Building a CIFAR-10 Classification Dataset: Preparation and Baseline Analysis

Aimilia Palaska 10453 - aimiliapm@ece.auth.gr
Academic Supervisor: Anastasios Tefas

November 24, 2024

1 Introduction

This report is created as part of the Neural Networks & Deep Learning course in Electrical and Computer Engineering department of Aristotle University of Thessaloniki. The developed code in python, along with detailed comments, can be found in the following GitHub Repository [1].

Neural Networks are increasingly being employed for many real-world problems, including image classification. For this course, we investigate the implementation and evaluation of neural networks for image classification on the CIFAR-10 dataset. By combining custom implementations from scratch with ready-made architectures, the study aims to understand the inner workings of neural networks while leveraging advanced, optimized models for benchmarking.

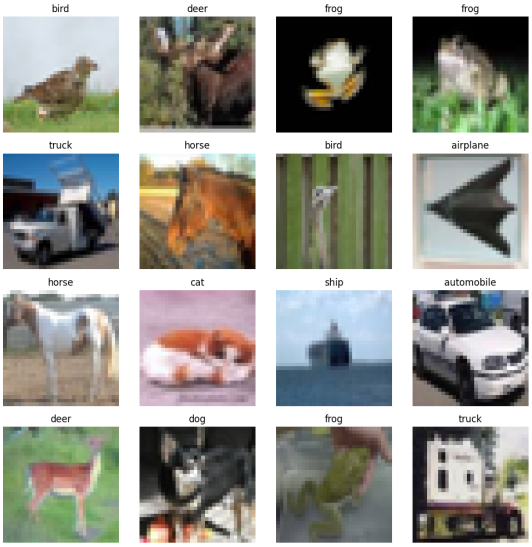

Figure 1: Random samples of the CIFAR-10 dataset and their respective labels.

2 Dataset Overview

2.1 Data Format and Structure

The CIFAR-10 dataset [2], originally introduced by Alex Krizhevsky, is a widely used benchmark in machine learning and computer vision, consisting of 60,000 color images categorized into 10 distinct classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Each image is 32x32 pixels in size and contains a single object from one of these classes, making it a relatively low-resolution dataset ideal for testing image classification models and algorithms. In Figure 1, a grid of 16 random samples, along with their respective label, is presented for the purpose of visualization. Split into 50,000 training images and 10,000 test images, CIFAR-10 offers a balanced and challenging dataset that facilitates evaluation and comparison of model performance across a broad range of supervised learning techniques.

The CIFAR-10 dataset has a uniform distribution of samples across its 10 classes, each class containing 6,000 images, which eliminates the need for downsampling and ensures that there is no inherent bias towards any particular class during training. This can be seen in Figure 2, where the ten classes are plotted with the corresponding number of instanced in the labels array.

2.2 Preprocessing

Since CIFAR-10 is a clean dataset without duplicates or missing values, it requires no preprocessing for data integrity.

However, as the images have pixel values ranging from 0 to 255, normalization is essential for training in order to standardize these values to facilitate more stable and effective learning. Two common normalization methods are applied: z-score normalization and min-max normalization. In z-score normalization, each pixel value x is transformed using $x' = \frac{x-\mu}{\sigma}$, where μ is the mean pixel value, and σ is the standard deviation of the pixel distribution. This method centers the values around 0 with a standard deviation of 1. Alternatively, in min-max normalization, the pixel value x is scaled as $x' = \frac{x-min}{max-min}$, setting the range between 0 and 1, where min and max are the minimum and maximum pixel values (0 and 255, respectively).

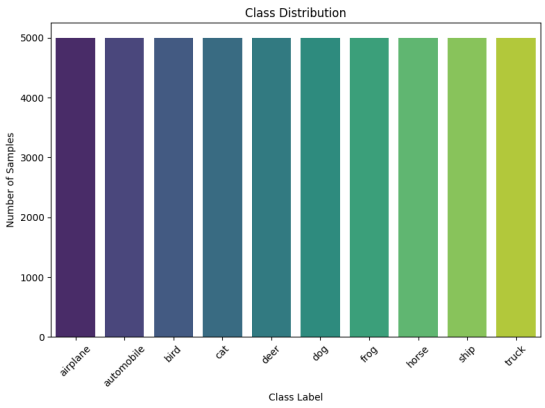

Figure 2: Class distribution of the CIFAR-10 dataset.

