# Phase 2 - Session 2

**Due** Saturday by 8:30am **Points** None **Available** Jan 25 at 11:30am - Feb 1 at 8:30am 7 days

## RECURRENT NEURAL NETWORKS

Administrative Stuff:

1. Hoping you have submitted the assignments.
2. From Session 2 we are changing the assignment format, more in the end.
3. Deadline will remain hard for this session too.

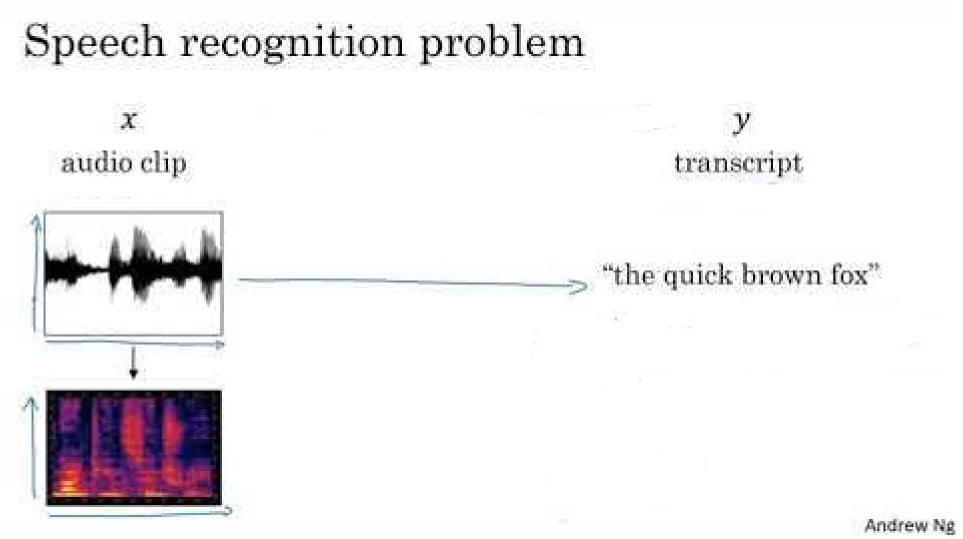
Today we are going to be talking about **Recurrent Neural Networks and LSTMS**.

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 them 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.

