Capstone Project

Machine Learning Engineer Nanodegree

Nicolás Alvarez

August 22nd, 2016

I. Definition

Project Overview

According to the CDC motor vehicle safety division, one in five car accidents[1] is caused by a distracted driver. Sadly, this translates to 425,000 people injured and 3,000 people killed by distracted driving every year in USA.

This project is based on a Kaggle[2] competition called "State Farm Distracted Driver Detection"[3]. Given a dataset of 2D dashboard camera images, I developed a ConvNet that is capable to classify each driver's behavior, that is, if they are driving attentively or if they are distracted, for example, taking a selfie with their friends in the backseat.

Problem Statement

The project's goal is to predict the likelihood of what the driver is doing in each picture. The main tasks involved are the following:

1. Download and preprocess the train, validation and test datasets.
2. Build and train the ConvNet classifier that can determine what the driver is doing in

1 http://www.cdc.gov/motorvehiclesafety/distracted_driving/
2 https://www.kaggle.com
3 https://www.kaggle.com/c/state-farm-distracted-driver-detection

each picture.

3. Measure performance on validation set and fine tune the model if necessary.
4. Predict images in the test dataset and submit the predictions file to the Kaggle competition.

Metrics

Two metrics are used to measure the performance of the ConvNet classifier: **Accuracy** and **Multi-class logarithmic loss**.

The first one, which is the proportion of correct results that the classifier achieved, is given by:

$$Accuracy = \frac{Correct\ Predictions}{Number\ of\ Data\ Points}$$

As is mentioned in the Exploratory Visualization section, the dataset is balanced between classes, so the accuracy paradox is not a problem in this case and no additional metrics, as precision or recall, are required to evaluate this model.

The multi-class logarithmic loss, the one used in Kaggle's competitions, is given by:

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

where ***N*** is the number of images in the test/validation set, ***M*** is the number of image class labels, log is the natural logarithm, ***y_ij*** is 1 if observation i belongs to class j and 0 otherwise, and ***p_ij*** is the predicted probability that observation i belongs to class j.

II. Analysis

Data Exploration

The data[4] provided by the competition consists of the following files:

- **imgs.zip:** zipped folder of all (train/test) images
- **sample_submission.csv:** sample submission file in the correct format
- **driver_imgs_list.csv:** list of training images, their subject (driver) id, and class id

The examples are driver images of 640px x 480px, each taken in a car with a driver doing something in the car (texting, eating, talking on the phone, makeup, reaching behind, etc).

The 10 classes to predict are:

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

- **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 train and test data are split on the drivers, such that one driver can only appear on either train or test set.

The train dataset has 22423 labeled images and the test dataset has 79726 unlabeled examples. The train data was split in two subsets, one of 17937 examples for training and another one of 4485 examples for validation.

Exploratory Visualization

The Figure 1 shows a bar diagram of the number of occurrences of each class. From a total of 22424 example in the training set, each class has between 2000 and 2500 examples. Despite it is a small dataset, it is balanced between classes.

On the other hand, in Figure 2 is shown the examples distribution in function of the drivers. Images of the dataset come from just 26 drivers, having approximately 80 images per every class per every driver. This is a very important thing to be considerer when splitting the dataset in training and validation sets for avoiding overfitting.

