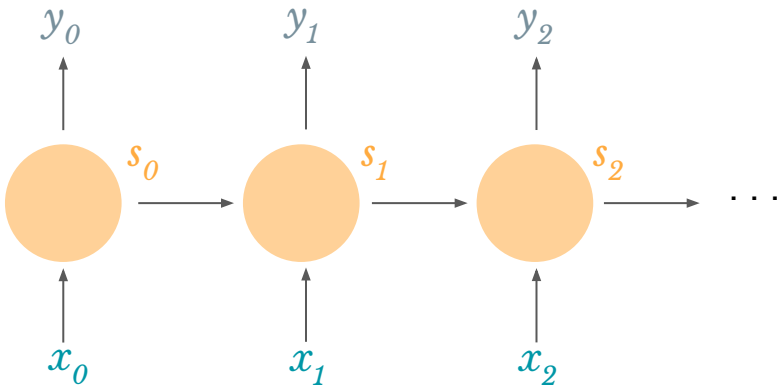
Computational Tutorial:
An introduction to LSTMs in Tensorflow



TensorFlow

Harini Suresh

Nick Locascio

CSAIL

Part 1: Neural Networks Overview

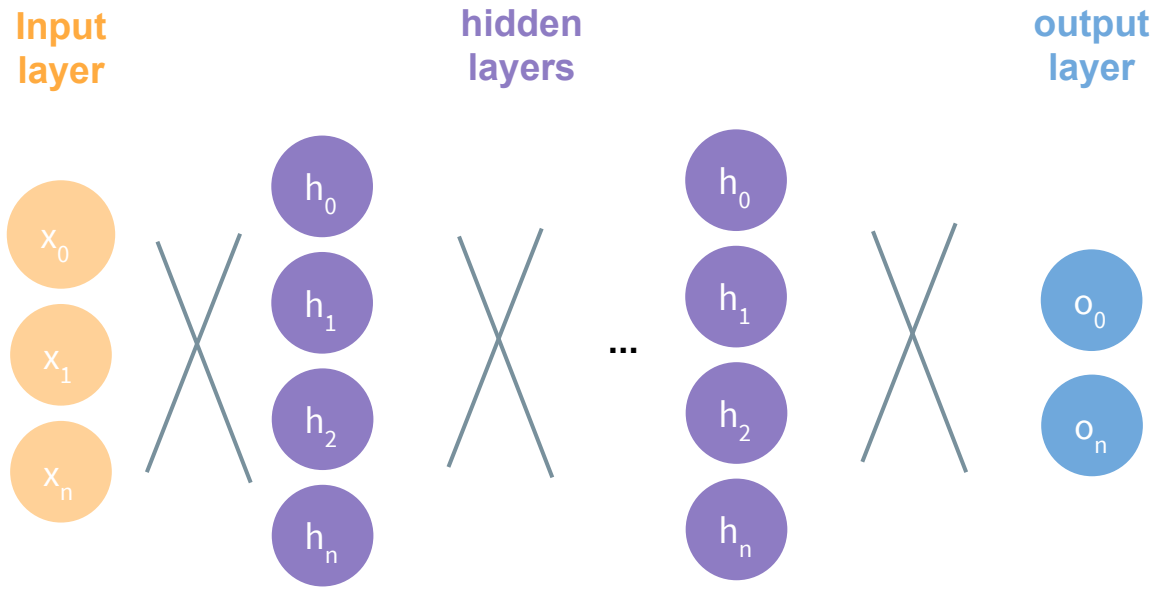
Part 2: Sequence Modeling with LSTMs

Part 3: TensorFlow Fundamentals

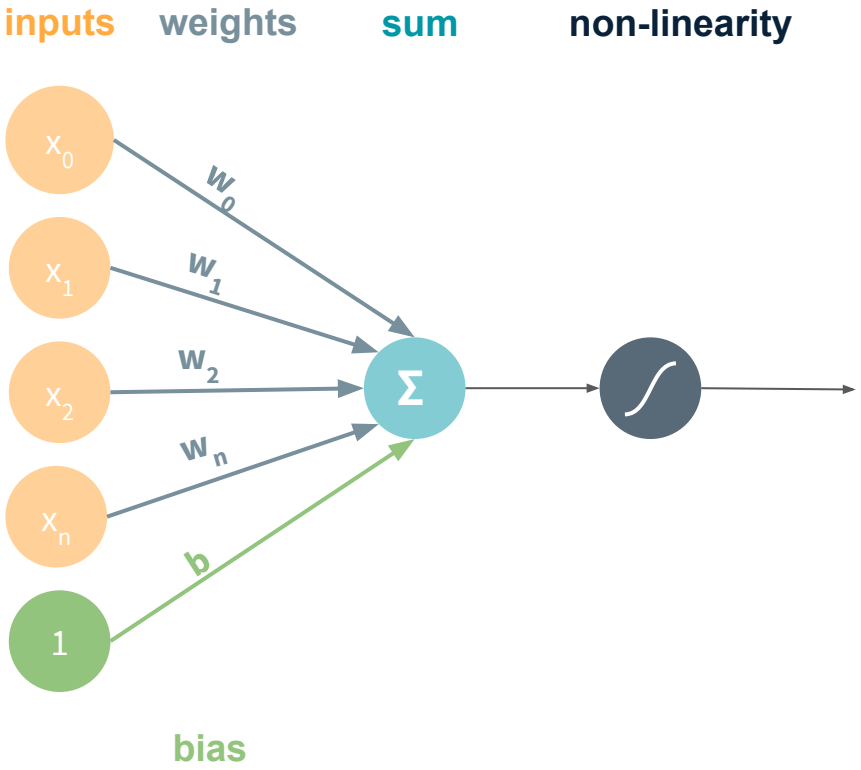
Part 4: LSTMs + Tensorflow Tutorial

Part 1: Neural Networks Overview

Neural Network



The Perceptron

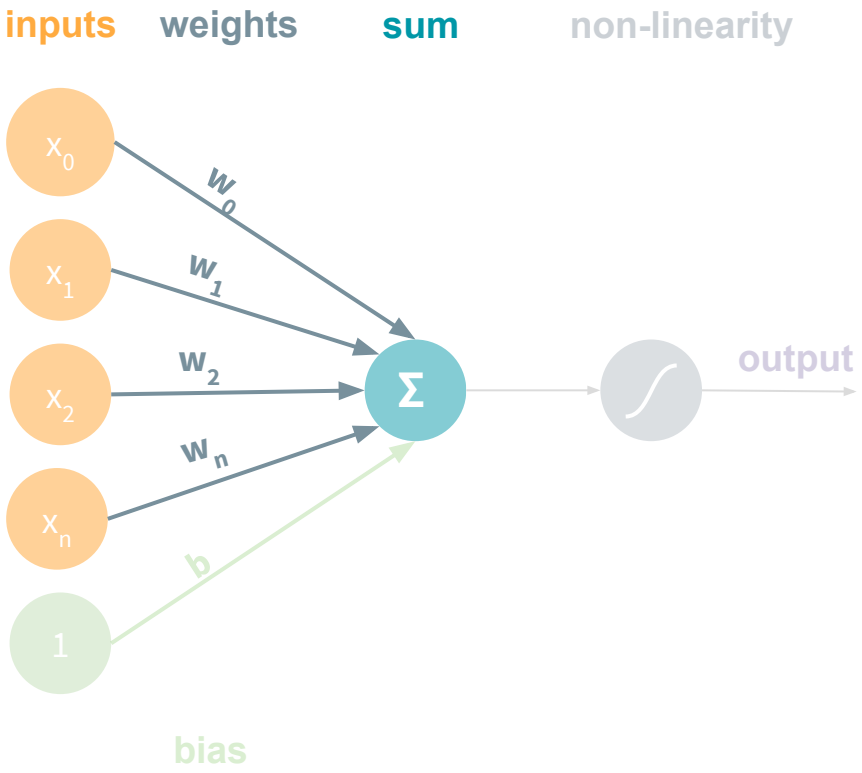


Perceptron Forward Pass

$$output =$$

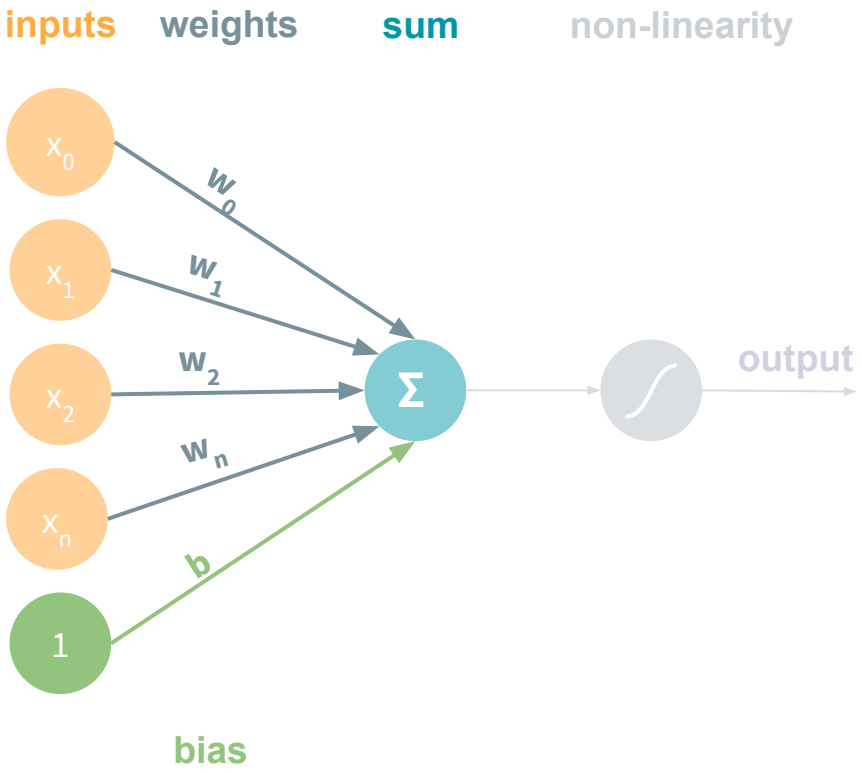
Perceptron Forward Pass

$$output = \sum_{i=0}^{N} x_i * w_i$$



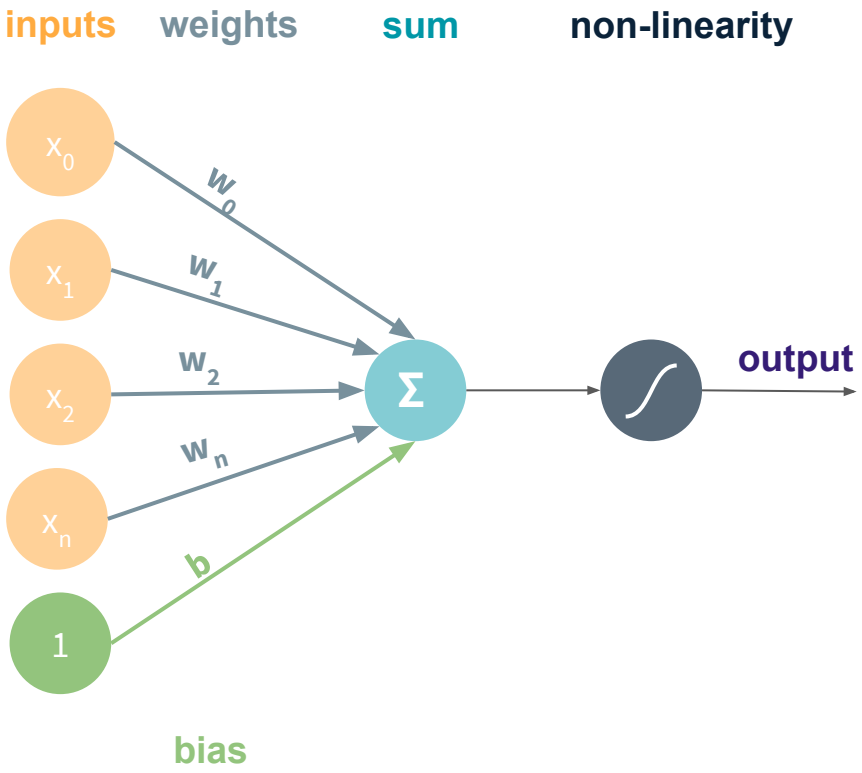
Perceptron Forward Pass

$$output = \sum_{i=0}^{N} (x_i * w_i) + b$$



Perceptron Forward Pass

$$output = g((\sum_{i=0}^{N} x_i * w_i) + b)$$

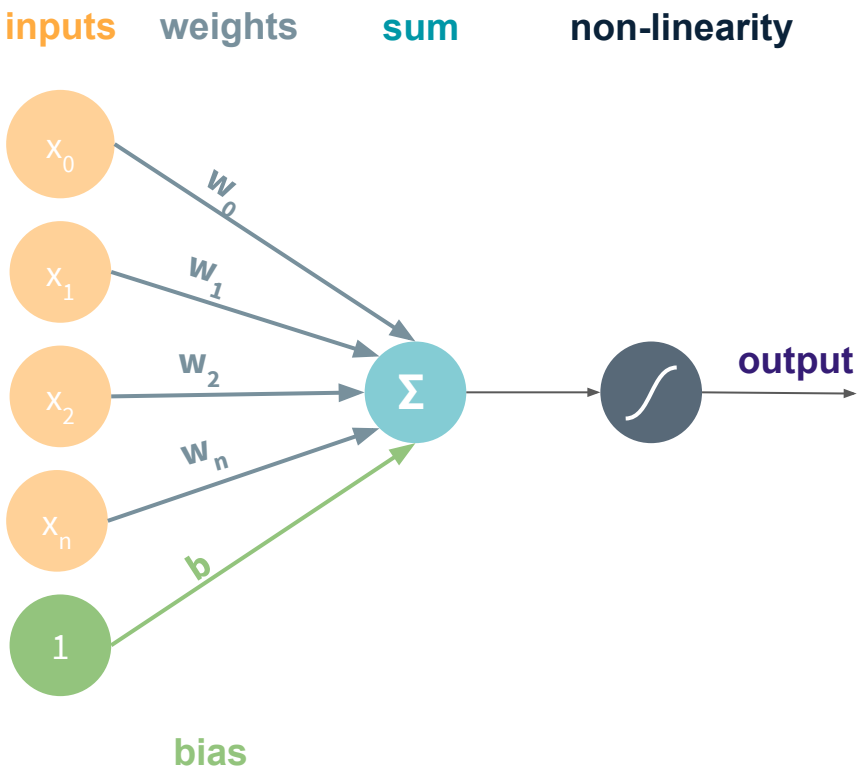


Perceptron Forward Pass

$$output = g(XW + b)$$

$$X = x_0, x_1, ...x_n$$

$$W = w_0, w_1, ...w_n$$



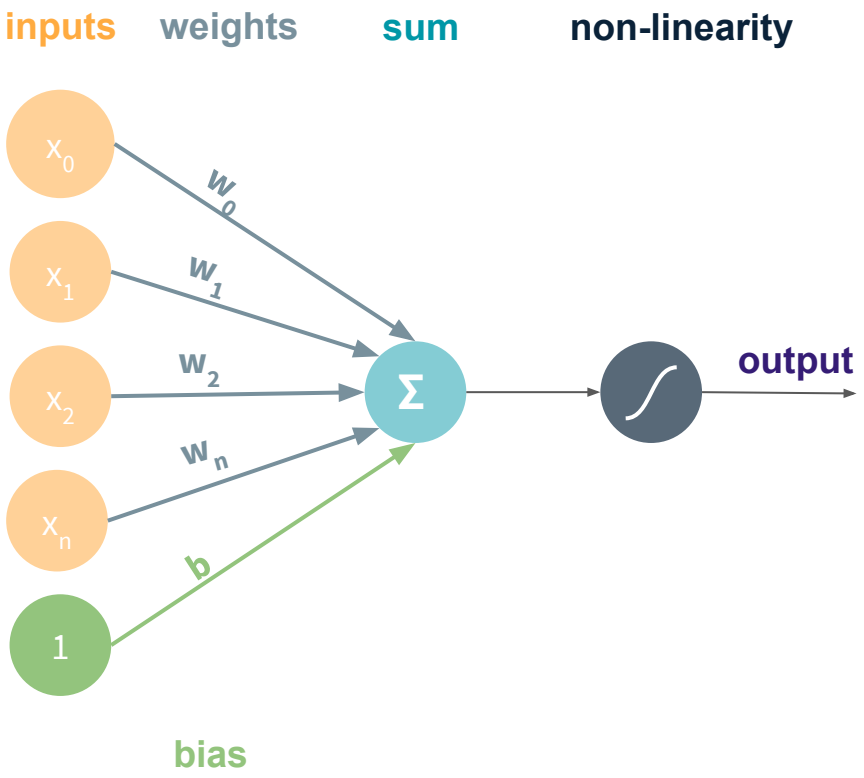
Perceptron Forward Pass

Activation Function

$$output = g(XW + b)$$

$$X = x_0, x_1, ...x_n$$

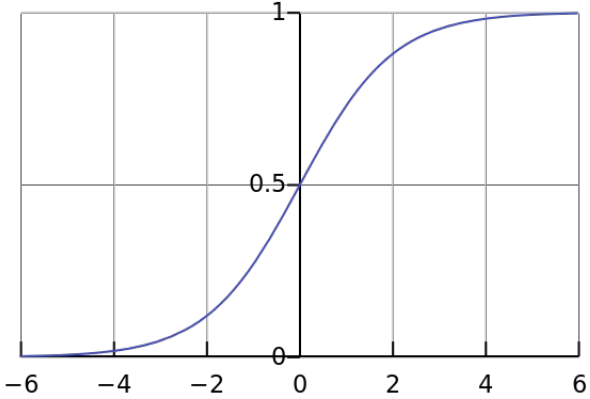
$$W = w_0, w_1, ...w_n$$

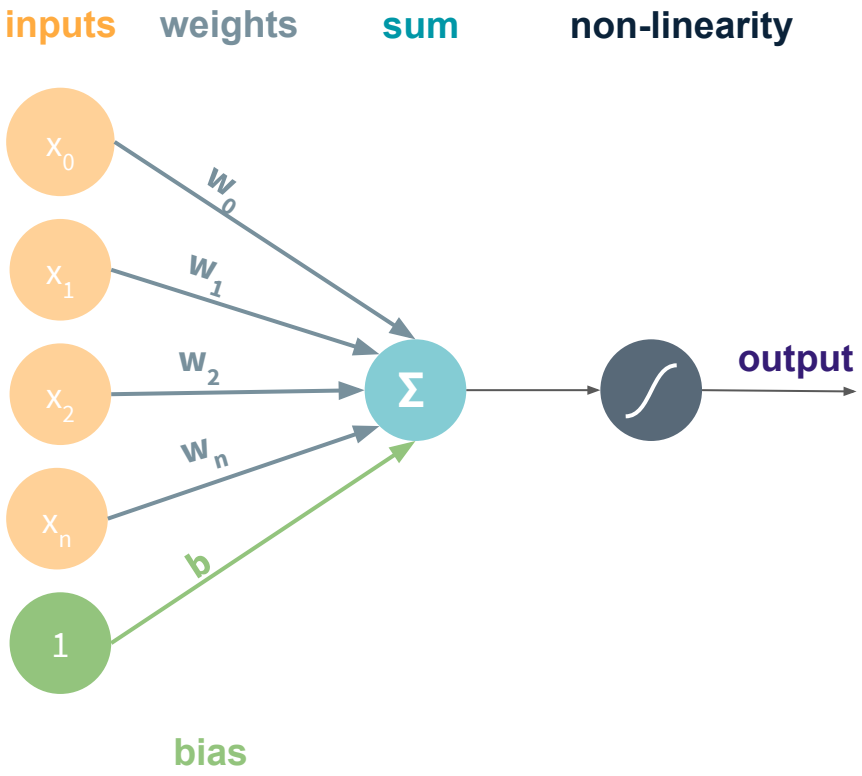


Sigmoid Activation

$$output = g(XW + b)$$

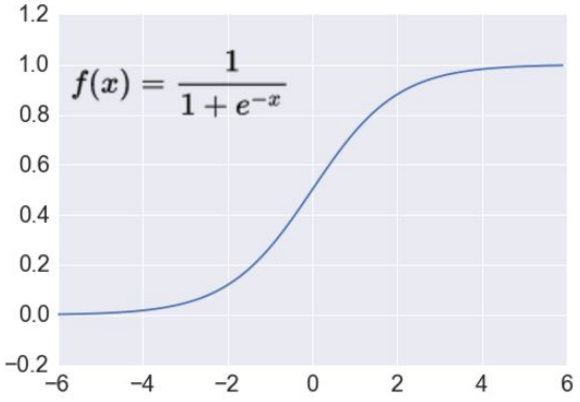
$$g(a) = \sigma(a) = \frac{1}{1 + e^{-a}}$$



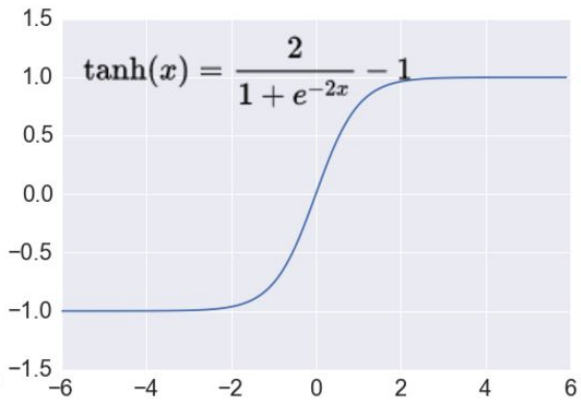


Common Activation Functions

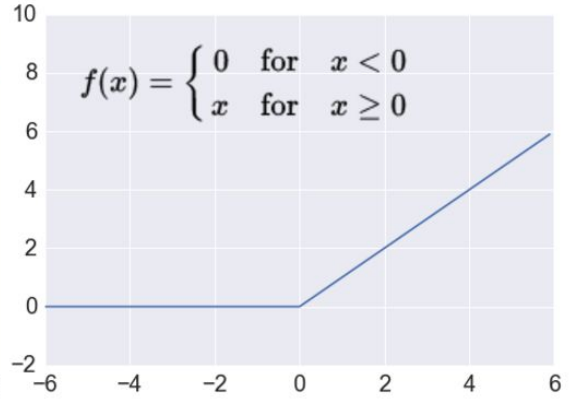
Sigmoid



TanH

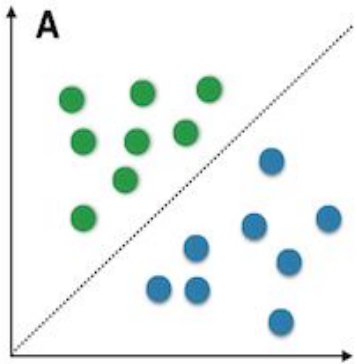


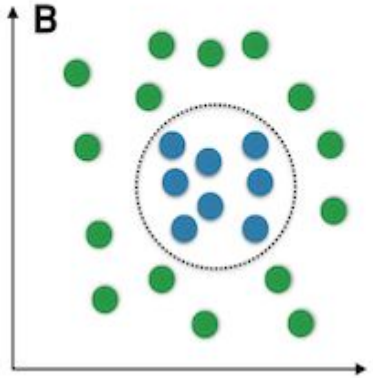
ReLU



Importance of Activation Functions

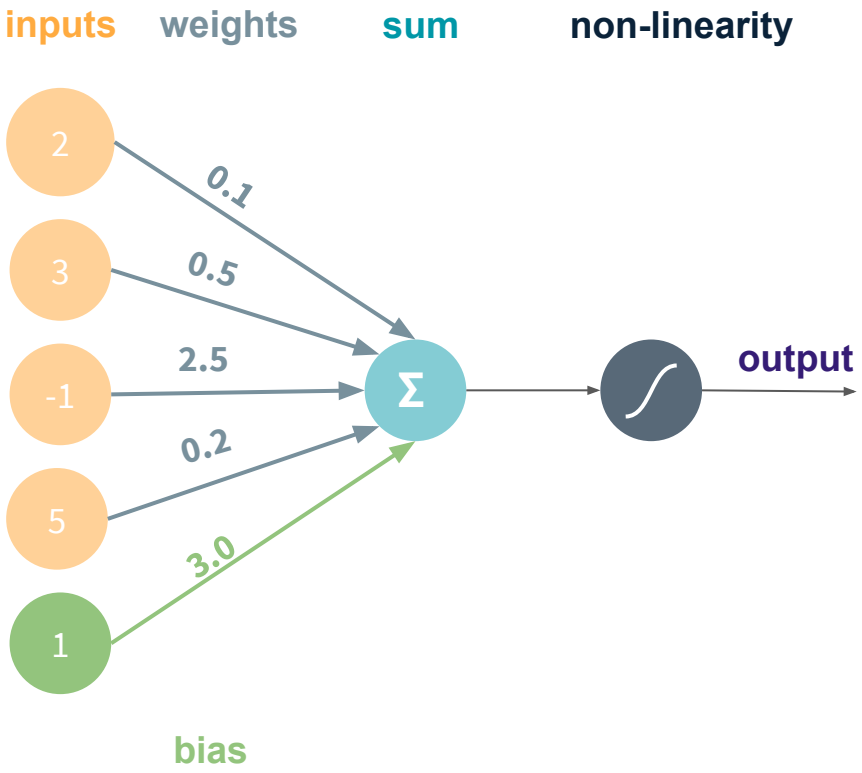
- Activation functions add non-linearity to our network's function
- Most real-world problems + data are **non-linear**





