DISTRACTED DRIVER DETECTION

JANUARY 7

Authored by: Akshat Sharma

DEFINITION

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.
This Project aims to implement a model which will help in predicting the behavior of the driver.

In this project, we will implement an algorithm that could be used to detect the behavior of the driver based on the images provided by the dashboard camera of the Car. At the end of this project, our code will accept the test-images provided by the State Farm and after analyzing the mage, it will predict the probability of various behaviors that driver in the image is engaged in. For example the program will predict that a driver in the image is engaged in safe driving with probability of 0.5, talking to passenger with a probability of 0.3 etc.

PROBLEM STATEMENT

For my Capstone project I will develop a Machine/Deep learning agent to address the challenge presented by the State Farm on the Kaggle to predict the behavior of the driver from the image provided by the dashboard camera of the car.

Given a dataset of 2D dashboard camera images, our aim is to classify the behavior of the driver in the image and predict whether they are driving attentively, wearing their seatbelt, taking a selfie with their friends in the backseat, or involved in any other distracted behavior. This program can be used to alert the drivers whenever they are getting engaged into any distraction while driving.

We will try to solve it using the Keras CNN with Tensorflow as the backend, since CNN works great in general for image classification problems. Depending on the performance of the CNN, we may need to fine tune the parameters and implement other techniques like transfer learning to improve the performance of the model and save on training time. After loading and splitting the data we will also need to create the 4D Tensor arrays of the images provided in the dataset as Keras CNN requires 4D arrays.

The final model is expected to achieve a score so that it is in top 50% of the Public Leaderboard submissions in Kaggle.

METRICS

Submissions are evaluated using the multi-class logarithmic loss. Each image has been labeled with one true class. For each image, you must submit a set of predicted probabilities (one for every image). The formula is then,

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

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

The submitted probabilities for a given image are not required to sum to one because they are rescaled prior to being scored (each row is divided by the row sum). In order to avoid the extremes of the log function, predicted probabilities are replaced with max(min(p,1−10−15),10−15).

Since ours is a multi-class classification problem I think having the predictions as probability of each class against straight yes or no gives us a more nuanced view into the performance of our model. Log Loss takes into account the uncertainty of our prediction based on how much it varies from the actual label. It is an information-theoretic measure to gauge the "extra noise" that comes from using a predictor as opposed to the true labels. By minimizing the cross entropy, we maximize the accuracy of the classifier. As it relates to our problem I think it's better for the model to be somewhat wrong rather than being wrong completely. Based on the above statements I believe log-loss is the ideal metric for this problem.

ANALYSIS

DATA EXPLORATION

The dataset for this challenge has been provided by the State farm and can be obtained from the following Url:-

https://www.kaggle.com/c/state-farm-distracted-driver-detection/download/
imgs.zip - zipped folder of all (train/test) images

The provided dataset has the images of drivers involved in the different behaviors while driving. These different behaviors can be using mobile phone for texting or talking, eating, reaching behind and talking to the passenger etc.

Along with the image dataset State farm has also provided couple of files which contains information about the provided dataset and can be obtained from the following urls:-

https://www.kaggle.com/c/state-farm-distracted-driver-detection/download/driver_imgs_list.csv.zip - a list of training images, their subject (driver) id, and

https://www.kaggle.com/c/state-farm-distracted-driver-detection/download/
sample_submission.csv.zip - a sample submission file in the correct format

The 10 classes to predict are:

