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

Capstone Project: State Farm Distracted Driver Detection

Project Definition

Overview

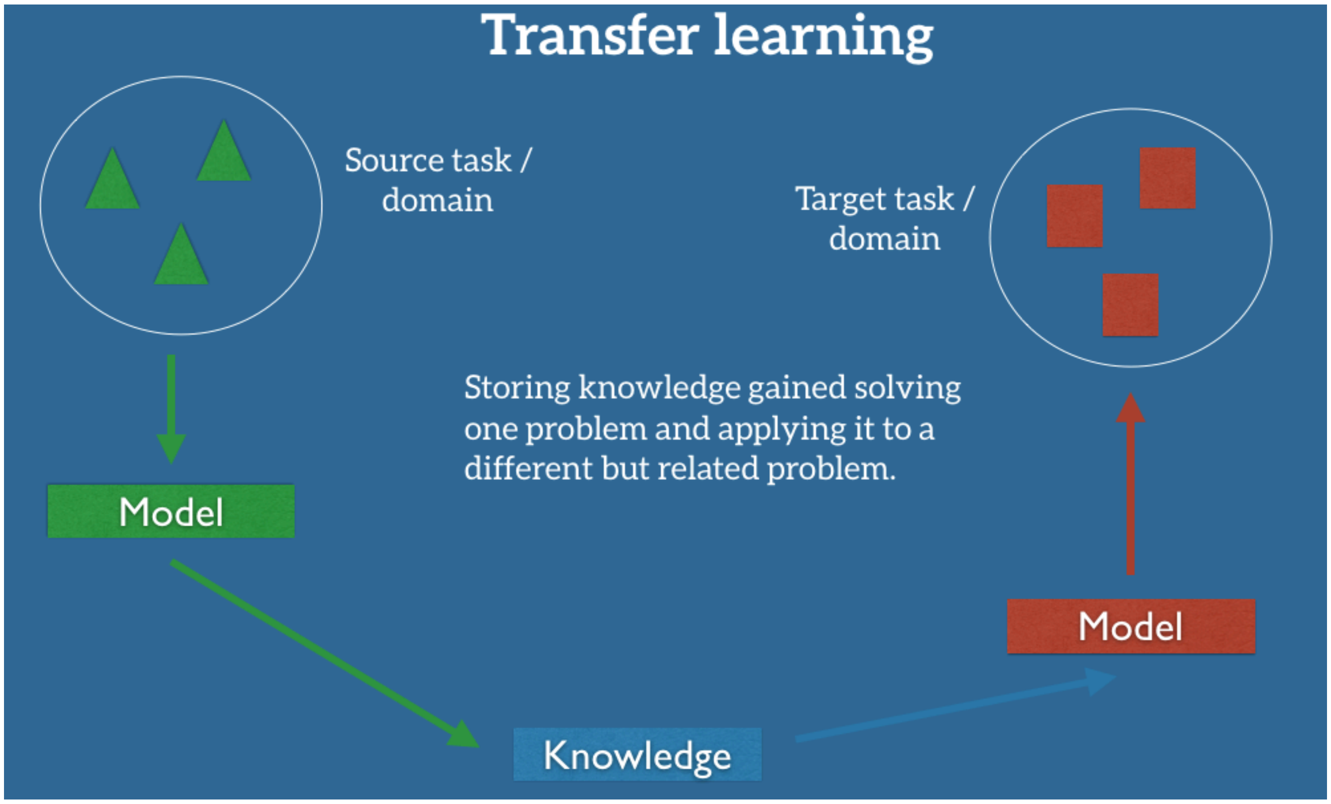
State Farm conducted a Kaggle competition to find the best approach to identifying distracted drivers. The following statistics are pulled directly from the competition overview. According to the CDC motor vehicle safety division, one in five car accidents is caused by a distracted driver. Sadly, this 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 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. 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 and is hyperlinked for further reference.


