UDACITY Capstone Project Report

State Farm Distracted Driver Detection
Can computer vision spot distracted drivers?

Bang-Sin Dai
May 5, 2018

Definition

Project Overview

Driving safety is an important issue. Many traffic accidents are caused by improper or distracted driving behaviors. The drivers may operate their mobile phones, talks to passengers, pick up some things around them and even drink beverages during driving. These inadvertent and dangerous driving behaviors often cause the loss of not only the driver's live but also other people's lives. Many people also get different levels of body hurts due to such traffic accidents.

According to the CDC motor vehicle safety division [i], 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 [ii] hopes to improve these alarming statistics and also to reduce the traffic accidents and thus better insure their customers, by testing whether dashboard cameras can automatically detect drivers engaging in distracted behaviors. Based on the motivation, State Farm provides prizes for competition participants with the best results on the problem of “distracted driver detection” [iii].

The distracted driver detection problem is a kind of image recognition problem. Given a data set of driver images as input, may the computer or machine learn from the input images and then automatically help us to find potential distracted behaviors of the drivers in the images?

The input data is a set of 2D driver images provided by State Farm [iv], each taken in a car from a dashboard camera with a driver doing something in this car, e.g., texting, eating, talking on the phone, makeup, reaching behind, etc. Each image is in one of the 10 classes (labels) listed below**:**

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		

Problem Statement

Given a data set of 2D driver images photoed from dashboard camera, State Farm is challenging data analysts and machine learning engineers to classify each driver's behavior. For example, are they driving attentively, wearing their seat-belt, or taking a selfie with their friends in the backseat? That is, by using computer vision or image recognition technology, can we detect and highlight these dangerous and distracted driving

behaviors and then automatically alarm the drivers in real time to stop their unsafe behaviors?

Formally speaking, the definition to the problem is that given an image of a driver on the seat, our algorithm requires to identify which one of the 10 classes mentioned above this image belongs to with the highest possibility (or maximum likelihood). There

If we can design an algorithm using deep learning technique, like convolutional neural network (CNN) model for example, to learn the driver behaviors labeled in the given input image data set and then to identify potential unsafe driving behaviors in the new coming images with a relatively high classification performance, the objective of the distracted driver detection problem can be regarded as achieved successfully. In other words, our mission is to train some CNN models to learn the labeled driver behaviors in train image data set given in input first, then we can validate the CNN models we construct using the validation images, and finally we can test prediction performance results of our CNN models on test images. For each test driver image, we define the output generated by our CNN prediction model to be consisted of a list of probabilities of the 10 classes this test driver image belongs to. For example, img_0.jpg below is the test driver image and it belongs to class c0 with probability 0.3 and class c1 with probability 0.1, and so on.

img,c0,c1,c2,c3,c4,c5,c6,c7,c8,c9
img_0.jpg,0.3,0.1,0.2,0,0,0,0.3,0,0,0.1
…

Metrics

We adopt the same evaluation metrics as that in Kaggle competition on this problem. Please see the Kaggle website [v] for a reference. The error function to evaluate the performance scores of models is the **(multi-class) logarithmic loss function**:
where *N* is the number of images in the test set, *M* is the number of image class labels,

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

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*.

Error function needs to properly represent the “distance” between prediction results and actual results. That is, the error function serves as a clear performance measuring metric, and we want to minimize (optimize) the error function to obtain best possible performance (or say maximum likelihood).

Log-loss error function has nice properties of differentiable and continuous, hence it is suitable to be applied gradient descent on it to improve the model performance iteratively. Since probability is the major tool used to evaluate and to improve the prediction model, given the overall actual results, the cross entropy form of log-loss function appropriately calculates the likelihood of a prediction model in a concrete way. This is the most important key for log-loss function to become a proper and also popular evaluation metric.

Analysis

Data Exploration

The data of driver images provided by State Farm include training and testing sets, and there are total 102150 images. The original train set consists of 22424 images, and each belongs to one of the 10 classes of driver behaviors shown above, i.e., c0, c1, …, and c9. The test image data set consists of 79726 driver images. Indeed it is a large number of data images.

For computer resource/performance limitation and time efficiency, we just randomly sample a ratio of 40% (8969 images) of the original train set of 22424 images provided by State Farm to be actually used and ignore the remaining 60% (13455 images) of the original train set. This randomly sampling step can be viewed as the first step of our data preprocessing.

Now we have sampled the 8969 images from original train data to be used and then we shuffle and split them into 7175 images for model training and 1794 images for validation. Note that we still have a test set of 79726 images.

Each driver image is colored and with size of 640 x 480 pixels, and we show some of them as examples in the following as a part of data characteristic exploration.

c4:talking on the phone – left

c8:hair and makeup

c5:operating the radio

The examples above show that the data images provided by State Farm are quite decent and neat. The images are photoed from almost the same viewing directions and angles. It is because State Farm set up these experiments in a controlled environment - a truck dragging the car around on the streets - so these "drivers" were not really driving. Therefore the picture quality of the driver images is also decent and neat. Such data image characteristic is quite good for us since we don't need to worry about the issues of noise, different picture quality like camera shaking, or different viewing angles too much.

