Machine Learning Engineer Nanodegree

Capstone Project

Ayşin Taşdelen
October 27th, 2017

I. Definition

State Farm is interested in alleviating the problem of car accidents caused by distracted drivers. According to AAA foundation, the foundation for traffic safety, more than 80% of drivers in the annual AAA Foundation Traffic Safety Culture Index cite distraction as a severe problem and behavior that makes them feel less safe on the road.Distracted driving is a deadly behavior. Federal estimates suggest that distraction contributes to 16% of all fatal crashes, leading to around 5,000 deaths every year. ***AAA's latest research*** has discovered that distraction "latency" lasts an average of 27 seconds, meaning that, even after drivers put down the phone or stop fiddling with the navigation system, drivers are not fully engaged with the driving task.
In this project, I created a web application which predicts the likelihood of what the driver is doing in each picture from the State Farm Kaggle competition dataset.

Project Overview

In this project, I have used training data of the State Farm Kaggle competition data and split it into training, validation and test datasets. Build a multiclass classifier using CNN. Before building my CNN, I was considering using Test dataset from the original competition to test my model. However, those images were unlabelled, and I could not measure my model's performance. Then I had to change my initial intent of using original test dataset. I have used original training dataset to build, test and measured my model; original test dataset to visualize and see how my model is doing on unlabelled images.

I have to build another model using transfer learning to compare my original model. In this new model, I have augmented my data to enhance performance. I have used pre-trained ResNet50 model and added fully connected layers and followed the other steps as I did for my original model.

I created a web application which a user can upload their images and classify them with my both models.The results of those images are printed on the page and user can see which of those ten classes those two models predicted their image as.

Problem Statement

The goal is to implement and train a multi-class classifier that can predict the likelihood of what actual action the driver is doing.

1. Split the original training data (labeled data) into train, validation and test data.
2. Preprocess the data to use in CNN.

3. Train a classifier with a preprocessed training data that can classify the image one of the ten classes.
4. Computing test accuracy of the model.
5. Computing log loss metric.
6. Plot some of the images
7. Predict some unlabelled images
8. Plot some of the unlabelled images.
9. Create a web application that a user can upload their images and see the classification results.

The final application is expected to be useful for determining distracted users and with what are they occupied.

Metrics

I used log-loss function evaluation metric for both models. State Farm Kaggle competition submissions were evaluated using the multi-class logarithmic loss. Each image has been labeled as one true class.

Logarithmic loss (related to cross-entropy) measures the performance of a classification model where the prediction input is a probability value between 0 and 1. The goal of our machine learning models is to minimize this value. A perfect model would have a log loss of 0. Log loss increases as the predicted probability diverge from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high log loss.

Keras also provides a way to specify a loss function during model training. The calculation is run after every epoch. In my case, I select categorical_crossentropy, which is another term for multi-class log loss.

```
 model.compile(optimizer='rmsprop', loss='categorical_crossentropy', met
rics=['accuracy'])
```

II. Analysis

Data Exploration

The dataset consists of imgs folder and driver_imgs_list.csv file. Imgs folder has test and train folder of 640 x 480 jpg files. Each image consists of a driver performing a task from the list below:

c0: safe driving 2489
c1: texting – right 2267
c2: talking on the phone – right 2317
c3: texting – left 2346
c4: talking on the phone – left 2326
c5: operating the radio 2312
c6: drinking 2325
c7: reaching behind 2002
c8: hair and makeup 1911

c9: talking to passenger 2129

Total training data: 22424
Maximum file number is: 2489
Minimum file number is: 1911
Mean is: 2242.4
Median is: 2314.5

Safe driving has the most examples, and it is close to median, so this does not affect our model drastically. Moreover, mean and median is close to each other meaning data is evenly divided around the mean and median is bigger than mean that more of our data is on the right side of the mean. Hair and makeup have the least examples. This case might be because we do not expect all of our drivers to perform this task at all.

