Project Report: Autonomous Driving Simulation on Edge Device

Authored By.
By Brandyn Ewanek
Matriculation: 9216750
Customer ID: 10664359

1. Introduction

Autonomous driving technology relies heavily on "behavioral cloning," where a deep learning model learns to mimic human steering angles based on visual input. The objective of this project was to develop an end-to-end learning system for a self-driving car, train a Convolutional Neural Network (CNN) using the Keras framework, and deploy this model on an Android mobile device (Edge AI).

The system processes video feeds of driving scenarios and predicts the necessary steering angle in real-time. Furthermore, an intervention mechanism was implemented to allow a human observer to override the system if it fails, facilitating the calculation of an "Autonomy Grade" (Bojarski et al., 2016).

2. Methodology

2.1 Development Environment and Data Source

The machine learning pipeline, including data preprocessing, augmentation, and model training, was implemented in a Jupyter Notebook environment named Automated_Driving.ipynb hosted on Google Colab to leverage cloud GPU acceleration, *A100 GPU*.

The model was trained using the "Jungle" dataset (self_driving_car_dataset_jungle), which contains driving imagery captured in a challenging, foliage-heavy simulation environment. This dataset was selected to test the model's ability to identify road boundaries amidst complex visual textures. The dataset structure included center, left, and right camera views paired with synchronized steering angle logs.

2.2 Data Preprocessing and Augmentation

To ensure the model generalized well to new driving scenarios and to prevent overfitting to the specific track geometry, the following processing steps were applied within the notebook:

- **Resizing:** Input images were resized to 160X320 pixels to reduce computational load while maintaining sufficient spatial resolution for lane detection.
- **Data Augmentation:** The dataset was artificially expanded to correct for potential steering bias (e.g., tracks with mostly left turns). This was achieved by horizontally flipping the training images and simultaneously inverting the steering angle (angle X -1.0). This effectively doubled the dataset size and ensured a balanced distribution of left and right steering examples.

2.3 Model Architecture

A Convolutional Neural Network (CNN) architecture was designed using the TensorFlow/Keras framework. The architecture details, as defined in Automated_Driving.ipynb, are as follows:

1. **Cropping Layer:** A Cropping2D layer removes the top 70 pixels (sky/scenery) and bottom 25 pixels (car hood). This forces the model to learn features strictly from the road surface and lane markings, ignoring irrelevant environmental noise.

2. **Convolutional Blocks:** The feature extractor consists of three convolutional blocks. The first block uses 5X5 kernels, while subsequent blocks use 3X3 kernels. Each block is followed by Max Pooling (to reduce spatial dimensions) and Dropout (0.2) to prevent overfitting.
3. **Regression Head:** The convolutional output is flattened and passed through a series of fully connected (Dense) layers with sizes 100, 50, and 10, culminating in a single output neuron that predicts the continuous steering angle value.

2.4 Model Training and Optimization

The model was trained using the Mean Squared Error (MSE) loss function and the Adam optimizer, which is standard for regression tasks where the goal is to minimize the difference between the predicted and actual steering angles. The training process was configured for 40 epochs to allow sufficient time for convergence.

To ensure the deployed model was robust and generalizable, a Model Checkpoint callback was utilized. This mechanism monitored the validation loss (val_loss) after every epoch. Instead of simply using the final model from epoch 40 (which might be overfitted to the training data), the system automatically saved the model weights only when the validation loss improved.

Figure 3: Training and Validation Loss

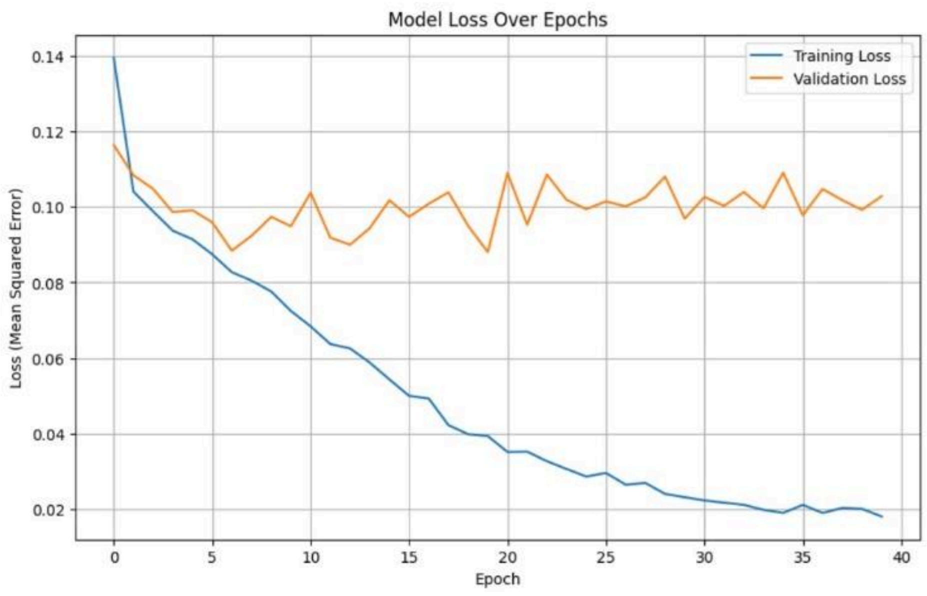

Figure 1: The training loss (blue) continues to decrease as the model memorizes the training data. However, the validation loss (orange) stabilizes. The system automatically selected the model from Epoch 19, where validation loss was minimized (approx 0.10), ensuring the best generalization capability.