Exploratory Visualization

First we read the “driver_imgs_list.csv” file, provided by State Farm, by using Pandas. This file is a list of original input training images, their subject (driver id), and classname (label id). We display the top 5 data items in “driver_imgs_list.csv” for data observation.

	subject	classname	img
0	p002	c0	img_44733.jpg
1	p002	c0	img_72999.jpg
2	p002	c0	img_25094.jpg
3	p002	c0	img_69092.jpg
4	p002	c0	img_92629.jpg

Now we realize that the first 5 data items in this csv file are all in class c0 (safe driving) and all with the same driver in the images. Then we try to display the description information of this csv file.

	subject	classname	img
count	22424	22424	22424
unique	26	10	22424
top	p021	c0	img_4530.jpg
freq	1237	2489	1

The description information shows that the number of train images State Farm provides is 22424, and the number of classes is 10. These are the same as what we count statistically before. We then explore the data set sizes of the different classes to check whether existence of data imbalance.

```
c0    2489
c3    2346
c4    2326
c6    2325
c2    2317
c5    2312
c1    2267
c9    2129
c7    2002
c8    1911
Name: classname, dtype: int64
```

The visualization above shows that there is just a little bit size imbalance between different classes, i.e., the data set can be regarded as roughly balance. Note that class c0 has the maximum number of 2489 images and class c8 has the minimum number of 1911 images among all the 10 classes. Therefore we don't require to worry about the data imbalance/skew issue too much. Generally speaking, the data images State Farm provides are decent and neat.

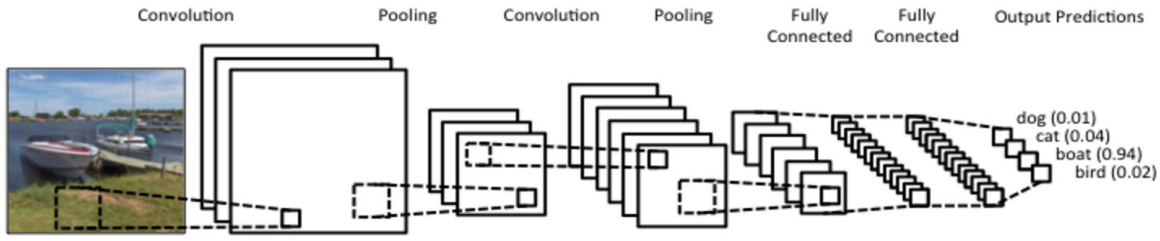
Algorithms and Techniques

To deal with the distracted driver detection problem, we can apply the **deep learning** technique, like the **CNN (convolutional neural network)** models, which are quite appropriate to handle problems in the fields of computer vision and image/pattern recognition. Our problem here is a multi-classes image classification problem. The algorithms we provide need to identify the objects (drivers) in the images and try the best to classify what they are doing. So a typical approach to solve this issue is to build and train the CNN models. The CNN models can be implemented by Keras and Tensorflow. We can first construct and train the CNN models by using the training set (let the machine learn from the correct labels on the training driver images) and then do validation by using the validation image set. We can record the model weights with the best validation loss. Then we can test the model prediction (classification) performance, e.g., the metric score of log-loss error function, by using the testing driver image set. In addition, we can also leverage the technique of transfer learning. That is, we can use good CNN models (model architectures and/or model weights) pre-trained by others before to be the initial parameter settings or a partial structure of our CNN models on this distracted driver detection problem, without training or fine-tuning the whole network parameters by ourselves

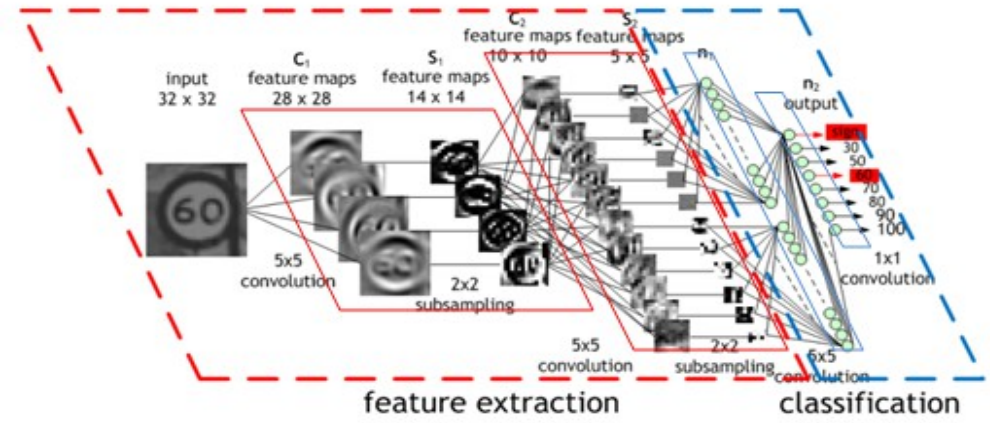
completely. Transfer deep learning is expected to save much training time and have better prediction quality. The popular pre-trained models for transfer deep learning are like VGG16, VGG19, Xception, ResNet50, InceptionV3, etc.

We simply introduce the **CNN** model [vi] as follows.

