Deep Convolutional Neural Network for Image Recognition with the MNIST Set

2) Implementation with TF 1.x

ME225 Winter 2023, Vedad Bassari Last edit: 3/7

32-3x3 filters

Declatations and Relevant Libraries

```
#Declarations and relevant libraries
import numpy as np
import math
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report

#Import pyplot library for plotting kernels and feature maps
plt.style.use('seaborn-poster')
%matplotlib inline
tf.compat.v1.disable_eager_execution()
sess = tf.compat.v1.InteractiveSession()
```

Load Data-set and Format Data

```
(train_images, train_labels), (test_images,test_labels) = keras.datasets.mnist.load_data() #Splitting MNIST data-set for training
train_images = train_images.astype("float32")/255 #Normalize data between [0,1] for ease of use
test_images = test_images.astype("float32")/255
train_images = np.expand_dims(train_images, -1) #Add one dimension for compatability with conv2d() function
test_images = np.expand_dims(test_images, -1)
train_labels = keras.utils.to_categorical(train_labels, 10) #Format the labels for compatibility with the training module
test_labels = keras.utils.to_categorical(test_labels, 10)
print(train_images.shape) #Print out the shape of input data for confirmation
print(test_images.shape)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
(60000, 28, 28, 1)
(10000, 28, 28, 1)
```

Define Tensorflow Placeholders

```
x = tf.compat.v1.placeholder(tf.float32,shape=[None,28,28,1]) #Define the structures of the input and output layers
y = tf.compat.v1.placeholder(tf.float32,shape=[None,10]) #10 classes for the classified digits
```

Define Tensorflow Variables

```
#Weights defined with Xavier initialization
w1 = tf.Variable(tf.initializers.GlorotUniform()(shape=(3,3,1,32))) #First convolution
w2 = tf.Variable(tf.initializers.GlorotUniform()(shape=(3,3,32,64))) #Second convoluton
w3 = tf.Variable(tf.initializers.GlorotUniform()(shape=(3,3,64,128))) #Third convolution
w4 = tf.Variable(tf.initializers.GlorotUniform()(shape=(4*4*128,128))) #Flatting Layer and first dense layer
w5 = tf.Variable(tf.initializers.GlorotUniform()(shape=(128,128))) #Second dense layer
w6 = tf.Variable(tf.initializers.GlorotUniform()(shape=(128,10))) #Output layer

#Biases defined with Xavier initialization
b1 = tf.Variable(tf.initializers.GlorotUniform()(shape=(1,32)))
b2 = tf.Variable(tf.initializers.GlorotUniform()(shape=(1,64)))
b3 = tf.Variable(tf.initializers.GlorotUniform()(shape=(1,128)))
b4 = tf.Variable(tf.initializers.GlorotUniform()(shape=(1,128)))
b5 = tf.Variable(tf.initializers.GlorotUniform()(shape=(1,128)))
b6 = tf.Variable(tf.initializers.GlorotUniform()(shape=(1,10)))
```

Define Network Layers

```
#First convolution layer
conv1 = tf.nn.conv2d(x, w1, strides=[1,2,2,1], padding='SAME') #First convolution: 32-3x3 filters, 2x2 stride
```

```
conv1 = tf.add(conv1,b1) #Add bias
conv1 = tf.nn.relu(conv1) #ReLU activation for convolution
mean_x, std_x = tf.nn.moments(conv1, axes = [0, 1, 2], keepdims=True)
conv1 = tf.nn.batch_normalization(conv1, mean_x, std_x, None, None, 1e-12) #Batch normalization after convolution

conv1 = tf.nn.max_pool(conv1, ksize=[1,3,3,1], strides=[1,2,2,1],padding='SAME') #First max-pooling: 3x3 pool size, 2x2 stride

#Second convolution layer
conv2 = tf.nn.conv2d(conv1, w2, strides=[1,1,1,1], padding='SAME') #Second convolution: 32x64-2x2 filters, 1x1 stride
conv2 = tf.add(conv2,b2) #Add bias
conv2 = tf.nn.relu(conv2) #ReLU activation for convolution
mean_x, std_x = tf.nn.moments(conv2, axes = [0, 1, 2], keepdims=True)
conv2 = tf.nn.batch_normalization(conv2, mean_x, std_x, None, None, 1e-12) #Batch normalization after convolution

#Third convolution layer
conv3 = tf.nn.conv2d(conv2, w3, strides=[1,1,1,1], padding='SAME') #Third convolution: 64x128-2x2 filters, 1x1 stride
conv3 = tf.add(conv3,b3) #Add bias
conv3 = tf.nn.relu(conv3) #ReLU acivation for convolution
mean_x, std_x = tf.nn.moments(conv3, axes = [0, 1, 2], keepdims=True)
conv3 = tf.nn.batch_normalization(conv3, mean_x, std_x, None, None, 1e-12) #Batch normalization after convolution

conv3 = tf.nn.max_pool(conv3, ksize=[1,3,3,1], strides=[1,2,2,1],padding='SAME') #Second max-pooling: 3x3 pool size, 2x2 stride

flatten = tf.reshape(conv3, [-1, w4.get_shape().as_list()[0]]) #Flattening layer
dense1 = tf.add(tf.matmul(flatten, w4), b4) #First dense layer, ReLU
dense1 = tf.nn.relu(dense1)

dense2 = tf.add(tf.matmul(dense1, w5), b5) #Second dense layer, Sigmoid
dense2 = tf.nn.sigmoid(dense2)

output = tf.add(tf.matmul(dense2,w6),b6) #Output layer
```

