Distracted Driver Detection

Elliot Orenstein
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
orens040@umn.edu

Sam Walczak
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455
walcz076@umn.edu

1 Project Description

When behind the wheel of a car, a driver's full attention should be devoted to the task of driving. However, as technology in cars and cell phones has accelerated, there are more distractions tempting a driver's attention away from the road. "Distracted driving" comes in many forms and can be caused by a myriad of sources, making it a difficult issue to solve. Part of the solution to this issue is identifying when distracted driving occurs. To help in this aspect, State Farm introduced a Kaggle competition that sought to have competitors use computer vision (i.e., image recognition) to identify distracted drivers. This is done by classifying driver behaviors into one of ten categories (e.g., driving safely, texting, talking on the phone) based on photos taken from dashboard cameras (on the passenger's side of the car). Our project will seek to analyze this data and develop model(s) that classify driver behavior and, thus, identify when drivers are distracted.

This issue affects everyone who drives - which is approximately 84.6% of Americans[1] - or, really, anyone who rides in motor vehicles. Every day, over one thousand people are injured in crashes involving distracted driving, according to the CDC[2]. Further, the auto insurance industry has seen increasing loss trends in recent years, which is likely influenced by an increase in distracted driving. In turn, these increased losses for insurance companies eventually lead to more expensive auto insurance costs for drivers. Identification of distracted driving is at the core of the solution to this wide-ranging problem, as it will allow distracted drivers to be held accountable and will provide additional incentive for drivers to focus on the road. Meaningful discovery on this problem would provide benefits for the insurance industry and improve driver safety.

Our goal for this project is two-fold: (1) generate a neural network model that classifies images and produces valid predictions, (2) generate a model that scores above the 75th percentile of the Kaggle competition (i.e., produces a log loss <= 0.57). Our focus will be on model accuracy rather than robustness. The following sections detail previous research on the topic, our work and findings, conclusions, and considerations for future research.

2 Previous Research

While there has likely been plenty of research on the broader topic of distracted driving, for purposes of this paper, we will focus on image classification of drivers using deep learning.

Image classification is a highly studied area of research in computer vision. Major advancements in the field have been made in the past decade using deep convolutional neural networks (CNNs) (e.g., AlexNet[3]), which has been made possible in large part by the advancement of high-powered computing (GPUs). Many papers on the topic discuss the use of pre-trained models, such as AlexNet, and their variations when approaching complex image recognition tasks, such as distracted driving. A paper by Abouelnaga et al.[4] trained AlexNet and InceptionV3 models on raw, face, hands, and "face+hands" images, and then used weighted ensemble methods via genetic algorithm to generate the final class probabilities. Another paper by Majdi et al.[5] proposed Drive-Net, a CNN network followed

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

Figure 1: Image Classes and Example Picture

by a random forest, to classify driving behavior, which achieved 95% accuracy in its five-fold cross validation experiments.

The Kaggle competition had over 1,400 entries, so there have presumably been a wide variety of models used in search of the optimal classification algorithm. Since individual algorithms are not visible to the public, we do not know whether certain types of models have worked better than others on this data. However, there is a public leaderboard showing scores based on multi-class log loss, which provides a good baseline for comparing the performance of our model. The Kaggle competition had a median score of 1.65, a 75th percentile score of 0.57, and a top score of 0.087. Beyond comparing model performance, there were also some useful posts on Kaggle that helped steer our model generation for the project (we will expand upon some specific examples later).

3 Our Work and Findings

The training data for this problem consisted of a set of 22K images, each from the dashboard camera of a car, with each image belonging to one of ten classes describing what the driver is doing (see Figure 1 for example image and class descriptions). The testing data consisted of 79K images of the same format, which were used to evaluate model performance on out-of-sample predictions.

We implemented our data and modeling processes in Google Colab, and primarily relied on PyTorch and NumPy functions to build our neural networks. Due to the size of the data and the computing power necessary to run neural networks, the data preparation/implementation process was not as simple as writing one line of code. Further, certain aspects of the process were new to us (e.g., working with large data sets stored on Google Drive, working on Colab Pro, refining the train and test loaders, transforming the images, running on GPUs). In short, it took us longer than anticipated before we were able to actually start building models!

