Machine Learning Capstone Project Report

Distracted Driver Detection

Project Overview

Distracted driving is a growing public safety hazard. The dramatic rise in texting and constant updates from social media platforms appear to be contributing to an alarming increase in distracted driving fatalities. In this project, we will classify a dataset of images of drivers at the wheel to identify distracted drivers and categorize them based on the type of distraction using CNNs.

Problem

The problem we are trying to solve is a multi-class classification problem. The model built should correctly classify driver's behaviour from an image of the driver at the wheel. The model will be trained using a dataset of images of drivers taken using a dashboard camera. The dataset used, "State Farm Distracted Driver Detection," is provided by State Farm and it is published on the Kaggle website.

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

Each image in the dataset is classified into one of the following ten classes.
c0: safe driving,
1: texting - right,
c2: talking on the phone - right,
c3: texting - left,
c4: talking on the phone - left,
c5: operating the radio,
c6: drinking,
c7: reaching behind,
c8: hair and makeup,
c9: talking to a passenger.

We will predict the likelihood of what the driver is doing in each picture. For each image in the test set, the probability of the image belonging to each of the classes is calculated. Since images in the test set are not labelled we will calculate the performance of the model by submitting the predictions on test data to Kaggle's State Farm Distracted Driver challenge.
https://www.kaggle.com/c/state-farm-distracted-driver-detection/submit

Submissions are evaluated using the multi-class logarithmic loss. Each image has been labelled with one true class. For each image, the submission file includes a set of predicted probabilities, one for every image. The formula is then,

$$F = -\frac{1}{N}\sum_{i}^{N}\sum_{j}^{M} y_{ij} \cdot Ln(p_{ij})) = \sum_{j}^{M}\left(-\frac{1}{N}\sum_{i}^{N} y_{ij} \cdot Ln(p_{ij}))\right) = \sum_{j}^{M} F_i$$

where N is the number of instances, M is the number of different labels, yij is the binary variable with the expected labels and pij is the classification probability output by the classifier for the i-instance and the j-label.

Metrics

I used Multi-class logarithmic loss as the metric to measure the performance of the model. This metric measures the accuracy of a classifier by penalizing false classifications which makes it a good metric for Computer Vision classification problem. Because, to calculate log-loss, the classifier must assign a probability to each class rather than outputting the most likely class. Log loss ensures that we are generalizing all categories well.

Related Work

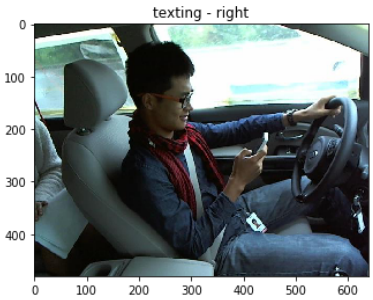
This section summarises some of the related work from literature for distracted driver detection. Since the primary cause of manual distractions is the usage of cell phones, there is a lot of research on cell phone usage detection while driving. Le et al. achieved higher accuracy(94.2%) than state-of-art methods on a dataset for cell phone usage detection by training on a Faster-RCNN based on face and hands segmentation to detect cell phone usage. [1].

Ohn-bar et al. [2] proposed a solution using a dataset with three types of Distractions; adjusting the radio, adjusting mirrors and driving gear. The proposed solution is a fusion of classifiers where the image is segmented into three regions: wheel, gear and instrument panel to infer actual activity. They also presented a region based classification approach to detect the presence of hands in certain pre-defined regions in an image [3]. A model was learned for each region separately and joined using a second-stage classifier. They extended their research to include eye cues to previously existing head and hands cues [4].
Yan et al. [5] designed a model on more comprehensive distracted driving dataset which comprised of video clips covering four driving postures, including normal driving, responding to a cell phone call, eating, and smoking. The proposed solution achieved 99.3% overall accuracy using a CNN model that was first pre-trained by an unsupervised feature learning method called sparse filtering and subsequently fine-tuned with classification.
The earlier datasets are not publicly available , and they concentrate on a limited set of

distractions. StateFarm's distracted driver detection competition on Kaggle defined ten postures to be detected, which include, safe driving and nine distracted behaviours. This dataset was one of the first publicly available datasets that considered a wide variety of distractions.
In 2017, Abouelnaga et al. [6] created a new dataset similar to StateFarm's dataset for distracted driver detection. Authors preprocessed the images by applying skin, face and hand segmentation and proposed the solution using a weighted ensemble of five different Convolutional Neural Networks. The system achieved a good classification accuracy but is computationally complex to be real-time which is utmost important in autonomous driving.