3 Baseline Models

3.1 Algorithm Overview

Baseline models provide a reference point, allowing for a meaningful comparison to assess the neural network's accuracy and improvement over simpler methods. For this purpose, k-nearest neighbors (k-NN) and nearest centroid classifiers were implemented, as they are straightforward, widely understood algorithms for classification with minimal development complexity.

K-Nearest Neighbor (KNN) The k-nearest neighbors (KNN) algorithm is a simple, non-parametric method used for classification tasks, relying on the concept of feature similarity. For a given input sample, KNN identifies the k nearest samples in the dataset and assigns the input to the class most common among those neighbors. By varying k, the algorithm can be fine-tuned for different levels of sensitivity to local patterns in the data. Although computationally intensive, especially for large datasets, KNN offers a straightforward way to predict labels based purely on proximity to labeled samples, making it a useful baseline.

Nearest Centroid (NC) The nearest centroid classifier, on the other hand, simplifies the classification process by representing each class with a single prototype—its centroid. In this approach, the centroid of each class is calculated by averaging the feature values of all samples within that class. To classify a new sample, the algorithm assigns it to the class with the closest centroid. This method is computationally efficient since it reduces each class to a single point, making it faster than KNN while still capturing basic patterns within the data. However, it may struggle with complex class boundaries, as it assumes each class can be effectively represented by a central point.

3.2 Distance Calculation

For the classification of the CIFAR-10 dataset, four methods of calculating the distance between the images were implemented, for experimental purposes.

- **Pair-wise distance**: This approach calculates the absolute difference between corresponding pixels of two images, summing up these differences to get an overall measure of similarity. Pair-wise distance captures the pixel-level discrepancies, making it sensitive to small changes but computationally light.
- **Mean squared error (MSE)**: MSE calculates the average of the squared differences between pixel values of two images, represented as $MSE = \frac{1}{n}\sum_{i=1}^{n}(x_i - y_i)^2$ where x and y are the pixel values of the two images. This method penalizes larger differences more heavily, making it useful for detecting significant deviations, though it can be sensitive to minor changes.
- **Cosine distance**: This method measures the angle between two image vectors in high-dimensional space rather than their absolute pixel values. Calculated as $1 - \frac{x \cdot y}{\|x\|\|y\|}$, cosine distance captures differences in orientation rather than intensity. This approach is often effective for comparing patterns in images while disregarding brightness variations.
- **Structural similarity (SSIM)**: Structural similarity [3] aims to measure perceptual differences by comparing local patterns of pixel intensities, focusing on luminance, contrast, and structure. SSIM values range from -1 to 1, with values closer to 1 indicating greater similarity. This method aligns more closely with human visual perception, making it valuable for detecting nuanced changes in image structure.

4 Evaluation Metrics

To evaluate model performance in this classification task, we employ four core metrics: accuracy, precision, recall, and F-score. These metrics offer distinct insights: accuracy, as the primary metric, measures the overall correctness of predictions, calculated as the ratio of correctly predicted instances to the total number of instances. Precision quantifies the model's ability to avoid false positives, while recall assesses its ability to capture true positives. The F-score, as a division of precision and recall, gives a balanced measure that is especially useful for imbalanced data scenarios.

Although test accuracy remains the most significant indicator of model performance, precision, recall, and F-score offer additional insights

into robustness, potential overfitting, and class imbalances. These metrics will help us better understand how the model performs beyond simple accuracy when evaluating Neural Networks.

