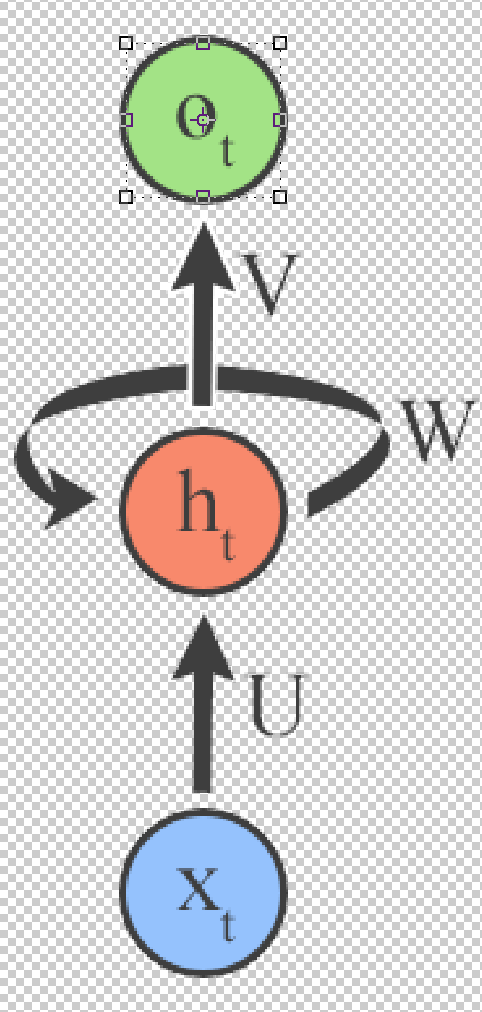
**Beginners Guide to RNNs in TensorFlow!**

**IF YOU KNOW ANSWERS TO THE QUESTIONS IN THE HEADLINE FEEL FREE TO SKIP THAT SECTION**

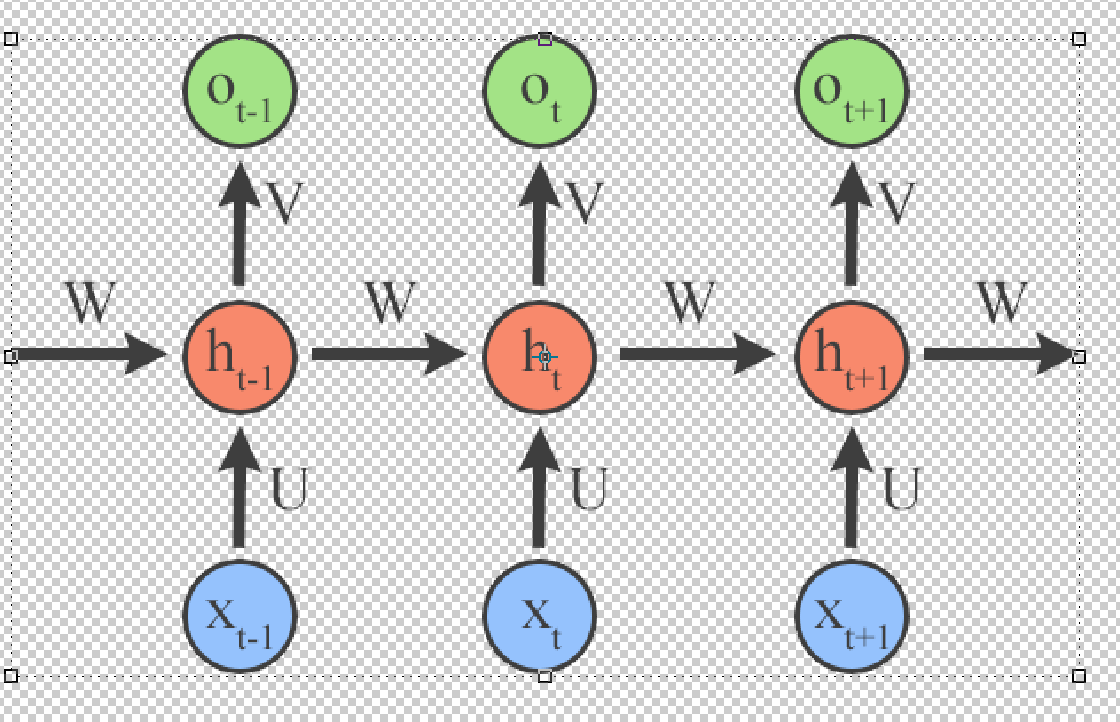
**What is a Recurrent Neural Network (RNN)?**

