Machine Learning Engineer Nanodegree

Capstone Project: **State Farm Distracted Driver Detection' from Kaggle 2016 competition.**

Nidhi Gupta

April14th, 2018

I. Definition

Project Overview

We've all been there: a light turns green and the car in front of you doesn't budge. Or, a previously unremarkable vehicle suddenly slows and starts swerving from side-to-side.

When you pass the offending driver, what do you expect to see? You certainly aren't surprised when you spot a driver who is texting, seemingly enraptured by social media, or in a lively hand-held conversation on their phone.

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. Given a dataset of 2D dashboard camera images, State Farm gave challenge to classify each driver's behavior. Are they driving attentively, wearing their seatbelt, or taking a selfie with their friends in the backseat.

We all know that it is risky to get distracted while driving, still I am surprised to see people texting and talking while driving specially teenagers. Recently I have heard of some teens having major accidents because of distracted driving. As a parent I want it to be safer for my kids when they start driving. If all insurance companies can have a good algorithm to classify distracted behavior then it can bring down the accidents drastically. Insurance companies will be able to pass the savings to their customers and the safe drivers can have their insurance premium significantly lowered. This will provide motivation for people to follow the guidelines of safe driving.

Problem Statement

In this project we are given driver images, each taken in a car with a driver doing something in the car (texting, eating, talking on the phone, makeup, reaching behind, etc). Our goal is to predict the likelihood of what the driver is doing in each picture.

The 10 classes to be predicted are:

c0: normal driving (*normal driving requires both hands on the steering wheel and forward gaze*" above)
c1: texting – right (*phone in the right hand and person texting*)
c2: talking on the phone – right (*phone in the right hand placed over the right ear*)
c3: texting – left (*phone in the left hand and person texting*)
c4: talking on the phone – left (*phone in the left hand placed over the left ear*)
c5: operating the radio (*right hand extended towards radio, so right hand is not on the steering wheel)*
c6: drinking (*right hand holding the cup*)
c7: reaching behind (*right hand trying to reach the back passenger seat*)
c8: hair and makeup (*hand on face or hair, so hand is not on the steering wheel*)
c9: talking to passenger (*no forward gaze, driver facing the passenger seat and talking*)

The train and test data are split on the drivers, such that one driver can only appear on either train or test set. The drivers' training images are labelled, but the test images are not labelled.

Metrics

I will use the following two functions to measure the performance of my model:

1. **Accuracy:**
   Accuracy is the percentage of predictions on test data which are correct when compared with their target labels.

   I have chosen this metrics because in our case the data is balanced i.e., it has almost equal number of images in each category. In a problem where there is a large class imbalance accuracy metrics may lead to wrong interpretation as a bad model can predict the value of the majority class for all predictions and achieve a high classification accuracy.
   Just as an example if the model just makes a dummy prediction that all samples belong to the bigger class i.e., first one (100 images in first class and 20 images in second class), the accuracy will be 83% (100/120). But that's usually not what we want to predict in an imbalanced dataset.
   But, since this is not the case with us it is okay to use accuracy metrics.

   Predictions for each test image is the maximum likelihood probabilities of the image belonging to each of the 10classes.

   Accuracy = $\frac{100}{N} * \sum_{i=1}^{N} Ci$

   Where N is number of test samples, and Ci =1 if prediction for image equals test labels, and Ci=0 if predictions for image do not equal test labels.

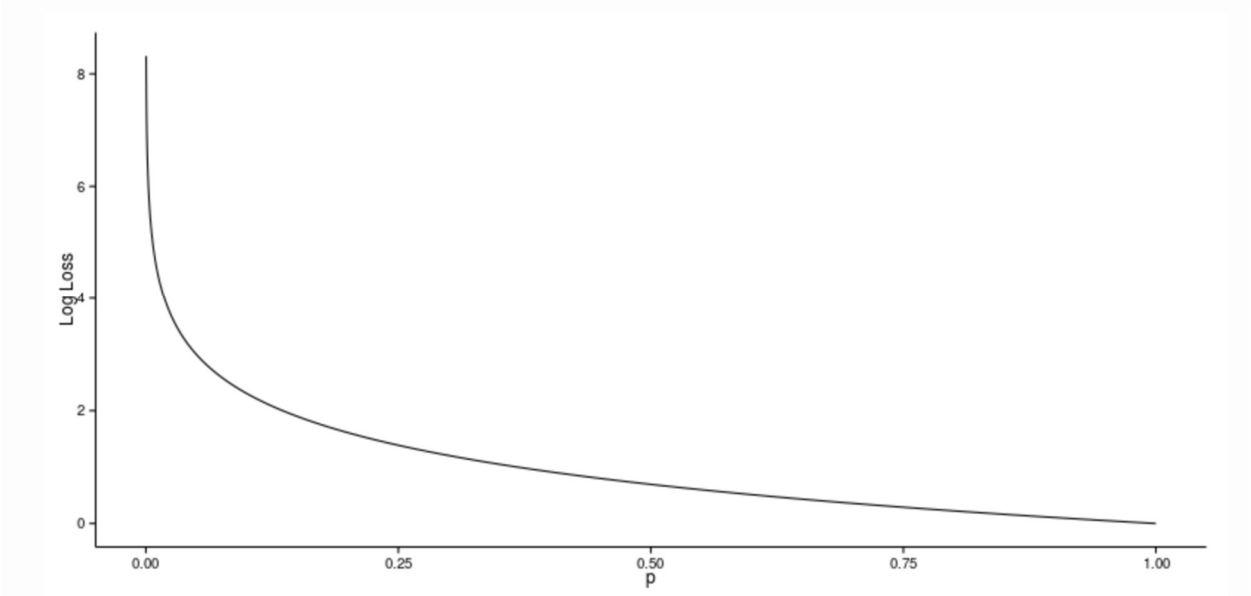
2. **Logrithmic loss (also known as categorical cross entropy)**

Log Loss takes into account the uncertainty of your prediction based on how much it varies from the actual label. Log Loss quantifies the accuracy of a classifier by penalising false classifications. Minimising the Log Loss is basically equivalent to maximising the accuracy of the classifier.

