Session 2

Submit Assignment

Due Tuesday by 11:59pm **Points** 500 **Submitting** a website url
Available after Jul 16 at 9:30am

RNNs & LSTMs

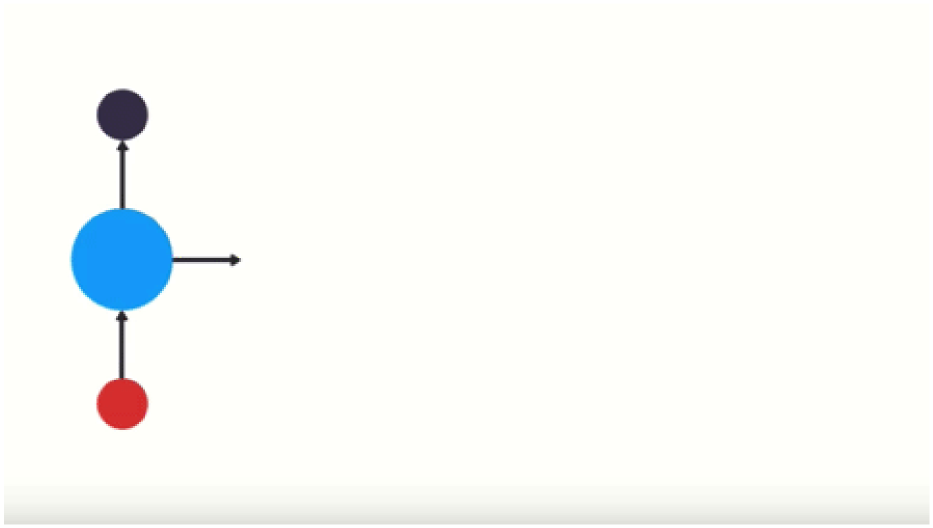
Administrative Stuff:

1. We are going to follow Phase 2 on LMS now.
2. Access emails has been sent to all.
3. You need to submit your assignments to LMS by tomorrow end.

Today we are going to be talking about **Recurrent Neural Networks and LSTMS**.

The source for the content below is shamelessly taken from **this (https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9)** and **this (https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21)** , and modified to suit our understanding. Please make sure that you go through the content below first, and then to the original sources.

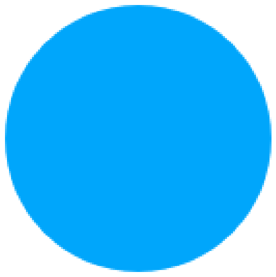
Recurrent Neural Networks



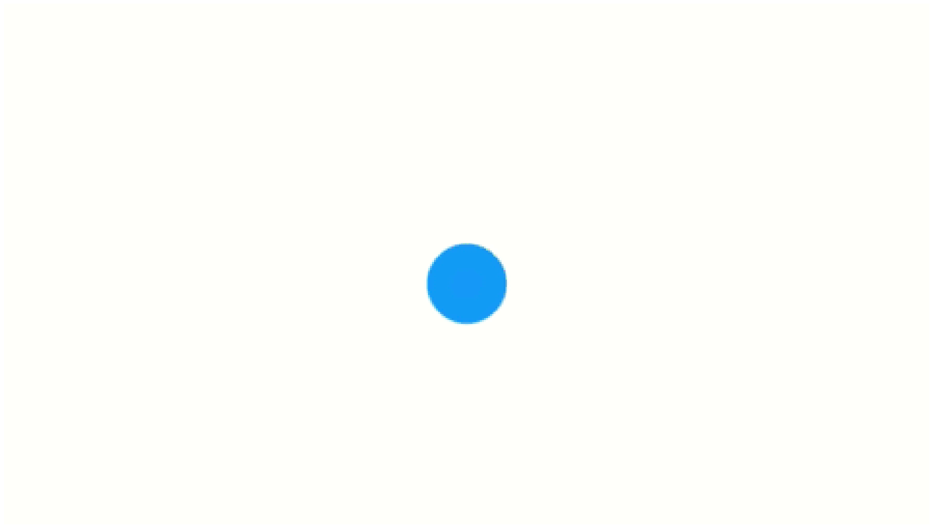
- RNNs are used in speech recognition, language translation, stock prediction, as well as image annotation. It is guaranteed that you have already used it through one of the apps installed on your phone.

SEQUENTIAL DATA

RNNs are neural networks which are great at modeling sequential data. To capture sequential data, we need a sequence (of course), but right now we are only understand how to work with a snapshot. Consider this ball below:



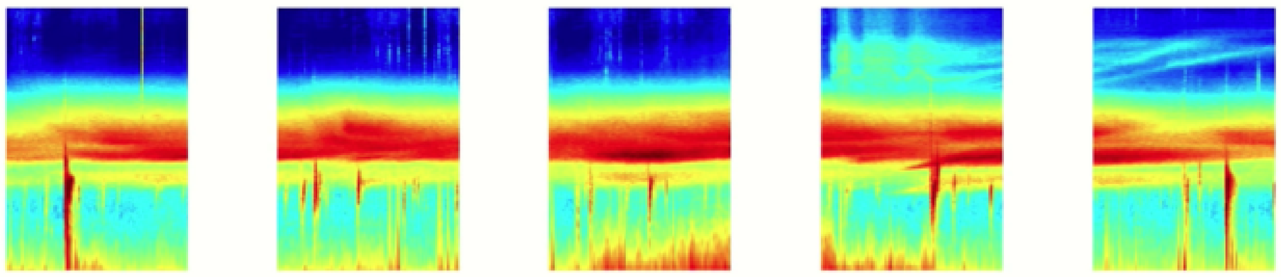
If we want to predict the direction in which this ball is moving we need something more than just this one frame. We would need past few frames to predict the motion.



Now with the above data, as can make our predictions.

Most of the problems we are trying to solve using DNNs actually are a sequence problem. We are converting smartly into a snapshot problem, but we can only go so far.

Take audio for example. There is no way we can look (hear in this case) only at a snapshot (say a 1 frame from 10k Hz audio) and predict what is being said. We however have converted this into a manageable format called spectrogram and then solved it.

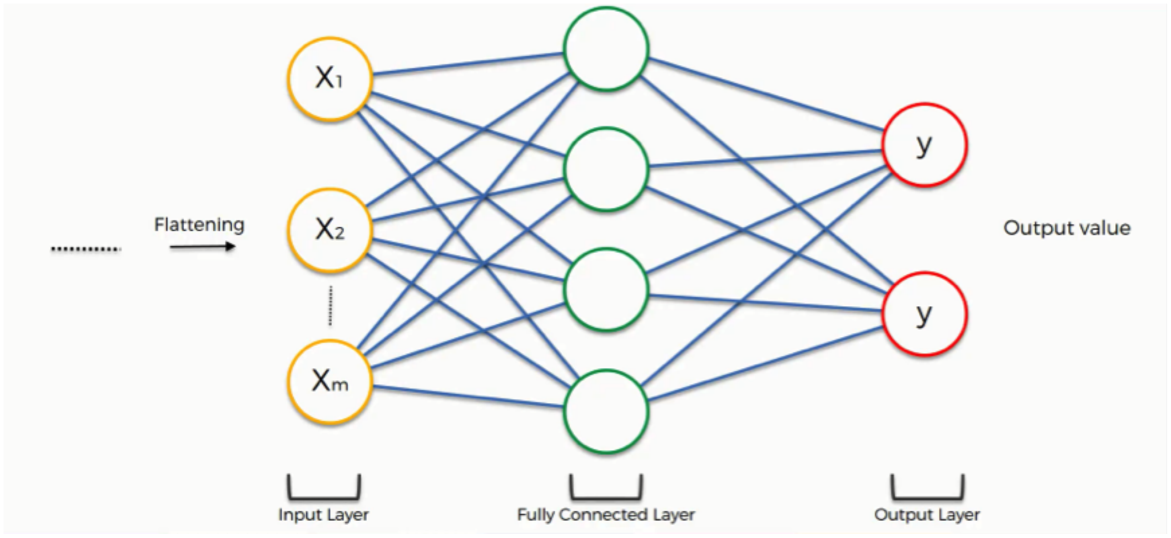


Text is another form of sequence. You can break up text into a sequence of characters or words.

RNNs are good at processing sequence data from predictions. Question is how?

Before we understand RNNs, we need to go back to the basics and re-look at fully connected layers.

FULLY CONNECTED LAYERS

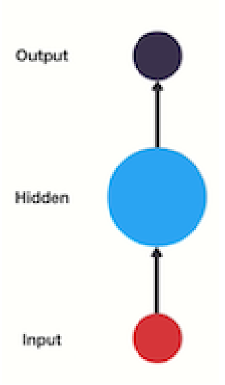


When we covered FCs we bad-mouthed it because it was destroying the 2D nature of our images. But FCs are really great while working with 1D data. We can look at the image above and realize that based on the set of input **X**s coming in, we can learn to predict set of **Y**s. This is great, because now we can send in a sequence to this FC one by one and allow it to predict things accordingly.

