Recurrent Neural Networks for Time Series Prediction

Frank Burkholder

12 September 2017

A little about me

I teach Data Science at Galvanize.

Background

Ph.D. thesis: “Transition Regime Heat Conduction in Parabolic Trough Receivers”

Worked in concentrating solar thermal power for a decade (NREL and Abengoa)

Present interests:

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

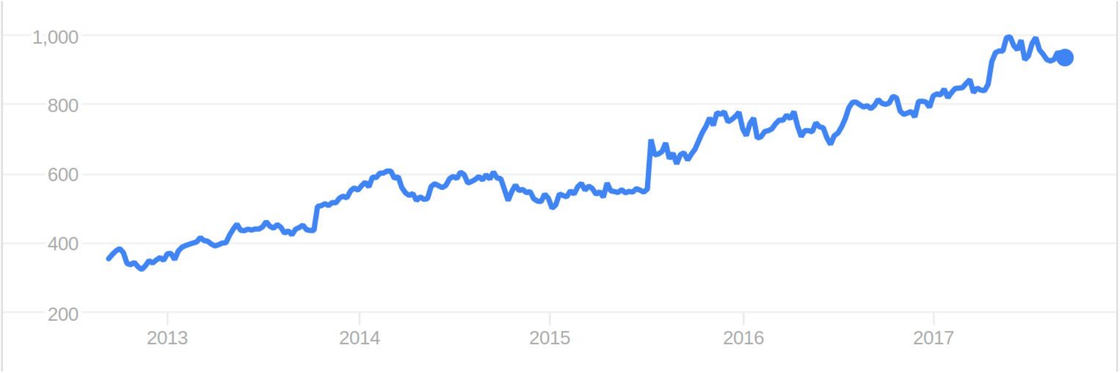
This presentation and source code:
https://github.com/GalvanizeOpenSource/Recurrent_Neural_Net_Meetup

Time series

A series of data points (usually in time).

Data are partly dependent on data that came before.

The order of data points matters.



The quick brown fox jumps over the lazy dog.

The quick brown dog jumps over the lazy fox.

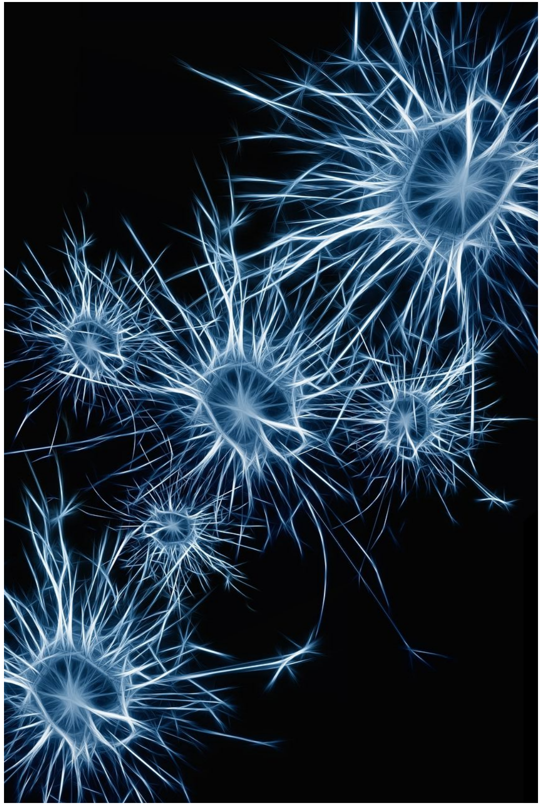
Recurrent Neural Networks (RNNs)

Models based on the connection of simple computational units.

Connections between units can form a directed cycle. This gives a network the ability to maintain **a state based on previous inputs**.

This **state** helps it model time series.

> *They pulled the boat up to the river **bank**.*
>
> *They had been casing the **bank** for months.*



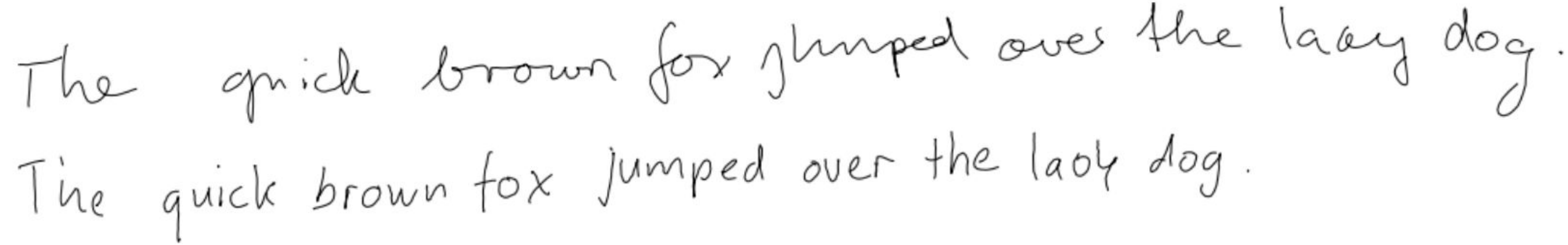
RNN use case - Handwriting generation

recurrent neural network handwriting generation demo

Type a message into the text box, and the network will try to write it out longhand (this paper explains how it works, source code is available here). Be patient, it can take a while!