If we look at the plot below showing the Log Loss contribution from a single positive instance where the predicted probability ranges from 0 (the completely wrong prediction) to 1 (the correct prediction). It's apparent from the gentle downward slope towards the right that the Log Loss

gradually declines as the predicted probability improves. Moving in the opposite direction though, the Log Loss ramps up very rapidly as the predicted probability approaches 0.

Log Loss heavily penalises classifiers that are confident about an incorrect classification. If the logloss comes very high then it is an indication that the model is overconfident (giving high probabilities) about the incorrect class for a particular image. It will give a better view of the classifier performance.



Since this project is from Kaggle we have leaderboard to show the score of 1440 contestants who competed this project in 2016. The metric used for this Kaggle competition is multi class Logarithmic loss (also known as categorical cross entropy) which is calculated as per the following formula.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} yij \log(pij)$$

where N is the number of images in the test set, M is the number of image class labels, log is the natural logarithm, yij is 1 if observation I belongs to class j and 0 otherwise, and pij is the predicted probability that observation I belongs to class j.

The submitted probabilities for a given image are not required to sum to one because they are rescaled prior to being scored (each row is divided by the row sum). In order to avoid the extremes of the log function, predicted probabilities are replaced with

$\max(\min(p,1-10^{-15}),10^{-15})$

II. Analysis

Data Exploration

This project is from 2016 Kaggle competition. The final set of data after removing noise and creation of validataion and testing folders are as follows:

File descriptions

imgs.zip - zipped folder of all labelled(train/ valid/test) images

There are 18930 training, 3490 validation images, and 370 test images.
There are 313 unlabelled test images in a separate folder. There are total 10categories to be tested.

- driver_imgs_list.csv - a list of training images, their subject (driver) id, and class id
  There are 26 unique drivers in the training set. This can be seen from the csv file provided driver_imgs_list.csv

As per the attached link of Kaggle website some background in the data collection is as follows.

https://www.kaggle.com/c/state-farm-distracted-driver-detection/discussion/20162

"Here is some background in the experiment. A number of volunteers were asked to perform specific driving actions while being recorded. The drivers were given some basic instructions on the action to take. Each action had a timeframe assigned throughout the video stream. Frames from that video at incremental time-steps were extracted to be used as a classification problem. The process was automated, at the process consisted of tens of thousands of images. It is expected that some frames could contain "misclassified" images as the driver could be in a transition state for the action.

The actions selected for this experiment have (for the most part) distinct features that can be identified, e.g. hand holding phone and driver looking down, or arm reached behind. Some features will have some classification crossover that makes this problem even more difficult.

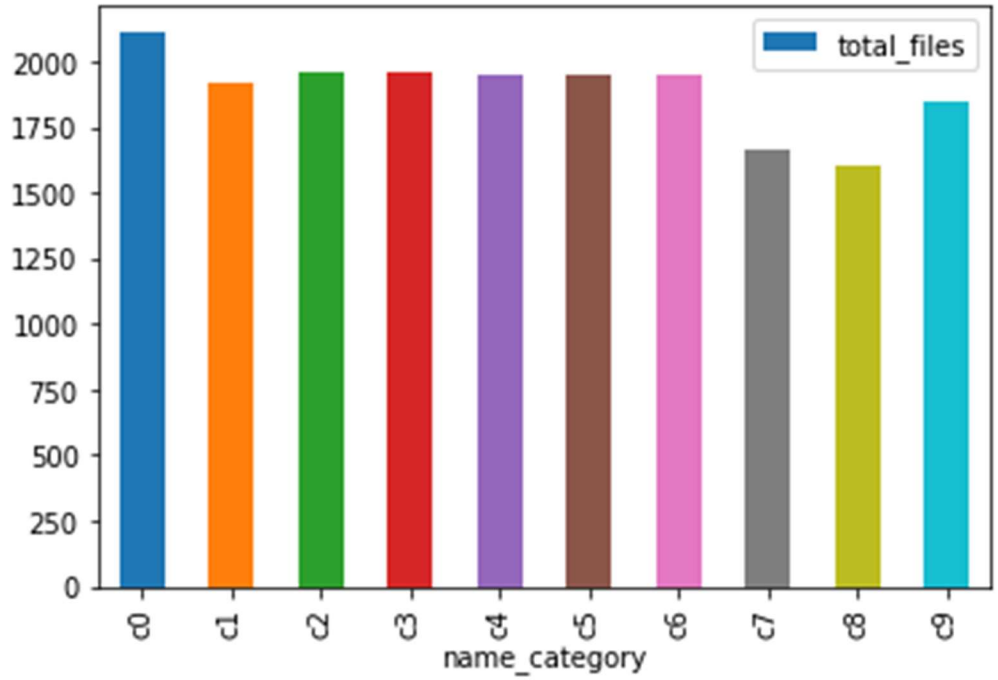
The most difficult action was "normal" driving, because it is so ambiguous. Plus, certain features of the driver could easily be confused with other classes, such as looking in blind spot (reaching behind or talking to passenger), checking rearview mirror (grooming), etc. The guidelines for "normal" driving were not strict other than simply "Do not do any of the other actions.". It was recommended that they keep both hands on the wheel, but whether it be subconscious or on purpose, the volunteers may have bent the guidelines slightly, but not to the point that they explicitly performed other "distracted driving" actions.

