CAPSTONE PROJECT

UDACITY MACHINE LEARNING NANODEGREE

1. Definition

1.1 Project Overview

According to the CDC motor vehicle safety division, one in five car accidents is caused by a distracted driver. Sadly, this translates to 425,000 people injured and 3,000 people killed by distracted driving every year.

State Farm hopes to improve these alarming statistics, and better insure their customers, by testing whether dashboard cameras can automatically detect drivers engaging in distracted behaviors. Given a dataset of 2D dashboard camera images, State Farm is challenging Kagglers to classify each driver's behavior. The aim is to determine whether the driver is paying attention to the road, wearing their seat belt, distracted by passengers in the car, drinking coffee, or using a mobile phone.

The aim of this Capstone project is to train a machine learning algorithm to classify the behavior of drivers using this dataset of 2D dashboard camera images.

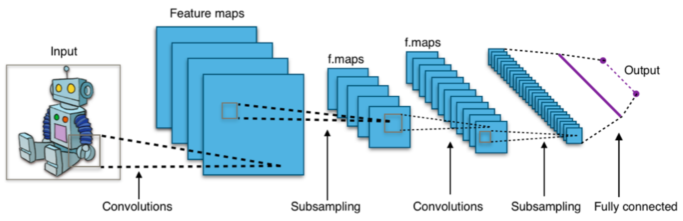
1.2 Problem Statement

The problem that this project aims to solve is to train a machine learning algorithm that can correctly identify the behavior of drivers based on 2D images captured by a dashboard camera. These images will be used to determine whether a driver spends more time focused and driving attentively, or whether they are distracted by other passengers, by drinking coffee, using a mobile phone or applying makeup.

There are a number of different machine learning approaches that could be taken, but in this study we look into CNN-based image recognition algorithms. CNNs, or Convolutional Neural Networks, are a type of feed-forward artificial neural network in which the connectivity pattern between neurons is inspired by the organization of an animal visual cortex, whose individual neurons are arranged in such a way that they respond to overlapping regions tiling the visual field. When used for image recognition, CNNs consist of multiple layers of small neuron collections which process portions of the input image, called receptive fields. 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. Tiling allows CNNs to tolerate translation of the input image.

Convolutional neural networks are often used in image recognition systems. They have set the standard in many image classification benchmarks such as MNIST4 and ILSVRC5. As this is an image classification problem, a CNN based machine learning algorithm will be used. The input of the network will be the 2D image taken from the dashboard camera and the output from the network is the predicted likelihood of what the driver is doing in each image.

The approach for processing is illustrated in the following diagram:

2D Driver's image	Neural Network Image Processing	Classification
	 Input Feature maps f.maps f.maps Output Convolutions Subsampling Convolutions Subsampling Fully connected	Drinking

The following have been defined by State Farm as the list of classes for this exercise:
c0: safe 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 neural network will be built using Python, Theano and Keras . Theano is 6 7 a python library that allows the programmer to define, optimise and evaluate mathematical expressions involving multidimensional arrays efficiently. It has the ability to use a GPU if one is available purely by changing configuration options. Keras is a minimalist, highly modular neural networks library written in Python and capable of running on top of Theano (or TensorFlow), and will be the basis for the CNN built to solve this problem.

1.3 Metrics

Kaggle's requirement is to predict the likelihood of what the driver is doing in each image across all 10 possible classifications. Performance of the classifier is to be measured using a categorical cross entropy function over validation data.
Log loss is a classification loss function often used as an evaluation metric and is defined as:

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

where N is the number of
samples, M is the number of possible labels/classes, yij is a binary indicator of whether or not label j is the correct classification for instance i, and pij is the model probability of assigning label j to instance i. A perfect classifier would have a log loss of precisely zero, and less ideal classifiers have progressively larger values of log loss.

In plain English, this error metric is used where we have to predict that something is true or false with a probability (likelihood) ranging from definitely true (1) to equally true (0.5) to definitely false(0). In this case, we have the similar situation where a driver might be texting with a right or left hand and our job was to identify that driver is texting and using left or right hand. In this case we need to go with the solution which can provide that driver is texting and may not be completely accurate whether driver is using right or wrong for texting. I guess Log loss is very suitable to provide this kind of prediction where we may not be completely true or completely false.

One of the challenges with Log Loss is that is heavily penalizes classifiers that are confident about an incorrect classification. The use of log on the error provides extreme punishments for being both confident and wrong. This means that it is better to be somewhat wrong than completely wrong, and suggests that smoothing the results set may provide a better overall benchmark. What it means is, based on above example of texting, it is better to identify that driver is texting than not identifying that driver is texting using right or left hand.

2. Analysis

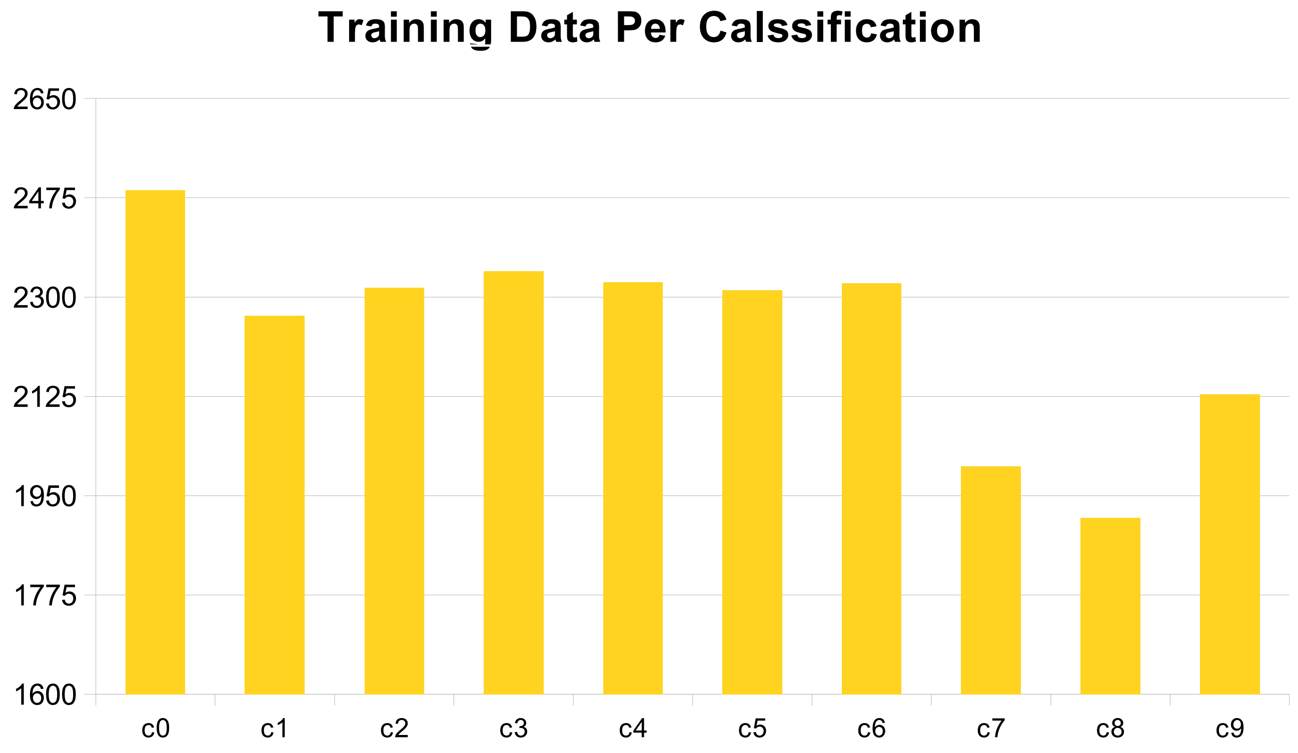
2.1 Data Exploration

For this particular Kaggle competition, the data has been provided by State Farm in the form of 2D JPEG images. Each image is sized 640x480 pixels and are in color. Each of the images show the inside of the car and a number of different drivers performing various actions.

