Spotting Distracted Drivers from a Dashboard Camera

Kyle Griswold
Stanford University
kggriswo@stanford.edu

Abstract

Distracted driving is a big problem in the United States, killing 3179 and injuring 431000 people in 2014 alone [1]. To help mitigate this problem, State Farm set up a competition on Kaggle [2] for people to create machine learning systems that take an image of a person driving a car and classify whether they are distracted or not. This paper details our participation in this competition, where we use state of the art techniques in Convolutional Neural Networks in an attempt to solve this problem

1. Introduction

The problem posed by State Farm is an image classification problem to create an architecture that can read in a single image of a driver (with no other metadata) and to classify whether that driver is distracted, and if so, in what way that driver is distracted (eg. talking on their cell phone, messing with the radio, etc.) The 10 classes that we need to classify each image into are as follows:

1. safe driving
2. texting - right
3. talking on phone - right
4. texting - left
5. talking on phone - left
6. operating the radio
7. drinking
8. reaching behind
9. hair and makeup
10. talking to passenger

Image classification is one of the classic problems in computer vision, so much so that the premier competition in computer vision, the ImageNet challenge [3], is primarily an image classification challenge. This competition was won by Google's Inception Network in 2014 [4] and Microsoft's Residual Network in 2015 [5], so we will be using some of Google's techniques for our architectures in this paper (we would use Microsoft's as well, but our hardware setup isn't large enough to effectively incorporate Microsoft's techniques).

As for what our architectures will be, we will be using Convolutional Neural Networks (CNNs) as a means of classifying these images. Since CNNs have been used to great success in many computer vision problems (including the ImageNet challenge - both Google's and Microsoft's models are CNNs), these seem to be the best approach to create a well performing machine learning model. We will be incorporating many state of the art techniques in an attempt to better classify the images, including Dropout, Batch and Spatial Normalization, Adam training, and several aspects of Google's Inception Network. We create several models incorporating these features and detail the results in our paper below.

2. Problem Statement

For the sake of clarity, we will explain the mathematics behind the image classification problem in this section. Let x be our input image, where $x_{i,j,k}$ is the value corresponding to the k'th color channel (using the RGB standard) of the pixel in the i'th row and j'th column. Our task is to build a function f (we will be using CNNs to create f in this paper) such that when we input x into f, we get $p = f(x)$ where p_c represents the estimated probability that x belongs to class c.

Now we need to have a way to evaluate how well our function f performs. A standard metric of success is accuracy, but the competition led by State Farm uses Cross Entropy Loss as a metric, so we will be reporting on both here.

Accuracy is exactly what you would expect from the name - it is simply the fraction of images that we guessed correctly. Mathematically, if we let N be the total number of images and y_i be the correct class for input image x_i for every i, then the mathematical formula for accuracy is

$$\frac{1}{N}\sum_{i=1}^{N} 1[y_i = argmax_c(f(x_i)_c)]$$

Which represents the fraction of images that f correctly classified like we wanted.

Cross Entropy Loss is a bit more complecated than accuracy. Instead of only taking into account whether f was right or not, cross entropy loss attempts to incorporate how confident f is in its determination (since a model that is more confident in its correct predictions is generally better than a model that is never confident about the predictions it gets right, or worse, one that is confident in the predictions it gets wrong). To accomplish this, cross entropy loss uses the following formula:

$$-\frac{1}{N}\sum_{i=1}^{N} log(f(x_i)_{y_i})$$

The properties of log and the fact that we are passing a probability into it mean that each term in the summation will be a negative number where the closer we are to 0, the closer our original probability was to 1 (and since the probability we used as the input to the log is the probability of the correct class, we want this probability as close to 1 as we can get it), which means that we want our result to be as close to 0 as possible. The negation then flips the result so that every number is positive, but still leaves us wanting results close to 0. This means that every cross entropy value we get will be positive, and we want it as low as possible, which will be important context to remember when we are analyzing our results.

3. Technical Content

In this section, we will detail all of the elements that went into our models. Note that I am assuming that the reader already has an understanding of standard neural networks, and will be relying on that knowledge in this section.

