Applying Deep Learning to Timer Series Forecasting with TensorFlow

https://mapr.com/blog/deep-learning-tensorflow/

Editor's Note: *This is the fourth installment in our blog series about deep learning. In this series, we will discuss the deep learning technology, available frameworks/tools, and how to scale deep learning using big data architecture. Read Part 1, Part 2, and Part 3.*

Time series analysis has significance in econometrics and financial analytics but can be utilized in any field, where understanding trends is important to decision making and reacting to changes in behavioral patterns. For example, a MapR Converged Data Platform customer, who is a major oil and gas provider, places sensors on wells, sending data to MapR Streams that is then used for trend monitoring well conditions, such as volume and temperature. In finance, time series analytics is used for financial forecasting for stock prices, assets, and commodities. Econometricians have long leveraged “autoregressive integrated moving average” (ARIMA) models to perform univariate forecasts.

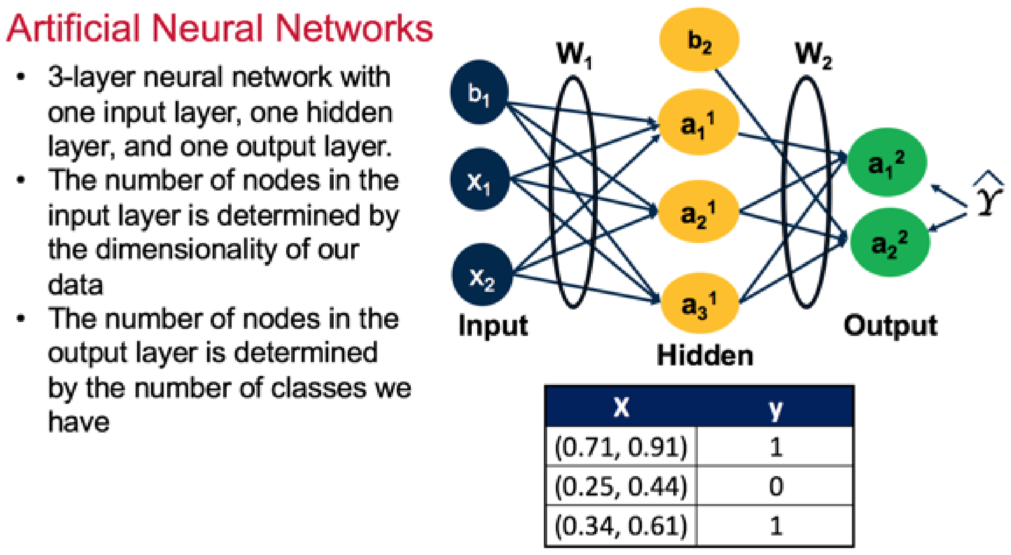
ARIMA models have been used for decades and are well understood. However, with the rise of machine learning and, more recently, deep learning, other models are being explored and utilized, either to support ARIMA results or replace them.

Deep learning (DL) is a branch of machine learning based on a set of algorithms that attempts to model high-level abstractions in data by using artificial neural network (ANN) architectures composed of multiple non-linear transformations. One of the more popular DL deep neural networks is the Recurrent Neural Network (RNN). RNNs are a class of neural networks that depend on the sequential nature of their input. Such inputs could be text, speech, time series, and anything else in which the occurrence of an element in the sequence is dependent on the elements that appeared before it. For example, the next word in a sentence, if someone writes “the grocery…” is most likely to be “store” instead of “school.” In this case, given this sequence, an RNN would likely predict store rather than school.

Artificial Neural Networks

Actually, it turns out that while neural networks are sometimes intimidating structures, the mechanism for making them work is surprisingly simple: stochastic gradient descent. For each of the parameters in our network (such as weights or biases), all we have to do is calculate the derivative of the parameter with respect to the loss, and nudge it a little bit in the opposite direction.

ANNs use a method known as backpropagation to tune and optimize the results. Backpropagation is a two-step process, where the inputs are fed into the neural network via forward propagation and multiplied with (initially random) weights and bias before they are transformed via an activation function. The depth of your neural network will depend on how many transformations your inputs should go through. Once the forward propagation is complete, the backpropagation step measures the error from your final output to the expected output by calculating the partial derivatives of the weights generating the error and adjusts them. Once the weights are adjusted, the model will repeat the process of the forward and backpropagation steps to minimize the error rate until convergence. If you notice how the inputs are aligned in Fig. 1, you will see that this is an ANN with only one hidden layer, so the back propagation will not need to perform multiple gradient descent calculations.