Transfer learning
Source task / domain
Target task / domain
Storing knowledge gained solving one problem and applying it to a different but related problem.
Model
Model
Knowledge

Another common practice in Machine Learning is Data augmentation, 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.

```
(480, 640, 3)
```

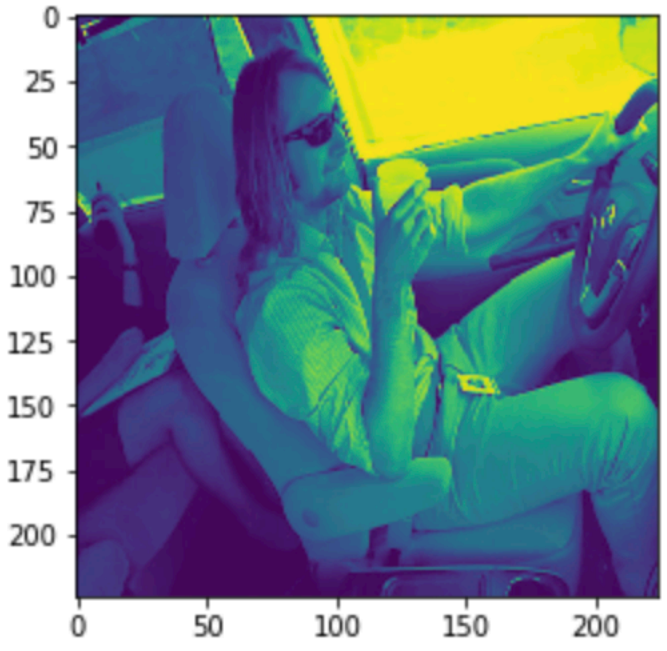
Benchmark

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. Additionally, there is not a direct correlation between multiclass loss score and validation accuracy, so determining an acceptable measure was an arbitrary process. I considered the task at hand. Accurately identify distracted drivers in 10 different classes, and this being my first end-to-end image recognition project, I decided 70%+ validation accuracy would be a good target.

Some performance metrics were available from the Kaggle discussion, in which one submission was experiencing 1760 seconds per epoch on a GPU processor. Given that this is a research project for State Farm, the time to train, validate and test a model, is not a major constraint. If State Farm moves forward with this project beyond research, there will need to be further design of how to support the end-to-end use case. Based upon these details, I chose to use twice the performance results listed in the Kaggle competition, 3520 seconds per epoch.

Implementation

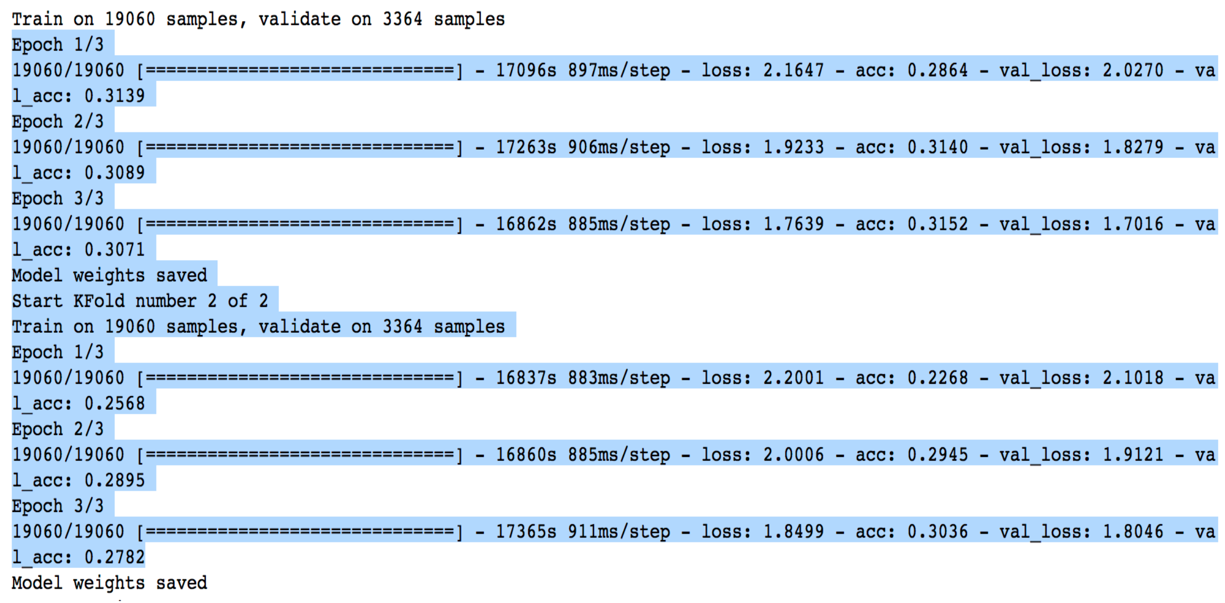
I chose to use Keras, an open source neural network library written in Python, to create the CNN using transfer learning. Keras has several different model architectures pretrained on Imagenet. This project is highly suitable for using Imagenet data as the datasets are very similar and the amount of 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.

