Image Classification Using Machine Learning and Deep Learning Approaches

Table of Contents

Abstract

This project focuses on performing image classification on the CIFAR-10 dataset using four models: Naïve Bayes (NB), Decision Tree (DT), Multilayer Perceptron (MLP), and Convolutional Neural Network (CNN). Feature extraction using ResNet-18 and Principal Component Analysis (PCA) was a key step for the NB, DT, and MLP models, significantly improving their performance. The best results were achieved by the MLP, which attained a testing accuracy of 81.9%, followed closely by NB at 79.4%. The DT and CNN models struggled with overfitting, limiting their generalization. A primary challenge encountered during the project was the computational limitations of the Google Colab Pro environment, particularly with memory usage when processing large images and batch sizes. To address this, compromises were made, including using smaller images to allow larger batch sizes and tuning a limited set of hyperparameters. These adjustments balanced resource constraints with achieving meaningful results. Detailed findings and insights are provided in the discussion.

1. Introduction

This project explores image classification on the CIFAR-10 dataset using both statistical and deep learning models. Two statistical models, Naïve Bayes (NB) and Decision Tree (DT), were implemented from scratch using the NumPy library, and their performance was compared with implementations from Scikit-learn. The Multilayer Perceptron (MLP) and Convolutional Neural Network (CNN) were developed using PyTorch.

Feature extraction for NB, DT, and MLP was performed using ResNet-18 to extract 512 features, followed by Principal Component Analysis (PCA) to reduce these features to 50 dimensions. The DT classifier was tuned for depth, with the optimal depth used to save the final model. The MLP was evaluated with variations in depth and hidden layer sizes, and the best-performing architecture was selected for final training and evaluation.

The CNN model is a re-implementation of the VGG-11 network, followed by three variations: a shallower network, a deeper network, and one with adjusted kernel sizes. CNNs were tuned for batch size and learning rate using a grid search. A custom training function was implemented that monitors train and validation accuracy and also uses early stopping for number of epochs. In this project, the test set was used also as the validation set due to dataset constraints.

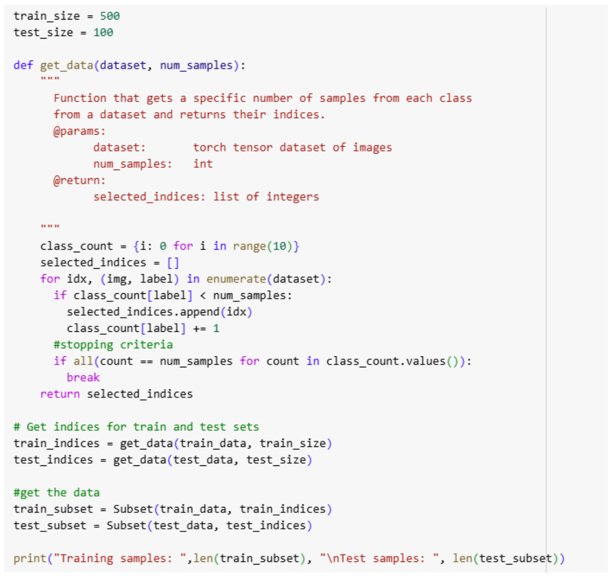
The performance of the implemented models were evaluated using confusion matrix, classification report and metrics such as accuracy, precision, recall, and F1-score.

2. Data Preprocessing

2.1 Subsample data

To reduce computational complexity, a subset of the CIFAR-10 dataset was used for this project. Specifically:

- **Training set**: 500 images per class, resulting in a total of 5000 images.
- **Test set**: 100 images per class, resulting in a total of 1000 images.



```python
train_size = 500
test_size = 100

def get_data(dataset, num_samples):
    """
      Function that gets a specific number of samples from each class
      from a dataset and returns their indices.
      @params:
            dataset:       torch tensor dataset of images
            num_samples:   int
      @return:
            selected_indices: list of integers

    """
    class_count = {i: 0 for i in range(10)}
    selected_indices = []
    for idx, (img, label) in enumerate(dataset):
      if class_count[label] < num_samples:
        selected_indices.append(idx)
        class_count[label] += 1
      #stopping criteria
      if all(count == num_samples for count in class_count.values()):
        break
    return selected_indices

# Get indices for train and test sets
train_indices = get_data(train_data, train_size)
test_indices = get_data(test_data, test_size)

#get the data
train_subset = Subset(train_data, train_indices)
test_subset = Subset(test_data, test_indices)

print("Training samples: ",len(train_subset), "\nTest samples: ", len(test_subset))
```

Figure 1.1. Subsampling process to extract balanced subsets for training and testing.

This subsampling ensured that the dataset remained balanced across the 10 object classes while being manageable for the computational resources available. Refer to Figure1.1 for implementation details in Python.

