# Capstone Project

Abdelrahman Saud

Machine Leaning Engineer Nanodegree

September 6, 2016

# 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.

Now that these risks are mentioned it’s important to note that the images were taken in a controlled environment, a truck dragging the car around the streets, so there was no risk as the drivers weren’t actually driving.

This project based on [this Kaggle competition]

## Problem Statement

The goal is to create a classifier that can classify driver behavior, therefor hopefully reducing the number of accidents caused by distraction. The driver behavior is categorized into 10 categorizes as described in this section: **Classes:**

The training and data sets can be downloaded from here. The data is then loaded and transformed using keras ImageDataGenerator, (*check **Preprocessing Steps** section*) which is used for loading training/validation/testing images, then a convolutional network is used for training (*check **CNN Architecture** section*) using convolutional layers, max and average pooling, fully connected layer for classification (*check **Training** section*). The data is split into train and validation by driver; this is to ensure the generalizability of the model.

## Metrics

As multi-class logarithmic loss is common for multi-class classifiers [1] it measures the negative log-likelihood of the true labels given the probabilistic prediction of the classifier. Logloss heavily penalizes classifiers that are confident about an incorrect classification, making it better for the model to be somewhat wrong, than wrong without doubt. Therefor the goal is to minimize the log loss, which is calculated as:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij}),$$

where N is the number of images in the test set, M is the number of image class labels, loglog is the natural logarithm, yijyij is 1 if observation ii belongs to class jj and 0 otherwise, and pijpij is the predicted probability that observation ii belongs to class jj.

A benchmark of 0.3588 log loss was already defined on the competition forums as the human performance on this dataset.

As the human accuracy was 94% over 10 classes, its log loss can be calculated as:

-(0.06*log(0.06/9) + 0.94*log(0.94)) ~ 0.3588

# Analysis

## Data Exploration

The drivers' data set consists of 22424 training images, and 79726 testing images.

An important feature of the dataset is that drivers in the training set don't appear in testing set.

The images are 640x480x3 RGB format (width, height, channels RGB) one dimension per color channel R, G, B. Each training image is labeled as one of the below classes, and present as one-hot encoding.

e.g. class c2 – talking on the phone – right will be encoded as

[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]

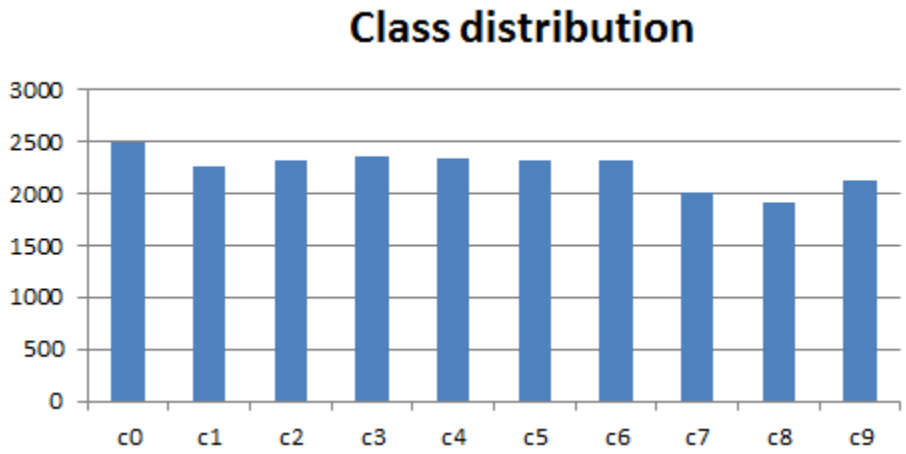
*Note: Right and left, refer to the hand performing the action, e.g. texting – right means the driver was using his right hand for texting.*