Data Exploration

The dataset consists of images of the driver captured from a side-view dashboard camera. Here are few sample images from the dataset.



All the images have a shape of 640 x 480 x 3.
The train set has 26 unique drivers and a total of 22424 training images that belong to 10 different classes with the following distributions per class.

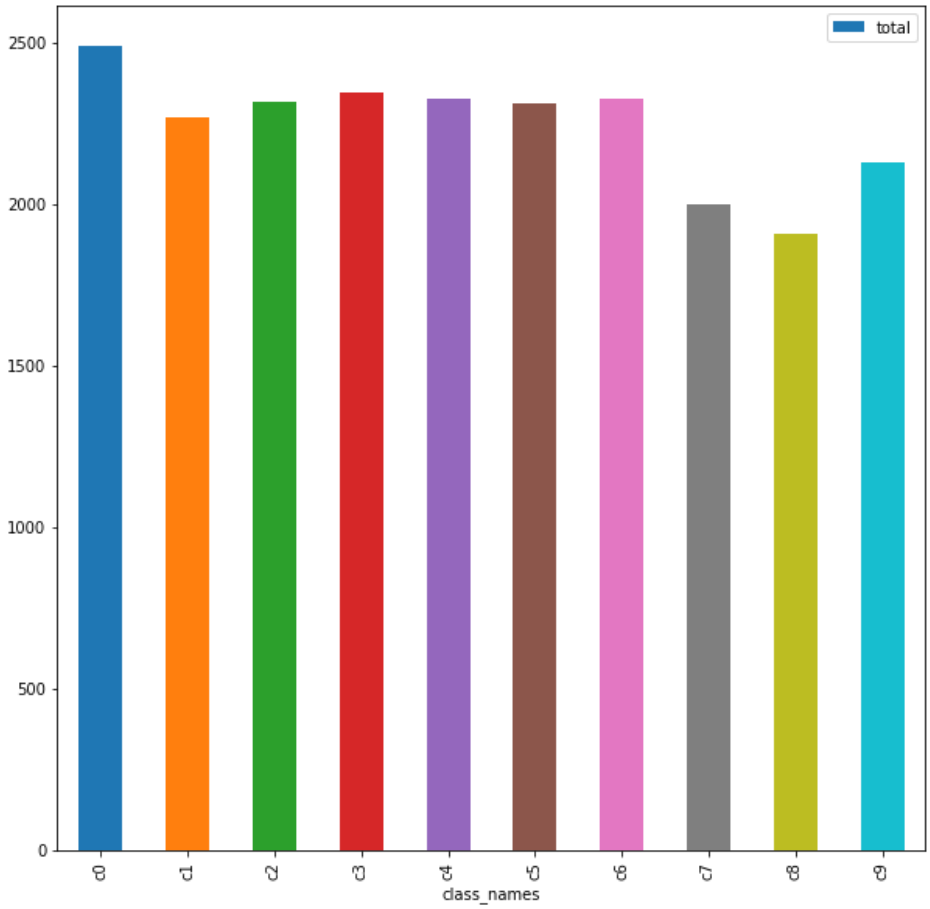


Figure: Distribution of classes in train dataset

Some classes have more images than the others but the difference is not significant.

