Machine Learning Engineer Nanodegree

Capstone Report

Michael Stevens

Octover 22th, 2018

Problem Definition

Proposal

Additional material for this project is available here, including the proposal.

https://github.com/mstevenscmu/capstone-project

The original proposal review is here: https://review.udacity.com/#!/reviews/1437790

Project Overview

The StateFarm Kaggle Competition represents an important application of machine learning classification to image data. Distracted driving is a large contributor to property damage and personal injury. By adding a camera facing the operator of a motorized vehicle, it may be possible to recognize when the operator is distracted. This could provide an opportunity to add a safety system similar to the audible chime when the seatbelts are not in use. The identification of a distracted operator could be used to remind the operator, alert other drivers with a signal, alter insurance billing, or be used as an input into a self-driving auto-brake system.

This problem appears to be feasible to solve using the skills taught in the Udacity MLND. This problem has not only been the subject of the Kaggle competition, but also the subject of academic projects. Other research has shown the ability of deep neural networks to perform pose estimation on subjects. These projects show that this task is possible to learn using machine learning.

- Kagggle StateFarm Distracted Driver Detection
    - https://www.kaggle.com/c/state-farm-distracted-driver-detection
- End-to-End Deep Learning for Driver Distraction Recognition (Koesdwiady)
    - https://www.springer.com/cda/content/document/cda_downloaddocument/9783319598758-c2.pdf?SGWID=0-0-45-1608335-p180889205
- DarNet: A Deep Learning Solution for Distracted Driving Detection
    - https://users.cs.duke.edu/~cdstreif/static/darnet_presentation.pdf
- DeepPose: Human Pose Estimation via Deep Neural Networks (Toshev)

- https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42237.pdf

Problem Statement

The problem is to classify images of automobile operators into one of ten possible classes. One of the classes represents normal driving and the other nine represent distracted driving. The classification algorithm to solve this problem will output probabilities for each possible class. The performance of the classification will be measured using accuracy and multi-class logarithmic loss.

This classification problem has several distracted classes that should be possible to classify with neural network deep learning systems, such as using a cell phone or reaching into the back seat. With a vast amount of training data it would be possible to train a network from scratch to classify novel images into the ten classes. With less labeled training data, it will require using transfer learning on bottleneck features of a pretrained neural network.

The full list of classes is the following.

- 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

To solve this problem, a Neural Network classifier will be constructed that takes input images and outputs the probabilities of the image belonging to each class.

```
class_probabilities = predict(input_image)
```

The implementation of this prediction function can be thought of as a large system of matrix operations. Instead of creating a neural network from scratch, a pretrained Resnet50 network will be the front of the network. A pretrained neural network has learned how to detect relevant features for image classification from the imagenet database. Those learned feature detectors eliminate the need for feature engineering and will provide meaningful signal from the input image. A pretrained network has created feature detectors that would not be possible without either vast amounts of training data or many epochs of training. The Resnet50 network was trained on 1.28 million images, which is significantly larger than approximately 20,000 labeled images in the StateFarm dataset.

The individual weights for each matrix will be learned through a training process where images with a known class probability are input into the predict function. An error value for the prediction will be computed using the log loss error metric that penalizes both misprediction and uncertainty in predictions. The error of the prediction will then be used by an optimizer to modify the weights

backwards through the matrix in a back-propagation algorithm to reduce the error. The Stochastic Gradient Descent and Adam optimizers will be used for this project.

This training process will be executed for several iterations called epochs. An epoch of training corresponds to each training image being passed through the predictor and an error computed. Epochs are divided into batches where the weights are updated after batch_size samples have been passed through the networks.

The final result of the training process will be a neural network that has optimized coefficients that minimize the error when presented with training samples. If the network generalizes well to unseen data, it should be able to output class probabilities with similar low log loss error values for the previously unseen images.

- http://wiki.fast.ai/index.php/Log_Loss
- https://en.wikipedia.org/wiki/Stochastic_gradient_descent
- https://keras.io/optimizers/#sgd
- https://keras.io/optimizers/#adam
- https://www.kaggle.com/keras/resnet50

Metrics

The two metrics that will be used for this project are accuracy and multi-class log loss. Accuracy is an appropriate metric for understanding performance because the data set does not contain skewed classes.

Accuracy = #Correct-Predictions/#All-Predictions

The multi-class log loss is a more sophisticated examination of the results and provides insights into how confident the predictions of class are.

This multi-class log loss metric is a standard for multi-class classifiers when the output is a predicted probability for each class. This is what the Kaggle competition originally used and is definitely suitable. Log loss takes into account the certainty result for the prediction. A standard accuracy measurement only accounts if the top prediction matched the actual label. In the case of a 10-class classification problem the actual class could be predicted with 0.1 + epsilon certainty or 1.0 certainty, but both would be counted the same by an accuracy score. Log-loss would rate the 1.0 certainty prediction much better than the 0.1 + epsilon prediction. A correct prediction of class A with 1.0 probability will score better on log loss than a prediction of A with 0.5, B with 0.25 and C with 0.25.

Log loss heavily penalizes mispredictions with a low predicted probability with extremely high loss values. Correct predictions with absolute certainty score close to zero. The undefined values of the formula for zero and one certainties are compensated for by applying a minimum and maximum to ensure the p values are within (0+epsilon, 1-epsilon).

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

- http://wiki.fast.ai/index.php/Log_Loss
- http://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html#sklearn.metrics.log_loss
- http://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html

Consequences of misclassification depend on how the classifier would be used. If the system was used to sound a warning chime or apply brakes, users might be more accepting of a bias more towards classifying the image as safe driving to avoid unwanted chimes or braking. If the system was used as a prescreen for humans reviewing videos of driver behavior, such as monitoring a school bus driver, it might be preferable to allow more false positives. A human could review the video and determine if it was really distracted driving.