Artificial Neural Networks

- 3-layer neural network with one input layer, one hidden layer, and one output layer.
- The number of nodes in the input layer is determined by the dimensionality of our data
- The number of nodes in the output layer is determined by the number of classes we have

x	y
(0.71, 0.91)	1
(0.25, 0.44)	0
(0.34, 0.61)	1

Figure 1

Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are called recurrent because they perform the same computations for all elements in a sequence of inputs. RNNs are becoming very popular due to their wide utility. They can analyze time series data, such as stock prices, and provide forecasts. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. They can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing (NLP) systems, such as automatic translation, speech-to-text, or sentiment analysis. It can be applied in situations where you have a sequences of “events” with events being a data point.

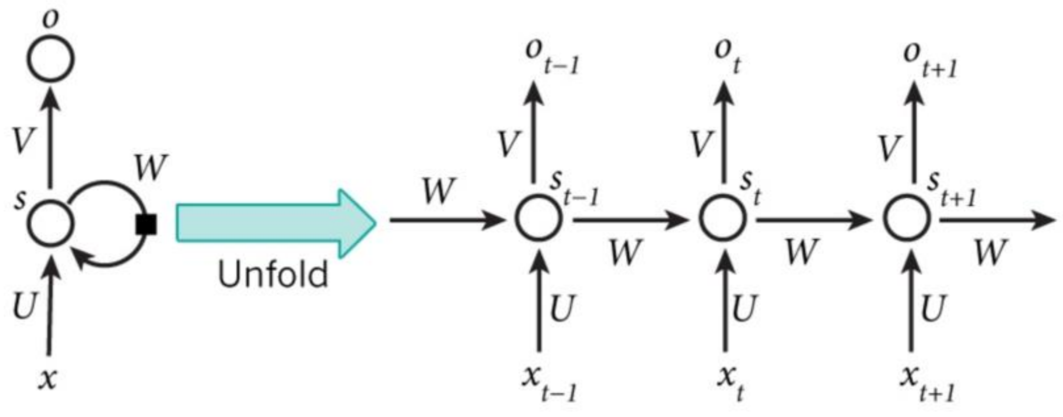


Figure 2

Fig. 2 shows an example of an RNN architecture, and we see xt is the input at time step t. For example, x1 could be the first price of a stock in time period one. st is the hidden state at time step tn and is calculated based on the previous hidden state and the input at the current step, using an activation function. St-1 is usually initialized to zero. ot is the output at step t. For example, if we wanted to predict the next value in a sequence, it would be a vector of probabilities across our time series.

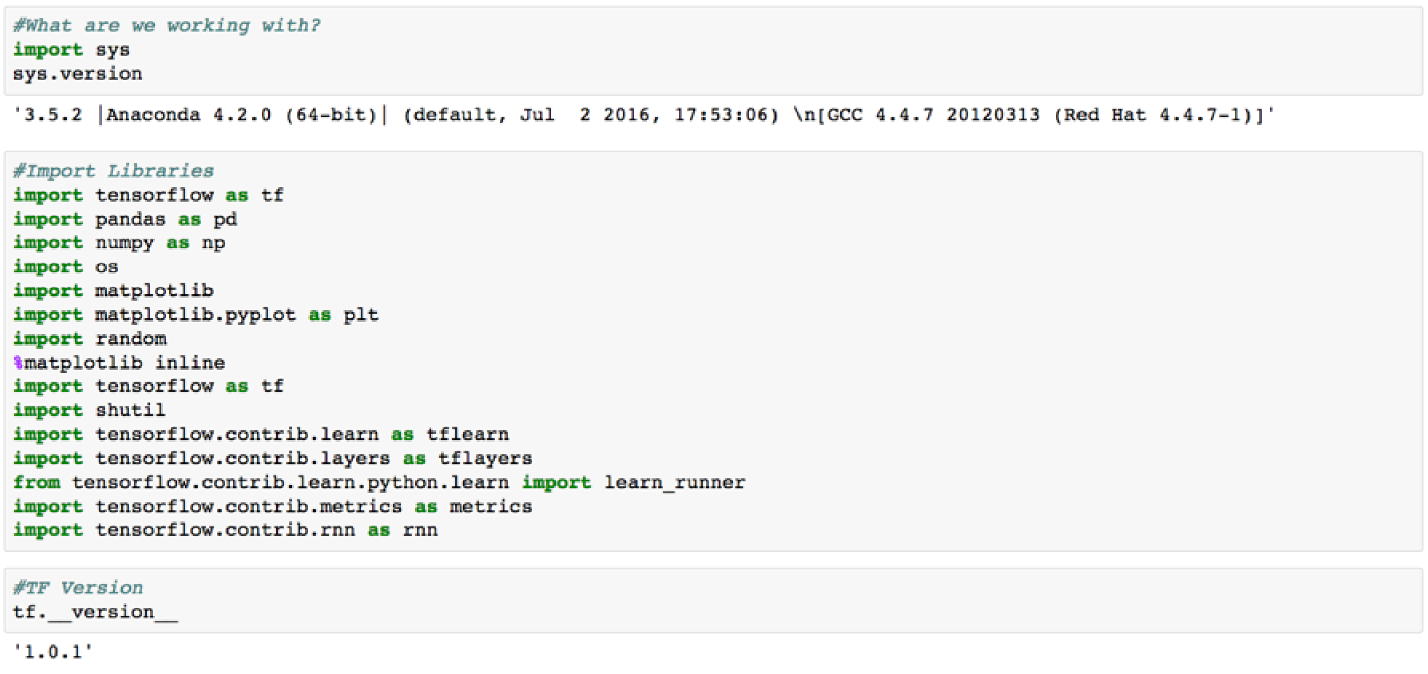
RNN cells are developed on the notion that one input is dependent on the previous input by having a hidden state, or memory, that captures what has been seen so far. The value of the hidden state at any point in time is a function of the value of the hidden state at the previous time step and the value of the input at the current time step. RNNs have a different structure than ANNs and use backpropagation through time (BPTT) to compute the gradient descent after each iteration.

