Power of Parallel Computing in Car Model Detection
Team 11
Angel Fernandes and Anish Navale

Introduction

Background:

In the dynamic arena of technological innovation, the automotive sector emerges as a crucible of progression, perennially advancing the limits of mobility and efficiency. The digital age has ushered in an era where deep learning is not merely a futuristic vision but a practical instrument, reshaping every aspect of automotive technology. From the intricacies of vehicle aesthetics to the nuanced algorithms governing safety mechanisms, the influence of sophisticated computational models is palpable. The ability to accurately recognize and classify automobile types is not just an academic exercise but a pivotal industry necessity that supports an array of critical applications, including the intricate dance of autonomous navigation, the meticulous orchestration of traffic systems, and the strategic fabric of urban infrastructural design. In the midst of this technological renaissance, expansive data repositories like the COMP-CARS dataset become invaluable treasure troves for researchers and engineers, offering a gateway to harness the prowess of deep learning for refined vehicular classification.

Motivation:

The impetus for this inquiry is rooted in the exigent demand for advanced car recognition systems, fueled by the burgeoning global vehicle population. The imperative for these technologies transcends the conventional, venturing into the realm of essential infrastructure for modern transport dynamics. The efficacy of such systems is critical in streamlining the pulsating rhythms of traffic flow, bolstering the integrity of road safety paradigms, and propelling the innovation curve in autonomous vehicle development. Our initiative to exploit the extensive and multifaceted dataset provided by COMP-CARS is driven by the vision to craft deep learning models of exceptional precision and efficiency. These models are not just academic prototypes but are envisioned as instrumental entities in the real-time identification and categorization of vehicles, replete with nuanced distinctions across an exhaustive spectrum of makes and models.

Goal:

At its heart, the project is an ambitious venture poised at the intersection of cutting-edge deep learning techniques and the practical exigencies of the automotive industry. It is a testament to the transformative potential of deep learning, with the ResNet architecture being a focal point of exploration for its proven efficacy in complex pattern recognition tasks, as well as its adaptability to a diversity of predictive analyses, including the intricate domain of biomolecular

structures. Parallel computing emerges as the linchpin in this scientific endeavor, a catalyst that promises to revolutionize the efficiency with which we process voluminous datasets and train sophisticated models. By harnessing the sheer computational might of CPUs and GPUs in tandem, we aim to shatter the temporal barriers traditionally associated with deep learning tasks. The ambitious endgame of this project is to distill a methodology that not only enhances the accuracy and speed of car classification models but also to distill insights that could have far-reaching implications in the broader context of computational biology and artificial intelligence. Our pursuit is not merely to contribute incremental advances in automotive engineering but to pave new avenues for research and development across interdisciplinary fields, anchored by the power and promise of parallel computing.

Dataset Description

The dataset is titled "Comprehensive Cars" aka "CompCars" and is available on Kaggle [1]. The dataset contains data from two scenarios, including images from web-nature and surveillance-nature. 163 automobile makes and 1,716 car models are included in the web-nature data. In total, 136,726 photos show the cars in their whole, while 27,618 photos show the individual car pieces. Bounding boxes and views are used to label the whole automobile photos. The maximum speed, displacement, number of doors, number of seats, and kind of automobile are the five attributes that are designated for each car model. There are 50,000 front-view automobile photos in the surveillance-nature data. The total size of the dataset is 17GB and we plan to use a subset of that data ranging from 2-5GB.

Methodology

1. **Data Preparation**
   Since the dataset is huge, we decided to utilize a subset of data which is relevant. We did this by performing initial exploratory data analysis according to a Kaggle notebook linked to the dataset [2] and then by reading in the data, and its respective attributes like Brand and Model Type. Data was combined by reading in the files 'car_type.mat' and 'make_model_name.mat'