RNNs are similar to Feedforward Neural Networks, but a RNN is used to process sequential data, where things that happened in the past could have an effect on the future. (What’s a Feedforward Neural Network? https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/) For instance if you want to predict what will be the next word in a sentence, you will want to know which words have already appeared in the sentence. A RNN is able to work with sequential data like this because it has cycles in its computational graph which send information at one time step t, to the same neuron at a future time step t+1. This way an RNN is able to have a “memory” of what has been seen thus far. This graph is depicted in figure 1.



**Figure** 1: The input comes in through the blue neuron, which is then processed by the hidden red neuron, which sends output to the green neuron. U, W, and V are weight matrices that are used in the computation. They are called input-to-hidden, hidden-to-hidden, and hidden-to-output connections respectively.

It is easier to visualize what is happening if we “unroll” this graph. Unrolling simply means to show every computation that will occur when the RNN is ran. In figure 2 you can see an unrolled graph for time steps t-1, t, and t+1 where t is the current time step.



**Figure 2**: Some of the output from the hidden neuron h\_t-1 is used as input to the hidden neuron at t.

As you can see from the unrolled graph, the weight matrices V, W, and U remain the same throughout every computation. This is known as parameter sharing. At each time step we use the same parameters to compute what the next element in the sequence will be. This allows us to use the model on varying length sequences and to recognize patterns in the data to when those patterns occur at different time steps. The alternative to parameter sharing would be to have different weight matrices at each time step. With different weight matrices, we would only be able to test on data with the same sequence length. For instance if we increased the sequence length when testing, we would not have weight matrices for the extra positions. Additionally, if we were trying to predict the next word in a sentence and were presented with two examples to learn from, “Alan Turing was a genius.” and “What did Alan Turing do?”, seeing ‘Alan’ then ‘Turing’ in the first sentence would not help us at all when having seen ‘Alan’ and trying to predict ‘Turing’ because the weight matrices from 1st and 2nd positions would be different from the matrices in the 3rd and 4th positions.

//It is important to note that when feeding in input to the RNN, we feed in input one //time step at a time, wait for the output from the current time step (t), and then //run the RNN with the input from the next time step (t+1) with the output from the //previous time step (t) depicted in figure 2.

*Mathematically how does an RNN work?*

a(t)= b + W h(t−1)+ Ux(t) (**eq 1**)

h(t)= tanh(a(t)) (**eq 2**)

o(t)= c + V h(t) (**eq 3**)

y(t)= softmax(o(t)) (**eq 4**)

We begin with equation 1 where we use our input from the current time step along with the state from the previous hidden layer to get a(t). b, as well as c in equation 3, is a bias term. Why do we need a bias term? https://stats.stackexchange.com/questions/185911/why-are-bias-nodes-used-in-neural-networks In equation 2 we use an activation function, tanh in this example, on a(t) to get the value for our current hidden state. The tanh function just squashes the input to a value between -1 and 1, and it allows for the model to learn non-linear models. For more on activation functions (https://medium.com/towards-data-science/activation-functions-and-its-types-which-is-better-a9a5310cc8f). In equation 3, we multiply our current hidden state with the hidden-to-output weight matrix V to obtain our output. In equation 4 we take the softmax of the output to obtain normalized probabilities of the output. the softmax function basically just takes a list of numbers and squashes them so that all numbers add up to 1.

*What are the dimensions of these variables?*

x [n x b]: n depends on your data. For instance, if your input was one hot encoded word vector (what?) with a dictionary size of 5,000, your input dimension would be 5,000 (the length of the vector). If your input was just a single number with values ranging from 0 to 10, your input dimension would be 1 (the dimension of a single number). Specifically, n is *equal to the number of features in your data*. b is your batch size.