2.2 Feature Extraction

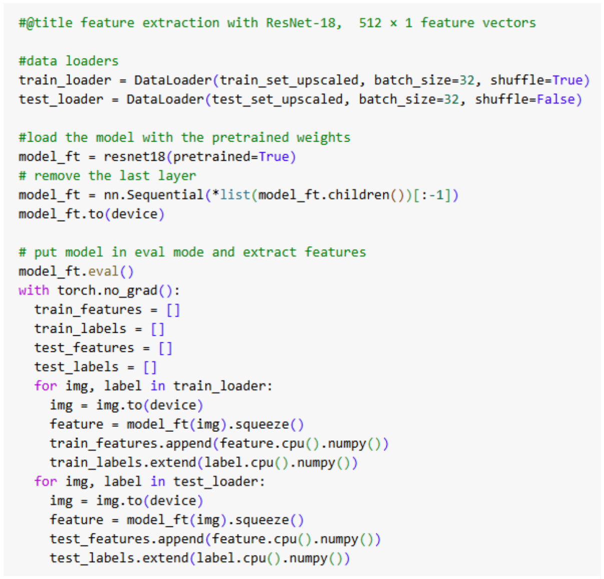
Feature extraction was performed to reduce the high dimensionality of the image data, making it suitable for models like Naïve Bayes, Decision Tree, and MLP. The following steps were taken:

Resizing and Normalization:

- Images were resized from 32×32×3 to 224×224×3 to meet the input size requirement of ResNet-18.
- Normalization was applied using the mean and standard deviation of the ImageNet dataset, ensuring consistency with the pre-trained ResNet-18 model.

ResNet-18 Feature Extraction:

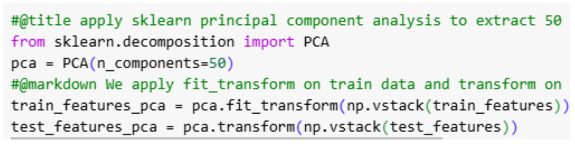
- The pre-trained ResNet-18 model (with its final layer removed) was used to extract a 512-dimensional feature vector for each image. See implementation in Figure 2.1.



```python
#@title feature extraction with ResNet-18,  512 x 1 feature vectors

#data loaders
train_loader = DataLoader(train_set_upscaled, batch_size=32, shuffle=True)
test_loader = DataLoader(test_set_upscaled, batch_size=32, shuffle=False)

#load the model with the pretrained weights
model_ft = resnet18(pretrained=True)
# remove the last layer
model_ft = nn.Sequential(*list(model_ft.children())[:-1])
model_ft.to(device)

# put model in eval mode and extract features
model_ft.eval()
with torch.no_grad():
  train_features = []
  train_labels = []
  test_features = []
  test_labels = []
  for img, label in train_loader:
    img = img.to(device)
    feature = model_ft(img).squeeze()
    train_features.append(feature.cpu().numpy())
    train_labels.extend(label.cpu().numpy())
  for img, label in test_loader:
    img = img.to(device)
    feature = model_ft(img).squeeze()
    test_features.append(feature.cpu().numpy())
    test_labels.extend(label.cpu().numpy())
```

Figure 2.1. Feature extraction pipeline showing the use of ResNet-18

Finally, sklearn PCA is used to extract the 50 features. The fit_transform() function is called on the train set and separately to avoid data leakage the transform() function is called on the test set. See Figure 2.2.



```python
#@title apply sklearn principal component analysis to extract 50
from sklearn.decomposition import PCA
pca = PCA(n_components=50)
#@markdown We apply fit_transform on train data and transform on
train_features_pca = pca.fit_transform(np.vstack(train_features))
test_features_pca = pca.transform(np.vstack(test_features))
```

Figure 2.2. PCA dimensionality reduction process and the resulting feature vectors.

The shapes of resulting vectors are:

- Train PCA feature vector: 5000 x 50
- Test feature vector: 1000 x 50

2.3 Validation Set Use

In this project, the provided test set was used as the validation set to monitor performance during training. This approach was necessary due to dataset constraints, which required balancing between training, validation, and computational feasibility. It should be noted that while this allowed for hyperparameter tuning and performance monitoring, the absence of a separate unseen test set limits the ability to assess true generalization.

3. Gaussian Naïve Bayes

3.1 Implementation

The Gaussian Naïve Bayes (NB) classifier was implemented from scratch using NumPy to better understand the algorithm's internal

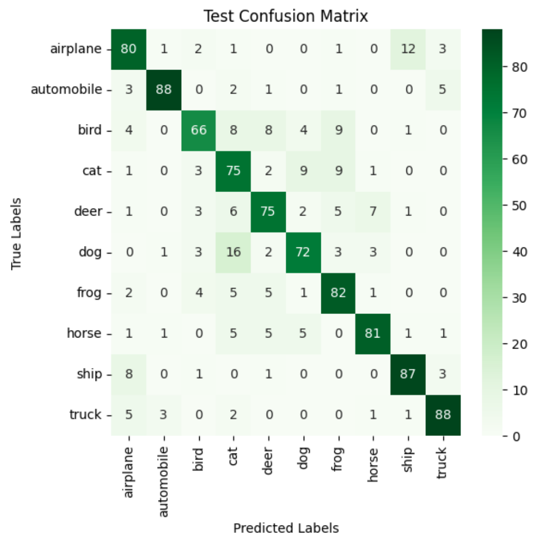
```
Define class GaussNaiveBayes with inputs: features (X) and labels (y).

  Define `fit (self)`:
      For each class in unique class in y:
          Extract class-specific rows from X
          Compute and store (mean, variance) of features of that class

  Define `predict (X)`:
      Initialize empty array `predictions`
      For each sample in `X`, calculate posterior probabilities:
          For each class, compute log-likelihoods using mean & variance
          Sum log-likelihoods and add log prior to get posterior probability
      Assign the class with the highest posterior to `predictions`
      Return `predictions`
```

Figure 3.1. Gaussian NB pseudocode of NumPy implementation

workings. The pseudocode is illustrated in Figure 3.1.