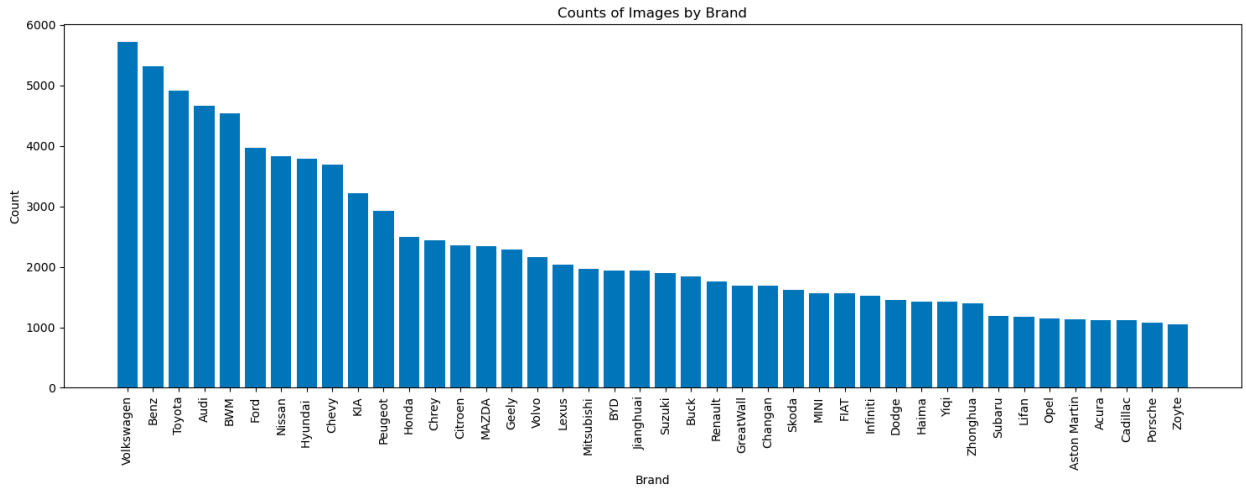


Fig. 1. Initial EDA – Number of Images by Brand

Anticipating a large computation time, we selected 4 brands having the highest number of images provided in the data across 4 model types. The brands are Audi, Volkswagen, BMW and Toyota and the 4 model types are SUV, Sedan, Hatchback and Sports. The number of Volkswagen images is 2919, the number of BMW images is 3017, the number of Audi images is 2187, and the number of Toyota images is 2551.

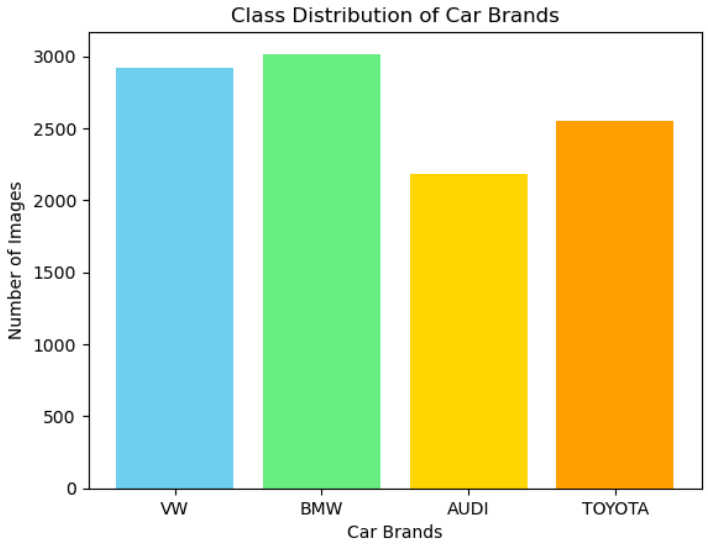


Fig. 2. Distribution of selected Car Brands

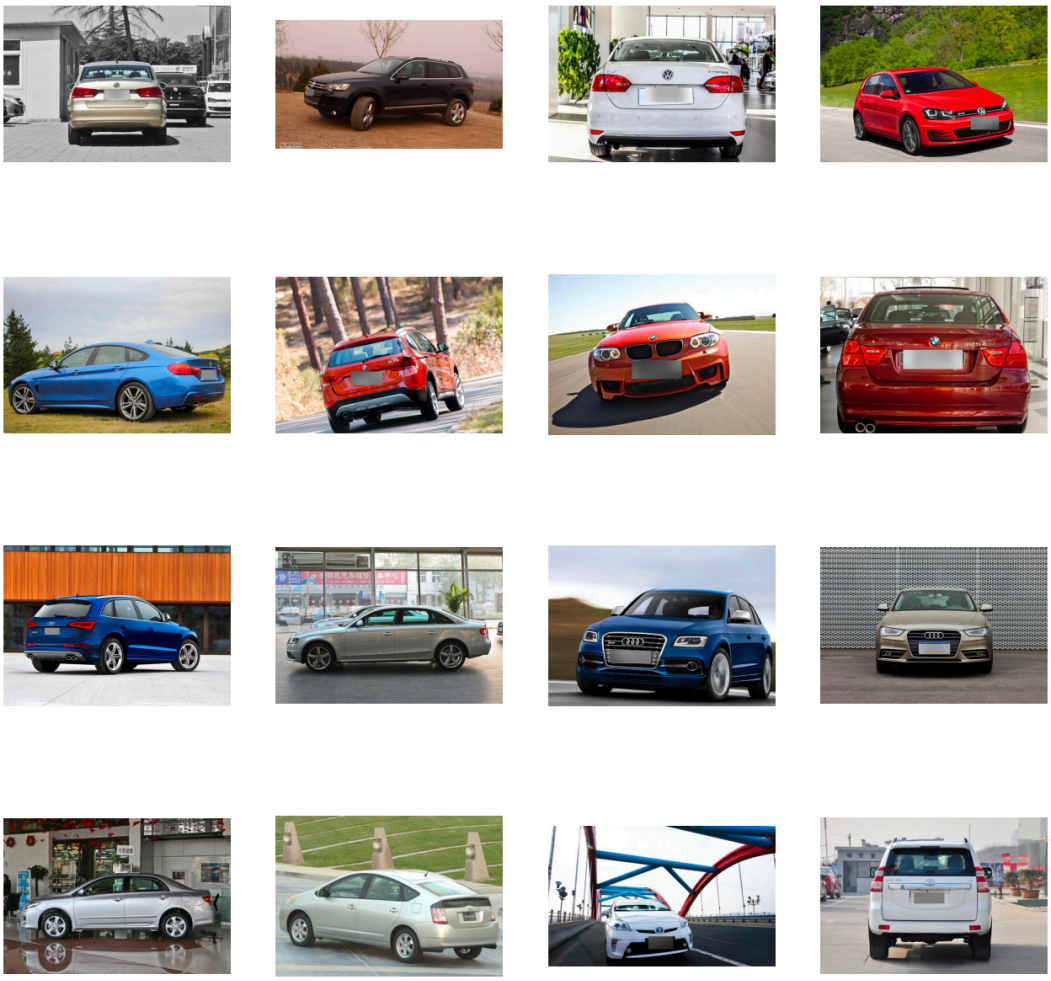


Fig. 3. Sample Images

2. **Data Processing**:

The data processing stage in machine learning pipelines is a critical phase where the raw data is transformed and prepared for the subsequent training of models. In this project, a meticulous approach has been adopted to manage the data effectively, ensuring that it is suitably structured for the training, testing, and validation of the model. This structured approach not only facilitates model training but also adheres to the best practices of data management and machine learning.