The second difficult one was "talking to passenger". Drivers tend to move their head from looking straight ahead to looking 90 degrees right to talk to the passenger. Although the head orientation is a feature that could be used in classifying these images, there are additional features that should be considered as well. For example, mouth movement, hand gestures, facial expressions, etc."

I have discussed the preprocessing of data later.

Exploratory Visualization

Distribution of data in the training directory (as per state-farm.pynb)



As seen from the plot in training folder each category has more than 1600 images. ‘c0’ has the most. Data is almost uniformly distributed among all categories.

‘c7’ and ‘c8’ have the least number of training images and ‘c0’ has the most.

Algorithms and Techniques

I will use transfer learning to save on training time and reach much higher accuracy in shorter amount of training time. I have used pretrained convolutional network VGG16 and finetuned the same. VGG16 is a CNN trained on imagenet database.

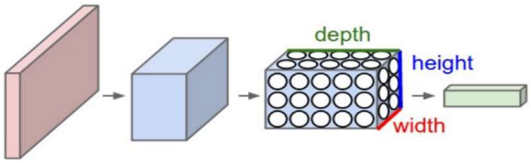
But, before that I first tried to make my own CNN from scratch and thereafter I tried making model by extracting bottleneck features from pretrained network on imagenet dataset.

First some details about the Convolutional Neural Networks are as follows:

As per http://cs231n.github.io/convolutional-networks/ Convolutional Neural Networks are very similar to ordinary Neural Networks i.e., they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer.

The only change is ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network. They take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. The end of the ConvNet

architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:



ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: **Convolutional Layer**, **Pooling Layer**, and **Fully-Connected Layer** (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet **architecture**.

In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image.

Spatial arrangement. Three hyperparameters control the size of the output volume: the **depth, stride** and **zero-padding**.

1. First, the **depth** of the output volume is a hyperparameter: it corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, for my scratch model, the first Convolutional Layer takes as input the raw image(tensor shape (1,224,224,3)), then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. We will refer to a set of neurons that are all looking at the same region of the input as a **depth column** (some people also prefer the term *fibre*).
2. Second, we must specify the **stride** with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. In first two convolution layer of scratch model I have chosen stride of 1 (convolutional layer will roughly be the same height and width as input layer), and thereafter stride of 2 (the height and width of convolutional layer will be reduced by half).
3. Sometimes it will be convenient to pad the input volume with zeros around the border. The size of this **zero-padding** is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes. I have kept padding as same to exactly preserve the spatial size of the input volume so the input and output width and height are the same.

Pooling Layer

(For scratch model)

I have inserted a Max Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. Using pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height. The same number of feature maps are retained, but reduced in width and height.

After the fourth max pooling layer I used 'Global Average Pooling layer' for extreme dimensionality reduction. Now, each feature is reduced to single (average) value. I used the Dropout and Dense combination to adjust for possible overfitting associated with the small training size. Dropout prevents a layer from seeing twice the exact same pattern.

Our dataset is fairly large and similar in some respects to imagenet dataset.

First, I will make the model from scratch. However, since our dataset is not large enough so learning on such dataset alone (scratch model) will not give good accuracy. I can leverage features already learned by some pretrained models on imagenet dataset. Imagenet dataset has human images, but we want to capture humans doing certain specific actions.

The transfer learning should give better accuracy than model from scratch. I will use ResNet50, InceptionResnetV2 just for training the model using the bottleneck features of a pre-trained network . A small fully-connected model on top of the stored features will be used for training. GlobalAveragePooling layer (for dimensionality reduction), and then the fully connected Dense layer with 'softmax' activation function on top of the stored feature will be used for training to give the final prediction and identify among 10 classes in our case.

But since the classes to be predicted are not identical to these pretrained networks, I had to Fine tune top layers of the neural network. I fine tuned only VGG16 for this project based on the following keras blog:https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html. Further details are in implement and Refinement section.

Challenges faced in implementation of this project:

The initial hurdle was how to copy such a big data on to AWS instance. I am working on windows 10 so I tried Filezilla. But, just downloading all the files from Kaggle occupied lot of space on my computer and made it super slow. Thereafter when I tried to upload from my laptop to AWS instance, the filezilla would abort the connection. Finally I realized the best way was to use kaggle's own tool to copy from Kaggle website directly to the AWS.

Thereafter I faced problem with Jupyter Notebook. I couldn't launch Jupyter notebook (problem started after working on it for a month successfully.) This problem sometimes got solved on its own and would reoccur frequently. I couldn't get help from AWS or Udacity forums regarding this. Then I realized that in Udacity lessons the AMI instance that was suggested 'Deep Learning on Ubuntu Linux' was no longer supported by AWS. I created a new instance 'Deep Learning AMI (Ubuntu) Version 5.0 (ami-c27af5ba)' and that solved my problem though I had to redo the whole thing.

Thereafter I had to launch another instance because I ran out of memory on the first one, this time 'Deep Learning AMI (Ubuntu) Version 7.0'. My instance would get terminated whenerver I start extracting bottleneck features. I had to take AWS business support to solve this and finally they figured out that the instance was working on CPU and not GPU. When I activated tensorflow, it would install the most optimized version for CPU. They couldn't figure out the reason for it. But, the solution was to install tensorflow gpu by pip install command.

