Background:

Waymo is a self-driving car company that is looking to revolutionize the world by bringing in cars that can safely drive people to and from where they need. Waymo has done a great job of training its cars near its hub of San Francisco. The company is even expanding to other American cities in the push to revolutionize the auto industry.

Meanwhile, some investors in Germany have been interested in the progress that Waymo has achieved in the US. German auto heavy-weights like Mercedes-Benz and BMW have similarly been testing autonomous driving in Germany with their fleet of cars and have achieved good success. To push the industry for more success, the German investors are thinking of bringing in Waymo to Germany to see if they can replicate their success outside of the US.

The German investors have reached out to Waymo to see if they have any interest in coming to Germany to test out their fleet. The investors have offered heavy financial backing if Waymo were to take on the challenge. While the task is challenging and does have risks, Waymo is very interested in testing its car outside of the country. This could be the start of its push to globalize its fleet and would offer a great opportunity to build its brand around the world.

After some internal discussions, Waymo has agreed that they would work to build an autonomous driving program in Germany. Before the program kicks off the German investors are very keen on testing Waymo’s model’s performance on German specific traffic signs and on general objects in images. If the models perform well then the investors will get some comfort on providing funds to Waymo. They decided to first test Waymo’s models on identifying German traffic signs as they are curious to see if their models or new models could adapt to non-US traffic signs.

Waymo has received 3 sets of data that represent German traffic signs. The 3 datasets are for training, validation and testing purposes. While Waymo already has many iterations of models that can identify traffic signs in the US, they do not have a very robust model for ones trained on German traffic signs. Waymo has decided to start from scratch and build a new model for this task.

Problem Statement:

Build a new image classification model that achieves high accuracy on all 3 datasets provided by the German investors. The goal is to build a model that doesn't overfit too much on the training data as this new model has to be robust for on-the-fly data when its implemented in the real world for autonomous driving. The hope is to get at least 95% accuracy with this initial model and then work on incremental changes for further improvement. My goal is to build a convolutional neural network (CNN) to classify if a traffic sign image is 1 of 43 signs.

Dataset Summary:

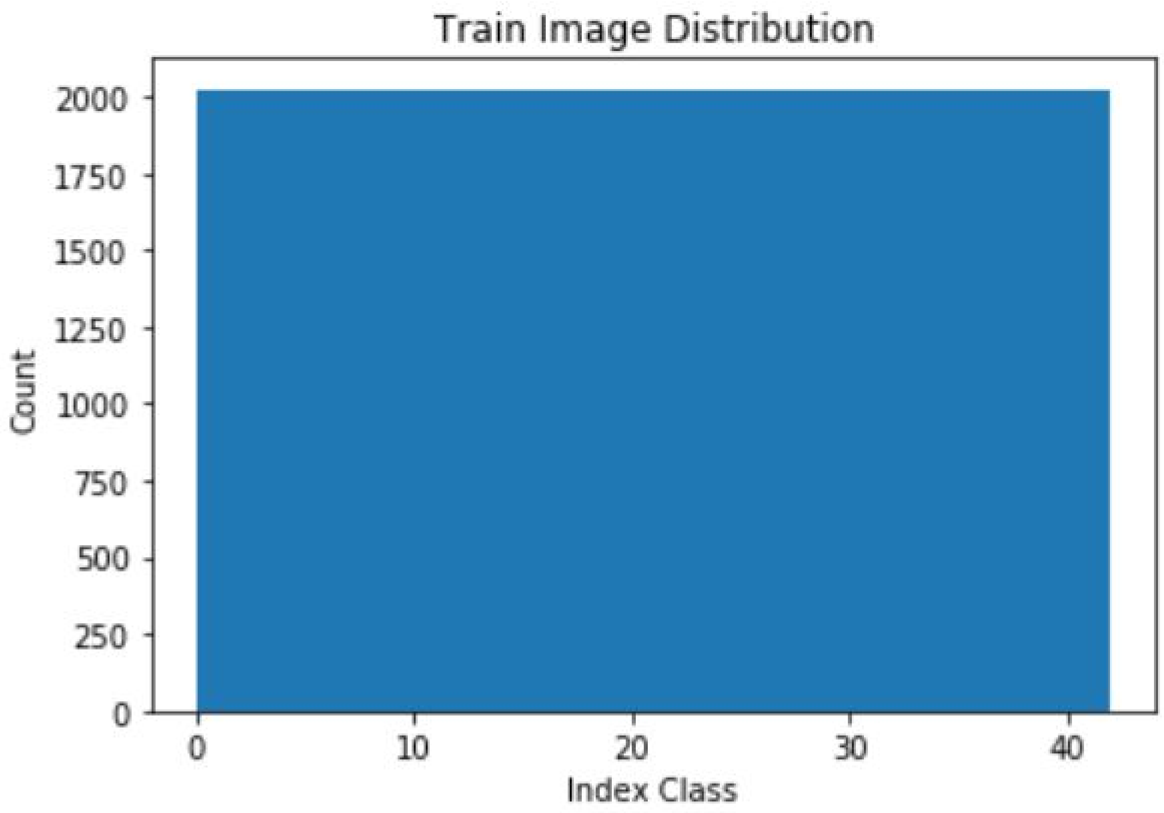
The German investors have provided us with 3 datasets of German traffic sign images. We were given a large amount of data with the training set having ~87K images, the validation set having ~4K images and the test set having ~13K images. The data is very clean and is in fact already in matrix form so I do not have to worry about manually changing any images to numerical representations. The data is already shuffled so I also do not have to worry about any bias in how the image data is ordered.

