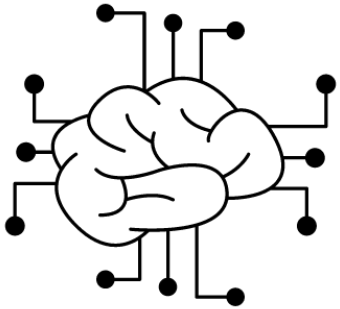
deeplearning.ai

Neural Networks for Sentiment Analysis

Outline

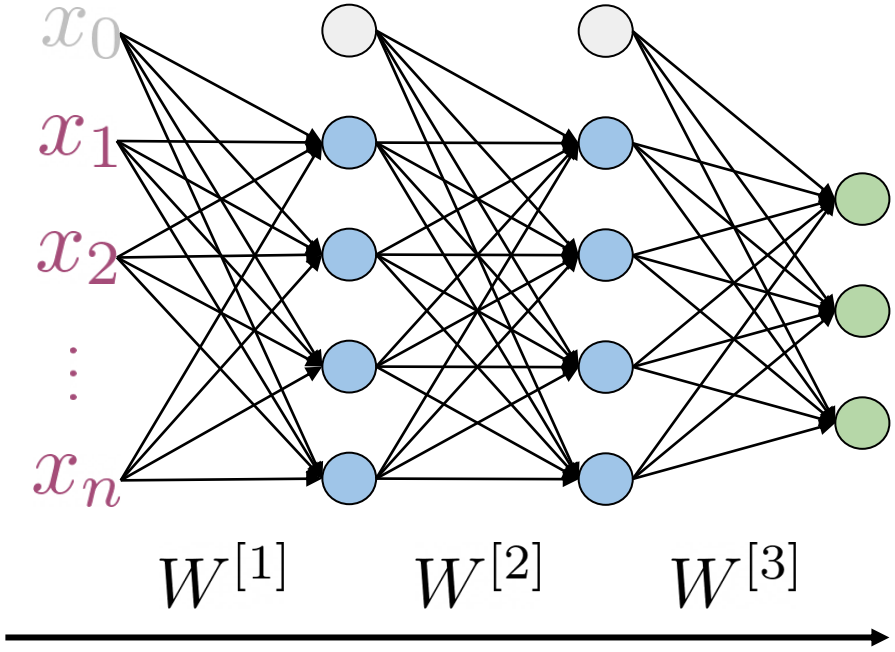
- Neural networks and forward propagation
- Structure for sentiment analysis



Neural Networks

$W^{[1]}$ $W^{[2]}$ $W^{[3]}$

Forward propagation



$a^{[i]}$ Activations ith layer

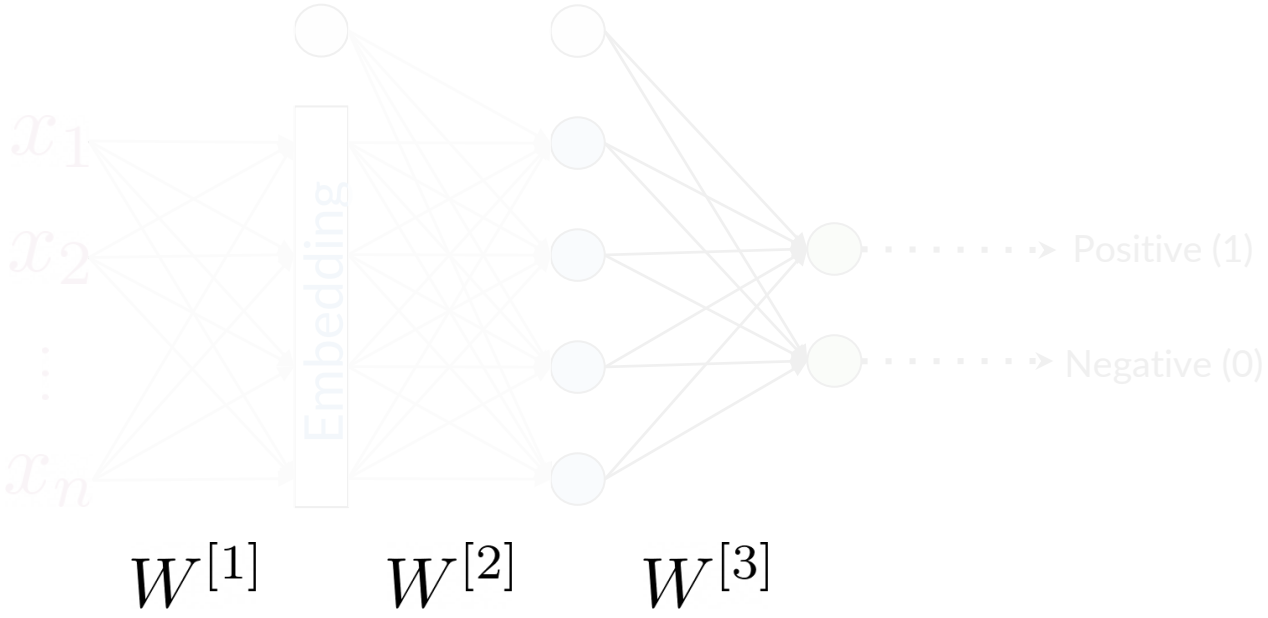
$$a^{[0]} = X$$

$$z^{[i]} = W^{[i]} a^{[i-1]}$$

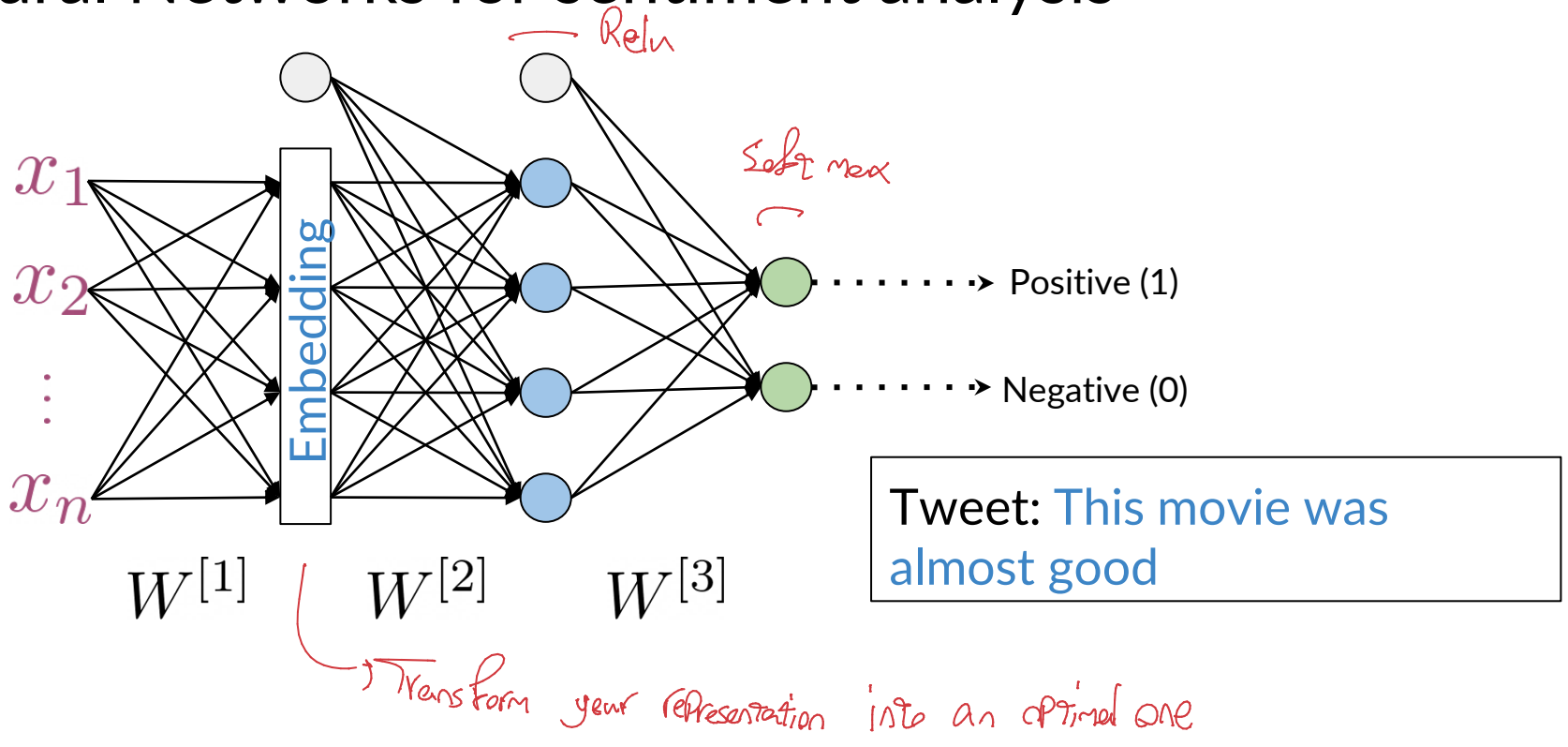
$$a^{[i]} = g^{[i]}(z^{[i]})$$

activation function

Neural Networks for sentiment analysis

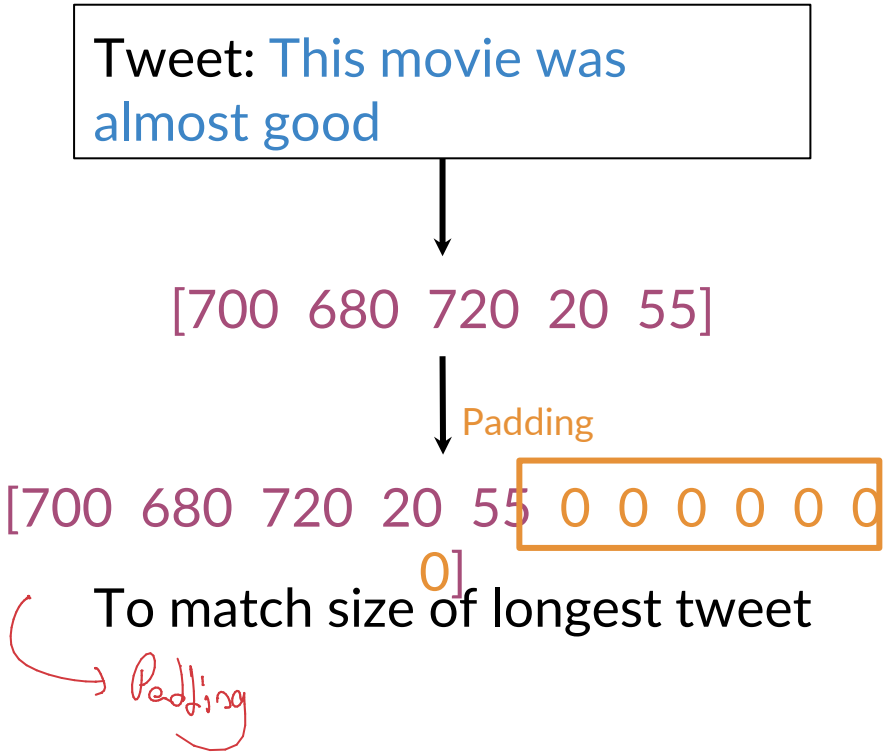


Neural Networks for sentiment analysis



Initial Representation

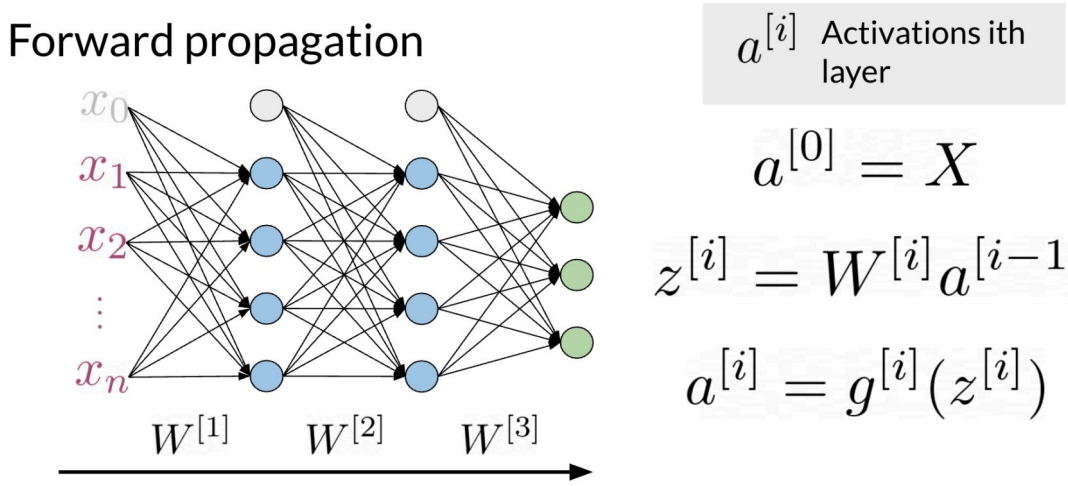
Word	Number
a	1
able	2
about	3
...	...
hand	615
...	...
happy	621
...	...
zebra	1000



Summary

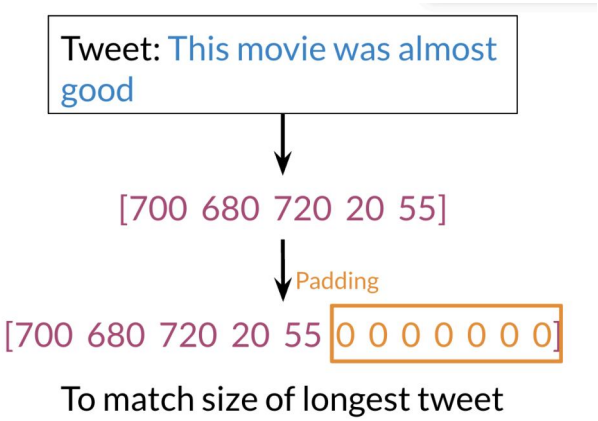
- Structure for sentiment analysis
- Classify complex tweets
- Initial representation

Previously in the course you did sentiment analysis with logistic regression and naive Bayes. Those models were in a sense more naive, and are not able to catch the sentiment off a tweet like: *"I am not happy "* or *"If only it was a good day"* . When using a neural network to predict the sentiment of a sentence, you can use the following. Note that the image below has three outputs, in this case you might want to predict, "positive", "neutral ", or "negative".



Note that the network above has three layers. To go from one layer to another you can use a W matrix to propagate to the next layer. Hence, we call this concept of going from the input until the final layer, forward propagation. To represent a tweet, you can use the following:

Word	Number
a	1
able	2
about	3
...	...
hand	615
...	...
happy	621
...	...
zebra	1000



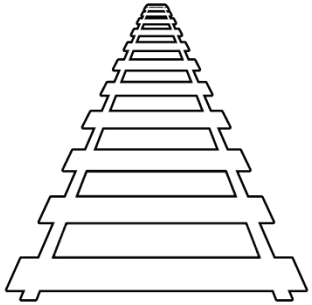
Note, that we add zeros for padding to match the size of the longest tweet.

deeplearning.ai

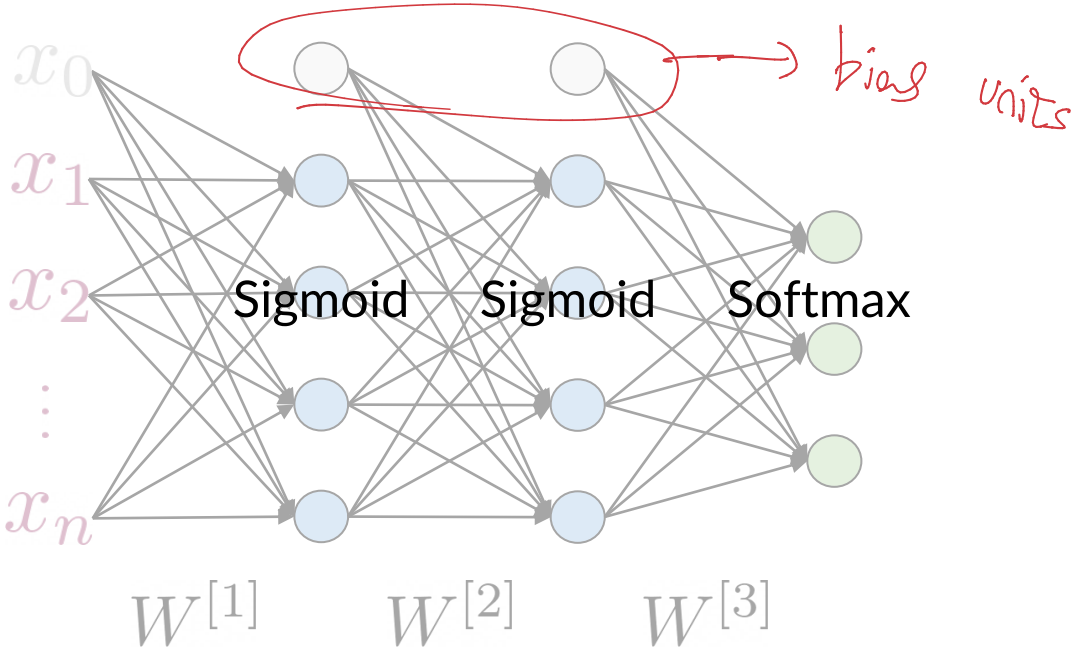
Trax: Neural Networks

Outline

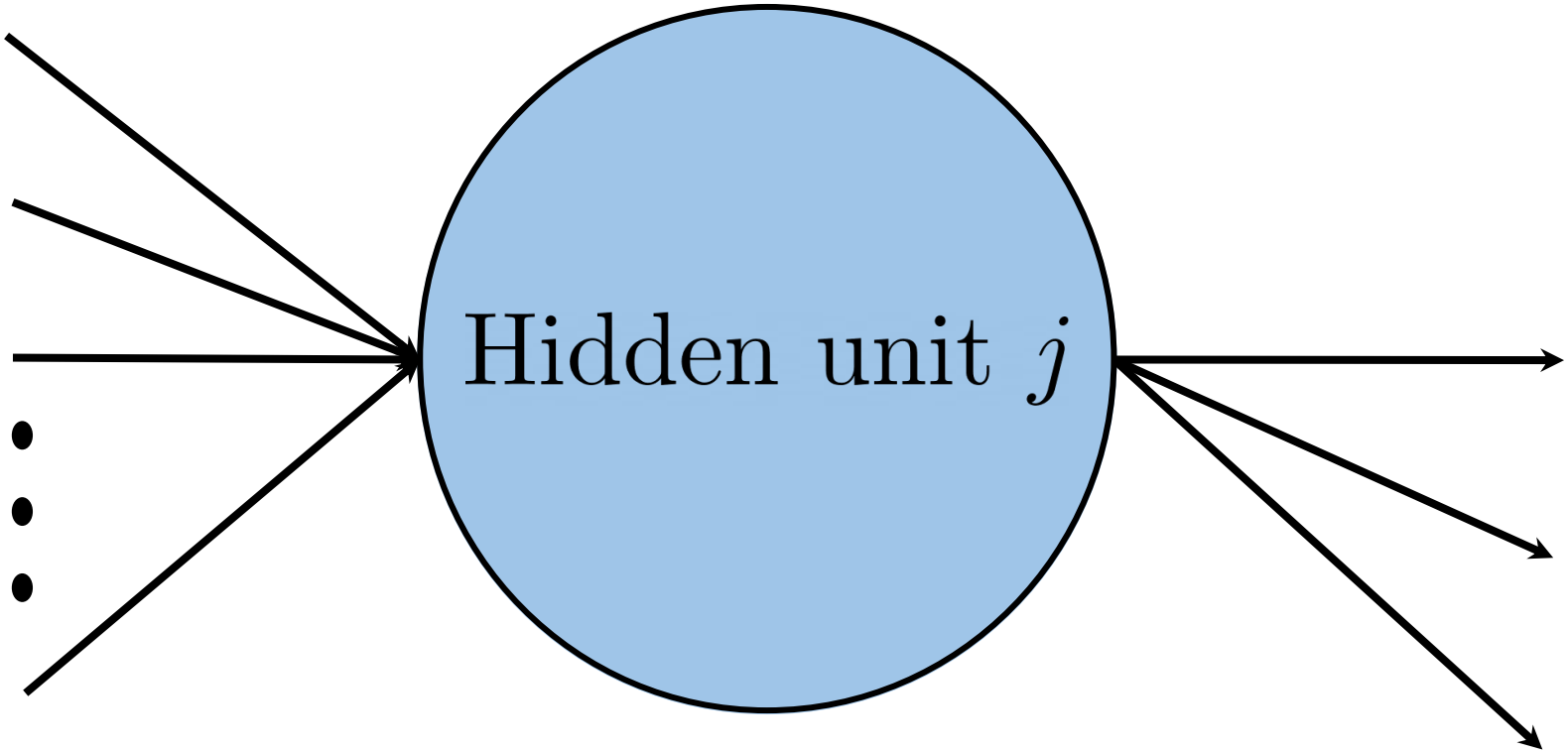
- Define a basic neural network using Trax
- Benefits of Trax



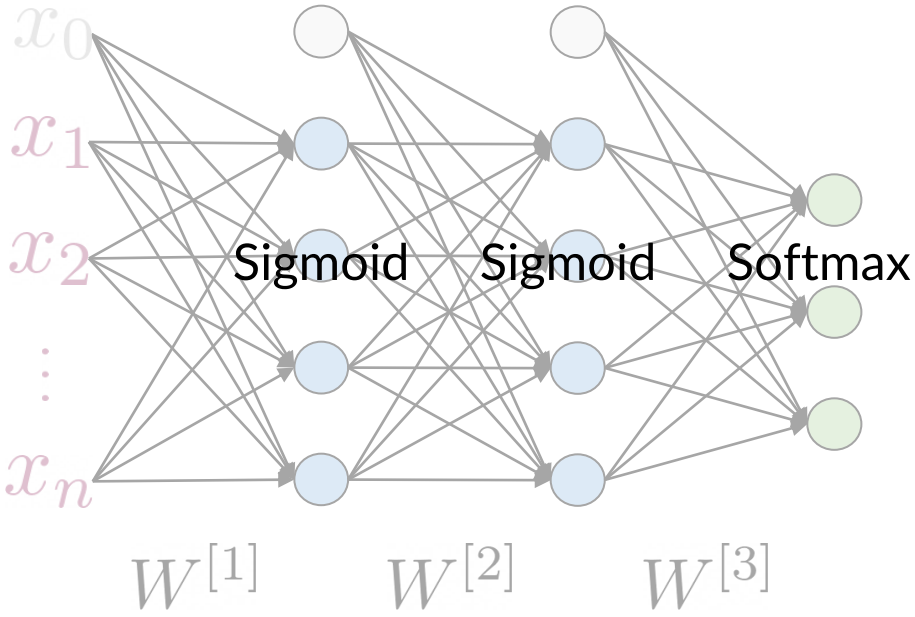
Neural Networks in Trax



Neural networks in Trax



Neural Networks in Trax




```
from trax import layers as tl
Model = tl.Serial(
        tl.Dense(4),
        tl.Sigmoid(),
        tl.Dense(4),
        tl.Sigmoid(),
        tl.Dense(3),
        tl.Softmax())
```

Advantages of using frameworks

- Run fast on CPUs, GPUs and TPUs
- Parallel computing
- Record algebraic computations for gradient evaluation

Tensorflow Pytorch JAX

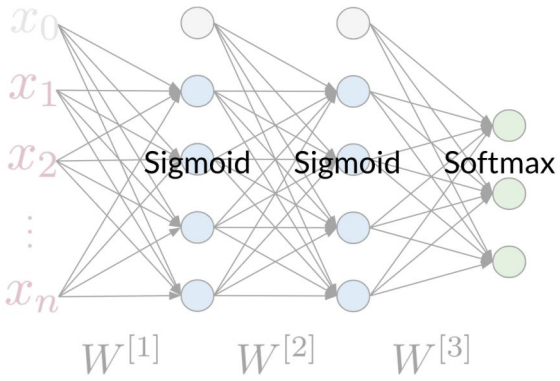
Summary

- Order of computation $\longrightarrow$ Model in Trax
- Benefits from using frameworks

Trax has several advantages:

- Runs fast on CPUs, GPUs and TPUs
- Parallel computing
- Record algebraic computations for gradient evaluation

Here is an example of how you can code a neural network in Trax:




```
from trax import layers as tl
Model = tl.Serial(
        tl.Dense(4),
        tl.Sigmoid(),
        tl.Dense(4),
        tl.Sigmoid(),
        tl.Dense(3),
        tl.Softmax())
```

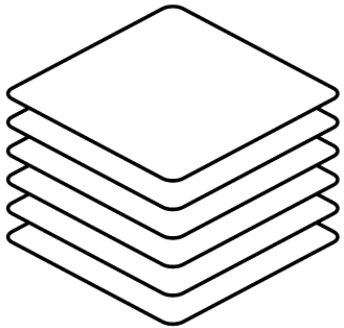
deeplearning.ai

Classes, Subclasses and Inheritance

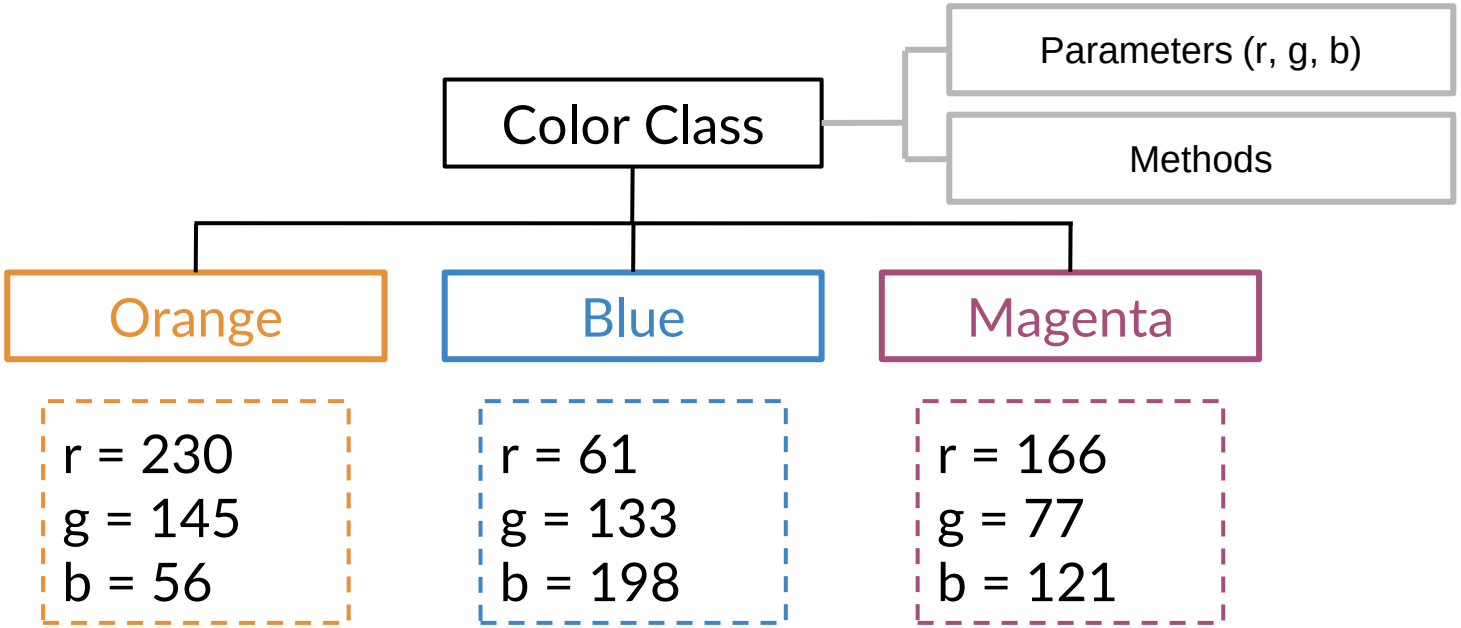
or Trax: Layers

Outline

- How classes work and their implementation
- Subclasses and inheritance



Classes


Color Class
Parameters (r, g, b)
Methods
Orange
Blue
Magenta
r = 230
g = 145
b = 56
r = 61
g = 133
b = 198
r = 166
g = 77
b = 121

Classes in Python

```
class MyClass:
    def __init__(self, y):
        self.y = y
    def my_method(self,x):
        return x + self.y
    def __call__(self, x):
        return
self.my_method(x)
```

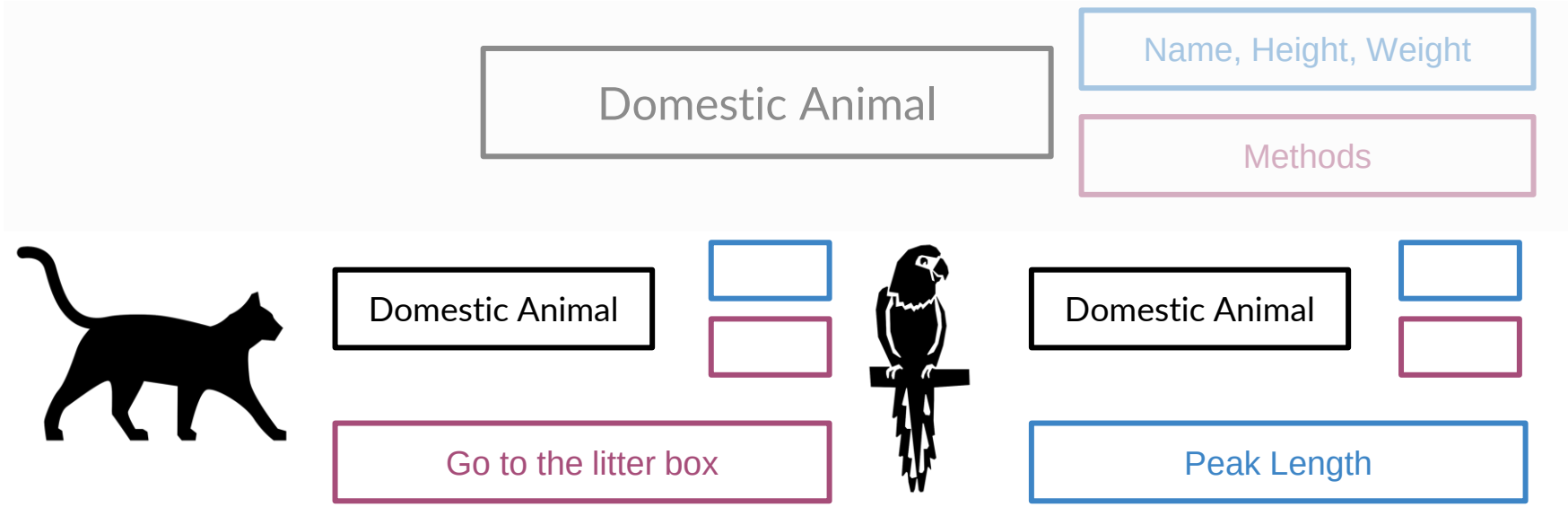
when initialize any instance of the class

```
f = MyClass(7)

print(f(3))
```

```
10
```

when call on already initialized instance

Subclasses and Inheritance



Convenient when classes share common parameters and methods.

