Vehicle Classification with Resnet

Oleg Russu, Rohan Walia

10/14/2024

Background

In autonomous driving, classifying nearby vehicles using the onboard vision system is crucial for ensuring safe and efficient navigation. This task involves identifying and categorizing vehicles in the surrounding environment, such as cars, trucks, buses, motorcycles, and sometimes even pedestrians or cyclists. Proper classification enables the autonomous vehicle to understand the type and behavior of each object on the road, which is essential for making real-time decisions such as lane changes, braking, or adjusting the velocity of the vehicle. The objective of this assignment is to fine-tune a ResNet model to classify images of road vehicles working with the Kaggle [1] dataset to achieve this goal.

Methodology

Architecture

We use Resnet50 already pretrained on the Imagenet dataset (`models.resnet50(pretrained=True)`). The final layer of Resnet is replaced with a new linear layer to match the number of road vehicle classes taken from the dataset (10 classes).

We freeze all the layers of the Resnet except for the new classification head (model.fc) by setting `param.requires_grad = False` for all parameters except the final layer.

Data Augmentation and Normalization

In our implementation, a dictionary named `data_transforms` is created to define a series of data preprocessing steps for different datasets: `'train'`, `'val'`, and `'test'`. The preprocessing for each dataset is implemented using PyTorch's `transforms.Compose` method, which allows for the sequential application of multiple transformations. Below is a breakdown of the process:

- **Training data (`'train'`)**:

  - The images are randomly resized and cropped to a size of 224×224 pixels using `RandomResizedCrop`.
  - Random horizontal flipping is applied to augment the data using `RandomHorizontalFlip`.
  - The images are then converted to PyTorch tensors via `ToTensor()`.
  - Finally, the pixel values are normalized using mean values $[0.485, 0.456, 0.406]$ and standard deviations $[0.229, 0.224, 0.225]$, which are standard values for pre-trained models like ResNet.

- **Validation and Test data (`'val'` and `'test'`)**:
  - Both sets are resized to 256×256 pixels using `Resize(256)` and then centrally cropped to 224×224 using `CenterCrop(224)`.
  - As with the training set, the images are converted to tensors using `ToTensor()` and normalized using the same mean and standard deviation values.

Training and Evaluation

We use the following hypermeters for training:

Parameter Name	**Value**
Batch Size	16
Number of Workers	4
Learning Rate	0.001
Momentum	0.9
Weight Decay	1e-4
Learning rate step size	7
Learning rate gamma	0.1
Number of epochs	25

Table 1: Hyperparameter Values

For our training process, we use cross-entropy loss for classification, with the Stochastic Gradient Descent optimizer applied to the fully connected layer. The optimizer is configured with momentum for accelerated convergence and weight decay for regularization. Additionally, a learning rate scheduler is used to reduce the learning rate at predefined intervals to ensure efficient convergence.

Results

Training and validation metrics were logged to Tensorboard for visualization. The training process logs the training loss and accuracy, while the validation and test sets are evaluated after each epoch. The best model is saved based on validation accuracy. After training, we visualize 10 test images along with their predicted and true labels. The results are listed below

