RAJUL GUPTA HW 6 EE258 010828818

2. **Build your own RNN and train and measure the performance using the MNIST data-set.**

Source Code

```
from __future__ import print_function

import tensorflow as tf
from tensorflow.contrib import rnn

# Import MNIST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)


# Training Parameters
learning_rate = 0.002
training_steps = 10000
batch_size = 128
display_step = 200

# Network Parameters
num_input = 28 # MNIST data input (img shape: 28*28)
timesteps = 28 # timesteps
num_hidden = 128 # hidden layer num of features
num_classes = 10 # MNIST total classes (0-9 digits)

# tf Graph input
X = tf.placeholder("float", [None, timesteps, num_input])
Y = tf.placeholder("float", [None, num_classes])

# Define weights
weights = {
    'out': tf.Variable(tf.random_normal([num_hidden, num_classes]))
}
biases = {
    'out': tf.Variable(tf.random_normal([num_classes]))
}


def RNN(x, weights, biases):

    # Prepare data shape to match `rnn` function requirements
    # Current data input shape: (batch_size, timesteps, n_input)
    # Required shape: 'timesteps' tensors list of shape (batch_size,
n_input)

    # Unstack to get a list of 'timesteps' tensors of shape
(batch_size, n_input)
    x = tf.unstack(x, timesteps, 1)
```

```
    # Define a lstm cell with tensorflow
    lstm_cell = rnn.BasicLSTMCell(num_hidden, forget_bias=1.0)

    # Get lstm cell output
    outputs, states = rnn.static_rnn(lstm_cell, x, dtype=tf.float32)

    # Linear activation, using rnn inner loop last output
    return tf.matmul(outputs[-1], weights['out']) + biases['out']

logits = RNN(X, weights, biases)
prediction = tf.nn.softmax(logits)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer =
tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:

    # Run the initializer
    sess.run(init)

    for step in range(1, training_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Reshape data to get 28 seq of 28 elements
        batch_x = batch_x.reshape((batch_size, timesteps, num_input))
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X:
batch_x,
                                                                 Y:
batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for 128 mnist test images
    test_len = 128
    test_data = mnist.test.images[:test_len].reshape((-1, timesteps,
num_input))
    test_label = mnist.test.labels[:test_len]
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: test_data, Y: test_label}))
```

