Problem Set 2: TensorFlow [70 pts]

Due date: Mar 13, 11:59pm

Submission instructions will be posted separately on Piazza

Note: The following has been verified to work with TensorFlow 2.0

* Adapted from official TensorFlow™ tour guide.

TensorFlow is a powerful library for doing large-scale numerical computation. One of the tasks at which it excels is implementing and training deep neural networks. In this assignment you will learn the basic building blocks of a TensorFlow model while constructing a deep convolutional MNIST classifier.

What you are expected to implement in this tutorial:

- Create a softmax regression function that is a model for recognizing MNIST digits, based on looking at every pixel in the image
- Use Tensorflow to train the model to recognize digits by having it "look" at thousands of examples
- Check the model's accuracy with MNIST test data
- Build, train, and test a multilayer convolutional neural network to improve the results

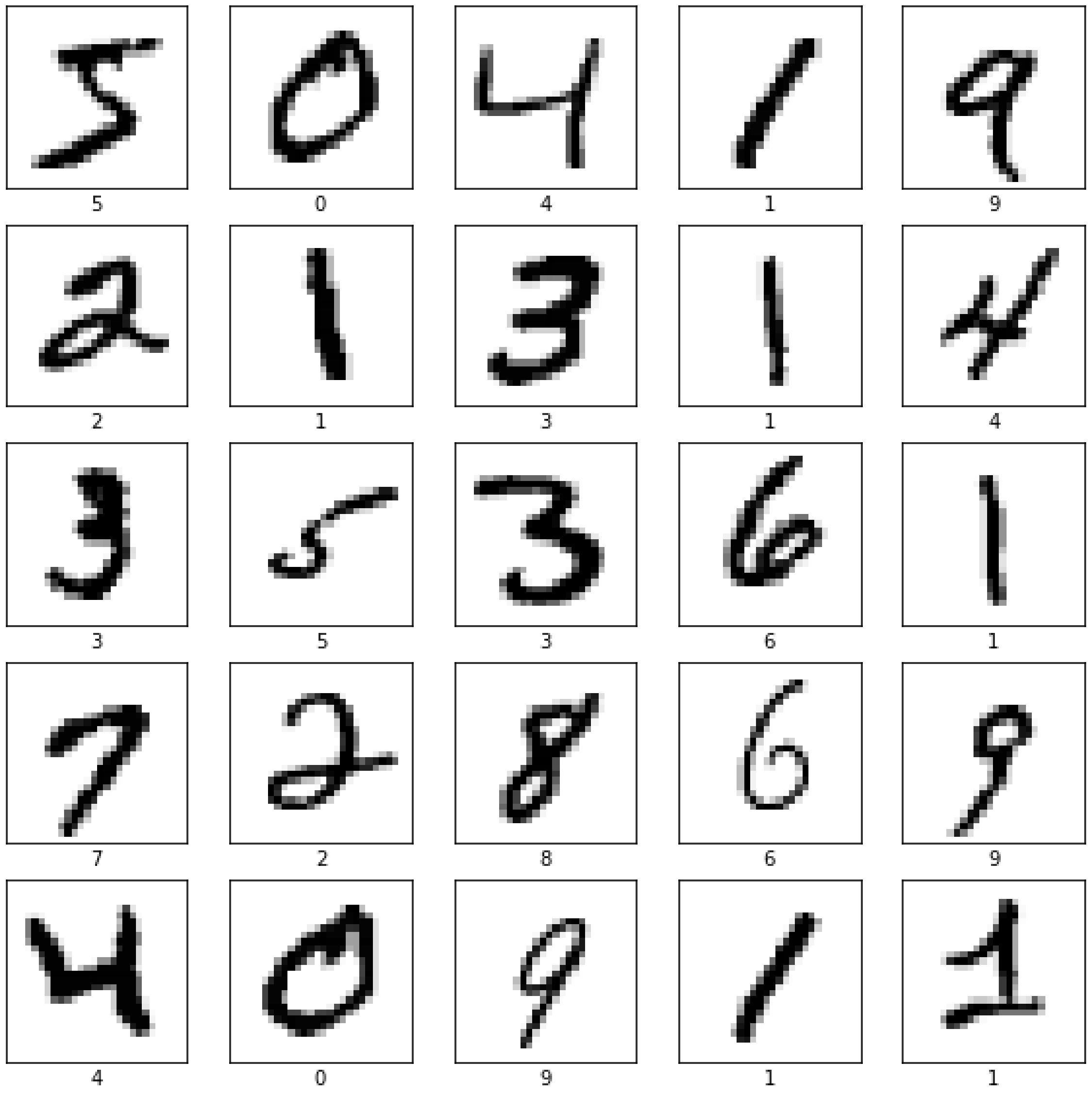
Data

After importing tensorflow, we can download the MNIST dataset with the built-in TensorFlow/Keras method.

In [101…
```
import os

import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

os.environ['OMP_NUM_THREADS'] = '1'
tf.__version__
```

Out[101…
```
'2.4.1'
```

In [116…
```
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load

class_names = ['0', '1', '2', '3', '4',
               '5', '6', '7', '8', '9']

plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(train_images[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[train_labels[i]])
plt.show()
```



[50 pts] 1. Build and Evaluate the CNN

In this part we will build a customized TF2 Keras model. As input, a CNN takes tensors of shape (image_height, image_width, color_channels), ignoring the batch size. For MNIST, you will configure our CNN to process inputs of shape (28, 28, 1), which is the format of MNIST images. You can do this by passing the argument input_shape to our first layer.

The overall architecture should be:

```
Model: "customized_cnn"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_2 (Conv2D)            multiple                  320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 multiple                  0         
_________________________________________________________________
conv2d_3 (Conv2D)            multiple                  18496     
_________________________________________________________________
flatten_1 (Flatten)          multiple                  0         
_________________________________________________________________
dense_2 (Dense)              multiple                  7930880   
_________________________________________________________________
dense_3 (Dense)              multiple                  10250     
=================================================================
