Comparing Machine Learning Models for CIFAR-10 Classification

Matteo Robidoux

Department of Computer Science and Software Engineering
Concordia University
Montreal, QC, Canada H3G 1M8

Abstract

This project examines the performance of various machine learning models for **image classification on the CIFAR-10 dataset**. The four models implemented include **Guassian Naive Bayes (GNB)**, **Decision Tree (DT)**, **Multi-Layer Perceptron (MLP)**, and a **Convolutional Neural Network (CNN) based on the VGG11 architecture**. The GNB, DT, and MLP models were trained using **image features extracted from a pre-trained ResNet-18 CNN model** and after feature vector **size reduction using PCA**. The CNN model was trained directly on the CIFAR-10 images. All models were trained and evaluated on subsets of the CIFAR-10 dataset, using **standard performance metrics**, including **accuracy**, **precision**, **recall**, **F1-score**, **training time**, and **confusion matrices** to compare and analyze. The study highlights how different models compare amongst each other and analyses the impact of changing **model depth**, **hidden layer size**, and **kernel size**, has on performance.

1 Introduction

Image classification is one of Artifical Intelligences most known problems, with datasets such as the **CIFAR-10** known for being the benchmark for image classification [1]. Neural networks have become the primary solution for dealing with image recognition, but other machine learning models can still be used to provide viable results. This project examines how both simple and advanced machine learning models perform on the CIFAR-10 dataset, and how they differ.

The goal of this project is to compare the four different type of models, including **Guassian Naive Bayes (GNB)**, **Decision Tree (DT)**, **Multi-Layer Perceptron (MLP)**, and a **Convolutional Neural Network (CNN) based on the VGG11 architecture**, and see how they perform on the CIFAR-10 dataset. GNB and DTs are two of the most basic machine learning models, while MLPs and CNNs are more advanced models that consist of a multi-layered architecture. Comparing them with eachother can show how model complexity affects performance on image classification.

Part of the goal of this project was not only to compare the models but also to see how different **hyperparameters** affect the performance of each model. This includes changing the depth of the DT, increasing or decreasing the number of layers and layer size of the MLP, and increasing or decreasing the number of layers and kernel size of the CNN. By changing these hyperparameters, we can see how they affect the performance of each model in regard to learning, overfitting, training time and general performance.

This project aims to not only evaluate the performance of each model but also understand why they perform the way they do. By analyzing the results of each model with different hyperparameters, we can get a better understanding of the positive and negatve aspects of each model and how it affects the results on a popular image classification dataset such as CIFAR-10.

2 Dataset Overview

The CIFAR-10 dataset is a commonly used benchmark dataset for image classification. It contains **60,000 color images**, **50,000 training images** and **10,000 test images**, of **size 32 x 32**, each belonging to one of **10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck** [2]. Due to its small size and simplicity, CIFAR-10 is often used for testing and evaluating machine learning models for image classification.

For this project, the CIFAR-10 datatset was sliced down to only **500 training images** and **100 test images** per class, resulting in a total of **6,000 images (5,000 training images and 1,000 test images)**, ultimately to reduce training time for models while still providing enough data for the models to learn from.

Since Gaussian Naive Bayes (GNB), Decision Trees (DT), and Multi-Layered Perceptrons (MLP) are not made to deal with high-dimensional RGB images due to the **curse of dimensionality** where model performance drops as dimension space grows [3], additional preprocessing is necessary to convert them into **low-dimensional vectors** through **feature extraction**. All CIFAR-10 images used for these models were first **normalized** and **resized to 224x224** to match the expected format for **ResNet-18**.

The images were then passed through a **pre-trained ResNet-18 CNN** to extract **512-dimensional feature vectors**. To further reduce the dimensionality of these feature vectors, **PCA** is applied reducing the 512-dimensional vectors to **50 dimensions**. With these reduced CIFAR-10 feature vectors, the GNB, DT, and MLP models can now be trained and evaluated effectively.

On the other hand, the CNN model (VGG11 architecture) was trained directly on the original CIFAR-10 images without any feature extraction or dimensionality reduction. This is becasue CNNs are designed to handle high-dimensional images directly.

These steps ensures that all models are trained under ideal conditions for their respective architectures, allowing for a fair comparison of their performance on the CIFAR-10 dataset.

3 Model Implementations and Training

3.1 Gaussian Naive Bayes

3.1.1 Custom Gaussian Naive Bayes Implementation

The custom Gaussian Naive Bayes (GNB) model was implemented in Python and NumPy, and was trained on a 50-dimensional feature vector extracted from the CIFAR-10 images. The model first calculates the mean, variance, and prior probability for each class during the training phase. During prediction, the model then calculates the log posterior probability for each class using the **Gaussian probability density function** shown in Equation 1.

$$log(P(Class|x)) = log(P(Class)) + \sum_{i=1}^{n} log(P(x_i|Class)) \tag{1}$$

where $P(Class)$ is the prior probability of class $Class$, and $P(x_i|Class)$ is the likelihood of feature x_i given class $Class$. The logarithm is added to avoid any potential underflow when dealing with small numbers. Finally, the class with the highest log posterior probability is then chosen as the predicted class. [4]

Unlike other models, the GNB model does not have hyperparameters to tune. It simply uses the training data to calculate the parameters needed for evaluation. [4]

3.1.2 Sklearn Gaussian Naive Bayes Implementation

The Sklearn Gaussian Naive Bayes (GNB) model was also implemented in order to compare its performance with the custom implementation. The model was trained on the same 50 dimensional feature vector extracted from the CIFAR-10 images. Unlike the custom implementation, the Sklearn GNB model handles all the calculations and optimizations in the background. This allows for a more easy to use GNB model but with less control over the details. Like the custom implementation, the Sklearn GNB model does not have hyperparameters to tune.

