AUTONOMOUS VEHICLE PATH PLANNING USING PARALLEL PROCESSING

Balasubramaniam Renganathan

Derek Koechlin

Shiva Kumar Dhandapani

Zachary Becker

Introduction and Problem Statement

Many Machine Learning based algorithms are recognized for their quick response and high performance, but are the results fast enough for real-time autonomous vehicle path planning?

As autonomous navigation prioritizes real-time factors like human security, vehicles must not only recognize the road and obstacles; but also, anticipate and adapt to the fast-paced environment, all while avoiding collisions. Traditional path planning algorithms are limited in unstructured or unpredictable environments, and demand significant manual effort to design and optimize. This project develops an end-to-end learning system capable of mapping raw video input directly to steering commands, leveraging parallelism to provide efficient training and validation. This will lead to an improvement in performance from a serial implementation of the model, enabling faster response times for real-world navigation tasks.

Dataset

One way the autonomous navigation models are trained is through the use of image frames from real driving scenarios for visual cognition and recognition. Each image frame is typically paired with steering angle data to mimic the driving characteristics of the vehicle. The dataset we use is the CARLA dataset, which contains "road images captured during simulated driving sessions" on the CARLA simulator, paired with the corresponding steering angle. Specifically, it contains 186k images and 15 text files of steering data. There is one large training dataset and 14 different testing datasets, each addressing different weather conditions. Given the large size and complexity of the CARLA dataset, a parallel and distributed approach is necessary.

CNN Model

Our CNN used in this project is inspired by NVIDIA's research on end-to-end learning for self-driving cars. This method bypasses traditional modular approaches such as lane detection, path planning, and control by training a unified CNN directly from raw image inputs to steering angle outputs. It simplifies the pipeline and potentially improves performance by optimizing the entire system jointly.

The CNN architecture is designed to process normalized input images, extract features through convolutional layers, and output a steering angle prediction. The model contains the following components:

- A flattening of the input image size
- Five convolutional layers for feature extraction:
    - The first three layers use a 5x5 kernel with a stride of 2, reducing spatial dimensions but preserving pattern

    - The last two layers use a 3x3 kernel with a stride of 1, enabling finer feature extraction
    - Each convolutional layer uses a ReLU activation function to introduce non-linearities and improve representational power
- Four fully connected layers act as a controller:
    - Layers have 1164, 100, 50, and 10 neurons respectively
    - A final output layer with a single neuron predicts the steering angle
    - The fully connected layers process the flattened output of the convolutional layers to produce steering predictions
- The final prediction represents the steering command as the inverse turning radius, avoiding singularities when the vehicle is moving straight

