ECSE 4850 Programming Assignment 3

Training a convolutional neural network to classify images using the CIFAR-10 dataset

---

Note: For information regarding loading and evaluating the model, see the end of this report.

Problem Statement

The objective of this programming assignment is to train a convolutional neural network to classify images from the CIFAR-10 dataset. For this assignment, full usage of the Tensorflow library is allowed, and therefore no gradient computations or weight update computations need to be done by hand. The structure of the model is fixed in terms of the number of layers that can be used, but hyper parameters are not fixed; the learning rate, batch size, and number of epochs can all be changed as needed. Additionally, techniques to combat overfitting are allowed, such as batch normalization, dropout, weight regularization, and data augmentation.

Environment Settings

For development of this project, an Anaconda virtual environment was created with Python 3.8 and Tensorflow 2.3.0, and programming and debugging was completed using the Spyder IDE. To improve model training speed, Tensorflow was connected with my system's GPU via CUDA version 10.1 and CuDNN version 7.6. For reference, my system has an Intel Core i7-7700HQ 4-core CPU and an NVIDIA GeForce GTX 1050 GPU.

Model Description

The model for this assignment consists of 3 convolutional layers, two max pooling layers, and one fully connected layer to generate the outputs. An overview of the structure can be seen in Figure 1 below.

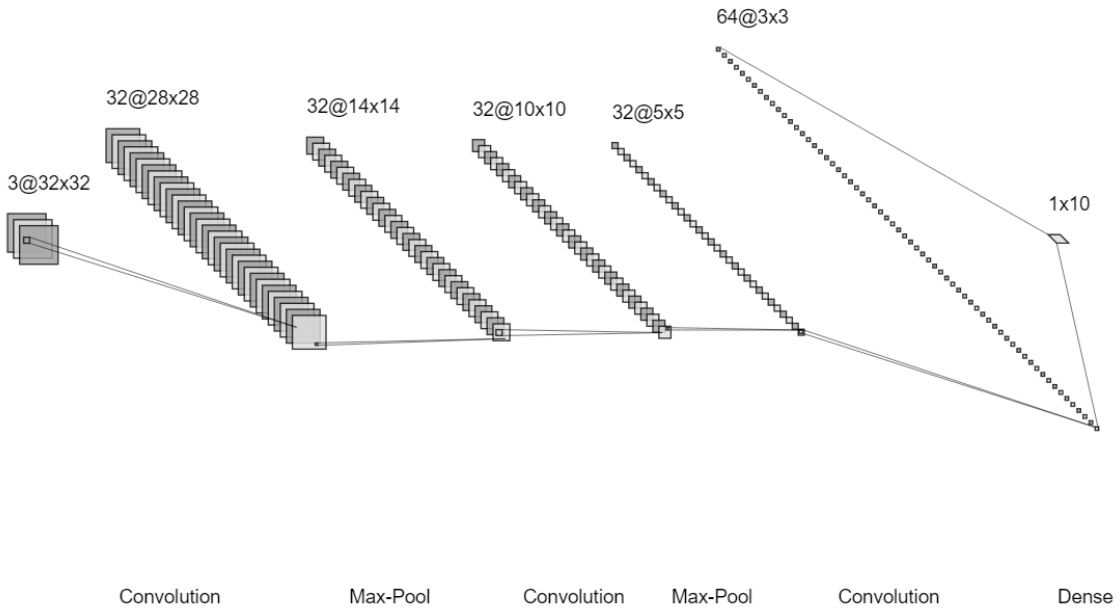

Figure 1: Model Structure

The specific details of each layer are as follows:

1. Convolutional Layer 1: This layer has 32 5×5 filters, no zero padding, and a stride of 1. This layer uses ReLU activation.
2. Max Pool Layer 1: This layer has a pooling window of 2×2 and a stride of 2.

3. Convolutional Layer 2: This layer has 32 5×5 filters, no zero padding, and a stride of 1. This layer uses ReLU activation.
4. Max Pool Layer 2: This layer has a pooling window of 2×2 and a stride of 2.
5. Convolutional Layer 3: This layer has 64 3×3 filters, no zero padding, and a stride of 1. This layer uses ReLU activation.
6. Fully Connected Layer: This layer has 10 nodes and is applied after flattening the output of the previous layer. Therefore, it has a 576×10 weight matrix and a 10×1 weight vector. This layer has softmax activation on its output.