Analysis

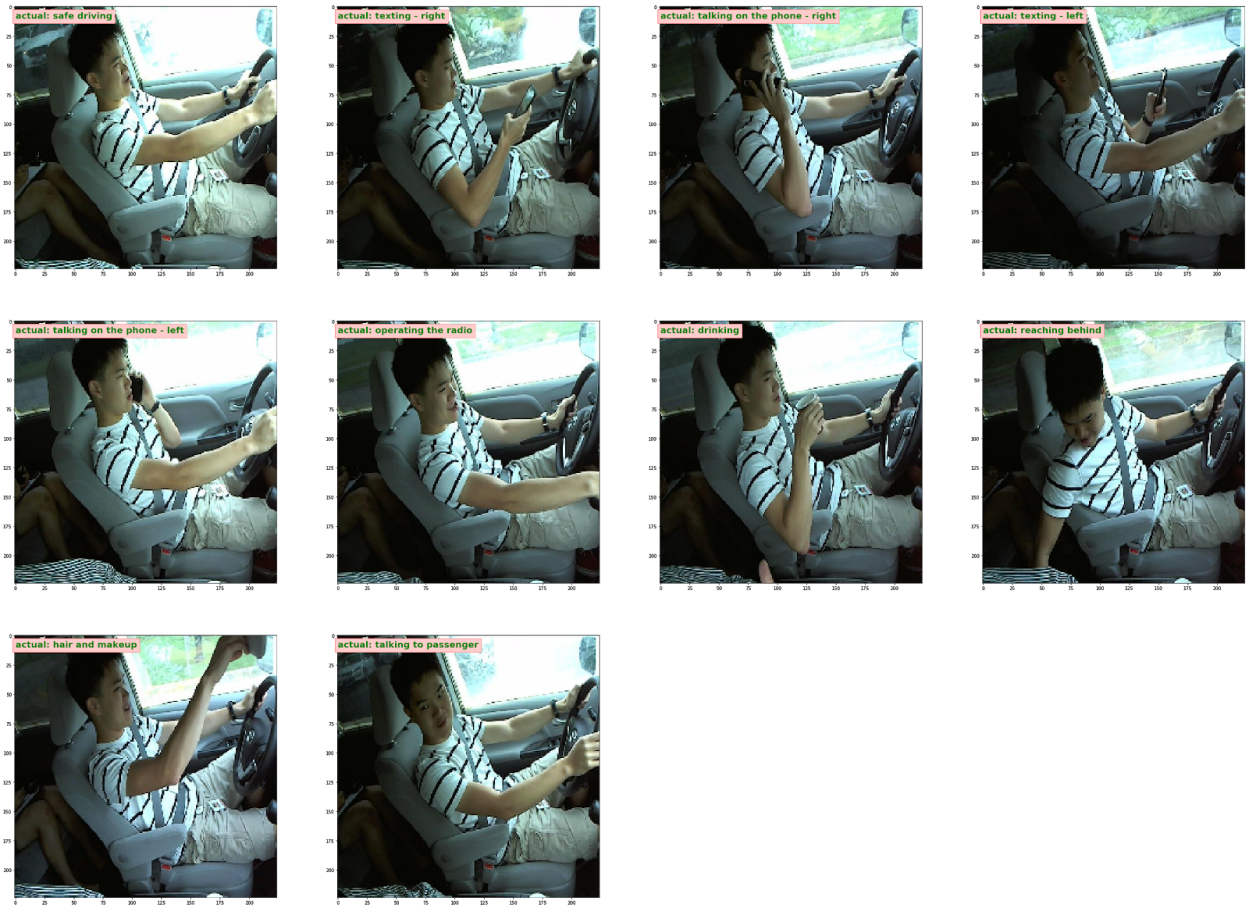
Data Exploration

The dataset for this problem was acquired by StateFarm. The drivers captured represent a mix of genders, clothing, eyewear, ethnicity, and weights. Several vehicles are represented in the training dataset. A number of the images show the drivers wearing StateFarm badges, so they may have used their employees for a majority of their captures. The subjects captured were not actually operating the vehicle because the car was being towed behind a truck. In some images, a person with a clipboard is visible in the back seat. Each labeled class has between 1911 and 2489 images.

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

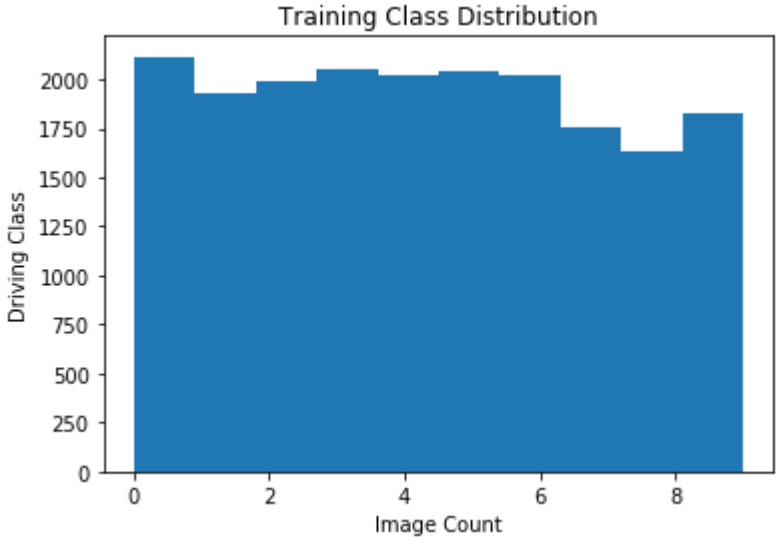
The training dataset is organized into ten folders, with one representing each class. Each image is a 640x480 pixel jpeg captured from a location near the passenger side A-pillar. A CSV file provides a driver identifier, class, and image name for each image in the training folder. A separate folder called test has 79,726 images, but no labels are provided as a part of the download, so they will not be used. Instead, the images in the test folder will be divided into train, validate, and test sets by splitting on the driver id.

Example images for each of the driving classes are shown here. Some clear potential challenges are already visible, such as very little obvious difference between operating the radio and normal driving. Hair/makeup and talking to passenger also have potential to be confused with checking in a mirror.



The 22,424 labeled images have a driver ID and driving behavioral class for time image. For this project, the labeled training data was split into train, validate, and test sets. Since the images represent captures from video, there are images that are very similar and would represent leakage across sets. To avoid this leakage, the sets are split by driver so that all images for a driver are only a member of one of the train, validate, or test classes. A total of 26 drivers are present in the dataset. 22 were used for training, 1 for validation, and 3 for test. This resulted in 19407 training images, 1034 validation images, and 1983 test images.

The distribution of labeled training data is evenly distributed across all of the driving classes. There is no presence of skewed classes in the labeled data, but a real world application would expect to see far more examples of normal driving than distracted driving.



a

Augmentation is frequently used to increase the amount of training data when building deep learning classifiers. Augmentation techniques of flipping and rotating images are very useful for training a classifier to recognize a cat in many possible orientations when only a single orientation is present. For the StateFarm competition the camera is always positioned in a fixed position, so flipped or rotated training images will not provide much value in training the network. Other augmentations such as lighting changes or non-affine transformations to simulate other camera lens may be useful.