3.2 Decision Tree

3.2.1 Custom Decision Tree Implementation

The custom Decision Tree (DT) model was implemented in Python and NumPy, and was trained on a 50-dimensional feature vector extracted from the CIFAR-10 images. The custom model uses **Gini Impurity** as the splitting criterion which is calculated using the formula shown in Equation 2.

$$Gini = 1 - \sum_{i=1}^{n} (p_i)^2 \tag{2}$$

where p_i is the probability of an image being classified to a particular class. During training, the model recursively splits the data into left and right subtrees based on the threshold. For each split, the **weighted Gini impurity** is calculated, and the split that results in the lowest weighted Gini impurity is chosen [5]. The tree continues to grow until a stopping criterion is met, such as:

- Maximum depth of the tree is reached.

- No more splits can be made.
- All samples in a node belong to the same class.

To analyze how the **maximum depth** of the tree affects the models performance, different variations of the model were trained and evaluated. Varying the depth of the tree allows us to compare how a shallow tree performs against a deeper tree, and how it affects the overall performance of the model.

3.2.2 Sklearn Decision Tree Implementation

The Sklearn Decision Tree (DT) model was implemented in order to compare its performance with the custom implementation. The model was trained on the same 50-dimensional feature vector extracted from the CIFAR-10 images. Like the custom DT, the Sklearn DT model by defualt uses **Gini Impurity** as the splitting criterion.

Similar to the custom DT, the Sklearn DT model was trained with different maximum depths to analyze how the depth of the tree affects the models performance and to compare it with the custom implementation.

3.3 Multi-Layer Perceptron

3.3.1 Custom Multi-Layer Perceptron Implementation

The custom Multi-Layer Perceptron (MLP) model was implemented in PyTorch and was trained on a 50-dimensional feature vector extracted from the CIFAR-10 images. The base architecture of the model consists of the layers shown in Table 3.3.1.

Layer	**Description**
Input Layer	50 neurons
Hidden Layer 1	Linear(50, 512) + ReLU
Hidden Layer 2	Linear(512, 512) + BatchNorm(512) + ReLU
Output Layer	Linear(512, 10)

Table 1: Base MLP Architecture

The model is trained using the settings shown in Table 3.3.1.

Parameter	**Value**
Loss Function	Cross-Entropy Loss
Optimizer	Stochastic Gradient Descent (SGD)
Momentum	0.9
Learning Rate	0.001
Batch Size	32
Epochs	100

Table 2: MLP Training Settings

During the training process, the model does a **forward pass** to calculate the output, calculates the **loss**, performs a **backward pass** to calculate gradients, and **updates the weights** using the optimizer [6]. After training, basic performance metrics for each epoch are recorded for analysis.

To analyze how the architecture of the MLP affects the models performance, different variations of the model were trained. This includes **adding/removing hidden layers**, and changing **the hidden layers sizes**. By doing so, we can compare how a shallow MLP performs against a deeper MLP, and how widder hidden layers perform against narrower hidden layers.

3.3.2 Sklearn Multi-Layer Perceptron Implementation

The Sklearn Multi-Layer Perceptron model was implemented in order to compare its performance with the custom implementation. The model was trained on the same 50-dimensional feature vector extracted from the CIFAR-10 images. The architecture of the Sklearn MLP is made to match the architecture of the custom MLP as closely as possible. Similar to the custom MLP, different variations of the Sklearn MLP were trained by adding/removing hidden layers, and changing the size of the hidden layers to analyze how these changes affect performance.

The Sklearn MLP model was trained using the same training settings as the custom MLP shown in Table 3.3.1.

3.4 Convolutional Neural Network

The custom Convolutional Neural Network (CNN) model was implemented in PyTorch using the **VGG11 architecture**. Unlike the other models, the CNN model was trained directly on the 32 x 32 x 3 CIFAR-10 images since CNNs are designed to handle high-dimensional images. The architecture of the VGG11 model is shown in Table 3.4.

Layer	**Description**
Conv Layer 1	Conv(3, 64, 3, 1) + ReLU + MaxPool
Conv Layer 2	Conv(64, 128, 3, 1) + ReLU + MaxPool
Conv Layer 3	Conv(128, 256, 3, 1) + ReLU
Conv Layer 4	Conv(256, 256, 3, 1) + ReLU + MaxPool
Conv Layer 5	Conv(256, 512, 3, 1) + ReLU
Conv Layer 6	Conv(512, 512, 3, 1) + ReLU + MaxPool
Conv Layer 7	Conv(512, 512, 3, 1) + ReLU
Conv Layer 8	Conv(512, 512, 3, 1) + ReLU + MaxPool
FC Layer 1	Linear(512, 4096) + ReLU + Dropout
FC Layer 2	Linear(4096, 4096) + ReLU + Dropout
Output Layer	Linear(4096, 10)

Table 3: VGG11 Architecture

The CNN architecture is divided into a **feature extractor** and a **classifier**, which together form the **forward pass** of the model. The feature extractor consists of a sequence of **convolutional layers**, followed by **batch normalization** and **ReLU activation**, with **max pooling** being applied at times. After the feature extractor processes the images, the output is flattened and passed into the classifier, which consists of fully connected layers, followed by ReLU activations and dropout. Finally, the output layer returns the class scores for each of the 10 CIFAR-10 classes [7].

