

RecurrentNeuralNetworks

December 21, 2017

1 Basic Manual RNN

```
In [42]: import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
```

1.1 Constants

```
In [43]: # Number of inputs for each example
num_inputs = 2

# Number of neurons in first layer
num_neurons = 3
```

1.2 Placeholders

```
In [44]: # We now need two Xs! One for each timestamp (t=0 and t=1)
x0 = tf.placeholder(tf.float32, [None, num_inputs])
x1 = tf.placeholder(tf.float32, [None, num_inputs])
```

1.3 Variables

```
In [45]: # We'll also need a Weights variable for each x
# Notice the shape dimensions on both!
Wx = tf.Variable(tf.random_normal(shape=[num_inputs, num_neurons]))
Wy = tf.Variable(tf.random_normal(shape=[num_neurons, num_neurons]))
b = tf.Variable(tf.zeros([1, num_neurons]))
```

1.4 Graphs

```
In [46]: # First Activation
y0 = tf.tanh(tf.matmul(x0, Wx) + b)
y1 = tf.tanh(tf.matmul(y0, Wy) + tf.matmul(x1, Wx) + b)
```

1.5 Initialize Variables

```
In [47]: init = tf.global_variables_initializer()
```

1.6 Run Session

```
In [48]: # Some data;
# BATCH 0:      example1 , example2, example 3
# DATA AT TIMESTAMP = 1
x0_batch = np.array(
    [[0,1], [2,3], [4,5]])
# BATCH 0:      example1 , example2, example 3
# DATA AT TIMESTAMP = 1
x1_batch = np.array(
    [[100,101], [102,103], [104,105]])

with tf.Session() as sess:
    sess.run(init)
    y0_output_vals , y1_output_vals = sess.run(
        [y0,y1],feed_dict={x0:x0_batch,x1:x1_batch})

In [49]: # The output of values at t=0
y0_output_vals

Out[49]: array([[ -0.79196179, -0.1730756 ,  0.91266376],
               [ -0.99970078, -0.98578572,  0.99794251],
               [-0.99999964, -0.99985462,  0.99995357]], dtype=float32)

In [50]: # Output at t=1
y1_output_vals

"""
This won't scale! So let's use tensorflow!
"""

Out[50]: "\nThis won't scale! So let's use tensorflow!\n"
```

2 RNN with TensorFlow API

2.1 The Data

```
In [51]: class TimeSeriesData():
    def __init__(self,num_points,xmin,xmax):
        self.xmin = xmin
        self.xmax = xmax
        self.num_points = num_points
        self.resolution = (xmax-xmin)/num_points
        self.x_data = np.linspace(xmin,xmax,num_points)
        self.y_true = np.sin(self.x_data)

    def ret_true(self,x_series):
        return np.sin(x_series)
```

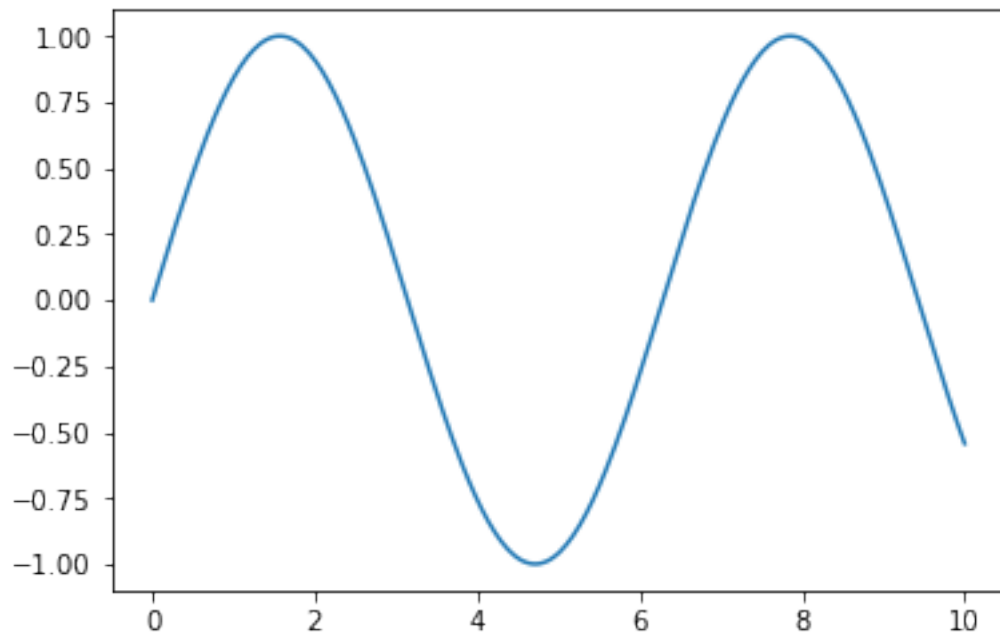
```

def next_batch(
    self, batch_size, steps, return_batch_ts=False):
    # Grab a random starting point for each batch
    rand_start = np.random.rand(batch_size,1)
    # Convert to be on time series
    ts_start = rand_start * (
        self.xmax- self.xmin - (steps*self.resolution) )
    # Create batch Time Series on t axis
    batch_ts = ts_start + np.arange(0.0,steps+1) * self.resolution
    # Create Y data for time series in the batches
    y_batch = np.sin(batch_ts)
    # Format for RNN
    if return_batch_ts:
        return y_batch[:, :-1].reshape(
            -1, steps, 1), y_batch[:, 1:].reshape(-1, steps, 1) ,batch_ts
    else:
        return y_batch[:, :-1].reshape(
            -1, steps, 1), y_batch[:, 1:].reshape(-1, steps, 1)

ts_data = TimeSeriesData(250,0,10)
plt.plot(ts_data.x_data,ts_data.y_true)

```

Out[51]: [<matplotlib.lines.Line2D at 0x22d78c587f0>]