Algorithms and Techniques

Transfer learning using a pretrained Keras network was used for the project. My experience with using Resnet50 in prior MLND projects has shown that it has potential to perform well in multi-class image classification problems. The pretrained model was configured in Keras to not be trainable and additional layers appended to it that were trainable.

A Convolutional Neural Network will be used to make predictions. A CNN gets the name from the NxN block, called a kernel, that scans across the span of the image. This scanning process with a smaller block allows the network to recognize features such as edges or curves no matter where they occur in the image. In addition to convolutional layers, there are a number of other layers such as MaxPooling, Dropout, and Dense layers. Each layer can have an activation function, such as ReLU. The ReLU function zeros any negative values and is linear for positive values. The slope is easy to compute for training it is either zero or one.

The MaxPooling layers reduce the dimensionality of the input layer by picking the maximum activation from a region of the input. This provides a lower resolution summary of if the Convolutional Filter found the pattern it was looking for. A maxpooling layer allows relative positions of features represented by the maximal activation in a region to be passed on to futher layers.

Dropout layers are primary used for ensuring the dense layers at the end of a neural network do not overfit. It works by randomly dropping a set percentage of nodes during training and reducing the output of all nodes by a percentage during prediction.

The Dense fully connected layers are the end of the network are where the reasoning about previous higher level features is transformed into a classification.

The final layer is a Softmax which outputs a set of probabilities that sum to 1 and are used as class prediction probabilities.

ResNet50 is a very deep CNN which was created by using a novel mechanism of propagating residuals forward in parallel to a block of layers to avoid the vanishing gradient problem for back-propagation in very deep networks. It effectively presents both F(x) and x for the prior layer as the input to the next layer. ResNet50 was trained on the ImageNet database, which should have images with similar high level features to the StateFarm driving images, making transfer learning from this network an appropriate choice.

During training, additional callbacks were provided to the fit function to checkpoint the model at the best point and to stop early if no progress was being made in training.

The training was done for up to 20 epochs at a batch size of 20. Computing the error in batches improves the training speed, but can reduce the accuracy and increase the loss versus using smaller batches. Stochastic Gradient Descent attempts to find the differential of the error function and tries to adjust the weights to minimize the computed error.

Plots of the accuracy and log loss after each epoch were generated to understand the impact of additional epochs of training.

Predictions were made by selecting the class with the highest probability. This was necessary for accuracy computations. Log loss was computed using the raw probabilities predicted for each class.

Benchmark

A benchmark model for distracted driving would be to always predict normal undistracted driving with complete certainty, 1.0, and all distracted driving classes as 0.0. The predictions will then be measured with accuracy and multi-class logarithmic loss. This benchmark should achieve slightly better than 10% accuracy, but a poor log-loss score. Beating this benchmark's log loss is a bare minimum. The Kaggle competition also provides other benchmarks using the leaderboard with the log loss of the winning entries being lower than 0.10.

Methodology

Data Preprocessing

Several steps were required to prepare the data for usage with a transfer learning model. The 640x480 images were resized to the input dimensions of the pretrained neural networks. The RGB 0->255 values was also converted to a floating point format compatible with the neural network's input expectations.

The data was also divide into train, validate, and test sets by driver identifier to prevent leakage from train to test sets where a test image was just the next frame in a video of a training image. The similarity of some test images to each other because they are frames from a video could cause high accuracy and low loss for training.

A cursory scan of file sizes was done to make sure there were no outliers such as an empty image that compressed to an unusually small size.

Significant effort in preparing and cleaning the dataset was done by StateFarm to ensure suitability for the contest. In addition, when images were displayed for examination of data or printing of misclassifications, the labels were added to allow checking that the actual label was represented what was contained in the image. No errors were noticed during the runs.

Implementation

This project is implemented using the Jupyter notebook system. This interface is superior to a normal editor because it integrates a live Python interpreter that continues running while making edits. Keeping the results of evaluated expressions in scope while editing allows faster feedback loops and the creation of a process for rapid exploration. The ability to show inline visual depictions allows seamless interpretation of complex results inside the combined editing and execution environment.

The Python language is used for the project primarily for the large amount of easy to use libraries available for machine learning. The interpreted nature versus compiling allows for faster iteration during exploration and design phases for machine learning projects. The computationally intensive numerical processes for training and prediction are offloaded to libraries not implemented using python. For these computationally intensive operations, python can be thought of as a control-plane for the linear algebra libraries written in C or the offloaded computation to a GPU. This allows a program to have the flexibility of an interpreted language but still have access to the speed of lower level languages, making it a great choice for machine learning.

The Python numpy library is used extensively through the project. This numerical computation library provides low level array structures that are used by many libraries. It also provides useful utility functions that perform operations such as converting the dimensions of arrays, returning the position of the maximum value, and filtering arrays for matching elements.

Pandas is a generic analysis library that builds on top of numpy concepts and structures. Pandas is used to read the csv file that maps images to driver numbers. Pandas allows for database style operations on the dataframe structure it provides, simplifying many sophisticated operations to a single line of code.

For creation and training of a neural network multi-class predictor function, this project used the Keras library. The goal of Keras is to provide a set of objects in native python that have an obvious mapping to the structure used to describe neural network construction. The interface provided avoids creation of a separate domain specific language and leverages the familiarity of Python. The interface provides an information dense description with very little unnecessary verbosity. The following code demonstrates the concise way that Keras allows neural network models to be expressed.

```
model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=100))
model.add(Dense(units=10, activation='softmax'))
```

Keras is a developer-friendly interface to a lower level library called Tensorflow. This is a library that is designed for the dataflow computations of linear algebra used to create the structure of neural networks. Tensorflow is where the majority of the compute intensive operations happen and has GPU offloading to accelerate the computation.