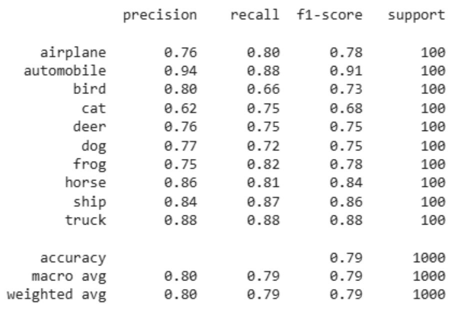
3.2 Results:

- Training accuracy: **81.70%**
- Testing accuracy: **79.40%**
- Confusion matrix (Figure 3.2).
- Classification report (Figure3.3).



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	80	1	2	1	0	0	1	0	12	3
automobile	3	88	0	2	1	0	1	0	0	5
bird	4	0	66	8	8	4	9	0	1	0
cat	1	0	3	75	2	9	9	1	0	0
deer	1	0	3	6	75	2	5	7	1	0
dog	0	1	3	16	2	72	3	3	0	0
frog	2	0	4	5	5	1	82	1	0	0
horse	1	1	0	5	5	5	0	81	1	1
ship	8	0	1	0	1	0	0	0	87	3
truck	5	3	0	2	0	0	0	1	1	88

Figure 3.2. NB from scratch confusion matrix



```
              precision    recall  f1-score   support

    airplane       0.76      0.80      0.78       100
  automobile       0.94      0.88      0.91       100
        bird       0.80      0.66      0.73       100
         cat       0.62      0.75      0.68       100
        deer       0.76      0.75      0.75       100
         dog       0.77      0.72      0.75       100
        frog       0.75      0.82      0.78       100
       horse       0.86      0.81      0.84       100
        ship       0.84      0.87      0.86       100
       truck       0.88      0.88      0.88       100

    accuracy                           0.79      1000
   macro avg       0.80      0.79      0.79      1000
weighted avg       0.80      0.79      0.79      1000
```

Figure 2.3. Classification report for NB classifier

The model achieves an overall accuracy of 79%, with macro-averaged precision, recall, and F1-score all 79%, demonstrating strong performance for a simple probabilistic classifier.

High-performing classes, such as Automobile, Truck, Ship, and Horse reflect the model's ability to distinguish well-separated feature spaces for these categories. However, the model struggles with Cat often misclassifying it as Dog, likely due to overlapping feature distributions and the assumption of feature independence in NB. The findings in the classification report are consistent with the confusion matrix where misclassifications often occur between visually similar classes.

3.3 Comparison with Sklearn NB model

To validate the correctness of the implementation, a comparison was made with the Scikit-learn implementation of Gaussian NB. Results were found to be nearly identical, indicating that the custom implementation functions as expected.

3.4 Overall performance of NB

The NB does well but shows some overfitting. The metrics suggest the model performs well for a simple NB classifier, particularly when enhanced with ResNet-18 and PCA feature extraction.

4. Decision tree classifier

4.1 Pseudocode:

```
Class Node:
  def __init__(self, feature, threshold, value, left, right):
    Initialize a tree node with:
    - `feature`: Index of the feature used for the split.
    - `threshold`: Value at which the split occurs.
    - `value`: Predicted class if it is a leaf node.
    - `left`: Left subtree.
    - `right`: Right subtree.
 def Gini(y):
      Compute Gini impurity: Gini = 1 - sum(probabilities^2),
      where probabilities are the frequencies of each class in
      `y`.
 def map_labels(y):
      Map class labels in `y` to contiguous integers for eas-
      ier computation.
      Return both the mapped labels and the original clas-
      ses.

 def Split_Node(X, y, metric):
        Identify the best feature and threshold for splitting:
        - Iterate through all features.
        - For each feature, compute impurity reduction using
      the given `metric` (e.g., Gini).
        - Select the feature and threshold with the highest
      gain.
```

Figure 4.1. Pseudocode for the DT Node class

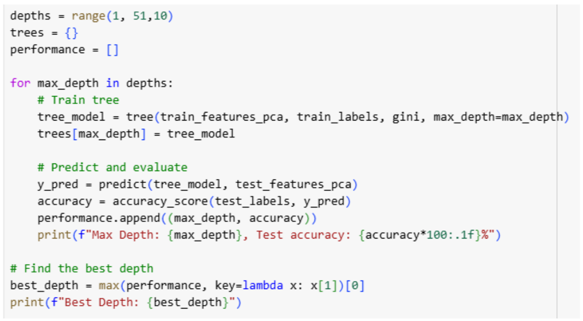
```
def Build_Tree(X, y, metric, depth, max_depth, min_sam-
ples_split):
     Build the decision tree recursively:
     Stop conditions:
          No improvement in split.
          Maximum depth reached.
          Number of samples in the dataset is less than
          `min_samples_split`.
     Create a leaf node with the most common class if any
     stop condition is met.
     Otherwise:
          Find the best feature and threshold using
          `Split_Node`.
          Split the data into left and right subsets.
          Recursively build the left and right subtrees.

def pred_node(node, input):
     Traverse the tree from root to leaf based on input fea-
     ture values.
     Return the predicted class at the leaf node.

def predict(tree, X):
     Apply `pred_node` to each sample in the dataset `X`.
     Return an array of predicted classes.
```