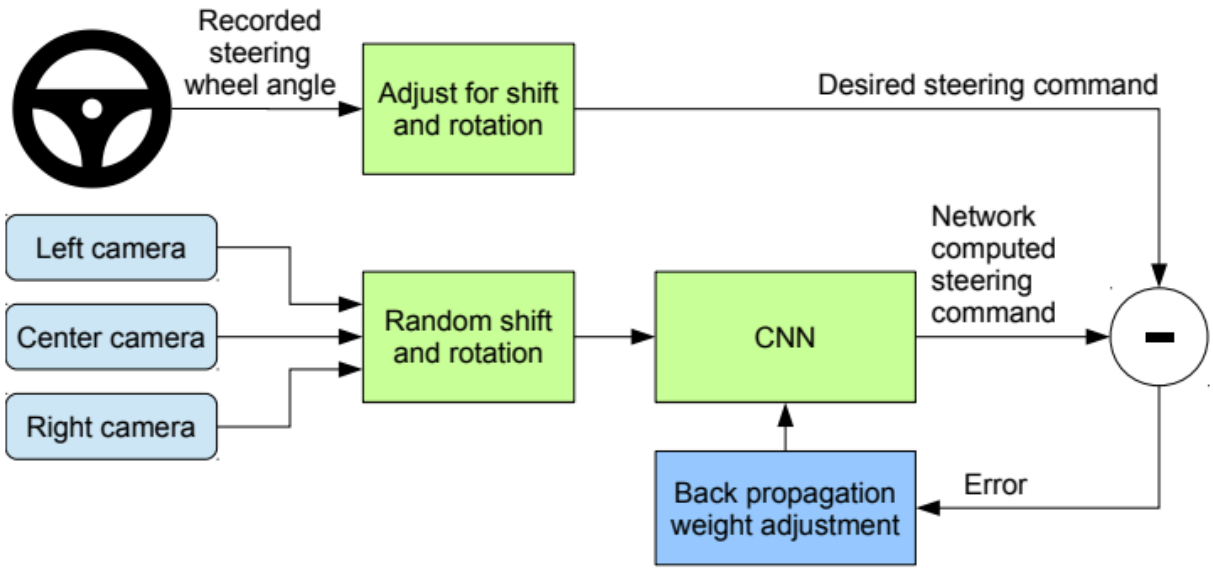


Fig. 1: Per Nvidia, images are given to a CNN which computes a proposed steering command. The proposed command is compared to the desired command for that image and the weights of the CNN are adjusted to bring the CNN output closer to the desired output.

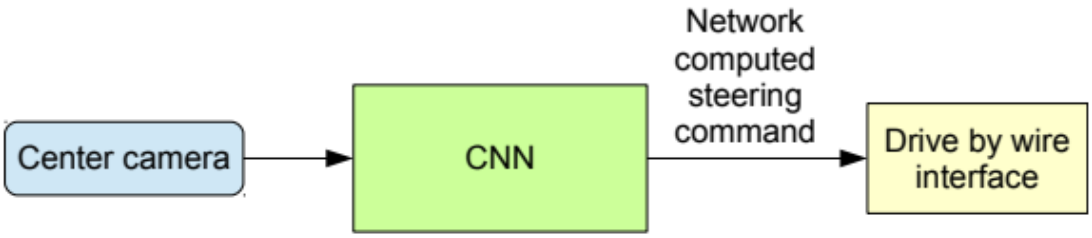


Fig. 2: Nvidia states that the network generates steering commands from a single front-facing center camera.

While the NVIDIA paper utilizes a similar structure, this implementation adapts the methodology to accommodate training on the CARLA dataset and focuses on efficient integration with PyTorch.

- The Mean Squared Error between the predicted and actual steering angles is used as the loss function and is minimized during training
- Input images are augmented with random rotations and horizontal flips in order to mimic off-center and rotated viewpoints
    - This ensures that the network learns to recover from deviations and generalizes well
- The Adam optimizer is used for its adaptive learning rate and robustness to gradients of varying scales
- Training is performed using a batch size of 2048 over 10 epochs
- The training module splits the dataset into training (80%) and validation (20%) sets for evaluation
    - Optimizes using the Adam optimizer and measures performance via metrics like MSE, MAE, R^2 score, and elapsed training time
    - Logs epoch duration, training/validation losses, and performance metrics

ResNet Model

ResNet-SteeringModel is a deep learning model based on a pre-trained ResNet-18 architecture, designed to predict a continuous value representing the steering angle. The model leverages transfer learning by fine-tuning the ResNet-18, originally trained on the ImageNet dataset, for a regression task rather than classification.