This project implemented Keras models by having functions return a configured Keras model object instance. This allowed saving the apparatus for the previous experiment while iterating with new models. In the following example the ResNet50 network is loaded from the Keras applications. It is loaded with the imagenet weights from training on 1+ million images. The fully connected layer is not included, which prepares it for being attached to a new custom network so it can be leveraged for feature extraction. The layers of ResNet50 are then set to be untrainable. The attached network is a simple arrangement of a flatten to convert the tensor shape, a dropout layer to reduce overfitting, and a final set output nodes with a softmax activation for the predicted probabilities. This ensures the output values are in the range of 0,1 and sum to 1. A name is returned with the model to use in plotting and result archival.

```
def model_v1():
    pretrain_model = applications.ResNet50(weights = "imagenet",
                                           include_top=False,
                                           input_shape = (img_width, img_height,
3))
    train_layers = 0
    for layer in pretrain_model.layers[:-train_layers]:
        layer.trainable = False
    x = pretrain_model.output
    x = Flatten()(x)
    x = Dropout(0.5)(x)
    predictions = Dense(10, activation="softmax")(x)

    model = Model(input = pretrain_model.input, output = predictions)
    return model, "resnet50-flat-drop0.5"
```

Training of a network requires a configured optimizer with a loss function to optimize. This preservation of experiment apparatus is done in a similar way by capturing the optimizer parameters in a function and returning a string representing the configuration.

```
def optimizer_1(model):
    lr=0.0001
    momentum=0.9
    opt="SGD"
    model.compile(loss = "categorical_crossentropy",
                  optimizer = optimizers.SGD(lr=lr, momentum=momentum),
                  metrics=["accuracy"])
    return "opt={},lr={},momentum={}".format(opt,lr, momentum)
```

Training of the network also requires configuration of epochs and the sizes of batches within the epochs. A similar function captures this by returning batch size and epochs. This function demonstrates the maximum batch size supported by my machine without failing on out of memory errors and only a single epoch. This fast training is useful for verifying a model functions and generates predictions.

```
def train_params_fast():
    return 75, 1
```

The training is then accomplished by calling the fit method on the model. The training tensors and categories are provided, along with the configured number of epochs and batch size within the epoch. Callbacks were configured to save the model at each step where an improved validation error metric was achieved and to stop training if no progress was made for a few epochs.

```
    history = model_final.fit(x=train_tensors,
                              y=train_classes_categories,
                              batch_size=batch_size,
                              epochs=epochs,
                              verbose=1,
                              validation_data=(validate_tensors,
validate_classes_categories),
                              callbacks = [checkpoint, early]
                              )
```

Plots of the accuracy and log_loss for train/validate were plotted per epoch and instrumental in understanding the progress of training. The plots were generated using the pyplot library. These graphs were useful in understanding the progress of the neural network training. These were saved to files in the analysis directory using a filename containing the description of the experiment parameters. The title of the graphs also included the parameters to ensure usability in reports.

Finally, the model was loaded with the best weights from the training process and the generalization of the model was tested using a collection of images that was not used in the training process. Predictions for these previously unseen images were made and evaluation metrics of accuracy and log loss were computed using the sklearn library.

```
# Load the best weights found during the experiment instead of leaving
# the final weights from the last training epoch.
model_final.load_weights("weights/{}.h5".format(model_desc))

# Generate test predictions using the best weights for the
# experiment
test_predictions = model_final.predict(x=test_tensors)
```

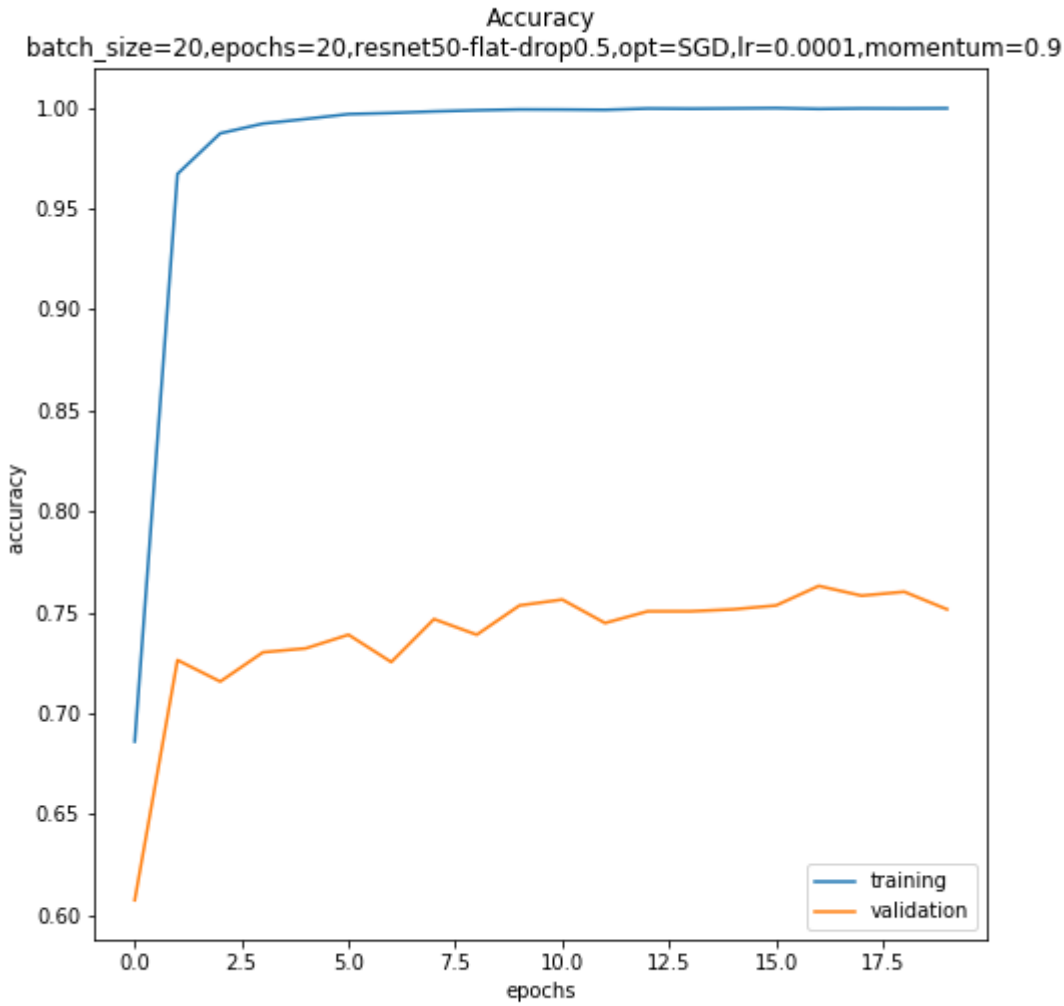
A confusion matrix was plotted using seaborn and was useful for understanding patterns in misclassifications. Additionally, sample images from the set of misclassifications were displayed with the true and predicted classes.

