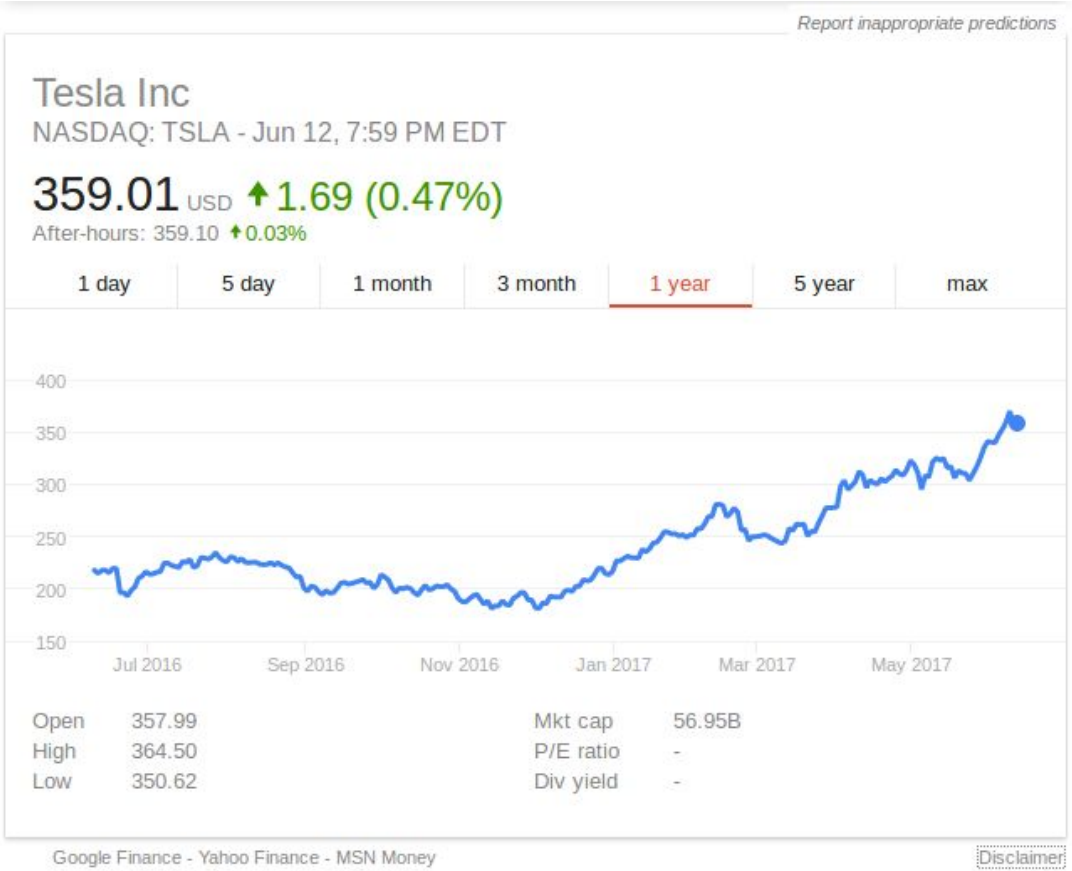
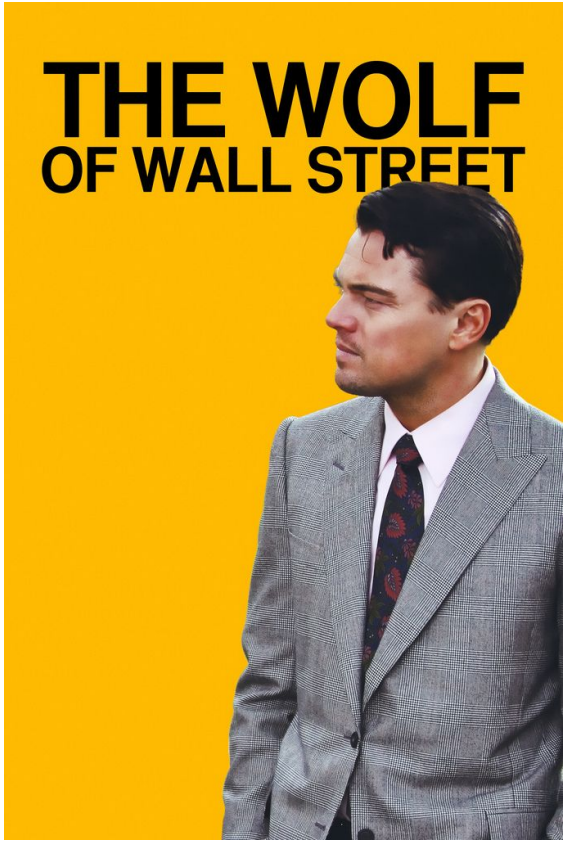
Live-code your own Stock Predictor using Recurrent Neural Nets

Why are we here?





Disclaimer

I teach Data Science here at Galvanize. I'm more engineer than financial analyst.