The model is trained using the settings shown in Table 3.4.

Parameter	**Value**
Loss Function	Cross-Entropy Loss
Optimizer	Stochastic Gradient Descent (SGD)
Momentum	0.9
Weight Decay	1e-4
Learning Rate	0.01
Batch Size	32
Epochs	50

Table 4: CNN Training Settings

Training the CNN model consists of a **forward pass** to calculate the output, compute the **loss**, performs a **backward pass** to calculate gradients, and updates the weights using the optimizer. Basic performance metrics for each epoch are recorded for analysis.

To analyze how the architecture of the CNN affects the models performance, different versions of the model were trained. This includes changing the kernel size and layer depth of the CNN. By doing so, we can compare how a shallower CNN performs against a deeper CNN, and how smaller kernels perform against larger kernels.

4 Results and Analysis

4.1 Environment Setup

All experiments were conducted on the same personal computer to ensure consistency in results. The specifications of the environment used for training and evaluating the models are detailed in Table 4.1.

Component	**Specification**
Processor	Intel Core i7-8650U CPU @ 1.90GHz
GPU	N/A
RAM	8 GB
Operating System	Windows 11

Table 5: Environment Specifications

4.2 Gaussian Naive Bayes Results

Both the custom Gaussian Naive Bayes (GNB) implementation and the Sklearn GNB model were evaluated on the 50-dimensional PCA feature vectors extracted from the CIFAR-10 test set. The performance metrics for both models are summarized in Table 4.2.

Metric	**Custom GNB**	**Sklearn GNB**
Accuracy	0.79	0.79
Train Accuracy	0.82	0.82
Precision	0.79	0.79
Recall	0.79	0.79
F1-Score	0.79	0.79
Training Time (s)	0.06	0.26

Table 6: GNB Results

Both models show an identical performance across all metrics, indicating that the custom implementation is working correctly. With an accuracy of **79%**, the GNB models perform surprisingly well considering its simplicity and training time.

The confusion matrices for both models are shown in Figure 4.2. Both models once again show identical results, with most classes being classified correctly. However, some misclassifications are seen, such as cats being misclassified as dogs which is expected due to the similar features between the two classes.

