|  |  |  |
| --- | --- | --- |
| **Udacity** |  | **Patrick Harris** |
| **Machine Learning Engineer Nano Degree** |  | **May 4, 2018** |
|  |  |  |

Capstone Project: State Farm Distracted Driver Detection

Project Definition

**Overview**

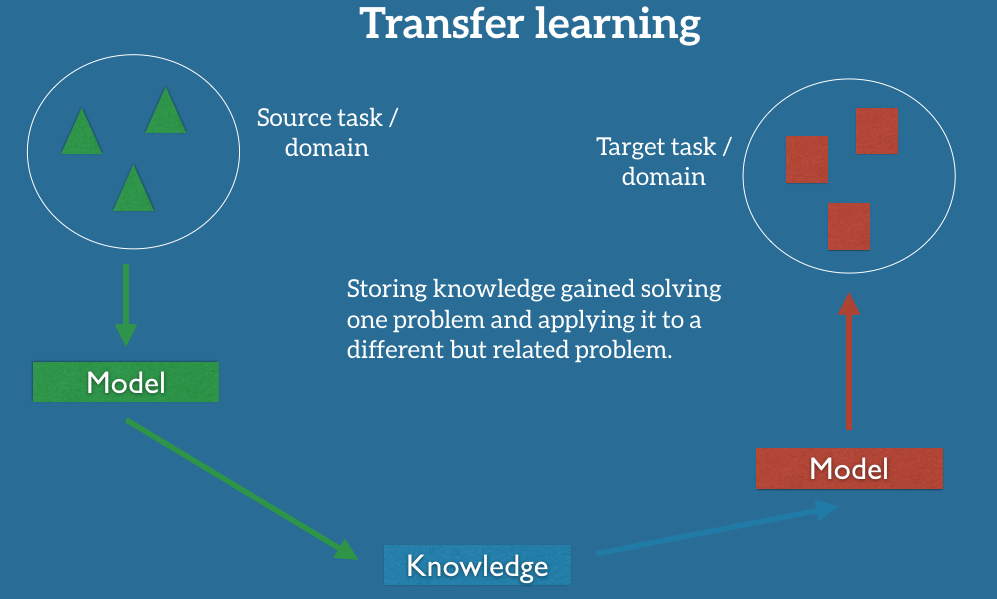
State Farm conducted a Kaggle competition to find the best approach to identifying distracted drivers. The following statistics are pulled directly from the competition overview. According to the CDC motor vehicle safety division, [one in five car accidents](http://www.cdc.gov/motorvehiclesafety/distracted_driving/) is caused by a distracted driver. Sadly, this translates to 425,000 people injured and 3,000 people killed by distracted driving every year. State Farm hopes to improve these alarming statistics, and better insure their customers, by testing whether dashboard cameras can automatically detect drivers engaging in distracted behaviors.

State Farm is attempting to change the auto insurance industry by leveraging dashboard cameras to detect distracted drivers. In order to achieve this goal, State Farm needs to first discover if Machine Learning models can correctly identify images of distracted drivers vs. non-distracted drivers.

**Solution Approach**

A common approach to image recognition today, is the use of a Convolutional Neural Network (CNN) with transfer learning and data augmentation. Images can be represented as a matrix of pixel values and a convolutional utilizes the matrices to extract and learn features of an image. To gain a better understanding of how a [CNN](https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/) works visit the link to see a visual representation.

Since CNNs rely heavily on very large datasets (millions of records) and require powerful computer processing, it is common practice to leverage [transfer learning](https://medium.com/@14prakash/transfer-learning-using-keras-d804b2e04ef8). Transfer learning is a technique in which the CNN uses the weights from a pre-trained network as initializations or a fixed feature extractor. This technique reduces the amount of computing power required to train the CNN and allows the model to learn on fewer images. The below graphic is taken from an excellent website on [transfer learning](http://ruder.io/transfer-learning/index.html#whytransferlearningnow) and is hyperlinked for further reference.



Another common practice in Machine Learning is [Data augmentation](https://medium.com/ymedialabs-innovation/data-augmentation-techniques-in-cnn-using-tensorflow-371ae43d5be9), which is a method that creates modified images from the original. This technique is used to supply additional images to train the model. Examples of data augmentation are rotating an image along an axis, flipping/mirroring an image or adding random distortions (noise) to an image.