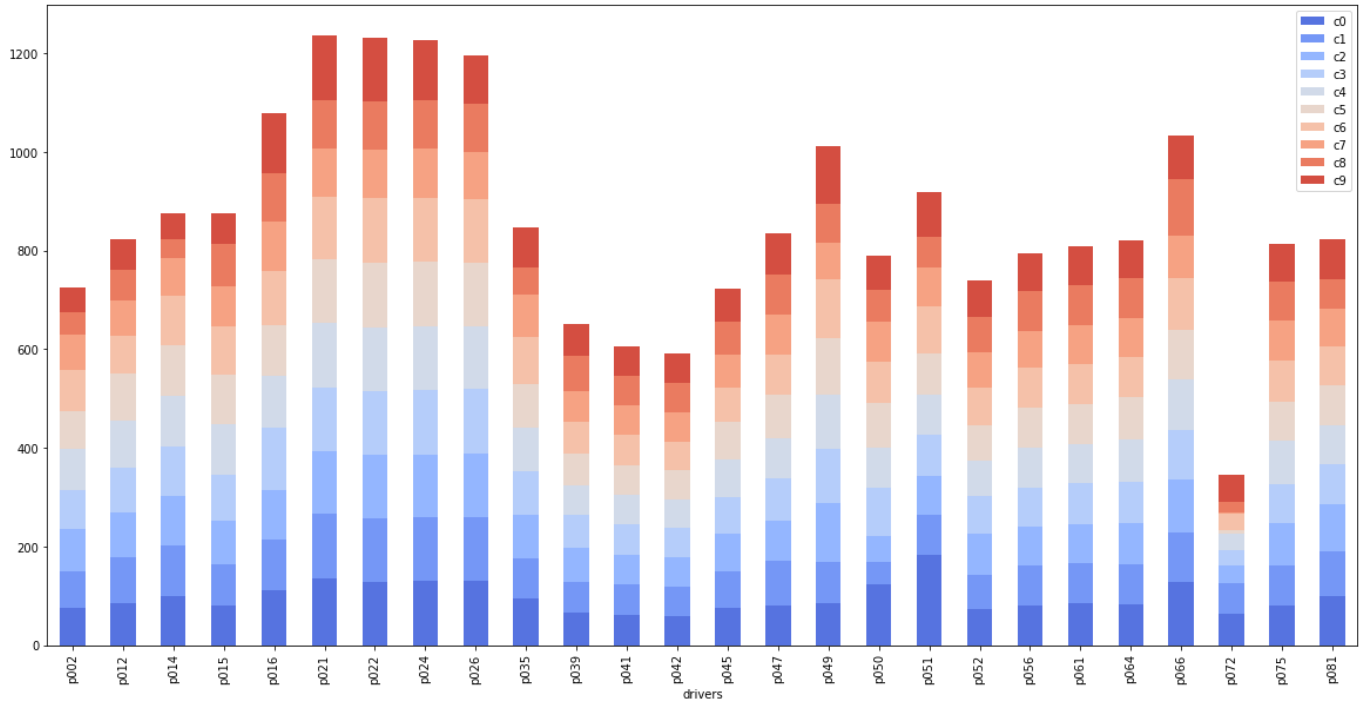


Figure: Distribution of class for drivers in the train dataset

The plot above shows the distribution of classes for each of the 26 drivers in the dataset. All the drivers have images across all the 10 classes. This feature of the dataset will guarantee that the model will generalize well and it will not memorize the driver and use it to identify the action. This is important because the test data has images of drivers that are not present in the train set and we want the model to perform well on unseen data.

The test dataset has 79726 images. These images are not labelled. To calculate the log loss on test dataset, the metric used to the performance of the model for this project, I made submissions on the Kaggle website. Additional details on data exploration that is done for this project is documented in Data_Expolaration.ipynb

Exploratory Visualization

Each driver data is distributed across all the available classes for all the drivers in the train set.

All the images posted here belong to the same class c0: Safe driving behaviour. From these examples, you can see that an image could be classified as safe driving even when the driver is not holding the wheel with both hands.

Also, Images vary in angle, the distance of the camera from the driver, lighting etc. These differences will help the model to generalize better.

One way to look at the features of the images is using edge detectors(countour) and a histogram on the contours of the image