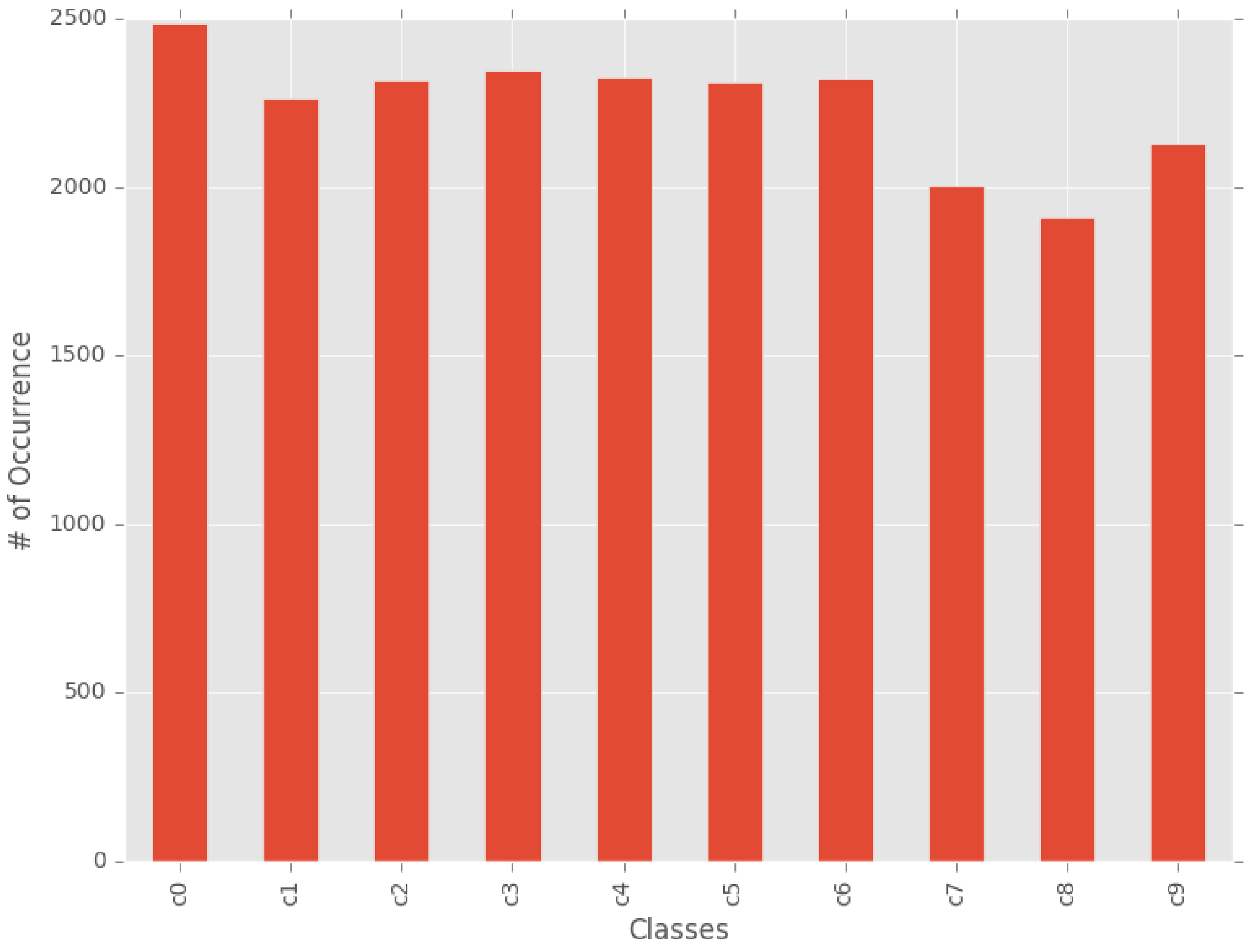


Figure 1: Occurrences per Class

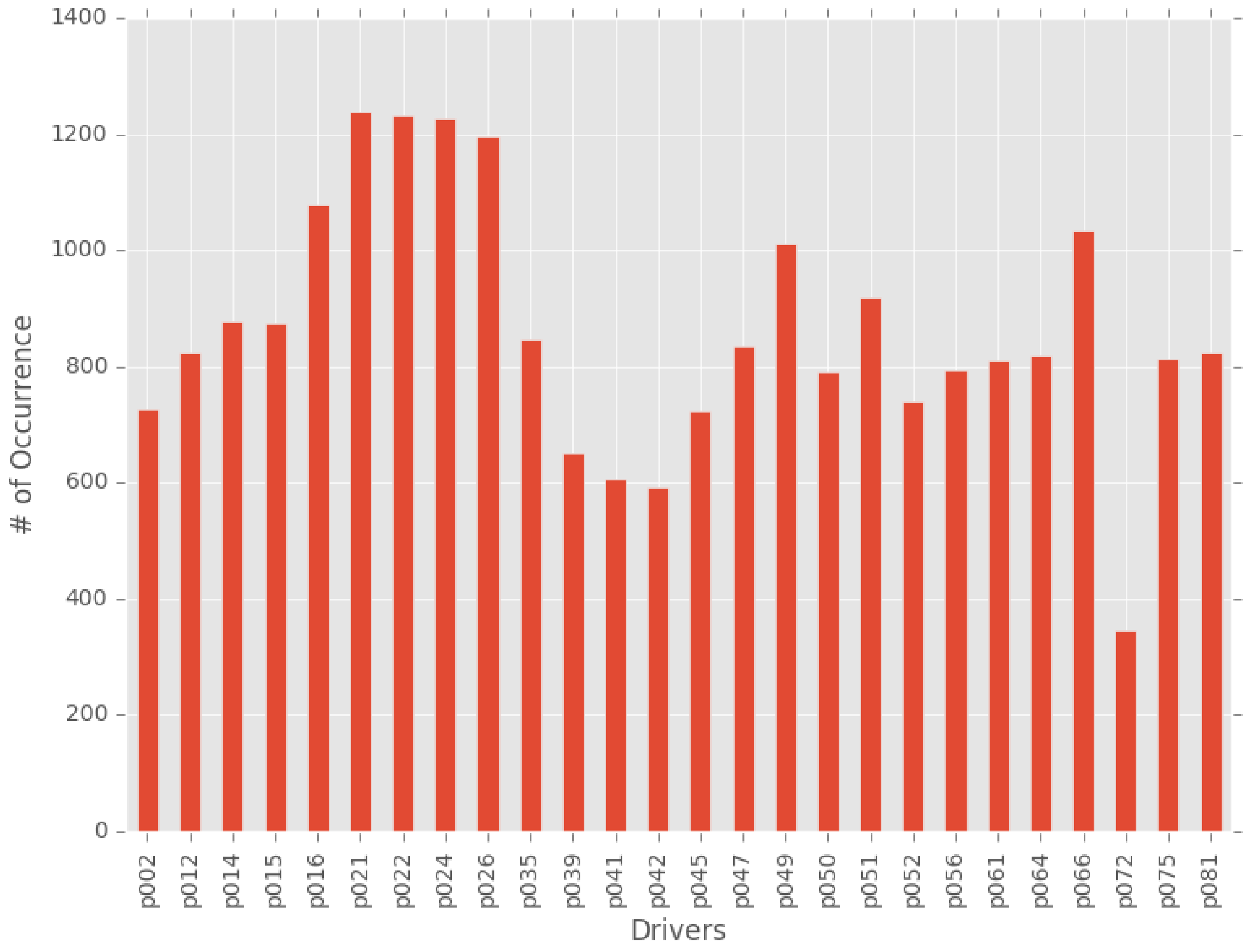


Figure 2: Occurrences per Driver

Algorithms and Techniques

Nowadays, Convolutional Networks[5][6], also known as ConvNets, or some close variant are used in most neural networks for image recognition. Making the explicit assumption that the inputs are images, these networks use a special architecture which is particularly well-adapted to classify images. This architecture vastly reduces the amount of parameters in the network, making ConvNets fast to train. Also ConvNets outputs an assigned probability for each class, as it is required by the competition rules.

The layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. As shown in Figure 3, the neurons in a layer are only connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner.

5 http://cs231n.github.io/convolutional-networks/
6 http://neuralnetworksanddeeplearning.com/chap6.html

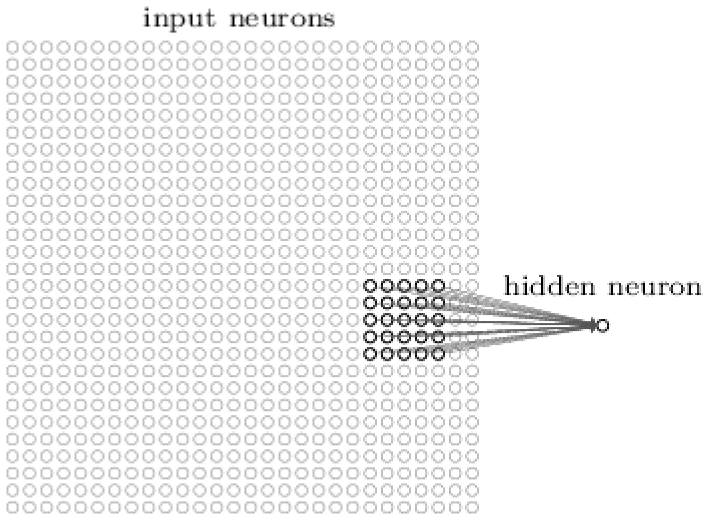


Figure 3: Hidden neuron in a ConvNet

Generally speaking, a ConvNet is a sequence of layers (Convolutional Layer, Pooling Layer and Fully-Connected Layer), and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores.

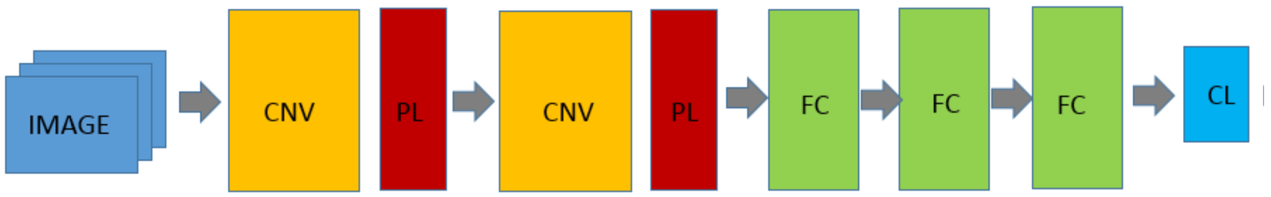


Figure 4: ConvNet Architecture Example

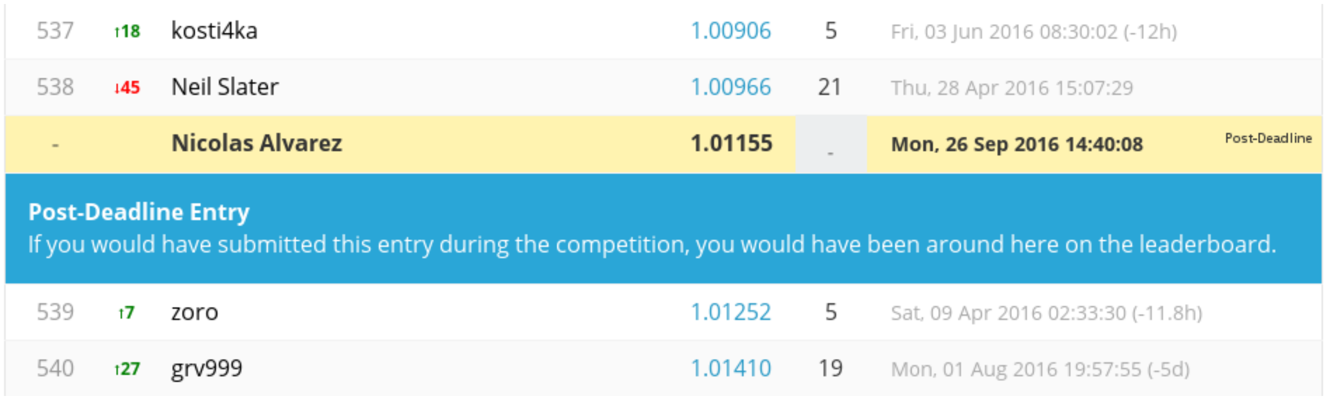
The following parameters can be tuned to optimize the ConvNet to be implemented:

- **Training parameters**
  - Number of epochs
  - Batch size
  - Optimization algorithms for learning, such as Gradient Descent or Adagrad
  - Learning rate (initial value and rate decay)
  - Dropout probability (for avoiding overfitting)
- **Neural network architecture**
  - Number of layers

    - Layer types (convolutional, fully-connected, or pooling)
    - Layer parameters (no. of neurons, bias and weight initialization)
- **Preprocessing parameters** (see the Data Preprocessing section)

Benchmark

The Private Leaderboard[7] of the State Farm Distracted Driver Detection Competition, which is computed on 75% of the test set, will be used as benchmark for this project. The goal is to arrive to a solution that is in the top half of the table in the competition. Because the competition has already finished, the Leaderboard does not change with time.



537	↑18	kosti4ka	1.00906	5	Fri, 03 Jun 2016 08:30:02 (-12h)	
538	↓45	Neil Slater	1.00966	21	Thu, 28 Apr 2016 15:07:29	
-		**Nicolas Alvarez**	**1.01155**	-	**Mon, 26 Sep 2016 14:40:08**	Post-Deadline

Post-Deadline Entry
If you would have submitted this entry during the competition, you would have been around here on the leaderboard.

539	↑7	zoro	1.01252	5	Sat, 09 Apr 2016 02:33:30 (-11.8h)
540	↑27	grv999	1.01410	19	Mon, 01 Aug 2016 19:57:55 (-5d)

Figure 5: LeaderBoard Entry Example

On the other hand, to use something more concrete and objective, it is expected that the model reaches an accuracy of 80%.

III. Methodology

Data Preprocessing

The training dataset is not large enough for training the model, so some distortions are apply to the images. The following preprocessing is applied to the training datasets' images:

- Check that all images have size 640px by 480px.
- Apply random distortions (Gaussian Blur, Unsharp Mask, Median Filter, Min Filter, Max Filter) to original images.
- Resize the original and distorted images from 640px by 480px to 64px by 48px.
- Save images in binary files, in batches up to 30MB per file, formatted as follows:

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

    - <1 x label><height*width*depth x pixel>
- Convert images to a squared form adding zero pixels as padding to a size of 1.2 the width of the image.
- Randomly crop the image to a size of 64px by 48px.
- In a random order, apply the following distortions:
    - Randomly adjust the brightness.
    - Randomly adjust the contrast.
    - Randomly adjust the saturation.
    - Randomly adjust the hue.
- Finally, linearly scales the image to have zero mean and unit norm.

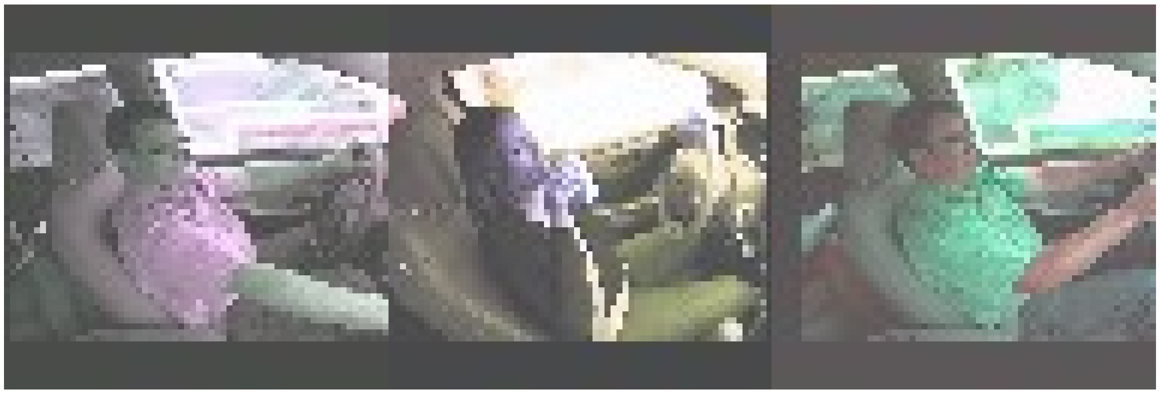


Figure 6: Processed images examples

Implementation

The model, called Distracted Driver Detection Model (DDDM), was implemented in Python using the TensorFlow[8] library. It is an open source software library for numerical computation using data flow graphs. TensorFlow was originally developed for the purposes of conducting machine learning and deep neural networks research, but the system is general enough to be applicable in a wide variety of other domains as well.