Perceptron Forward Pass

$$output = g(XW + b)$$



Perceptron Forward Pass

$$output = g($$

(2*0.1) +

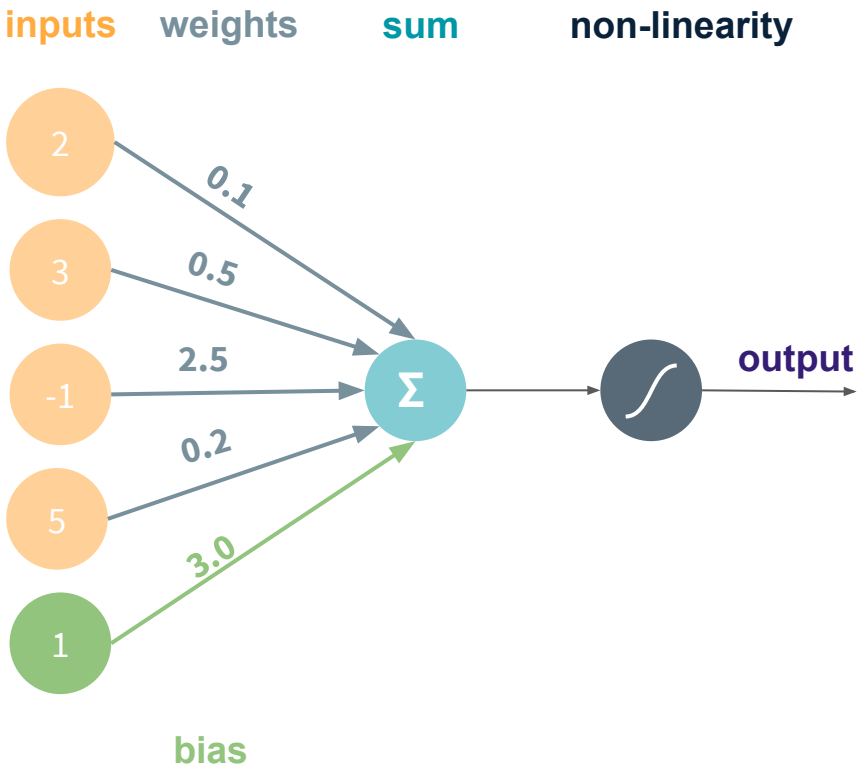
(3*0.5) +

(-1*2.5) +

(5*0.2) +

(1*3.0)

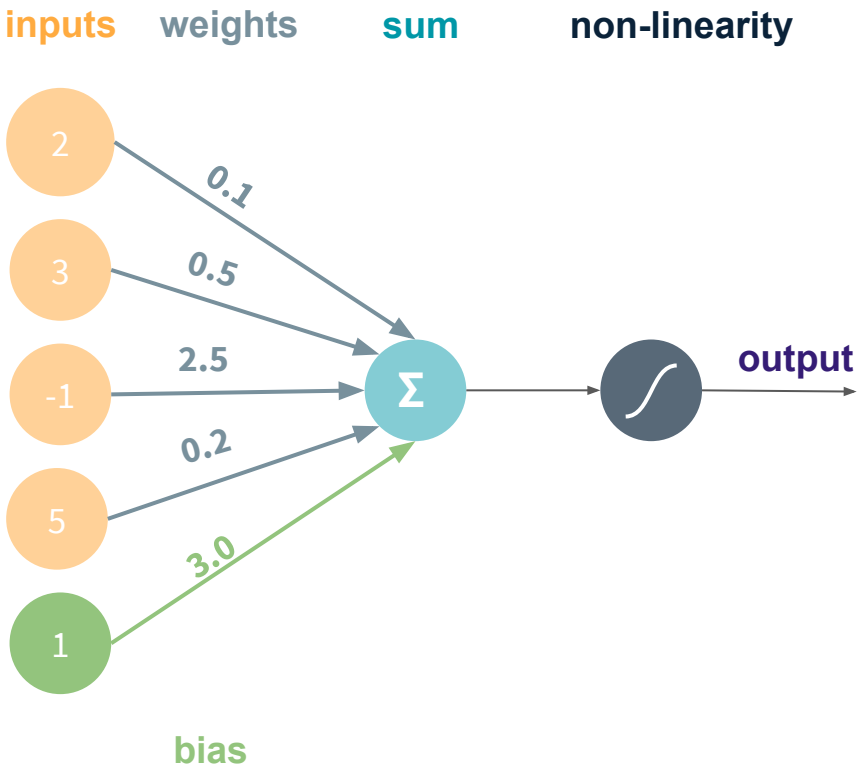
)



Perceptron Forward Pass

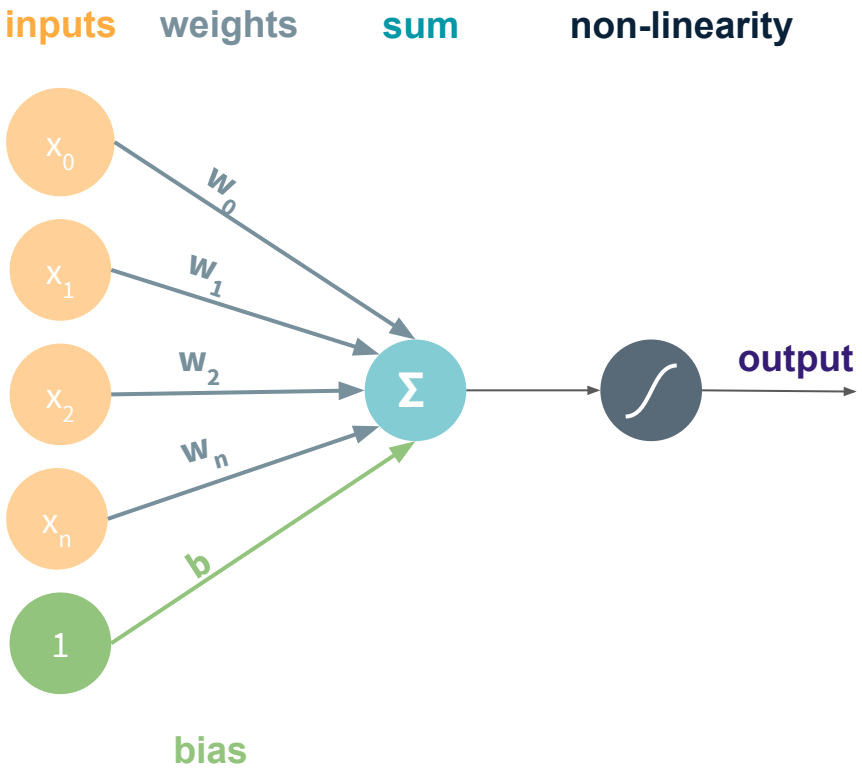
$$output = g(3.2) = \sigma(3.2)$$

$$= \frac{1}{(1 + e^{-3.2})} = 0.96$$

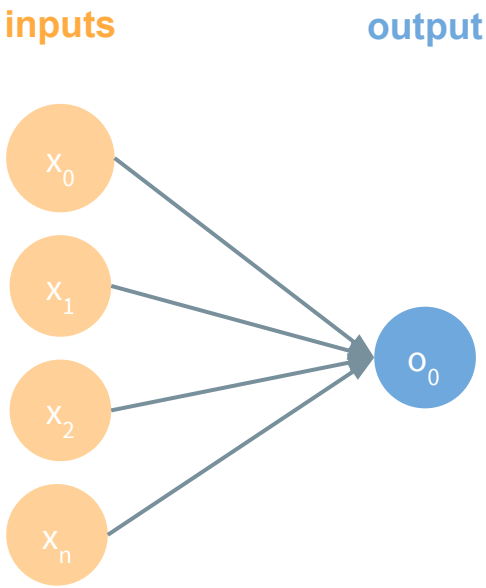


How do we build neural networks with perceptrons?

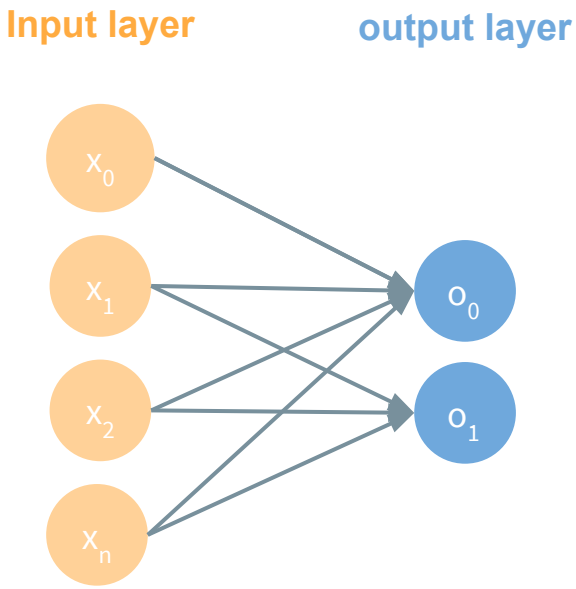
Perceptron Diagram Simplified



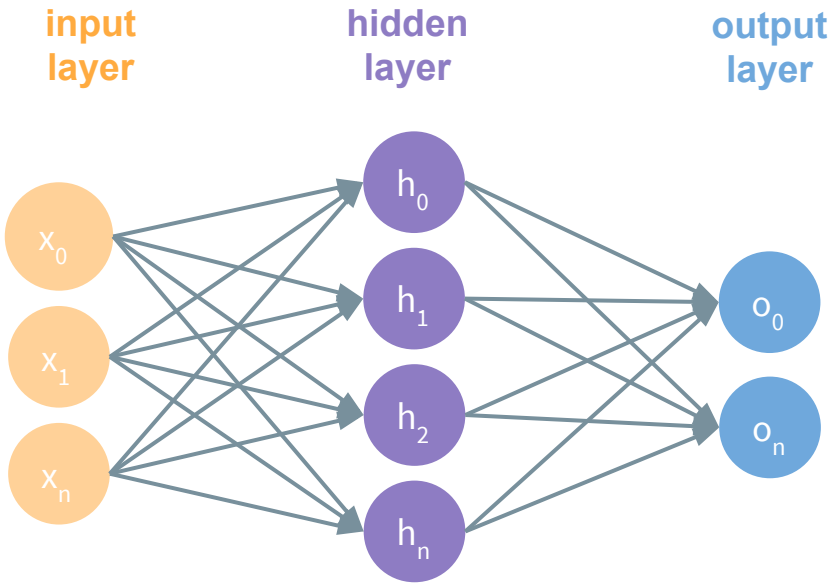
Perceptron Diagram Simplified



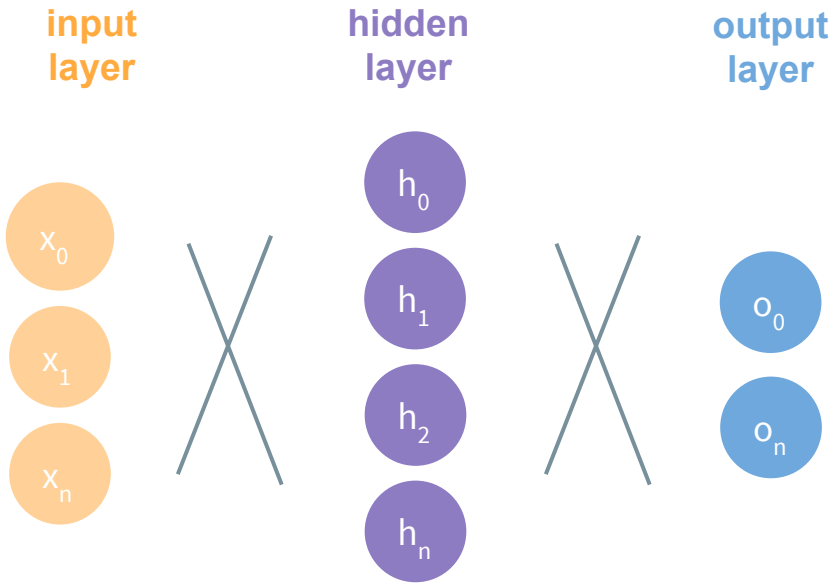
Multi-Output Perceptron



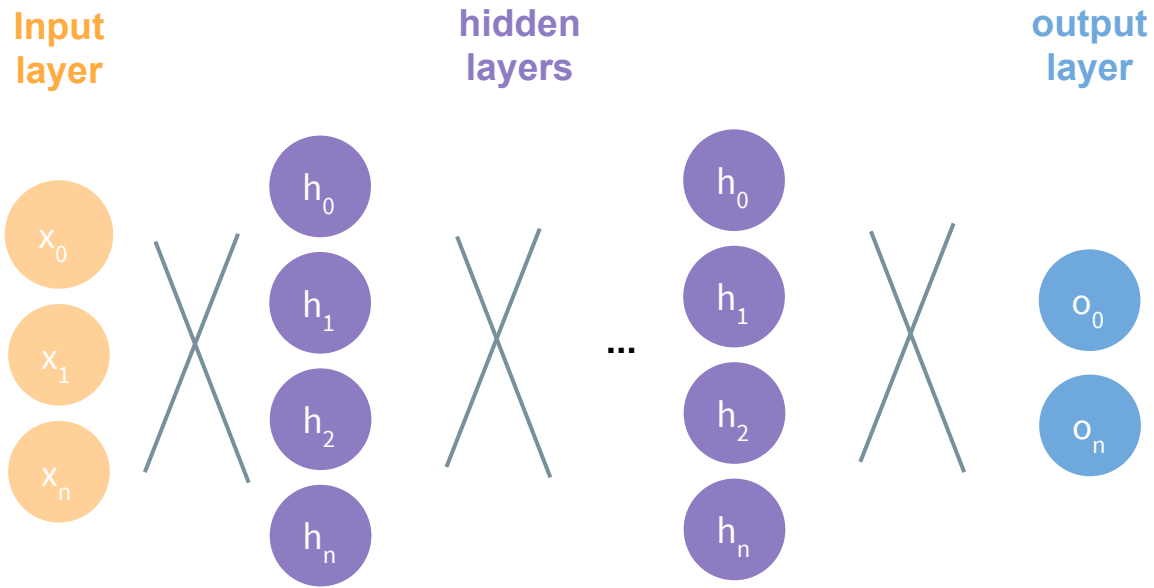
Multi-Layer Perceptron (MLP)



Multi-Layer Perceptron (MLP)



Deep Neural Network



Training Neural Networks

Training Neural Networks: Loss function

$$\text{loss} := J(\theta) = \frac{1}{N} \sum_{i}^{N} loss(f(x^{(i)}; \theta), y^{(i)}))$$

N = # examples

Predicted

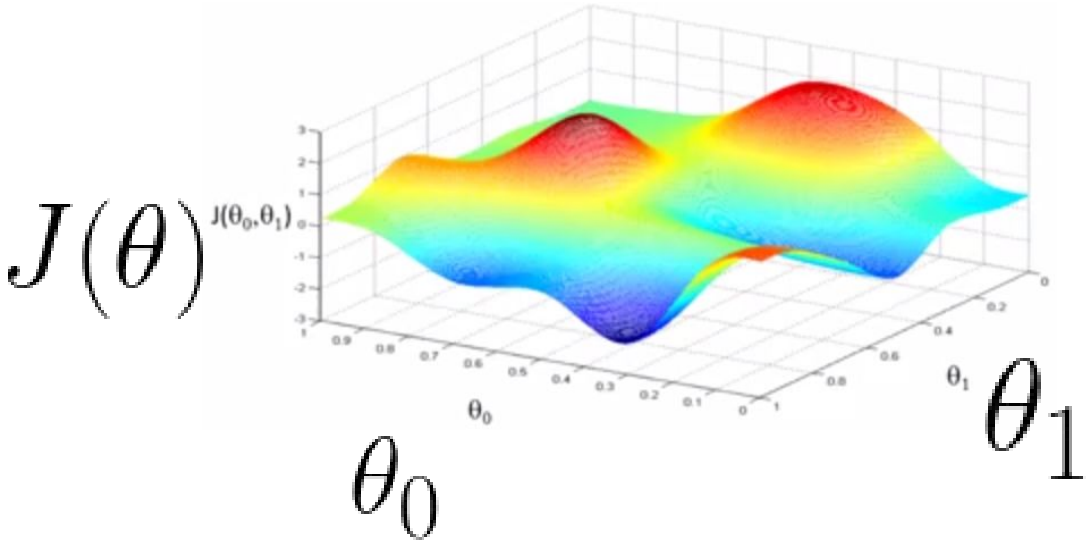
Actual

Training Neural Networks: Objective

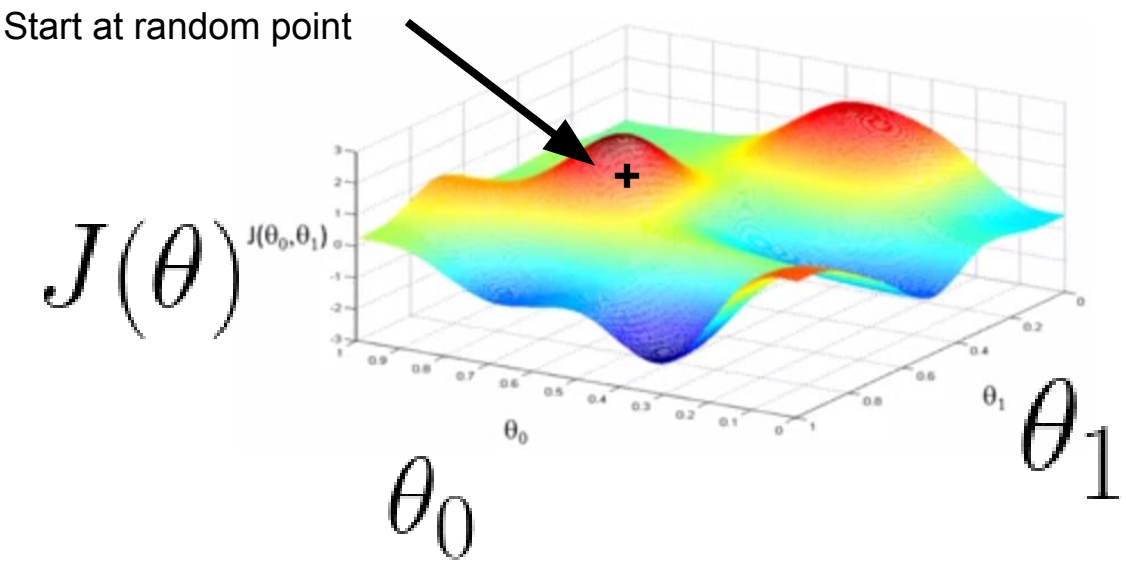
$$arg_\theta \min \underbrace{\frac{1}{N}\sum_i^N \text{loss}(f(x^{(i)};\theta), y^{(i)})}_{J(\theta)}$$

$$\theta = W_1, W_2 \ldots W_n$$

Loss is a **function** of the model's parameters

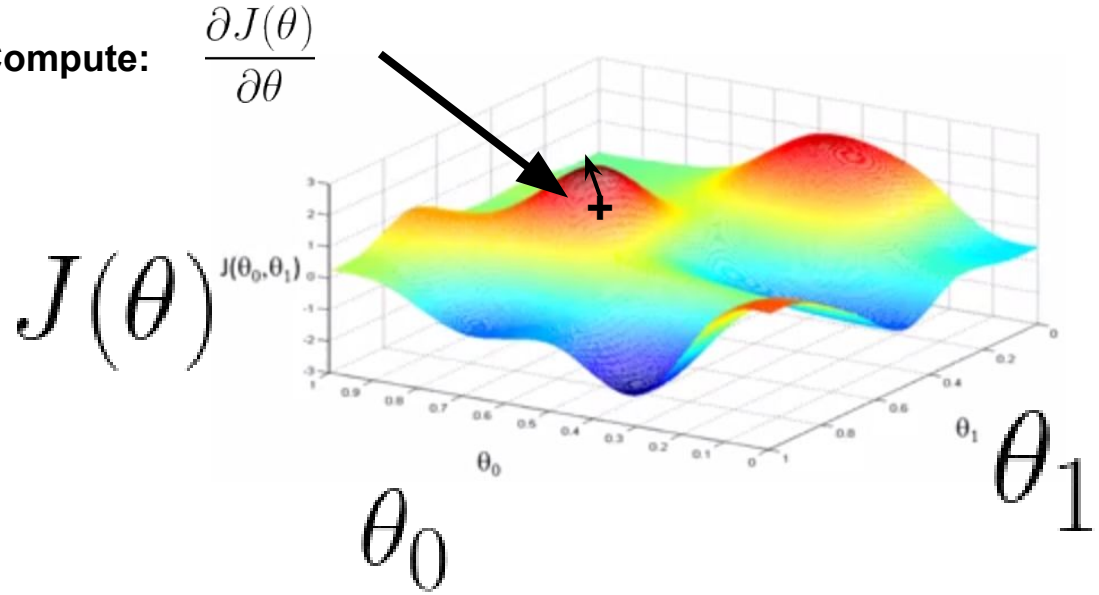


How to minimize loss?



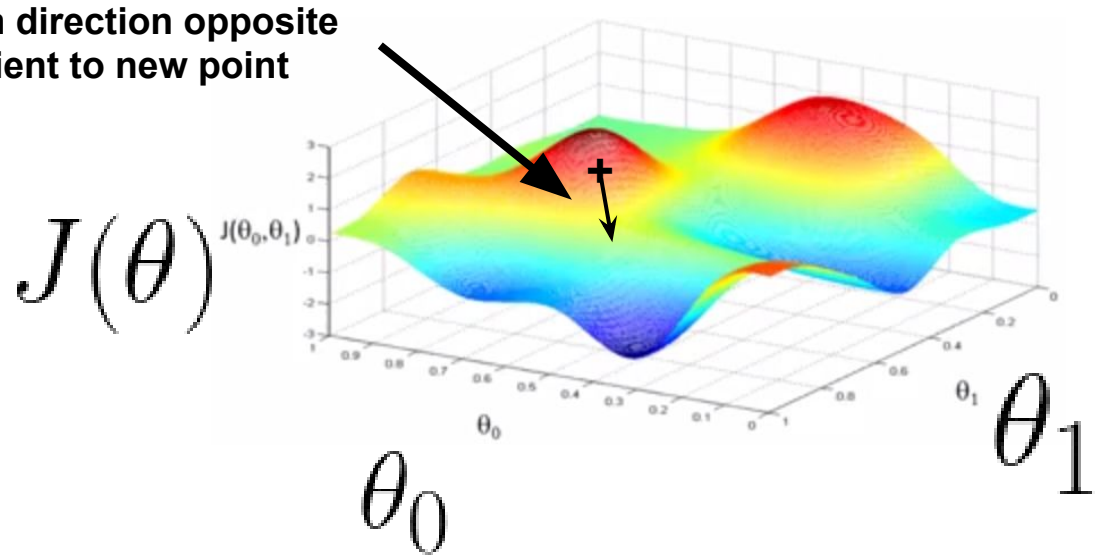
How to minimize loss?

Compute: $\frac{\partial J(\theta)}{\partial \theta}$



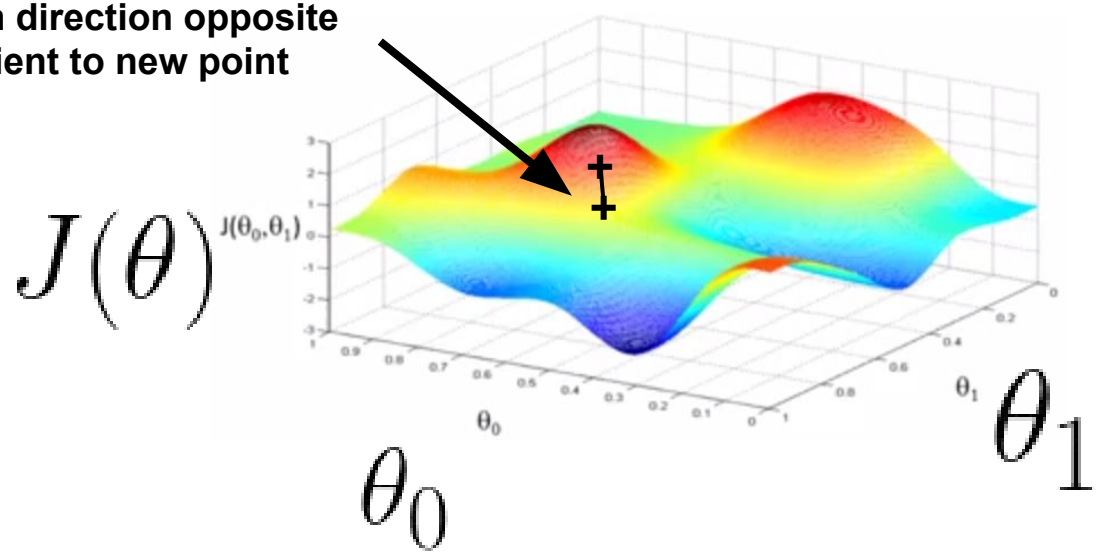
How to minimize loss?