Example

This example was done with a small MapR cluster of 3 nodes. This example will use the following:

- Python 3.5
- TensorFlow 1.0.1
- Red Hat 6.9

If you are using Anaconda, you should be able to install TensorFlow version 1.0.1 on your local machine and Jupyter Notebook. This code will not work with versions of TensorFlow < 1.0. It can be run on your local machine and conveyed to a cluster if the TensorFlow versions are the same or later. Other deep learning libraries to consider for RNNs are MXNet, Caffe2, Torch, and Theano. Keras is another library that provides a python wrapper for TensorFlow or Theano.



```
#What are we working with?
import sys
sys.version
```

```
'3.5.2 |Anaconda 4.2.0 (64-bit)| (default, Jul  2 2016, 17:53:06) \n[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]'
```

```
#Import Libraries
import tensorflow as tf
import pandas as pd
import numpy as np
import os
import matplotlib
import matplotlib.pyplot as plt
import random
%matplotlib inline
import tensorflow as tf
import shutil
import tensorflow.contrib.learn as tflearn
import tensorflow.contrib.layers as tflayers
from tensorflow.contrib.learn.python.learn import learn_runner
import tensorflow.contrib.metrics as metrics
import tensorflow.contrib.rnn as rnn
```

```
#TF Version
tf.__version__
```

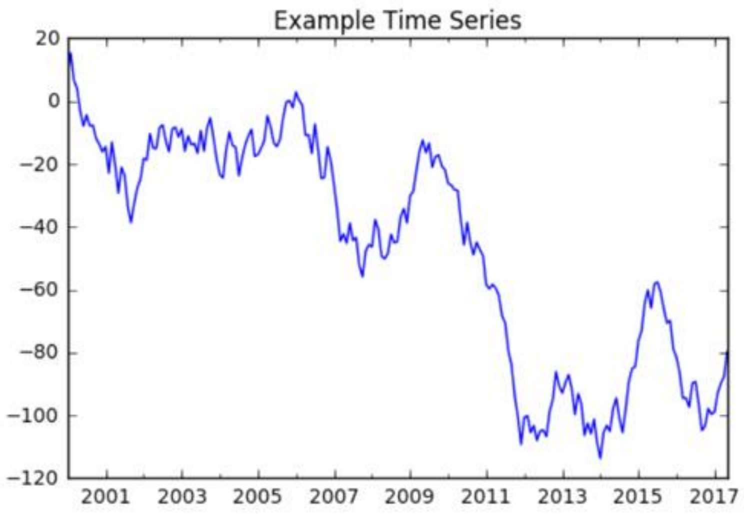
```
'1.0.1'
```

MapR provides the ability to integrate Jupyter Notebook (or Zeppelin) at the user's preference. What we are showing here would be the end of a data pipeline. The true value of running a RNN time series model in a distributed environment is the data pipelines you can construct to push your aggregated series data into a format that can be fed into the TensorFlow computational graph.