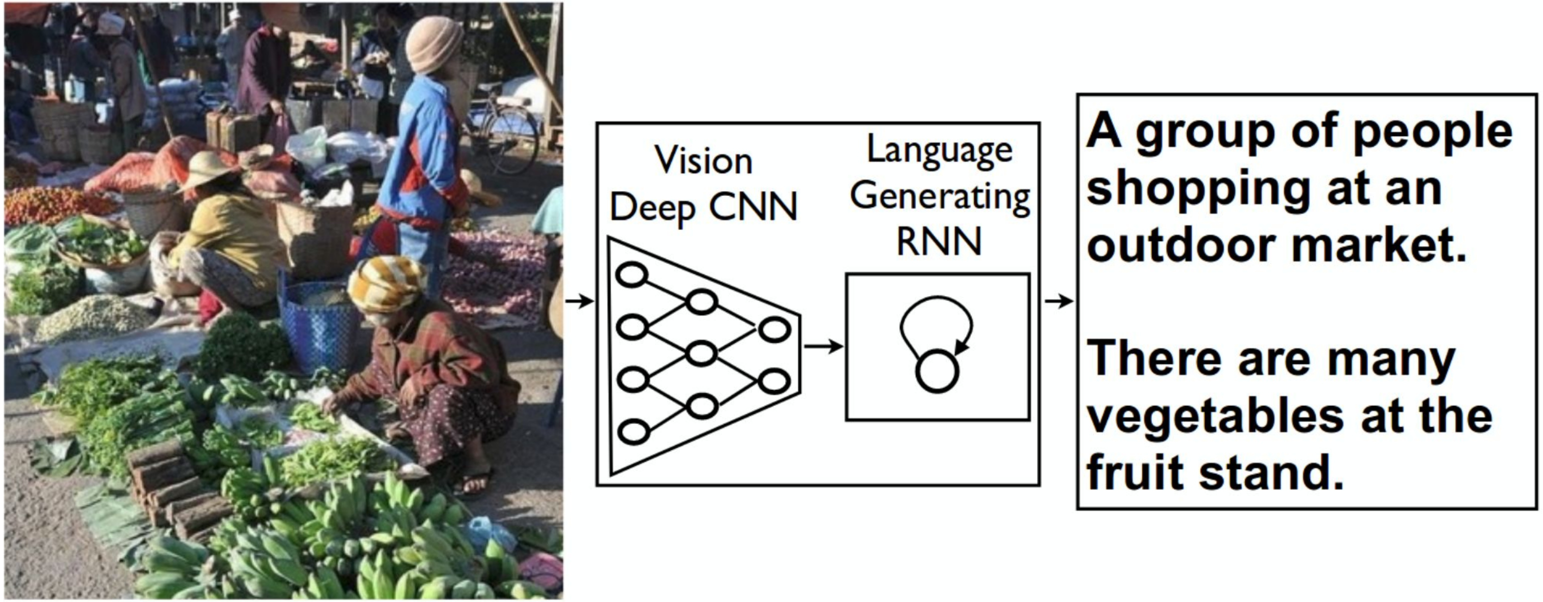
Text --- up to 100 characters, lower case letters work best

The quick brown fox jumped over the lazy dog.



https://www.cs.toronto.edu/~graves/handwriting.html

RNN use case - Image Captioning



Google: Neural Image Caption Generator
https://arxiv.org/pdf/1411.4555.pdf

RNN use case - predicting stock closing price

epochs = 1, window size = 50, sequence shift = 50

http://www.jakob-aungiers.com/articles/a/LSTM-Neural-Network-for-Time-Series-Prediction

Mathematical & code understanding of RNNs

Start with a simple neural network (multilayer perceptron)

- Show how you would code a simple one in Python

Extrapolate from that to a simple recurrent neural network

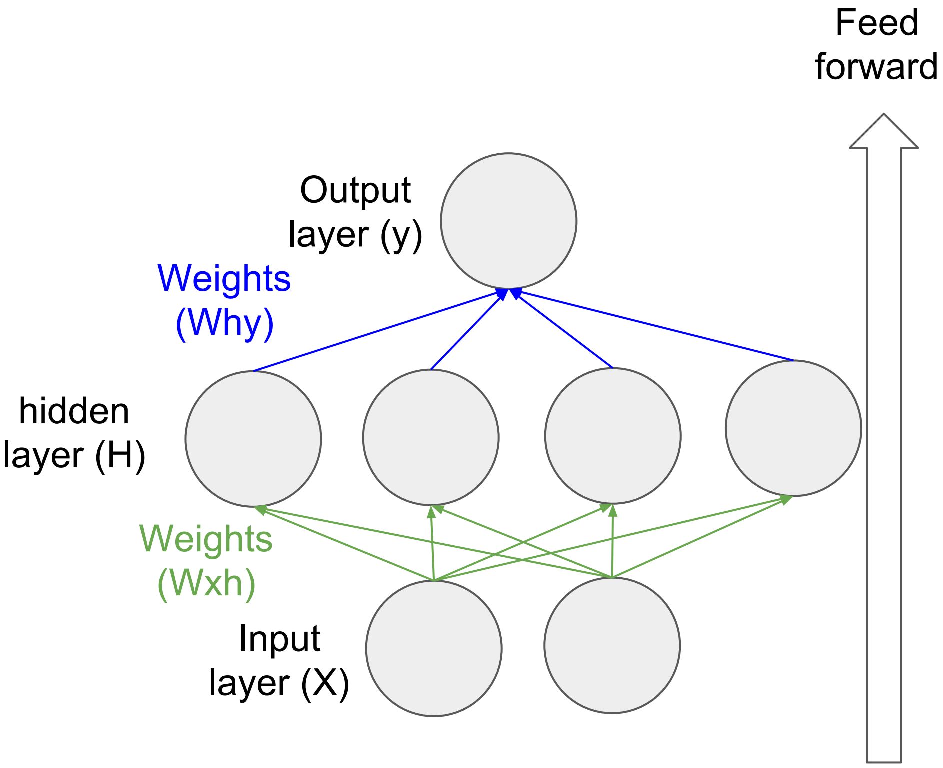
- Show how the code is modified from the MLP

Will use similar code to make a new Dr. Seuss book.