Move in direction opposite of gradient to new point



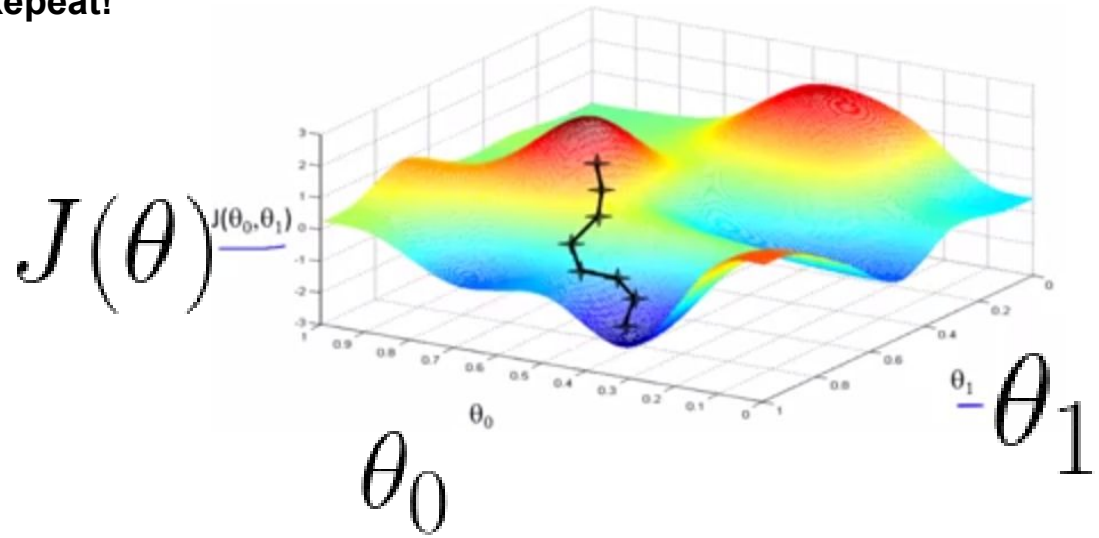
How to minimize loss?

Move in direction opposite of gradient to new point



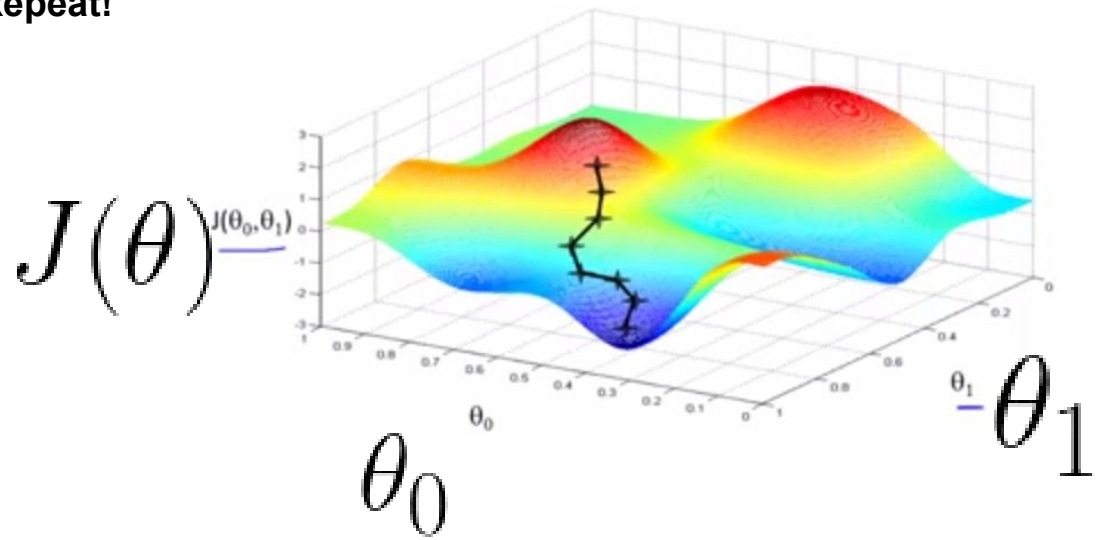
How to minimize loss?

Repeat!



This is called Stochastic Gradient Descent (SGD)

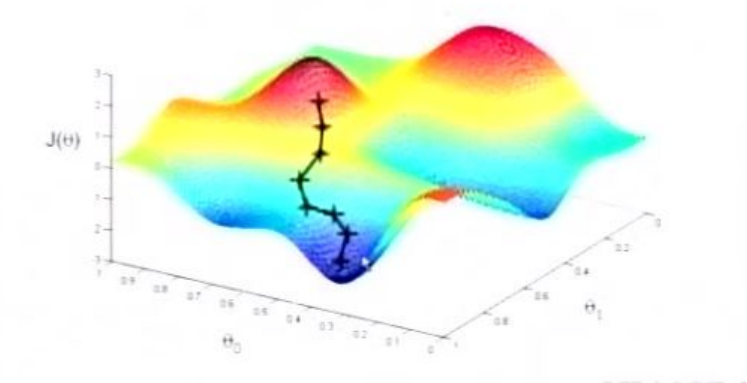
Repeat!



Stochastic Gradient Descent (SGD)

- Initialize θ randomly
- For N Epochs
  - For each training example (x, y):
    - Compute Loss Gradient: $\frac{\partial J(\theta)}{\partial \theta}$
    - Update θ with update rule:

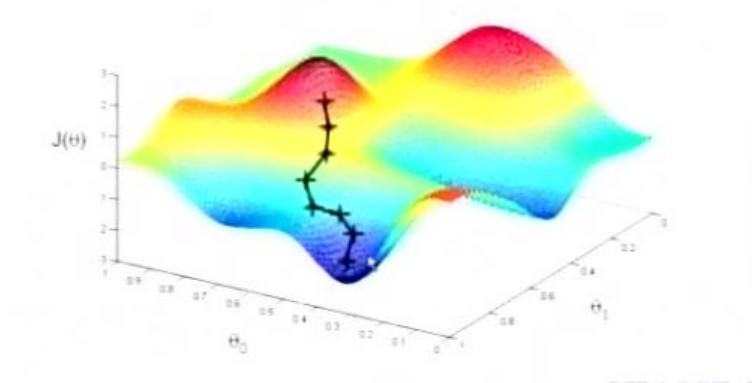
$$\theta := \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$$



Stochastic Gradient Descent (SGD)

- Initialize θ randomly
- For N Epochs
  - For each training example (x, y):
    - Compute Loss Gradient: $\frac{\partial J(\theta)}{\partial \theta}$
    - Update θ with update rule:

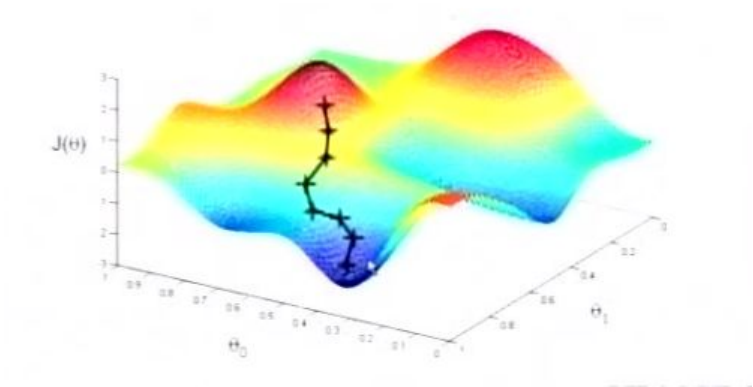
$$\theta := \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$$



Stochastic Gradient Descent (SGD)

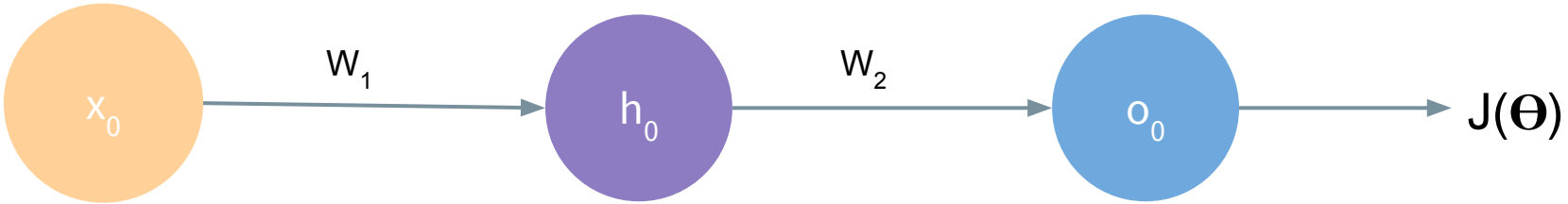
- Initialize θ randomly
- For N Epochs
  - For each training example (x, y):
    - Compute Loss Gradient: $\frac{\partial J(\theta)}{\partial \theta}$
    - Update θ with update rule:

$$\theta := \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$$

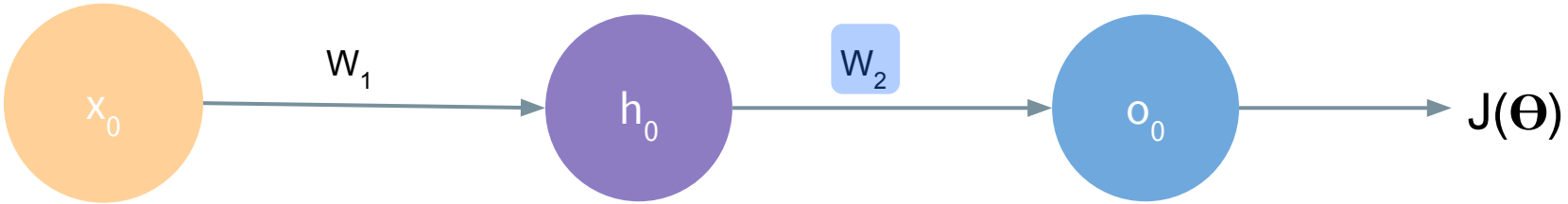


- How to Compute Gradient?

Calculating the Gradient: Backpropagation

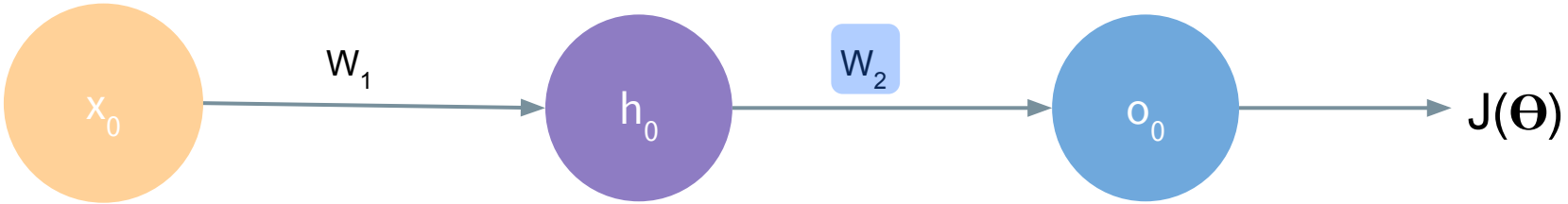


Calculating the Gradient: Backpropagation



$$\frac{\partial J(\theta)}{\partial W_2} =$$

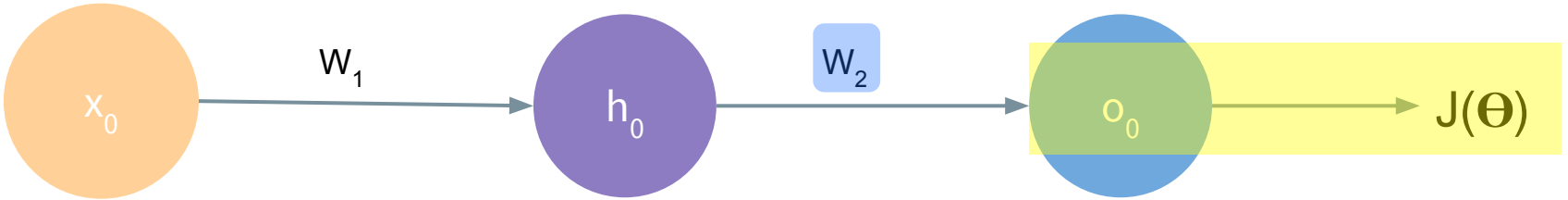
Calculating the Gradient: Backpropagation



Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_2} =$$

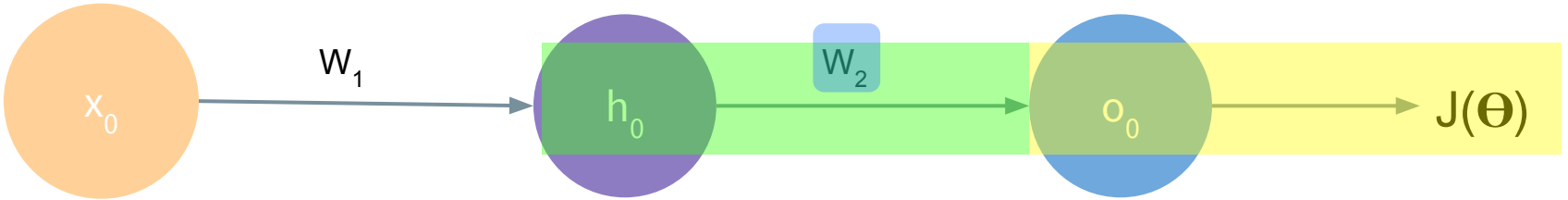
Calculating the Gradient: Backpropagation



Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_2} = \frac{\partial J(\theta)}{\partial o_0}$$

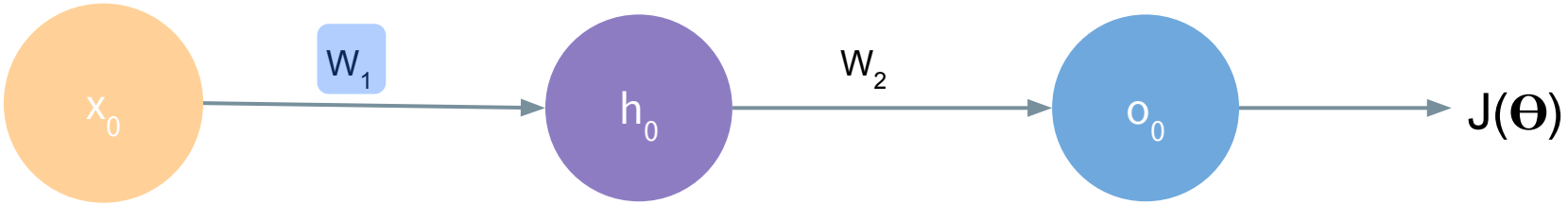
Calculating the Gradient: Backpropagation



Apply the chain rule

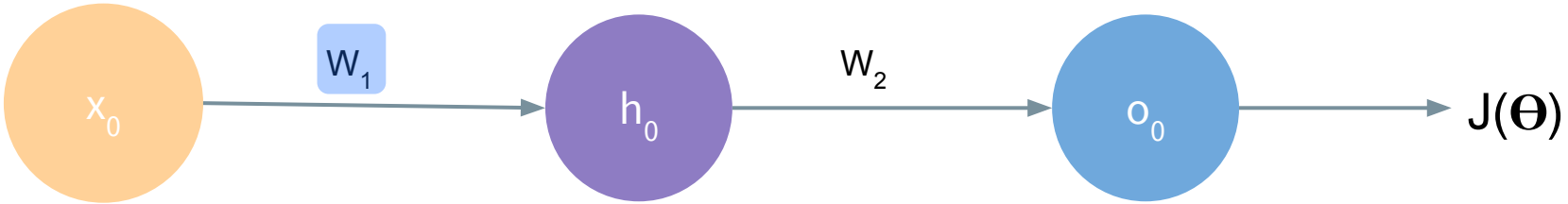
$$\frac{\partial J(\theta)}{\partial W_2} = \frac{\partial J(\theta)}{\partial o_0} * \frac{\partial o_0}{\partial W_2}$$

Calculating the Gradient: Backpropagation



$$\frac{\partial J(\theta)}{\partial W_1} =$$

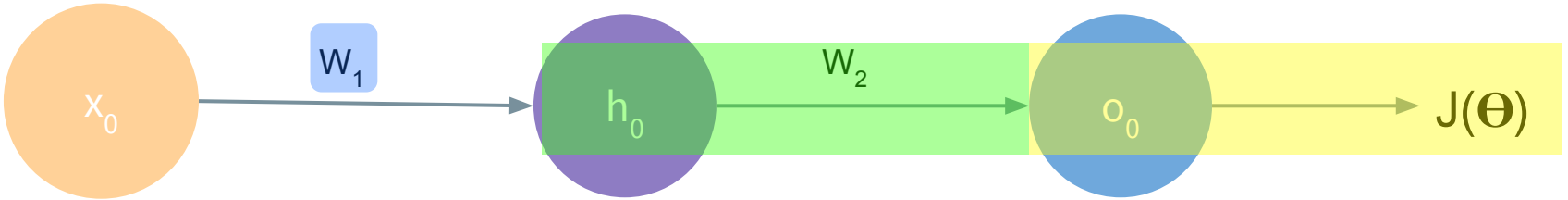
Calculating the Gradient: Backpropagation



Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_1} =$$

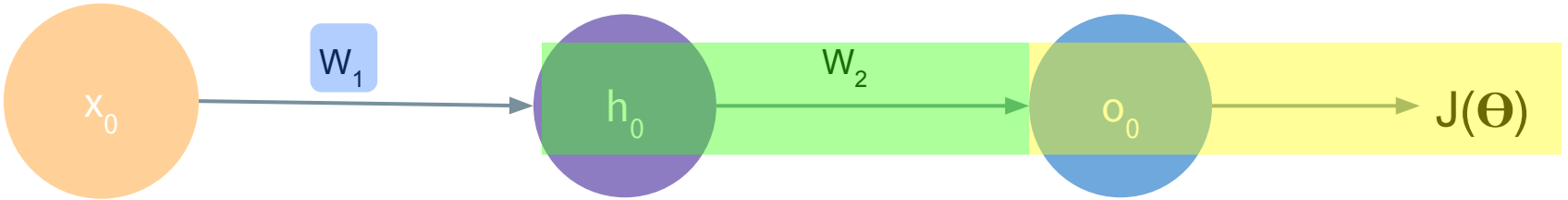
Calculating the Gradient: Backpropagation



Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_1} = \frac{\partial J(\theta)}{\partial o_0} * \frac{\partial o_0}{\partial h_0}$$

Calculating the Gradient: Backpropagation

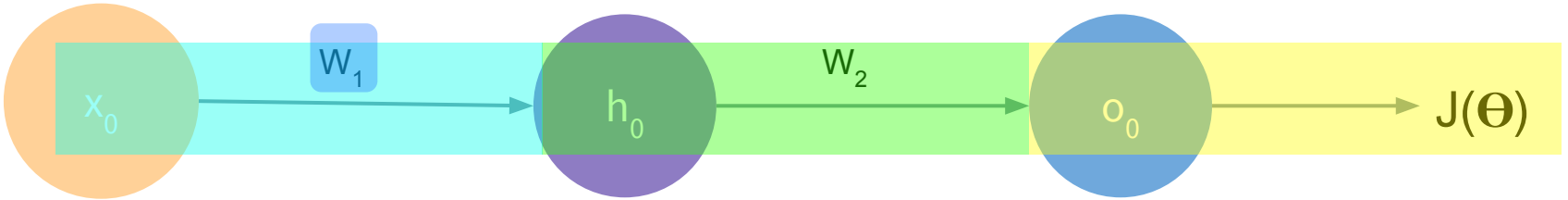


Apply the chain rule

Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_1} = \frac{\partial J(\theta)}{\partial o_0} * \frac{\partial o_0}{\partial h_0}$$

Calculating the Gradient: Backpropagation



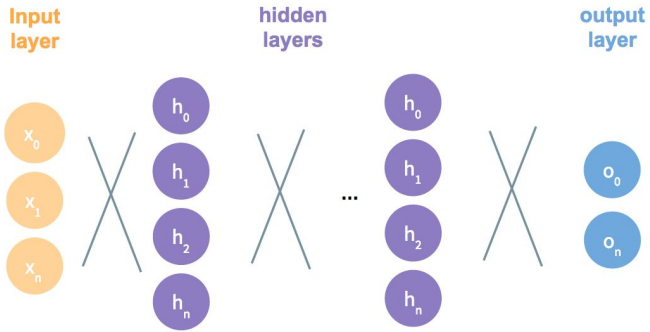
Apply the chain rule

Apply the chain rule

$$\frac{\partial J(\theta)}{\partial W_1} = \frac{\partial J(\theta)}{\partial o_0} * \frac{\partial o_0}{\partial h_0} * \frac{\partial h_0}{\partial W_1}$$

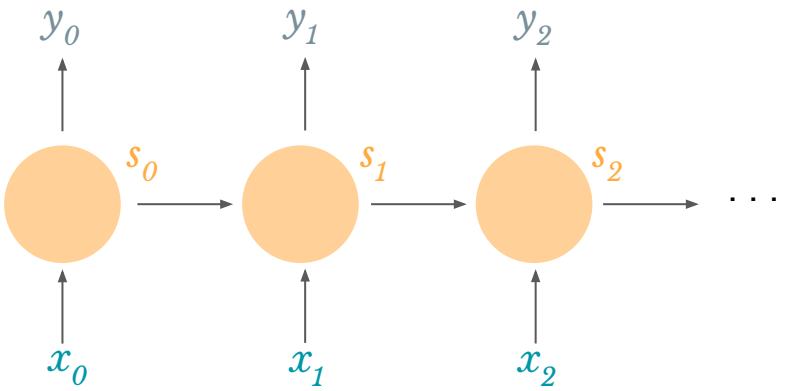
Core Fundamentals Review

- Perceptron Classifier
- Stacking Perceptrons to form neural networks
- How to formulate problems with neural networks
- Train neural networks with backpropagation

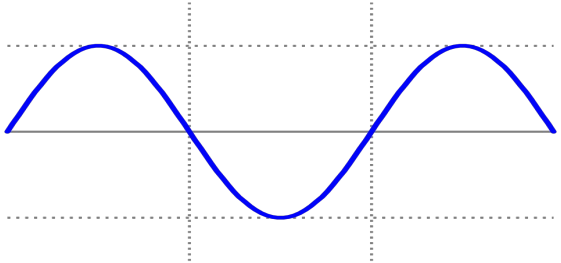


Part 2: Sequence Modeling with Neural Networks

Harini Suresh

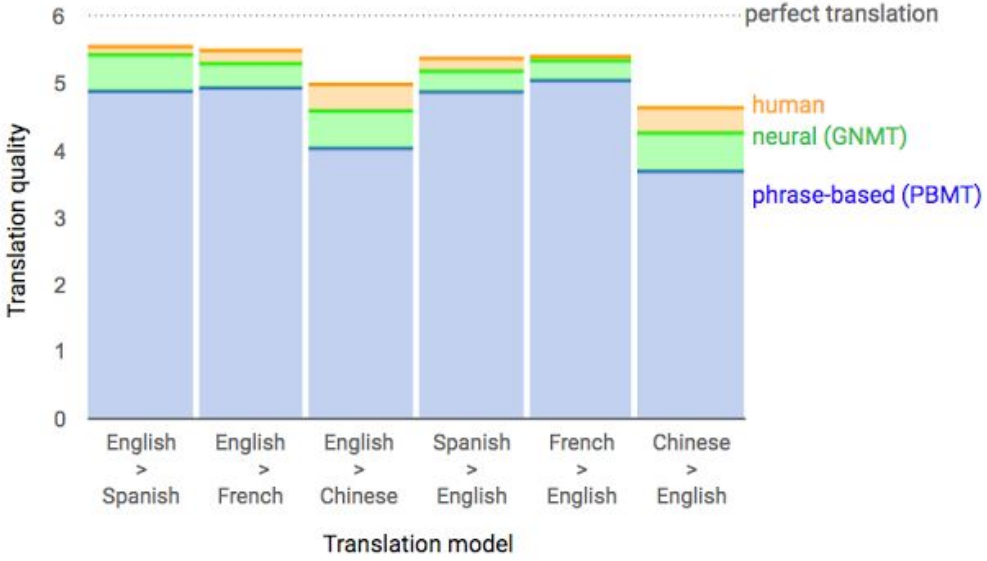


What is a sequence?

- “I took the dog for a walk this morning.” sentence
-  function
-  speech waveform

Successes of deep models

Machine translation



Question Answering

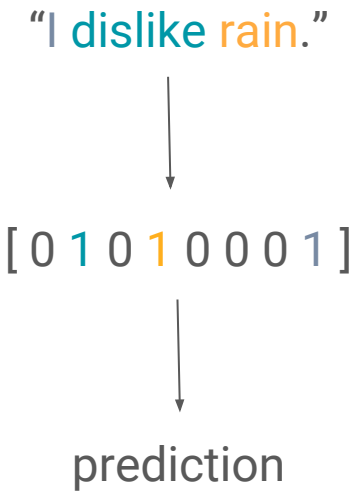
Super Bowl 50 was an American football game to determine the champion of the National Football League (NFL) for the 2015 season. The American Football Conference (AFC) champion Denver Broncos defeated the National Football Conference (NFC) champion Carolina Panthers 24–10 to earn their third Super Bowl title. The game was played on February 7, 2016, at Levi's Stadium in the San Francisco Bay Area at Santa Clara, California. As this was the 50th Super Bowl, the league emphasized the "golden anniversary" with various gold-themed initiatives, as well as temporarily suspending the tradition of naming each Super Bowl game with Roman numerals (under which the game would have been known as "Super Bowl L"), so that the logo could prominently feature the Arabic numerals 50.