Data Splitting and Augmentation:
At the outset, the **copy_files** function serves as the workhorse for segregating the dataset into distinct training and testing subsets. This function is meticulously crafted to handle class-specific directories, allowing for the organized distribution of images across training and testing sets. Each class, represented by automotive brands in this context, is allocated a predetermined number of images—1750 for training and 436 for testing. This allocation ensures a robust and comprehensive dataset that is capable of training the deep learning models effectively.
The random selection algorithm embedded within the **copy_files** function is designed to mitigate sampling bias, a critical aspect when the objective is to construct a model that is capable of

generalizing well beyond the scope of the training data. In scenarios where the available images are fewer than the desired count, the function is programmed to adapt by utilizing the entirety of the available dataset, thereby maintaining the integrity and balance of the training and testing sets.

Parallel Processing for Efficiency:
Recognizing the potentially time-consuming nature of the copying process, the project leverages parallel processing to enhance efficiency. The **run_parallel_copy** function is a testament to the application of modern computing techniques to traditional data handling problems. It invokes the multiprocessing capabilities of Python, specifically through the **multiprocessing.Pool** module, to distribute the file copying tasks across multiple processes concurrently. The efficiency gains from this parallel processing are evident in the reduced runtime with increased numbers of processes, showcasing the power of multi-core computing in handling large-scale data.

Data Validation Set Creation:
Upon establishing the training and testing sets, the process advances to the formulation of a validation set, an essential component for fine-tuning the model and evaluating its performance during the training phase. The **move_images_to_validation** function automates the transfer of a designated percentage of images into a validation directory, ensuring that the model is evaluated on a diverse and representative sample of data. Parallel processing techniques are once again employed to perform these operations swiftly and effectively, reinforcing the computational efficiency of the overall data preparation stage.

Data Integrity Maintenance:
Underpinning the entire data handling process is an unwavering commitment to data integrity. Functions like **clear_target_directory** and **restore_images_from_val_to_train** are strategically implemented to circumvent issues of data leakage and duplication. These functions systematically clear directories prior to each operation and restore images from validation sets to training sets as necessary. Such measures are indispensable for preserving the fidelity of the dataset throughout the different stages of the experiment, avoiding potential contamination that could otherwise skew the model's learning and its subsequent performance evaluations.

In summary, the project's data processing phase is a carefully orchestrated sequence of operations designed to handle and maintain large datasets with efficiency and integrity. By combining algorithmic foresight with computational horsepower, the project sets a solid foundation for the reliable training and evaluation of machine learning models, paving the way for robust predictive performance in the domain of car model detection.