The following key points have been identified:

2.1.1 Training Data

There are 22,424 training images split across each of the 10 classifications, although this split is not equal. Therefore some classes have more training samples than others as can be seen in this graph.



Drawing 1: Training Data Calssification

What this means is that the learning algorithm is likely to find it harder to classify images from c7 (reaching behind) and c8 (applying hair and make-up) based strictly on the number of images it has to learn from. In theory, the algorithm should find it easier to predict normal driving, and therefore might actually predict that this is the case for any borderline samples (e.g. the network may prefer to predict normal driving). In terms of the outcome of the exercise, predicting normal driving when the driver is distracted is not an ideal outcome.

Additionally there are only 26 drivers in the complete training data set, which means there is significant risk of overfitting during training. This can be highlighted when comparing similar images that are from different classes:

If the machine learning algorithm is not structured correctly, then it is likely to take features that identify the driver or the vehicle rather than from the action that the driver is taking. This may make it inherently hard to train a model that has a good chance of predicting the driver's behavior.

2.1.2 Validation Data

The dataset does not provide pre-selected validation images, so these will need to be selected from the training images. Due to the limited number of drivers, it is envisaged that the validation set contains drivers that are not used to train the model.

2.1.3 Testing Data

There are 79,726 testing images, significantly higher than the number of training images. None of the drivers in the test data set are in the training data set.

2.2 Algorithm and Techniques

2.2.1 Introduction to CNN

Convolutional Neural Networks are very similar to ordinary Neural Networks. 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 architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features.

2.2.2 Building Blocks of CNN

A CNN architecture is formed by a stack of distinct layers that transform the input volume into an output volume (e.g. holding the class scores) through a differentiable function. A few distinct types of layers are commonly used.

2.2.2.1 Convolutional layer

The convolutional layer is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume.

- **Local Connectivity -** When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume because such a network architecture does not take the spatial structure of the data into account. Convolutional networks exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers: each neuron is connected to only a small region of the input volume. The extent of this connectivity is a hyperparameter called the receptive field of the neuron. The connections are local in space (along width and height), but always extend along the entire depth of the input volume. Such an architecture ensures that the learnt filters produce the strongest response to a spatially local input pattern.
- **Spatial Arrangement –** Three hyperparameters control the size of the output volume of the convolutional layer: the depth, stride and zero-padding.
  1. **Depth** of the output volume controls the number of neurons in the layer that connect to the same region of the input volume. All of these neurons will learn to activate for different features in the input.
  2. **Stride** controls how depth columns around the spatial dimensions (width and height) are allocated. When the stride is 1, a new depth column of neurons is allocated to spatial positions only 1 spatial unit apart. This leads to heavily overlapping receptive fields between the columns, and also to large output volumes. Conversely, if higher strides are used then the receptive fields will overlap less and the resulting output volume will have smaller dimensions spatially.
  3. Sometimes it is convenient to pad the input with zeros on the border of the input volume. The size of this **zero-padding** is a third hyperparameter. Zero padding provides control of the output volume spatial size. In particular, sometimes it is desirable to exactly preserve the spatial size of the input volume.
- **Parameter Sharing -** Parameter sharing scheme is used in convolutional layers to control the number of free parameters. It relies on one reasonable assumption: That if one patch feature is useful to compute at some spatial position, then it should also be useful to compute at a different position. In other words, denoting a single 2-dimensional slice of depth as a **depth slice**, we constrain the neurons in each depth slice to use the same weights and bias.

2.2.2.2 Pooling Layer

Another important concept of CNNs is pooling, which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling among which max pooling is the most common. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features. The function of the pooling layer 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. It is common to periodically insert a pooling layer in-between successive conv layers in a CNN architecture. The pooling operation provides a form of translation invariance.

2.2.2.3 ReLU Layer

ReLU is the abbreviation of Rectified Linear Units. This is a layer of neurons that applies the non-saturating activation function. It increases the nonlinear properties of the decision function and of the overall network without affecting the receptive fields of the convolution layer. Other functions are also used to increase nonlinearity, for example the saturating hyperbolic tangent.

2.2.2.4 Fully Connected Layer

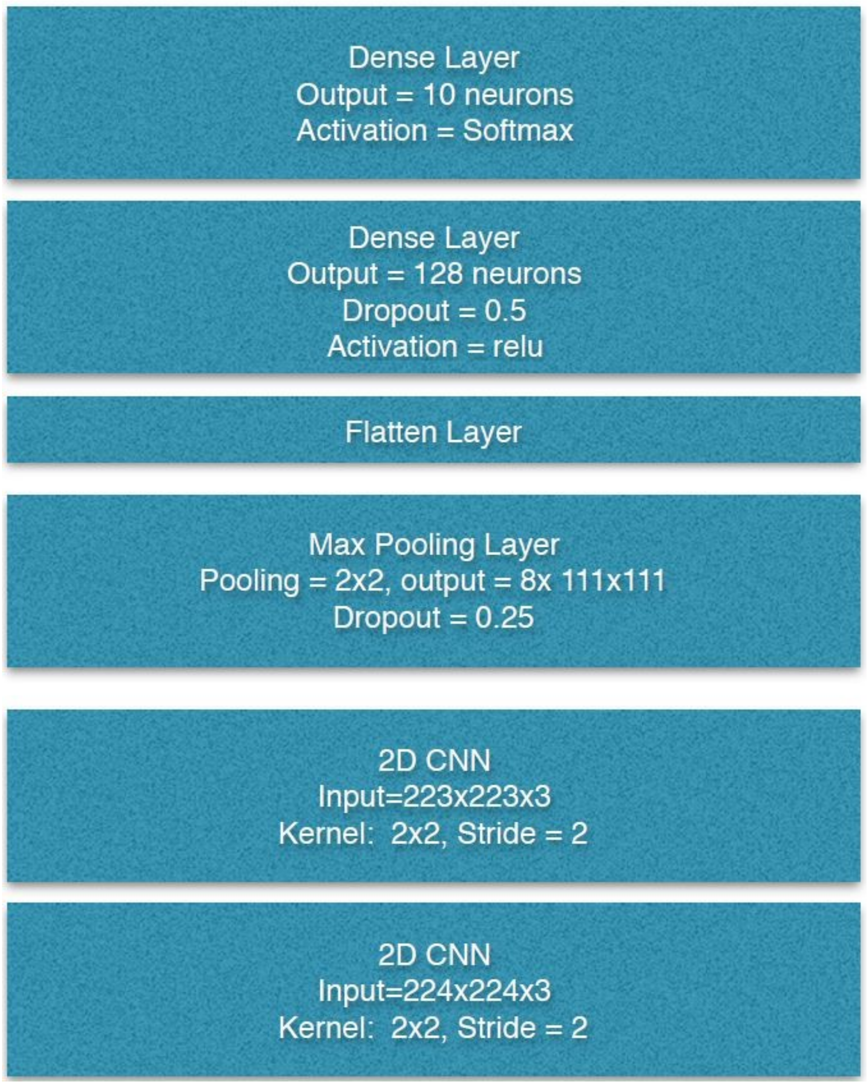
Finally, after several convolutional and max pooling layers, the high-level reasoning in the neural network is done via fully connected layers. Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

2.2.2.5 Loss Layer

The loss layer specifies how the network training penalizes the deviation between the predicted and true labels and is normally the last layer in the network. Various loss functions appropriate for different tasks may be used there. Softmax loss is used for predicting a single class of K mutually exclusive classes. Sigmoid cross-entropy loss is used for predicting K independent probability values in [0, 1].