### Classes:

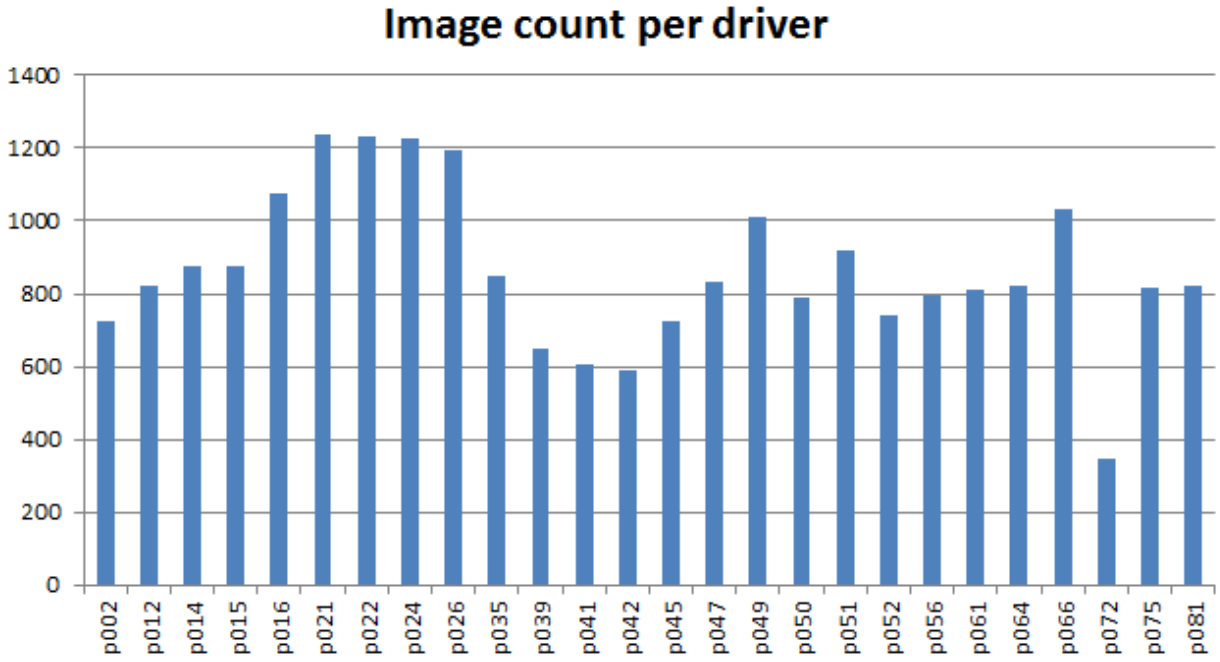
c0: normal driving
c1: texting - right
c2: talking on the phone - right
c3: texting - left
c4: talking on the phone - left
c5: operating the radio
c6: drinking
c7: reaching behind
c8: hair and makeup
c9: talking to passenger

The data can be downloaded here.

The classes distribution is close to uniform as illustrated by the below figure




Most of the drivers (27 drivers in total) have more than 600 images except only 3 as illustrated below




By looking at the training images, it seems that a lot of the images were taken in consecutive shots within few seconds, which results in very similar images, as shown by the below images.

This fact and the distribution in the above figure affected the decision made as explained in Improvements section.

