Feb 19 2018

Digit Classification using MNIST Dataset

Overview:

I used following two approaches:

1. Used **tensorflow library to build neural network** on given dataset (60000 training images and 10000 test images) and attained a modest accuracy of 96.26%. For memory computation, used memory-profiler / resource and for time computation, I used both time() and tensorflow's own timeline library to obtain time resource graph. I also explored tfprof library and think this would be the best option for a comprehensive profiling of both time and memory especially on large datasets.

2. Used **machine learning classification model (support vector machine or SVM)** on different groups of scaled-down datasets (5000 images) and plotting graph of memory / time consumed to extrapolate the profile for 60 Bn training dataset and 10 Bn test dataset

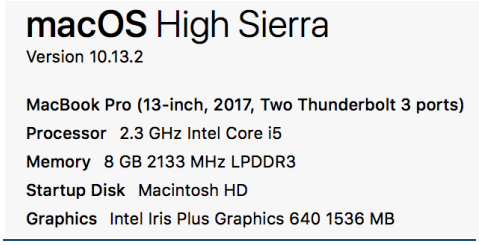
Approach 1: Using Tensorflow

About the Neural Network:

- Three hidden layers each with 400, 400, and 500 nodes respectively
- Trained the data in a batch size of 100
- No of epochs = 20
- Activation function used is Rectified Linear Unit (ReLU)
- Cost computation -> softmax_cross_entropy_with_logits
- Optimizer -> AdamOptimizer algorithm

Memory Usage on Local Machine

- **Local Machine Configuration**
    - 8GB RAM
    - 2.3 GHz 64-bit processor (i5)



- **Memory Usage: Storage**
    - Size of data**: ~12 MB**

```
train-images-idx3-ubyte.gz:  training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:  training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:   test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:   test set labels (4542 bytes)
```

Note – We have used the preloaded data in tensorflow library

- **Memory Usage: Training (using memory-profiler)**

```
Line #    Mem usage    Increment   Line Contents
================================================
    44  476.883 MiB  476.883 MiB   @profile
    45                             def train_neural_network(x):
    46  478.023 MiB  478.023 MiB     prediction = neural_network_model(x)
    47  478.453 MiB    0.430 MiB     cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = prediction, labels = y))
    48                               #learning_rate = 0.001 (by default which is anyway good so we are not changing that)
    49  481.070 MiB    2.617 MiB     optimizer = tf.train.AdamOptimizer().minimize(cost)
    50
    51                               #cycles of feedforward + backpropagation
    52  481.070 MiB    0.000 MiB     hm_epochs = 20
    53
    54  481.285 MiB    0.215 MiB     with tf.Session() as sess:
    55                                 #sess.run(tf.initialize_all_variables())
    56  497.496 MiB   16.211 MiB      sess.run(tf.global_variables_initializer())
    57  497.496 MiB -2624.938 MiB      for epoch in range(hm_epochs):
    58  497.496 MiB -2750.133 MiB        epoch_loss = 0
    59  501.855 MiB -1602955.852 MiB         for _ in range(int(mnist.train.num_examples/batch_size)):
    60  501.855 MiB -1600087.168 MiB           epoch_x, epoch_y = mnist.train.next_batch(batch_size)
    61  501.855 MiB -1600201.395 MiB           _, c = sess.run([optimizer, cost], feed_dict = {x: epoch_x, y: epoch_y})
    62  501.855 MiB -1600205.699 MiB           epoch_loss += c
    63  483.105 MiB -2999.965 MiB         print('Epoch', epoch, 'completed out of', hm_epochs, 'loss: ', epoch_loss)
    64  335.188 MiB -162.309 MiB       correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y,1))
    65
    66  335.312 MiB    0.125 MiB       accuracy = tf.reduce_mean(tf.cast(correct, 'float'))
    67
    68  437.445 MiB  102.133 MiB       print('Accuracy', accuracy.eval({x:mnist.test.images, y:mnist.test.labels}))
```