2.2.3 CNN Architecture

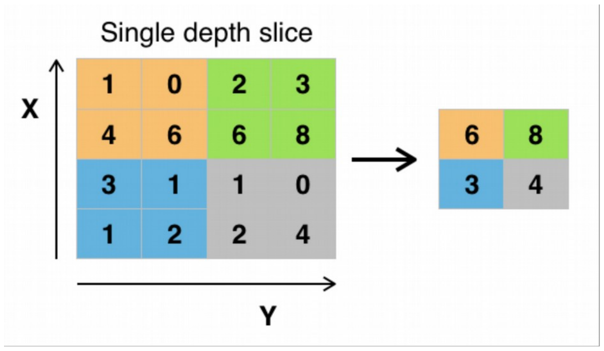
The overall aim of this architecture is to take the input image (in this case re-sized) and gradually identify features within the image, reduce it's dimensionality, and then reason as to which features provide the best indication of the driver's behavior.
As discussed previously, the algorithm to be used is a deep convolutional neural network. The proposed network architecture is:



Drawing 2: CNN Architecture

As discussed previously, CNN-based deep-learning architectures are ideally suited to image recognition tasks. In this architecture, the bottom 2 layers are used to identify features in each image. The bottom layer is likely to identify such things as edges or textures, and the 2nd layer aims to identify more complex structures such as an arm, a face or a phone.

A max-pooling layer is 8 common approach to perform non-linear down-sampling of the identified features. Max-pooling partitions the image into non-overlapping sub-regions, and for each sub-region output the maximum. The intuition is that once a feature is found, it doesn’t matter where in the image it is. The maxpooling layer also reduces the size of the representation of the image, and therefore also reduces the amount of parameters and computation in the network. It also is a way to control over-fitting, and also provides a form of translation invariance.



Drawing 3: Max Pooling

The flatten layer takes the 2D representation of the image and converts it into a 1D representation suitable for feeding into a normal dense layer of neurons. The 2 dense layers, also known as fully-connected layers, are designed to translate the identified features into a prediction of the driver’s action. This is the higher-level reasoning of the network.

The final layer is the loss layer and determines how the network penalizes the deviation between the predicted labels and the true labels. In this case, there is a single mutually-exclusive class to be predicted, so Softmax is 9 ideal loss function.

‘ReLU’, or Rectified Linear Units are used across the network as they provide a number of distinct advantages of other types of neurons. Although they help mitigate the risk of vanishing gradients, in this case they have been chosen as they can be trained effectively10 without pre-training the network on other data.

Dropout11 has been used in the architecture as this strengthens the network by forcing it to learn a number of different representations of the data, and additionally it also reduces overfitting. As described earlier, there is a risk of overfitting due to the limited number of drivers in the training data set.

The following parameters can be tuned to optimize the classifier:

- Training parameters
  - Training length (number of epochs)
  - Batch size (how many images to train with in a single training step)
  - Weight decay and momentum
  - Learning rate
  - Size of the training vs validation sets
- Neural network architecture
  - Number and type of layers
  - Network parameters, such as drop outs, stride, kernel size
- Pre-processing of the images
  - Image size
  - Greyscale vs color
  - Cropping
  - Random ordering of training images

During training, both training and validation sets are loaded into RAM. After that, batches are selected and to be loaded into the GPU memory for processing, and training is done using standard gradient descent without momentum.

2.3 Benchmark

To create an initial benchmark for the classifier, I aimed to achieve a Kaggle score in the top 500 which represented a log loss score of 0.8895212. Additionally, with an end-user market in mind, I wanted to ensure that the trained neural network could predict the behavior of a driver in under 2 seconds using a typical laptop (in my case a Macbook Pro) without GPU support in order to represent a low-power, 13 non-GPU based device within the car. I assumed that 2 seconds on a Mac would translate to being able to predict driver behavior every 10 seconds in-car. So if any analysis or action needs to be taken on driver behavior, 10 seconds interval is reasonable.

3. Methodology

3.1 Data Preprocessing

It would be possible for a CNN based deep network to learn directly from the images provided by State Farm. However, in order to reduce the size of the network and accordingly the number of parameters, the images are re-sized from full color 640x480 pixels to grey scale 224x224 pixels.

The training set is also split into training and validation sets. The number of drivers in the training set vs the number in the validation set is a tunable parameter, currently set to 95% (this means 2 drivers are allocated to the validation set).

Pre-processing is done entirely before the neural net is trained. Here are examples of preprocessed images:

Due to the risk of overfitting, it would be ideal to increase the size of training images by rotation, skewing or other image manipulation techniques. This will be addressed in the section on “Improvements”.

3.2 Implementation

The following steps have been taken to implement the pre-processing, training and testing:

3.2.1 Pre-processing

A complete list of images and their labels is provided in the file “driver_imgs_list.csv”. The format of this csv file is “subject, classname, img”. This file is read in for future use. There is no list of testing images provided directly, so this list is created by reading the directory of testing images.

Next, the 2 arrays are created, these are an array of image names and an array of the classes (or labels). As the pre-processing of images takes a long period of time, the images are processed and then stored in a separate directory (defined by the variables train_images_dir and test_images_dir). The option to create pre-processed images is handled by setting the variable create_repository to true (to create a new set of pre-processed images) or false (to use an existing set).

Next, a set of district drivers is collated. This is to ensure that the split between training and validation sets are split by driver in an aim to reduce overfitting, and to provide better metrics in training and validation.

The training images can then be split into training and validation sets. This could be done at training time using the “validation_split” parameter in the Keras mode.fit command, but as mentioned previously, the intuition is that the network would learn drivers as opposed to drivers behavior, and therefore the train/validation split is performed separately and is determined from the list of drivers and then their associated images.

3.2.2 Defining the Neural Network

The next step is to create a network using Keras. Keras has been chosen due its portability (it works on any Python platform with Theano or Tensorflow installed) and has a simple approach for defining and training neural networks. Caffe was originally considered for this but there was a higher level of complexity around creating training, validation and test data sets.

The neural network is based on the LeNet model as detailed previously. Categorical cross-entropy is used a loss function in line with recommendations for a multi-class classifier implemented in Keras.

Standard Gradient Descent is used as the optimizer and metrics are captured for accuracy.

3.2.3 Training

Each training run has 10 epochs, and each one uses the same training and validation data. A Keras callback is created in order to display the loss history and display a graph at the end of training.

3.2.4 Testing

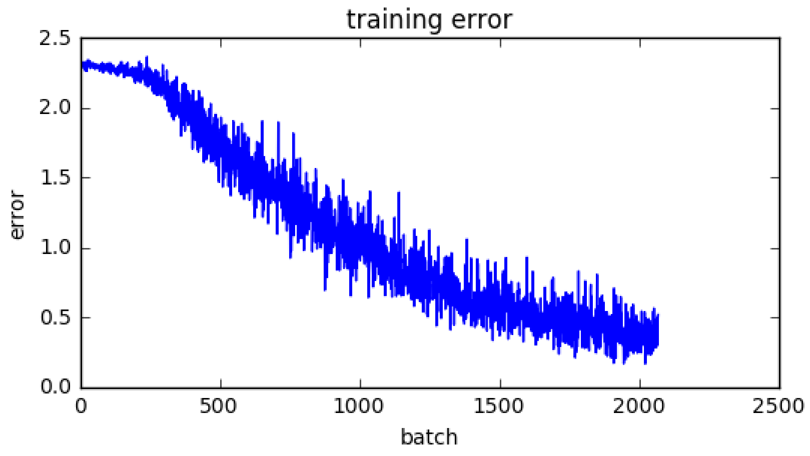
The final step is to predict the classes for the test data provided by Kaggle. This uses the model.predict function within Keras. Once the predictions have been made, the predicted classes are converted to CSV and saved to the local file system

