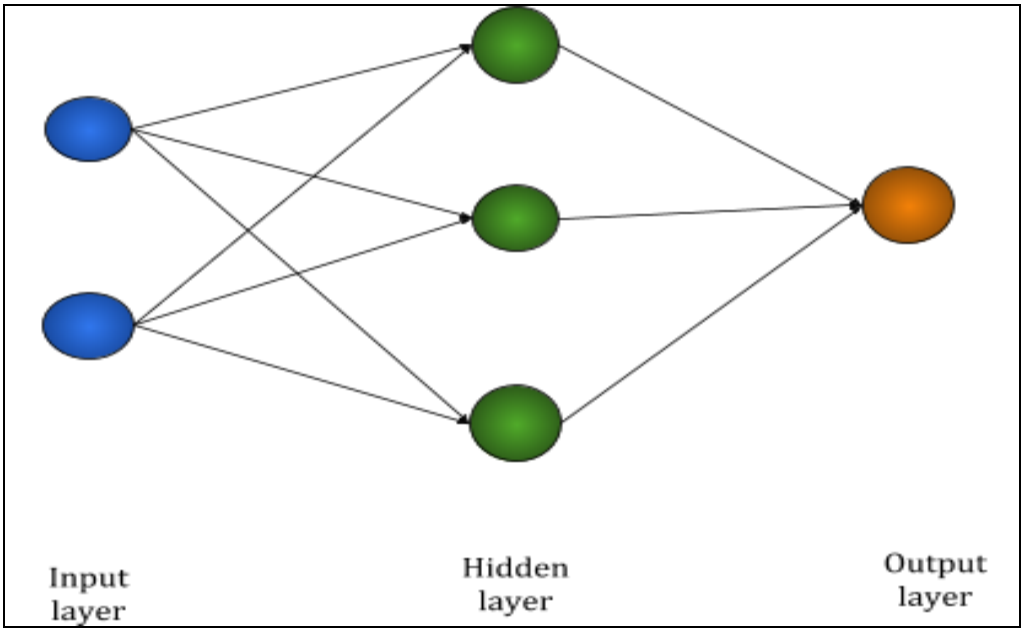
RNN:

To understand Recurrent Neural Network, let's start with the simple neural network.



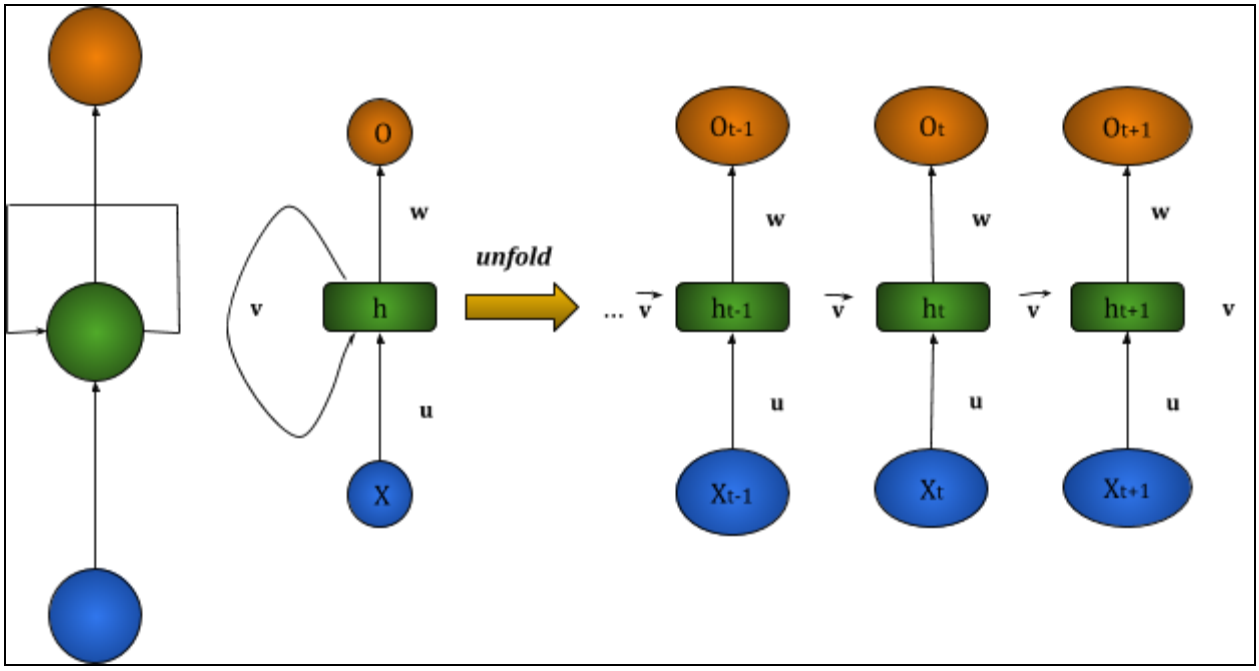
Input Layer: A simple neural network as studied earlier, will have some inputs - can be one. Two or multiple.