- **Memory Usage: Model (using memory-profiler)**

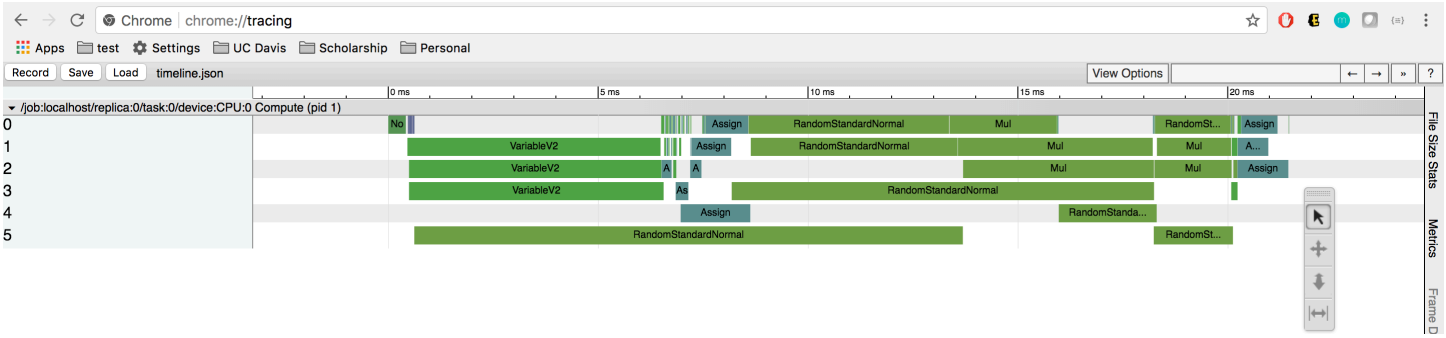
```
Line #    Mem usage    Increment   Line Contents
================================================
    18  476.883 MiB  476.883 MiB   @profile
    19                             def neural_network_model(data):
    20  477.098 MiB    0.215 MiB     hidden_1_layer = {'weights': tf.Variable(tf.random_normal([784, n_nodes_hl1])),
    21  477.223 MiB    0.125 MiB                       'biases': tf.Variable(tf.random_normal([n_nodes_hl1]))}
    22  477.352 MiB    0.129 MiB     hidden_2_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_hl2])),
    23  477.461 MiB    0.109 MiB                       'biases': tf.Variable(tf.random_normal([n_nodes_hl2]))}
    24  477.570 MiB    0.109 MiB     hidden_3_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl2, n_nodes_hl3])),
    25  477.684 MiB    0.113 MiB                       'biases': tf.Variable(tf.random_normal([n_nodes_hl3]))}
    26  477.797 MiB    0.113 MiB     output_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl3, n_classes])),
    27  477.902 MiB    0.105 MiB                     'biases': tf.Variable(tf.random_normal([n_classes]))}
    28
    29                               # Model: (input_data * weights) + biases
    30
    31  477.930 MiB    0.027 MiB     l1 = tf.add(tf.matmul(data, hidden_1_layer['weights']), hidden_1_layer['biases'])
    32  477.941 MiB    0.012 MiB     l1 = tf.nn.relu(l1)
    33
    34  477.957 MiB    0.016 MiB     l2 = tf.add(tf.matmul(l1, hidden_2_layer['weights']), hidden_2_layer['biases'])
    35  477.965 MiB    0.008 MiB     l2 = tf.nn.relu(l2)
    36
    37  477.992 MiB    0.027 MiB     l3 = tf.add(tf.matmul(l2, hidden_3_layer['weights']), hidden_3_layer['biases'])
    38  478.004 MiB    0.012 MiB     l3 = tf.nn.relu(l3)
    39
    40  478.023 MiB    0.020 MiB     output = tf.matmul(l3, output_layer['weights']) + output_layer['biases']
    41
    42  478.023 MiB    0.000 MiB     return output
```

- **Overall Memory Usage (using resource)**

  69488640 Bytes = 66.3 MB

Time Usage on Local Machine

- **Using timeline ~ 22 milliseconds for tensorflow session**



- **Using time() ~ 135 seconds for overall model implementation**

Time and resources required to handle a training set of 60B images and a test set of 10B

I am going to assess the performance of this model (both time and resources) on the newly launched Google TPUs (Tensor Processing Units)

- **How fast is TPU compared to GPU?**

  Following calculations are based on this article where a Google spokesperson said, "To put this into perspective, our new large-scale translation model takes a full day to train on 32 of the world's best commercially available GPU's—while one 1/8th of a TPU pod can do the job in an afternoon," Google wrote in a statement.

  Full Day = 24 hours * 32 GPUs = 768 hours / GPU
  1/8TPU * 8 hours = 1 hour / TPU

  ➔ **1 hour / TPU = 768 hours / GPU.** ---------- 1

- **How fast is GPU compared to CPU?**

  Based on this blog which is a conservative estimate of a GPU's speed by Intel,

  ➔ **1 hour / GPU = 14 hours / CPU.** ---------- 2

  From 1 and 2, we can derive

1 hour / TPU = 10752 hours / CPU ---------- 3

Based on all of this information and our own results from the model implementation, it is safe to assume that running the same model on TPU will take following time:

135 seconds * 1 Million / (10752 * 3600) = 3.49 seconds ☺

This is quite an amazing result that even after scaling up the data by a million, the same model can be implemented at a speed of ~3.5 seconds which is 1/38th times the speed of a normal CPU (in our case, a 2.3 GHz 64-bit processor (i5)).