Finish up with LSTMs and stock prediction.

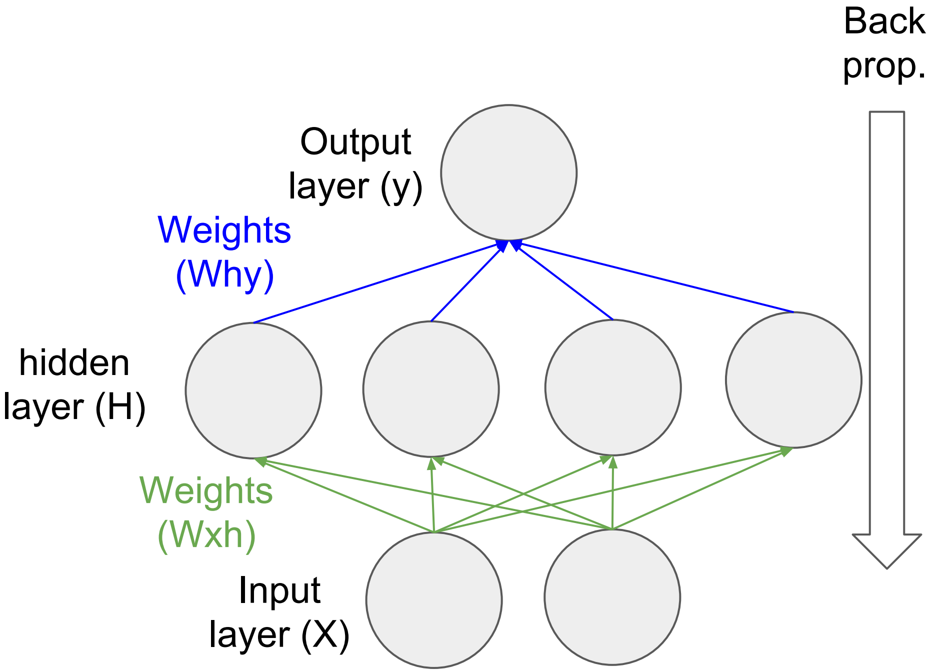
Multilayer perceptron

- 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.

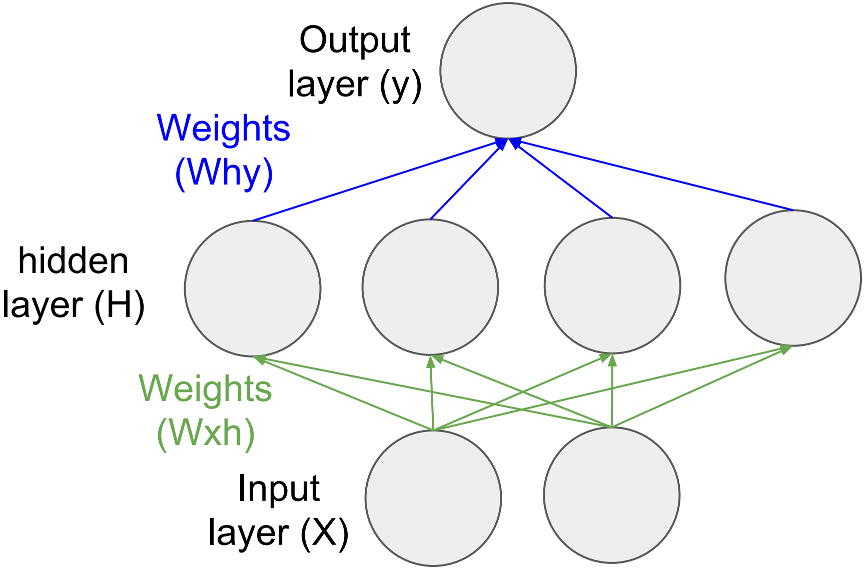


Multilayer perceptron - learning the weights

- After feed forward yields a predicted output ($\mathbf{y}_p$), its difference (loss) is calculated from the real output ($\mathbf{y}$).
- 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 trained and ready to predict.



MLP code



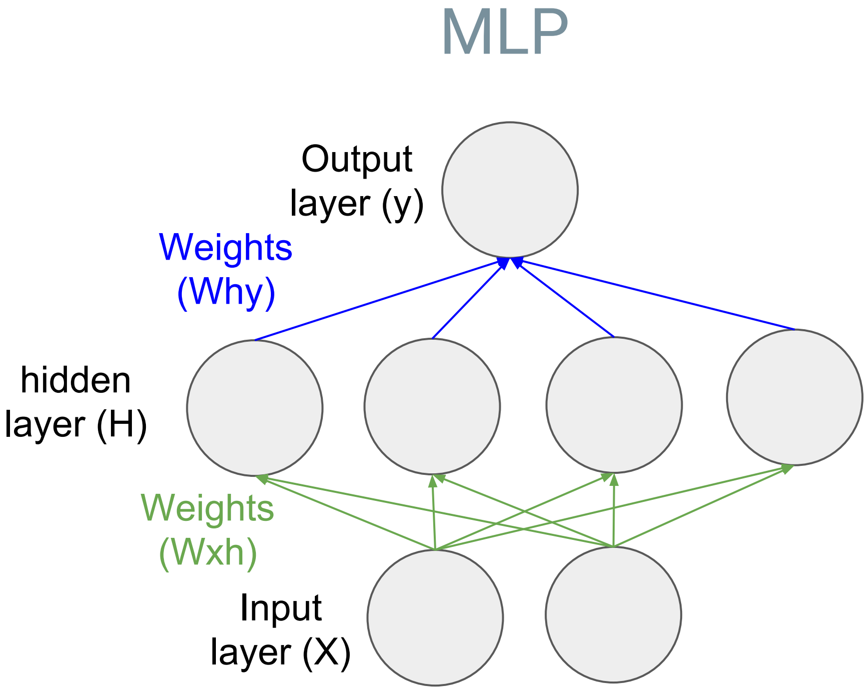
https://github.com/GalvanizeOpenSource/Recurrent_Neural_Net_Meetup/blob/master/mlp_soln.py