CNN is an important neural network model in deep learning techniques, especially for dealing with image or speech recognition problems. CNN is a feed-forward neural network, composed of three types of layers: **convolutional layers**, **pooling layers**, and then followed by **fully connected layers**. The CNN architecture is designed to be fed with 2D image data as input. The output of CNN is typically a classification label or the probabilities of the classes the input image belongs to. The conceptual architectures of some simple CNNs are shown in the figures below.

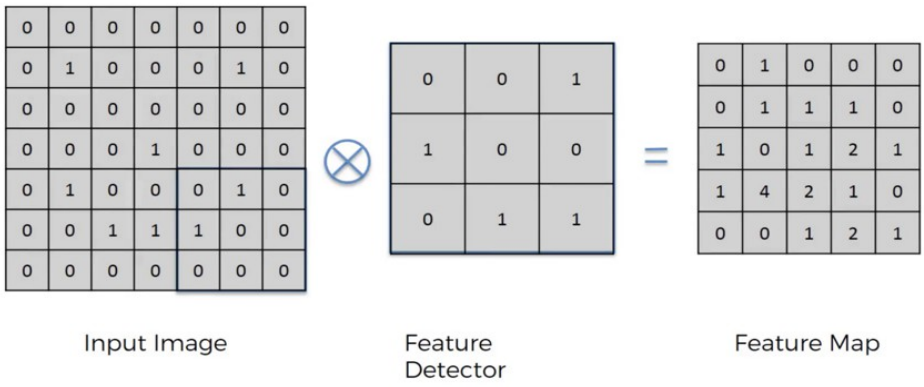


CNN conceptual architecture 1

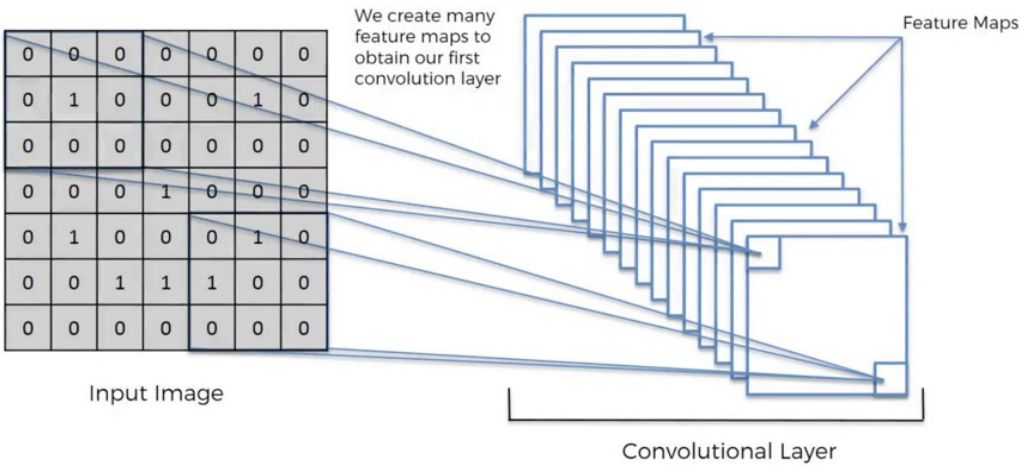


CNN conceptual architecture 2

Convolutional layers:
Convolutional operations are taken on the input array of the image with some specific **feature detectors/filters** (for example a 3x3 matrix). The convolutional operation is just like the vector dot product (linear production and then summation). Through the particular feature detectors/filters, the specific features (regional information like edges and colors) of the original image array are extracted by the feature detectors “scanning” the image array from left to right and from top to bottom and then doing convolutional operations, and the so-called **“feature map”** is generated. The diagrams of convolutional operations and feature maps are shown in the following figures.