Further, this article suggests that one TPU can handle the processing of over 1M regular Google images per day (which have far more memory consumption due to enhanced quality, color, density etc. compared to the MNIST greyscale images). Further, a cloud TPU resource accelerates the performance of linear algebra computation, minimizes the time-to-accuracy training large, complex neural network models like in our case. Models that previously took weeks to train on other hardware platforms can converge in hours on TPUs. Following table is a comparative snapshot which gives the pricing as well.

	Cloud TPU (4x TPU2)	AWS P3.2xlarge (1xV100)	AWS P3.8xlarge (4xV100)	AWS P3.16xlarge (8xV100)
RN50 Image/Second	1369	819	3242	6309
RN50 Time to Train (hr)	23.4	39.1	9.9	5.1
$/Hr	$6.88	$3.06	$12.24	$24.48
$ to Train ImageNet, 90 epochs	$161	$120	$121	$124
Availability	Beta	Now	Now	Now

Moor Insights & Strategy, Google, NVIDIA, Amazon AWS

Table 1: Comparing Google Cloud TPU vs. NVIDIA's Volta (V100) GPU for training convolutional[+]

Recommendation

In conclusion, based on the research, I recommend using Google cloud TPUs for better performance and scalability in minimum time even though the cost is a bit on the higher side. Here is the official tutorial link we found for MNIST analysis on TPU

Tensorflow Code:

```
import resource
from memory_profiler import profile
from time import time
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
from tensorflow.python.client import timeline

t00 = time()
```

```
mnist = input_data.read_data_sets("/tmp/data", one_hot = True)
n_nodes_hl1 = 400
n_nodes_hl2 = 400
n_nodes_hl3 = 500
n_classes = 10
batch_size = 100
x = tf.placeholder('float', [None, 784])
y = tf.placeholder('float')

@profile
def neural_network_model(data):
  hidden_1_layer = {'weights': tf.Variable(tf.random_normal([784, n_nodes_hl1])),
                    'biases': tf.Variable(tf.random_normal([n_nodes_hl1]))}
  hidden_2_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_hl2])),
                    'biases': tf.Variable(tf.random_normal([n_nodes_hl2]))}
  hidden_3_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl2, n_nodes_hl3])),
                    'biases': tf.Variable(tf.random_normal([n_nodes_hl3]))}
  output_layer = {'weights': tf.Variable(tf.random_normal([n_nodes_hl3, n_classes])),
                    'biases': tf.Variable(tf.random_normal([n_classes]))}

  # Model: (input_data * weights) + biases

  l1 = tf.add(tf.matmul(data, hidden_1_layer['weights']), hidden_1_layer['biases'])
  l1 = tf.nn.relu(l1)

  l2 = tf.add(tf.matmul(l1, hidden_2_layer['weights']), hidden_2_layer['biases'])
  l2 = tf.nn.relu(l2)

  l3 = tf.add(tf.matmul(l2, hidden_3_layer['weights']), hidden_3_layer['biases'])
  l3 = tf.nn.relu(l3)

  output = tf.matmul(l3, output_layer['weights']) + output_layer['biases']

  return output

@profile
def train_neural_network(x):
  prediction = neural_network_model(x)
  t11 = time()
  cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits = prediction, labels = y))
  #learning_rate = 0.001 (taking the default value. Changing this does not boost the accuracy much)
  optimizer = tf.train.AdamOptimizer().minimize(cost)
  hm_epochs = 20

  with tf.Session() as sess:
    run_options = tf.RunOptions(trace_level=tf.RunOptions.FULL_TRACE)
    run_metadata = tf.RunMetadata()
    sess.run(tf.global_variables_initializer(), options=run_options, run_metadata=run_metadata)

    for epoch in range(hm_epochs):
```

```
    epoch_loss = 0
    for _ in range(int(mnist.train.num_examples/batch_size)):
      epoch_x, epoch_y = mnist.train.next_batch(batch_size)
      _, c = sess.run([optimizer, cost], feed_dict = {x: epoch_x, y: epoch_y})
      epoch_loss += c
    print('Epoch', epoch, 'completed out of', hm_epochs, 'loss: ', epoch_loss)
  correct = tf.equal(tf.argmax(prediction, 1), tf.argmax(y,1))

  accuracy = tf.reduce_mean(tf.cast(correct, 'float'))

  print('Accuracy', accuracy.eval({x:mnist.test.images, y:mnist.test.labels}))
  tl = timeline.Timeline(run_metadata.step_stats)
  ctf = tl.generate_chrome_trace_format()
  with open('timeline.json', 'w') as f:
    f.write(ctf)

m1 = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
train_neural_network(x)
m2 = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss - m1

t22 = time()

print ("start time: " + str(t00))
print ("end time: " + str(t22))
print ("memory usage: " + str(m2))
```