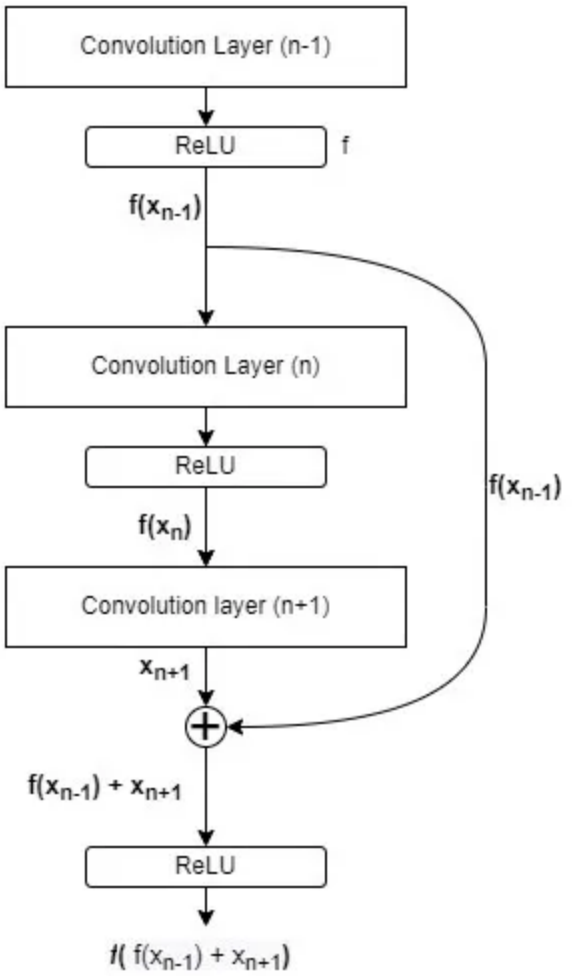


Fig. 3: Architecture of Residual Block in a ResNet Model

- The model utilizes ResNet-18, a residual network with 18 layers. Residual networks are known for their use of skip connections, which help mitigate the vanishing gradient problem and enable training of deeper networks.

- The model is initialized with ResNet-18 pre-trained weights from the ImageNet dataset (IMAGENET1K_V1). These weights provide a strong feature extraction backbone for visual tasks.
- The final fully connected layer of ResNet-18, originally designed to output 1,000 classes, is replaced with a single-node fully connected layer (nn.Linear(self.resnet.fc.in_features, 1)). This change converts the model into a regression model capable of predicting a single continuous output (steering angle).
- Just like CNN model Adam optimizer is used here for its adaptive learning rate and robustness to gradients of varying scales

Model Architecture

The model architecture is based on the ResNet structure and consists of the following components:

- Input Layer: Images are resized to 224x224 pixels and normalized for optimal feature extraction.
- Residual Blocks: Multiple convolutional layers with skip connections, enabling the network to learn residual mappings instead of direct transformations:
    - Residual connections add the input of each block to its output, allowing efficient backpropagation and gradient flow.
    - All convolutional layers use the ReLU activation function to introduce non-linearities and enhance representational power.
- Fully Connected Layers: The output of the convolutional blocks is flattened and passed through fully connected layers to produce a steering angle prediction.
    - Final output layer consists of a single neuron, representing the predicted steering angle as the inverse turning radius.
    - This approach avoids singularities for straight driving paths.

Layer Name	Output Size	ResNet-18 Blocks	Custom Modifications
conv1	112x112	7x7, 64, stride 2	Same
conv2_x	56x56	3x3, 64, x2	Same
conv3_x	28x28	3x3, 128, x2	Same
conv4_x	14x14	3x3, 256, x2	Same
conv5_x	7x7	3x3, 512, x2	Same
fc	1x1	Linear (1)	Output: Steering Angle

Fig. 4: Custom ResNet-18 Model Architecture

Training Process

The model is trained using the following strategy:

- Loss Function: The Mean Squared Error (MSE) loss is minimized to reduce the difference between predicted and actual steering angles.
- Data Augmentation: Input images are augmented with random rotations and horizontal flips to simulate different driving conditions and improve generalization.
- Optimizer: The Adam optimizer is employed for its adaptive learning rate and robustness to gradient variations.
- Training and Validation:
    - The dataset is split into training (80%) and validation (20%) sets.
    - Training is conducted over 10 epochs with a batch size of 32.
    - The model performance is evaluated using metrics such as MSE and RMSE (Root Mean Squared Error).

- Device Utilization: The training process is optimized to leverage GPU acceleration if available.
- Logging and Monitoring: Training progress, including epoch duration, training/validation losses, and performance metrics, is logged for analysis and debugging.

Parallelism

- Serial Code:
    - Uses a single CPU core for training torch.set_num_threads(1).
    - Data is loaded sequentially with a single worker per data loading task.
    - No parallelization in model training.
- Parallel Code (using torch's built-in method):
    - Uses all 24 CPU cores for training, default torch setting is all cores. The behavior of PyTorch's CPU threading occurs during training, when tensor operations are encountered (i.e. matrix multiplication or convolutions), threads from a thread pool break up the work into chunks and operate on it in parallel. This is achieved through an OpenMP backend C/C++ library in PyTorch.
    - Data is loaded in a distributed parallel fashion and is defined by num_workers: The number of workers for data loading is set to 8, which means it scales with the number of available devices. 8-cores for data loading is a sweet spot between increasing training speeds while minimizing additional overhead of creating and managing additional processes
    - The use of multiple workers helps in loading and augmenting data in parallel while training the model on GPU