Output

```
Extracting /tmp/data/train-images-idx3-ubyte.gz
Extracting /tmp/data/train-labels-idx1-ubyte.gz
Extracting /tmp/data/t10k-images-idx3-ubyte.gz
Extracting /tmp/data/t10k-labels-idx1-ubyte.gz
Step 1, Minibatch Loss= 2.4430, Training Accuracy= 0.172
Step 200, Minibatch Loss= 1.8269, Training Accuracy= 0.492
Step 400, Minibatch Loss= 1.5858, Training Accuracy= 0.531
Step 600, Minibatch Loss= 1.3762, Training Accuracy= 0.578
Step 800, Minibatch Loss= 1.3985, Training Accuracy= 0.578
Step 1000, Minibatch Loss= 1.1324, Training Accuracy= 0.664
Step 1200, Minibatch Loss= 1.2087, Training Accuracy= 0.609
Step 1400, Minibatch Loss= 1.0200, Training Accuracy= 0.703
Step 1600, Minibatch Loss= 0.8794, Training Accuracy= 0.758
Step 1800, Minibatch Loss= 0.9698, Training Accuracy= 0.711
Step 2000, Minibatch Loss= 0.8113, Training Accuracy= 0.695
Step 2200, Minibatch Loss= 0.9789, Training Accuracy= 0.719
Step 2400, Minibatch Loss= 0.7549, Training Accuracy= 0.734
Step 2600, Minibatch Loss= 0.8442, Training Accuracy= 0.750
Step 2800, Minibatch Loss= 0.7803, Training Accuracy= 0.711
Step 3000, Minibatch Loss= 0.7811, Training Accuracy= 0.711
Step 3200, Minibatch Loss= 0.7105, Training Accuracy= 0.766
Step 3400, Minibatch Loss= 0.6378, Training Accuracy= 0.781
Step 3600, Minibatch Loss= 0.5158, Training Accuracy= 0.789
Step 3800, Minibatch Loss= 0.5505, Training Accuracy= 0.820
Step 4000, Minibatch Loss= 0.6523, Training Accuracy= 0.773
Step 4200, Minibatch Loss= 0.5077, Training Accuracy= 0.820
Step 4400, Minibatch Loss= 0.5642, Training Accuracy= 0.828
Step 4600, Minibatch Loss= 0.4113, Training Accuracy= 0.859
Step 4800, Minibatch Loss= 0.5170, Training Accuracy= 0.797
Step 5000, Minibatch Loss= 0.4607, Training Accuracy= 0.836
Step 5200, Minibatch Loss= 0.5135, Training Accuracy= 0.867
Step 5400, Minibatch Loss= 0.5950, Training Accuracy= 0.859
Step 5600, Minibatch Loss= 0.4522, Training Accuracy= 0.852
Step 5800, Minibatch Loss= 0.3427, Training Accuracy= 0.898
Step 6000, Minibatch Loss= 0.3268, Training Accuracy= 0.898
Step 6200, Minibatch Loss= 0.4513, Training Accuracy= 0.859
Step 6400, Minibatch Loss= 0.3670, Training Accuracy= 0.914
Step 6600, Minibatch Loss= 0.3068, Training Accuracy= 0.922
Step 6800, Minibatch Loss= 0.2806, Training Accuracy= 0.883
Step 7000, Minibatch Loss= 0.3138, Training Accuracy= 0.906
Step 7200, Minibatch Loss= 0.3641, Training Accuracy= 0.898
Step 7400, Minibatch Loss= 0.4025, Training Accuracy= 0.859
Step 7600, Minibatch Loss= 0.2231, Training Accuracy= 0.930
Step 7800, Minibatch Loss= 0.3419, Training Accuracy= 0.867
Step 8000, Minibatch Loss= 0.3425, Training Accuracy= 0.883
Step 8200, Minibatch Loss= 0.3338, Training Accuracy= 0.898
Step 8400, Minibatch Loss= 0.3814, Training Accuracy= 0.875
Step 8600, Minibatch Loss= 0.2636, Training Accuracy= 0.930
Step 8800, Minibatch Loss= 0.2011, Training Accuracy= 0.930
Step 9000, Minibatch Loss= 0.1418, Training Accuracy= 0.984
Step 9200, Minibatch Loss= 0.2995, Training Accuracy= 0.906
Step 9400, Minibatch Loss= 0.2842, Training Accuracy= 0.930
Step 9600, Minibatch Loss= 0.2671, Training Accuracy= 0.914
Step 9800, Minibatch Loss= 0.2638, Training Accuracy= 0.922
Step 10000, Minibatch Loss= 0.2476, Training Accuracy= 0.914
Optimization Finished!
Testing Accuracy: 0.960938
```

3. **Create a time series (see Fig 14-7 for an example). Create an RNN and train and measure the performance using the time series signal you created.**