I used this website of keras for transfer learning.https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html. They have provided help with the code also. I checked and rechecked numerous time but my code was not working. After some research I realized that total number of images should be exactly divisible by batch size chosen. There should not be any remainder images left because then it gives error in calculation of number of batches. I had to add or delete three or four files to come to the perfect number of total images in training and validation folder (since the number of batches are 10, the number of images should be multiple of 10).

Another thing which I realized after some time was not to keep shuffle as True, because otherwise it shuffles targets also and accuracy is reduced significantly.

I got feedback from Udacity after first submission to perform a K-Fold Cross Validation and include a table of the results of the different folds' scores. I need to split the data making sure that the same driver does not come in training and validation data otherwise the model will learn the drivers and not the specific action that we have to capture. This will help in avoiding overfitting and making sure that testing data/ validation data is not similar to training data.

I couldn't find a way to split the data meeting this requirement in a single line of code. Since I could not use simple kfold. I took original big training data having 22420 images and splitting into validation folder by taking just 20% of total driver(5drivers). So, first trial will be with validation set of driver 1to5 and training set of driver 6 to 266; 2nd trial will have driver6 to 10 in validation and remaining in training; and so on. I will save these five trials in five different files. The first jupyter notebook will be name as State_farm1.pynb, and last file will be named as State_farm5.pynb.

This serves the same purpose as kfold, but cumbersome. Also the bottleneck features generated with these files were so big that I could not do the whole thing in one AWS instance. But, it certainly helped in good analysis of my final model.

Benchmark

I will benchmark my model in the following two ways:

1. **Final Fine Tuned VGG16 model against my model from Scratch for accuracy metrics**

The accuracy of all the models made in this project have been mentioned in the table below in the decreasing order of accuracy:

Model Description	**Accuracy as satefarm.pynb**	**Accuracy as satefarmcopy1.pynb**	**Accuracy as satefarmcopy2.pynb**	**Accuracy as satefarmcopy3.pynb**	**Accuracy as satefarmcopy4.pynb**	**Average Accuracy**
Final Fine tuned VGG16 model (split on training and validation folder made on driver id)	Random selection of four drivers for validation	p002', 'p012', 'p014', 'p015 drivers in validation folder	'p022', 'p024', 'p026', p045 drivers in validation	'p039', 'p041', 'p042', 'p047' drivers in validation	p049', 'p050', 'p052, p056 drivers in validation	
	88.65%	**87.84%**	**87.83%**	**86.21%**	**88.6%**	**87.83%**
Model from scratch	54.59%					54.59%
Using the bottleneck features of ResNet50	14.59%					14.59%
Using the bottleneck features of VGG16**(split on training**	11.35%	12.7%	12.7027%	12.7%	13.24%	11.35%

and validation folder made on driver id)						
Using the bottleneck features of InceptionResNetV2	10.27%					10.27%

2. Comparing the log loss or categorical crossentropy loss with Kaggle leaderboard

Since this project is from Kaggle we have leaderboard https://www.kaggle.com/c/state-farm-distracted-driver-detection/leaderboard to show the score of 1440 contestants who competed this project in 2016. I will compare my 'log loss' with the leaderboard results.

In private leader board Rank1 and Rank 1440 has logarithmic loss of 0.08739 and 31.06614 respectively. In public leader board Rank1 and Rank 1440 has logarithmic loss of 0.08689 and 30.74658respectively.

III. Methodology

Data Preprocessing

There is some noise in the data because of the reasons mentioned in 'Data Exploration'. I would move some images in their right folder as it will ensure better learning.

Since there is no validation data provided I will take 20% of the data from training set and save it in validation folder. I will separate the data into validation folder while taking care that the drivers who appear in validation set do not appear in training set. Since the test data is not labelled I will manually move them to the right labelled folder for testing.

After creation of testing and validation folder there are 18,930 training images, 3490validation images, and 370 labelled test images. There are total 10categories to be tested.

I have used 'to_categorical' to transform the target to vectors before passing it on to the models. The target 1 i.e., C0 is changed to [1, 0, 0, 0, 0, 0, 0, 0, 0, 0], C1 is changed [0,1, 0, 0, 0, 0, 0, 0, 0, 0], ……, C10 is changed to [0, 0, 0, 0, 0, 0, 0, 0, 0, 1].

Converting to tensors:

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

(nb_samples,rows,columns,channels),(nb_samples,rows,columns,channels),

where nb_samples corresponds to the total number of images (or samples), and rows, columns, and channels correspond to the number of rows, columns, and channels for each image, respectively. The path_to_tensor function below takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224×224224×224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three

channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shape

(1,224,224,3).(1,224,224,3).

The paths_to_tensor function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

(nb_samples,224,224,3).(nb_samples,224,224,3).

Here, nb_samples is the number of samples, or number of images, in the supplied array of image paths. It is best to think of nb_samples as the number of 3D tensors (where each 3D tensor corresponds to a different image) in dataset!

Data augmentation

Our data is not very small, but still not large enough and overfitting may become a concern in case of smaller data. Overfitting happens when a model exposed to too few examples learns patterns that do not generalize to new data, i.e. when the model starts using irrelevant features for making predictions. Data augmentation is one way to fight overfitting. Augmentation tend to disrupt random correlations occuring in data. Image augmentation artificially creates training images through different ways of processing or combination of multiple processing, such as random rotation, shifts, shear and flips, etc.

Shear mapping or shearing displaces each point in the vertical direction by an amount proportional to its distance from an edge of the image. Shear_range and corresponds to the deviation angle (in radians) between a horizontal line in the original picture and the image (in the mathematical sense) of this line in the transformed image. The transformation zooms the initial image in or out. The zoom_range parameter controls the zooming factor. I have considered shear range and zoom range both as 0.2. I did not consider doing flip or rotation as they may alter the image to an extent that it may fall into another category (example left hand will become right hand).