U [m x n]: m can be chosen to be whatever you desire

h [m x b]

W [m x m]

V [k x m]

o [k x b]: k is determined by what type of problem you are solving. If you are solving a classification problem (such as deciding a students grade, A+, A, A-… ) then k will be the number of classes. If you are solving a regression problem (such as a predicting a students GPA), k will be 1 since you are just predicting one value.

If the NN is a regressor, then the output layer has a single node.

If the NN is a classifier, then it also has a single node unless softmax is used in which case the output layer has one node per class label in your model.

If you are looking for a more detailed and mathematical description see here:

**What is TensorFlow and How does it Work?**

TensorFlow is an open source framework that runs on Python, Java, and C++. It allows you to easily implement a Neural Network without having to worry about most of the math involved.

*What are Tensors?*

Tensors are basically just arrays/vectors/lists of any dimension. The formal definition is…

*How do you download and run TensorFlow?*

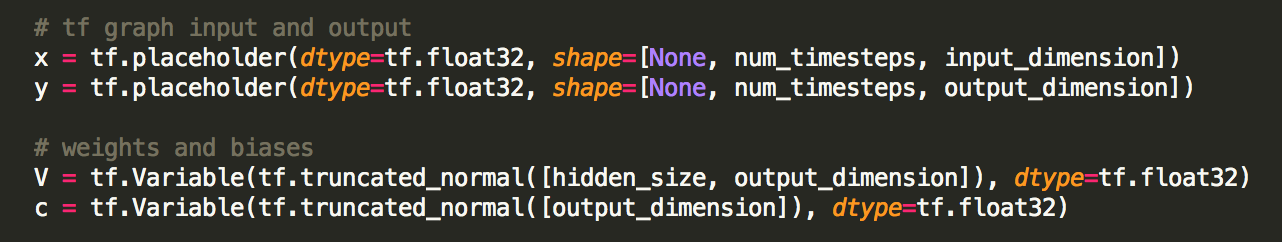
https://www.tensorflow.org/install/

*What are variables and placeholders?*

In TensorFlow, there are a few main types of tensors that you can create. The two that are important for us are variables and placeholders. Variables allow for values inside of the tensor to change. The weight matrices will be variable tensors. This means that the values in the weight matrix can change, otherwise how would our model learn? Placeholder tensors are exactly what they sound like. They are a tensor of a certain shape that alone are just to fill in a node of our model. The placeholder will later be filled in with values that we assign to it. Placeholders are helpful when we want to feed in input to our model. The values of the input will change with each example we run, but the location of the input in our model and the shape of the input will be the same for each example. In image 3 you can see where we would have variable tensors and where we would have placeholders.

