CEE 498 Applied Machine Learning - HW8

Rini Jasmine Gladstone (rjg7) and Maksymilian Podraza (podraza3)

May 7 2020

1 Problem 1

We are using tensorflow for our homework and we downloaded the tutorial code for MNIST in tensorflow, trained it and ran a classifier. The tutorial code was downloaded from https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d.

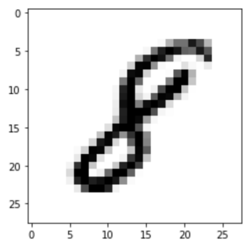
Please find below the screenshots of the code and the accuracy results.

```
In [3]: import tensorflow as tf
        (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
```

In [4]:
```python
import matplotlib.pyplot as plt
image_index = 7777 # You may select anything up to 60,000
print(y_train[image_index]) # The label is 8
plt.imshow(x_train[image_index], cmap='Greys')
```

```
8
```

Out[4]:
```
<matplotlib.image.AxesImage at 0x7f866a2cab70>
```



In [5]:
```python
x_train.shape
```

Out[5]:
```
(60000, 28, 28)
```

In [6]:
```python
# Reshaping the array to 4-dims so that it can work with the Keras API
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
input_shape = (28, 28, 1)
# Making sure that the values are float so that we can get decimal points after division
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
# Normalizing the RGB codes by dividing it to the max RGB value.
x_train /= 255
x_test /= 255
print('x_train shape:', x_train.shape)
print('Number of images in x_train', x_train.shape[0])
print('Number of images in x_test', x_test.shape[0])
```

```
x_train shape: (60000, 28, 28, 1)
Number of images in x_train 60000
Number of images in x_test 10000
```

In [8]:
```python
# Importing the required Keras modules containing model and layers
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D
# Creating a Sequential Model and adding the layers
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten()) # Flattening the 2D arrays for fully connected layers
model.add(Dense(128, activation=tf.nn.relu))
model.add(Dropout(0.2))
model.add(Dense(10,activation=tf.nn.softmax))
```

```
WARNING:tensorflow:From /home/podrazque/.local/lib/python3.6/site-packages/keras/backend/tensorflo
w_backend.py:4070: The name tf.nn.max_pool is deprecated. Please use tf.nn.max_pool2d instead.

Using TensorFlow backend.
```

In [9]:
```python
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
model.fit(x=x_train,y=y_train, epochs=10)
```

```
WARNING:tensorflow:From /home/podrazque/.local/lib/python3.6/site-packages/keras/backend/tensorflo
w_backend.py:422: The name tf.global_variables is deprecated. Please use tf.compat.v1.global_varia
bles instead.

Epoch 1/10
60000/60000 [==============================] - 11s 183us/step - loss: 0.2066 - accuracy: 0.9388
Epoch 2/10
60000/60000 [==============================] - 11s 180us/step - loss: 0.0819 - accuracy: 0.9748
Epoch 3/10
60000/60000 [==============================] - 11s 184us/step - loss: 0.0574 - accuracy: 0.9814
Epoch 4/10
60000/60000 [==============================] - 11s 188us/step - loss: 0.0448 - accuracy: 0.9852
Epoch 5/10
60000/60000 [==============================] - 11s 186us/step - loss: 0.0362 - accuracy: 0.9880
Epoch 6/10
60000/60000 [==============================] - 11s 183us/step - loss: 0.0283 - accuracy: 0.9908
Epoch 7/10
60000/60000 [==============================] - 11s 183us/step - loss: 0.0256 - accuracy: 0.9913
Epoch 8/10
60000/60000 [==============================] - 11s 186us/step - loss: 0.0228 - accuracy: 0.9921
Epoch 9/10
60000/60000 [==============================] - 11s 179us/step - loss: 0.0200 - accuracy: 0.9931
Epoch 10/10
```

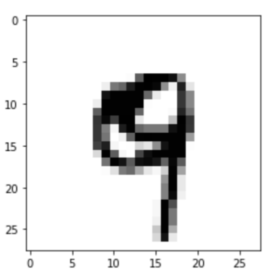
In [10]:
```python
model.evaluate(x_test, y_test)
```

```
10000/10000 [==============================] - 0s 35us/step
```

Out[10]:
```
[0.06483541152450016, 0.9836000204086304]
```

In [12]:
```python
image_index = 4444
plt.imshow(x_test[image_index].reshape(28, 28),cmap='Greys')
pred = model.predict(x_test[image_index].reshape(1, 28, 28, 1))
print(pred.argmax())
```

```
9
```



We got an accuracy of **98.36%** in the testing data.