3.1. Convolutional Layers

The Convolutional Layer is the main part of a Convolutional Neural Network. It is designed to extract the same features from different locations in the image so that the model can better understand each part of the image in relation to every other location on the image. Mathematically, if we have an input x of size $BxHxWxD_1$ where B is the batch size, H is the height of the input, W is the width, and D_1 is the depth of the input (D_1 will be 3 for a raw image, but may be different for future layers), and we want an output y of size $BxHxWxD_2$ (where B, H, and W are the same as before, but D_2 may be a new depth), then we will need a weight vector w of size $D_2xS_HxS_WxD_1$ and a bias vector b of size D_2 where S_H and S_W are the height and width fields of view for this layer, which represent how much of the image we look at for this layer. Putting these together, the formula for the convolutional layer is:

$$y_{b,i,j,d_2} = b_{d_2} + \sum_{d_1=1}^{D_1} \sum_{s_h=-\frac{S_H-1}{2}}^{\frac{S_H-1}{2}} \sum_{s_w=-\frac{S_W-1}{2}}^{\frac{S_W-1}{2}} w_{d_2,s_h,s_w,d_1} * x_{b,i+s_h,j+s_w,d_1}$$

Intuitively, this formula runs the weight matrix w over the length and width of the input and returns the weighted sum of the values it sees. Note that in our implementation we zero pad the edges of the input to allow us to maintain the same height and width before and after the convolution, and we use $S_H = S_W = 3$ for our normal convolutional layers (with one exception we will explain later on). There are many other ways to implement these layers, but since these are the ones we use, we will leave the other implementations to other papers.

3.2. Non-Linearities

If all we did was concatenate one convolutional layer after another, then the result would be little different than just a single layer with a larger field of view, since the model would still be linear like a single layer. The classic solution to this problem is to introduce a non-linearity after each layer to prevent this issue from coming up. The non-linearity we use is the Rectified Linear Unit (ReLU), which simply takes every negative feature value and sets it to 0. Mathematically, if we had an input x and output y as in section 3.1 (with $D_1 = D_2$), then the formula for a ReLU layer would be:

$$y_{b,i,j,d} = max(0, x_{b,i,j,d})$$

This is a standard non-linearity to use, since it is fast to run and it disrupts the linear nature of the convolutions enough to prevent the stacked convolutions from essentially being a single convolution, which is all we need it to do. There are other non-linearities we could use for this purpose (eg. sigmoid, tanh, leaky ReLU, etc.), but we only use ReLU in this paper, so that is all we will detail.

3.3. Pooling

At some points within the model, we will want to reduce the height and width of the input activations to better consolodate the information in our activation values. Since none of the other layers change the height and width, we need to create a new layer to do so. One standard layer that does this is a pooling layer. This layer takes in a group of pixels in the input (a 2x2 square in our case) and combines them together into a single pixel in the output (by taking the max of each color channel in our case). Mathematically, using the input x from above and a y of size $Bx\frac{H}{2}x\frac{W}{2}xD_2$ (with $D_1 = D_2$ again), then a pooling layer uses the formula

$$y_{b,i,j,d} = max_{k,l\in\{0,1\}}(x_{b,2*i+k,2*j+l,d})$$

While you can use different size groups and use a different function to combine the inputs (eg. min, average, etc.), we only used 2x2 max pooling, so that is all we will go over in this section.

3.4. Dropout

One of the main problems in building computer vision models is over-fitting, which is when the model learns too much from the training examples, and while it works really well for classifying the training examples, it doesn't generalize to outside examples. One way to counteract this is regularization, which is designed to distort the model enough so that it doesn't learn too much from the training data. We used Dropout for our regularization, which takes each activation value at a layer, and randomly decides to set each one to 0 (with probability $1 - p$) or leave it like it is (with probability p). Mathematically, if we used the input x and output y from above (again with $D_1 = D_2$), then dropout would generate a mask m of uniformly random numbers in [0,1] of the same size as x and y, and would then use this formula to set y:

$$y_{b,i,j,d} = 1[m_{b,i,j,d} < p] * x_{b,i,j,d}$$

For our purposes, we place a Dropout layer after the ReLU non-linearity in each layer, and we use a p of $p = 0.5$. There are other ways one can implement this, and other regularization methods that one can use, but as before, since we aren't using them, we won't detail them here.