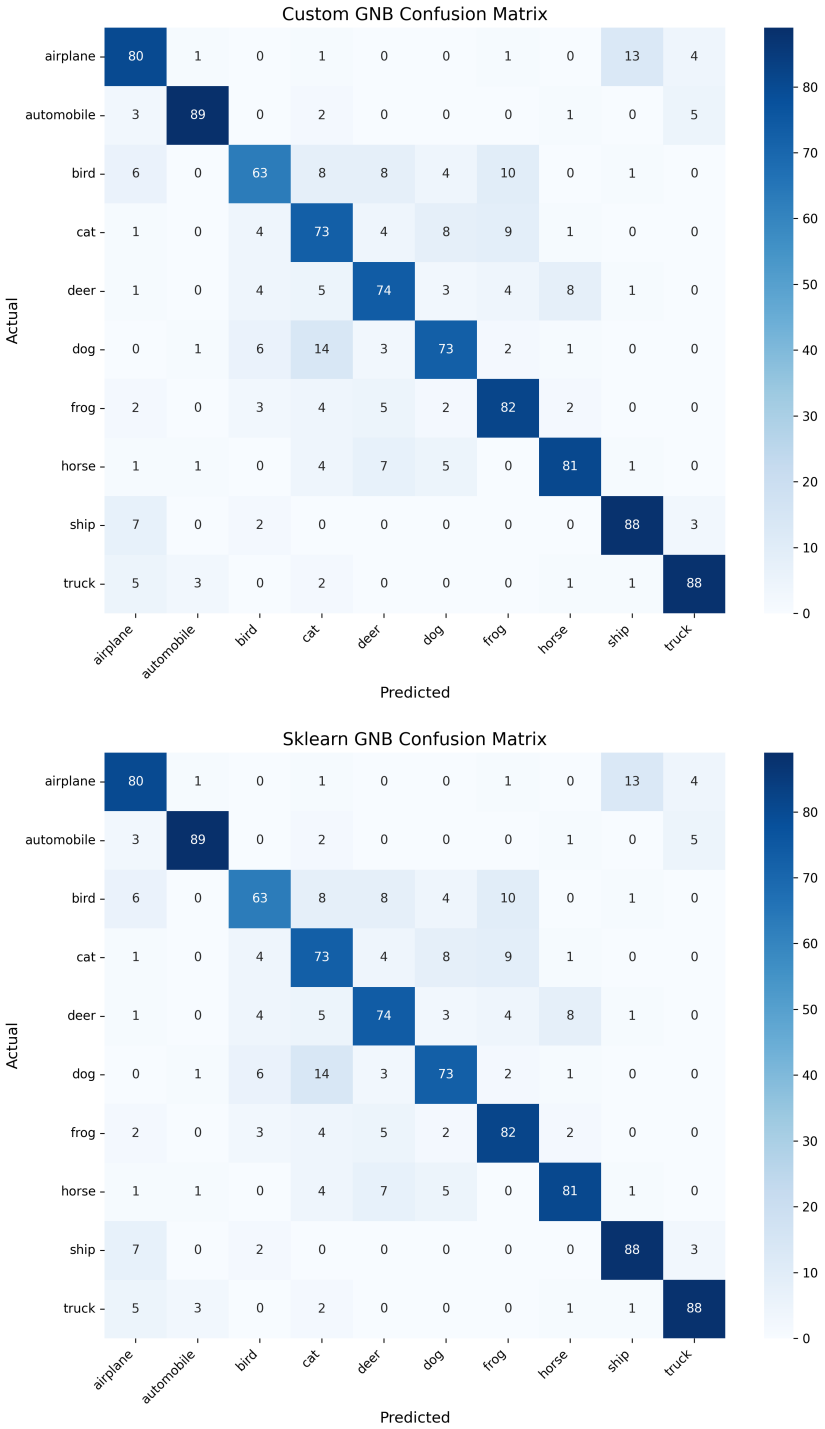

Figure 1: Confusion matrices for the Custom GNB (top) and Sklearn GNB (bottom).

Overall, the results show that both models result in identical classification performance on the CIFAR-10 dataset, with the custom implementation having a faster training time. The GNB model performs decently well considering its simplicity, but is limited due to its lack of complexity.

4.3 Decision Tree Results

4.3.1 Effect of Maximum Depth on Performance

Both the custom Decision Tree (DT) and the Sklearn DT were trained and evaluated on the 50-dimensional PCA feature vectors extracted from the CIFAR-10 data. To analyze the effect tree depth

has on performance, the models were trained with various maximum depths:

- Maximum Depth = 5
- Maximum Depth = 10
- Maximum Depth = 20
- Maximum Depth = 50

The peformance of the custom DT across these depths is summarized in Table 4.3.1.

Metric	**Depth 5**	**Depth 10**	**Depth 20**	**Depth 50**
Accuracy	0.55	0.60	0.57	0.57
Train Accuracy	0.62	0.88	1.00	1.00
Precision	0.56	0.61	0.57	0.57
Recall	0.55	0.60	0.57	0.57
F1-Score	0.54	0.60	0.57	0.57
Training Time (s)	206.57	284.12	369.03	406.09

Table 7: Custom DT Results with Varying Depths

The results show that the depth of the tree has a significant impact on the performance of the custom DT model. With a maximum depth of 5, the model is **underfitting** the data with a training accuracy of only **62%** and a test accuracy of **55%**. As the depth increases to 10, the model's performance improves with a training accuracy of **88%** and a test accuracy of **60%**. However, when the depth is increased to 20 and 50, the model achieves perfect training accuracy but the test accuracy drops to **57%**, indicating **overfitting**. Ultimately, a maximum depth of 10 provides the best outcome with a good balance between training and test accuracy.

4.3.2 Comparison Between Custom and Sklearn Decision Tree

After analyzing the effect of maximum depth on the custom Decision Tree (DT) model, a maximum depth of 10 was selected as the best performing depth due to its balance between training and test accuracy. Table 4.3.2 summarizes the performance metrics of both the custom and Sklearn DT models at this depth.

Metric	**Custom DT (Depth 10)**	**Sklearn DT (Depth 10)**
Accuracy	0.60	0.60
Train Accuracy	0.88	0.88
Precision	0.61	0.61
Recall	0.60	0.60
F1-Score	0.60	0.60
Training Time (s)	284.12	0.38

Table 8: Comparison of Custom and Sklearn DT Results

The results show that both models achieve identical performance across all metrics, indicating that the custom implementation is functioning correctly. However, the training time for the custom DT is significantly higher than that of the Sklearn implementation. This is largely because sckit-learns tree is implemented in optimized C/Cython code with several advanced libraries, while the custom implementation is done purely in Python/NumPy [8].

The confusion matrices for for both the custom and Sklearn models with a maximum depth of 10 are shown in Figure 4.3.2.