### Speech Recognition Example



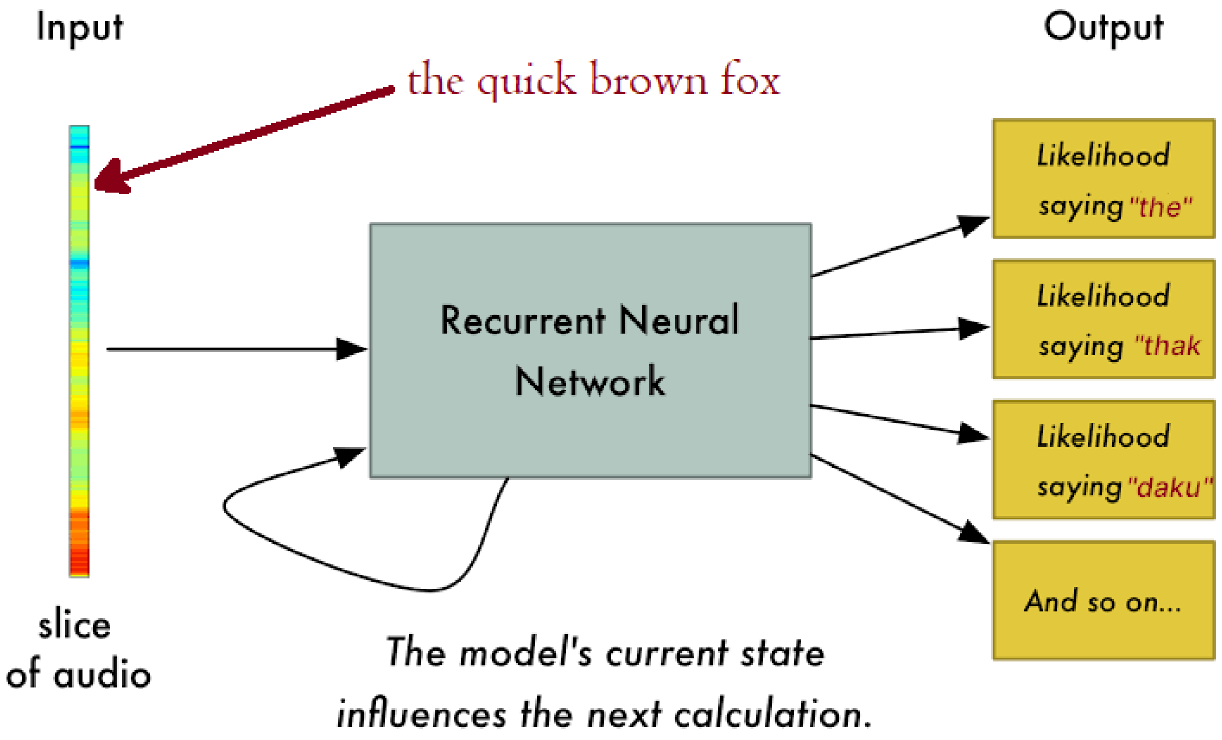
## Speech recognition problem

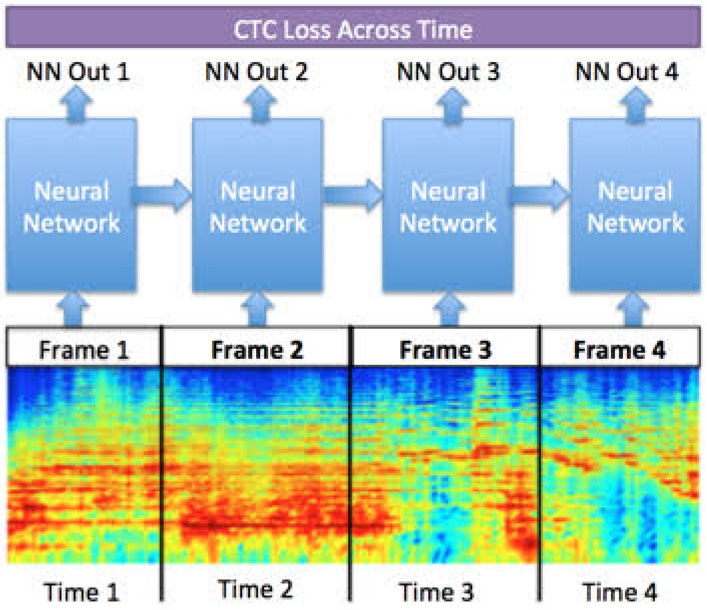
$x$
audio clip

$y$
transcript

"the quick brown fox"

Andrew Ng

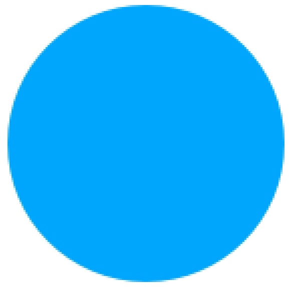

Input
Output
the quick brown fox
Recurrent Neural Network
Likelihood saying "the"
Likelihood saying "thak
Likelihood saying "daku"
And so on...
slice of audio
The model's current state influences the next calculation.



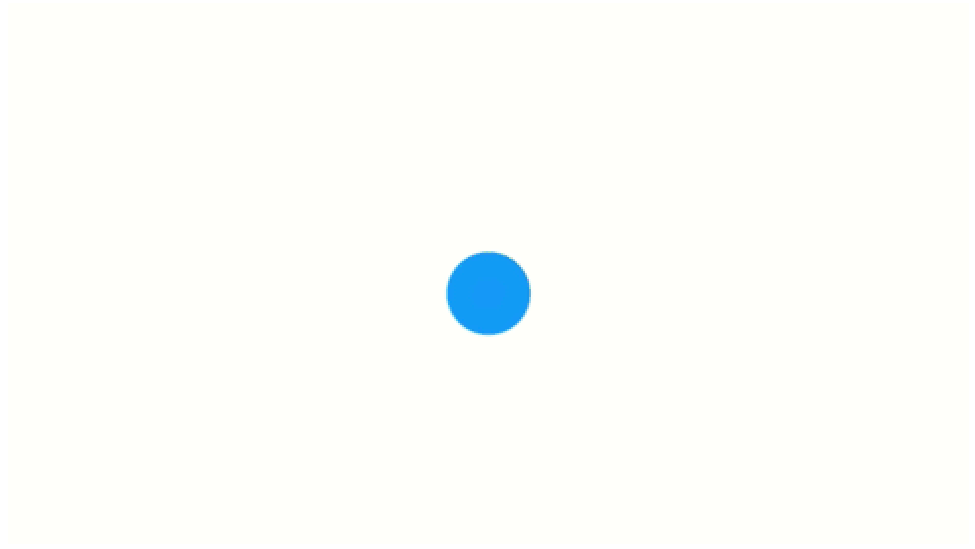
CTC: Connectionist temporal classification ([CTC (https://en.wikipedia.org/wiki/Connectionist_temporal_classification)](https://en.wikipedia.org/wiki/Connectionist_temporal_classification)) is a type of neural network output and associated scoring function, for training recurrent neural networks (RNNs) such as LSTM networks to tackle sequence problems where the timing is variable.