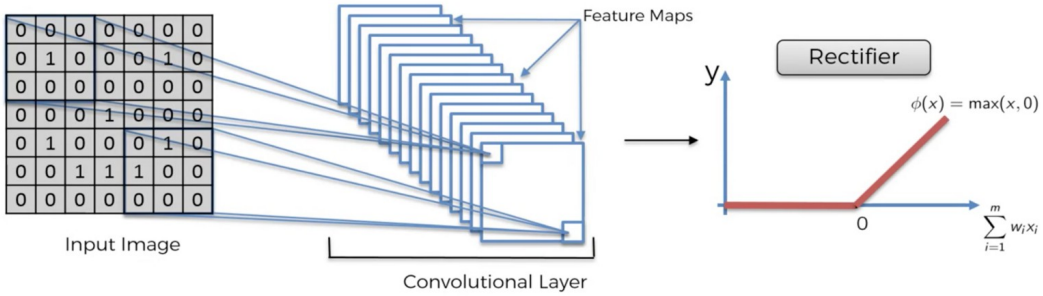


Convolutional operations

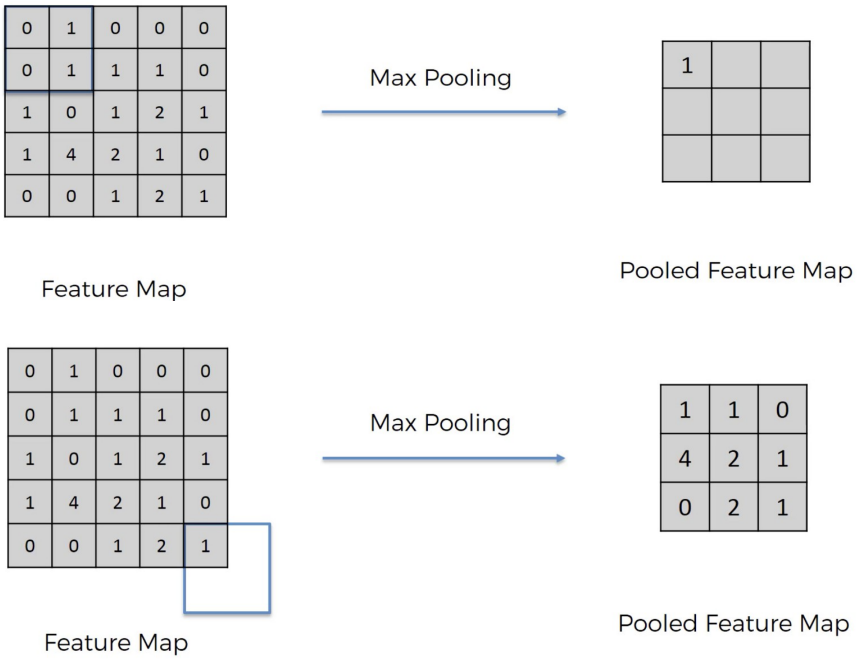


Use many feature detectors to extract many feature maps.

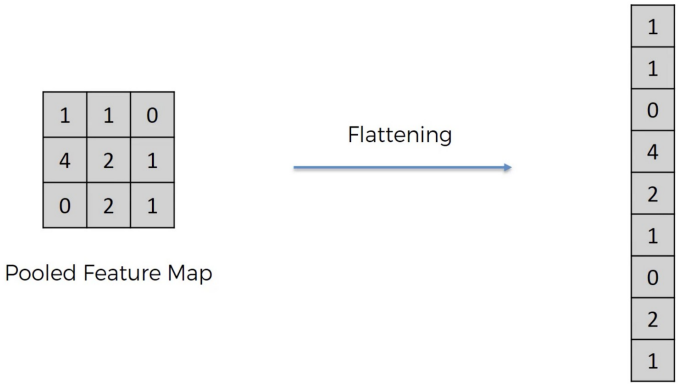
Then the ReLU activation function is applied to throw out negative values, as shown in the following figure.

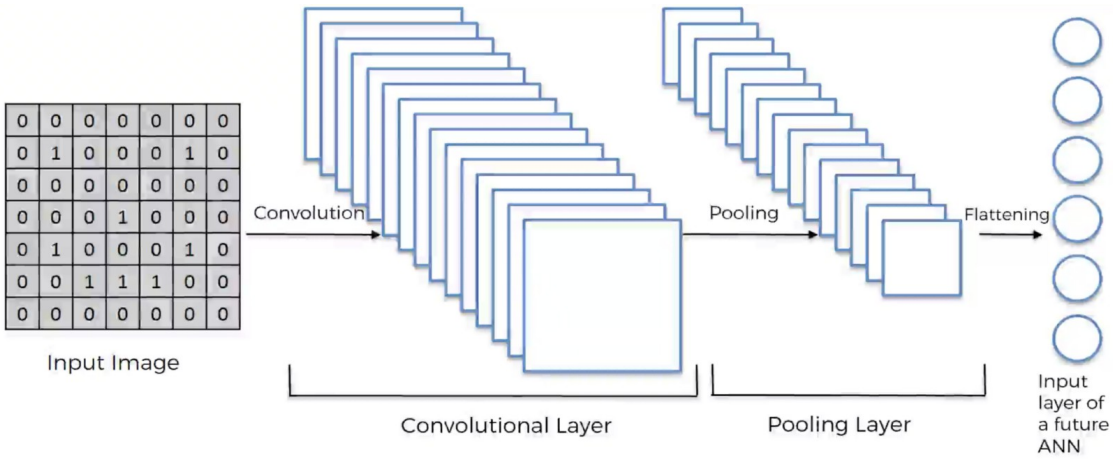


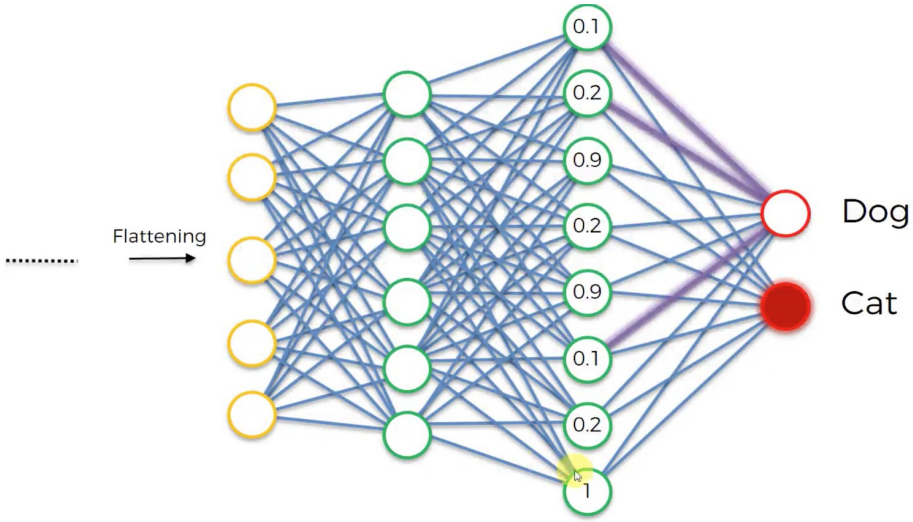
Pooling layers:
There are two major types of pooling layers: **max pooling** and **global average pooling (GAP)**. The max pooling uses a window/pool to “scan” the input array (a feature map for example) from left to right and from top to bottom (similar to feature detector before) and then records only the maximum value in the window/pool. The global average pooling only records the average value of the entire feature map. The main benefits and functionalities of pooling layers are noise-resistance and also dimensional reduction (avoidance of overfitting). The schematic diagram of max pooling is shown in the following figures.