2 Problem 2

We built the neural network as mentioned in section 17.2.1

```
##### Initialization ##########################################################
num_epochs = 500
LR = 0.0001
nunit = 32
batch_size = 10000
```

```
weights = {
    # Convolution Layers
    'c1': tf.get_variable('W1', shape=(5,5,1,20), \
            initializer=tf.contrib.layers.xavier_initializer()),
    'c2': tf.get_variable('W2', shape=(5,5,20,50), \
            initializer=tf.contrib.layers.xavier_initializer()),
    'c3': tf.get_variable('W3', shape=(4,4,50,500), \
            initializer=tf.contrib.layers.xavier_initializer()),
    'c4': tf.get_variable('W4', shape=(1,1,500,10), \
            initializer=tf.contrib.layers.xavier_initializer()),
}
biases = {
    # Convolution Layers
    'c1': tf.get_variable('B1', shape=(20), initializer=tf.zeros_initializer()),
    'c2': tf.get_variable('B2', shape=(50), initializer=tf.zeros_initializer()),
    'c3': tf.get_variable('B3', shape=(500), initializer=tf.zeros_initializer()),
    'c4': tf.get_variable('B4', shape=(10), initializer=tf.zeros_initializer()),
}
```

```
def conv2d_1(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1,1,1,1], padding='VALID')
    x = tf.nn.bias_add(x, b)
    return x
def conv2d_2(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1,1,1,1], padding='VALID')
    x = tf.nn.bias_add(x, b)
    return tf.nn.relu(x)
def conv2d_3(x, W, b, strides=1):
    x = tf.nn.conv2d(x, W, strides=[1,1,1,1], padding='VALID')
    x = tf.nn.bias_add(x, b)
    return tf.nn.softmax(x)
```

Figure 1: Setting up the weights and biases

```python
def conv_net(data, weights, biases, training=False):
    # Convolution layers
    conv1 = conv2d_1(data, weights['c1'], biases['c1'])
    pool1 = tf.nn.max_pool(conv1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')
    conv2 = conv2d_1(pool1, weights['c2'], biases['c2'])
    pool2 = tf.nn.max_pool(conv2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')
    conv3 = conv2d_2(pool2, weights['c3'], biases['c3'])
    conv4 = conv2d_3(conv3, weights['c4'], biases['c4'])

    out = conv4 # [10]
    return out
```

```python
Xtrain = tf.placeholder(tf.float32, shape=(None, 28, 28, 1))
logits = conv_net(Xtrain, weights, biases)
#logits = conv_net_dropout(Xtrain, weights, biases)
```

```python
n_classes=10
ytrain = tf.placeholder(tf.float32, shape=(None, n_classes))
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=logits,
                                                                   labels=ytrain))
```

```python
optimizer = tf.train.GradientDescentOptimizer(0.1)
#optimizer = tf.train.MomentumOptimizer(learning_rate=0.1,momentum=0.9)
#optimizer = tf.train.AdamOptimizer(1e-3)
train_op = optimizer.minimize(loss)
```

```python
test_images = tf.placeholder(tf.float32, shape=(None, 28, 28, 1))
tests_labels=tf.placeholder(tf.float32, shape=(None, n_classes))
test_predictions = tf.nn.softmax(conv_net(test_images, weights, biases))
#acc,acc_op = tf.metrics.accuracy(predictions=tf.argmax(test_predictions,1),
 #                                labels=tests_labels )
```

Figure 2: Creating the neural network architecture

```python
import sklearn.preprocessing as skp
enc = skp.OneHotEncoder(handle_unknown='ignore')
enc.fit(train_set[1].reshape(-1,1))
Ytrain_data=enc.transform(train_set[1].reshape(-1,1)).toarray()
```

```python
num_epochs = 50
batch_size=5000
sess = tf.Session()
sess.run(tf.global_variables_initializer())
sess.run(tf.local_variables_initializer())
for epochs in range(num_epochs):
    res_avg = 0.
    for i in range(0, 60000, batch_size):
        feed_dict={Xtrain: Train_data[i:i+batch_size].reshape(-1,28,28,1), ytrain:Ytrain_data[i:i+batch_size].reshape(-
        _,res = sess.run([train_op,loss], feed_dict=feed_dict)
        res_avg += np.sum(res)
    if epochs%5==0.:
        print(int(60000/batch_size)*epochs, res_avg/60000)
        print(np.mean(res))
```