Combatting Overfitting

When the model was built and was trained for the first time, an overfitting problem was noticed. This can be seen in Figure 2 below.

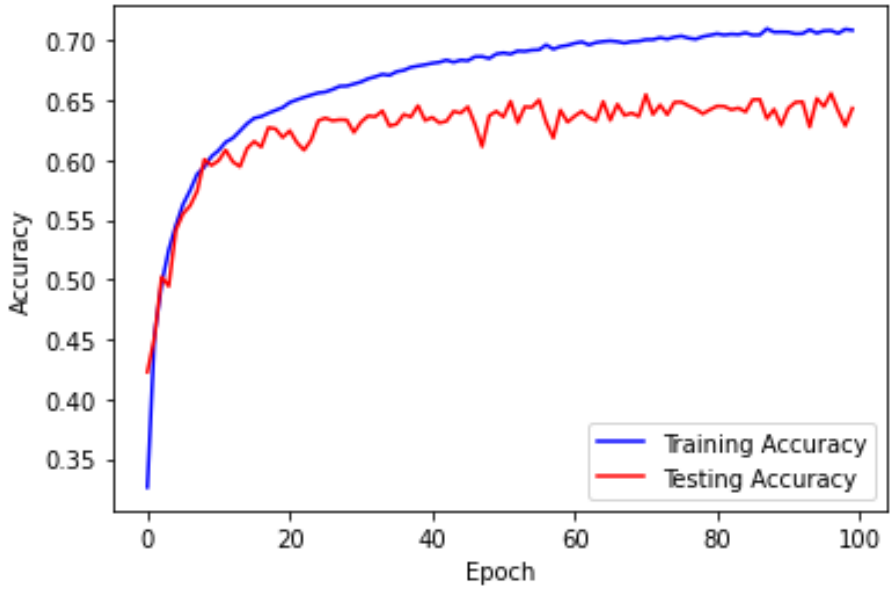

Figure 2: Model accuracy plotted as a function of epochs. Overfitting can be seen.

The plot shows a clear deviation between the training accuracy and the testing accuracy. The training accuracy continues to increase while the testing accuracy levels off at around 64%. Part of this can be alleviated by adjusting the learning hyper parameters (this plot was obtained with a learning rate of 0.001, and batch size of 25, and 100 epochs), but it can be reduced even further by implementing various techniques such as data augmentation, dropout, batch normalization, and weight regularization. A few of these were implemented to train this model and their details are described below.

- *Data Augmentation*: A Keras `ImageDataGenerator` object was used, which would apply random horizontal flips on the images in the training set. Originally I had set it to apply horizontal and vertical shifts to the images as well, but I found that this hindered the model's ability to learn features.
- *Dropout*: To apply dropout to the model, Keras `Dropout` layers were applied to the model after each max pooling layer and after the last convolutional layer. This layer would randomly set inputs to 0 at a rate set by the user. I set a base dropout rate that would be used for the first dropout layer, and then each subsequent dropout layer would have an increased rate compared to the previous one. This is described more in the **Hyper Parameters** section.
- *Batch Normalization*: A Keras `BatchNormalization` layer was added to the model after each convolutional layer. This layer would normalize its inputs by using the mean and standard deviation.

Hyper Parameters

- *Optimizer*: For training of the model, the Adam optimizer was used.
- *Learning Rate*: The initial learning rate η was set to 0.002. This learning rate is modified over time by the Adam optimizer. I found that increasing this learning rate too high, for example 0.01, would cause a lot of oscillation in the training and testing accuracy.
- *Weight and Bias Initialization*: All weights were initialized to a small random value from a normal distribution, i.e. $W \sim 0.01 \cdot N(0, 1)$. All biases were initialized to 0.
- *Batch Size*: The batch size was set to 100. I tried running the model with a batch size of 25, 50, 100, 200, and 500, and determined that 100 was the best option. A batch size of 25 and 50 were a bit too noisy for the model to learn features well, and a batch size of 200 and 500 would lead to a small amount of overfitting.
- *Number of Epochs*: I let the model train for 250 epochs. I set this limit to 250 since that seemed to be long enough to let the model learn features well and approach the learning plateau. However, I'm sure that if I let the model train for much longer, maybe 400 epochs, it might gain 1-2% more accuracy, but the tradeoff between time and performance improvement didn't seem worth it.
- *Dropout Rate*: I set the base dropout rate to 0.1. This means that about 10% of the inputs for the dropout layer would be set to 0. Additionally, each subsequent dropout layer had a higher dropout rate than the previous. Dropout layer 2 had a dropout rate of 0.2, and dropout layer 3 had a dropout rate of 0.3. This forces the model to adapt more as it gets closer to generating its outputs which has a more effective overfitting dampening effect.