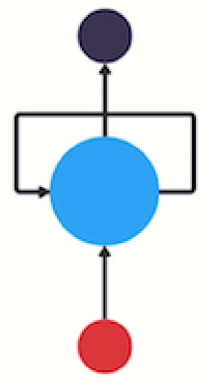
Understanding this concept is important, else we will not be able to understand what exactly is going on **inside** an RNN cell.

RNN CELL

Let's look at a traditional neural network (not CNN, FCs). It has its input layer (a snapshot), hidden layer (FC) and the output layer.



What if we add a loop in the neural network that can pass prior information forward?

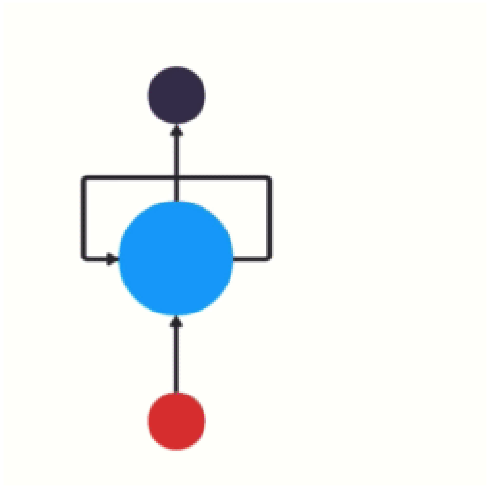


That is essentially what an RNN does. An RNN has a looping mechanism that acts as a highway to allow information to flow from one step to another.

Let's dig deeper and look at that FC we spoke about.

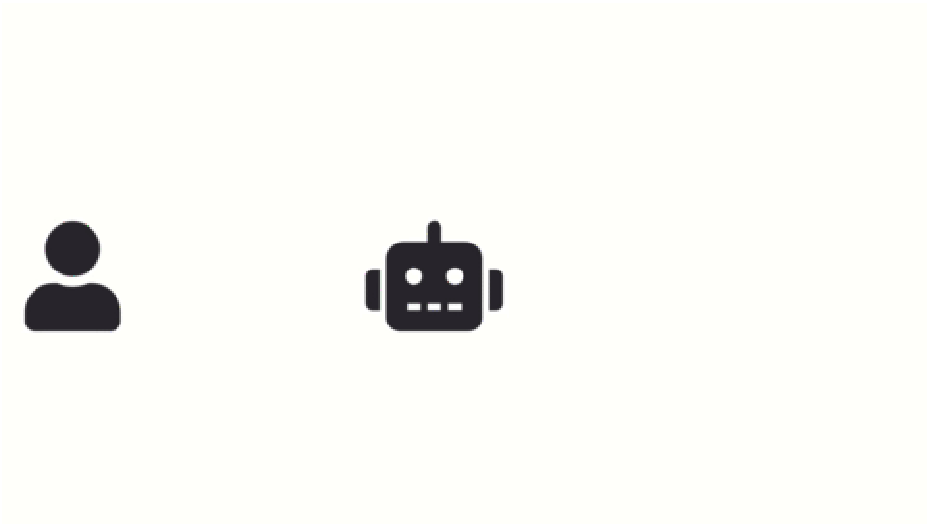
Let us assume (for the original traditional network) that our one input had 100 units as the input (1D) information, and the output had 10 units. Our FC layer must have now 100x10 weights.

In the re-purposed RNN, we are feeding the last output of the FC to itself. Past output had 10 units. This means the input to the RNN is 100 + 10 units. In RNN, our FC must have 110x10 weights. This is important to understand. We will initialize these 10 units for the first time randomly.



This 10 additional units we added is the representation of the previous stage (and no one stops us from saying that we will add 100 units and not 10, it would be a different kind of RNNs then).

Let us say we want to build a chatbot which is tasked with classifying the intent of the user's input text.



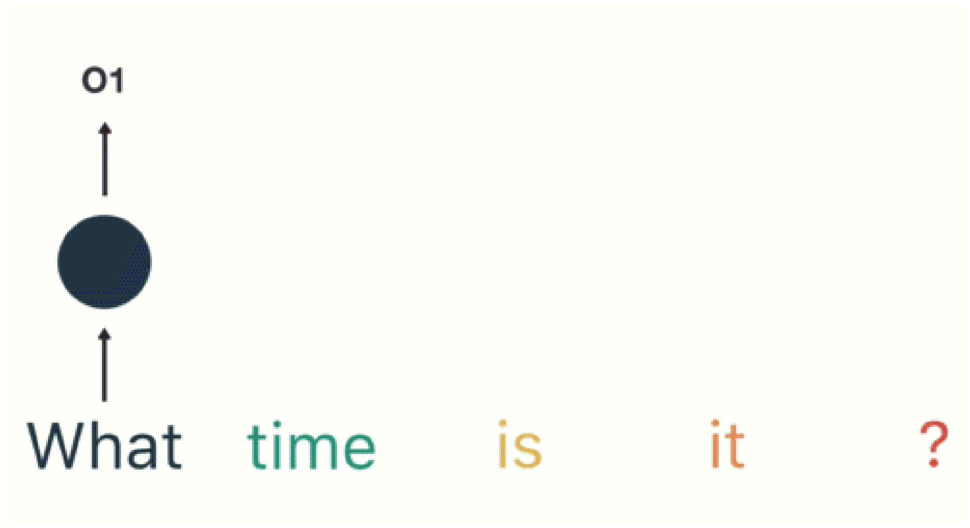
To tackle this problem. First, we are going to encode the sequence of text using an RNN. Then, we are going to feed the RNN output into a feed-forward neural network which will classify the intents.

Ok, so a user types in… ***what time is it?***. To start, we break up the sentence into individual words. RNN's work sequentially so we feed it one word at a time.

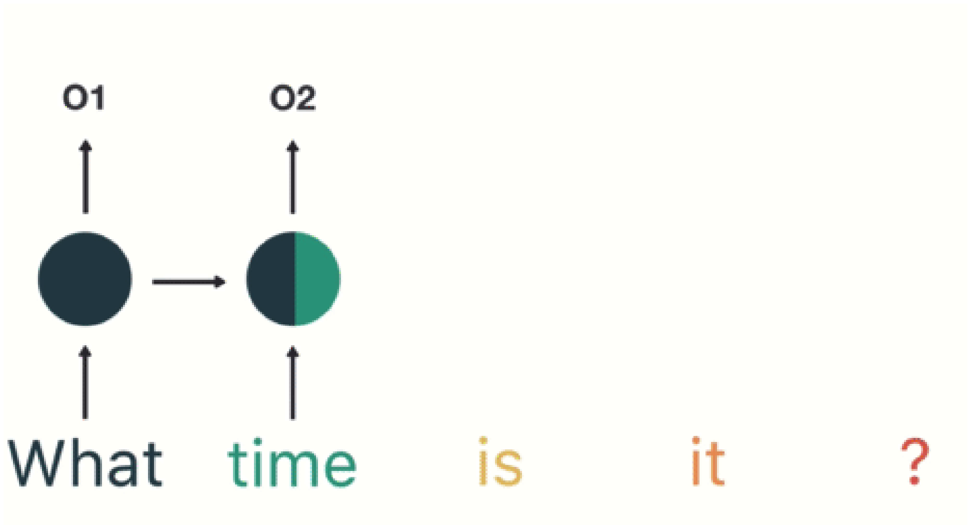
The first step is to feed "What" into the RNN. The RNN encodes "What" and produces an output.

What
time
is
it
?

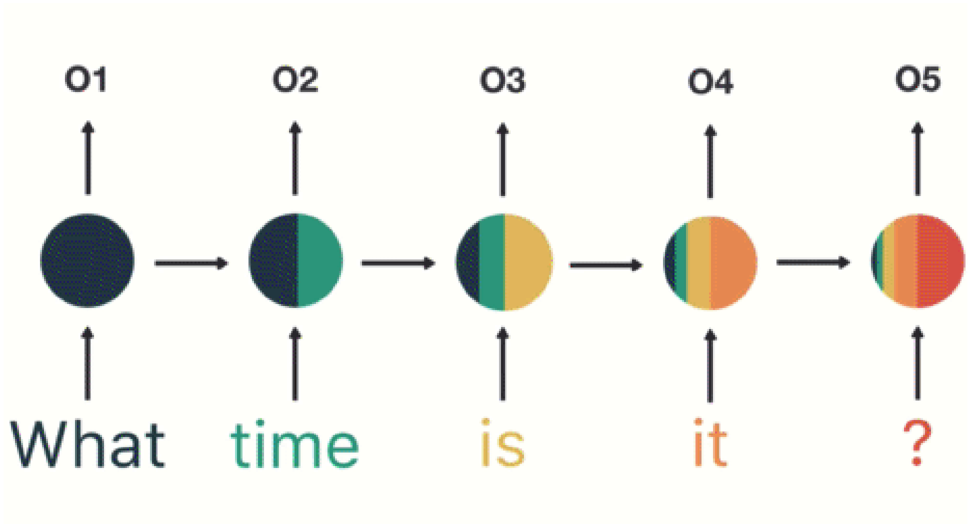
For the next step, we feed the word “time” and the hidden state from the previous step. The RNN now has information on both the word “What” and “time.”