I don't trade stocks! Stocks are one type of sequential data you can model with RNNs. **Use what we develop here tonight at your own risk.**

I've been playing with neural nets for about a year. Useful resources:

- Andrej Karpathy's NN course, Karpathy Github
- Ian Goodfellow's Deep Learning book
- Iamtrask's blog
- Christopher Olah's blog
- Jakob Aungiers's blog, Aungiers Github

Github page for this Meetup (all code and this presentation)

Goals for Meetup

Introduce neural nets

From the ground up, live-code a small multilayer perceptron (MLP) neural network using only numpy and Python. (45 min)

Change the MLP into a recurrent neural network (RNN) and learn a sine wave. (30 min)

Detour - play with a text generating RNN. Make a new Dr. Seuss book. (15 min)

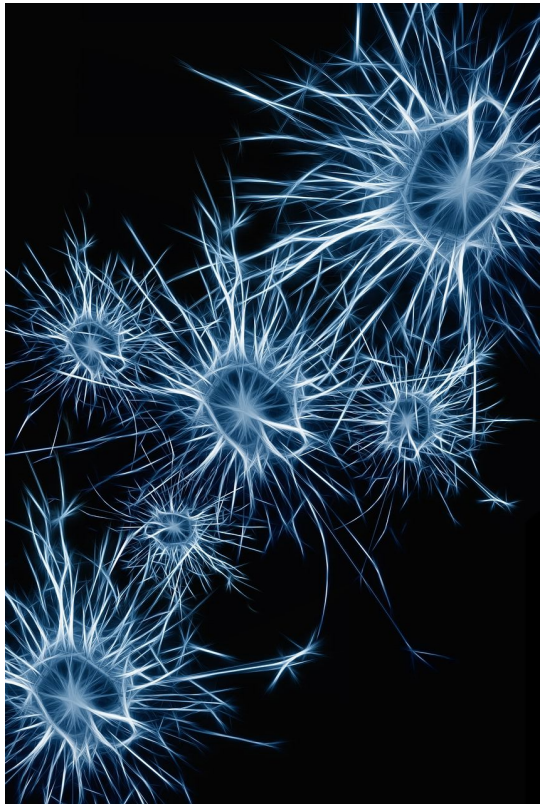
Switch over to Keras and use LSTM layers to learn the same sine wave, then predict stock price. (15 min)

Neural Networks

Models based on the connection of simple computational units, loosely analogous to neurons in the human brain.

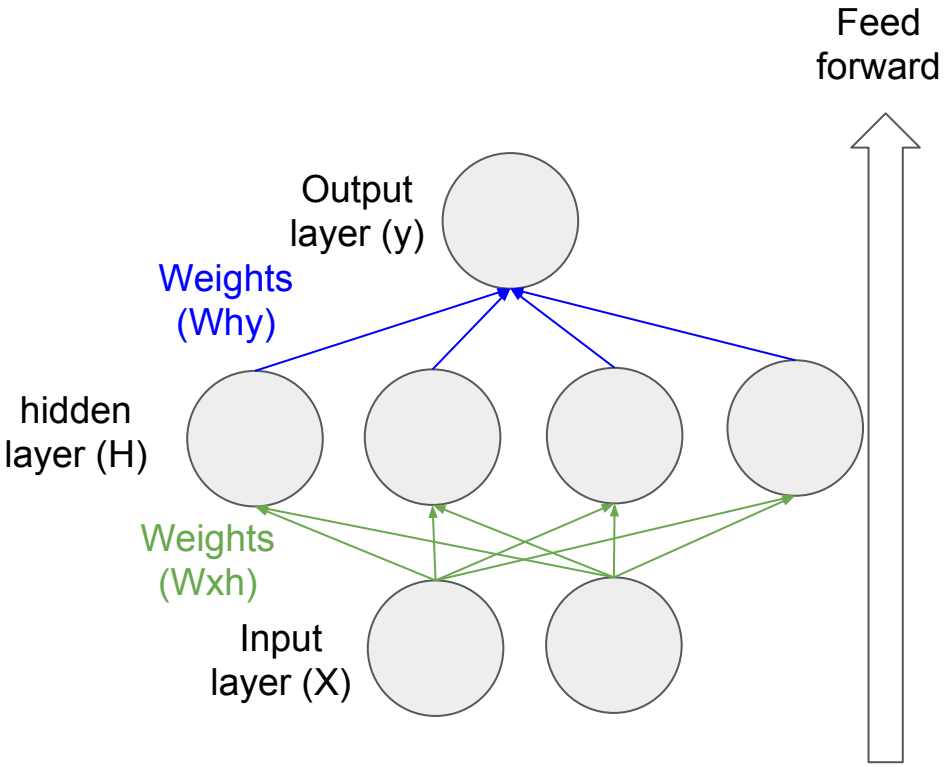
Data availability, computational power, and algorithm development have contributed to their resurgence.