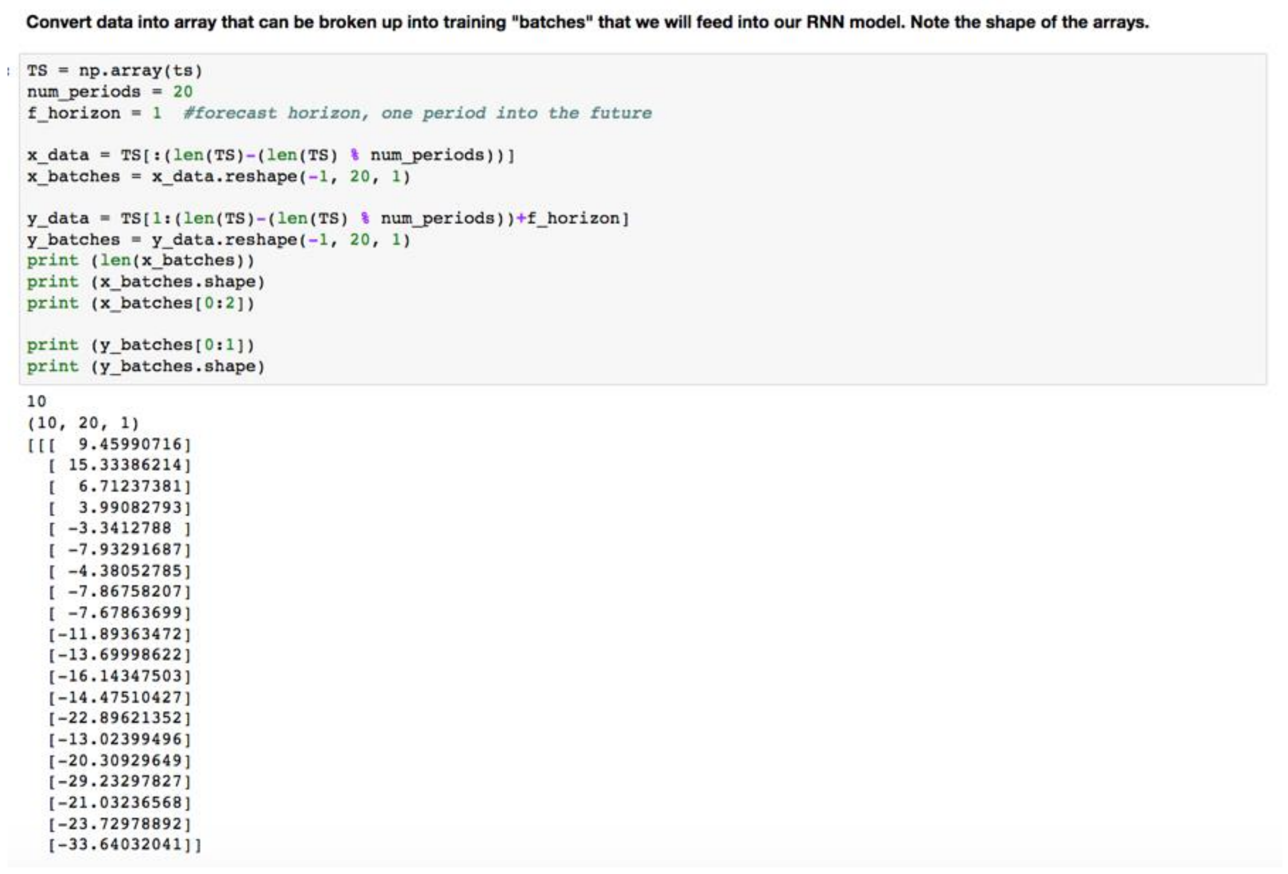
If I am aggregating network flows from multiple devices (IDS, syslogs, etc.), and I want to forecast future network traffic pattern behavior, I could set up a real-time data pipeline using MapR Streams that aggregates this data into a queue that can be fed into my TensorFlow model. For this example, I am using only a single node on my cluster, but I could have installed TensorFlow on the two other nodes and could have three TF models running with different hyper-parameters. For this example, I generated some dummy data.

Generate some data

```
random.seed(111)
rng = pd.date_range(start='2000', periods=209, freq='M')
ts = pd.Series(np.random.uniform(-10, 10, size=len(rng)), rng).cumsum()
ts.plot(c='b', title='Example Time Series')
plt.show()
ts.head(10)
```



```
2000-01-31     9.459907
2000-02-29    15.333862
2000-03-31     6.712374
2000-04-30     3.990828
2000-05-31    -3.341279
2000-06-30    -7.932917
2000-07-31    -4.380528
2000-08-31    -7.867582
2000-09-30    -7.678637
2000-10-31   -11.893635
Freq: M, dtype: float64
```