Class Id	Behavior
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

Description of the provided images:-

Total Images	Training	Validation	Test
102150	17939	4485	79726

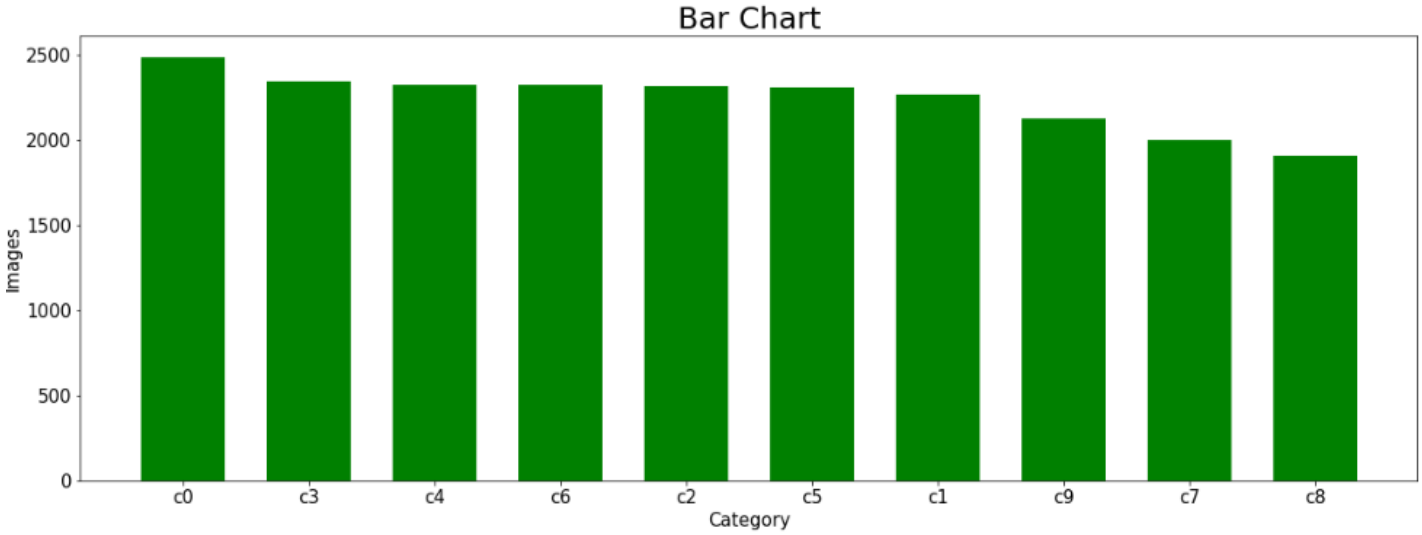
All the training, validation and test images are colored and have 640 x 480 pixels each and belong to the 10 categories mentioned above.

Data provided in the driver_imgs_list.csv.zip file contains data for the training images along with the driver Id's. Below table shows the frequency of different behaviors in the training images provided –

Class Id	Count
C0	2489
C1	2346
C2	2326
C3	2325
C4	2317
C5	2312
C6	2267
C7	2129
C8	2002
C9	1911

EXPLORATORY VISUALIZATION

The below bar chart plots the count of images for each class in the training dataset. From the graph it is clear that the distribution of the training images among the different classes is uniform.



ALGORITHMS AND TECHNIQUES

After analyzing the data and based on my learning from one of the earlier project, I believe that a CNN algorithm can help us get the good results for this problem. I have used CNN for other image classification projects before like predicting breed of the dog from the image and cancer detection. In both the projects CNN have yielded good results, making it a good fit for this project.

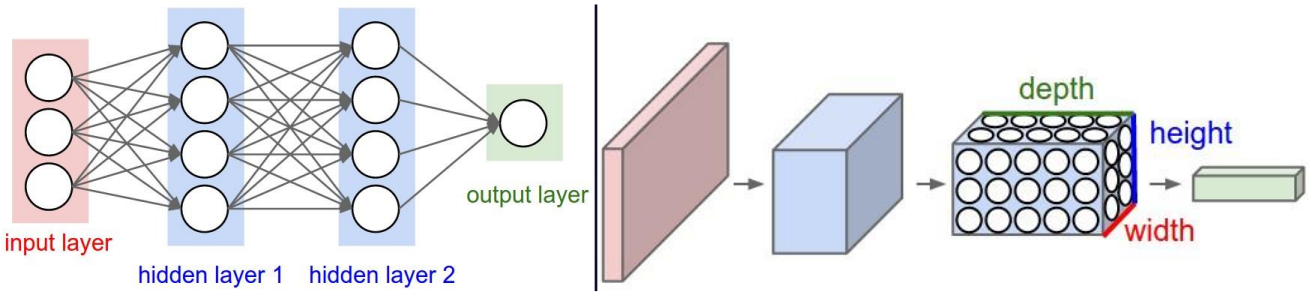
CNN – Convolutional Neural Network

Convolutional Neural Networks are just like normal Neural Networks which 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 expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. CNN's also have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer.