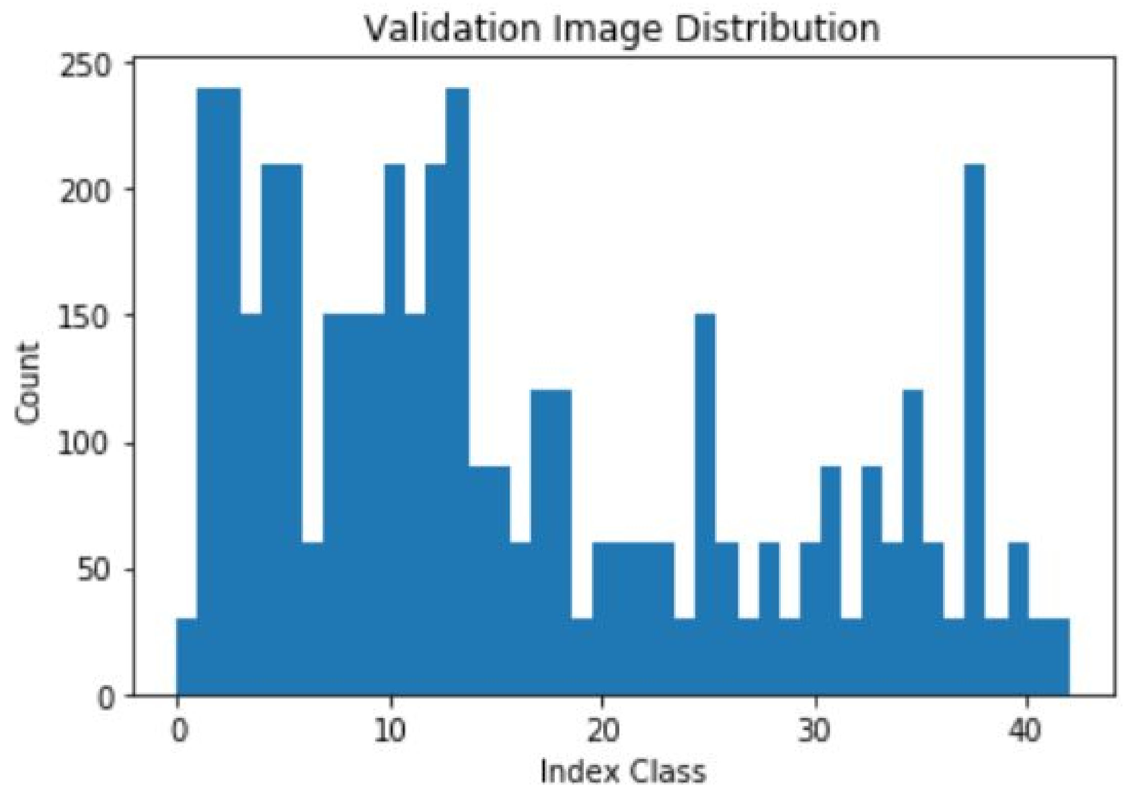
It is standard practice to scale the pixels of the images to a range from 0-1 and then to normalize that scaled data by subtracting the mean and then dividing by the standard deviation. The scaling is simply handled by dividing the pixel data by 255 in all 3 sets of data. For the normalization, I first find the mean and standard deviation of the training data and save that. Then for all 3 datasets, I subtract the training data mean and divide by training standard deviation. Generally speaking, best practice is to use only the training set to figure out how to scale/normalize. If you use the whole dataset to figure out the feature mean and variance, you are using knowledge about the distribution of the test set to set the scale of the training set thus 'leaking' information.