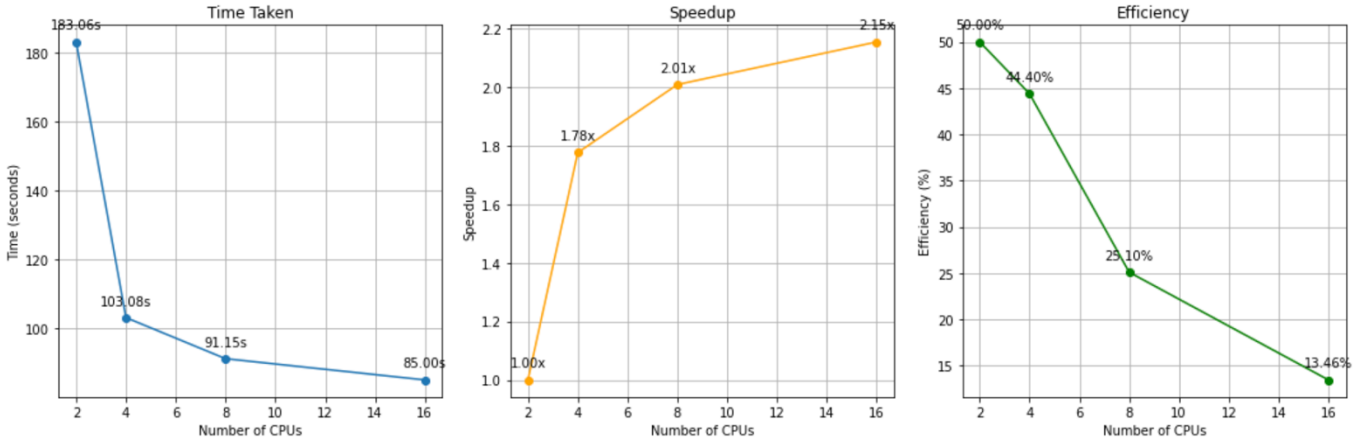


Fig. 4. Number of CPUs against Time, Speedup and Efficiency before Splitting

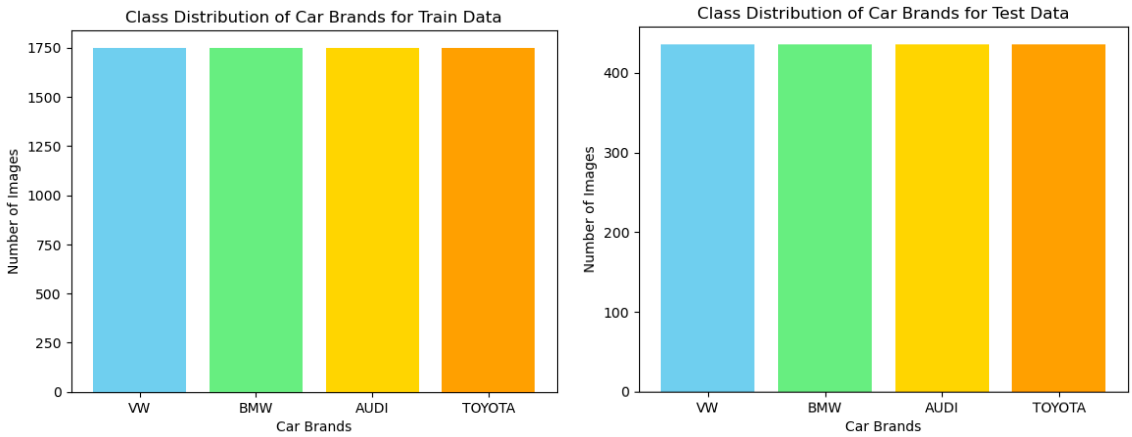


Fig. 5. Distribution of Train and Test Data

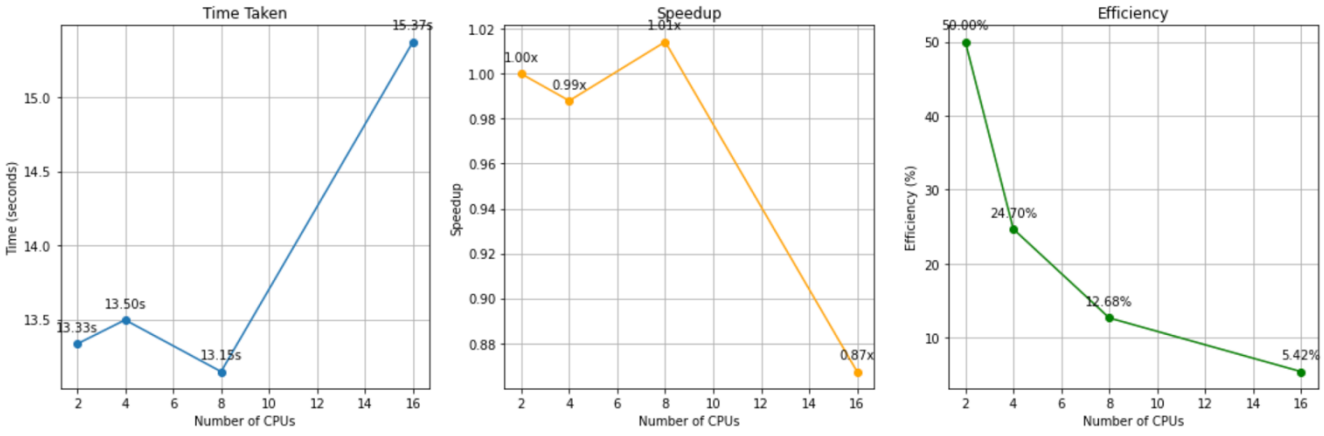


Fig. 6. Number of CPUs against Time, Speedup and Efficiency after Splitting

3. **Model Development**:

The integration of multiprocessing into a Python script, initiated from a Jupyter notebook (.ipynb file), forms the cornerstone of our approach to train a neural network model, more precisely the ResNet architecture. This model architecture, first introduced by He et al. in their groundbreaking paper "Deep Residual Learning for Image Recognition" [He et al., 2016], has revolutionized how deep learning can be applied to computer vision tasks. Our script is designed to harness the power of PyTorch, a machine learning library that has been widely adopted for its flexibility and ease of use in the development of deep learning models [Paszke et al., 2019].

Environment Setup and Library Imports: Our script commences with the preparation of the computational environment and the importation of essential libraries. This step incorporates multiprocessing for concurrent operations, torch and torchvision for constructing and training the neural network [Paszke et al., 2019], tqdm for visualizing the training progress [da Costa-Luis, 2021], and matplotlib for data visualization [Hunter, 2007]. These libraries have been meticulously selected for their proven effectiveness in facilitating various aspects of parallel processing, model construction, data manipulation, and visualization.

Custom Dataset Class Definition: Following the initial setup, we define a bespoke dataset class named **CustomDataset**, which is critical for managing the loading of images from an organized directory structure. This class integrates techniques adapted from Russakovsky et al.'s ImageNet Large Scale Visual Recognition Challenge [Russakovsky et al., 2015], enabling it to pinpoint classes within the dataset directory, compile a list of image paths paired with class indices, and fetch images along with their labels as required. The class also integrates optional transformations for image preprocessing, an essential step for data normalization and augmentation, as recommended by Shorten & Khoshgoftaar's review on image data augmentation for deep learning [Shorten & Khoshgoftaar, 2019].