## 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.

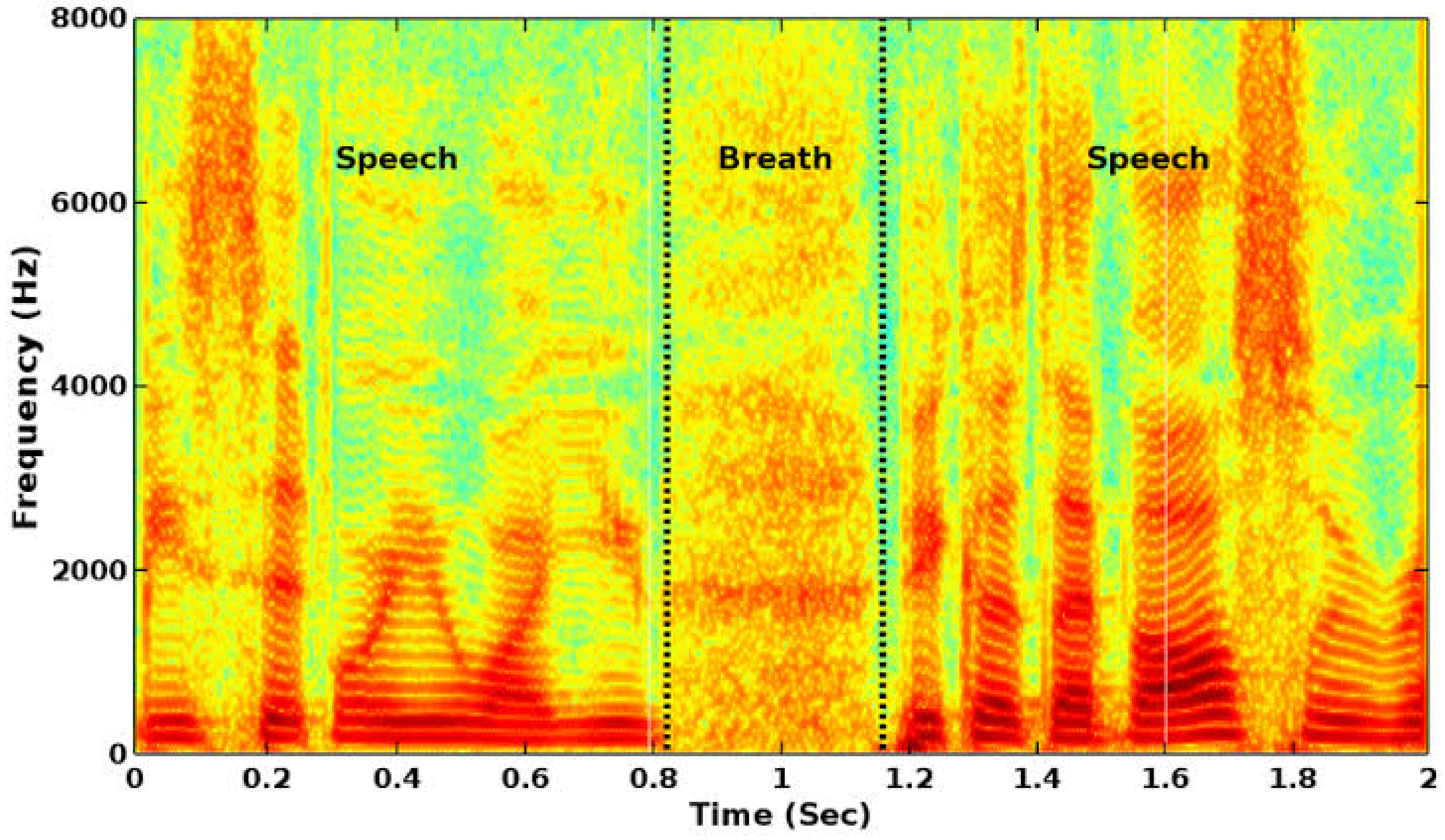


Now with the above data, as can make our predictions.

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** **(https://en.wikipedia.org/wiki/Spectrogram)** and then solved it.