Before starting to code the model in TensorFlow, one difficult question to answer is: Do I use TensorFlow or tf.contrib.learn, also known as skflow and TF.Learn. Referencing the tf.contrib.learn documentation:

8 https://www.tensorflow.org/

“TF.Learn is a library that wraps a lot of new APIs by TensorFlow with nice and familiar Scikit Learn API.

TensorFlow is all about a building and executing graph. This is a very powerful concept, but it is also cumbersome to start with.”

The two sentences above are really true. TF.learn is very similar to sklearn, with just few lines of code, it is possible to load and preprocess the data, define a model, train and validate it. In just a few hours, the job is completely done, it is something “magical”. But it has some disadvantages:

- It is not very well documented (yet, I hope).
- Using just TF.learn means that you will not know anything about TensorFlow, being unable to take advantage of all that powerful staff that the library offers.

In several blogs talk about TF.learn as a transition from sklearn to TensorFlow. So, I decided learning TensorFlow as a secondary objective for this project. Of course, it wasn't free, it took me a lot of work and time learn it, as the documentation says, *it is cumbersome to start with.* I could say that almost every complication I had during the coding process was because I didn't know TensorFlow, but there was nothing that could not be resolved reading the documentation and making some little example programs.

Going on, the model's code is organized as shown in the following table:

File	**Purpose**
DDDM.py	DDDM ConvNet model.
DDDM_data_exploration.py	Routine for DDDM data exploration
DDDM_input.py	Routine for convert and decode the DDDM dataset.
DDDM_train.py	Routine for training the DDDM.
DDDM_val.py	Routine for validating the DDDM.
DDDM_test.py	Routine for testing the DDDM.
DDDM_retrain_test.py	Routine for testing the retrained Inception V3 model.

Model Inputs

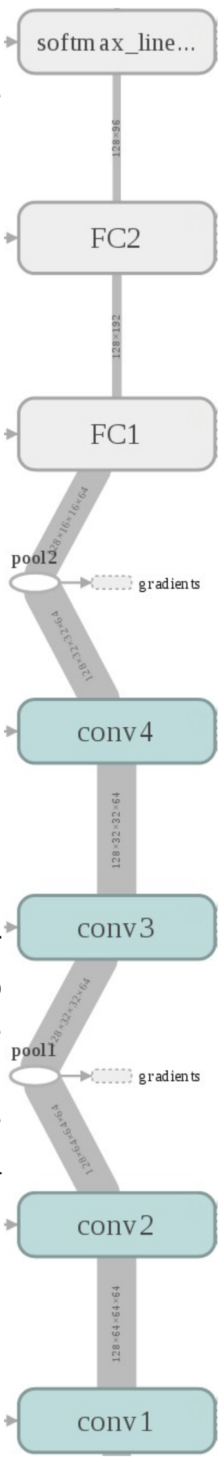
The input part of the model (DDDM_input.py) is built by the functions “inputs()” and “distorted_inputs()”, which read the batch binary data files. If these files do not exists, they are automatically generated from the original dataset.

The "distorted_input()", which perform the distortion mentioned in Data Preprocessing Section, is used for the training of the model, while that "input()" function is used for validate the model.

Model Training

The prediction part of the model is constructed by the inference() function, implemented in DDDM.py, which adds operations to compute the logits of the predictions. The ConvNet hast the following architecture:

Layer Name	Description
conv1	Convolution layer with 64 filters of 5x5; and rectified linear activation.
conv2	Convolution layer with 64 filters of 5x5; and rectified linear activation.
pool1	Max Pooling layer.
conv3	Convolution layer with 64 filters of 5x5; and rectified linear activation.
conv4	Convolution layer with 64 filters of 5x5; and rectified linear activation.
pool2	Max Pooling layer.
FC1	Fully Connected Layer with 192 neurons.
FC2	Fully Connected Layer with 96 neurons.
softmax_linear	Linear transformation to produce logits.



Multinomial logistic regression, also known as Softmax regression, is used for training the ConvNet. The Softmax regression applies a softmax nonlinearity to the output of the network and calculates the cross-entropy between the normalized predictions and a 1-hot encoding of the label. For regularization, dropout is applied in the layers FC1 and FC2 with a drop probability of 0.5. The DDDM model is trained with the standard gradient descent algorithm with a learning rate that exponentially decays over time.

