CS 510 - Start YR Neural Networks
Matthew Ernst & James Yost

TensorFlow 2 Set-Up:

For this assignment, we transition into modern neural networks-based machine learning. The ultimate goal for this assignment was to familiarize ourselves with TensorFlow 2. Specifically, how to run specific neural networks with it and how to manipulate and build our own. For this assignment, we were tasked with setting up and running the “TensorFlow 2 quickstart for beginners” Jupyter notebook. This notebook demonstrates the basics of TensorFlow as well as serves as a way to make sure that TensorFlow is set up correctly for our environments. This seemed to be the first challenge that we ran into. Seeing as one of us is running in Linux and the other MacOS, we had to download the specific TensorFlow versions and make sure it worked for both systems. We tried multiple attempts at this with python virtual environments and had no luck. Luckily, we were able to put everything into PyCharm and create a requirements.txt to file which with the use of pip allowed PyCharm to figure which TensorFlow worked for a specific environment. From here we were able to run the quickstart notebook and verify our environments were working correctly.

Failed Images:

With our TensorFlow set up correctly, we were tasked with displaying the failed MNIST images from the trained model that was provided by the quickstart notebook. We did this with the use of OpenCV to display them in individual windows. This can be shown in Figure 1. Here, we display the image that failed along with a text overlay showing what the model labeled it as and its true label. We also enlarged the image by a factor of ten to better display the images in the windows. Even though this model is roughly 98% accurate, there were many failed cases. Going through it by hand, we noticed that a lot of the failed ones were even hard for our human vision to reconigze.

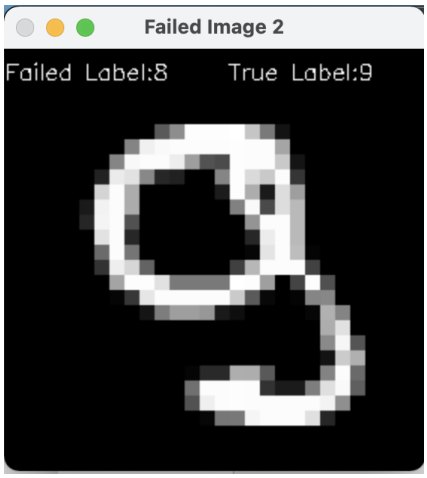

Figure 1

Training Epochs:

After displaying the images that had failed, we were tasked with starting to become more comfortable with TensorFlow. We did this first by training the original provided model but with various epochs. We did this by running the original model through a for loop that varied the number of epochs from 0 to 10. Shown in Figure 2 are the results in loss and accuracy of the model as epochs increased.

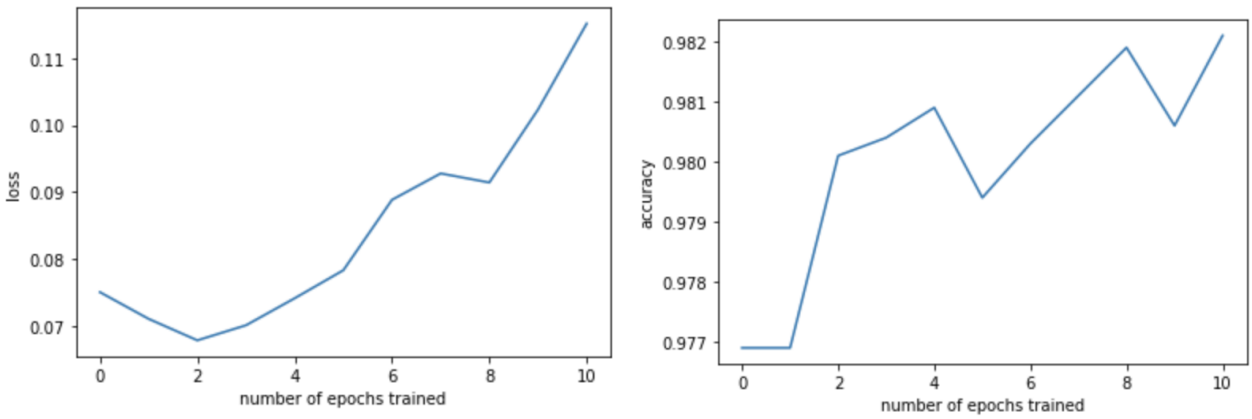

Figure 2

As we can see there is not a true linear relationship as the epochs vary. This is shown in the first graph in the very beginning with no epochs actually having more loss than 2. However, as we increase the epochs, the loss increases dramatically until around 6 in which starts to plateau until it increases again. A somewhat similar but more sporadic trend can be seen in the second graph. Here, we see that as the epochs increase, we generally see accuracy increase as well. However, there a momentary dip in accuracy depending on the epoch. Specifically, these dips can be seen at epoch 5 and 9. Here there is a dip from the previous amount of epochs that results in less accuracy. It is important to note though that after 5 epochs we are looking at an increase in accuracy of around 0.01%. This isn't such a vast jump in accuracy that there is a need to increase epochs. This allows the user to choose a lower epoch number and have to wait less time with training.