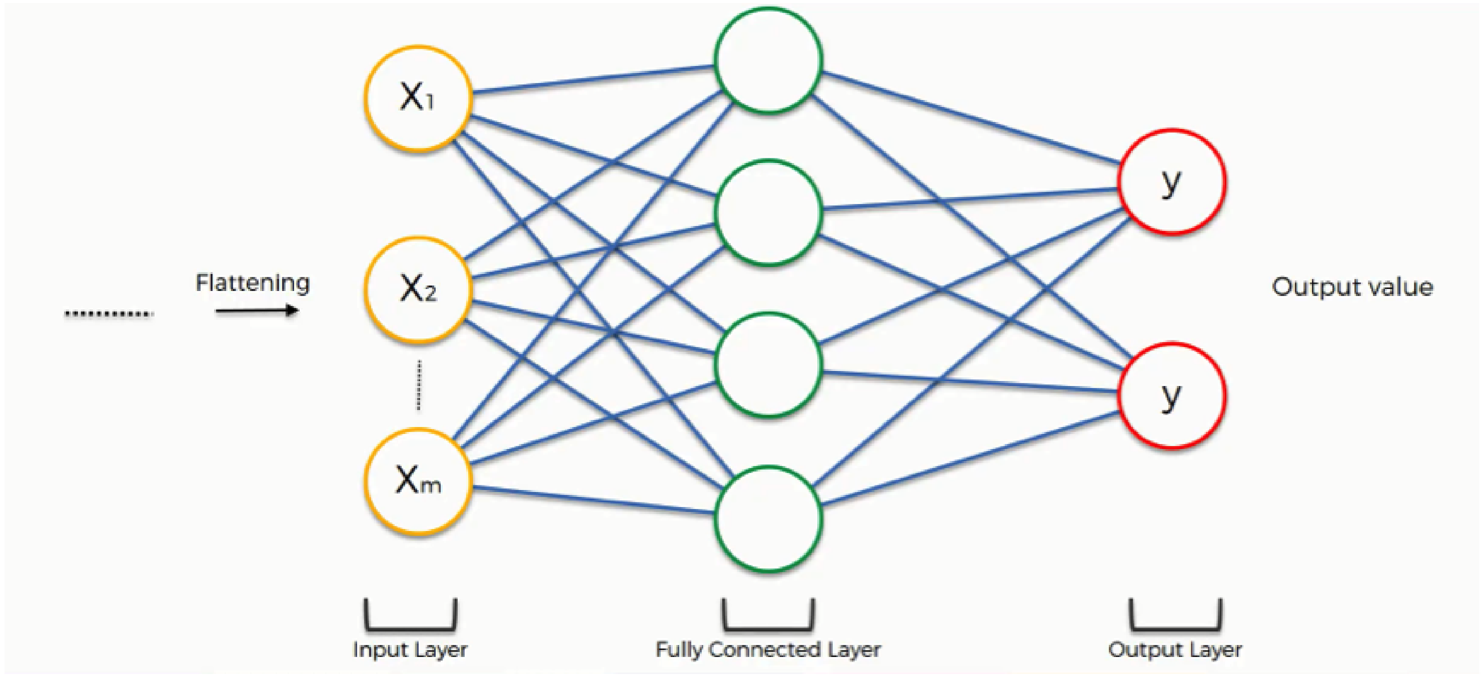


Text is another form of sequence. You can break up text into a sequence of characters or words.

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

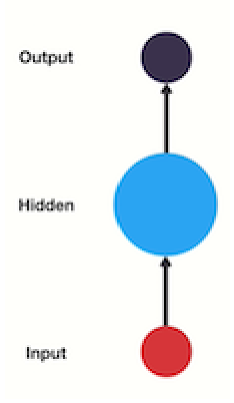


When we covered FCs we bad-mouthed it because it was destroying the 2D nature of our images. But FCs are really great while working with 1D data. We can look at the image above and realize that based on the set of input **X**s coming in, we can learn to predict set of **Y**s. This is great, because now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

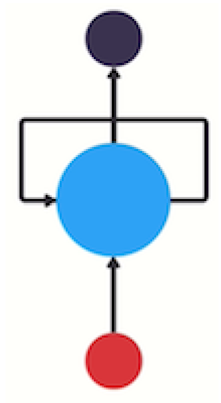
Understanding this concept is important, else we will not be able to understand what exactly is going on **inside** an RNN cell.

## 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?

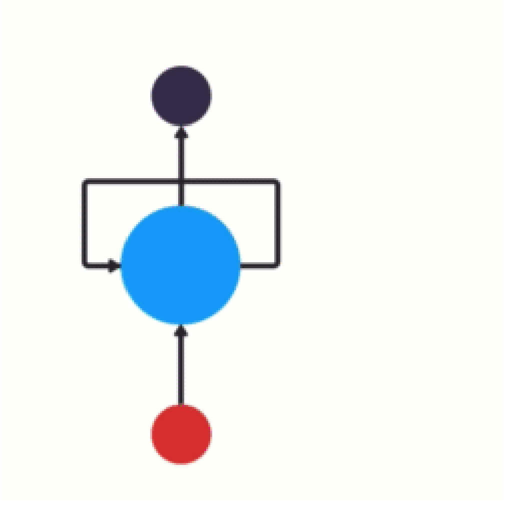


That is essentially what an RNN does. An RNN has a looping mechanism that acts as a highway to allow information to flow from one step to another.