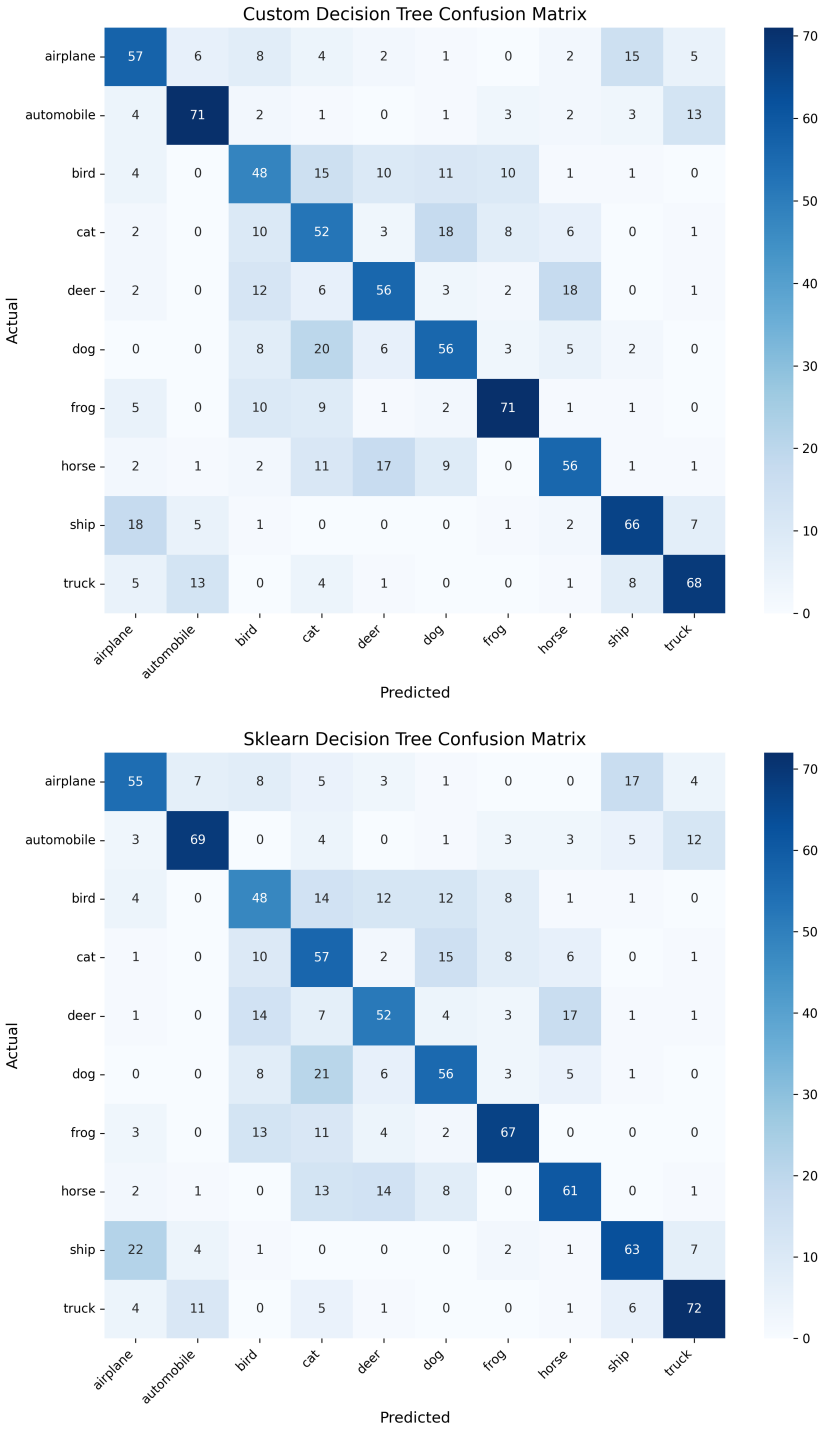

Figure 2: Confusion matrices for the Custom DT (top) and Sklearn DT (bottom) with a maximum depth of 10.

Both models show similar classification patterns. Classes with similar sizes and shapes, such as animals as a group, automobiles and trucks, airplanes and ships, tend to be misclassified more often. This is expected due to their similar features.

Overall, both the custom and Sklearn DT models perform similarly, with Sklearns implementation being significantly faster in training time. Despite this, the custom implementation successfully matches Sklearns performance, confirming that it is functioning correctly.

4.4 Multi-Layer Perceptron Results

4.4.1 Effect of Depth and Hidden Layer Size on Performance

Both the custom Multi-Layer Perceptron (MLP) and the Sklearn MLP were trained and evaluated on the 50-dimensional PCA feature vectors extracted from the CIFAR-10 data. To analyze how the depth and hidden layer size affect performance, five different custom architectures were tested:

- **Single-Layer:** $50 \rightarrow 10$
- **Shallow:** $50 \rightarrow 128 \rightarrow 10$
- **Base:** $50 \rightarrow 512 \rightarrow 512 \rightarrow 10$
- **Deep:** $50 \rightarrow 512 \rightarrow 512 \rightarrow 512 \rightarrow 512 \rightarrow 10$
- **Wide:** $50 \rightarrow 1024 \rightarrow 1024 \rightarrow 10$

All architectures were trained using the same training settings shown in Table 3.3.1. The performance of these architectures are summarized in Table 4.4.1.

Metric	Single-Layer	Shallow	Base	Deep	Wide
Accuracy	0.82	0.84	0.82	0.80	0.83
Train Accuracy	0.87	0.95	1.0	1.0	1.0
Precision	0.83	0.85	0.82	0.81	0.83
Recall	0.82	0.84	0.82	0.80	0.83
F1-Score	0.82	0.84	0.82	0.90	0.83
Training Time (s)	21.25	30.49	51.95	120.66	113.82

Table 9: Custom MLP Results with Varying Architectures

The results show that the increase of depth and width do not always lead to better performance. The shallow architecture had the best overall results with a test accuracy of **84%** and a training accuracy of **95%**, stipulating that it was able to learn the data well without overfitting. The single-layer MLP also performed well with a test accuracy of **82%** and a training accuracy of **87%**, showing that even a simple architecture can achieve good results. Where as the deeper and wider models attained **100%** training accuracy but had lower test accuracies of **80%** and **83%** respectively, suggesting overfitting.

4.4.2 Comparison Between Custom and Sklearn Multi-Layer Perceptron

Since the shallow architecture provided the best performance for the custom MLP, we will compare it against the Sklearn MLP with the same architecture. The performance metrics for both the custom and Sklearn MLP with the shallow architecture are summarized in Table 4.4.2.

Metric	Custom MLP	Sklearn MLP
Accuracy	0.84	0.83
Train Accuracy	0.95	0.97
Precision	0.85	0.83
Recall	0.84	0.83
F1-Score	0.84	0.83
Training Time (s)	30.49	6.33

Table 10: Comparison of Custom and Sklearn MLP Results with Shallow Architecture

Both implementations of the MLP with the shallow architecture achieve similar performance, with the custom MLP slightly outperforming the Sklearn MLP in every metric but training time where it takes nearly 5 times longer to train. This is likely due to the optimized nature of the Sklearn implementation compared to the custom PyTorch implementation. Despite the minor differences, both models demonstrate strong performance on the CIFAR-10 dataset with the shallow architecture, indicating that not only is this architecture and model suitable for this problem, but also that the custom implementation is working correctly.