**Data Review**

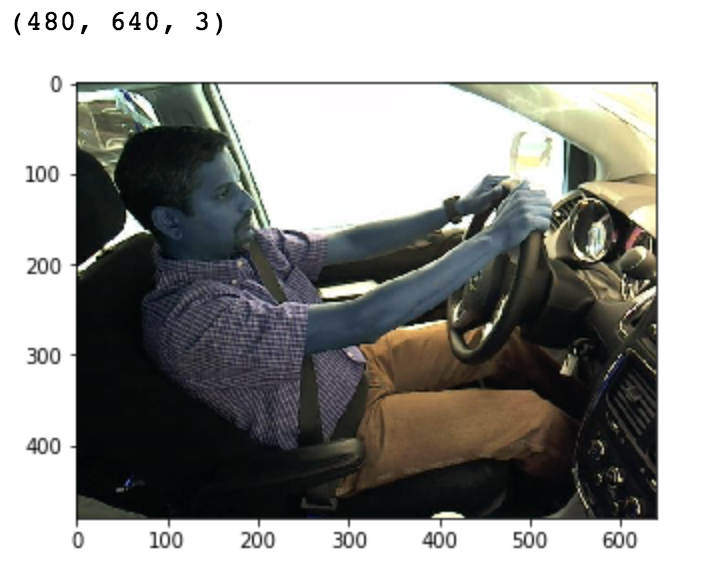
**Volume:**

There are 22,424 training images provided by State Farm in this competition. Additionally, the data set contains nearly 80,000 test images. In order to increase the number of images available to training, I’ve decided to use data augmentation. By using random rotation of the images and adding random noise to each image this increases the images by 3 times. I chose to only use these two methods of data augmentation, as each is highly likely to occur in a dashcam view. The camera can easily be shifted or rotated causing the images to be off center. Additionally, the camera can get covered with dust, debris or fingerprints to cause noise in the image.

Another method to increase the amount of training is to leverage K-Fold cross validation, which is not typically used in CNNs. However, it is leveraged in cases like this, where the training data set is limited. The main drawback of using K-Folds cross validation is the added time to training and testing. As each fold has to run through the series of specified epochs and each fold is tested using the weights from each training.

**Dimensions:**

In any image recognition project, the dimensions of the images are an important aspect. As previously stated, CNNs rely on the image’s pixels being represented in a matrix and the shape of the matrix depends on the dimensions of the image. Below I show a sample of the images supplied in this project. The first number is the number of pixel rows in the image, the second number is the number of pixel columns, the last number is the number of color channels. 3 channels represent red, blue, green; whereas 1 channel represents a grayscale image.

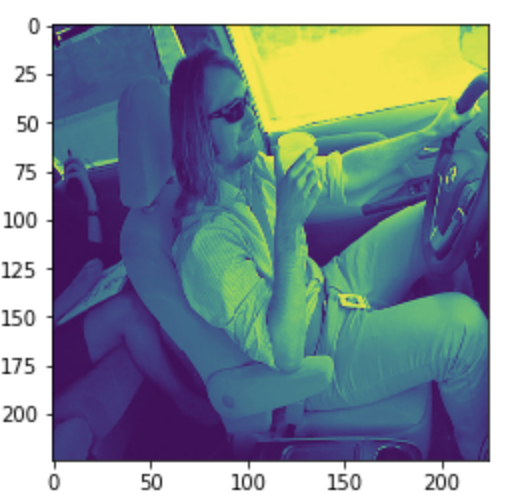


**Metrics**

In the capstone proposal, I had suggested that success would be measured using the multi-class loss score that is used in the Kaggle competition. However, I did not realize this would not be possible, as the competition does not provide the target variable for each of the test images, which is needed to calculate the score. Since the test results cannot be validated, I will use the Validation Accuracy as a measure of success, with the goal of achieving above 70%.