Total params: 7,959,946
Trainable params: 7,959,946
Non-trainable params: 0
_________________________________________________________________
```

First Convolutional Layer [5 pts]

We can now implement our first layer. The convolution will compute 32 features for each 3x3 patch. The first two dimensions are the patch size, the next is the number of input channels, and the last is the number of output channels.

Max Pooling Layer [5 pts]

We stack max pooling layer after the first convolutional layer. These pooling layers will perform max pooling for each 2x2 patch.

Second Convolutional Layer [5 pts]

In order to build a deep network, we stack several layers of this type. The second layer will have 64 features for each 3x3 patch.

Fully Connected Layers [10 pts]

Now that the image size has been reduced to 11x11, we add a fully-connected layer with 128 neurons to allow processing on the entire image. We reshape the tensor from the second convolutional layer into a batch of vectors before the fully connected layer.

The output layer should also be implemented via a fully connect layer.

Complete the Computation Graph [15 pts]

Please complete the following function:

```
 def call(self, inputs, training=None, mask=None):
```

To apply the layer, we first reshape the input to a 4d tensor, with the second and third dimensions corresponding to image width and height, and the final dimension corresponding to the number of color channels (which is 1).

We then convolve the reshaped input with the first convolutional layer and then the max pooling followed by the second convolutional layer. These convolutional layers and the pooling layer will reduce the image size to 11x11.

Dropout Layer [5 pts]

Please add dropouts during training before each fully connected layers, as this helps avoid overfitting during training. https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

In [120…

```
from tensorflow.keras.layers import *

class CustomizedCNN(tf.keras.models.Model):

    def __init__(self, *args, **kwargs):
        super(CustomizedCNN, self).__init__()
        self.conv1 = Conv2D(32,(3,3), activation = 'relu', input_shape = (28,28,1))
        self.pool = MaxPooling2D(pool_size = (2,2))
        self.conv2 = Conv2D(64,(3,3), activation = 'relu')
        self.flat = Flatten()
        self.drop1 = Dropout(0.5)
        self.dense1 = Dense(128)
        self.drop2 = Dropout(0.5)
        self.dense2 = Dense(10)


    def call(self, inputs, training=None, mask=None):
        x = inputs[..., tf.newaxis]
        #x = tf.reshape(inputs,(32,28,28,1))
        x = self.conv1(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.flat(x)
        x = self.drop1(x)
        x = self.dense1(x)
        x = self.drop2(x)
        x = self.dense2(x)
        return x
```