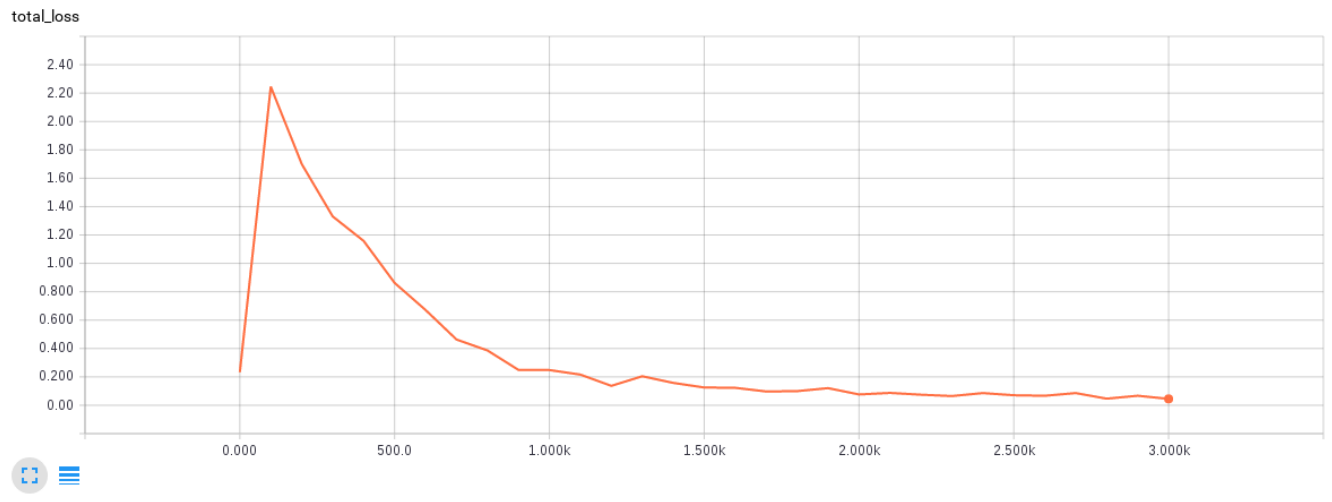


Figure 7: Training Cross Entropy Loss

For train the model, just is needed to execute the script DDDM_train.py.

Model Validation

Every 1000 training steps, the model is validated running the script DDDM_val.py. This script uses the 20% of the dataset, taken images of six drivers from a total of twenty six, for compute the model accuracy and the multi-class logarithmic loss with and without Laplace smoothing.

Model Testing

With the model trained, it can be tested against the test dataset running the script DDDM_test.py. Basically, this script estimates the probability of every class for every of the 79726 images in the test set and generate the submission file for the Kaggle competition. Once generated, the submission file could be uploaded to Kaggle to check the rank position in the competition.

Model Visualization

The model periodically exports summaries that may be visualized in TensorBoard. To run TensorBoard, execute the following command:

```
$ tensorboard --logdir=model_path
Starting TensorBoard 22 on port 6006
(You can navigate to http://0.0.0.0:6006)
```

Then load the provided URL (here, http://0.0.0.0:6006) in your browser.

Refinement

In this section I will describe a ConvNet architecture with the following way:

[(filter_size x no_filters) CONV -> RELU -> (filter_size x no_filters)CONV -> RELU -> POOL]*2 -> [(no_neurons_per_FC)FC ->RELU]*2 -> FC

For example, the following network

[(3x3x64)CONV -> RELU -> (3x3x64)CONV -> RELU -> POOL]*2 -> [(384, 192)FC ->RELU]*2 -> FC

has two sequence of:

- 1convolution layer of 64 filters of size 3x3,
- 1 RELU,
- 1convolution layer of 64 filters of size 3x3,
- 1 RELU,
- 1 Pooling layer

and to fully connected layers with 384 and 192 neurons each, ending with the linear transformation to produce the logits.

The first ConvNet had the following architecture:

- [(3x3x64)CONV -> RELU -> (3x3x64)CONV -> RELU -> POOL]*2 -> [(384, 192)FC ->RELU]*2 -> FC → **Accuracy: 0.3**

In the best case, this ConvNet archive a accuracy of 0.30 in the validation set, and a multi-class logarithmic loss of 2.32032. This model was in the position 1302/1440 in the rank table.

I've tried multiple ConvNet architectures, getting the following accuracy values:

- [(5x5x64)CONV -> RELU -> (5x5x64)CONV -> RELU -> POOL]*2 -> [(192, 96)FC ->RELU]*2 -> FC → **Accuracy: 0.4**
- [(5x5x32)CONV -> RELU -> (5x5x32)CONV -> RELU -> POOL]*2 -> [(48)FC ->RELU]*1 -> FC → **Accuracy: 0.34**
- [(5x5x64)CONV -> RELU -> (5x5x64)CONV -> RELU -> POOL]*3 -> [(96, 192)FC ->RELU]*2 -> FC → **Accuracy: 0.37**
- [(3x3x128)CONV -> RELU -> (5x5x64)CONV -> RELU -> POOL]*3 -> [(96, 48)FC ->RELU]*2 -> FC → **Accuracy: 0.4**
- [(3x3x128)CONV -> RELU -> (3x3x64)CONV -> RELU -> POOL]*3 -> [(4096, 2048)FC

->RELU]*2 -> FC → **Accuracy: 0.375**

All this results was obtained without distorting the original images, just the resized ones. Running similar experiments but applying distortions on the original images, the ConvNet with the best performance has the following architecture:

- [(5x5x64)CONV -> RELU -> (5x5x64)CONV -> RELU -> POOL]*2 -> [(192, 96)FC ->RELU]*2 -> FC → **Accuracy: 0.44**

I've tried multiple learning rate initial values and decay rates and, using learning rates less than 0.1, the model always converges to a loss value less than 0.05.