**Implementation**

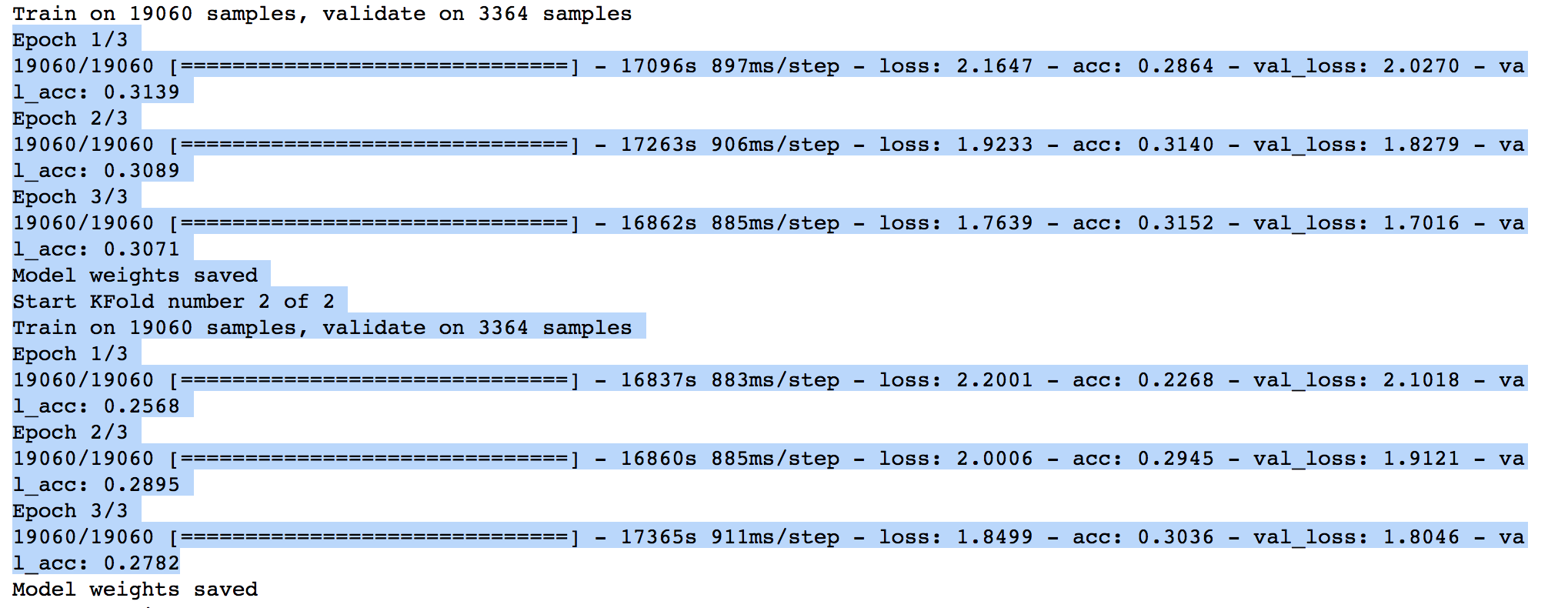
I chose to use [Keras](https://keras.io/applications/), an open source neural network library witten in Pyton, to create the CNN using transfer learning. Keras has several different model architectures pretrained on [Imagenet](http://www.image-net.org/). This project is highly suitable for using Imagenet data as the datasets are very similar and the amount of training data available is limited. In order to utilize a pretrained VGG16 the images must to be resized to 224 rows and 224 columns. Below is another sample that has been resized to 224, 224, 3.



I encountered several challenges with this project, primarily with computing resources. My first attempt at running the Jupyter Notebook was on a MacBook Pro with a 3.5GHz Intel Core i7 and 16GB RAM. The first VGG16 model architecture was configured with 2 folds and 3 epochs, training 15 of the 16 layers. The timing to process one epoch was 180 hours. 180 \* 3 epochs \* 2 folds = 1080 / 24hrs = 45 days. This does not account for time to process the test data.

At this point, I moved to Amazon Web Services (AWS) on an EC2 GPU instance. I saw a vast improvement using the same model architecture with each epoch taking 4.75 hours to train. However, when the model was done training and ready to process the test data, the system encountered an Out of Memory (OOM) error. AWS customer support recommended that I use a spot instance, track memory usage and apply a memory SWAP if needed.