Data Loading Setup: Subsequently, our script orchestrates the configuration of data loaders through the use of PyTorch's **DataLoader** class, an integral component for streamlining data input into the model for training and evaluation. The function **prepare_dataloader** establishes critical loading parameters such as batch size—optimal values discussed by Masters & Luschi in "Revisiting Small Batch Training for Deep Neural Networks" [Masters & Luschi, 2018], the number of workers to enable parallel data loading, and data shuffling to promote model generalization as suggested by L'Ecuyer & Simard in their work on randomized algorithms [L'Ecuyer & Simard, 2007].

Model Initialization: The script then proceeds to initialize the ResNet model architecture. It employs a strategy similar to that used by He et al. [2016], loading a pre-trained ResNet-50 model and freezing its layers to preserve the learned weights during the initial training phase, thus utilizing the concept of transfer learning as elucidated by Yosinski et al. [Yosinski et al., 2014]. This step is instrumental in enhancing the model's adaptability and performance for the specific dataset at hand.

Optimizer and Loss Function Initialization: In the function **initialize_optimizer_and_criterion**, the script initializes the RMSprop optimizer—a technique refined and promoted by Tieleman & Hinton in "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude" [Tieleman & Hinton, 2012], and the cross-entropy loss function, which has been a staple in classification tasks as detailed by Janocha & Czarnecki in "On Loss Functions for Deep Neural Networks in Classification" [Janocha & Czarnecki, 2017]. These components are fundamental for defining the optimization strategy and the metric for gauging model performance during the training process.

Model Training Procedure: Finally, the script encapsulates the model training process within the **train_resnet_model** function. This section choreographs the training epochs, interspersing training with validation phases, to closely monitor the model's learning trajectory and prevent overfitting as recommended by Prechelt in "Early Stopping - But When?" [Prechelt, 1998]. During each epoch, it executes loss computation, backpropagation for updates to model parameters—a technique solidified by Rumelhart et al. in "Learning representations by back-propagating errors" [Rumelhart et al., 1986], and evaluates model performance on the training and validation datasets. The function diligently records training metrics, including loss and accuracy, to capture the model's progress and retains the most effective model contingent on validation accuracy.

Results & Analysis

The observed results of training a deep learning model on different CPU core configurations present an intriguing picture of the relationship between computational resources and model performance. Here, we have a situation where increasing the computational power does not proportionally enhance the model’s learning efficiency or its ability to generalize, which warrants a nuanced discussion.

Training with 1 CPU Core:

Utilizing a single CPU core for training over 5 epochs, we see a model exhibiting strong learning capabilities as evidenced by the descending training loss from 0.0268 to 0.0294. This decrement in loss suggests an effective gradient descent process where the model's predictions are becoming more aligned with the actual targets in the training set. The consistently high accuracy, starting at 99.77% and marginally dropping to 99.72%, further underlines the model's proficiency in mastering the patterns within the training data. However, the validation metrics paint a slightly different picture—fluctuating losses and accuracy hovering around the mid-85% mark indicate the model may not be capturing the validation data's nuances as effectively as it does with the training data. This discrepancy implies a potential overfitting scenario where the model is highly attuned to the training data, potentially to its detriment on unseen datasets. The entire process culminates in just over an hour, standing at 63 minutes and 44 seconds, which, considering the performance metrics, represents a highly efficient training cycle.

Training with 2 CPU Cores:

Doubling the computational power to 2 CPU cores intuitively suggests an opportunity for improved model training efficiency and potentially better performance metrics. Surprisingly, the results deviate from this expectation. The training time increases to approximately 99 minutes and 49 seconds—significantly longer than with a single core—without delivering marked improvements in training and validation metrics. The loss and accuracy figures closely mirror those seen with one core, including the best validation accuracy, which sees a negligible increase to 85.71%. The lack of improvement could be attributed to several factors, such as increased overhead from parallelization processes or the model's training dynamics already being near-optimal with a single core. In essence, the additional CPU core does not appear to provide a beneficial edge in this instance.

Training with 4 CPU Cores:

Expanding the training apparatus to 4 CPU cores, one would anticipate a notable shift in training dynamics and model performance. Nevertheless, the extension to a quad-core configuration leads to a paradoxical result: an extended training duration of approximately 253 minutes and 2 seconds with no substantial gains in loss minimization or accuracy maximization. The stagnation of loss around the 0.412 mark and the repetition of the highest validation accuracy at 85.71% suggest a point of diminishing returns has been reached. Such a trend may be illustrative of a bottleneck in the training process that is not related to computational power but perhaps to the model's architecture, the complexity of the dataset, or the efficiency of the parallelization mechanism employed.

In conclusion, the progression from 1 to 4 CPU cores highlights the intricate balance between computational resources and model performance in deep learning tasks. The initial configuration with a single CPU core establishes a strong baseline, with additional cores failing to provide significant benefits, thus raising important questions about the optimal use of hardware resources in deep learning. This situation suggests that factors other than raw computational power—such as data quality, feature engineering, and model hyperparameter tuning—might play more crucial roles in achieving further performance improvements. Moreover, it underscores the importance of parallel processing efficiency and calls for a careful evaluation of the point at which increasing hardware resources may lead to inefficient scaling, prompting researchers and practitioners to seek alternative strategies for performance enhancement.