Super Bowl 50 decided the NFL champion for what season?
Ground Truth Answers: 2015 | the 2015 season | 2015
Prediction: 2015

Left:
https://research.googleblog.com/2016/09/a-neural-network-for-machine.html
Right:
https://rajpurkar.github.io/SQuAD-explorer/

how do we model sequences?

idea: represent a sequence as a *bag of words*



problem: bag of words does not preserve order

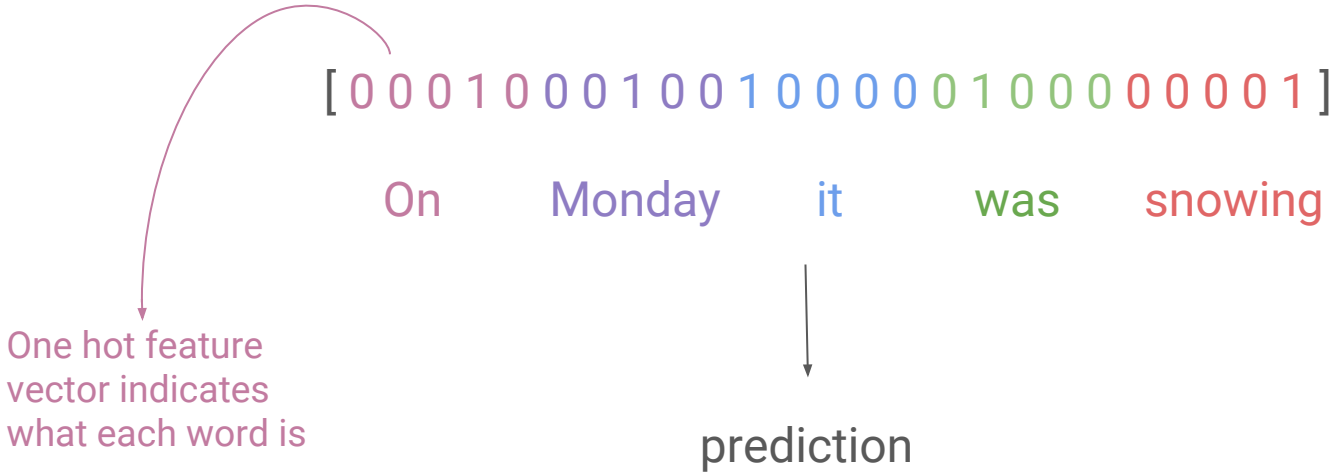
problem: bag of words does not preserve order

“The food was good, not bad at all.”

vs

“The food was bad, not good at all.”

idea: maintain an ordering within feature vector



problem: hard to deal with different word orders

“On Monday, it was snowing.”

vs

“It was snowing on Monday.”

problem: hard to deal with different word orders

[0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1]

On Monday it was snowing

vs

[1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0]

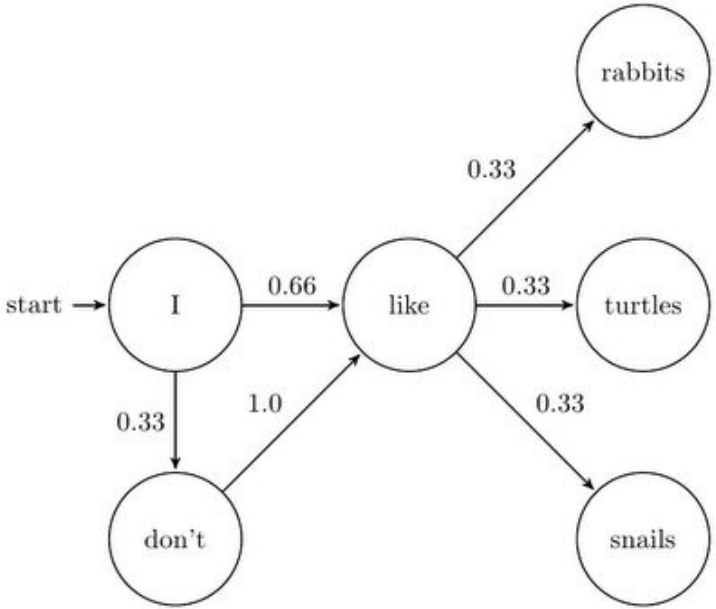
It was snowing on Monday

problem: hard to deal with different word orders

“On Monday it was snowing.”
vs
“It was snowing on Monday.”

We would have to relearn the rules of language at each point in the sentence.

idea: markov models



problem: we can’t model long-term dependencies

markov assumption: each state depends only on the last state.

problem: we can’t model long-term dependencies

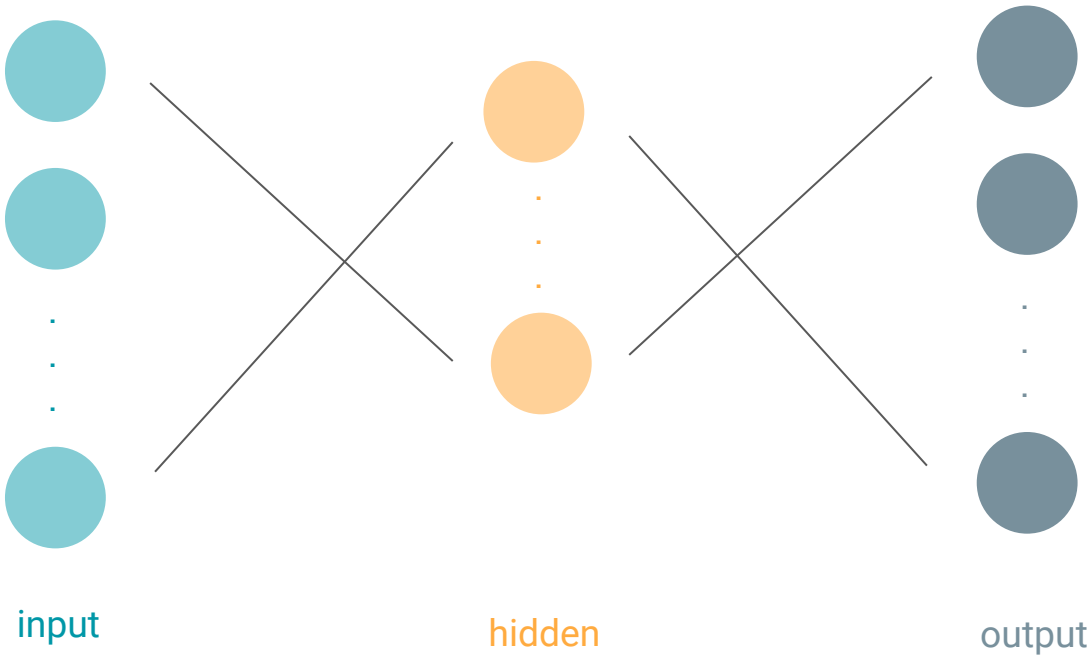
“In France, I had a great time and I learnt some of the _____ language.”

We need information from the far past and future to accurately guess the correct word.

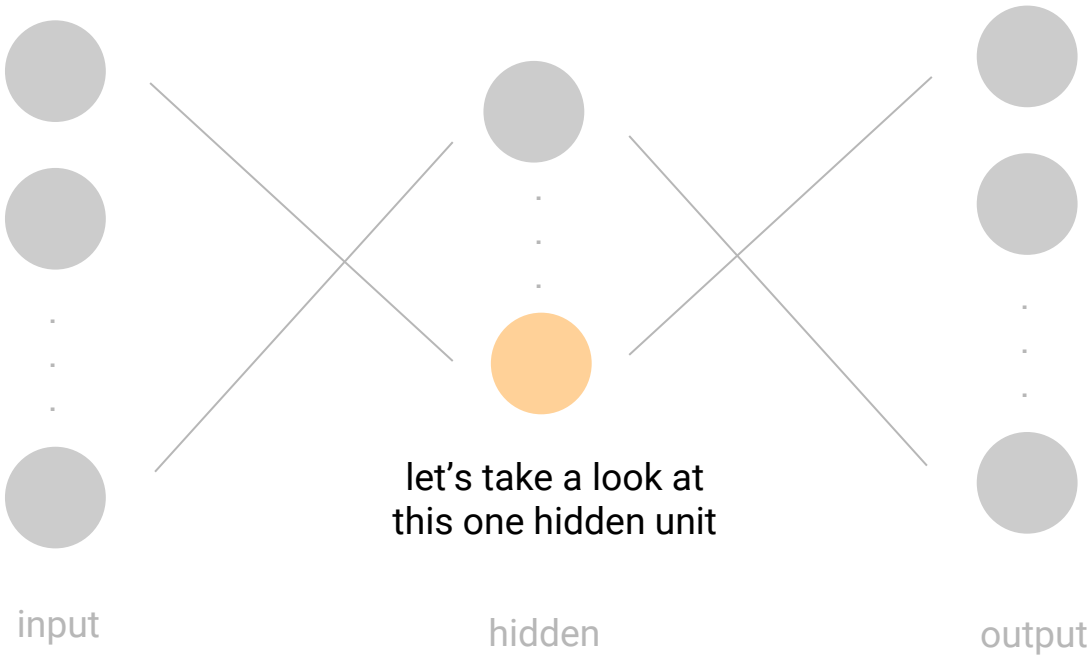
let’s turn to **recurrent neural networks**! (RNNs)

1. to maintain word order
2. to share parameters across the sequence
3. to keep track of long-term dependencies

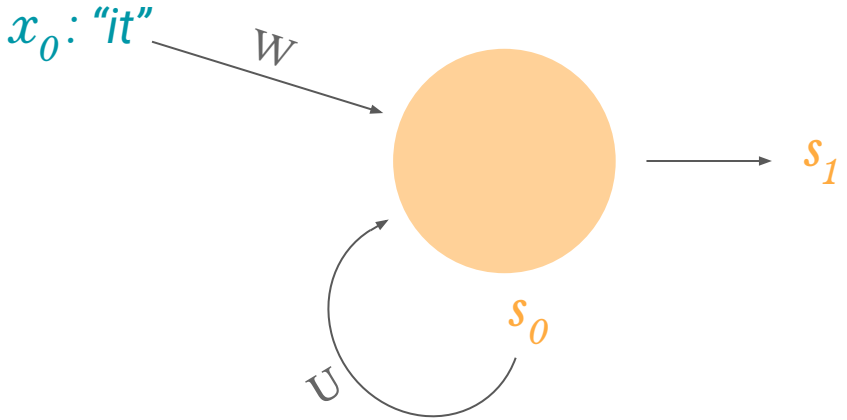
example network:



example network:



RNNS **remember** their previous state:

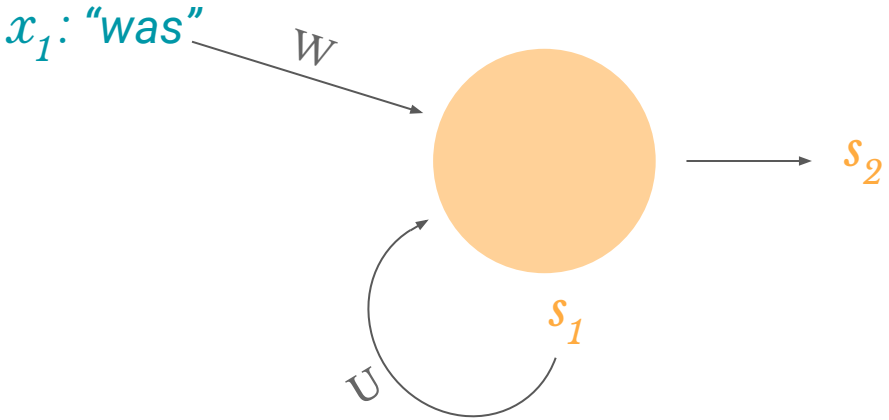


x_0 : vector representing first word
s_0 : cell state at $t = 0$ (some initialization)
s_1 : cell state at $t = 1$

$$s_1 = tanh(Wx_0 + Us_0)$$

$W,\ U$: weight matrices

RNNS **remember** their previous state:

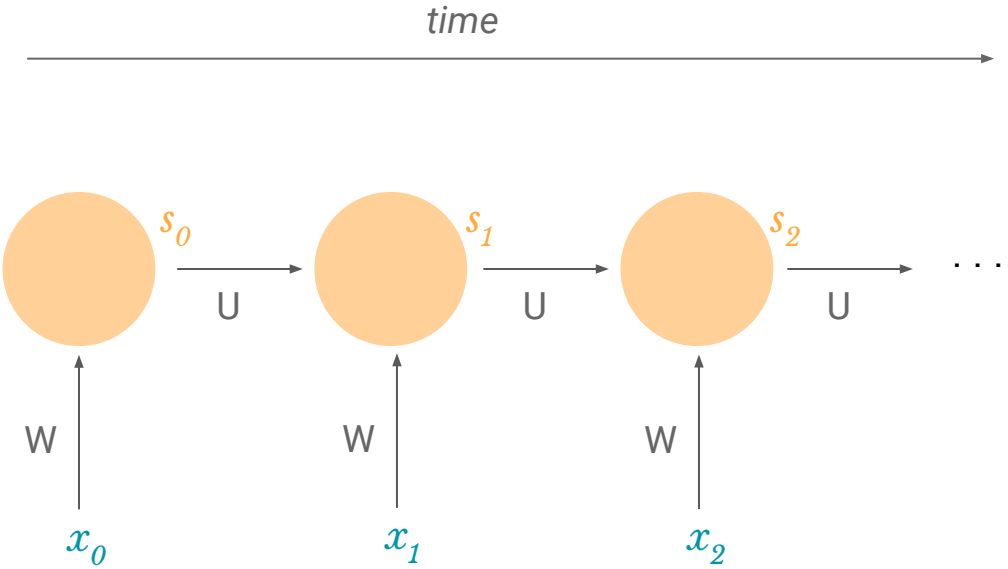


x_1 : vector representing second word
s_1 : cell state at $t = 1$
s_2 : cell state at $t = 2$

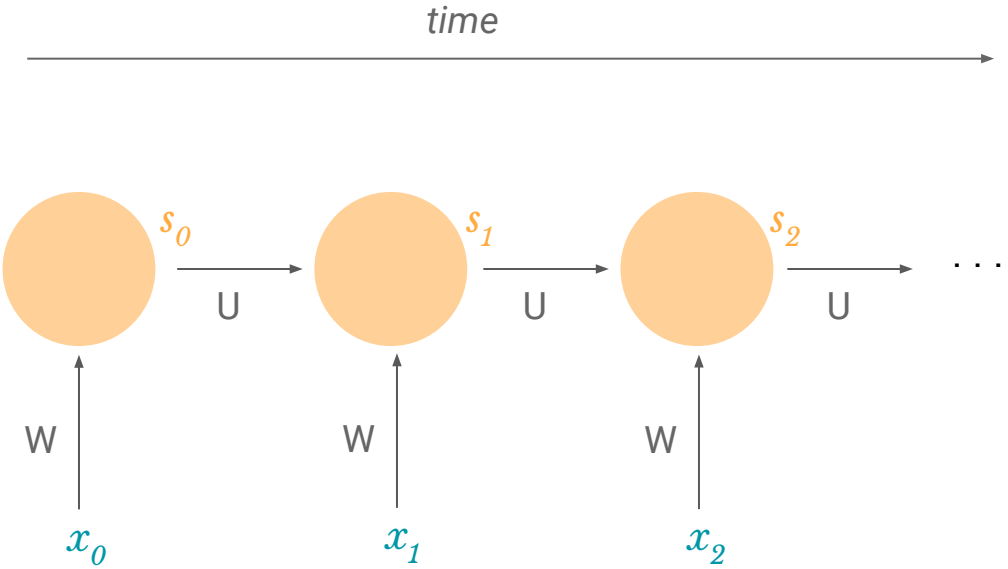
$$s_2 = tanh(Wx_1 + Us_1)$$

$W,\ U$: weight matrices

“unfolding” the RNN across time:

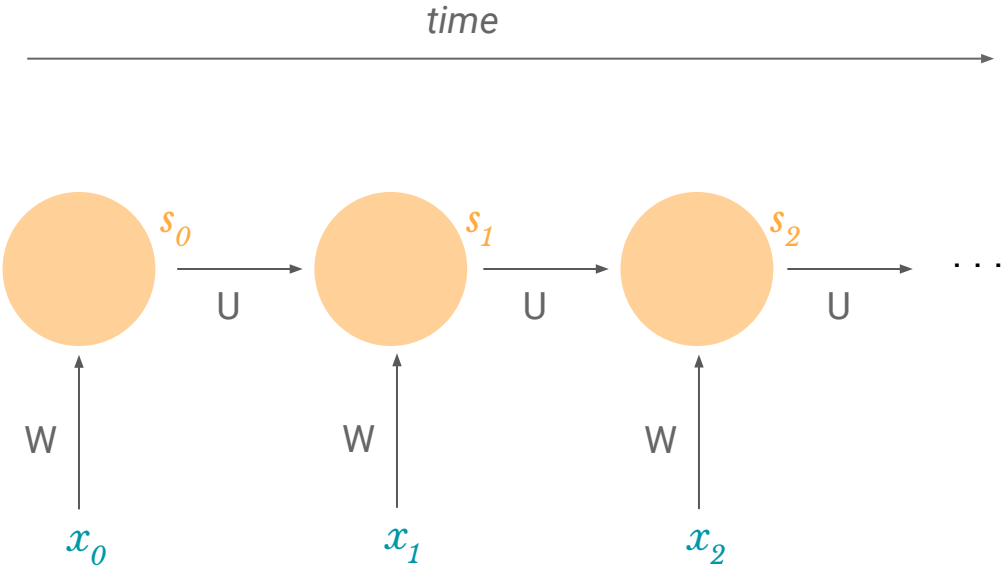


“unfolding” the RNN across time:



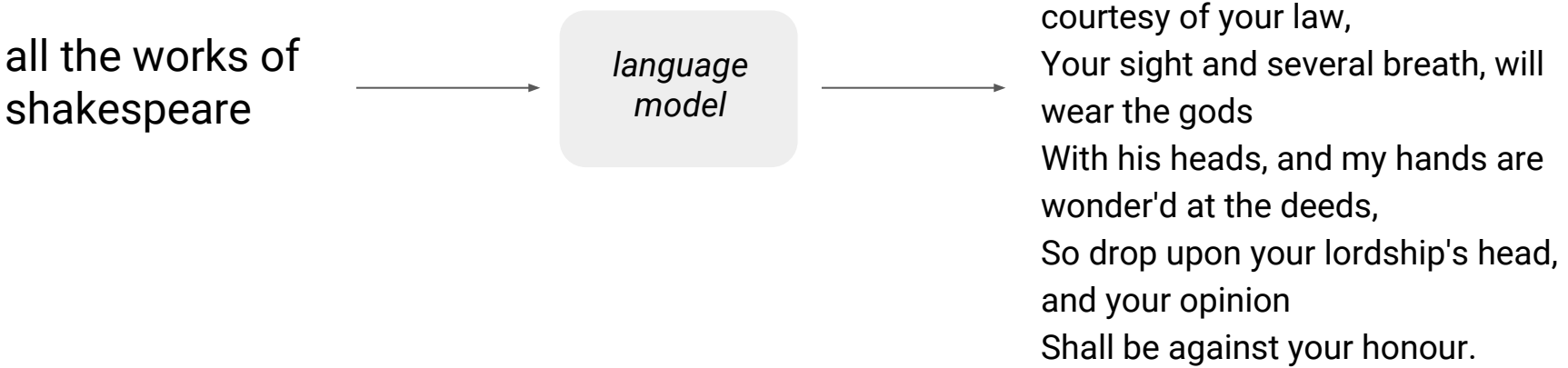
notice that W and U stay the same!

“unfolding” the RNN across time:

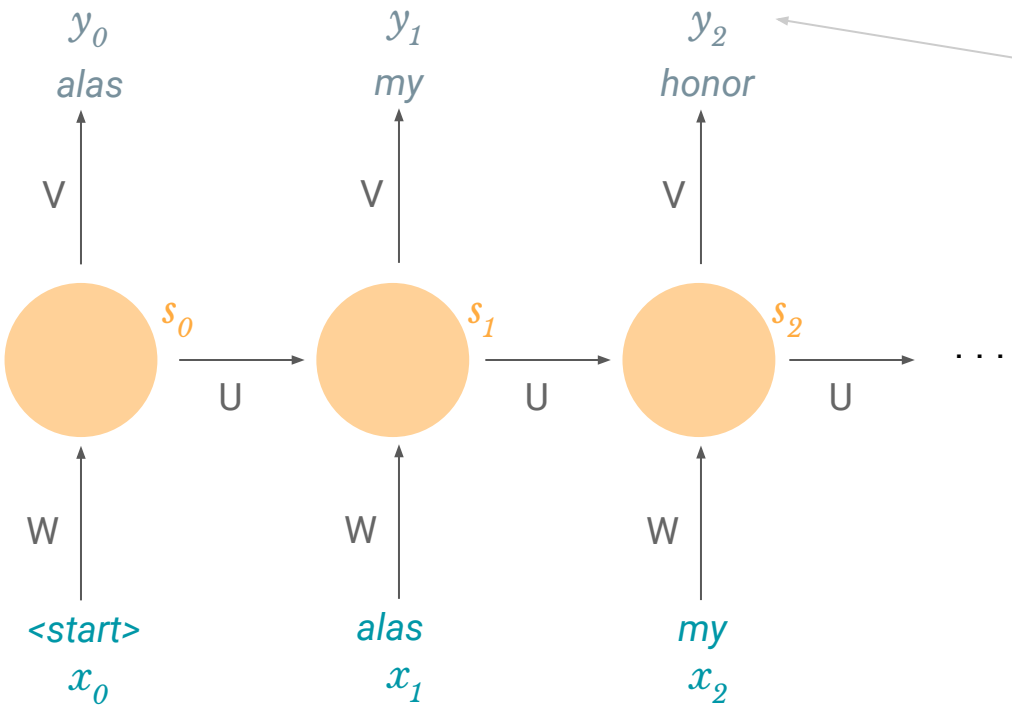


s_n can contain information from all past timesteps

possible task: language model

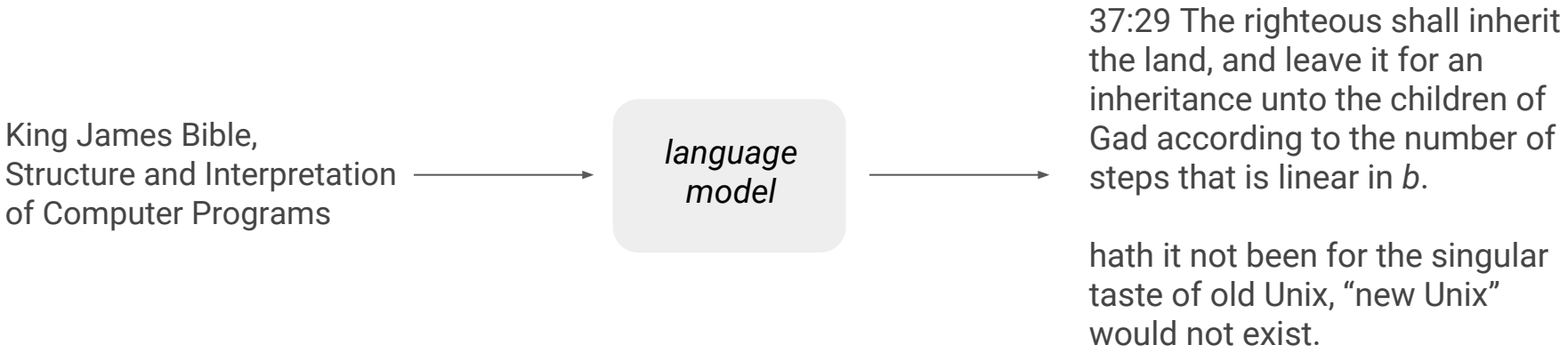


possible task: language model



y_i is actually a probability distribution over possible next words, aka a *softmax*

possible task: language model



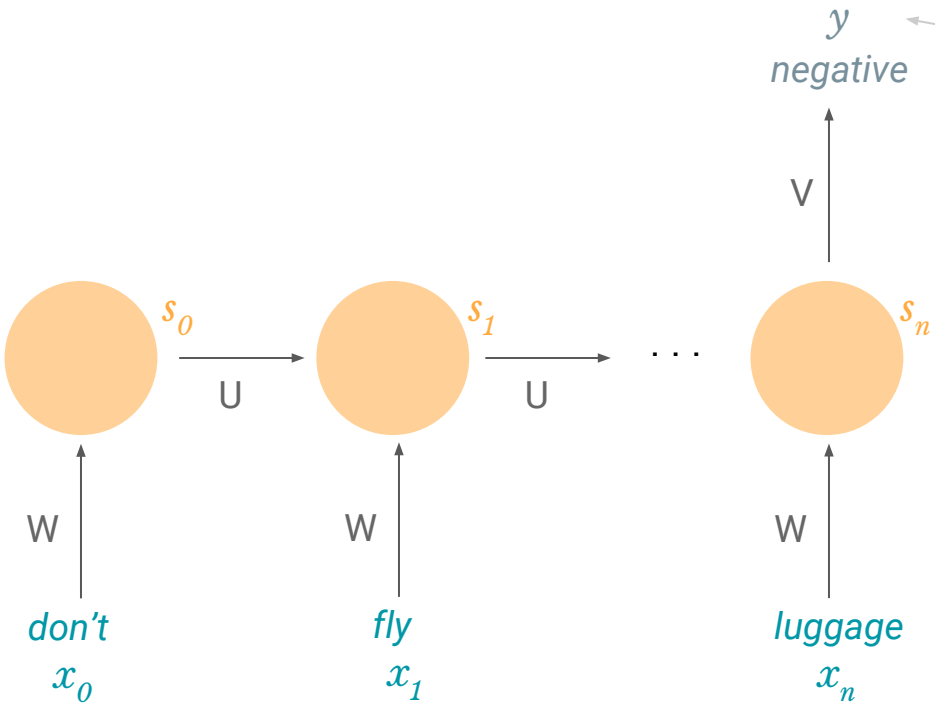
http://kingjamesprogramming.tumblr.com/

possible task: classification (i.e. sentiment)

:(

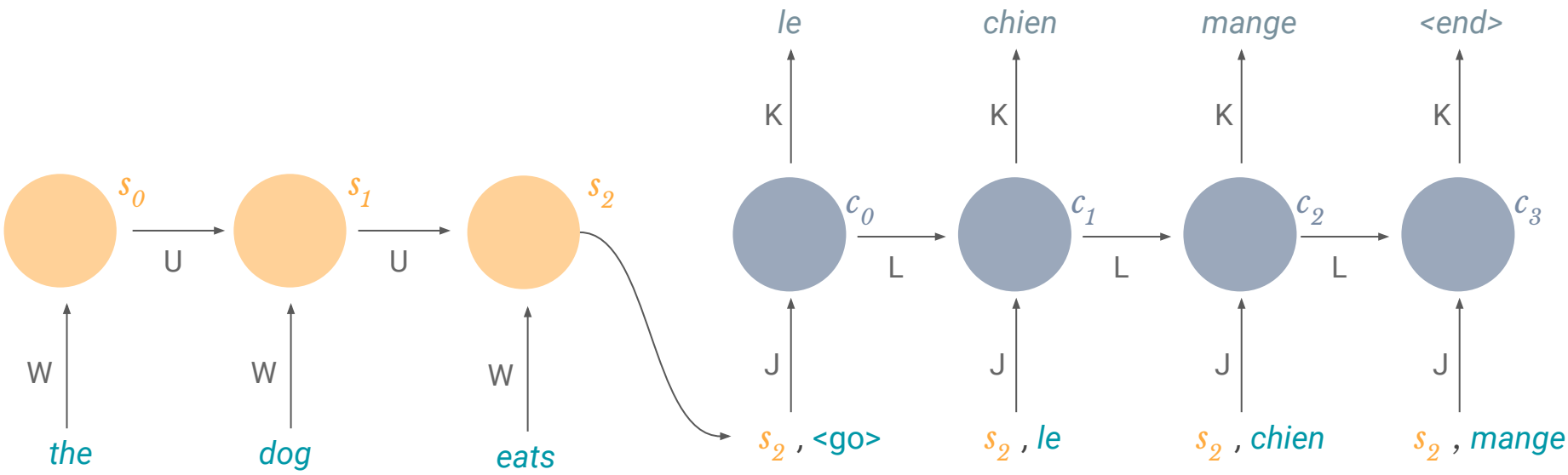
:)

possible task: classification (i.e. sentiment)



y is a probability distribution over possible classes (like positive, negative, neutral), aka a *softmax*

possible task: machine translation



how do we **train** an RNN?

how do we **train** an RNN?

backpropagation!