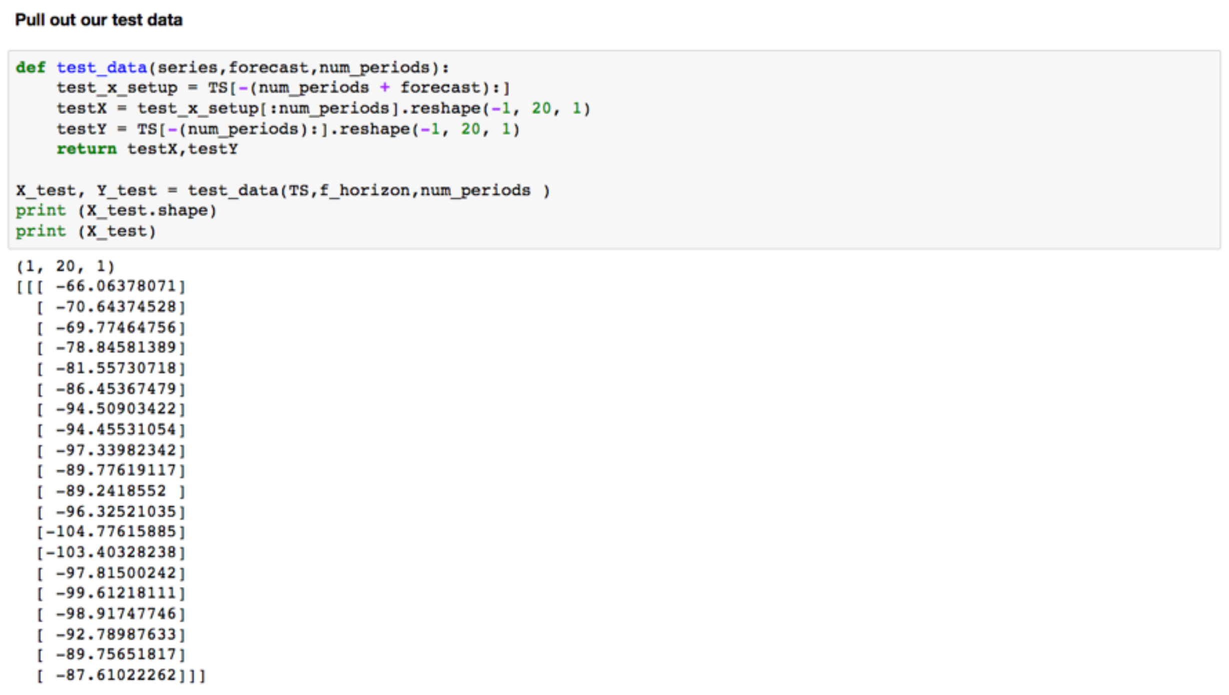

Convert data into array that can be broken up into training "batches" that we will feed into our RNN model. Note the shape of the arrays.

```
TS = np.array(ts)
num_periods = 20
f_horizon = 1  #forecast horizon, one period into the future

x_data = TS[:(len(TS)-(len(TS) % num_periods))]
x_batches = x_data.reshape(-1, 20, 1)

y_data = TS[1:(len(TS)-(len(TS) % num_periods))+f_horizon]
y_batches = y_data.reshape(-1, 20, 1)
print (len(x_batches))
print (x_batches.shape)
print (x_batches[0:2])

print (y_batches[0:1])
print (y_batches.shape)
```

```
10
(10, 20, 1)
[[[  9.45990716]
  [ 15.33386214]
  [  6.71237381]
  [  3.99082793]
  [ -3.3412788 ]
  [ -7.93291687]
  [ -4.38052785]
  [ -7.86758207]
  [ -7.67863699]
  [-11.89363472]
  [-13.69998622]
  [-16.14347503]
  [-14.47510427]
  [-22.89621352]
  [-13.02399496]
  [-20.30929649]
  [-29.23297827]
  [-21.03236568]
  [-23.72978892]
  [-33.64032041]]
```

We have 209 total observations in our data. I want to make sure I have the same number of observations for each of my batch inputs.