Define Loss Function

```
#Using categorical cross-entropy
loss_function = tf.reduce_mean(tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=output, labels =y)))
```

Training Block

```
show_every= 1 #Displaye parameter
Learning_Rate = 0.005 #Learning rate for gradient descent
#Execute feedforward
Session_Optimizer = tf.compat.v1.train.AdamOptimizer(Learning_Rate).minimize(loss_function)
result = sess.run(tf.compat.v1.global_variables_initializer())
epochs = 2
for i in range(epochs):
  #Execute backprop and gradient descent in batches
  for j in range(240):
    sess.run(Session_Optimizer,feed_dict={x:train_images[j:j+749],y:train_labels[j:j+749]})
  if i % show_every == 0:
    cur_loss_train = sess.run(loss_function,feed_dict={x:train_images,y:train_labels})
    print("Training loss at Epoch", i, "is: ", cur_loss_train)
```

```
 Training loss at Epoch 0 is:  0.19067742
 Training loss at Epoch 1 is:  0.18050241
```

Testing Block

```
predicted = output.eval(feed_dict={x:test_images,y:test_labels}) #Make prediction on MNIST test data with the model
predicted=np.argmax(predicted, axis=1)
test_labels=np.argmax(test_labels, axis=1)
print(predicted)
print(test_labels)
print(
    f"{classification_report(test_labels, predicted)}\n" #Evaluate test data with classification report
)
test_labels = keras.utils.to_categorical(test_labels, 10) #Re-format test labels for future evaluation
```

```
[7 2 1 ... 4 5 6]
[7 2 1 ... 4 5 6]
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       980
```

```
                1       0.98      0.99      0.99      1135
                2       0.95      0.98      0.97      1032
                3       0.95      0.96      0.96      1010
                4       0.97      0.97      0.97       982
                5       0.95      0.94      0.95       892
                6       0.95      0.97      0.96       958
                7       0.95      0.96      0.95      1028
                8       0.98      0.90      0.94       974
                9       0.95      0.94      0.94      1009

         accuracy                           0.96     10000
        macro avg       0.96      0.96      0.96     10000
     weighted avg       0.96      0.96      0.96     10000
```