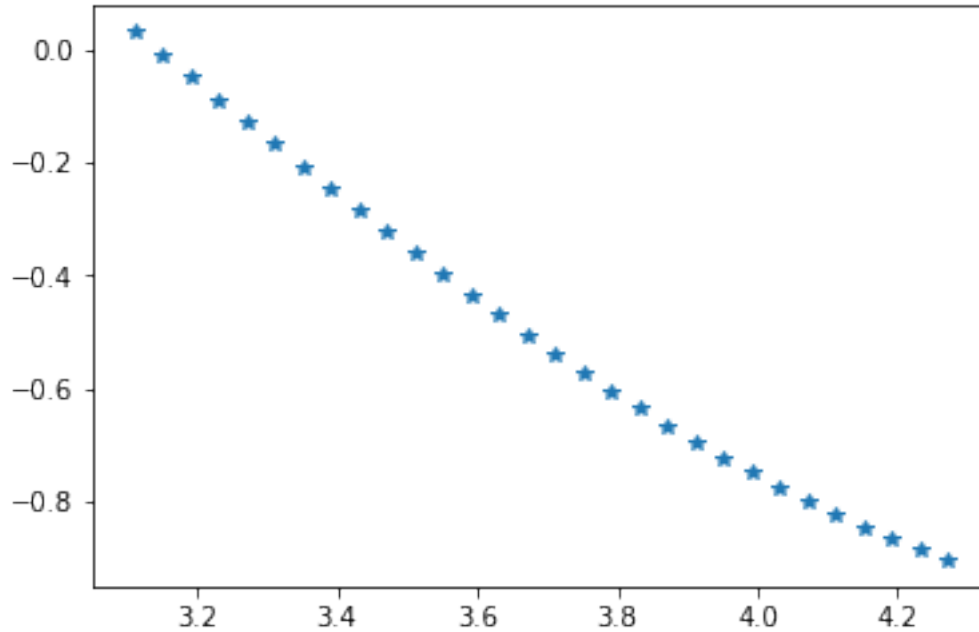
```

In [52]: # Num of steps in batch (also used for prediction steps into the future)
num_time_steps = 30

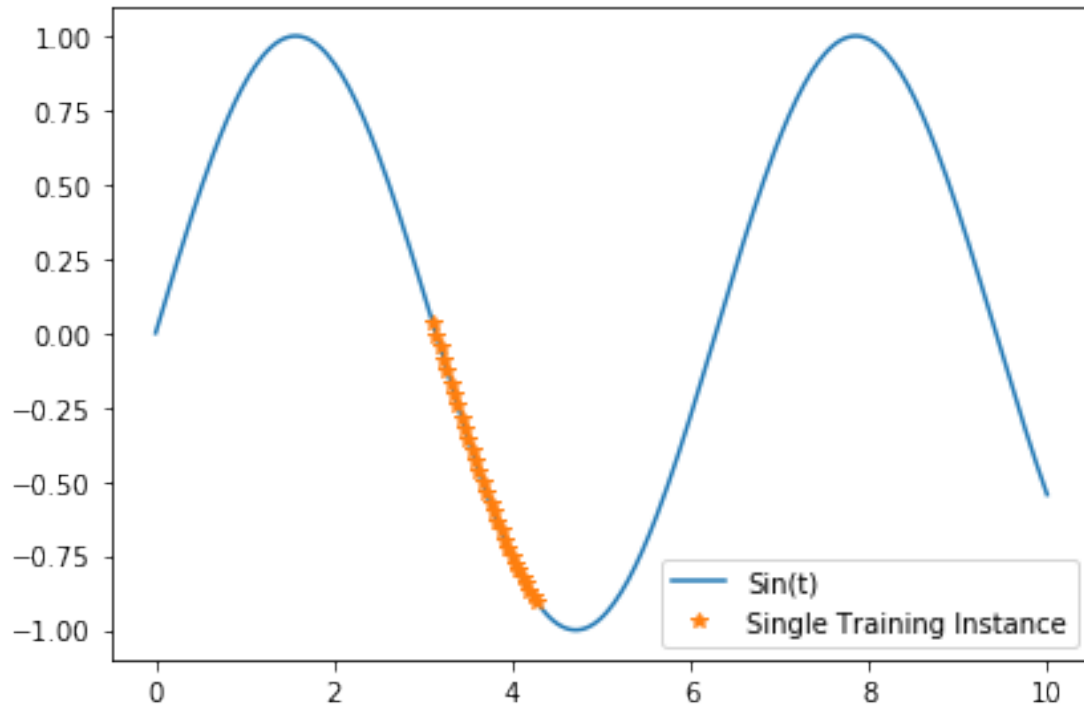
```

```
y1,y2,ts = ts_data.next_batch(1,num_time_steps,True)
plt.plot(ts.flatten()[1:],y2.flatten(),'*')
```

Out[52]: [<matplotlib.lines.Line2D at 0x22d78a85048>]



```
In [53]: plt.plot(ts_data.x_data,ts_data.y_true,label='Sin(t)')
plt.plot(ts.flatten()[1:],y2.flatten(),'*',label='Single Training Instance')
plt.legend()
plt.tight_layout()
```



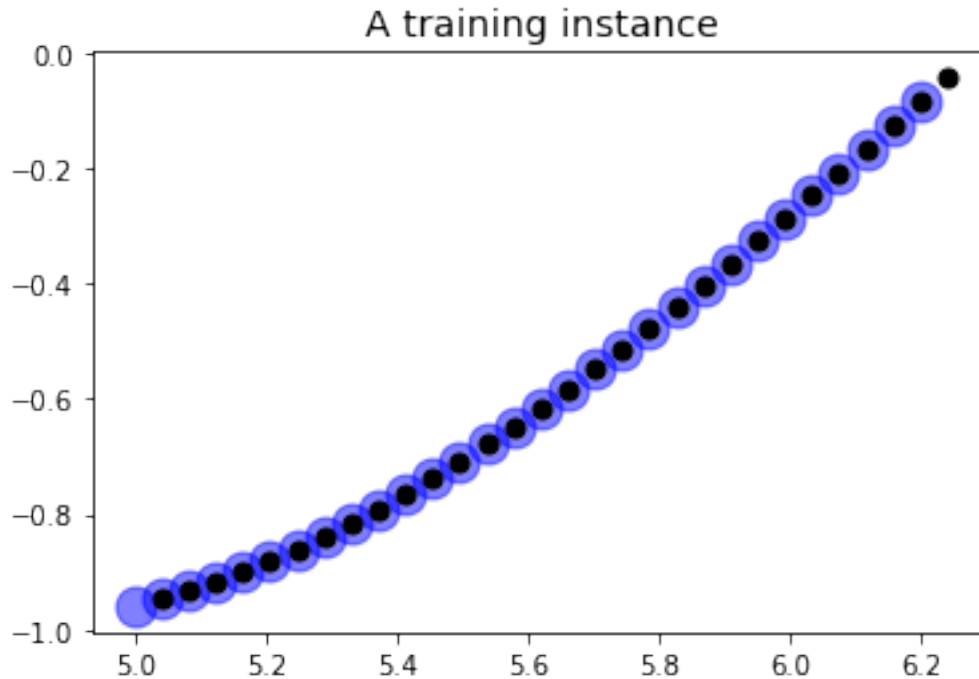
2.2 Training Instance and what to Predict