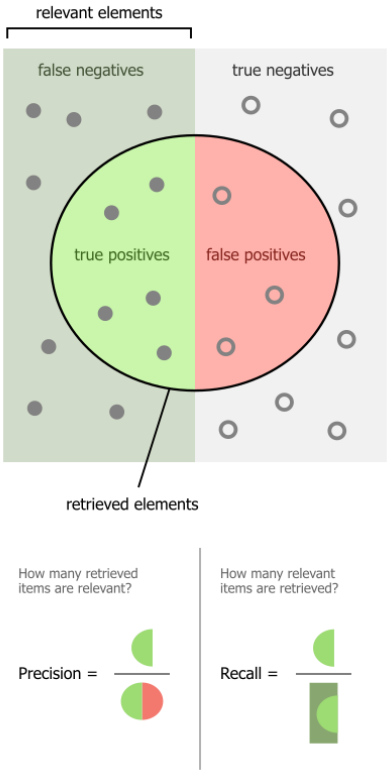

Figure 3: Precision and recall formula visualized for better understanding. Image by Wallber on Wikipedia [4].

In this baseline analysis, simpler models like k-nearest neighbor (KNN) and nearest centroid are expected to perform modestly, as they lack the complexity to capture the full structure of image data. However, these baseline metrics are essential as they establish a performance threshold. This threshold serves as a benchmark that the neural network model, with its greater complexity and adaptability, is expected to surpass, demonstrating its suitability for CIFAR-10 classification.

5 Preliminary Results with Baseline Models

This section presents the performance of the K-Nearest Neighbors (KNN) and Nearest Centroid (NC) algorithms on the CIFAR-10 dataset, using the predefined metrics. Results are analyzed for each model with a discussion on the strengths and limitations observed, setting expectations for the neural network's anticipated performance.

Comparison of Normalization Methods: To assess the impact of normalization on model performance, we tested the KNN algorithm with two common normalization techniques: Z-score and Min-Max. Results are summarized in Table 1, using pairwise distance metrics and evaluating performance at $k = 1$ and $k = 3$. Both normalization techniques performed similarly across most metrics, suggesting limited impact on the model's effectiveness. Given these similarities, Z-score normalization was selected for subsequent experiments, as it provided slightly more stable results overall.

	Z-score		Min-Max	
	$k = 1$	$k = 3$	$k = 1$	$k = 3$
Accuracy	0,3548	0,3287	0,3539	0,3303
Precision	0,4118	0,4283	0,4112	0,4304
Recall	0,3548	0,3287	0,3539	0,3303
F-score	0,3503	0,318	0,3495	0,3192

Table 1: Performance of KNN model for two normalization methods, Z-score and Min-Max. It is observed that the performance of the models differs slightly, indicating similar behavior of the KNN algorithm for these normalization methods.

Effect of Different k-values in KNN: Table 3 compares the KNN model's performance for $k = 1$ and $k = 3$, using several distance metrics. Generally, models with $k = 1$ achieved higher accuracy than those with $k = 3$, indicating that a closer, single nearest neighbor may yield better classification results in this dataset. However, despite this advantage, none of the tested configurations achieved accuracy scores above 50%, with the lowest performance observed in models using cosine similarity. This highlights a limitation in the baseline models' ability to classify image features accurately.

Comparison of KNN and Nearest Centroid (NC): A comparison of KNN and NC models in Table 2 reveals a trade-off between execution time and accuracy. While the NC algorithm demonstrated faster computation times, it generally delivered lower accuracy scores compared to KNN, except in the cosine similarity case. This indicates that NC may be a viable option when computational efficiency is prioritized, but its reduced accuracy makes it less ideal for more accurate classifications.

Distance Metric Performance Analysis: We observed that the Mean Squared Error (MSE) and Pairwise metrics produced similar results, as expected, given their calculation similarity (MSE incorporates squared differences compared to Pairwise). Among all tested metrics, the Structural distance metric emerged as the most effective for this classification problem, yielding the highest accuracy scores. The best-performing baseline model overall was the KNN model with Z-score normalization, $k = 1$, and the Structural distance calculation.