Figure 3: Training the neural network

With this neural network, trained with SGD and mini batch, we got an accuracy of **93.98 %**.

```
In [126]: Test_data = test_set[0].reshape(10000,28,28)
          pred=sess.run(test_predictions,feed_dict={test_images:Test_data.reshape(-1,28,28,1)})

In [127]: pred_label=[]
          for i in range(10000):
              pred_label.append(np.argmax(pred[i]))

In [128]: pred_label=np.array(pred_label)

In [129]: import sklearn.metrics
          accuracy = sklearn.metrics.accuracy_score(test_set[1], pred_label)

In [130]: accuracy

Out[130]: 0.9398
```

Figure 4: Accuracy on the test data - SGD+Mini Batch

2.1 Part A

To improve the accuracy, we added momentum to the optimizer. We used a momentum of 0.9.

```
In [44]: #optimizer = tf.train.GradientDescentOptimizer(0.1)
         optimizer = tf.train.MomentumOptimizer(learning_rate=0.1,momentum=0.9)
         #optimizer = tf.train.AdamOptimizer(1e-3)
         train_op = optimizer.minimize(loss)
```

Figure 5: Optimizer changed to include the momentum

For this, we got an accuracy of **98.23%**.

```
In [48]: Test_data = test_set[0].reshape(10000,28,28)
         pred=sess.run(test_predictions,feed_dict={test_images:Test_data.reshape(-1,28,28,1)})

In [49]: pred_label=[]
         for i in range(10000):
             pred_label.append(np.argmax(pred[i]))

In [50]: pred_label=np.array(pred_label)

In [51]: import sklearn.metrics
         accuracy = sklearn.metrics.accuracy_score(test_set[1], pred_label)

In [52]: accuracy

Out[52]: 0.9823
```

Figure 6: Accuracy for the momentum

2.2 Part B

We added two drop out layers, one after first convolutional layer and one after second convolutional layer.

```python
def conv_net_dropout(data, weights, biases, training=False):
    # Convolution layers
    conv1 = conv2d_1(data, weights['c1'], biases['c1'])
    dropout1 = tf.nn.dropout(conv1,0.8)
    pool1 = tf.nn.max_pool(dropout1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')
    conv2 = conv2d_1(pool1, weights['c2'], biases['c2'])
    dropout2 = tf.nn.dropout(conv2,0.8)
    pool2 = tf.nn.max_pool(dropout2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')
    conv3 = conv2d_2(pool2, weights['c3'], biases['c3'])
    conv4 = conv2d_3(conv3, weights['c4'], biases['c4'])

    out = conv4 # [10]
    return out
```

Figure 7: Dropout layers added

We ran this network with SGD (no momentum). The accuracy is **94.65%**.

```
    for i in range(10000):
        pred_label.append(np.argmax(pred[i]))

In [25]: pred_label=np.array(pred_label)

In [26]: import sklearn.metrics
         accuracy = sklearn.metrics.accuracy_score(test_set[1], pred_label)

In [27]: accuracy

Out[27]: 0.9465
```

Figure 8: Accuracy for SGD+dropout

2.3 Part C

We tried two techniques to improve the accuracy. Normalized layer and data augmentation.

We added 3 normalized layers, one after each convolutional layer.

```python
epsilon = 1e-3
def conv_net(data, weights, biases, training=False):
    # Convolution layers
    conv1 = conv2d_1(data, weights['c1'], biases['c1'])
    batch_mean1, batch_var1 = tf.nn.moments(conv1,[0])
    scale1 = tf.Variable(tf.ones([20]))
    beta1 = tf.Variable(tf.zeros([20]))
    normalized_layer_1 = tf.nn.batch_normalization(conv1, batch_mean1,batch_var1,beta1,scale1,epsilon)
    pool1 = tf.nn.max_pool(normalized_layer_1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')
    conv2 = conv2d_1(pool1, weights['c2'], biases['c2'])
    batch_mean2, batch_var2 = tf.nn.moments(conv2,[0])
    scale2 = tf.Variable(tf.ones([50]))
    beta2 = tf.Variable(tf.zeros([50]))
    normalized_layer_2 = tf.nn.batch_normalization(conv2, batch_mean2,batch_var2,beta2,scale2,epsilon)
    pool2 = tf.nn.max_pool(normalized_layer_2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')
    conv3 = conv2d_2(pool2, weights['c3'], biases['c3'])
    batch_mean3, batch_var3 = tf.nn.moments(conv3,[0])
    scale3 = tf.Variable(tf.ones([500]))
    beta3 = tf.Variable(tf.zeros([500]))
    normalized_layer_3 = tf.nn.batch_normalization(conv3, batch_mean3,batch_var3,beta3,scale3,epsilon)
    conv4 = conv2d_3(normalized_layer_3, weights['c4'], biases['c4'])
    out = conv4 # [10]
    return out
```