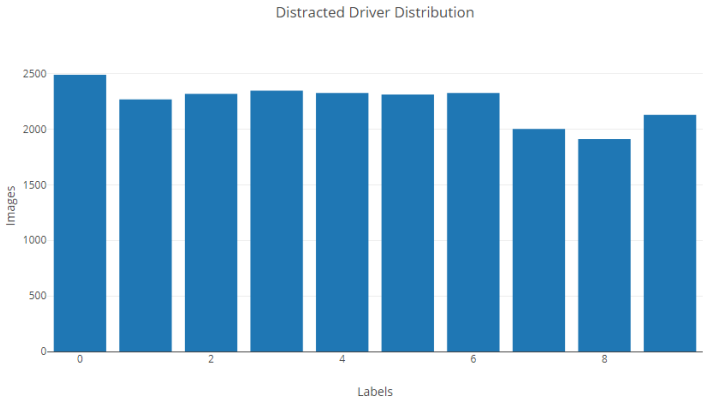
Total testing data: 79726

Total testing data is close to 3 times more than training data, and they are not labeled, and we did not use them to test our model. The size of that set is not important for our model. driver_imgs_list.csv file lists training images, driver names, and tasks. In the figure below, there are 26 drivers. This number seems to be small, but this number might be intentional.

	subject	classname	img
count	22424	22424	22424
unique	26	10	22424
top	p021	c0	img_97080.jpg
freq	1237	2489	1

Exploratory Visualization

I plotted labeled data as shown in the figure below. As we can see here, data is distributed almost like a uniform distribution



Algorithms and Techniques

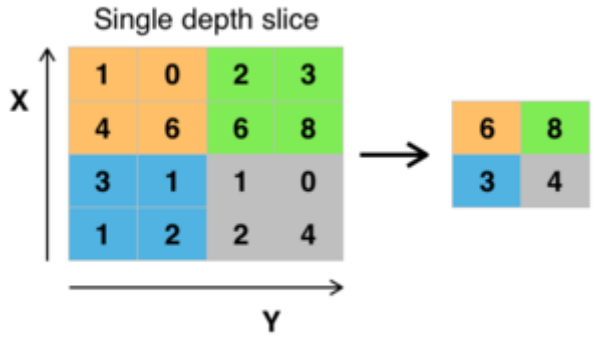
In machine learning, a convolutional neural network (CNN, or ConvNet) is a class of deep, feed-forward artificial neural networks that has successfully been applied to analyzing visual imagery. A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers are either convolutional, pooling or fully connected.

The **convolutional layer** is the core building block of a CNN. The layer's parameters consist of a set of learnable filters (or kernels), which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the width and height of the input volume, computing the dot product between the entries of the filter and the input and producing a 2-dimensional activation map of that filter. As a result, the network learns filters that activate when it detects some specific type of feature at some spatial position in the input.

When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume because such a network architecture does not take the spatial structure of the data into account. Convolutional networks exploit spatially local correlation by enforcing a **local connectivity** pattern between neurons of adjacent layers: each neuron is connected to only a small region of the input volume.

Three hyperparameters control the size of the output volume of the convolutional layer: the depth, stride and zero-padding.The depth of the output volume controls the number of neurons in a layer that connect to the same region of the input volume. Stride controls how depth columns around the spatial dimensions (width and height) are allocated. Sometimes it is convenient to pad the input with zeros on the border of the input volume. The size of this padding is a third hyperparameter

Pooling layer Max pooling(with a 2x2 filter and stride = 2) is a form of non-linear down-sampling. It partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum. The intuition is that the exact location of a feature is less important than its rough location relative to other features. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting.



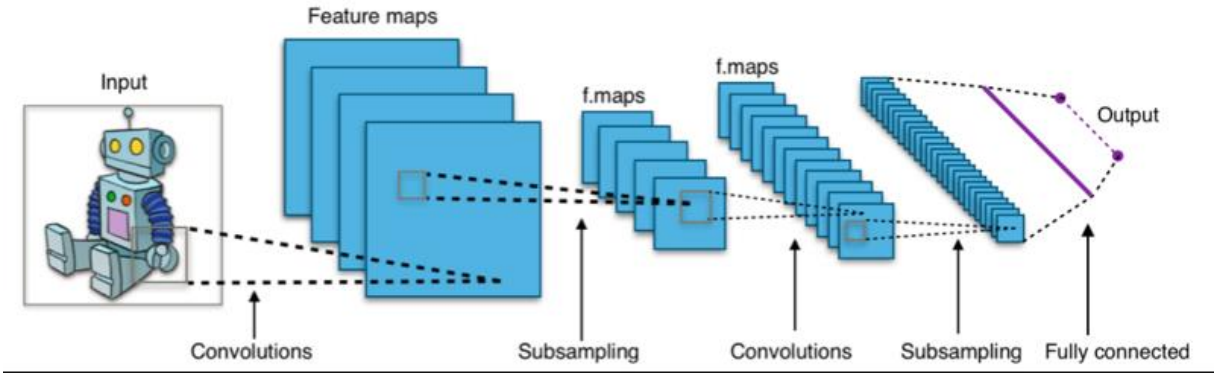
Max pooling with a 2x2 filter and stride = 2