Because I expected a better accuracy, I tried retraining a Inception V3 model[9]. This model, stored in capstone/retrained_model, is tested running the DDDM_retrain_test.py script. This model archives a accuracy of 0.80 on the validation set, 0.82 on the testing set, and a multi-class logarithmic loss of 1.01155. This model was in the position 539/1440 in the rank table, a very big improvement.

IV. Results

Model Evaluation and Validation

The final architecture of the Ad-Hoc ConvNet and hyperparameters were chosen because they performed the best in the rank table of the competition and archived one of the highest accuracy over the validation set.

As mentioned in the previous section, the final model has the following architecture:

- [(5x5x64)CONV -> RELU -> (5x5x64)CONV -> RELU -> POOL]*2 -> [(192, 96)FC ->RELU]*2 -> FC → **Accuracy: 0.44**

This model applies distortions to the original images and the the resized ones and uses a initial learning rate of 0.1 with rate decay. It archives a accuracy of 0.44 and a multi-class logarithmic loss of 1.92777. This model was in the position 1017/1440 in the rank table, a very big improvement.

On the other hand, using the Inception V3 model the results was much better reaching the position 539/1440 in the rank table, with a multi-class logarithmic loss of 1.01155.

Justification

This Kaggle competition was very complex, specially because the dataset provided is not large enough and it has few drivers (only 26). Because the dataset has more images per

9 http://arxiv.org/abs/1512.00567

driver (~600) that drivers (26), the ad-hoc ConvNets not only learned the drivers action, it learned the drivers too. So, when there was a driver in the test set much different from drivers in the training set, the ConvNet miss the prediction. On the other hand, too many classes are very similar between then, thing that produce the ConvNet to miss predictions. For example, sometimes it is very difficult to distinguish a driver that is touching its ear with another one that is talking on the phone.

Because the complexity of the problem and considering the simplicity of the Ad-Hoc ConvNet and the hardware availability (a laptop without GPU), it is possible to consider the accuracy of 0.44 as acceptable.

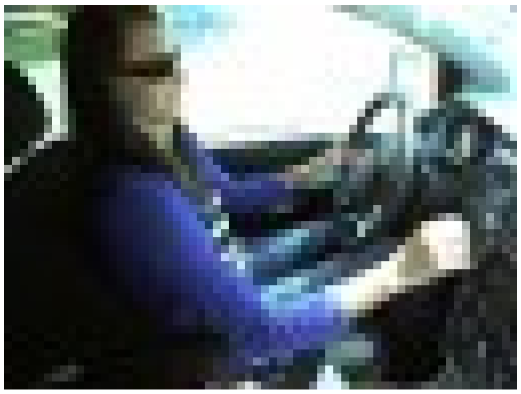
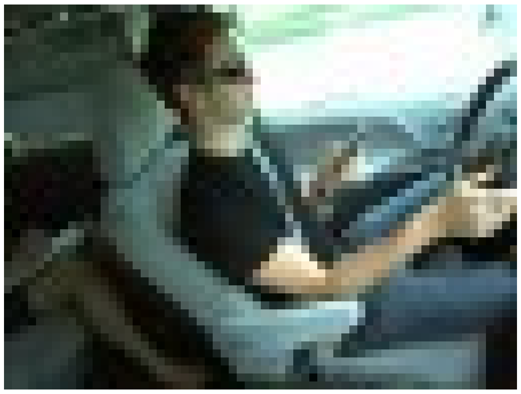
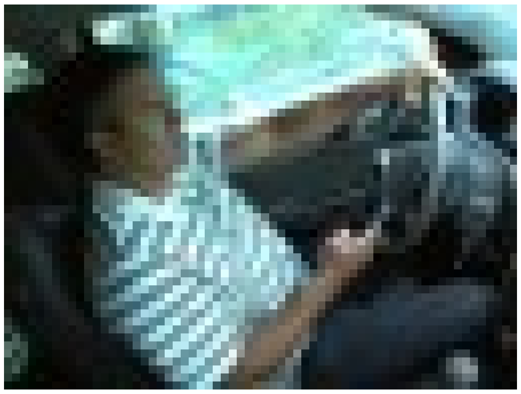
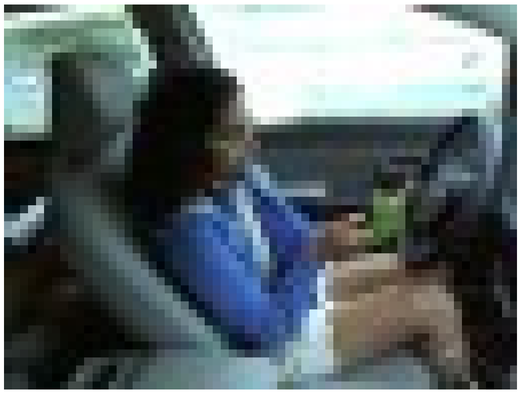
On the other hand, retraining the Inception V3 model, the performance obtained is much better, reaching a solution that accomplish the expected performance specified in the benchmark section. This approach of retrain models, is the one chosen by the the majority of the Kaggle competitors.