As far as the actual implementation of neural networks, we began by using the models provided in the homework assignments for this class as a starting point / framework for our project. Specifically, we began with a basic, fully-connected, two-layer neural network with ReLU activation functions as our baseline model. This model also utilized the AdaGrad algorithm to update parameter values. We spent a great deal of time troubleshooting to get this model working properly, and making sure that the output produced was valid. As previously discussed, there was a Kaggle leaderboard for this problem where we could compare our model's performance (on the test data) to other submissions. One complication in the model evaluation process was that the Kaggle-provided testing data for this problem did not contain the class labels for the images (i.e., we only had the images, no class labels). So, while we could generate predictions for the test images, we were required to upload those predictions to Kaggle in order to evaluate our model's performance on the test data. Recall that Kaggle scored predictions based on a multi-class log loss function (see Figure 2). As a way around this complication, we built a function to split out a validation set from the training set, which was then implemented along with our trainloader, prior to running our models. This allowed us to set aside a group of images to use as a validation set within Colab, such that we could get a general idea of model performance without having to upload to Kaggle for every single model run.

As a barometer for log loss scores, note that using "random" predictions (i.e., predicting 0.1 for all ten classes on all images) produced a log loss of 2.30258 on the testing data. After implementing

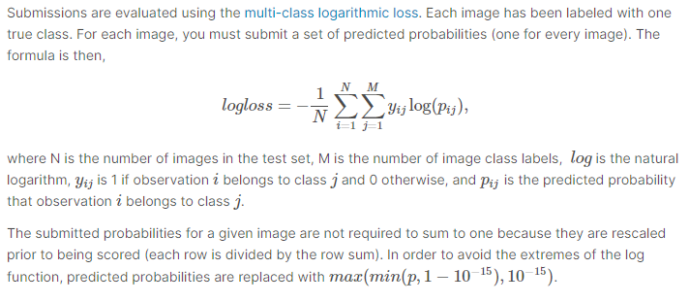

Figure 2: Log Loss Function Used in Model Evaluation

variations of our initial two-layer model (described above), we were able to make marginal progress on that score, producing values in the 2.26 - 2.29 range (measured on the testing data, by Kaggle). However, when we attempted to improve on that score, further training and parameter tuning generally did not result in improved scores. In fact, some models actually produced a worse score than the random predictions (i.e., > 2.30258). This was a confusing result at first, as our training appeared to be "working" in that it was producing monotonically decreasing training loss (i.e., the loss function was decreasing as the number of epochs/batches increased) and performance on the validation set was very strong. As we investigated this trend, we found multiple Kaggle forums stating that using all images of a specific driver as the validation set (rather than a random collection of images) provides a more reliable testing scenario. This way, the validation would be performed on a "new" driver that the model had not already seen in training. After implementing a validation set that used all images of one specific driver (rather than a random collection of images), we found that our scores on the validation set were, in fact, performing very poorly. So, we were then seeing poor performance on both: (1) the validation set, and (2) the testing set via Kaggle, despite a decreasing loss function on the training set. We believed this could suggest that the model was overfitting the training data, however, given that these results were produced from a relatively simple model after just a few (2-10) epochs, it is also possible that it was simply a poorly performing model, rather than an overfitted model. After observing these results, we focused on using fewer epochs, fewer nodes within hidden layers, and frequently checked the error on the validation set (in hopes of avoiding overfitting). These changes did help to improve on our poor performance on the validation set. However, even after making these changes, we did not immediately achieve dramatic improvement in performance on the testing data. We did eventually find some success with this model after (1) increasing the batch size above 64, and/or (2) decreasing the step size to be smaller than 0.001. Adjusting these parameter values allowed us to improve our scores on the test data from the 2.26 - 2.29 range down to 2.06 (with similar performance seen on the validation set). After seeing this initial benefit, we did more exhaustive parameter tuning to see if we could improve on the 2.06 score.