In Keras this can be done via the keras.preprocessing.image.ImageDataGenerator class. We will use .flow_from_directory() to generate batches of image data (and their labels) directly from our jpgs in their respective folders.

Implementation and Refinement:

Create a CNN to Classify Distracted Drivers (from scratch)

The first convolution layer has 32filters, 2x2 in size. The input shape is 224x224, 3channels. I took padding as 'same' because I did not want to loose any information. Stride of 1(by default) makes the convolutional layer the same height and width as input layer. The number of weights/ parameters for the first CNN layer is (2X2X3)X32+32= 416

I planned to use at least four CNN layers to extract relevant feature maps. The number of filters in successive CNN layers will be increased in sequence of 32, 64, 128, 256. The more the number of filters means more stacks of feature maps. So, dimensionality of convolutional layers will increase drastically. More parameters can lead to overfitting. Hence pooling layers are used to reduce dimensionality. Using 'Pooling layers' will help in keeping the same number of feature map, but each feature map will reduce in width and height. I chose the first two pooiling layer with stride of 1 and third and fourth pooling layers with stride of 2.

For Conv. layers I used 'relu' as activation function, because it returns positive as it is and converts all negatives to zero.

After the fourth max pooling layer I used 'Global Average Pooling layer' for extreme dimensionality reduction. Now, each feature is reduced to single (average) value. I used the Dropout and Dense combination to adjust for possible overfitting associated with the small training size. Dropout prevents a layer from seeing twice the exact same pattern It took almost an hour to finish training but could achieve fairly good accuracy of 54.6% with just 18930 images

With model from scratch I got accuracy of 54.6% and validation loss of 1.6

Prediction by extracting bottleneck features from VGG16, ResNet50, and Inception ResNetV2

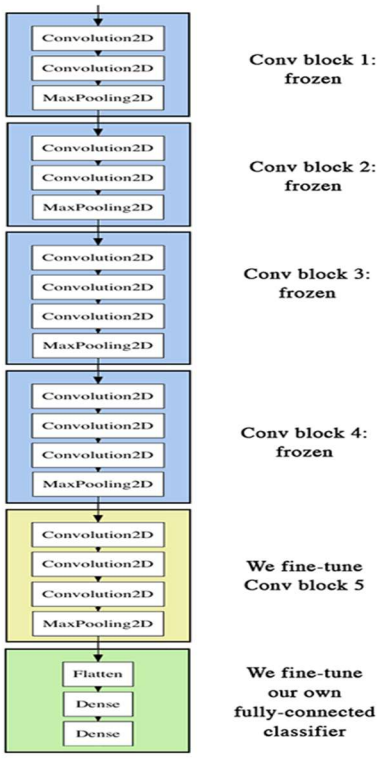
The transfer learning should give better accuracy than model from scratch. I used ResNet50, InceptionResnetV2 just for **training the model using the bottleneck features** of a pre-trained network . A small fully-connected model on top of the stored features will be used for training. GlobalAveragePooling layer (for dimensionality reduction), and then the fully connected Dense layer with 'softmax' activation function on top of the stored feature will be used for training to give the final prediction and identify among 10 classes in our case.

Using this approach I got the accuracy and validation loss much lower than the model from scratch.

Fine Tuning VGG16

But since the classes to be predicted are not identical to these pretrained networks, I had to Fine tune top layers of the neural network. I **fine tuned only VGG16** for this project based on the following keras blog:https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

- To perform fine-tuning, all layers should start with properly trained weights because the large gradient updates triggered by the randomly initialized weights would wreck the learned weights in the convolutional base. I will first train the top-level classifier (by extracting bottleneck features from VGG16), and only then start fine-tuning convolutional weights alongside it.
- We choose to only fine-tune the last convolutional block rather than the entire network in order to prevent overfitting, since the entire network would have a very large entropic capacity and thus a strong tendency to overfit. The features learned by low-level convolutional blocks are more general, less abstract than those found higher-up, so it is sensible to keep the first few blocks fixed (more general features) and only fine-tune the last one (more specialized features).
- Fine-tuning should be done with a very slow learning rate. I will use SGD optimizer for fine tuning. This is to make sure that the magnitude of the updates stays very small, so as not to wreck the previously learned features.