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

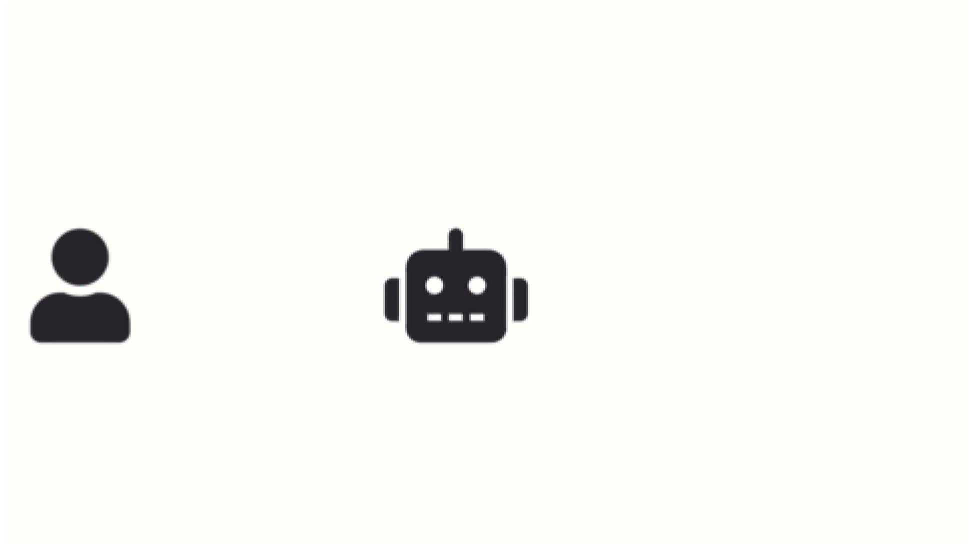
Let us assume (for the original traditional network) that our one input had 100 units as the input (1D) information, and the output had 10 units. Our FC layer must have now 100x10 weights.

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.



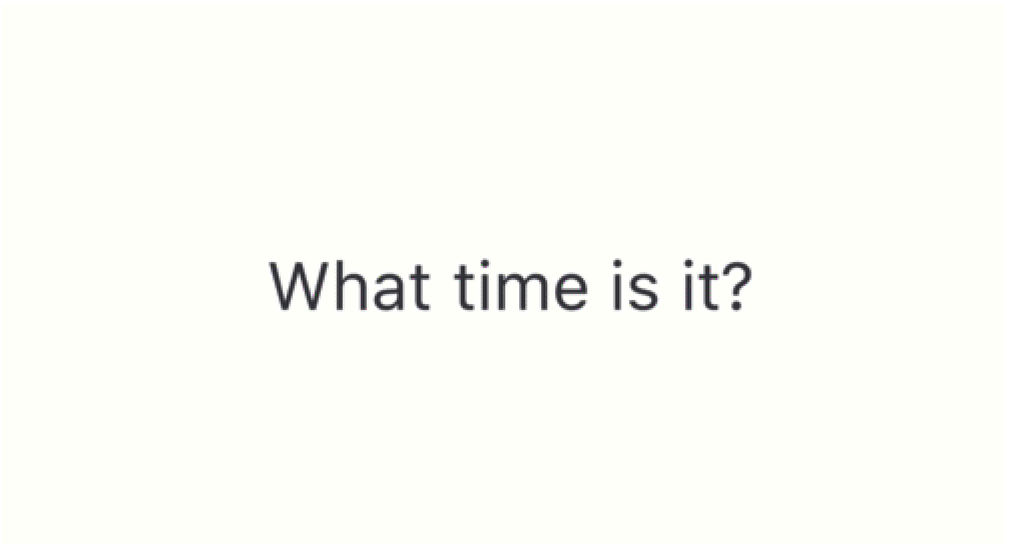
This 10 additional units we added is the representation of the previous stage (and no one stops us from saying that we will add 100 units and not 10, it would be a different kind of RNNs then).

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.