Figure 4.2. Pseudocode for DT build function. and prediction function

The Decision Tree (DT) classifier was implemented from scratch using NumPy as depicted in Figures 4.1 and 4.2. While the implementation functioned correctly, it suffered from slow computation and high memory usage, likely due to double recursion during tree-building, which can lead to inefficiencies when handling large datasets.

4.2 Training trees of different depth

To analyze the impact of tree depth on model performance, the decision tree classifier was trained and evaluated across varying depths. The process involved systematically varying the maximum depth of the tree and recording the performance at each depth (See figure 4.3 for Python code).



```python
depths = range(1, 51,10)
trees = {}
performance = []

for max_depth in depths:
    # Train tree
    tree_model = tree(train_features_pca, train_labels, gini, max_depth=max_depth)
    trees[max_depth] = tree_model

    # Predict and evaluate
    y_pred = predict(tree_model, test_features_pca)
    accuracy = accuracy_score(test_labels, y_pred)
    performance.append((max_depth, accuracy))
    print(f"Max Depth: {max_depth}, Test accuracy: {accuracy*100:.1f}%")

# Find the best depth
best_depth = max(performance, key=lambda x: x[1])[0]
print(f"Best Depth: {best_depth}")
```

Figure 4.3. Python implementation for training decision trees at different depths

Implementation Steps:

1. **Depth Range**: from 1 to 50, stepping by 10.
2. **Storage**: A dictionary to store trained trees with their corresponding depths and a list to record the accuracy at each depth.
3. **Loop Over Depths**: For each depth in the specified range:

   Train Tree with the current depth as a parameter, along with the training features and labels.

   Store Trained Tree in the dictionary, using depth as key.

   Predict Labels: Use the predict function on the test dataset.

   Evaluate Test Accuracy by comparing the predicted labels with the true test labels.

   Record Performance of the current depth and its accuracy in the performance list.

   Print Results: depth and accuracy.

4. **Find Optimal Depth**: Identify the depth with the highest accuracy from the performance list by finding the maximum accuracy value.
5. **Plot Results**: See Figure 4.4.
6. **Retrain Optimal Tree**: Train a final decision tree using the optimal depth determined in Step 4 on the full training dataset.

```
Best Depth: 11
```

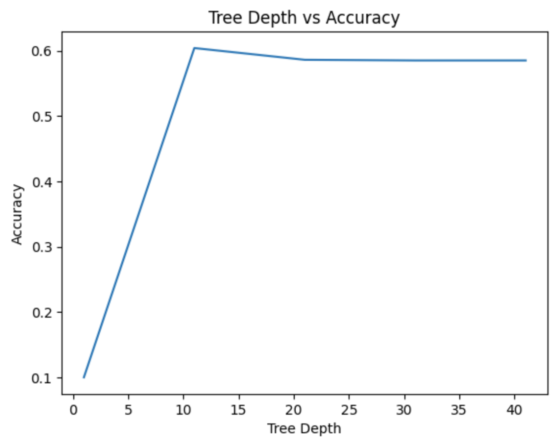
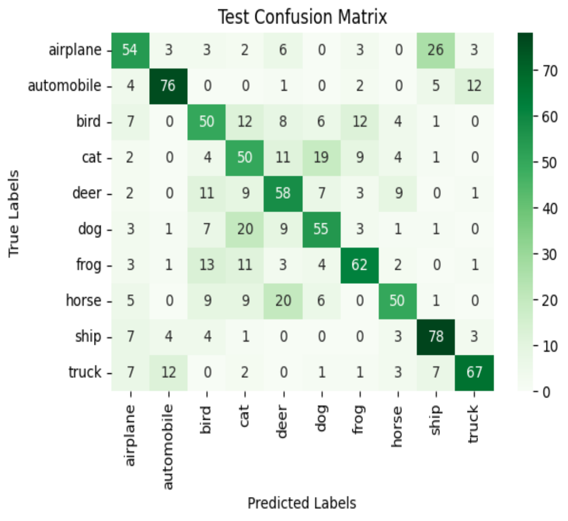


Figure 4.4. Graph depicting tree depth vs accuracy

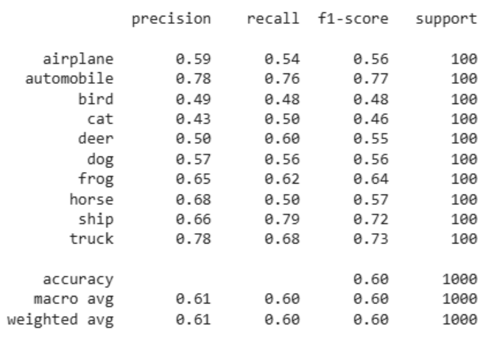
4.3 Results:

- Train accuracy: **87.34%**
- Test accuracy: **60.40%**
- Confusion matrix (Figure 4.5)
- Classification report (Figure 4.6)
- Best depth: 11



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	54	3	3	2	6	0	3	0	26	3
automobile	4	76	0	0	1	0	2	0	5	12
bird	7	0	50	12	8	6	12	4	1	0
cat	2	0	4	50	11	19	9	4	1	0
deer	2	0	11	9	58	7	3	9	0	1
dog	3	1	7	20	9	55	3	1	1	0
frog	3	1	13	11	3	4	62	2	0	1
horse	5	0	9	9	20	6	0	50	1	0
ship	7	4	4	1	0	0	0	3	78	3
truck	7	12	0	2	0	1	1	3	7	67

Figure 4.5. DT Confusion Matrix



```
              precision    recall  f1-score   support

    airplane       0.59      0.54      0.56       100
  automobile       0.78      0.76      0.77       100
        bird       0.49      0.48      0.48       100
         cat       0.43      0.50      0.46       100
        deer       0.50      0.60      0.55       100
         dog       0.57      0.56      0.56       100
        frog       0.65      0.62      0.64       100
       horse       0.68      0.50      0.57       100
        ship       0.66      0.79      0.72       100
       truck       0.78      0.68      0.73       100

    accuracy                           0.60      1000
   macro avg       0.61      0.60      0.60      1000
weighted avg       0.61      0.60      0.60      1000
```

Figure 4.6. DT Classification Report

The model's overall performance is moderate, reflecting its limited ability to generalize from the feature vectors. Strong classes include Automobile, Ship and Truck, while weak classes bird and cat. Similarly to the NB, the weighted averages (Precision = 0.61, Recall = 0.60, F1-Score = 0.60) align with the overall accuracy, indicating that the model does not significantly favor any specific class. Like the NB, DT also struggles with classifying objects with similar features like animals or some that have similar backgrounds (for e.g. Airplane and Ship might both have sky)

4.4 Comparison with Scikit-learn decision tree

Computational Efficiency:

The Scikit-learn implementation was significantly faster than the custom implementation.

This efficiency is due to Scikit-learn's highly optimized code.

Performance Comparison:

Both implementations identified similar optimal tree depths, with Scikit-learn's best depth also being 11. Scikit-learn's model achieved a training accuracy of 91.74%, slightly higher than the custom implementation, likely due to fine-tuned internal optimizations. The test accuracy for Scikit-learn's model was 60.60%, very close to the 60% achieved by the custom implementation, demonstrating the correctness of the custom implementation.

Limitations of the Custom Tree:

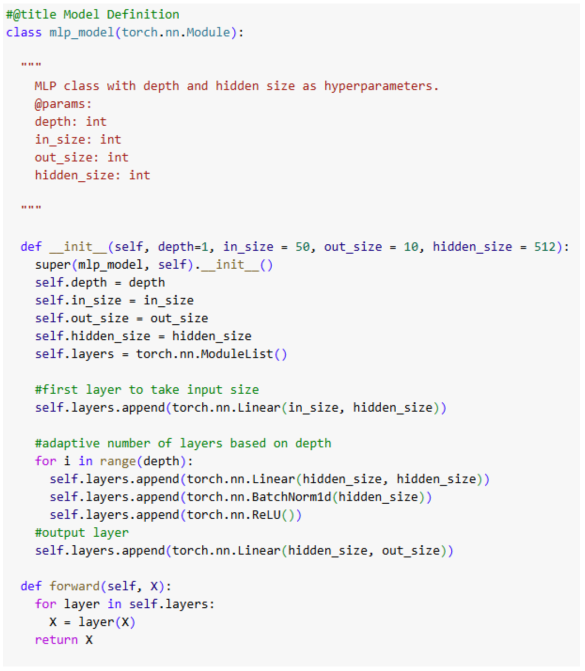
Speed: The double recursion in the custom implementation led to slower training times, particularly for deeper trees.

Memory Usage: Recursive splitting resulted in higher memory consumption, making it less efficient for large datasets.

Improvements: The custom DT can benefit from parallelization or vectorization to reduce computational time.

5. MLP implementation

5.1 Code



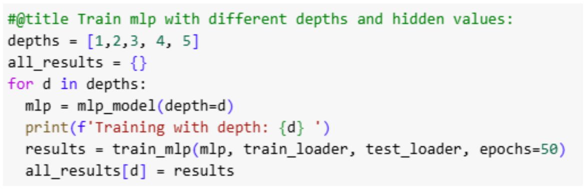
```
#@title Model Definition
class mlp_model(torch.nn.Module):

  """
    MLP class with depth and hidden size as hyperparameters.
    @params:
    depth: int
    in_size: int
    out_size: int
    hidden_size: int

  """

  def __init__(self, depth=1, in_size = 50, out_size = 10, hidden_size = 512):
    super(mlp_model, self).__init__()
    self.depth = depth
    self.in_size = in_size
    self.out_size = out_size
    self.hidden_size = hidden_size
    self.layers = torch.nn.ModuleList()

    #first layer to take input size
    self.layers.append(torch.nn.Linear(in_size, hidden_size))

    #adaptive number of layers based on depth
    for i in range(depth):
      self.layers.append(torch.nn.Linear(hidden_size, hidden_size))
      self.layers.append(torch.nn.BatchNorm1d(hidden_size))
      self.layers.append(torch.nn.ReLU())
    #output layer
    self.layers.append(torch.nn.Linear(hidden_size, out_size))

  def forward(self, X):
    for layer in self.layers:
      X = layer(X)
    return X
```

Figure 5.1. PyTorch Implementation of an MLP that takes depth and hidden size as arguments

5.2 Train MLP at different depths

As the depth increases, the model generally learns more complex patterns. However, beyond a certain depth, overfitting may occur, reducing test accuracy. Optimal depth provides a balance between model complexity and generalization.