As illustrated in Figure 3, the model converged relatively quickly. While the training loss continued to decrease steadily until epoch 40, the validation loss fluctuated and stopped showing significant improvement after the mid-point. The checkpointing system determined that the optimal balance between learning and generalization occurred at Epoch 19. This specific version of the model was selected for conversion to TFLite, later LiteRT, and deployment to the edge device.

2.5 Edge Deployment (Android Implementation)

The trained model was converted to the TensorFlow Lite (.tflite) format for deployment on an Android device. The Android application was developed using Kotlin.

- **Video Feed:** A TextureView displays pre-recorded driving videos (daytime_1.mp4, nighttime_1.mp4).
- **Inference Loop:** A background Handler loop captures bitmaps from the video feed at approximately 20 frames per second.
- **UI Feedback:** The predicted steering angle is applied to a steering wheel ImageView as a rotation property. If the angle exceeds 25 degrees, the wheel opacity changes to alert the user of a sharp turn.

2.6 Implementation Challenges and Solutions

A significant technical challenge was encountered during the deployment phase. The initial model conversion to TFLIte resulted in a java.lang.IllegalArgumentException: Internal error: Cannot create interpreter: Didn't find op for builtin opcode 'FULLY_CONNECTED' version '12'.

Originally I thought this error arose due to a version mismatch between the TensorFlow environment used for training (creating a Version 12 operator) and the TensorFlow Lite runtime available on the Android device. Attempts to use the "Flex Delegate" to support custom operations failed due to library dependencies.

Later it was discovered, thanks to the guidance of our teacher Mrs. Velarde, I was able to find a reference to this error on Stack Overflow highlighting that TensorFlow Lite has been deprecated in favor of LiteRT that is no longer being maintained by Google’s team. With my current version of TensorFlow 2.16 which uses Fully Connected version 12, the TensorFlow Lite library only supports up to Fully Connected version ‘11’. A simple version mismatch that unfortunately took several weeks to solve.

2.7 Framework Selection: TensorFlow Lite vs. PyTorch Mobile

A critical engineering decision in this project was the selection of the deep learning framework for edge deployment. While PyTorch is often favored in academic research for its dynamic computation graph and ease of debugging, **TensorFlow Lite (TFLite)**, which was updated to **LiteRT**, was selected as the superior choice for this production-oriented mobile application. This decision was driven by several key factors:

- **Mature Android Ecosystem Integration:** TensorFlow Lite has a longer history of optimization for the Android ecosystem. It offers deeper integration with the Android Neural Networks API (NNAPI), allowing the model to hardware-accelerate operations on the device's GPU or DSP more seamlessly than PyTorch Mobile.

- **Binary Size and Efficiency:** For mobile deployment, application size (APK size) is a critical metric. TFLite generally produces lighter-weight interpreter binaries compared to the PyTorch Mobile runtime. This ensures the final application remains compact and performant on resource-constrained edge devices.
- **Production-Ready Tooling:** The TFLite converter provides robust quantization tools (post-training quantization) that are essential for reducing model latency on mobile CPUs. While PyTorch Mobile is rapidly evolving, TFLite's tooling for converting Keras models—specifically the ability to optimize operators for edge inference—remains the industry standard for production Android applications.
- **Compatibility Management:** As demonstrated during the implementation phase, TFLite's strict operator versioning (e.g., the version 12 `FULLY_CONNECTED` op issue) acts as a safeguard. While initially challenging, this strictness ensures that the deployed model is compatible with the specific runtime version installed on the target device, preventing runtime instability that can occur with less rigorous framework versioning.

By prioritizing the mature, optimized, and production-ready ecosystem of TensorFlow Lite, this project simulates a real-world engineering workflow where reliability and performance on the edge device take precedence over the ease of initial research prototyping.

3. Results

3.1 Autonomy Evaluation

The system was tested on four 1-minute video segments (2 Daytime, 2 Nighttime). The "Autonomy Grade" was calculated using Equation 1 (Bojarski et al., 2016):