We are trying to predict a time series shifted over by $t+1$

```
In [54]: train_inst = np.linspace(5, 5 + ts_data.resolution *
                                   (num_time_steps + 1), num_time_steps+1)

In [55]: plt.title("A training instance", fontsize=14)
plt.plot(train_inst[:-1], ts_data.ret_true(
    train_inst[:-1]), "bo", markersize=15, alpha=0.5, label="instance")
plt.plot(train_inst[1:], ts_data.ret_true(
    train_inst[1:]), "ko", markersize=7, label="target")

Out[55]: [<matplotlib.lines.Line2D at 0x22d78a51da0>]
```



2.3 Creating the model

2.3.1 Constants

```
In [56]: tf.reset_default_graph()
         # Just one feature, the time series
         num_inputs = 1
         # 100 neuron layer, play with this
         num_neurons = 100
         # Just one output, predicted time series
         num_outputs = 1
         # learning rate, 0.0001 default,
         #but you can play with this
         learning_rate = 0.0001
         # how many iterations to go through
         #(training steps), you can play with this
         num_train_iterations = 2000
         # Size of the batch of data
         batch_size = 1
```

2.3.2 Placeholders

```
In [57]: X = tf.placeholder(tf.float32, [None, num_time_steps, num_inputs])
         y = tf.placeholder(tf.float32, [None, num_time_steps, num_outputs])
```

2.3.3 RNN Cell Layer

Play around with the various cells in this section, compare how they perform against each other.

```
In [58]: cell = tf.contrib.rnn.OutputProjectionWrapper(
          tf.contrib.rnn.BasicRNNCell(num_units=num_neurons, activation=tf.nn.relu),
          output_size=num_outputs)
        """
        cell = tf.contrib.rnn.OutputProjectionWrapper(
            tf.contrib.rnn.BasicLSTMCell(num_units=num_neurons, activation=tf.nn.relu),
            output_size=num_outputs)

        n_neurons = 100
        n_layers = 3

        cell = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.BasicRNNCell(num_units=n_neurons)
                                             for layer in range(n_layers)])

        cell = tf.contrib.rnn.BasicLSTMCell(num_units=num_neurons, activation=tf.nn.relu)

        n_neurons = 100
        n_layers = 3

        cell = tf.contrib.rnn.MultiRNNCell([tf.contrib.rnn.BasicLSTMCell(num_units=n_neurons)
                                             for layer in range(n_layers)])
        """

Out[58]: '\ncell = tf.contrib.rnn.OutputProjectionWrapper(\n      tf.contrib.rnn.BasicLSTMCell(nu
```

2.3.4 Dynamic RNN Cell

```
In [59]: outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

2.3.5 Loss function and Optimizer

```
In [60]: loss = tf.reduce_mean(tf.square(outputs - y)) # MSE
          optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
          train = optimizer.minimize(loss)
```

2.3.6 Init Variables

```
In [61]: init = tf.global_variables_initializer()
```

2.4 Session

```
In [62]: # ONLY FOR GPU USERS:
# https://stackoverflow.com/questions/34199233/
# how-to-prevent-tensorflow-from-allocating-the-totality-of-a-gpu-memory
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.75)
saver = tf.train.Saver()
with tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)) as sess:
    sess.run(init)

    for iteration in range(num_train_iterations):

        X_batch, y_batch = ts_data.next_batch(batch_size, num_time_steps)
        sess.run(train, feed_dict={X: X_batch, y: y_batch})

        if iteration % 100 == 0:

            mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
            print(iteration, "\tMSE:", mse)

    # Save Model for Later
    saver.save(sess, "./rnn_time_series_model")

0      MSE: 0.233602
100    MSE: 0.40067
200    MSE: 0.00483153
300    MSE: 0.0175155
400    MSE: 0.0176565
500    MSE: 0.0470966
600    MSE: 0.00438213
700    MSE: 0.00019475
800    MSE: 0.00806291
900    MSE: 0.00293631
1000   MSE: 0.00346564
1100   MSE: 0.000276441
1200   MSE: 0.00789621
1300   MSE: 0.00506087
1400   MSE: 0.00580755
1500   MSE: 0.00288942
1600   MSE: 0.00449396
1700   MSE: 0.00464371
1800   MSE: 0.00366746
1900   MSE: 0.00159508
```

2.4.1 Predicting a time series $t+1$

```
In [63]: with tf.Session() as sess:
    saver.restore(sess, "./rnn_time_series_model")
```

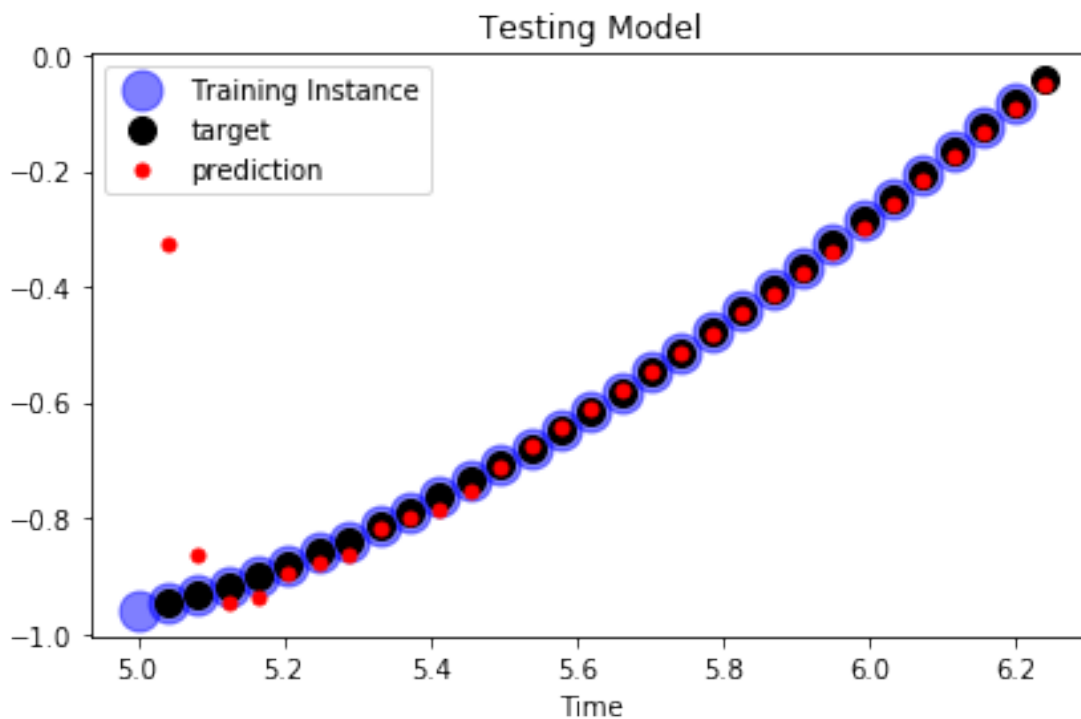