What's different in CNN from normal neural network is that the CNN architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

Convolutional Neural Networks take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular Neural Network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third

dimension of an activation volume, not to the depth of a full Neural Network, which can refer to the total number of layers in a network.) For example, the input images in CIFAR-10 are an input volume of activations, and the volume has dimensions 32x32x3 (width, height, depth respectively). In CNN, the neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer would for CIFAR-10 have dimensions 1x1x10, because by the end of the ConvNet architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension. Here is a visualization:



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Layers used to build ConvNets

As we described above, a simple ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We use three main types of layers to build ConvNet architectures: Convolutional Layer, Pooling Layer, and Fully-Connected Layer (exactly as seen in regular Neural Networks). We will stack these layers to form a full ConvNet architecture.

Convolutional Layer

The Conv layer is the core building block of a Convolutional Network that does most of the computational heavy lifting. The CONV layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. For example, a typical filter on a first layer of a ConvNet might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because images have depth 3, the color channels). During the forward pass, we slide (more precisely, convolve) each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on

higher layers of the network. Now, we will have an entire set of filters in each CONV layer (e.g. 12 filters), and each of them will produce a separate 2-dimensional activation map. We will stack these activation maps along the depth dimension and produce the output volume.

Pooling Layer

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function 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. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice). The depth dimension remains unchanged.

Fully-connected layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, just like regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.
It is worth noting that the only difference between FC and CONV layers is that the neurons in the CONV layer are connected only to a local region in the input, and that many of the neurons in a CONV volume share parameters. However, the neurons in both layers still compute dot products, so their functional form is identical. Therefore, it turns out that it's possible to convert between FC and CONV layers:

Above explanation of CNN is taken from this link. If you are interested in learning about CNN in detail, provided link is a very good source.

Transfer Learning

Since it is an image classification problem and we know that there are many popular pre-trained models which have been trained on millions of images from popular imagenet dataset, we can also use the transfer learning technique to reduce the training time of our model without compromising with the accuracy. Popular pre-trained models like VGG16, Resnet, VGG19 etc. can be utilized for this purpose.

In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest. The three major Transfer Learning scenarios look as follows:

ConvNet as fixed feature extractor

Take a ConvNet pretrained on ImageNet, remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet), then treat the rest of the ConvNet as a fixed feature extractor for the new dataset. In an AlexNet, this would compute a 4096-D vector for every image that contains the activations of the hidden layer immediately before the classifier. We call these features CNN codes. It is important for performance that these codes are ReLUd (i.e. thresholded at zero) if they were also thresholded during the training of the ConvNet on ImageNet (as is usually the case). Once you extract the 4096-D codes for all images, train a linear classifier (e.g. Linear SVM or Softmax classifier) for the new dataset.

Fine-tuning the ConvNet

The second strategy is to not only replace and retrain the classifier on top of the ConvNet on the new dataset, but to also fine-tune the weights of the pretrained network by continuing the backpropagation. It is possible to fine-tune all the layers of the ConvNet, or it's possible to keep some of the earlier layers fixed (due to overfitting concerns) and only fine-tune some higher-level portion of the network. This is motivated by the observation that the earlier features of a ConvNet contain more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but later layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset. In case of ImageNet for example, which contains many dog breeds, a significant portion of the representational power of the ConvNet may be devoted to features that are specific to differentiating between dog breeds.

Pretrained models

Since modern ConvNets take 2-3 weeks to train across multiple GPUs on ImageNet, it is common to see people release their final ConvNet checkpoints for the benefit of others who can use the networks for fine-tuning. For example, the Caffe library has a Model Zoo where people share their network weights.

BENCHMARK

For Benchmarking, the model can be compared against the top model from the Public Leaderboard for this competition and the goal for this project would be to gain the rank in top 50% of the public leaderboard, which equates to the score of 1.507 for the public score.

METHODOLOGY

DATA PREPROCESSING

Data Pre-processing is a vital step before creating a model and running it. First and foremost data is loaded from the datasets provided. After loading the training data, it has been split into training and validation data for the purpose of validating the model before using it on test data for making predictions.

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

(nb_samples,rows,columns,channels),

where nb_samples corresponds to the total number of images (or samples), and rows, columns, and channels correspond to the number of rows, columns, and channels for each image, respectively.

We implemented a function that takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224×224224×224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shape

(1,224,224,3).

Other function we implemented takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

(nb_samples,224,224,3).