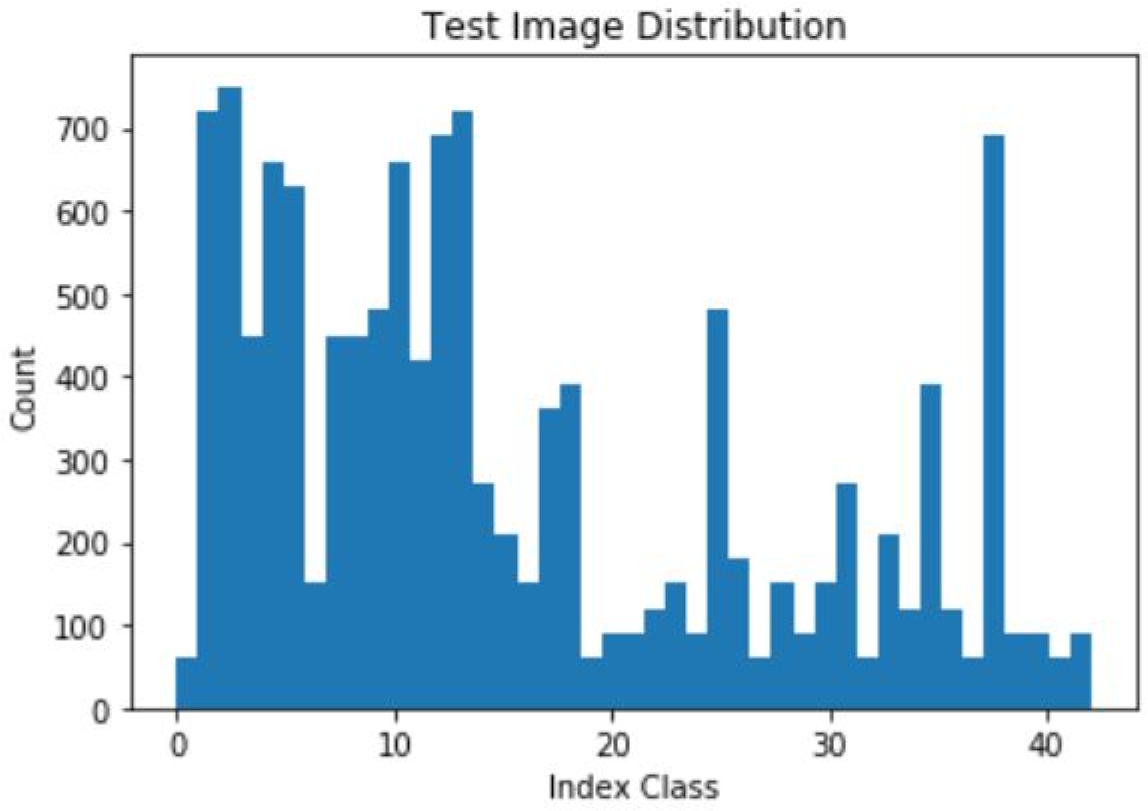
Given the clean data, I have not done any further data wrangling or data preprocessing. If I get further in my project and my classification model does not work as well as I planned, I may revisit data preprocessing as my image data may need some more advanced preprocessing to boost model performance. For example, perhaps I can use PCA or image whitening.

Given that is project is revolved around image classification there really is not much work to do in terms of statistical analysis. It is not very common to do this type of process when dealing with an image classification neural network. I also chatted with my colleague mentor and he agreed that robust statistical analysis would not be needed. His only recommendation was to test the class distribution among the 3 datasets.