Figure 9: Normalized layers added

We ran it for SGD+Momentum (no dropout). We got an accuracy of **99.16%**. Adding a normalized layer improved the accuracy on the testing data.

```
In [31]:
         Test_data = test_set[0].reshape(10000,28,28)
         pred=sess.run(test_predictions,feed_dict={test_images:Test_data.reshape(-1,28,28,1)})

In [32]: pred_label=[]
         for i in range(10000):
             pred_label.append(np.argmax(pred[i]))

In [33]: pred_label=np.array(pred_label)

In [34]: import sklearn.metrics
         accuracy = sklearn.metrics.accuracy_score(test_set[1], pred_label)

In [35]: accuracy

Out[35]: 0.9916
```

Figure 10: Accuracy for the SGD +momentum + normalized layer

Next,we tried data augmentation. We rotated all the training images by 90 degrees and created additional 60,000 training images. We also flipped the training images and created another additional training images. Now our training dataset is of size 180,000.

```
def rotate_all(data_all, degrees, flat="True"):
    return_rotated = []
    for img in data_all:
        if flat == "True":
            return_rotated.append(crop_center(rotate(img, degrees), 28, 28).flatten())
        else:
            return_rotated.append(crop_center(rotate(img, degrees), 28, 28))
    return return_rotated
def rotate_single(data_single, degrees):
    return crop_center(rotate(data_single, degrees), 28, 28)
def flip_single(data_single, flag="h"):
    if flag == "h":
        return np.flip(data_single, axis=1)
    elif flag == "v":
        return np.flip(data_single, axis=0)
    elif flag == "hv":
        return np.flip(data_single, axis=None)
    else:
        return "error"
def flip_all(data_all, flag="h", flat="True"):
    return_flipped = []
    for img in data_all:
        if flat == "True":
            return_flipped.append(flip_single(img, flag=flag).flatten())
        else:
            return_flipped.append(flip_single(img, flag=flag))
    return return_flipped
def flatten(data):
    returned_flat = []
    for img in data:
        returned_flat.append(img.flatten())
    return returned_flat
def append_augment(original, augmented):
    appended = []
    for img in original:
        appended.append(img)
    for img in augmented:
        appended.append(img)
    return appended
```

```python
def append_labels(labels, labels_):
    labels_appended = []
    for label in labels:
        labels_appended.append(label)
    for label in labels_:
        labels_appended.append(label)
    return labels_appended
```

```python
from scipy.ndimage.interpolation import rotate
training_rotated = rotate_all(Train_data, 90)
```

```python
training_flipped = flip_all(Train_data, "h")
```

```python
flattened = flatten(Train_data)
```

```python
total = append_augment(flattened, training_flipped)
```

```python
total=append_augment(total, training_rotated)
```

```python
len(total)
```

```
180000
```

```python
labels = append_labels(train_set[1],train_set[1])
```

```python
labels = append_labels(labels,train_set[1])
```

```python
len(labels)
```

```
180000
```

For this, we got an accuracy of **98.23%** which is the same as the SGD+Momentum. Therefore, data augmentation didnt help to improve the accuracy for us.

```python
Test_data = test_set[0].reshape(10000,28,28)
pred=sess.run(test_predictions,feed_dict={test_images:Test_data.reshape(-1,28,28,1)})
```

```python
pred_label=[]
for i in range(10000):
    pred_label.append(np.argmax(pred[i]))
```

```python
pred_label=np.array(pred_label)
```

```python
import sklearn.metrics
accuracy = sklearn.metrics.accuracy_score(test_set[1], pred_label)
```

```python
accuracy
```

```
0.9823
```

Figure 11: Accuracy for the SGD +momentum + Data Augmentation