3.5. Batch and Spatial Normalization

Like any values that are derived from random sources (the images we are presented with come from the distribution of all possible images of the kind we are dealing with, and so can be considered random), the feature activations in each convolutional layer will come from a distribution with its own mean and variance. Unfortunately, the next layer is always a ReLU layer, and since the ReLU layer can't learn from the training examples (since it has no parameters), the distribution that the convolutional layer outputs may not be the best distribution to use when combined with the ReLU layer. Batch and Spatial normalization [6] are designed to solve this problem. The first step in Normalization is to normalize each feature over the other examples in the batch (this is Batch Normalization) as well as the other values of the same feature from all the other locations in the input (this is Spatial Normalization). We do this by subtracting out the mean of each feature and dividing by the variance. This means that each feature will now have mean 0 and variance 1, but that still might not be the right distribution. In order to fix this, the second step is to use learnable parameters as the new mean and variance. This gives each feature the best distribution the model can learn when it heads into the ReLU layer.

Mathematically, if we use the input x and output y as above (again with $D_1 = D_2 = D$), then we will create learnable parameters m and v of size D each, a small constant ϵ to prevent divide-by zero errors, and use the following formulas to get our output:

$$\mu_d = \frac{1}{B*H*W} * \sum_{b,i,j=1}^{B,H,W} x_{b,i,j,d}$$

$$\sigma_d^2 = \frac{1}{B*H*W} * \sum_{b,i,j=1}^{B,H,W} (x_{b,i,j,d} - \mu_d)^2$$

$$\hat{x}_{b,i,j,d} = \frac{x_{b,i,j,d} - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}}$$

$$y_{b,i,j,d} = v_d * \hat{x}_{b,i,j,d} + m_d$$

While this technique is relatively new (only developed in 2015), it has still been shown to be useful in a variety of situations, which is why we incorporate it into our models.

3.6. Full Pooling Layer

One technique we incorporate from Google's Inception Network [4] is the use of a Full Pooling Layer between the convolutional and the fully connected parts of the architecture. One issue in normal networks is that there are usually many elements in the last layer of convolutions, and by connecting them all to a fully connected layer we are creating a lot of parameters that can cause us to over-fit our training data (Note that depending on the architecture, this single layer can be the source of most of the parameters in the model). Google's Inception Network attempts to solve this by instead of using a fully connected layer to reduce the dimensions of the activations, they pool the input across the entire height and width of the input instead. There is one main difference between full pooling and the 2x2 pooling we mentioned above: 2x2 pooling only pools a small section of the image and leaves it with height and width dimensions, and full pooling completely removes the height and with dimensions by pooling over every spacial coordinate. As in 2x2 pooling above, we use max as our pooling function. Since a pooling layer doesn't have any parameters, this prevents the enormous amount of parameters from being generated in that layer, which should help prevent over-fitting.

Mathematically, if we use the input x as above and and output y of size BxD, then the formula for Full pooling will be:

$$y_{b,d} = max_{i \in [0,H-1], j \in [0:W-1]}(x_{b,i,j,d})$$

As in normal pooling, we can use functions aside from max (eg. min, average, etc.), but we use max in our architectures, so that is what we detail here.

3.7. Split Convolutional Layers

The other technique we incorporate from Google's Inception Network [4] is splitting each convolutional layer into two seperate layers. We do this by instead of having a single convolutional layer with a 3x3 field of view (that is, $S_H = S_W = 3$), we have one convolutional layer with a 3x1 field of view, and after that we attach one convolutional layer with a 1x3 convolutional field of view. The reasoning behind this is that each neuron in the second layer has the same field of view as before (3x3), but we now have more non-linearities (which is generally good because it allows us to move away from a strictly linear model), and fewer parameters (which helps with over-fitting). This means that this technique should help create a more accurate model, and since it worked for Google, I will be incorporating it into my model as well.

3.8. Overall Models

I will need to set up a notation to describe the architectures I built concisely. I will be detailing an archtecture by writing a string like this: "Conv8 - Poolx2 - SplitConv16 - FullPool - FC32". Here is how I am describing each layer in my notation:

1. Conv<D> - This represents a sequence of 4 layers - a single 3x3 convolutional layer of depth D, a batch normalization layer, a ReLU layer, and finally a Dropout layer.
2. SplitConv<D> - This represents a sequence of 8 layers - the first 4 are the same sequence of layers as in a Conv layer, but the convolutional layer has a field of view of 3x1 instead of 3x3. The last 4 are also the same sequence of layers as in a Conv layer, but the convolutional layer has a field of view of 1x3 instead of 3x3.
3. Pool - This represents a 2x2 pooling layer.
4. FullPool - This represents a full pooling layer.
5. FC<D> - This represents a fully-connected layer of size D.