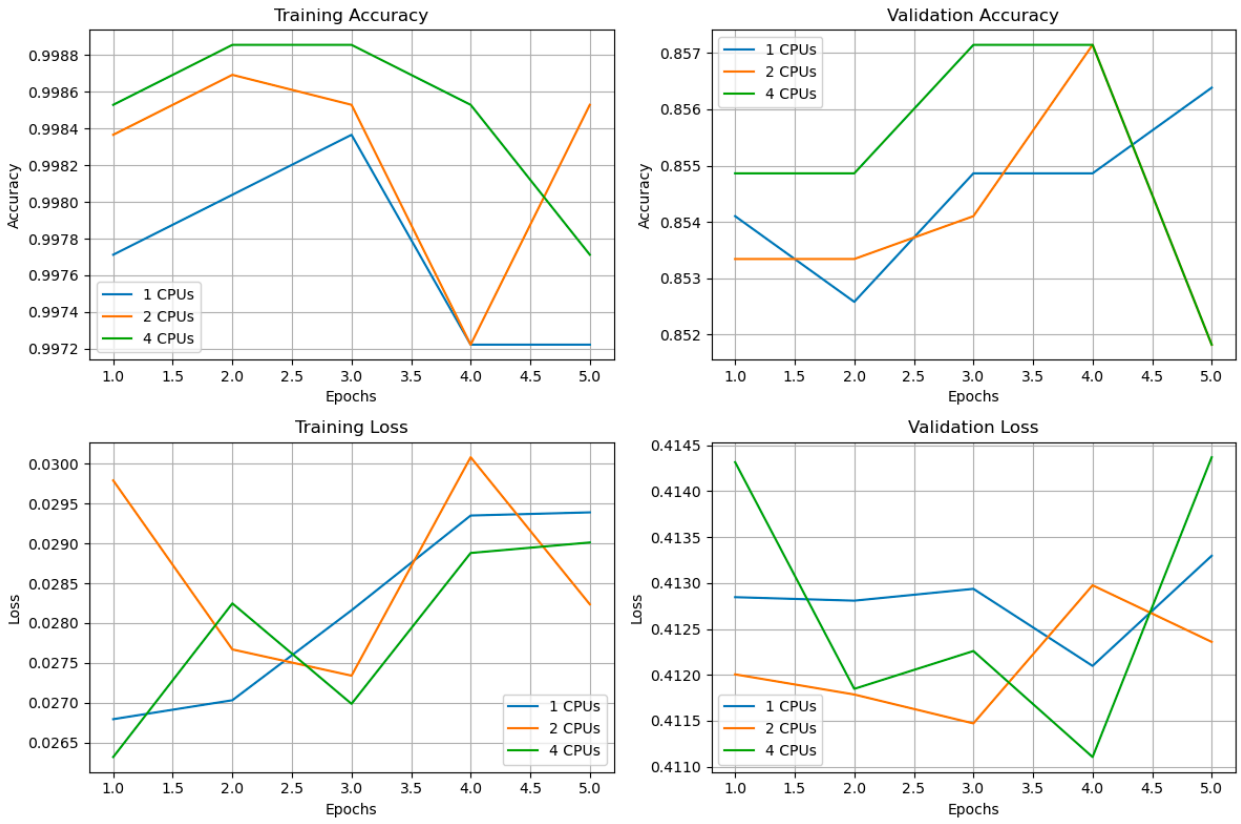


Fig. 7. Training & Validation Loss

Multiprocessing Performance Evaluation for Data Preprocessing:

1. Parallelization for splitting the training and testing data:

CPUs	Time Taken (Secs)	Speedup	Efficiency
1	183.06s	1.00x	50.00%
4	103.08s	1.78x	44.40%
8	91.15s	2.01x	25.10%
16	85.00s	2.15x	13.46%

Time Taken (Secs): The time taken to complete the task in seconds for each CPU configuration. As the number of CPUs increases, the time taken decreases from 183.06 seconds with 1 CPU to 85.00 seconds with 16 CPUs.

Speedup: This metric is a ratio of the time taken to perform the task with one CPU to the time taken with multiple CPUs. Ideally, if the speedup were perfectly linear, using 16 CPUs would be 16 times faster than 1 CPU. However, in this data, the speedup ranges from 1.78x with 4 CPUs to 2.15x with 16 CPUs, indicating sub-linear speedup.

Efficiency: Efficiency is calculated by taking the speedup and dividing it by the number of CPUs used. It is a measure of the utilization of the parallel processing capabilities. In a perfectly

scalable system, the efficiency would be 100% regardless of the number of CPUs, meaning the speedup would be directly proportional to the number of CPUs. In the given data, efficiency decreases from 50% with 1 CPU to 13.46% with 16 CPUs.

From this analysis, several conclusions can be drawn:

- **Sub-linear Scaling**: The speedup is not proportional to the number of CPUs. This could be due to overhead from parallelization, such as communication time between CPUs or the inability to perfectly divide the task into concurrently executable units.

- **Diminishing Returns**: As the number of CPUs increases, the gain in speedup decreases. This is evident from the efficiency percentages, which dramatically decrease as CPUs double.