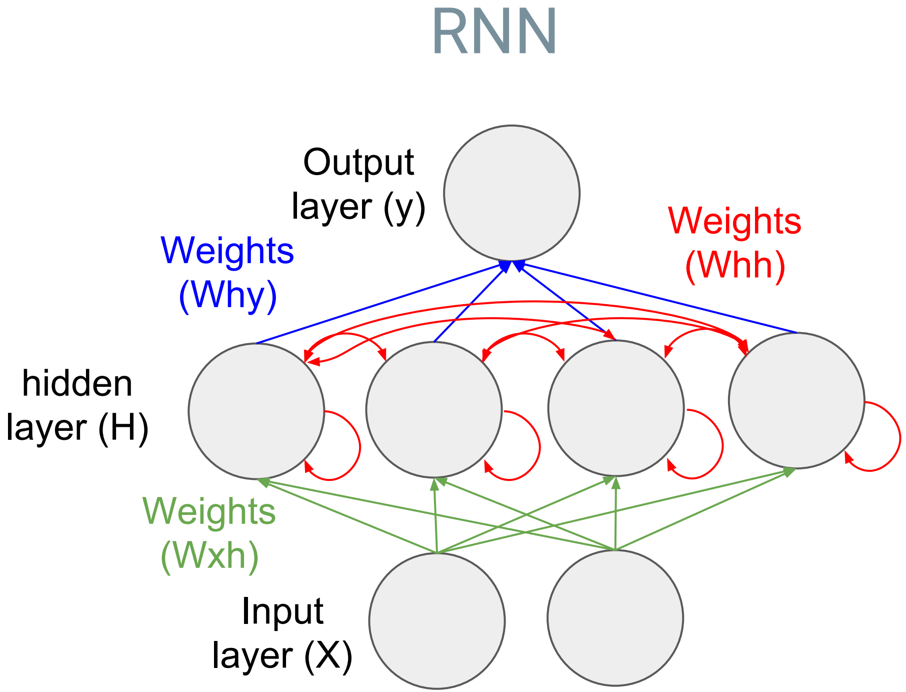
```python
# initialize weights
Wxh = 2*np.random.uniform(size=(nodes_input, nodes_hidden)) - 1
Why = 2*np.random.uniform(size=(nodes_hidden, nodes_target)) - 1

# simulation parameters
np.random.seed(1)
alpha = 1 # learning rate
num_epochs = 20000 # number of epochs

# training
print("\nTraining:")
for e in range(num_epochs):
    yp_lst = [] # predictions
    error_lst = [] # differences between target and predictions
    for X, y in zip(inputs, targets):
        X = X.reshape((1, X.shape[0])) # for row, column shape
        # Feed forward
        H = activation(np.dot(X,Wxh))
        yp = activation(np.dot(H,Why))
        # Back propogate to find gradients
        # Why gradients
        yp_error = y - yp
        yp_delta = yp_error*activation(yp,deriv=True)
        grad_Why = np.dot(H.T, yp_delta)
        # Wxh gradients
        H_error = np.dot(yp_delta, Why.T)
        H_delta = H_error * activation(H,deriv=True)
        grad_Wxh = np.dot(X.T, H_delta)
        # Use gradient descent to update weights
        Why += alpha * grad_Why
        Wxh += alpha * grad_Wxh
```

Comparing the simplest version of these neural nets

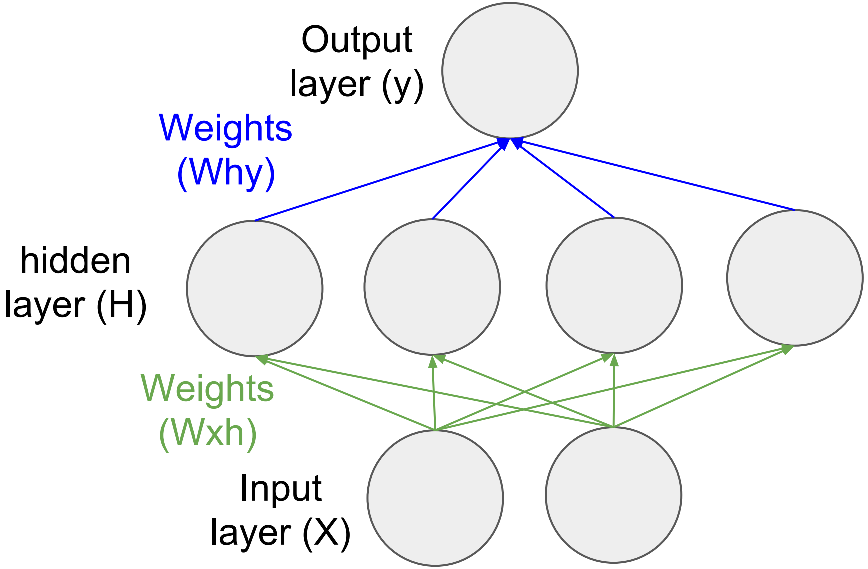




A double arrow indicates a weight in each direction (2 weights).