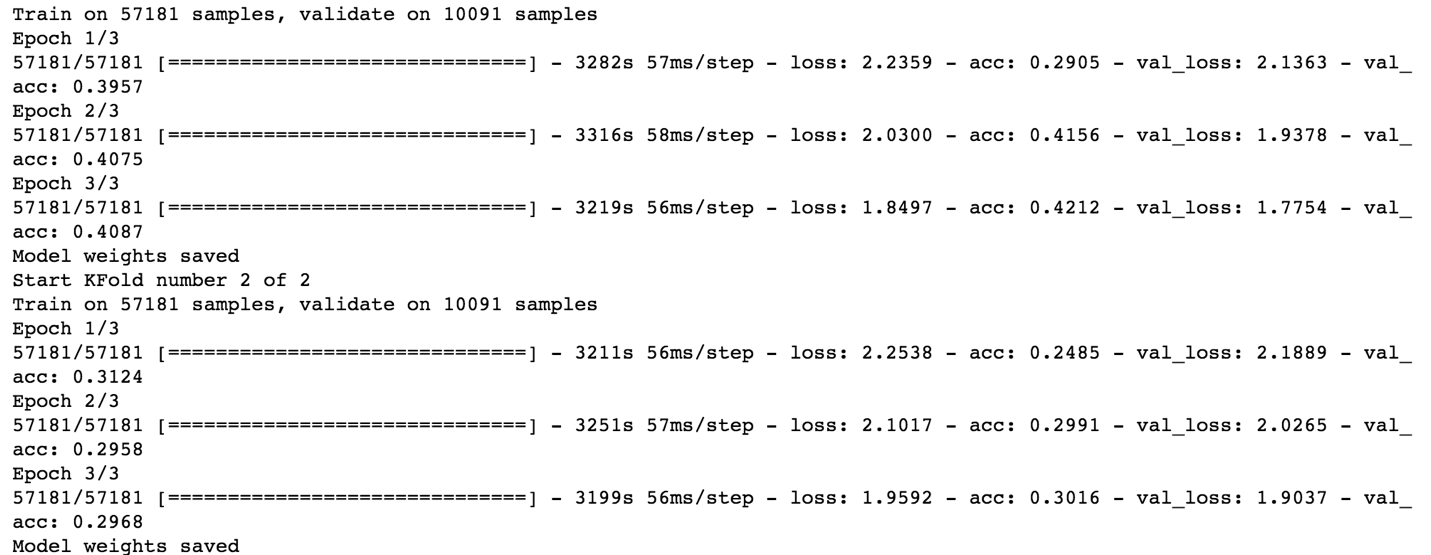
Moving to the spot instance proved to be the difference, getting me past the computing resource issue. My new challenge, was that my first set of results was less than desired.



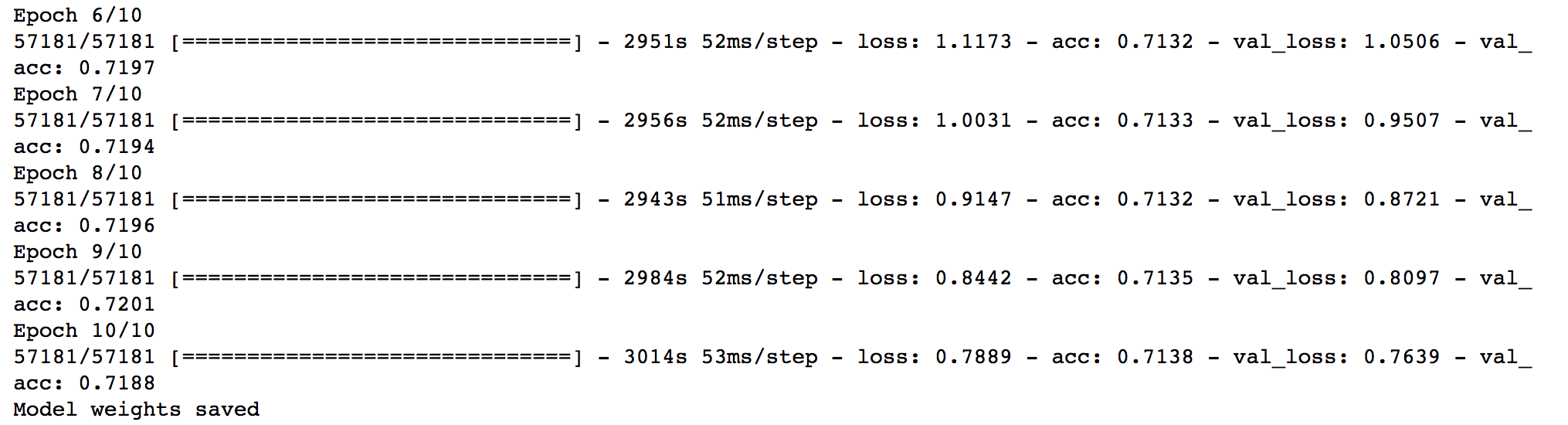
The model accuracy was only ~31%. At this point, I started to look for methods to further reduce the training time. This lead me to a simplified model architecture, in which I set the first 8 layers of the model to non-trainable, meaning the weights would not be updated for first 8 layers. This provided a giant boost in processing time as each epoch took roughly 50 minutes, but the accuracy in the model only improved slightly if at all.