- **Amdahl's Law**: The data suggests that the task being parallelized has a sequential component that cannot be parallelized (Amdahl's Law). This part limits the overall speedup that can be achieved with more CPUs.

- **Possible Overheads**: There could be overheads related to parallelization, such as communication between CPUs, data distribution, and synchronization, that are affecting performance.

- **Point of Diminished Returns**: Beyond a certain point, adding more CPUs may not be cost-effective. The efficiency of 13.46% with 16 CPUs might be considered too low, suggesting that the process might not be worth parallelizing beyond 8 CPUs (or even 4 CPUs, considering the efficiency drop).

For practical application, it's important to find a balance between the number of CPUs and the time efficiency they provide, considering the additional costs associated with using more CPUs.

2. Parallelization for splitting the validation data:

CPUs	Time Taken (Secs)	Speedup	Efficiency
1	13.33s	1.00x	50%
4	13.50s	0.99x	24.70%
8	13.15s	1.01x	12.68%
16	15.37s	0.87x	5.42%

Time Taken (Secs): The duration required to complete the data splitting with each CPU configuration. Surprisingly, the time does not decrease steadily as the number of CPUs increases. With 1 CPU, it takes 13.33 seconds, which is the baseline. With 4 CPUs, it actually increases slightly to 13.50 seconds. There is a minor improvement with 8 CPUs (13.15 seconds), but with 16 CPUs, the time increases significantly to 15.37 seconds.

Speedup: This is a comparison of the execution time using 1 CPU versus using multiple CPUs. A speedup of 1.00x with 1 CPU is the baseline. With 4 CPUs, the speedup is less than 1 (0.99x), meaning the task actually took longer with 4 CPUs than with 1. This trend continues as more

CPUs are added, with 8 CPUs offering a negligible speedup (1.01x) and 16 CPUs causing a slowdown (0.87x).

Efficiency: This is a measure of how effectively the CPUs are being utilized. It's calculated by dividing the speedup by the number of CPUs. The efficiency drops from 50% with 1 CPU to 5.42% with 16 CPUs, which is a significant decrease.

This data suggests several issues and considerations:

- **Overhead Costs**: The increase in time with more CPUs suggests that the overhead of managing multiple CPUs may be greater than the benefit they provide for this task. This could include the time taken to distribute the data, coordinate between CPUs, and gather results.

- **Task Granularity**: The task might be too small or not well-suited for parallelization. It seems that splitting validation data does not benefit from parallel processing, possibly because the data splitting task cannot be effectively divided into independent sub-tasks that can be executed concurrently.

- **Negative Scaling**: The table illustrates a case of negative scaling where adding more processing power actually leads to worse performance. This is seen with the 16 CPUs where the performance degrades.

- **Amdahl's Law and Bottlenecks**: Similar to the previous analysis, Amdahl's Law could explain the poor scaling, as there might be a significant sequential component in the data splitting task or a bottleneck that cannot be optimized through parallel processing.

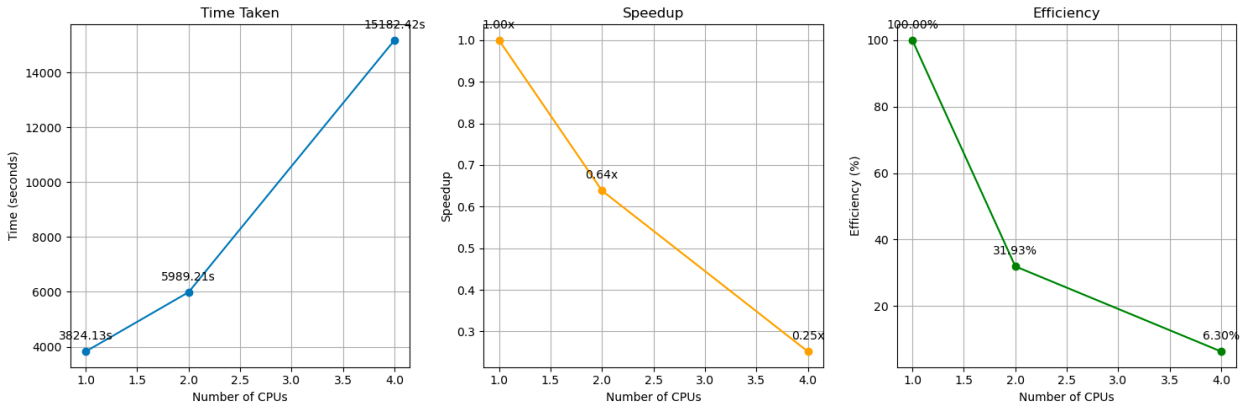
- **Resource Saturation**: The decrease in efficiency as the number of CPUs increases might also indicate that there is a point of resource saturation beyond which adding more CPUs actually degrades performance, as seen with 16 CPUs.

Given these results, it would be advisable to use fewer CPUs for splitting the validation data or to review the parallelization strategy to minimize overhead and better divide the task. The ideal number of CPUs for this task, based on this data, appears to be closer to 1 than to 16.