```

X_new = np.sin(np.array(train_inst[:-1].reshape(-1, num_time_steps, num_inputs)))
y_pred = sess.run(outputs, feed_dict={X: X_new})
plt.title("Testing Model")
# Training Instance
plt.plot(train_inst[:-1], np.sin(
    train_inst[:-1]), "bo", markersize=15, alpha=0.5, label="Training Instance")
# Target to Predict
plt.plot(train_inst[1:], np.sin(train_inst[1:]), "ko", markersize=10, label="target")
# Models Prediction
plt.plot(train_inst[1:], y_pred[0,:,0], "r.", markersize=10, label="prediction")
plt.xlabel("Time")
plt.legend()
plt.tight_layout()

```

INFO:tensorflow:Restoring parameters from ./rnn_time_series_model



2.5 Generating New Sequences

Sometimes can give interesting and wacky results, tread carefully

```

In [64]: with tf.Session() as sess:
          saver.restore(sess, "./rnn_time_series_model")

```

```

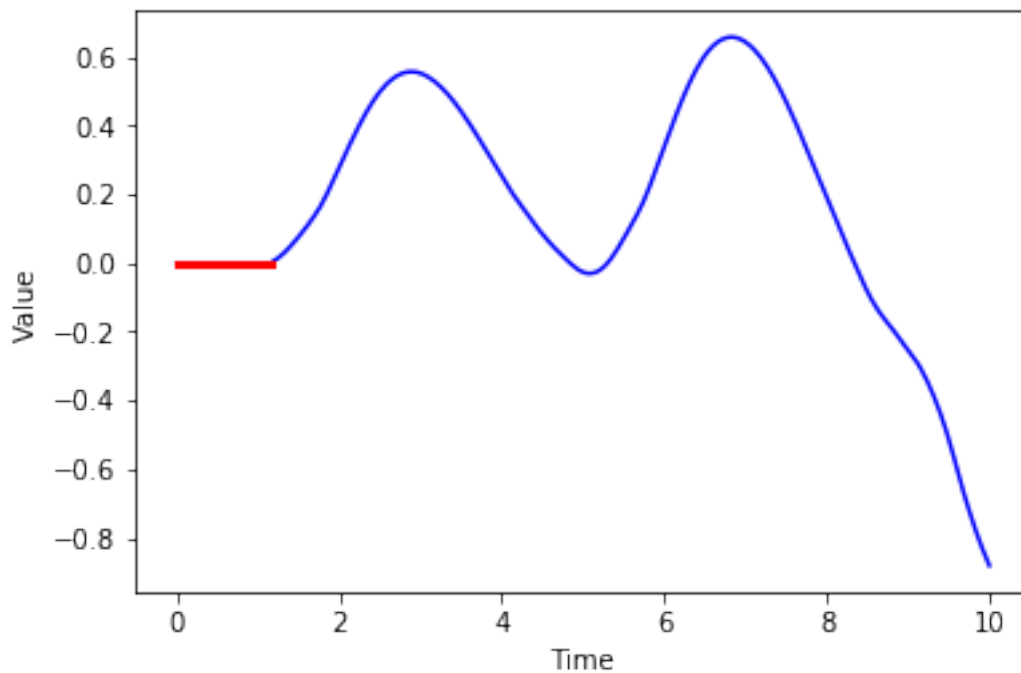
# SEED WITH ZEROS
zero_seq_seed = [0. for i in range(num_time_steps)]
for iteration in range(len(ts_data.x_data) - num_time_steps):
    X_batch = np.array(
        zero_seq_seed[-num_time_steps:]).reshape(1, num_time_steps, 1)
    y_pred = sess.run(outputs, feed_dict={X: X_batch})
    zero_seq_seed.append(y_pred[0, -1, 0])

plt.plot(ts_data.x_data, zero_seq_seed, "b-")
plt.plot(ts_data.x_data[:num_time_steps],
        zero_seq_seed[:num_time_steps], "r", linewidth=3)
plt.xlabel("Time")
plt.ylabel("Value")

```

INFO:tensorflow:Restoring parameters from ./rnn_time_series_model

Out[64]: Text(0,0.5,'Value')



```

In [65]: with tf.Session() as sess:
    saver.restore(sess, "./rnn_time_series_model")
    # SEED WITH Training Instance
    training_instance = list(ts_data.y_true[:30])
    for iteration in range(len(ts_data.x_data) - num_time_steps):
        X_batch = np.array(

```

```

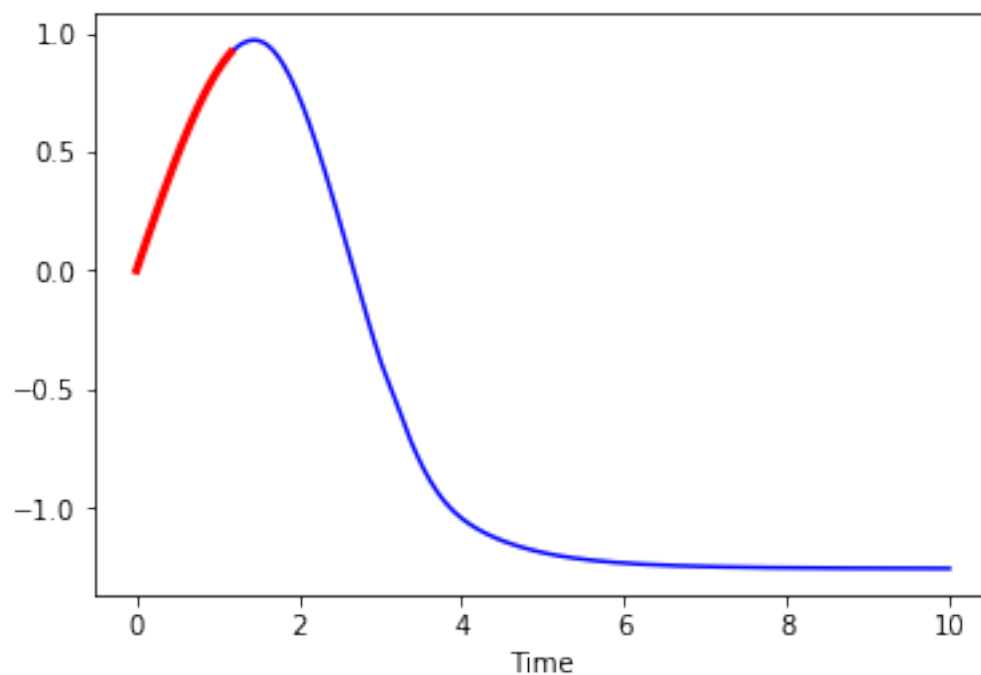
        training_instance[-num_time_steps:]).reshape(1, num_time_steps, 1)
    y_pred = sess.run(outputs, feed_dict={X: X_batch})
    training_instance.append(y_pred[0, -1, 0])