(through time)

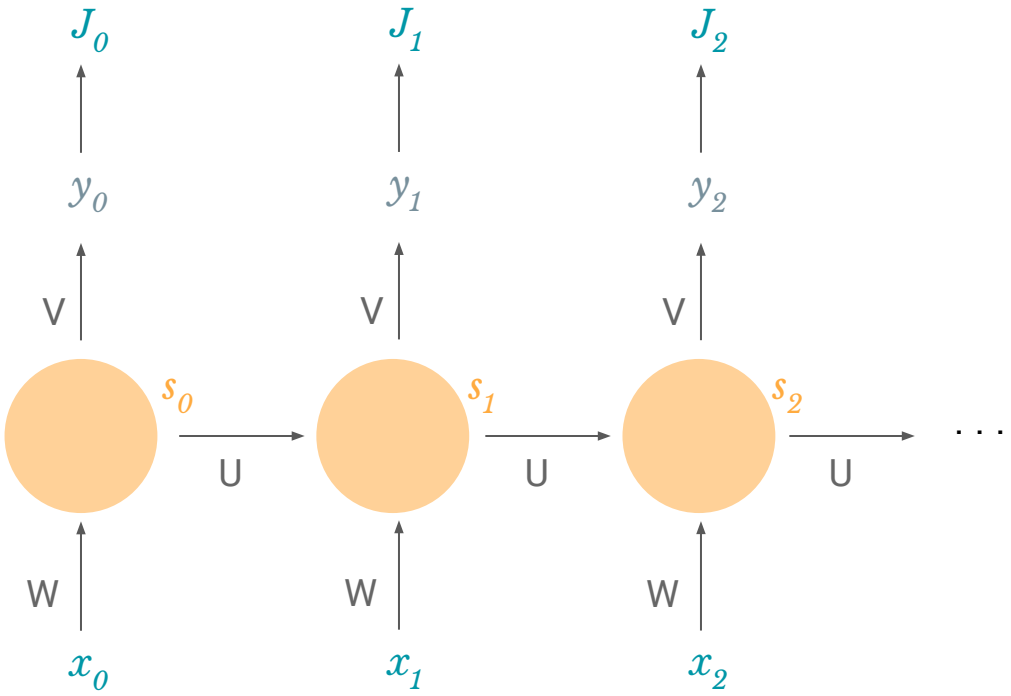
remember: **backpropagation**

1. take the derivative (gradient) of the loss with respect to each parameter

2. shift parameters in the opposite direction in order to minimize loss

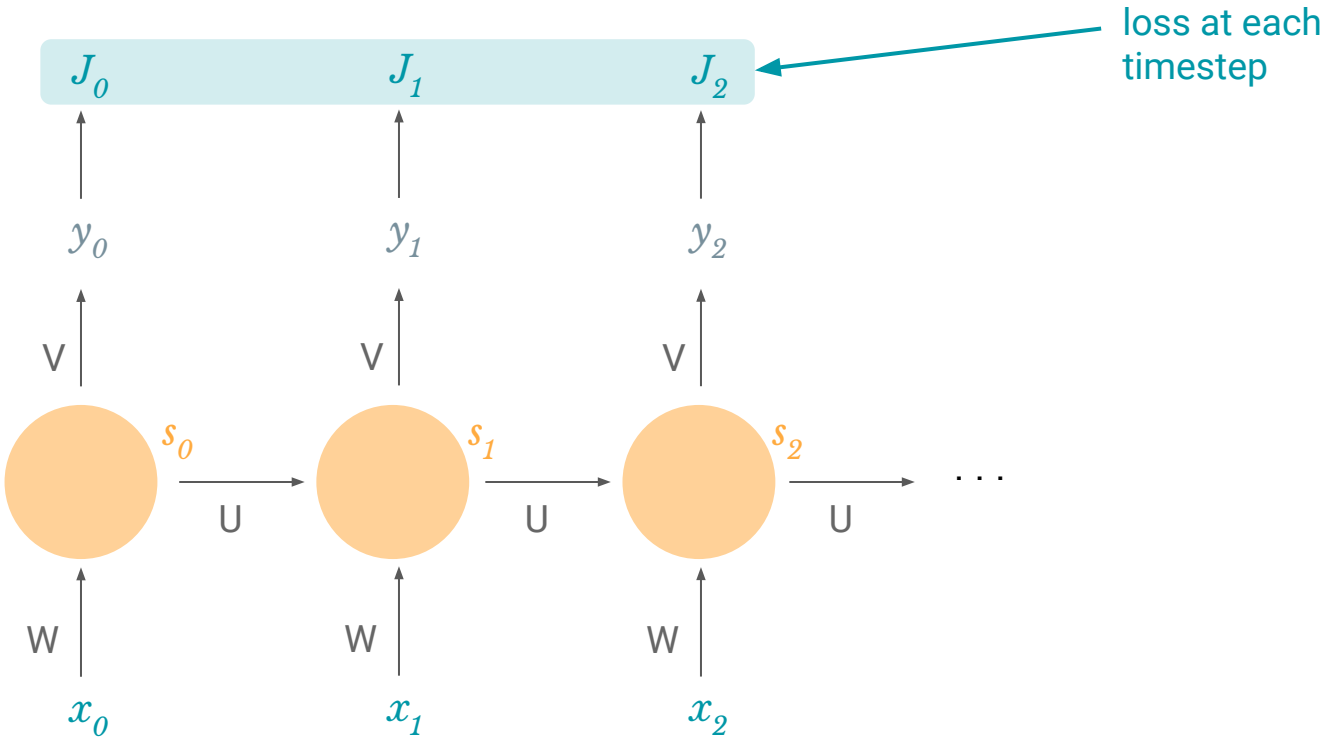
we have a **loss at each timestep**:

(since we're making a prediction at each timestep)



we have a **loss at each timestep**:

(since we're making a prediction at each timestep)



we **sum the losses** across time:

loss at time t = $J_t(\Theta)$

Θ = our parameters, like weights

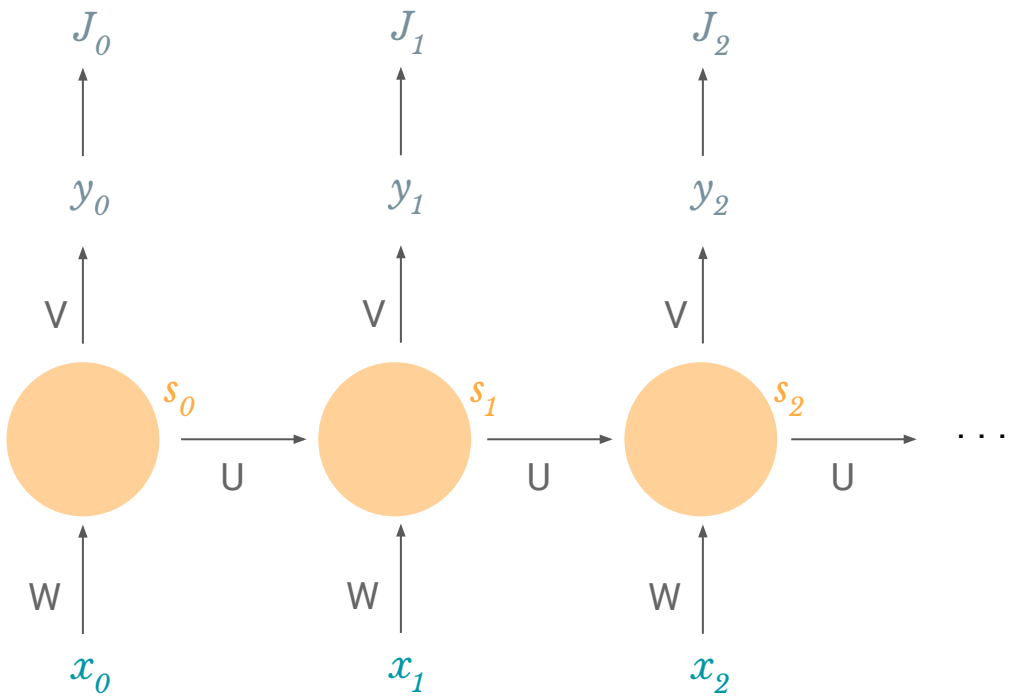
total loss = $J(\Theta) = \Sigma_t J_t(\Theta)$

what are our **gradients**?

we sum gradients across time for each parameter *P*:

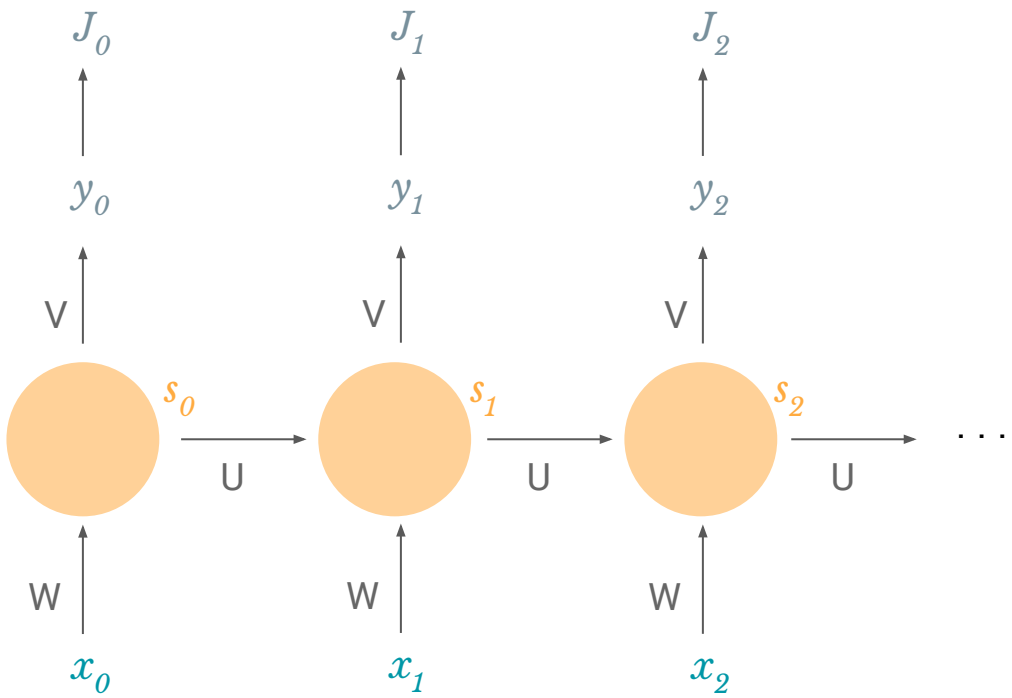
$$\frac{\partial J}{\partial P} = \sum_t \frac{\partial J_t}{\partial P}$$

let's try it out for *W* with the **chain rule**:



$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

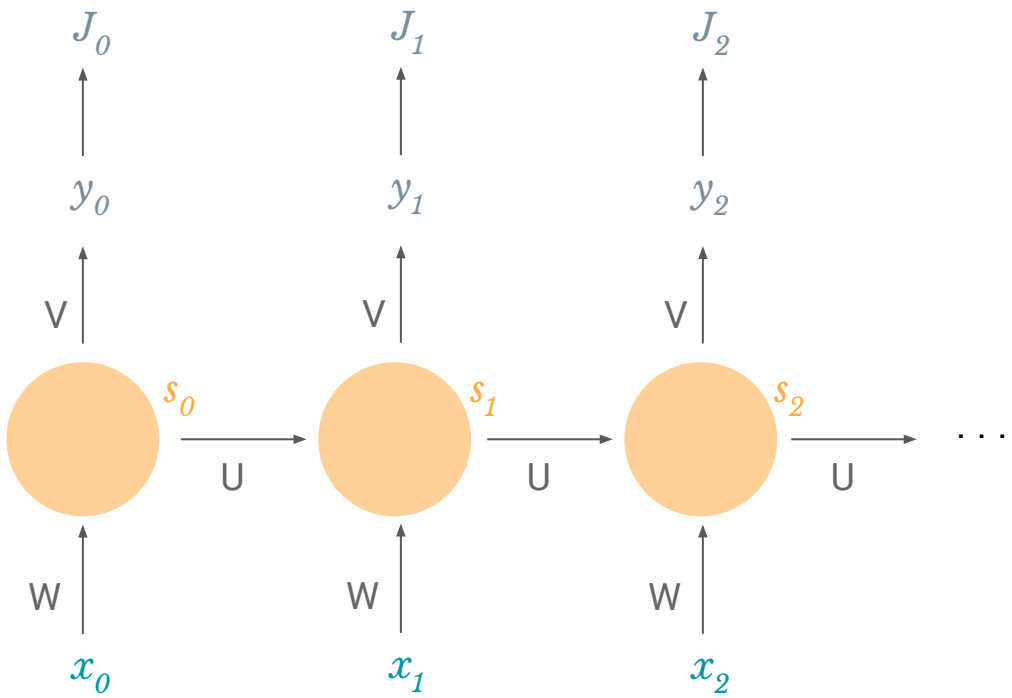
let's try it out for *W* with the **chain rule**:



$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

so let's take a single timestep *t*:

let's try it out for *W* with the **chain rule**:

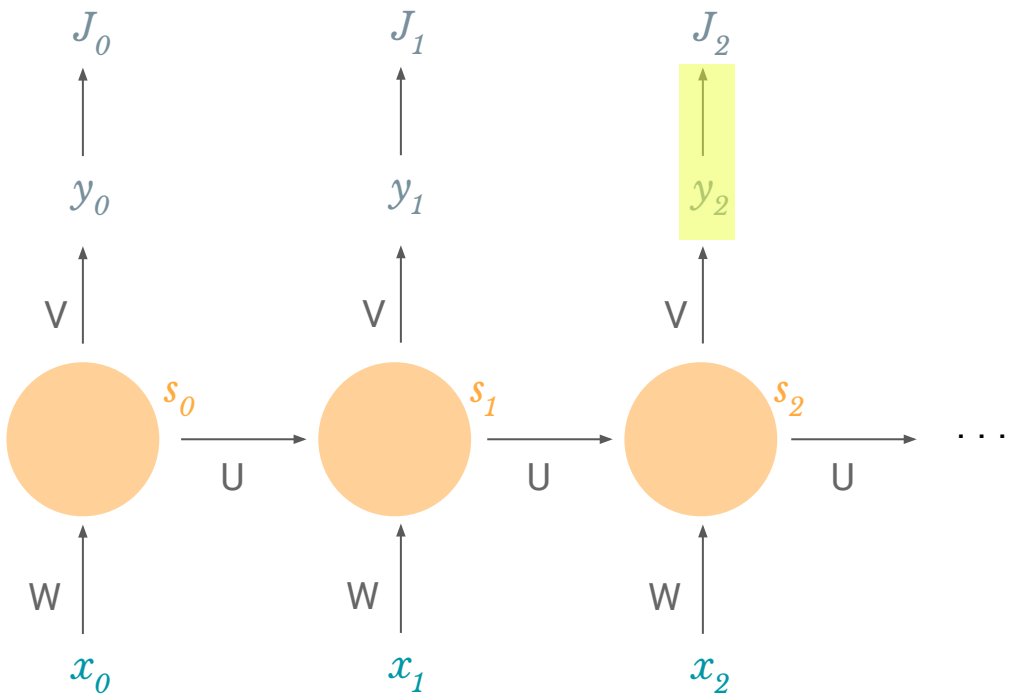


$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

so let's take a single timestep *t*:

$$\frac{\partial J_2}{\partial W}$$

let's try it out for *W* with the **chain rule**:

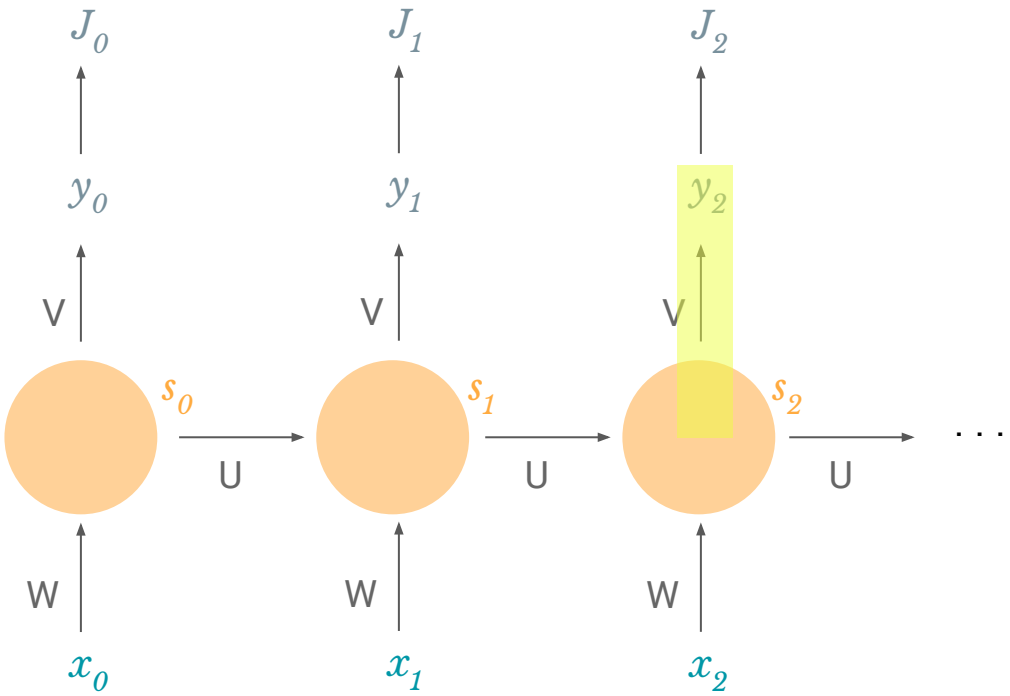


$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

so let's take a single timestep *t*:

$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2}$$

let's try it out for *W* with the **chain rule**:

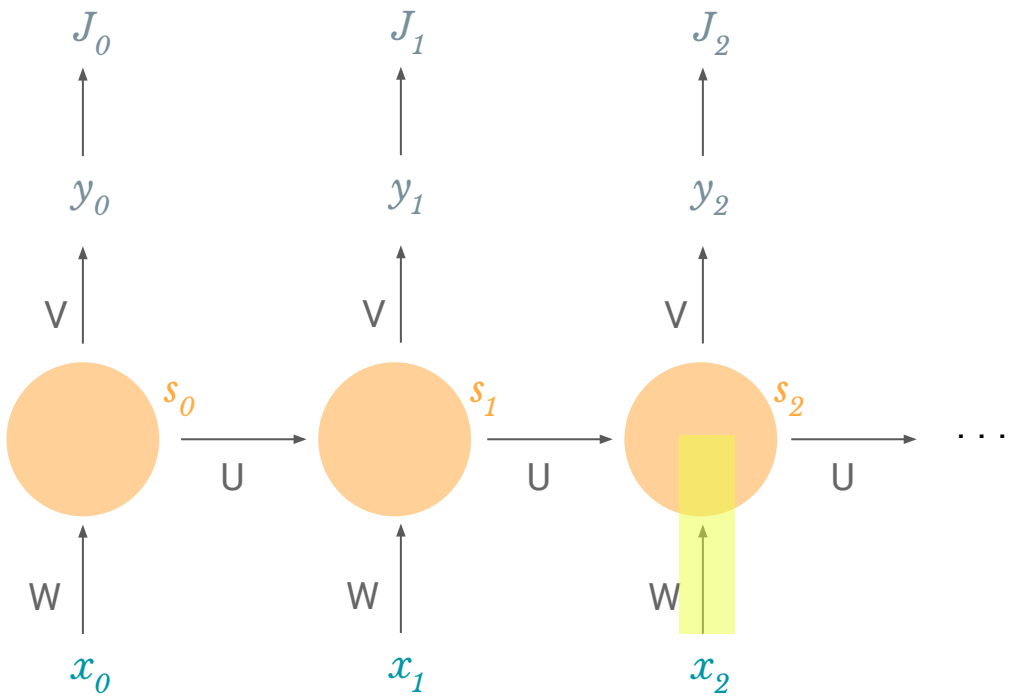


$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

so let's take a single timestep *t*:

$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2}\frac{\partial y_2}{\partial s_2}$$

let's try it out for *W* with the **chain rule**:

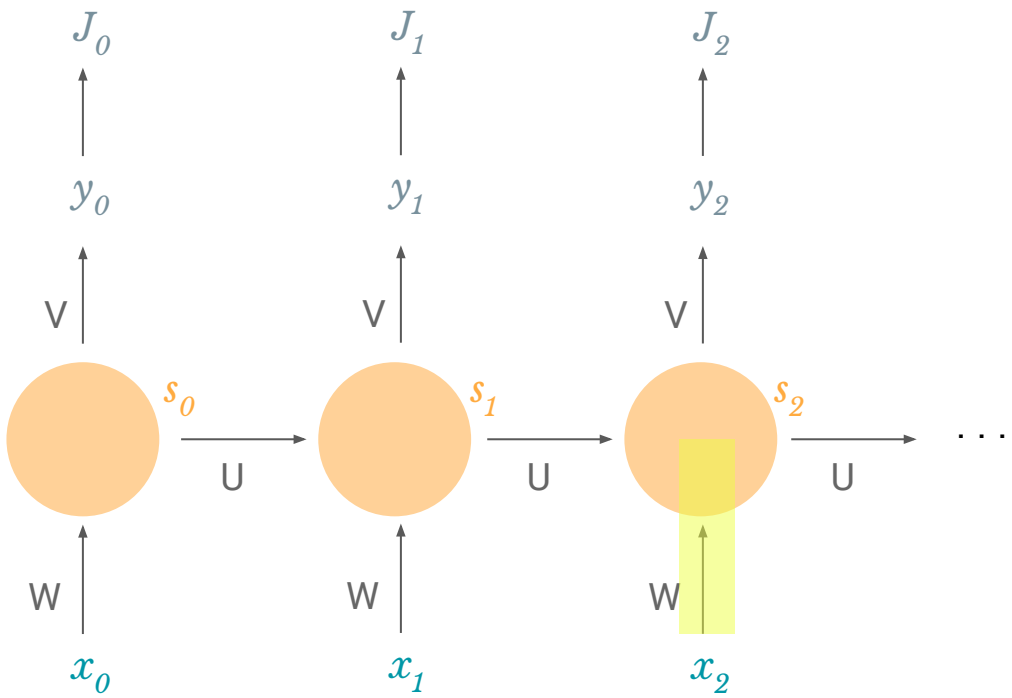


$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

so let's take a single timestep *t*:

$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2}\frac{\partial y_2}{\partial s_2}\frac{\partial s_2}{\partial W}$$

let's try it out for *W* with the **chain rule**:



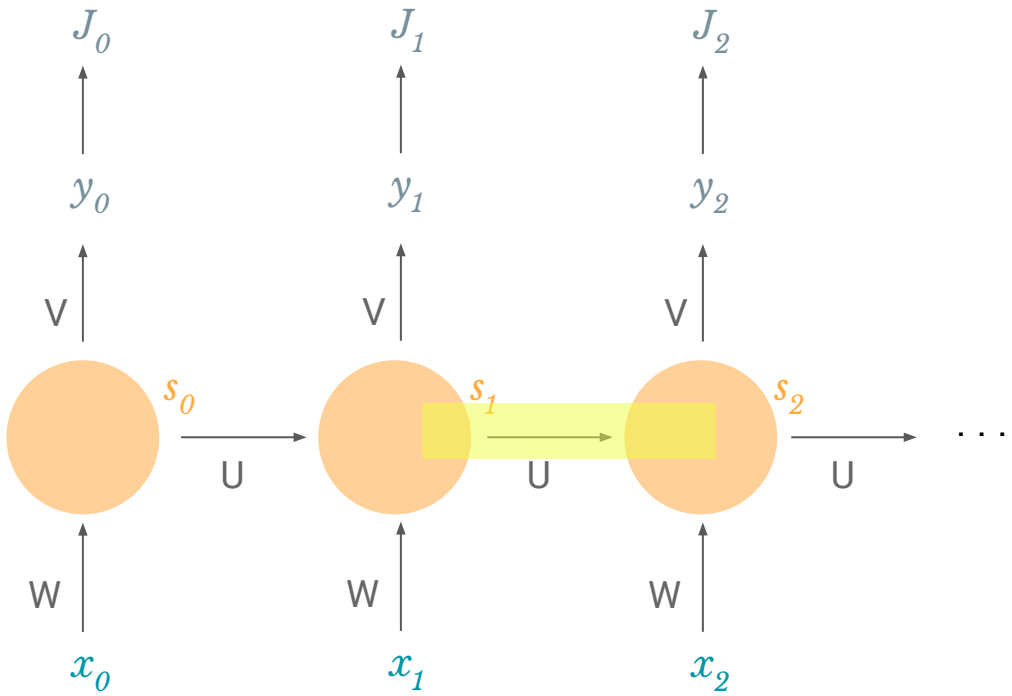
$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

so let's take a single timestep *t*:

$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2}\frac{\partial y_2}{\partial s_2}\frac{\partial s_2}{\partial W}$$

but wait…

let's try it out for *W* with the **chain rule**:



$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

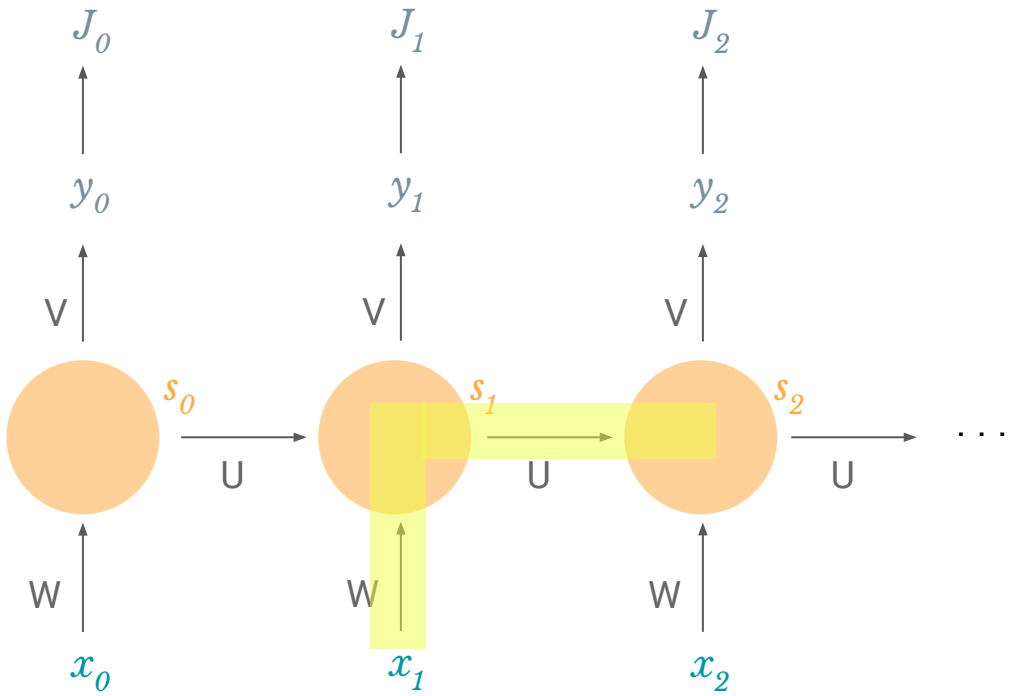
so let's take a single timestep *t*:

$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2}\frac{\partial y_2}{\partial s_2}\frac{\partial s_2}{\partial W}$$

but wait...

$$s_2 = tanh(U s_1 + W x_2)$$

let's try it out for *W* with the **chain rule**:



$$\frac{\partial J}{\partial W} = \sum_t \frac{\partial J_t}{\partial W}$$

so let's take a single timestep *t*:

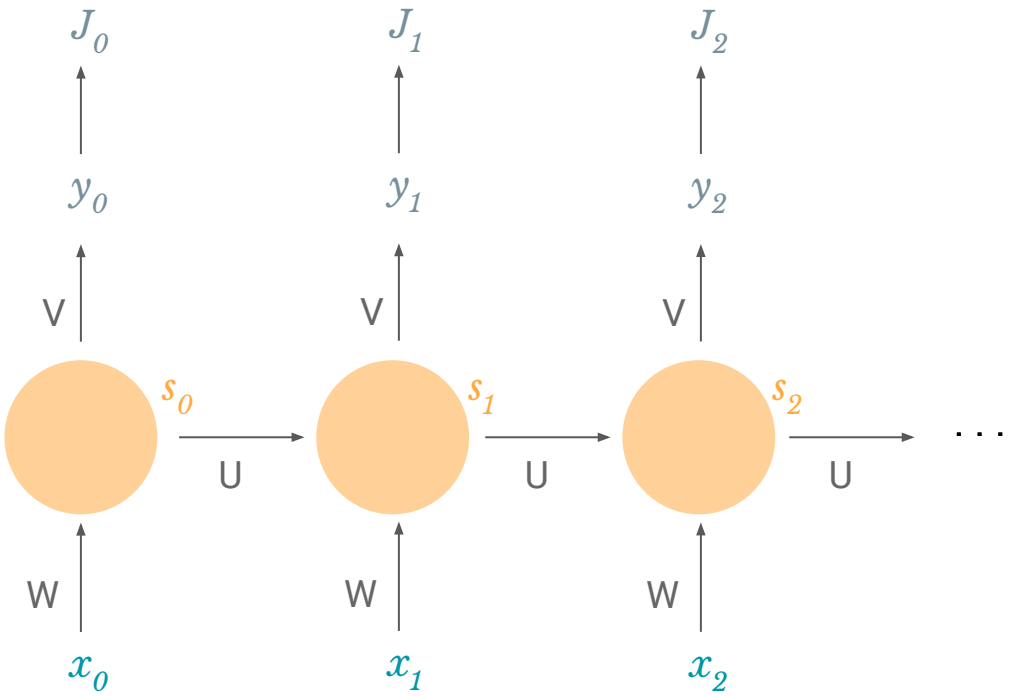
$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2}\frac{\partial y_2}{\partial s_2}\frac{\partial s_2}{\partial W}$$

but wait...

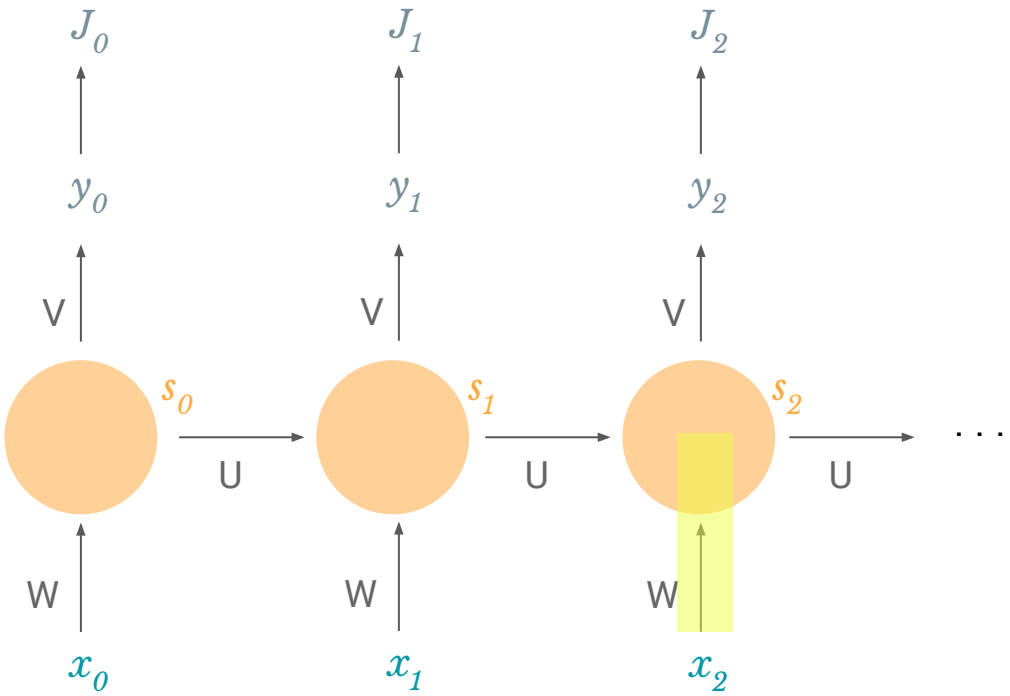
$$s_2 = tanh(U s_1 + W x_2)$$

s_1 also depends on *W* so we can't just treat $\frac{\partial s_2}{\partial W}$ as a constant!

how does s_2 depend on W?

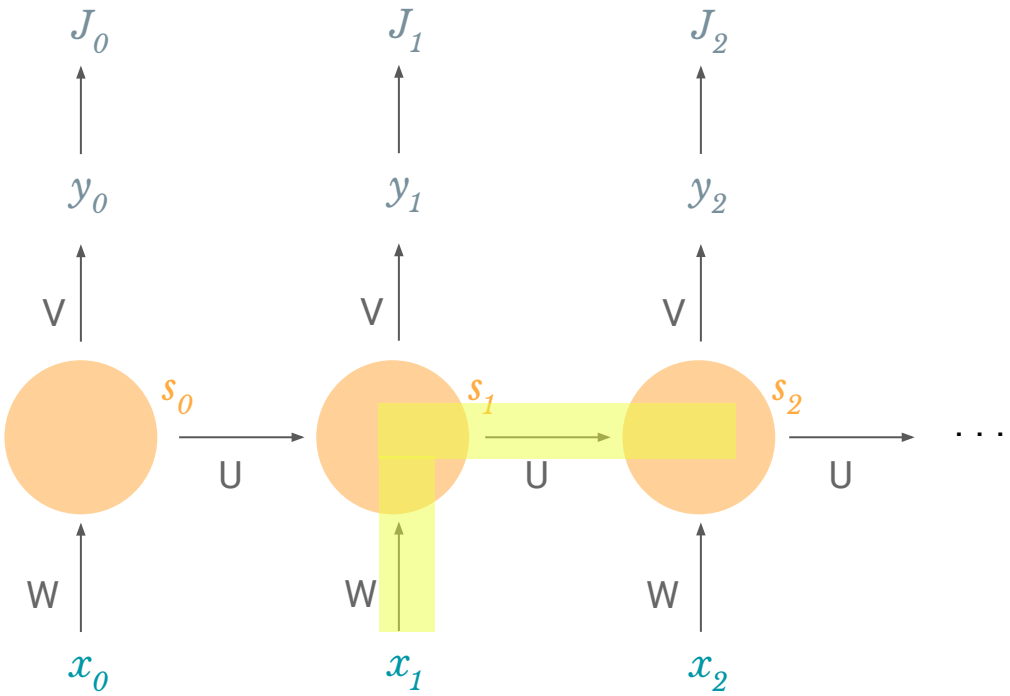


how does s_2 depend on W?



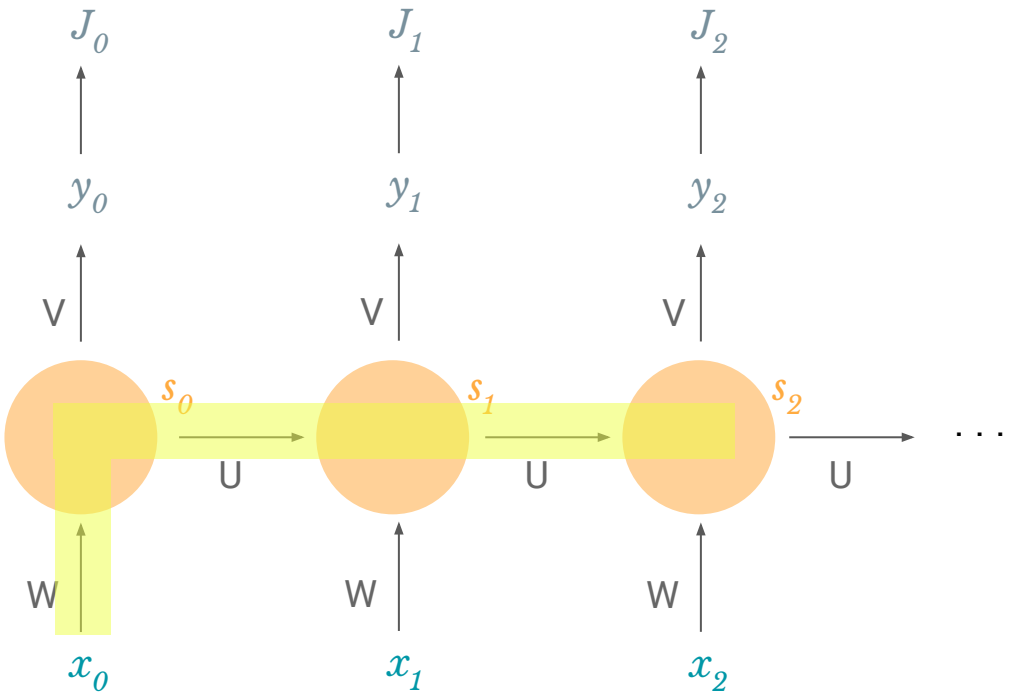
$$\frac{\partial s_2}{\partial W}$$

how does s_2 depend on W?



$$\frac{\partial s_2}{\partial W} + \frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W}$$

how does s_2 depend on W?



$$\frac{\partial s_2}{\partial W} + \frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W} + \frac{\partial s_2}{\partial s_0}\frac{\partial s_0}{\partial W}$$

backpropagation through time:

$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^{2} \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \underbrace{\frac{\partial s_2}{\partial s_k} \frac{\partial s_k}{\partial W}}$$

Contributions of *W* in previous timesteps to the error at timestep *t*

backpropagation through time:

$$\frac{\partial J_t}{\partial W} = \sum_{k=0}^{t} \frac{\partial J_t}{\partial y_t} \frac{\partial y_t}{\partial s_t} \underbrace{\frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W}}$$

Contributions of *W* in previous timesteps to the error at timestep *t*

why are RNNs **hard to train**?

problem: vanishing gradient

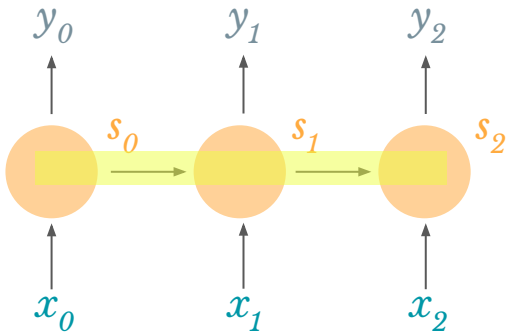
$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^{2} \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial s_k} \frac{\partial s_k}{\partial W}$$

problem: vanishing gradient

$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^{2} \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial s_k} \frac{\partial s_k}{\partial W}$$

problem: vanishing gradient

$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^{2} \frac{\partial J_2}{\partial y_2} \frac{\partial y_2}{\partial s_2} \frac{\partial s_2}{\partial s_k} \frac{\partial s_k}{\partial W}$$



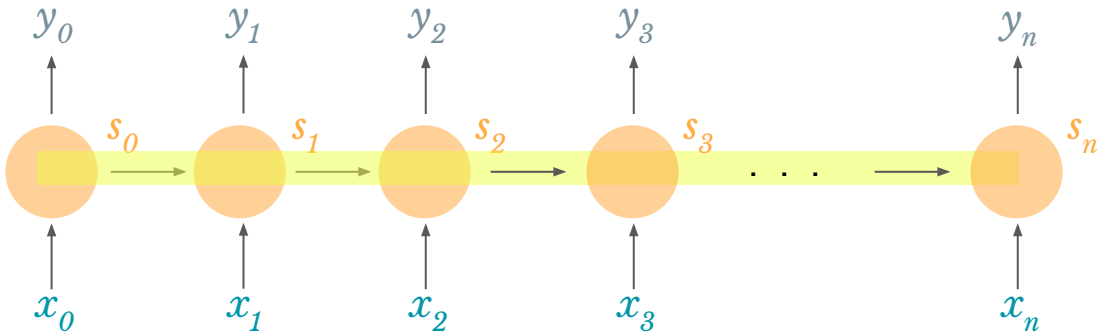
at $k = 0$:

$$\frac{\partial s_2}{\partial s_0} = \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

problem: vanishing gradient

$$\frac{\partial J_n}{\partial W} = \sum_{k=0}^{n} \frac{\partial J_n}{\partial y_n} \frac{\partial y_n}{\partial s_n} \frac{\partial s_n}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

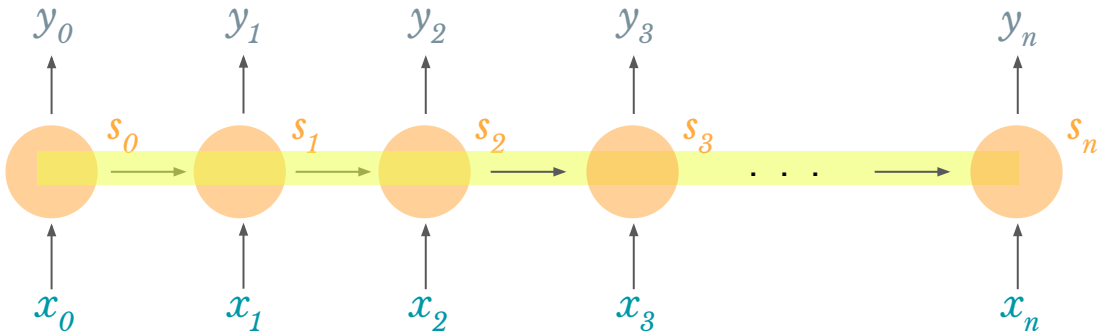


problem: vanishing gradient

$$\frac{\partial J_n}{\partial W} = \sum_{k=0}^{n} \frac{\partial J_n}{\partial y_n} \frac{\partial y_n}{\partial s_n} \frac{\partial s_n}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

as the gap between timesteps gets bigger, this product gets longer and longer!



problem: vanishing gradient

$$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

problem: vanishing gradient

what are each of these terms?

$$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

problem: vanishing gradient

what are each of these terms?

$$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

$$\frac{\partial s_n}{\partial s_{n-1}} = W^T \, diag[f'(W_{s_{j-1}} + Ux_j)]$$

W = sampled from standard normal distribution = mostly < 1

f = tanh or sigmoid so **f'** < 1

problem: vanishing gradient

what are each of these terms?

$$\frac{\partial s_n}{\partial s_{n-1}} \frac{\partial s_{n-1}}{\partial s_{n-2}} \cdots \frac{\partial s_3}{\partial s_2} \frac{\partial s_2}{\partial s_1} \frac{\partial s_1}{\partial s_0}$$

$$\frac{\partial s_n}{\partial s_{n-1}} = W^T \, diag[f'(W_{s_{j-1}} + Ux_j)]$$

W = sampled from standard normal distribution = mostly < 1

f = tanh or sigmoid so **f'** < 1

we're multiplying a lot of **small numbers** together.

we’re multiplying a lot of **small numbers** together.

so what?

errors due to further back timesteps have increasingly **smaller gradients.**

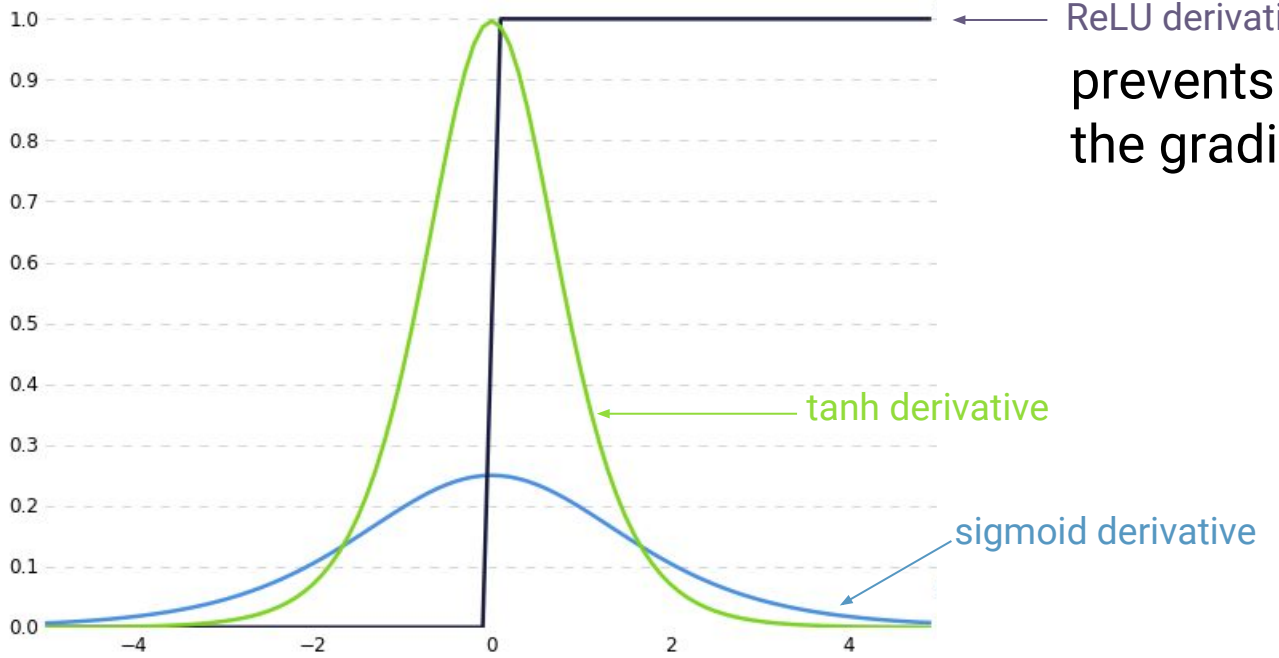
so what?

parameters become biased to **capture shorter-term** dependencies.

“In France, I had a great time and I learnt some of the _____ language.”

our parameters are not trained to capture long-term dependencies, so the word we predict will mostly depend on the previous few words, not much earlier ones

solution #1: **activation functions**



prevents f’ from shrinking the gradients

solution #2: **initialization**

weights initialized to identity matrix

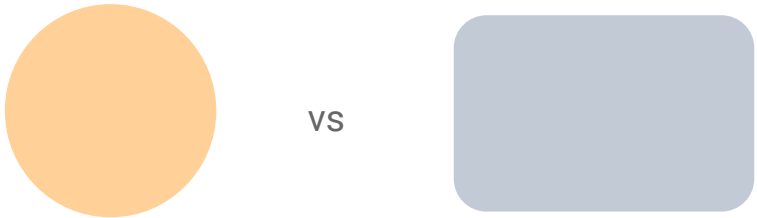
biases initialized to zeros

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix}$$

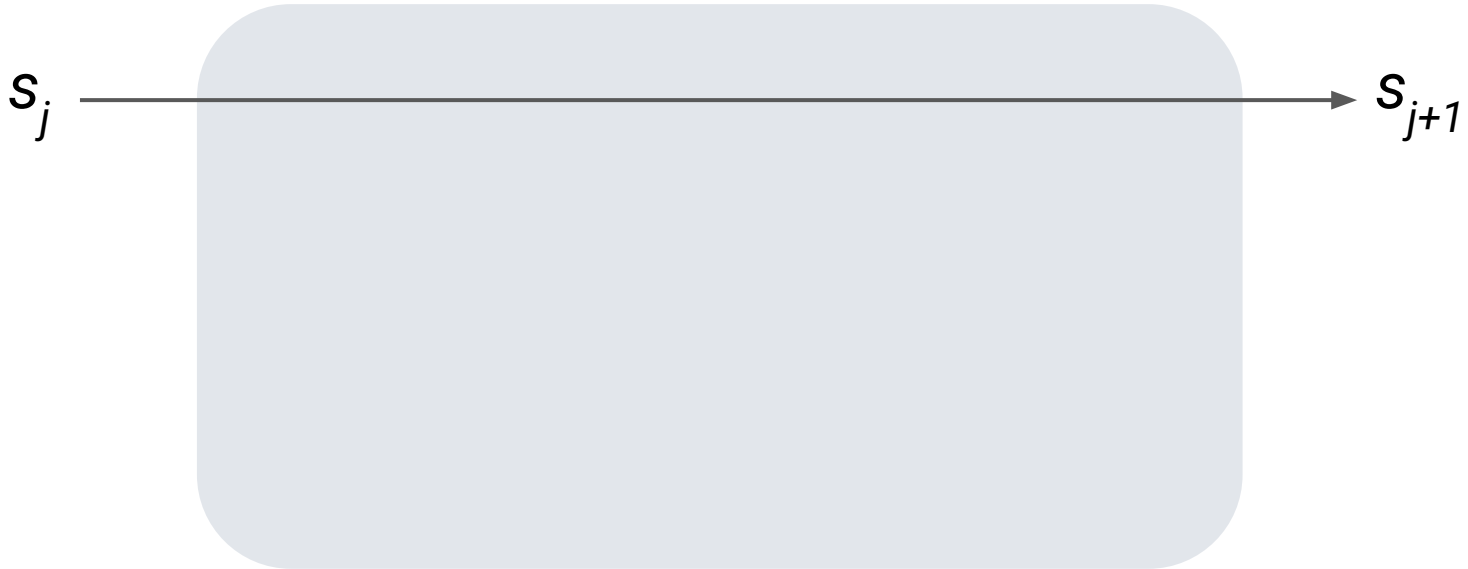
prevents W from shrinking the gradients

solution #3: **gated cells**

rather each node being just a simple RNN cell, make each node a more **complex unit with gates** controlling what information is passed through.