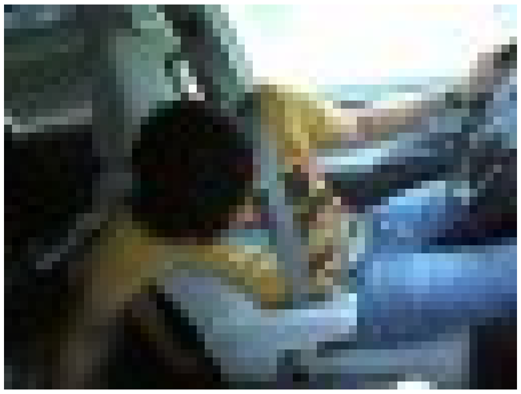
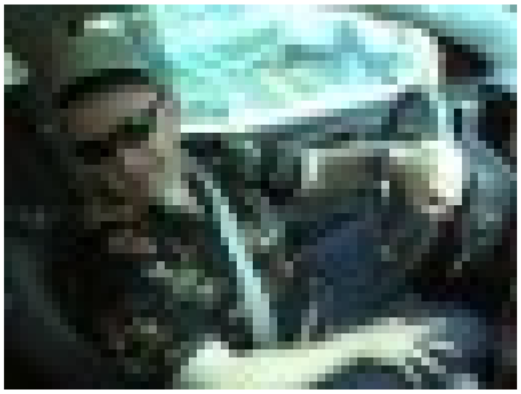
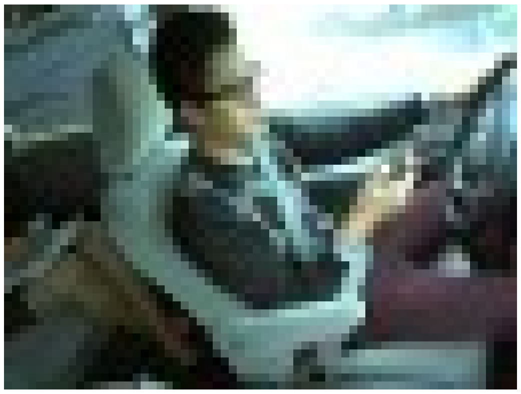
V. Conclusion

Free-Form Visualization

In the following table several specific cases are presented.

Predicted Class (probability)	**Right Class (probability)**	**Observations**	**Image**
c0: safe driving (0.99992)	c0: safe driving (0.99992)	Correct prediction with very high confidence.	

c0: safe driving (0.53428)	c5: operating the radio (0.26853)	Image incorrectly predicted. The prediction is not very confidence and the right class has the second larger probability.	
c3: texting - left (0.98143)	c3: texting - left (0.98143)	Correct prediction with very high confidence.	
c1: texting – right (0.95742)	c1: texting – right (0.95742)	Correct prediction with very high confidence despite that the image is not very clear, it's difficult to distinguish the phone.	
c6: drinking (0.70562)	c1: texting - right (0.09131)	Here the model makes a mistake, giving a very low probability to the right class. Is not easy to distinguish the cell phone from, for example, a paper glass, but I think no one takes a glass with the hand in that	

		position.	
c7: reaching behind (0.99999)	c7: reaching behind (0.99999)	Correct prediction with very high confidence.	
c9: talking to passenger (0.99999)	c9: talking to passenger (0.99999)	Correct prediction with very high confidence.	
c0: safe driving (0.65227)	c1: texting – right (0.11160)	Image incorrectly predicted. The prediction is relatively confidence and the right class has the second larger probability.	

c5: operating the radio (1.0)	c5: operating the radio (1.0)	Correct prediction with very high confidence.	

Reflection

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

1. A machine learning competition were found with an associate problem and dataset to resolve it.
2. The data was downloaded and preprocessed.
3. The classifier was trained and validated using the data (multiple times, until a good set of parameters and ConvNet architecture were found)
4. The model was tested against the test dataset and the results were submitted to the competition.
5. Repeated steps 4 and 5 for a retrained Inception V3 model, getting better performance.

I found steps 4 and 5 the most difficult, as I had to familiarize myself with TensorFlow framework together with the theory related to Convolutional Networks. Because the characteristics of the dataset, reach a good solution is very difficult without a pre-trained model, specially without the appropriate hardware (that is, several GPUs).

During the project, I realize the importance of reuse pre-trained models. Nowadays, I think that this is the way to face new projects, not the development from scratch like I tried in this competition. I'm completely satisfied with the results that I get retraining the Inception ConvNet, specially considering that retrain it was very easy.

Improvement

The faced problem is very complex an a lot of improvements can be done. Because the dataset is not large enough, multiple improvements could be done in the preprocessing stage For example, having a model that just detects phones in hands could ensure a very high accuracy estimating classes where the driver was texting or talking on the phone.

The other kind of improvements are related to the model. Multiple methods could be tried, like ensembles of several models (resNet, VGG, inception, googlenetvx), semi supervised learning, etc. The competition forum is a place where there are a lot of great ideas and solutions to try for improvements.