Some use cases:
Pattern recognition: handwriting, captioning images
Text analysis: describe the linguistic context of words
Sequential data: stock market, generating text and articles and click-bait headlines



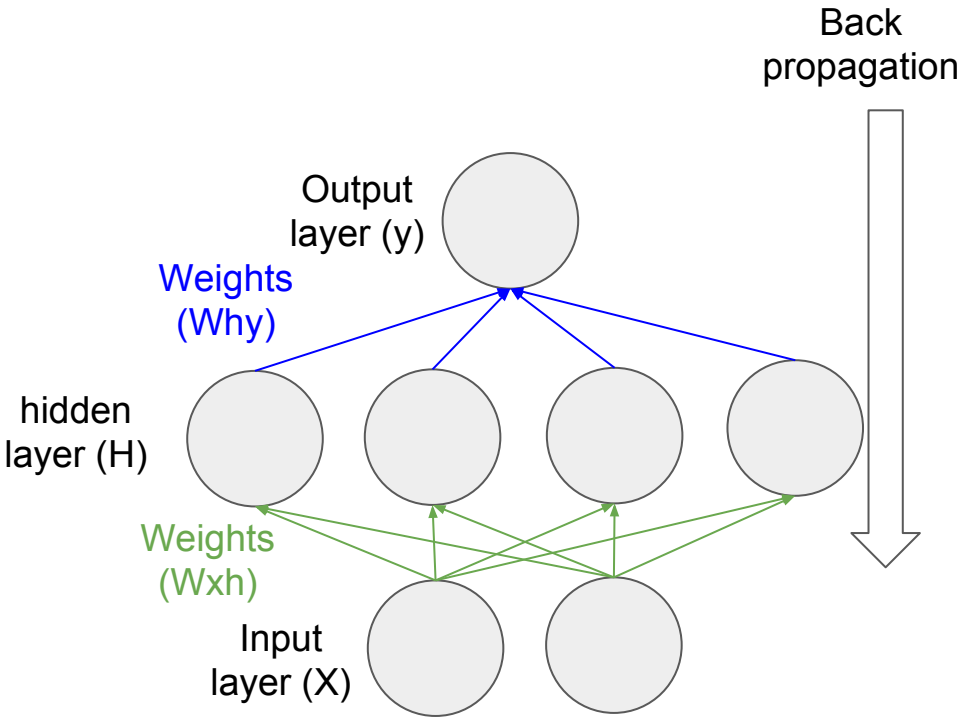
Multilayer perceptron (vanilla neural network)

- Maps input data to corresponding outputs. Calculation *feeds forward* through the network.
- Nodes are arranged in layers.
- Nodes have values for any input given the sum of inputs to the node and an *activation function* that transforms the sum to a non-linear output.
- The “learning” in the network is held by the trained values of the connections (the weights). These weights start with random values.



Multilayer perceptron - learning the weights

- After feed forward yields a predicted output (yp), its difference (loss) is calculated from the real output (y).
- Back propagation estimates how much the loss varies due to each weight in the network (the gradient).
- Gradient descent uses the gradient and learning rate to tweak each of the weights so that the network predicts a little better next time.
- When the total loss reaches an acceptable level, stop. Now it's a trained network ready to predict on new data.



MLP - let's get to coding

Build a simple MLP that predicts the correct output for three logic gates: AND, OR, and XOR. Solve one gate before moving on to the next.

AND

INPUT		OUTPUT
A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

OR

INPUT		OUTPUT
A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

XOR

INPUT		OUTPUT
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

MLP pseudo code - let's get to coding in mlp.py

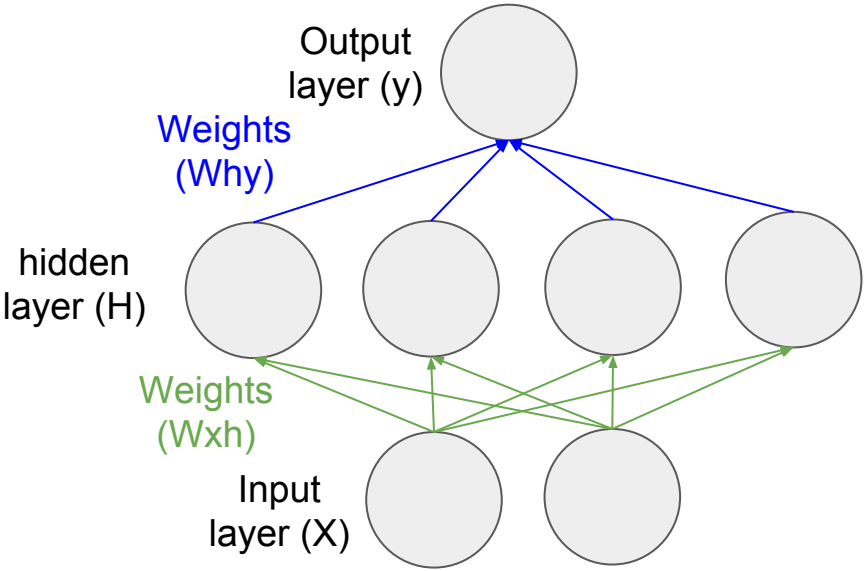
```
# training pseudo code
# for each epoch:
    # for each row of X, y in inputs, targets
        # Feed forward to find values of:
        # H
        # yp (the prediction)

        # Back propogate to find the gradient of the loss with respect to:
        # Why
        # Wxh

        # Use gradient descent to update the weights

    # for this epoch, print training error
```