Dataset Class Distributions:



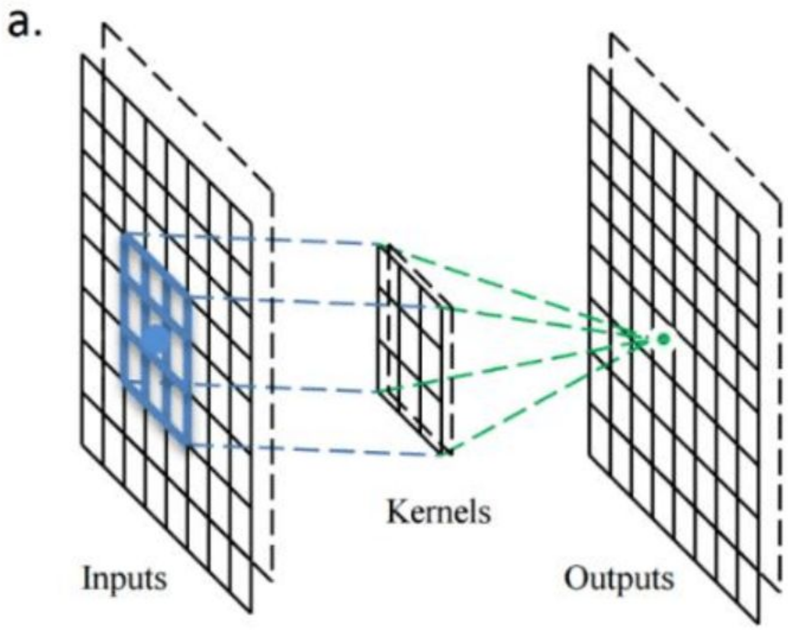

Validation Image Distribution
Count
Index Class
250
200
150
100
50
0
0
10
20
30
40


Test Image Distribution
Count
Index Class
700
600
500
400
300
200
100
0
0
10
20
30
40

The good news is that my training data has an equal balance of all 43 types of traffic signs. This means that my model should have good training experience on all types of signs. Thus the model should be good at classifying all signs instead of being biased to the signs that it experienced the most. The things to be cautious about are that both my validation and test datasets have different distributions compared to the distribution of the training data set. Usually one desires the distributions to be the same across all the datasets. In this case perhaps the true distribution of signs that will be experienced in the real-world will be closer to the distribution of the validation/test data sets. If this is the case, then this is an allowable caution as the model will be better equipped for the real world.

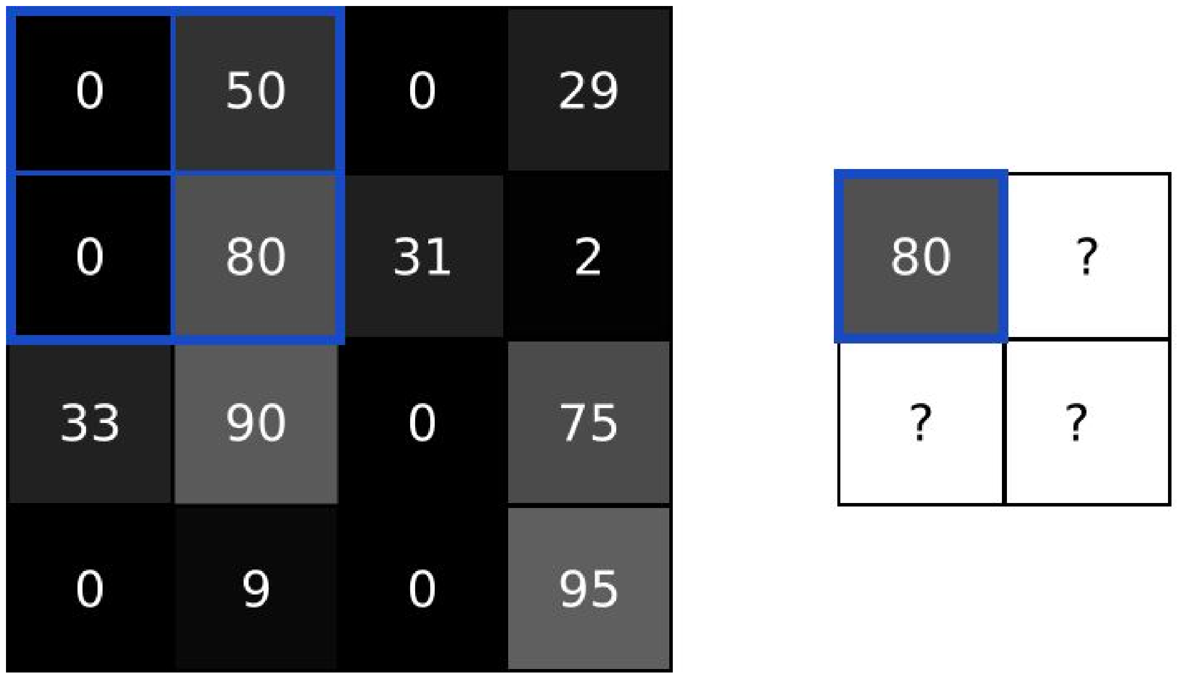
Model Background:

Given that the goal is to classify images properly, I will look to use a convolutional neural network (CNN) as this has been a proven model for image classification. Rather than looking at an entire image at once to find certain features it can be more effective to look at smaller portions of the image. A convolution is essentially sliding a filter over the input. Each convolutional layer contains a series of filters known as convolutional kernels. The filter is a matrix of integers that are used on a subset of the input pixel values, the same size as the kernel. Each pixel is multiplied by the corresponding value in the kernel, then the result is summed up for a single value for simplicity representing a grid cell, like a pixel, in the output channel/feature map.

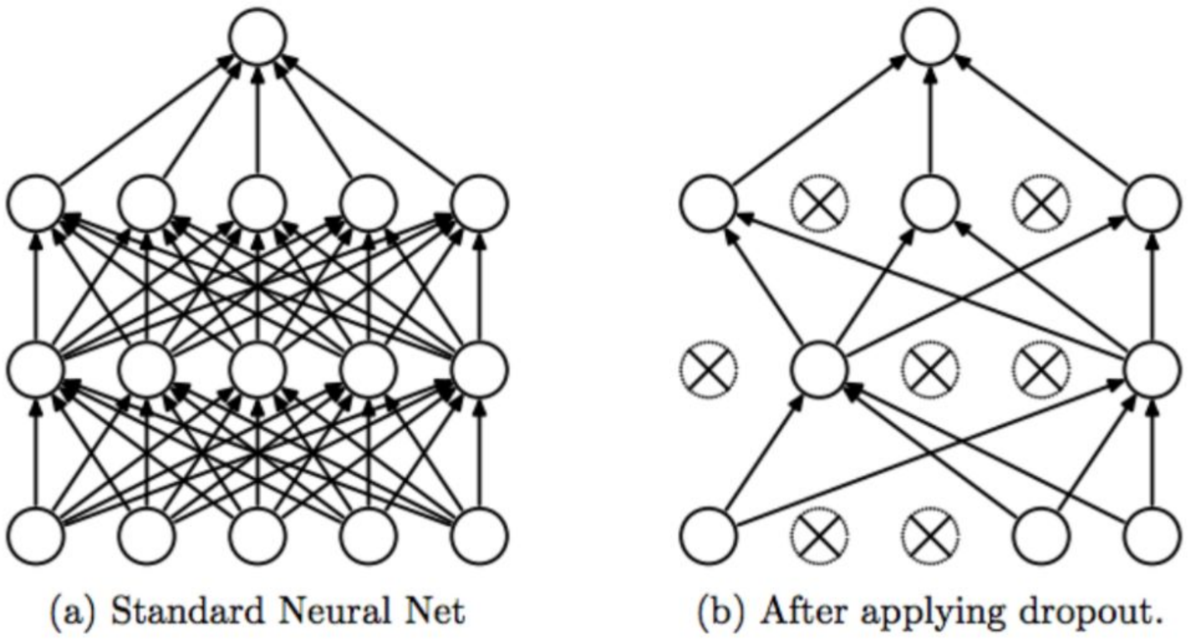


A stride one 3x3 convolutional kernel acting on a 8x8 input image, outputting an 8x8 filter/channel. Source: https://www.researchgate.net/figure/a-Illustration-of-the-operation-principle-of-the-convolution-kernel-convolutional-layer_fig2_309487032