3.3 Refinement

This is the training results over 5 epochs with an initial model described above:

```
Starting training iteration 1 with learning rate 0.01

Train on 20690 samples, validate on 1734 samples
Epoch 5/5
20690/20690 [==============================] - 490s - loss: 0.4203 - acc: 0.8707 - val_loss: 1.7365 - val_acc: 0.4343
```



Drawing 4: Initial Model - Training Result over 5 Epochs, Learning Rate 0.01

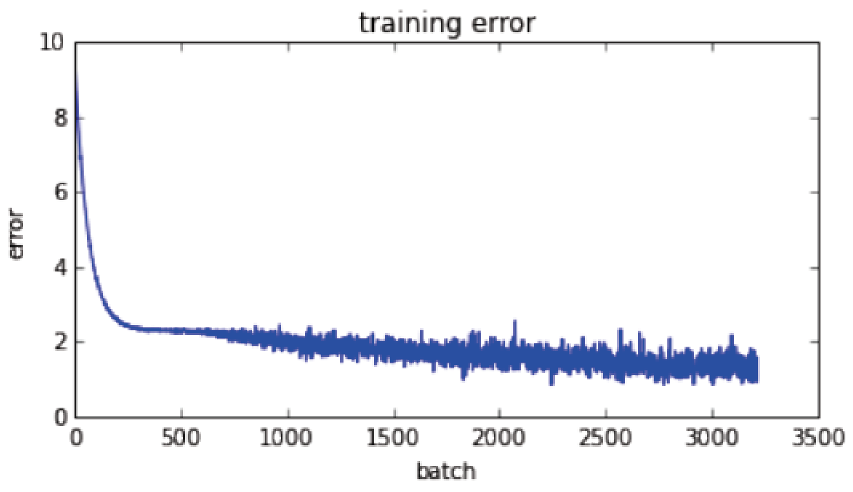
The following approach were taken to improve the algorithm:

3.3.1 L2 regularizers

Regularizers are a staple part of many deep networks and are designed to help the network perform well on training data and also on previously unseen (test) data . 14 My original network did not have any regularizers. L2 are the most common form of regularizers and therefore these were tried first.

Below is the training results with L2 regularizers with the following parameters:
W_regularizer = l2(0.1)
b_regularizer = l2(0.1)

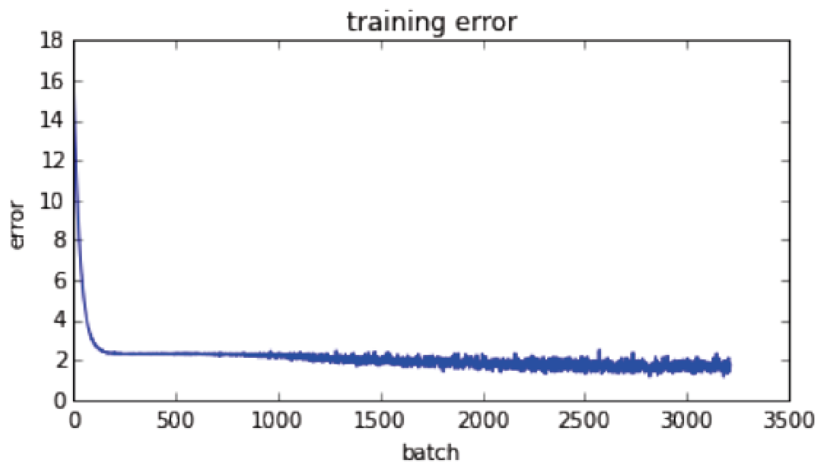
```
**********************************
Starting training iteration 1 with learning rate 0.01
Train on 20555 samples, validate on 1869 samples
Epoch 5/5
20555/20555 [==============================] - 458s - loss: 1.3620 - acc: 0.5743 - val_loss: 1.8434 - val_acc: 0.4216
```



Drawing 5: L2 Regularizer - Training Result over 5 Epochs, Learning Rate 0.01

Below is the training results with L2 regularizers with the following parameters:
W_regularizer = l2(0.2)
b_regularizer = l2(0.2)

```
**********************************
Starting training iteration 1 with learning rate 0.01
Train on 20555 samples, validate on 1869 samples
Epoch 5/5
20555/20555 [==============================] - 472s - loss: 1.6770 - acc: 0.4408 - val_loss: 1.8905 - val_acc: 0.3210
```



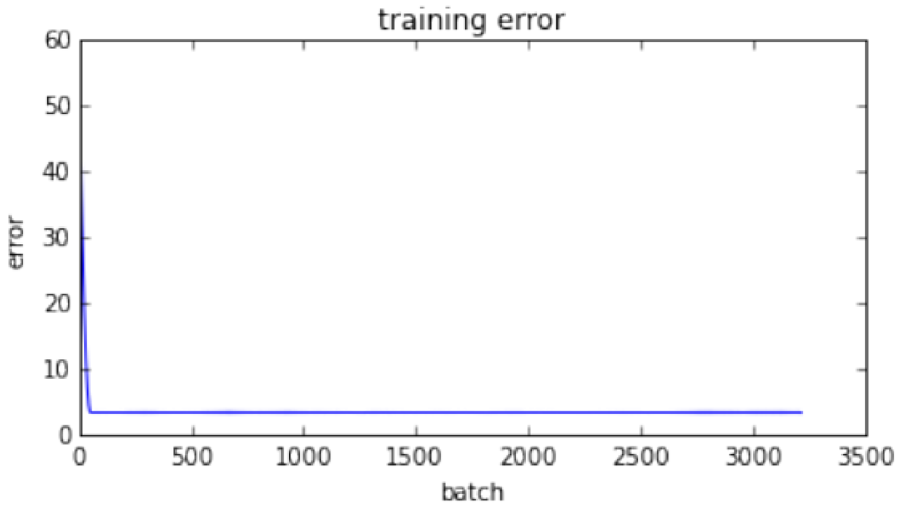
Drawing 6: L2 Regularizer - Training Result over 5 Epochs, Learning Rate 0.01

It can be seen that the training loss and accuracy are reduced by the introduction of the regularizers, but the accuracy of the validation metrics increased. However, the network also seems to learn much quicker.

3.3.2 L1 regularizers

L1 regularizers do not improve training performance as can be seen from these results and graph:

```
**********************************
Starting training iteration 1 with learning rate 0.01
Train on 20555 samples, validate on 1869 samples
Epoch 5/5
20555/20555 [==============================] - 483s - loss: 3.3349 - acc: 0.1025 - val_loss: 2.3027 - val_acc: 0.0883
```



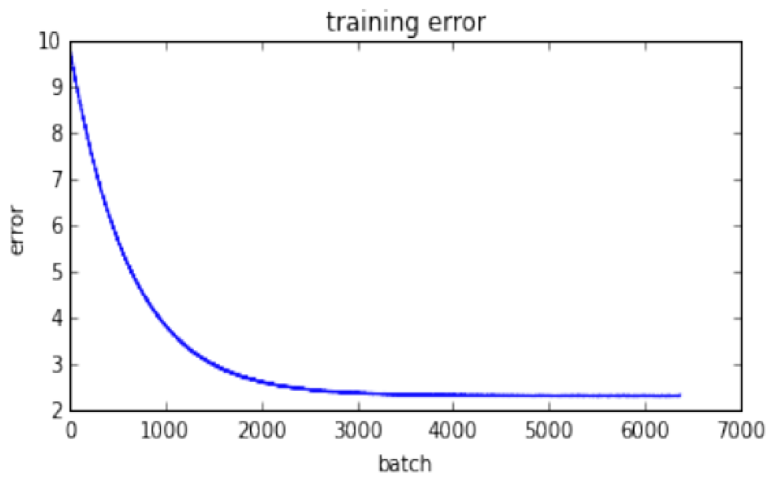
Drawing 7: L1 Regularizer - Training Result over 5 Epochs, Learning Rate 0.01

3.3.3 Learning Rates

Now that I had ascertained that L2 regularizers improved performance, different learning rates were used in order to determine an optimal rate. The learning rates trained against are 0.001, 0.003, 0.01, 0.03 and 0.1.