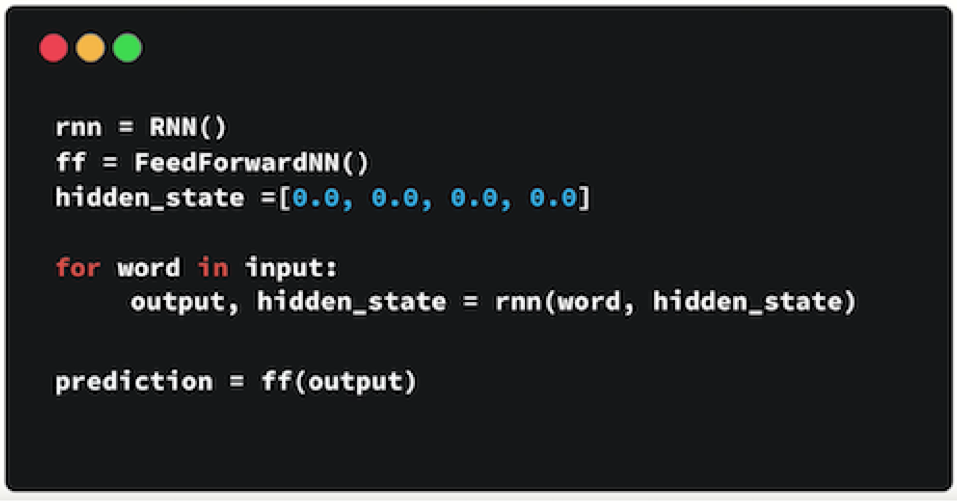
We repeat this process, until the final step. You can see by the final step the RNN has encoded information from all the words in previous steps.



Since the final output was created from the rest of the sequence, we should be able to take the final output and pass it to the feed-forward layer to classify an intent.



For those of you who like looking at code here is some python showcasing the control flow



```
rnn = RNN()
ff = FeedForwardNN()
hidden_state =[0.0, 0.0, 0.0, 0.0]

for word in input:
     output, hidden_state = rnn(word, hidden_state)

prediction = ff(output)
```

First, you initialize your network layers and the initial hidden state. The shape and dimension of the hidden state will be dependent on the shape and dimension of your recurrent neural network. Then you loop through your inputs, pass the word and hidden state into the RNN. The RNN returns the output and a modified hidden state. You continue to loop until you're out of words. Last you pass the output to the feed-forward layer, and it returns a prediction. And that's it! The control flow of doing a forward pass of a recurrent neural network is a for loop.

VANISHING GRADIENTS

You may have noticed the odd distribution of colors in the hidden states. That is to illustrate an issue with RNN's known as short-term memory.

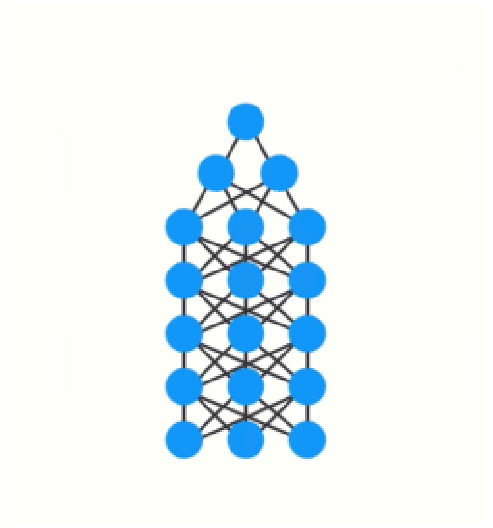
Short-term memory is caused by the infamous vanishing gradient problem, which is also prevalent in other neural network architectures. As the RNN processes more steps, it has troubles retaining information from previous steps. As you can see, the information from the word "what" and "time" is almost non-existent at the final time step. Short-Term memory and the vanishing gradient is due to the nature of back-propagation; an algorithm used to train and optimize neural networks. To understand why this is, let's take a look at the effects of back propagation on a deep feed-forward neural network.

Training a neural network has three major steps.

First, it does a forward pass and makes a prediction.

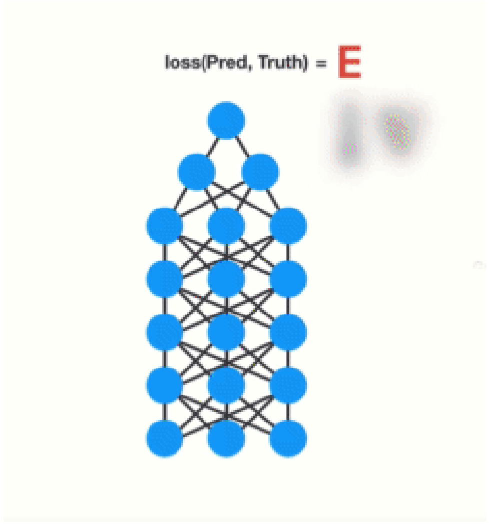
Second, it compares the prediction to the ground truth using a loss function. The loss function outputs an error value which is an estimate of how poorly the network is performing.

Last, it uses that error value to do back propagation which calculates the gradients for each node in the network.

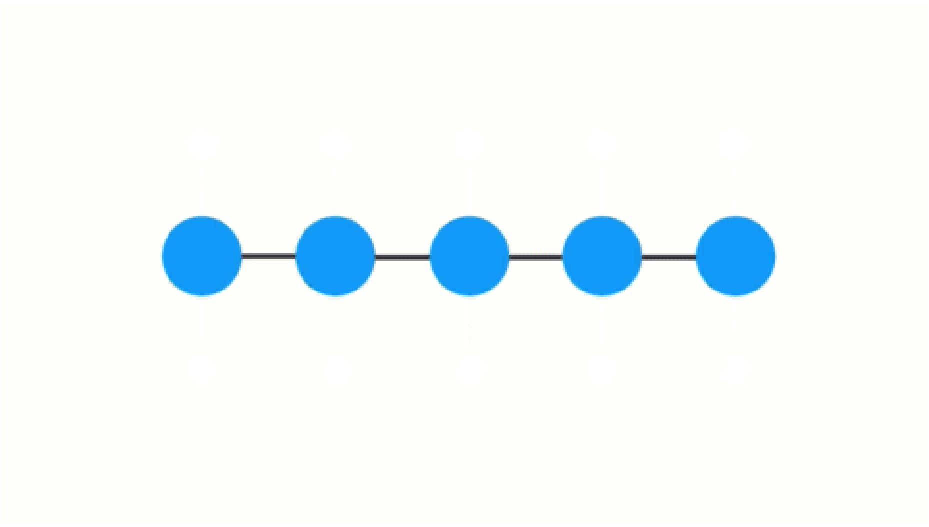


The gradient is the value used to adjust the networks internal weights, allowing the network to learn. The bigger the gradient, the bigger the adjustments and vice versa. Here is where the problem lies. When doing back propagation, each node in a layer calculates it's gradient with respect to the effects of the gradients, in the layer before it. So if the adjustments to the layers before it is small **(why would it be small?)**, then adjustments to the current layer will be even smaller.

That causes gradients to exponentially shrink as it back propagates down. The earlier layers fail to do any learning as the internal weights are barely being adjusted due to extremely small gradients. And that's the vanishing gradient problem.



Let's see how this applies to recurrent neural networks. You can think of each time step in a recurrent neural network as a layer. To train a recurrent neural network, you use an application of back-propagation called back-propagation through time. The gradient values will exponentially shrink as it propagates through each time step.



Again, the gradient is used to make adjustments in the neural networks weights thus allowing it to learn. Small gradients mean small adjustments. That causes the early layers not to learn.

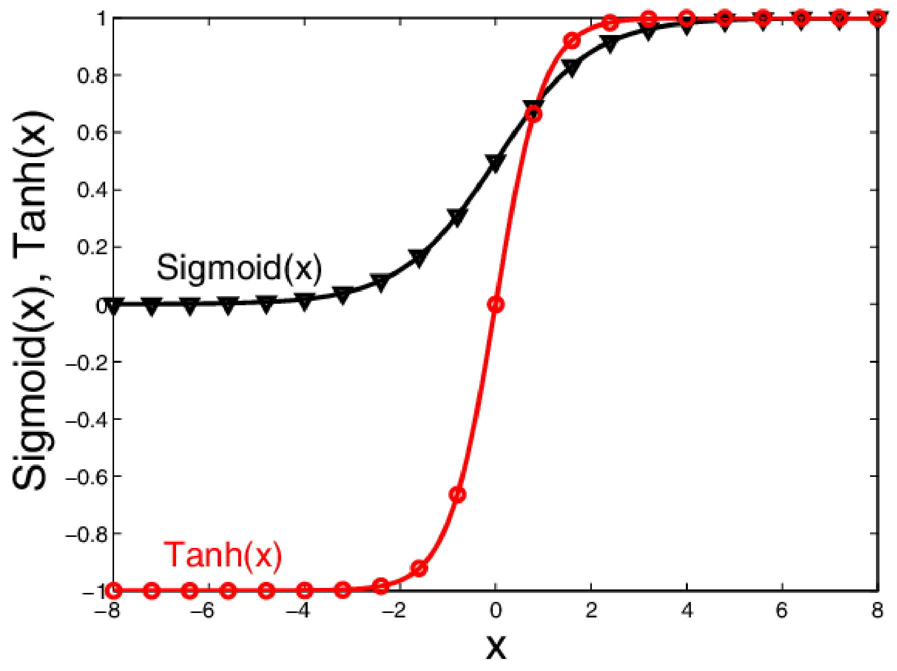
Because of vanishing gradients, the RNN doesn't learn the long-range dependencies across time steps. That means that there is a possibility that the word "what" and "time" are not considered when trying to predict the user's intention. The network then has to make the best guess with "is it?". That's pretty ambiguous and would be difficult even for a human. So not being able to learn on earlier time steps causes the network to have a short-term memory.