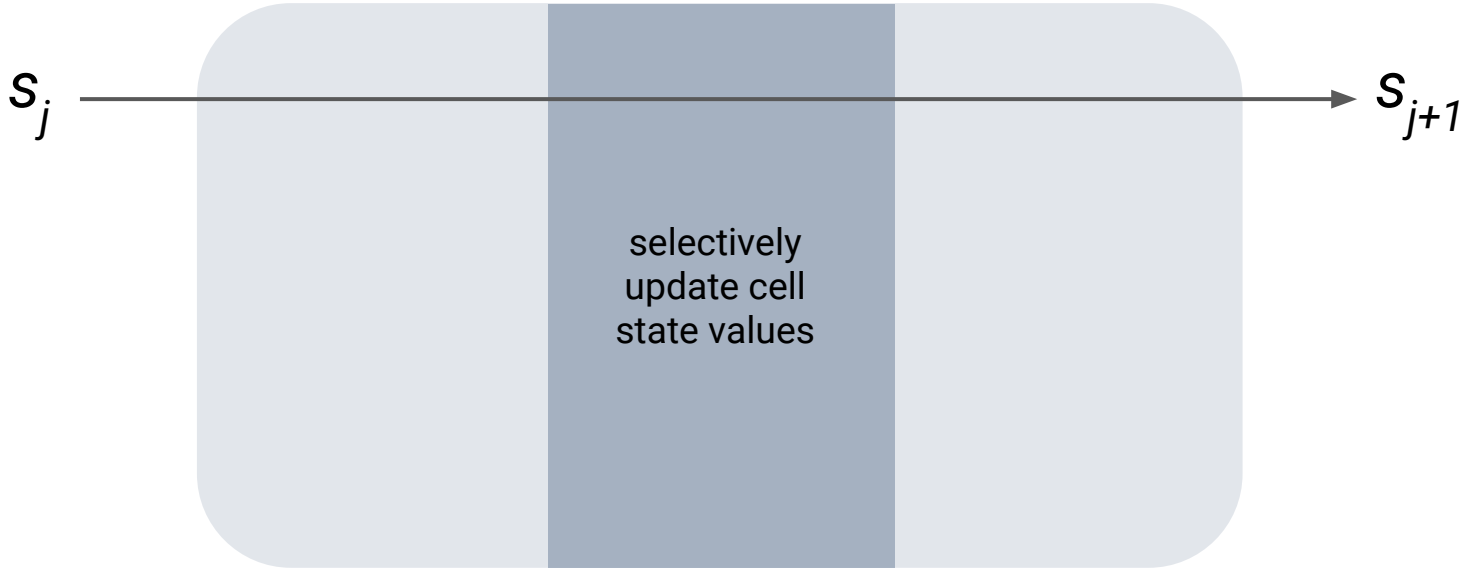


solution #3: more on **LSTMs**

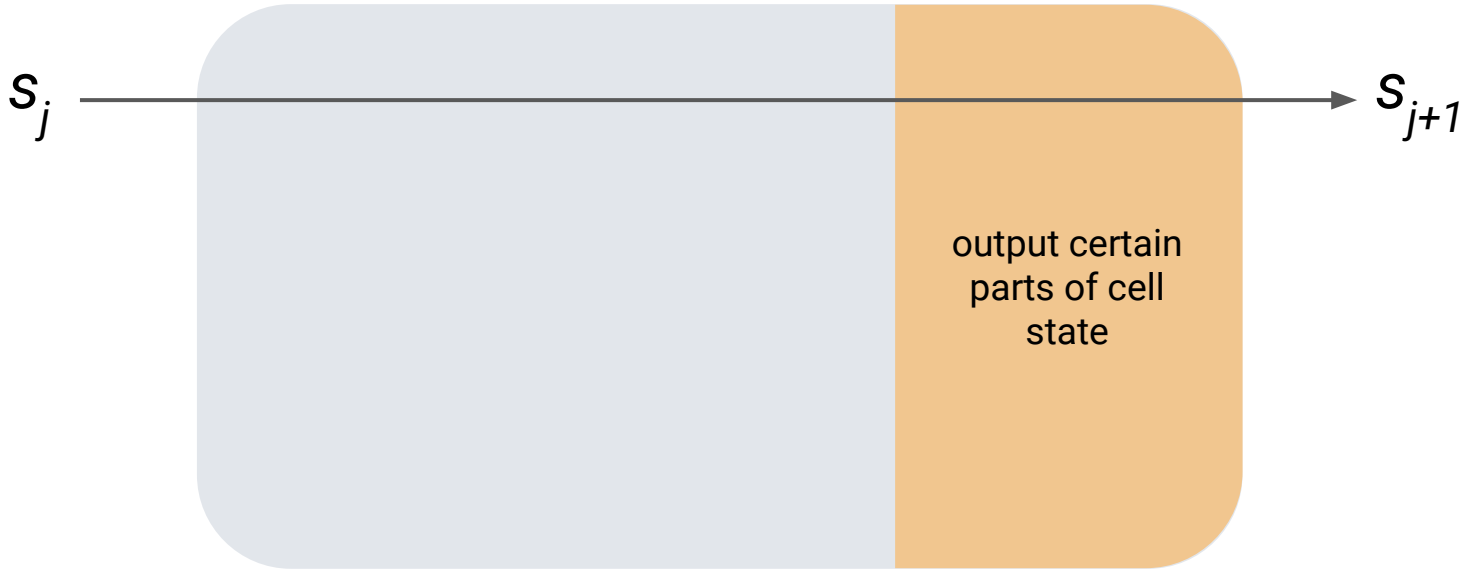


solution #3: more on **LSTMs**

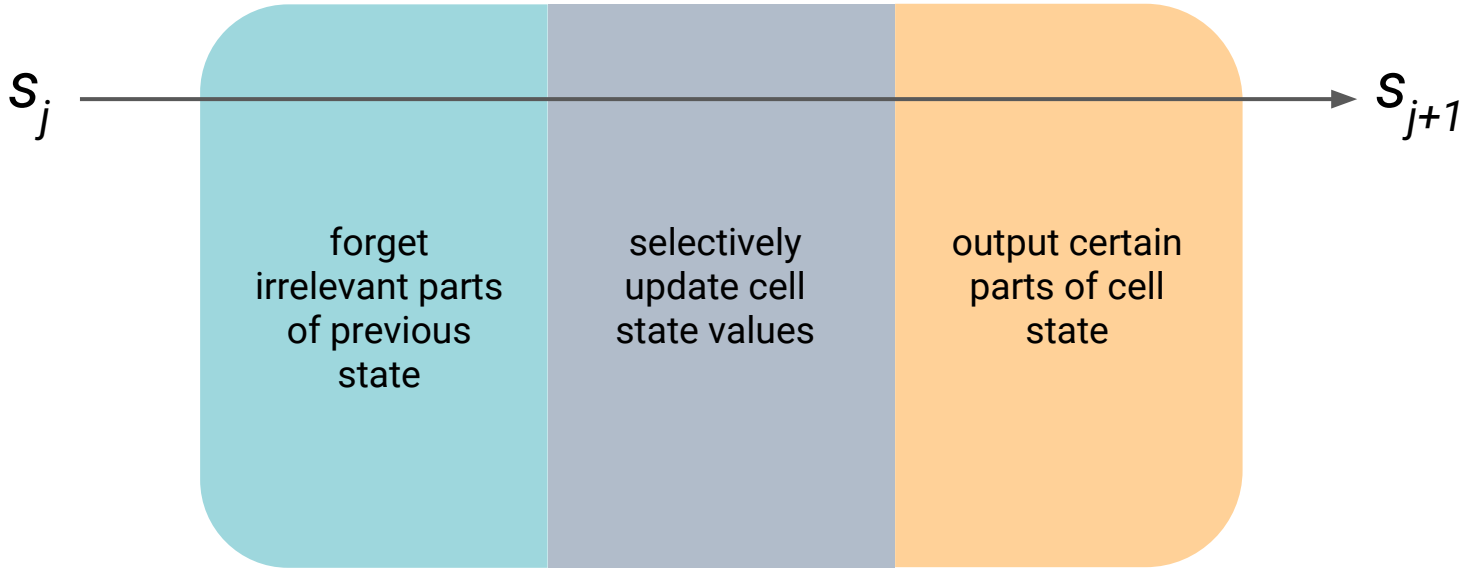
solution #3: more on **LSTMs**



solution #3: more on **LSTMs**



solution #3: more on **LSTMs**



why do LSTMs help?

1. forget gate allows information to pass through unchanged

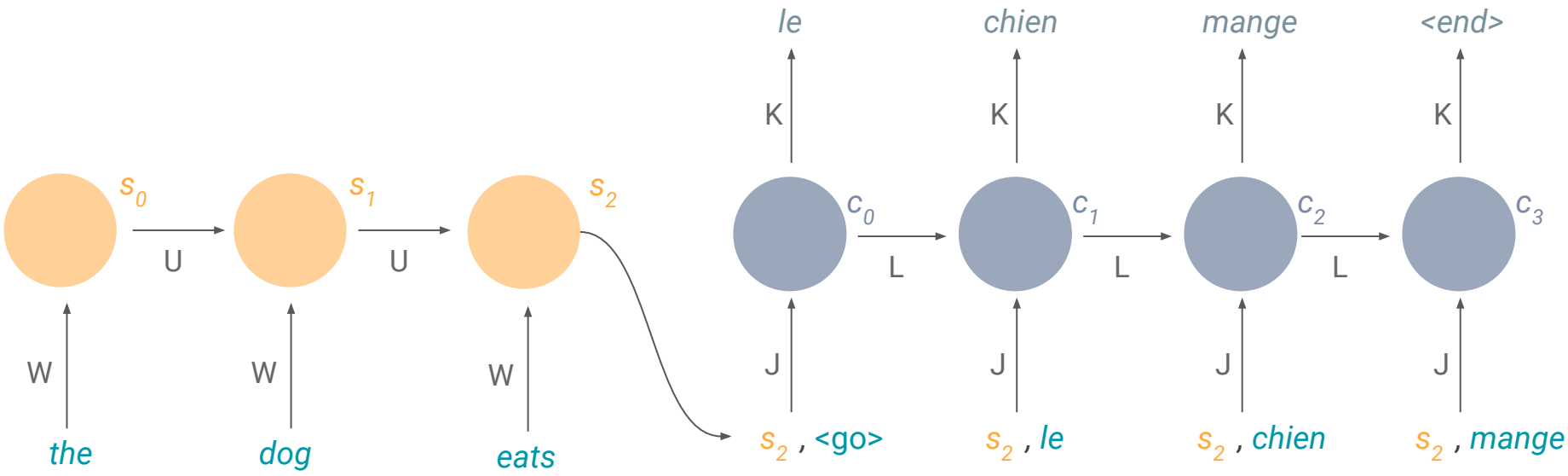
   → when taking the derivative, **f'** is 1 for what we want to keep!

2. s_j depends on s_{j-1} through addition!

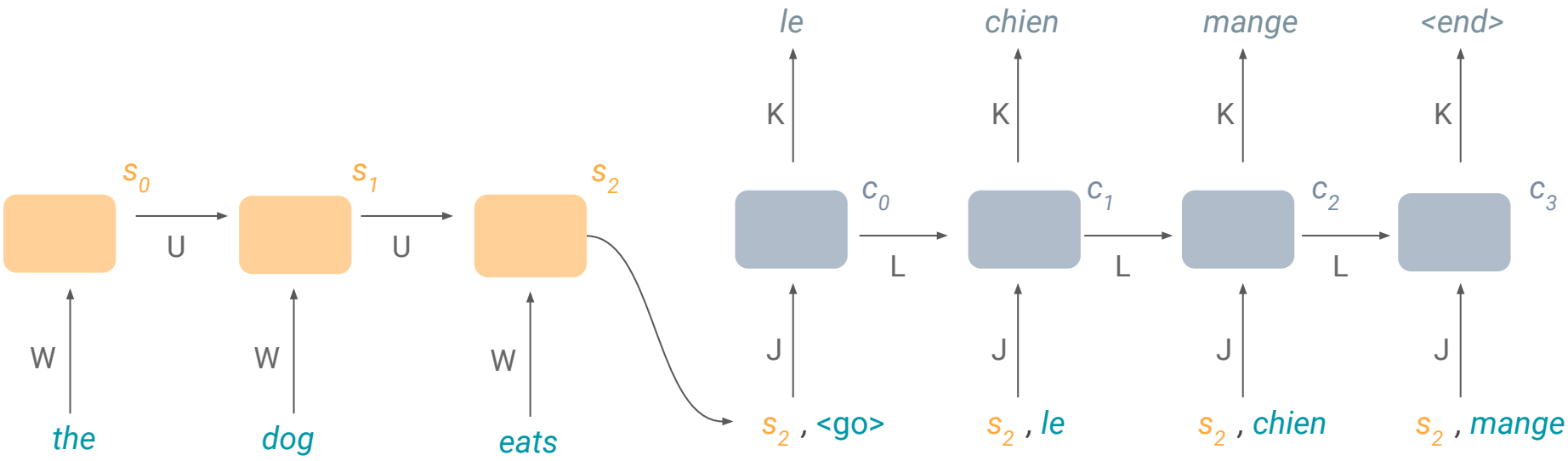
   → when taking the derivative, not lots of small **W** terms!

in practice: **machine translation**.

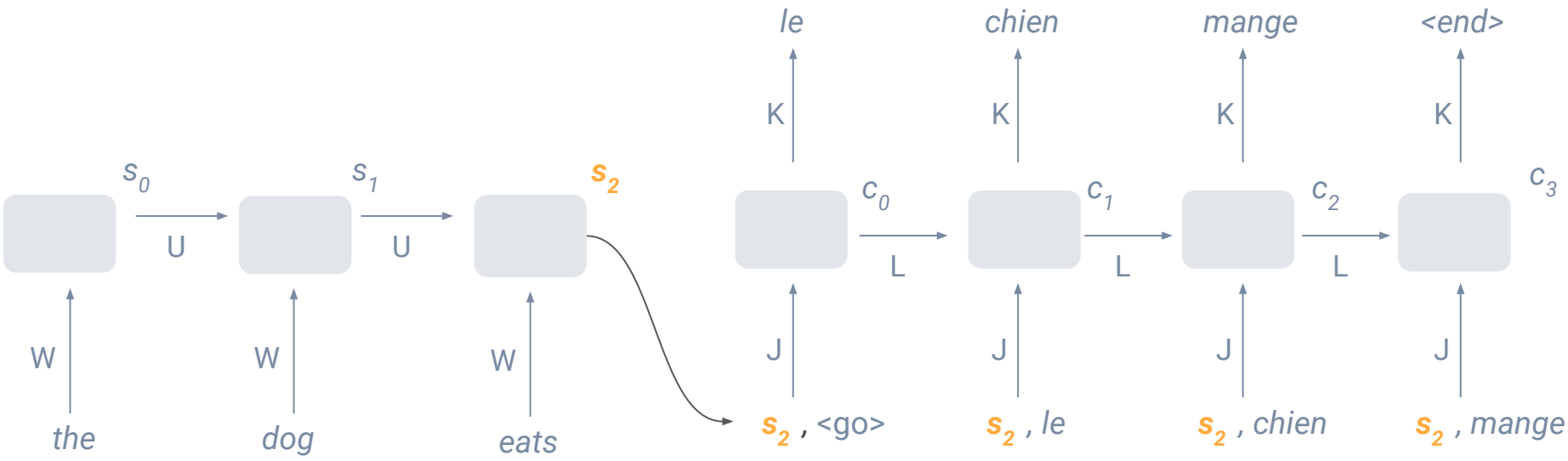
basic encoder-decoder model:



add LSTM cells:

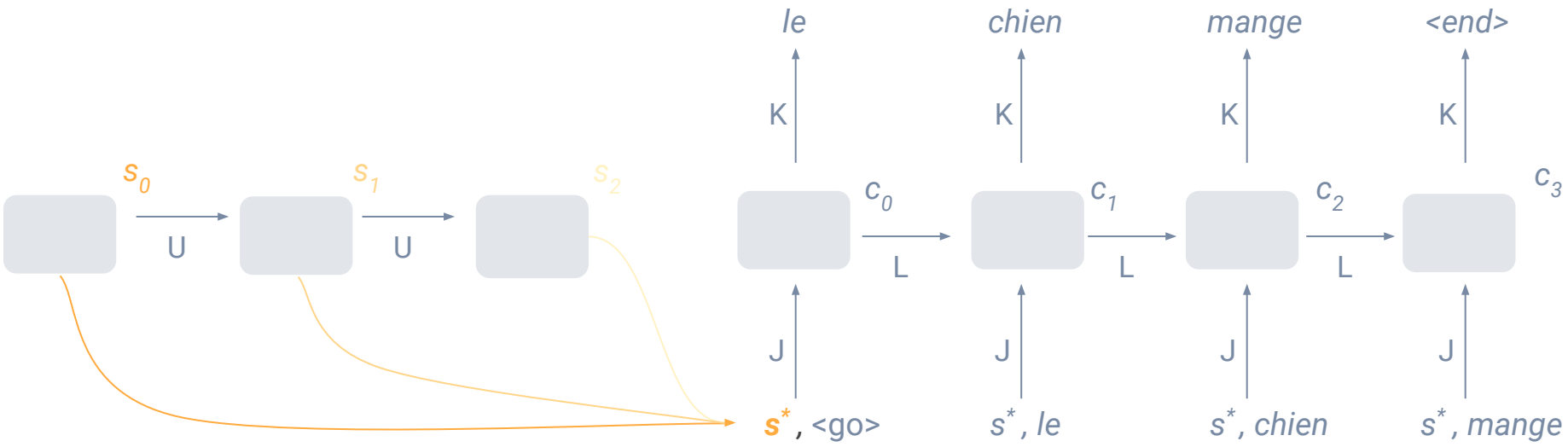


problem: a fixed-length encoding is limiting

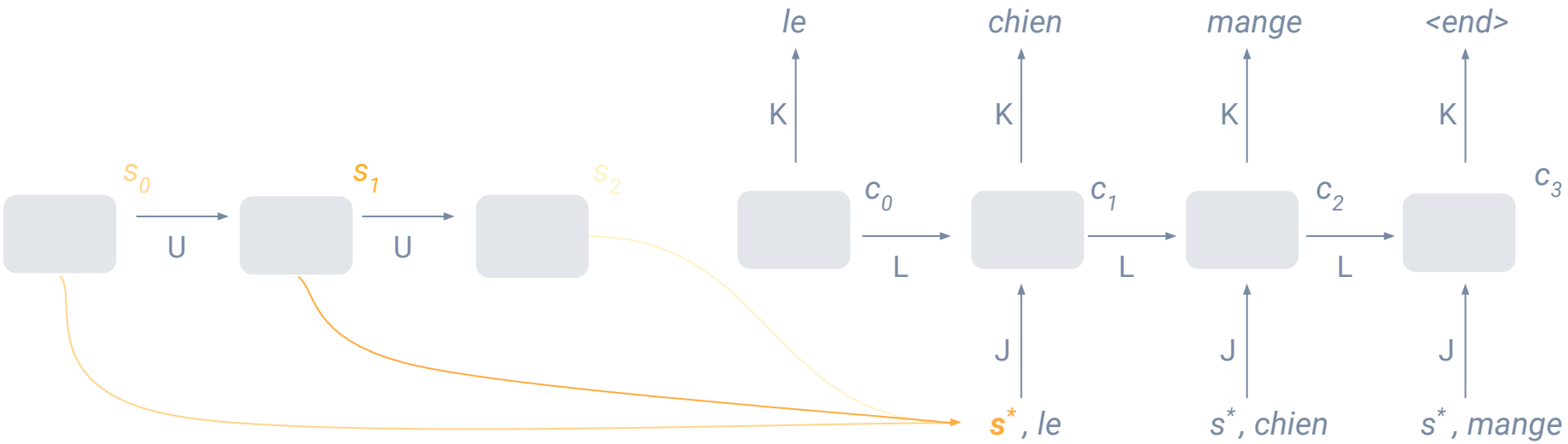


all the decoder knows about the input sentence is in one fixed length vector, s_2

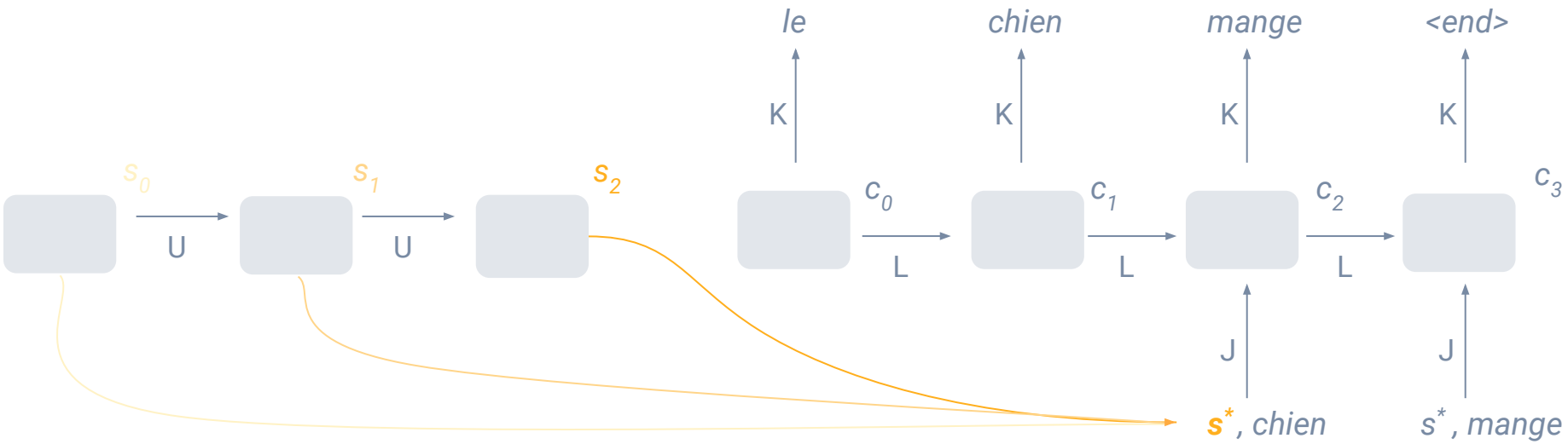
solution: attend over all encoder states



solution: attend over all encoder states



solution: attend over all encoder states



now we can model **sequences**!

- why recurrent neural networks?
- building models for language, classification, and machine translation
- training them with backpropagation through time
- solving the vanishing gradient problem with activation functions, initialization, and gated cells (like LSTMs)
- using attention mechanisms

and there’s lots more to do!

- extending our models to timeseries + waveforms
- complex language models to generate long text or books
- language models to generate code
- controlling cars + robots
- predicting stock market trends
- summarizing books + articles
- handwriting generation
- multilingual translation models
- … many more!

Using TensorFlow

Deep Learning Frameworks

- GPU Acceleration
- Automatic Differentiation
- Code Reusability + Extensibility
- Speed up Idea -> Implementation

Whats out there?

What is a Tensor?

