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

Capstone Project: State Farm Distracted Driver Detection

**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.

**Objective**

The goal of this project is to develop an image recognition model to detect distracted drivers.

**Solution Approach**

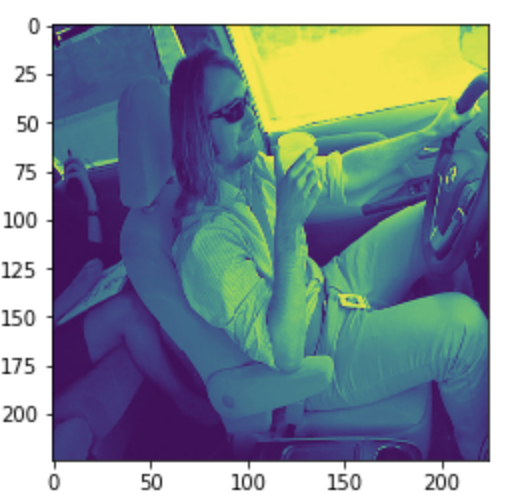
1. Review the data to gain an understanding of the dataset
2. Rescale images
3. Split data into training and validation sets (test images separate file)
4. Create an augmented dataset
5. Construct CNN model architecture (transfer learning)
6. Compile model
7. Train model
8. Load model with best weights
9. Score model
10. Repeat steps 6-10 until fine-tuned and experiment with combination of augmented dataset and without

**Analysis**

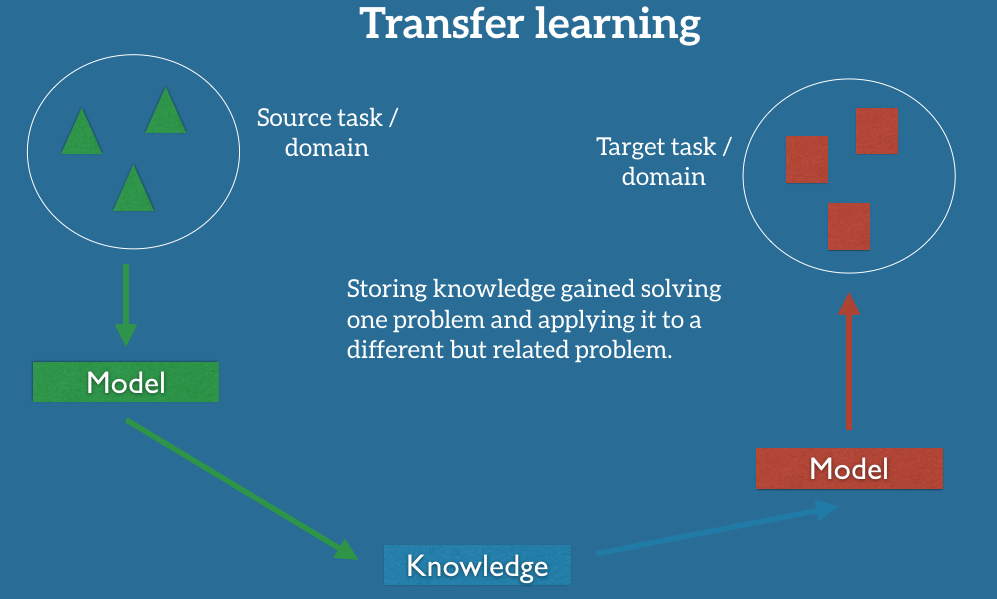
There are 22,424 training images available with the data set and nearly 80,000 test images. In order to increase the number of images available to training, I’ve decided to use data augmentation. By using random rotation of the images and adding random noise to each image this increases the images by 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.

To support [transfer learning](http://ruder.io/transfer-learning/index.html#whytransferlearningnow) of a pretrained VGG16 the images needed to be resized to 224 rows and 224 columns.



Transfer learning allows us to leverage already existing labeled data of a related domain. This knowledge gained in solving the root task are then applied to the related task. The below graphic is taken from an excellent website on transfer learning and is hyperlinked above.