MLP debrief

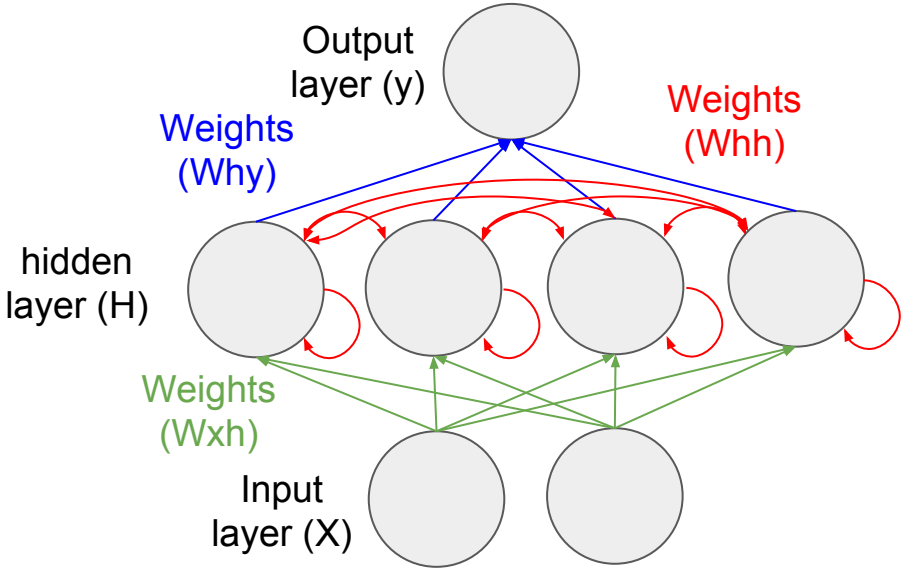
- The MLP was able to perfectly learn each logic gate, even the XOR. (see the XOR affair)
- These inputs, though learned in order with SGD, are not a time series or sequence. One data point does not partially depend on an earlier data point.
- However, in many cases the order of data matters. Think musical notes, order of characters in text, stock market closing price, next point in a sine wave.
- MLPs *can* learn sequential data, but a more elegant solution has emerged: the recurrent neural net.

Comparing the simplest version of these neural nets

Vanilla MLP



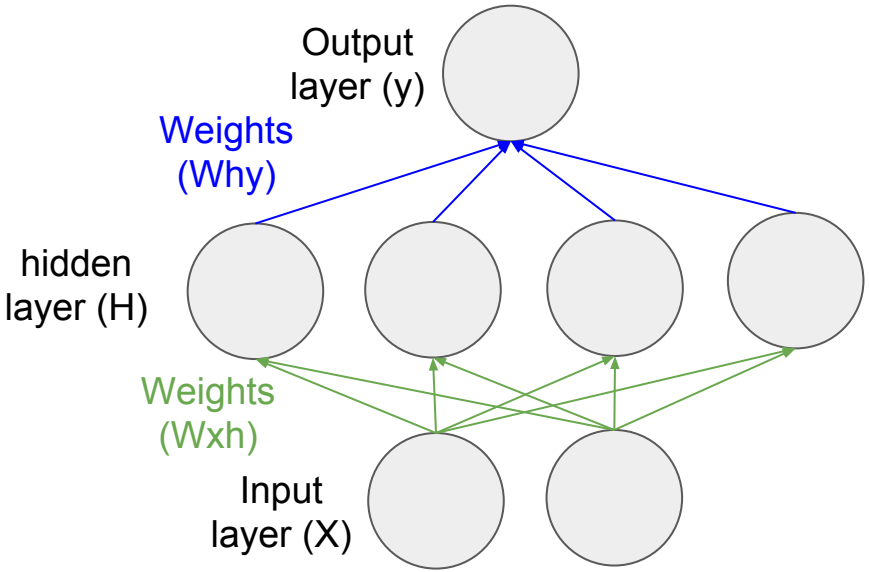
Vanilla RNN



A double arrow indicates a weight in each direction (2 weights).

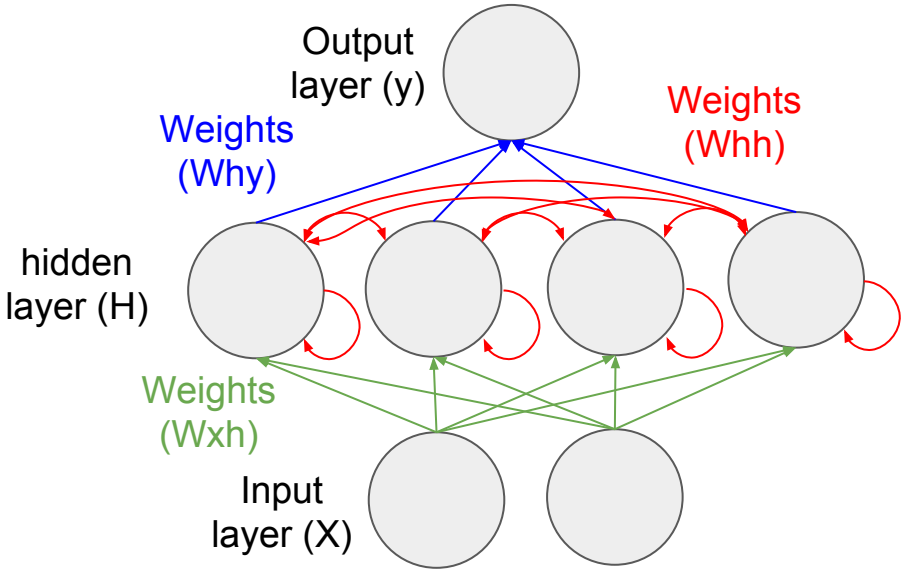
So how many weights in each architecture?

