Nathan Morris ICP_4

All of my different attempts to get to 99%:

8 hidden elu layers, 50 epochs, batch size = 256

98.39000105857849

8 hidden relu layers, 50 epochs, batch size = 256

98.47000241279602

8 alternating elu and relu layers, 50 epochs, batch size = 256

98.54000210762024

10 alternating elu and relu layers, 50 epochs, batch size = 256

98.36999773979187

6 alternating elu and relu layers, 50 epochs, batch size = 256

97.06000089645386

6 alternating elu and relu layers, 50 epochs, batch size = 256 pt. 2

96.97999954223633

8 alternating elu and relu layers, 50 epochs, batch size = 256

98.07000160217285

8 layers of tanh, 50 epochs, batch size = 512

97.89000153541565

2 relu, 8 tanh, 50 epochs, batch size = 512

88.84999752044678%

2 tanh, 6 alternating elu, relu, 50 epochs, batch size = 512

97.43000268936157%

2 tanh, 2 sigmoid, 2 elu, 2 relu

96.97999954223633%

4 sigmoid, 2 elu, 2 relu

96.32999897003174%

1 sigmoid, 7 alternating relu and elu

97.28999733924866%

8 relu Adam

97.50999808311462%

8 relu SGD 50 epochs

92.97000169754028%

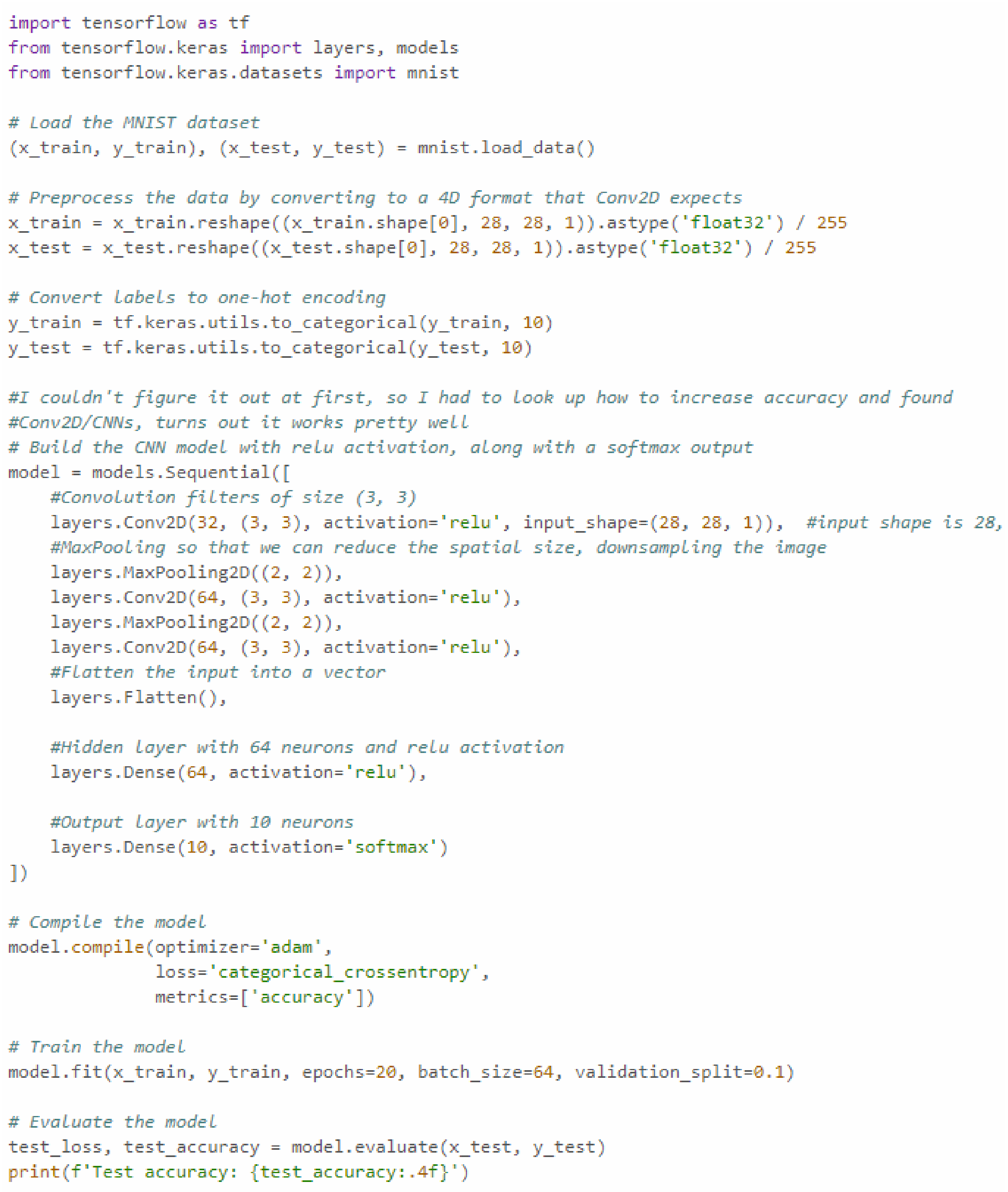
8 relu  RMSProp

98.1000006198883%

10 relu RMSProp

98.50999712944031%

After a number of messing with other settings I looked it up on youtube and found out about Conv2S and MaxPooling2D. Once I found these, it made the rest of my work significantly easier. So I ended up moving forward with it until I got my final Product of:



```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess the data by converting to a 4D format that Conv2D expects
x_train = x_train.reshape((x_train.shape[0], 28, 28, 1)).astype('float32') / 255
x_test = x_test.reshape((x_test.shape[0], 28, 28, 1)).astype('float32') / 255

# Convert labels to one-hot encoding
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)

#I couldn't figure it out at first, so I had to look up how to increase accuracy and found
#Conv2D/CNNs, turns out it works pretty well
# Build the CNN model with relu activation, along with a softmax output
model = models.Sequential([
    #Convolution filters of size (3, 3)
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),  #input shape is 28,
    #MaxPooling so that we can reduce the spatial size, downsampling the image
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    #Flatten the input into a vector
    layers.Flatten(),

    #Hidden layer with 64 neurons and relu activation
    layers.Dense(64, activation='relu'),

    #Output layer with 10 neurons
    layers.Dense(10, activation='softmax')
])

# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(x_train, y_train, epochs=20, batch_size=64, validation_split=0.1)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy:.4f}')
```