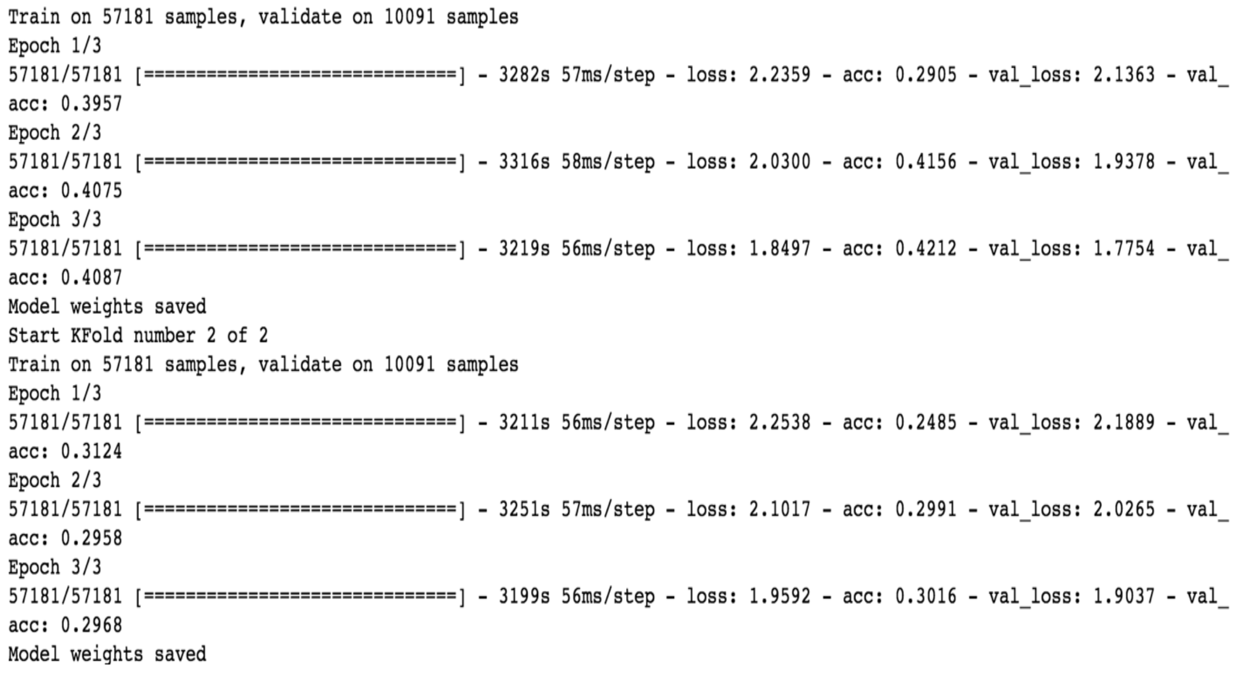
Moving to the spot instance proved to be the difference, getting me past the computing resource issue. My new challenge, was that my first set of results was less than desired.



```
Train on 19060 samples, validate on 3364 samples
Epoch 1/3
19060/19060 [==============================] - 17096s 897ms/step - loss: 2.1647 - acc: 0.2864 - val_loss: 2.0270 - va
l_acc: 0.3139
Epoch 2/3
19060/19060 [==============================] - 17263s 906ms/step - loss: 1.9233 - acc: 0.3140 - val_loss: 1.8279 - va
l_acc: 0.3089
Epoch 3/3
19060/19060 [==============================] - 16862s 885ms/step - loss: 1.7639 - acc: 0.3152 - val_loss: 1.7016 - va
l_acc: 0.3071
Model weights saved
Start KFold number 2 of 2
Train on 19060 samples, validate on 3364 samples
Epoch 1/3
19060/19060 [==============================] - 16837s 883ms/step - loss: 2.2001 - acc: 0.2268 - val_loss: 2.1018 - va
l_acc: 0.2568
Epoch 2/3
19060/19060 [==============================] - 16860s 885ms/step - loss: 2.0006 - acc: 0.2945 - val_loss: 1.9121 - va
l_acc: 0.2895
Epoch 3/3
19060/19060 [==============================] - 17365s 911ms/step - loss: 1.8499 - acc: 0.3036 - val_loss: 1.8046 - va
l_acc: 0.2782
Model weights saved
```