To show how the shallow MLP trains over time, a subset of the training logs for the custom implementation is shown in Table 4.4.2.

Epoch	Train Loss	Train Accuracy
1	1.7782	48.70%
10	0.5128	83.64%
25	0.4067	85.88%
50	0.3350	88.38%
75	0.2726	90.88%
100	0.2421	92.06%

Table 11: Custom MLP Shallow Architecture Training Log (Subset)

The model starts with an accurary of **48.70%** in the first epoch and steadily improves over time, reaching a training accuracy of **92.06%** by the 100^{th} epoch but may have started to overfit near the 75^{th} epoch. This indicates that the model is effectively learning from the training data over time.

The confusion matrices for both models with the shallow architecture are shown in Figure 3.

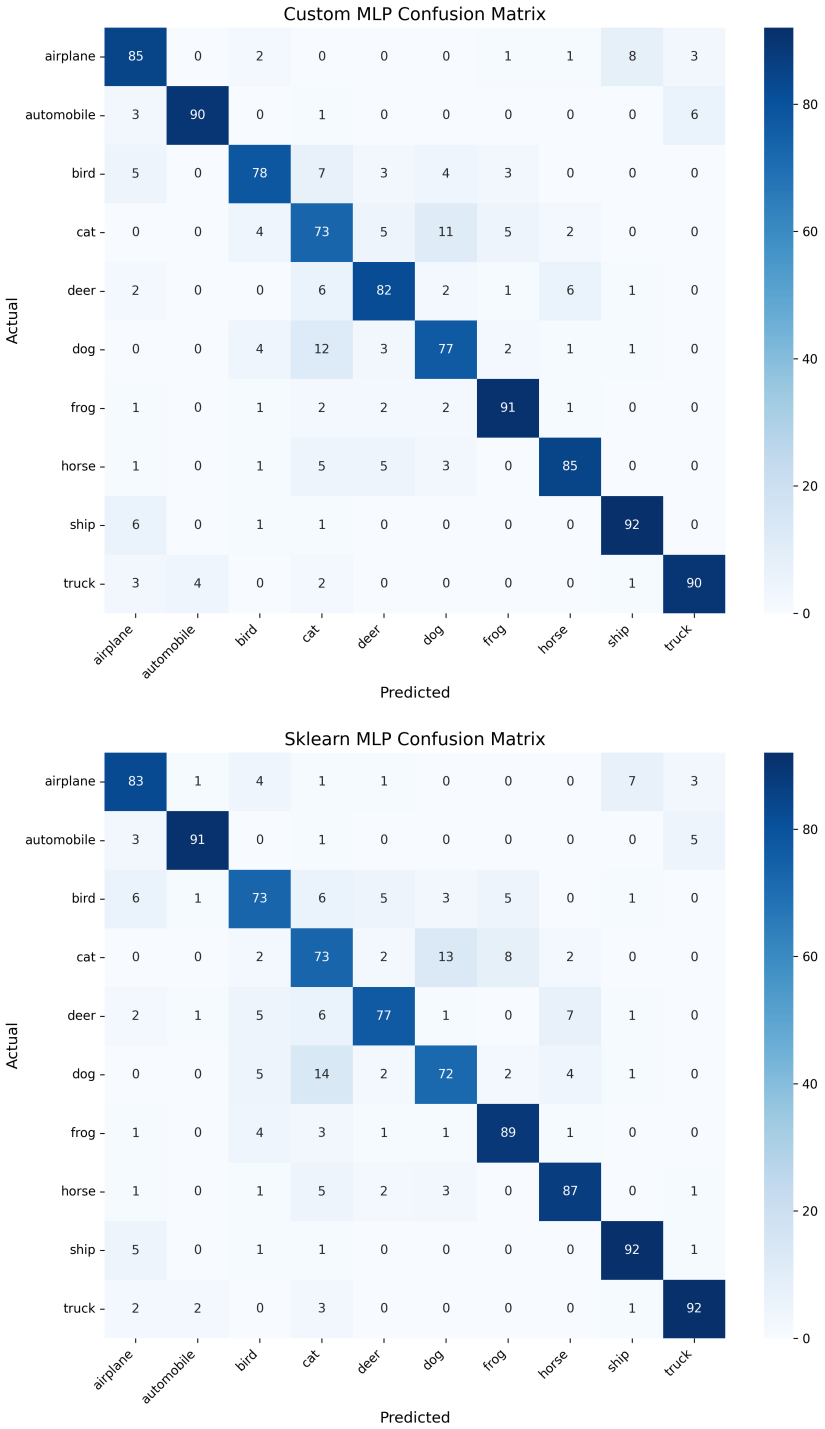

Figure 3: Confusion matrices for the Custom MLP (top) and Sklearn MLP (bottom) with shallow architecture.

Both models show similar classification patterns, with most classes being classified correctly. However, some misclassifications occur, such as mistaking cats and dogs for each other (12), which is expected due to their similar features. The similarities between both confusion matrices reinforces the idea that both implementations are functioning correctly.

In summary, the results show that both implementations of the MLP with the shallow architecture perform well on the CIFAR-10 dataset, with the custom implementation being slightly better in every metric except training time. Based on these results and analysis, the shallow MLP architecture seems to be an effective solution to the CIFAR-10 image classification problem.