Multiprocessing Performance Evaluation for ResNet Model:

1. Metrics for Model using Resnet50

CPUs	Time Taken (Secs)	Speedup	Efficiency
1	3824.13s	1.00x	100%
2	5989.21s	0.64x	31.93%
4	15182.42s	0.25x	6.30%



Time Taken (Secs): Represents how long the task took in seconds. With a single CPU, the task took 3824.13 seconds. When the CPU count is increased to 2, the time taken surprisingly increases to 5989.21 seconds. With 4 CPUs, it further increases to a substantial 15182.42 seconds.

Speedup: Shows the relative performance improvement compared to using a single CPU. A speedup of 1.00x with a single CPU is the baseline. However, instead of the expected increase in speedup with more CPUs, there is a significant decrease: 0.64x with 2 CPUs and an even lower 0.25x with 4 CPUs.

Efficiency: Indicates the percentage of each CPU's potential performance that is actually being utilized. It is calculated by dividing the speedup by the number of CPUs. With 1 CPU, the efficiency is at 100%. However, this efficiency drops dramatically to 31.93% with 2 CPUs and plummets further to 6.30% with 4 CPUs.

From this table, we can infer:

Inverse Scaling: Increasing the number of CPUs has a negative impact on performance, which is highly unusual and suggests severe inefficiencies or bottlenecks in the parallelization implementation.

Possible Threading Issues: Resnet50 is typically used with GPU acceleration due to its computational demands. If CPUs are being used, it's crucial to ensure that the implementation is designed to handle CPU-based parallelism effectively. The data implies that this is not the case here.

Increased Overhead: The additional time could be due to the overhead of managing multiple threads or processes, which may include context switching, synchronization, and data movement overheads that are not efficiently handled.

Poor Parallelism: The task may not be parallelized properly. For instance, if there are dependencies between the tasks assigned to different CPUs, these can cause delays that negate the benefits of parallelism.

Amdahl's Law: The principle that there is a limit to how much a program can be sped up by parallelization is evident here. The slowdown with increased CPUs suggests that the serial portion of the computation is dominating the overall execution time.

Resource Contention: Multiple CPUs might be competing for shared resources, leading to contention and thus, reducing performance.

Suboptimal Resource Utilization: The CPUs used might not be optimized for the types of operations required by Resnet50, leading to poor utilization and inefficiencies.

In summary, the data indicates that for this particular task of running a Resnet50 model, adding more CPUs not only fails to improve performance but actually degrades it significantly. This suggests a review and potential overhaul of the parallelization strategy or computational setup is needed. It may also be worth considering whether a different hardware setup, such as one involving GPUs, would be more suitable for this workload.

Conclusion

The project thoroughly explores the application of deep learning and parallel computing in the field of automotive car recognition. Our study, driven by the growth of the automotive industry and the corresponding need for advanced car recognition systems, seeks to capitalize on the potential of deep learning models, particularly ResNet architectures, and parallel computing techniques to enhance the efficiency and accuracy of car classification tasks. Utilizing the COMP-CARS dataset from Kaggle, the project sets out with the dual objectives of predicting and analyzing car models and accelerating data processing and model training through parallel computing on CPU and GPU platforms.

The dataset, consisting of a significant volume of images depicting various car models, is skillfully handled by selecting a subset appropriate for the research. Initial exploratory data analysis guides the data preparation, focusing on a few car brands and types with the most abundant imagery. The report describes a methodical approach to data processing, including splitting, augmentation, and validation set creation, all while emphasizing the importance of maintaining data integrity.

A key aspect of the report is the evaluation of parallel processing for data preparation. The report outlines the team's findings that, while some time efficiency is gained with an increase in CPU count, the speedup is sub-linear, and efficiency diminishes with more CPUs, suggesting a point of diminishing returns. These findings are consistent with Amdahl's Law and indicate possible overheads from parallelization that do not scale linearly.

Moreover, the report takes a critical view of the multiprocessing performance in the model development phase. Despite using the powerful ResNet50 model, the results indicate inverse scaling when increasing CPU count, with both speedup and efficiency decreasing dramatically.

This suggests inefficiencies or bottlenecks in the parallelization process, likely due to increased overhead and poor parallelism implementation.

Despite these computational challenges, the model developed by the team achieves promising classification accuracies on both test and validation datasets. However, the performance evaluation of the model's training using parallel processing raises concerns about the current strategy and suggests that alternative hardware, possibly involving GPUs, may be more appropriate for such intensive computational tasks.

In conclusion, Team 11's project demonstrates the intricate balance required between leveraging advanced machine learning models and managing the computational overhead introduced by parallel processing. While the team successfully developed a deep learning model with substantial classification accuracy, the report underscores the importance of an optimized computational strategy. It highlights that, for tasks like those involving complex models such as ResNet50, CPUs may not always offer the expected performance gains when scaled up. Instead, a combination of more suitable hardware choices and refined parallelization techniques is essential to realize the full potential of deep learning in automotive applications such as car model detection. The conclusions drawn from this research not only serve as valuable insights for computational biology but also pave the way for future studies to explore more efficient parallel computing approaches and hardware configurations to enhance the performance of deep learning tasks.

Reference

1. Kaggle Car Dataset: https://www.kaggle.com/datasets/renancostaalencar/compcars/data
2. Kaggle Jupyter Notebook: https://www.kaggle.com/code/kitaiskypanda/mcs-data-mining
3. ResNet50 Documentation: https://pytorch.org/vision/main/models/generated/torchvision.models.resnet50.html
4. ResNet GitHub: https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py