plt.plot(ts_data.x_data, training_instance, "b-")
plt.plot(ts_data.x_data[:num_time_steps],
        training_instance[:num_time_steps], "r-", linewidth=3)
plt.xlabel("Time")

INFO:tensorflow:Restoring parameters from ./rnn_time_series_model

```

Out[65]: Text(0.5,0,'Time')



3 Time Series Exercise

3.1 The Data

Source: <https://datamarket.com/data/set/22ox/monthly-milk-production-pounds-per-cow-jan-62-dec-75#!ds=22ox&display=line>

Monthly milk production: pounds per cow. Jan 62 - Dec 75

1. Import numpy pandas and matplotlib

```

In [1]: import pandas as pd
        import numpy as np

```

```
import tensorflow as tf
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Use pandas to read the csv of the monthly-milk-production.csv file and set index_col='Month'

3. Check out the head of the dataframe

```
In [2]: milk = pd.read_csv('monthly-milk-production.csv', index_col='Month')
milk.head()
```

```
Out[2]:
```

Milk Production	
Month	
1962-01-01 01:00:00	589.0
1962-02-01 01:00:00	561.0
1962-03-01 01:00:00	640.0
1962-04-01 01:00:00	656.0
1962-05-01 01:00:00	727.0

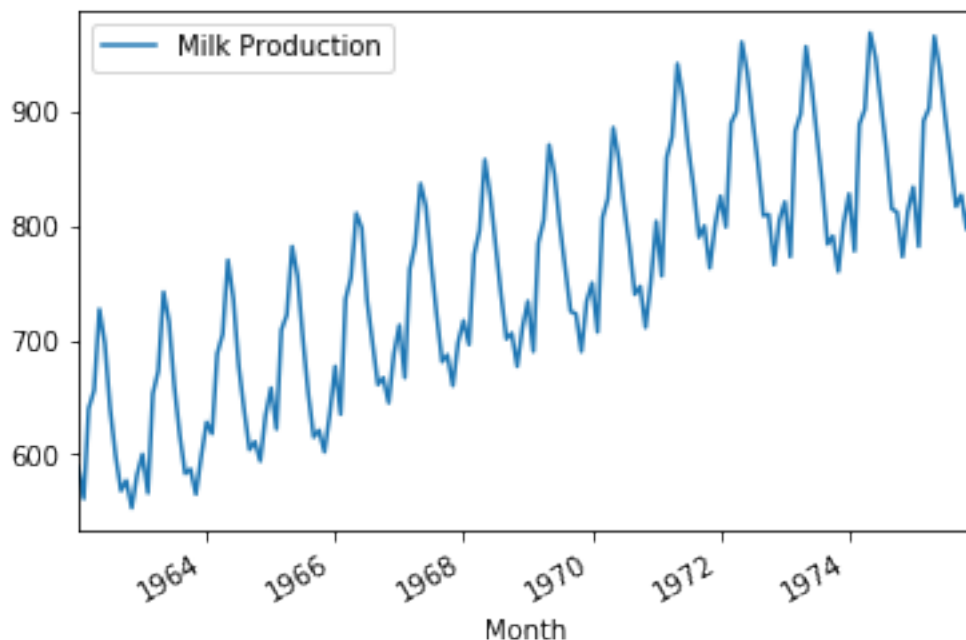
4. Make the index a time series by using:

```
milk.index = pd.to_datetime(milk.index)
```

5. Plot out the time series data.

```
In [3]: milk.index = pd.to_datetime(milk.index)
milk.plot()
```

```
Out[3]: <matplotlib.axes._subplots.AxesSubplot at 0x240d8d23860>
```



3.1.1 Train Test Split

6. Let's attempt to predict a year's worth of data. (12 months or 12 steps into the future)

7. Create a test train split using indexing (hint: use `.head()` or `.tail()` or `.iloc[]`). We don't want a random train test split, we want to specify that the test set is the last 3 months of data is the test set, with everything before it is the training.

```
In [4]: train_set = milk.head(156)
        test_set = milk.tail(12)
        milk.info()

<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 168 entries, 1962-01-01 01:00:00 to 1975-12-01 01:00:00
Data columns (total 1 columns):
Milk Production    168 non-null float64
dtypes: float64(1)
memory usage: 2.6 KB
```

3.1.2 Scale the Data

8. Use `sklearn.preprocessing` to scale the data using the `MinMaxScaler`. Remember to only `fit_transform` on the training data, then transform the test data. You shouldn't fit on the test data as well, otherwise you are assuming you would know about future behavior!

```
In [5]: from sklearn.preprocessing import MinMaxScaler
        scaler = MinMaxScaler()
        train_scaled = scaler.fit_transform(train_set)
        test_scaled = scaler.fit_transform(test_set)
```

3.1.3 Batch Function

9. We'll need a function that can feed batches of the training data. We'll need to do several things that are listed out as steps in the comments of the function. Remember to reference the previous batch method from the lecture for hints. Try to fill out the function template below, this is a pretty hard step, so feel free to reference the solutions!

```
In [6]: def next_batch(training_data, batch_size, steps):
        """
        INPUT: Data, Batch Size, Time Steps per batch
        OUTPUT: A tuple of y time series results. y[:, :-1] and y[:, 1:]
        """

        # STEP 1: Use np.random.randint to set a random starting point index for the batch.
```

```

# Remember that each batch needs have the same number of steps in it.
# This means you should limit the starting point to len(data)-steps

# Grab a random starting point for each batch
rand_start = np.random.randint(0, len(training_data)-steps)

# STEP 2: Now that you have a starting index you'll need to index the data from
# the random start to random start + steps. Then reshape this data to be (1, steps)

# Create Y data for time series in the batches
y_batch = np.array(training_data[rand_start:rand_start+steps+1]).reshape(1, steps+1)

# STEP 3: Return the batches. You'll have two batches to return y[:, :-1] and y[:, 1:]
# You'll need to reshape these into tensors for the RNN. Depending on your indexing
# will be either .reshape(-1, steps-1, 1) or .reshape(-1, steps, 1)

return y_batch[:, :-1].reshape(-1, steps, 1), y_batch[:, 1:].reshape(-1, steps, 1)