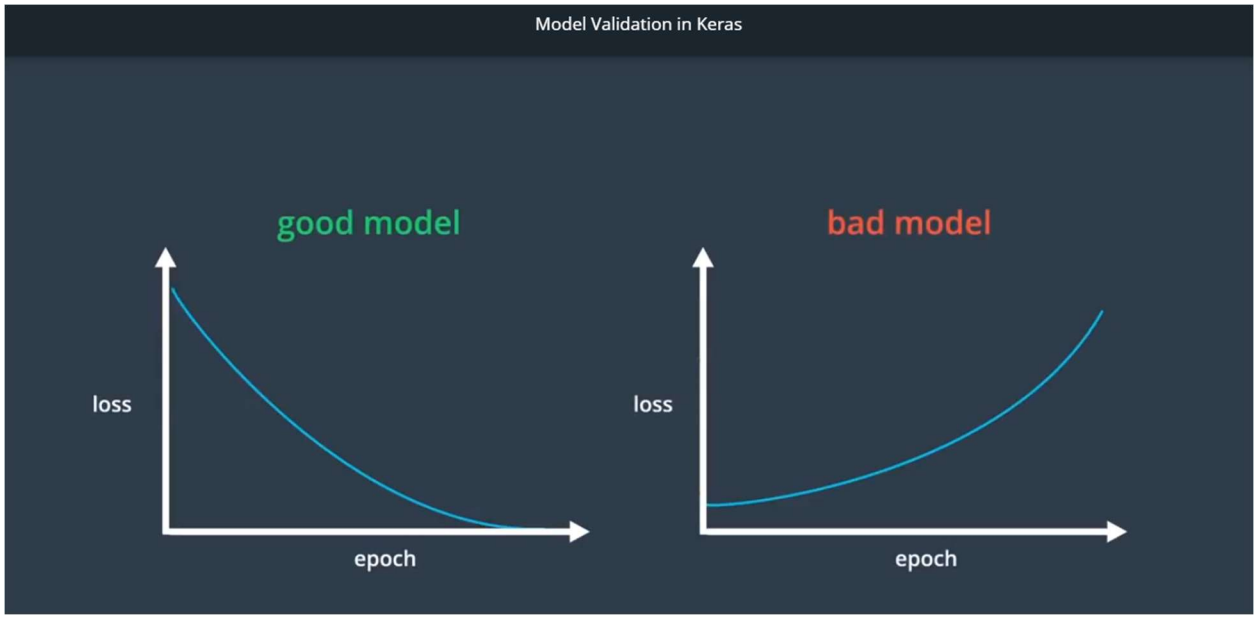


IV. Results

Model Evaluation and Validation

We used almost 20% of the training data for validation, and test data was totally separate and never seen by the model before.

I have used log_loss and accuracy as the metrics. As shown in the graph below a good model is the one in which logloss reduces with every epoch. The training results from the Final fine tuned model is as given below.



Epoch 1/5
1893/1893 [==============================] - 355s 188ms/step - loss: 2.1005 - acc: 0.2136 - val_loss: 1.3234 - val_acc: 0.4579

Epoch 00001: val_loss improved from inf to 1.32336, saving model to saved_models/weights.best.finetuneVGG16.hdf5
Epoch 2/5
1893/1893 [==============================] - 356s 188ms/step - loss: 0.4310 - acc: 0.8606 - val_loss: 0.7058 - val_acc: 0.7398

Epoch 00002: val_loss improved from 1.32336 to 0.70583, saving model to saved_models/weights.best.finetuneVGG16.hdf5
Epoch 3/5
1893/1893 [==============================] - 356s 188ms/step - loss: 0.1516 - acc: 0.9537 - val_loss: 0.6002 - val_acc: 0.8074

Epoch 00003: val_loss improved from 0.70583 to 0.60022, saving model to saved_models/weights.best.finetuneVGG16.hdf5
Epoch 4/5
1893/1893 [==============================] - 356s 188ms/step - loss: 0.0863 - acc: 0.9756 - val_loss: 0.6575 - val_acc: 0.7885

Epoch 00004: val_loss did not improve
Epoch 5/5
1893/1893 [==============================] - 356s 188ms/step - loss: 0.0575 - acc: 0.9838 - val_loss: 0.6531 - val_acc: 0.7865

We could get a very good accuracy in just five epochs. The cross validation loss reduced from 1.3 to 0.6, and on test data we got log_loss of 0.4,, and accuracy of 88.6%.

As per analysis in later sections if we increase the number of training images in c7, c8, and c9 categories we can expect accuracy to improve drastically. Since the training loss is much less than the validation loss the model could be overfitting. The validation accuracy achieved while training is 95.3%, however on test

data set we could get accuracy of 88.6% only. In future I can increase the training set size by more data augmentation to avoid overfitting.

Justification

I benchmarked my model in the following two ways:

1. **Final Fine Tuned VGG16 model against my model from Scratch for accuracy metrics**

The accuracy of all the models made in this project have been mentioned in the table below in the decreasing order of accuracy:

Model Description	**Accuracy as satefarm.pynb**	**Accuracy as satefarmcopy1.pynb**	**Accuracy as satefarmcopy2.pynb**	**Accuracy as satefarmcopy3.pynb**	**Accuracy as satefarmcopy4.pynb**	**Average Accuracy**
Final Fine tuned VGG16 model (split on training and validation folder made on driver id)	Random selection of four drivers for validation	p002', 'p012', 'p014', 'p015 drivers in validation folder	'p022', 'p024', 'p026', p045 drivers in validation	'p039', 'p041', 'p042', 'p047' drivers in validation	p049', 'p050', 'p052, p056 drivers in validation	
	88.65%	**87.84%**	**87.83%**	**86.21%**	**88.6%**	**87.83%**
Model from scratch	54.59%					54.59%
Using the bottleneck features of ResNet50	14.59%					14.59%
Using the bottleneck features of VGG16**(split on training and validation folder made on driver id)**	11.35%	12.7%	12.7027%	12.7%	13.24%	11.35%
Using the bottleneck features of InceptionResNetV2	10.27%					10.27%

The average accuracy of final Finetuned model on VGG16 is 87.83%. The average is from models with different training and validation set and we can see that accuracy has remained almost consistent. The variation of maximum and minimum accuracy value is within 1% of the mean value. So it is a consistent and reliable model.

Extracting bottleneck features in VGG16, ResNet50, and InceptionResNetV2 gave very low accuracy, even lower than my model from scratch, because these networks are trained on imagenet database which had human images, but not the same category of classification as required in our project. Hence, finetuning is essential to extract the higher level features and improve prediction of the right class. Model from scratch still gave a little better accuracy because it learned the features from our project database which is mostly balanced with almost equal number of training images in each category. We have 18923 training images which is medium size data base and not very small.