Ok so RNN's suffer from short-term memory, so how do we combat that? To mitigate short-term memory, two specialized recurrent neural networks were created. One called Long Short-Term Memory or LSTM's for short. The other is Gated Recurrent Units or GRU's. LSTM's and GRU's essentially function just like RNN's, but they're capable of learning long-term dependencies using mechanisms called "gates." These gates are different tensor operations that can learn what information to add or remove to the hidden state. Because of this ability, short-term memory is less of an issue for them.

LSTM's

Before we jump into the complex world of LSTMs we need to re-look at some of the basics again.

Consider what a Sigmoid and TanH function does:



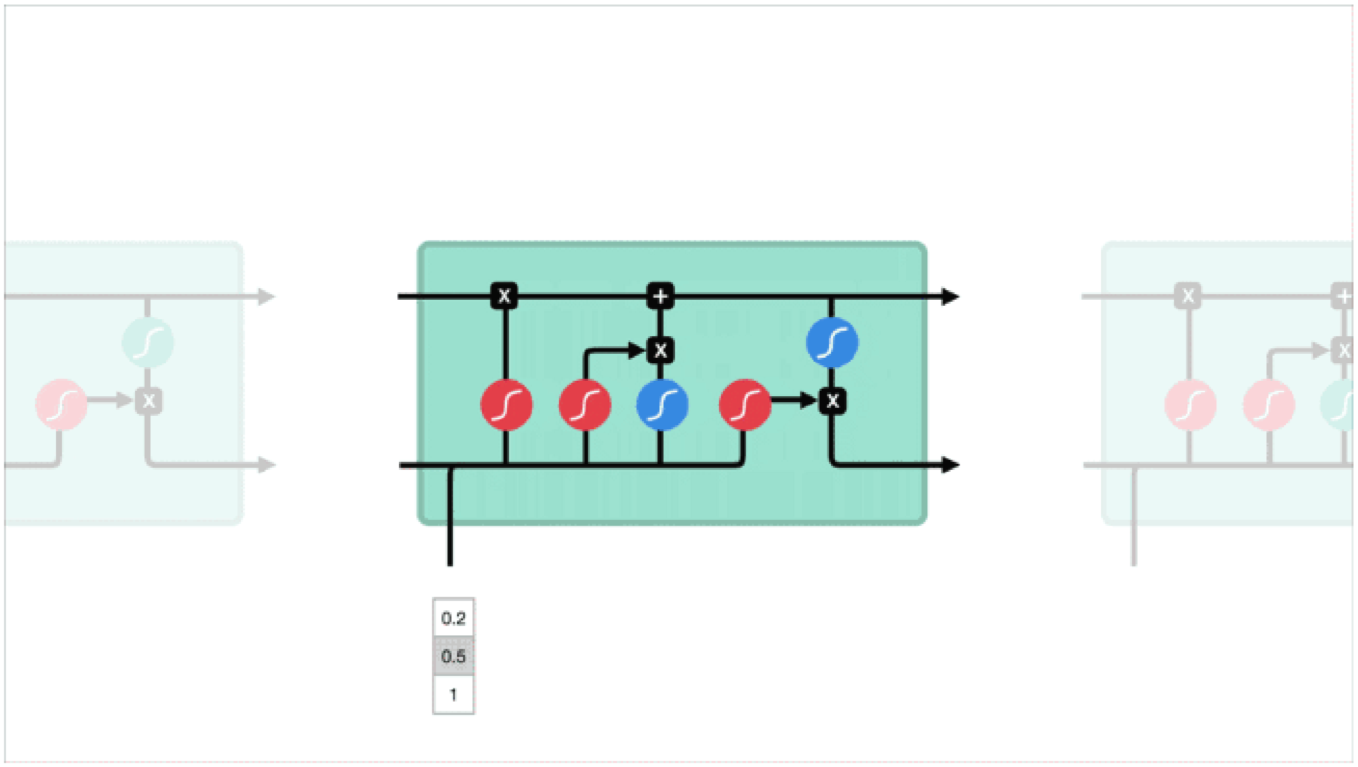
Sigmoid squishes the values between 0 and 1, while **tanh** squishes the values between -1 and 1.

Sigmoid function has the capability to convert things into 0 or 1. This function thus, can be used to **remember** or **forget** things. Imagine a set of 100 numbers. Sigmoid function (with help of hidden weights) can convert this set of 100 numbers into 0s and 1s, thus remember or forgetting the past state. This is a very important concept to understand.

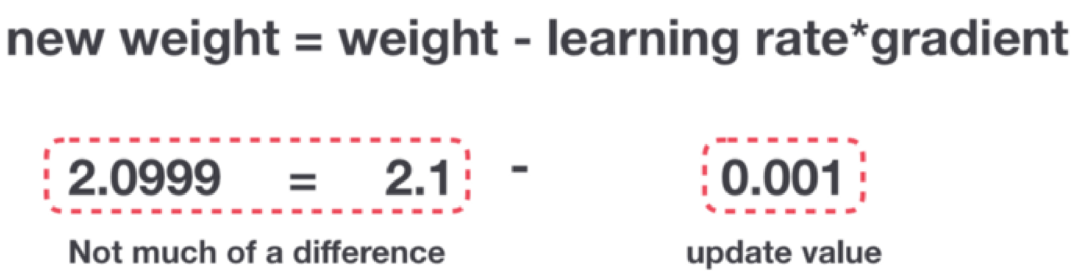
Tanh function has the capability to act like a counter (with the help of hidden weights). We can use it as **adding** or **subtracting** values from a set of 100 numbers.

We are going to use a complex mix of these function to remember, forget, add and subtract numbers, and that in turn will help us make a better sequence modeler.

LSTMs



During back propagation, recurrent neural networks suffer from the vanishing gradient problem. Gradients are values used to update a neural networks weights. The vanishing gradient problem is when the gradient shrinks as it back propagates through time. If a gradient value becomes extremely small, it doesn't contribute too much learning.



So in recurrent neural networks, layers that get a small gradient update stops learning. Those are usually the earlier layers. So because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory.

LSTM's **as a solution**

LSTM 's and GRU's were created as the solution to short-term memory. They have internal mechanisms called gates that can regulate the flow of information.

These gates can learn which data in a sequence is important to keep or throw away. By doing that, it can pass relevant information down the long chain of sequences to make predictions. Almost all state of the art results based on recurrent neural networks are achieved with these two networks. LSTM's and GRU's can be found in speech recognition, speech synthesis, and text generation. You can even use them to generate captions for videos.

INTUITION

The core to the concept of LSTMs (and GRUs) is "memory state". In RNN we were passing the past output to the current cell, rather we should have also passed on "its memory" or "its feeling" or "its internal state which made it make that output". This *internal state* is what makes LSTMs robust to the problems of RNNs.

So we have 2 things at our hand, the last input as well as the internal state (also called cell state).

To make a new decision we must:

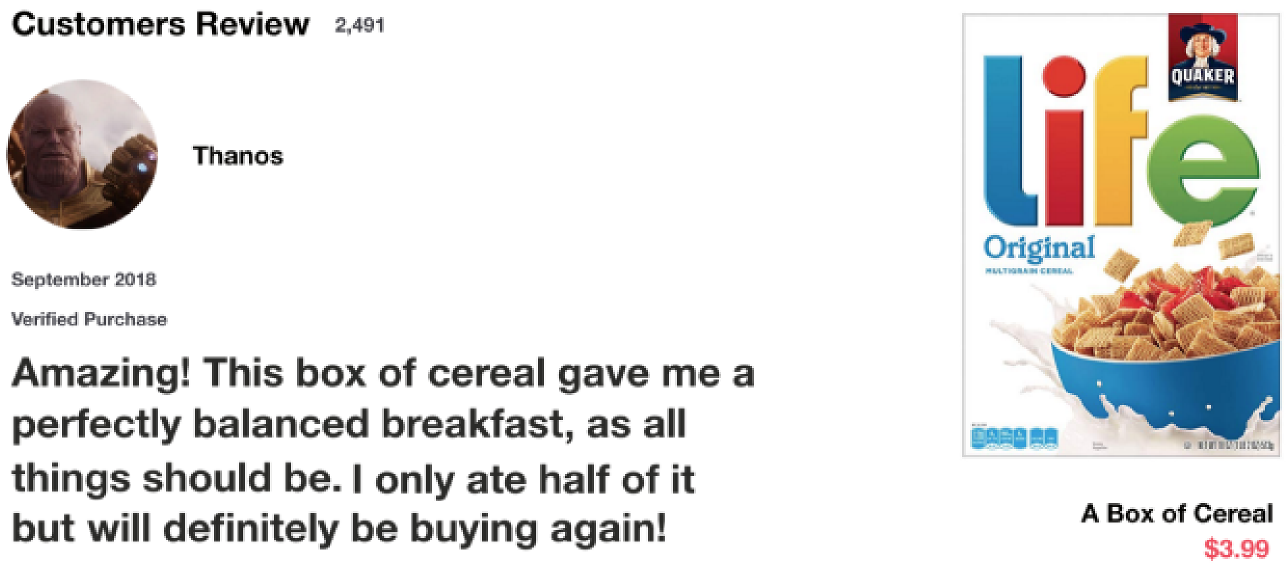
- use the new and old information to change the internal state. We need to forget somethings given the new information and what we predicted (may be the role of some of the units is over or may be we need to trigger something). Now, please make sure that you understand the next line. We can fuse (concatenate) the past output and new input and send it to a FC layer, and send this to a **sigmoid** function to create a new FORGET-REMEMBER vector, and multiply it with the internal state. This will make those units 0 whose role is over, and save those which are still required.
- now we need to add new information to cell state. We can again use our last output and new input, concate them, send it to a new FC layer, then to a tan function and get the information we need to add. We still need to make sure that remove the information (which we already decided in the last step to not add again). For this step we again send our LO and NI to another FC and then to Sigmoid function to get forget vector. Why do we need to use a new FC here, may be we need to refresh a block we wanted to forget with a new update (going from 1 to 0 to -1). We will multiply the values we want to add with the new forget-remember vector and then add this to out cell state above. Our new cell state has a better understand of what is going on right now.
- next, we again take our LO and NI, concatenate them, send it through another sigmoid function to get a new "output" vector. Why a new sigmoid and FC here? Well, we might want to remember something, but