## CNN Architecture

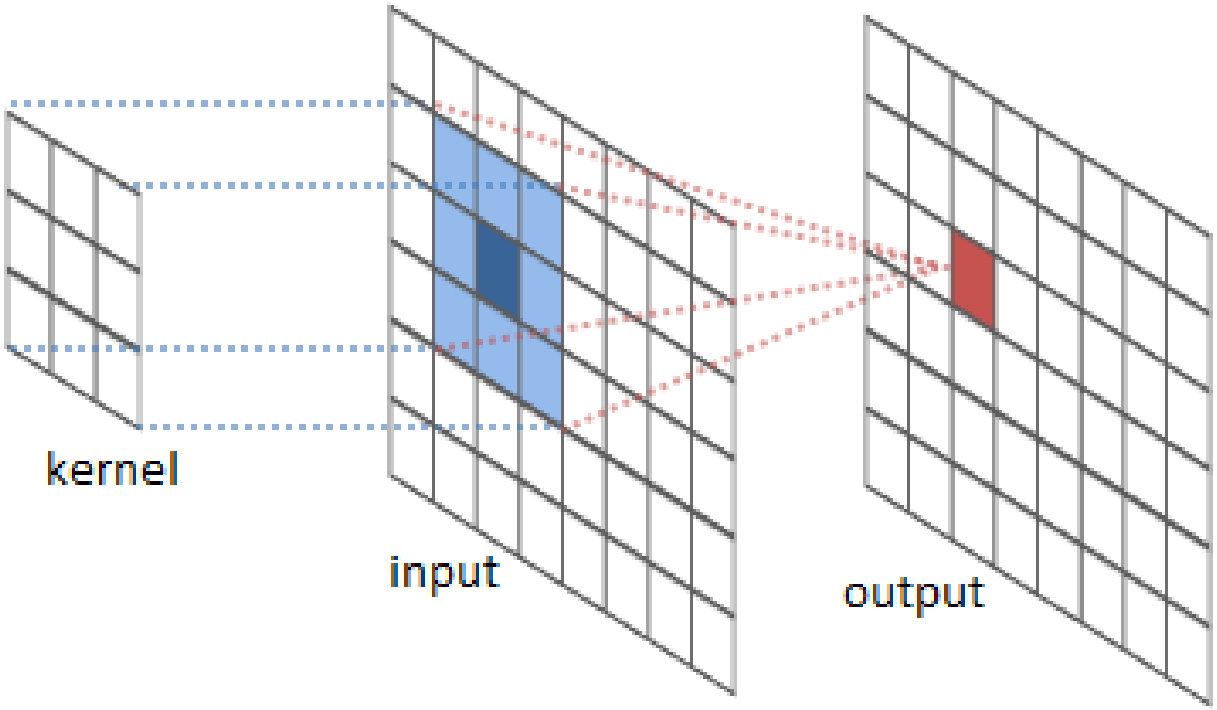
CNNs are proven to outperform other image recognition algorithms, so it was the natural choice for this image classification problem.

A typical CNN architecture consists of a mixture of convolutional (CNV) and pooling (PL) layers followed by fully connected layers (FCN)

The input to the CNN is usually a 2D image in color space RGB or YUV.

While CNV consist of multiple layers of small neuron collections which process portions of the input image, called receptive fields/filters/kernels. The outputs of these collections are then tiled so that their input regions overlap, to obtain a better representation of the original image; this is repeated for every such layer, and each of them is 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. [2]

The below figure illustrates a 3x3 kernel applied to part of the input, and the dot product output.




The actual learning happens through backpropagation algorithm, and gradient descent or one of its improvements.

Activation layers usually follow each CNV layer applying element wise activation function such as ReLU (rectified linear units) max(0, x), this leaves the volume of the input unchanged.

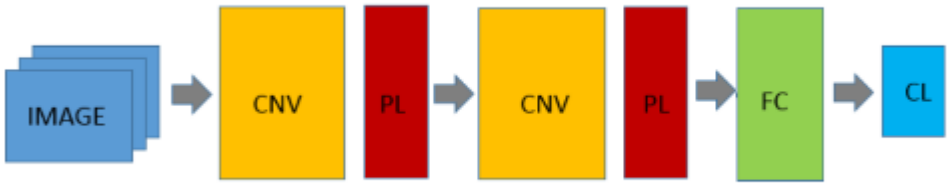
POOL layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume reduction typically by ½

FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers corresponds to a class score. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

Classification layer: The final layer of the network that converts the output of FC to probability of each driver being a certain class. In classification problems like this it's typically a soft-max algorithm.

## Simple initial CNN Architecture

I used a simple CNN architecture of 2 CNV layers followed by 2 PL max layer, then followed by FCN, and finally a soft-max classification layer.




The input image is resized to 64x64, the first CNV layer consisted of 16 3x3 filters followed by ReLU activation, then 2x2 max pooling layer, the second CNV layer consisted of 32 3x3 filters followed by ReLU activation, then 2x2 max pooling layer the output of it is then flattened into 1D to be received by FCN of 128 neurons followed by 10 neurons which represent the number of classes, and finally a classification layer with a soft-max algorithm.

## Training

In the training phase I've used Adam algorithm [3] (*adaptive moment estimation*), and training is done offline through back-propagation. The CNN is build using Keras [4] over Theano [5].