3. Comparing the log loss or categorical crossentropy loss with Kaggle leaderboard

Model Description	Logloss as per satefarm.pynb	logloss as per satefarmcopy1.pynb	logloss as per satefarmcopy2.pynb	logloss as per satefarmcopy3.pynb	logloss as per satefarmcopy4.pynb	Average Logloss
Final Fine tuned VGG16 model (split on training and validation folder made on driver id)	Random selection of four drivers for validation	p002', 'p012', 'p014', 'p015 drivers in validation folder	'p022', 'p024', 'p026', p045 drivers in validation			
	0.49	**0.43**	**0.43**	**0.52**	**0.45**	**0.464**

Since this project is from Kaggle we have leaderboard to show the score of 1440 contestants who competed this project in 2016. I will compare my 'log loss' with the leaderboard results.

The log_loss achieved is 0.46 If we compare the leaderbord of Kaggle https://www.kaggle.com/c/state-farm-distracted-driver-detection/leaderboard . This places at 316 position in private leaderboard and 310 in public leaderboard out of 1440 teams that participated. This is approximately first 21% of the contestants.

V. Conclusion

Free-Form Visualization

To summarize the performance of a final fine tuned VGG16 model I have plotted the confusion matrix (without normalization, and with normalization) as given below. This helps in figuring out the following.

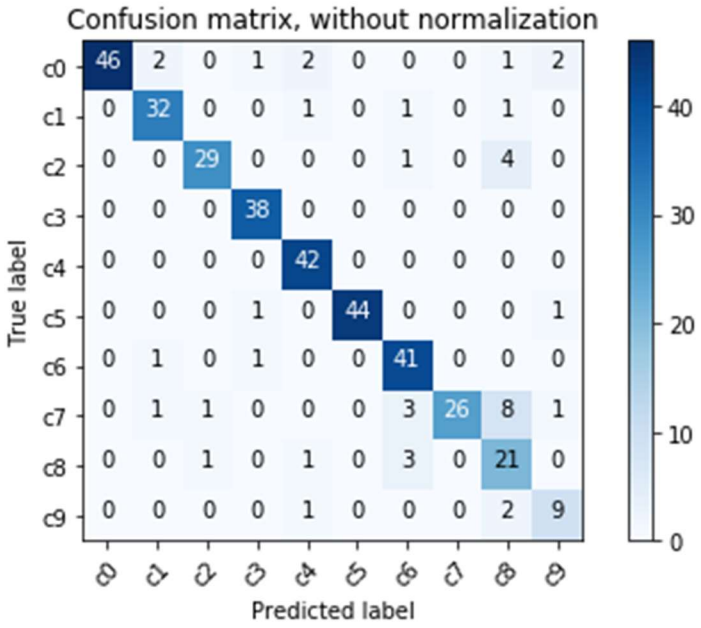
When our data has more than 2 classes. With 3 or more classes we may get a classification accuracy of 80%, but we don't know if that is because all classes are being predicted equally well or whether one or two classes are being neglected by the model. Also when our data does not have an even number of classes. We may achieve accuracy of 90% or more, but this is not a good score if 90 records for every 100 belong to one class and we can achieve this score by always predicting the most common class value.

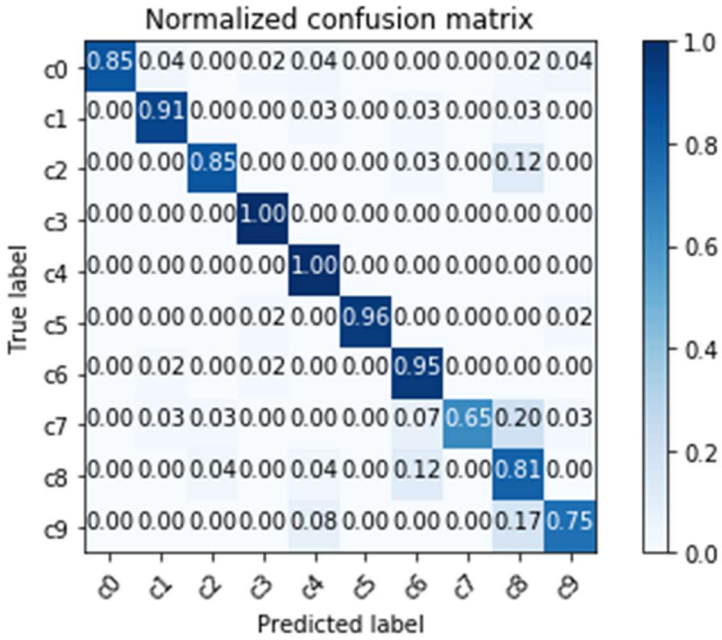
In our case c9 has just 11 images. Other than c9 and c8 all other classes are well represented by having almost equal number of images. If we look at normalized confusion matrix we can see that c7 has the least accuracy. C3, c4, c5 , and c6 have almost perfect predictions.

Overall accuracy = (sum of all True positives along the diagonal) / (Total test images)

$$Accuracy = \frac{46 + 32 + 29 + 38 + 42 + 44 + 41 + 26 + 21 + 9}{370}$$

$$Acccuracy = 88.6\%$$





As seen from the graph ‘Distribution of data in training directory’ above we notice that C7, C8 and C9 have the least number of training images as compared to other categories. Even though the difference is not much (around 250 images each), the effect on the accuracy is quite evident here. 8images out of 40 which were actually c7 have been classified as c8.

If the person is actually driving normally, but he is wrongly classified in either of the categories from c1 to c9 then it is not fair for him. Out of 51 actual images of c0, only 46 are True positives i.e, correctly classified. Rest of the five are classified as distracted driver which is misjudging the driver and penalizing him for something which he did not do.

None of the actual distracted drivers have been misclassified as normal driving (they have been misclassified as other category of distracted drivers).