not want to use it right now and so on. We take our cell state, send it to a FC layer to make a prediction vector, multiply it with our "output" vector to get the final output.

If you do not understand these steps above, there is no point reading the material below. Please make sure understand what we did above.

Let's go through all of this again (in different format now)

Ok, Let’s start with a thought experiment. Let’s say you’re looking at reviews online to determine if you want to buy Life cereal (don’t ask me why). You’ll first read the review then determine if someone thought it was good or if it was bad.



Customers Review 2,491

Thanos

September 2018

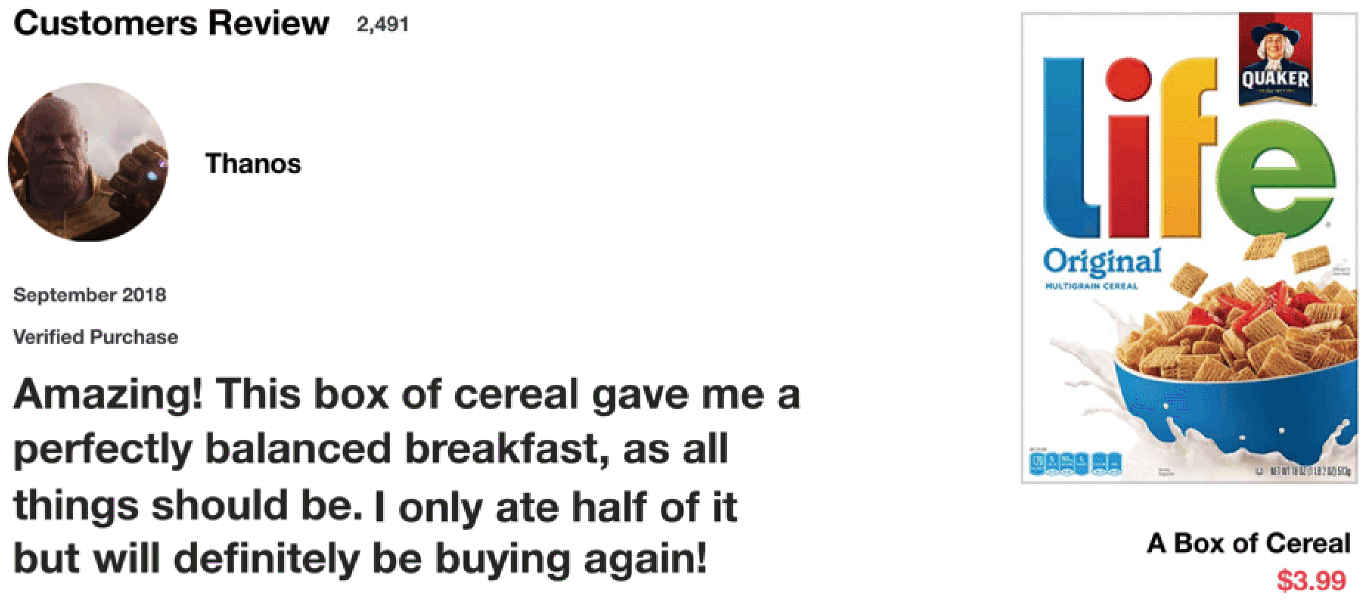
Verified Purchase

Amazing! This box of cereal gave me a perfectly balanced breakfast, as all things should be. I only ate half of it but will definitely be buying again!

A Box of Cereal

$3.99

When you read the review, your brain subconsciously only remembers important keywords. You pick up words like “amazing” and “perfectly balanced breakfast”. You don’t care much for words like “this”, “gave“, “all”, “should”, etc. If a friend asks you the next day what the review said, you probably wouldn’t remember it word for word. You might remember the main points though like “will definitely be buying again”. If you’re a lot like me, the other words will fade away from memory.



Customers Review 2,491

Thanos

September 2018

Verified Purchase

Amazing! This box of cereal gave me a perfectly balanced breakfast, as all things should be. I only ate half of it but will definitely be buying again!

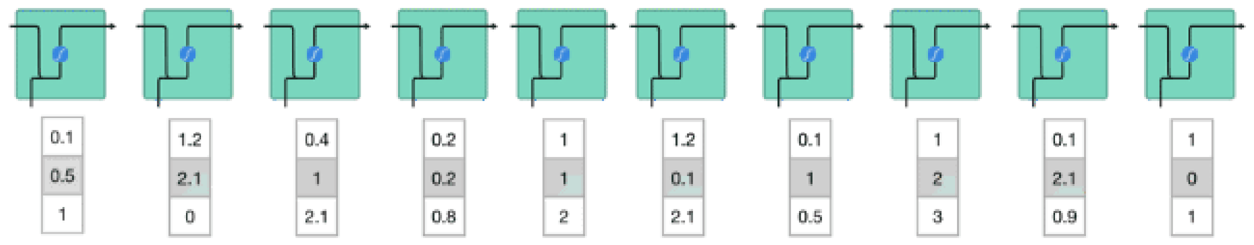
A Box of Cereal

$3.99

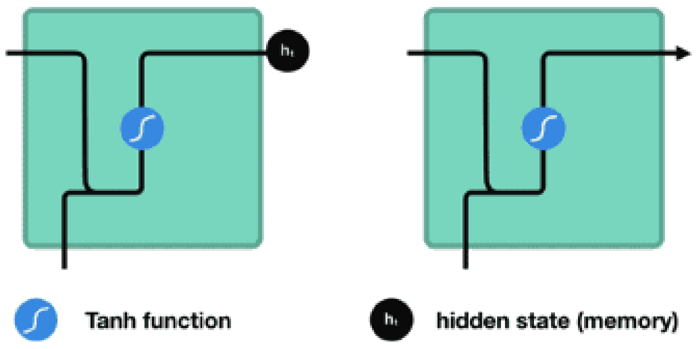
And that is essentially what an LSTM or GRU does. It can learn to keep only relevant information to make predictions, and forget non relevant data. In this case, the words you remembered made you judge that it was good.

Review of Recurrent Neural Networks

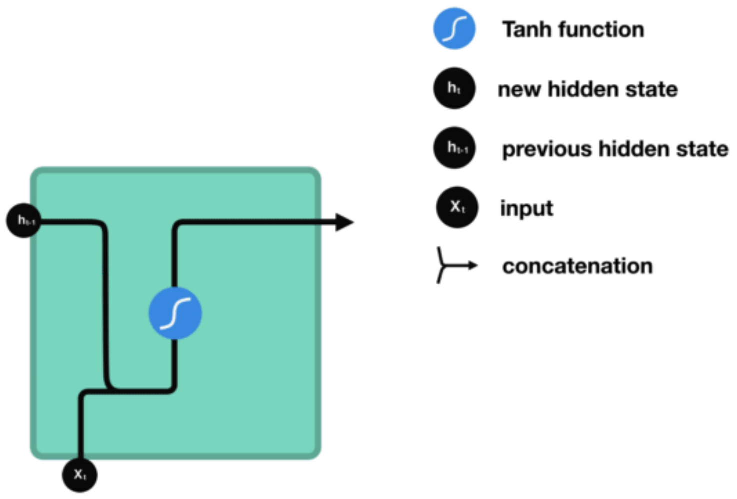
To understand how LSTM's or GRU's achieves this, let's review the recurrent neural network. An RNN works like this; First words get transformed into machine-readable vectors. Then the RNN processes the sequence of vectors one by one.