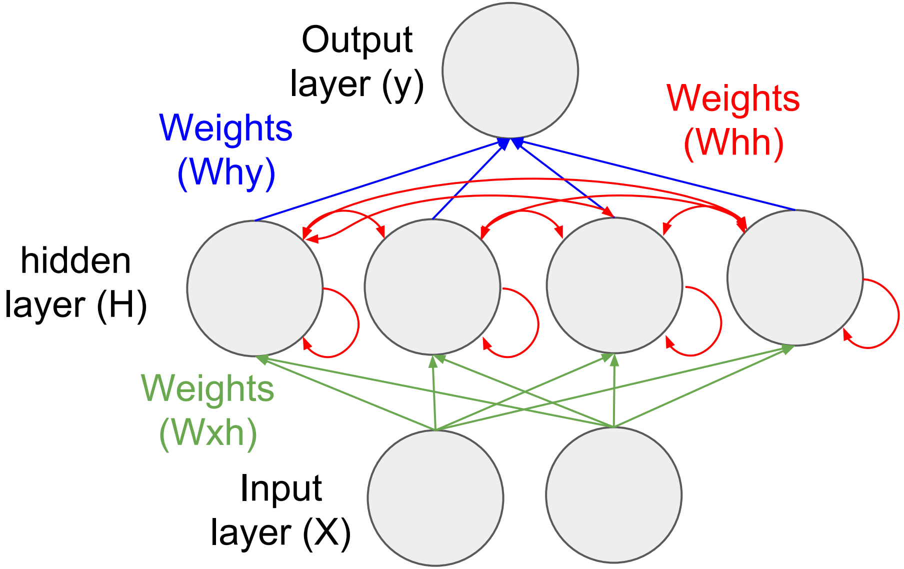
So how many weights in each architecture?

MLP



Why =
Wxh =

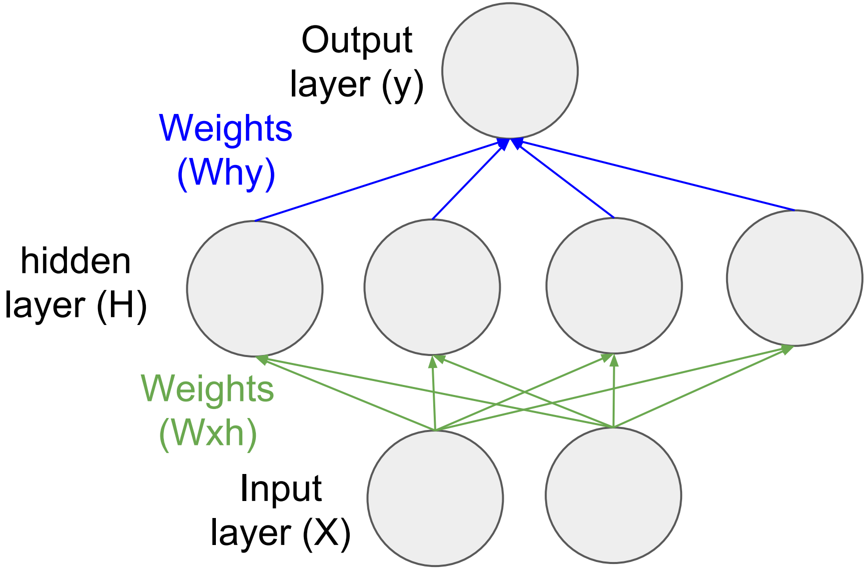
RNN



Why =
Whh =
Wxh =

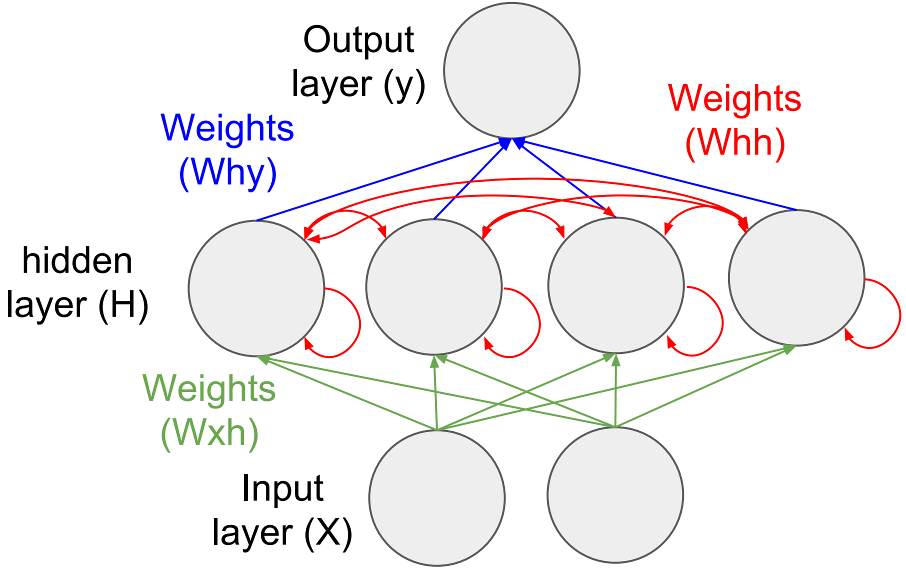
So how many weights in each architecture?