Vanilla MLP



Why =
Wxh =

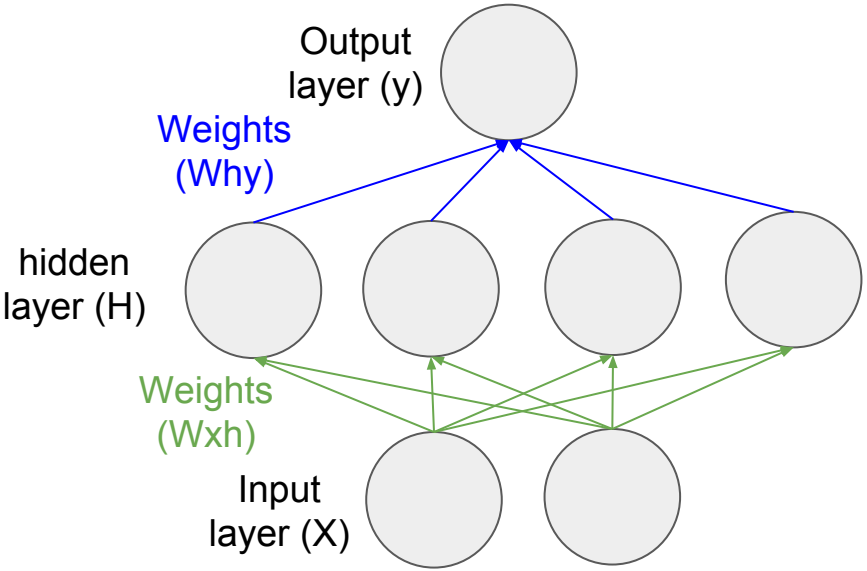
Vanilla RNN



Why =
Whh =
Wxh =

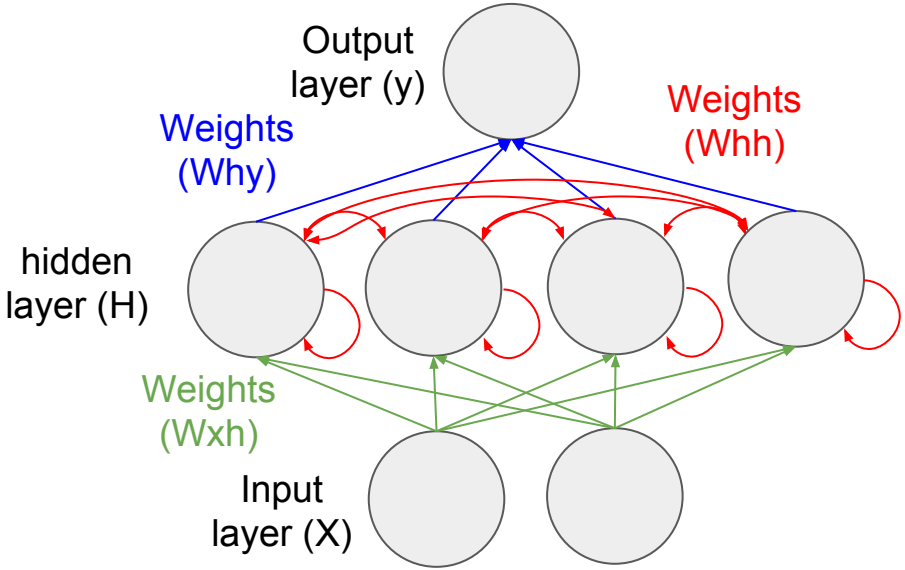
So how many weights in each architecture?