Hidden Layer: contains neurons. Depending on the model that we are using, there can be multiple hidden layers. One hidden layer may contain thousands of neurons. The inputs may be connected to all the neurons or may be connected to single neurons. It varies from model to model. In regression, we saw that the neural network are connected in a similar fashion as the figure above. In case of CNNs, we need to extract some portion, do a downsampling to some other things to get get an output and there can be multiple output.

Output Layer: Gives the output. Can be one or multiple.

Basic working philosophy of neural network -
In chapter 4, we have understood that we have inputs and on top of that some weights are applied to form the hidden neural layer and from that we get the output. In order to minimize the loss, you keep on applying weights using gradient descent methods until a satisfactory weight is obtained which significantly reduces the loss i.e. the neurons tries to give a value which is close to the expected value.

Now in case of dependent output, say in case of time series, where the consecutive values are dependent on the previous (second value is dependent on the first, third value is dependent on the second and so on….), say for example financial markets. In such case it is not possible to use a neural network such as above in which the inputs are not interlinked with each other. Hence, a different kind of approach is required.



Modifying the three hidden layer at the left, we have the input layer, the hidden layer which forms a loop to itself, and then we have the output. At the right hand side, we have unfolded the diagram to see what it looks like. Initially, there is no data to feed into h = t-1 layer. So we pass it with zero or some other default value to start with.

Now, from the above neural network, we have 2 input layer, 3 hidden layer and output. We will refer that. At time x = t-1, we use the input values, fetch the value of the hidden layer from the previous layer, or a default value or weight. The output of this hidden layer will give output of this layer which will serve as an input for the next hidden layer. From the figure, it is clearly reflected that the prediction of each layer is dependent on the previous layer.

Therefore, getting back to the definition of Recurrent Neural Network, it is used when the prediction is dependent. E.g: time series data. In this, the layers are connected in such a way (as seen above) that it will give the output which will be the input of the next step. It creates a cycle in a network graph, so that internal state can be maintained and used for the next element.

Applications:

- Word prediction

- Machine Translation
- Speech Recognition
- Image captioning
- Predicting finance markets, share prices etc.
- Movie subtitle (e.g: youtube)

RNN in PyTorch:

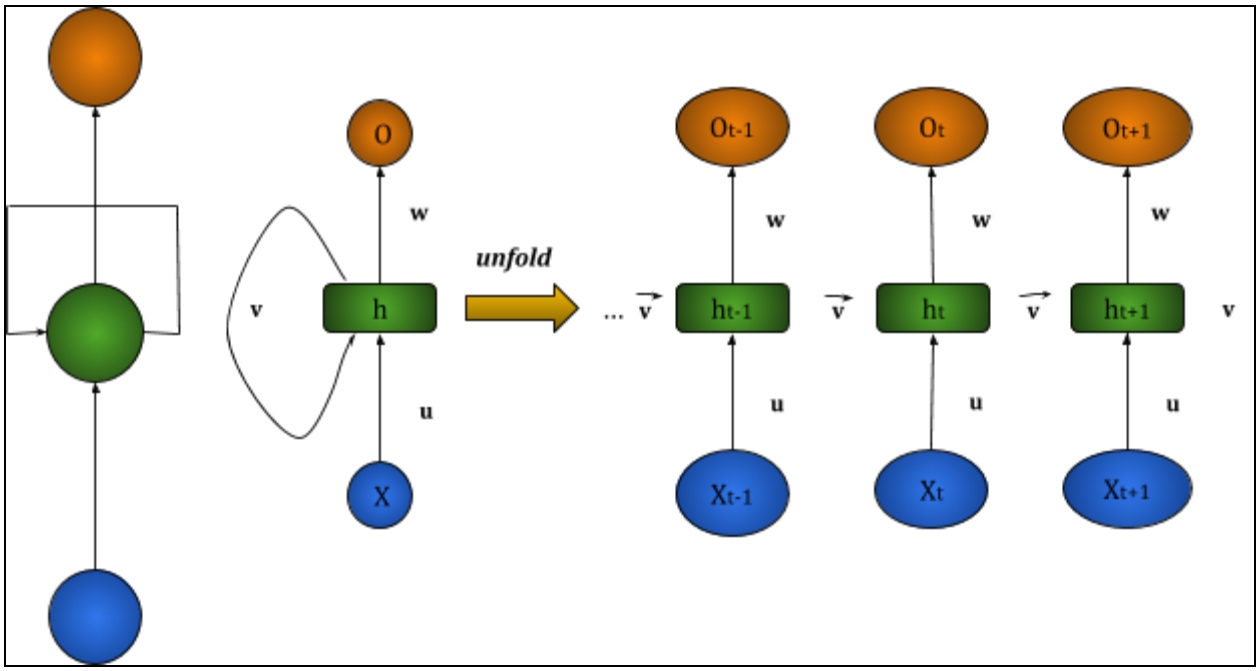
RNN to predict character sequence.

CODE FROM COLAB NOTEBOOK

LSTM:

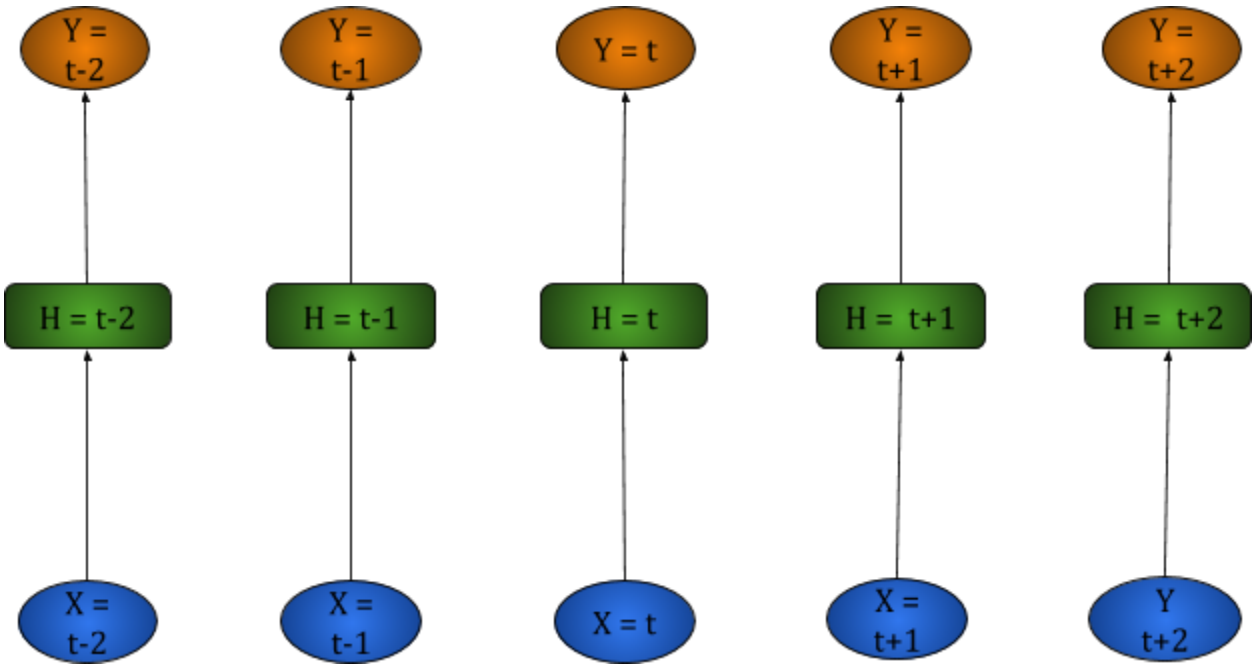
Problems of RNN: Vanishing gradient problem an exploding gradient problem.