- Learning Rate 0.001

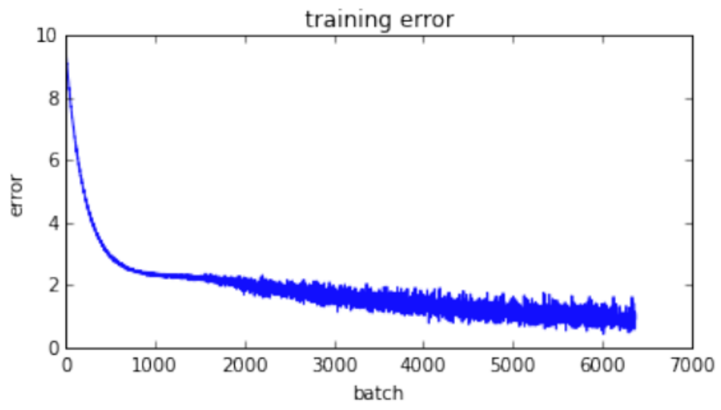
```
Starting training iteration 1 with learning rate 0.001
Epoch 10/10
20368/20368 [==============================] - 458s - loss: 2.3014 - acc: 0.1113 - val_loss: 2.3005 - val_acc: 0.1114
```



Drawing 8: Training Result over 10 Epochs, Learning Rate 0.001

- Learning Rate 0.003

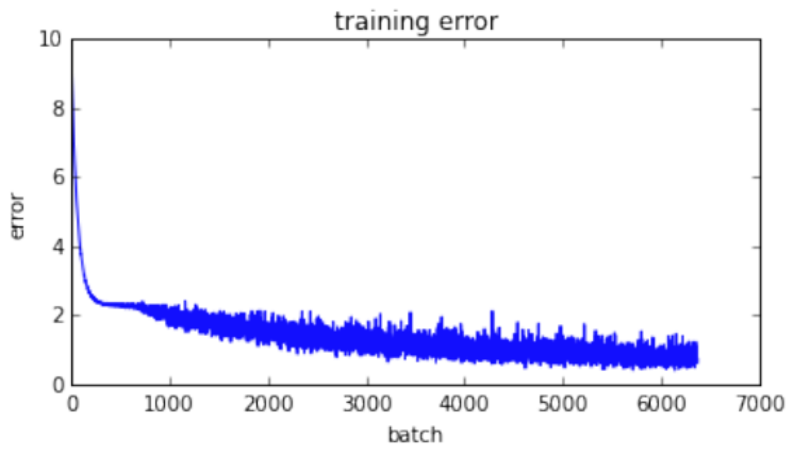
```
**********************************
Starting training iteration 2 with learning rate 0.003
Epoch 10/10
20368/20368 [==============================] - 468s - loss: 0.9574 - acc: 0.7202 - val_loss: 2.5641 - val_acc: 0.1659
```



Drawing 9: Training Result over 10 Epochs, Learning Rate 0.003

- Learning Rate 0.01

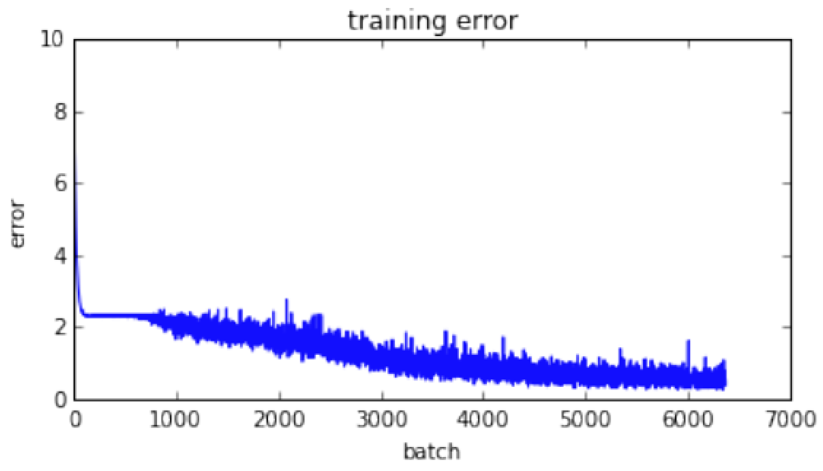
```
**********************************
Starting training iteration 3 with learning rate 0.01
Epoch 10/10
20368/20368 [==============================] - 414s - loss: 0.8108 - acc: 0.7857 - val_loss: 2.2433 - val_acc: 0.2549
```



Drawing 10: Training Result over 10 Epochs, Learning Rate 0.01

- Learning Rate 0.03

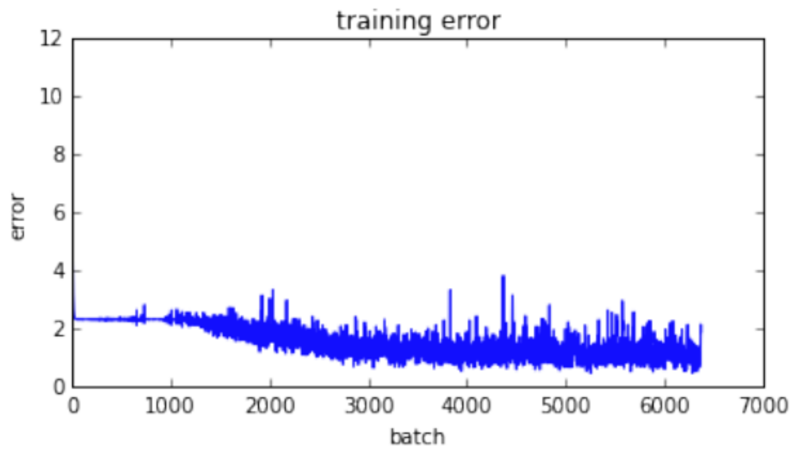
```
Starting training iteration 4 with learning rate 0.03
Epoch 10/10
20368/20368 [==============================] - 1446s - loss: 0.5534 - acc: 0.8908 - val_loss: 1.9968 - val_acc: 0.4611
```



Drawing 11: Training Result over 10 Epochs, Learning Rate 0.03

- Learning Rate 0.1

```
**********************************
Starting training iteration 5 with learning rate 0.1
Epoch 10/10
20368/20368 [==============================] - 8570s - loss: 1.1201 - acc: 0.7904 - val_loss: 2.2572 - val_acc: 0.3867
```



Drawing 12: Training Result over 10 Epochs, Learning Rate 0.1

It can be seen from these training runs that a learning rate of 0.001 or 0.003 provides the best compromise between training time and accuracy. However, it can be seen that the learning rate does cause accuracy to vary wildly between each training batch, which means that higher learning rates are not converging to a local minima. Intuition suggests that a learning rate of 0.001 does not converge to a global optima, or takes an inordinate amount of time to reach it, therefore a larger number of epochs combined with a learning rate (of 0.003) would provide the best training parameters.

Subsequent parameter tuning has been performed with a learning rate of 0.003.