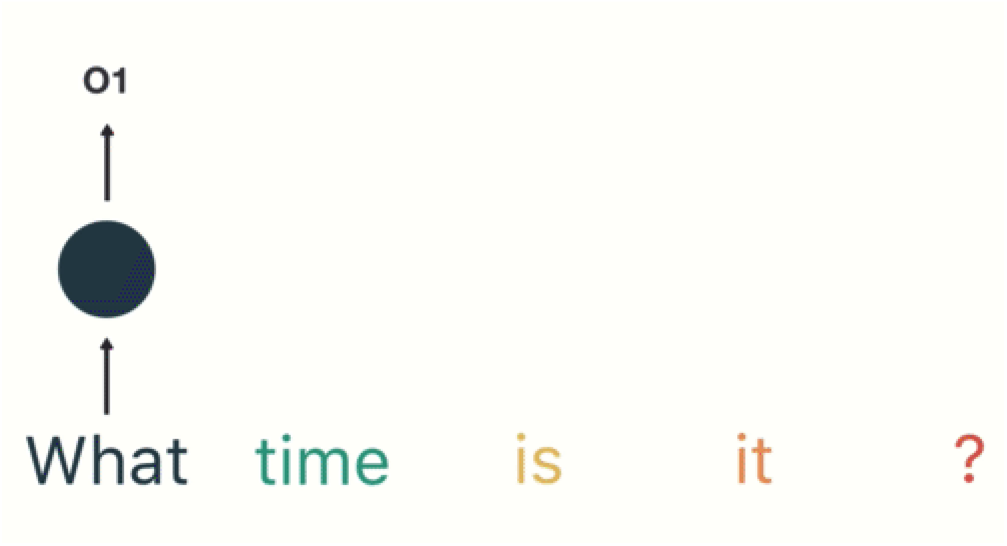
Ok, so a user types in… ***what time is it?***. To start, we break up the sentence into individual words. RNN’s work sequentially so we feed it one word at a time.



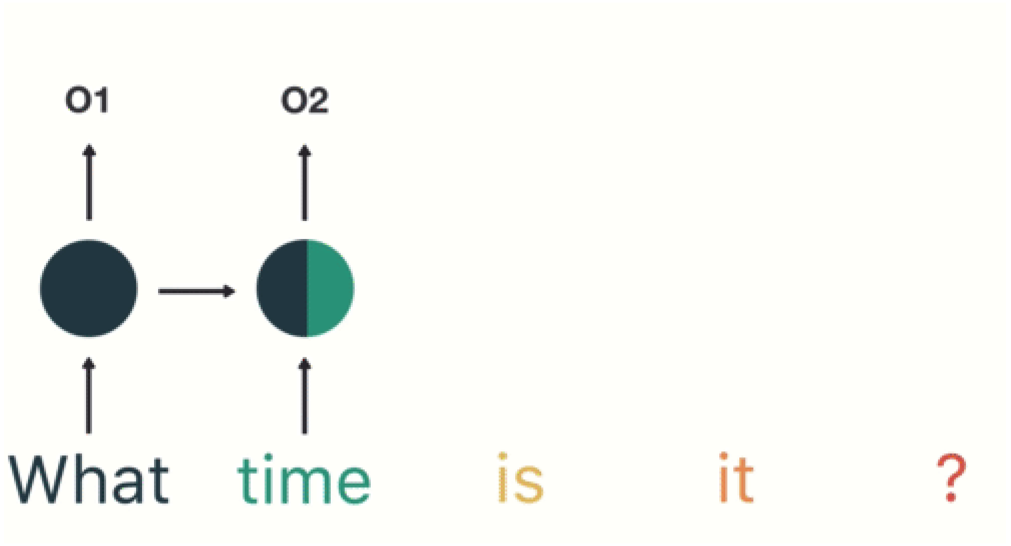
What time is it?

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

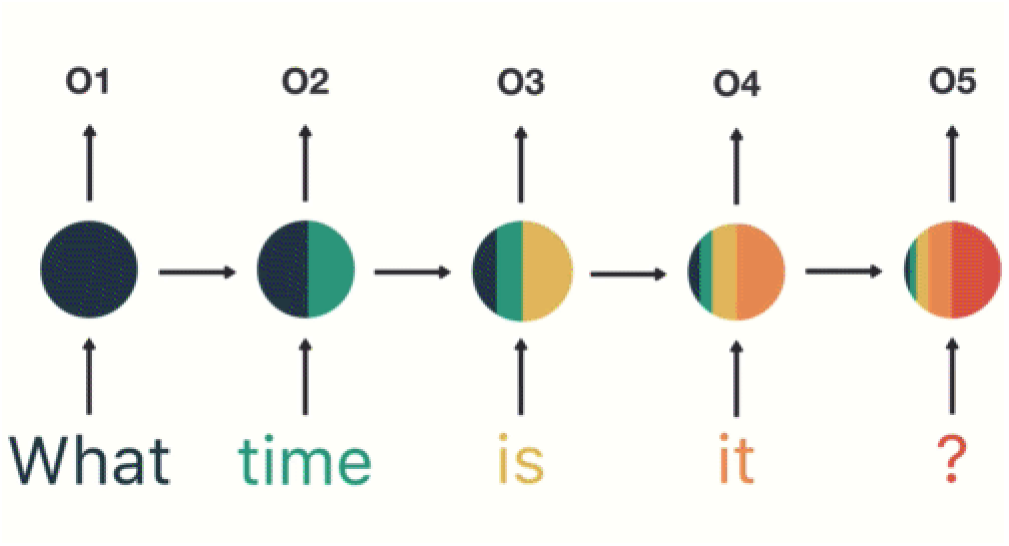
For the next step, we feed the word “time” and the hidden state from the previous step. The RNN now has information on both the word “What” and “time.”