Plotting Kernels

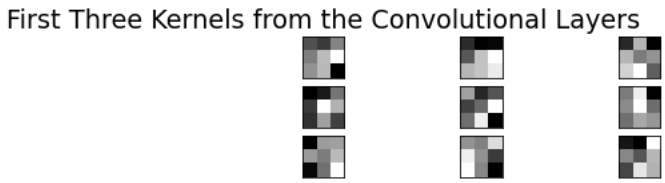
```
#Evaluate the filters
w1v = w1.eval()
w2v = w2.eval()
w3v = w3.eval()

#Representetive filters from layer 1
p1 = plt.subplot(6,3,1)
p1.set_xticks([])
p1.set_yticks([])
plt.title("First Three Kernels from the Convolutional Layers")
plt.imshow(w1v[:,:,:,1], cmap='gray')
p1 = plt.subplot(6,3,2)
p1.set_xticks([])
p1.set_yticks([])
plt.imshow(w1v[:,:,:,2], cmap='gray')
p1 = plt.subplot(6,3,3)
p1.set_xticks([])
p1.set_yticks([])
plt.imshow(w1v[:,:,:,3], cmap='gray')

#Representative filters from layer 2
p1 = plt.subplot(6,3,4)
p1.set_xticks([])
p1.set_yticks([])
plt.imshow(w2v[:,:,1,1], cmap='gray')
p1 = plt.subplot(6,3,5)
p1.set_xticks([])
p1.set_yticks([])
plt.imshow(w2v[:,:,2,1], cmap='gray')
p1 = plt.subplot(6,3,6)
p1.set_xticks([])
p1.set_yticks([])
plt.imshow(w2v[:,:,3,1], cmap='gray')

#Representative filters from layer 3
p1 = plt.subplot(6,3,7)
p1.set_xticks([])
p1.set_yticks([])
plt.imshow(w3v[:,:,1,1], cmap='gray')
p1 = plt.subplot(6,3,8)
p1.set_xticks([])
p1.set_yticks([])
plt.imshow(w3v[:,:,2,1], cmap='gray')
p1 = plt.subplot(6,3,9)
p1.set_xticks([])
p1.set_yticks([])
plt.imshow(w3v[:,:,3,1], cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f78b0b23fa0>
```



Plotting Feature Maps

```
#Evaluate feature maps
conv1v = conv1.eval(feed_dict={x:test_images,y:test_labels})
conv2v = conv2.eval(feed_dict={x:test_images,y:test_labels})
conv3v = conv3.eval(feed_dict={x:test_images,y:test_labels})

#Representetive feature maps from layer 1
p2 = plt.subplot(6,3,1)
p2.set_xticks([])
p2.set_yticks([])
plt.title("First Three Feature Maps from the Convolutional Layers")
plt.imshow(conv1v[0,:,:,1], cmap='gray')
p2 = plt.subplot(6,3,2)
p2.set_xticks([])
p2.set_yticks([])
plt.imshow(conv1v[0,:,:,2], cmap='gray')
p2 = plt.subplot(6,3,3)
p2.set_xticks([])
p2.set_yticks([])
plt.imshow(conv1v[0,:,:,3], cmap='gray')

#Representative feature maps from layer 2
p2 = plt.subplot(6,3,4)
p2.set_xticks([])
p2.set_yticks([])
plt.imshow(conv2v[0,:,:,1], cmap='gray')
p2 = plt.subplot(6,3,5)
p2.set_xticks([])
p2.set_yticks([])
plt.imshow(conv2v[0,:,:,2], cmap='gray')
p2 = plt.subplot(6,3,6)
p2.set_xticks([])
p2.set_yticks([])
plt.imshow(conv2v[0,:,:,3], cmap='gray')

#Representative feature maps from layer 3
p2 = plt.subplot(6,3,7)
p2.set_xticks([])
p2.set_yticks([])
plt.imshow(conv3v[0,:,:,1], cmap='gray')
p2 = plt.subplot(6,3,8)
p2.set_xticks([])
p2.set_yticks([])
plt.imshow(conv3v[0,:,:,2], cmap='gray')
p2 = plt.subplot(6,3,9)
p2.set_xticks([])
p2.set_yticks([])
plt.imshow(conv3v[0,:,:,3], cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f78b08604f0>
```

Analysis and Commentary

1. We observe that the tensorflow implementation matches the accuracy of the keras model.
2. In the auxilary editions of this model, the following changes are made to the convolutional layers.
    - The filter count is cut in half: We observe that reducing the filter count makes the feature maps less distinguishable. This is expected, as the filter count is proportional to model expressivity. However, note that the expressivity of the feature map is not adversely affected in a readilly visible manner before the starting filter count is reduced to 4. Thus, while accuracy is reduced, starting filter counts less than 32 can be used succesfully.
    - The filter size is increased by 1x1: The features maps increasingly resemble the features of the input data as their dimension is increased. This, once again, is expected since using smaller feature maps compresses the data more and makes it more difficult to identify similarities by inspection. Increasing the size also causes a disproportionate rise in training time.

3. The above script highlights the flexibility of tensorflow 1.x compared to the keras filters attribute. We note that instead of using a function call, tensorflow allows us to directly access the content of the kernels and the feature maps using the eval() function. This gives us the flexibility to perform algebraic manipulation of the kernels. Moreover, the same kernel can be re-used throughout a network. As long as each instance of the kernel is defined as a separate tensor, tensorflow allows for the extraction of each discrete instance of a kernal using the same method.