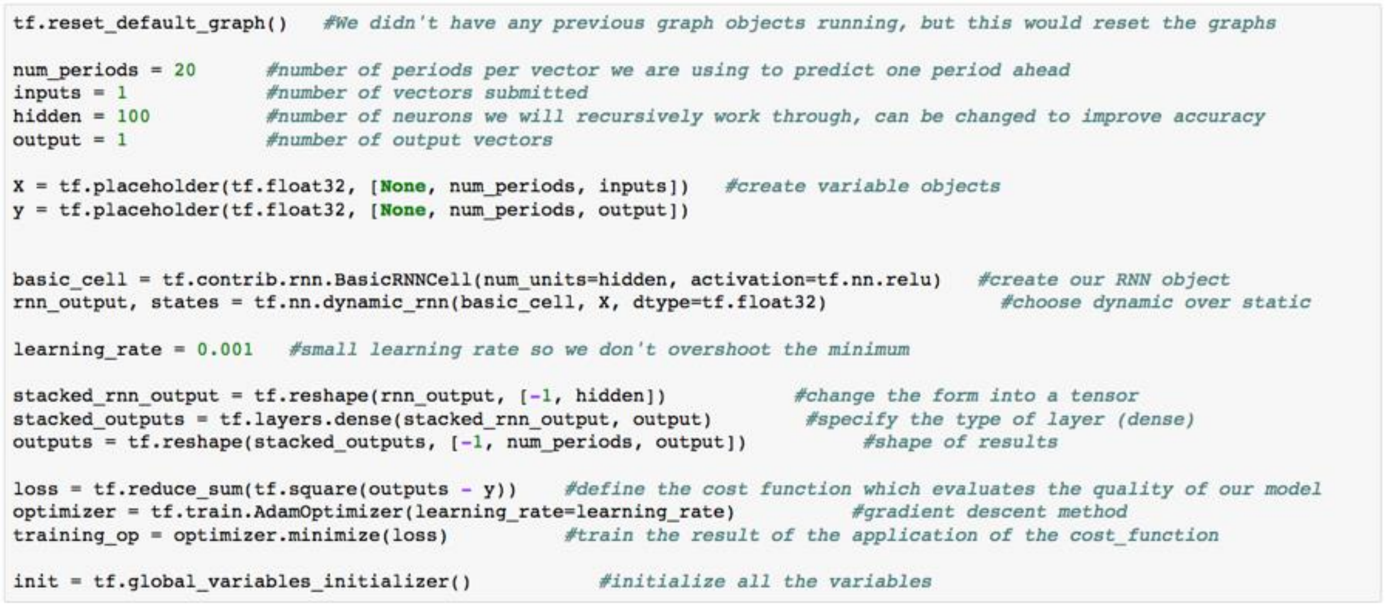
What we see is our training data set is made up of 10 batches, containing 20 observations. Each observation is a sequence of a single value.


Pull out our test data

```
def test_data(series,forecast,num_periods):
    test_x_setup = TS[-(num_periods + forecast):]
    testX = test_x_setup[:num_periods].reshape(-1, 20, 1)
    testY = TS[-(num_periods):].reshape(-1, 20, 1)
    return testX,testY

X_test, Y_test = test_data(TS,f_horizon,num_periods )
print (X_test.shape)
print (X_test)
```

```
(1, 20, 1)
[[[ -66.06378071]
  [ -70.64374528]
  [ -69.77464756]
  [ -78.84581389]
  [ -81.55730718]
  [ -86.45367479]
  [ -94.50903422]
  [ -94.45531054]
  [ -97.33982342]
  [ -89.77619117]
  [ -89.2418552 ]
  [ -96.32521035]
  [-104.77615885]
  [-103.40328238]
  [ -97.81500242]
  [ -99.61218111]
  [ -98.91747746]
  [ -92.78987633]
  [ -89.75651817]
  [ -87.61022262]]]
```

Now that we have our data, let’s create our TensorFlow graph that will do the computation. ^1^



```
tf.reset_default_graph()    #We didn't have any previous graph objects running, but this would reset the graphs

num_periods = 20        #number of periods per vector we are using to predict one period ahead
inputs = 1              #number of vectors submitted
hidden = 100            #number of neurons we will recursively work through, can be changed to improve accuracy
output = 1              #number of output vectors

X = tf.placeholder(tf.float32, [None, num_periods, inputs])    #create variable objects
y = tf.placeholder(tf.float32, [None, num_periods, output])


basic_cell = tf.contrib.rnn.BasicRNNCell(num_units=hidden, activation=tf.nn.relu)   #create our RNN object
rnn_output, states = tf.nn.dynamic_rnn(basic_cell, X, dtype=tf.float32)               #choose dynamic over static

learning_rate = 0.001   #small learning rate so we don't overshoot the minimum

stacked_rnn_output = tf.reshape(rnn_output, [-1, hidden])          #change the form into a tensor
stacked_outputs = tf.layers.dense(stacked_rnn_output, output)        #specify the type of layer (dense)
outputs = tf.reshape(stacked_outputs, [-1, num_periods, output])          #shape of results

loss = tf.reduce_sum(tf.square(outputs - y))    #define the cost function which evaluates the quality of our model
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)          #gradient descent method
training_op = optimizer.minimize(loss)          #train the result of the application of the cost_function

init = tf.global_variables_initializer()           #initialize all the variables
```

There is a lot going on there, so let's examine one step at a time. We are specifying the number of periods we are using to predict. In this case, it is the number of sequences that we are feeding into the model as a single input. We specify our variable placeholders. We initialize a type of RNN cell to use (size 100) and the type of activation function we want. ReLU stands for “Rectified Linear Unit” and is the default activation function, but it can be changed to Sigmoid, Hyberbolic Tangent (Tanh), and others, if desired.