Fully connected layers:
Fully connected layers are just like dense layers in the basic deep neural network. The resulting arrays of previous convolutional and pooling layers are flatten and then connected to the dense (fully connected) layers of neural networks. The schematic diagram of fully connected layers is shown in the figures below.







The convolutional and pooling layers of CNN have much less parameters than the dense layers in deep neural networks with the same architecture of layers and nodes since they keep only regional information (features) rather than global information in deep neural networks. The first convolutional layer records fundamental underlying features (e.g., edge, color), and the second convolutional layer records the advanced features (e.g., circles and stripe), and the third convolutional layer records the more complicated and also the higher-level features, and so on. Therefore the CNN model transforms the original single image (input) into a large number of high-level features instead finally (the spatial information in the original image will be vanished). The fully connected layers behind can thus help us to identify what the object in the image is based on these high-level features.

Benchmark

In the public leader board of Kaggle competition on the distracted driver detection problem [vii], there are many models (1440 in total) constructed by different teams. The top one model has the score (multi-class logarithmic loss) of 0.08689. For simplicity and convenience, we can choose the median of the 1440 scores of all the models, i.e., score 1.50719, as the benchmark model score to be compared with our model score.

Methodology

Data Preprocessing

Besides the randomly sampling step for original large image data set, we also need to do one hot encoding preprocessing to deal with the string-valued classification labels (class names) for the CNN prediction models.

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. It is best to think of **nb_samples** as the number of 3D tensors (where each 3D tensor corresponds to a different image) in the data set.

We need to preprocess the data for Keras. To take a string-valued file path to a color image as input and finally returning a 4D tensor suitable for supplying to a Keras CNN, we

first load the image and resize it to a square image that is 224×224 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: 'r', 'g', and 'b', i.e., 3D tensor with shape (224, 224, 3). Likewise, since we are processing a single image (or sample), the returned 4D tensor will always have shape (1, 224, 224, 3).

Finally we re-scale (normalize) the images by dividing every pixel in every image by 255. These are our preprocessing steps.

Implementation

We use Keras and Tensorflow as backend to implement our CNN models. We create the first CNN model to classify driver behaviors from scratch. The model architecture of our first CNN model is shown in the figure below.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 224, 224, 16)	208
max_pooling2d_1 (MaxPooling2	(None, 112, 112, 16)	0
batch_normalization_1 (Batch	(None, 112, 112, 16)	64
dropout_1 (Dropout)	(None, 112, 112, 16)	0
conv2d_2 (Conv2D)	(None, 112, 112, 32)	2080
max_pooling2d_2 (MaxPooling2	(None, 56, 56, 32)	0
batch_normalization_2 (Batch	(None, 56, 56, 32)	128
dropout_2 (Dropout)	(None, 56, 56, 32)	0
conv2d_3 (Conv2D)	(None, 56, 56, 64)	8256
max_pooling2d_3 (MaxPooling2	(None, 28, 28, 64)	0
batch_normalization_3 (Batch	(None, 28, 28, 64)	256
dropout_3 (Dropout)	(None, 28, 28, 64)	0
global_average_pooling2d_1 (	(None, 64)	0
dense_1 (Dense)	(None, 64)	4160
batch_normalization_4 (Batch	(None, 64)	256
activation_1 (Activation)	(None, 64)	0
dropout_4 (Dropout)	(None, 64)	0
dense_2 (Dense)	(None, 10)	650
batch_normalization_5 (Batch	(None, 10)	40
activation_2 (Activation)	(None, 10)	0

Total params: 16,098
Trainable params: 15,726
Non-trainable params: 372

We use three convolutional layers followed by three max pooling layers interleavely and then use two fully connected layers behind in the CNN architecture. We also adopt batch_normalization layers between each convolutional layer or dense layer and their activation layer to avoid co-variate shift and to accelerate the training process. The number of filters in each convolution layer is twice to the previous one (this is a common practice), and we choose 16, 32, and 64 filters to extract the feature maps (regional information). Both the window size of feature filter in each convolution layer and the pool size in each max pooling layer are (2,2), and it's also a kind of typical choices. We set the padding parameter to be “same” for not lost information near matrix boundaries. The activation function in each layer beside the output layer is ReLU for dealing with the vanishing gradient problem, and that in output layer is SoftMax for calculation of probabilities on the

multi-classes. In max pooling layers, we set the strides parameter to be 2 for half both length and width of each 2D feature map (dimensional reduction), and such strides setting is also typical. Before fully connected layers, we use the GlobalAveragePooling2D layer, which can immediately reduce the amount of parameters and avoid overfitting as well as save much time. We adopt the dropout layers with probability 0.2 to reduce opportunity of overfitting. We choose the number of nodes to be 64 in the first fully connected layer for an initial try, and due to the 10 classes of driver behaviors, the number of nodes in output layer is 10.