Build the Model

In [121…

```
model = CustomizedCNN()
model.build(input_shape=(None, 28, 28))
model.summary()
```

```
Model: "customized_cnn_26"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_52 (Conv2D)           multiple                  320
_________________________________________________________________
max_pooling2d_26 (MaxPooling multiple                  0
_________________________________________________________________
conv2d_53 (Conv2D)           multiple                  18496
_________________________________________________________________
flatten_26 (Flatten)         multiple                  0
```

```
_________________________________________________________________
dropout_52 (Dropout)         multiple                  0         
_________________________________________________________________
dense_52 (Dense)             multiple                  991360    
_________________________________________________________________
dropout_53 (Dropout)         multiple                  0         
_________________________________________________________________
dense_53 (Dense)             multiple                  1290      
=================================================================
Total params: 1,011,466
Trainable params: 1,011,466
Non-trainable params: 0
_________________________________________________________________
```

We can specify a loss function just as easily. Loss indicates how bad the model's prediction was on a single example; we try to minimize that while training across all the examples. Here, our loss function is the cross-entropy between the target and the softmax activation function applied to the model's prediction. As in the beginners tutorial, we use the stable formulation:

In [122…

```
model.compile(optimizer='adam',
                loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                metrics=['accuracy'])
```

Train and Evaluate the Model [5 pts]

We will use a more sophisticated ADAM optimizer instead of a Gradient Descent Optimizer.

Feel free to run this code. Be aware that it does 10 training epochs and may take a while (possibly up to half an hour), depending on your processor.

The final test set accuracy after running this code should be approximately 98.7% -- not state of the art, but respectable.

We have learned how to quickly and easily build, train, and evaluate a fairly sophisticated deep learning model using TensorFlow.

In [123…

```
train_images = train_images.astype("float32")
history = model.fit(
    x = train_images,
    y = train_labels,
    batch_size = 25,
    epochs = 10,
    validation_split = 0.2) # Use correct args here.
```

```
Epoch 1/10
1920/1920 [==============================] - 33s 17ms/step - loss: 2.6676 - accuracy: 0.
8152 - val_loss: 0.1088 - val_accuracy: 0.9652
Epoch 2/10
1920/1920 [==============================] - 32s 17ms/step - loss: 0.2217 - accuracy: 0.
9368 - val_loss: 0.0800 - val_accuracy: 0.9750
Epoch 3/10
1920/1920 [==============================] - 32s 17ms/step - loss: 0.1556 - accuracy: 0.
9549 - val_loss: 0.0724 - val_accuracy: 0.9780
Epoch 4/10
1920/1920 [==============================] - 32s 17ms/step - loss: 0.1298 - accuracy: 0.
9620 - val_loss: 0.0579 - val_accuracy: 0.9832
Epoch 5/10
1920/1920 [==============================] - 32s 17ms/step - loss: 0.1156 - accuracy: 0.
```