ReLU layer ReLU is the abbreviation of Rectified Linear Units. This layer applies the non-saturating activation function . It increases the nonlinear properties of the decision function and of

the overall network without affecting the receptive fields of the convolution layer.. ReLU is preferable to other functions, because it trains the neural network several times faster without a significant penalty to generalisation accuracy.

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

Loss layer The loss layer specifies how training penalizes the deviation between the predicted and true labels and is normally the final layer. Various loss functions appropriate for different tasks may be used there. **Softmax** loss is used for predicting a single class of K mutually exclusive classes.



Typical CNN Architecture

In this project, my classifier is a Convolutional Neural Network, which is the state-of-the-art algorithm for most image processing tasks, including classification. My target is to train a multi-class classification model using CNN.

```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

model.add(Conv2D(filters=64, kernel_size=2, padding='same',activation='relu',input_shape=(224, 224,1)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32,kernel_size=2, padding='same',activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=128,kernel_size=2, padding='same',activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32,kernel_size=2, padding='same',activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=16,kernel_size=2, padding='same',activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0,3))
model.add(Flatten())
model.add(Dense(8, activation='relu'))
model.add(Dense(10, activation='softmax'))

model.summary()
```

I have 5 convolutional, 5 pooling, 1 dropout,1 flatten and 2 dense layers. The first convolutional layer is the input layer. I defined input shape as (224,224,1) which means my model is expecting 224-pixel wide 224-pixel high grayscale images. The computer interprets grayscale images as a 2D array and my features or not correlated with color; thus, it would be simpler to use grayscale images, and our “relu activation” calculation for a single filter is 1 /3 of the color images. Other convolutional layers

help us extract more features from the images. I increased filter numbers from beginning to 3rd convolutional layer to detect more patterns in the images because we have sophisticated image patterns to detect like driver reaching behind. The number of total filters is high, and dimensionality of our convolutional layers can get quite large. Higher dimensionality means, we need to use more parameters which can lead to overfitting.Therefore, we need a method reducing dimensionality. For this purpose, I used pooling layers after each convolutional layers. Pooling layers take a stack of feature maps as input and reduce each feature layer by using 2*2 window calculating the maximum of the pixels contained in the window starting from top left to bottom right.

Before building the model, I had to preprocess the data which I explained in depth in Data preprocessing section. Then I built my CNN model and run test and computed its accuracy. Then, I have used transfer learning to build another multi-class classifier to benchmark my model. After fitting each model, I have plotted some random test data and then predict unlabelled data and plot some random unlabelled data predictions.

Benchmark

When it comes to image classification, the only method better than CNN seems to be “Transfer learning.” Taking a pre-trained model like ResNet remove last fully connected layers, define the input-output shape and train new model.

I used *categorical_crossentropy,* and validation loss only improved the first epoch. Test accuracy is low, and log loss is too high. Because Log loss increases as the predicted probability diverge from the actual label, with a log loss 30, this model is not useful to predict any of the 10 classes I intend to classify.

```
# evaluate and print test accuracy
score = new_model.evaluate(X_test, y_test, verbose=0)
print('\n', 'Test accuracy:', score[1])
```

```
Test accuracy: 0.102564102564
```

```
from sklearn.metrics import log_loss
#Calculate logloss
log_loss = log_loss(y_test, y_hat)
print('\n', 'Log loss:', log_loss)
```

```
Log loss: 30.9963065421
```

III. Methodology

Data Preprocessing

Until now, We have talked about given data. Before running our CNN model, we selected labeled data (initially training) to train, validate and test our model. There are 10 total categories,102150 total images, 22424 labeled images and 79726 unlabelled images.

```
# load datasets. labelled files will be used
labelled_files = load_labelled_dataset('imgs/train')
unlabelled_files = load_unlabelled_dataset('imgs/test')

# load list of class names
categories = [item[20:-1] for item in sorted(glob("imgs/train/*/"))]

# print statistics about the dataset
print('There are %d total categories.' % len(categories))
print('There are %s total  images.\n' % len(np.hstack([labelled_files, unlabelled_files])))
print('There are %d labelled images.' % len(labelled_files))
print('There are %d unlabelled images.'% len(unlabelled_files))
```

```
Using TensorFlow backend.
```

```
There are 10 total categories.
There are 102150 total  images.

There are 22424 labelled images.
There are 79726 unlabelled images.
```

We have matched the labeled class names with labeled images. Read the labels from the csv file and match them with the images.