MLP



Why = 4
Wxh = 8

RNN

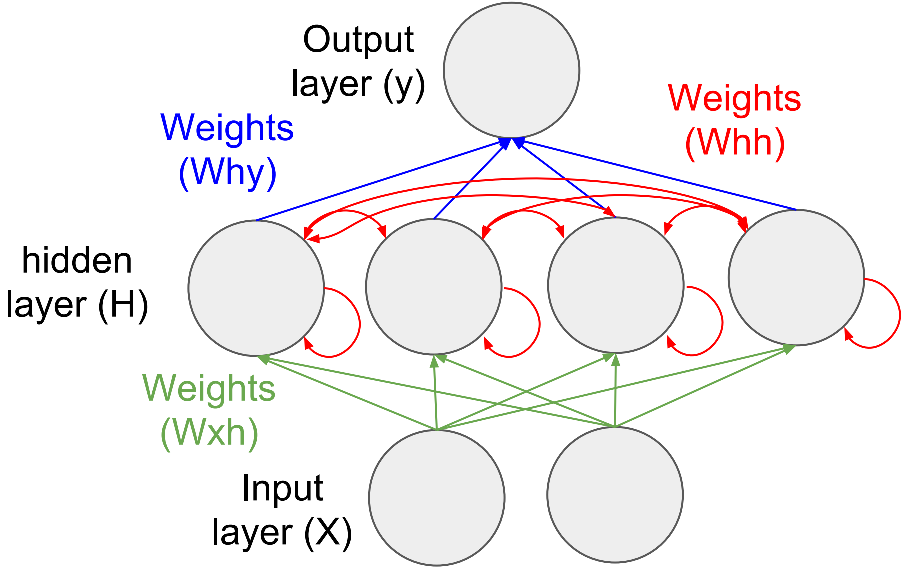


Why = 4
Whh = 16
Wxh = 8

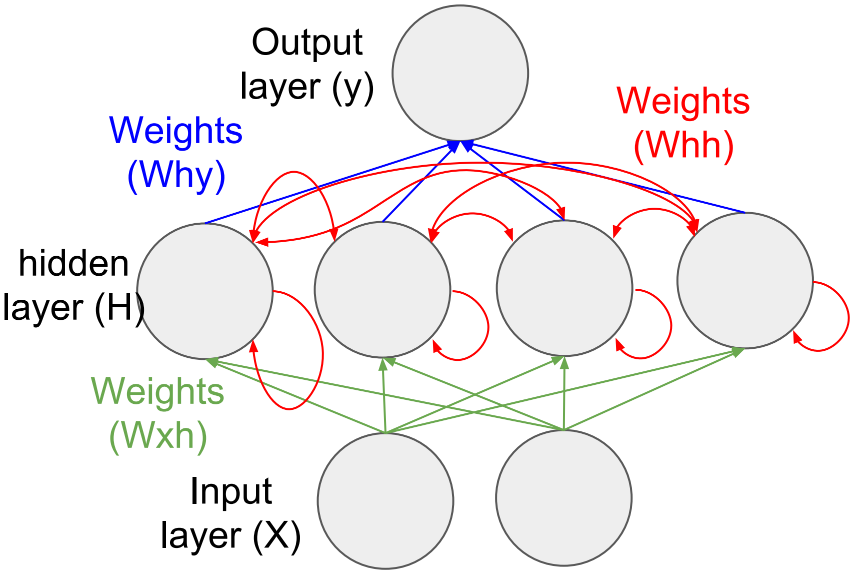
Benefit of the intra - layer recurrent connections

- 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.
- This gives the net the ability to model sequential data.
- Feedforward and backpropagation 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.

Vanilla RNN



RNN code



https://github.com/GalvanizeOpenSource/Recurrent_Neural_Net_Meetup/blob/master/rnn_soln.py