Similarly CNNs often use max-pooling layers. Neighboring pixels in images tend to have similar values, so convolutional layers will typically also produce similar values for neighboring pixels in outputs. As a result, much of the information contained in a convolution layer's output is redundant. For example, if we use an edge-detecting filter and find a strong edge at a certain location, chances are that we will also find relatively strong edges at locations 1 pixel shifted from the original one. However, these are all the same edge thus we are not finding anything new. Pooling layers solve this problem. All they do is reduce the size of the input it is given by pooling values together in the input. The pooling is usually done by a simple operation like max, min, or average. A max-pool example is below.



In an initial iteration of the model, there was a lot of overfit on the training data. To counteract this I have implemented drop-out layers. Dropout forces a neural network to learn more robust features that are useful in conjunction with many different random subsets of the other neurons. During training, some number of layer outputs are randomly ignored or “dropped out.” This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer. In effect, each update to a layer during training is performed with a different “view” of the configured layer. Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs. This conceptualization suggests that perhaps dropout breaks-up situations where network layers co-adapt to correct mistakes from prior layers, in turn making the model more robust

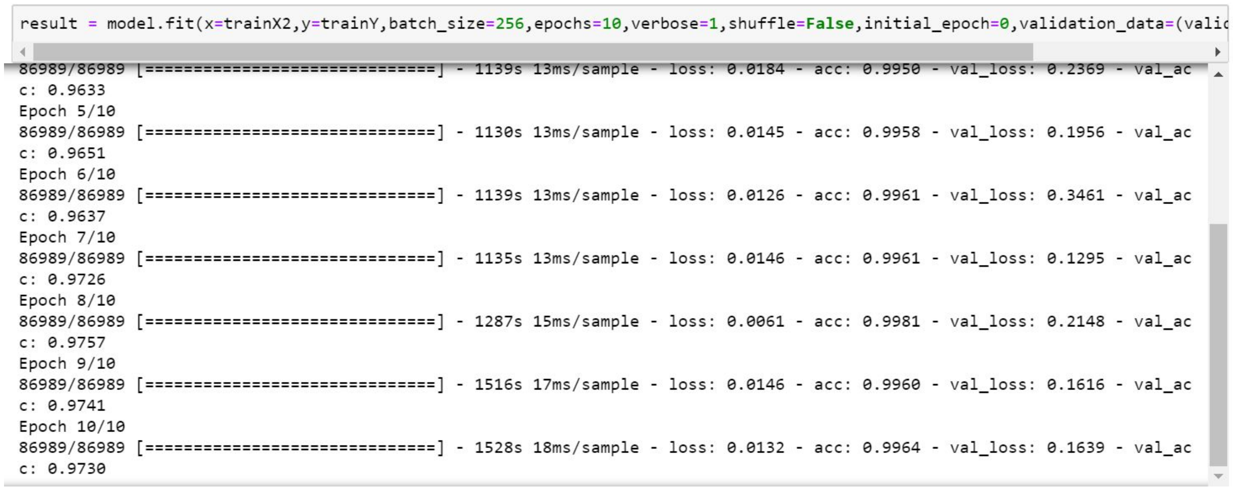


Srivastava, Nitish, et al. "Dropout: a simple way to prevent neural networks from overfitting", JMLR 2014

I look to build my own basic CNN by relying on heuristics that many other practitioners have come across. In the first stage, I look to use a few convolutional layers along with some max-pooling layers. Then I plan to flatten out the data and put it through a dense layer and finally have a softmax activation in the last dense layer to calculate the class probabilities. Code for the model can be seen at https://github.com/jvel76/git.