Additionally, if there is an x<C> after a layer, then that means that I used C copies of that layer in a row. There is also a single fully connected layer at the end of every model that generates the class probabilities, but since that is the same in every model I won't be including it in my notation. This should allow you to understand the architectures behind my models, so I will now detail my experiments.

4. Experimental Setup and Results

4.1. Dataset

State Farm provided the training and test data in their competition listing on Kaggle [2], and that is where I got all of my data. The test set contains 79726 images without their corresponding class labels, and the training set contains 22424 images with their class labels. It is important to note that State Farm specifically seperated out the drivers in the training and test sets so that no driver has an image in both the training and test sets. Because of this, I decided to experiment with how much of an effect training and validating on the same drivers effected the reported performance of the model. To do this, I came up with two ways to seperate the provided training set into training and validation sets.

The first method is the way to get accurate results from the validation set - namely splitting drivers between the training and validation sets. To do this, I randomly ordered the drivers and included each driver in order in the training set until I had at least 80% of the examples in the training set, and then I put the rest of the drivers in the validation set. Due to the fact that the number of images per driver varied wildly, I wasn't able to get an exact 80-20 split between training and validation, but I was able to get close, with 18158 training images and 4266 validation images for a 80.9757% training split.

The second method is the one that will supposedly give you biased results - mixing the images from a single driver between the training and validation sets (note that I am not mixing a single image between the training and validation sets - I am only allowing different images from the same driver to be in different sets). This method is much easier to split, because all I have to do is randomly order all of the images and put the first 80% in the training set. Running this split give me 17939 training examples and 4485 validation images for a 79.9991% training split (you can't get exactly 80% because 22424 isn't divisible by 5).

Note that I did make sure to split the images the into the same sets for each method and architecture, so no architecture had an advantage over the others. I did this by setting the random seed to a constant before doing the split (the constant being 42 of course :)) and resetting the random seed to the default after doing the split.

4.2. Architectures

The 6 architectures I used in my experiments are detailed in Table 1. Note that Model 1 was used to calibrate the training regimen (eg. batch size) for my hardware and was not used in the experiments.

4.3. Hardware

All experiments were run on an MSI Dominator Pro laptop, with an NVIDIA GTX 980M graphics card, 32GB of

Model Number	Architecture
2	Pool - Conv16 - Pool - Conv32 - Pool - FC64
3	Pool - Conv16 - Pool - Conv32 - Poolx3 - FC64
4	Pool - Conv16 - Pool - Conv32 - FullPool - FC32
5	Poolx2 - Conv8 - Pool - Conv16 - Poolx2 - FC32
6	Pool - SplitConv16 - Pool - SplitConv32 - Pool - FC64
7	Pool - SplitConv8x2 - Pool - SplitConv16x2 - Pool - FC32x2

Table 1. Architectures used in experiments

RAM, and an Intel i7-4980HQ CPU. I also implemented all of my models using Google's Tensorflow framework [7].

4.4. Training Regimen

In order to train the models, I used Cross Entropy Loss for the loss, a batch size of 50 images, 2000 batches for each model, and an Adam update rule [8] with learning rate of 0.0001. Most of these are standard for neural network classification challenges, but the Adam update is relatively new. Adam was chosen because it is designed to allow for minimal adjustment of the learning rate, which allows me to focus on improving the model as opposed to adjusting the learning rate parameter.

4.5. Results

The Training and Validation Accuracy and Cross Entropy Loss are detailed in Tables 2 and 3. Since Model 3 gives us the best validation accuracy when we split the drivers between the training and validation sets properly, that is the model we will evaluate on the test set by submitting its test set predictions (for both training methods) to Kaggle. The results are included in Table 4.

The loss graphs for model 3 under these two training regimens are also included as Figure 1. The loss graphs for the other models look similar with few exceptions, so I didn't include them in order to save space in the paper.

These are the results that I was able to obtain from my experiments, so I will now analyze them in the next section.

5. Conclusions

5.1. Analysis of Results

We first note that both loss graphs have almost exactly the shape that we want - curving down sharply and then flattening out, with a small enough variation in the loss for it to not affect the performance of the model. This tells us that both models were well trained (as were the other models, since most of them were basically the same shape.), which means that the results we have correspond to the actual performance of the models, and aren't artificially worsened by poor training practices. This means that we can continue to analyze the remaining results.