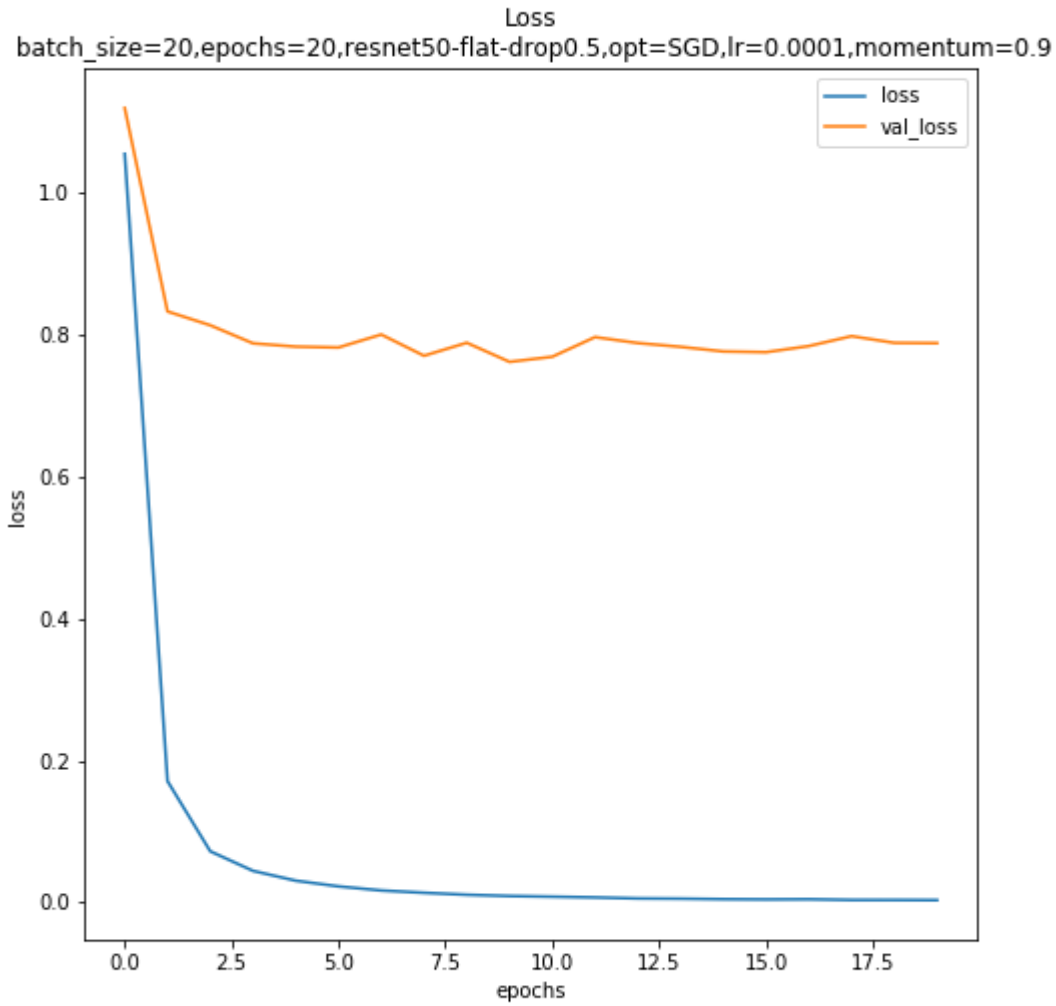
The initial implementation solution of a single layer of output nodes preceded by dropout(0.5) achieved the following test results.

- Test accuracy: 61.9768%
- Log loss: 1.113071

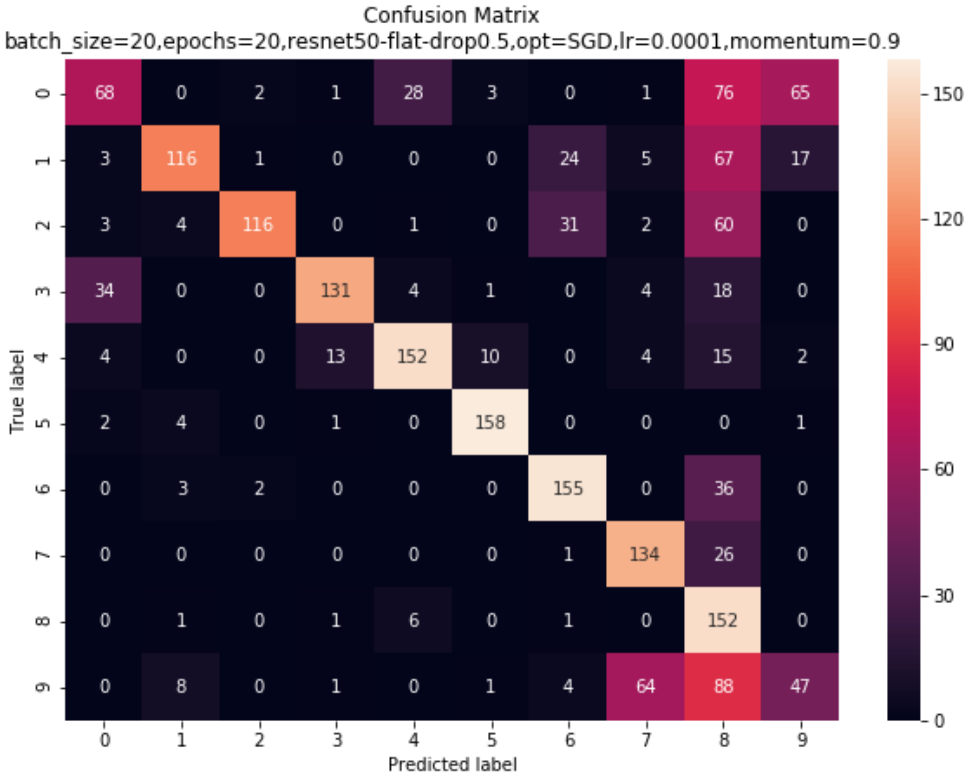
The accuracy and log loss plots for training and validation per epoch show the training quickly achieves very good accuracy and log loss after only one epoch and only has modest gains for subsequent epochs. This kind of learning curve indicates that it may be possible to reduce the learning rate.