Fig1: c3 texting - left

Fig2: c0 safe driving

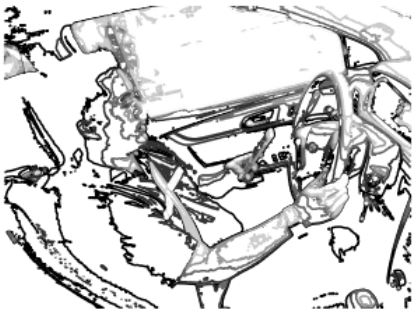


Fig3: Contour for texting

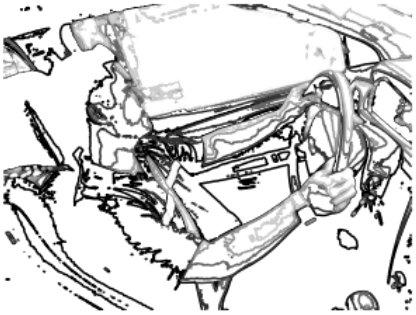


Fig4: Contour for safe driving

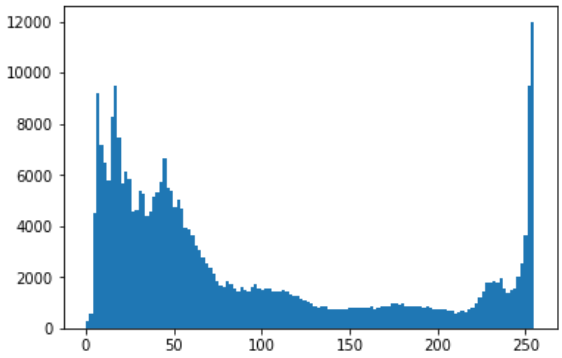


Fig5:Histogram for texting right

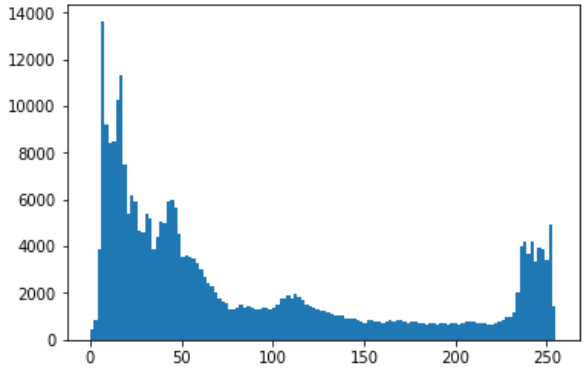


Fig6: Histogram for safe driving

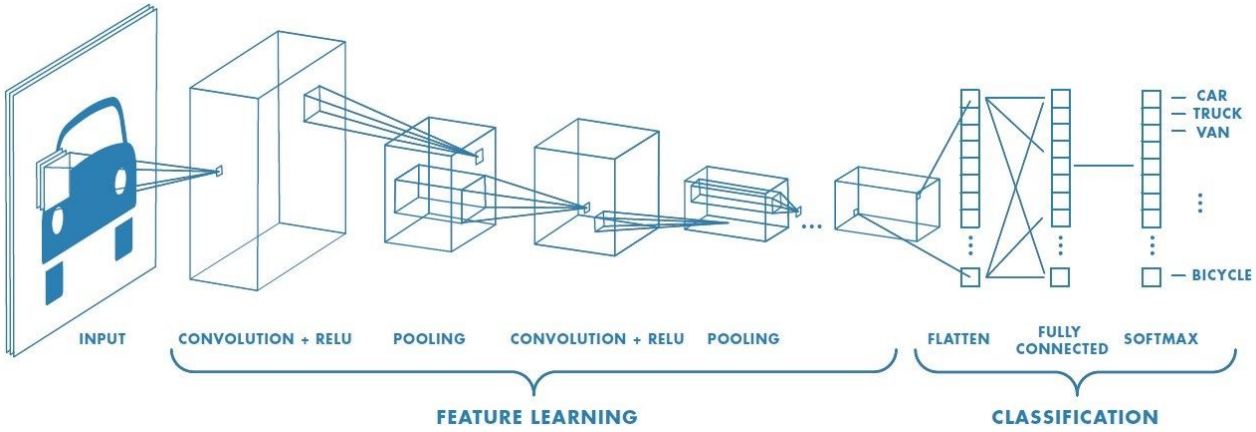
From the above figures you can notice how pixel density variation is different in case of histograms for safe driving and distracted driving of type texting with left hand.

Algorithms and Techniques

I used Convolutional Neural Networks(CNN) to classify the images. Convolutional Neural Networks are a category of Neural Networks that have proven to be very useful in areas such as image recognition and image classification. CNNs are particularly helpful in finding patterns in

images to recognize objects, faces, and scenes. They learn directly from image data, using patterns to classify images eliminating the need for manual feature extraction.

A convolutional neural network can have tens or hundreds of layers. Each of these layers learns to detect various features of an image.



A CNN with many convolutional layers

Like other neural networks, a CNN is composed of an input layer, an output layer, and many hidden layers in between. These layers perform operations that alter the data with the intent of learning features specific to the data.

The most common layers are: convolution, activation or ReLU, pooling and fully connected layers.