Performance Analysis

1. CNN Model

Table 1: Training and Validation Metrics Comparison (Serial vs. Parallel Implementation for CNN)

Epoch	Implementation	Training Loss	Validation Loss	MAE	R^2	Epoch Duration (s)
1	Serial	0.0189	0.0087	0.0388	-0.0391	3001.36

	Parallel	0.0104	0.0074	0.0330	-0.0060	244.75
2	Serial	0.0078	0.0084	0.0384	0.0007	2857.27
	Parallel	0.0072	0.0061	0.0372	0.1808	176.86
3	Serial	0.0076	0.0082	0.0388	0.0270	2590.18
	Parallel	0.0059	0.0048	0.0339	0.3514	197.64
4	Serial	0.0074	0.0077	0.0403	0.0815	2630.67
	Parallel	0.0050	0.0044	0.0311	0.4144	204.11
5	Serial	0.0069	0.0072	0.0409	0.1470	2762.20
	Parallel	0.0044	0.0038	0.0298	0.4912	175.71
6	Serial	0.0064	0.0065	0.0409	0.2358	2746.46
	Parallel	0.0049	0.0044	0.0327	0.4432	336.58
7	Serial	0.0057	0.0056	0.0354	0.3418	2736.98
	Parallel	0.0042	0.0039	0.0308	0.5027	340.64
8	Serial	0.0051	0.0051	0.0347	0.4089	2765.28
	Parallel	0.0037	0.0035	0.0294	0.5558	341.13
9	Serial	0.0046	0.0046	0.0313	0.4592	2707.20
	Parallel	0.0032	0.0031	0.0270	0.6097	337.09

10	Serial	0.0041	0.0044	0.0324	0.4898	2496.96
	Parallel	0.0029	0.0027	0.0247	0.6570	336.67

Reduced Epoch Duration: Parallel processing reduced the epoch duration from over 3000 seconds in the serial implementation to approximately 336 seconds, translating to an approximately 90% reduction in computation time.

Improved Training and Validation Metrics: Training and validation losses steadily decreased across epochs for both implementations. However, the parallel model consistently achieved better loss values:

- Final training loss: 0.0029 (parallel) vs. 0.0041 (serial)
- Final validation loss: 0.0027 (parallel) vs. 0.0044 (serial)

Higher Model Accuracy: The parallel implementation showed a marked improvement in the R^2 score, increasing from 0.4898 (serial) to 0.6570 (parallel) in the final epoch. This indicates a more accurate representation of steering angles.

CNN Parallel test results:

Mean Squared Error (MSE): 0.0009
Mean Absolute Error (MAE): 0.0256

CNN Serial test results:

Mean Squared Error (MSE): 0.0029
Mean Absolute Error (MAE): 0.0529

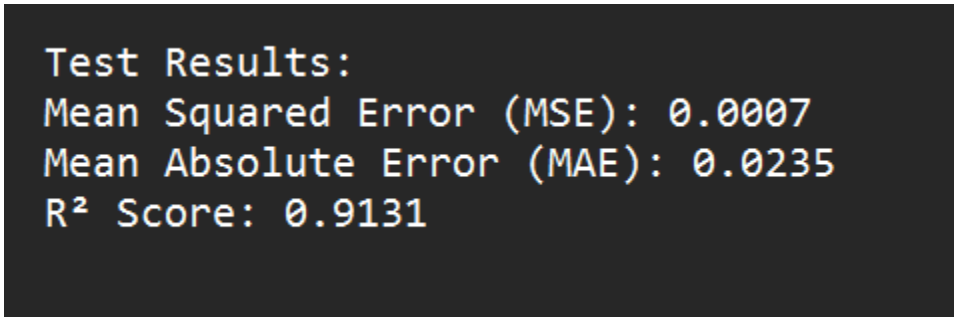
2. ResNet Model

Table 2: Training and Validation Metrics Comparison (Serial vs. Parallel Implementation for ResNet)