Subclasses

Overwrite

```
class MyClass:
    def __init__(self,y):
        self.y = y
    def my_method(self,x):
        return x +
    def __call__(self,x):
        return
self.my_method(x)
```

```
class SubClass(MyClass):
    def my_method(self,x):
        return x +
self.y**2
```

```
f = SubClass(7)

print(f(3))
```

```
52
```

Summary

- Classes, subclasses, instances and inheritance.

Trax makes use of classes. If you are not familiar with classes in python, don't worry about it, here is an example.

```
class MyClass(object):
    def __init__(self, y):
        self.y = y
    def my_method(self,x):
        return x + self.y
    def __call__(self, x):
        return self.my_method(x)
```

```
f = MyClass(7)
print(f(3))
```

```
10
```

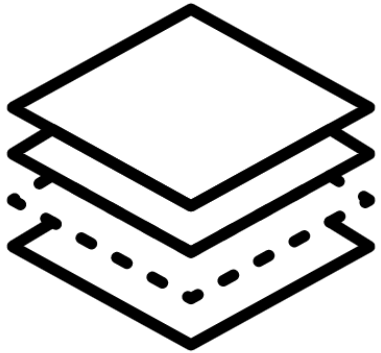
In the example above, you can see that a class takes in an *__init__* and a *__call__* method. These methods allow you to initialize your internal variables and allow you to execute your function when called. To the right you can see how you can initialize your class. When you call *MyClass(7)* , you are setting the *y* variable to 7. Now when you call *f(3)* you are adding *7 + 3*. You can change the *my_method* function to do whatever you want, and you can have as many methods as you want in a class.

deeplearning.ai

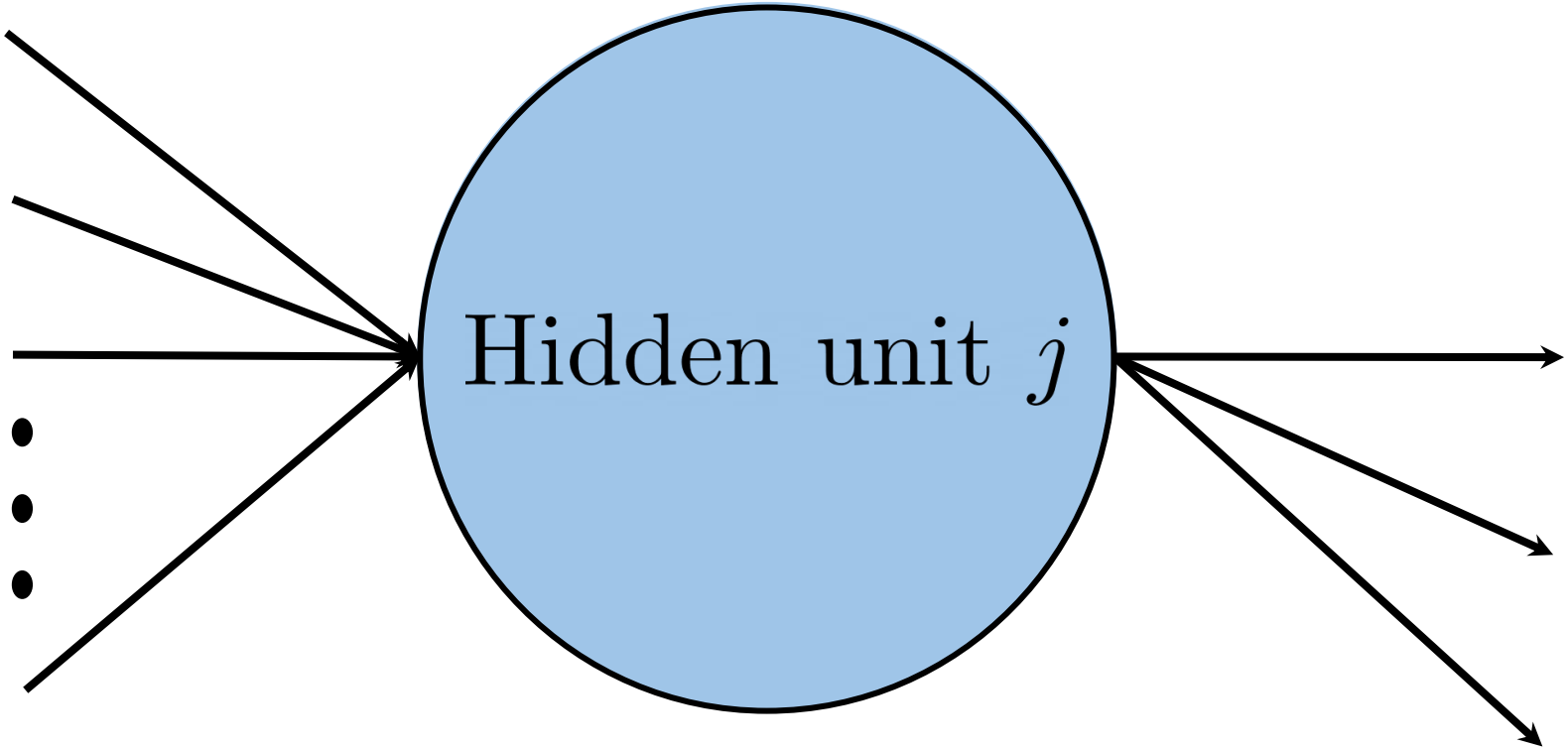
Dense and ReLU Layers

Outline

- Dense layer in detail
- ReLU function

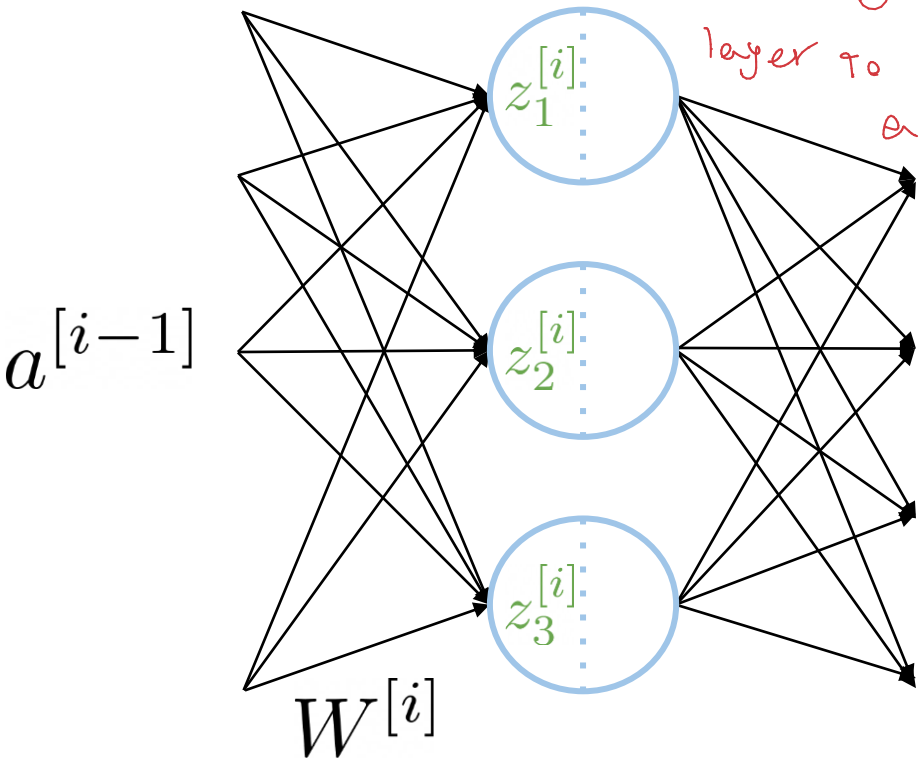


Neural networks in Trax



Dense Layer

: Allows you to go from one layer to another inside a network



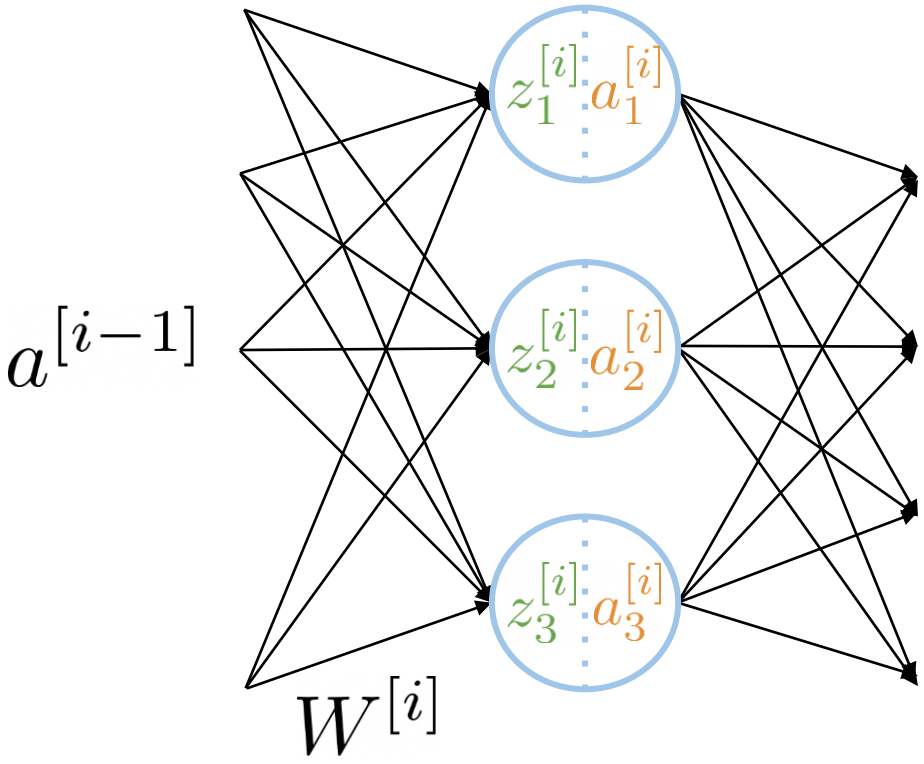
$$z_j^{[i]} = w_j^{[i]T} a^{[i-1]}$$

Dense layer

$$z^{[i]} = W^{[i]} a^{[i-1]}$$

Trainable parameters

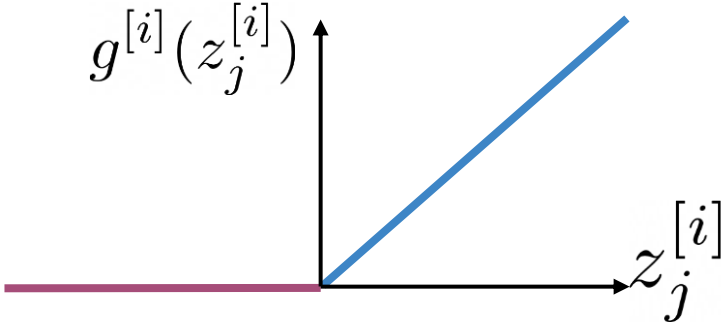
ReLU Layer : keeps your network stable



$$a_j^{[i]} = g^{[i]}(z_j^{[i]})$$

ReLU = Rectified linear unit

$$g(z^{[i]}) = \max(0, z^{[i]})$$

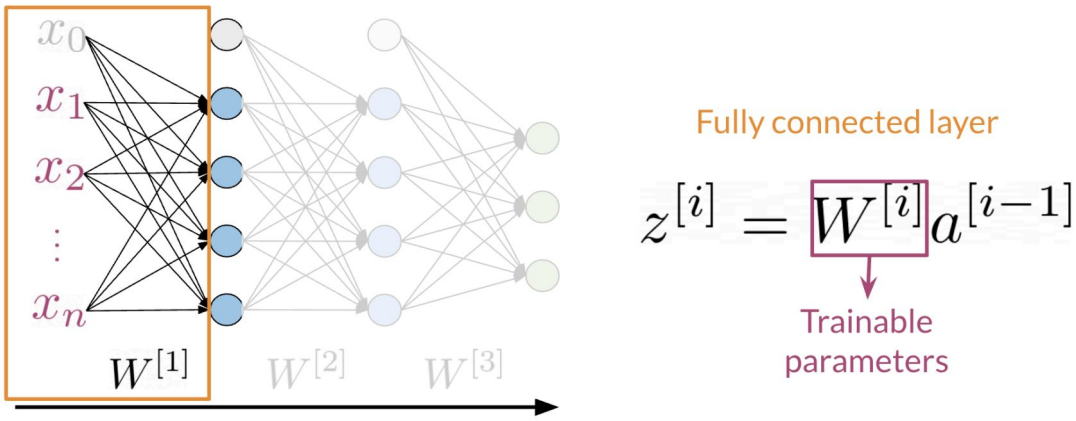


```
tf.keras.layers.Dense(32, activation='relu')
```

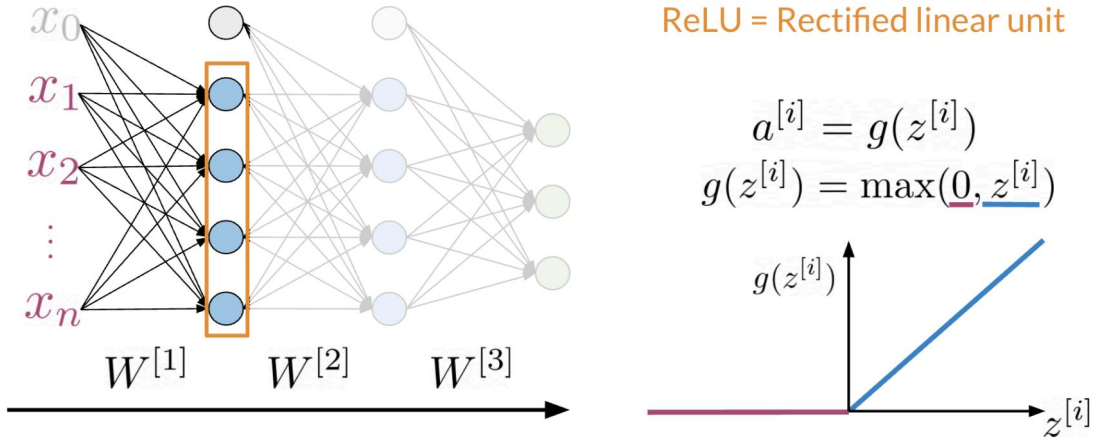
Summary

- Dense Layer $\longrightarrow z^{[i]} = W^{[i]} a^{[i-1]}$
- ReLU Layer $\longrightarrow g(z^{[i]}) = \max(0, z^{[i]})$

The Dense layer is the computation of the inner product between a set of trainable weights (weight matrix) and an input vector. The visualization of the dense layer could be seen in the image below.



The orange box shows the dense layer. An activation layer is the set of blue nodes. Concretely one of the most commonly used activation layers is the rectified linear unit (ReLU).

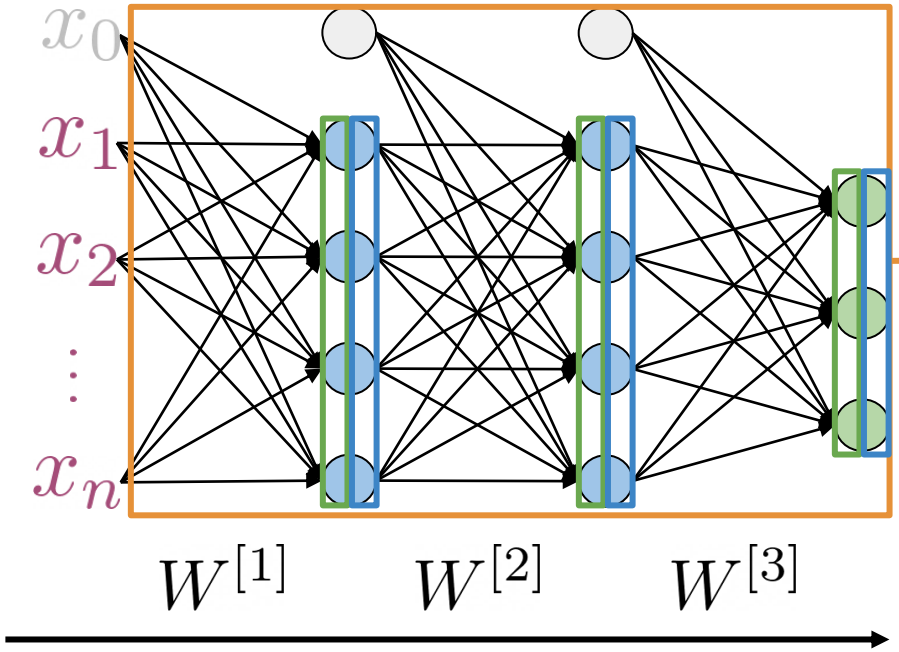


ReLU(x) is defined as *max(0,x)* for any input x.

deeplearning.ai

Serial Layer

Serial Layer



Composition of layers in *serial* arrangement

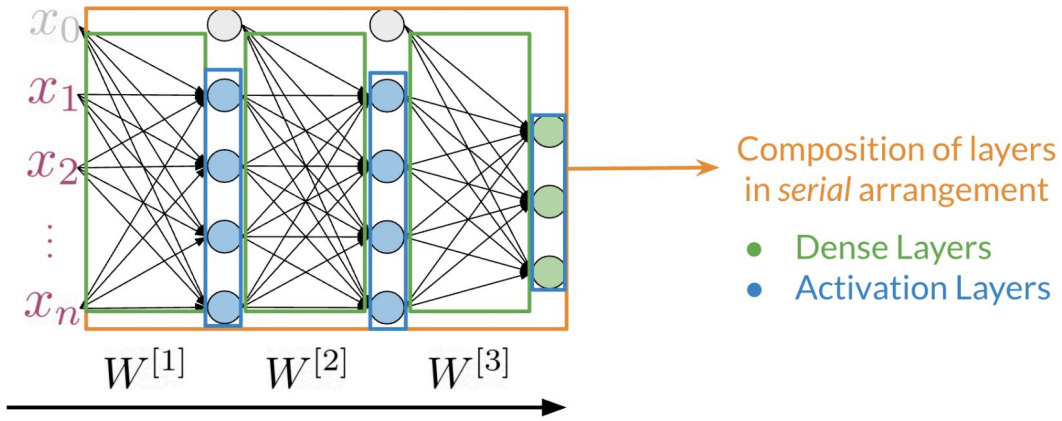
- Dense Layers
- Activation Layers

Compute forward propagation of the entire model

Summary

- Serial layer is a composition of sublayers
- Forward propagation by calling the method from the serial layer

A serial layer allows you to compose layers in a serial arrangement:



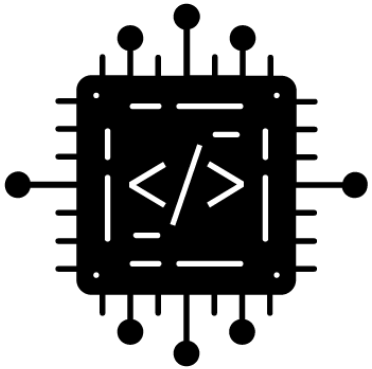
It is a composition of sublayers. These layers are usually dense layers followed by activation layers.

deeplearning.ai

Other Layers

Outline

- Embedding layer
- Mean layer



Embedding Layer

Vocabulary	Index		
I	1	0.020	0.006
am	2	-0.003	0.010
happy	3	0.009	0.010
because	4	-0.011	-0.018
learning	5	-0.040	-0.047
NLP	6	0.009	0.050
sad	7	-0.044	0.001
not	8	0.011	-0.022

Trainable weights

Vocabulary x Embedding

Mean Layer

Tweet: I am happy

Vocabulary	Index		
I	1	0.020	0.006
am	2	-0.003	0.010
happy	3	0.009	0.010

0.020	0.006
-0.003	0.010
0.009	0.010

Mean of the word embeddings

0.009
0.009

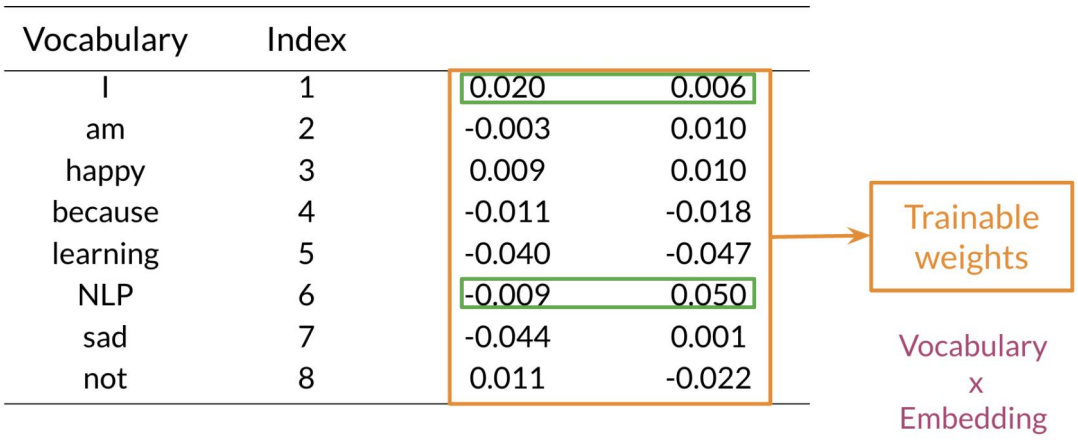
= embedding size

No trainable parameters

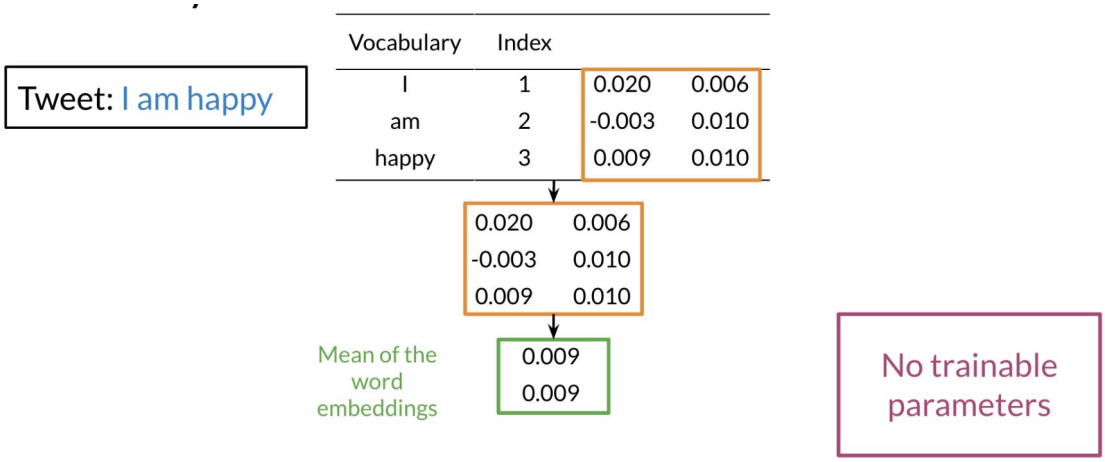
Summary

- Embedding is trainable using an embedding layer
- Mean layer gives a vector representation

Other layers could include embedding layers and mean layers. For example, you can learn word embeddings for each word in your vocabulary as follows:



The mean layer allows you to take the average of the embeddings. You can visualize it as follows:



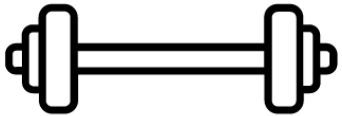
This layer does not have any trainable parameters.

deeplearning.ai

Training

Outline

- Computing gradients
- Training



Computing gradients in Trax

$$f(x) = 3x^2 + x$$

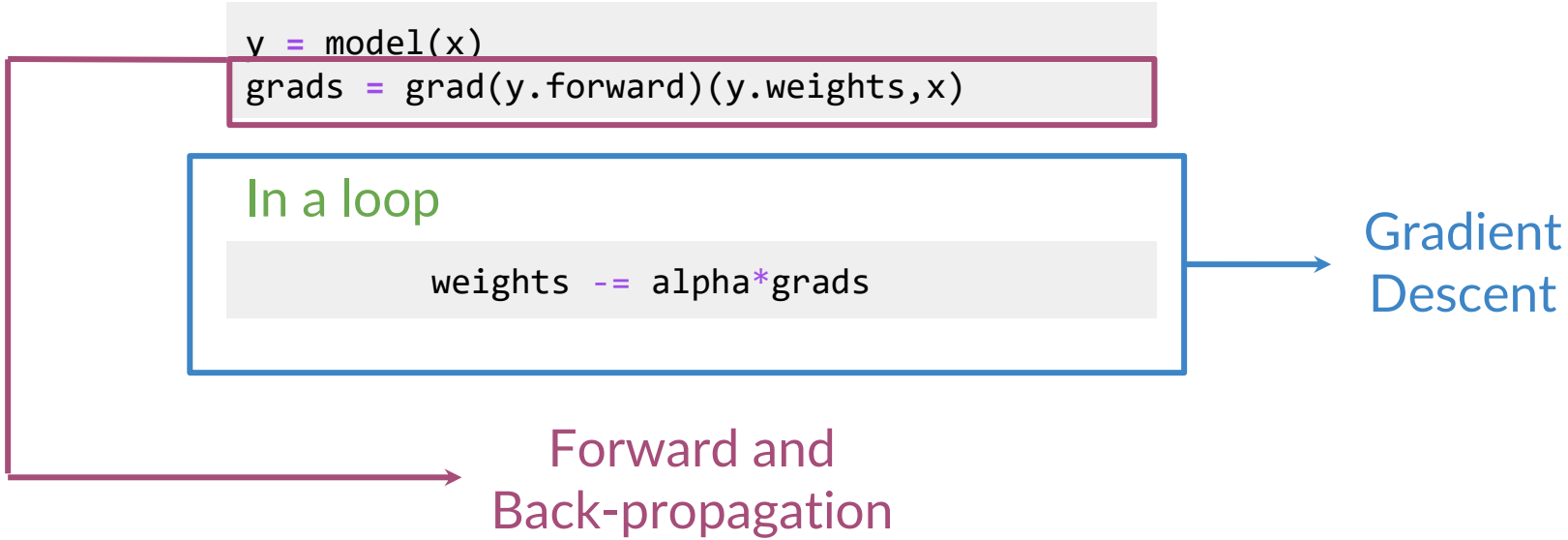
$$\frac{\delta f(x)}{\delta x} = 6x + 1$$

Gradient

```
def f(x):
        return 3*x**2 + x

grad_f = trax.math.grad(f)
```

Returns a function

Training with grad()



Summary

- grad() allows much easier training
- Forward and backpropagation in one line!