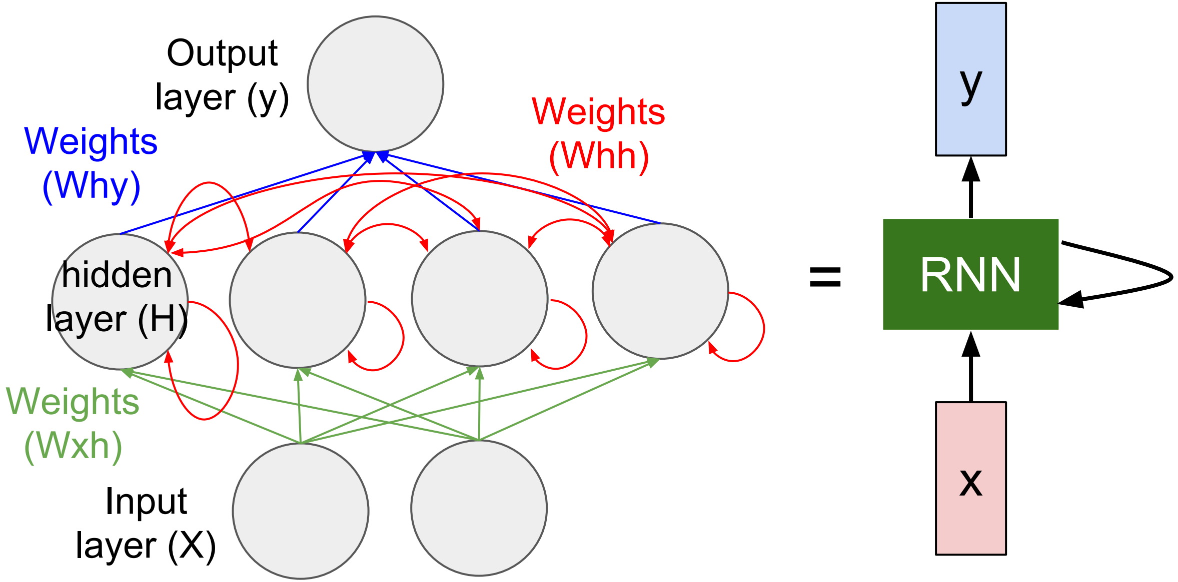
```python
# training
print("\nTraining:")
H_prev = np.zeros((1, nodes_hidden))
H_delta_fut = np.zeros(nodes_hidden)
for e in range(num_epochs):
    error_lst = [] # differences between target and predictions
    for X, y in zip(X_train, y_train):
        X = X.reshape((1, X.shape[0])) # for row, column shape
        y = y.reshape((1,1))
        # Feed forward
        H = activation(np.dot(X,Wxh) + np.dot(H_prev,Whh))
        yp = activation(np.dot(H,Why))
        # Back propogate to find gradients
        # Why gradients
        yp_error = y - yp
        yp_delta = yp_error*activation(yp,deriv=True)
        grad_Why = np.dot(H.T, yp_delta)
        # Wxh gradients
        H_error = np.dot(yp_delta, Why.T) + np.dot(H_delta_fut, Whh.T)
        H_delta = H_error * activation(H,deriv=True)
        #H_delta_fut = np.copy(H_delta) crashes simulation
        grad_Wxh = np.dot(X.T, H_delta)
        # Whh gradients
        grad_Whh = np.dot(H_prev.T, H_delta)
        # Use gradient descent to update weights
        Why += alpha * grad_Why
        Whh += alpha * grad_Whh
        Wxh += alpha * grad_Wxh
        # save for future use
        H_prev = np.copy(H)
```

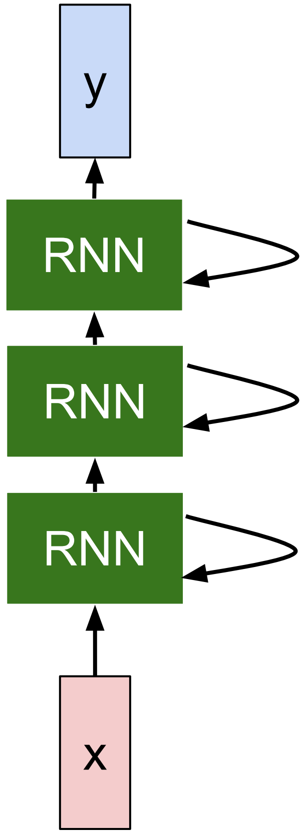
RNN - text is sequential data

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.





Keras

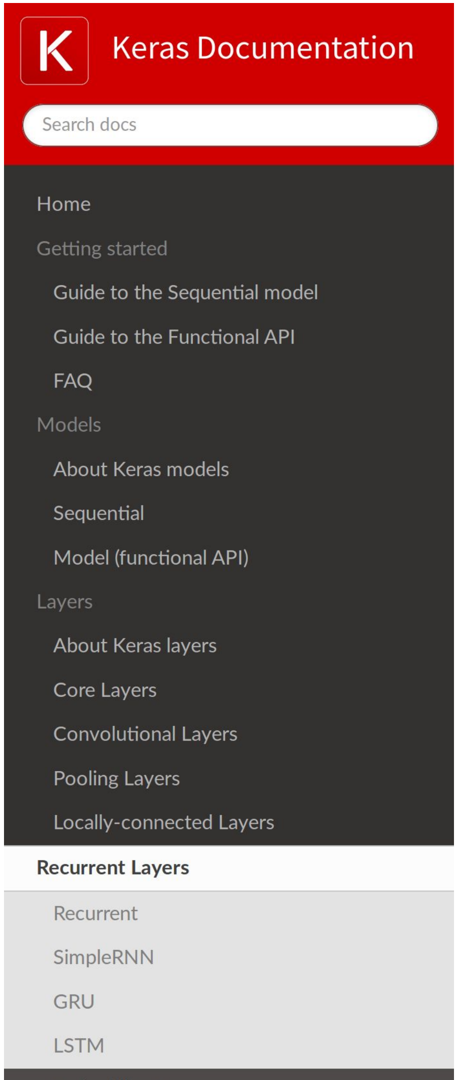
[Keras](#) is a high-level neural networks API, written in Python and capable of running on top of either TensorFlow, CNTK or Theano.