While processing, it passes the previous hidden state to the next step of the sequence. The hidden state acts as the neural networks memory. It holds information on previous data the network has seen before.

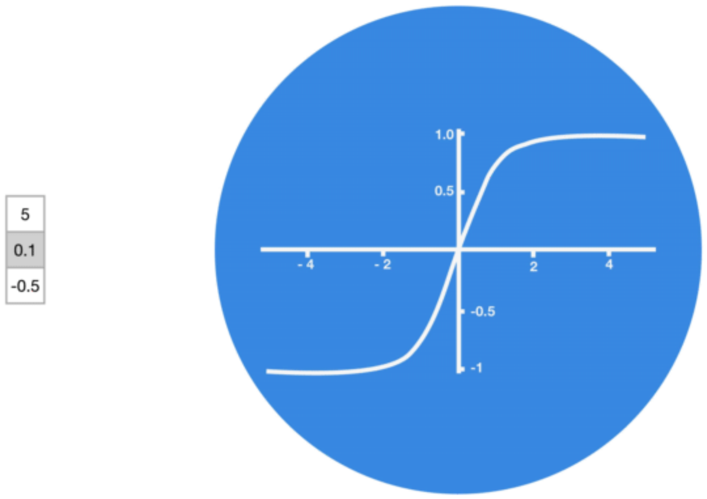


Let's look at a cell of the RNN to see how you would calculate the hidden state. First, the input and previous hidden state are combined to form a vector. That vector now has information on the current input and previous inputs. The vector goes through the tanh activation, and the output is the new hidden state, or the memory of the network.

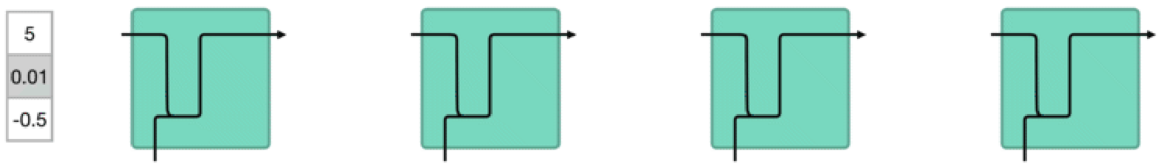


Tanh activation

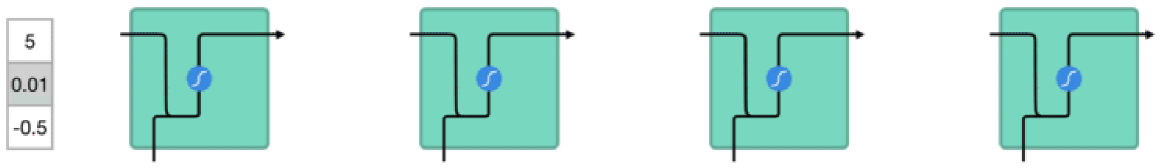
The tanh activation is used to help regulate the values flowing through the network. The tanh function squishes values to always be between -1 and 1.



When vectors are flowing through a neural network, it undergoes many transformations due to various math operations. So imagine a value that continues to be multiplied by let's say ***3***. You can see how some values can explode and become astronomical, causing other values to seem insignificant.



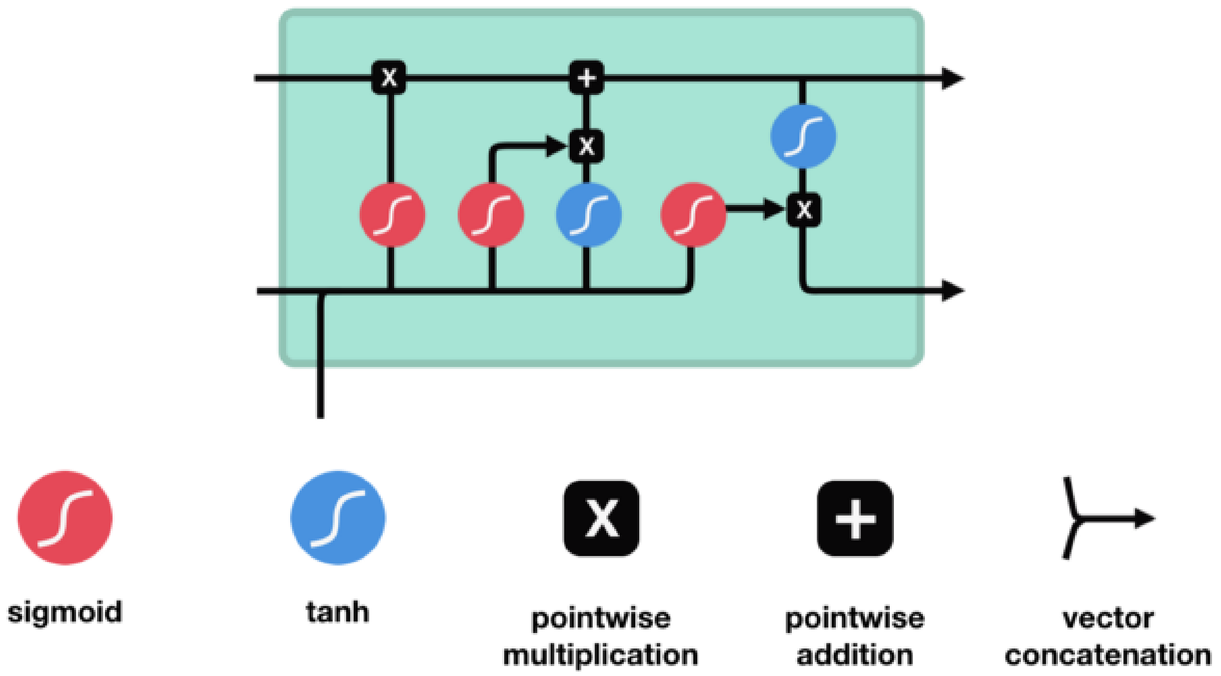
A tanh function ensures that the values stay between -1 and 1, thus regulating the output of the neural network. You can see how the same values from above remain between the boundaries allowed by the tanh function.



So that's an RNN. It has very few operations internally but works pretty well given the right circumstances (like short sequences). RNN's uses a lot less computational resources than it's evolved variants, LSTM's and GRU's.

LSTM

An LSTM has a similar control flow as a recurrent neural network. It processes data passing on information as it propagates forward. The differences are the operations within the LSTM's cells.



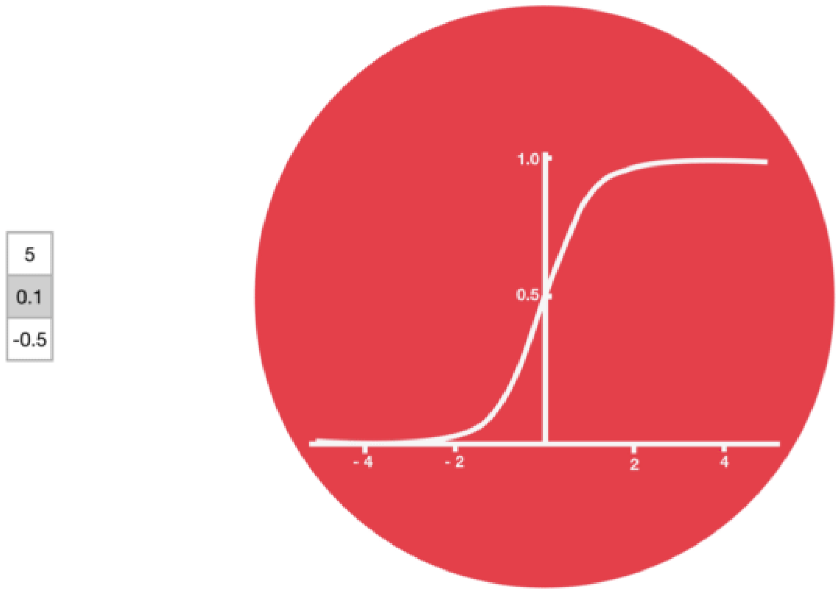
These operations are used to allow the LSTM to keep or forget information. Now looking at these operations can get a little overwhelming so we'll go over this step by step.

Core Concept

The core concept of LSTM's are the cell state, and it's various gates. The cell state act as a transport highway that transfers relative information all the way down the sequence chain. You can think of it as the "memory" of the network. The cell state, in theory, can carry relevant information throughout the processing of the sequence. So even information from the earlier time steps can make it's way to later time steps, reducing the effects of short-term memory. As the cell state goes on its journey, information get's added or removed to the cell state via gates. The gates are different neural networks that decide which information is allowed on the cell state. The gates can learn what information is relevant to keep or forget during training.