Convolutional Layers:

The first layer in a CNN is always a Convolutional Layer and it is responsible for learning features. The input to this layer is an array of pixel values. These inputs are processed by a set of convolutional filters which are also an array of numbers, called features or kernels, each of which activates certain features from the images. The filters start as simple features like brightness and edges and increase in complexity to features that uniquely define the object.

Filters slide or convolve around the input image and multiplying the values in the filter with the original pixel values of the image. These multiplications are all summed up to a single number. After sliding the filter over all the locations, what you're left with an array of numbers, which is called activation map or feature map.

In practice, a CNN learns the values of these filters on its own during the training process although we still need to specify parameters such as number of filters, filter size, architecture of the networketc. before the training process. The more number of filters we have, the more

image features get extracted and the better our network becomes at recognizing patterns in unseen images.

An additional operation called activation is used after every Convolutional layer which activates the features that are carried forward into the next layer for faster and more effective training. Rectified linear unit (ReLU) is most commonly used for CNNS. ReLU maps negative values to zero and maintains positive values.

Pooling Layers

After the convolutional layer, it is a common practice to pass these values into the pooling layer. Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information. Pooling can be of different types: Max, Average, Sum etc.

In case of Max Pooling, we define a spatial neighborhood (filter) and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) or sum of all elements in that window. In practice, Max Pooling has been shown to work better.

These operations are repeated over tens or hundreds of layers, with each layer learning to identify different features. After learning features in many layers, the architecture of a CNN shifts to classification.

Fully Connected Layers

A Fully Connected layer looks at the output of the previous layer , which are activation maps of high level features and determines what high level features most strongly correlate to a particular class.

Finally we have an activation function such as softmax or sigmoid to classify the outputs as a vector of K dimensions where K is the number of classes that the network will be able to predict. This vector contains the probabilities for each class of any image being classified.

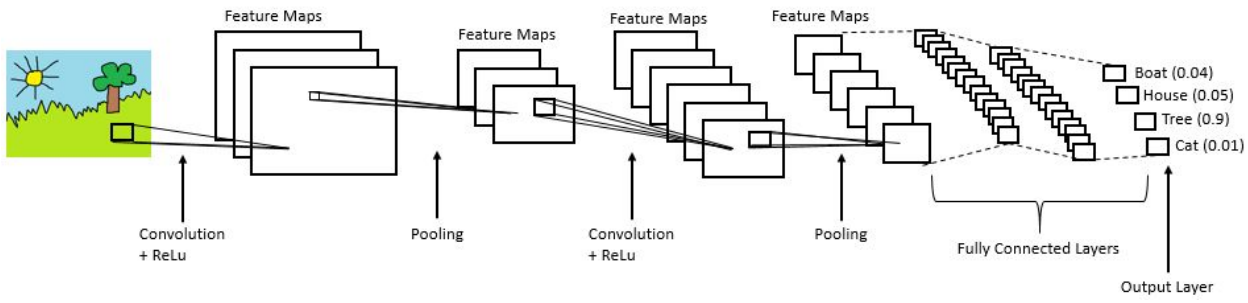
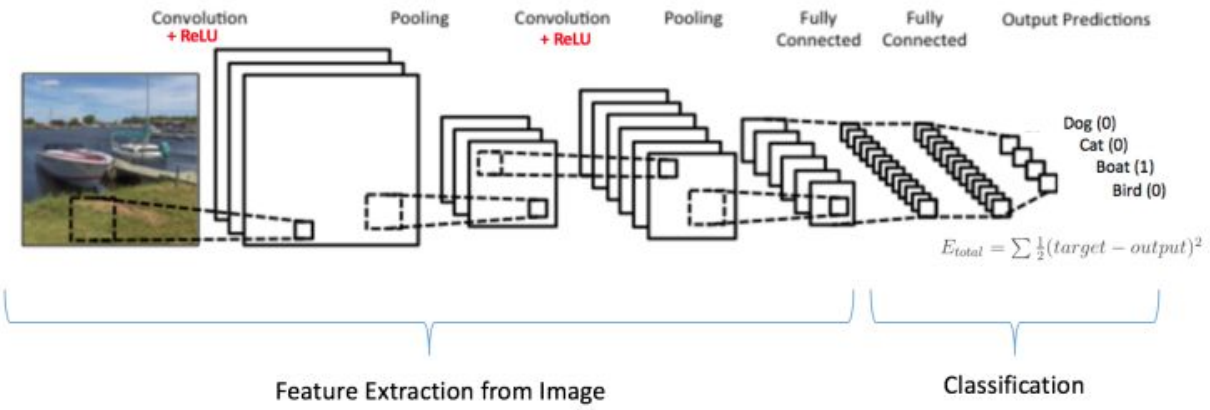


