Rewriting the Berkeley Deep Drive Driving Model

Jordan Patterson, Austin Hendy, Daria Sova, Maxwell Borden
University of Victoria
{jpatts, ahendy, dariasv, mborden}@uvic.ca

Abstract

We rewrote the Berkeley Deep Drive (BDD) Driving Model with modern best practices, to ensure that future developers don't have to sift through the original codebase. Documentation was a focus, with required code and function comments describing functionality. Proper naming schemes and structure were also implemented, in order to present the information as fluently as possible. Finally, everything is written with Python 3.6, and as such use the most up to date libraries possible. We were able to implement all major components of the paper, including alexnet and the LSTM.

1. Introduction

Communication of ideas is very important, and a tenant of research anywhere. It is very difficult to build upon the work of others when it is either convoluted or not properly documented. Both of these issues plague the BDD Driving Model project, where it seems much of the coding was rushed and poorly thought through. To remedy this, we rewrote practically everything in the project, much of which from scratch. As such, we have come to have a strong understanding of the goal for the BDD Driving Model, and how it was implemented. We successfully preprocessed the data and fed it into AlexNet. Then we created the LSTM, and had training working on that to build a model predicting the next direction for a car given a driving scene. We improve upon the original by having better documentation, naming schemes and overall code structure.

2. Contributions of Original Paper

2.1. Background

Huazhe Xu, Yang Gao, Fisher Yu, and Trevor Darrel of University of California Berkeley published a novel End-to-End deep learning driving model in 2017[1]. Inspired by previous works, such as the successful End-to-End fully convolutional model published by NVIDIA corp. in 2016[2], they combined temporal sequences along with axillary sensor data to improve accuracy.

2.2. Dataset

One of the primary contributions introduced in the original paper is the Berkley Deep Drive dataset. It is primarily designed to stand on its own against other large video datasets, advertising itself as the largest publicly available driving video dataset, compared to others such as Robotcar and KITTI. Although publicly available, it was a contribution of the paper, and used to permit further research in the field of vehicle control and analysis by way of deep learning. The dataset contains high quality dashboard camera footage, with one frame every 10 seconds being segmented by hand into areas classified as one of 41 object categories. Along with the videos are auxiliary sensor data such as accelerometer, GPS, and current speed. The footage includes a wide variety of various driving environments and situations including urban and rural settings with diverse lighting, weather, and time.

2.3. Model

The BDD Driving Model project created a new variant of a recursive neural network with a convolutional neural network combined with a long term short term memory network (CNN+LSTM). The CNN+LSTM uses the joint loss of the predictions and loss of an intermediate convolutional layer generated with privileged information training on the segmentation frames. BDD was able to show that the addition of this secondary source of information was able to help the network converge faster than using just the loss from the final prediction of the network. The segmentation focused layers of the network are based upon a modified version of the AlexNet[3] network. The 'LSTM' aspect of the CNN+LSTM is a two layer stacked LSTM consuming the activations of the fc7 layer of the AlexNet section.

2.4. Previous State of the art works

As deep learning and computer vision are currently very active areas of research in the academic computer science community, there are a fair number of competing state of the art works involved with the topic of end-to-end neural

network based self driving vehicles. The BDD team has done a comprehensive comparison to the competing state of the art techniques, along with a few different configurations of their new model. The models compared against in the paper are an LSTM predicting solely based on the previous speed values, a CNN predicting the previous frame, an LRCNN and an FCN-LSTM training only on the previous frames. The new architecture is able to provide a similar accuracy to the other state of the art methods while converging faster and requiring less memory to utilize.

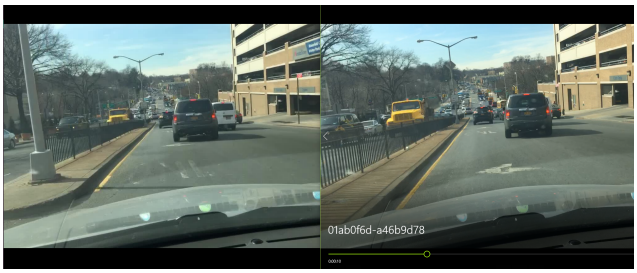

Figure 2. Frame-10s is on left, video at $\tilde{1}0$ seconds on right

3. Contributions of this Project

3.1. Difficulties

This project overhauled the entire BDD Driving Model codebase, and was successful in creating a smaller, better documented version of the original. This was not an easy project; despite the rating of 3/5 on the paper list given to us, it really belonged at a 4-5/5 once it was realized that the original codebase was very scrapped together and poorly documented. Preprocessing alone took 30-40 hours to figure out, with the research paper as the only thing to go off of (along with Dr. Kwang Moo Yi's help). This problem with documentation reoccurred at several key points in development as well, particularly with the LSTM implementation, where it was very unclear how the network parameters should be configured.