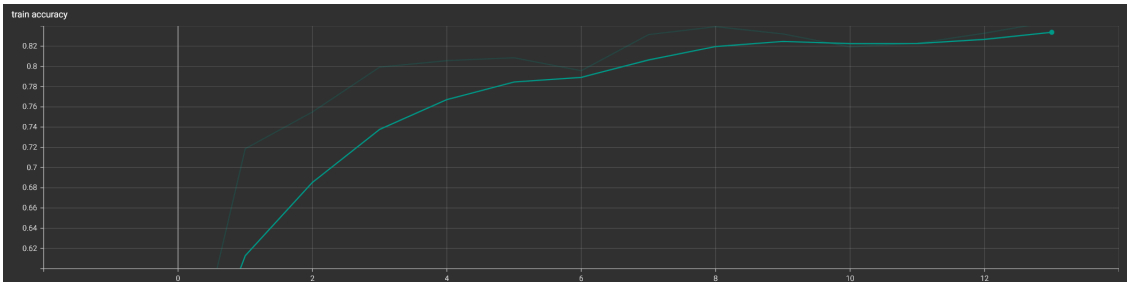

Figure 1: Train Accuracy

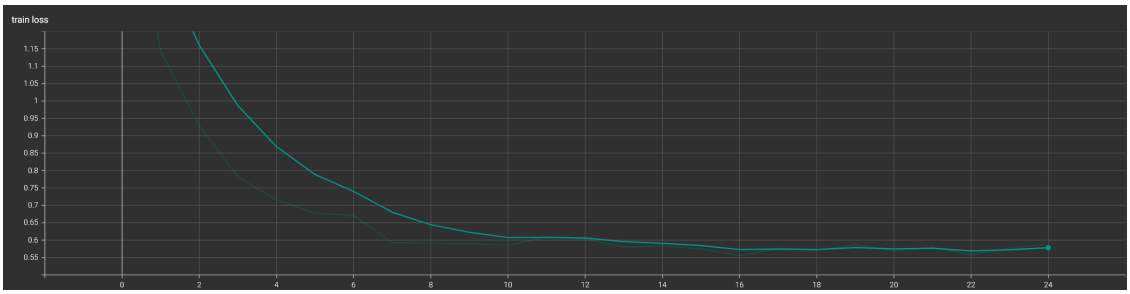

Figure 2: Train Loss

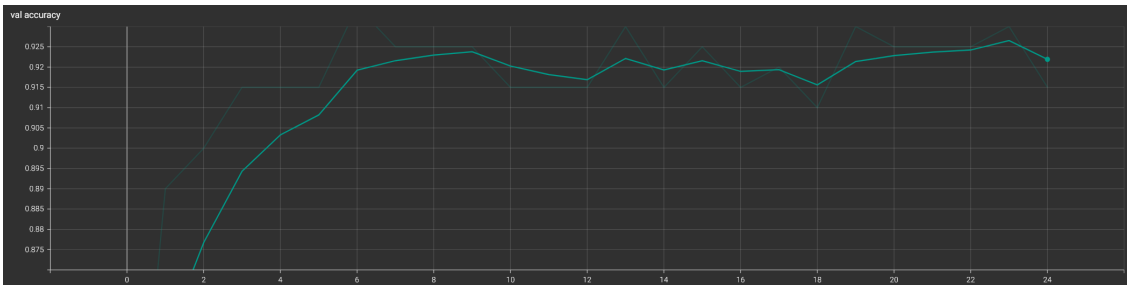

Figure 3: Validation Accuracy

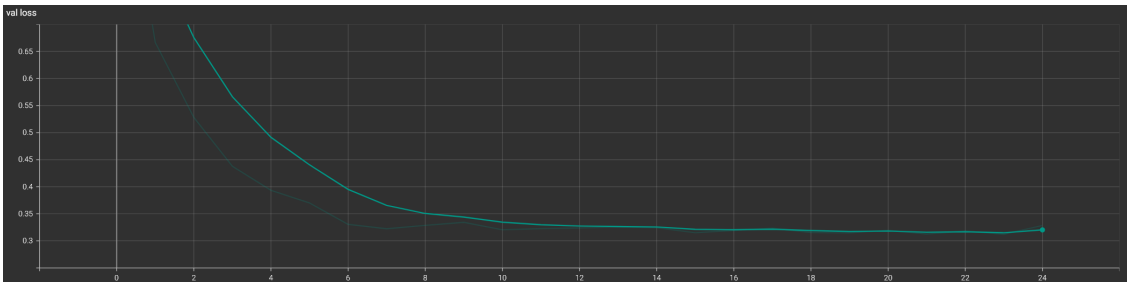

Figure 4: Validation Loss

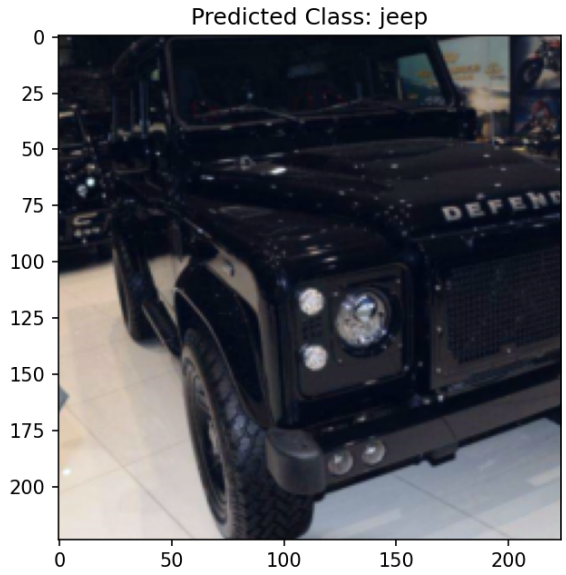

Figure 5: Predict Jeep Class

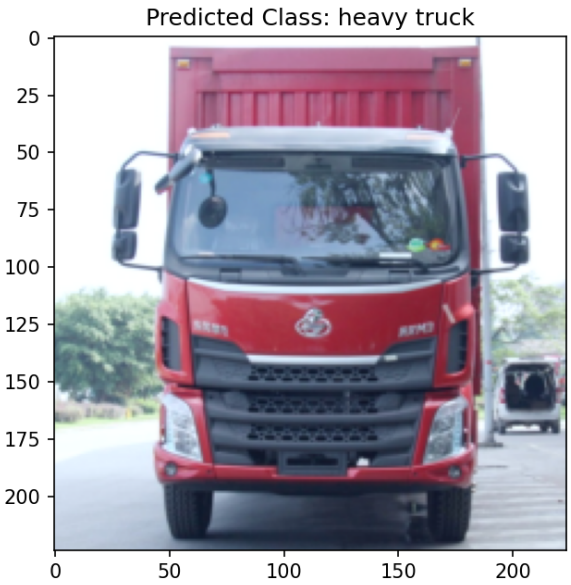

Figure 6: Predict Heavy Truck Class