Vanilla MLP



Why = 4
Wxh = 8

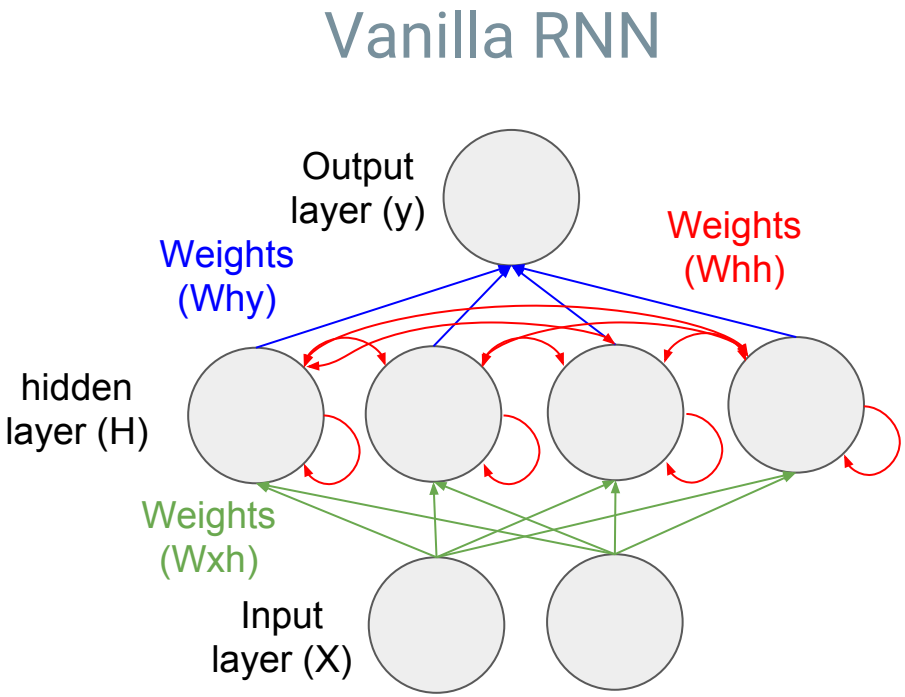
Vanilla RNN



Why = 4
Whh = 16
Wxh = 8

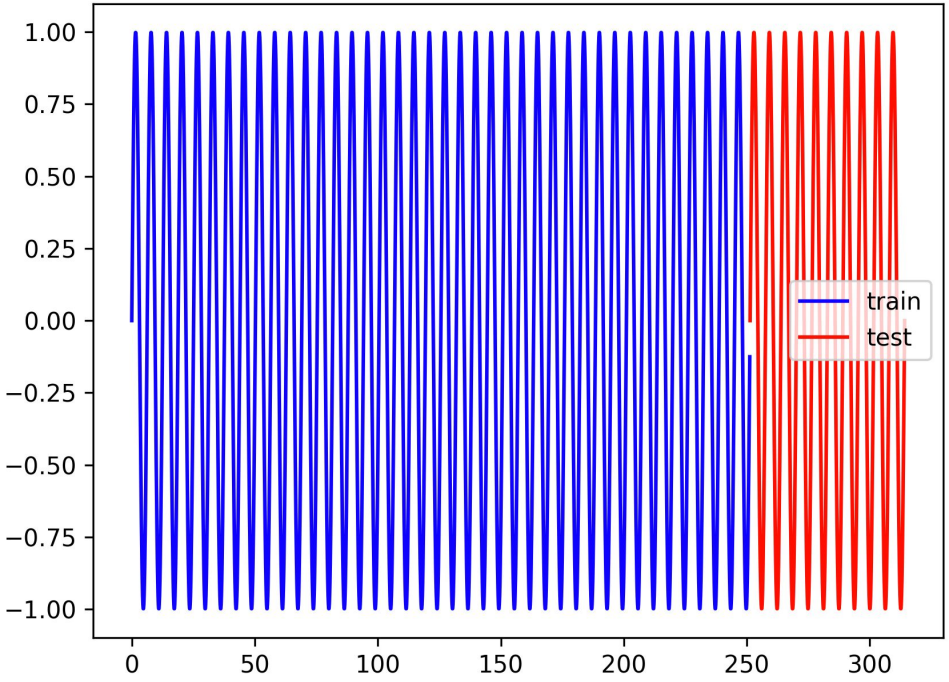
Benefit of the intra - layer recurrent connections

- The previous state of a node in a recurrent hidden layer (H_prev for coding purposes) can affect the value of itself or other nodes in the layer for the next prediction.
- This gives the net the ability to model sequential data.
- Feed forward and back propagation work the same way.
- Learn Whh like all the other weights. In a trained model all the weights are fixed. It's the activations of the nodes that changes with changes in sequence.



RNN - let’s get back to coding

Building with what you’ve learned in mlp.py, start with rnn.py and make it recurrent. Use rnn.py to predict a sine wave.