Looking at the model 3 validation and test accuracies gives us some interesting results. The validation loss from splitting the drivers into the train and validation sets gives us a much more accurate loss value (by an order of magnitude) than sharing the drivers between the train and validation sets, but sharing the drivers between the train and validation sets gives a better test loss than splitting them up. The more accurate validation loss from splitting up the drivers is to be expected - State Farm purposefully split the drivers up between the original test and training sets, and the only reason they would think to do that is because they have seen results like these in their own work on the subject. Getting better test performance by sharing the drivers is unexpected though - considering that splitting the drivers gives a more accurate validation loss, and since splitting the drivers even results in slightly more training examples (due to the number of images per driver not dividing up evenly), one would think that splitting the drivers would also give a better test performance. In fact, the only advantage that sharing the drivers gives the model (since the drivers aren't shared with the test set) is that the model has seen more drivers in general. This is the key though - by simply seeing more examples of different drivers, the model that was shown the Shared Drivers training data was able to increase its performance by almost 10%. This means that the more drivers we are able to show our model, the better it will perform. We will detail ways we can accomplish this in the Future Work section.

When we compare the results of all the models between the Splitting and Sharing Drivers datasets, they mostly tell the same basic story as when we looked at just model 3. The training performance is approximately the same, but the validation performance when we are sharing drivers is much greater (at training performance levels) than when we are splitting the drivers. This difference in performance is by more than an order of magnitude for some of the models. This tells us that not only does sharing the drivers give us inaccurate information, but the information is usually off by extreme amounts, not by small differences. The difference is also present between the training and validation performance when we split the drivers - again varying by orders of magnitude in some cases. This tells us that we are over-

Model Number	Train Accuracy	Train Loss	Val Accuracy	Val Loss
2	0.595991	1.710525	0.287623	3.460873
3	0.992510	0.075897	0.651899	1.075854
4	0.139057	2.278056	0.129864	2.306801
5	0.976870	0.278310	0.601969	1.288893
6	0.991684	0.094351	0.444679	1.867791
7	0.804494	0.945188	0.423347	1.896609

Table 2. Results of Training by Splitting Drivers

Model Number	Train Accuracy	Train Loss	Val Accuracy	Val Loss
2	0.999610	0.019854	0.997324	0.033223
3	0.990969	0.094291	0.985730	0.110870
4	0.127320	2.294936	0.122854	2.293173
5	0.960198	0.362087	0.956522	0.382223
6	0.994091	0.058322	0.989521	0.076038
7	0.706282	1.325765	0.705017	1.327055

Table 3. Results of Training by Sharing Drivers

fitting the training set by a large margin with our present models, and we need to make adjustments to our models in order to correct that issue (this was what I attempted to do with the later models, but it didn't help like I had hoped). I will detail possible improvements in the Future Work section.

Comparing the performance of the various models also gives us interesting results. We see that adding techniques from Google's Inception Network doesn't seem to have helped improve the performance of the model, even though the changes were intended to reduce over-fitting, which seems to be the main problem with our current models. The only model that was even close to the performance of model 3 was model 5, and the only differences between those models are that one of the final pool layers in 3 is moved to the beginning in 5, and 5 has the number of channels in each layer halved with respect to 3. The fact that this doesn't seem to improve the performance of the model, or even reduce the amount of over-fitting (since the training and validation loss went up by approximatly the same amount) is important, because reducing the number of parameters and activation values in the model (which is what all the changes from 3 to 5 are intended to do) is one of the primary ways to reduce over-fitting, but it didn't work in this case. The inclusion of the FullPool layer in 4 was also intended to reduce over-fitting in this manner, but it ended up choking the model so much that the performance ended up little better than random. This tells us that while reducing the number of parameters and activation values in the model may work in other cases, it doesn't seem to be working for this problem. This indicates which directions not to go in for improving the model, which we take into consideration in the Future Work section.