Here, nb_samples is the number of samples, or number of images, in the supplied array of image paths. It is best to think of nb_samples as the number of 3D tensors (where each 3D tensor corresponds to a different image) in our dataset!

Creating the array of tensor for the dataset was both time and memory extensive process. In fact it required such a huge amount of memory that just the array tensors fir the training and validation data ended up consuming major part of RAM. Tensor array for the test data would have been 4-5 times bigger than the arrays for the training and validation data combined. Hence we opted for other solution and rather than creating the tensor array for the test data and storing it in memory, we passed the tensor for each test image file to the predictor function one by one. By the time we figured out this solution it end up taking a big bunch of time as the attempt to create the tensor array took 5-6 hours of run time each time before failing.

IMPLEMENTATION

Once the data was processed as per the requirement, next we moved on to creating a CNN from the scratch. Following steps were implemented for creating, training, compiling and testing the CNN.

1. Model was initialized as a sequential model from Keras.Model.

2. Then three convolutional layers were added of kernel size 2 and having activation function as relu.
3. For each consecutive convolutional layer number of filters increased from 16 to 32 and to 64.
4. Each convolutional layer was followed by a MaxPooling layer.
5. Then a GlobalAveragePooling layer was added.
6. Finally a fully connected layer with 10 nodes and activation function as softmax was added at the end of CNN.
7. The model is then compiled with 'rmprop' as the optimizer, 'categorical_crossentropy' as the loss function and accuracy as the metrics.
8. Before training the model, a ModelCheckPoint has been implemented to save the model weights at each epoch whenever validation loss is improved.
9. The model is then trained for 40 epochs with the batch size of 30.
10. After training the model is loaded with the best validation loss weights from the training.
11. Finally model was used to make predictions for each image in the test dataset.
12. Those predictions were saved in the suitable format for submission and obtaining the score.
13. First submission with this CNN landed us in the top 78% on the public leadership board which was below our expected benchmark performance.

    This model took a lot of time for getting trained and even much more time during testing. Training the model took approximately 3 hours and around 10-12 hours for testing each time. Though these times were improved by a lot after enabling the GPU computing, which was necessary for other reasons. After enabling the GPU capabilities training time got reduced to less than an hour and testing/prediction time got reduced to 3.5 hours.

REFINEMENT

To improve the model's performance and run time we implemented the transfer learning. In the transfer learning instead of training a CNN from scratch a pre-trained model is used as an initialization or fixed feature extractor.

For our model we used the pre-trained VGG16 model which has been trained on millions of images of the popular imagenet dataset. Steps involved in transfer learning process for model refinement were as follows:-

1. Created the instance of VGG16 model.
2. Used this instance to obtain the bottleneck features for training, validation and test dataset.
3. For the model architecture, a GlobalAveragePooling Layer with input shape as the train bottleneck features followed by a fully connected layer of 10 nodes have been added to the sequential model.
4. The model has been compiled using same parameters as the basic CNN with 'rmprop' as the optimizer, 'categorical_crossentropy' as the loss function and accuracy as the metrics.
5. The model has been trained on the traing bottleneck features and validation bottleneck features using the modelCheckPoint for storing the model with best validation losses.
6. The model is then trained for 50 epochs with the batch size of 20.

7. After training the model is loaded with the best validation loss weights from the training.
8. Finally model was used to make predictions for each feature in the test bottleneck features.
9. Those predictions were saved in the suitable format for submission and obtaining the score.
10. This submission with the transfer learning CNN landed us in the top 42% on the public leadership board and help us meet the expected benchmark performance.

    It should be mentioned that the process of obtaining the bottleneck features took large amount of time and space. Even after enabling the GPU capability, obtaining the bottleneck features for the test dataset took about 1.5 hours and occupied 8 Gbs of disk space.

RESULTS

MODEL EVALUATION

Final model where CNN trained with a pre-trained VGG16 model using transfer learning gave us the expected results. First bottleneck features were obtained for the train, validation and test data using the pre-trained VGG16 model. Model has been developed using VGG16 where BGG16 has been used as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model. We only added a global average pooling layer to get our final model.