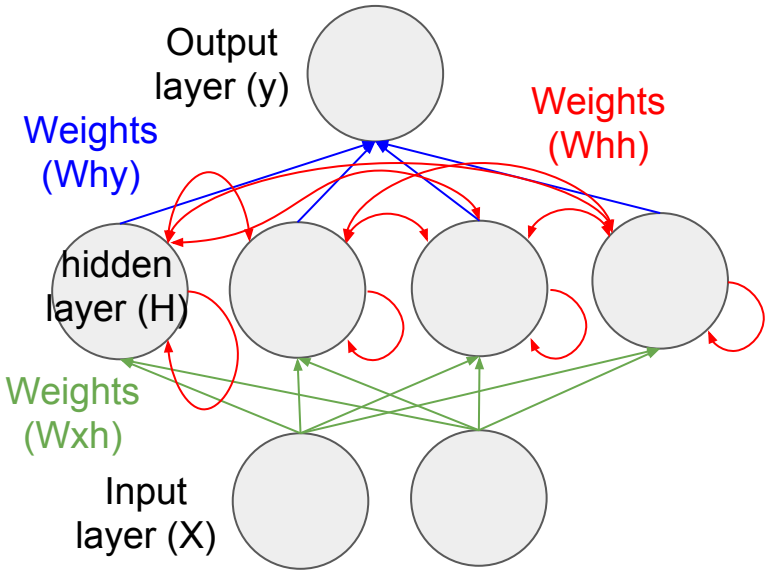


RNN - text is sequential data

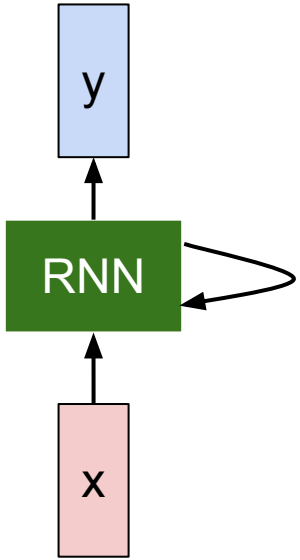
Use the min-char-rnn.py code to learn Dr. Seuss. As the model trains it will eventually write some new books!

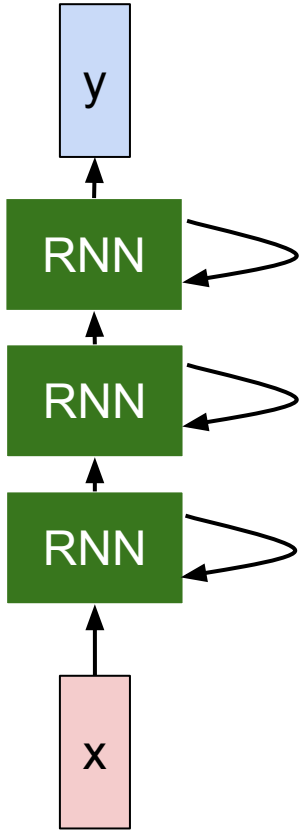
Moving into multilayer RNNs

Multiple layers (and more nodes in each layer) allow more difficult sequences to be learned. They are also harder to train. Exploding and vanishing gradients cause convergence problems, too. Let's stop building things from scratch.



=





Keras

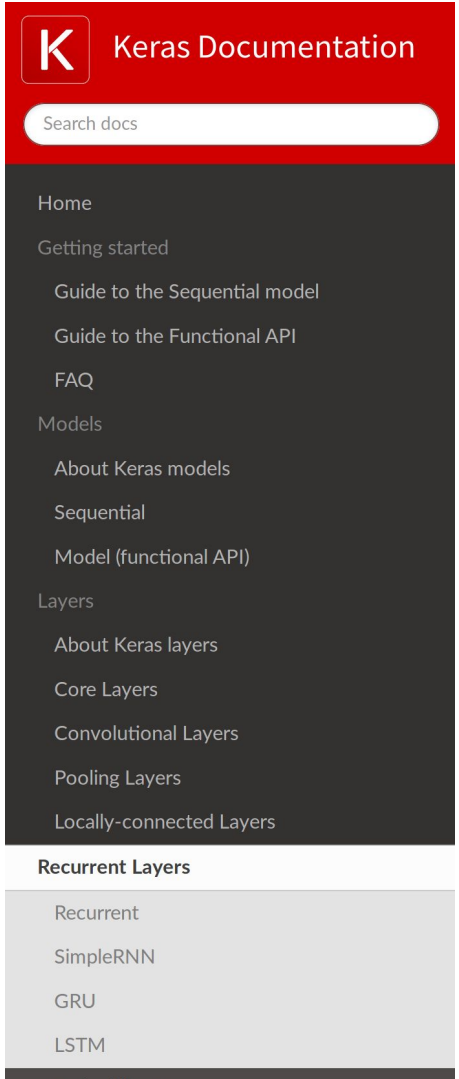
Keras is a high-level neural networks API, written in Python and capable of running on top of either TensorFlow, CNTK or Theano.

We use it in the DSI for capstone projects. MLPs, CNNs, RNNs, some Reinforcement Learning too.

Will become TensorFlow's default API.

Available recurrent layers:

- Recurrent
- SimpleRNN
- Long Short-Term Memory (LSTM)
- Gated Recurrent Unit (GRU)