In Trax, the function grad allows you to compute the gradient. You can use it as follows:

$$f(x) = 3x^2 + x$$

$$\frac{\delta f(x)}{\delta x} = 6x + 1$$

Gradient

```
def f(x):
    return 3*x**2 + x

grad_f = trax.math.grad(f)
```

Returns a function

Now if you were to evaluate *grad_f* at a certain value, namely *z* , it would be the same as computing *6z+1* . Now to do the training, it becomes very simple:

```
y = model(x)
grads = grad(y.forward)(y.weights,x)
```

In a loop

```
    weights -= alpha*grads
```

Gradient Descent

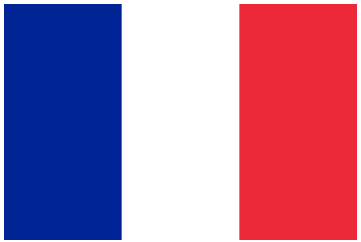
Forward and Back-propagation

You simply compute the gradients by feeding in y.forward (the latest value of y), the weights, and the input x, and then it does the back-propagation for you in a single line. You can then have the loop that allows you to update the weights (i.e. gradient descent!).

deeplearning.ai

Traditional Language models

Traditional Language Models



J'ai vu le match de foot

Sequence	$P(\text{Sequence})$
I saw the game of soccer	4.5 e-5
I saw the soccer game	6.0 e-5
I saw the soccer match	4.6 e-5
Saw I the game of soccer	2.6 e-9

N-grams

$$P(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)} \longrightarrow \text{Bigrams}$$

$$P(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)} \longrightarrow \text{Trigrams}$$

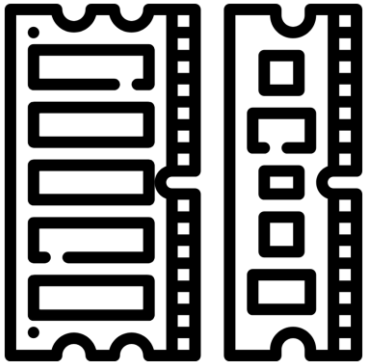
$$P(w_1, w_2, w_3) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_2)$$

★ Bigram assumption: $P(w_3|w_1, w_2) \sim P(w_3|w_2)$

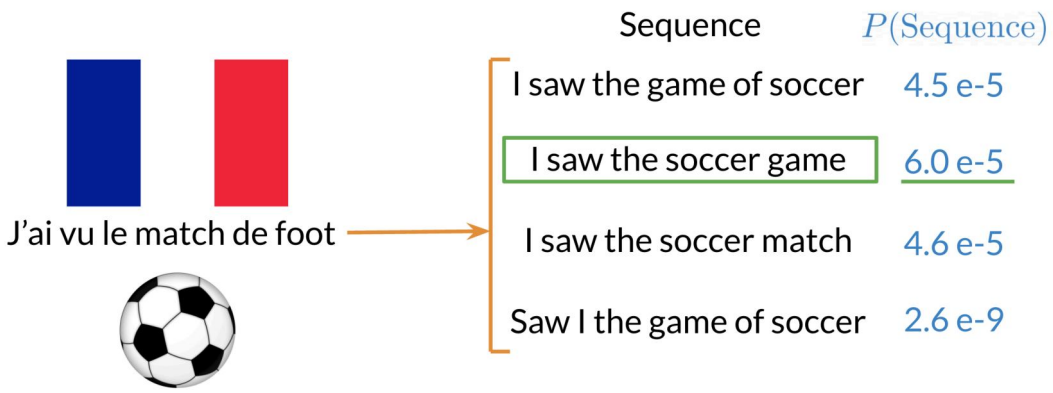
- Large N-grams to capture dependencies between distant words
- Need a lot of space and RAM

Summary

- N-grams consume a lot of memory
- Different types of RNNs are the preferred alternative



Traditional language models make use of probabilities to help identify which sentence is most likely to take place.



In the example above, the second sentence is the one that is most likely to take place as it has the highest probability of happening. To compute the probabilities, you can do the following:

$$P(w_2|w_1) = \frac{\text{count}(w_1, w_2)}{\text{count}(w_1)} \longrightarrow \text{Bigrams}$$

$$P(w_3|w_1, w_2) = \frac{\text{count}(w_1, w_2, w_3)}{\text{count}(w_1, w_2)} \longrightarrow \text{Trigrams}$$

$$P(w_1, w_2, w_3) = P(w_1) \times P(w_2|w_1) \times P(w_3|w_2)$$

Large N-grams capture dependencies between distant words and need a lot of space and RAM. Hence, we resort to using different types of alternatives.

deeplearning.ai

Recurrent Neural Networks

Advantages of RNNs

Nour was supposed to study with me. I called her but she did not answer

Q: answer

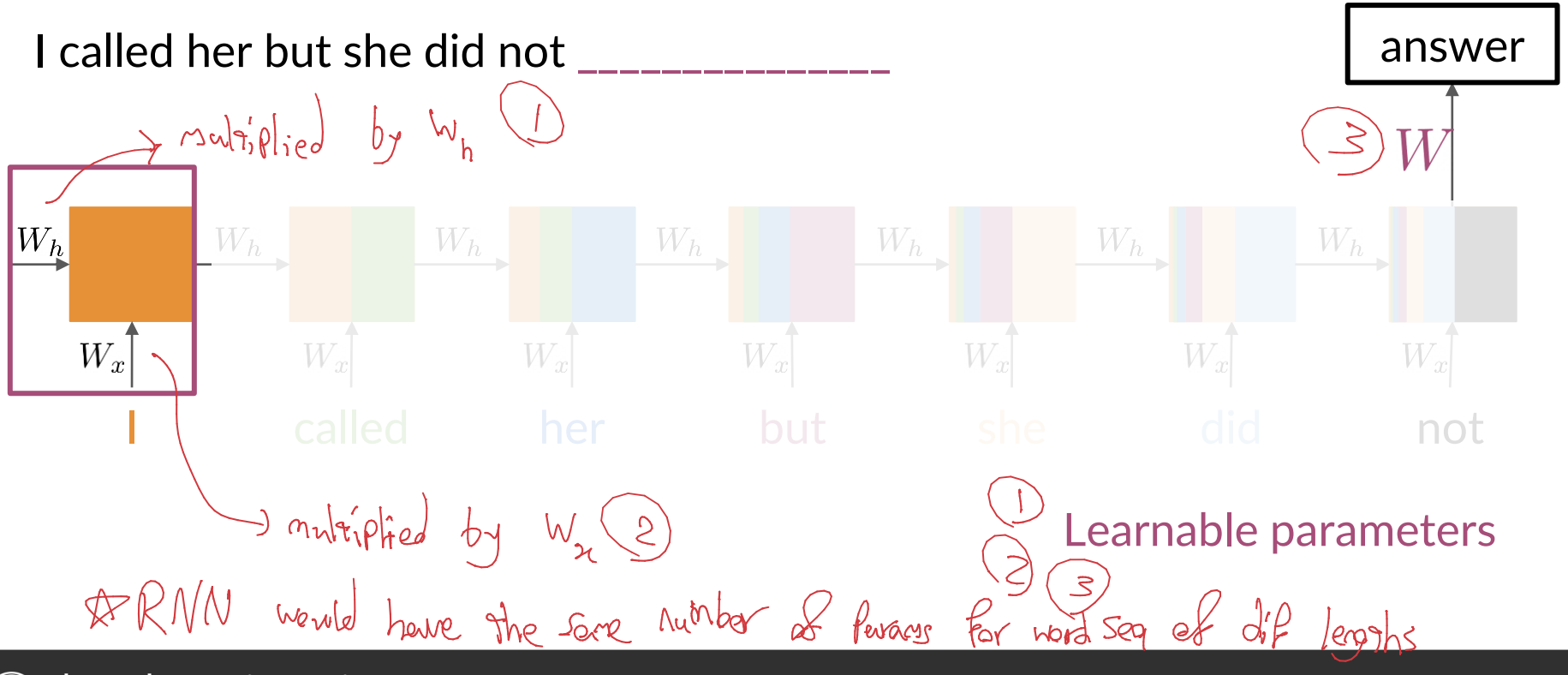
~~want~~
respond
choose
want
have
ask
attempt
answer
know

RNNs look at every previous word

Similar probabilities with trigram

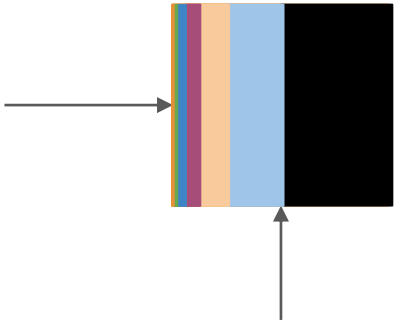
RNNs Basic Structure

I called her but she did not _____

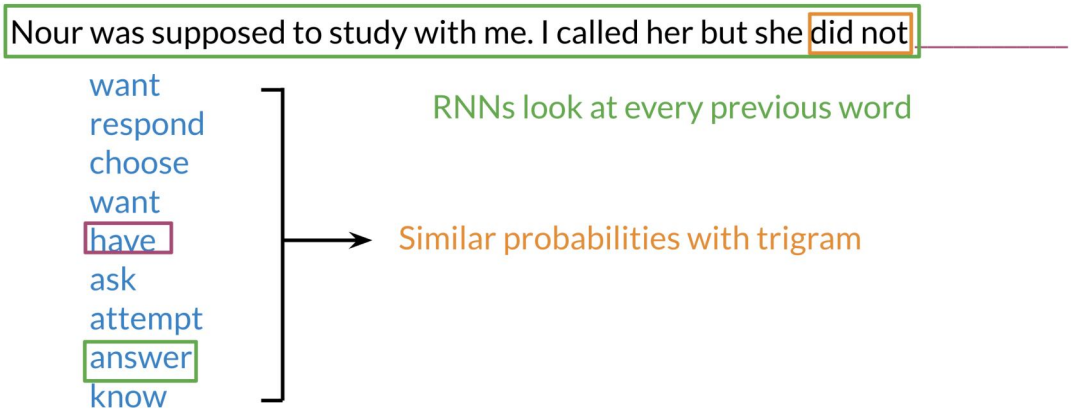


Summary

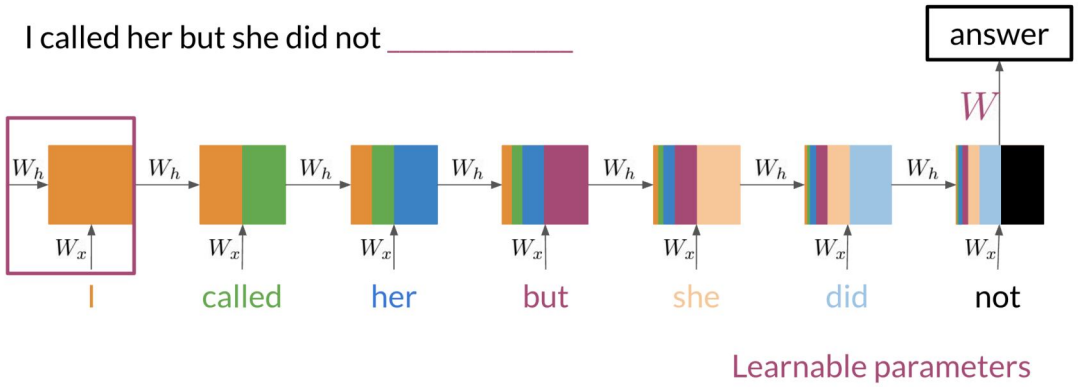
- RNNs model relationships among distant words
- In RNNs a lot of computations share parameters



Previously, we tried using traditional language models, but it turns out they took a lot of space and RAM. For example, in the sentence below:



An N-gram *(trigram)* would only look at "did not" and would try to complete the sentence from there. As a result, the model will not be able to see the beginning of the sentence "I called her but she". Probably the most likely word is *have* after "did not". RNNs help us solve this problem by being able to track dependencies that are much further apart from each other. As the RNN makes its way through a text corpus, it picks up some information as follows:



Note that as you feed in more information into the model, the previous word's retention gets weaker, but it is still there. Look at the orange rectangle above and see how it becomes smaller as you make your way through the text. This shows that your model is capable of capturing dependencies and remembers a previous word although it is at the beginning of a sentence or paragraph. Another advantage of RNNs is that a lot of the computation shares parameters.

deeplearning.ai

Applications of RNNs

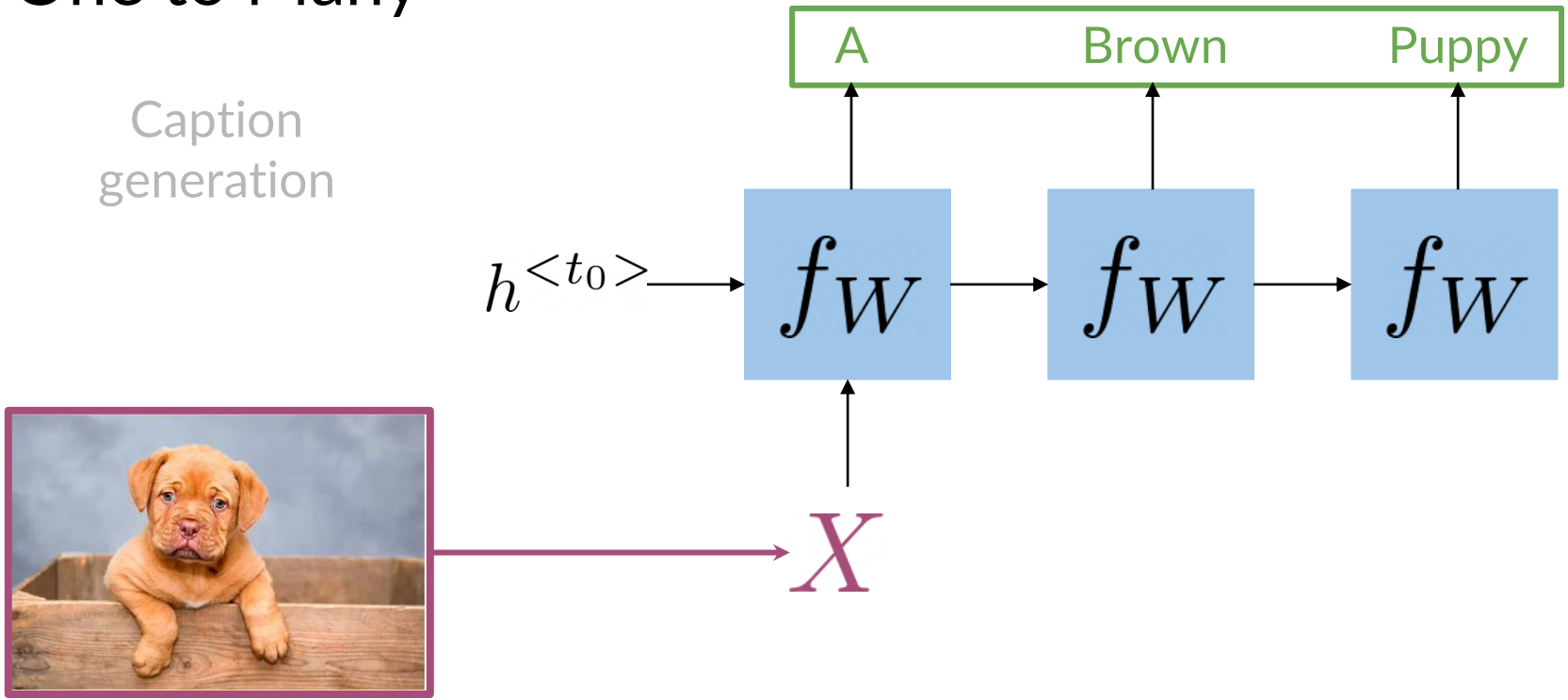
One to One


for this type of task RNN aren't all that useful
Position of RM in leaderboard
LaLiga Santander
Real Valladolid
0
1
Real Madrid
Real Zaragoza
0
4
Real Madrid
Atletico Madrid
0
1
Real Madrid
$h^{<t_0>}$
f_W
X

RNN is good for this

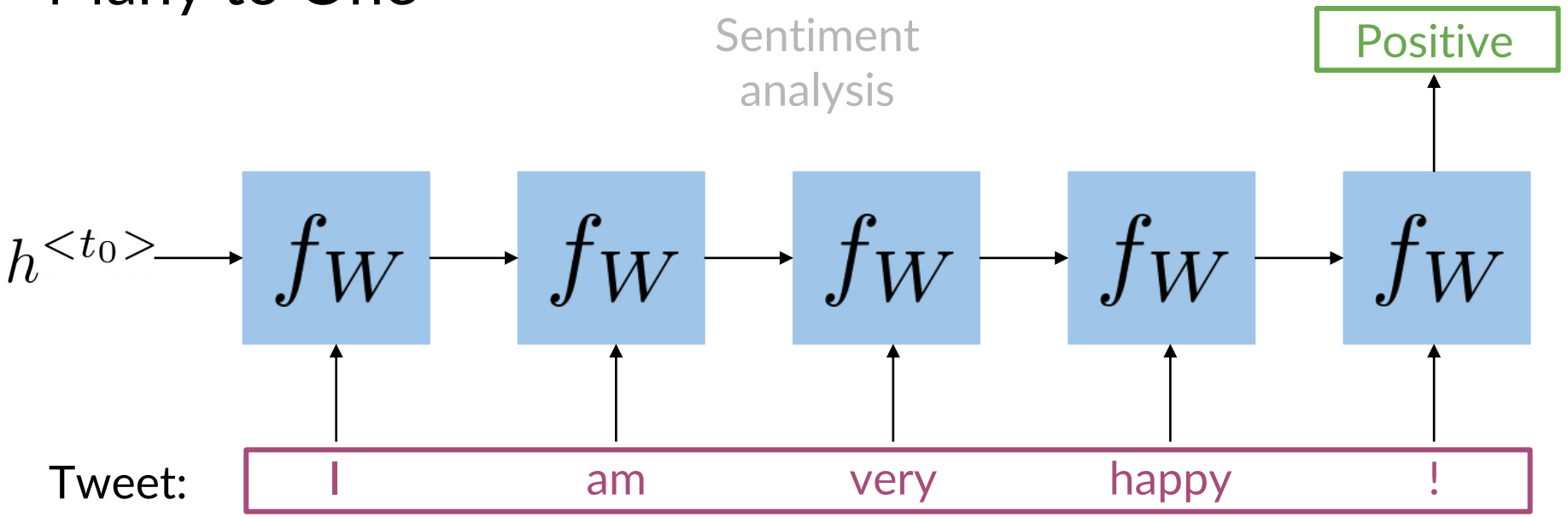
One to Many

Caption generation



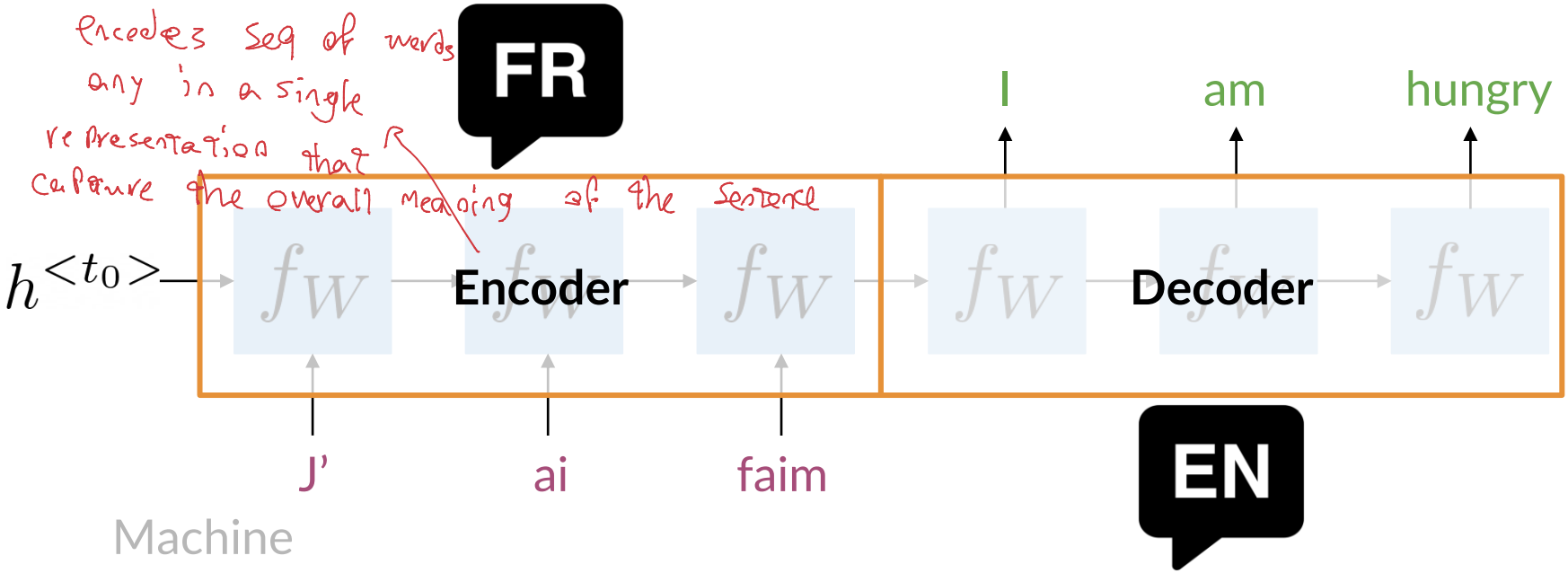
Many to One

RNN ✓



Many to Many

RNN ✓



Machine Translation

Summary

- RNNs can be implemented for a variety of NLP tasks
- Applications include Machine translation and caption generation

RNNs could be used in a variety of tasks ranging from machine translation to caption generation. There are many ways to implement an RNN model:

- One to One: given some scores of a championship, you can predict the winner.
- One to Many: given an image, you can predict what the caption is going to be.
- Many to One: given a tweet, you can predict the sentiment of that tweet.
- Many to Many: given an english sentence, you can translate it to its German equivalent.

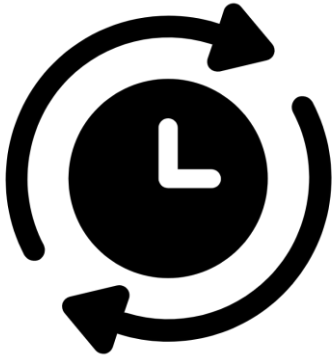
In the next video, you will see the math in simple RNNs.

deeplearning.ai

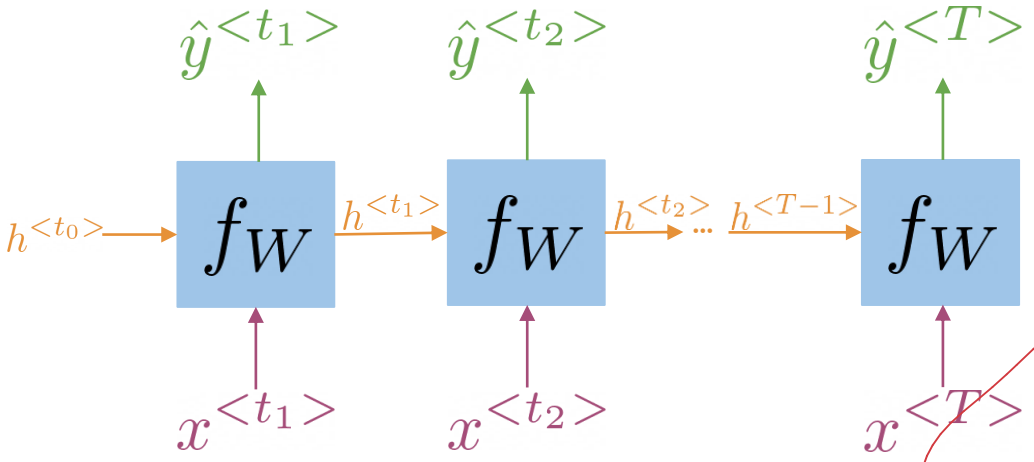
Math in Simple RNNs

Outline

- How RNNs propagate information (Through time!)
- How RNNs make predictions



A Vanilla RNN



activation func

$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

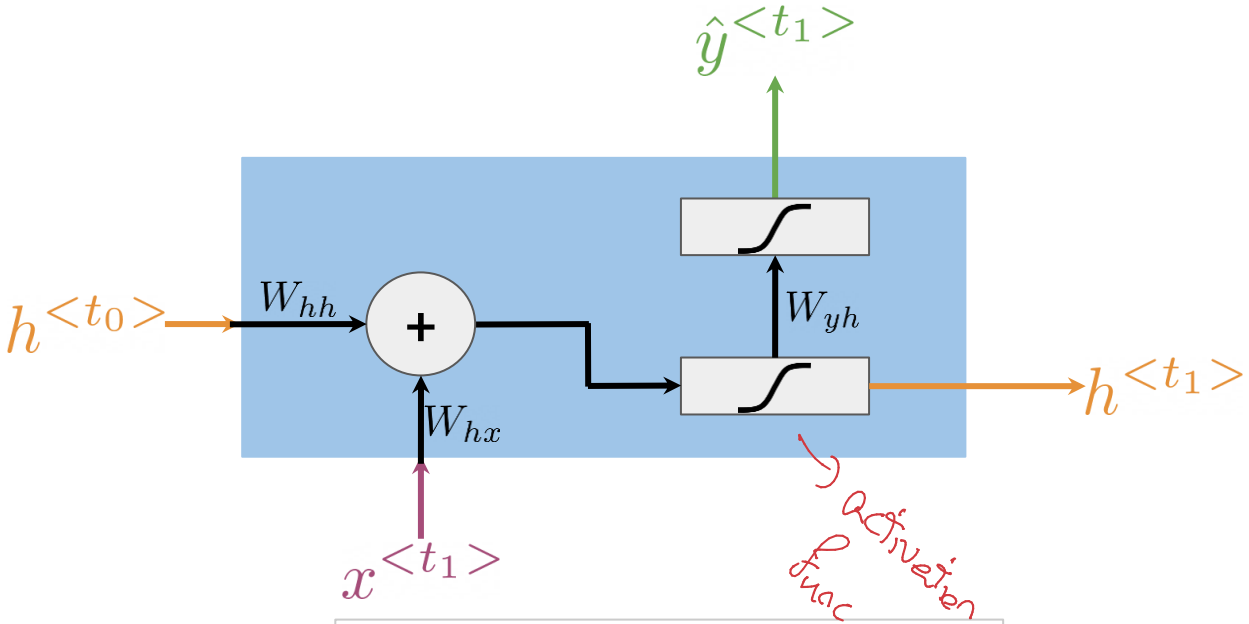
$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

$$h^{<t>} = g(W_{hh}h^{<t-1>} + W_{hx}x^{<t>} + b_h)$$

$W_h = [W_{hh} \quad W_{hx}]$

$[h^{<t-1>}, x^{<t>}] = \begin{bmatrix} h^{<t-1>} \\ x^{<t>} \end{bmatrix}$

A Vanilla RNN

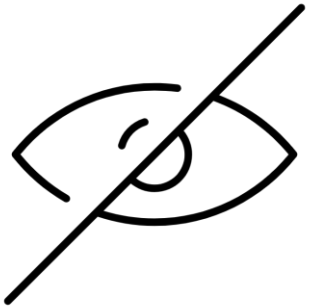


$$h^{<t>} = g(W_{hh}h^{<t-1>} + W_{hx}x^{<t>} + b_h)$$

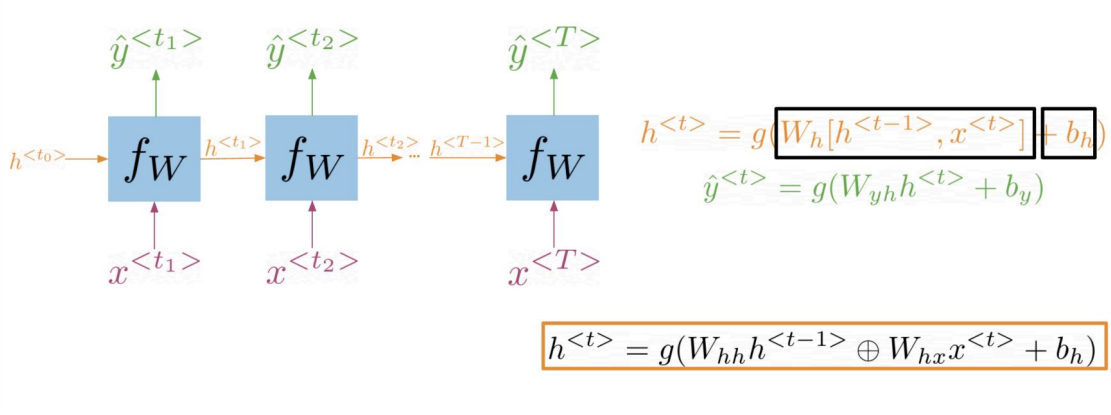
$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

Summary

- Hidden states propagate information through time
- Basic recurrent units have two inputs at each time: $h^{<t-1>}$ $x^{<t>}$,



It is best to explain the math behind a simple RNN with a diagram:



Note that:

$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

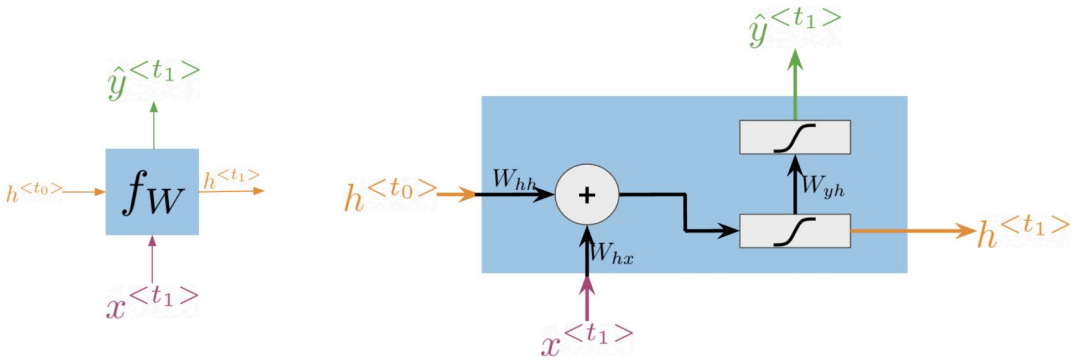
Is the same as multiplying W_{hh} by h and W_{hx} by x. In other words, you can concatenate it as follows:

$$h^{<t>} = g\left(W_{hh}h^{<t-1>} \oplus W_{hx}x^{<t>} + b_h\right)$$

For the prediction at each time step, you can use the following:

$$\hat{y}^{<t>} = g\left(W_{yh}h^{<t>} + b_y\right)$$

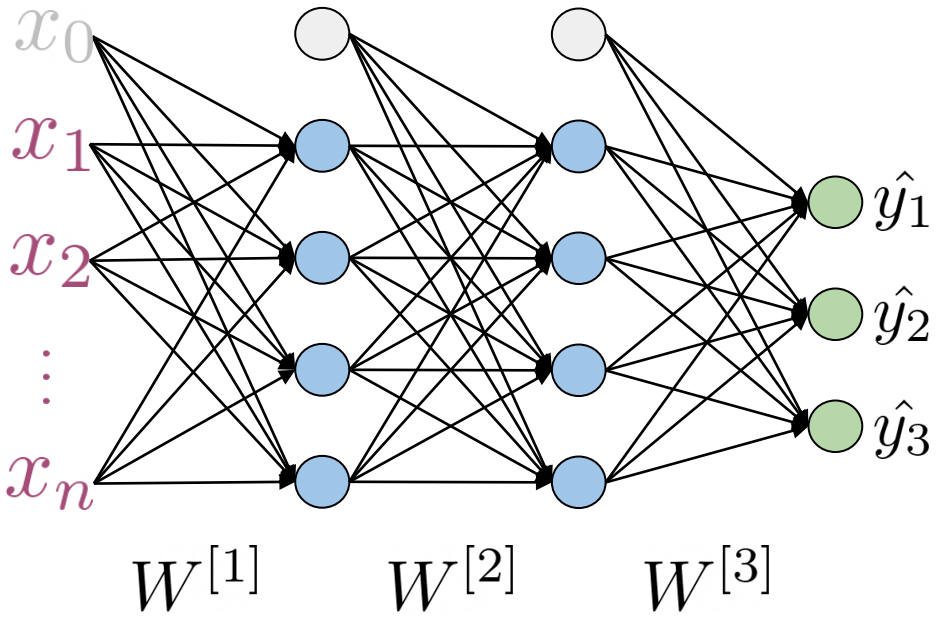
Note that you end up training $W_{hh}, W_{hx}, W_{yh}, b_h, b_y$. Here is a visualization of the model.



deeplearning.ai

Cost Function for RNNs

Cross Entropy Loss



K - classes or possibilities

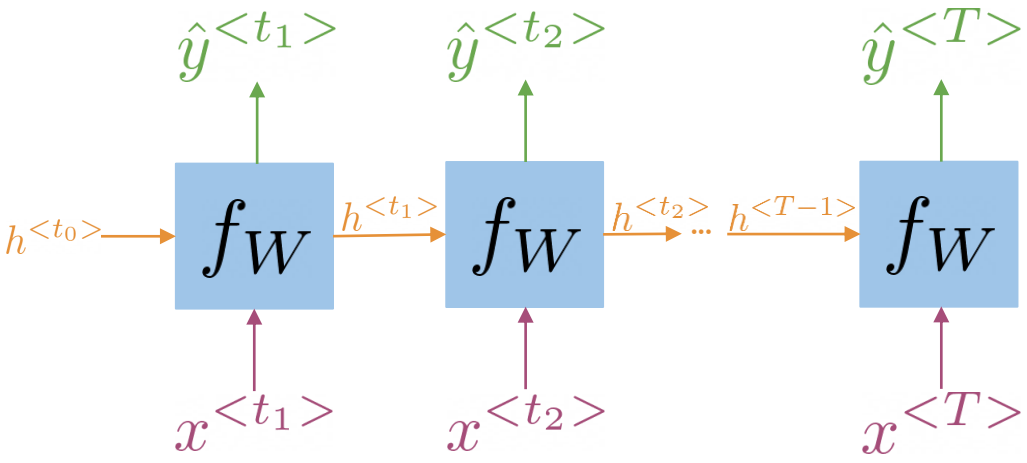
$$J = -\sum_{j=1}^{K} y_j \log \hat{y}_j$$

Either 0 or 1

Looking at a single example (x, y)

only one of $y_j = 1$

Cross Entropy Loss



$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$

$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$

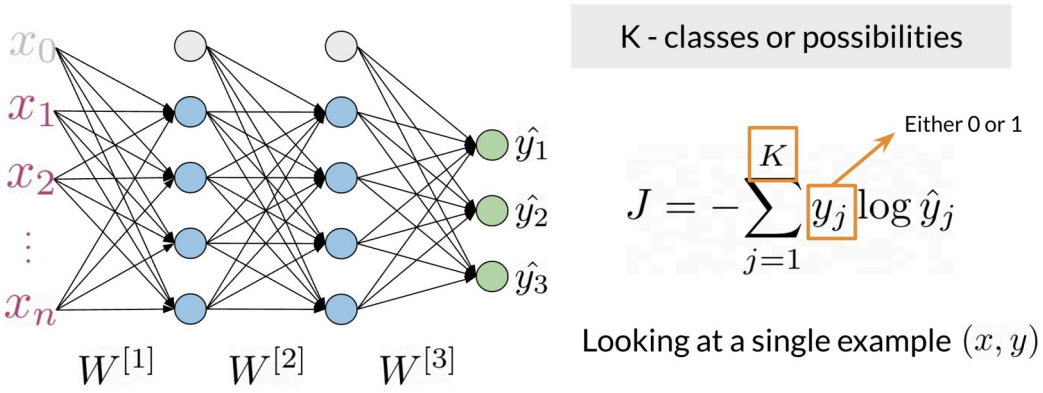
$$J = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{K} y_j^{<t>} \log \hat{y}_j^{<t>}$$

Average with respect to time

Summary

For RNNs the loss function is just an average through time!

The cost function used in an RNN is the cross entropy loss. If you were to visualize it



you are basically summing over the all the classes and then multiplying y_j times $\log \hat{y}_j$. If you were to compute the loss over several time steps, use the following formula:

$$J = -\frac{1}{T} \sum_{t=1}^{T} \sum_{j=1}^{K} y_j^{<t>} \log \hat{y}_j^{<t>}$$

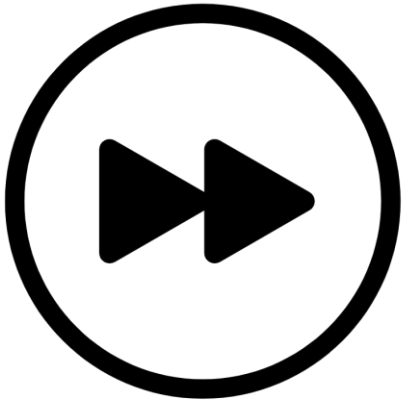
Note that we are simply summing over all the time steps and dividing by *T,* to get the average cost in each time step. Hence, we are just taking an average through time.

deeplearning.ai

Implementation Note

Outline

- scan() function in tensorflow
- Computation of forward propagation using abstractions



tf.scan() function

Take func fn and apply it to all of the elements in the list elems

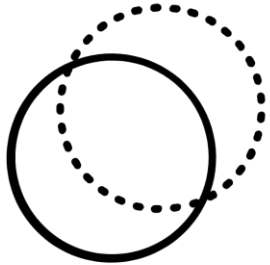
Could be used in the final computation of fn


```
def scan(fn, elems, initializer=None,
...):
    cur_value = initializer
    ys = []

    for x in elems:
        y, cur_value = fn(x, cur_value)
        ys.append(y)

    return ys, cur_value
```

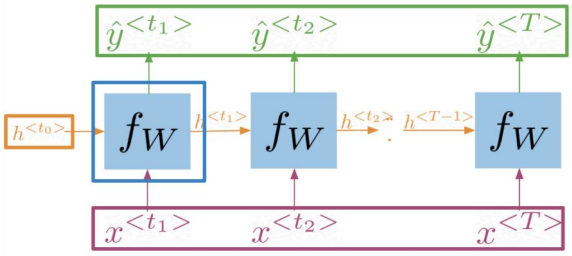
Frameworks like Tensorflow need this type of abstraction in order to perform Parallel computations and GPU usage

Summary

- Frameworks require abstractions
- tf.scan() mimics RNNs



The scan function is built as follows:




```
def scan(fn, elems, initializer=None, ...):
    cur_value = initializer
    ys = []
    for x in elems:
        y, cur_value = fn(x, cur_value)
        ys.append(y)
    return ys, cur_value
```

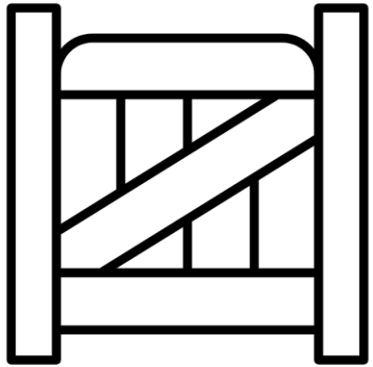
Note, that is basically what an RNN is doing. It takes the initializer, and returns a list of outputs (ys), and uses the current value, to get the next y and the next current value. These type of abstractions allow for much faster computation.

deeplearning.ai

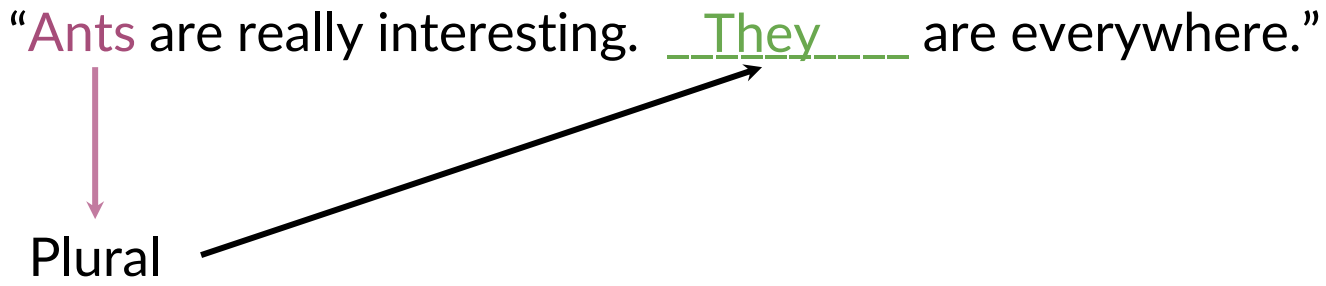
Gated Recurrent Units

Outline

- Gated recurrent unit (GRU) structure
- Comparison between GRUs and vanilla RNNs

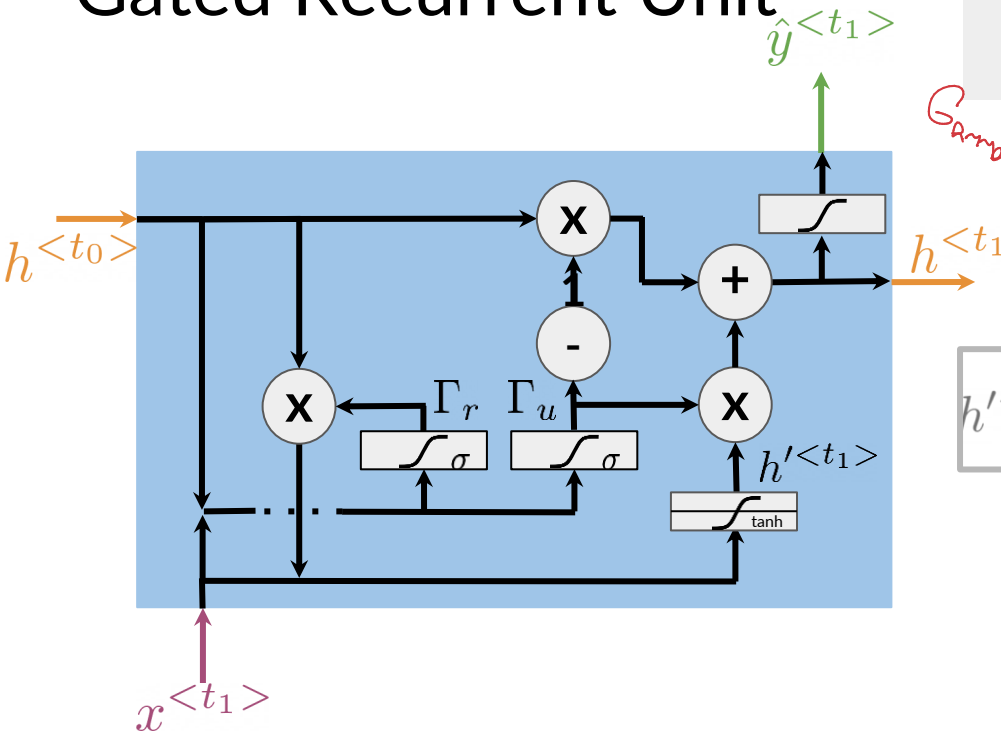


Gated Recurrent Units



Relevance and update gates to remember important prior information

Gated Recurrent Unit



Gates to keep/update relevant information in the hidden state

Gamma

$$\Gamma_r = \sigma(W_r[h^{<t_0>}, x^{<t_1>}] + b_r)$$

$$\Gamma_u = \sigma(W_u[h^{<t_0>}, x^{<t_1>}] + b_u)$$

$$h'^{<t_1>} = \tanh(W_h[\Gamma_r * h^{<t_0>}, x^{<t_1>}] + b_h)$$

Hidden state candidate

$$h^{<t_1>} = (1 - \Gamma_u) * h^{<t_0>} + \Gamma_u * h'^{<t_1>}$$

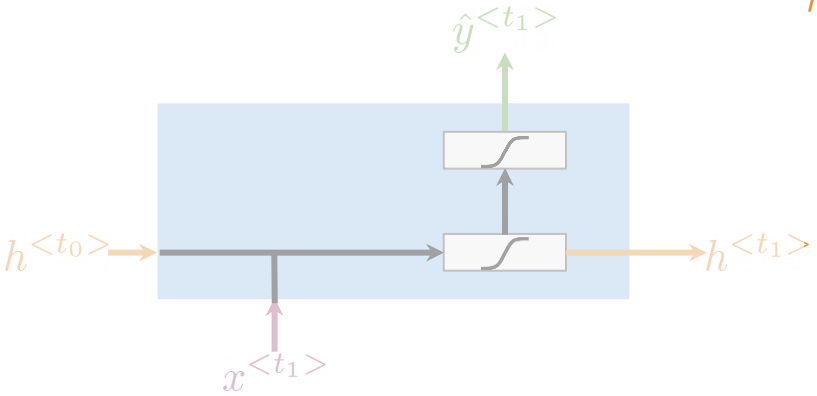
$$\hat{y}^{<t_1>} = g(W_y h^{<t_1>} + b_y)$$

Γ_r : relevance gate : Help to determine which info from the previous hidden state is relevant

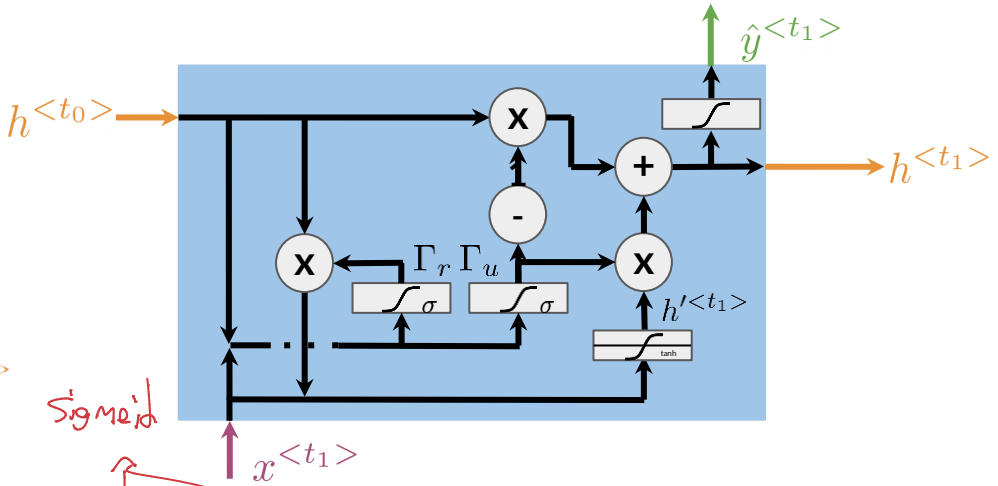
Γ_u : updates gate : How much of the info from the previous hidden state will be overwritten

$h'^{<t,>}$: Stores all the candidates for info thus could override the one contained in the previous hidden states

Vanilla RNN vs GRUs



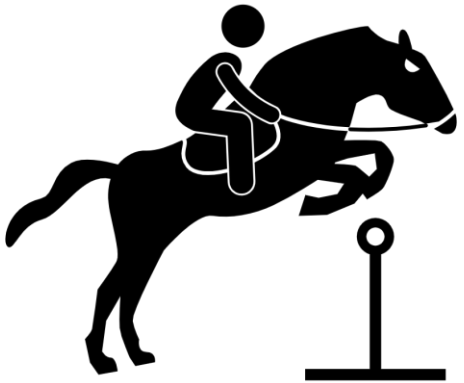
$$h^{<t>} = g(W_h[h^{<t-1>}, x^{<t>}] + b_h)$$
$$\hat{y}^{<t>} = g(W_{yh}h^{<t>} + b_y)$$



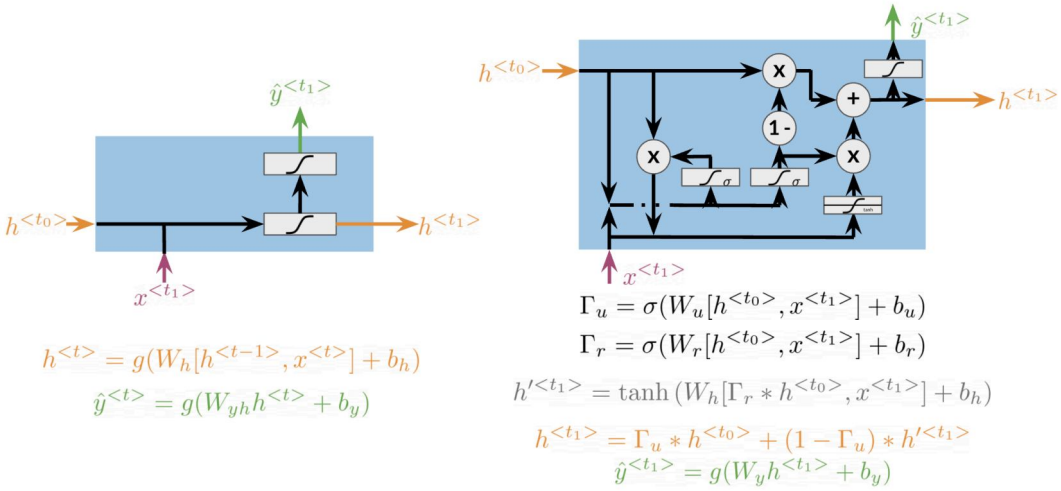
$$\Gamma_u = \sigma(W_u[h^{<t_0>}, x^{<t_1>}] + b_u)$$
$$\Gamma_r = \sigma(W_r[h^{<t_0>}, x^{<t_1>}] + b_r)$$
$$h'^{<t_1>} = \tanh(W_h[\Gamma_r * h^{<t_0>}, x^{<t_1>}] + b_h)$$
$$h^{<t_1>} = (1 - \Gamma_u) * h^{<t_0>} + \Gamma_u * h'^{<t_1>}$$
$$\hat{y}^{<t_1>} = g(W_y h^{<t_1>} + b_y)$$

Summary

- GRUs “decide” how to update the hidden state
- GRUs help preserve important information



Gated recurrent units are very similar to vanilla RNNs, except that they have a "relevance" and "update" gate that allow the model to update and get relevant information. I personally find it easier to understand by looking at the formulas:



To the left, you have the diagram and equations for a simple RNN. To the right, we explain the GRU. Note that we add 3 layers before computing *h* and *y*.

$$\Gamma_u = \sigma\left(W_u\left[h^{<t_0>}, x^{<t_1>}\right] + b_u\right)$$
$$\Gamma_r = \sigma\left(W_r\left[h^{<t_0>}, x^{<t_1>}\right] + b_r\right)$$
$$h'^{<t_1>} = \tanh\left(W_h\left[\Gamma_r * h^{<t_0>}, x^{<t_1>}\right] + b_h\right)$$

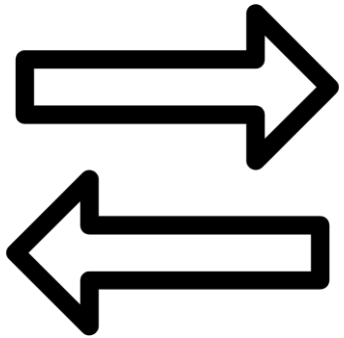
The first gate Γ_u allows you to decide how much you want to update the weights by. The second gate Γ_r, helps you find a relevance score. You can compute the new h by using the relevance gate. Finally you can compute *h*, using the update gate. GRUs "decide" how to update the hidden state. GRUs help preserve important information.

deeplearning.ai

Deep and Bi-directional RNNs

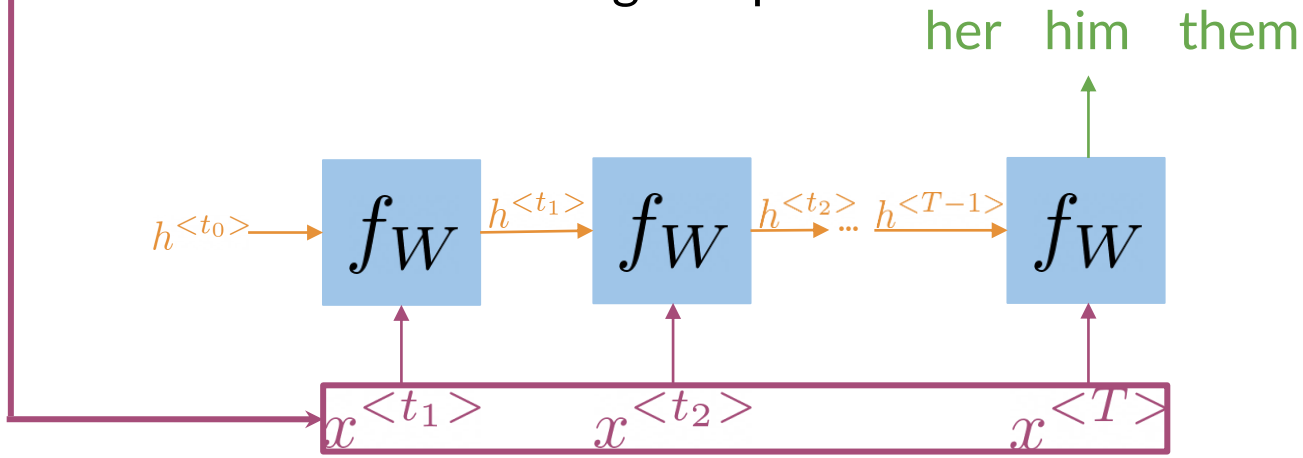
Outline

- How bidirectional RNNs propagate information
- Forward propagation in deep RNNs



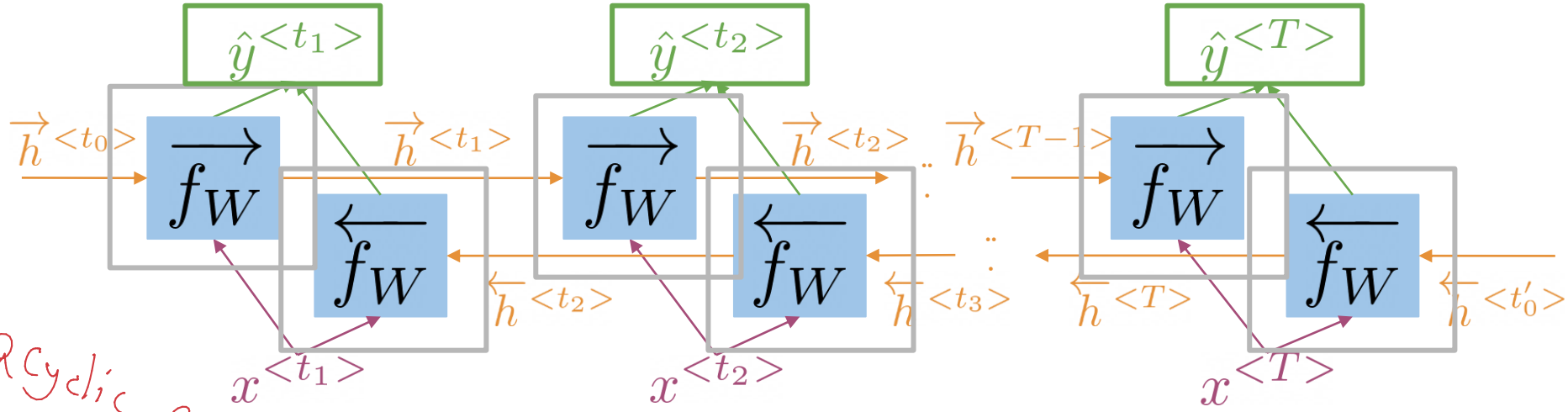
Bi-directional RNNs

I was trying really hard to get a hold of _____. **Louise,** finally answered when I was about to give up.