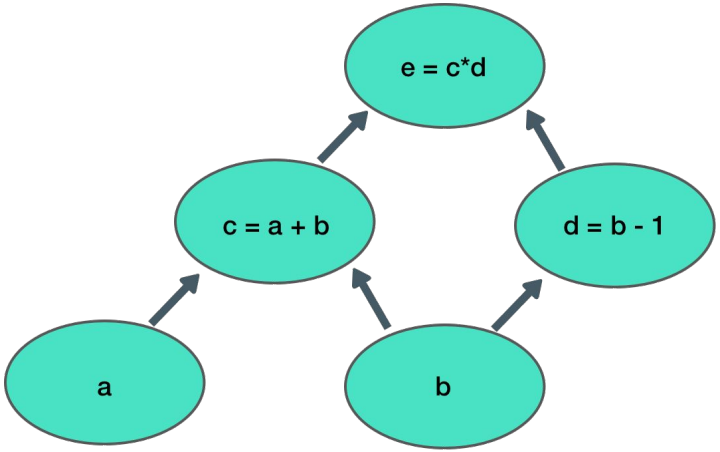
- Tensorflow Tensors are very similar to numpy ndarrays

Numpy	**TensorFlow**
`a = np.zeros((2,2)); b = np.ones((2,2))`	`a = tf.zeros((2,2)), b = tf.ones((2,2))`
`np.sum(b, axis=1)`	`tf.reduce_sum(a,reduction_indices=[1])`
`a.shape`	`a.get_shape()`
`np.reshape(a, (1,4))`	`tf.reshape(a, (1,4))`
`b * 5 + 1`	`b * 5 + 1`
`np.dot(a,b)`	`tf.matmul(a, b)`
`a[0,0], a[:,0], a[0,:]`	`a[0,0], a[:,0], a[0,:]`

https://cs224d.stanford.edu/lectures/CS224d-Lecture7.pdf

TensorFlow Basics

- Create a session
- Define a computation Graph
- Feed your data in, get results out



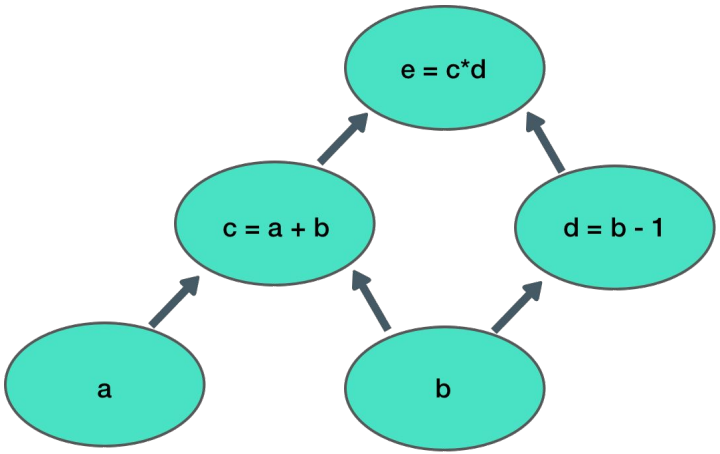
Sessions

- Encapsulates environment to run graph
- How to create the session

```
import tensorflow as tf
session = tf.InteractiveSession()
or
session = tf.Session()
```

What is a graph

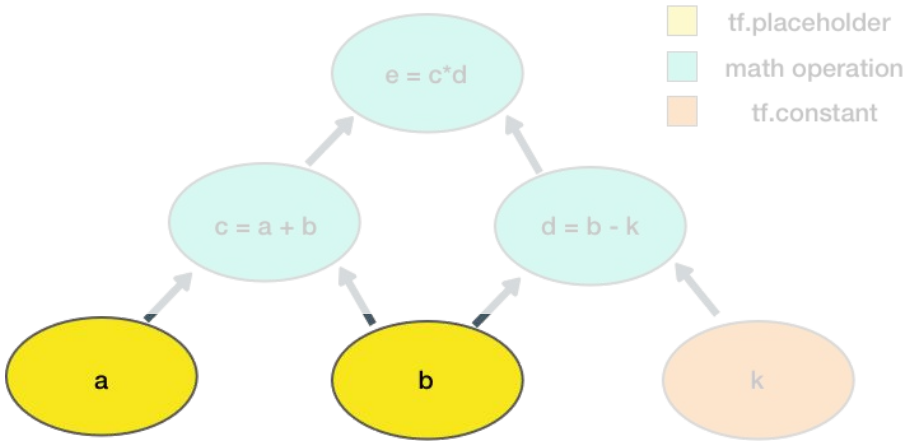
- Encapsulates the computation you want to perform



What are graphs made of?

- Placeholders (aka Graph Inputs)

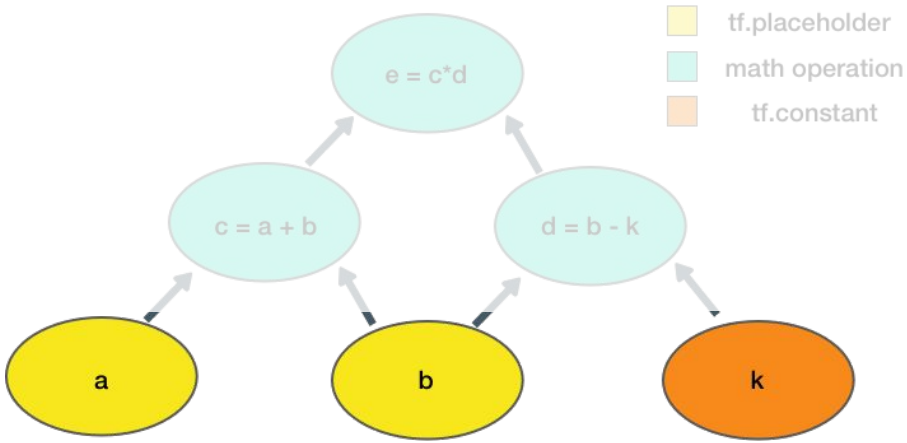
```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
```



What are graphs made of?

- Constants

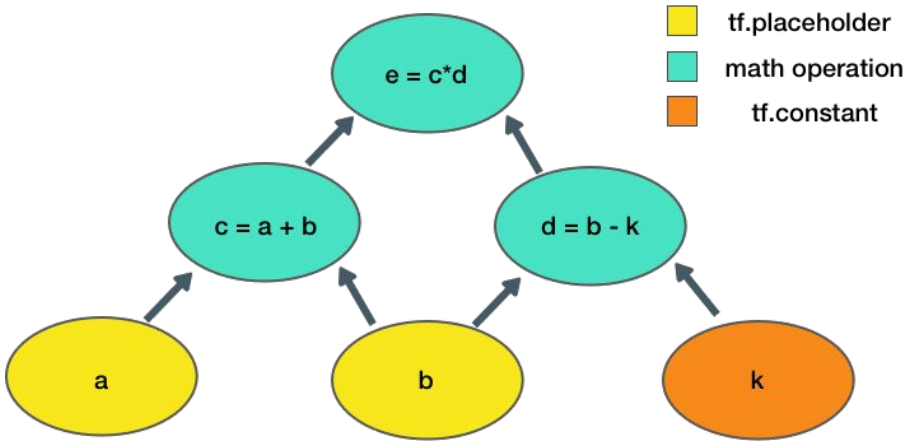
```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
k = tf.constant(1.0)
```



What are graphs made of?

- Operations

```
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
k = tf.constant(1.0)
c = tf.add(a, b)
d = tf.subtract(b, k)
e = tf.multiply(c, d)
```



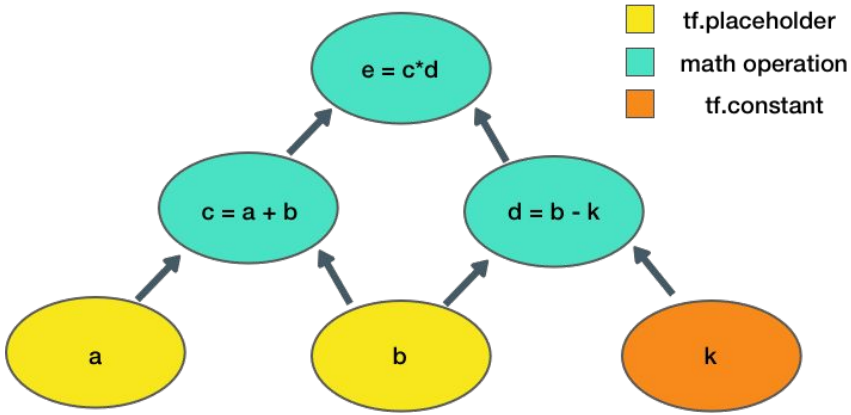
How do we run the graph?

- Select nodes to evaluate
- Specify values for placeholders

```
session.run(e, feed_dict={a:2.0, b:0.5})
>>> -1.25

session.run(c, feed_dict={a:2.0, b:0.5})
>>> 2.5

session.run([e,c], feed_dict={a:2.0, b:0.5})
>>> [2.5, -1.25]
```



Building a Neural Network Graph

- The previous graph performed a **constant** computation
- Network weights need to mutable
- Enter: **tf.Variable**

tf.Variable: Initialization

- Can initialize to specific values

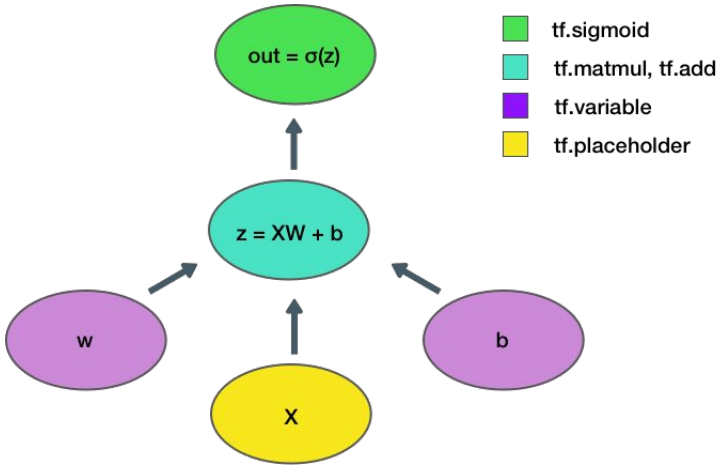
```
b1 = tf.Variable(tf.zeros((2,2)), name="bias")
```

- Can initialize to random values

```
w1 = tf.Variable(tf.random_normal((2,2)), name="w1")
```

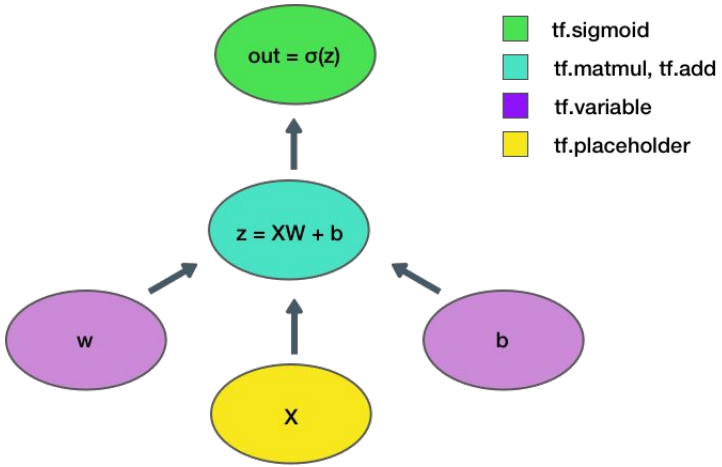
Building a Neural Network Graph

```
n_input_nodes = 2
n_output_nodes = 1
x = tf.placeholder(tf.float32, (None, 2))
y = tf.placeholder(tf.float32, (None, 1))
W = tf.Variable(tf.random_normal((n_input_nodes,
                                  n_output_nodes)))
b = tf.Variable(tf.zeros(n_output_nodes))
z = tf.matmul(x, W) + b
out = tf.sigmoid(z)
```



Adding a loss function

```
n_input_nodes = 2
n_output_nodes = 1
x = tf.placeholder(tf.float32, (None, 2))
W = tf.Variable(tf.random_normal((n_input_nodes,
                                  n_output_nodes)))
b = tf.Variable(tf.zeros(n_output_nodes))
z = tf.matmul(x, W) + b
out = tf.sigmoid(z)
loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
     logits=z, labels=y))
```

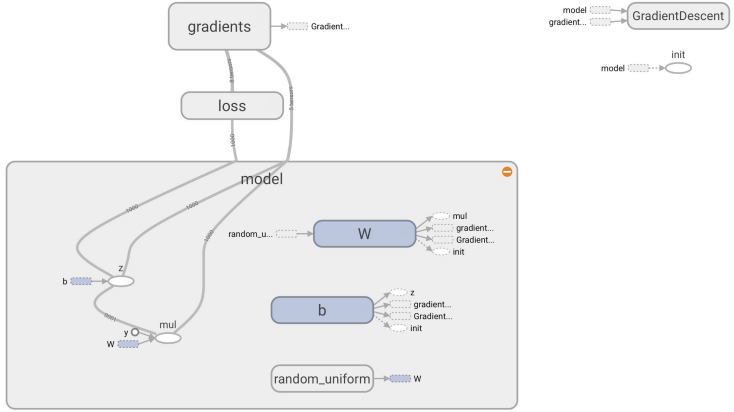


Add an optimizer: SGD

```
learning_rate = 0.02
loss = tf.reduce_mean(
    tf.nn.sigmoid_cross_entropy_with_logits(
     logits=output, labels=y))

optimizer = tf.train.GradientDescentOptimizer(
                learning_rate).minimize(loss)

sess.run(optimizer, feed_dict={x: inputs, y:labels})
```

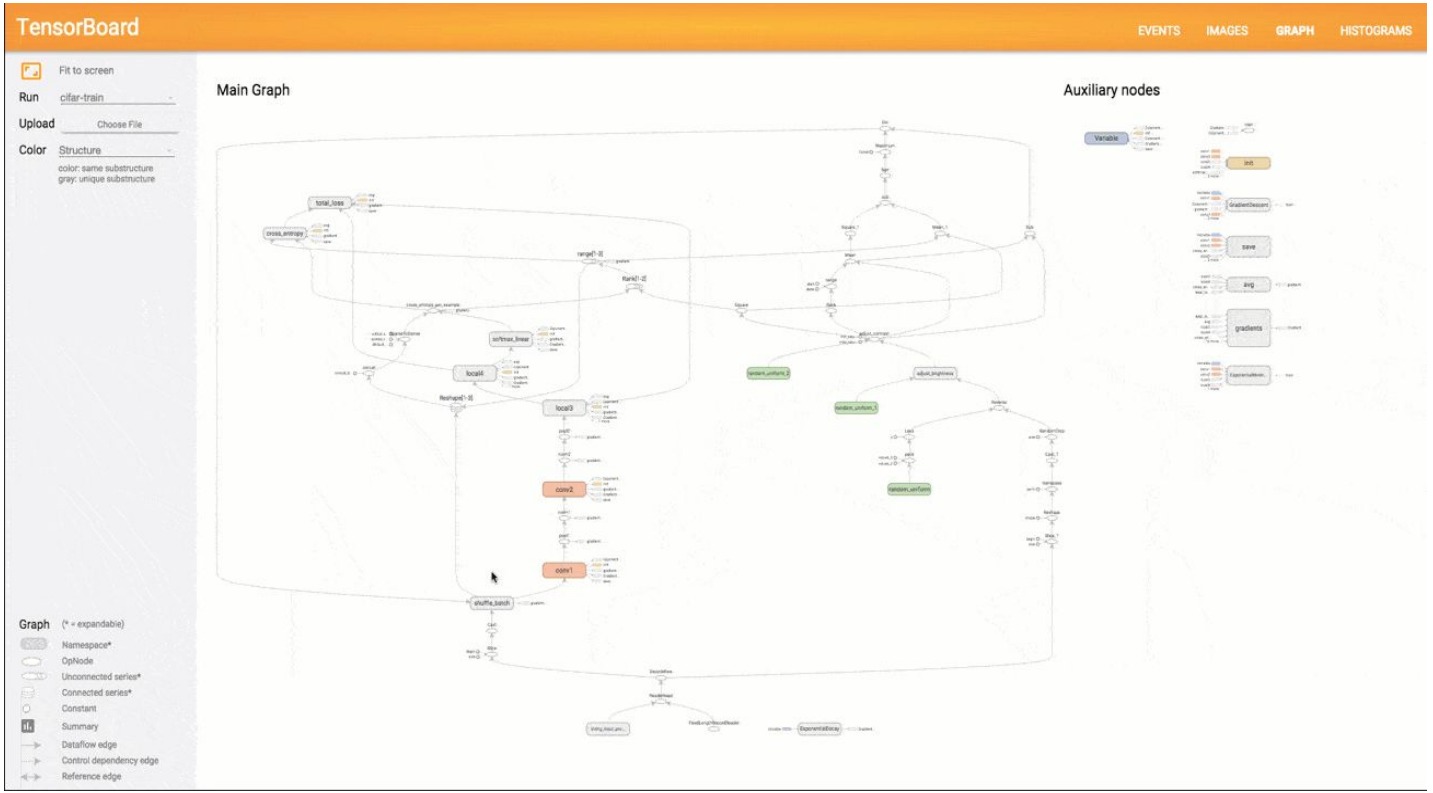


Run the graph

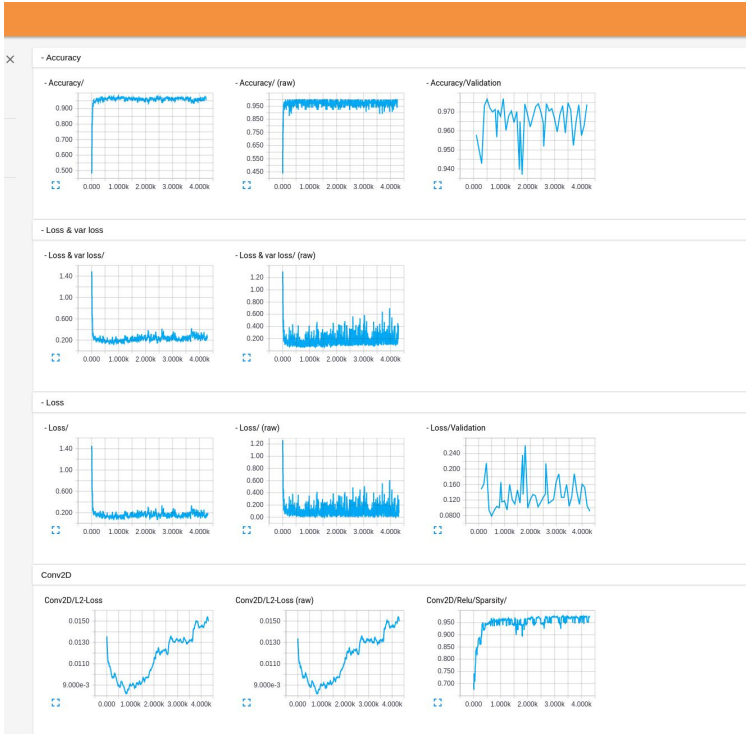
- Feed in training data in batches
- Each run of the graph updates the variables
  - SGD applies an op to all variables
- Feed in dev/test data to evaluate
  - Do not fetch the train op

Useful Features of TensorFlow

TensorBoard: Model Visualization



TensorBoard: Logging



How to use TensorBoard

- Write to Tensorboard using **Summary Logs**

Open your TensorBoard with the terminal command:

```
tensorboard --logdir=path/to/log-directory
```

Summary Logs

- Summaries are operations! So just part of the graph:

```
loss_summary = tf.summary.scalar('loss', loss)
```

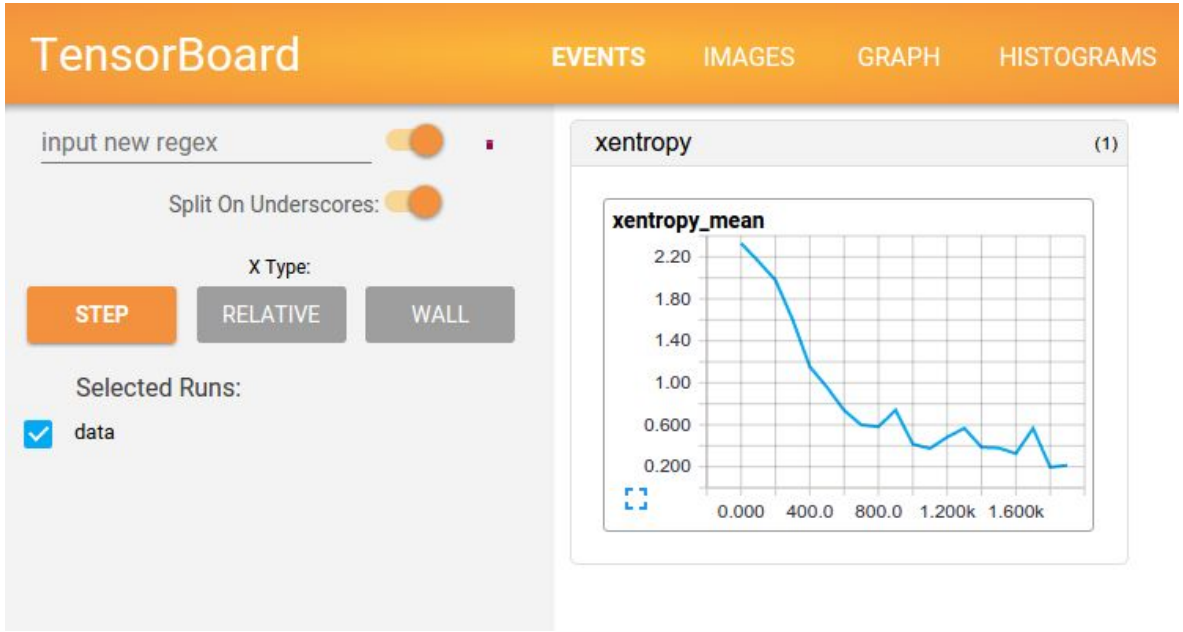
- Summary writers save summaries to a log file

```
summary_writer = tf.summary.FileWriter('logs/', session.graph)
```

- Summaries are operations - so just run them!

```
pred, summary = sess.run([out, loss_summary], feed_dict={
     x: inputs, labels_placeholder:labels})
summary_writer.add_summary(summary, global_step)
```

Summary Logs



Name Scoping

```
with tf.variable_scope("foo"):
    with tf.variable_scope("bar"):
        v = tf.Variable("v", [1])

v.name
>>> "foo/bar/v:0"
```

Sharing weights tf.get_variable()

```
with tf.variable_scope("foo"):
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])

v.name
>>> "foo/bar/v:0"
```

Why share weights?

- Imagine we want to learn a feature detector that we run over multiple inputs, and aggregate features and produce a prediction, all in 1 graph
- Need to share the weights to ensure:
  - A shared, single representation is learned
  - Gradients get propagated for all inputs

Attempt 1

```
def cnn_feature_extractor(image):
    ...
    with tf.variable_scope("feature_extractor"):
            v = tf.Variable("v", [1])
    ...
    features = tf.relu(h4)
    return features

feat_1 = cnn_feature_extractor(image_1)
feat_2 = cnn_feature_extractor(image_2)
pred = predict(feat_1, feat_2)
```

Name Scoping for cleaner code

- Networks often re-use similar structures, gets tedious to write each of them

```
def make_layer(input, input_size, output_size, scope_name):
    tf.variable_scope(scope_name):
        W = tf.Variable("w", tf.random_normal((input_size,
        output_size)))
        b = tf.Variable("b", tf.zeros(output_size))
        z = tf.matmul(input, W) + b
    return z
```

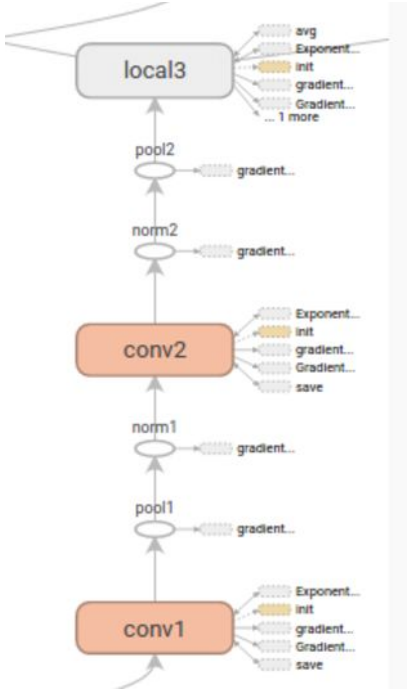
Name Scoping for cleaner code

- Networks often re-use similar structures, gets tedious to write each of them

```
...

input = ...
h0 = make_layer(input, 10, 20, "h0")
h1 = make_layer(h0, 20, 20, "h1")
...
tf.get_variable("h0/w")
tf.get_variable("h1/b")
```

Name Scoping Makes for Clean Graph Visualizations



Checkpointing + Saving Models

```
# Create a saver.
saver = tf.train.Saver(...variables...)
# Launch the graph and train, saving the model every 1,000 steps.
sess = tf.Session()
for step in xrange(1000000):
    sess.run(..training_op..)
    if step % 1000 == 0:
        # Append the step number to the checkpoint name:
        saver.save(sess, 'my-model', global_step=step)
```

Loading Models

```
# Add ops to save and restore all the variables.
saver = tf.train.Saver()

# Later, launch the model, use the saver to restore variables from disk, and
# do some work with the model.
with tf.Session() as sess:
  # Restore variables from disk.
  saver.restore(sess, "/tmp/model.ckpt")
  print("Model restored.")
  # Do some work with the model
```

TensorFlow as core of other Frameworks

- Keras, TFLearn, TF-slim, others all based on TensorFlow
- Research often means tinkering with inner workings - worthwhile to understand the core of any framework you are using

TensorFlow Tutorial:

- Pair up into pairs of 2
- Go to https://github.com/yala/introdeeplearning
- Follow install instructions
- If you need help, come down to the front

- Hint for Lab 2: Fix map(lambda...) to list(map(lambda...

TensorFlow Tutorial:

- Pair up into pairs of 2
- Go to https://github.com/yala/introdeeplearning
- Follow install instructions
- If you need help, hop on the HelpQ:
  - HelpQ is here: http://deepqueue.herokuapp.com/
  - Click “Log in with GitHub”
  - (or just raise your hand)