In [2]:
```
import pandas as pd

#read image names and labels from csv file
def get_driver_data():
    df = pd.read_csv('driver_imgs_list.csv')
    return df

import numpy as np

#matches labels to images
def match_labels_images(img_paths):
    df = get_driver_data()
    images = []
    labels = []
    for img_path in img_paths:
        image = img_path.split("/")
        im_name = image[-1]
        label = df['classname'][df['img']==im_name]
        str_label = str(label)
        images.append(img_path)
        labels.append(str_label)
    np_images = np.array(images)
    np_labels = np.array(labels)

    return np_images, np_labels

X_labelled, y_labelled = match_labels_images(labelled_files)

print(X_labelled.shape[0], 'labelled samples')
```

```
22424 labelled samples
```

I have split labeled data into train, test and validation sets. I have split my data approximately 60% train; 20% test and 20% validation.

```
from sklearn.model_selection import train_test_split

# split labelled data into train,validation and test sets
X_trainVal, X_test, y_trainVal, y_test = train_test_split(X_labelled, y_labelled, test_size=0.2)
X_train, X_validation, y_train, y_validation = train_test_split(X_trainVal, y_trainVal, test_size=0.2)

# print number of training and test images
print(X_train.shape[0], 'train samples')
print(X_validation.shape[0], 'validation samples')
print(X_test.shape[0], 'test samples')
```

```
14351 train samples
3588 validation samples
4485 test samples
```

Reshape the images and labels to use as input to my model.

```
from keras.utils import np_utils

def reshape_labelled(X , y):

    data = np.array(paths_to_tensor(X), dtype=np.uint8)
    target = np.array(split_target(y), dtype=np.uint8)
    target = np_utils.to_categorical(target, 10)

    #Rescaling images by dividing every pixel in every image by 255
    data = data.astype('float32') / 255

    return data, target

def reshape_unlabelled(X):

    data = np.array(paths_to_tensor(X), dtype=np.uint8)

    #Rescaling images by dividing every pixel in every image by 255
    data = data.astype('float32') / 255

    return data

X_train, y_train = reshape_labelled(X_train , y_train)
X_validation, y_validation = reshape_labelled(X_validation , y_validation)
X_test, y_test = reshape_labelled(X_test , y_test)

#we select some sample of unlabelled data to plot and see how our model does on unlabelled data
X_unlabelled = reshape_unlabelled(unlabelled_files[:100])

# print shape of training, validation and test.
print('X_train shape:', X_train.shape)
print('X_validation shape:', X_validation.shape)
print('X_test shape:', X_test.shape)
# print number of training, validation, and test images
print(X_train.shape[0], 'train samples')
print(X_validation.shape[0], 'validation samples')
print(X_test.shape[0], 'test samples')

# print shape of unlabelled set
print('X_unlabelled shape:', X_unlabelled.shape)

# print number of unlabelled images
print(X_unlabelled.shape[0], 'unlabelled samples')
```

After that operation, my dataset is ready to train validate and test my model.

```
X_train shape: (14351, 224, 224, 1)
X_validation shape: (3588, 224, 224, 1)
X_test shape: (4485, 224, 224, 1)
14351 train samples
3588 validation samples
4485 test samples
X_unlabelled shape: (100, 224, 224, 1)
100 unlabelled samples
```

Different than the original model, I augmented data before running the benchmark model to improve its performance. Because I thought transfer learning would do better and with the augmented data I was expecting it would do even better, so I used augmented data on my benchmark model.

```
from keras.preprocessing.image import ImageDataGenerator

# create and configure augmented image generator
datagen = ImageDataGenerator(
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True)

#Augment the training data
datagen.fit(X_train)
```

Implementation

The implementation process contains three main stages:

1. CNN classifier training, testing, and evaluation
2. Transfer model training, testing, and evaluation
3. The web application development