Pseudocode

```
import training and testing data sets
normalize images and build one-hot label vectors
initialize model
create ImageDataGenerator

for epoch = 1 to 250
    for each batch in training data set
        compute predictions
        compute loss and accuracy
        compute gradient and update weights
    end loop

    for each batch in testing data set
        compute predictions
        compute loss and accuracy
    end loop

    compute metrics for plotting
end loop

plot metrics
visualize 1st convolutional layer filters
```

Model Performance

Training & Testing Loss

While the model was training, metrics of its training/testing loss and training/testing accuracy were collected. The average loss of the model (the average of the losses computed for each batch of training images) is shown in the plot below.

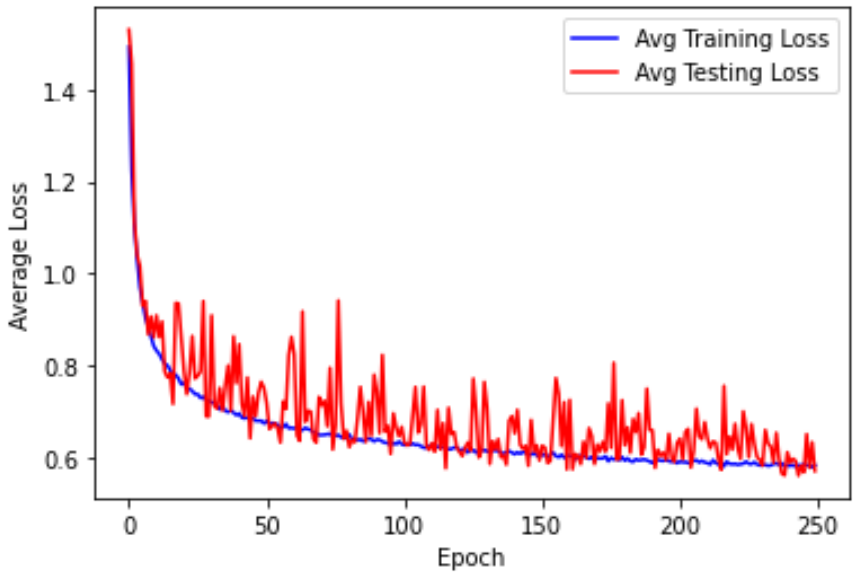

Figure 3: Training and testing loss as a function of epochs

The plot above shows a few important things. First, the training and testing loss curves follow a downward trajectory, first decreasing steeply and then levelling off as more epochs occurred. The testing loss has significantly more variation, but it still follows this trend. Second, the training and testing loss curves follow each other closely, indicating that the model is not overfitting. At the end of Epoch 250, the average training loss was 0.58 and the average testing loss was 0.57.

Training & Testing Accuracy

The accuracy of the model was evaluated during training as well. The training and testing accuracy of the model is plotted in the figure below. Note here that "accuracy" is defined as the ratio of correct classifications to the total number of images.