Method	Distance	Accuracy	Precision	Recall	F-score
KNN	Pair	0.3548	0.4118	0.3548	0.3503
	MSE	0.3548	0.4118	0.3548	0.3503
	Cosine	0.0992	0.1096	0.0992	0.0963
	Structural	0.4070	0.4178	0.407	0.4055
NC	Pair	0.2784	0.2882	0.2784	0.2551
	MSE	0.2784	0.2882	0.2784	0.2551
	Cosine	0.2853	0.2931	0.2853	0.2696
	Structural	0.1955	0.2190	0.1955	0.1626

Table 2: Comparison of k-nearest-neighbor ($k = 1$) and nearest centroid models for four distance calculation methods.

KNN	Distance	Accuracy	Precision	Recall	F-score
$k = 1$	Pair	0.3548	0.4118	0.3548	0.3503
	MSE	0.3548	0.4118	0.3548	0.3503
	Cosine	0.0992	0.1096	0.0992	0.0963
	Structural	0.407	0.4178	0.407	0.4055
$k = 3$	Pair	0.3287	0.4283	0.3287	0.318
	MSE	0.3287	0.4283	0.3287	0.318
	Cosine	0.0900	0.112	0.0900	0.0812
	Structural	0.3740	0.4031	0.3740	0.3720

Table 3: Comparison of k-nearest-neighbor models for $k = 1$ and $k = 3$ neighbors.

6 Experimental Setup

6.1 Implementation from Scratch

For the training pipeline, only the NumPy library [5] was used, ensuring that no pre-built neural network methods influenced the results. During the evaluation stage, the scikit-learn library [3] was utilized for metrics calculation, as it did not interfere with the core functionality or themes of the experiments. Neural networks were implemented from scratch to gain a deeper understanding of their mechanisms through a "white-box" approach. Debugging the implementations provided critical insights into the inner workings and expected behavior of neural networks.

Portions of the code were generated using ChatGPT, which are available in the repository under the `first_attempt` module. While most of the generated code was accurate, minor issues related to data types and dimensional inconsistencies were encountered and rectified. These references were retained to illustrate ChatGPT's potential for aiding in white-box neural network implementation.

To enhance clarity and modularity, the codebase was divided into modules and classes. Core components included classes for a fully connected layer, a convolutional layer, and utility functions, which were organized within the `scratch` module. This design enabled flexible layer configurations through the main files.

6.1.1 Fully Connected Layers

Initial tests focused on fully connected architectures with 3 and 5 layers. The 3-layer model used (1024, 512, 256) neurons, while the 5-layer model had (2048, 1024, 512, 256, 128) neurons, activated by the ReLU function [6]. Both models were tested with Xavier [7] and He [8] initialization techniques and learning rates of 0.01 and 0.001.

While results from these tests were not expected to match modern frameworks like PyTorch [9], the primary objective was to observe loss reduction across epochs and identify potential overfitting trends. *Cross-validation* and *dropout* mechanisms were implemented from scratch to enhance model robustness. These experiments highlighted the challenges and nuances of building effective fully connected networks manually.

6.1.2 Convolutional Layers

Convolutional layers presented additional challenges due to the complexity of convolution operations, which required precise handling of input/output dimensions and padding. Testing these networks was also significantly more computationally intensive. For instance, an initial test with a 4-layer convolutional model using filter sizes (32, 32, 64, 64), a kernel size of 3, stride 1, and padding 1 took approximately 50021.81 seconds for a single epoch with a batch size of 10,000.

These experiments underscored the importance of GPU parallelization in modern deep

learning frameworks like TensorFlow and PyTorch, as convolution operations are well-suited for distributed computation. Due to these constraints, testing on the NumPy-based convolutional networks (CNN) was limited to smaller batch sizes. The primary focus of these tests was to validate the correctness of gradient descent and parameter updates in more complex architectures.

6.2 Ready Architectures

In addition to the custom implementations, several pre-built architectures were utilized to benchmark performance, analyze the impact of GPU acceleration, and compare the achieved metrics against more complex, well-established models. These ready architectures were trained and evaluated using both CPU and GPU, highlighting the computational efficiency of modern frameworks.

The selected architectures were ResNet-18, VGG-16, and MobileNetV2, each of which represents a distinct design philosophy and serves as a valuable point of comparison. The implementations are organized inside the `ready_models` module.