Next, I added the data augmentation (random rotation and random noise) to see what boost in accuracy that would provide.

The increase was significant, accuracy moved up to ~40%

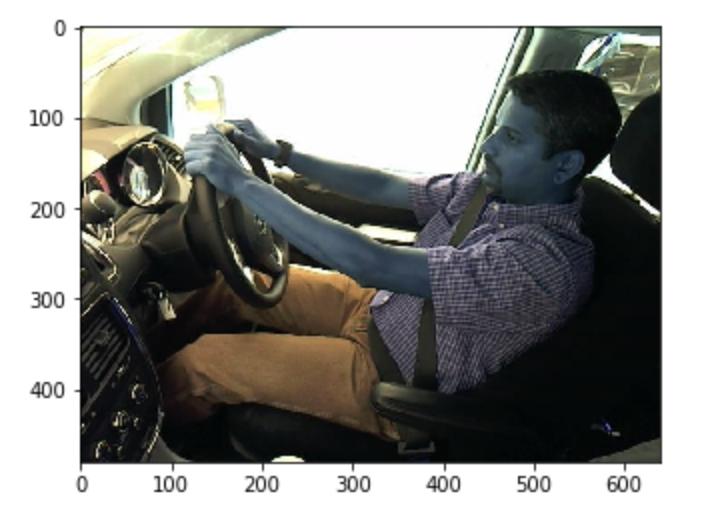


Given that both loss categories (training loss/validation loss) were still quite high, above ~1.7, I moved on to increasing the number of epochs. Based on the amount of time per epoch I chose to increase the number to 10 epochs but kept the folds at 2. The addition of epochs netting very positive results, ~71% accuracy.