And the output of running this after tweaking and figuring out that one hidden layer works best with this model, Here is the output

```
Epoch 1/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.8580 - loss: 0.4584 - val_accuracy: 0.9823
- val_loss: 0.0625
Epoch 2/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9818 - loss: 0.0567 - val_accuracy: 0.9843
- val_loss: 0.0518
Epoch 3/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9887 - loss: 0.0366 - val_accuracy: 0.9905
- val_loss: 0.0318
Epoch 4/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9907 - loss: 0.0282 - val_accuracy: 0.9898
- val_loss: 0.0369
Epoch 5/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9928 - loss: 0.0221 - val_accuracy: 0.9912
- val_loss: 0.0338
Epoch 6/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9942 - loss: 0.0182 - val_accuracy: 0.9888
- val_loss: 0.0398
Epoch 7/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9954 - loss: 0.0145 - val_accuracy: 0.9918
- val_loss: 0.0331
Epoch 8/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9959 - loss: 0.0125 - val_accuracy: 0.9900
- val_loss: 0.0396
Epoch 9/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9965 - loss: 0.0107 - val_accuracy: 0.9893
- val_loss: 0.0473
Epoch 10/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9976 - loss: 0.0076 - val_accuracy: 0.9915
- val_loss: 0.0414
Epoch 11/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9975 - loss: 0.0072 - val_accuracy: 0.9912
- val_loss: 0.0388
Epoch 12/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9981 - loss: 0.0056 - val_accuracy: 0.9918
- val_loss: 0.0361
Epoch 13/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9976 - loss: 0.0069 - val_accuracy: 0.9915
- val_loss: 0.0444
Epoch 14/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9990 - loss: 0.0036 - val_accuracy: 0.9907
- val_loss: 0.0424
Epoch 15/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9975 - loss: 0.0064 - val_accuracy: 0.9922
- val_loss: 0.0430
Epoch 16/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9984 - loss: 0.0053 - val_accuracy: 0.9927
- val_loss: 0.0485
Epoch 17/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9995 - loss: 0.0017 - val_accuracy: 0.9892
- val_loss: 0.0633
Epoch 18/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.9973 - loss: 0.0078 - val_accuracy: 0.9918
- val_loss: 0.0443
Epoch 19/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - accuracy: 0.9991 - loss: 0.0032 - val_accuracy: 0.9928
- val_loss: 0.0475
Epoch 20/20
844/844 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9985 - loss: 0.0035 - val_accuracy: 0.9915
- val_loss: 0.0450
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9882 - loss: 0.0522
Test accuracy: 0.9913
```