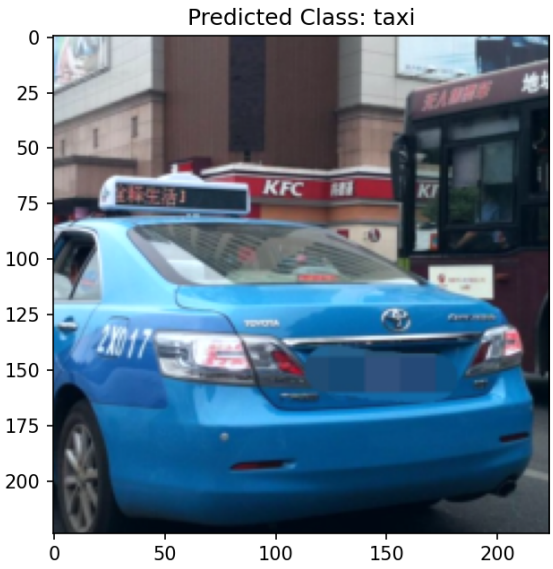

Figure 7: Predict Taxi Class

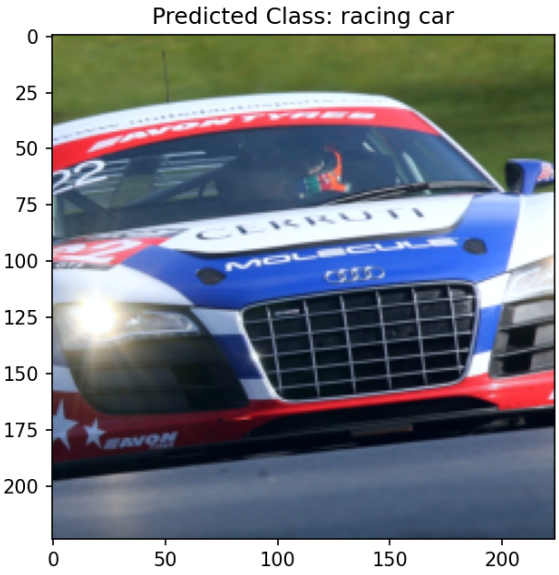

Figure 8: Predict Racecar Class

Figure 9: Predict Bus Class

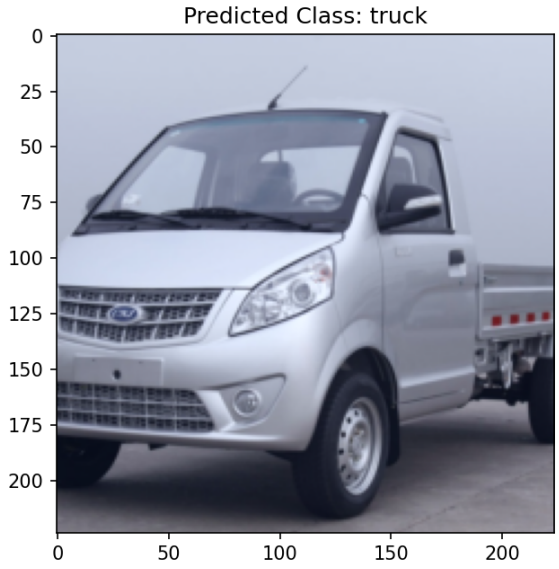

Figure 10: Predict Truck Class

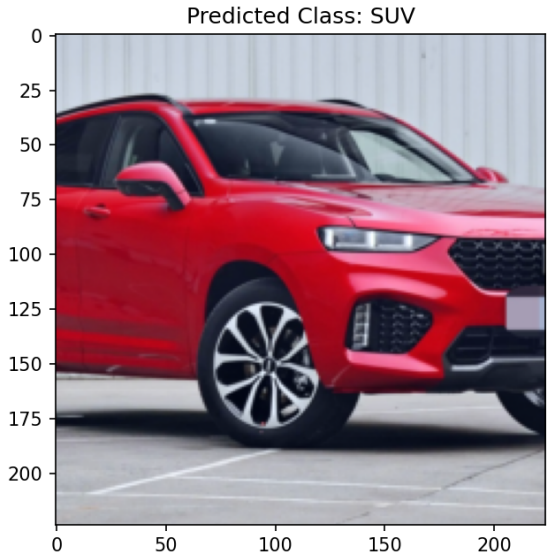

Figure 11: Predict SUV Class

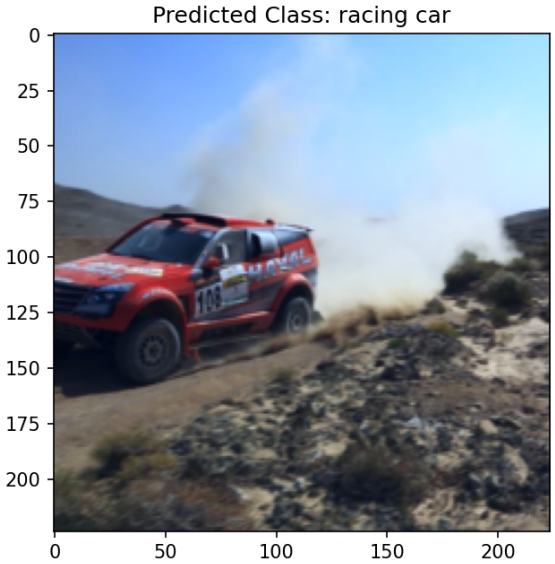

Figure 12: Predict Racecar Class 2