In terms of implementation this is done by looping and having results stored in a dictionary and later plotted. See figure 5.2 for the Python code and Figure 5.3 for the plots depicting training and testing performance at different depths.



```
#@title Train mlp with different depths and hidden values:
depths = [1,2,3, 4, 5]
all_results = {}
for d in depths:
  mlp = mlp_model(depth=d)
  print(f'Training with depth: {d} ')
  results = train_mlp(mlp, train_loader, test_loader, epochs=50)
  all_results[d] = results
```

Figure 5.2. Code to implement the training at different depths an mlp

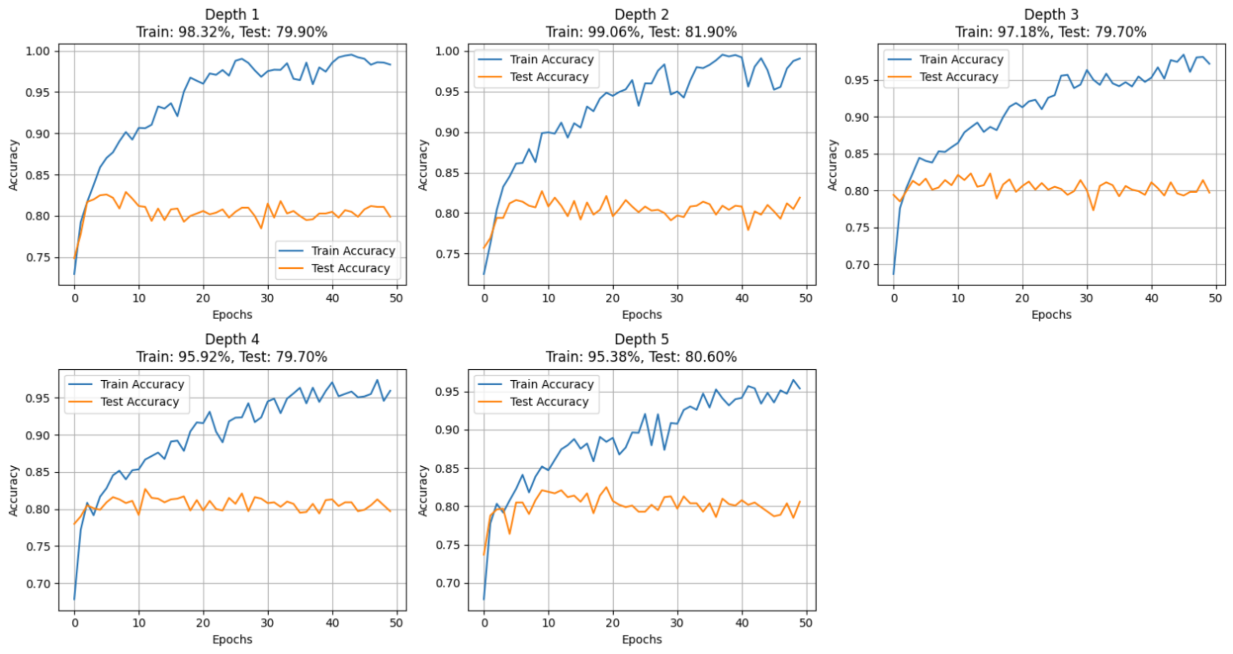


Figure 5.3. Graphs depicting various performances of the mlp at different layer depth

Optimal depth was achiever at 2. In Figure 5.3 second graph from the left on first row depicts the performance and overall average testing accuracy of 81.9%. At depth 1 the model is not complex enough however the parameter space is significantly more complex at higher depths. However, adding more layers beyond 2 did not improve performance.

5.3 Train with different hidden sizes

Smaller hidden sizes are computationally efficient but may underfit the data. Larger hidden sizes increase the model's complexity and training accuracy but risk overfitting and provide minimal gains in test accuracy.

In terms of implementation the code is very similar to the depth analysis. The best depth was used to search for best hidden size.

Figure 5.4 depicts the graphs provided by the search. The best hidden size was found to be 32. Although 128 shows similar results, hidden size 32 achieves the second-highest test accuracy (81.20%) while maintaining a small gap between training and test performance.