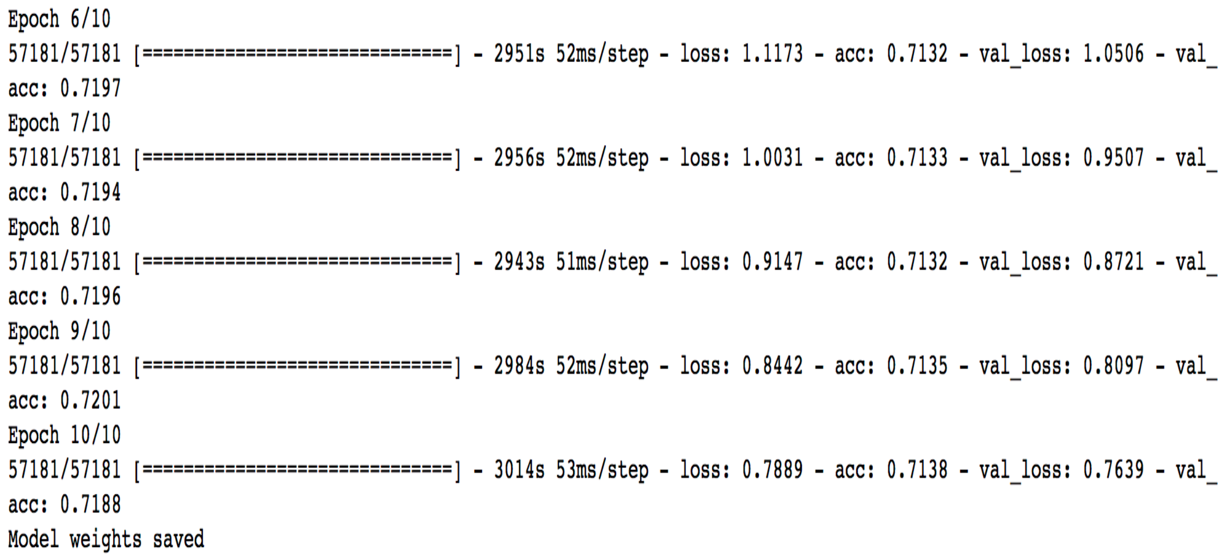
The model accuracy was only ~31%. At this point, I started to look for methods to further reduce the training time. This lead me to a simplified model architecture, in which I set the first 8 layers of the model to non-trainable, meaning the weights would not be updated for first 8 layers. This provided 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%



```
Train on 57181 samples, validate on 10091 samples
Epoch 1/3
57181/57181 [==============================] - 3282s 57ms/step - loss: 2.2359 - acc: 0.2905 - val_loss: 2.1363 - val_
acc: 0.3957
Epoch 2/3
57181/57181 [==============================] - 3316s 58ms/step - loss: 2.0300 - acc: 0.4156 - val_loss: 1.9378 - val_
acc: 0.4075
Epoch 3/3
57181/57181 [==============================] - 3219s 56ms/step - loss: 1.8497 - acc: 0.4212 - val_loss: 1.7754 - val_
acc: 0.4087
Model weights saved
Start KFold number 2 of 2
Train on 57181 samples, validate on 10091 samples
Epoch 1/3
57181/57181 [==============================] - 3211s 56ms/step - loss: 2.2538 - acc: 0.2485 - val_loss: 2.1889 - val_
acc: 0.3124
Epoch 2/3
57181/57181 [==============================] - 3251s 57ms/step - loss: 2.1017 - acc: 0.2991 - val_loss: 2.0265 - val_
acc: 0.2958
Epoch 3/3
57181/57181 [==============================] - 3199s 56ms/step - loss: 1.9592 - acc: 0.3016 - val_loss: 1.9037 - val_
acc: 0.2968
Model weights saved
```