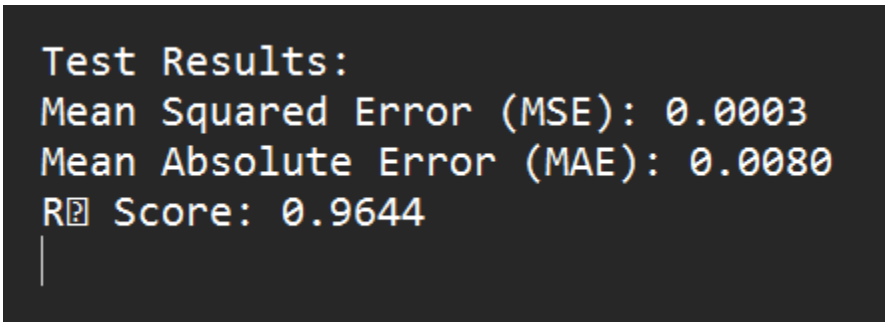
Epoch	Serial Avg Loss	Serial Val Loss	Serial Val RMSE	Parallel Avg Loss	Parallel Val Loss	Parallel Val RMSE

1	0.0086	0.0143	0.1192	0.0145	0.0019	0.0433
2	0.0014	0.0008	0.0264	0.0018	0.0011	0.0330
3	0.0010	0.0006	0.0229	0.0012	0.0057	0.0753
4	0.0011	0.0005	0.0199	0.0009	0.0005	0.0221
5	0.0010	0.0044	0.0660	0.0009	0.0013	0.0355
6	0.0009	0.0017	0.0400	0.0008	0.0013	0.0358
7	0.0006	0.0004	0.0166	0.0005	0.0009	0.0273
8	0.0006	0.0004	0.0176	0.0006	0.0011	0.0322
9	0.0004	0.0006	0.0225	0.0006	0.0006	0.0240
10	0.0004	0.0007	0.0265	0.0005	0.0003	0.0214

Serial Test Results:



Parallel Test Results:



Reduced Validation Loss: The parallel implementation achieved consistently lower validation losses compared to the serial version, with the final value reaching 0.0003 for the parallel model versus 0.0007 for the serial model.

Lower RMSE: The Root Mean Squared Error (RMSE) for validation decreased notably in the parallel model, with a final value of 0.0214, demonstrating improved prediction accuracy.

Faster Convergence: The parallel implementation converged faster, achieving optimal metrics earlier in the training process

Impact of Parallelism

The integration of parallel processing provided several benefits:

1. Efficiency: Computation time was dramatically reduced, enabling faster training and testing cycles.
2. Scalability: Parallelism facilitated handling of large-scale datasets, making the system suitable for real-time applications.
3. Enhanced Model Tuning: Faster iterations allowed for better hyperparameter tuning, further refining model's performance.