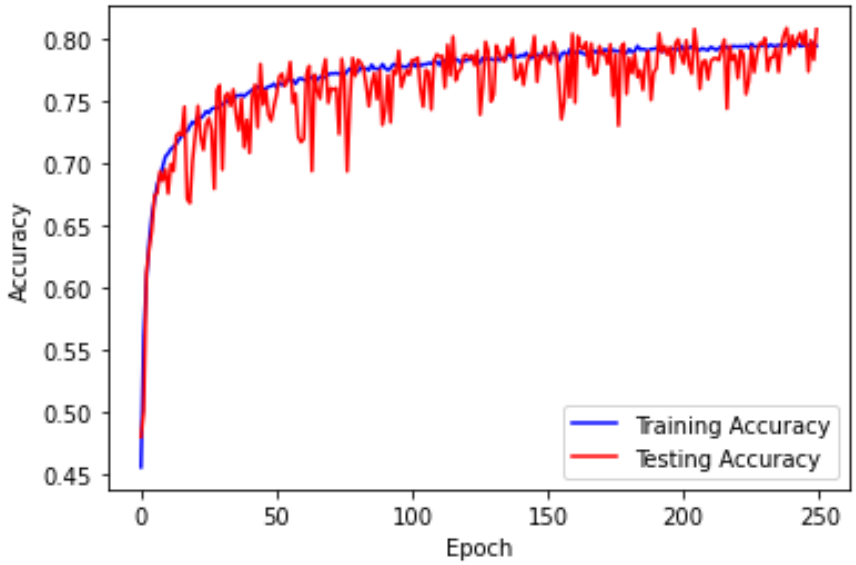

Figure 4: Training and testing accuracy as a function of epochs

The above plot shows that the accuracy of the model increased steeply at first, and then started to increase more slowly as more epochs occurred. Again, there is a large amount of variance in the the testing accuracy curve, but it still follows the general increasing trend. Additionally, this plot shows that the training accuracy and testing accuracy curves followed each other very closely, indicating that the model is not overfitting. At the end of Epoch 250, the model's training accuracy was 79% and the model's testing accuracy was 80.7%. To get a better idea of the model's performance for each class, we can look at the class-wise testing inaccuracy values. "Class-wise inaccuracy" is defined as the ratio of incorrect classifications for a specific class to the total number of images that appear for that specific class. These values are summarized in the table below.

Class	**Inaccuracy**
airplane	0.1810
automobile	0.0996
bird	0.2182
cat	0.3224
deer	0.2181
dog	0.3287
frog	0.1576
horse	0.2109
ship	0.0583
truck	0.1355
Average	**0.1930**

Table 1: Class-wise testing inaccuracy values

It is difficult to know exactly what the CNN model is thinking when classifying, but the above table can give us some insight. First, it appears that the model has similar classification inaccuracy for each class (that is, there are no classes that really stand out with a lot of error). Second, it looks like the model has the most difficulty with classifying cats and dogs, which can be reasonably expected. It looks like the model has the least difficulty classifying ships and automobiles, which can also be reasonably expected given the unique features those classes have when compared to the other classes in the dataset. For reference, the inaccuracy values for each class have been plotted over time in the figure below. Each of the plots show that the training and testing inaccuracy values decreased over time, although the testing inaccuracy values varied greatly. It appears that this variance generally decreases over time, so it might have been decreased if I let the model train for a longer period of time. (Note: the plots in the figure below have different y-axis scales.)