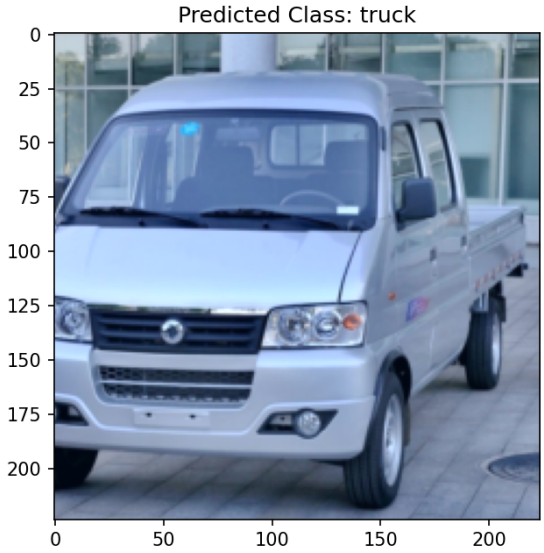

Figure 13: Predict Truck Class 2

Figure 14: Predict Truck Class 3

Optimizing the Model

The code was run on Ryzen 9 7945HX CPU and RTX 4090 GPU. We observed spikes in both the CPU and GPU utilization. The GPU in particular was sitting idle for most of the run but spiking up to 100 percent utilization for a split second every 10 or so seconds. We tried the following in the first attempt:

1. Increased the batch size to 64 to improve GPU utilization.

2. Increased the number of workers (`num_workers=8`) for data loading to better use our CPU.

3. Added to DataLoader to optimize host-to-GPU memory transfers (`pin_memory=True`).

These changes made the program run about 50 percent slower (with less GPU utilization). We brought the batch size and workers down a little with a couple of other changes for our next run:

1. Batch Size reduced to 32.

2. Workers reduced back to 4. Increasing workers should help if CPU utilization becomes the bottleneck, but according to our tests, beyond a certain point, it saturates the CPU or disk I/O capacity.

3. Keep `pin_memory=True` to help with faster transfers between CPU and GPU.

We ran the code and monitored our resource utilization (CPU, GPU, disk) via Task Manager and the results did show a modest improvement (see figure 15). Code ran about 5 percent faster than the original (before we made it slower). Based on our trials, increasing the batch size and keeping the workers at a moderate level ensures more data is fed into the GPU per iteration. This can improve utilization as long as the CPU isn't bottlenecked, and the data pipeline isn't overloaded. GPU utilization spikes will still happen (especially with a high end GPU) if the CPU is working on pre-processing (like augmentations) while the GPU waits for more data. However, given that we observed more frequent spikes suggest the CPU is keeping up a little better, and the GPU isn't sitting idle as much as before. After this run, we tried to increase the batch size back to 64, but that significantly increase the time the program took to process.