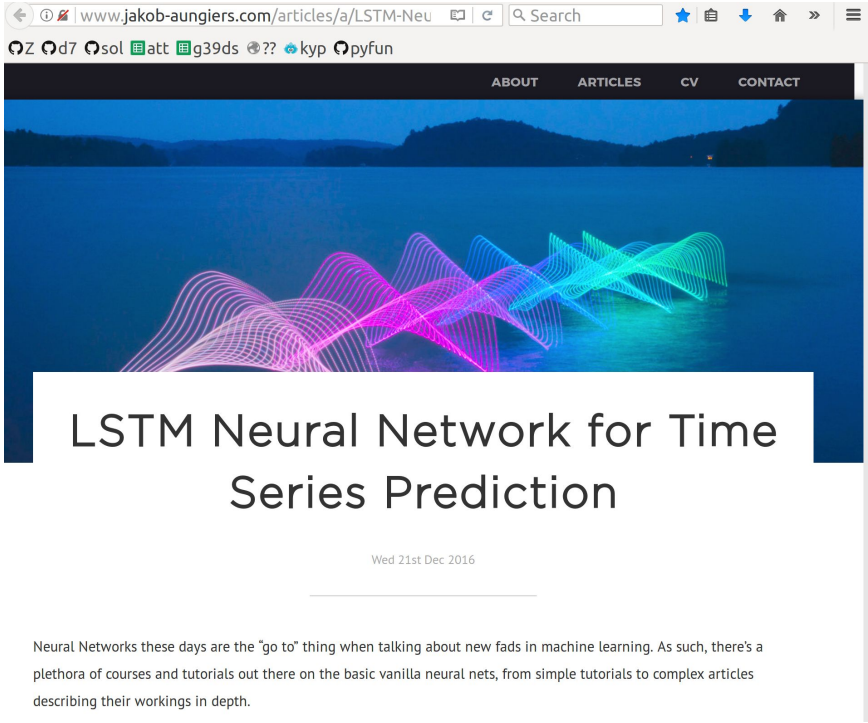


Using Keras LSTM layers to predict stock price

See `lstm.py` and `run_lstm.py` in the Github repo. These files were written by Jakob Aungiers. Check out his blog and Github.

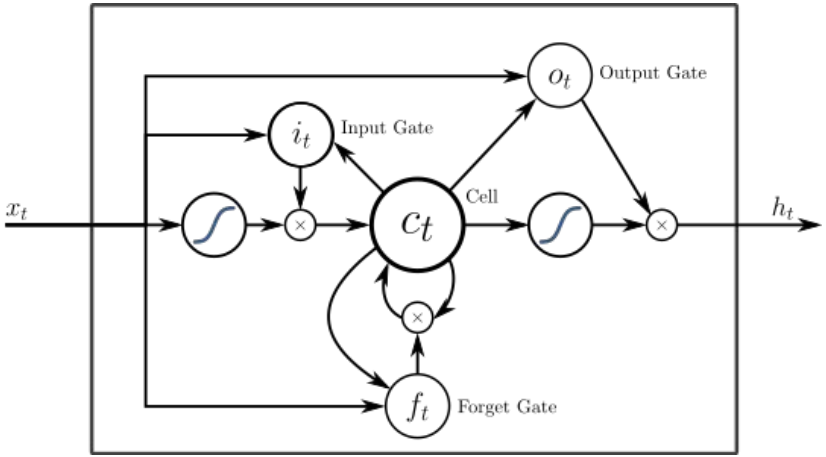
Jakob uses 2 LSTM layers to predict a sine wave (familiar?) and the S&P500.

So what is an LSTM layer?



LSTM

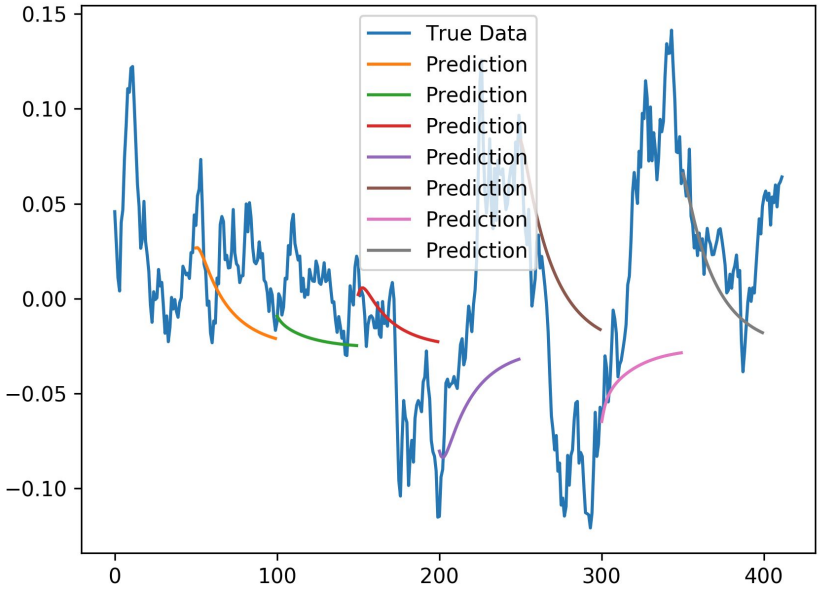
- Long short-term memory (LSTM) is an architecture (an artificial neural network) proposed in 1997.
- LSTM network is well-suited … when there are time lags of unknown size and bound between important events.
- LSTM practical applications: natural language text compression, handwriting recognition, speech recognition, translation.
- I’m still trying to figure out how it works.



Attribute

Play with it

See lstm.py and run_lstm.py in the Github repo. Predict the sine wave, and then the S&P500. Or your own stock!



Thank you!

You can contact me at frank.burkholder@galvanize.com