However, the largest issue turned out to be the data itself, which was largely confusing, poorly explained, not documented, and sometimes blatantly wrong. There are $\tilde{2}8{,}000$ videos in the BDD database, yet only $\tilde{4}000$ of them have segmentation data, thus rendering $\tilde{2}4{,}000$ videos without any form of label. This was annoying, but easy to solve, by only choosing the videos with segmentation data. This was just the beginning of problems with the data however, as Figures 1 and 2 highlight the frame that occurs at 10 seconds in every video we analyze.

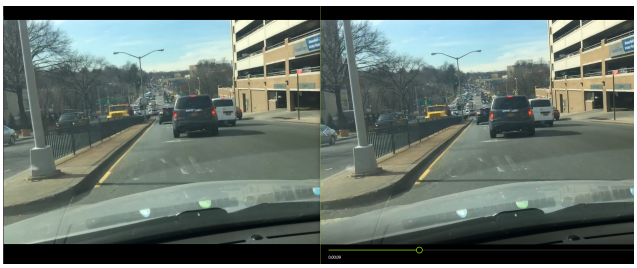

Figure 1. Frame-10s is on left, video at $\tilde{9}.2$ seconds on right

Looking at the figures, it is clear that the frame-10s image does not actually occur 10 seconds into the video. In fact, testing revealed it occurred at $\tilde{9}.18$ seconds in this specific video, which was significantly off the 10 second mark. This of course makes finding the exact frame in the video that the frame-10s occurs in quite difficult, but we decided that it could be done by comparing each frame with the frame-10s. What we learned was that the frame-10s doesn't even occur in any frame in the video. There were no matches across any of the videos frames with the frame-10s. This means that they likely used ffmpeg and requested the frame at 10 seconds, which would end up being an interpolated frame.

This should be documented, rather than requiring other users and developers hours of their time to understand. This ends up being a great example of why this project should not have been released in its current state. Furthermore, it illustrates just how useful our refactor is, as it explains these issues in the code.

3.2. Analyzing their Code

As an example of how the code is written, refer to figure 3, where many convoluted and undocumented operations are occurring. Checking lines 113, 129 and 147, the operation "inspect.stack()[0][3]" is being used. This essentially just gets the name of the function that the inspect.stack() operation occurs in. It is typically used for reverse engineering code, yet here it is being used instead of a simple hard coded string with the functions name contained. This is a great example of something that added a lot of time to completing this rewrite, as we had to check what pieces of code such as this were doing, often to learn how unnecessary they were to begin with.

```
def continuous_linear_bin(phase):
    tag = inspect.stack()[0][3]
    set_gpu_ids(phase, "0", "5")
    common_final_settings_continous(phase,
                                    tag,
                                    7260)
    set_train_stage(False, 110001)

    FLAGS.discretize_max_angle = math.pi / 2 * 0.99
    FLAGS.discretize_max_speed = 30 * 0.99
    FLAGS.discretize_label_gaussian_sigma = 0.5

    FLAGS.discretize_bin_type = "linear"
    FLAGS.discretize_n_bins = 180


def continuous_log_bin(phase):
    tag = inspect.stack()[0][3]
    set_gpu_ids(phase, "1", "6")
    common_final_settings_continous(phase,
                                    tag,
                                    7297)
    set_train_stage(False, 149001)

    FLAGS.discretize_max_angle = math.pi / 2 * 0.99
    FLAGS.discretize_min_angle = 0.1 / 180 * math.pi
    FLAGS.discretize_max_speed = 30 * 0.99
    FLAGS.discretize_min_speed = 0.1
    FLAGS.discretize_label_gaussian_sigma = 0.5

    FLAGS.discretize_bin_type = "log"
    FLAGS.discretize_n_bins = 179


def continuous_datadriven_bin(phase):
    tag = inspect.stack()[0][3]
    set_gpu_ids(phase, "2", "5")
    common_final_settings_continous(phase,
                                    tag,
                                    7298)
    set_train_stage(False, 91001)

    FLAGS.discretize_max_speed = 30 * 0.99
    FLAGS.discretize_label_gaussian_sigma = 0.5

    FLAGS.discretize_bin_type = "datadriven"
    FLAGS.discretize_n_bins = 181
    FLAGS.discretize_datadriven_stat_path = "data/" + tag + "/empirical_dist_dataDriven.npy"
```

Figure 3. Code snippet from ”config.py” in the BDD Driving Model Github project

Looking throughout the rest of this code, we can see that unexplained numbers are being used, alongside poorly explained functions, all without a single line of documentation for any of it. This was not an isolated case, and most of their codebase ended up being similar, pushing the total contributed time between all members to well over 150 hours for the coding alone, far more than was expected. The reason for this is that we got to a point where looking at the code became a last resort, only happening if we couldn't figure it out from the paper or guidance from Dr. Kwang.

3.3. Implementation

Data preprocessing was rewritten from scratch, with only the JSON info data code having any basis on the original codebase. Everything else was written without a point of reference, only drawing off of what the paper suggests is happening (with help from Dr. Kwang again). As such, our preprocessing is much more fluent and understandable than the one present on their Github.