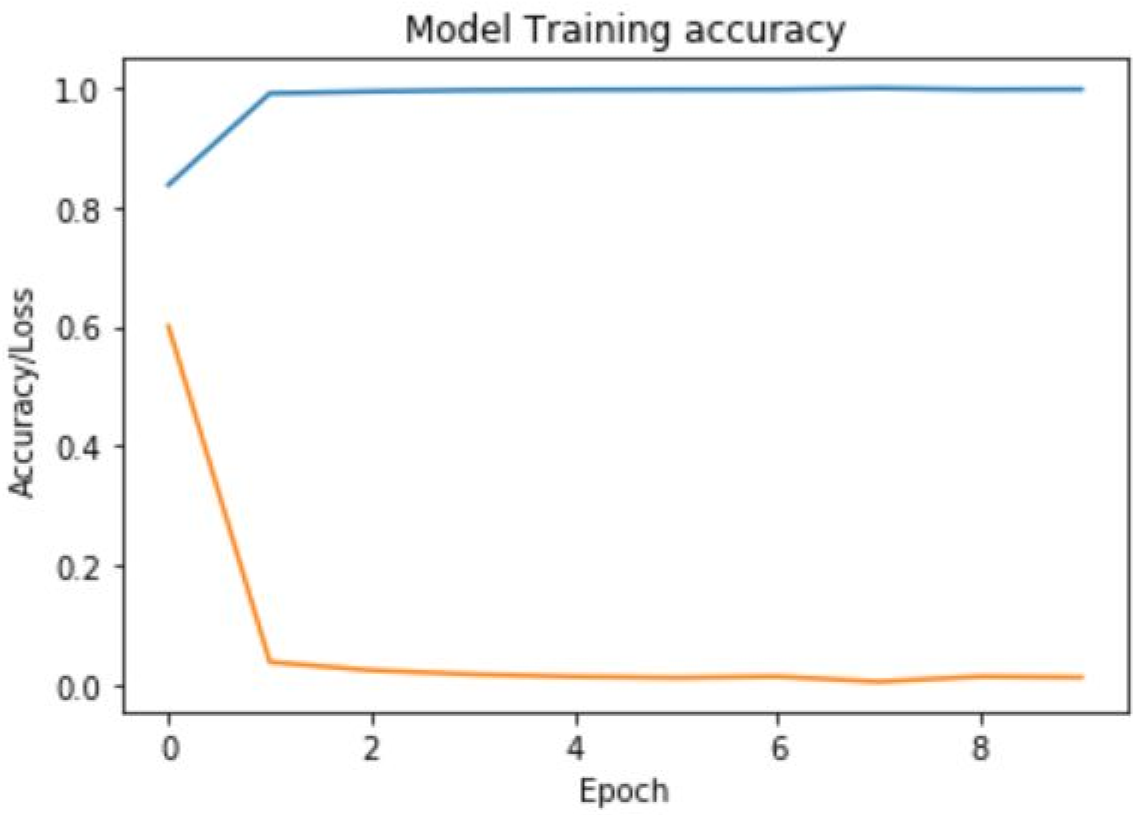
Model Results:

So the model is doing very well on the training data with over 99% accuracy. This is not very surprising given the good quality and size of training data I am working with. The validation and test accuracies are hovering over 97% and 96%, respectively. Compared to a previous iteration of my CNN, I have significantly decreased the overfitting which makes the model better for real world implementation.



```
result = model.fit(x=trainX2,y=trainY,batch_size=256,epochs=10,verbose=1,shuffle=False,initial_epoch=0,validation_data=(valic
```

```
86989/86989 [==============================] - 1139s 13ms/sample - loss: 0.0184 - acc: 0.9950 - val_loss: 0.2369 - val_ac
c: 0.9633
Epoch 5/10
86989/86989 [==============================] - 1130s 13ms/sample - loss: 0.0145 - acc: 0.9958 - val_loss: 0.1956 - val_ac
c: 0.9651
Epoch 6/10
86989/86989 [==============================] - 1139s 13ms/sample - loss: 0.0126 - acc: 0.9961 - val_loss: 0.3461 - val_ac
c: 0.9637
Epoch 7/10
86989/86989 [==============================] - 1135s 13ms/sample - loss: 0.0146 - acc: 0.9961 - val_loss: 0.1295 - val_ac
c: 0.9726
Epoch 8/10
86989/86989 [==============================] - 1287s 15ms/sample - loss: 0.0061 - acc: 0.9981 - val_loss: 0.2148 - val_ac
c: 0.9757
Epoch 9/10
86989/86989 [==============================] - 1516s 17ms/sample - loss: 0.0146 - acc: 0.9960 - val_loss: 0.1616 - val_ac
c: 0.9741
Epoch 10/10
86989/86989 [==============================] - 1528s 18ms/sample - loss: 0.0132 - acc: 0.9964 - val_loss: 0.1639 - val_ac
c: 0.9730
```