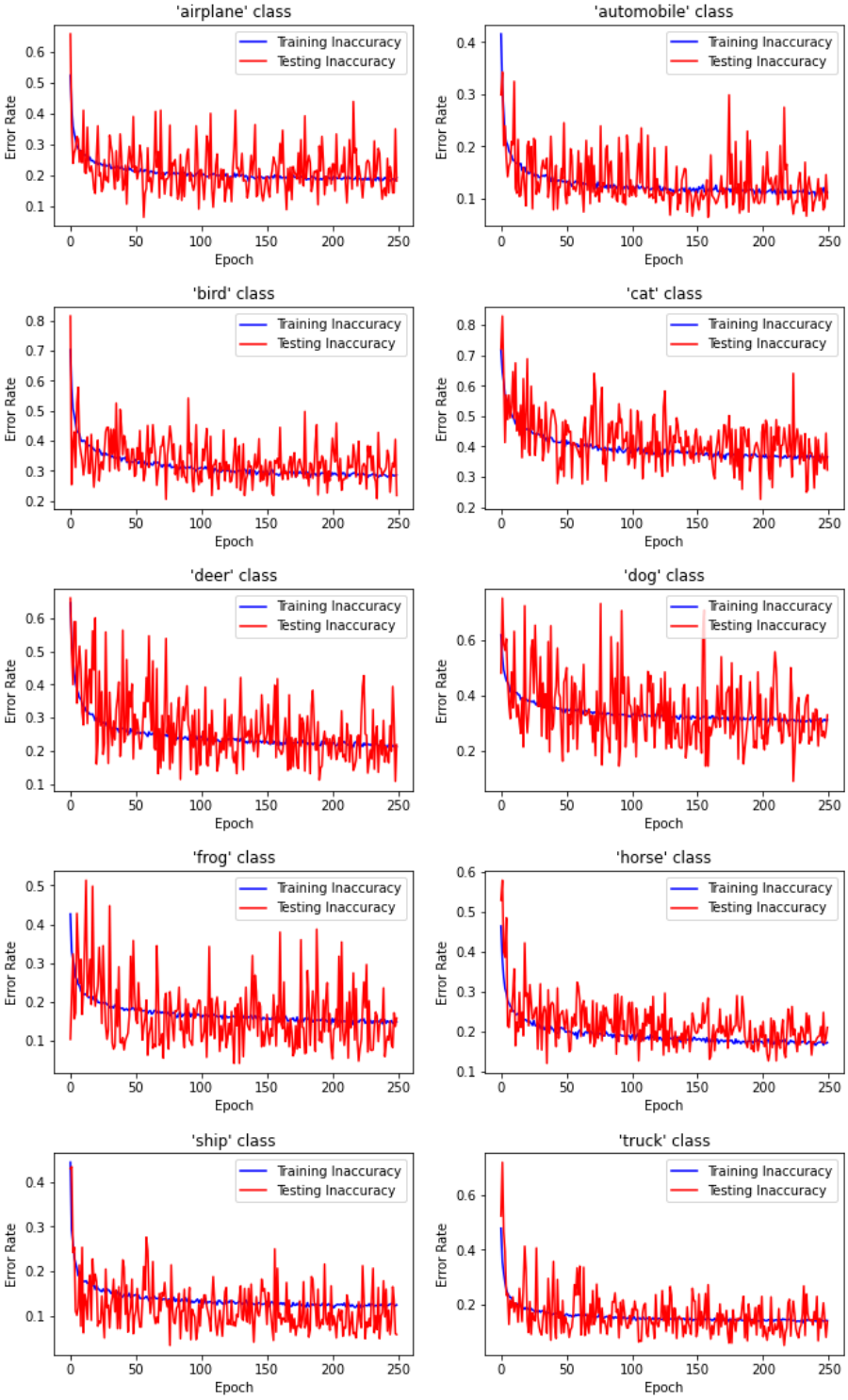

Figure 5: Training and testing inaccuracy values for each class plotted as a function of epochs

Visualization

1st Convolutional Layer Filters

To get a sense of the features that the model is learning to look for in the images, we can visualize the filters for the first convolutional layer. This visualization is shown in the figure below.

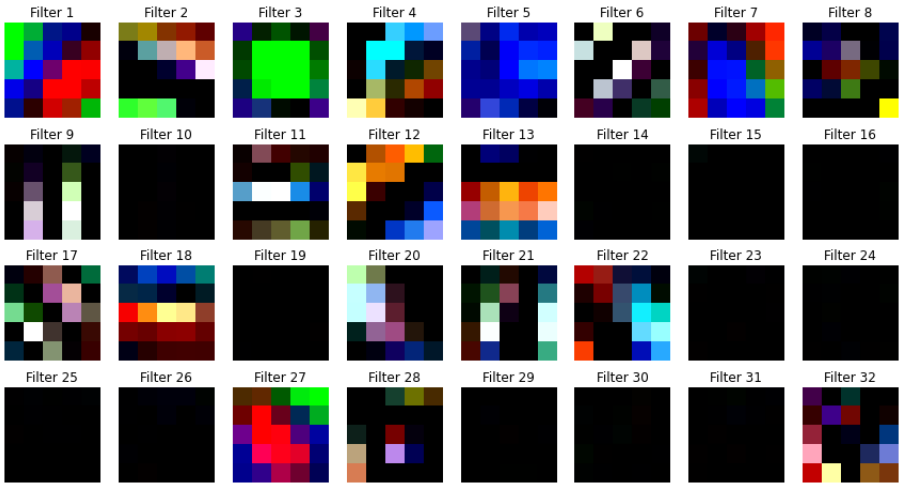

Figure 6: Visualization of the filters for the first convolutional layer

As we can see in the above figure, there are specific features that each filter has learned to recognize in the inputted image. For example, Filter 3 appears to be looking for a small green region in the image, Filter 6 appears to be looking for diagonal lines, and Filter 27 appears to be looking for a small group of red pixels next to an even smaller region of green pixels. We can also see that a portion of the filters appear to be empty; this is likely due to the filter weights being negative, since the plotting library will clip the float values to the range [0, 1] before plotting the image.

Side Note

One of the ways I verified that my implementation was correct was by looking to see if the model learned similar features each time it was trained. Below is a visualization of the filters that were learned when I trained the model using a learning rate of 0.005 and the Exponential Linear Unit (ELU) activation function. These altered parameters resulted in a slightly less accurate model, but I wanted to note that the filters generated strongly resemble the filters mentioned above! Although they are much brighter (due to the higher learning rate), it is clear to see that Filter 3 above and Filter 13 below look very similar, and so do Filter 21 above and Filter 18 below. I thought this was just something interesting I wanted to mention.