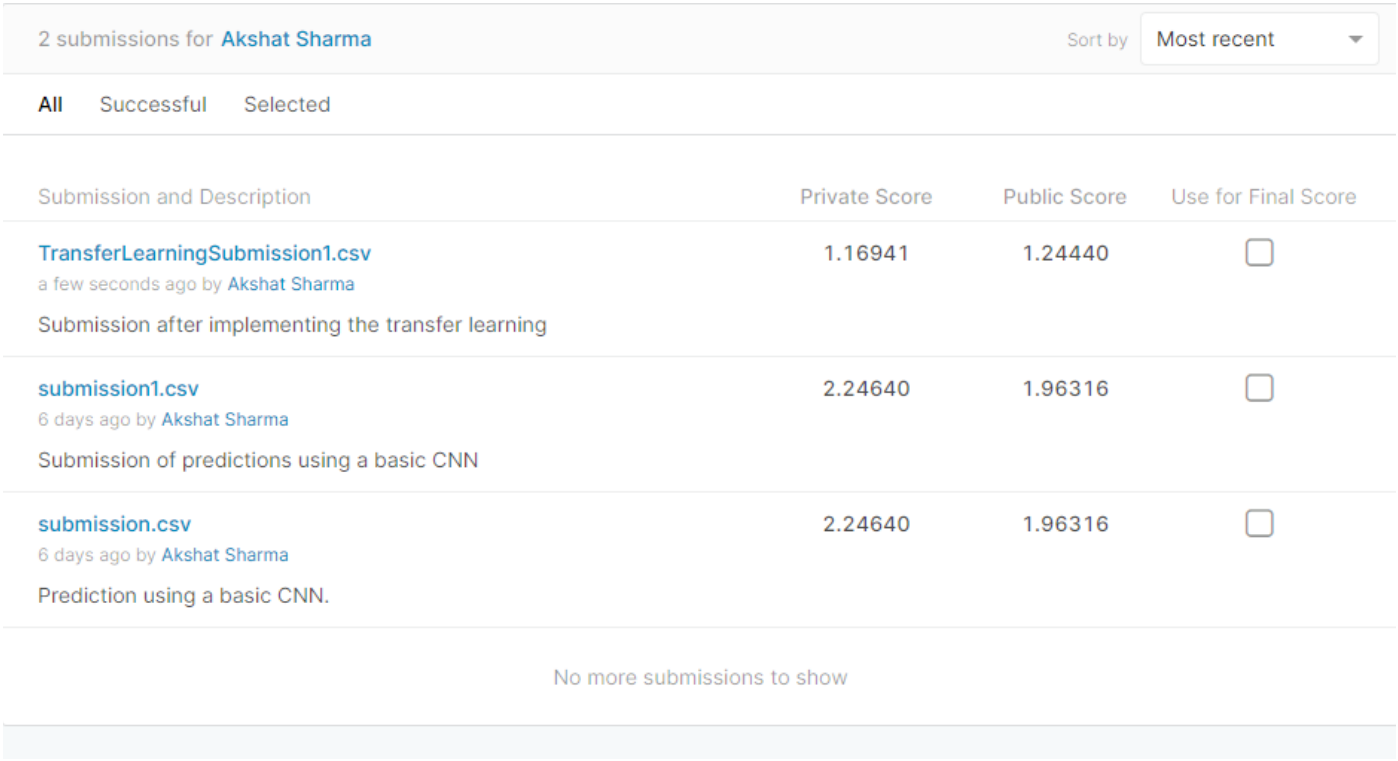
Model has been training using the Train and Validation bottleneck features against the respective targets and the model with the best validation loss has been saved offline. Model with best validation loss has been tested against the test data.
The predictions from the final model places us in the top 42% of the public leadership board of Kaggle which is better than our expectation of being in top 50%. Test dataset has huge amount of test images and our model has been tested against all of them and hence we can claim that the results from this model are both convincing and reliable.

JUSTIFICATION

Since our expectation were to rank among top 50% of the public leadership board, our model was required to obtain the score of 1.507 or better for a public score. Our final solution obtained the score of 1.24440 and ranked among the top 42%.

Transfer Learning and bottleneck features have helped reducing the training time by a significant factor making the model getting trained in the reasonable time. Model VGG16 has been used for the transfer leaning which

has been trained on millions of images of the imagenet dataset. Combining this robustness of VGG16 with the thorough testing of our model on huge test dataset makes our model robust and trustworthy. Results from our model are significant enough to help detect a driver getting involved in the distracted behavior during the driving and this might help in alarming the driver to prevent any accidents that might happen due to the distracted behavior.