To compile the CNN model, we choose RMSprop optimizer (RMS: root mean squared error). It decreases the learning rate by dividing it by an exponentially decaying average of squared gradients. We use categorical cross entropy to be the loss (error) function. To train the model, we adopt epochs parameter to be 10 and batch_size parameter to be 20 for an initial try. Besides, we also use model checkpoint to save the model that attains the best validation loss.

The test image set is quite large and there are 79726 test driver images. Each of them needs to be tested for result submission format defined by Kaggle, to generate the test score of our proposed CNN models. Because of the computer performance limitation (memory, CPU, ...), when we implement the code to build and to train the CNN models, we need to try our best to save the use of memory as much as possible to avoid running out of memory (and thus leading to extreme slow program execution) and long CNN model training time. Once the memory spaces of some variables (like lists, Numpy arrays, tensors, etc) are not used, we release them instantly. We also adopt coding skills to avoid record of a large number of test tensors or test bottleneck features by computing them individually when we need them. Based on the same reason, we also cannot use the whole original train image data set (we do randomly sampling to pick only 40% ratio) or a larger number of epochs to train our CNN models actually. During the transfer learning process, we also spend quite much time to obtain the bottleneck features.

Refinement

To reduce training time without sacrificing accuracy, we train our next two CNN models using transfer learning techniques for improvements. Our next two CNN models on transfer learning adopt the pre-trained **VGG16** model and the pre-trained **ResNet50** model as **fixed feature extractors**, respectively, where the last convolutional outputs of VGG16 and ResNet50 are fed as the inputs to our own CNN models. Due to the computer resource/performance limitation, we just simply select to **freeze** the parameter weights of the pre-trained VGG16 model and also ResNet50 model, i.e., fixed feature extractors, and only train the fully connected layers we add behind these two pre-trained models.

Note that actually the data set size of driver images in our problem is quite large, and hence we don't need to worry about the overfitting possibility, which often occurs in the smaller data set. In other words, instead of freezing the pre-trained model parameters, we can choose to re-train the parameters of the whole CNN model (or the higher-level convolutional layers) of transfer learning. We may decide to use the weights of VGG16 and ResNet50 models as the initial values to re-train the whole CNN model if the driver images are similar to the images used to train the VGG16 and ResNet50 models previously, or directly use random values as parameter initialization to re-train the whole CNN if these images are not similar.

One reason to choose ResNet50 as the transfer learning model is also the computer resourse/performance issue since the bottleneck features of ResNet50 is quite smaller

than other popular models like VGG16, VGG19, Xception, and InceptionV3. Another reason to choose ResNet50 is that Resnet50 neural network model won the 1st place on the ILSVRC & COCO 2015 competitions of classification tasks of ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation [viii]. I think Resnet50 is one of good CNN candidates for using transfer learning.

When using transfer learning, we don't include the fully-connected layers at the top of the VGG16 and ResNet50 models. Then we add the fully connected layers we design, as shown in the figures below. That is, we calculate the bottleneck features of the pre-trained VGG16 and ResNet50 models, and then we feed the bottleneck features to our fully connected layers shown below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_1 ( (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                32832
_________________________________________________________________
batch_normalization_1 (Batch (None, 64)                256
_________________________________________________________________
activation_1 (Activation)    (None, 64)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
_________________________________________________________________
batch_normalization_2 (Batch (None, 10)                40
_________________________________________________________________
activation_2 (Activation)    (None, 10)                0
=================================================================
Total params: 33,778
Trainable params: 33,630
Non-trainable params: 148
_________________________________________________________________
```

Fully connected layers we add behind VGG16

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_1 ( (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                131136
_________________________________________________________________
batch_normalization_1 (Batch (None, 64)                256
_________________________________________________________________
activation_50 (Activation)   (None, 64)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
_________________________________________________________________
batch_normalization_2 (Batch (None, 10)                40
_________________________________________________________________
activation_51 (Activation)   (None, 10)                0
=================================================================
Total params: 132,082
Trainable params: 131,934
Non-trainable params: 148
_________________________________________________________________
```

Fully connected layers we add behind ResNet50

The fully connected layers we add behind VGG16 and ResNet50 models are the same as that in our first CNN model (from scratch) above. Please see the “implementation” part for a reference. Such network layer design is neat for making comparisons (with the first CNN model).

To compile the model, we choose RMSprop optimizer again. Also we use categorical cross entropy to be the loss error function. To train the model, we adopt epochs parameter to be

30 to reduce the opportunity of overfitting and batch_size parameter to be 20 to accelerate the gradient descent iterations. Besides, we also use model checkpoint to save the model that attains the best validation loss.