Validation Analysis

CNN Model

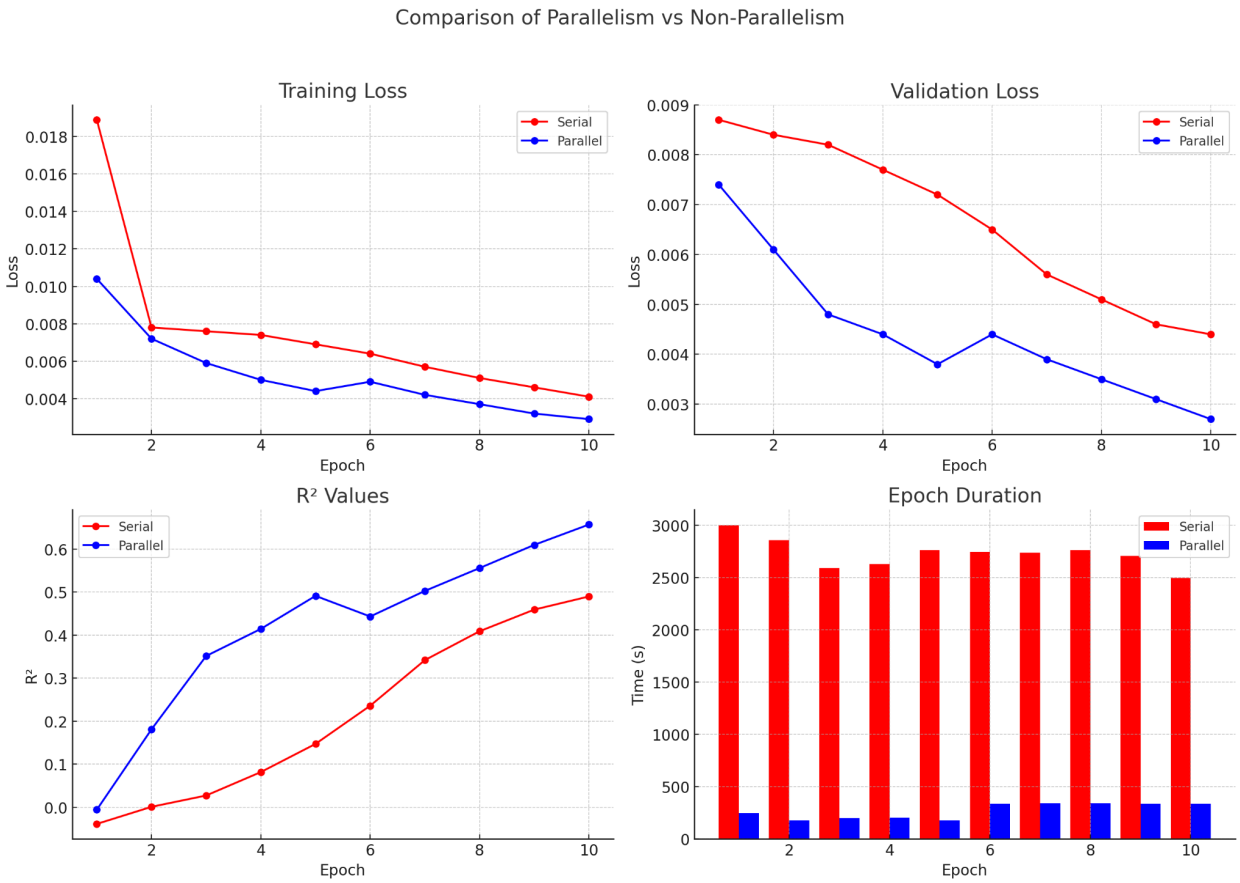


Fig. 5: Comparison of Serial and Parallel Analyses for CNN Model

Serial Analysis:

- The training and validation losses decrease consistently over the 10 epochs, with the training loss steadily improving from 0.0189 in Epoch 1 to 0.0041 in Epoch 10.
- The validation loss also shows improvement, starting from 0.0087 in Epoch 1 and reaching 0.0044 by Epoch 10, indicating that the model is generalizing well to the validation data.
- Mean Absolute Error (MAE) starts at 0.0388 and drops to 0.0324 by Epoch 10, reflecting better predictions as the model trains.
- R^2 value increases progressively, starting from a negative value (-0.0391) in Epoch 1 and reaching 0.4898 by Epoch 10, which is a significant improvement, showing that the model is learning the underlying data distribution.

- The training time per epoch gradually decreases, with the longest epoch (3001 seconds) being at Epoch 1, and the shortest (2496 seconds) being at Epoch 10, demonstrating better resource utilization over time.

Parallel Analysis:

1. The training loss decreases from 0.0104 in Epoch 1 to 0.0029 in Epoch 10, with the validation loss also improving from 0.0074 in Epoch 1 to 0.0027 in Epoch 10. This shows faster convergence compared to the serial model.
2. The MAE decreases from 0.0330 in Epoch 1 to 0.0247 in Epoch 10, indicating more accurate predictions as training progresses.
3. The R^2 value increases sharply over the epochs, starting from a low value near 0 and reaching 0.6570 by Epoch 10, which suggests the parallel implementation enhances the model's ability to fit the data better.
4. Epoch durations are significantly shorter compared to the serial implementation, ranging from 175.71 seconds in Epoch 5 to 350.20 seconds in Epoch 1, showcasing the speedup due to parallelization.