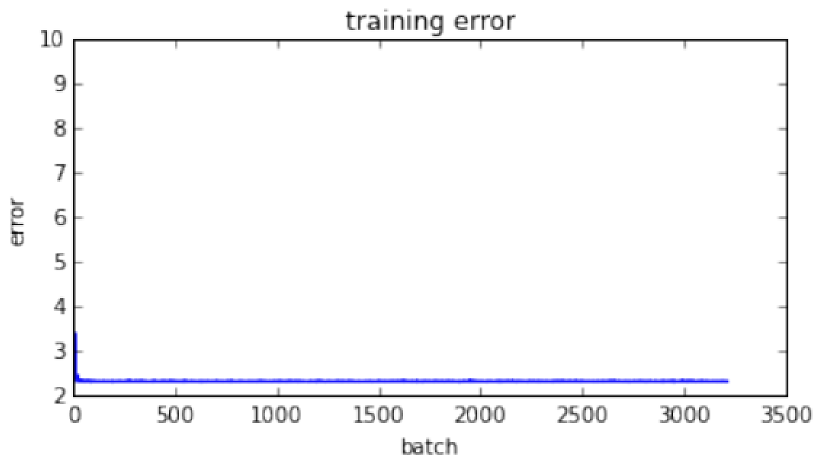
3.3.4 Tuning SGD parameters

I started with initial values for SGD of decay=0, momentum=0 and nesters=False. I then looked at alternative versions, for example using those from VGG-16. These are:
Learning rate=0.1
Decay=1e-6
Momentum=0.9
Nesterov=True

Momentum aims to help the algorithm navigate local optima by accelerating SGD in the relevant direction16. Nesterov is an accelerated gradient approach, it effectively looks ahead to see where the gradient might be in a subsequent iteration.

With these settings, early epochs have a lower loss metric, but the network does not converge any more beyond this point:

```
Starting training iteration 1 with learning rate 0.003
Train on 20555 samples, validate on 1869 samples
Epoch 5/5
20555/20555 [==============================] - 462s - loss: 2.3071 - acc: 0.1030 - val_loss: 2.3034 - val_acc: 0.1001
```



Drawing 13: With SGD Tuning - Training Result over 5 Epochs, Learning Rate 0.003

3.3.5 Pre-Trained Implementation

An pre-trained implementation of the VGG-16 network was also tested but this did not converge with the available data so was discarded.
Due to the time taken to train this model (circa 3 days), results have not been included in this.

- **VGG-16 Introduction** - This is the Keras model of the 16-layer network used by the VGG team in the ILSVRC-2014 competition. It has been obtained by directly converting the Caffe model provided by the authors. The aim of this project is to investigate how the ConvNet depth affects their accuracy in the large-scale image recognition setting. The main reason to develop this to have a rigorous evaluation of networks of increasing depth, which shows that a significant improvement on the prior-art configurations can be achieved by increasing the depth to 16-19 weight layers, which is substantially deeper than what has been used previously. To reduce the number of parameters in such very deep networks, they use very small 3×3 filters in all convolutional layers.
  Details about the network architecture can be found in the following arXiv paper:
  http://arxiv.org/pdf/1409.1556

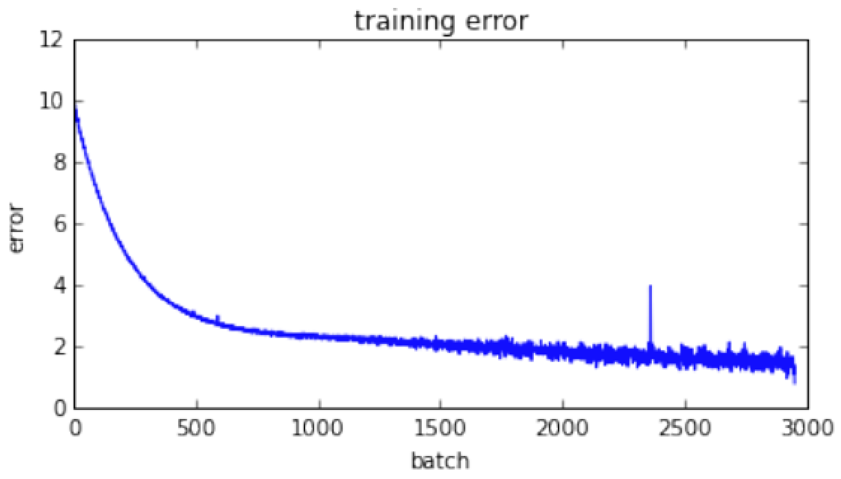
3.3.6 Size of Data Sets

Varying the size of the training and validation data sets, particularly by increasing the amount of training data

The became a trade-off between how much data could be used to train and the size of the dataset to be used for validation. Using too much data for validation reduced the effectiveness of the training, and reducing the size of the validation set to a single driver didn't necessarily provide confidence that the network would generalize to different drivers.

With 85% of drivers used for training (4 drivers remaining for validation), the results are:

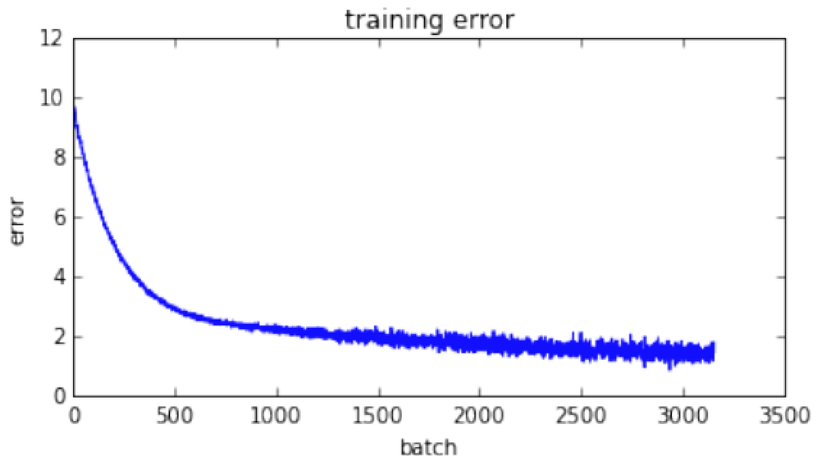
```
**********************************
Starting training iteration 1 with learning rate 0.003
Train on 18849 samples, validate on 3575 samples
Epoch 5/5
18849/18849 [==============================] - 426s - loss: 1.5604 - acc: 0.4771 - val_loss: 2.9294 - val_acc: 0.2537
```



Drawing 14: With 85% Drivers - Training Result over 5 Epochs, Learning Rate 0.003

With 95% of drivers used for training (2 drivers remaining for validation), the results are:

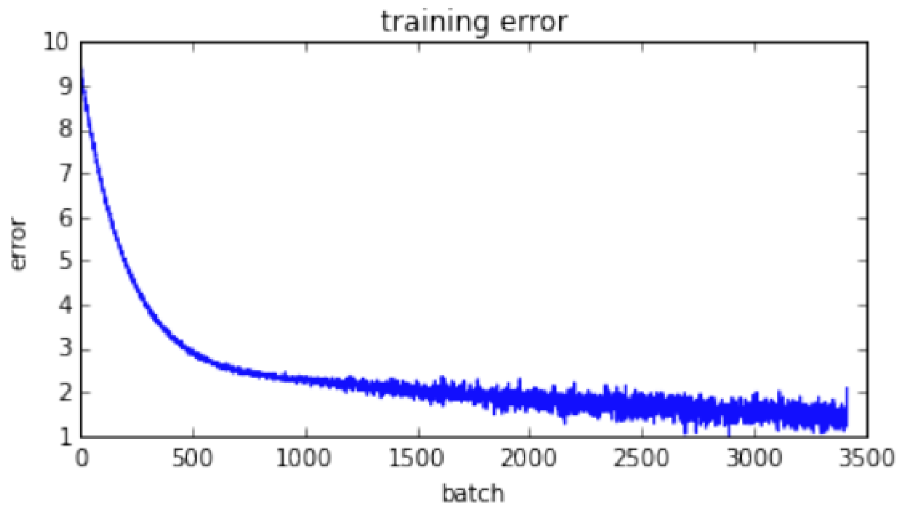
```
**********************************
Starting training iteration 1 with learning rate 0.003
Epoch 5/5
20187/20187 [==============================] - 433s - loss: 1.4555 - acc: 0.5211 - val_loss: 1.4536 - val_acc: 0.5422
```



Drawing 15: With 95% Drivers - Training Result over 5 Epochs, Learning Rate 0.003

With 98% of drivers used for training, (1 driver remaining for validation), the results are:

```
**********************************
Starting training iteration 1 with learning rate 0.003
Epoch 5/5
21833/21833 [==============================] - 460s - loss: 1.5045 - acc: 0.5014 - val_loss: 2.2921 - val_acc: 0.2047
```



Drawing 16: With 98% Drivers - Training Result over 5 Epochs, Learning Rate 0.003

3.3.7 Increased the batch size

The final run increased the batch size to 64 and then subsequently to 128 and achieved much greater performance. See the final results section below for details.