We want our outputs to be in the same format as our inputs so we can compare our results using the loss function. In this case, we are using mean squared error (MSE), since this is a regression problem, in which our goal is to minimize the difference between the actual and the predicted. If we were dealing with a classification outcome, we might use cross-entropy. Now that we have this loss function defined, it is possible to define the training operation in TensorFlow that will optimize our network of input and outputs. To execute the optimization, we will use the Adam optimizer. Adam optimizer is a great general-purpose optimizer that performs our gradient descent via backpropagation through time. This allows faster convergence at the cost of more computation.

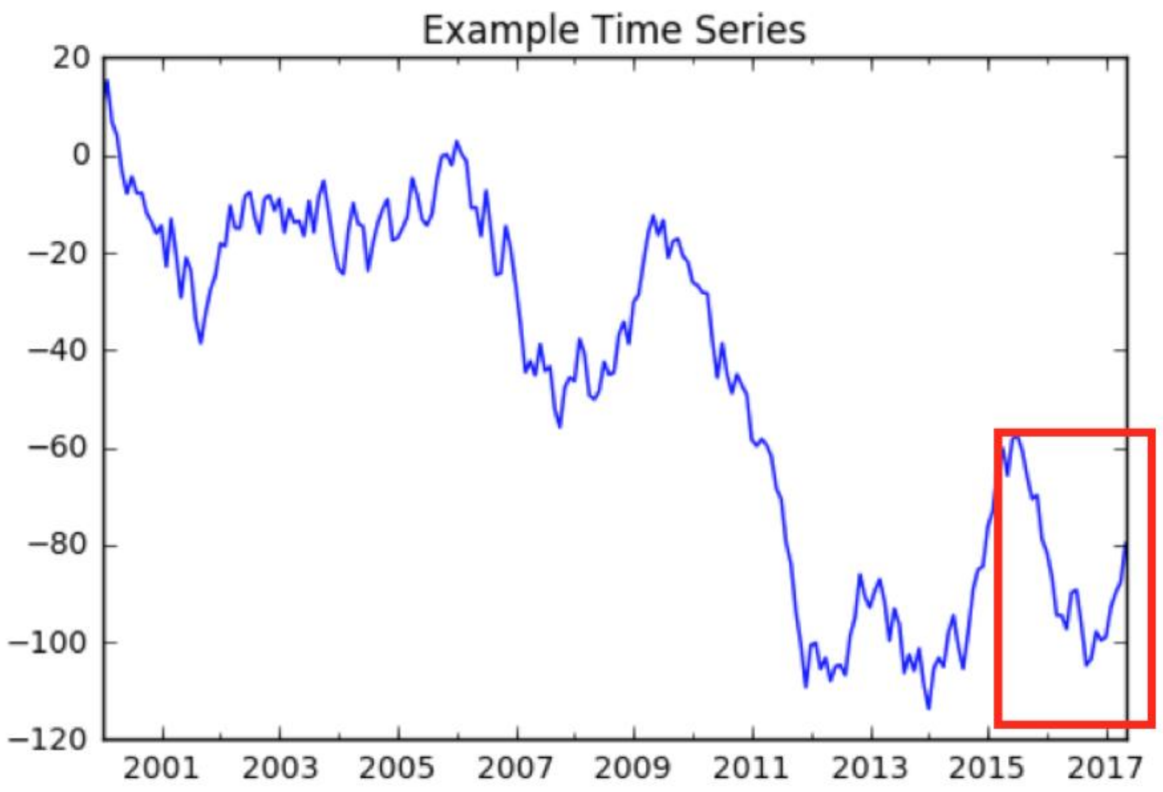
Now it is time to implement this model on our training data.

```
epochs = 1000     #number of iterations or training cycles, includes both the FeedFoward and Backpropogation

with tf.Session() as sess:
    init.run()
    for ep in range(epochs):
        sess.run(training_op, feed_dict={X: x_batches, y: y_batches})
        if ep % 100 == 0:
            mse = loss.eval(feed_dict={X: x_batches, y: y_batches})
            print(ep, "\tMSE:", mse)

    y_pred = sess.run(outputs, feed_dict={X: X_test})
    print(y_pred)
```