During the first stage, the classifier trained on the preprocessed training data. Training run in a Jupyter notebook (titled “Capstone_CNN”), and contains the following steps:

1) Load both the labeled and unlabelled images into memory, preprocessing them as described in the previous section
2) Implement helper functions:
   a) load_labelled_dataset(path): load labelled data
   b) load_unlabelled_dataset(path): load unlabelled data
   c) get_driver_data(): read image names and labels from csv file
   d) match_labels_images(img_paths): file matches labels to images
   e) path_to_tensor(img_path): and paths_to_tensor(img_paths): convert 3D tensor to 4D tensor with shape (1, 224, 224, 1) and return 4D tensor
   f) split_target(y_labels): reshape label values into numerical values 0,1,2,3,4,5,7,8 and 9
   g) reshape_labelled(X , y): used to reshape train, validation and test datasets
   h) reshape_unlabelled(X):used to reshape unlabelled data
   At the end of datapreprocessing steps, I have (1, 224, 224, 1) shaped images and size 10 array labels.
3) Define the network architecture and training parameters
4) Evaluate and print test accuracy
5) Get predictions on the test set
6) Calculate log-loss
7) Plot a random sample of test images, their predicted labels, and ground truth
8) Get predictions on the unlabelled set and plot random samples

During the second stage, the transfer learning classifier trained on the preprocessed training data. Transfer learning run in another Jupyter notebook (titled “Capstone_Transfer_Learning”), and contains the following steps:

1) Load both the labeled and unlabelled images into memory, preprocessing them as described in
   the previous section
2) Implemented helper functions are the same with the CNN model. Different than CNN model I augmented data to enhance performance. At the end of data preprocessing steps, I have (1, 224, 224, 3) shaped images and size 10 array labels. IN this model images are left RGB because ResNet model used RGB images and leaving input images as RGB would increase overall model performance.
3) Define the network architecture and training parameters
4) Evaluate and print test accuracy
5) Get predictions on the test set
6) Calculate log-loss
7) Plot a random sample of test images, their predicted labels, and ground truth
8) Get predictions on the unlabelled set and plot random samples

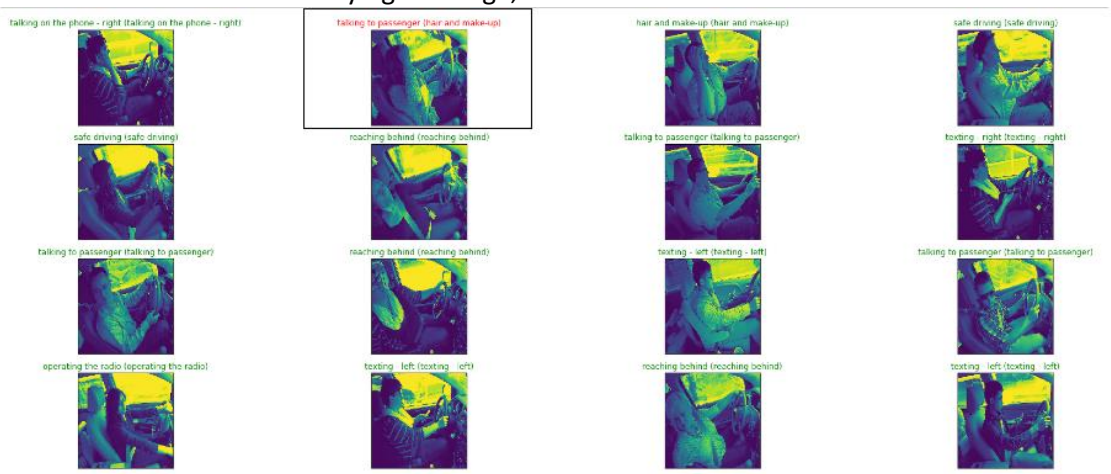
During the third stage, I designed a web application using C# (titled “Capstone”) following steps:

1) A simple .aspx page is designed for users to upload their images (Default.aspx), call the trained classifiers and display the results of each model's predictions.