4. Results

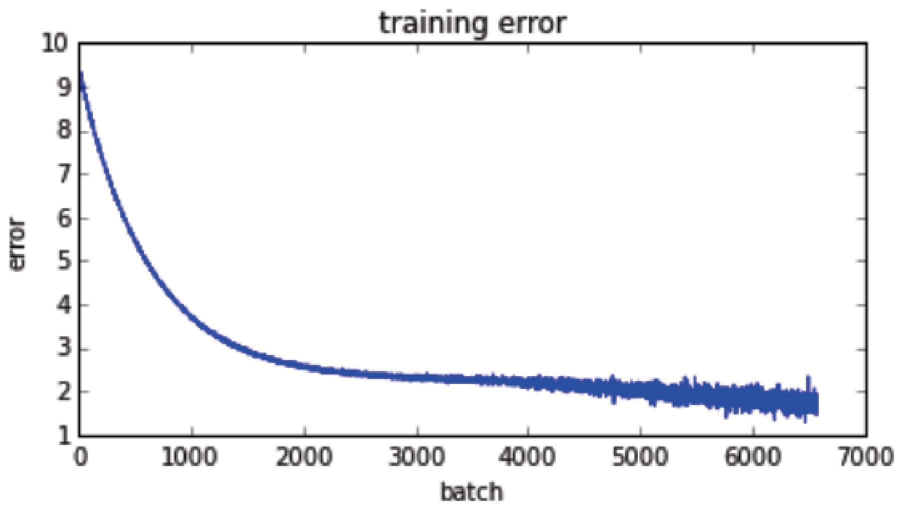
4.1 Model Evaluation and Validation

The final model is deemed to be reasonable but does not fully align with the expectations set out in the introduction. It typically results in a leaderboard position around 1300th, and not in the top 500 as originally targeted.

The final model was derived using the following approaches:

1. Initially, small training runs were performed with limited numbers of epochs (perhaps 2-5 epochs) to understand the model's behavior.
2. I observed the training and validation metrics of the model to determine whether it was converging when trained and how it performed on validation data. Results of each training run were determined based on data such as:

```
Starting training iteration 1 with learning rate 0.001
Train on 20996 samples, validate on 1428 samples
Epoch 10/10
20996/20996 [==============================] - 369s - loss: 1.7738 - acc: 0.4002 - val_loss: 1.6683 - val_acc: 0.3410
```



Drawing 17: Training Result over 10 Epochs, Learning Rate 0.003

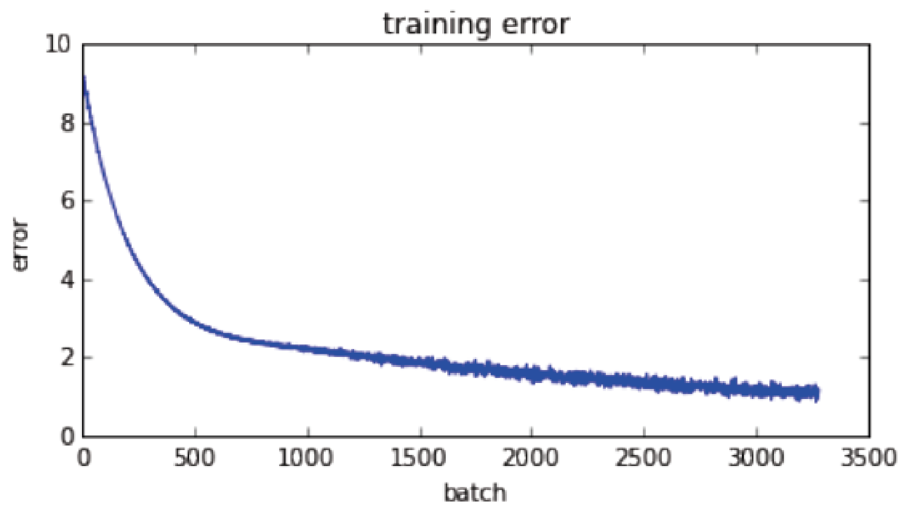
Train on 20996 samples, validate on 1428 samples

3. When the model didn't behave as expected, I investigated best practice for deep CNNs and made modifications accordingly. These changes included:
   - ➔ regularizers such as trying L1 and L2.
   - ➔ Modifying the kernel size (from 2x2 to 3x3)
   - ➔ Changing the number of filters in the CNN layers
   - ➔ Running for different number of epochs
4. I made modifications to the training parameters, such as the values for L2, the learning rate, and the SGD values. Some of these values were taken from pre-existing models that I researched (for example, I tried SGD values from a VGG-16 model), and others were determined by training using different values.

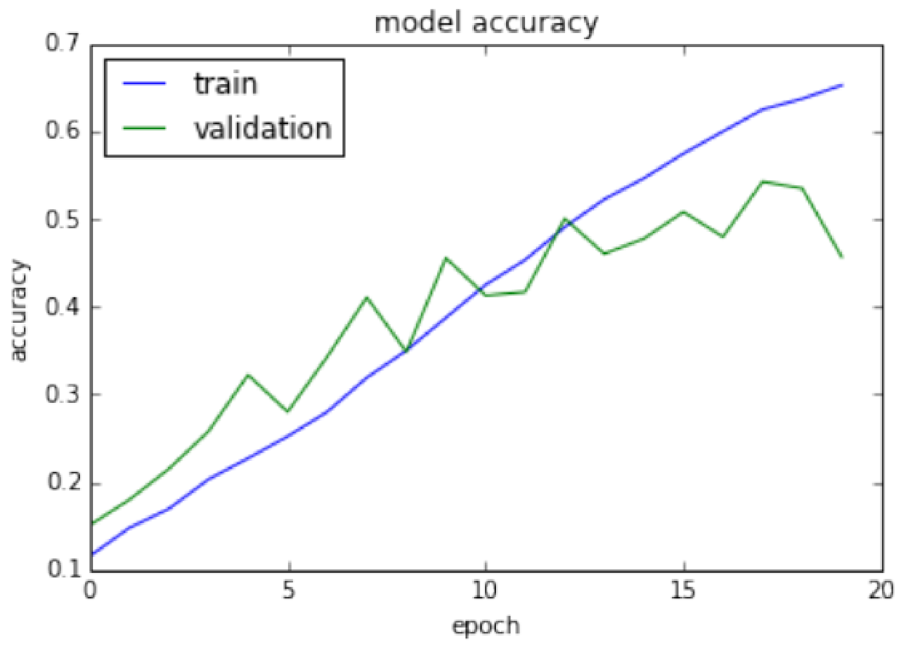
4.2 Final Run Metrics

The final run metrics are:

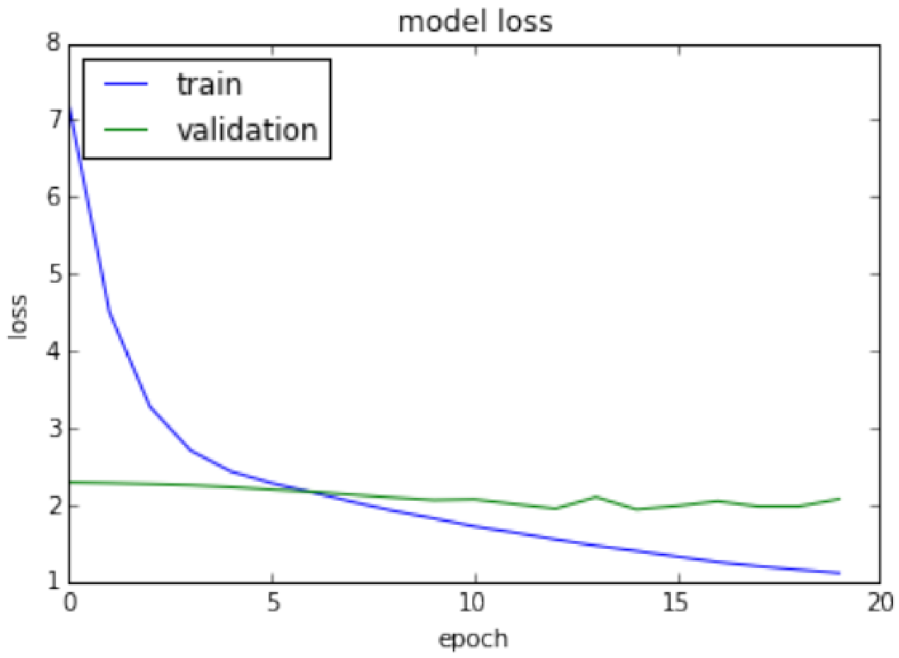
```
Train on 20975 samples, validate on 1449 samples
Epoch 20/20
20975/20975 [==============================] - 394s - loss: 1.1143 - acc: 0.6524 - val_loss: 2.0758 - val_acc: 0.4569
```



Drawing 18: Final Run - Training Result - Train on 20975 samples, validate on 1449 samples



Drawing 19: Final Run - Model Accuracy Graph



Drawing 20: Final Run - Model Loss Graph

This model was use to predict against the test dataset and the results show a MSE loss of **2.09736**

The following observations have been made with the final model:

- It tends to converge to similar training and validation loss metrics irrespective of which samples are in the train and validation data sets. This can be tested by created new training and validation data sets and re-running the model.
- It generalizes reasonably well to unseen data.
- After 12-13 epochs, the validation metrics tend not to improve (and potentially even start to degrade) as seen in the graphs above.
- If the initial learning rate is too large, then there are occasional and dramatic increases in the validation loss.
- Implementation of the L2 regularizer has reduced the smoothness of the loss over the training runs, but has improved the validation metrics.

4.3 Justification

The model is able to predict on the 77k testing samples in approximately 10 minutes on a Macbook , this equates to circa 130 samples per second which is well within the benchmark defined earlier. However, the predictive performance of the model is not deemed to be suitably reliable for real world performance as its log loss metric on the test data is 2.5x larger than that of the targeted 500th place benchmark.

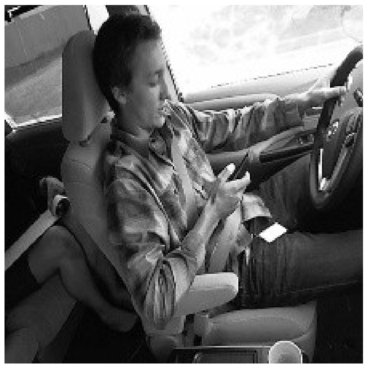
To become a suitable model, it would be necessary to implement some of the improvements mentioned in section 5.3 of this document below.

5. Conclusion

5.1 Free-form Visualization

Here are examples of validation images that have been correctly predicted:

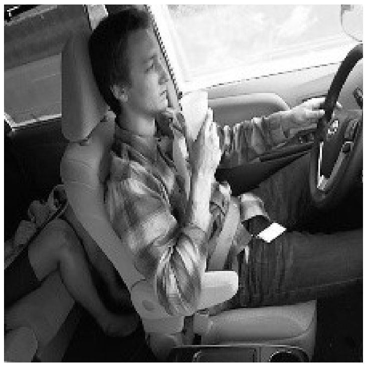
However, these images have classified incorrectly:



Drawing 21:

Class c1: texting – right

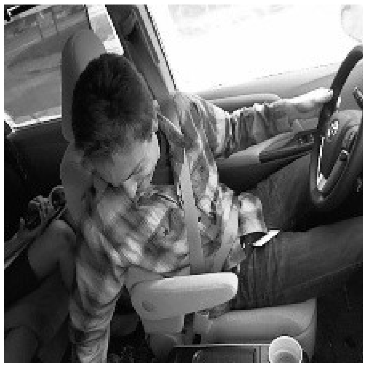
Predicted as c3: texting - left



Drawing 22:

Class c6: drinking

Predicted as c0: safe driving



Drawing 23:

Class c7: reaching behind

Predicted as c1: texting – right

What is obvious from these is that the CNN is confused between left and right arms, arm rests in the car and the arms of the driver, and the general background of the car. However, 2 of the 5 images above are correctly classified as texting, but the CNN was not able to correctly determine which of the arms the phone was in.

5.2 Reflection

The process used for this project can be summarized using the following steps:

1. An initial problem was found on Kaggle that was deemed to be interesting and provided a way to explore computer vision and deep learning.
2. The data was provided on Kaggle and I downloaded from there.
3. The images were pre-processed to reduce their size. Initially I did this for each training run, resizing them and putting them into training and validation folders. But later I pre-processed all of the images in a single batch

and then use an array to determine which images were for training and which were for validation. This greatly reduced the time between training runs.

4. A benchmark was created for the classifier. This was an estimate based on pure guesswork.
5. The available training data was split into training and validation data sets
6. The classifier was trained on the available data over a number of epochs. There were some issues getting Theano and Keras to play well with the inbuilt GPU on my Macbook.
7. After each training run, the parameters for the network were tuned, and additional controls such as regularizers were added to the network in order to determine an optimal network configuration. Initial training runs contained a lot of NaNs. Some of the issues came in my code which took a long time to debug as I am new to both deep neural nets and Keras.
8. Once a suitable model was created, predictions were made against the test data and the results.

I also faced the following additional difficulties:

1. Consistent time to work on the Capstone due to commitments in work.
2. Lack of available documentation relating to using Keras in different real-world scenarios (I note that this has improved significantly over recent months)
3. Instability of GPU features in Theano and Cuda on OSX. I have had occasions where code that worked on GPU one day didn't work the next, requiring reboots or restarting the ipython notebook process.

5.3 Improvements

The following improvements are suggested in order to make this a more usable model:

1. Reduce learning rate after 3000 iterations, or as it becomes evident that the loss is becoming erratic (symptomatic that the learning rate is too high). It is not clear how to do this in Keras as the learning rate is part of the model.compile function.
2. Increase the volume of training data by rotating and skewing each of the training images.
3. Trying alternative deep networks, such as a VGG16 model. Introduction for VGG-16 is provided above. Detail of the VGG-16 model can be found on following link. http://arxiv.org/pdf/1409.1556
4. Using a pre-trained network. A pre-trained VGG16 model was used but it did not converge with the existing training data set. This should be tried in conjunction with increasing the amount of training data as described above
5. Trying alternative optimizers such as Adam.
6. Labellings the images in more detail, for example building a network that learn purely from the arm position and face positions (perhaps either by using a labeling tool, or by training a network on individual parts of an image)
7. Instead of randomly picking training and validation data, find a set of training and validation images that perform the best in terms of log loss.
8. There is an option to crop the images to just the driver, or more specifically the driver's body, to reduce the amount of irrelevant information in each image. A mask of the body would also remove such irrelevant information such as the color or patterns of the drivers' clothing, etc. This may help with the issues identified previously where the network was confused between the left and right arm, and the arms of the driver and the inside of the car.
9. Cross-validation could be implemented, for example rotating through the data set and holding out 2 different drivers for validation on each set of training runs (so effectively training 13 different models). Visual checking could then determine what was unique about the best-fit images. An addition to this would be to implement ensemble learning.