- **ResNet-18** [10], part of the Residual Network family, is known for its use of skip connections, which mitigate the vanishing gradient problem in deep networks. This architecture contains 18 layers, including both convolutional and fully connected layers, and is structured around residual blocks that allow gradients to flow more easily during backpropagation. This simplicity and robustness make ResNet-18 an excellent starting point for experiments with GPU acceleration.
- **VGG-16** [11] is a deeper network consisting of 16 layers, predominantly convolutional layers followed by max-pooling and fully connected layers. This architecture is characterized by its uniform use of 3x3 convolutional kernels and its simplicity in stacking layers, resulting in a high capacity to learn intricate features. While computationally intensive on a CPU, it demonstrates significant speed improvements and scalability on GPUs.
- **MobileNetV2** [12] is a lightweight architecture optimized for mobile and resource-constrained devices. It employs depthwise separable convolutions and inverted residual blocks to reduce computational cost while maintaining competitive accuracy. Despite its smaller size, MobileNetV2 achieves excellent performance on benchmark datasets, showcasing the benefits of modern architectural innovations for efficient computation.

These models provided valuable insights into the performance trade-offs between CPU and GPU, as well as the benefits of using optimized and bench-marked architectures for improving accuracy and training efficiency.

7 Neural Network Results

All models were tested on an Intel(R) Core(TM) i5-8400 CPU. Additionally, the pre-trained models were trained and evaluated on an NVIDIA A100 GPU (40GB DDR6) provided by the Aristotelis HPC infrastructure. The metrics of every training and evaluation experiment presented in this section are located inside the `results` folder of the repository in JSON format.

7.1 Fully Connected from Scratch

The performance of fully connected architectures implemented from scratch was analyzed with varying initialization methods, learning rates, and training configurations. A comparison of loss reduction across epochs for Xavier and He initialization is shown in Figure 4. Xavier initialization begins closer to the loss convergence point but both methods ultimately achieve similar levels of minimization, demonstrating their effectiveness in this simple setup.

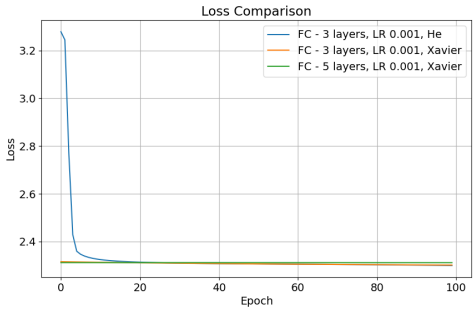

Figure 4: Loss comparison over epochs for different fully connected architectures.

Performance metrics for various configurations are summarized in Table 4. Overall, the models exhibit poor performance, with accuracy and F-score lower than baseline models, namely k-NN and nearest centroid. Notably, the network's behavior with a larger learning rate is particularly interesting, as illustrated in Figure 5. Here, abrupt fluctuations in loss values are observed due to the larger parameter update steps, resulting in significantly lower precision.

These results highlight the challenges of implementing fully connected architectures from scratch without benefiting from advanced optimization techniques commonly found in modern frameworks. Despite the modest outcomes, the experiments provide valuable insights into the effects of initialization, learning rate, and regularization on training dynamics.

Configuration	Accuracy	Precision	Recall	F-score
FC(3)	0.0908	0.1216	0.0908	0.0515
FC(3), LR 0.0100	0.1000	0.0100	0.1000	0.0182
FC(3), DR, 30 EP	0.1247	0.0984	0.1247	0.0813
FC(3), DR, 100 EP	0.1127	0.1059	0.1127	0.0596
FC(3), DR, 30 EP, 5 FOLDS	0.1049	0.0320	0.1049	0.0313
FC(5)	0.1086	0.0741	0.1086	0.0723

Table 4: Performance of different fully connected models from scratch in the test dataset. Unless specified, the learning rate is assumed to be 0.001.