* Forward Prop will start in both ways

Bi-directional RNNs



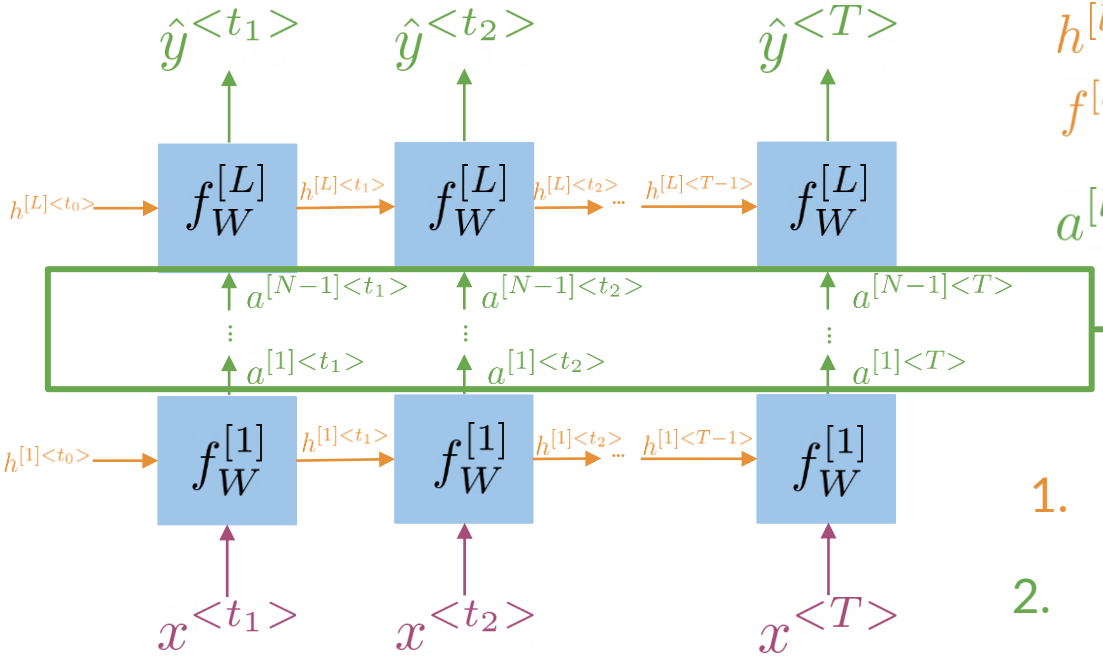
Q Cyclic Graph:

Information flows from the past and from the future **independently**

$$\hat{y}^{<t>} = g(W_y[\overrightarrow{h}^{<t>}, \overleftarrow{h}^{<t>}] + b_y)$$

$\hat{y}^{<t>} = g(W_y[\overrightarrow{h}^{<t>}, \overleftarrow{h}^{<t>}] + b_y)$

Deep RNNs



$$h^{[l]<t>} = f^{[l]}(W_h^{[l]}[h^{[l]<t-1>}, a^{[l-1]<t>}] + b_h^{[l]})$$

$$a^{[l]<t>} = f^{[l]}(W_a^{[l]} h^{[l]<t>} + b_a^{[l]})$$

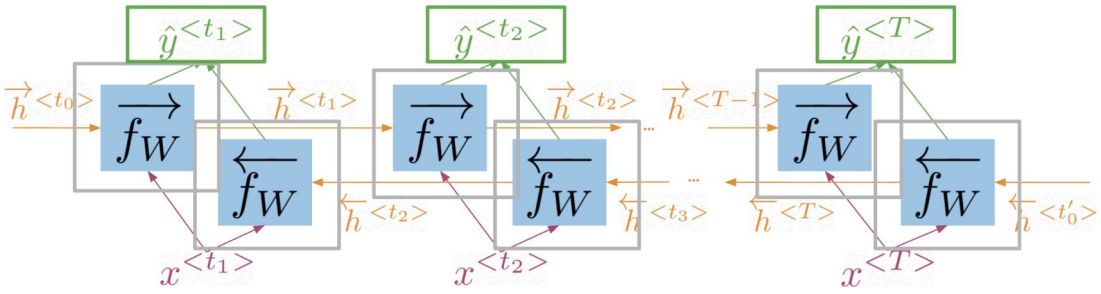
Intermediate layers and activations

1. Get hidden states for current layer
2. Pass the activations to the next layer

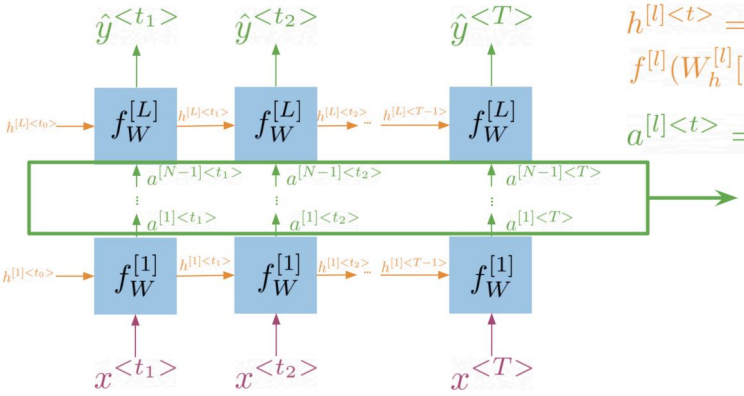
Summary

- In bidirectional RNNs, the outputs take information from the past and the future
- Deep RNNs have more than one layer, which helps in complex tasks

Bi-directional RNNs are important, because knowing what is next in the sentence could give you more context about the sentence itself.



So you can see, in order to make a prediction $\hat{y}$, you will use the hidden states from both directions and combine them to make one hidden state, you can then proceed as you would with a simple vanilla RNN. When implementing Deep RNNs, you would compute the following.



$$h^{[l]<t>} = f^{[l]}(W_h^{[l]}[h^{[l]<t-1>}, a^{[l-1]<t>}] + b_h^{[l]})$$

$$a^{[l]<t>} = f^{[l]}(W_a^{[l]} h^{[l]<t>} + b_a^{[l]})$$

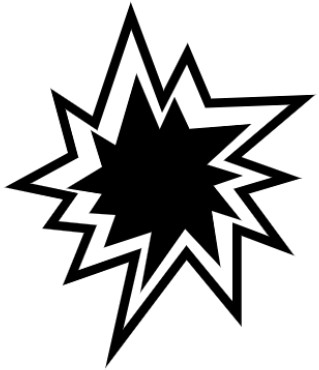
Note that at layer l, you are using the input from the bottom $a^{[l-1]}$ and the hidden state h^l. That allows you to get your new h, and then to get your new a, you will train another weight matrix W_a, which you will multiply by the corresponding h add the bias and then run it through an activation layer.

deeplearning.ai

RNNs and Vanishing Gradients

Outline

- Backprop through time
- RNNs and vanishing/exploding gradients
- Solutions



RNNs: Advantages

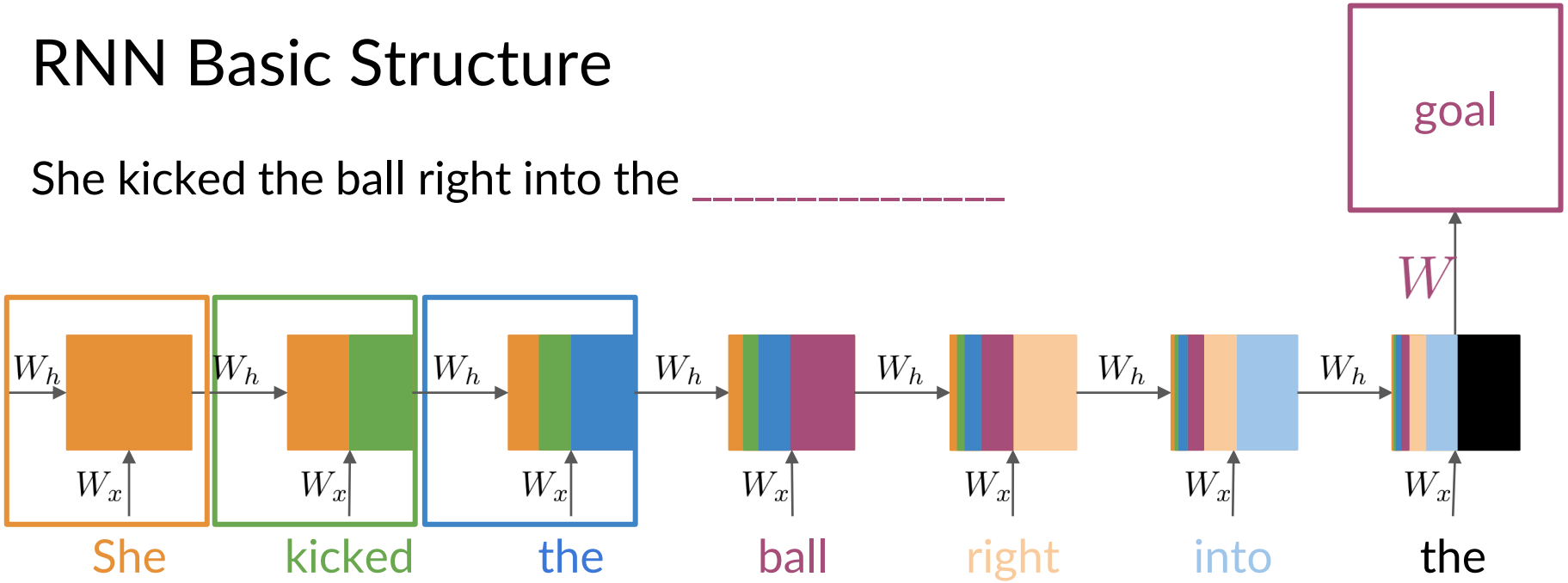
- Captures dependencies within a short range
- Takes up less RAM than other n-gram models

RNNs: Disadvantages

- Struggles to capture long term dependencies
- Prone to vanishing or exploding gradients

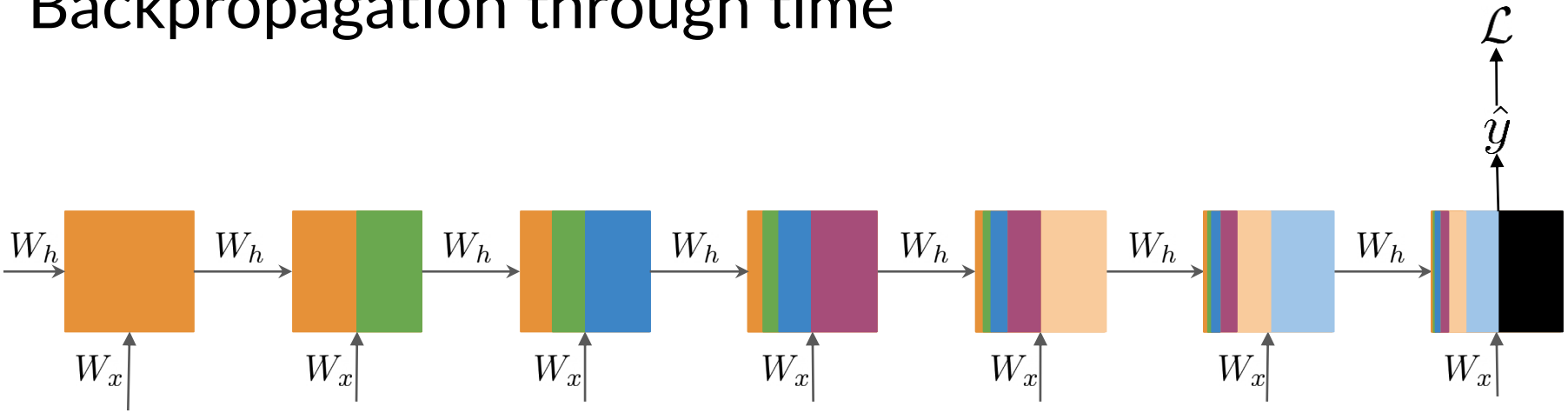
RNN Basic Structure

She kicked the ball right into the _____



Learnable parameters

Backpropagation through time



W_x
W_h
Same at every step

$$\frac{\partial L}{\partial W_h} \propto \sum_{1 \leq k \leq t} \left(\prod_{t \geq i > k} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W_h}$$

Gradient is proportional to a sum of partial derivative products

Backpropagation through time

$$\frac{\partial L}{\partial W_h} \propto \sum_{1 \le k \le t} \left(\prod_{t \ge i > k} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W_h}$$

Contribution of hidden state ***k***

Length of the product proportional to how far ***k*** is from ***t***

$$\frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \frac{\partial h_{t-2}}{\partial h_{t-3}} \frac{\partial h_{t-3}}{\partial h_{t-4}} \frac{\partial h_{t-4}}{\partial h_{t-5}} \frac{\partial h_{t-5}}{\partial h_{t-6}} \frac{\partial h_{t-6}}{\partial h_{t-7}} \frac{\partial h_{t-7}}{\partial h_{t-8}} \frac{\partial h_{t-8}}{\partial h_{t-9}} \frac{\partial h_{t-9}}{\partial h_{t-10}} \frac{\partial h_{t-10}}{\partial W_h}$$

Contribution of hidden state ***t-10***

Backpropagation through time

$$\frac{\partial L}{\partial W_h} \propto \sum_{1 \leq k \leq t} \left(\prod_{t \geq i > k} \frac{\partial h_i}{\partial h_{i-1}} \right) \frac{\partial h_k}{\partial W_h}$$

→ Contribution of hidden state ***k***

Length of the product proportional to how far ***k*** is from ***t***

low acc and gets worse over time

Partial derivatives <1	Contribution goes to 0	Vanishing Gradient
Partial derivatives >1	Contribution goes to infinity	**Exploding** Gradient

Convergence not during train

Solving for vanishing or exploding gradients

Vanishing

init W to: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$-1 \longrightarrow 0$

- Identity RNN with ReLU activation

Exploding

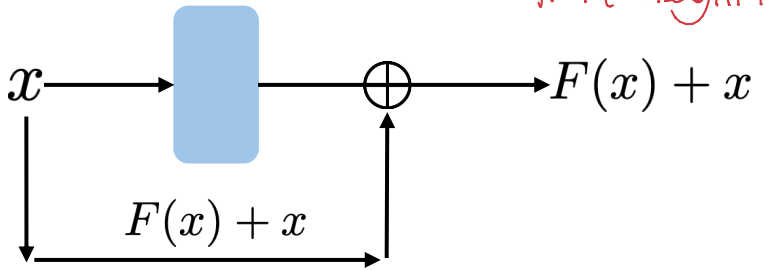
- Gradient clipping

$32 \longrightarrow 25$

any value $> 25 \Rightarrow 25$

limit magnitude of gradients

- Skip connections



skip over activation func, and add value from init inputs

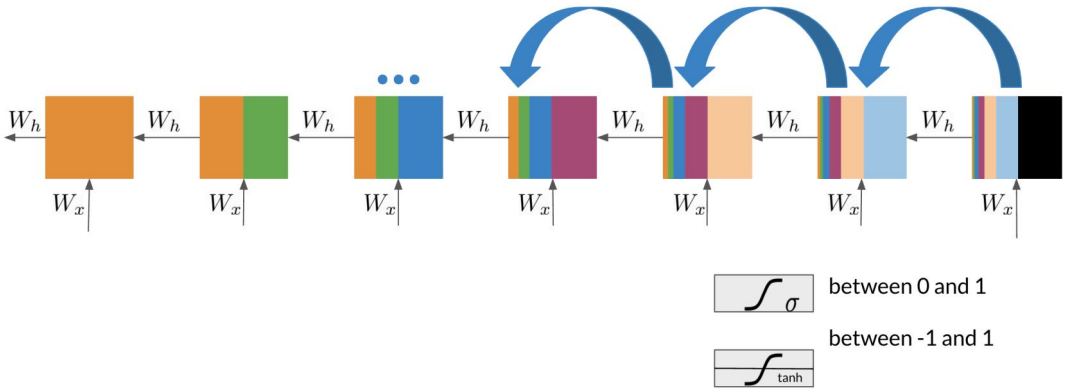
Advantages of RNNs

RNNs allow us to capture dependancies within a short range and they take up less RAM than other n-gram models.

Disadvantages of RNNs

RNNs struggle with longer term dependencies and are very prone to vanishing or exploding gradients.

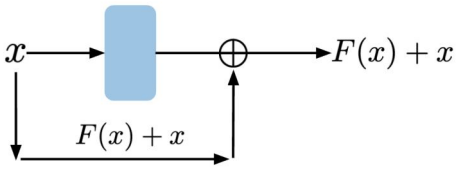
Note that as you are back-propagating through time, you end up getting the following:



Note that the *sigmoid* and *tanh* functions are bounded by 0 and 1 and -1 and 1 respectively. This eventually leads us to a problem. If you have many numbers that are less than |1|, then as you go through many layers, and you take the product of those numbers, you eventually end up getting a gradient that is very close to 0. This introduces the problem of vanishing gradients.

Solutions to Vanishing Gradient Problems

- Identity RNN with ReLU activation $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ -1 ⟶ 0
- Gradient clipping 32 ⟶ 25
- Skip connections

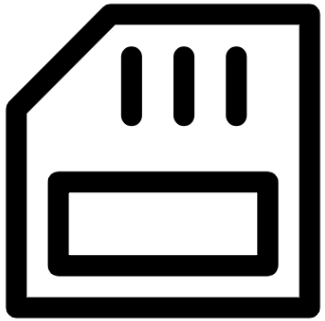


deeplearning.ai

Introduction to LSTMs

Outline

- Meet the Long short-term memory unit!
- LSTM architecture
- Applications



LSTMs: a memorable solution

- Learns when to remember and when to forget
- Basic anatomy:
  - A cell state : It's memory
  - A hidden state : where computations are performed during training to decide on what changes to make
  - Multiple gates : their transform the state in the network
- Gates allow gradients to avoid vanishing and exploding

LSTMs: Based on previous understanding

Starting point with some irrelevant information

↓

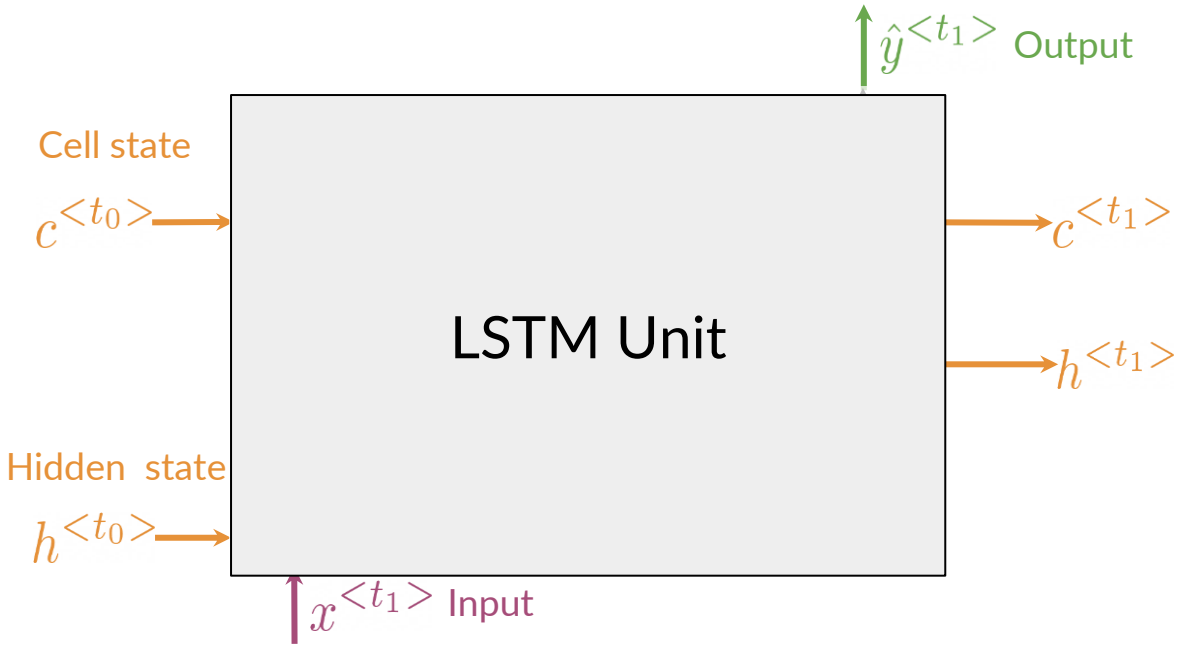
Update Cell and Hidden States constantly

Discard anything irrelevant

Add important new information → Gates

Produce output

Gates in LSTM



1. Forget Gate: information that is no longer important

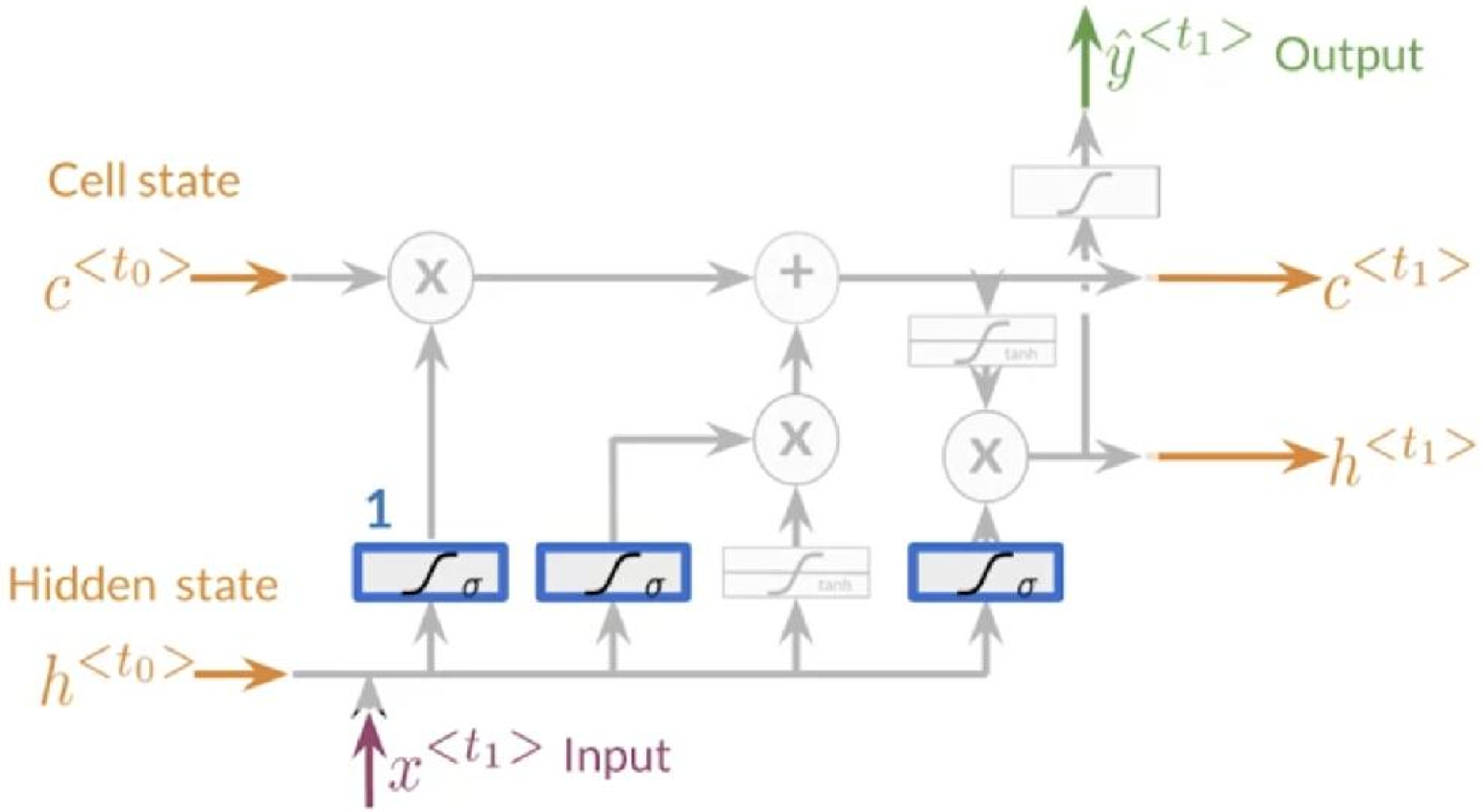
2. Input Gate: information to be stored

3. Output Gate: information to use at current step

in forget gate:

input & Previous hidden states is used to find which info from cell state is useless

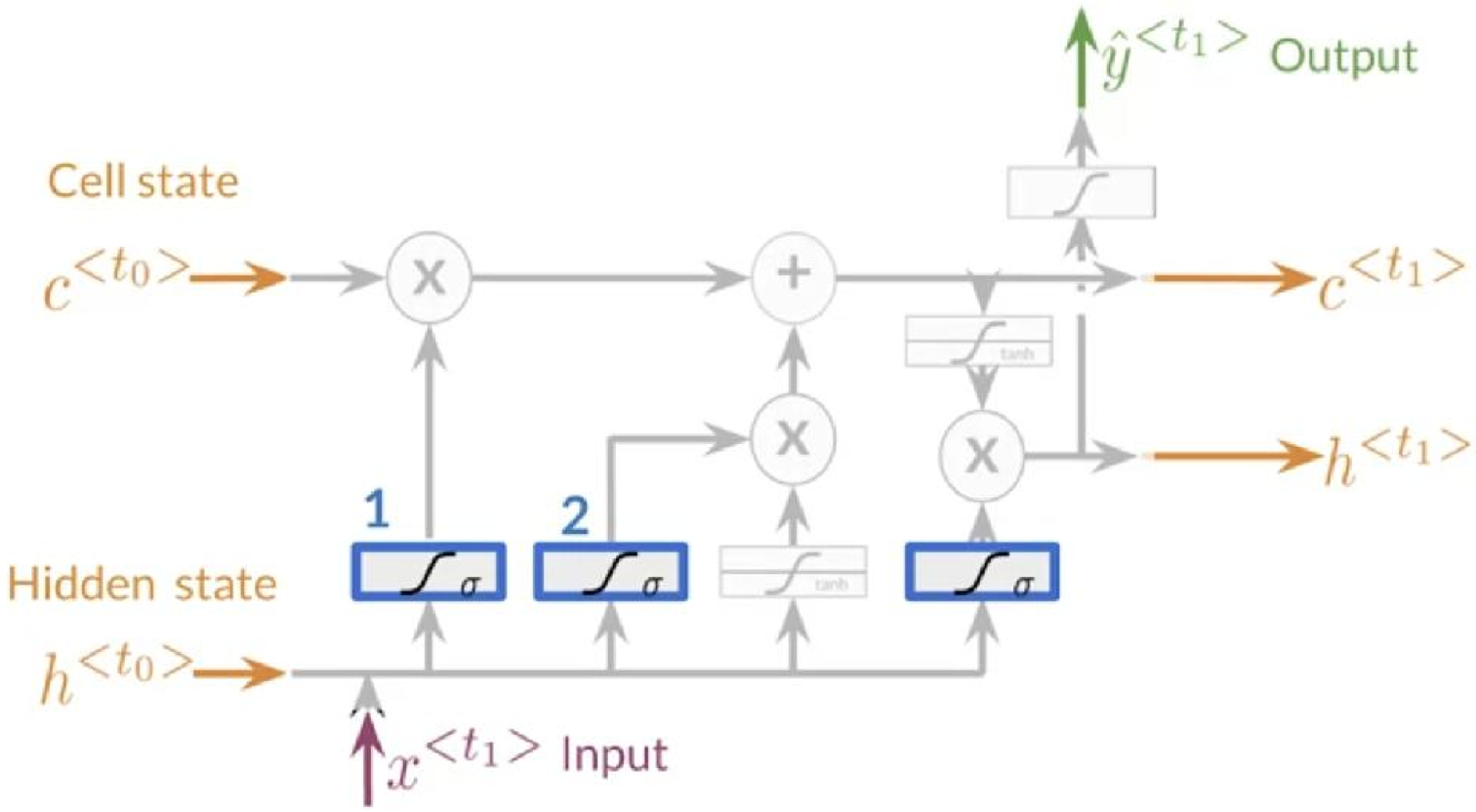
Gates in LSTM



1. Forget Gate:
information that is no longer important

input gate is used to decide which info from input & prev h states is relevent so it is added to cell state

Gates in LSTM

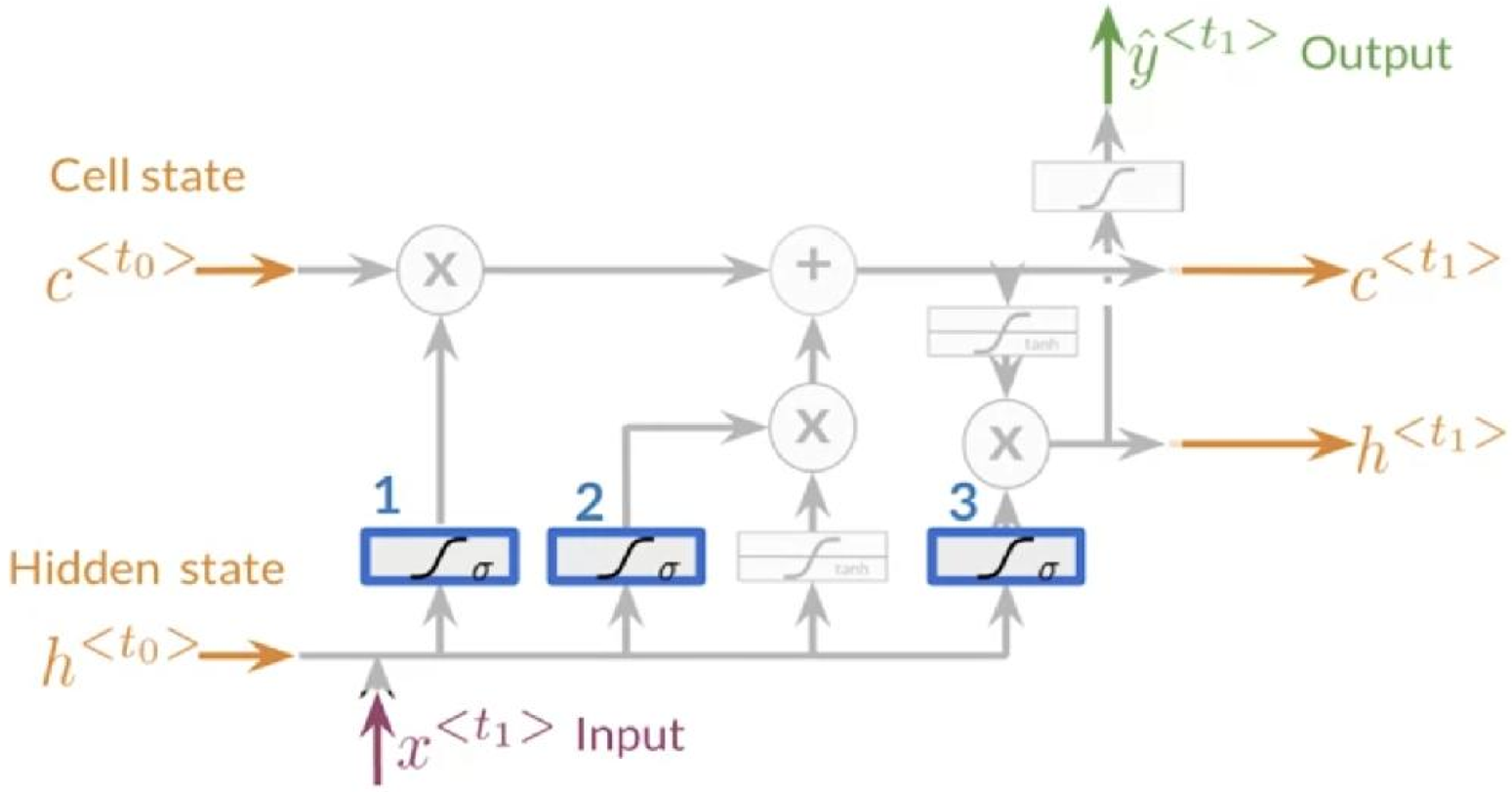


1. Forget Gate: information that is no longer important

2. Input Gate: information to be stored

the output gate determines the info from the cell state that is stored in the hidden state and used to construct an output at the given timestep

Gates in LSTM



1. **Forget Gate:** information that is no longer important

2. **Input Gate:** information to be stored

3. **Output Gate:** information to use at current step

Applications of LSTMs

Next-character prediction

Chatbots

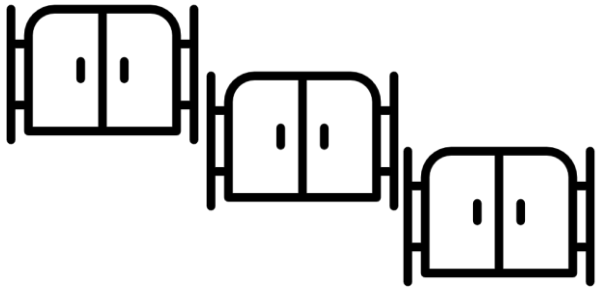
Music composition

Image captioning

Speech recognition

Summary

- LSTMs offer a solution to vanishing gradients
- Typical LSTMs have a cell and three gates: and h states
  - Forget gate
  - Input gate
  - Output gate



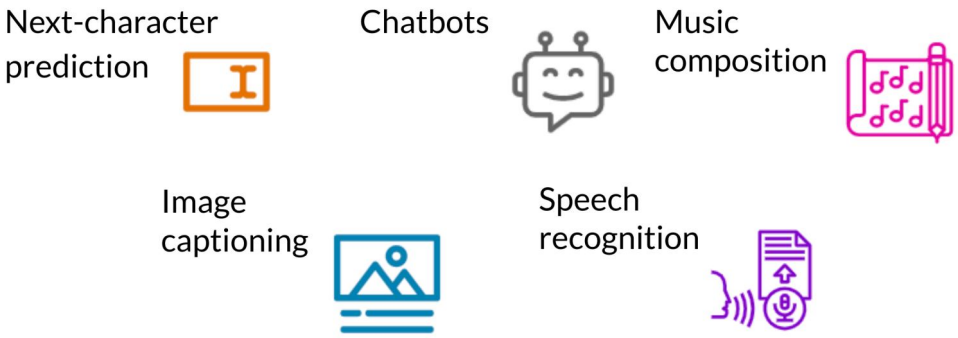
The LSTM allows your model to remember and forget certain inputs. It consists of a cell state and a hidden state with three gates. The gates allow the gradients to flow unchanged. You can think of the three gates as follows:

Input gate: tells you how much information to input at any time point.