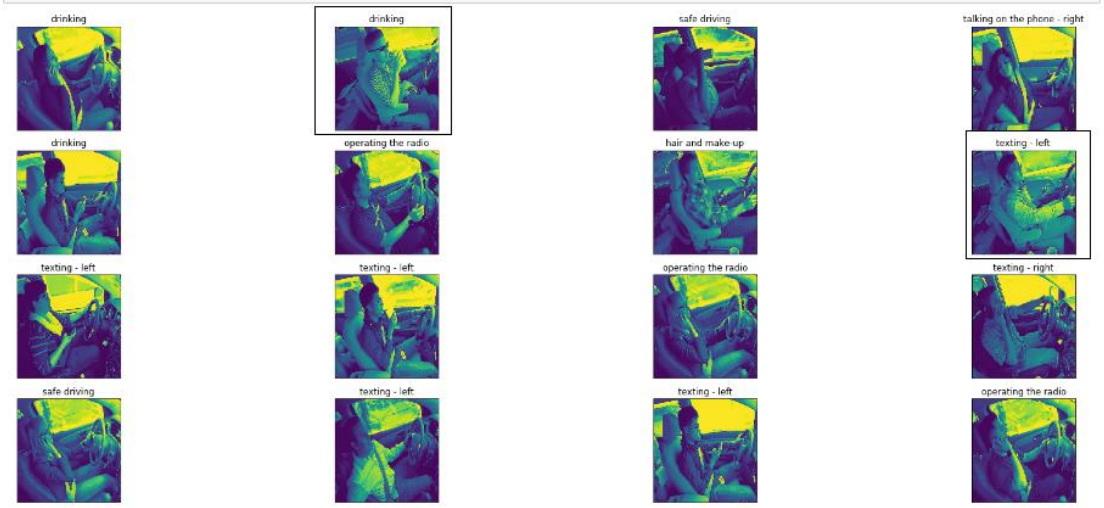
2) `string RunPython(string pythonFileName)` function which calls each model predict methods.

Refinement

My CNN model performs well regarding model's Test accuracy: 0.921070234114 and log loss result: 0 .321802023058. After plotting random test results. We see only one out of sixteen results shows our model make a mistake classifying an image, but other fifteen is successful.



With 92% test accuracy my model seems to be trustworthy but after predicting some unlabelled images and plotting them. My model predicts two out of sixteen images successfully.



That increase the probability of overfitting. The model can be fine-tuned looking at the unlabelled images.

After couple of times reconstructing the model and re running, my model's performance improved. The improved model predicted 3 out of sixteen images' true labels.I used StratifiedShuffleSplit before reconstructing my model and I added more dropout layersto prevent overfitting. To find the dense

solution I added some more Convolutional layers and an additional dense layer . My model's accuracy improved from 92% to 94% and I can see the change on unlabelled data.

I trained a transfer learning model to compare and benchmark my model, but benchmark model did worse than my model. Maybe benchmark model can be considered as a separate model and fine-tune its parameters.

IV. Results

Model Evaluation and Validation

My refined model is better than the initial model. It is more robust than initial model but it is not the perfect model because it does not predict half of the true labels of the unlabelled data. The model is not robust enough because it does poorly on unlabelled images. If I trust the result of the model, I would expect the model to predict the correct class of unlabelled images from one out of ten classes, but it did not. Therefore, this model needs to be improved.

To refine the model, I have augmented data and added more dropout layers to my model. But, outcomes were worse than initial model. Then ,I totally removed the augmentation section and used StratifiedShuffleSplit to split my data into train validation and test which shufles and splits the data Which would be good idea to prevent over fitting.

To visualize my model performance I've both tried using TensorBoard and model_to_dot from keras.utils.vis_utils ; However, both tools require pydot when displaying results and my environment is python3.6 which is not compatiable with pydot. I managed to install pydot3 but keras does not use pydot3 behind.

	Initial Model	Refined Model
Data splitting	`train_test_split`	`StratifiedShuffleSplit`
Convolutional Layer Size	5	6
Dropout layer Size	1	3
Maxpooling layer size	5	3

Dense layers	2	3
Flatten Layers	1	1
`input_shape`	`(224, 224,3)`	`(224, 224,1)`
`Test accuracy`	`0.921070234114`	`0.946046373365`
`Log loss`	`0.321802023058`	`0.773139418623`

Log loss of initial model is close to 0 which means it looks like the better model. However, this better number can be because of overfitting , numbers show better results but unlabelled data does not.