As previously mentioned, increasing the batch size above 64 helped performance. We saw roughly equivalent performance across batch sizes in the 100-200 range. Pushing that value any higher resulted in our GPU overloading, so we did not test anything larger. Adjusting the number of hidden nodes (in our two layers) did not appear to have a dramatic impact on model performance, so long as they were larger than 64 / 32 in the first and second layers, respectively (e.g., we saw similar performance using values of 128 / 64 compared to 64 / 32). Also, similar to the batch size, increasing the number of nodes by too much caused our GPU to overload. Step size was perhaps the most sensitive parameter within our initial model. Moving off of our initial value of 0.001 gave us a boost in model performance, and we eventually found that values of 0.000001 (1e-6) or 0.00001 (1e-5) produced optimal results in this framework. At this stage, we tried pivoting to stochastic gradient descent (SGD), RMSProp, or the Adam algorithms (as opposed to AdaGrad) for our updating method, but none resulted in improved scores based on our trials. The number of epochs we used varied depending on the set of parameters we were testing. We installed a check in our code that stopped training the model if the performance on the validation set was plummeting (i.e., stop training rather than going through the remaining epochs). With the smaller learning rates, we were occasionally able to train a model for 30+ epochs, however, in general, the model performance would start to deteriorate sometime within the first 10 epochs. After all this further refinement, we were able to produce a score of 1.73 on the test data (down from 2.06). While an improvement, this score seemed to be a lower bound on the performance for this model framework. Again using the trajectory of this

class as our guide, this was a logical time for us to pivot away from our fully-connected network to a convolutional neural network (CNN). In hindsight, due to the high dimensionality and nature of our data, we may have spent too much time training/optimizing our fully-connected network, as the CNN is generally a better fit for these high-dimensional images. Further, a CNN is generally less susceptible to the overfitting / poor performance issues we initially ran into. However, we did learn a lot about parameter tuning in this process, which proved useful in training the future networks we applied to this problem.

Moving to the CNN framework did not produce the immediate improvement that we were hoping for. We started with a CNN composed of two convolutional layers. First, we reduced the size of the images by a factor of two (to reduce training time) and normalized the images. The model applied two layers of 3×3 filters over the transformed images, with optional zero padding and stride modifications. After each convolutional layer, we implemented dropout, ReLU activations, batch normalization, and finally 2×2 max-pooling. The outputs from the feature extraction were then fed into a two-layer fully-connected neural network for the purpose of classification. The motivation for the architecture of this CNN came from the Drive-Net model proposed by Majdi, and was further reinforced based on our class lectures. Under this framework, we found that the Adam update method performed superiorly to AdaGrad, RMSProp, and SGD. We also found that a smaller step size of 0.000001 (1e-6) again worked best, as larger values would cause validation loss to increase after just 1-2 epochs. Generally, our CNN model did not benefit from training beyond 5-10 epochs. Even when using a smaller step size that allowed for a more stable loss on the validation set, models that trained for 20+ epochs did not see improvement in performance compared to models that trained on 5-10 epochs (this was a common theme throughout the project). The number of input and output channels of the network did not have a dramatic impact on model performance. We eventually achieved a top score of 1.55 on the testing data (via Kaggle) using the CNN model, with broader scores from the CNN ranging from 1.55 - 1.94, depending on the parameters used. So, this model did provide some improvement on the 1.73 score from the fully-connected model, but we were expecting to see something better than 1.55, especially considering the amount of parameters/methods that we explored. From here, based on our class lectures, we had thankfully learned that pre-trained models were utilized frequently for image recognition problems, and often performed very well. With this in mind, we decided to explore pre-trained models.