Tensorflow Output:

```
use: SSE4.2 AVX AVX2 FMA
('Epoch', 0, 'completed out of', 20, 'loss: ', 1648911.9029693604)
('Epoch', 1, 'completed out of', 20, 'loss: ', 371495.22474861145)
('Epoch', 2, 'completed out of', 20, 'loss: ', 209670.77573281527)
('Epoch', 3, 'completed out of', 20, 'loss: ', 127917.65152192116)
('Epoch', 4, 'completed out of', 20, 'loss: ', 80780.739795818008)
('Epoch', 5, 'completed out of', 20, 'loss: ', 51415.149344441743)
('Epoch', 6, 'completed out of', 20, 'loss: ', 34864.970039913082)
('Epoch', 7, 'completed out of', 20, 'loss: ', 24879.979579467446)
('Epoch', 8, 'completed out of', 20, 'loss: ', 20328.23952158451)
('Epoch', 9, 'completed out of', 20, 'loss: ', 14718.877096552435)
('Epoch', 10, 'completed out of', 20, 'loss: ', 13380.721331574097)
('Epoch', 11, 'completed out of', 20, 'loss: ', 11586.850689145327)
('Epoch', 12, 'completed out of', 20, 'loss: ', 11925.184685918881)
('Epoch', 13, 'completed out of', 20, 'loss: ', 11868.325995282361)
('Epoch', 14, 'completed out of', 20, 'loss: ', 10741.610335360652)
('Epoch', 15, 'completed out of', 20, 'loss: ', 8585.6293483148511)
('Epoch', 16, 'completed out of', 20, 'loss: ', 8170.4177761713509)
('Epoch', 17, 'completed out of', 20, 'loss: ', 9945.5633442637336)
('Epoch', 18, 'completed out of', 20, 'loss: ', 8624.9282187074423)
('Epoch', 19, 'completed out of', 20, 'loss: ', 9204.0658708125356)
('Accuracy', 0.96259999)
```

Approach 2: Using Machine Learning (SVM)

Importing the dependencies :

In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt, matplotlib.image as mpimg
from sklearn.model_selection import train_test_split
from sklearn import svm
%matplotlib inline
```

Loading the data

- We use panda's read_csv to read train.csv into a dataframe. We have converted the MNSIT dataset to a csv file for easy processing.
- We also do a train_test_split to break our data into two sets, one for training and one for testing. This let's us measure how well our model was trained by later inputting some known test data.

For the sake of time, we're only using 5000 images. The whole dataset was taking more than 10 hours to train.

In [3]:
```
labeled_images = pd.read_csv('./train.csv')   # converted the MNSIT dataset to a csv
images = labeled_images.iloc[0:5000,1:]
labels = labeled_images.iloc[0:5000,:1]

# suing sklearn's train_test split
train_images, test_images,train_labels, test_labels = train_test_split(images, labels, train_size=0.8, random_state=0)
```