```

3.2 Setting Up The RNN Model

10. Import TensorFlow

3.2.1 The Constants

11. Define the constants in a single cell. You'll need the following (in parenthesis are the values I used in my solution, but you can play with some of these): * Number of Inputs (1) * Number of Time Steps (12) * Number of Neurons per Layer (100) * Number of Outputs (1) * Learning Rate (0.003) * Number of Iterations for Training (4000) * Batch Size (1)

```

In [7]: # Just one feature, the time series
num_inputs = 1
# Num of steps in each batch
num_time_steps = 12
# 100 neuron layer
num_neurons = 100
# Just one output, predicted time series
num_outputs = 1
## You can also try increasing iterations, but decreasing learning rate
# learning rate you can play with this
learning_rate = 0.001
# how many iterations to go through (training steps), you can play with this
num_train_iterations = 10000
# Size of the batch of data
batch_size = 1

```

12. Create Placeholders for X and y. (You can change the variable names if you want). The shape for these placeholders should be [None, num_time_steps-1, num_inputs] and [None, num_time_steps-1, num_outputs] The reason we use num_time_steps-1 is because each of these

will be one step shorter than the original time steps size, because we are training the RNN network to predict one point into the future based on the input sequence.

```
In [8]: X = tf.placeholder(tf.float32, [None, num_time_steps, num_inputs])
        y = tf.placeholder(tf.float32, [None, num_time_steps, num_inputs])
```

13. Now create the RNN Layer, you have complete freedom over this, use `tf.contrib.rnn` and choose anything you want, `OutputProjectionWrappers`, `BasicRNNCells`, `BasicLSTMCells`, `MultiRNNCell`, `GRUCell` etc... Keep in mind not every combination will work well! (If in doubt, the solutions used an `Outputprojection Wrapper` around a basic LSTM cell with `relu` activation.

14. Now pass in the cells variable into `tf.nn.dynamic_rnn`, along with your first placeholder (X)

```
In [9]: cell = tf.contrib.rnn.OutputProjectionWrapper(
        tf.contrib.rnn.GRUCell(num_units=num_neurons,
                                activation=tf.nn.relu), output_size=num_outputs)
        outputs, states = tf.nn.dynamic_rnn(cell, X, dtype=tf.float32)
```

3.2.2 Loss Function and Optimizer

15. Create a Mean Squared Error Loss Function and use it to minimize an `AdamOptimizer`, remember to pass in your learning rate.

16. Initialize the global variables

17. Create an instance of `tf.train.Saver()`

```
In [10]: loss = tf.reduce_mean(tf.square(outputs-y)) # MSE
         optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
         train = optimizer.minimize(loss)
         init = tf.global_variables_initializer()
```

3.2.3 Session

18. Run a `tf.Session` that trains on the batches created by your `next_batch` function. Also add an a loss evaluation for every 100 training iterations. Remember to save your model after you are done training.

```
In [11]: gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction = .9)
        saver = tf.train.Saver()
        with tf.Session(config = tf.ConfigProto(gpu_options=gpu_options)) as sess:
            sess.run(init)
            for iteration in range(num_train_iterations):
                X_batch, y_batch = next_batch(train_scaled, batch_size, num_time_steps)
                sess.run(train, feed_dict={X: X_batch, y: y_batch})
                if iteration % 500 == 0:
                    mse = loss.eval(feed_dict={X: X_batch, y: y_batch})
                    print(iteration, "\tMSE:", mse)
            saver.save(sess, "./rnn_time_series_model")
```

```

0          MSE: 0.414051
500        MSE: 0.0144065
1000       MSE: 0.00730563
1500       MSE: 0.00640915
2000       MSE: 0.00585742
2500       MSE: 0.00704521
3000       MSE: 0.00467488
3500       MSE: 0.00589041
4000       MSE: 0.00556548
4500       MSE: 0.00506271
5000       MSE: 0.0048962
5500       MSE: 0.00477827
6000       MSE: 0.0031558
6500       MSE: 0.00205747
7000       MSE: 0.003039
7500       MSE: 0.00401108
8000       MSE: 0.00270145
8500       MSE: 0.00405873
9000       MSE: 0.00410118
9500       MSE: 0.00550826

```

3.3 Predicting Future (Test Data)

19. Show the test_set (the last 12 months of your original complete data set)

```
In [12]: test_set
```

```

Out[12]:
           Milk Production
Month
1975-01-01 01:00:00      834.0
1975-02-01 01:00:00      782.0
1975-03-01 01:00:00      892.0
1975-04-01 01:00:00      903.0
1975-05-01 01:00:00      966.0
1975-06-01 01:00:00      937.0
1975-07-01 01:00:00      896.0
1975-08-01 01:00:00      858.0
1975-09-01 01:00:00      817.0
1975-10-01 01:00:00      827.0
1975-11-01 01:00:00      797.0
1975-12-01 01:00:00      843.0

```

3.3.1 Generative Session

Now we want to attempt to predict these 12 months of data, using only the training data we had. To do this we will feed in a seed training_instance of the last 12 months of the training_set of data to predict 12 months into the future. Then we will be able to compare our generated 12 months to our actual true historical values from the test set!

NOTE: Recall that our model is really only trained to predict 1 time step ahead, asking it to generate 12 steps is a big ask, and technically not what it was trained to do! Think of this more as generating new values based off some previous pattern, rather than trying to directly predict the future. You would need to go back to the original model and train the model to predict 12 time steps ahead to really get a higher accuracy on the test data. (Which has its limits due to the smaller size of our data set)

20. Fill out the session code below to generate 12 months of data based off the last 12 months of data from the training set. The hardest part about this is adjusting the arrays with their shapes and sizes. Reference the lecture for hints.