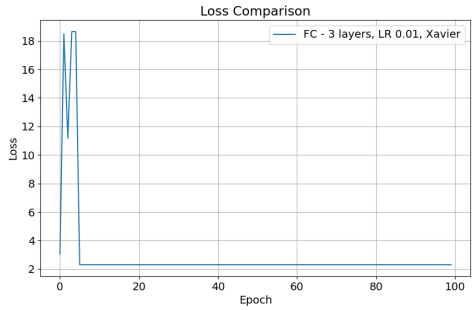

Figure 5: Loss fluctuations observed with a larger learning rate, caused by larger parameter update steps during training.

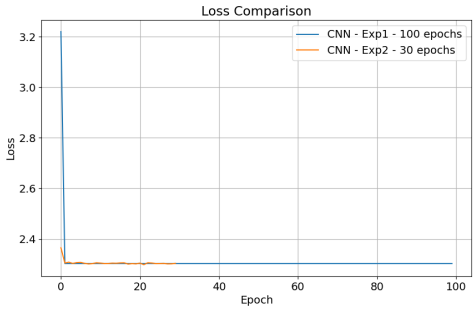

Figure 6: Loss convergence of convolutional networks from scratch. The rapid convergence suggests potential errors in back-propagation.

7.2 Convolutional from Scratch

As previously noted, the high computational cost of convolutional networks significantly limited the scope of experiments. Table 5 summarizes the average duration of an epoch for different configurations, illustrating how execution time grows exponentially with increased depth and batch size. This computational overhead underscores the necessity of GPU parallelization for training convolutional models efficiently.

Interestingly, the loss converged almost instantaneously during training, as shown in Figure 6. This behavior suggests potential issues in the implementation of backpropagation, a particularly challenging aspect of convolutional networks when developed from scratch. Despite these challenges, test metrics hovered around 10%, indicating the need for additional training and refinement to improve performance.

7.3 Ready Architectures

The results from the ready-made architectures were significantly better than those achieved with the custom implementations, as expected. These architectures are more complex, thoroughly tested, and optimized for image classification tasks.

To mitigate overfitting, a StepLR learning rate scheduler [9] was integrated into the ResNet-18 architecture. This modification slightly improved the F-score, as shown in Table 6. Additionally, these models generally required fewer epochs to converge, demonstrating the efficiency of modern frameworks and training techniques.

Among the tested architectures, VGG-16 performed the best, achieving the highest accuracy and F-score pair when trained for 20 epochs with a learning rate of 0.0001. The predictions of this model are visualized in the confusion matrix presented in Figure 7. This matrix highlights the model's ability to distinguish between the ten classes effectively. However, ResNet18 was a close second, with very similar results and one tenth of the computational cost to train it.

Overall, the ready architectures showed a 25% improvement over the best-performing baseline model, underscoring the potential of neural networks in the field of image classification. These results confirm that employing advanced architectures and training methodologies is critical for achieving state-of-the-art performance in challenging datasets like CIFAR-10.

7.4 CPU vs GPU Execution Time

The High-Performance Computing (HPC) infrastructure provided by the university enabled experiments to be conducted on a GPU, offering an opportunity to understand the immense power of parallelization in deep learning. This capability was particularly beneficial for running the ready-made models implemented in PyTorch, as these frameworks are designed to leverage GPU acceleration effectively.

As shown in Table 7, the average execution

	Epoch Duration	Filters per Layer	Kernel Size	Batch Size
Exp 1	50022 sec	(32, 32, 64, 64)	3x3	10,000
Exp 2	76 sec	(16, 16)	3x3	1,000
Exp 3	1476 sec	(32, 32)	3x3	1,000

Table 5: Average execution time per epoch for different CNN configurations.

Architecture	Accuracy	Precision	Recall	F-score
ResNet18	0.5026	0.5126	0.5026	0.5011
ResNet18 (StepLR)	0.5071	0.5026	0.5071	0.5033
VGG-16	0.5147	0.5226	0.5147	0.5124
MobileNetV2	0.4023	0.3981	0.4023	0.3951

Table 6: Performance metrics of the ready model architectures provided by PyTorch on the test dataset.