Vanishing Gradient Problem:
Recapping from RNN section, we saw this kind of a diagram as below -

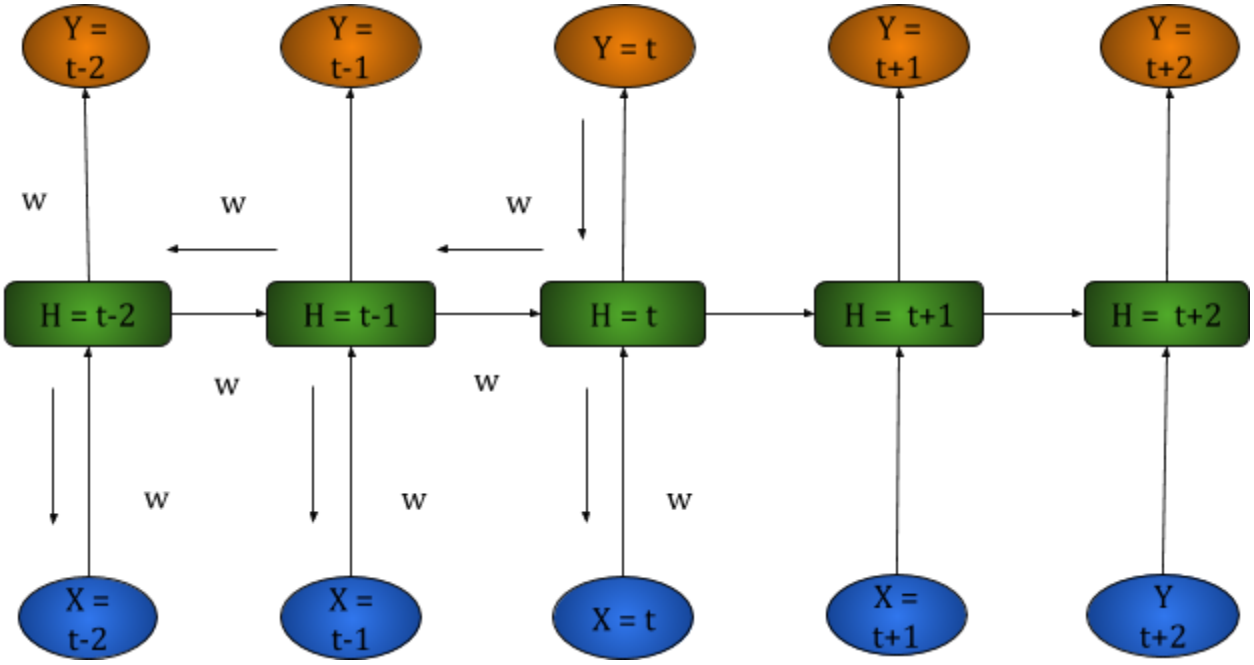


We considered 2 inputs, 3 hidden layers and an output. At each time steps, we have different hidden layer, which stores some states and then after some computation, it forwards it to the next hidden layer. Therefore at each layer, we require the output of its previous layer and feed the input and get the output accordingly. Ideally, at each epoch, we also compute the loss.