Fig: CNN Architecture

Putting it all together – Training using Backpropagation

As discussed above, the Convolution + Pooling layers act as Feature Extractors from the input image while Fully Connected layer acts as a classifier.



In the Figure above, since the input image is a boat, the target probability is 1 for Boat class and 0 for other three classes, i.e.
Input Image = Boat
Target Vector = [0, 0, 1, 0]

The overall training process of the Convolution Network may be summarized as below:

1. We initialize all filters and parameters / weights with random values.
2. The network takes a training image as input, goes through the forward propagation step (convolution, ReLU and pooling operations along with forward propagation in the Fully Connected layer) and finds the output probabilities for each class.
3. Calculate the total error at the output layer
4. Use Backpropagation to calculate the gradients of the error with respect to all weights in the network and use gradient descent to update all filter values / weights and parameter values to minimize the output error.
5. Repeat steps 2 to 4 with all images in the training set.

The above steps train the Convolutional Network which means that all the weights and parameters of the ConvNet have now been optimized to correctly classify images from the training set. When a new (unseen) image is input into the ConvNet, the network will output a probability for each class using the optimized weights.

The following techniques were used to optimize the model:

1. Transfer Learning: I applied transfer learning using the pertained models available in Keras Application model, to reduce training time and still produce good results. The architectures of these models has been carefully selected and the pre-trained weights for Imagenet are obtained by training on million images. Making use of these trained weights and architectures has drastically reduced the training time.

2. Preprocessing of images using ImageDataGenerator.
3. Early stopping to reduce the training time when validation loss is not improving
4. Model checkpoints to save the best model while training, which was useful in restoring the model, either to optimize or evaluate the model.

I used VGG16 as the benchmark model. The VGGNet has a simple layout of basic conv nets: a series of convolutional, max-pooling, and activation layers before some fully-connected classification layers at the end.

To improve on the benchmark model, I trained the dataset on VGG19, ResNet50, InceptionV3, InceptionResNetV2 networks. I expected ResNet50, InceptionV3 model to perform better than the benchmark model because of their better architecture and higher performance on imageNet. But they did not work well for this dataset. I tried VGG19 which is an improvement on VGG16 and got good results. I fine-tuned the VGG19 model and got an improved score of 1.20704 after training on 50 epochs.
I used Python and Keras framework to implement this project and trained it on an AWS EC2 instance.

Benchmark

I used VGG16's bottleneck features as the benchmark model.
This involved initializing a model with pre-trained weights from VGG16 adding a Dense layer with 10 outputs one for each class, training the model for few epochs and using it as a feature extractor.

The test images are not labelled. So I created a submission file with the predictions from the test set and got a score of 1.3770. Submissions file is included in the Submission folder on the Github project.

Name	Submitted	Wait time	Execution time	Score
VGG16_Benchmark_Model.csv	a few seconds ago	1 seconds	2 seconds	1.37710

The result is calculated using multi-class logarithmic loss, log loss value of the predicted labels against the actual labels of the 79726 test images.

Methodology

Data Preprocessing

The following preprocessing steps were applied to the data:

- Resize images to 100 x 100 to make it easier to load into memory and train in a reasonable amount of time.
- Shuffle images to change the default order.
- Normalize pixel values of the image by diving all the values in the image array by 255 which is the maximum RGB value and reduces our data to be within the range of 0 and 1

- Data augmentation to improve generalization like Zoom, height shift, width shift and rotation.

Implementation

I followed the following steps to implement the model.