First, before any data preprocessing, we analyze it to ensure it is valid across all requirements, with many helpful print statements and error checks for the user. Next, once it has been determined that all data in a given directory is valid and matched to the other required data, the videos are opened and parsed frame by frame at a rate of 3hz. Every chosen frame is downsampled to size (640, 360, 3), as mentioned in the paper. These are written to an HD5 file, alongside the segmentation, frame-10s, and info data, with a unique dataset for each, and a unique group for each video. The info data is processed to ensure it is valid, removing videos with 3 or more invalid data pieces present. It is then interpolated between each frame, such that there is proper acceleration between each location update.

Once all the data is written to the HD5 file, it is opened in the network for processing. Relevant values are parsed into variables, which are split into 70% training, 20% validation, and 10% test. Data is then fed into our network, which uses an Alexnet implementation with pre-trained weights combined with the LSTM. Results can be seen in figures 4 and 5, where we do achieve overfitting but have problems doing actual segmentation. LSTM accuracy is very inconsistent too, which suggests an error is occurring. Looking at the segmentation image, the model appears to get the ground and sky mostly correct, but has difficulties with other items. This was a downsampled image for sake of computation time, which explains the low pixel density. With another day to work on this, we believe somewhat accurate training results could have been achieved.

Figure 4. Overfitting correctly on Segmentation, LSTM having issues

Figure 5. Example segmentation classification

3.4. Conclusion

Despite all these setbacks, we are very proud of what we created, as it is far more readable than what we had to work with. Everything major is implemented from the paper, and functioning to some degree. The documentation is a major step up from the original source project, as is the code writing, and we agree that this would be a useful project for someone to work off of.

4. Discussion and Future Directions

4.1. Teamwork

Communication was never really an issue, with the slack channel containing many questions, answers and meet-up suggestions. Members were able to make it to a majority of the meetings. The only thing that was a struggle between us was coordinating available time during finals, as practically all of us had very little breaks between, and were busy with other requirements. Looking back, starting earlier and keeping a consistent schedule likely would have been very helpful.

4.2. Weaknesses

Rewriting an entire codebase of the original project highlights the do's and don'ts of programming, especially when collaboration and longevity are concerned. Naming schemes for variables and functions, comments highlighting the not obvious, and concise code that is still readable all play an important part in creating a project that others might use. While the number 2000 or 1100 might seem relevant and intuitive to the writer, the reader often will have no idea what this means if a comment or name is not applied explaining so. Out of everything that this project has failed to do, this was perhaps its greatest error (see figure 6).

```
def read_json(json_path, video_filename):
    with open(json_path) as data_file:
        seg = json.load(data_file)

    locs = seg['locations']
    loc2nparray = lambda locs, key: np.array([x[key] for x in locs]).ravel()
    res = {}
    bad_video_c = 0
    bad_video_t = 0
    bad_video_same = 0
    for ifile, f in enumerate(locs):
        if int(f['course']) == -1 or int(f['speed']) == -1:
            bad_video_c += 1   # Changed for interpolation
            if bad_video_c >= 3:
                break
        if ifile != 0:
            if int(f['timestamp'])-prev_t > 1100:
                bad_video_t = 1
                break
            if abs(int(f['timestamp']) - int(prev_t)) < 1:
                bad_video_same = 1
                break
        prev_t = f['timestamp']
    if bad_video_c >= 3:
        print('This is a bad video because course or speed is -1', json_path, video_filename)
        return None
    if bad_video_t:
        print('This is a bad video because time sample not uniform', json_path, video_filename)
        return None
    if len(locs)==0:
        print('This is a bad video because no location data available', json_path, video_filename)
        return None
    if bad_video_same:
        print('This is a bad video because same timestamps', json_path, video_filename)
        return None

    for key in locs[0].keys():
        res[key]=loc2nparray(locs, key)

```

Figure 6. Code snippet from "json_to_speed.py" in the BDD Driving Model Github project

One can look past the chicken scratch coding, and lack of overall documentation, and still understand what is happening by analyzing the code. But to understand a number that means nothing to the reader, and has no documentation, is extremely difficult if not impossible to do in any reasonable amount of time. In practically every function and file this can be found, making it a guessing game as to what the original purpose was to begin with. With more time, we would have liked to understand what these numbers mean, as often if we couldn't figure out the purpose, we were forced to replicate their mistake and leave the number there, without understanding.

4.3. Continuing

The BDD Driving Model project had very nice graphical effects showing the angle the car is driving in real time during a video. For comparison, all of our results were terminal based, and projecting them onto a video is something we think would have been nice to add. One of the future goals of the original BDD Driving Model project is to implement real time object detection overlay on the videos. We initially wanted to do this using the YOLO 9000 implementation discussed in class, but unfortunately ran out of time to do so. We will consider picking this up in the future to implement. We may also continue polishing and developing this after the deadline, because this is a very interesting topic of research that excited us as we worked on it.

References

[1] H. Xu, Y. Gao, F. Yu, and T. Darrell, "End-to-end learning of driving models from large-scale video datasets," *CoRR*, vol. abs/1612.01079, 2016.

[2] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, (USA), pp. 1097–1105, Curran Associates Inc., 2012.