After switching to a pre-trained model, we immediately saw a substantial improvement in model performance, with scores on the test data dropping below 1.00 (recall that the Kaggle leaderboard had a median score of 1.65). Given that the dataset was relatively large, we employed two transfer learning strategies on the VGG pre-trained model. Our first was to assume our dataset was large and different from the ImageNet dataset (which is debatable), which resulted in us finetuning all of the pretrained weights through the entire network. Under this framework, we found that the model trained very quickly, with validation loss minimizing after just 1-2 epochs. This likely occurred due to the large number of trainable parameters, making the model very sensitive to overfitting. We eventually settled on 2 epochs (with a batch size of 50 images) as the optimal amount of training for this model, and we again found that the Adam update method performed better than others. Even while using just 2 epochs, we struggled to slow the training of this model, as the log loss on the training data was converging to zero very quickly, and performance on the validation set would begin to deteriorate within those 2 epochs. While Adam helps control for the magnitude of the gradient in each dimension, we used a learning rate scheduler to further slow training. We implemented a multiplicative learning rate scheduler (from PyTorch) that would periodically decrease the learning rate within our training cycle (e.g., the scheduler would multiply the learning rate by a factor, say 0.95, to slow down the training). We explored a number of different options for the implementation of this multiplicative learning rate, ranging from the extremes of: (1) applying the multiplier every single batch, to (2) applying the multiplier only after a full epoch had finished. In the interest of time, it was not possible to perform a fully exhaustive search of the options here (keep in mind that the "normal" step size parameter and the update method used are also related to this process), but we eventually found that applying the multiplier of 0.95 every $\sim$50 batches worked well for this problem (when using a step size around 1e-4 and the Adam update method). Ultimately, our best and "final" model used this VGG framework, using 2 epochs (batch size of 50), step size of 1e-4, Adam update method, and a multiplicative learning rate scheduler of 0.95 (applied every 50 batches). This model produced a log loss score of 0.347 (via Kaggle), with similar performance seen on the validation set.

While we were able to achieve much lower loss using the pretrained VGG with the aforementioned transfer learning strategy, we also tried employing a second strategy to compare performance. Here, we assumed that our dataset was large and similar to the ImageNet dataset. In this strategy, we froze early feature extractor layers and finetuned from there through the entire network, under the assumption that the similarity of the datasets would yield similar features in early convolutional layers. This strategy allowed us to benefit from finetuning the later feature extractor layers, which could potentially improve performance while also mitigating overfitting concerns (by reducing the number of trainable parameters) that we experienced when finetuning through the entire network. In turn, we hoped to find an optimal balance that would generalize better to our test set. This strategy introduced a hyperparameter of the cutoff convolutional layer, or, the layer at which to freeze all layers below it. Through dozens of runs, we found that freezing the first 3-4 layers achieved the best results. While freezing these early layers slowed down training marginally, the results produced were generally similar to the previous VGG model. Ultimately, these two transfer learning methods performed similarly across a variety of different hyperparameters we tested, but the score of 0.347 (previously mentioned) was the best that we were able to achieve.

4 Conclusions

All of our work on this problem has produced a number of interesting takeaways. Perhaps the most noteworthy is the improvement in performance we saw from using a pre-trained model (as opposed to one that we built from scratch). Considering the time and effort we spent implementing, understanding, and tuning our initial model, when weighed against the marginal improvement in results, we would have been much better off starting with the pre-trained model (at least from a model performance standpoint). However, there was undoubtedly some benefit for us from a learning standpoint in spending the time we did on our initial model. We certainly have a much better understanding of neural networks and their parameters after our work on this project, and after our work on our initial model, in particular.

Another conclusion is the importance of parameter tuning and choosing modeling methods. As described in the previous section, results varied quite a bit depending on the type of model and value of parameters that went into the model. Based on this, we would be hesitant to quickly dismiss a particular method in future work, as it's possible that performance could drastically change once the parameters are optimized. Also, we could have benefited from additional time tuning the parameters of our models. While we did spend a great deal of time tuning the parameters as it is, there is certainly room for improvement in that area, and additional time spent exploring other options may result in marginally better results than we were able to achieve.

One aspect of our project we have not yet discussed is our implementation of a function to run our models. In short, we wrote one large, all-encompassing function that we could utilize to train our model using different parameters (e.g., step size was a hyperparameter to the function, and we could call the function with a new step size to test model performance, rather than changing the code for our full model every single time). An example is provided below in Figure 3. Later on in the project, as we began exploring other types of models, we expanded this function to include options that could specify "CNN" or "VGG" which dictated the type of model to be utilized. This provided a very efficient framework for training our model and tuning parameters, and thus is something we wanted to highlight in our conclusions to our project.