Results

Model Evaluation and Validation

We create three CNN models: the first CNN model from scratch, the CNN model using transfer learning on VGG16, and the CNN model using transfer learning on ResNet50.

During the CNN model training/fitting process (feed forward and back propagation), a validation data set is used to evaluate the CNN model (to do gradient descent) and also to keep the parameter weights of the model with the best validation loss by model checkpoint. Note that the ratio between our training image set and validation image set is 4 (7175 images for training and 1794 images for validation).

The test results (on 79726 test driver images) of our three CNN models are as follows:

(1) the first CNN model from scratch
Log-loss score: 2.60072
Public leader board rank: 1355/1440
Private leader board rank: 1340/1440
Best validation loss: 1.75698 (while 1.3517 training loss)

(2) the CNN model using transfer learning on VGG16
Log-loss score: 1.93189
Public leader board rank: 1120/1440
Private leader board rank: 1021/1440
Best validation loss: 0.10374 (while 0.1340 training loss)

(3) the CNN model using transfer learning on ResNet50
Log-loss score: 3.08525
Public leader board rank: 1380/1440
Private leader board rank: 1380/1440
Best validation loss: 1.22959 (while 0.3610 training loss)

The CNN model we create with the **best testing score** is **transfer learning on VGG16**. And this CNN model will be our final model choice. Not only it has the best testing score, but also during the CNN model training/fitting process, we can see that transfer learning on VGG16 also has **better best validation loss** than the other two CNN models. This information also shows the robustness of transfer learning on VGG16 is also better than the other two CNN models.

Justification

Our benchmark is defined to be the median of the total 1440 scores (the average of 720th and 721th) on the public leader board of Kaggle competition, i.e., log-loss score 1.50719. It is obvious that our final CNN model (transfer learning on VGG16) with the score 1.93189 is worse than the benchmark performance. In addition, our final CNN model is ranked 1120 out of 1440 in public leader board, and ranked 1021 out of 1440 in private leader board. Based on the statistical results, our final CNN model is not significant enough to have solved the distracted driver detection problem since it is worse than half number of total models presented on the leader board. This also means that our final model can be

easily substituted by many other CNN models. Our final CNN model is actually not good enough. But in fact we can expect such results at all and it is really not surprised. It is because: (1) due to our computer hardware performance limitation (small memory and slow CPU), we just only pick 40% of the original training data provided by State Farm to train and validate our final CNN model. (2) when using transfer learning on VGG16, due to the same computer performance issue, we are almost “forced” to choose to freeze the parameter weights of the pre-trained VGG16 model, i.e., we just simply use them as fixed feature extractors (to compute bottleneck features). We have no chance to re-train or optimize the parameters of the whole CNN (or partial higher-level convolutional layers). Since in fact the size of driver image data set in our problem is huge, we don't need to worry about the overfitting possibility, which often occurs in the smaller data set. We could try to use the weights of pre-trained VGG16 model as initial values of parameters to re-train the whole CNN model (or partial higher-level convolutional layers), or we could directly use random values as parameter initialization to re-train the whole CNN. The above-mentioned two reasons badly affect the prediction performance of our CNN models in a large degree. Another CNN model worth to be discussed is transfer learning on ResNet50. It is obvious that this CNN model is serious overfitting since it has large log-loss score gap between training loss and validation loss. It even obtains a worse score (3.08525) than the CNN model we design from scratch (with score 2.60072). It is also not surprised since ResNet50 model has quite deeper model architecture as well as quite more parameters than VGG16 model, and we just use them as fixed feature extractors.

Conclusion

Free-Form Visualization

Here we demonstrate some examples of prediction results of the test driver images by our CNN model using transfer learning on VGG16.

c5: operating the radio (correct)

c4: talking on the phone – left (wrong)

c0: safe driving (wrong)

c2: talking on the phone – right (correct)


c0: safe driving (correct)


c7: reaching behind (correct)

It is interesting about the wrong prediction cases: the 2nd figure and the 3rd figure. In the 2nd figure, the driver lifts his left hand and just touches his face (or mouse). This action is quite similar to talking on the phone using the left hand but actually it is NOT. This case makes our CNN model giving a wrong prediction. In my opinion, it is really hard to classify it correctly, right? Let's take a look at the 3rd figure. The wrong prediction may be caused by the “phone” feature. It seems that our CNN model cannot clearly and concretely capture the high-level feature of “phone”. Hence our CNN model cannot successfully detect the “phone” object in the image in good identification performance. So I guess the root causes of the wrong predictions on the 2nd and 3rd figures are actually the same (the “phone” feature is the key point). Through the visualization and demonstration of real examples, we can continue to refine and improve our CNN model based on our observation and analysis.

Reflection

We summarize the outlines and processes in our approach workflow as follows.

1. Clearly realize and define the extracted driver detection problem: background, motivation, topic (field), scope, property, etc. Obtain the data set of driver images and realize basic data characteristic: size of data set, labels, image format.

2. Propose potential ML solutions/approaches/algorithms (CNN models), and define suitable performance evaluation metrics (log-loss error function) and also the benchmark for performance comparisons of our solutions and results.