There isn't evidence of overfitting where the train performance continues to improve, but the validate performance decreases. The gap between train and validate curves shows there is room to improve the generalization. The training of the network has achieved near perfect results on the training network, but significantly worse for validation data. The self-similarity of many training images to each other because of video likely has much to do with this rapid learning.


Accuracy
batch_size=20,epochs=20,resnet50-flat-drop0.5,opt=SGD,lr=0.0001,momentum=0.9
accuracy
epochs
training
validation
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5



A curious property of this simplistic initial solution is the frequent misclassification of images as hair/makeup.



The biggest complication was making sure the input data was correctly loaded and split into train, validate, and test sets. A simple mistake here could invalidate many hours of later experimentation and iteration. To avoid problems here I made extensive use of assertions to catch potential errors where the wrong variable was used in the path to generating the input tensors.

Another complication was iterating on the plots. Retraining the model just to generate some plots would take significant time, so I used a mechanism to save the weights of the model that would include the parameters in the filename. This allowed easily commenting out the training call and loading the prior weights from a file on disk. This also preserves the value of the very costly computations of training.

Refinement

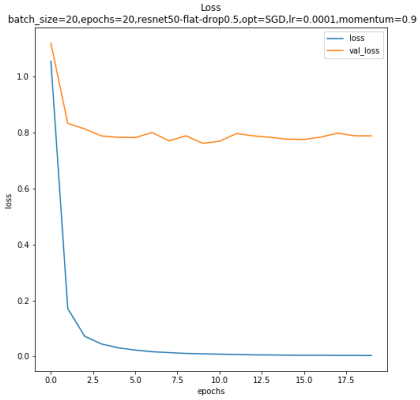
The biggest implementation detail key to refinement was keeping track of the experiments in an orderly fashion. This was accomplished by having functions return configured variables for the optimizer, keras model, and the training parameters. The returned variables also had a stringified form that was used when saving the output images and added to the titles. Instead of modifying the existing functions, new functions could be created that would encapsulate the experiment in a read-only way. The results of the experiment were recorded with the description of the experiment parameters in the chart titles and the filename saved to the analysis/ directory.

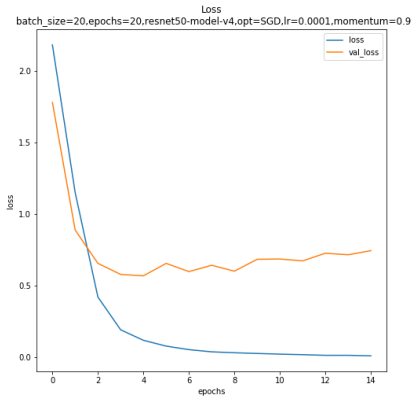
The majority of the changes were to the model layered after the pretrained deep neural network. The changes to this section of the network were supported by the test/validation accuracy and log_loss graphs for training epochs. Parameters were changed one at a time to determine if they represented a positive or negative refinement to validation accuracy and log_loss.

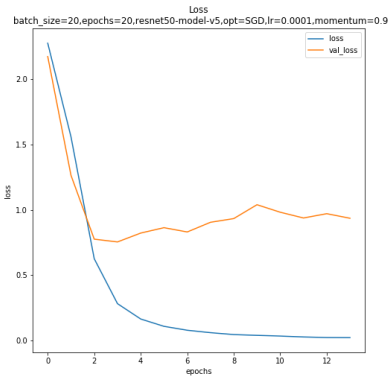
Additional modifications to the optimizer, optimizer parameters, batch size, and maximum number of epochs was attempted. In many of the runs the early stopping callback prevented additional rounds from executing if the model wasn't improving.

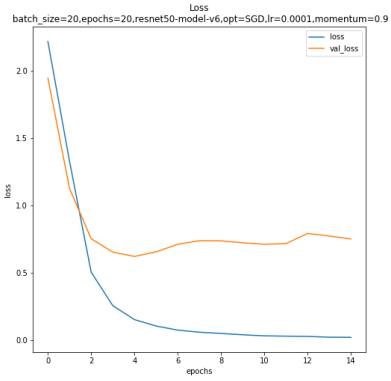
Substantial improvements from the initial single dense layer, single epoch training were achieved through the modification process and guided by the graphs.

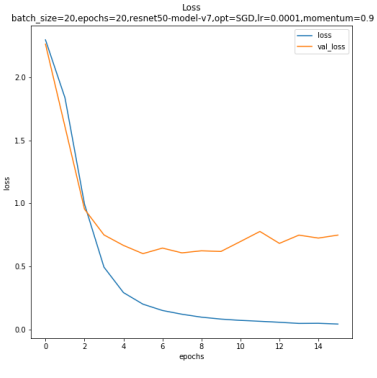
- Model v1
    - Drop(0.5) -> Dense(10)
    - Test accuracy: 61.9768%
    - Log loss: 1.113071
- Model v2
    - Dense(1024) -> Drop(0.5) -> Dense(1024) -> Dense(10)
- Model v3
    - Drop(0.5) -> Dense(512) -> Drop(0.5) -> Dense(512) -> Drop(0.5) -> Dense(10)
- Model v4
    - Drop(0.5) -> Dense(256) -> Drop(0.5) -> Dense(256) -> Drop(0.2) -> Dense(10)
    - Test accuracy: 66.3137%
    - Log loss: 1.010329
- Model v5
    - Drop(0.5) -> Dense(128) -> Drop(0.5) -> Dense(128) -> Drop(0.2) -> Dense(10)
    - Test accuracy: 69.1377%
    - Log loss: 0.974119
- Model v6
    - GAP -> Drop(0.5) -> Dense(128) -> Drop(0.5) -> Dense(128) -> Drop(0.2) -> Dense(10)
    - Test accuracy: 67.1710%
    - Log loss: 1.060210
- Model v7
    - Drop(0.5) -> Dense(64) -> Drop(0.5) -> Dense(64) -> Drop(0.2) -> Dense(10)
    - Test accuracy: 60.2118%
    - Log loss: 1.080033


Loss
batch_size=20,epochs=20,resnet50-flat-drop0.5,opt=SGD,lr=0.0001,momentum=0.9
loss
val_loss
loss
epochs


Loss
batch_size=20,epochs=20,resnet50-model-v4,opt=SGD,lr=0.0001,momentum=0.9
loss
val_loss
loss
epochs