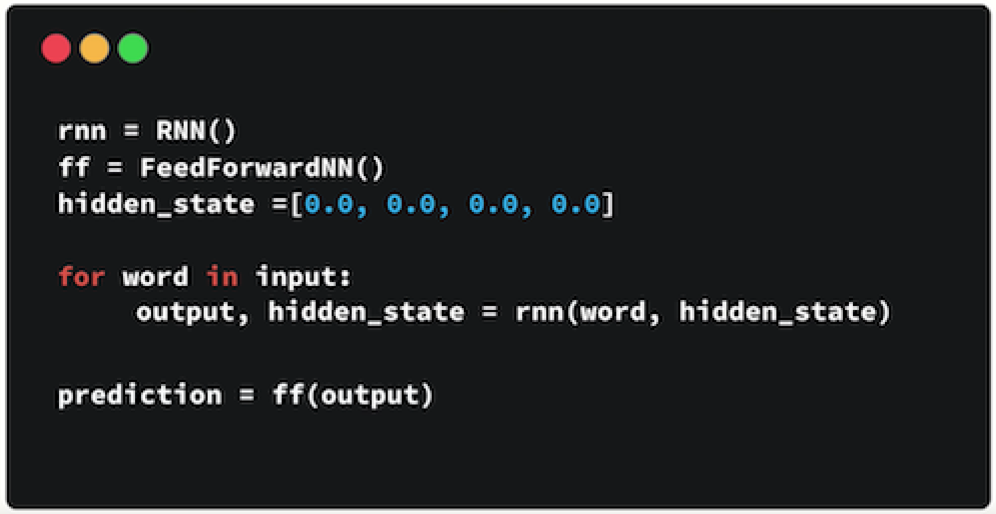
We repeat this process, until the final step. You can see by the final step the RNN has encoded information from all the words in previous steps.



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.