Source Code and Output

```
from __future__ import print_function

import numpy as np

import matplotlib.pyplot as plt

from keras.models import Sequential

from keras.layers import Dense, LSTM

# since we are using stateful rnn tsteps can be set to 1

tsteps = 1

batch_size = 25

epochs = 3

# number of elements ahead that are used to make the prediction

lahead = 1

def gen_tanh_amp(amp=100, period=1000, x0=0, xn=50000, step=1, k=0.0001):
    """Generates an absolute cosine time series with the amplitude
    exponentially decreasing
    Arguments:
        amp: amplitude of the cosine function
        period: period of the cosine function
        x0: initial x of the time series
        xn: final x of the time series
        step: step of the time series discretization
        k: exponential rate
    """
    tanh = np.zeros(((xn - x0) * step, 1, 1))
    for i in range(len(tanh)):
        idx = x0 + i * step
        tanh[i, 0, 0] = amp * np.tanh(2 * np.pi * idx / period)
        tanh[i, 0, 0] = tanh[i, 0, 0] * np.exp(-k * idx)
    return tanh
```

```
print('Generating Data')

tanh = gen_tanh_amp()

print('Input shape:', tanh.shape)

print (tanh[1:5])
```

```
Generating Data
Input shape: (50000, 1, 1)
[[[ 0.62824743]]

 [[ 1.25631963]]

 [[ 1.88416704]]

 [[ 2.51174019]]]
```

```
expected_output = np.zeros((len(tanh), 1))

for i in range(len(tanh) - lahead):

    expected_output[i, 0] = np.mean(tanh[i + 1:i + lahead + 1])


print('Output shape')

print(expected_output.shape)

print (expected_output[1:5])
```

```
Output shape
(50000, 1)
[[ 1.25631963]
 [ 1.88416704]
 [ 2.51174019]
 [ 3.13898963]]
```

```
print('Creating Model')
model = Sequential()
model.add(LSTM(20,
        batch_input_shape=(batch_size, tsteps, 1),
        return_sequences=True,
        stateful=True))
model.add(LSTM(20,
        batch_input_shape=(batch_size, tsteps, 1),
        return_sequences=False,
        stateful=True))
model.add(Dense(1))
model.compile(loss='mse', optimizer='rmsprop')
print('Training')
for i in range(epochs):
   print('Epoch', i, '/', epochs)
   model.fit(tanh,
         expected_output,
         batch_size=batch_size,
         verbose=1,
         nb_epoch=1,
         shuffle=False)
   model.reset_states()
```

```
Creating Model
```

```
Training
Epoch 0 / 3
C:\Users\rajul\Anaconda3\lib\site-packages\keras\models.py:848: UserWarning: The `nb_epoch` argument in `fit` has been
`epochs`.
  warnings.warn('The `nb_epoch` argument in `fit` '
Epoch 1/1
50000/50000 [==============================] - 10s - loss: 674.8979
Epoch 1 / 3
Epoch 1/1
50000/50000 [==============================] - 8s - loss: 414.9826
Epoch 2 / 3
Epoch 1/1
50000/50000 [==============================] - 8s - loss: 281.2387
```

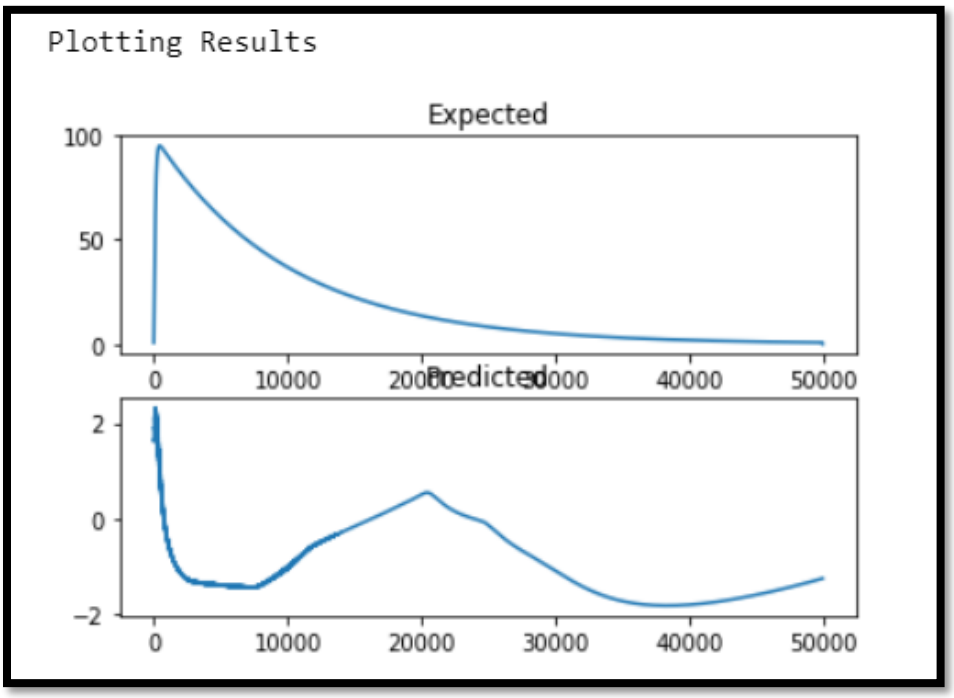
```
print('Predicting')

predicted_output = model.predict(tanh, batch_size=batch_size)

print (expected_output[1:5])

print (predicted_output[1:5])

print (predicted_output.shape)
```

```
Predicting
[[ 1.25631963]
 [ 1.88416704]
 [ 2.51174019]
 [ 3.13898963]]
[[ 1.62699127]
 [ 1.61865747]
 [ 1.61314189]
 [ 1.6100986 ]]
(50000, 1)
```