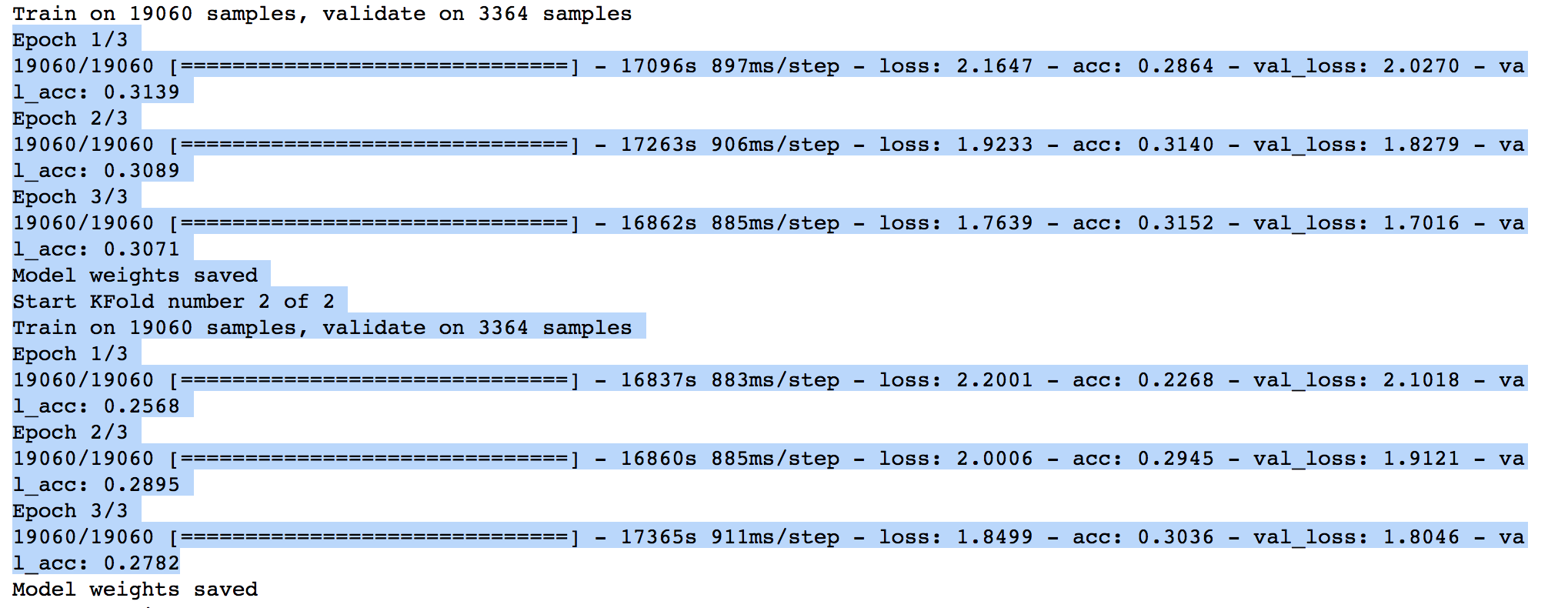
I chose to use transfer learning for this project for three key reasons. First, [Keras](https://keras.io/applications/) has several different models pretrained on [Imagenet](http://www.image-net.org/). Second, this project is highly suitable for using Imagenet data for image recognition. Lastly, it helps reduce the amount of time to train the model, as you are able to specify which layers in the CNN are non-trainable, meaning the those weights are not updated during training.

**Challenges**

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.