Ultimately, we achieved our initial goal - we generated a valid model that produced a log loss of 0.347, scoring in the 84th percentile of the Kaggle competition (our original goal was to beat the 75th percentile). We are proud of this achievement, as it certainly looked more grim at other points in the semester! The project as a whole proved a more challenging task than we initially believed. This project, in concert with the other work for this course, provided an excellent framework for us to gain a deeper understanding of neural networks, and provided a practical application of the methods taught in the course. Our work has also given us perspective on real-world implementations of neural networks. The difficulties that we encountered were a good reminder that applying these neural networks in practice requires a detailed understanding of: the problem, the underlying data, the theoretical foundation of the model, and patience!

```
▾ 03 - Apply Function

[ ]  mod13= DDD_Model_v3(batch_size_used = 50, stepsize = 1e-4, method = 'adam', epochs =2, testOutput = True, use = 'gpu', mod_type='vgg', buffer = 0.99, lr_fact = 0.95)

     You're on GPU!
     Data Loaded!
     Parameters Set!
        Validation Loss before Epoch 0: 2.393
     Done With epoch 0 | Loss = 5115.0
        Validation Loss before Epoch 1: 0.49
     Done With epoch 1 | Loss = 327.0
        Validation Loss after Epoch 1: 0.597
     Beginning to apply test data
     0.06
     0.13
     0.19
     0.25
     0.32
     0.38
     0.44
     0.51
     0.57
     0.63
     0.7
     0.76
     0.82
     0.89
     0.95
     Model Run Complete!
```

Figure 3: Example of Comprehensive Modeling Function Used for our Project

5 Considerations for Future Research

In the framework we used for our project, we did experience some instability in our model results. For example, we would train a model twice, using the exact same hyperparameters, and the two resulting models would be slightly different (i.e., they would produce different scores on the test data). It would be interesting to learn more about why this occurs, and to explore if there are any worthwhile techniques available to prevent this type of instability, for sake of reproducibility, if nothing else. It would seem that our choice to use small, shuffled batches to train the model was a large contributor to this issue. Thus, increasing the batch size and/or not shuffling the data (on each model run) might be something to consider in future research, at least once a model is close to finalized.

As previously discussed, there is certainly additional parameter tuning that could be explored for our final model. Additionally, there are likely other worthwhile modeling methods we could explore further. In particular, there are other pre-trained models that would be worth implementing to compare against VGG. It is possible that pretrained ResNet models could perform even better than our existing model, as its depth may be well-suited to handle the complexity and nuance of our task. Previous work has also successfully used AlexNet and InceptionV3, so those would also be worth exploring. Additionally, appending a random forest (or other machine learning technique) to the end of the neural network would be another idea to apply.

We could also use transfer learning to identify important components of the image, such as the face and hands, in future work. Abouelnaga explores this in their work and then uses an ensemble of classifiers to predict classes. Potentially expanding this concept to other objects, such as phones, could be pertinent information to help achieve better results.

References

[1] Number of Licensed Drivers in US, & How Many Americans Drive. 25 Aug. 2020. Available at hedgescompany.com/blog/2018/10/number-of-licensed-drivers-usa/.

[2] National Highway Traffic Safety Administration. Traffic Safety Facts Research Notes 2016: Distracted Driving.S. Department of Transportation, Washington, DC: NHTSA; 2015. Available at https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812517

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. Commun. ACM 60, 6 (June 2017), 84–90. DOI:https://doi.org/10.1145/3065386

[4] Yehya Abouelnaga, Hesham M. Eraqi, Mohamed N. Moustafa. Real-time Distracted Driver Posture Classification. arXiv:1706.09498, 2018.

[5] M. S. Majdi, S. Ram, J. T. Gill and J. J. Rodríguez. Drive-Net: Convolutional Network for Driver Distraction Detection". 2018 IEEE Southwest Symposium on Image Analysis and Interpretation (SSIAI), Las Vegas, NV, 2018, pp. 1-4, DOI: 10.1109/SSIAI.2018.8470309.