Now, let's look at a complex model.

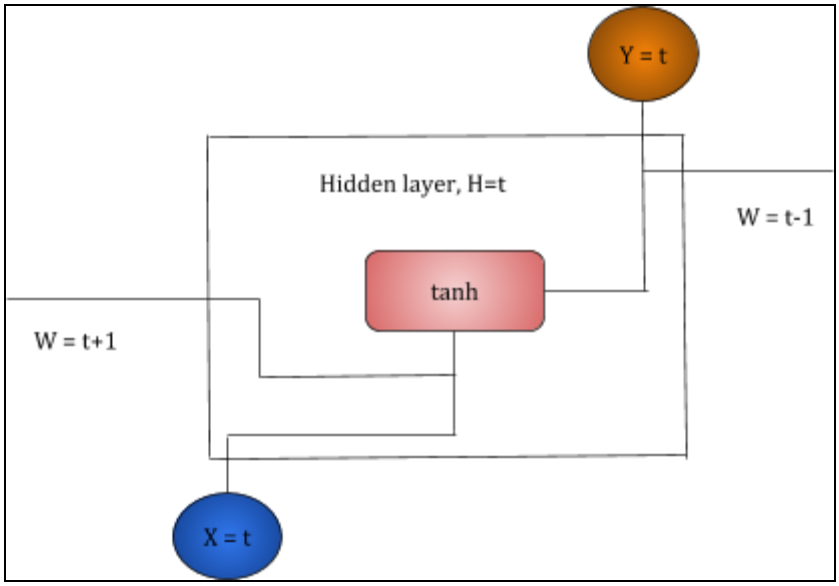


This above diagram is a case of many input an manyt output sitiation. So, at x=t, you feed in data from two input layers and also the output of the previous hidden layer. Now let's calculate the loss at this layer. We can see that the output value at y=t is dependent on the hidden layer h=t which is further dependent on the output of its previous layer, h=t-1 and this is dependent on the two inputs at x=t-1 and the output of its previous hidden layer which is h=t-2 and so on. Now let's try to compute the loss at y=t.



Gradient descent involves updating weights so as to train neurons. But, from the above diagram, you can see that there are multiple weights involved. So, when gradients starts propagating throughout the layer, it is possible that it may miss some weights which has a very small value. Everytime, with backpropagation, we are trying to adjust the weights. So if eh gradient is high, your weight distribution would be high. It is dependent on the weights, how fast we get a solution. In simple terms, if the weight is very small < 1, then it is a case of a vanishing gradient problem. There is a math behind it, but that is beyond the current scope. If the weight is one, it may take more time and it is an exploding gradient problem. Therefore, weights are multiplied, if the weights are large, we get the output very fast and if it is small, it is generally slow to get the output.

The use of non-linearity as a tanh activation function:



Again here, as above, you have 2 inputs, 3 hidden layer and 1 output. The weights of the previous hidden layer are also used. We combine these and feed it through a non-linearity function which which convert the prediction from -1 to 1 and in case of sigmoid function, it will convert the prediction in the range of 0 to 1. Once the tanh function converts the values, the predicted value and this value is passed on to the next layer. Adjusting weights through tanh function is done at all the connections and not just at the output.

LSTM is the solution to this vanishing gradient descent problem. It uses forget gates and hence prevents vanishing or gradient descent problem. These gates are similar to logic gates. These gates decide whether or not to pass the information ahead. So, it is kind of a filter. In this we will be using **cells** which will act as gates allowing or preventing entering of data.