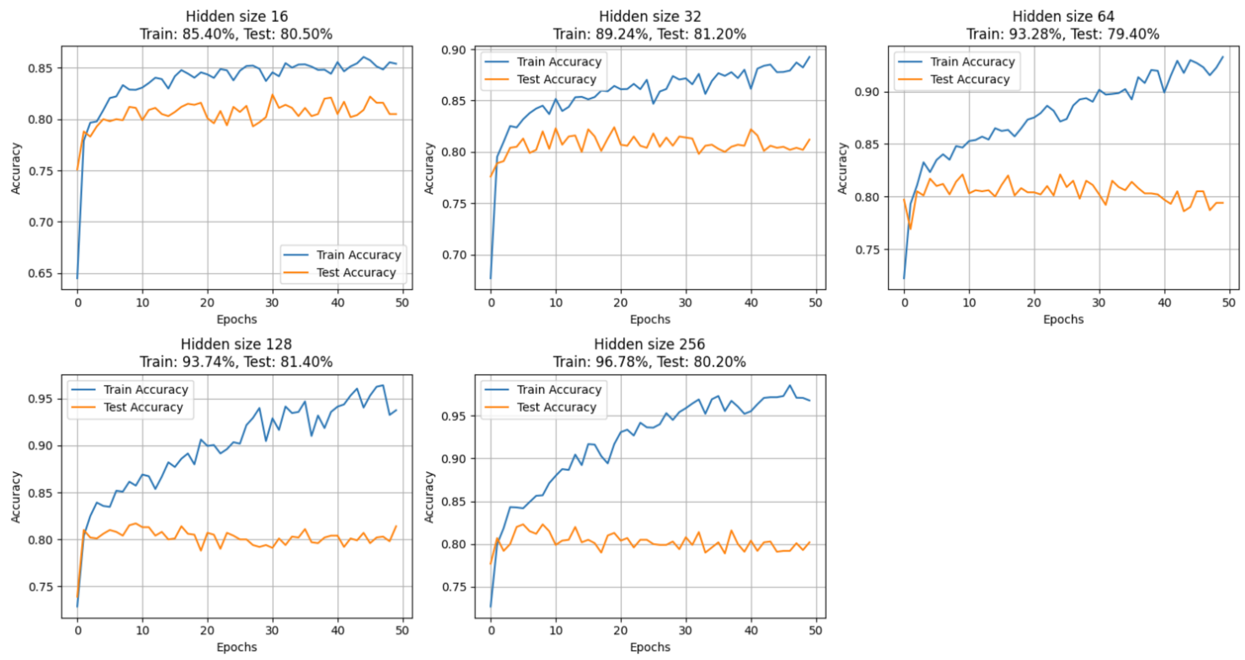
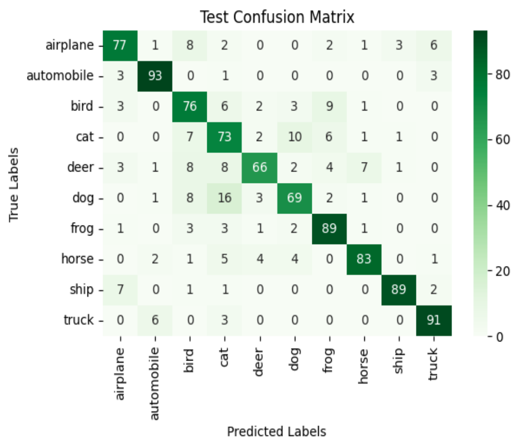


Figure 5.4. Performance of mlp with different hidden sizes

Smaller gaps between train and test accuracies indicate better generalization. Increasing hidden size beyond 32 results in marginal improvements or decreases in test accuracy, while significantly increasing computational cost and overfitting risk.
A final best model was trained using these two parameters and its performance is reported. The MLP was trained with Categorical cross entropy loss, batch size of 64, a learning rate of 0.1, and SGD optimizer with momentum of 0.9.

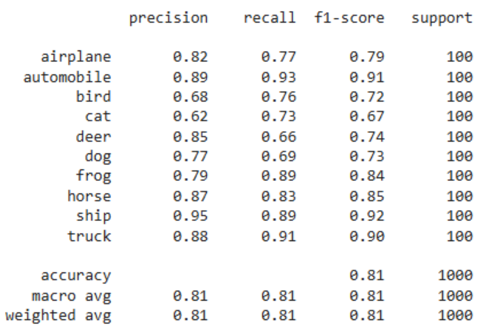
5.4 Results

- training accuracy: **87.32%**
- test accuracy: **81.9%**
- confusion matrix (Figure 5.5)
- classification report (Figure 5.6)



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	77	1	8	2	0	0	2	1	3	6
automobile	3	93	0	1	0	0	0	0	0	3
bird	3	0	76	6	2	3	9	1	0	0
cat	0	0	7	73	2	10	6	1	1	0
deer	3	1	8	8	66	2	4	7	1	0
dog	0	1	8	16	3	69	2	1	0	0
frog	1	0	3	3	1	2	89	1	0	0
horse	0	2	1	5	4	4	0	83	0	1
ship	7	0	1	1	0	0	0	0	89	2
truck	0	6	0	3	0	0	0	0	0	91

Figure 5.5. Confusion matrix for MLP



```
              precision    recall  f1-score   support

    airplane       0.82      0.77      0.79       100
  automobile       0.89      0.93      0.91       100
        bird       0.68      0.76      0.72       100
         cat       0.62      0.73      0.67       100
        deer       0.85      0.66      0.74       100
         dog       0.77      0.69      0.73       100
        frog       0.79      0.89      0.84       100
       horse       0.87      0.83      0.85       100
        ship       0.95      0.89      0.92       100
       truck       0.88      0.91      0.90       100

    accuracy                           0.81      1000
   macro avg       0.81      0.81      0.81      1000
weighted avg       0.81      0.81      0.81      1000
```