Gradient descent algorithms are very popular in solving quadratic equations using dynamic programing, the algorithm has many variants. Adam is designed to combine the advantages of AdaGrad, which works well with sparse gradients, and RMSProp, which works well on-line and non-stationary settings [3].

One of the most important parameters of gradient descent is the learning rate. While a high learning rate helps gradients to converge faster but with decent quality of the weights, a low learning rate is slower to converge but achieves better weights.

I have used drop-out of connections randomly to increase the randomization of each stream from one layer to another, the data going from one layer to another is being

dropped out with certain probability provided, a probability of 50% drop out rate is used.

## Preprocessing Steps

1- Normalize the images to 0-1 by multiplying by 1/255, which is done for faster learning.
2- Image resized to 64x64 for computing power limitations, switched later to 224x224 but cross validation is stopped at that point.
3- I have used random rotation of the image, and random vertical, horizontal flipping to increase the size and variability of the dataset; this was later removed when larger model was used because of computing power limitations.
4- I have used one-hot encoding for the categories, e.g. label 3 is converted to a vector [ 0 0 0 1 0 0 0 0 0 0 ] (labels start from 0 to 9), this is required as the output of the network has 10 neurons, one for each class

## Improvements

During training several enhancements were done to the architecture to enhance its performance.

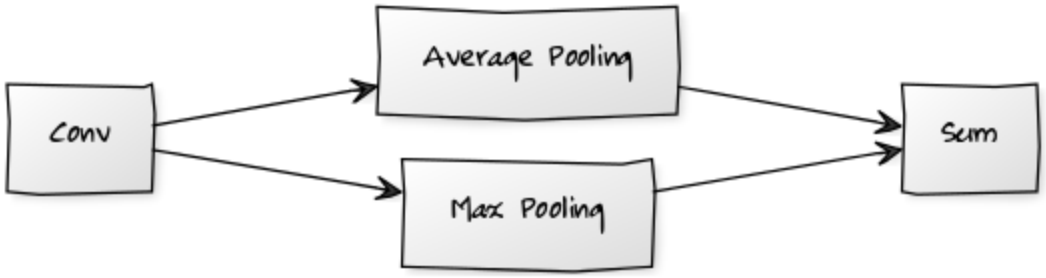
The initial architecture described above achieved logloss of 2.35 on grey scale images, by implemented early stopping method which stops the training once the validation logloss stops decreasing the logloss was improved to 2.03.

As the current architecture suffered under-fitting, I modified it to be the following

- Conv 32-3x3, ReLU, MaxPooling 2x2, dropout 0.3
- Conv 64-3x3, ReLU, MaxPooling 2x2, dropout 0.3
- Conv 128-3x3, ReLU, MaxPooling 8x8, dropout 0.3
- Fully connected 10 (number of classes), softmax activation

Using the mentioned architecture improved the loss to 0.68 and adding the three channels instead of grey scale images, improved loss to 0.65 (it seems since all objects are similar – drivers- not totally different objects, the color did not make much difference), then I changed the pooling layers as recommended by the document linked in there references [6], instead of using max pooling layers, we

apply to pooling layers after a conv layer, max and average and then use the sum of them, which has improved the loss further to 0.52




To improve the loss further I have used ensemble of predictions of cross validation, using 13 folds, which resulted in training on 25 drivers and validating on 2 for each fold, and improved the loss slightly to 0.5

Folds summary

| Fold number | Log loss |
|---|---|
| 1 | 0.546 |
| 2 | 0.748 |
| 3 | 0.621 |
| 4 | 0.583 |
| 5 | 0.388 |
| 6 | 0.701 |
| 7 | 0.440 |
| 8 | 0.320 |
| 9 | 0.613 |
| 10 | 0.574 |
| 11 | 0.589 |
| 12 | 0.544 |
| 13 | 0.475 |

The best fold (5) had drivers 16, 21 left out, and based on the distribution of the drivers, those were among the drivers with the highest count, and driver 21 is indeed the highest.