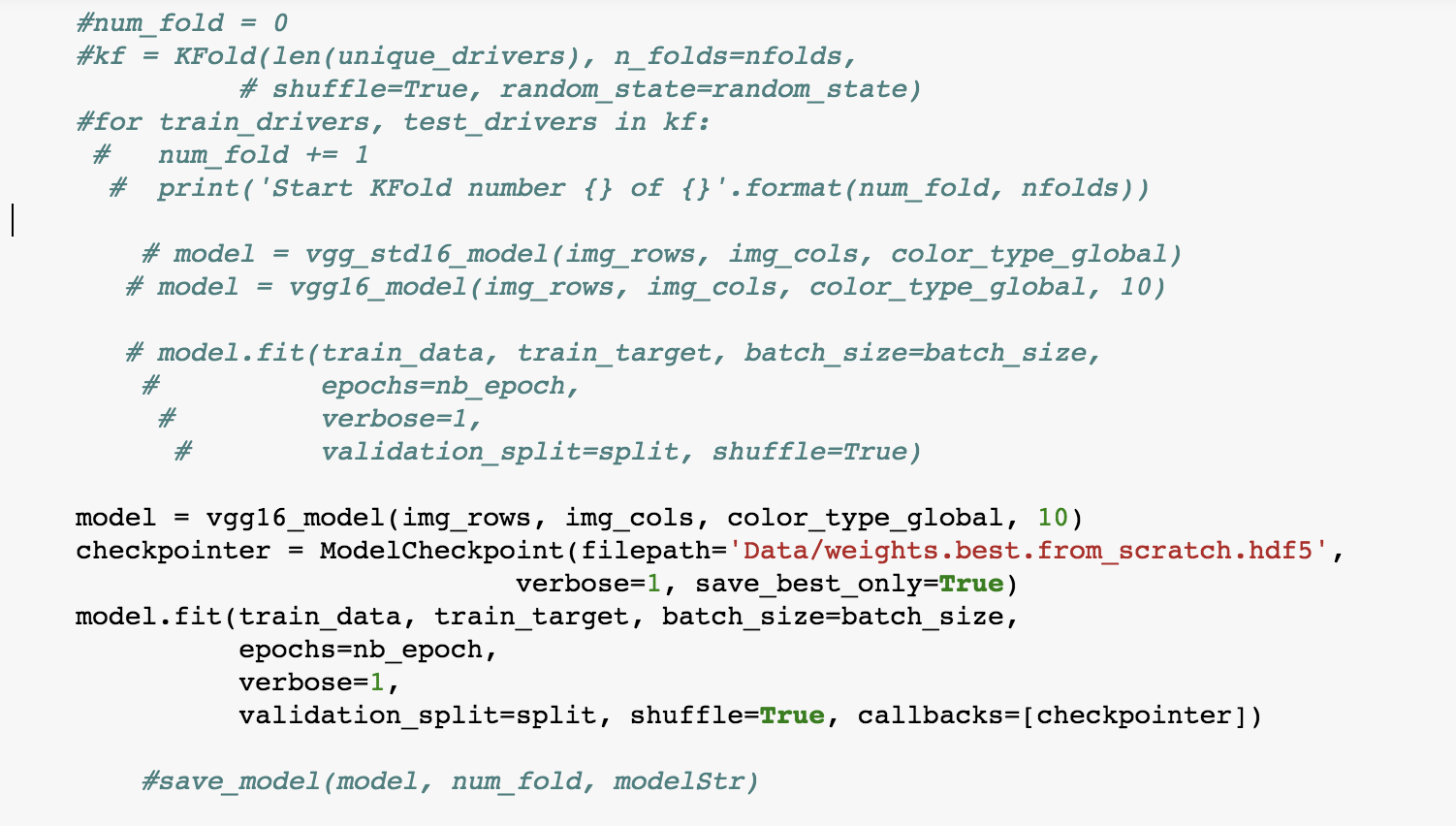
**Continued Improvements**

To continue increasing the amount of training data, I added another data augmentation which flips/mirrors the image.

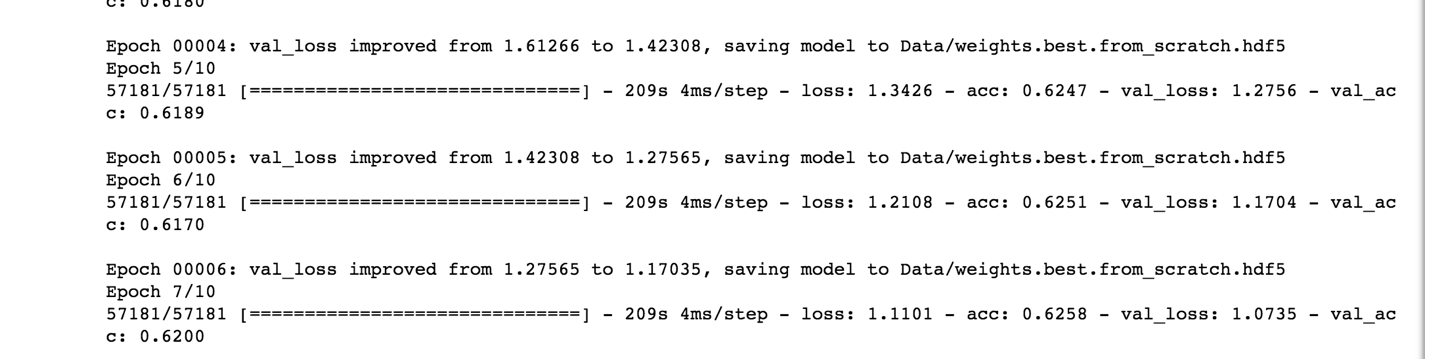


Additionally, since the validation loss was still quite a bit above zero, at .75, it made sense to increase epochs to 20. Unfortunately, this plan did not net the expected results. The addition of data seemed to cause the data to overfit, as validation accuracy was eroding as validation loss was converging towards 0. Next, I tested increasing the folds to 5 and lowering the epochs to 6, as I was noticing the accuracy seemed to max out near the 3rd and 4th epoch. At this point, I began experiencing OOM, out of memory, errors within my AWS instance. Only 4 of the 5 folds would execute, and without any improvement in the accuracy.

Lastly, I decided to move away from using k-fold cross validation and use the ModelCheckpoint approach, which allows for saving only the best weights.



Unfortunately, this did not provide any gains in validation accuracy. However, I did not encounter any OOM issues on the AWS p3.8xlarge instance and the overall speed in which the data was processed through the model increased drastically using the latest GPU processors.



**Final Thoughts**

This project was very challenging and gained a better understanding on how to solve Machine Learning problems. I learned how to leverage the many different libraries in Python to solve image classification problems, such as Keras, numpy, and scikit-learn. As well as, how to use a pre-trained network and apply transfer learning. Additionally, I have learned how to augment datasets to create additional training data. Lastly, I learned how to properly utilize AWS; creating an image for easy reuse on any instance, leveraging spot instances, fleet instances and dedicated instances, and the variations in GPU instances