```
/home/ec2-user/.env/lib64/python3.6/dist-packages/sklearn/model_selection/_split.py:2010: FutureWarning: From version 0.21, tes
t_size will always complement train_size unless both are specified.
  FutureWarning)
```

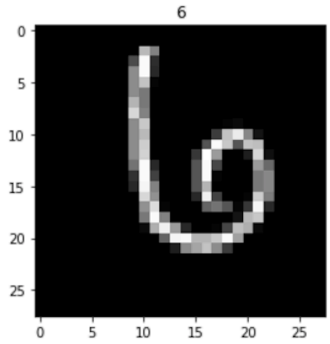
Viewing an Image

- Since the image is currently one-dimension, we load it into a numpy array and reshape it so that it is two-dimensional (28x28 pixels)
- Then, we plot the image and label with matplotlib

You can change the value of variable *i* to check out other images and labels.

In [4]:
```
i=1
img=train_images.iloc[i].as_matrix()
img=img.reshape((28,28))
plt.imshow(img,cmap='gray')
plt.title(train_labels.iloc[i,0])
```

Out[4]:
```
Text(0.5,1,'6')
```



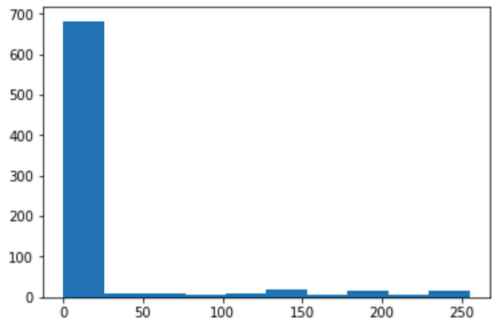
Examining the Pixel Values

Note that these images aren't actually black and white (0,1). They are gray-scale (0-255).

- A histogram of this image's pixel values shows the range.

In [5]:
```
plt.hist(train_images.iloc[i])
```

Out[5]:
```
(array([682.,   9.,  10.,   7.,  10.,  18.,   7.,  17.,   7.,  17.]),
 array([  0. ,  25.5,  51. ,  76.5, 102. , 127.5, 153. , 178.5, 204. ,
        229.5, 255. ]),
 <a list of 10 Patch objects>)
```



Training our model

- First, we use the sklearn.svm module to create a vector classifier.
- Next, we pass our training images and labels to the classifier's fit method, which trains our model.
- Finally, the test images and labels are passed to the score method to see how well we trained our model. Fit will return a float between 0-1 indicating our accuracy on the test data set

Try playing with the parameters of svm.SVC to see how the results change.

In [6]:
```
import time
```

In [ ]:
```
start = time.time()

clf = svm.SVC()
clf.fit(train_images, train_labels.values.ravel())
clf.score(test_images,test_labels)


end = time.time()
```

In [8]:
```
print("Training time for 5k images is : {}".format(end - start))
```

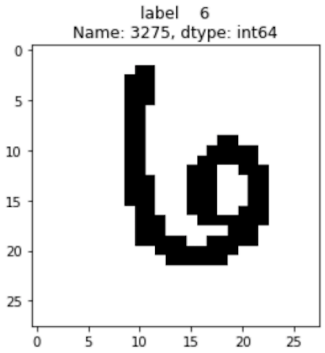
```
Training time for 5k images is : 30.257941007614136
```

How did our model do?

- To make this easy, any pixel with a value simply becomes 1 and everything else remains 0.
- We'll plot the same image again to see how it looks now that it's black and white. Look at the histogram now.

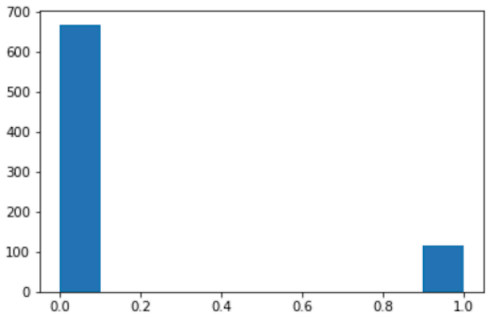
In [9]:
```
test_images[test_images>0]=1
train_images[train_images>0]=1

img=train_images.iloc[i].as_matrix().reshape((28,28))
plt.imshow(img,cmap='binary')
plt.title(train_labels.iloc[i])
```



In [10]:
```
plt.hist(train_images.iloc[i])
```

Out[10]:
```
(array([668.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0., 116.]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <a list of 10 Patch objects>)
```



We follow the same procedure as before, but now our training and test sets are black and white instead of gray-scale. Our score still isn't great, but it's a huge improvement.

In [11]:
```
start = time.time()

clf = svm.SVC()
clf.fit(train_images, train_labels.values.ravel())
clf.score(test_images,test_labels)


end = time.time()


print("The training time for 5k black and white images is {}".format(end-start))
```

```
The training time for 5k black and white images is 9.086803913116455
```

We improved the training time by 70%

Labelling the test data

Now for those making competition submissions, we can load and predict the unlabeled data from test.csv. Again, for time we're just using the first 5000 images. We then output this data to a results.csv for competition submission.

In [15]:
```
start = time.time()


test_data=pd.read_csv('./test.csv')
test_data[test_data>0]=1
results=clf.predict(test_data[0:5000])


end = time.time()

print("The testing time for 5k images is {}".format(end-start))
```

```
The testing time is 16.32559823989868
```