Problem Set 4: TensorFlow

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

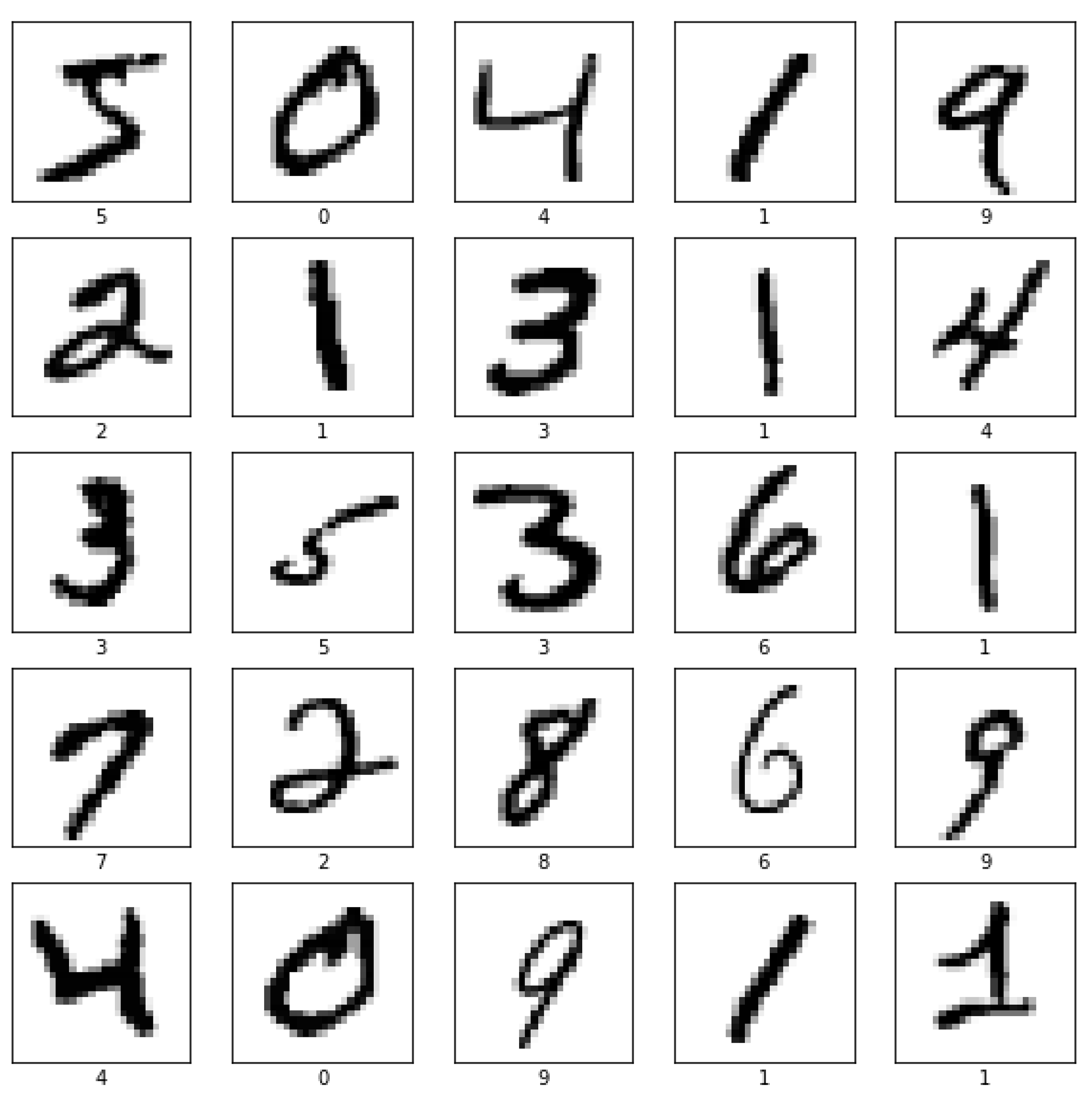
Data

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

In [124]:
```
import os

import tensorflow as tf
import matplotlib.pyplot as plt

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

Out[124]: '2.3.0'

In [157]:
```
(train_images, train_labels), (test_images, test_labels) = tf.keras.datasets.mnist.load_data()
train_images_f16 = train_images.astype("float16")
test_images_f16 = test_images.astype("float16")
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()
```



Build 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 [10 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

These docs could be very helpful:

https://www.tensorflow.org/api_docs/python/tf/keras/layers

https://www.tensorflow.org/api_docs/python/tf/nn/dropout

In [168]:
```
class CustomizedCNN(tf.keras.models.Model):

    def __init__(self, *args, **kwargs):
      super(CustomizedCNN, self).__init__()
      self.conv2d_2 = tf.keras.layers.Conv2D(32, (3,3), input_shape=(28,28,1),activation="relu")
      self.max_pooling2d_1 = tf.keras.layers.MaxPool2D(pool_size=(2, 2))
      self.conv2d_3 = tf.keras.layers.Conv2D(64, (3,3),activation="relu")
      self.flatten_1 = tf.keras.layers.Flatten()
      self.dropout = tf.keras.layers.Dropout(.1)
      self.dense_2 = tf.keras.layers.Dense(128, activation="relu")
      self.dense_3 = tf.keras.layers.Dense(10)

    def call(self, inputs, training=True, mask=None):
      x = tf.keras.layers.Reshape((28,28,1))(inputs)
      a = self.conv2d_2(x)
      b = self.max_pooling2d_1(a)
      e = self.conv2d_3(b)
      c = self.flatten_1(e)
      if training:
        c = self.dropout(c)
      d = self.dense_2(c)
      if training:
        d = self.dropout(d)
      return self.dense_3(d)
```

Build the Model

In [169]:
```
model = CustomizedCNN()
model.build(input_shape=(None, 28, 28))
model.summary()
```