Sigmoid

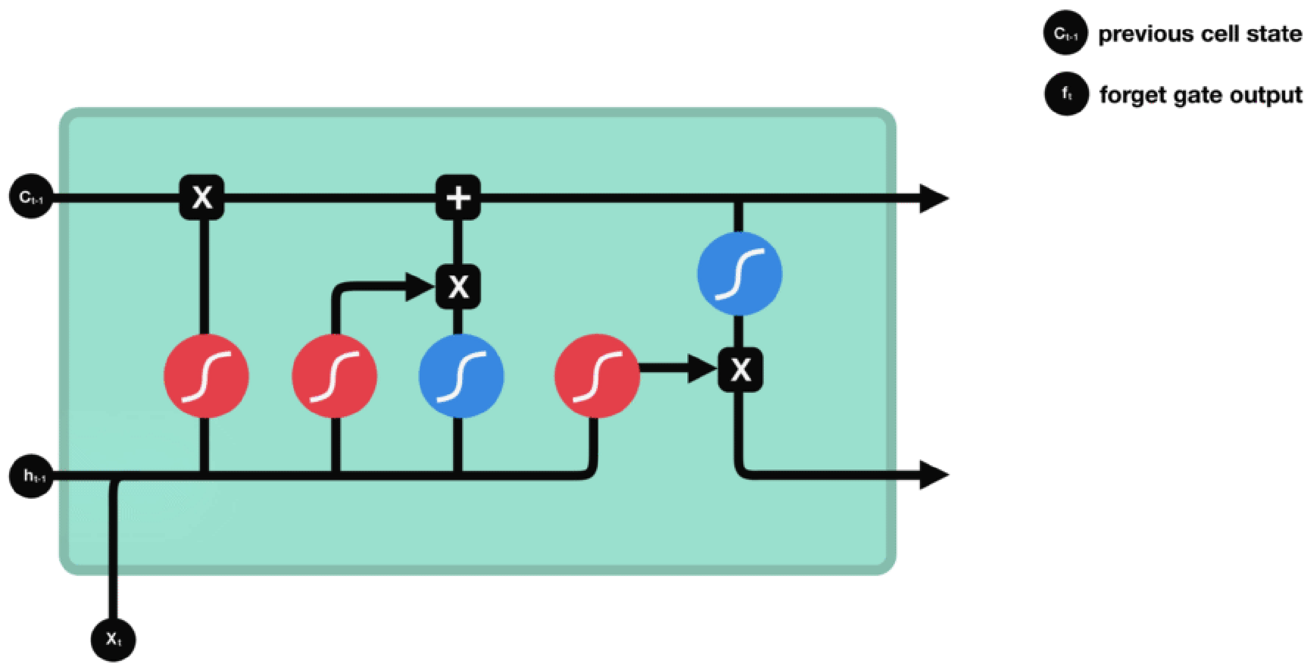
Gates contains sigmoid activations. A sigmoid activation is similar to the tanh activation. Instead of squishing values between -1 and 1, it squishes values between 0 and 1. That is helpful to update or forget data because any number getting multiplied by 0 is 0, causing values to disappears or be "forgotten." Any number multiplied by 1 is the same value therefore that value stay's the same or is "kept." The network can learn which data is not important therefore can be forgotten or which data is important to keep.



Let’s dig a little deeper into what the various gates are doing, shall we? So we have three different gates that regulate information flow in an LSTM cell. A forget gate, input gate, and output gate.

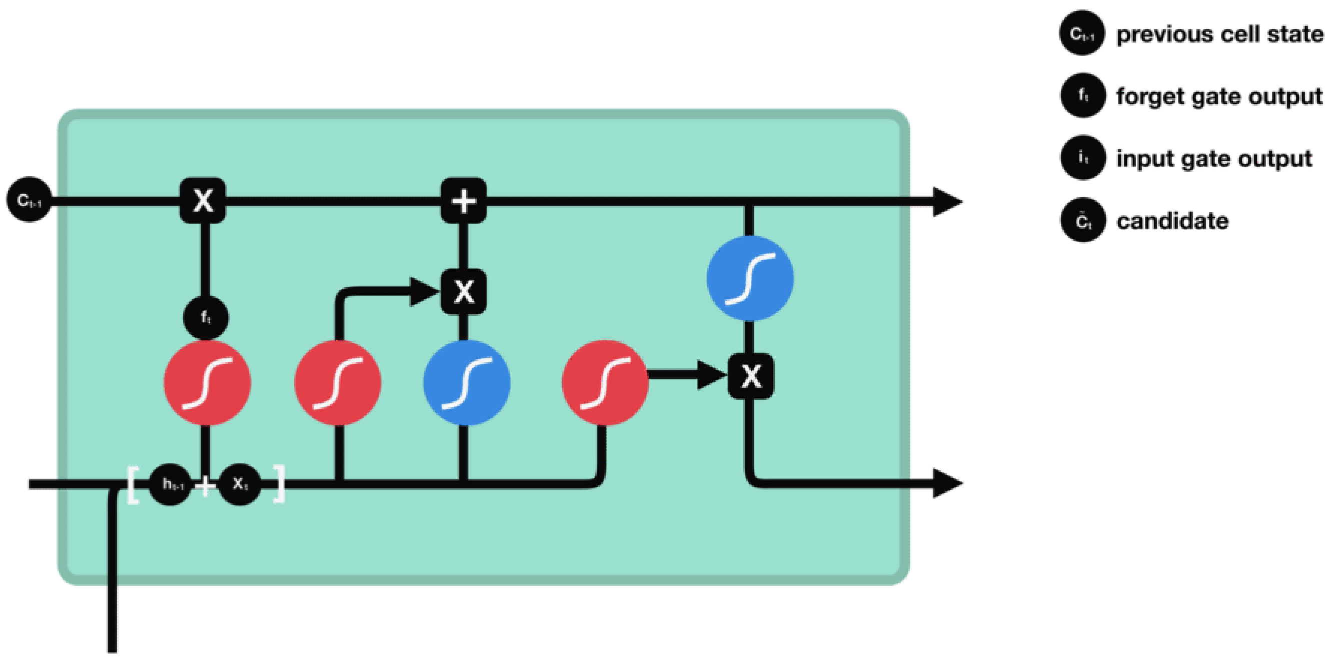
Forget gate

First, we have the forget gate. This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep.



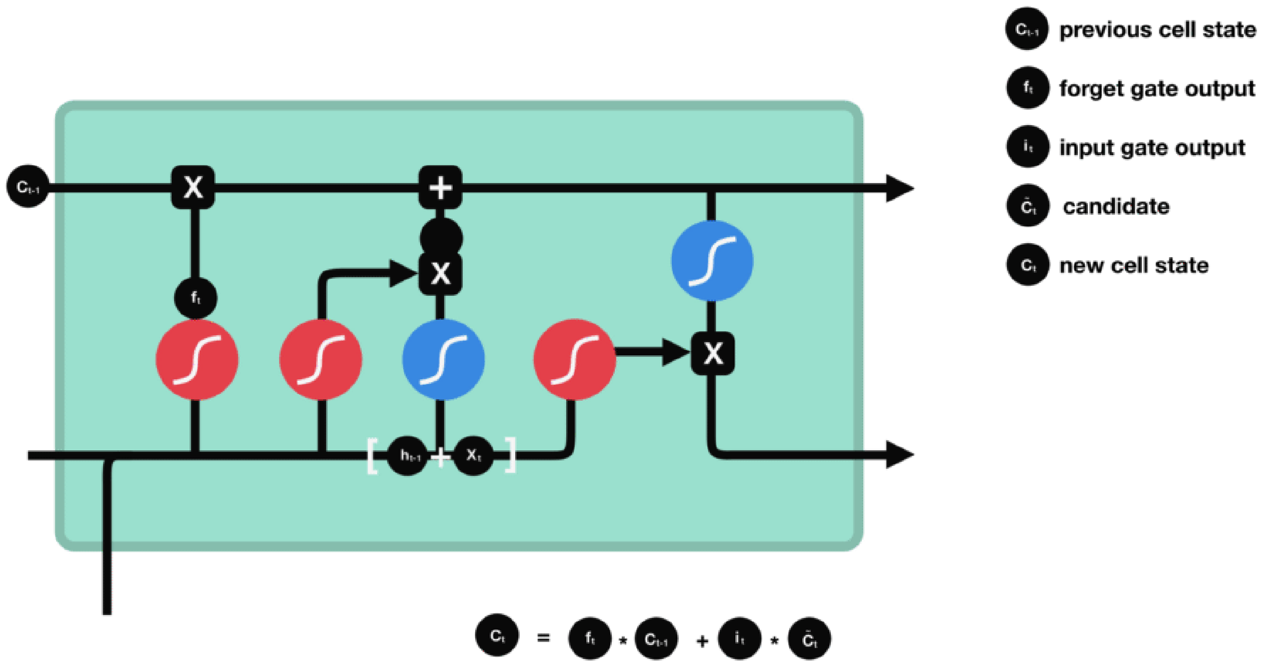
Input Gate

To update the cell state, we have the input gate. First, we pass the previous hidden state and current input into a sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important. You also pass the hidden state and current input into the tanh function to squish values between -1 and 1 to help regulate the network. Then you multiply the tanh output with the sigmoid output. The sigmoid output will decide which information is important to keep from the tanh output.