This is due to images taken within seconds of each other, resulting in very similar images, and for drivers with high amount of images that led to over fitting to those drivers, which is why I decided to use drivers 21 and 22 for validation when using bigger model and removing cross validation, as they have highest amount of images.

I didn't include driver 24 in validation set also, because removing three drivers from a limited dataset might reduce the predictive power.

As the validation log loss is currently at 0.5 and the selected benchmark is 0.42, I switched to transfer learning and fine tuning a pre-trained model to overcome the dataset size limitation, and switched to VGG16 [7].

The idea of transfer learning is taking an already trained model over a large dataset (imagenet in this case) and only fine tune specific layers on the dataset at hand.

Typically the last layer (classifier) is removed and instead we add one that match the classes we have, image net has 1000 classes while this dataset has 10.

A key point to consider is the added layer has randomly initialized weights while the rest of the layers the learned weights, thus it is important to split training into multi-stages otherwise the gradient steps of the new layer would wreck the learned weights.

The first stage would be with all the layers with learned weights frozen, and training only the classifier.

The second stage would be to un-freeze the last convolutional block (or more) for fine tuning.

Summary of VGG16 result:

1- Training classifier block only and splitting data by driver randomly resulted in 0.64 log loss
2- Training classifier block only and splitting by the selected drivers (21, 22 for validation) resulted in 0.41 log loss, the same split method is used afterwards.
3- Training classifier then fine tuning last conv block resulted in 0.30
4- Training classifier then fine tuning the whole network resulted in 0.28

Comparing to the benchmark of 0.3588, the model passed the benchmark after fine tuning of the last conv block 0.3 and full network fine tuning 0.28

Example prediction

As seen in the image, the driver is actually distracted, let's see the predictions for this image.

In the below table are the probabilities given by the classifier that the image belongs to a specific class

| c0 | c1 | c2 | c3 | c4 | c5 | c6 | c7 | c8 | c9 |
|---|---|---|---|---|---|---|---|---|---|
| 2.34E-08 | 3.68E-08 | 1.21E-08 | 1.57E-08 | 1.96E-08 | 0.999999 | 9.38E-08 | 7.54E-08 | 1.09E-07 | 1.22E-07 |

The probability of the image being class 5 is 0.999999, which means the classifier is very confident about the image's class, checking the classes list mentioned earlier, the class c5 is: operating the radio.

# Conclusion

Splitting the data by driver has better generalizability for other drivers, by using a CNN architecture of 3 convolutional layers each followed by ReLU activation, pooling layer, and drop out, and classification layer, resulted in log loss 0.65 which was further improved, by replacing the max pooling layer by sum of max and average pooling, to 0.52, the main challenge to reach this model was finding an architecture that would improve the loss more than 1.8, the key point was using large pooling layers (8x8) after the last conv block, another interesting thing I noticed during selection of architecture and hyper-parameters is that batch size and learning rate need to be changed proportionally, for example, when using batch size of 64 and learning rate 0.001, changing the batch size to 16 would results in too high learning rate and the loss will be stuck around 14.5, but when the rate is decreased to a number between 0.0003 (64/3) and 0.0001, it would start learning again.

In situation like this with limited size dataset it's good idea to use transfer learning, here VGG16 was used and by splitting the drivers randomly and not implementing cross validation, we got log loss 0.64.

And by just selecting the split drivers according to the exploratory data analysis done improves the loss to 0.41, and fine tuning the the last conv block improves the loss significantly to 0.30. This is further improved to 0.28 by fine tuning the whole network instead of the last layers only.

This result could be improved further by implementing cross validation with multi-stage fine tuning. Other ways to improve it might include trying other pre-trained models .e.g. VGG19, or ResNet.

# References


[1] https://www.kaggle.com/wiki/LogarithmicLoss

[2] http://cs231n.github.io/convolutional-networks/

[3] https://arxiv.org/pdf/1412.6980v8.pdf

[4] https://keras.io/

[5] http://deeplearning.net/software/theano/

[6] https://arxiv.org/pdf/1606.02228v1.pdf

[7] http://www.robots.ox.ac.uk/~vgg/research/very_deep/

https://blog.keras.io/