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

[An API for TensorFlow](#)

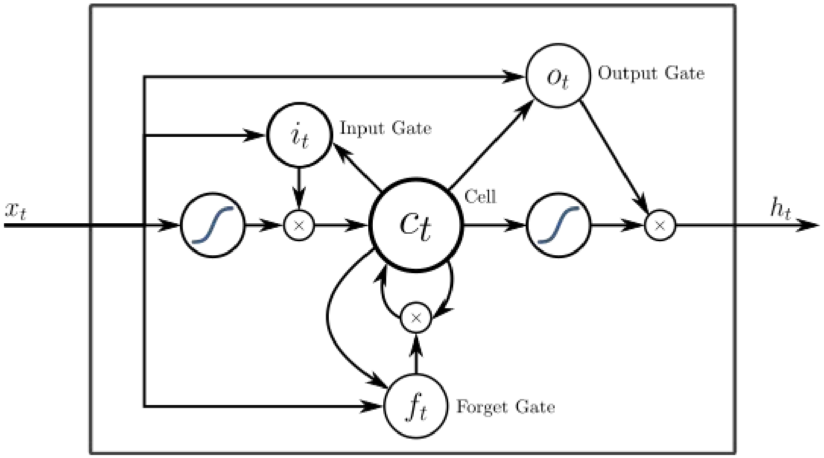
Available recurrent layers:

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



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. (See Wikipedia)
- Addresses the vanishing gradient problem.
- Christopher Olah's blog to gain insight.

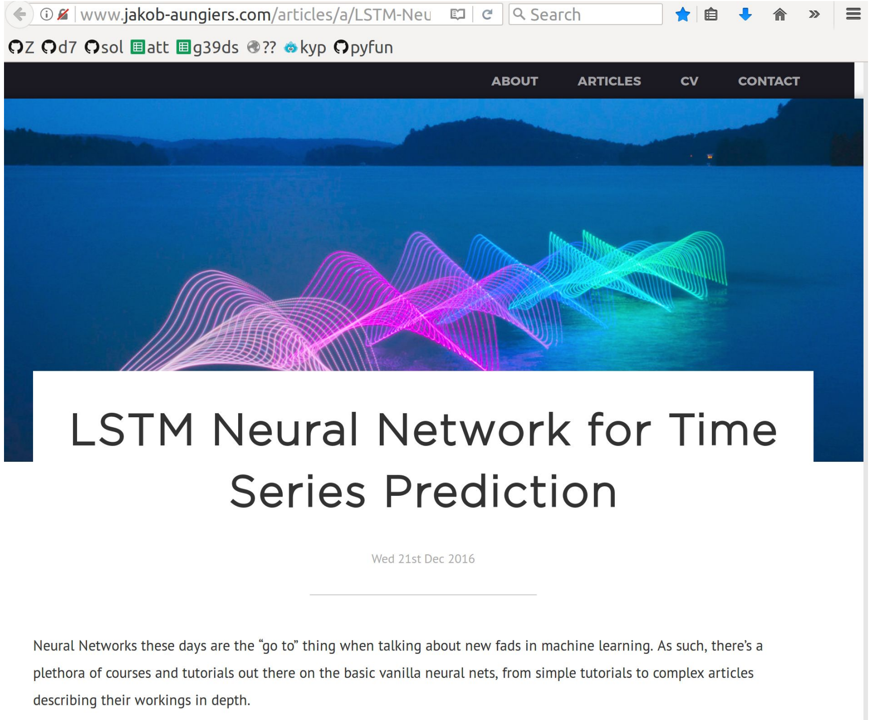


Attribute

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 the S&P500.



Code and results

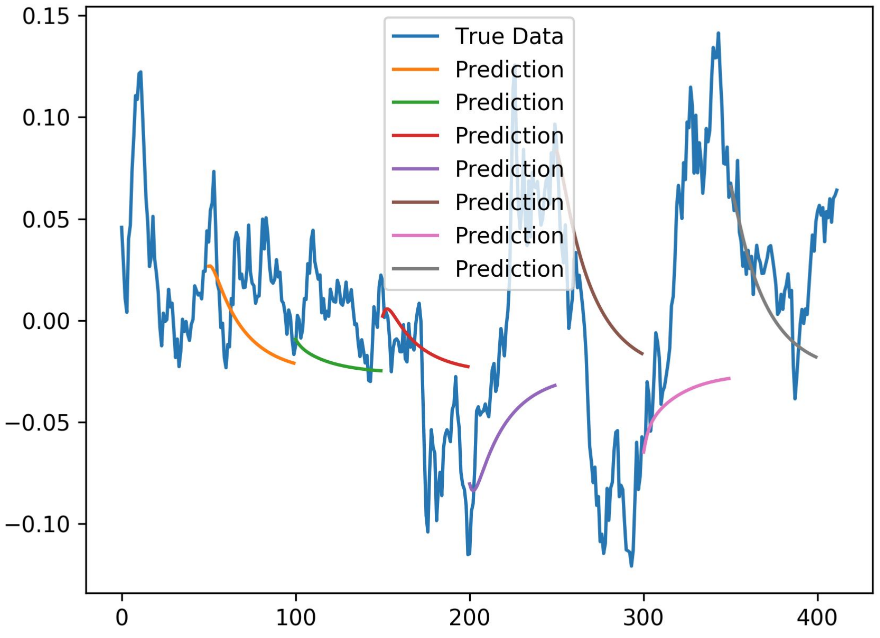
```python
def build_model(layers):
    model = Sequential()

    model.add(LSTM(
        input_shape=(layers[1], layers[0]),
        output_dim=layers[1],
        return_sequences=True))
    model.add(Activation("tanh"))
    model.add(Dropout(0.2))

    model.add(LSTM(
        layers[2],
        return_sequences=False))
    #model.add(Activation("tanh"))
    model.add(Dropout(0.2))

    model.add(Dense(
        output_dim=layers[3]))
    model.add(Activation("linear"))

    start = time.time()
    model.compile(loss="mse", optimizer="rmsprop")
    print("> Compilation Time : ", time.time() - start)
    return model
```



https://github.com/GalvanizeOpenSource/Recurrent_Neural_Net_Meetup/blob/master/lstm.py

Thank you!

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