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.



```
Epoch 6/10
57181/57181 [==============================] - 2951s 52ms/step - loss: 1.1173 - acc: 0.7132 - val_loss: 1.0506 - val_
acc: 0.7197
Epoch 7/10
57181/57181 [==============================] - 2956s 52ms/step - loss: 1.0031 - acc: 0.7133 - val_loss: 0.9507 - val_
acc: 0.7194
Epoch 8/10
57181/57181 [==============================] - 2943s 51ms/step - loss: 0.9147 - acc: 0.7132 - val_loss: 0.8721 - val_
acc: 0.7196
Epoch 9/10
57181/57181 [==============================] - 2984s 52ms/step - loss: 0.8442 - acc: 0.7135 - val_loss: 0.8097 - val_
acc: 0.7201
Epoch 10/10
57181/57181 [==============================] - 3014s 53ms/step - loss: 0.7889 - acc: 0.7138 - val_loss: 0.7639 - val_
acc: 0.7188
Model weights saved
```

Continued Improvements

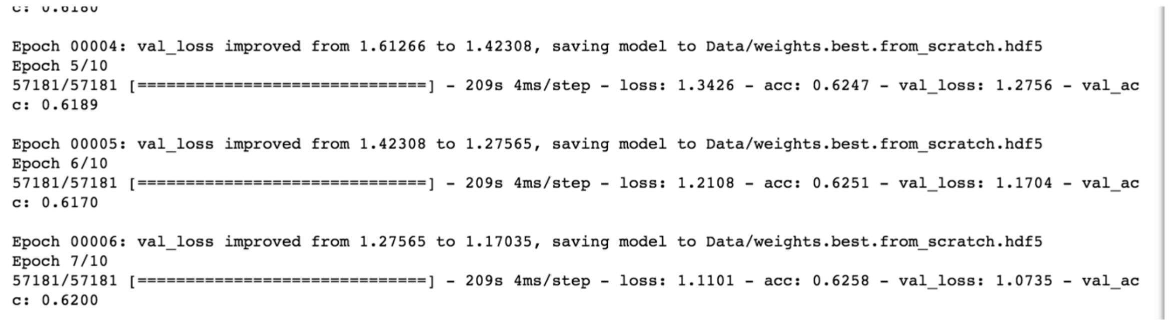
To continue increasing the amount of training data, I added another data augmentation which flips/mirrors the image.

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.

```
    #num_fold = 0
    #kf = KFold(len(unique_drivers), n_folds=nfolds,
              # shuffle=True, random_state=random_state)
    #for train_drivers, test_drivers in kf:
     #   num_fold += 1
      #  print('Start KFold number {} of {}'.format(num_fold, nfolds))

        # model = vgg_std16_model(img_rows, img_cols, color_type_global)
       # model = vgg16_model(img_rows, img_cols, color_type_global, 10)

       # model.fit(train_data, train_target, batch_size=batch_size,
        #          epochs=nb_epoch,
         #         verbose=1,
          #        validation_split=split, shuffle=True)

    model = vgg16_model(img_rows, img_cols, color_type_global, 10)
    checkpointer = ModelCheckpoint(filepath='Data/weights.best.from_scratch.hdf5',
                               verbose=1, save_best_only=True)
    model.fit(train_data, train_target, batch_size=batch_size,
              epochs=nb_epoch,
              verbose=1,
              validation_split=split, shuffle=True, callbacks=[checkpointer])

        #save_model(model, num_fold, modelStr)
```

Unfortunately, this did not provide any gains in validation accuracy. However, I did not encounter any OOM issues on the AWS p3.8xlarge instance. The overall speed in which the data was processed through the model increased drastically using the latest GPU processors at 209 seconds per epoch. This is significantly lower than what was reported on the Kaggle discussion of 1760 seconds.



```
c: 0.6180

Epoch 00004: val_loss improved from 1.61266 to 1.42308, saving model to Data/weights.best.from_scratch.hdf5
Epoch 5/10
57181/57181 [==============================] - 209s 4ms/step - loss: 1.3426 - acc: 0.6247 - val_loss: 1.2756 - val_ac
c: 0.6189

Epoch 00005: val_loss improved from 1.42308 to 1.27565, saving model to Data/weights.best.from_scratch.hdf5
Epoch 6/10
57181/57181 [==============================] - 209s 4ms/step - loss: 1.2108 - acc: 0.6251 - val_loss: 1.1704 - val_ac
c: 0.6170

Epoch 00006: val_loss improved from 1.27565 to 1.17035, saving model to Data/weights.best.from_scratch.hdf5
Epoch 7/10
57181/57181 [==============================] - 209s 4ms/step - loss: 1.1101 - acc: 0.6258 - val_loss: 1.0735 - val_ac
c: 0.6200
```

Final Thoughts

This project was very challenging, and I 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