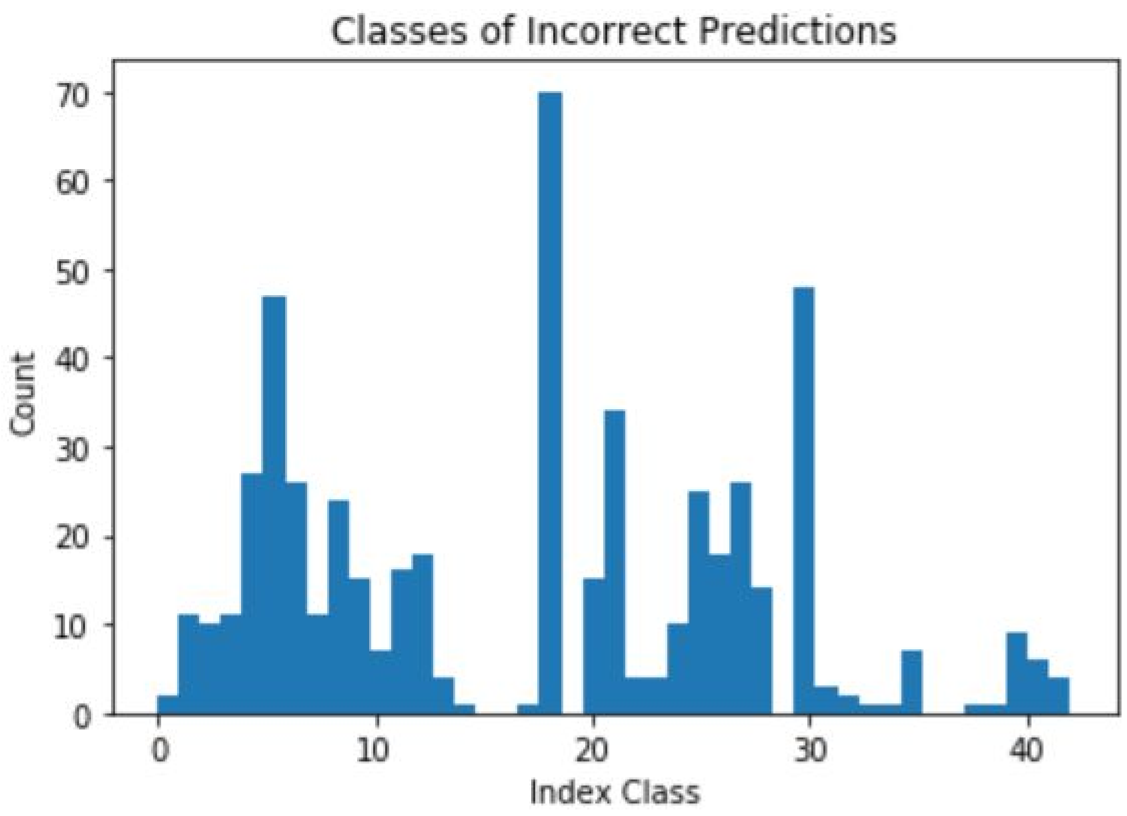
```
valid = model.evaluate(validationX2,validationY,verbose=2)
```

```
4410/4410 - 29s - loss: 0.1639 - acc: 0.9730
```

```
test = model.evaluate(testX2,testY,verbose=2)
```

```
12630/12630 - 86s - loss: 0.2651 - acc: 0.9577
```

One interesting thing to notice is that relatively, the model is doing really badly at identifying Pedestrian, Double Curve and Beware of ice/snow signs. This is something we should caution when presenting our results. Overall the model passes the initial target of 95% accuracy across all 3 datasets but there does seem to be room for improvement in the incorrect classes previously mentioned. The next steps could be to see if PCA or image whitening preprocessing can help with improvement, especially on the badly mislabeled classes.


Classes of Incorrect Predictions
Count
Index Class