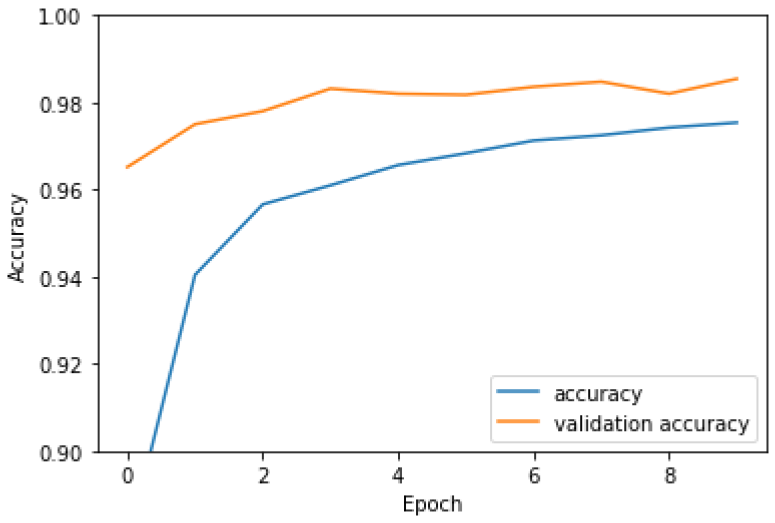
```
9662 - val_loss: 0.0618 - val_accuracy: 0.9820
Epoch 6/10
1920/1920 [==============================] - 32s 17ms/step - loss: 0.1034 - accuracy: 0.
9697 - val_loss: 0.0575 - val_accuracy: 0.9818
Epoch 7/10
1920/1920 [==============================] - 32s 17ms/step - loss: 0.1016 - accuracy: 0.
9707 - val_loss: 0.0666 - val_accuracy: 0.9836
Epoch 8/10
1920/1920 [==============================] - 32s 17ms/step - loss: 0.0908 - accuracy: 0.
9731 - val_loss: 0.0596 - val_accuracy: 0.9847
Epoch 9/10
1920/1920 [==============================] - 32s 17ms/step - loss: 0.0848 - accuracy: 0.
9755 - val_loss: 0.0636 - val_accuracy: 0.9820
Epoch 10/10
1920/1920 [==============================] - 35s 18ms/step - loss: 0.0785 - accuracy: 0.
9766 - val_loss: 0.0599 - val_accuracy: 0.9854
```

In [124…

```python
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label = 'validation accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.ylim([0.9, 1])
plt.legend(loc='lower right')
```

Out[124…

```
<matplotlib.legend.Legend at 0x1466810dbb0>
```



In [125…

```python
test_images = test_images.astype("float32")
test_loss, test_acc = model.evaluate(test_images,  test_labels)
print(test_acc)
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.0558 - accuracy: 0.9836
0.9836000204086304
```

[20pts] 2. Simple Regularization Methods

You may have noticed the `reg` parameter in `TwoLayerMLP.loss` , controlling "regularization strength". In learning neural networks, aside from minimizing a loss function $\mathcal{L}(\theta)$ with respect to the network parameters θ, we usually explicitly or implicitly add some regularization term to reduce overfitting. A simple and popular regularization strategy is to penalize some *norm* of θ.

[10pts] Q2.1: L2 regularization

We can penalize the L2 norm of θ: we modify our objective function to be $\mathcal{L}(\theta) + \lambda\|\theta\|^2$ where λ is the weight of regularization. We will minimize this objective using gradient descent with step size η. Derive the update rule: at time $t+1$, express the new parameters θ_{t+1} in terms of the old parameters θ_t, the gradient $g_t = \frac{\partial \mathcal{L}}{\partial \theta_t}$, η, and λ.

$$\begin{aligned}\theta_{t+1} &= \theta_t - \eta * (g_t + 2\lambda\theta_t) \\ &= \theta_t - \eta * \frac{\partial \mathcal{L}}{\partial \theta_t} - 2\eta\lambda\theta_t\end{aligned}$$

[10pts] Q2.2: L1 regularization

Now let's consider L1 regularization: our objective in this case is $\mathcal{L}(\theta) + \lambda\|\theta\|_1$. Derive the update rule.

(Technically this becomes *Sub-Gradient* Descent since the L1 norm is not differentiable at 0. But practically it is usually not an issue.)

$$\begin{aligned}\theta_{t+1} &= \theta_t - \eta * (g_t + \lambda * sgn(\theta_t)) \\ &= \theta_t - \eta * \frac{\partial \mathcal{L}}{\partial \theta_t} - \eta\lambda * sgn(\theta_t))\end{aligned}$$ **where sgn is the sign function**

$sgn(\theta)$ **= 1 when** θ **> 0, and** $sgn(\theta)$ **= - 1 when** θ **< 0, sign(0) = 0**