4.5 Convolutional Neural Network Results

4.5.1 CNN Training Environment

Unlike the other models, the Convolutional Neural Network (CNN) model was trained directly on the 32 x 32 x 3 CIFAR-10 images. Due to the high computational power needed to train the CNN model, all experiments were conducted on Google Colab using the environment specifications detailed in Table 4.5.1.

Component	Specification
Processor	Intel Xeon CPU @ 2.20GHz
RAM	13 GB
GPU	NVIDIA Tesla T4

Table 12: CNN Training Environment Specifications

With the use of a GPU, the CNN model was able to be trained in a reasonable amount of time compared to training on a CPU [9].

4.5.2 Effect of Depth and Kernel Size on CNN Performance

To analyze how the depth and kernel size affect performance, three different CNN architectures were trained and evaluated:

- **Shallow CNN:** Shorter depth version of VGG11 with 3x3 kernels
- **Base:** full VGG11 architecture as described in Table 3.4
- **VGG11 with 5x5 Kernels:** Deeper version of VGG11 with 5x5 kernels instead of 3x3

All three models were trained with the same training settings shown in Table 3.4. The performance metrics for the different CNN architectures are summarized in Table 4.5.2.

Metric	Shallow CNN	VGG11 (Base)	VGG11 (Larger Kernels)
Accuracy	0.62	0.64	0.62
Train Accuracy	0.99	1.00	0.99
Precision	0.64	0.65	0.64
Recall	0.62	0.64	0.62
F1-Score	0.63	0.64	0.62
Training Time (s)	28.75	139.84	633.09

Table 13: CNN Architecture Results with Varying Depths and Kernel Sizes

The results show that by increasing the depth of the CNN from the shallow version to the full VGG11 architecture, there is a slight improvement in performance with the test accuracy increasing from **62%** to **64%**. However, comes at the cost of a significant increase in training time from **28.75 seconds** to **139.84 seconds**. On the other hand, increasing the kernel size from 3x3 to 5x5 in the VGG11 architecture does not lead to any performance improvement, with the test accuracy dropping back to **62%** and the training time increasing drastically to **633.09 seconds**. This is because larger kernels are more expensive and can blur critical features that require fine-grained detection [10].

Since the base VGG11 architecture had the best performance, we will only analyze its confusion matrix shown in Figure 4.5.2 since the other two models had similar classification patterns.

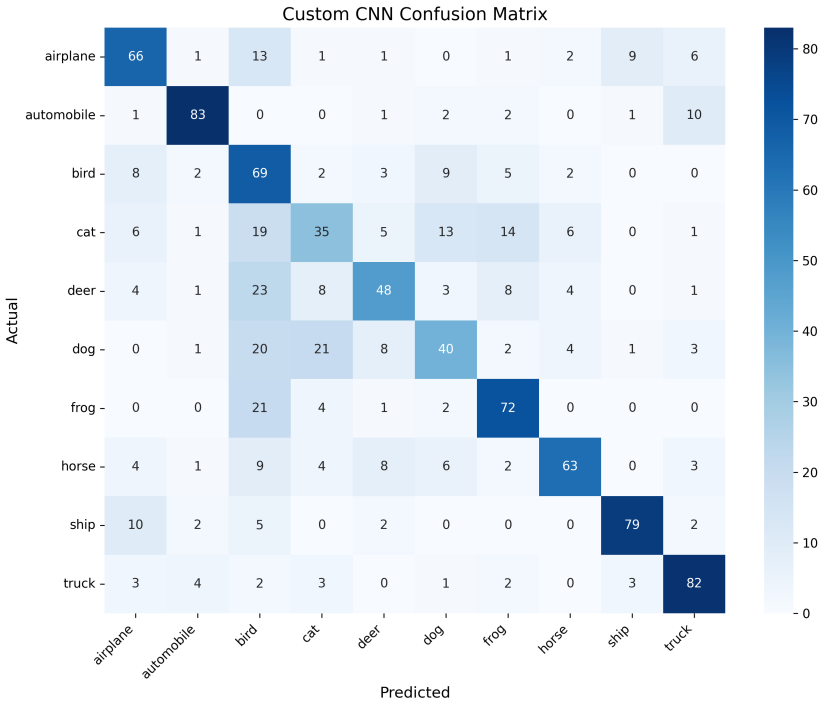

Figure 4: Confusion matrix for the VGG11 CNN architecture

The strongest classifications were seen mainly in the automobile and truck classes, due to their distinct shapes and features, it was either correctly classified or misclassified as each other. However, the model struggled the most with classes representing animals, such as cats, dogs, deers, and birds, likely due to their similar features and appearances.

4.5.3 Best CNN Model Analysis

While the VGG11 architecture was the best performing CNN model with an accuracy of **64%** (see Table 4.5.2). It is clear that the CNN model is not performing as well as expected on the CIFAR-10 dataset. This could be due to several factors, such as:

- **Insufficient Training Epochs:** The model was only trained for 50 epochs, which may not be enough for it to fully learn the complex features of the CIFAR-10 images.
- **Overfitting:** The model achieved a training accuracy of **100%**, indicating that it may have overfitted to the training data and is not classifying unseen data well.
- **Insufficient Data:** The model was trained on **5,000 training images** and not the full **50,000 training images** of the CIFAR-10 dataset, which may have limited its ability to learn different features.