Forget gate: tells you how much information to forget at any time point.

Output gate: tells you how much information to pass over at any time point.

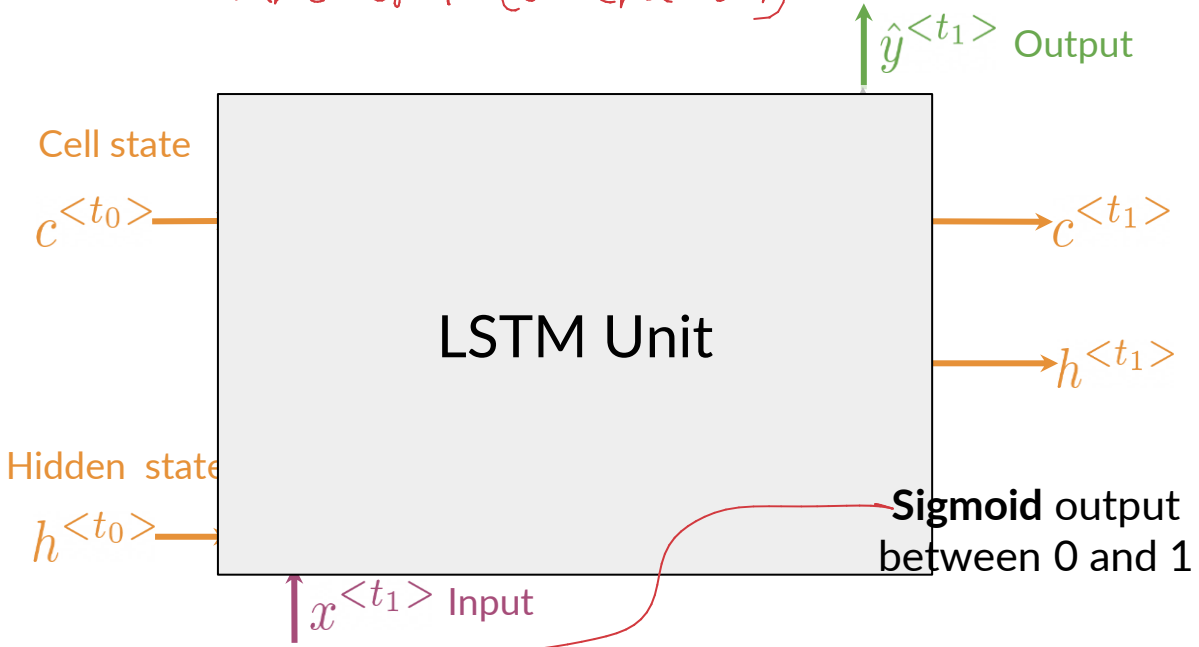
There are many applications you can use LSTMs for, such as:



deeplearning.ai

LSTM Architecture

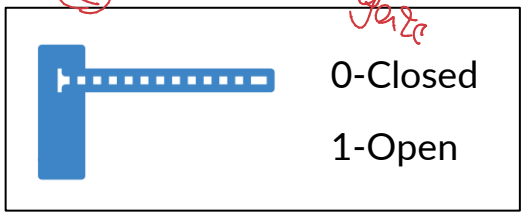
Gates in LSTM



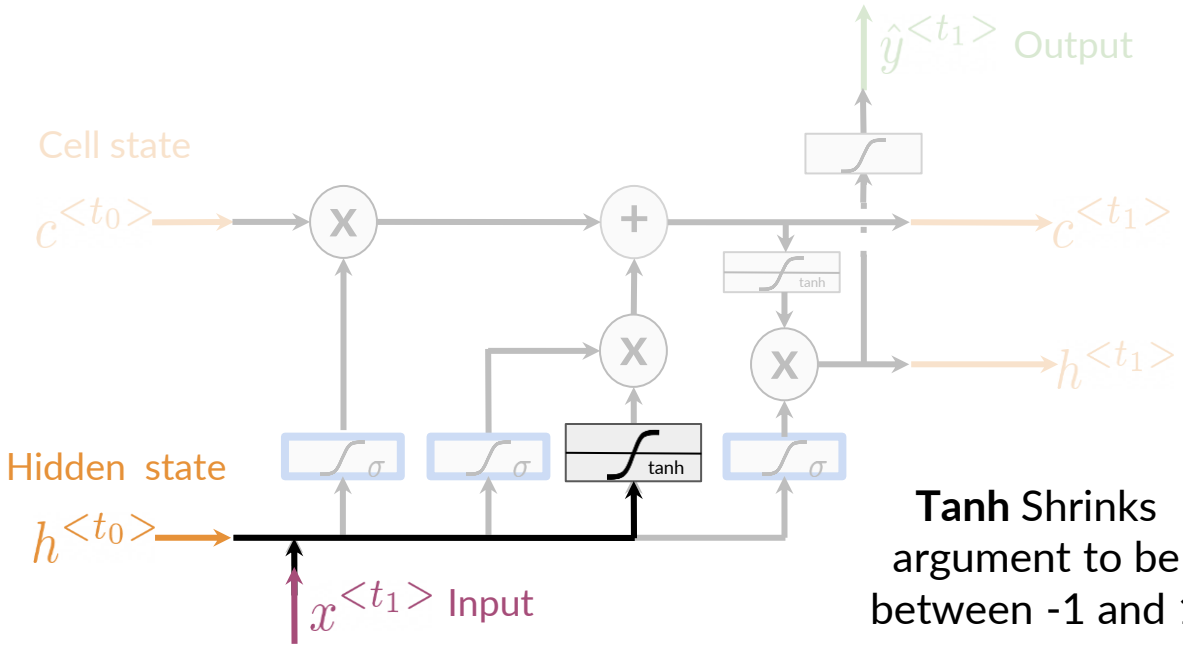
1. Forget Gate: information that is no longer important

2. Input Gate: information to be stored

3. Output Gate: information to use at current step



Candidate Cell State



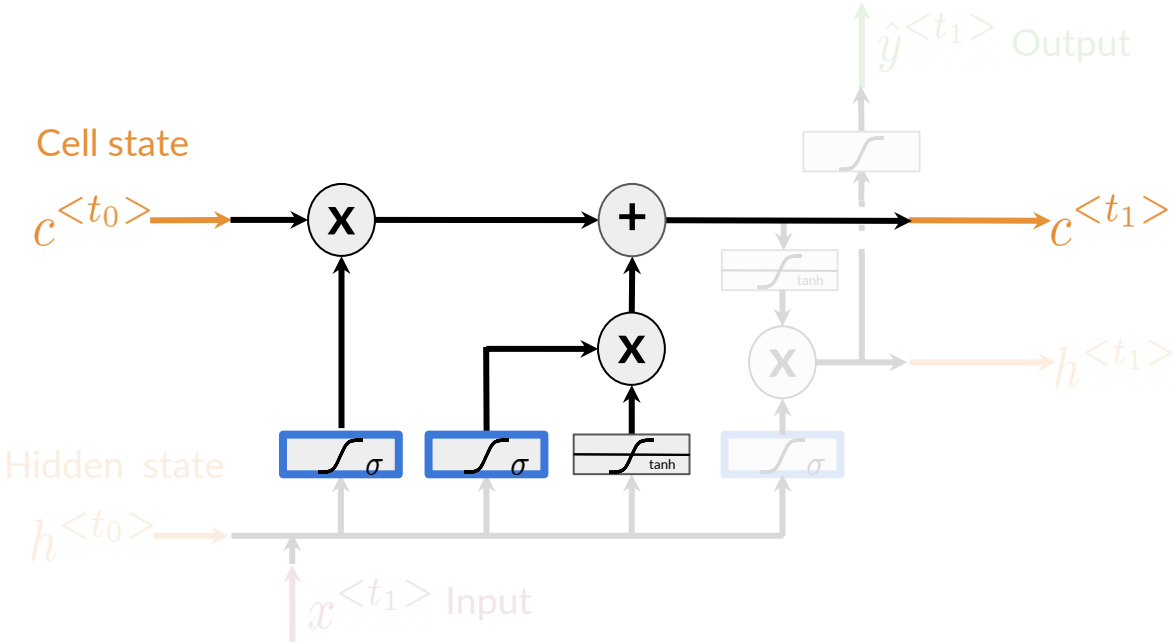
Candidate cell state

Information from the previous **hidden state** and current **input**

Tanh Shrinks argument to be between -1 and 1

→ **Other activations could be used**

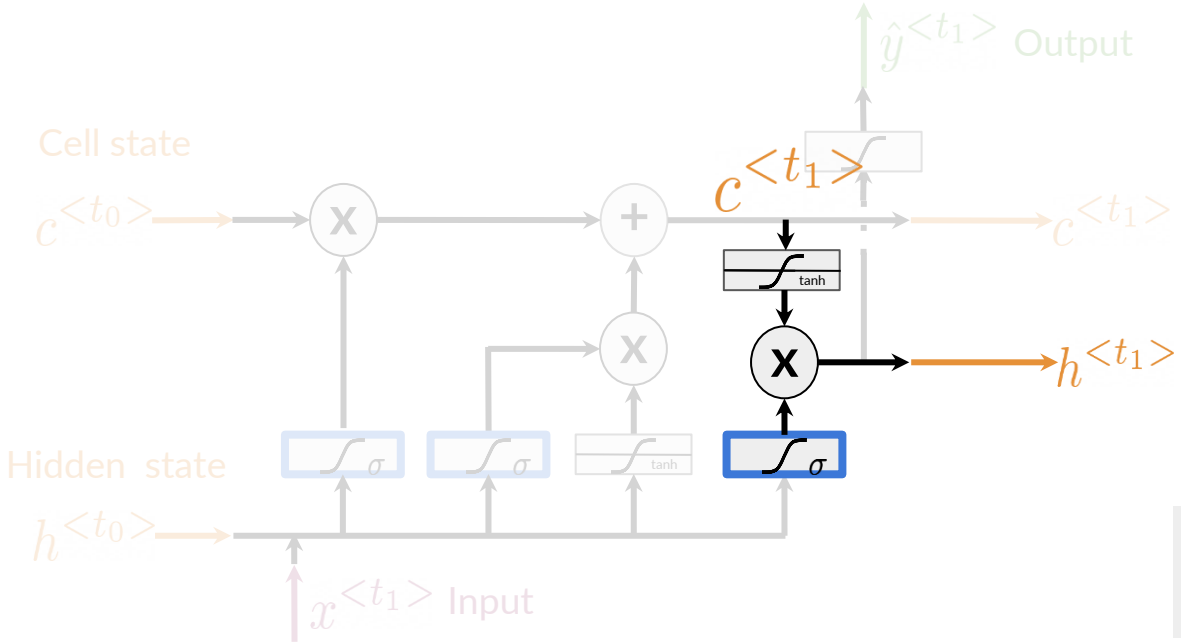
New Cell State



New Cell state

Add information from the **candidate cell state** using the **forget** and **input gates**

New Hidden State



New Hidden State

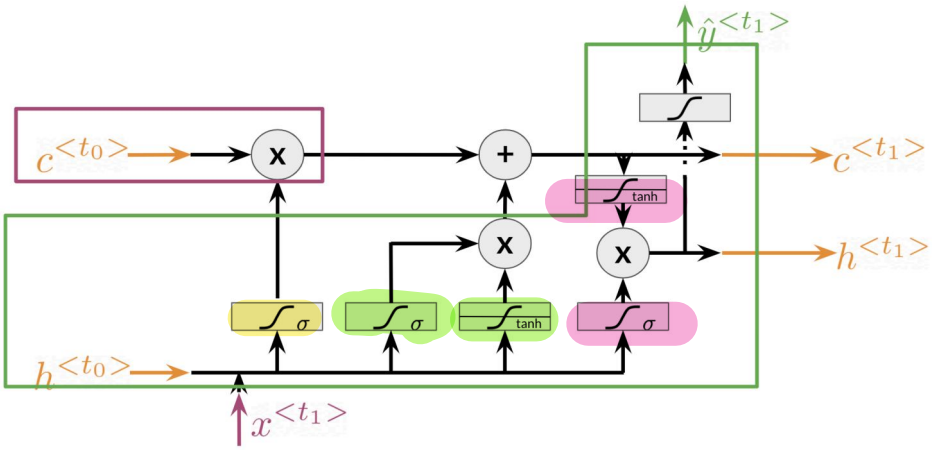
Select information from the **new cell state** using the **output gate**

The **Tanh** activation could be omitted

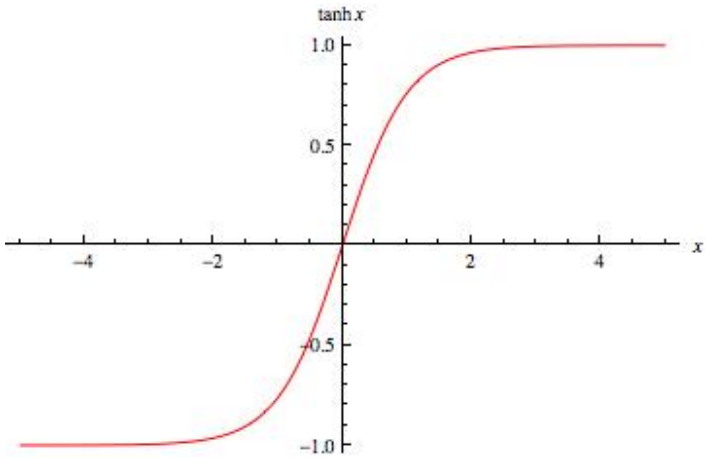
Summary

- LSTMs use a series of gates to decide which information to keep:
  - Forget gate decides what to keep
  - Input gate decides what to add
  - Output gate decides what the next hidden state will be

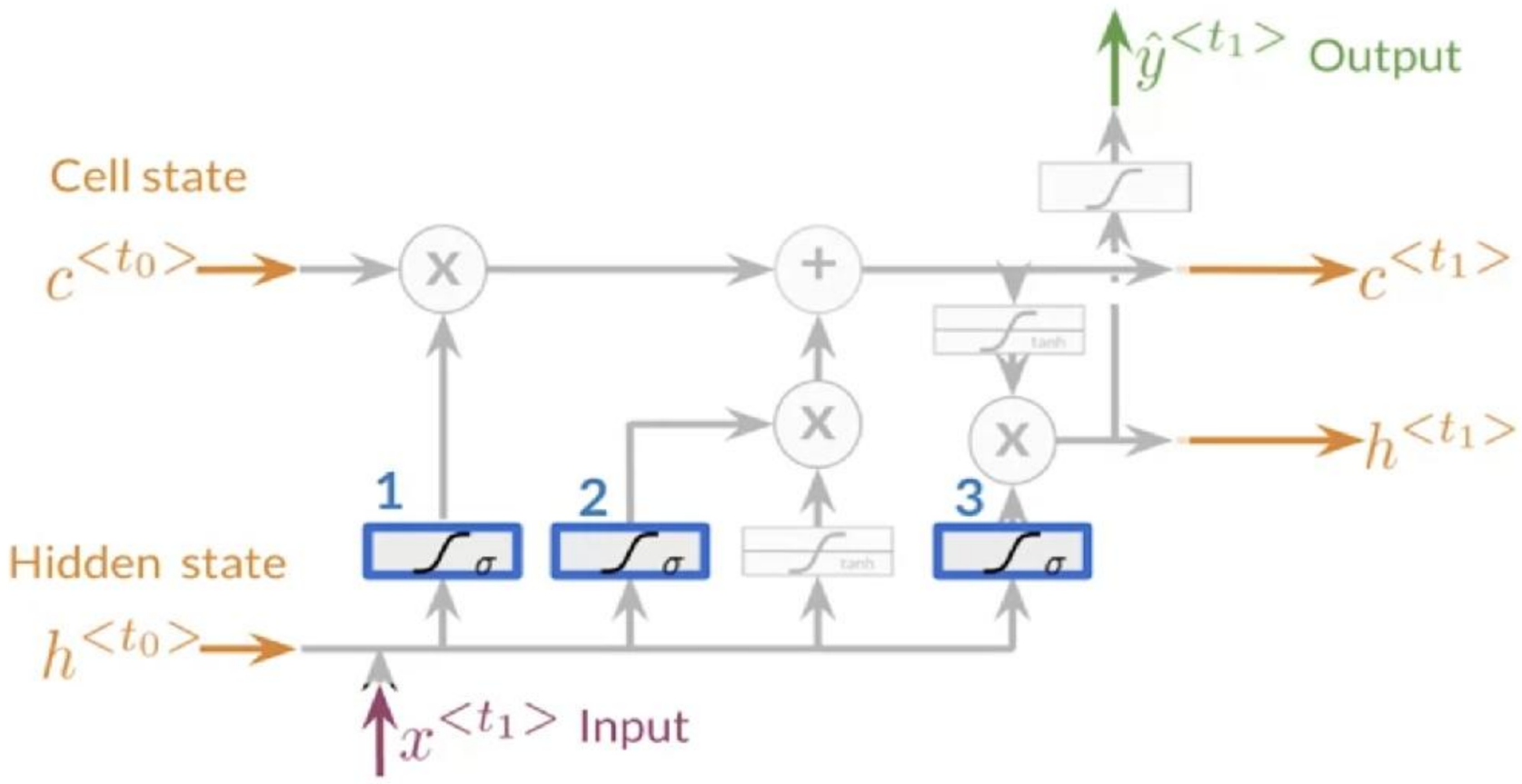
The LSTM architecture could get complicated and don't worry about it if you do not understand it. I personally prefer looking at the equation, but I will try to give you a visualization for now and later this week we will take a look at the equations.



Note that there is the cell state and the hidden state, and then there is the output state. The **forget gate** is the first activation in the drawing above. It makes use of the previous hidden state $h^{<t_0>}$ and the input $x^{<t_1>}$. The **input gate** makes use of the next two activations, the *sigmoid* and the *tanh* . Finally the **output gate** makes use of the last activation and the *tanh* right above it. This is just an overview of the architecture, we will dive into the details once we introduce the equations.



Gates in LSTM



1. Forget Gate: information that is no longer important

2. Input Gate: information to be stored

3. Output Gate: information to use at current step

Note the forget gate (1), input gate (2) and output gate (3) marked in blue. In contrast with vanilla RNNs, there is the cell state in addition to the hidden state. The idea of the **forget gate** to drop the information that is no longer important. It makes use of the previous hidden state $h^{<t_0>}$ and the input $x^{<t_1>}$. The **input gate** makes sure to keep the relevant information that needs to be stored. Finally the **output gate** creates an output that is used at the current step.

LSTM equations (optional):

For better understanding, take a look at the LSTM equations and relate them to the figure above.

The forget gate: $f = \sigma\left(W_f\left[h_t{-}_1; x_t\right] + b_f\right)$ (marked with a blue 1)

The input gate: $i = \sigma\left(W_i\left[h_t{-}_1; x_t\right] + b_i\right)$ (marked with a blue 2)

The gate gate (candidate memory cell): $g = \tanh\left(W_g\left[h_t{-}_1; x_t\right] + b_g\right)$

The cell state: $c_t = f \odot c_{t-1} + i \odot g$

The output gate: $o = \sigma\left(W_o\left[h_{t-1}; x_t\right] + b_o\right)$ (marked with a blue 3)

The output of LSTM unit: $h_t = o_t \odot tanh(c_t)$

deeplearning.ai

Introduction to Named Entity Recognition

What is Named Entity Recognition?

- Locates and extracts predefined entities from text
- Places, organizations, names, time and dates

Types of Entities

Thailand:
Geographical

Google:
Organization

Indian:
Geopolitical

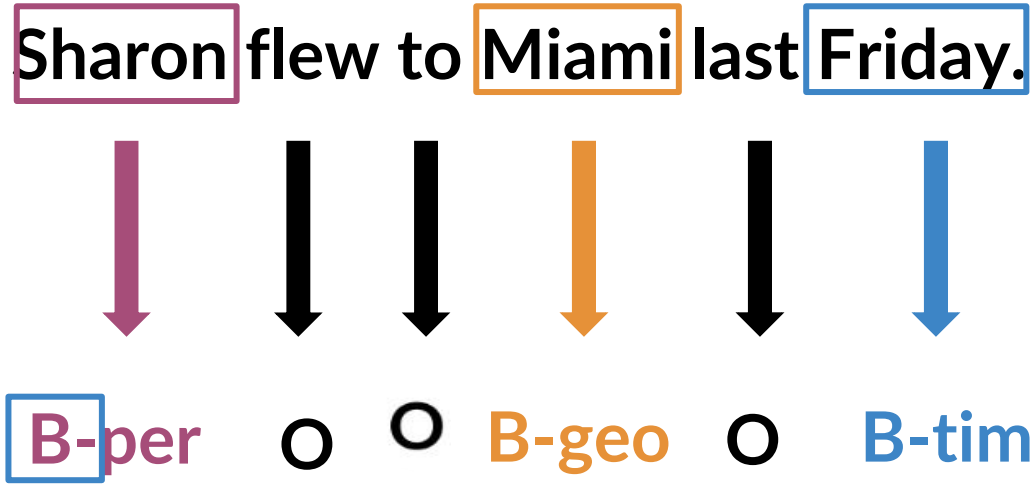
More Types of Entities

December:
Time Indicator

Egyptian statue:
Artifact

Barack Obama:
Person

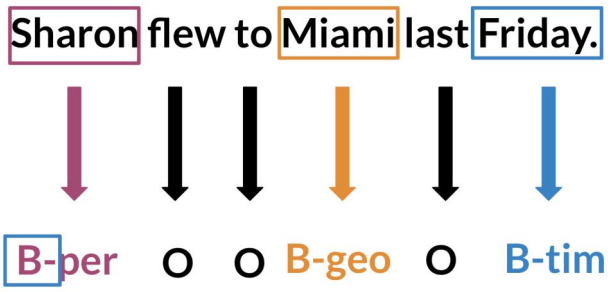
Example of a labeled sentence



Applications of NER systems

- Search engine efficiency
- Recommendation engines
- Customer service
- Automatic trading

Named Entity Recognition (NER) locates and extracts predefined entities from text. It allows you to find places, organizations, names, time and dates. Here is an example of the model you will be building:



NER systems are being used in search efficiency, recommendation engines, customer service, automatic trading, and many more.

These are the LSTM equations related to the gates I had previously spoken about:

The forget gate: $f = \sigma\left(W_f\left[h_t - 1; x_t\right] + b_f\right)$

The input gate: $i = \sigma\left(W_i\left[h_t\ - 1; x_t\right] + b_i\right)$

The gate gate: $g = \tanh(W_g\left[h_t\ - 1; x_t\right] + b_g)$

The cell state: $c_t\ = f \odot c_{t-1} + i \odot g$

The output gate: $o = \sigma\left(W_o\left[h_{t-1}; x_t\right] + b_o\right)$

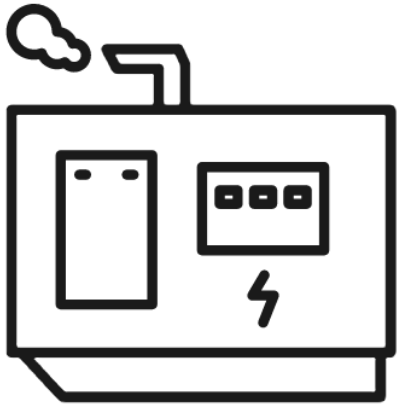
Take a look at the equations. You can think of the forget gate as a gate that tells you how much information to forget, the input gate, tells you how much information to pick up. I like to think of the gate gate as the gate containing information. This is multiplied by the input gate (which tells you how much of that information to keep).

deeplearning.ai

Training NERs: Data Processing

Outline

- Convert words and entity classes into arrays
- Token padding
- Create a data generator



Processing data for NERs

- Assign each class a number
- Assign each word a number

Sharon flew to Miami last Friday.

[4282, 853, 187, 5388, 2894, 7]

B-Per O O B-geo O B-tim

1 2 3

Token padding

For LSTMs, all sequences need to be the same size.