21. Show the result of the predictions.

```
In [13]: with tf.Session() as sess:
          # Use your Saver instance to restore your saved rnn time series model
          saver.restore(sess, "./rnn_time_series_model")

          # Create a numpy array for your genreative seed from the last 12 months of the
          # training set data. Hint: Just use tail(12) and then pass it to an np.array
          train_seed = list(train_scaled[-12:])

          for iteration in range(12):
              X_batch = np.array(
                  train_seed[-num_time_steps:]).reshape(1, num_time_steps, 1)
              y_pred = sess.run(outputs, feed_dict={X: X_batch})
              train_seed.append(y_pred[0, -1, 0])

          train_seed
```

```
INFO:tensorflow:Restoring parameters from ./rnn_time_series_model
```

```
Out[13]: [array([ 0.66105769]),
          array([ 0.54086538]),
          array([ 0.80769231]),
          array([ 0.83894231]),
          array([ 1.]),
          array([ 0.94711538]),
          array([ 0.85336538]),
          array([ 0.75480769]),
          array([ 0.62980769]),
          array([ 0.62259615]),
          array([ 0.52884615]),
          array([ 0.625]),
          0.68514168,
          0.58138394,
          0.83863091,
          0.87113422,
          1.0321299,
          0.96059185,
```

```
0.86626852,  
0.77739108,  
0.68386054,  
0.68219084,  
0.61764836,  
0.70163476]
```

21. Grab the portion of the results that are the generated values and apply `inverse_transform` on them to turn them back into milk production value units (lbs per cow). Also reshape the results to be (12,1) so we can easily add them to the `test_set` dataframe.

22. Create a new column on the `test_set` called "Generated" and set it equal to the generated results. You may get a warning about this, feel free to ignore it.

23. View the `test_set` dataframe.

```
In [14]: results = scaler.inverse_transform(np.array(train_seed[12:]).reshape(12,1))  
         test_set['Generated'] = results  
         test_set
```

```
C:\Users\Ripti\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: SettingWithCopyWarning:  
A value is trying to be set on a copy of a slice from a DataFrame.  
Try using .loc[row_indexer,col_indexer] = value instead
```

See the caveats in the documentation: <http://pandas.pydata.org/pandas-docs/stable/indexing.html#>

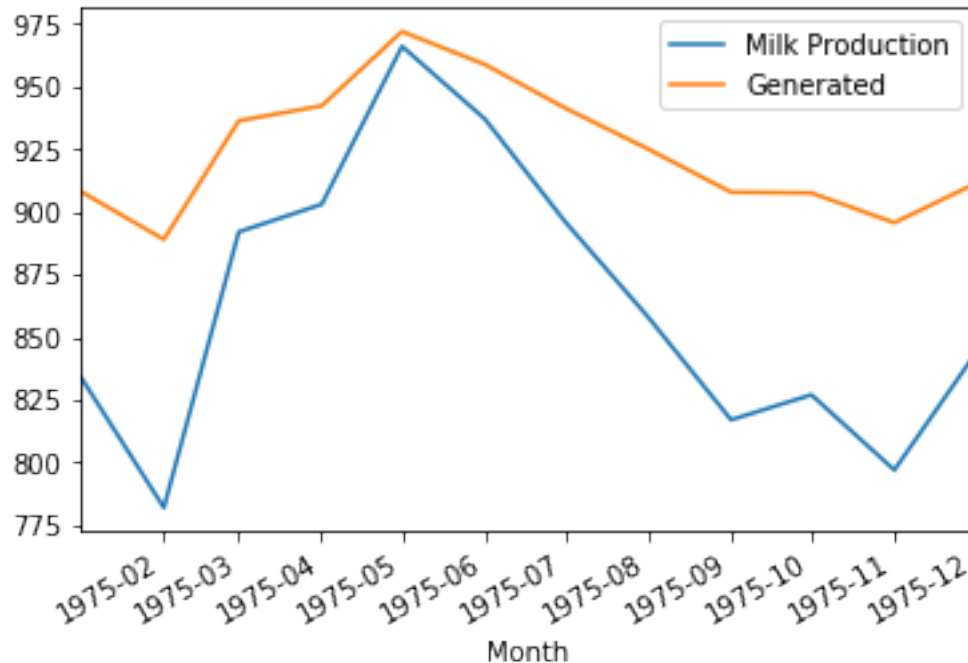
```
Out[14]:
```

	Milk Production	Generated
Month		
1975-01-01 01:00:00	834.0	908.066040
1975-02-01 01:00:00	782.0	888.974609
1975-03-01 01:00:00	892.0	936.308044
1975-04-01 01:00:00	903.0	942.288696
1975-05-01 01:00:00	966.0	971.911865
1975-06-01 01:00:00	937.0	958.748901
1975-07-01 01:00:00	896.0	941.393433
1975-08-01 01:00:00	858.0	925.039917
1975-09-01 01:00:00	817.0	907.830383
1975-10-01 01:00:00	827.0	907.523132
1975-11-01 01:00:00	797.0	895.647278
1975-12-01 01:00:00	843.0	911.100830

24. Plot out the two columns for comparison.

```
In [15]: test_set.plot()
```

```
Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x240ddd3cf60>
```



We're asking the model to ask 12 steps ahead, so this is a relatively good result.

4 Word2Vec tutorial

4.1 Imports

```
In [24]: import collections
import math
import os
import errno
import random
import zipfile
import numpy as np
from six.moves import urllib
from six.moves import xrange
import tensorflow as tf
```

4.2 The Data

```
In [25]: data_dir = "word2vec_data/words"
data_url = 'http://mattmahoney.net/dc/text8.zip'

def fetch_words_data(url=data_url, words_data=data_dir):
    # Make the Dir if it does not exist
    os.makedirs(words_data, exist_ok=True)
```

```

# Path to zip file
zip_path = os.path.join(words_data, "words.zip")
# If the zip file isn't there, download it from the data url
if not os.path.exists(zip_path):
    urllib.request.urlretrieve(url, zip_path)
# Now that the zip file is there, get the data from it
with zipfile.ZipFile(zip_path) as f:
    data = f.read(f.namelist()[0])
# Return a list of all the words in the data source.
return data.decode("ascii").split()

# Use Defaults (this make take awhile!!)
words = fetch_words_data()
# Random slice of words
for w in words[9000:9014]:
    print(w, end=' ')

```

feelings and the auditory system of a person without autism often cannot sense the

4.3 Building Word Counts

```

In [26]: from collections import Counter
mylist = ["one", 'one', 'two']
Counter(mylist).most_common(1)

```

```

Out[26]: [('one', 2)]

```

4.4 Create Word Data and Vocabulary

```

In [27]: def create_counts(vocab_size=50000):
# Begin adding vocab counts with Counter
vocab = [] + Counter(words).most_common(vocab_size )
# Turn into a numpy array
vocab = np.array([word for word, _ in vocab])
dictionary = {word: code for code, word in enumerate(vocab)}
data = np.array([dictionary.get(word, 0) for word in words])
return data, vocab
vocab_size = 50000
# This may take awhile
data, vocabulary = create_counts(vocab_size=vocab_size)