```
Model: "customized_cnn_46"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_93 (Conv2D)           multiple                  320
_________________________________________________________________
max_pooling2d_46 (MaxPooling multiple                  0
_________________________________________________________________
conv2d_94 (Conv2D)           multiple                  18496
_________________________________________________________________
flatten_45 (Flatten)         multiple                  0
_________________________________________________________________
dropout_36 (Dropout)         multiple                  0
_________________________________________________________________
dense_90 (Dense)             multiple                  991360
_________________________________________________________________
dense_91 (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 [170]:
```
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. Do 10 training epochs with 128 images per batch, which may take a while (possibly up to half an hour), depending on your processor. Don't forget to set validation set.

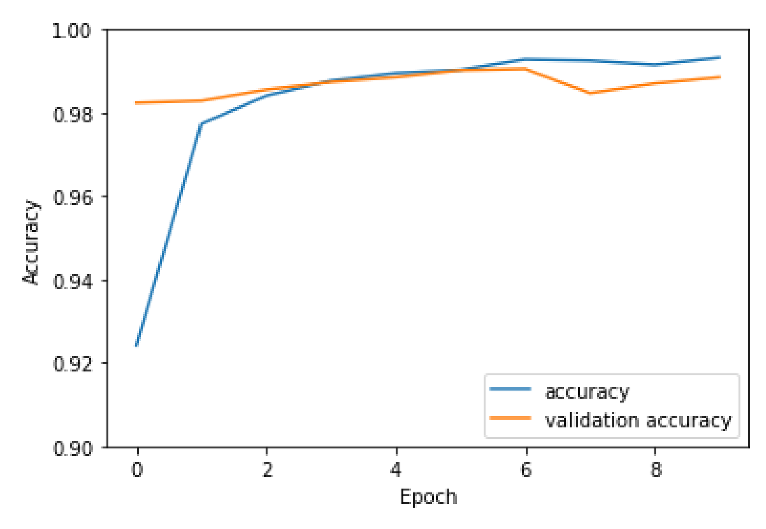
The final test set accuracy after running this code should be approximately 98.7% -- not state of the art, but respectable. We have learned so far how to quickly and easily build, train, and evaluate a fairly sophisticated deep learning model using TensorFlow.

In [171]:
```
history = model.fit(train_images_f16, train_labels, batch_size=128, epochs=10, validation_split=.1) # Use appropiate args here.
```

```
Epoch 1/10
422/422 [==============================] - 51s 121ms/step - loss: 0.9397 - accuracy: 0.9241 - val_loss: 0.0662 - val_accuracy: 0.9823
Epoch 2/10
422/422 [==============================] - 52s 122ms/step - loss: 0.0732 - accuracy: 0.9773 - val_loss: 0.0601 - val_accuracy: 0.9828
Epoch 3/10
422/422 [==============================] - 51s 122ms/step - loss: 0.0513 - accuracy: 0.9840 - val_loss: 0.0542 - val_accuracy: 0.9855
Epoch 4/10
422/422 [==============================] - 51s 122ms/step - loss: 0.0394 - accuracy: 0.9877 - val_loss: 0.0514 - val_accuracy: 0.9873
Epoch 5/10
422/422 [==============================] - 52s 123ms/step - loss: 0.0316 - accuracy: 0.9895 - val_loss: 0.0515 - val_accuracy: 0.9885
Epoch 6/10
422/422 [==============================] - 52s 123ms/step - loss: 0.0300 - accuracy: 0.9902 - val_loss: 0.0511 - val_accuracy: 0.9902
Epoch 7/10
422/422 [==============================] - 51s 122ms/step - loss: 0.0223 - accuracy: 0.9928 - val_loss: 0.0557 - val_accuracy: 0.9905
Epoch 8/10
422/422 [==============================] - 52s 124ms/step - loss: 0.0239 - accuracy: 0.9925 - val_loss: 0.0673 - val_accuracy: 0.9847
Epoch 9/10
422/422 [==============================] - 51s 122ms/step - loss: 0.0266 - accuracy: 0.9915 - val_loss: 0.0711 - val_accuracy: 0.9870
Epoch 10/10
422/422 [==============================] - 52s 122ms/step - loss: 0.0208 - accuracy: 0.9932 - val_loss: 0.0613 - val_accuracy: 0.9885
```

In [172]:
```
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[172]: <matplotlib.legend.Legend at 0x7f6cdef5dc18>



In [173]:
```
test_loss, test_acc = model.evaluate(test_images_f16,  test_labels)
print(test_acc)
```

```
313/313 [==============================] - 3s 10ms/step - loss: 0.0621 - accuracy: 0.9845
0.984499990940094
```