As a result, the VGG11 CNN model is able to memorize the training data but struggles with unseen data, leading to mediocre performance on the CIFAR-10 test set.

In order to show how the VGG11 CNN trains over time, a subset of the training logs is shown in Table 4.5.3.

Epoch	Train Loss	Train Accuracy
1	1.9861	26.84%
5	1.1953	57.72%
10	0.6251	78.86%
20	0.1165	96.46%
30	0.0701	98.08%
40	0.0231	99.28%
50	0.0259	99.32%

Table 14: VGG11 Base CNN Training Log (Subset)

The model starts with a low accuracy of **26.84%** in the first epoch and steadily improves over time, reaching a training accuracy of **96.46%** by the 20^{th} epoch and from there, starts to show signs of overfitting as it reaches a training accuracy of 99.32% by the 50^{th} epoch. This indicates that while the model is effectively learning from the training data, it is also starting to memorize it, which could explain its poor performance on the test set.

Overall, while the VGG11 CNN model shows some promise with a test accuracy of **64%**, it is clear that there are several areas for improvement. By addressing the issues of insufficient training epochs, overfitting, and insufficient data, the performance of the CNN model on the CIFAR-10 dataset could be significantly improved.

5 Conclusion

This project explored the implementation and evlauation of four different mahcine learning models for image classification on the CIFAR-10 dataset: Gaussian Naive Bayes, Decision Tree, Multi-Layer Perceptron, and Convolutional Neural Network. Each model was implemented both from scratch and using the Sklearn library, with the exception of the CNN model, in order to compare their performance and ensure the custom models were implemented correctly. The goal was not only to achieve good performance results, but also to gain a better understanding of how these models work and perform when changing various hyperparameters and architectures such as depth, hidden layer size, and kernel size.

Three of the four models (GNB, DT, MLP) were trained and evaluated on a 50-dimensional PCA feature vector extracted from the CIFAR-10 subset, while the CNN model was trained directly on the subset of CIFAR-10 images. The PCA based models benefited from the reduced dimensionality, allowing them to train faster and more efficiently. The CNN model, on the other hand, had to handle the high-dimensional images directly, leading to longer training times and the need for more computational power.

The Gaussian Naive Bayes model, both custom and Sklearn, while being the simplest and having no tunable hyperparameters, still achieved a surprisingly strong performance. The custom Decision Tree results from Table 4.3.2, clearly shows that increasing depth does not always improve results. The deeper trees were obsiously overfitted, and the best-performing version was the model of depth 10. Both the custom and Sklearn version had nearly identical metrics, with the custom DT taking significally longer to train simply due to the optimizations in the Sklearn alogorithm. For the MLP, testing different depths and hidden layer sizes showed that the shallow architecture generalized better, while deeper and wider models memorized the training data while taking longer. The custom shallow MLP slightly outperformed the Sklearn version in every metric except training time (see Table 4.4.2). The CNN, which was expected to outperform the other models due to its ability to learn image features directly from the data, instead struggled. Training on only a small subset of CIFAR-10 caused the models to overfit, resulting in weaker test performances compared to the simpler models trained on PCA features.

Overall, this evaluation highlights how preprocessing, architecture and hyperparameter tuning, and training data all have a significant impact on model performance. Among all four models, the shallow MLP provided the best balance of performance and training time on the CIFAR-10 subset. Through these findings, we gained a deeper understanding of how different machine learning models behave under the same conditions with varying parameters, and how they can affect their performance as a whole.

References

[1] Ultralytics, "CIFAR-10 Dataset for Image Classification," Accessed: Jan. 2025. [Online]. Available: https://docs.ultralytics.com/datasets/classify/cifar10/

[2] A. Krizhevsky, "The CIFAR-10 Dataset," University of Toronto, Accessed: Jan. 2025. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

[3] Datacamp, "Curse of Dimensionality in Machine Learning," Accessed: Jan. 2025. [Online]. Available: https://www.datacamp.com/blog/curse-of-dimensionality-machine-learning

[4] IBM, "Naive Bayes Explained," Accessed: Jan. 2025. [Online]. Available: https://www.ibm.com/think/topics/naive-bayes

[5] Quantinsti, "Understanding the Gini Index," Accessed: Jan. 2025. [Online]. Available: https://blog.quantinsti.com/gini-index/

[6] Datacamp, "Multilayer Perceptrons in Machine Learning," Accessed: Jan. 2025. [Online]. Available: https://www.datacamp.com/tutorial/multilayer-perceptrons-in-machine-learning

[7] IBM, "Convolutional Neural Networks (CNNs): Explained," Accessed: Jan. 2025. [Online]. Available: https://www.ibm.com/think/topics/convolutional-neural-networks

[8] Scikit-learn, "Parallelism, Performance, and Optimization," Accessed: Jan. 2025. [Online]. Available: https://scikit-learn.org/stable/computing/parallelism.html

[9] Saturn Cloud, "Hardware Specs for Google Colab," Accessed: Jan. 2025. [Online]. Available: https://saturncloud.io/blog/whats-the-hardware-spec-for-google-colaboratory/

[10] Massed Compute, "Does a Larger CNN Filter Size Improve Performance?" Accessed: Jan. 2025. [Online]. Available: https://massedcompute.com/faq-answers/?question=Can%20a%20larger%20filter%20size%20in%20a%20CNN%20always%20lead%20to%20better%20performance?#:~:text=Increased%20Computational%20Cost:%20Larger%20filters,the%20model%20lacks%20contextual%20understanding