1. Data loading and visualization: Load the data and get familiar with the data and distribution amongst categories using visualizations.
All the data exploration steps are documented in IPython notebook, “Data_Expolaration.ipynb”
2. Create a benchmark model using pre-trained VGG16 by adding a fully-connected softmax layer with ten outputs for classification.
The steps to train and test this model are documented in “VGGNet.ipynb” I saved the model in h5 format to load and compare results with other models.
3. Train models using fine-tuned InceptionV3, VGG19, ResNet50 and InceptionResNetV2 models pre-trained on Imagenet data, compare it with the benchmark model and choose the model that performs best on the training set as the feature encoder.
5. Enhance the results of the model chosen using hyperparameter tuning and data augmentation.
Final Model is documented in “Final Model - VGG19.ipynb ”

For all the models training was a time-consuming process and I had to make tweaks to batch size and image shape to optimize memory usage and time to train.

Refinement

This section will discuss following three topics
- Improving upon the algorithms and performance
- Learning Rate and Hyperparameter Manipulations
- Challenges

Improving algorithms and performance

I started off with VGG16 as the benchmark model. I trained and tested the model and got a log loss score of 1.37710 when submitted to the Kaggle submission page. I expected InceptionV3 and ResNetV2 to perform better than VGG16. But Inception's log loss of 2.23553 and ResNet50’s score of 2.78177 on test data. Inception models and ResNetV2 are deeper architecture, and for this dataset, these modes are overfitting and not generalizing well even after tweaking the hyperparameters and adding dropout layers.

Name	Submitted	Wait time	Execution time	Score
Inception_submission (1).csv	just now	0 seconds	2 seconds	2.23553

Name	Submitted	Wait time	Execution time	Score
Inception_submission (1).csv	just now	0 seconds	2 seconds	2.23553

I trained on multiple models multiple models ResNet50, InceptionV3, InceptionResNetV2 and VGG19.

VGG19 performed the best of all, so I fine-tuned the VGG19 model to improve the performance of the model. To further refine the model, I had to tweak the optimizers used, learning rate, momentum and batch size. In addition to hyperparameter tuning, another refinement made was to use Early stopping technique. This technique reduced the time it took to train when there are no improvements in validation loss.

I started off with VGG16 as the benchmark model. I trained and tested the model and got a log loss score of 1.37710 when submitted to the Kaggle submission page.

The final model is an improved VGG19 model for this dataset with a log loss score of 1.20704 on test data.

Name	Submitted	Wait time	Execution time	Score
VGG19_submission.csv	just now	0 seconds	2 seconds	1.20704

Learning Rate and Hyperparameter Manipulations

When I started off building the VGG16 model, I set a learning rate of 0.0001 and an Adam optimizer with a decay rate of 1e-5. I set early stopping to monitor the validation loss and to stop with a patience of 10.

- I then went on to build three more models, InceptionV3, Resnet50 and VGG19.
- For InceptionV3, I set a learning rate of 0.0001 with an Adam optimizer and no decay. Even then, after 50 epochs, reduced right until the last epoch, which meant that the learning rate was too low. I increased it to 0.001, and it finally converged
- For ResNet50, I used a learning rate of 0.002 instead of 0.001. I increased it ever so slightly because although it was learning, I saw scope that it could learn better with a little more learning rate, because it was learning very slowly. Increasing the rate to 0.01 made it stop learning with a higher loss.
- For VGG19, I set a learning rate of 0.01 with an SGD optimizer and decay of 1e-5. This did something completely opposite to the InceptionV3 model. The learning rate started to go up and down erratically by the 10th epoch. This looked like a sign of a too large a learning rate, I changed it to 0.001 and it finally converged.

Challenges

- The first challenge I faced was the amount of compute I had, I only had one GPU access and each image was 480 x 640. Which was huge for a machine with just one GPU. I had to reshape the images to 100 x 100 x 3 just to get it to run within 15 minutes for each epoch.
- The second challenge was the use of a Validation generator. This was both a boon and a bane. For example, the generator would generate completely new set of images unless shuffle was set to False which messed up the results when measuring log loss of the model while evaluating the model. This one problem alone took me a couple of hours to figure out.
- Another problem I encountered was that Amazon EC2 (p2.xlarge) instances randomly lose connection while running models using jupyter notebooks. This created a lot of issues because I had to continuously monitor the training process and restart my model training everytime the connection was lost, and with each other them running for 8 hours, this was a considerable challenge. I solved this issue by creating a daemon process that ran in the background using nohup (no hangup) command in the ubuntu terminal.