3. Import data sets:
First we import original “train” and “test” data sets provided by State Farm. Since the original image data set is too huge and due to computer performance limitation, we do randomly sampling to select only a smaller subset (ratio 40%) of data to use. Then we shuffle and split the sampled image data set into the training set and the validation set.

4. Data analysis, exploration, and preprocessing:
First we explore the data set in detail: data visualization, image quality, data balance degree, noise, photo angles, etc. We check the number of images in each class and whether the data set is roughly balanced. Then we preprocess the images for Keras by converting each image to a 3D tensor with (224, 224, 3) shape, i.e., 224 x 224 pixels with three channels of '*r*', '*g*', and '*b*'. Then transform each 3D tensor to a 4D tensor with shape (1, 224, 224, 3). Finally re-scale the images by dividing every pixel in every image by 255.

5. Create a CNN from scratch to classify driver images:
We use Keras and Tensorflow as the backend to implement our CNN model. In this step, we provide the first architecture of the CNN model we design. And then test the performance result of the first CNN model.

6. Create a CNN using transfer learning on pre-trained VGG16 to classify driver images:
To reduce training time without sacrificing accuracy, we train a CNN model using transfer learning. In this step, our CNN model use the pre-trained VGG16 model as a fixed feature extractor, where the last convolutional output of VGG16 is fed as input to our model. Then we train and validate the CNN model and finally test the performance result.

7. Create a CNN using transfer learning on pre-trained ResNet50 to classify driver images:
In this step, instead of VGG16, we try to use another pre-trained model, ResNet50, for potential different pre-trained model choice. Besides the pre-trained model of ResNet50, the remaining model architecture and parameter setting in training process are similar to previous VGG16. Then we test the performance result of the CNN model using the testing driver image set.

8. Analysis and comparison of algorithm test results:
We choose the best (the lowest log-loss score) testing result among the three CNN models we construct to be the final CNN model we propose. We compare and analyze the scoring results as well as performance of the three CNN models in detail. Also check whether the results meet our expectation.

9. Conclusion and improvement:
The justification of comparing the benchmark performance with our solution results is provided. Through visualization and demonstration of real prediction examples of test driver images, some drawbacks and weak points of our CNN model are found. Based on the observation and analysis, our CNN model can be further refined. Finally we make the conclusions and also propose the improvement for our CNN model.

Interesting aspect: Neural network is like a black box, and there is a large number of parameters in it. We have many different choices of number of layers, nodes, and parameters in model architecture model. Even the training and fine-tuning processes of CNN, we also have many different choices on optimizer, epochs, learning rate, activation function, batch size, drop out, window size, etc. We need to do many trials and experiments on the CNN creating (model architecture design), training, and fine-tuning processes to construct the experience of ourselves. The experience of dealing with a huge image data set is also quite interesting, exciting, and important to me.

Difficult aspect: CNN training and testing time may be quite long. So the trial-based fine-tuning approach may be time costly and not efficient. Actually it is also hard to fine-tune a good CNN model by ourselves completely. CNN models require high performance computer environment to execute them smoothly (for training and testing processes and even the data/image preprocessing step). If the computer hardware performance is not strong enough (like my condition), the coding skills need to be good.

Improvement
We provide some potential methods to further improve our CNN models as follows.

1. Use entire original train image data set provided by State Farm. That is, don't do random sampling for speed or computer performance limitation concern.

2. When preprocessing/resizing the original image, use larger size, e.g., 480 x 480 pixels rather than the current 224 x 224 pixels, to feed the CNN. Note that the original image size is 640 x 480 pixels.

3. Try to use the technique of image augmentation as one data preprocessing step to enhance the scale invariance, rotation invariance, and translation invariance of prediction performance.

4. Fine-tune the CNN model parameters: optimizer, epochs, batch size, number of layers, number of nodes in each layer, filters, kernel size, pool size, strides, activation, dropout, kernel initializer, learning rate, momentum, etc.

5. Try all the popular pre-trained models for transfer learning: VGG16, VGG19, InceptionV3, Xception, ResNet50, etc.

6. When using transfer learning, don't just freeze the parameter weights of the pre-trained models, i.e., don't just use them as fixed feature extractors. Instead, we can re-train the parameters of the whole CNN (or partial higher-level convolutional layers). Note that in fact the data set size of driver images in our problem is quite large, hence we don't need to worry about the overfitting possibility, which often occurs in the smaller data set. We could decide to use the weights of pre-trained models as initial values of parameters to re-train the whole CNN model if the driver images are similar to the images in ImageNet, which are used to train VGG16, ResNet50, etc, or we could directly use random values as parameter initialization to re-train the whole CNN if these images are considered not similar.

i https://www.cdc.gov/motorvehiclesafety/distracted_driving/
ii https://www.statefarm.com/
iii https://www.kaggle.com/c/state-farm-distracted-driver-detection#description
iv https://www.kaggle.com/c/state-farm-distracted-driver-detection/data
v https://www.kaggle.com/c/state-farm-distracted-driver-detection#evaluation
vi http://0rz.tw/XLlUs
vii https://www.kaggle.com/c/state-farm-distracted-driver-detection/leaderboard
viii https://www.kaggle.com/keras/resnet50