Width and Depth:

Our next manipulation of the TensorFlow was to build our own networks and run them with the provided MNIST dataset and compare loss and accuracy. We ended up building 4 models. The first one being the provided model but without Dropout, the second was a bow-tie or autoencoder-shaped network with the nodes of the hidden layer being 128, 64, 32, 16, 8, 4, 8, 32, 64, and 128. Our third model had three hidden layers with 128 as the number of nodes at each layer. Lastly, our final model was similar to the third but instead of the same activation

function, we had the first layer use relu, the second use sigmoid, and the last use tanh. The results of these models after training for 5 epochs can be shown in Figure 3.

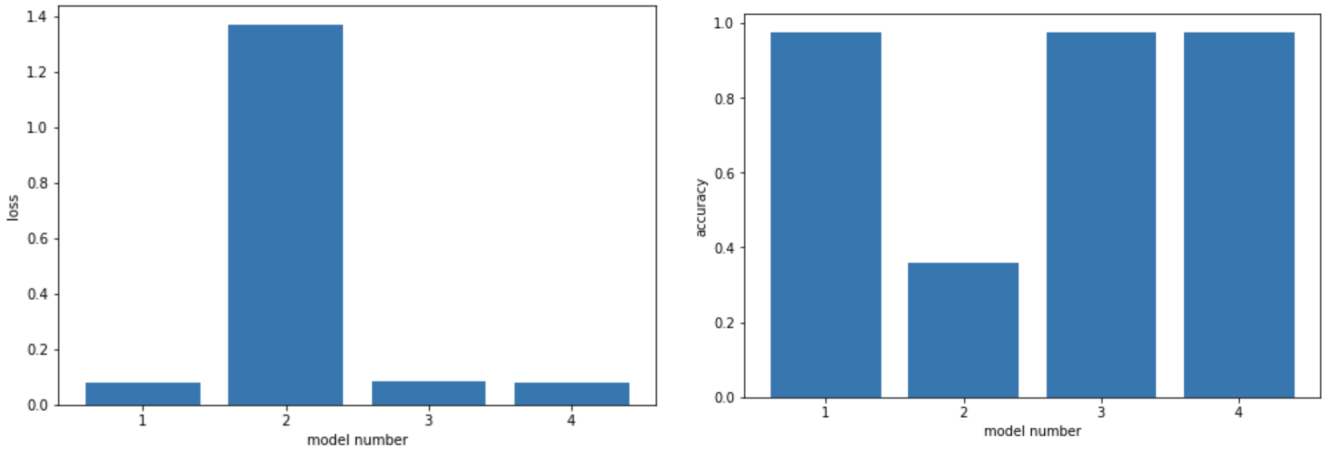

Figure 3

It is fairly apparent from these graphs which model did not do the best. Model 2 had a significant loss compared to the other models, almost a factor of 10. We can also see this in the accuracy graph with the accuracy of this model being extremely low, around 39%. However, the other models all performed roughly the same. The loss and accuracy of these models were all within the same range as themselves as well as the original provided model. This begs the question that if similar loss and accuracy can be taken from these models, wouldn't the simpler most intuitive model be the easiest to use? We believe this to be true and thus using the provided model from TensorFlow would be best.

Sobel Edge Detection:

Our last task was to use OpenCV's Sobel edge detection. We did this by testing different variations of the edge detection such as horizontal, vertical, and laplacian. Shown in Figure 4 are the results from the edge detection. We believe that the edges in the horizontal and vertical direction tended to do the best. We also saw that changing the gradient on the images also drowned them out. Due to this, we kept the gradient as the standard cv2.CV_64F.

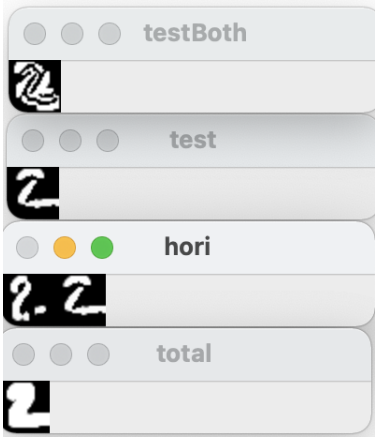

Figure 4