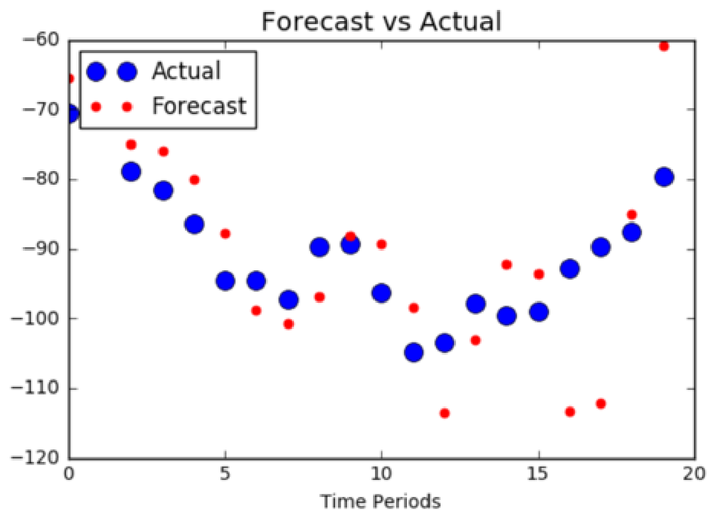
```
0       MSE: 724844.0
100     MSE: 19297.0
200     MSE: 10030.5
300     MSE: 4453.94
400     MSE: 2486.23
500     MSE: 1849.27
600     MSE: 1537.43
700     MSE: 1365.78
800     MSE: 1141.08
900     MSE: 1062.75
[[[ -65.43389893]
  [ -70.15782166]
  [ -74.97548676]
  [ -76.0067215 ]
  [ -80.11599731]
  [ -87.79607391]
  [ -98.87727356]
  [-100.79344177]
  [ -96.78859711]
  [ -88.2572937 ]
  [ -89.33888245]
  [ -98.35159302]
  [-113.49594879]
  [-103.05888367]
  [ -92.25975037]
  [ -93.55337524]
  [-113.38123322]
  [-112.15855408]
  [ -85.15632629]
  [ -60.79482269]]]
```

We specify the number of iterations/epochs that will cycle through our batches of training sequences. We create our graph object (tf.Session()) and initialize our data to be fed into the model as we cycle through the epochs. The abbreviated output shows the MSE after each 100 epochs. As our model feeds the data forward and backpropagation runs, it adjusts the weights applied to the inputs and runs another training epoch. Our MSE continues to improve (decrease). Finally, once the model is done, it takes the parameters and applies them to the test data to give us our predicted output for Y.

Let’s check our predicted versus actual. For our test data, we were focused on the last 20 periods of the entire 209 periods.



```
plt.title("Forecast vs Actual", fontsize=14)
plt.plot(pd.Series(np.ravel(Y_test)), "bo", markersize=10, label="Actual")
#plt.plot(pd.Series(np.ravel(Y_test)), "w*", markersize=10)
plt.plot(pd.Series(np.ravel(y_pred)), "r.", markersize=10, label="Forecast")
plt.legend(loc="upper left")
plt.xlabel("Time Periods")

plt.show()
```



It would appear there is some room for improvement ☺. However, this can be done by changing the number of hidden neurons and/or increasing the number of epochs. Optimizing our model is a process of trial and error, but we have a great start. This is random data, so we were expecting great results, but perhaps applying this model to a real-time series would give the ARIMA models some quality competition.

RNNs (and Deep Learning in general) are expanding the options available to data scientists to solve interesting problems. One issue that many data scientists face is how can we automate our analysis to run, once we have optimized it? Having a platform like MapR allows for this ability because you can construct, train, test, and optimize your model on a big data environment. In this example, we only used 10 training batches. What if my data allowed me to leverage hundreds of batches, not merely of 20 periods, but 50 or 100 or 500? I think I could definitely improve this model's performance. Once I did, I could package it up into an automated script to run on an individual node, a GPU node, in a Docker container, or all of the above. That's the power of doing data science and deep learning on a converged data platform.

Additional Resources

- Read blog 'TensorFlow on MapR Tutorial: A Perfect Place to Start'
- Read blog 'Deep Learning: What Are My Options?'
- Read blog 'Scalable Machine Learning on the MapR Converged Data Platform via SparkR and H2O'

^1^ Portions of this model were taken from the fantastic book *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, 1st Edition, by Aurélien Géron.

This blog post was published June 10, 2017.