**How do you setup a RNN in TensorFlow?**

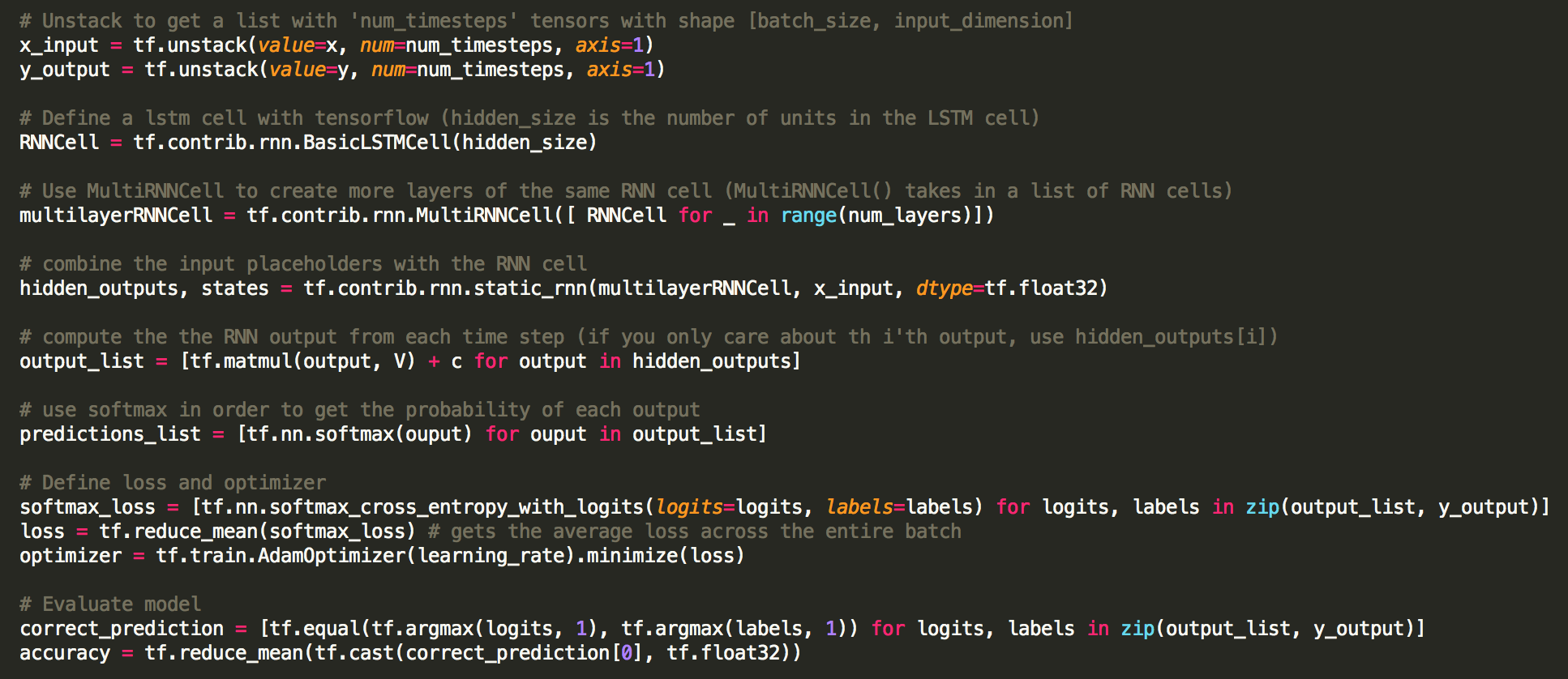
We will first setup our placeholders that will contain values for the input (x) and output (y) of each example. The “None” dimension of these tensors corresponds to the batch size. What is batch size? We will also create two variable tensors that will be used as the weight and bias matrices. These variables are initialized with a “truncated normal” distribution which means that values more than 2 standard deviations (default 1) away from the mean (default 0) will be recalculated.



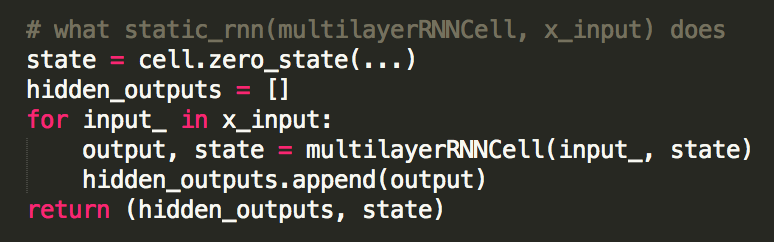
*Why use a truncated distribution?*

Truncating the weight matrices’ values makes your network learn quicker compared to having many values of the matrices outside the truncated range. If you allowed for values greater than 2 standard deviations away from the mean and used an activation function such as a sigmoid (where the function is basically flat at values greater than 2), then when updating your weights, the gradient will be very small which means it will take much longer for your neurons to learn.

To create the RNN we can use one of the many RNN types of cells that TensorFlow provides. I will use an LSTM (Long Short Term Memory) cell because those have been shown to retain information for a longer period of time. What are LSTMs? (https://medium.com/@shiyan/understanding-lstm-and-its-diagrams-37e2f46f1714). See more RNN cell types you can use. (https://www.tensorflow.org/api\_docs/python/tf/contrib/rnn)



The input and output had to be unstacked to the correct dimensions that static\_rnn accepts [*number of time steps* x *batch size* x *input dimension*]. You can avoid unstacking your input if you provide the correct dimensions initially (with my data set it was easier to unstack here). static\_rnn() needs these dimensions because it works as follows.