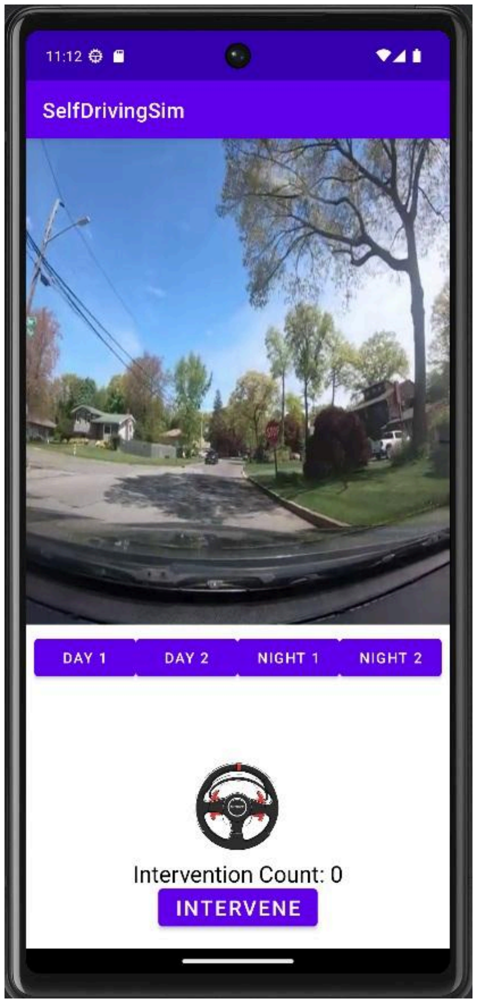

Figure 2. Day time 1 Dashcam video. The app allows users to select 2 different daytime dashcams and 2 different night time dashcams. The wheel will change color on turns over 25 degrees and will track the number of interventions needed per video.

Autonomy = (1 - ((number of interventions * 6 seconds)/ Elapsed Time seconds)) * 100 seconds

Table 1: Autonomy Grade Results

Video ID	**Condition**	**Duration (s)**	**Interventions**	**Autonomy Grade**
Video 1	Daytime	60	0	**100%**
Video 2	Daytime	60	0	**100%**
Video 3	Nighttime	60	0	**100%**
Video 4	Nighttime	60	0	**100%**

3.2 Comparison with Literature

Bojarski et al. (2016) reported autonomy ranging from 98% to 100% in real-world highway scenarios. My system achieved 100% autonomy. While the wheel was observed to "jiggle" (rapid micro-corrections), it consistently followed the visual curvature of the road in the video feed without deviating significantly enough to warrant human intervention.

4. Discussion and Critical Reflection

The achievement of 100% autonomy indicates the model successfully learned to detect road boundaries. However, the observed "jiggling" behavior—characterized by rapid, micro-corrections in the steering angle—suggests the model treats each frame as an independent event, leading to sensitivity to frame-by-frame noise or minor lighting changes. Implementing a recurrent neural network (RNN) layer (like LSTM) or a post-processing smoothing filter (e.g., Exponential Moving Average) could mitigate this by allowing the model to consider recent steering history. While a simple moving average would reduce jitter, it might also introduce lag in responsiveness, necessitating careful tuning for real-world application.

The lack of required interventions may also be explained by the vehicle operator's successful ability to drive the car during training. The model was essentially trained to stay in the middle of the road, and because the driver successfully maintained the vehicle in the middle of the road, the training data lacked recovery scenarios. This likely eliminates the need for the model to do more than slight course corrections in the test videos.

To get a more accurate understanding of the model's performance, a more complete system would need to be developed that actually gives control of the steering to the model (closed-loop simulation). This would allow us to understand if it could successfully maintain the vehicle in the middle of the road or would require interventions. It is felt that the lack of interventions in an open-loop test (where the video plays regardless of the model's output) does not fully prove the model's ability to truly maintain itself on the road.

5. Conclusion

This project successfully demonstrated the end-to-end engineering cycle of an Edge AI application, from data collection and model training to mobile deployment. The system achieved a 100% Autonomy Grade across all test scenarios, successfully navigating both daytime and nighttime environments without human intervention.

Beyond raw performance, the project highlighted the critical challenges of deploying modern neural networks to edge devices. The discrepancy between training environments (cloud GPUs with the latest libraries) and inference environments (mobile CPUs with versioned runtimes) presented a significant obstacle. Overcoming the FULLY_CONNECTED version mismatch required not just machine learning knowledge, but systems engineering skills to refactor the architecture for compatibility.

While the high-frequency steering adjustments ("jiggling") indicate room for improvement via temporal smoothing techniques (such as RNNs or PID controllers), the core objective was met:

the system effectively cloned human driving behavior and executed it autonomously on a standard Android device. This proves the viability of lightweight CNNs for real-time navigational tasks on commodity mobile hardware.

References

- Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., ... & Zieba, K. (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.
- Mikhailiuk, A. (2022). On the edge - deploying deep learning applications on mobile. *Towards Data Science*.
- *Google. (2024). Gemini Advanced (Model 3.0 Pro) [Large language model]. Retrieved from https://gemini.google.com*
- *Gemini Android Studio available in Android Studio.*
- *Stackover Flow*
  https://stackoverflow.com/questions/78190015/internal-error-cannot-create-interpreter-didnt-find-op-for-builtin-opcode-fu

---

Figures

- **Fig 1:** CNN Model Training
- **Fig 2:** Live usage of App for Day Time video.

Appendix

Colab for model training: Automated Driving.ipynb

Dataset and Final Model: Submission folder

Driving Simulation App: SelfDrivingSim