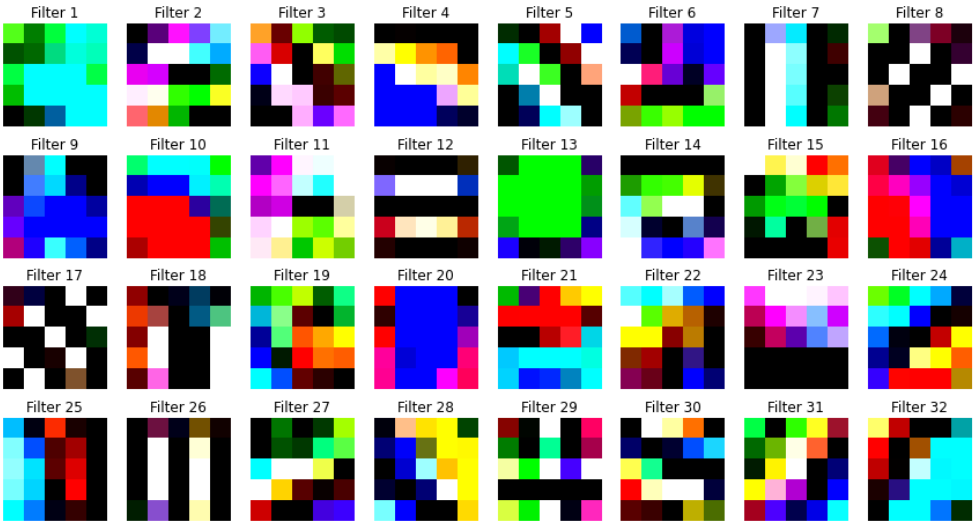

Figure 7: Visualization of the filters for the first convolutional layer after a previous training run

Conclusion

In conclusion, my trained model is able to classify images from the CIFAR-10 dataset with about 81% accuracy. Based on some benchmarking research, it seems that this is the accuracy I should expect for a model like this one with only three convolutional layers.

Future Considerations

In the future, I would employ a more robust method of tuning the hyper parameters. My method for this assignment was to train the model, look at the results, change a bunch of the parameters as I thought would best help the model, and then run it again. In hindsight, it would have been helpful to have a more strategic method of tuning one parameter at a time, although that might take a long time to optimize. Additionally, I would look into a method of loading a model I trained previously and continuing to train it to improve its accuracy. Every time I changed the hyper parameters, I had to start the training over from scratch, which might have wasted a lot of time that could have otherwise been spent fine-tuning a previously trained model.

Additionally, I did run into a memory issue when training the model. I'm not 100% sure of the cause of the error, but my entire model training process crashed at one point with no error message to be found. It seemed that training a model from scratch over and over again eventually consumed a significant portion of my computer's memory, causing it to run out of available memory space eventually. Restarting my computer seemed to fix it, but just to be sure I added a bit of code into my program to allow memory allocation growth on my GPU. In the future I would look into this error more and put in safeguards to ensure it did not happen again.

Model Loading & Evaluation

The model I created was built by subclassing the `tf.keras.Model` class. As such, it was saved using the standard Keras model saving process (`model.save("cnn_model")`) which saves the model in the `SavedModel` format. However, the model was not trained using the standard Keras `model.fit` method, and therefore it cannot be evaluated using the normal `model.evaluate` method. Included in the submitted .zip archive is a Python file named `evaluate.py`. This file can be used to evaluate my model's performance on a given set of testing data as follows:

1. At the top of the file, change the values for the parameters to match the locations of the files needed:
    - `TEST_DATA_PATH` should have the path to the testing images .npy file. This data should be non-normalized, i.e. each pixel is an integer in the range [0, 255].
    - `TEST_LABEL_PATH` should have the path to the testing labels .npy file. The labels should be integers in the range [0, 9], since the one-hot vectors will be built when the file is run.
    - `MODEL_PATH` should have the path to the folder containing the saved model to be evaluated. This will likely not need to be changed.
2. Run the python file.
3. The file will print the model's average loss, overall accuracy, class-wise error rate, and average error rate.

Please let me know if you have any issues with evaluating this model!