Here is some python showcasing the control flow

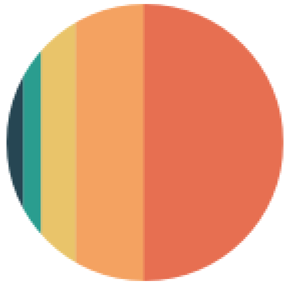


```
rnn = RNN()
ff = FeedForwardNN()
hidden_state =[0.0, 0.0, 0.0, 0.0]

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.

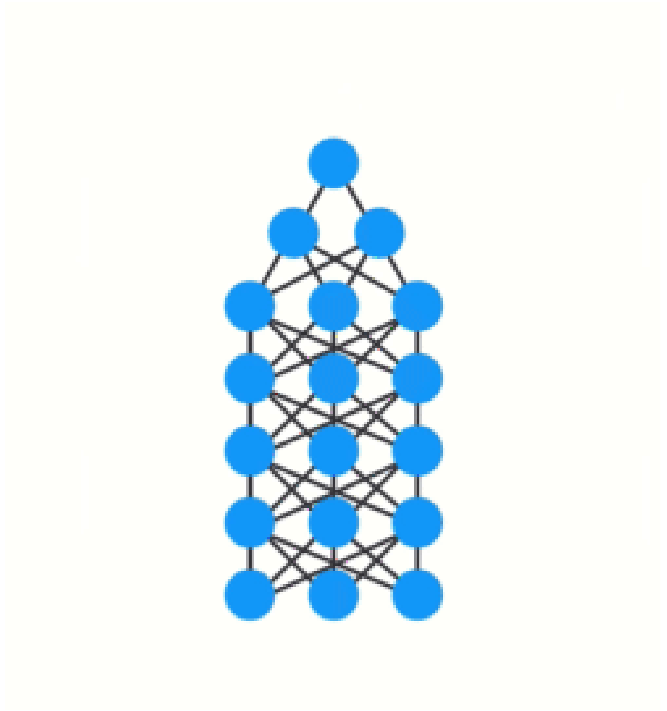


Short-term memory is caused by the infamous vanishing gradient problem, which is also prevalent in other neural network architectures. As the RNN processes more steps, it has troubles retaining information from previous steps. As you can see, the information from the word "what" and "time" is almost 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

Training a neural network has three major steps.

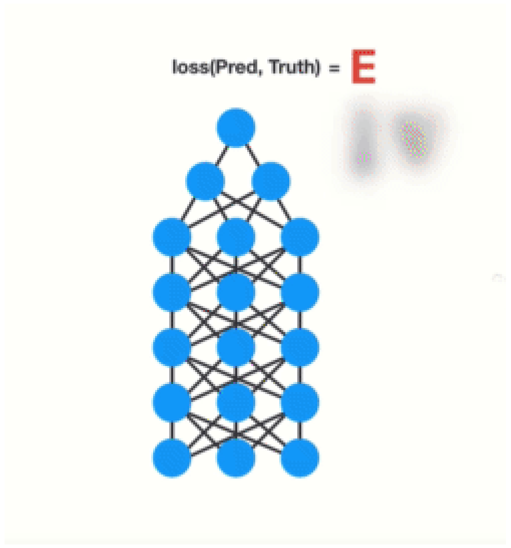
1. First, it does a forward pass and makes a prediction.
2. Second, it compares the prediction to the ground truth using a loss function. The loss function outputs an error value which is an estimate of how poorly the network is performing.
3. Last, it uses that error value to do back propagation which calculates the gradients for each node in the network.



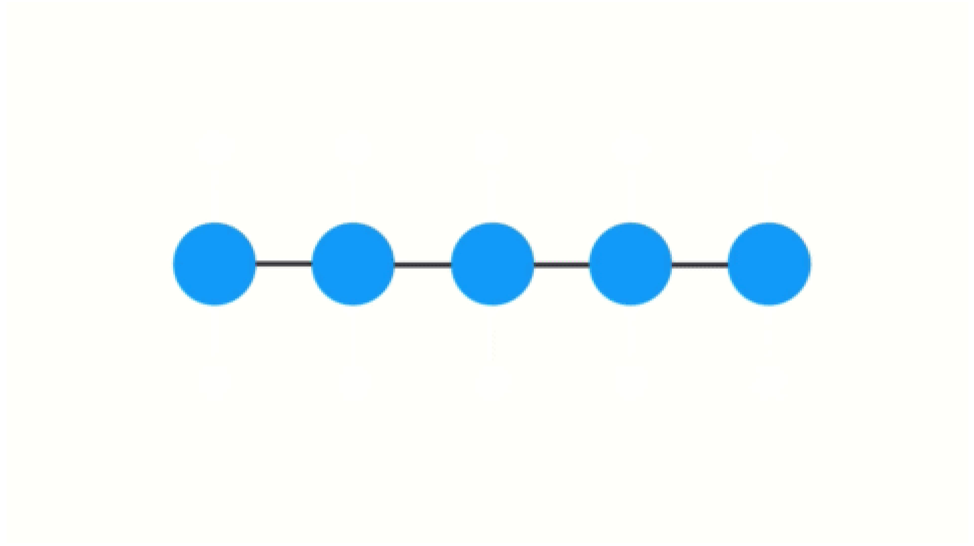
The gradient is the value used to adjust the networks internal weights, allowing the network to learn. The bigger the gradient, the bigger the adjustments and vice versa. Here is where the problem lies. When doing back propagation, each node in a layer calculates it’s gradient with respect to the effects 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.

Let's look at some code:

- [File 0](https://colab.research.google.com/drive/13Ty9j9_CD8YzKo18ZseY_JC8PLrNILE0) (https://colab.research.google.com/drive/13Ty9j9_CD8YzKo18ZseY_JC8PLrNILE0)
- [File 1](https://colab.research.google.com/drive/1XjdVj4wzkrza-p8lZzwY1Zt-pbWtS-5U) (https://colab.research.google.com/drive/1XjdVj4wzkrza-p8lZzwY1Zt-pbWtS-5U)
- [File 2](https://colab.research.google.com/drive/1bezOutxxTinsbrc2-2fGhO7K1bQeg5kV) (https://colab.research.google.com/drive/1bezOutxxTinsbrc2-2fGhO7K1bQeg5kV)
- [File 3](https://colab.research.google.com/drive/1Nim2d75wOM9cpZcJ7Fnbnzdr60f3UmJf) (https://colab.research.google.com/drive/1Nim2d75wOM9cpZcJ7Fnbnzdr60f3UmJf)

Assignment:

- Please go through these 4 cod files mentioned above "a lot" of times.
- Please go through the content above as well, as we may have question from the content as well! We are expecting you to change things in these files and be ready with answers, like changing activation function, RNN units, etc!
- You need to answer the questions in the quiz. There is only quiz and no assignment. From now on your assignment if your quiz and vice-versa.
- Your quiz is time, so make sure that you have gone through the quiz, else you will not be able to finish the quiz on time.
- All the best!

Wednesday Batch Video:

*Sunday Batch Video:*

EVA Phase 2 Session 2 Sunday