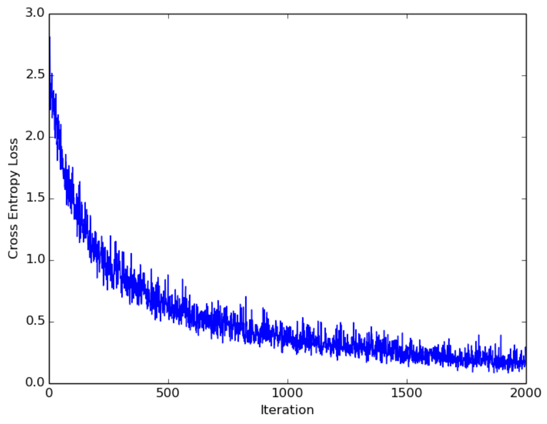
5.2. Future Work

We mentioned above that we need to include more drivers in our training set in order to improve the performance of our model. The simplest way to do this would be to simply collect more images for the training set from new drivers, but this is generally expensive and time consuming, so we would like to find a better way. One way to train on more drivers would be to proceed as normal by using the split drivers dataset to train and validate our models, but when we have finally decided on the final model we want to use, we take the final architecture that gives us the best validation results and re-train it on the entire training dataset, without using a validation set. This would mean that our model is being trained on all of the drivers that we have, which should allow it to achieve more accurate predictions. Another, much more involved way to train on more drivers would be to incorporate unsupervised learning on the test set (eg. training an auto-encoder for use as a feature extractor) and include that in our model. The competition doesn't allow for a human to go through the test data set, but it doesn't explicitly prohibit using unsupervised learning on it, so this technique should be legal for use in the competition.

We also noticed above that our primary issue with getting good performance from our models is over-fitting. My first attempt at fixing this issue was to reduce the expressiveness of my model by reducing the number of parameters and activation values in the model, but this turned out not to work very well. There are other methods to reduce over-fitting though. The most straight forward method is to increase

Model Number	Training Method	Test Loss
3	Split Drivers	1.71393
3	Share Drivers	1.57714

Table 4. Results of Testing on Model 3



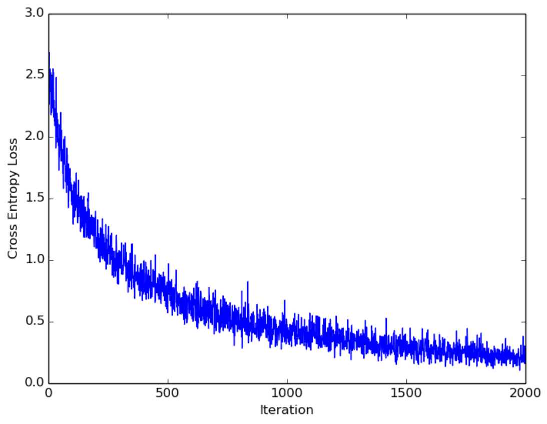


Figure 1. Left: Loss Graph of Model 3 with Split Drivers - Right: Loss Graph of Model 3 with Shared Drivers

the regularization on the model (since regularization is explicitly designed to combat over-fitting), which can either mean increasing the dropout regularization (by lowering the probability we keep an activation value), adding in another type of regularization (eg. regularizing on the magnitude of the weight matricies), or some of both. A more advanced way to combat over-fitting would be to use transfer learning. We could take a pre-trained network (eg. AlexNet, VGGNet, Google's Inception Network, Microsoft's Residual Network, etc.), use some of the layers from that network as a feature extractor, and train our network on top of that feature extractor to output the final probabilities. This would help because we know that the feature extractor is already good for most purposes (especially if we train on the ImageNet challenge winners like Google's Inception Network and Microsoft's Residual Network), and we can't overfit it to our dataset because we don't change it during the training process. Additionally, since we just need to train a small neural network on top of it, the reduced size of the network should help prevent over-fitting as well (this might not help as much as expected considering our earlier results, but it should still help some).

This gives us several directions to go in order to improve our models - from training an auto-encoder on the test set for use as a feature extractor, to taking advantage of past highly performing networks. This means that there is plenty of room for improvement in our models, and our performance can only improve as we keep trying new techniques.

6. Code

The following page should give you access to my code on github - https://github.com/kggriswold/CS231ProjectSubmission.git If you have trouble downloading it, just send me an email.

7. References

[1] Distraction.gov, http://www.distraction.gov/stats-research-laws/facts-and-statistics.html

[2] Kaggle State Farm Distracted Driver Detection Competition. https://www.kaggle.com/c/state-farm-distracted-driver-detection

[3] ImageNet Challenge. http://www.image-net.org/

[4] Google Inception Network v4. https://static.googleusercontent.com/media/ research.google.com/en//pubs/archive/45169.pdf

[5] Microsoft Residual Network. http://arxiv.org/abs/1512.03385

[6] Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, http://arxiv.org/pdf/1502.03167v3.pdf

[7] TensorFlow. https://www.tensorflow.org/

[8] Adam: A Method for Stochastic Optimization, http://arxiv.org/pdf/1412.6980.pdf