- Set sequence length to a certain number
- Use the <PAD> token to fill empty spaces

Training the NER

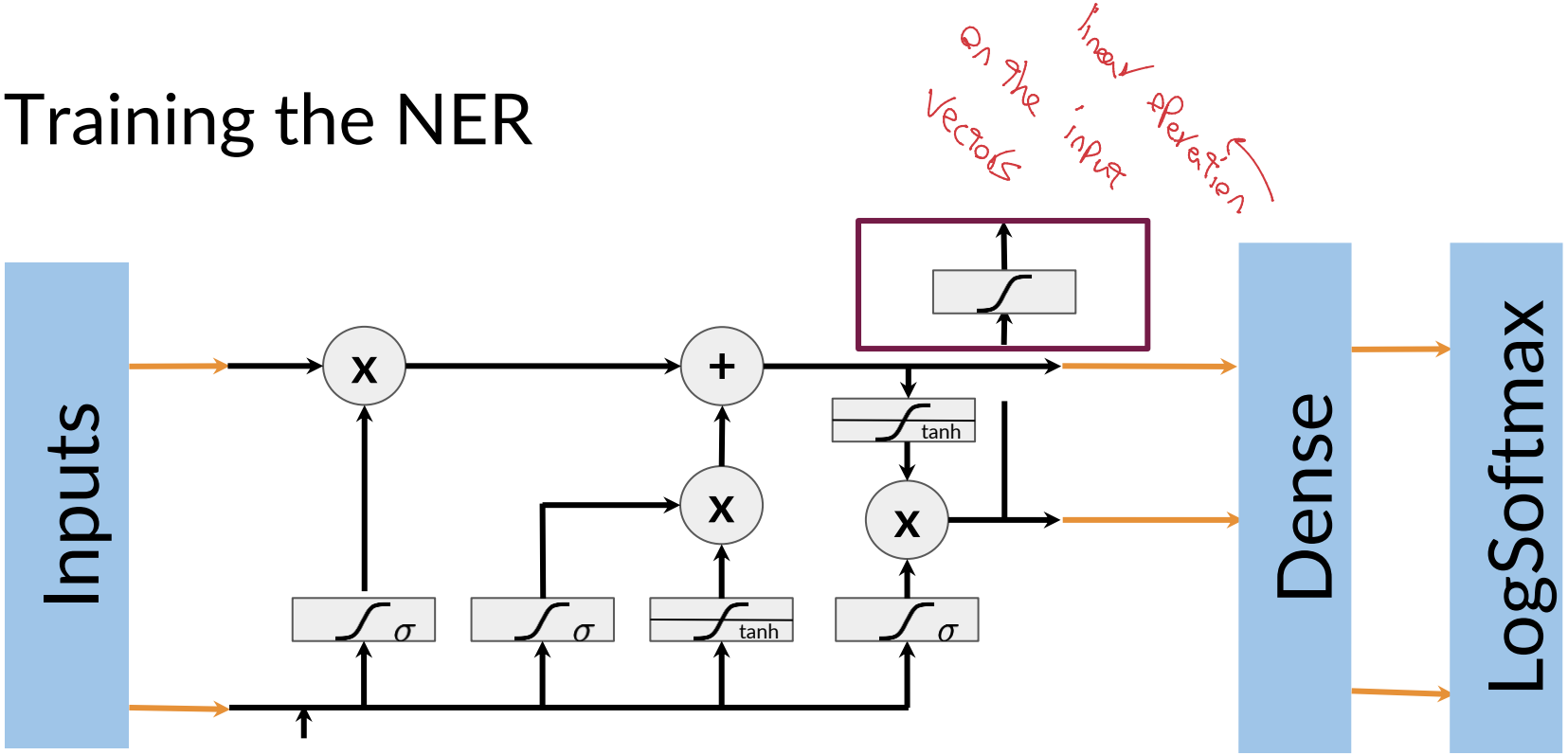
1. Create a tensor for each input and its corresponding number
2. Put them in a batch → better to use
3. Feed it into an LSTM unit
4. Run the output through a dense layer
5. Predict using a log softmax over K classes

64, 128, 256, 512 ... (Power of 2)

☆ log softmax: gives better numerical performance and gradient optimization

↳ Number of possible outputs

Training the NER



Layers in TensorFlow

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(),
    tf.keras.layers.LSTM(),
    tf.keras.layers.Dense(),
])
```

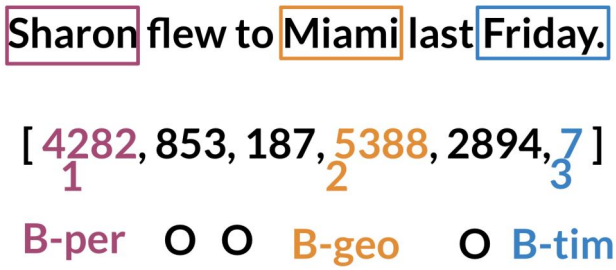
Summary

- Convert words and entities into same-length numerical arrays
- Train in batches for faster processing
- Run the output through a final layer and activation

Processing data is one of the most important tasks when training AI algorithms. For NER, you have to:

- Convert words and entity classes into arrays:
- Pad with tokens: Set sequence length to a certain number and use the <PAD> token to fill empty spaces
- Create a data generator:

Once you have that, you can assign each class a number, and each word a number.



Training an NER system:

1. Create a tensor for each input and its corresponding number

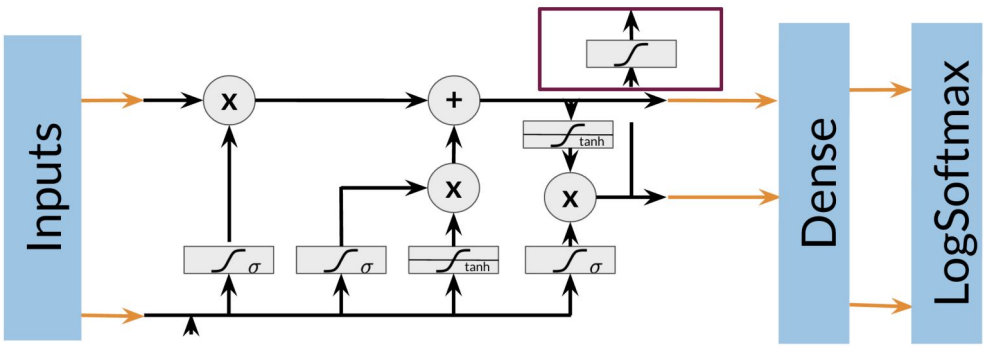
2. Put them in a batch ==> 64, 128, 256, 512 ...

3. Feed it into an LSTM unit

4. Run the output through a dense layer

5. Predict using a log softmax over K classes

Here is an example of the architecture:



Note that this is just one example of an NER system. You can have different architectures.

deeplearning.ai

Computing Accuracy

Evaluating the model

1. Pass test set through the model
2. Get arg max across the prediction array
3. Mask padded tokens
4. Compare outputs against test labels

Evaluating the model in Python

```
def masked_accuracy(y_true, y_pred):
    mask = ...
    y_pred_class = tf.math.argmax(y_pred, axis=-1)
    matches_true_pred = tf.equal(y_true, y_pred_class)
    matches_true_pred *= mask
    masked_acc = tf.reduce_sum(acc) / tf.reduce_sum(mask)