ResNet Model

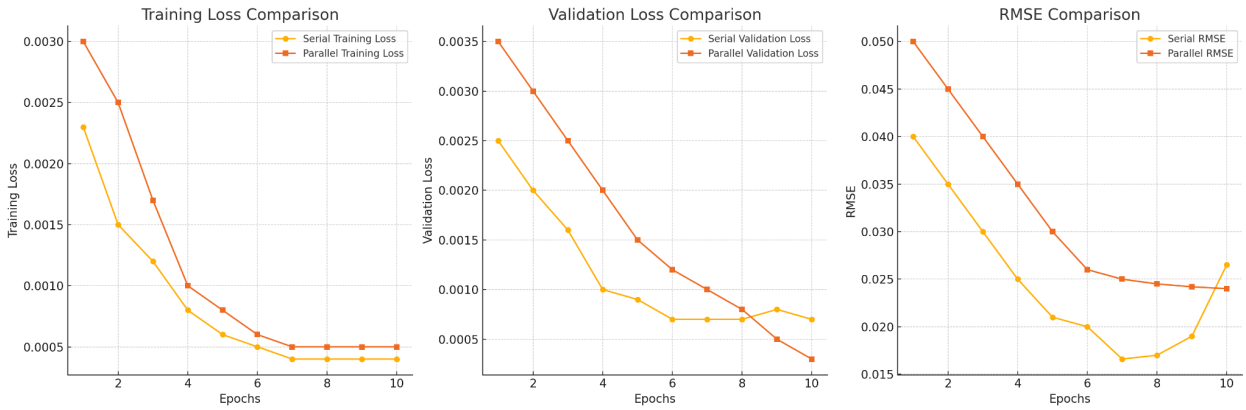


Fig. 6: Comparison of Serial and Parallel Analyses for ResNet Model

Serial Analysis:

- The training loss shows a rapid decrease in the first few epochs, from 0.0086 in Epoch 1 to 0.0006 in Epoch 7, with small variations later on.
- Validation loss also drops, starting at 0.0143 in Epoch 1 and reaching 0.0004 by Epoch 8, with the lowest value (0.0004) occurring in multiple epochs, indicating stable performance.

- The Validation RMSE decreases consistently over time, indicating better generalization to the validation set. It starts at 0.1192 in Epoch 1 and improves to 0.0166 by Epoch 7.
- The model shows very stable results, with the validation loss fluctuating minimally between epochs after reaching its lowest point around Epoch 4.

Parallel Analysis:

- The training loss decreases from 0.0086 in Epoch 1 to 0.0010 by Epoch 5, which is slightly more gradual than in the serial implementation.
- Validation loss improves from 0.0143 in Epoch 1 to 0.0027 in Epoch 10, indicating efficient generalization.
- Validation RMSE shows a continuous drop from 0.1192 in Epoch 1 to 0.0264 in Epoch 10, demonstrating improved performance over the epochs.
- Epoch durations are relatively shorter in the parallel implementation compared to the serial version, with each epoch taking between 244.75 and 350.20 seconds.

Conclusion

In conclusion, both CNN and ResNet benefit from parallelization in terms of training time and generalization to validation data. The ResNet model provides slightly better performance in terms of validation loss and RMSE, making it a stronger choice for the task at hand. However, the CNN model offers a good trade-off between performance and computational efficiency when parallelized. This project emphasizes the feasibility of leveraging parallelism and distributed computation for large-scale deep learning tasks in autonomous driving. The results validate the quality of the learned model and highlight the potential to scale the system further if needed for various industrial applications. Future work may focus on further optimizing hyperparameters and exploring more complex architectures or separate models to handle dynamic and unpredictable driving scenarios more effectively.

How work was split up among team members

We collectively decided that we liked the idea of parallelizing an autonomous driving system task, and started to create initial model and testing scripts. Balasubramaniam and Shiva handled the ResNet analyses, Derek created the bulk of the CNN python scripts, and Zach and Shiva helped to log more things in the model and refine the code. We split up the validation analyses, running the task with serial and parallel implementations. Zach worked to create the outline of the report and slides, and Derek, Balasubramaniam and Shiva inserted the results when they were gathered.

References:

Dataset

https://www.kaggle.com/datasets/zahidbooni/alltownswithweather?resource=download-directory

Torch Multithreading

https://pytorch.org/docs/stable/notes/cpu_threading_torchscript_inference.html

Nvidia CNN Model

https://arxiv.org/pdf/1604.07316

ResNet Model

https://arxiv.org/abs/1512.03385