Screenshot of various submissions on Kaggle along with the respective scores

CONCLUSION

FREE-FORM VISUALIZATION

Below are some of images classified by the VGG16 model.

This image is correctly classified by the model as c5 category i.e operating the radio. Classifier was able to predict the behavior correctly even after the right hand of the driver and radio were not visible in the image.

This image is correctly classified by the model as c8 category i.e hair and makeup.

This image is misclassified by the model as c2 category i.e talking on the phone-right, while the driver is actually drinking.

REFLECTION

As mentioned in the project report and as we know from the other image classification projects, a CNN model can produce great results for such kind of tasks. Even a basic CNN was good enough to get the rank among top 78% of the public leadership board. Since our benchmark was to rank amount the top 50% we needed to improve the performance of the model.
This performance improvement was achieved using the Transfer Learning from the well know VGG16 model and obtaining the bottleneck features for the datasets.

One very interesting aspect of this project was the large amount of data provided with it. Specially the test dataset which consisted of approximately 80,000 images of drivers. Testing the model against such a big dataset provides a great sense of reliability towards the performance of the model. Reasonable amount of training time along with the good prediction results makes our model a good fit to solve the problem.

One aspect of the project which made it very difficult to implement the model was none other than the interesting aspect of this project, yes the data itself. While initially choosing this problem to solve in my capstone project, I didn't realized that with such a big amount of data, the model will require huge amount of

computing and storage resource for it to run, especially when making predictions. To provide some insight just assigning an array of tensor for the train, validation and test data made my RAM of 32GBs looked very small. Just the array for the test data itself required 41 GBs of RAM. I had to literally make additional 70 GBs available for the virtual memory apart from my 32 GBs of RAM so that the memory requirements of this model can be fulfilled. Bottleneck features for just the test data ended up taking 8 Gbs of disk data. To add more to this both interesting and difficult aspect of the project, just running the predictions for the basic CNN took around 12-14 hours to run for my system's powerful Intel Xeon processor. It actually became very annoying at a point when after running for the model for 12 hours you have to realize that you will need to run the whole code again (sometimes due to some required coding improvement and other times due to storage capacity). In fact, while trying to obtain the bottleneck features for the training, validation and the test dataset, I realized that it would take almost a month's (actually more than that) continuous run of the program. It is then it became impossible to move ahead in my project without enabling the GPU computability of my system, luckily my system has the kind of hardware that's required. Obviously enabling the tensorflow-GPU was also not a smooth road for me. For known or unknown reasons and issues it ended up taking almost 10 days(millions of time installing and uninstalling Anaconda and other software and packages) of mine to successfully establish the environment where I can run my model using the GPU capabilities. Yes, thanks to this project, it made me a troubleshooting expert too for Anaconda and Tensorflow and also get to learn a lot of new things. I can keep writing about my experience/struggle/learning but I think this should be enough to explain why this project was both difficult and interesting at the same time.

IMPROVEMENT

If I get a chance I would definitely like to try more combination of fine-tuning the model to improve the performance of the model but since I am already working on it for more than a month for now I will convince myself by performance of the model better than the expected benchmark. Long runs for training and prediction made it difficult to try various combinations of parameters. I was specifically interested in trying different type of optimizers with different value of parameters to find the optimal one.

One other technique that I can think of to improve the model is to use the Functional Model capability from Keras which provides a more flexible way for defining models and specifically allows you to define multiple input or output models as well as models that share layers.

I believe that this model can certainly be used as the new benchmark and at the same time I do believe that a better solution can be implemented if provided with enough and powerful computing and storage resources.

References

https://medium.com/@viveksingh.heritage/how-to-install-tensorflow-gpu-version-with-jupyter-windows-10-in-8-easy-steps-8797547028a4

https://github.com/Akshat2127/Dog-Breed-Identifier/blob/master/dog_app.ipynb
https://towardsdatascience.com/building-a-deep-learning-model-using-keras-1548ca149d37
https://www.oreilly.com/ideas/evaluating-machine-learning-models/page/3/evaluation-metrics
http://wiki.fast.ai/index.php/Log_Loss
http://cs231n.github.io/convolutional-networks/
http://cs231n.github.io/transfer-learning/