Loss
batch_size=20,epochs=20,resnet50-model-v5,opt=SGD,lr=0.0001,momentum=0.9
loss
val_loss
loss
epochs


Loss
batch_size=20,epochs=20,resnet50-model-v6,opt=SGD,lr=0.0001,momentum=0.9
loss
val_loss
loss
epochs


Loss
batch_size=20,epochs=20,resnet50-model-v7,opt=SGD,lr=0.0001,momentum=0.9
loss
val_loss
loss
epochs

The final model was the following.

```
def model_v5():
    pretrain_model = applications.ResNet50(weights = "imagenet",
include_top=False, input_shape = (img_width, img_height, 3))
    train_layers = 0
    for layer in pretrain_model.layers[:-train_layers]:
        layer.trainable = False
    x = pretrain_model.output
    x = Flatten()(x)
    x = Dropout(0.5)(x)
    x = Dense(128, activation="relu")(x)
    x = Dropout(0.5)(x)
    x = Dense(128, activation="relu")(x)
    x = Dropout(0.2)(x)
    predictions = Dense(10, activation="softmax")(x)

    model = Model(input = pretrain_model.input, output = predictions)
    return model, "resnet50-model-v5"
```

The test loss improved from 1.11 to 0.97 from the initial model. Additional attempts to further reduce the number of nodes and make use of a global average pooling layer for ResNet resulted in worse validation performance.

Results

Model Evaluation and Validation

The benchmark model achieved accuracy of 12.30% and a log loss of 30.28 with the test set. This was very close to the last-placed entrant into the StateFarm contest leaderboard, so perhaps.

The final model achieved 69.1377% test accuracy and a test log loss of 0.974119, which places it in the top 50% of leaderboard entries. The test log loss of 0.974119 was slightly worse than validation loss of 0.6712 showing the model is somewhat sensitive to changes in the dataset, but generalizes to unseen data.

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

The final model was done by having the following parameters for the training, optimizer, and model.

- batch_size=20
- epochs=20
- Resnet50-model-v5
    - Resnet(50)
    - Flatten()
    - Dropout(0.5)
    - Dense(128) + ReLU
    - Dropout(0.5)
    - Dense(128) + ReLU
    - Dropout(0.2)
    - Dense(10) + softmax
- optimizer=SGD
    - lr=0.0001
    - momentum=0.9

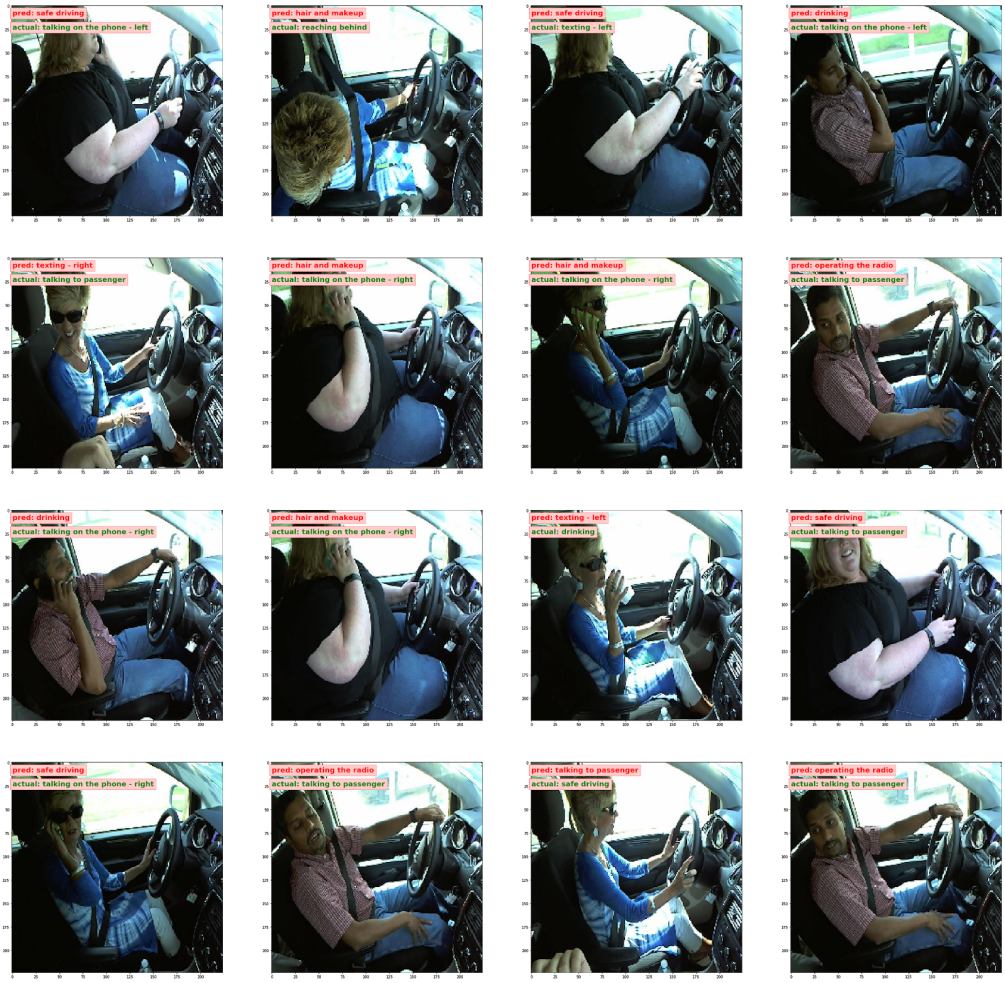
Justification

The results of the transfer learning solution at above 66% accuracy and 1.01 log loss on the test set compared to the 12.3% accuracy and 30.28 log loss of the benchmark solution clearly show an ability to classify images of distracted drivers. There is clearly room for improvement with the top Kaggle Competition entries posting a log loss of below 0.1.