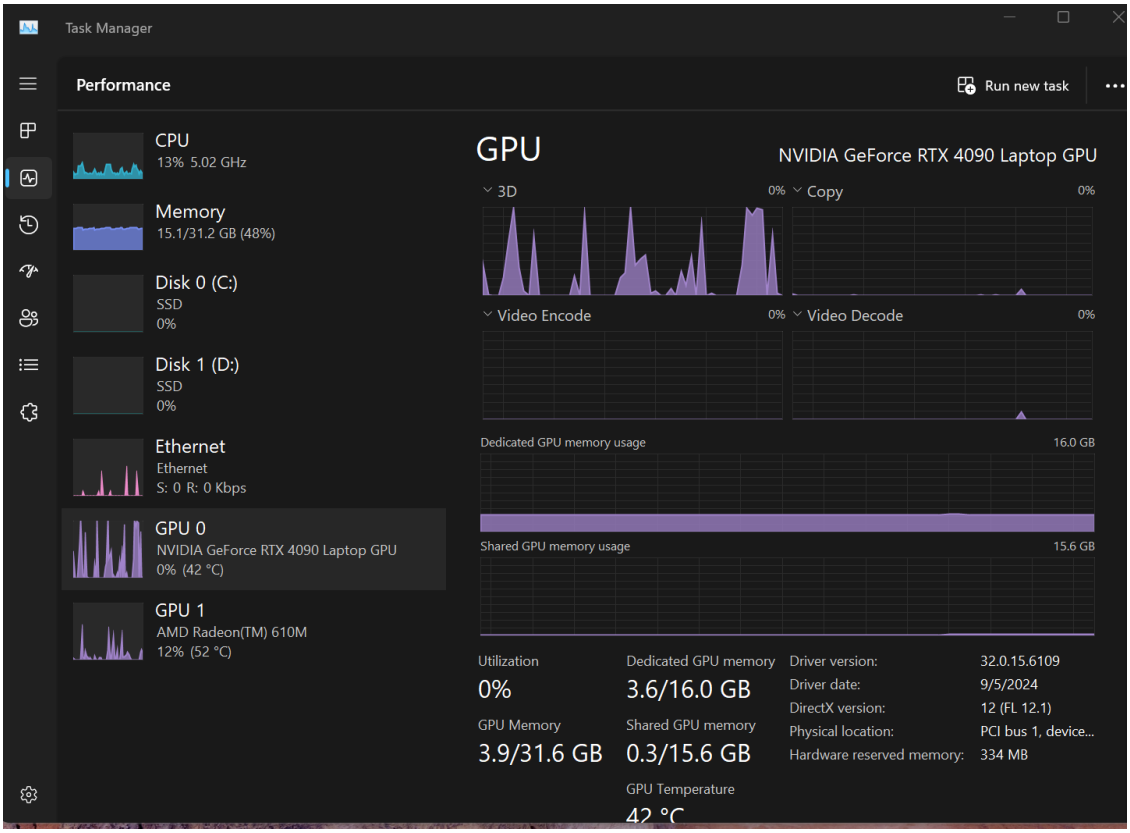

Figure 15: Best GPU Utilization Training the Vehicle Dataset

We also attempting to train the entire Resnet 152 dataset (160 GB). Initially, the GPU utilization was relatively low with big spikes to 100 (similar to what we saw when we trained the vehicle database), but the longer the program ran, the more utilization we saw. After about 4-5 minutes of ruining the program, the GPU utilization was staying at 100 almost the entire time (see figure 16)

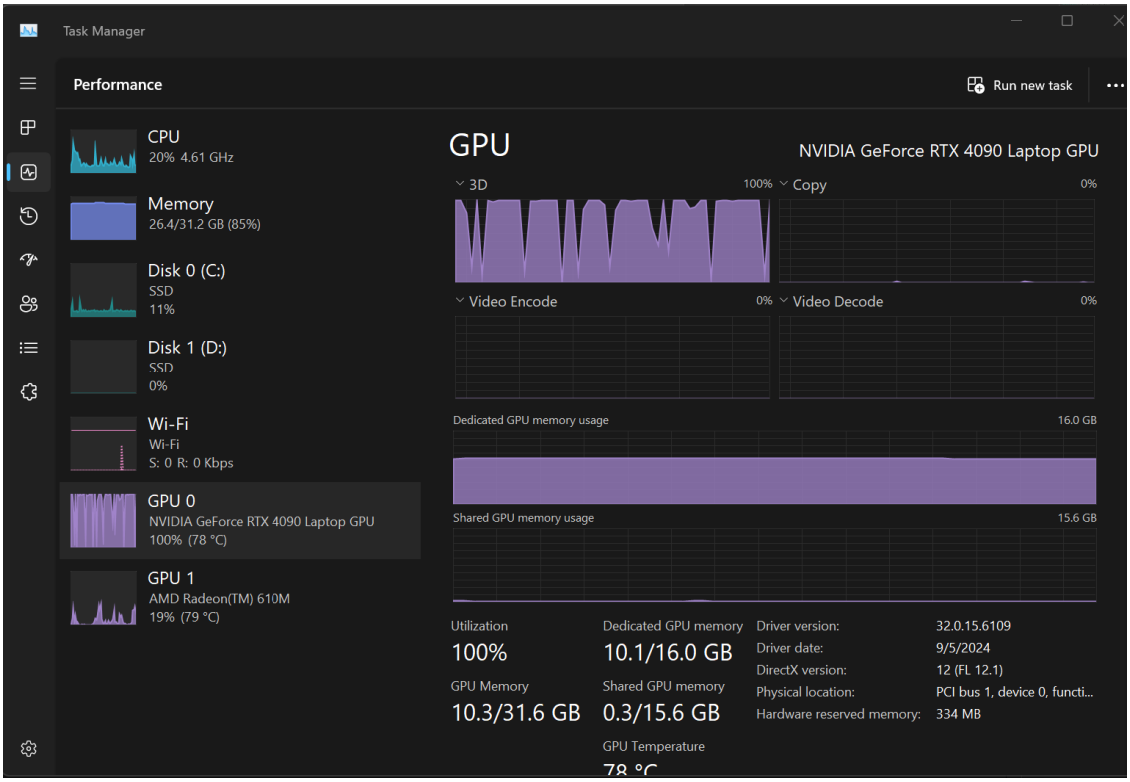

Figure 16: Best GPU Utilization Training the Entire Resent 152 Dataset

References

[1] Marquis03. Vehicle classification dataset, 2023. Accessed: 2024-09-29.