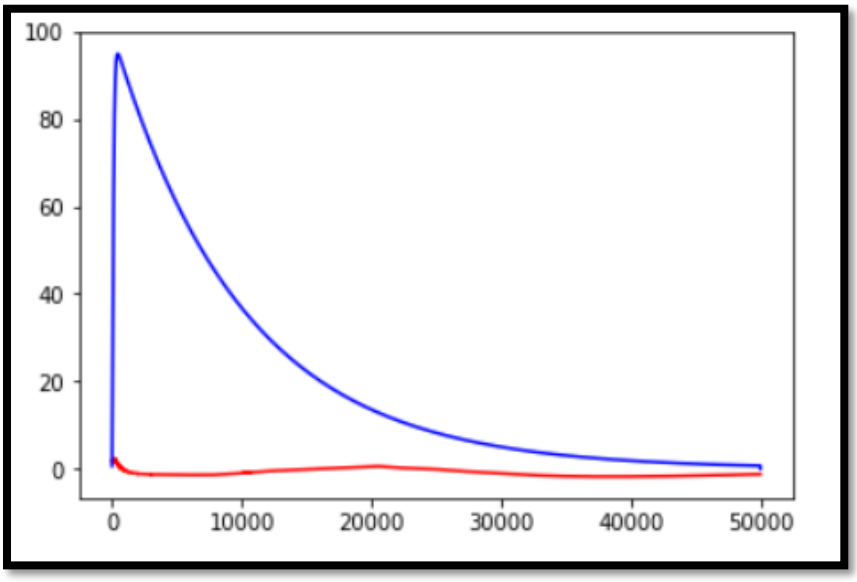
```
print (expected_output[4001:4005])

print (predicted_output[4001:4005])

print (expected_output[40001:40005])

print (predicted_output[40001:40005])
```

```
[[ 67.01859954]
 [ 67.01189802]
 [ 67.00519716]
 [ 66.99849698]]
[[-1.35776436]
 [-1.36440599]
 [-1.36872494]
 [-1.37070191]]
[[ 1.83119761]
 [ 1.8310145 ]
 [ 1.83083141]
 [ 1.83064834]]
[[-1.79970729]
 [-1.79968679]
 [-1.79966629]
 [-1.79964626]]
```

```
print('Plotting Results')
plt.subplot(2, 1, 1)
plt.plot(expected_output)
plt.title('Expected')
plt.subplot(2, 1, 2)
plt.plot(predicted_output)
plt.title('Predicted')
plt.show()
```




```
plt.plot(predicted_output, 'r-', expected_output, 'b-')
plt.show()
```



Red – Predicted Output

Blue – Expected Output