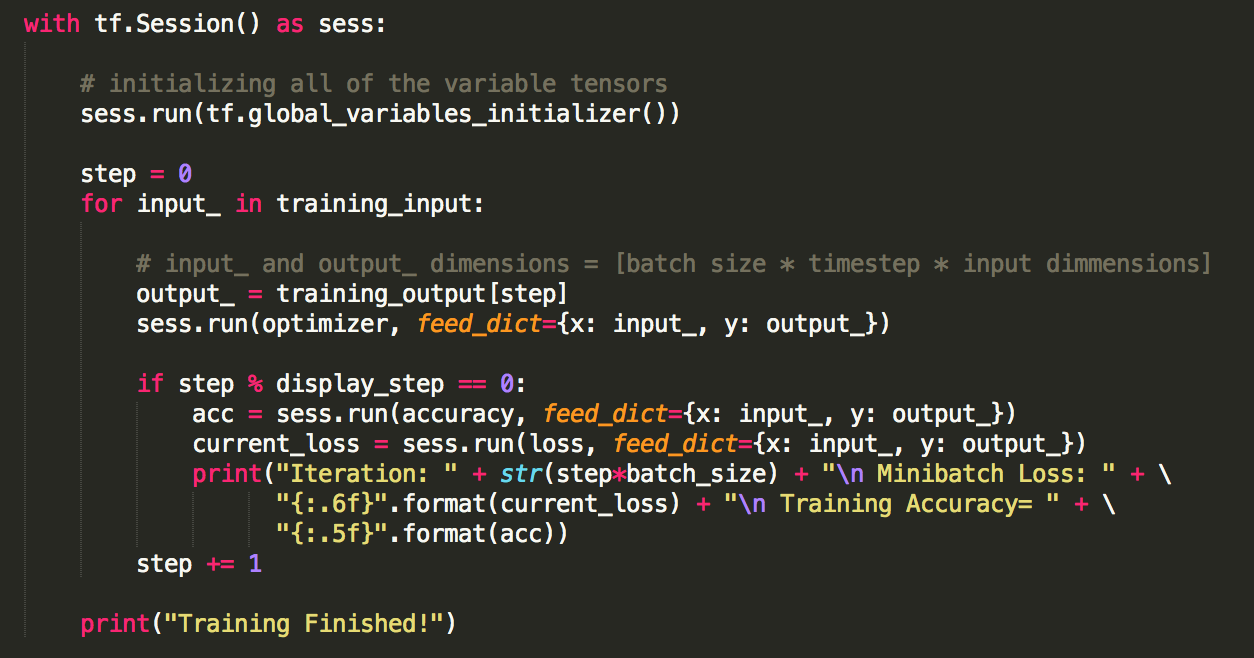
What does “for \_ in …” do?

softmax\_cross\_entropy\_with\_logits?

AdamOptimizer?

**How do you run a RNN in TensorFlow?**

It is important to note that up until this point, no for the RNN has been executed. We have simply set up what our RNN will be like. In order to make our RNN learn, we must run our RNN on a TensorFlow Session. A session encloses the information necessary for the RNN to be executed. Sess.run() takes in a node of our computation graph (optimizer in this example), and a dictionary that contains any placeholder values that the node is dependent on.



In the example above we are just training our model. To test it, you would simply make another for loop but with test\_input rather than training\_input and use the same TensorFlow session when evaluating the test\_input (because the session contains the RNN information, such as the weight matrices).

**What now?**

I did not use a specific example when making this walk through because I wanted the code to be understandable when you use your own data that may be of a different form or where you only need certain outputs from your model. Use this as an outline to create a RNN to fit your sequential task at hand and please reach out to me with any questions!

The output would be of the shape [time\_steps \* *batch\_size \* input\_dimmension*]

Visuals!!