    return masked_acc
```

① : where you identify any token IDs you need to skip over during evaluation like Pad token

Summary

- If padding tokens, remember to mask them when computing accuracy
- Coding assignment!

To compare the accuracy, just follow the following steps:

- Pass test set through the model
- Get arg max across the prediction array
- Mask padded tokens
- Compare with the true labels.

deeplearning.ai

Siamese Networks

Question Duplicates

How old are you? = What is your age?

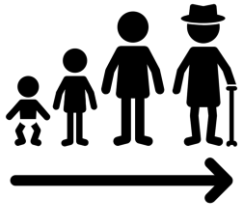
Where are you from? ≠ Where are you going?

What do Siamese Networks learn?

I am happy because I am learning

Classification: categorize things

Siamese Networks: Identify similarity between things



What is your age?

How old are you?

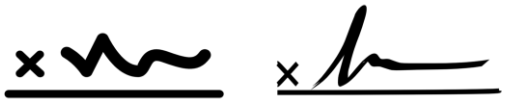
Difference or Similarity

Siamese Networks in NLP

What is your age?

How old are you?

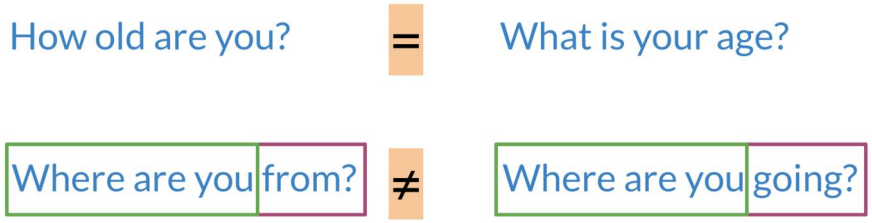
Handwritten checks



Question duplicates

Queries

It is best to describe what a Siamese network is through an example.

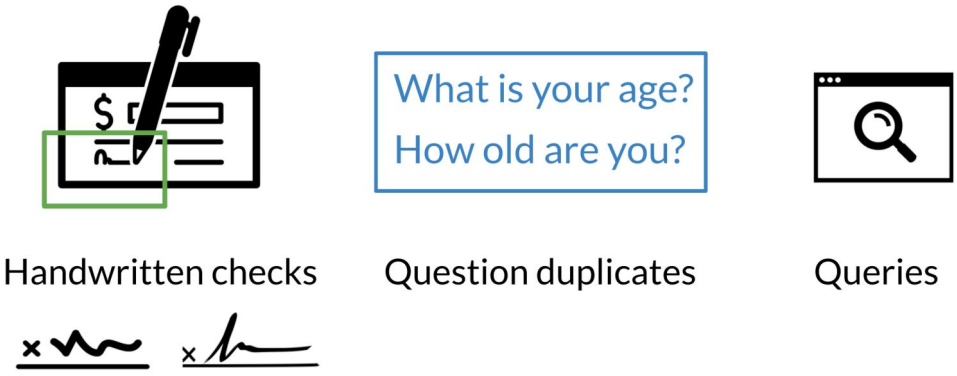


Note that in the first example above, the two sentences mean the same thing but have completely different words. While in the second case, the two sentences mean completely different things but they have very similar words.

Classification : learns what makes an input what it is.

Siamese Networks : learns what makes two inputs the same

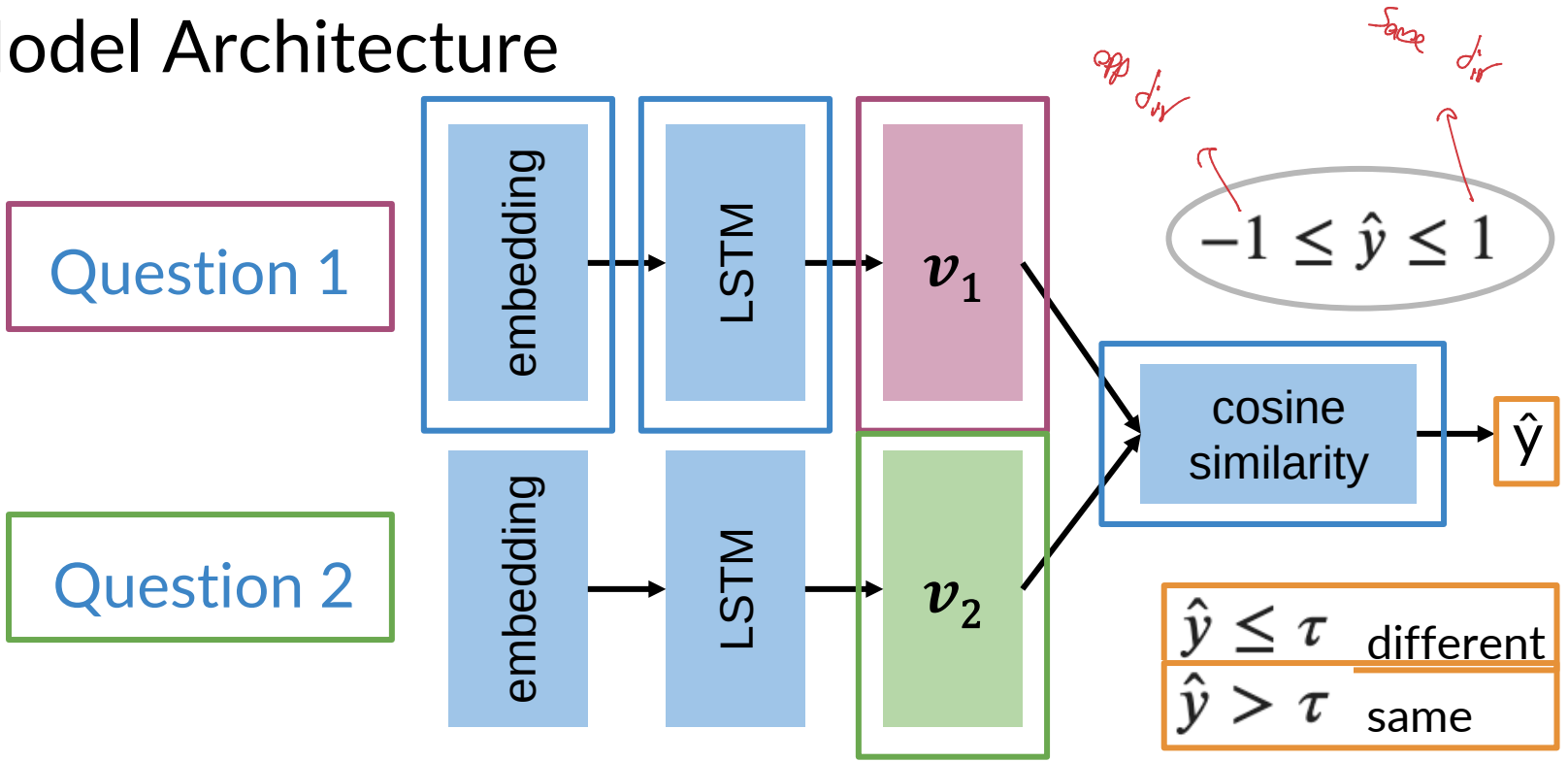
Here are a few applications of siamese networks:



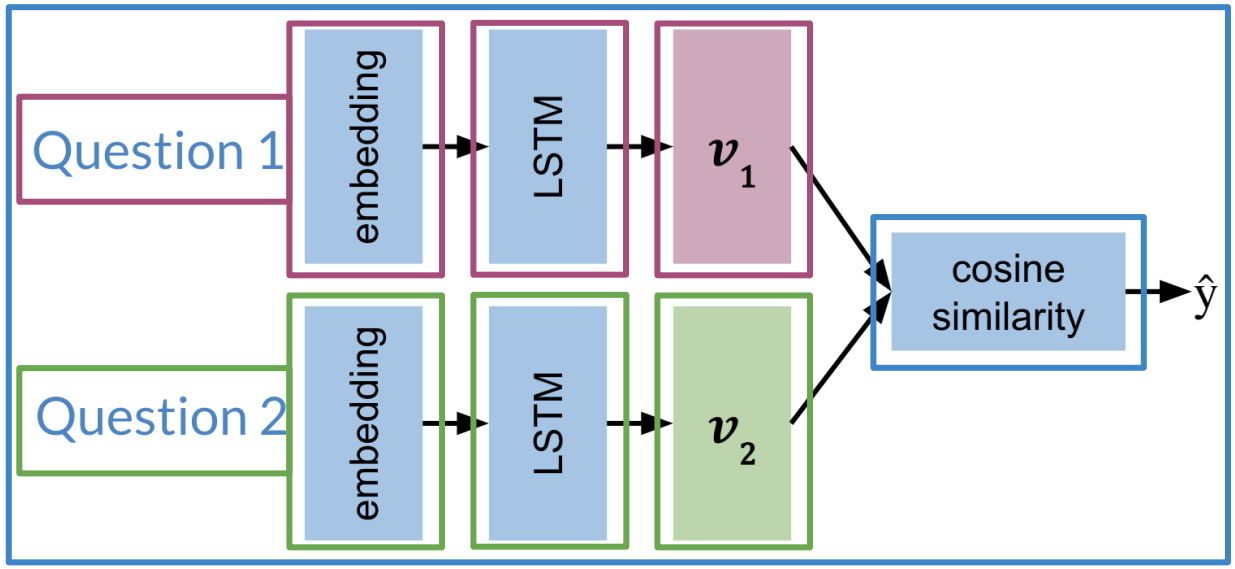
deeplearning.ai

Architecture

Model Architecture

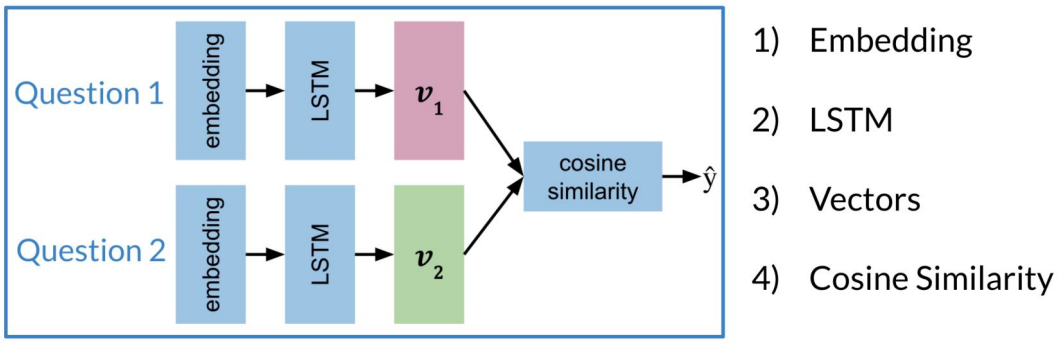


Model Architecture



1) Inputs
2) Embedding
3) LSTM
4) Vectors
5) Cosine Similarity

The model architecture of a typical siamese network could look as follows:



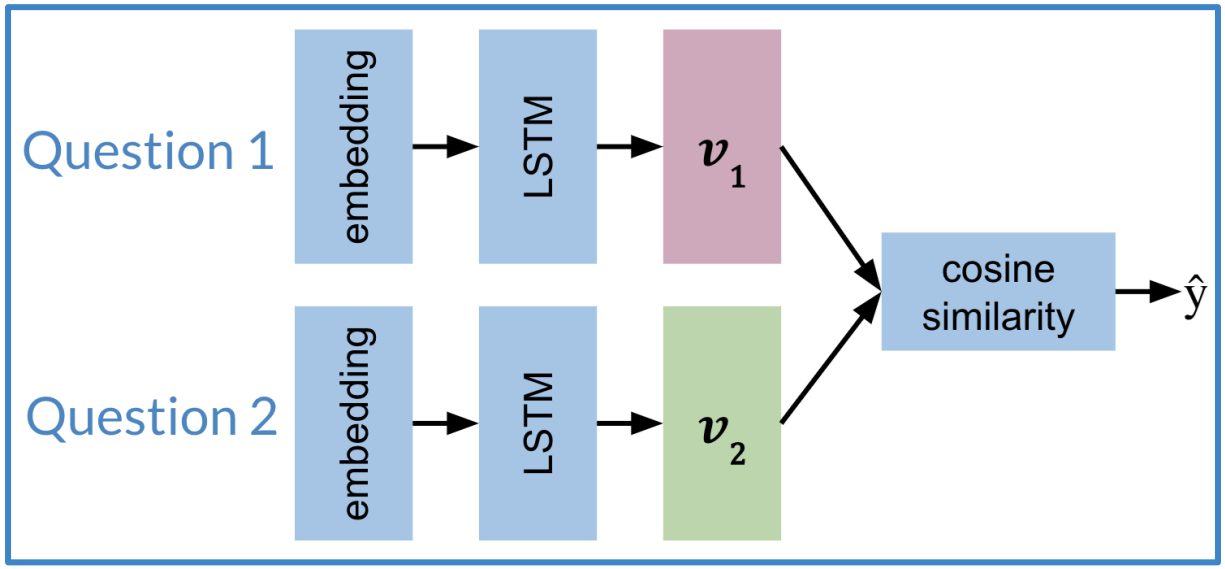
These two sub-networks are sister-networks which come together to produce a similarity score. Not all Siamese networks will be designed to contain LSTMs. One thing to remember is that sub-networks share identical parameters. This means that you **only** need to train one set of weights and not two.

The output of each sub-network is a vector. You can then run the output through a cosine similarity function to get the similarity score. In the next video, we will talk about the cost function for such a network.

deeplearning.ai

Loss Function

Loss Function



$$\hat{y} = s(v_1, v_2)$$

Loss Function

$$\cos(v_1, v_2) = \frac{v_1 \cdot v_2}{||v_1||\,||v_2||}$$

$$s(v_1, v_2)$$

How old are you? — Anchor

What is your age? — Positive

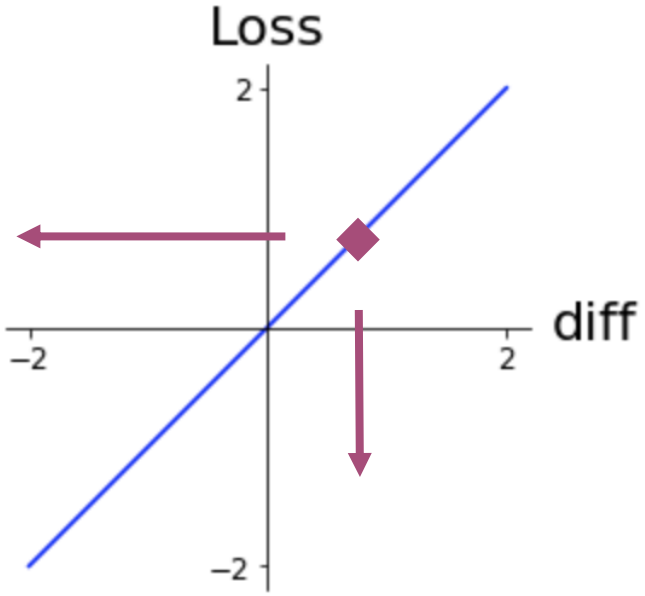
$$s(A, P) \sim 1$$

Where are you from? — Negative

$$s(A, N) \approx -1$$

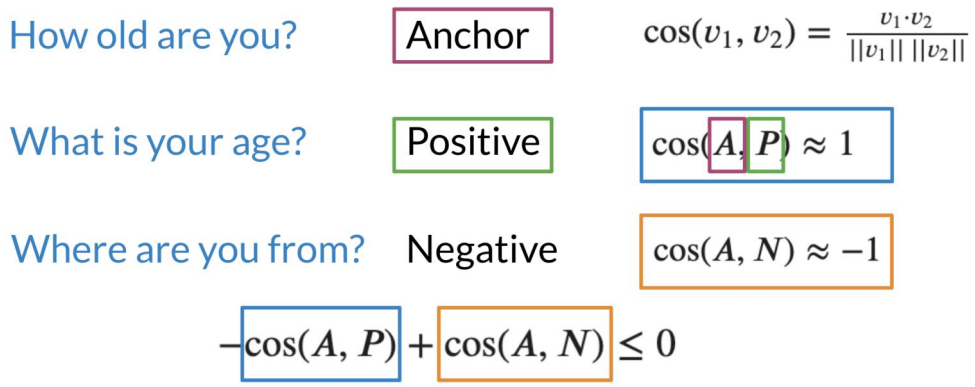
$$s(A, N) - s(A, P)$$

Loss Function



$$\text{diff} = \text{s}(A, N) - \text{s}(A, P)$$

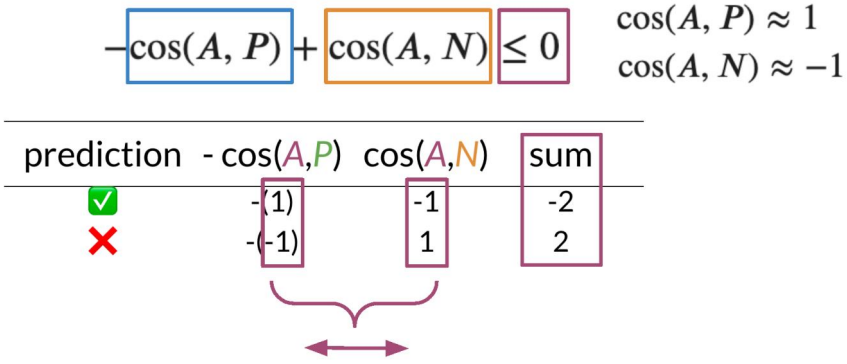
Let us take a close look at the following slide:



Note that when trying to compute the cost for a siamese network we use the triplet loss. The triplet loss usually consists of an Anchor and a Positive example. Note that the anchor and the positive example have a cosine similarity score that is very close to one. On the other hand, the anchor and the negative example have a cosine similarity score close to -1. Now we are ideally trying to optimize the following equation:

$$-\cos(A, P) + \cos(A, N) \leq 0$$

Note that if $\cos(A, P) = 1$ is 1 and $\cos(A, N) = -1$, then the equation is definitely less than 0. However, as cos(A,P) deviates from 1 and cos(A,N) deviates from -1, then you can end up getting a cost that is > 0. Here is a visualization that would help you understand what is going on. Feel free to play with different numbers.



deeplearning.ai

Triplets

Triplets

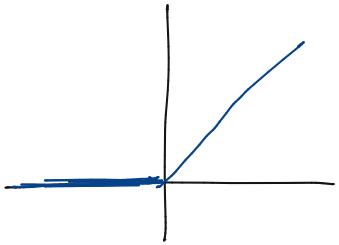
How old are you? — Anchor

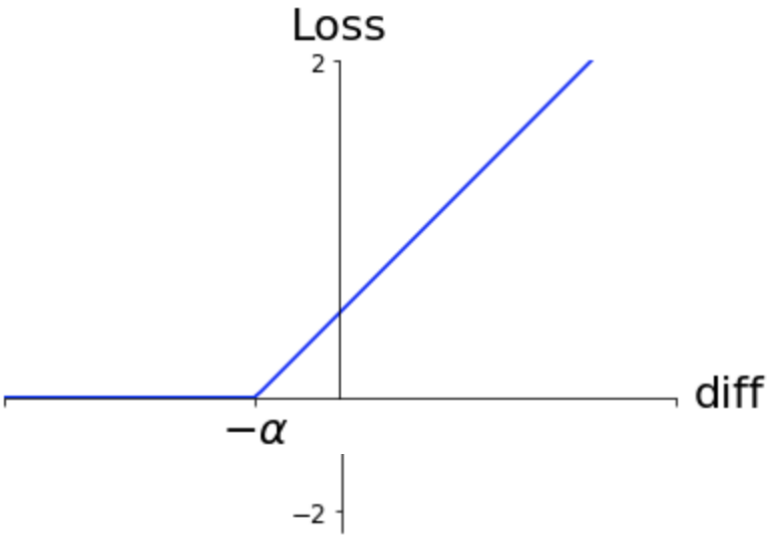
What is your age? — Positive

Where are you from? — Negative

Whether or not a question has the same meaning as the anchor

Triplet Loss





Simple loss:

$$\text{diff} = s(A, N) - s(A, P)$$

With non-linearity

$$\mathcal{L} = \begin{cases} 0; & \text{if } diff \leq 0 \\ diff; & \text{if } diff > 0 \end{cases}$$

With alpha margin

$$\mathcal{L} = \begin{cases} 0; & \text{if } diff + \alpha \leq 0 \\ diff + \alpha; & \text{if } diff + \alpha > 0 \end{cases}$$

hyper Param

Triplet Loss

$$\mathcal{L} = \begin{cases} 0; & \text{if } diff + \alpha \leq 0 \\ diff; & \text{if } diff + \alpha > 0 \end{cases}$$

Simplified

$$\mathcal{L}(A, P, N) = \max(diff + \alpha, 0)$$

From the neural network

You can use any similarity function or distance metric

distance metric opposite similarity fun

Triplet Selection

Questions

Triplet A, P, N

- duplicate set: A, P
- non-duplicate set: A, N

$$\mathcal{L} = \max(diff + \alpha, 0)$$

$$\text{diff} = \text{s}(A, N) - \text{s}(A, P)$$

Random

Easy to satisfy. Little to learn

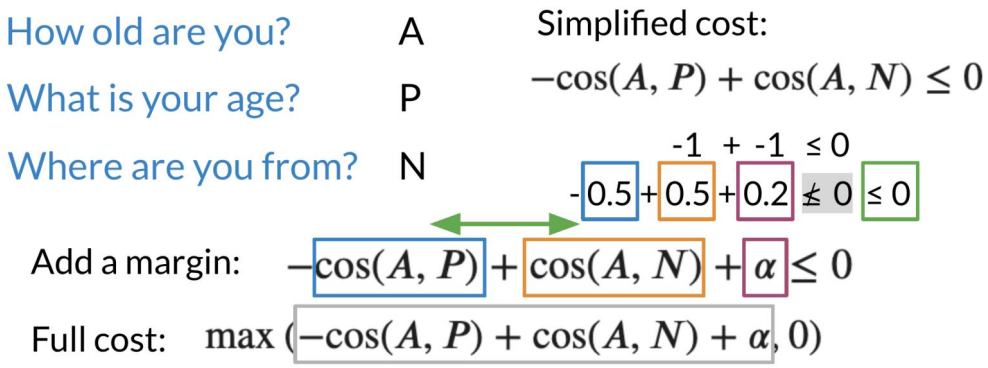
(loss goes to 0)

$$\text{s}(A, N) \approx \text{s}(A, P)$$

Hard

Harder to train. More to learn

We will now build on top of our previous cost function. To get the full cost function you will add a margin.



Note that we added an α in the equation above. This allows you to have a margin of "safety". When computing the full cost, we take the max of that the outcome of $-\cos(A, P) + \cos(A, N) + \alpha$ and 0. Note, we do not want to take a negative number as a cost.

Here is a quick summary:

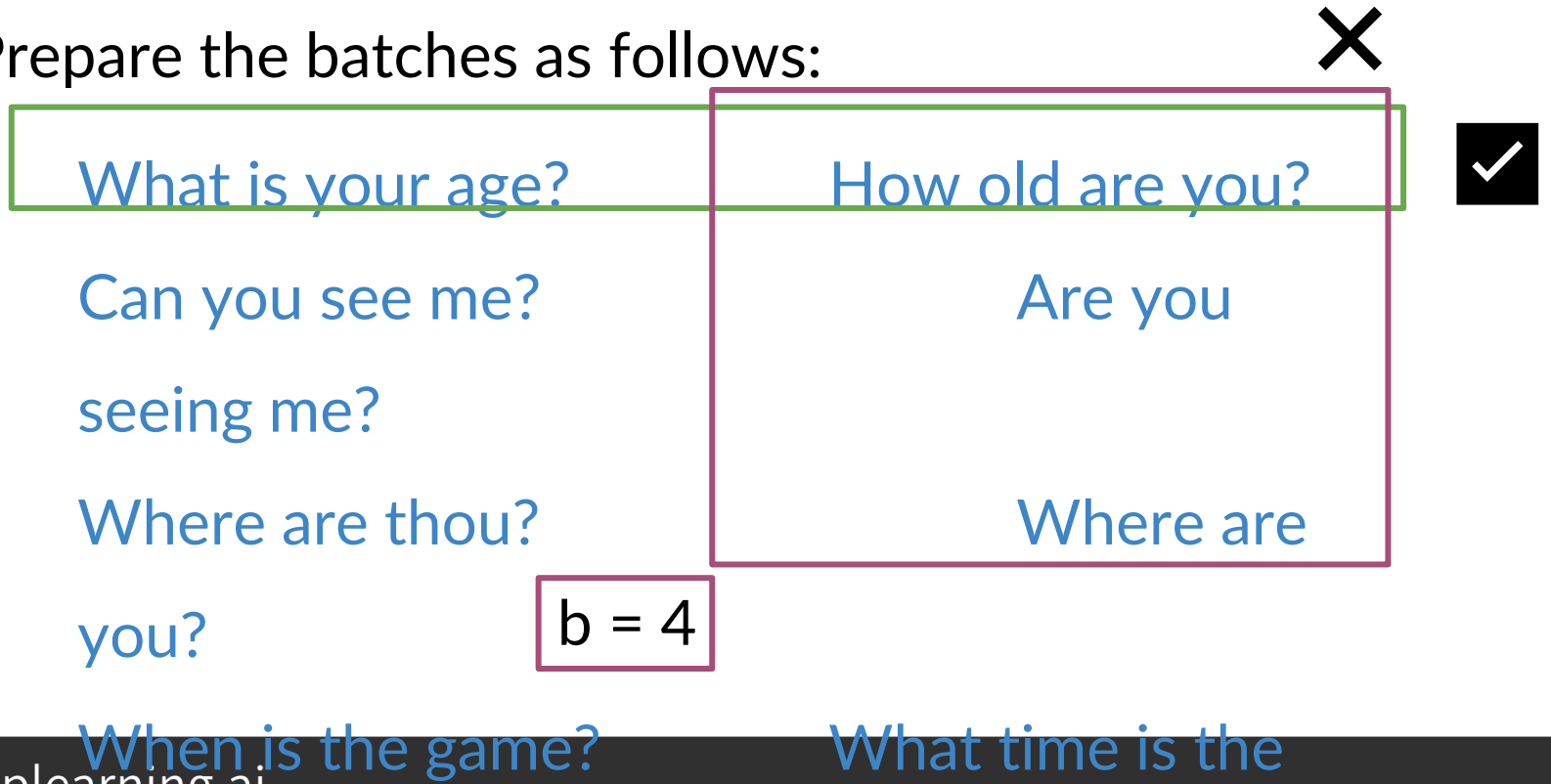
- $\boldsymbol{\alpha}$**:** controls how far cos(A , P) is from cos(A , N)
- **Easy** negative triplet: cos(A , N) < cos(A , P)
- **Semi-hard** negative triplet: cos(A , N) < cos(A , P) < cos(A , N) + $\boldsymbol{\alpha}$
- **Hard** negative triplet: cos(A , P) < cos(A , N)

deeplearning.ai

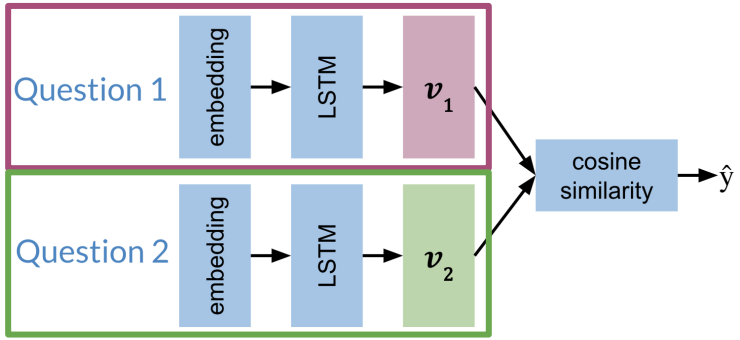
Computing The Cost I

Computing The Cost

Prepare the batches as follows:



Computing The Cost



Batch 1

What is your age?

Can you see me?

Where are thou?

When is the game?

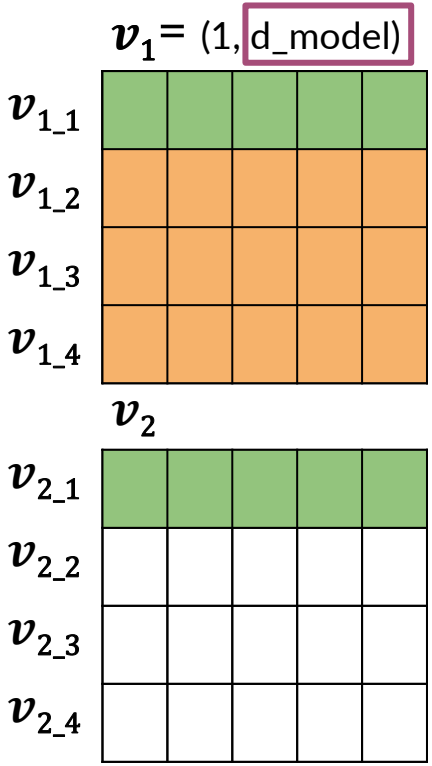
Batch 2

How old are you?

Are you seeing me?

Where are you?

What time is the game?



Computing The Cost

$s(v_1, v_2)$

v_2 \ v_1	_1	_2	_3	_4
_1	0.9	-0.8	0.3	-0.5
_2	-0.8	0.5	0.1	-0.2
_3	0.3	0.1	0.7	-0.8
_4	-0.5	-0.2	-0.8	1.0

Computing The Cost

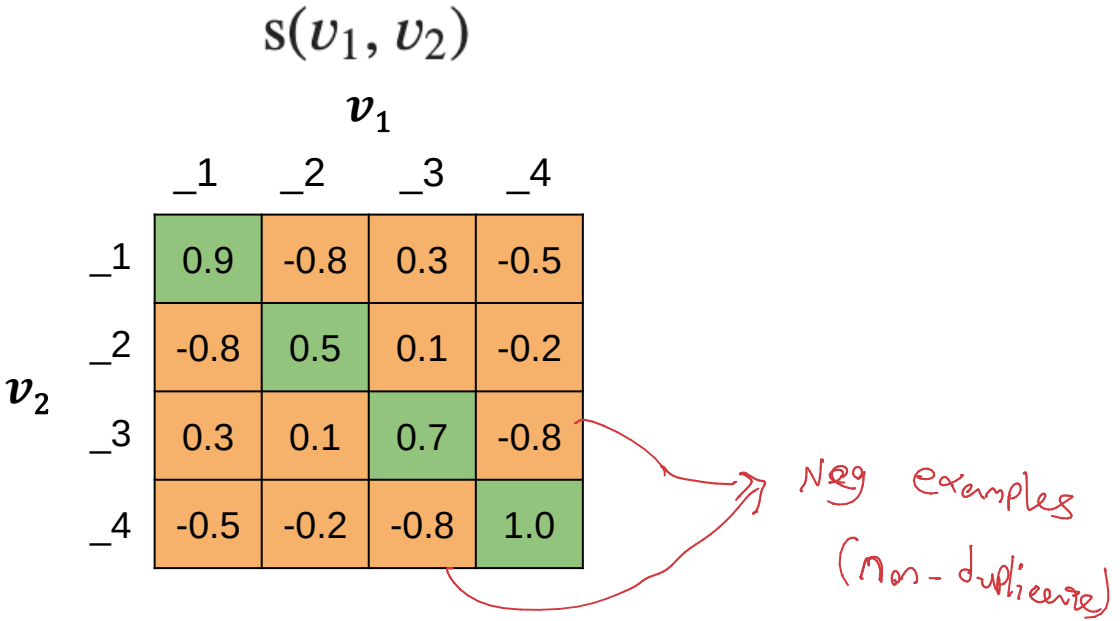
$s(v_1, v_2)$

v_2 \ v_1	_1	_2	_3	_4
_1	0.9	-0.8	0.3	-0.5
_2	-0.8	0.5	0.1	-0.2
_3	0.3	0.1	0.7	-0.8
_4	-0.5	-0.2	-0.8	1.0

similarities

→ Pos example

Computing The Cost



Computing The Cost

$s(v_1, v_2)$

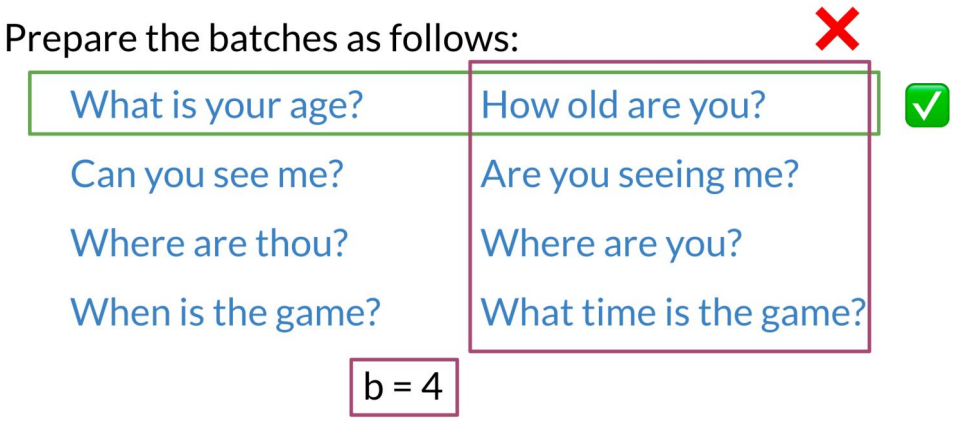
v_2 \ v_1	_1	_2	_3	_4
_1	0.9	-0.8	0.3	-0.5
_2	-0.8	0.5	0.1	-0.2
_3	0.3	0.1	0.7	-0.8
_4	-0.5	-0.2	-0.8	1.0

$$\mathcal{L}(A, P, N) = \max(diff + \alpha, 0)$$

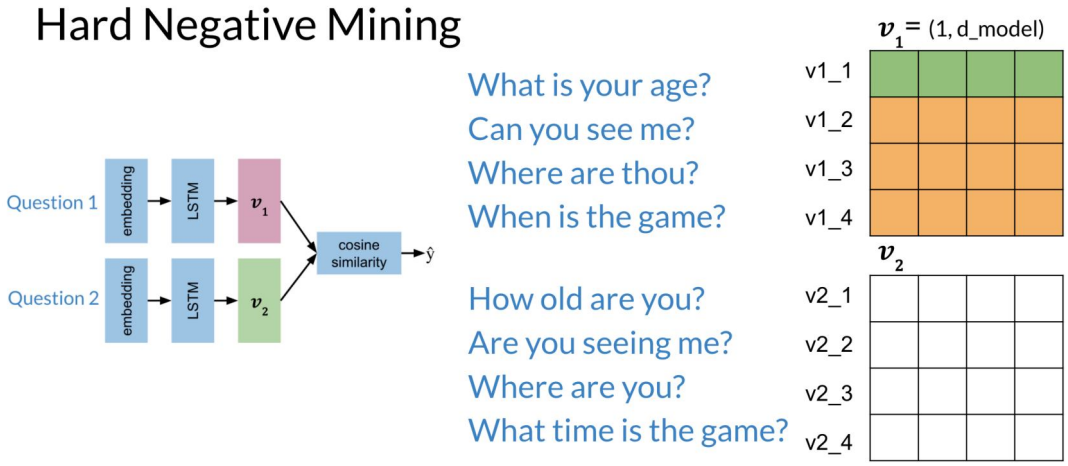
$$\text{diff} = s(A, N) - s(A, P)$$

$$\mathcal{J} = \sum_{i=1}^{m} \mathcal{L}(A^{(i)}, P^{(i)}, N^{(i)})$$

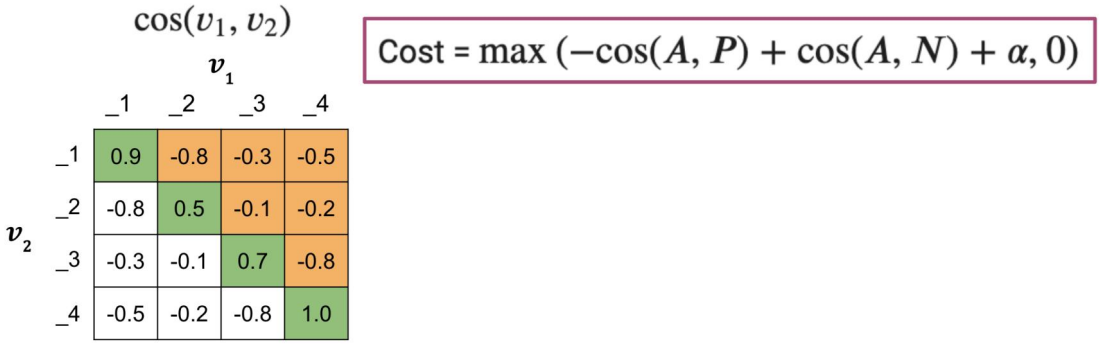
To compute the cost, we will prepare the batches as follows:



Note that each example, has a similar example to its right, but no other example means the same thing. We will now introduce hard negative mining.



Each horizontal vector corresponds to the encoding of the corresponding question. Now when you multiply the two matrices and compute the cosine, you get the following:

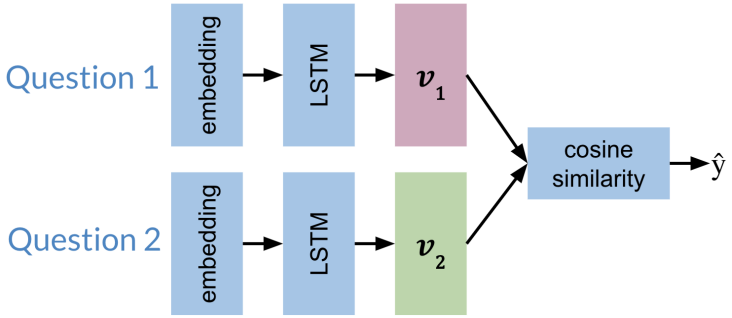


The diagonal line corresponds to scores of similar sentences, (normally they should be positive). The off-diagonals correspond to cosine scores between the anchor and the negative examples.

deeplearning.ai

Computing The Cost II

Computing The Cost



Batch 1

What is your age?

Can you see me?

Where are thou?

When is the game?

Batch 2

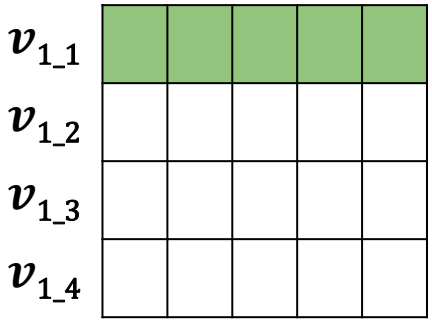
How old are you?

Are you seeing me?

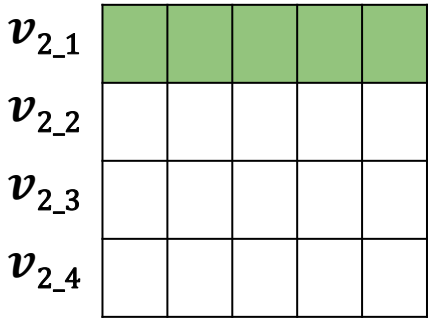
Where are you?

What time is the game?

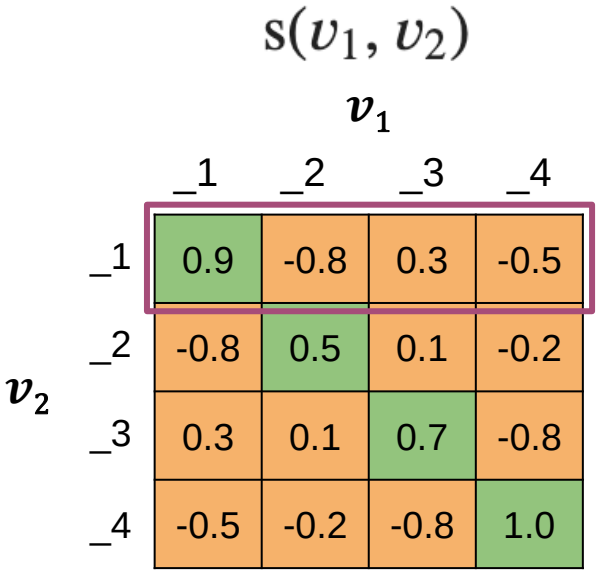
v_1= (1, d_model)



v_2



Hard Negative Mining



mean negative:
mean of off-diagonal values in each row

closest negative:
off-diagonal value closest to (but less than) the value on diagonal in each row

Hard Negative Mining

mean negative: mean of off-diagonal values

closest negative: closest off-diagonal value

$$\mathcal{L}_{\text{Original}} = \max(\underbrace{s(A, N) - s(A, P)}_{\text{diff}} + \alpha, 0)$$

we want diff < 0

$L_1 = \max(\text{mean neg} - s(A,P) + \alpha, 0)$

↳ helps the model to converge faster during training by reducing noise

$$L_2 = \max(\text{closest neg} - S(A,P) + \alpha, 0)$$

(, help create larger Penalty by diminishing the effects of the otherwise more negative $S(A,N)$ that it replaces (smallest diff)

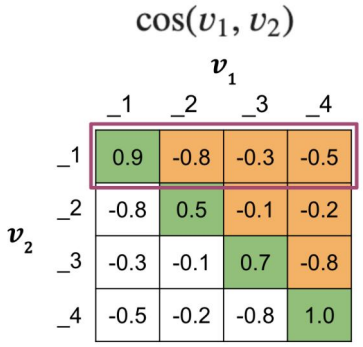
$$L_{FWM} = L_1 + L_2$$

Hard Negative Mining

$$\mathcal{L}_{\text{Full}}(A, P, N) = \mathcal{L}_1 + \mathcal{L}_2$$

$$\mathcal{J} = \sum_{i=1}^{m} \mathcal{L}_{\text{Full}}(A^{(i)}, P^{(i)}, N^{(i)})$$

Now that you have the matrix with cosine similarity scores, which is the product of two matrices, we go ahead and compute the cost.



mean_neg: speeds up training

closest_neg: helps penalize the cost more

We now introduce two concepts, the **mean_neg,** which is the mean negative of all the other off diagonals in the row, and the **closest_neg,** which corresponds to the highest number in the off diagonals.

$$\text{Cost} = \max(-\cos(A, P) + \cos(A, N) + \alpha, 0)$$

So we will have two costs now:

$$\text{Cost 1} = \max(-\cos(A, P) + mean_neg) + \alpha, 0)$$

$$\text{Cost 2} = \max(-\cos(A, P) + closest_neg + \alpha, 0)$$

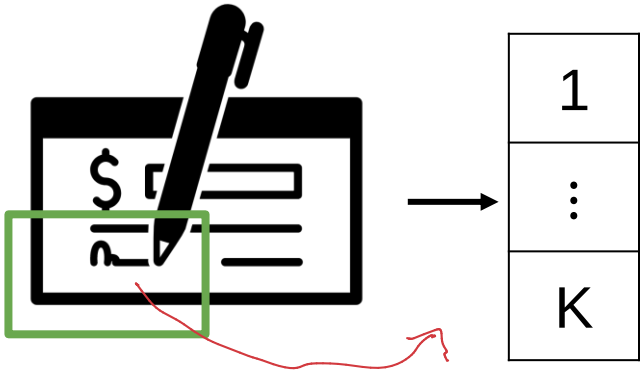
The full cost is defined as: Cost 1 + Cost 2.

deeplearning.ai

One Shot Learning

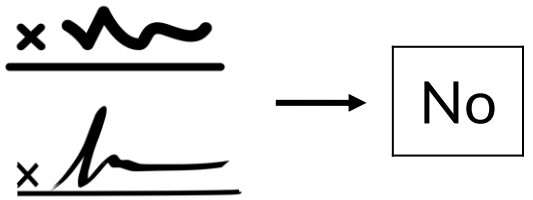
Classification vs One Shot Learning

Classification



Classify as 1 of K classes

One Shot Learning



Measure similarity between 2 classes

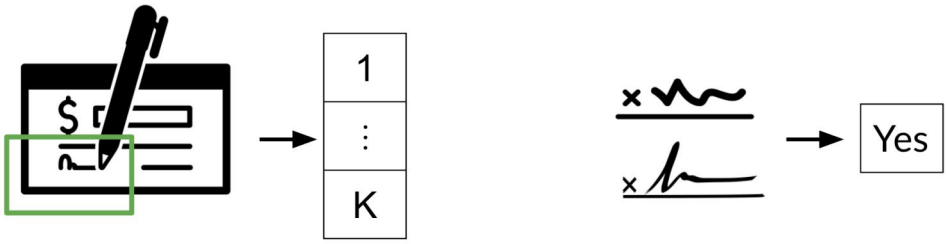
One Shot Learning

No need for retraining !

Learn a similarity score!

$$s(sig1, sig2) > \tau$$ ✓

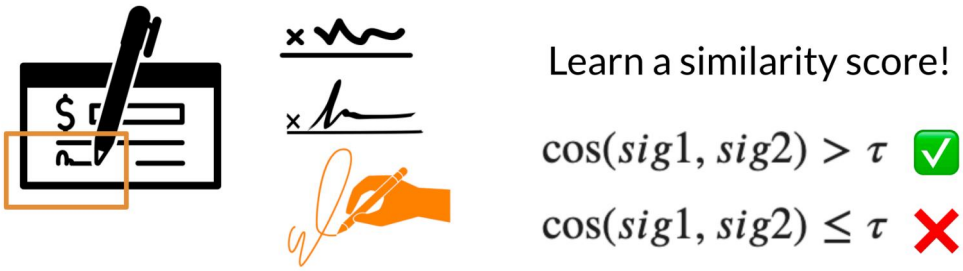
Imagine you are working in a bank and you need to verify the signature of a check. You can either build a classifier with K possible signatures as an output or you can build a classifier that tells you whether two signatures are the same.



Classification: classifies input as 1 of K classes

One Shot Learning: measures similarity between 2 classes

Hence, we resort to one shot learning. Instead of retraining your model for every signature, you can just learn a similarity score as follows:



deeplearning.ai

Training / Testing

Dataset

Question 1	Question 2	is_duplicate
What is your age?	How old are you?	true
Where are you from?	Where are you going?	false
⋮	⋮	⋮

Prepare Batches

Question 1:
batch size b

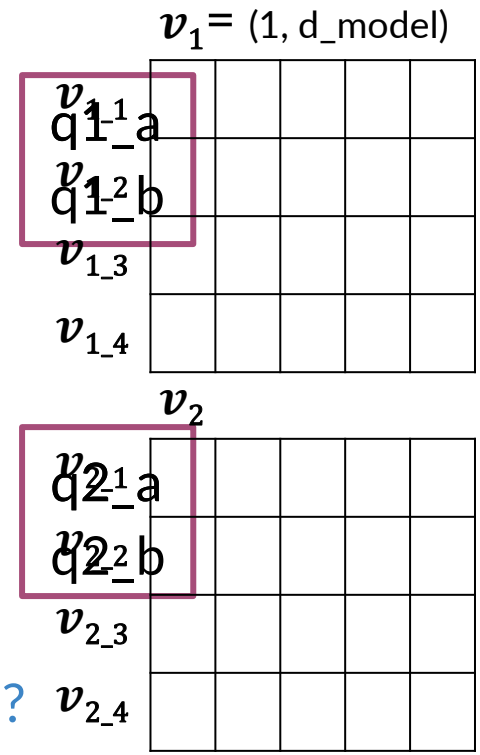
Batch 1

- What is your age?
- Can you see me?
- Where are thou?
- When is the game?

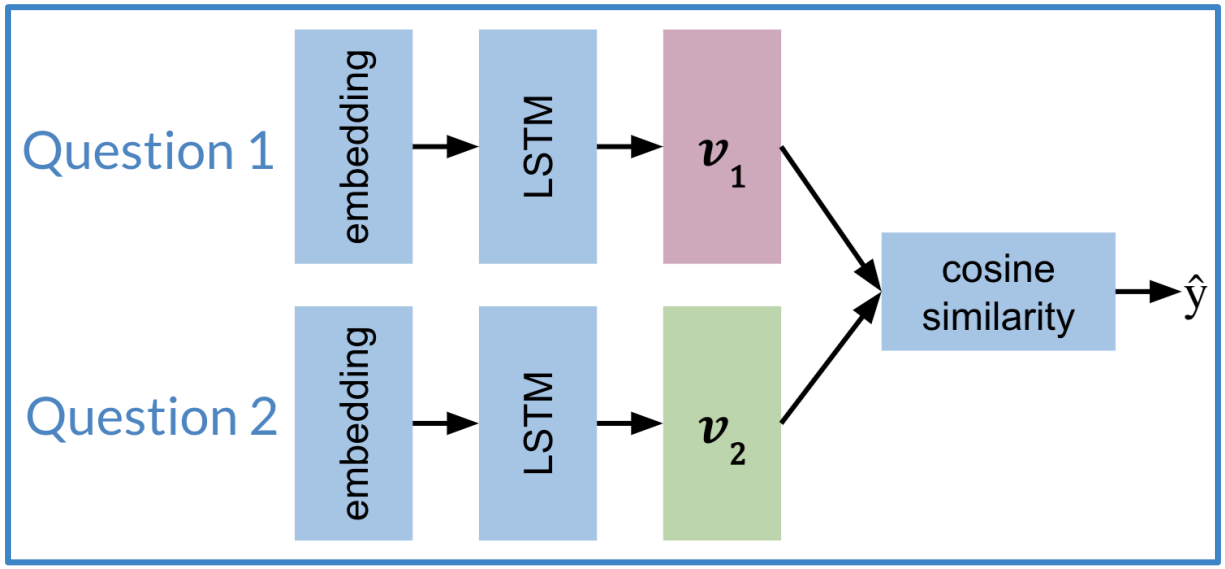
Question 2:
batch size b

Batch 2

- How old are you?
- Are you seeing me?
- Where are you?
- What time is the game?



Siamese Model



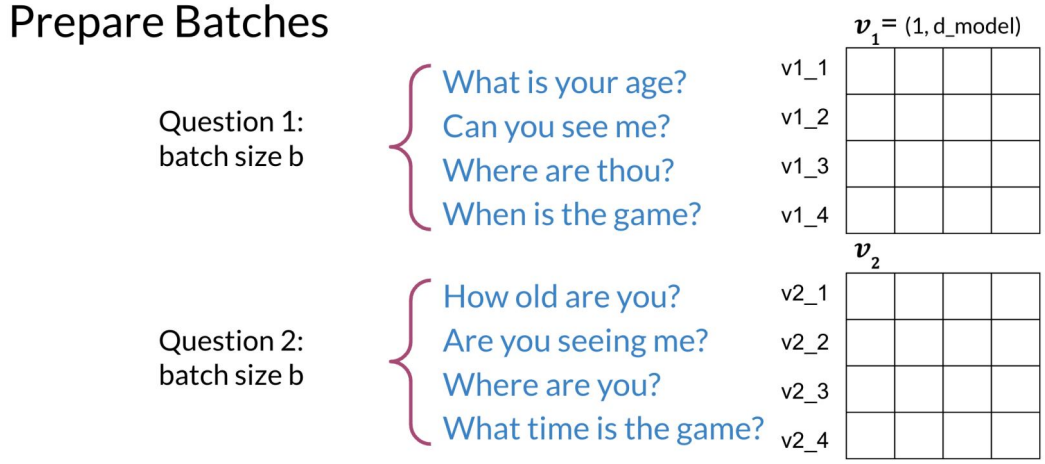
Create a subnetwork:

1) Embedding
2) LSTM
3) Vectors
4) Cosine Similarity

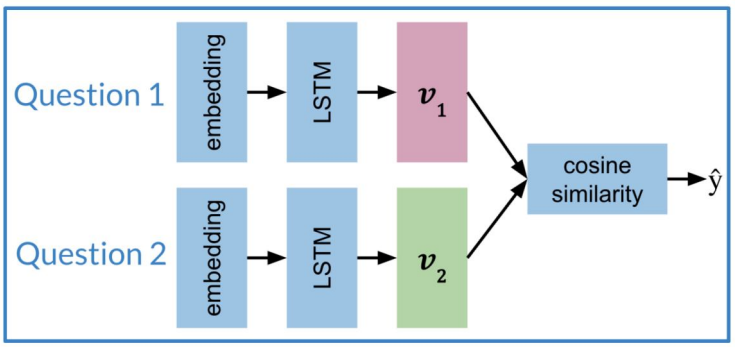
Testing

1. Convert each input into an array of numbers
2. Feed arrays into your model
3. Compare v_1, v_2 using cosine similarity
4. Test against a threshold τ → hyper Param

After preparing the batches of vectors, you can proceed to multiplying the two matrices. Here is a quick recap of the first step:



The next step is to implement the siamese model as follows:



Create a subnetwork:

1) Embedding
2) LSTM
3) Vectors
4) Cosine Similarity

Finally when **testing** :

1. Convert two inputs into an array of numbers
2. Feed it into your model
3. Compare $v1$,$v2$ using cosine similarity
4. Test against a threshold τ