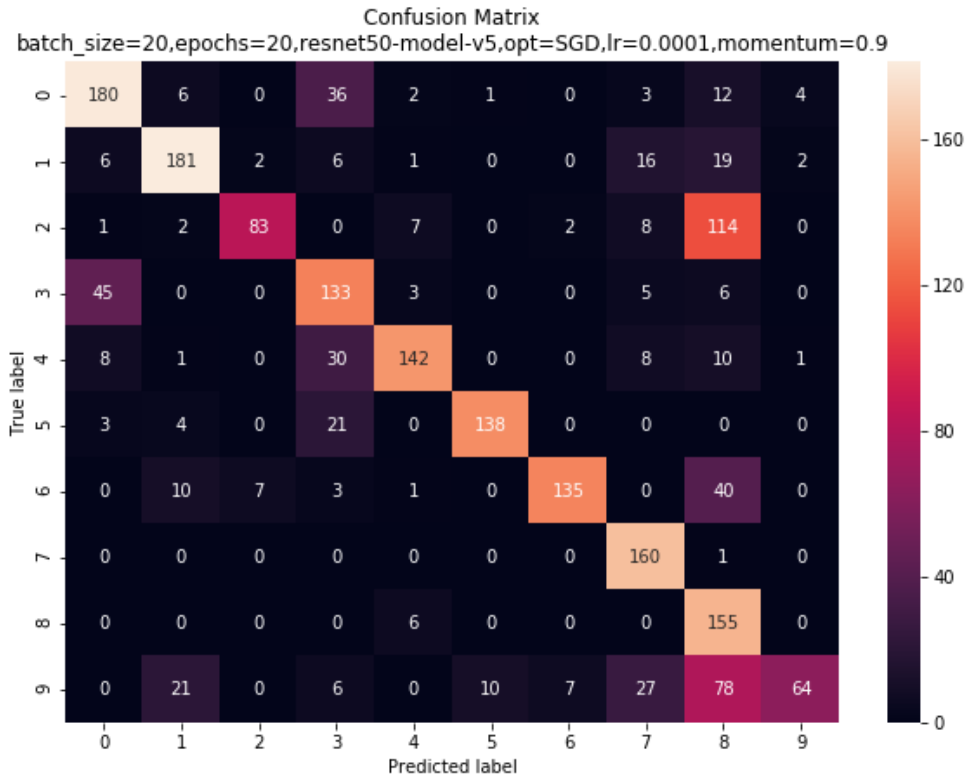
The level of accuracy and log loss required depends on the application. The level I achieved is not enough for a system integrated into a vehicle and used to alert a reminder chime. It is sufficient to show the potential of the technique and justify additional resources into improving the solution.

Conclusion

A few misclassified examples are presented in the following image with the true class and the mispredicted class. These examples were useful in understanding why the classification system was performing incorrectly by highlighting potential issues with the data or nuances with the data.



One of the very interesting things that is obvious when looking at misclassified images is how very similar images that are adjacent frames from each other are misclassified in the same way. This shows one of the biggest weaknesses in the dataset, which is significant redundancy because many of the images are near duplicates of each other. Stating that each class has 2000 images is a bit misleading because many of those images are very similar to each other. The approximately 60 images per class per person are effectively lower.



The confusion matrix shows the classifier was great at classifying reaching behind (c7) and hair/makeup (c8) when presented with them.

Talking on the phone right (c2) was frequently misclassified as hair/makeup (c8), perhaps because of the similar position of the right hand near the face for both activities. Additional training may allow the neural network to detect the presence or absence of a phone to distinguish the two.

Texting left (c3) and safe driving (c0) are confused in both directions, likely because of the difficulty of seeing the phone on the left size from the camera's angle.

In summary, the goal for this project was to write a classifier that could determine if a driver was attentive or performing one of nine distracted driving activities. A deep learning convolutional neural network was created using a pretrained network a start and augmented with additional layers. The weights for this network were trained using the labeled images in the StateFarm dataset and by using an optimizer that back-propagated errors from misclassifications. The quality of the model was evaluated using log-loss on test data that was withheld from the training process. Refinements of the model were made based on the results and then the training process repeated. The log loss and accuracy demonstrated this is a feasible problem to solve with neural networks and could likely be improved with additional investment.

Reflection

One of the hardest aspects of this project was the extremely slow iteration time, even with using an Amazon EC2 GPU accelerated instance. A more effective way to work might be to generate numerous experiments that would all be executed in a batch job. Instead of paying by the hour for an active EC2 instance, it might be nice to have an extremely powerful system that jobs could be submitted, quickly returned, and charged by the time spent in the job. Paying for machine time while coding and reviewing results incentives selection of a less powerful system to avoid excessive charges while not actively training.

Improvement

A very obvious way to improve the results would be to have an even wider set of training data captured and labeled. If a driver and capture assistant were paid 15 per hour each, it might be possible to capture more subjects at a cost starting at $30 per subject. It is possible this dataset could be doubled in size for a thousand dollars.

K-fold cross validation could be used to get the most out of the training data by not losing some of it to a validation set. This could be implemented by using the driver identifier as the split for the folds in the same way the train, validate, and test sets were created.

I initially assumed that most of the time spent in training was on the optimization/back-propagation and that marking layers as not trainable would avoid costs associated with that back-propagation. Given the speed of training even on a GPU, it seems that it might be worth investigating the use of extracted bottleneck features, especially if the weights of the pretrained network aren't going to be fine-tuned because of the limited amount of data available.

After getting deep into a lot of internet posts on transfer learning, I found several posts stating that ResNet wasn't designed for transfer learning. This appears to mainly be around the topic of truncating layers other than the final fully-connected layers, which I did not do. Investigating other models available in Keras would be a good use of time to determine are relevant to the driver classification problem and the way transfer learning is used in this model.

Ensemble methods are a proven way to create better predictions. Creating an ensemble from the output of several models using different pretrained networks would likely increase the accuracy and lower the log_loss.

Although rotation and flipping augmentation would not be useful, there are possible augmentations that may provide advantages. Blurring, lighten/darken, translation, and masking out everything but the segmented subject may be useful. The lighten/darken could compensate for an uncontrolled free variable of lighting conditions in the source data. Translation may be useful to handle different seating or camera positions. Masking a segmented subject has the potential to eliminate many unrelated signals from the windows, visor, wheel position, visor, mirrors, and vehicle contents. Implementation of non-affine transformations to model different camera intrinsic parameters would be useful if the model needed to handle different camera intrinsic parameters from multiple camera vendors or lenses.

It is also possible to unfreeze training of the pretrained layers after the trailing dense layers had been initially trained. Additional epochs of training could then allow back-propagation into the pretrained network to customize it for the distracted driving dataset.

A much more complex change would be altering the problem from identifying distracted driving from a single image into using a sequence of images of the video. The additional information could allow the model to make more accurate predictions using multiple images that represent the same activity.