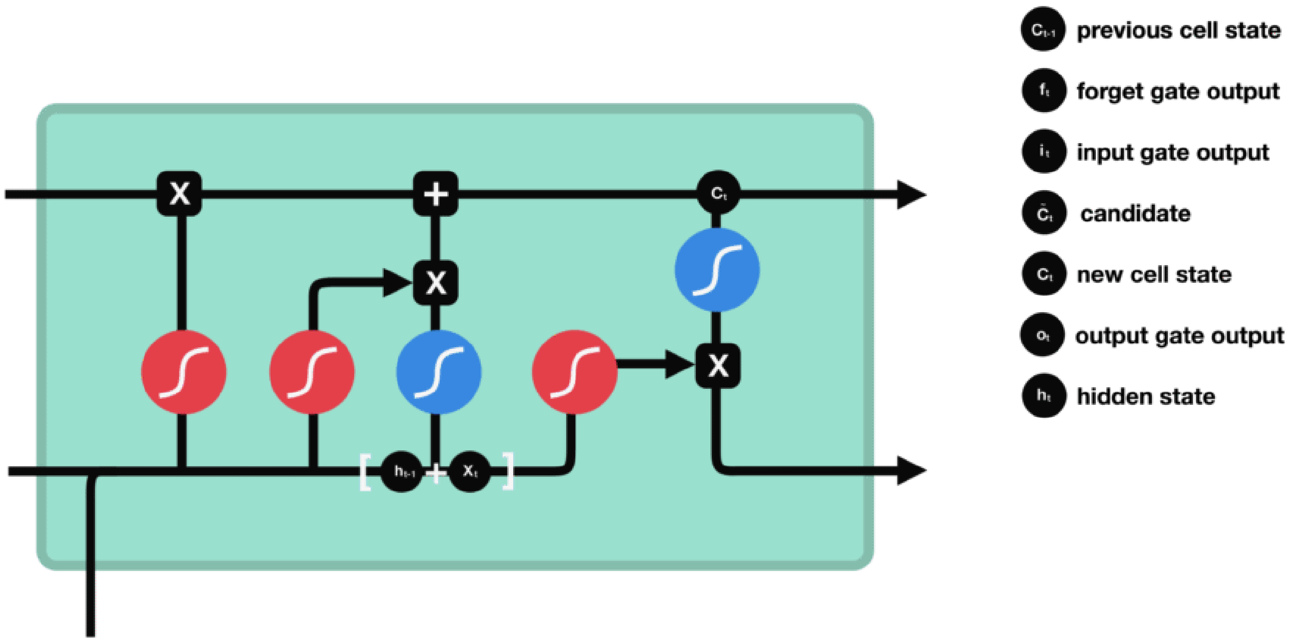
Cell State

Now we should have enough information to calculate the cell state. First, the cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.



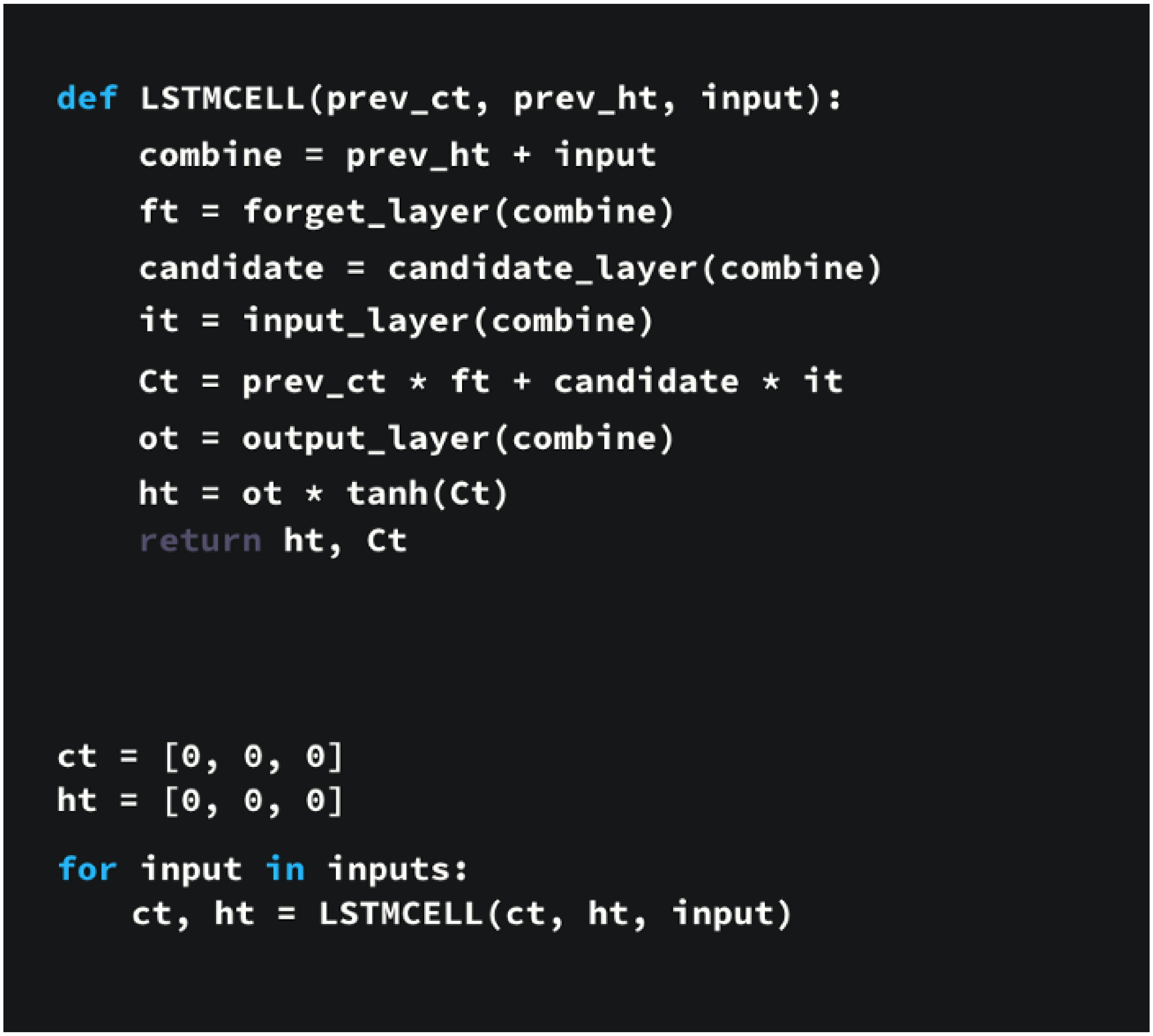
Output Gate

Last we have the output gate. The output gate decides what the next hidden state should be. Remember that the hidden state contains information on previous inputs. The hidden state is also used for predictions. First, we pass the previous hidden state and the current input into a sigmoid function. Then we pass the newly modified cell state to the tanh function. We multiply the tanh output with the sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.



Code Demo

For those of you who understand better through seeing the code, here is an example using python pseudo code.



```
def LSTMCELL(prev_ct, prev_ht, input):
    combine = prev_ht + input
    ft = forget_layer(combine)
    candidate = candidate_layer(combine)
    it = input_layer(combine)
    Ct = prev_ct * ft + candidate * it
    ot = output_layer(combine)
    ht = ot * tanh(Ct)
    return ht, Ct



ct = [0, 0, 0]
ht = [0, 0, 0]

for input in inputs:
    ct, ht = LSTMCELL(ct, ht, input)
```

1. First, the previous hidden state and the current input get concatenated. We'll call it *combine*.
2. *Combine* get's fed into the forget layer. This layer removes non-relevant data.
4. A candidate layer is created using *combine*. The candidate holds possible values to add to the cell state.
3. *Combine* also get's fed into the input layer. This layer decides what data from the candidate should be added to the new cell state.
5. After computing the forget layer, candidate layer, and the input layer, the cell state is calculated using those vectors and the previous cell state.
6. The output is then computed.
7. Pointwise multiplying the output and the new cell state gives us the new hidden state.

That's it! The control flow of an LSTM network are a few tensor operations and a for loop. You can use the hidden states for predictions. Combining all those mechanisms, an LSTM can choose which information is relevant to remember or forget during sequence processing.

REFERENCES:

1. **https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9 (https://towardsdatascience.com/illustrated-guide-to-recurrent-neural-networks-79e5eb8049c9)**
2. **https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21 (https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21)**
3. **https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (https://colah.github.io/posts/2015-08-Understanding-LSTMs/)**
4. **https://hackernoon.com/understanding-architecture-of-lstm-cell-from-scratch-with-code-8da40f0b71f4 (https://hackernoon.com/understanding-architecture-of-lstm-cell-from-scratch-with-code-8da40f0b71f4)**
5. **https://www.topbots.com/exploring-lstm-tutorial-part-1-recurrent-neural-network-deep-learning/ (https://www.topbots.com/exploring-lstm-tutorial-part-1-recurrent-neural-network-deep-learning/)**

Assignment

1. Go through this Post: **https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/ (https://machinelearningmastery.com/text-generation-lstm-recurrent-neural-networks-python-keras/)**
2. Add these improvements to the final code described in the post:
   1. Predict 500 characters only
   2. Remove all the punctuation from the source text
   3. Train the model on **padded sequences (https://machinelearningmastery.com/data-preparation-variable-length-input-sequences-sequence-prediction/)** rather than random sequences of characters.
   4. Train the model for 100 epochs
   5. Add dropout to the input layer, remove it from the layer before dense layer. Use Dropout value of 0.1 everywhere
   6. Submit!

Video for the session

S6Tuesday