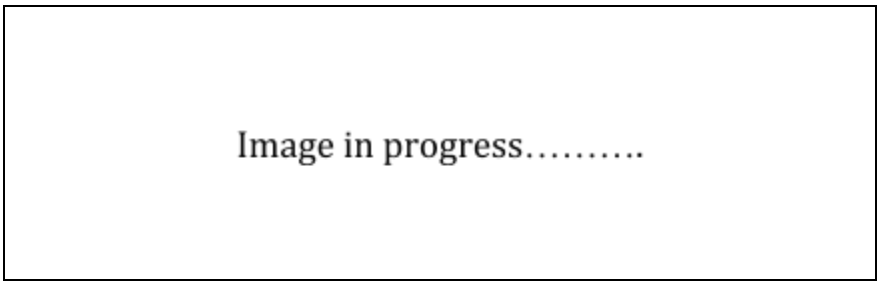


Fig: LSTM flow

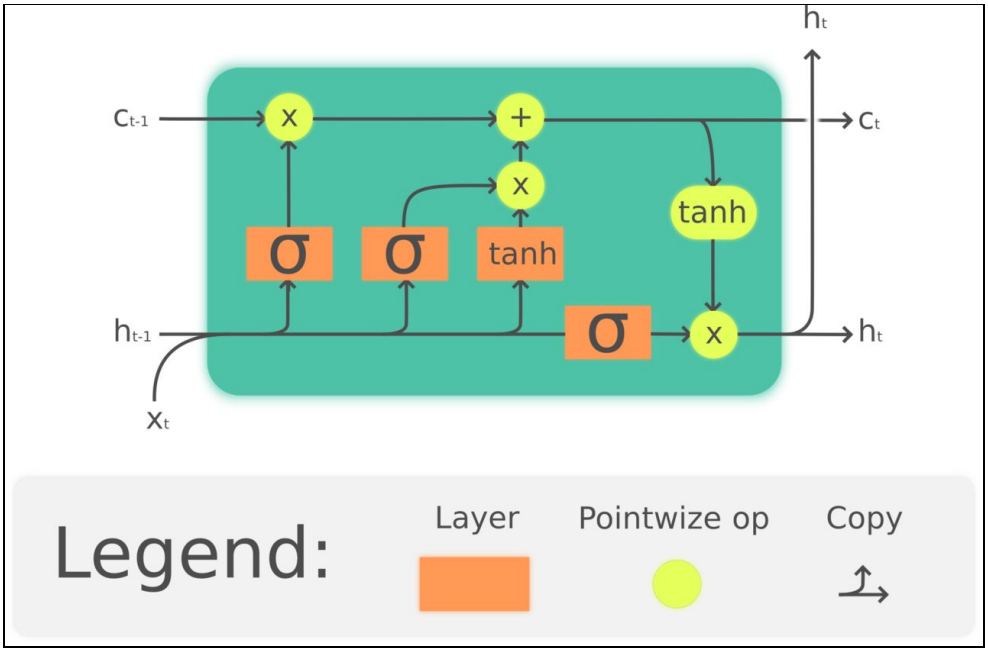


Fig: just the LSTM unit. source: wikipedia

In the LSTM unit , the hidden layers and input layers and combined, then we pass this through a sigmoid function, then again through a sigmoid function, the through a tanh function and then again through a sigmoid function. Again, as stated above, sigmoid function will convert the value in the range of 0 to 1 and tanh function will convert the value in the range of -1 to 1. The gates are well reflected, we have the multiplication and addition gates in this. Accordingly, the gates will calculate the value from the hidden layer as well as the output of the sigmoid and tanh function. If the value turns out to be zero, then it will be locked at that gate. If so, then in the next gate, new values will be updated in the cell and it will be passed to the next hidden layer. Again, there are mathematical explanations to this, but it is beyond the scope of this book.

LSTM in PyTorch:

In this, we will modify the previous RNN project in LSTM.

CODE FROM COLAB NOTEBOOK