```

```

In [28]: (words[100], data[100])

```

```

Out[28]: ('interpretations', 4186)

```

4.5 Function for Batches

```

In [29]: def generate_batch(batch_size, num_skips, skip_window):
global data_index

```

```

assert batch_size % num_skips == 0
assert num_skips <= 2 * skip_window
batch = np.ndarray(shape=(batch_size), dtype=np.int32)
labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
span = 2 * skip_window + 1 # [ skip_window target skip_window ]
buffer = collections.deque(maxlen=span)
if data_index + span > len(data):
    data_index = 0
buffer.extend(data[data_index:data_index + span])
data_index += span
for i in range(batch_size // num_skips):
    target = skip_window # target label at the center of the buffer
    targets_to_avoid = [skip_window]
    for j in range(num_skips):
        while target in targets_to_avoid:
            target = random.randint(0, span - 1)
        targets_to_avoid.append(target)
        batch[i * num_skips + j] = buffer[skip_window]
        labels[i * num_skips + j, 0] = buffer[target]
if data_index == len(data):
    buffer[:] = data[:span]
    data_index = span
else:
    buffer.append(data[data_index])
    data_index += 1
# Backtrack a little bit to avoid skipping words in the end of a batch
data_index = (data_index + len(data) - span) % len(data)
return batch, labels
data_index=0
batch, labels = generate_batch(8, 2, 1)

```

4.6 Constants

```

In [30]: # Size of the bath
batch_size = 128

# Dimension of embedding vector
embedding_size = 150

# How many words to consider left and right (the bigger, the longer the training)
skip_window = 1

# How many times to reuse an input to generate a label
num_skips = 2

# We pick a random validation set to sample nearest neighbors. Here we limit the
# validation samples to the words that have a low numeric ID, which by
# construction are also the most frequent.

```

```

# Random set of words to evaluate similarity on.
valid_size = 16

# Only pick dev samples in the head of the distribution.
valid_window = 100
valid_examples = np.random.choice(valid_window, valid_size, replace=False)

# Number of negative examples to sample.
num_sampled = 64

# Model Learning Rate
learning_rate = 0.01

# How many words in vocab
vocabulary_size = 50000

```

4.7 TensorFlow Placeholders and Constants

```

In [31]: tf.reset_default_graph()
         # Input data.
         train_inputs = tf.placeholder(tf.int32, shape=[None])
         train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
         valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

```

4.8 Variables

```

In [32]: # Look up embeddings for inputs.
         init_embeds = tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0)
         embeddings = tf.Variable(init_embeds)
         embed = tf.nn.embedding_lookup(embeddings, train_inputs)

```

4.9 NCE Loss

```

In [33]: # Construct the variables for the NCE loss
         nce_weights = tf.Variable(
             tf.truncated_normal(
                 [vocabulary_size, embedding_size], stddev=1.0 / np.sqrt(embedding_size)))
         nce_biases = tf.Variable(tf.zeros([vocabulary_size]))
         # Compute the average NCE loss for the batch.
         # tf.nn.nce_loss automatically draws a new sample of the negative labels each
         # time we evaluate the loss.
         loss = tf.reduce_mean(
             tf.nn.nce_loss(nce_weights, nce_biases, train_labels, embed,
                           num_sampled, vocabulary_size))

```

4.10 Optimizer

```
In [34]: # Construct the Adam optimizer
optimizer = tf.train.AdamOptimizer(learning_rate=1.0)
trainer = optimizer.minimize(loss)
# Compute the cosine similarity between minibatch examples and all embeddings.
norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), axis=1, keep_dims=True))
normalized_embeddings = embeddings / norm
valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings, valid_dataset)
similarity = tf.matmul(valid_embeddings, normalized_embeddings, transpose_b=True)
# Add variable initializer.
init = tf.global_variables_initializer()
```

4.11 Session

```
In [35]: gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.9)
```

```
In [38]: # Usually needs to be quite large to get good results,
# training takes a long time!
num_steps = 200001
with tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)) as sess:
    sess.run(init)
    average_loss = 0
    for step in range(num_steps):
        batch_inputs, batch_labels = generate_batch(batch_size, num_skips, skip_window)
        feed_dict = {train_inputs : batch_inputs, train_labels : batch_labels}
        # We perform one update step by evaluating the training op (including it
        # in the list of returned values for session.run()
        empty, loss_val = sess.run([trainer, loss], feed_dict=feed_dict)
        average_loss += loss_val
        if step % 20000 == 0:
            if step > 0:
                average_loss /= 20000
                # The average loss is an estimate of the loss over the last 1000 batches.
                print("Average loss at step ", step, ": ", average_loss)
            average_loss = 0
    final_embeddings = normalized_embeddings.eval()
```

```
Average loss at step 0 : 317.766845703
Average loss at step 20000 : 39455.5291396
Average loss at step 40000 : 54623.6779175
Average loss at step 60000 : 66518.9150379
Average loss at step 80000 : 70374.9083885
Average loss at step 100000 : 76435.327955
Average loss at step 120000 : 82592.4306402
Average loss at step 140000 : 93608.1246127
Average loss at step 160000 : 94800.8771616
Average loss at step 180000 : 91514.3004474
```

Average loss at step 200000 : 94899.6697334

4.12 Visualizing Results

```
In [39]: def plot_with_labels(low_dim_embs, labels):
    assert low_dim_embs.shape[0] >= len(labels), "More labels than embeddings"
    plt.figure(figsize=(18, 18)) #in inches
    for i, label in enumerate(labels):
        x, y = low_dim_embs[i,:]
        plt.scatter(x, y)
        plt.annotate(label,
                      xy=(x, y),
                      xytext=(5, 2),
                      textcoords='offset points',
                      ha='right',
                      va='bottom')
```

4.12.1 TSNE

- <https://lvdmaaten.github.io/tsne/>
- https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding

Dimensionality reduction to 2-D vectors (down from 150), this takes awhile.

```
In [40]: from sklearn.manifold import TSNE
    tsne = TSNE(perplexity=30, n_components=2, init='pca', n_iter=5000)
    plot_only = 2000
    low_dim_embs = tsne.fit_transform(final_embeddings[:plot_only,:])
    labels = [vocabulary[i] for i in range(plot_only)]
    plot_with_labels(low_dim_embs, labels)
```