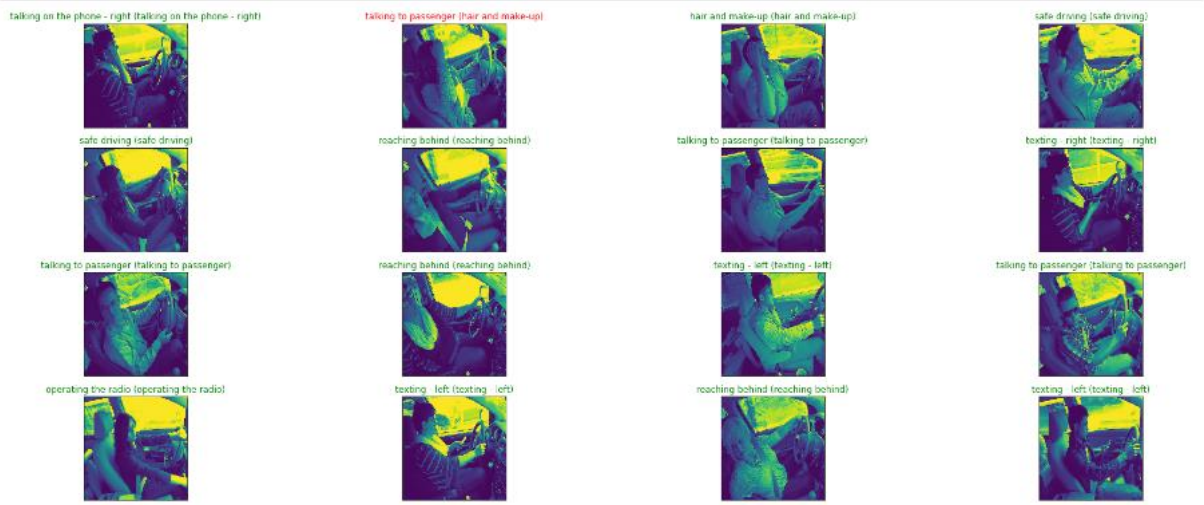
Justification

My CNN model refinement should be contuniued. 2% of improvement is a start but not enough. My benchmark model did even worse than the initial model. Those two models can be considered as seperate models and both can be improved.

V. Conclusion

Free-Form Visualization

The below plot shows results of my CNN model predictions on test data and correct labels. Only one of the prediction predicts the wrong label others are correct.



Plot below shows my refined model prediction on the labeled data. From the below plot it seems my refined model does worse but on the unlabelled data refined model performs better.

The below plot shows results of my Transfer Learning model predictions on test data and correct labels. Only one of the prediction predicts the label correctly others are false.

Reflection

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

1. An initial problem and relevant, public datasets found
2. The data was downloaded and preprocessed
3. The CNN classifier trained, tested and log loss calculated
4. Transfer learning classifier trained, tested and log loss calculated.
5. Web application created

Training models took at least two and a half hours each, and I could not run them on my local computer. I have used GPU compute instance to train my models, test them and calculate their accuracies using Jupyter notebooks. I wanted to create a web application I had to create it on my local machine. Because I could not run those two on the same computer, I could not test my project end to end.

Improvement

Initially, I was expecting CNN to do good and generate the best results with the unlabelled data. However, everything seems to be fine until I saw how my model done on the random unlabelled images. This model can be fine-tuned by changing parameters convolutional layers or augmenting the data or using kfold validation. In this project, my CNN is not the only thing to be improved. I expected transfer learning to do even better than my CNN.Those pre-trained models can detect more features than my small CNN. There might be other ways to construct a transfer learning architecture than mine. Also, other pre-trained models might be more useful than ResNet50. All of those probabilities can be constructed and compared by log loss function and predict random unlabelled data and visualize.

I look images as whole in this project but two local places are really important for me, people's hands and their head area using Feature Matching + Homography to find Objects like hand or head would increase my model's performance.

One minor detail to be improved I could not test my web application's predict method because I could not run my models locally.

References

https://github.com/fchollet/keras/issues/4465 *(transfer learning)*

http://wiki.fast.ai/index.php/Log_Loss *(Los loss function definition)*

https://plot.ly *(used to generate visual)*

http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html *(shuffle test split for refinement section)*

https://keras.io/preprocessing/image/ *(Data Augmentation)*

http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedShuffleSplit.html *(Stratified ShuffleSplit)*

http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html *(Improvement)*

https://www.kaggle.com/the1owl/object-expression-farmer (*Idea about improvement)*

https://en.wikipedia.org/wiki/Convolutional_neural_network *(Definition of Convolutional Neural Network)*