Reflection

It was very interesting to get 54.58% percent accuracy from model from scratch (I never expected it to be so high for model from scratch). I was expecting that using just the bottle neck features will give better accuracy than model from scratch, but again it was not in line of my expectation and the accuracy was less than 15% in all the tree models(VGG16, ResNet50, and InceptionResNetV2). I expected the accuracy of InceptionResNetV2 to be the most because it is the deepest network, but on the contrary accuracy of ResNet50 was the most.

It was fun to do FineTuning of VGG16 and see the accuracy rise to 88.64% in just five epochs. The advantages of transfer learning were so much evident here. I tried to Fine tune Inception ResNetV2, but since it has more than 400layers, it was difficult to count the layers and figure out how many to freeze. I will do the same as part of further improvement on this project.

Final fine tuned project gave accuracy of 88.68% which is very good accuracy, however the log loss was 0.4. In future I would aim to bring down log loss further. To actually implement this for distracted driver implementation by insurance companies I would like accuracy to be at least 99% because if I am a customer and aim to lower down my insurance premium I would not like insurance companies to penalize me for inaccuracies in their algorithm.

Also, the images need to be seen in totality like we need to analyze the images in sequence. For example, if the driver is trying to see the blind spot then as per current scenario it will be treated as 'talking to passenger' because the gaze is not forward. If the driver is stopped at Stop sign and makes hand motion to other drivers to communicate it will be considered as distracted driving. This can be done by may be nearest neighbor technique which needs to be further explored.

Improvement

I would like to try following improvements in this project:

Model Description	**Accuracy**
Final Fine tuned VGG16 model	88.6486%
Model from scratch	54.5946%
Using the bottleneck features of ResNet50	14.5946%
Using the bottleneck features of VGG16	11.3514%
Using the bottleneck features of InceptionResNetV2	10.2703%

1. **Fine tuning of ResNet50**: Using just the bottleneck features of ResNet50 I could get an accuracy of 14.6% which is better than similar models of VGG16 and InceptionResNetV2. This makes me believe ResNet50 will give better accuracy as compared to VGG16 when finetuned.

2. **Freezing fewer layers**

   In order for the model to learn features better I would like to try freezing fewer layers next time. (instead of freezing first 15 layers, I will try freezing first 11layers in VGG16). It will increase the training time, but it will learn features better.

3. **Supplying more training images for c7(reaching behind) and c8(hair and make up) category**

   As seen from data visulalization plot no. of images in c7 and c8 are lesser than c0(which is most). I should supply with more training images in c7 and c8 to make the training data perfectly balanced. As seen from the prediction results in python notebook, these two categories are causing major errors.

References for this project:

I referred the following websites for understanding of concepts, help with the code, and with the images to include in the project.

- Kaggle discussion forum for state farm distracted driver,
- Udacity lessons
- https://machinelearningmastery.com/image-augmentation-deep-learning-keras/
- https://towardsdatascience.com/some-tricks-learned-from-kaggle-statefarm-competition-8419e032d1f1
- https://arxiv.org/abs/1409.1556 (for VGG16)
- http://stackoverflow.com/questions/36927025/how-to-use-keras-multi-layer-perceptron-for-multi-class-classification
- https://www.kaggle.com/c/state-farm-distracted-driver-detection/discussion/37323
- http://www.codesofinterest.com/2017/08/bottleneck-features-multi-class-classification-keras.html
- https://github.com/keras-team/keras/issues/4040
- http://cs231n.github.io/convolutional-networks/, https://gist.github.com/baraldilorenzo/07d7802847aaad0a35d3
- https://keras.io/preprocessing/image/,
- http://scikit-learn.org/stable/modules/model_evaluation.html,
- machinelearningmastery.com
- https://towardsdatascience.com/image-augmentation-for-deep-learning-histogram-equalization-a71387f609b2
- https://software.intel.com/en-us/articles/hands-on-ai-part-14-image-data-preprocessing-and-augmentation
- https://stats.stackexchange.com/questions/260505/machine-learning-should-i-use-a-categorical-cross-entropy-or-binary-cross-entropy
- https://www.reddit.com/r/MachineLearning/comments/39bo7k/can_softmax_be_used_with_cross_entropy/
- https://keras.io/metrics/
- https://stackoverflow.com/questions/33119169/accuracy-metric-for-multi-class-classifier
- https://cs231n.github.io/transfer-learning/#tf
- https://stackoverflow.com/questions/2084069/create-a-csv-file-with-values-from-a-python-list
- https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html
- http://wiki.fast.ai/index.php/Log_Loss
- http://markhneedham.com/blog/2016/09/14/scikit-learn-first-steps-with-log_loss/
- https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html

- http://pandas-ml.readthedocs.io/en/latest/conf_mat.html
- http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
- https://machinelearningmastery.com/confusion-matrix-machine-learning/
- http://cs231n.github.io/convolutional-networks
- https://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/
- http://www.exegetic.biz/blog/2015/12/making-sense-logarithmic-loss/
- https://datascience.stackexchange.com/questions/28227/why-will-the-accuracy-of-a-highly-unbalanced-dataset-reduce-after-oversampling
- http://www.ritchieng.com/machine-learning-cross-validation/
- https://www.kaggle.com/c/state-farm-distracted-driver-detection/discussion/22614
- http://mortada.net/tips-for-running-tensorflow-with-gpu-support-on-aws.html
- https://faroit.github.io/keras-docs/1.2.2/regularizers/
- http://www.chioka.in/differences-between-l1-and-l2-as-loss-function-and-regularization/
- http://cv-tricks.com/keras/fine-tuning-tensorflow/