Final Model Parameters and Validation results

Parameters:

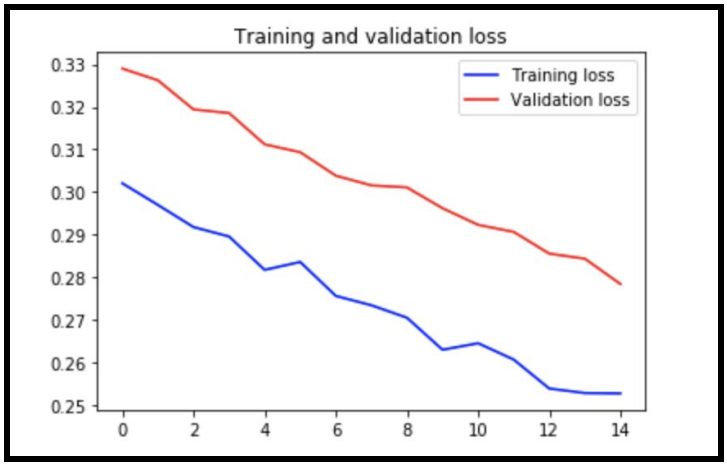
Model	Learning Rate	Optimizer	Input Shape
VGG19	0.001	SGD with decay of 1e-5	(100,100,3)
VGG16	0.001	Adam with decay of 1e-5	(100,100,3)
InceptionV3	0.001	Adam with decay of 1e-5	(100,100,3)
ResNet50	0.002	SGD with decay of 1e-5	(224,224,3)

Results:

Model	Val Loss	Test score
VGG16	0.3571	1.37

VGG19 (Final Model)	0.2784	1.20
InceptionV3	1.41	2.23
ResNet50	2.78	2.78

Final Model Loss graph, VGG19



Test Results and Discussion

The final solution for this project is a Convolutional Neural Network-based system for distracted driver detection. The pre-trained ImageNet model is used for weight initialization and the concept of transfer learning is applied. Weights of all the layers of the network are updated with respect to the dataset. After rigorous experimentation, all the hyperparameters are finet-tuned. The training is carried out using Stochastic Gradient Descent with a learning rate of 0.002, the decay rate of 1e-5. The batch size and number of epochs are set to 32 and 50 respectively. Training and testing are carried out using GPU instance on AWS(p2.xlarge with Ubuntu 16.04+k80 GPU). The framework used to build the project is Keras.

On VGG-16 model, the Benchmark model a log loss score of 1.37710 on the validation set and log loss score of 1.37710 for the Kaggle submission on the test set.

Performance of the system has significantly improved with the fine-tuned VGG19 model which resulted in log loss score of 1.20704 on the test set.

Name	Submitted	Wait time	Execution time	Score
VGG19_submission.csv	just now	0 seconds	2 seconds	1.20704

Conclusion and Future Work

Driver distraction is a serious problem leading to a large number of road crashes worldwide. Hence detection of the distracted driver becomes an essential system component until we achieve level 5, complete automation, with self-driving cars. Here, I built a robust Convolutional Neural Network based system to detect distracted driver and also identify the cause of distraction. We modified the VGG19 architecture for this particular task which has several regularization techniques to prevent overfitting to the training data. With the log loss score of 1.20704, this proposed system outperforms the benchmark approach of distracted driver detection by a significant amount.
As an extension of this work, we can create an ensemble model to improve the accuracy further. We can also incorporate a temporal context that may help in reducing misclassification errors and thereby increasing accuracy.

References:

1. Multiple Scale Faster-RCNN (MSFRCNN) approach
http://openaccess.thecvf.com/content_cvpr_2016_workshops/w3/papers/Le_Multiple_Scale_Faster-RCNN_CVPR_2016_paper.pdf
2. Driver hand activity analysis in naturalistic driving studies: challenges, algorithms, and experimental studies
http://cvrr.ucsd.edu/publications/2013/hand_JEI13.pdf
3. Head, Eye, and Hand Patterns for Driver Activity Recognition
http://cvrr.ucsd.edu/publications/2014/headhandeye.pdf
4. In-vehicle hand activity recognition using integration of regions
https://ieeexplore.ieee.org/document/6629602
5. Driving posture recognition by convolutional neural networks. Available
https://www.researchgate.net/publication/283433254_Driving_posture_recognition_by_convolutional_neural_networks
6. Real-time Distracted Driver Posture Classification
https://arxiv.org/abs/1706.09498
7. https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/