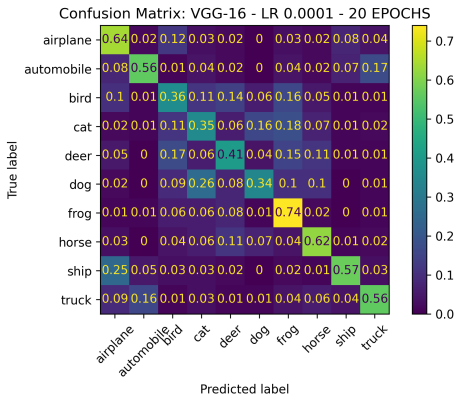

Figure 7: Confusion matrix of the best-performing network (VGG-16) on the test dataset

Architecture	CPU	GPU
ResNet18	16.22 sec	1.07 sec
VGG-16	532.6 sec	6.55 sec
MobileNetV2	241.67 sec	10.34 sec

Table 7: Computational cost of models based on the average time to complete one epoch on CPU and GPU.

time per epoch dropped to approximately 10% of the CPU time when utilizing the GPU. This dramatic reduction underscores the efficiency gains offered by parallelized computations, which are essential for training large, complex architectures.

These results illustrate the critical role of GPUs in modern deep learning workflows, particularly for training deeper architectures like VGG-16, which would otherwise be computationally prohibitive on CPUs. The experiments further highlight how parallel processing can dramatically accelerate training and make the exploration of larger models feasible within academic and research settings.

8 Conclusion

This report explored the implementation and evaluation of neural networks for image classification using the CIFAR-10 dataset. The experiments were conducted across three main avenues: custom implementations of neural networks from scratch, ready-made architectures provided by PyTorch, and an analysis of computational performance using both CPUs and GPUs for training.

The custom implementations highlighted the challenges and insights gained from developing neural networks manually. Fully connected networks demonstrated basic functionality but suffered from poor performance compared to baseline models, such as k-NN and nearest centroid. The addition of dropout and cross-validation slightly improved the results, but overall accuracy and F-score remained low. Convolutional networks, while more complex and computationally intensive, showed promising loss convergence but faced potential back-propagation issues, emphasizing the steep learning curve and computational demands of implementing such models without modern frameworks.

The ready-made architectures, including ResNet18, VGG-16, and MobileNetV2, performed significantly better, leveraging the advanced design and optimization techniques of PyTorch. Among these, VGG-16 achieved the best results, with an accuracy of 51.47% and an F-score of 51.24%, marking a 25% improvement over the best-performing baseline model. These results demonstrate the effectiveness of neural networks in tackling image classification tasks and underscore the importance of sophisticated architectures and robust training methodologies.

A key insight from this project was the transformative role of GPU acceleration. With the university's HPC infrastructure, experiments utilizing GPU parallelization achieved a tenfold reduction in execution time compared to CPU-based training. This efficiency gain was particularly pronounced for deeper networks like

VGG-16, which would otherwise be infeasible to train within reasonable time constraints.

In conclusion, this report highlights the potential of neural networks for image classification, the value of understanding their inner workings through custom implementation, and the critical importance of leveraging advanced tools and parallel computing resources.

Acknowledgments

Results presented in this work have been produced using the Aristotle University of Thessaloniki (AUTh) High Performance Computing Infrastructure and Resources. Additionally, the open-source language model ChatGPT [13] was utilized in parts of these experiments. It generated portions of the code which were then tested and analyzed, as well as enhanced the overall readability and clarity of this report.

References

[1] E. Palaska, "cifar10-nn." https://github.com/emily-palaska/cifar10-nn, 2024.

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," tech. rep., University of Toronto, 2009.

[3] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, T. Yu, and the scikit-image contributors, "scikit-image: Image processing in python." https://doi.org/10.7717/peerj.453, 2014.

[4] Wallber, "Precision and recall." https://commons.wikimedia.org/w/index.php?curid=36926283, 2021.

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, *et al.*, "Array programming with NumPy," 2020.

[6] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines." https://www.cs.toronto.edu/~hinton/absps/reluICML.pdf, 2010.

[7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 9, pp. 249–256, 2010.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

[9] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library." https://pytorch.org/, 2019.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[11] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[12] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 4510–4520, 2018.

[13] OpenAI, "Chatgpt." https://chat.openai.com/, 2024.