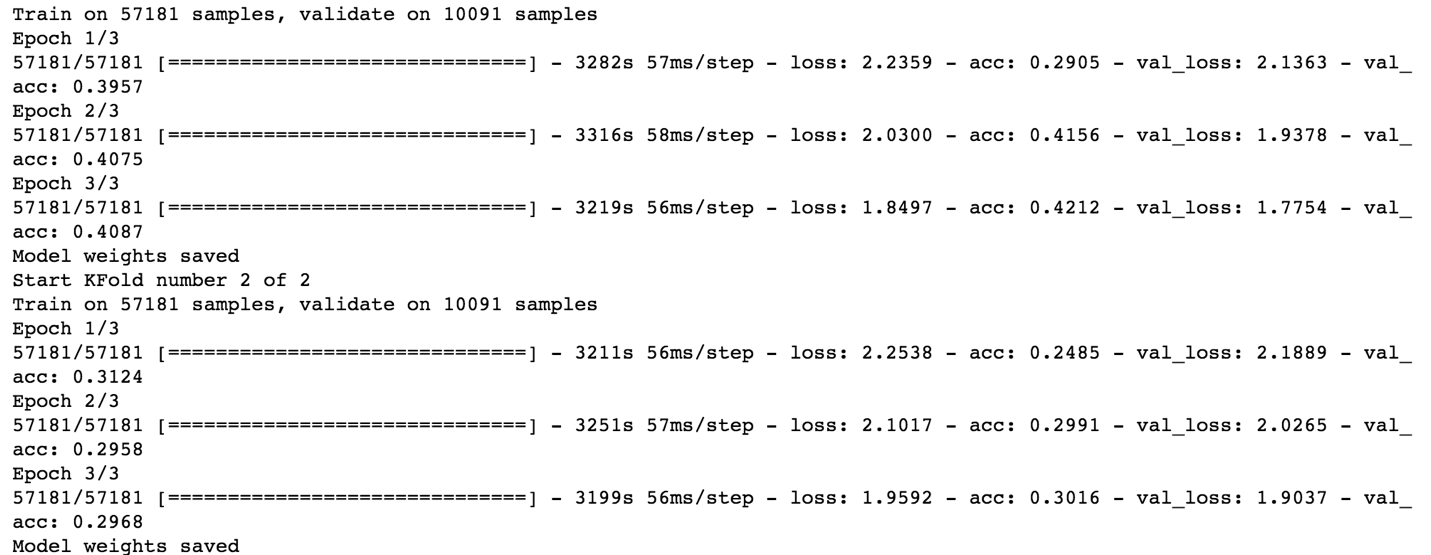
Moving to the spot instance proved to be the difference, getting me past the computing resource issue. My new challenge, was that my first set of results was less than desired.



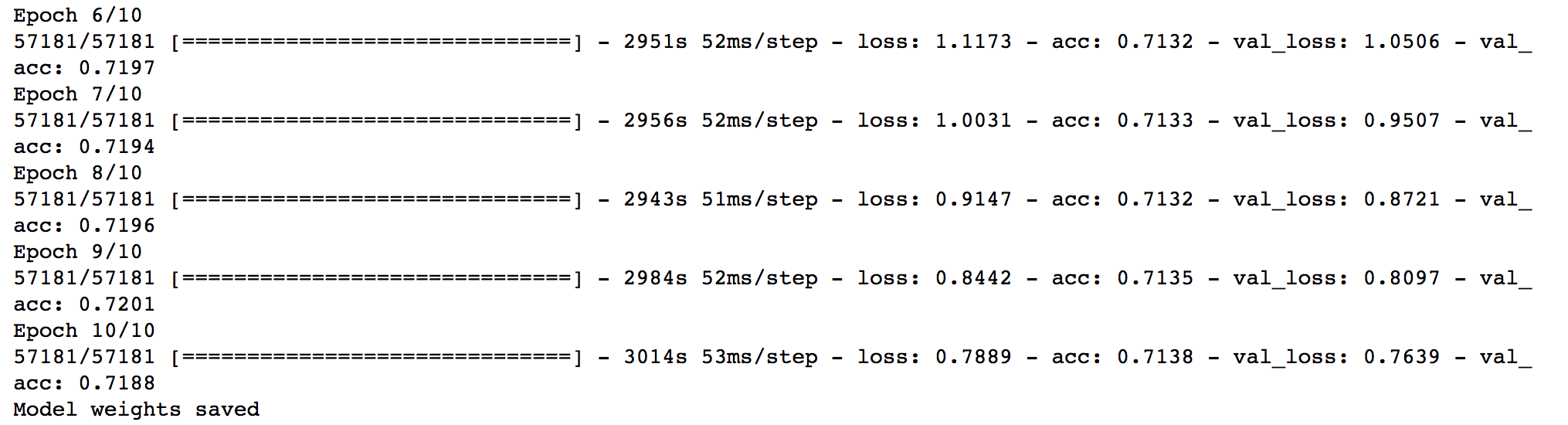
The model accuracy was only ~31%. At this point, I started to look for methods to further reduce the training time. This lead me to a simplified model architecture, in which I set the first 8 layers of the model to non-trainable, meaning the weights would not be updated for first 8 layers. This provided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



**Improvements**

1. Continue to increase the number of epochs, as the loss values are still near .70
2. Add image scaling data augmentation, which would increase the training data set
3. Test with additional folds
4. Use a different pre-trained model, such as InceptionResNetV2 or DenseNet