Figure 5.6. Classification report for MLP

The confusion matrix demonstrates that the MLP classifier also achieves higher accuracy for structured classes like Automobile Ship, and Truck, reflecting the strength of the extracted feature vectors. However, significant misclassifications occur between animal classes similarly to NB and DT, particularly Cat, highlighting challenges in distinguishing visually similar patterns. Future work could involve fine-tuning the feature extraction process, introducing regularization techniques to address these limitations, and tuning hyperparameters such as batch size and learning rate. However, overall, it performed better than the previous two models.

6. CNN

The CNN models implemented are based on variations of the base model, VGG11, architecture. These models were designed, trained, and tuned to optimize performance on the CIFAR-10 dataset. For this section the ResNet-18 features were not used. Instead, the raw images were fed directly into the model. However, the subset of 5000 train and 1000 test images was maintained as per requirements of the project. Following sections detail the process and findings.

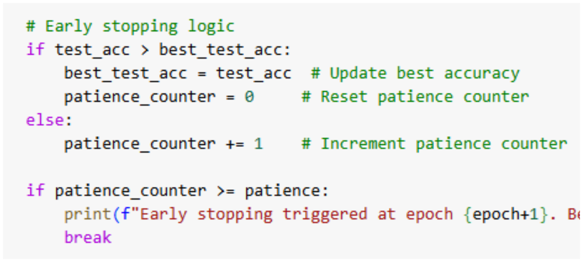
Model Variations: Their architectures were derived with modifications shallow, deeper, and kernel-size variations.

Preprocessing: Input images were not resized from 32×32×3 (native size for CIFAR-10) due computational constraints on Google Colab Pro. The program would run out of memory when trying to train large image sizes (larger than 64x64x3) with high batch sizes (higher than 128). They were normalized using the ImageNet mean and standard deviation. The reader is reminded that the validation set used in this section is the test set as stated in the introduction.

Training Function: A training function was developed to incorporate validation during training to monitor generalization on unseen data for each epoch and early stopping based on validation accuracy to prevent overfitting. Models were trained with SGD optimizer with momentum 0.9 and Categorical Cross Entropy loss.

Early Stopping Logic
Early stopping was implemented to monitor validation accuracy and halt training when no improvement was observed for a defined number of epochs (patience). This technique helped to prevent unnecessary computation.

The Python code snippet below in Figure 6.0. illustrates the logic used for early stopping:



```
# Early stopping logic
if test_acc > best_test_acc:
    best_test_acc = test_acc  # Update best accuracy
    patience_counter = 0     # Reset patience counter
else:
    patience_counter += 1    # Increment patience counter

if patience_counter >= patience:
    print(f"Early stopping triggered at epoch {epoch+1}. Be
    break
```

Figure 6.0. Python implementation of early stopping logic.

This implementation resets the patience counter whenever a new best accuracy is achieved. If no improvement is observed for patience consecutive epochs, training is terminated early, and the best model state is saved.

Hyperparameter Tuning: A grid search was performed to identify the optimal batch size and learning rate for each model using validation accuracy as the metric. Hyperparameter ranges were:

Batch sizes: [128, 256, 512]

Learning rates: [0.01, 0.001, 0.0001]

The combination yielding the highest validation accuracy was used for final training and evaluation.

6.1 Re-implementation of VGG11

The classification report and confusion matrix for the VGG-11 model reveal notable overfitting (See gap in train and validation per epoch in figure 6.1). Best hyperparameters were batch size: 512 and learning rate: 0.01. The model reached optimal validation accuracy after 34 epochs. It is complex and has 28 million learnable parameters (see appendix i. for model summary).

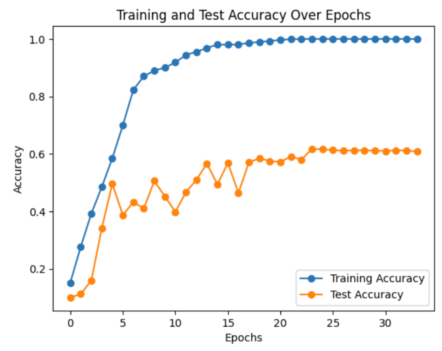
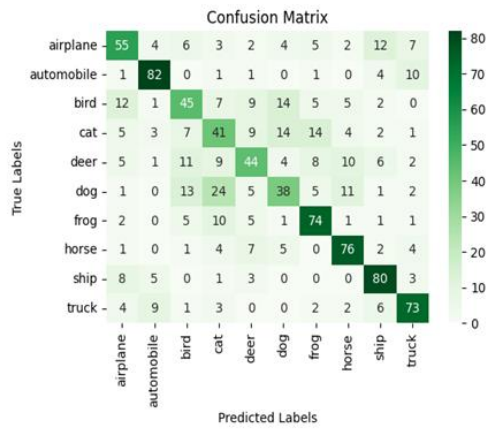


Figure 6.1. Training and validation per epoch for VGG11

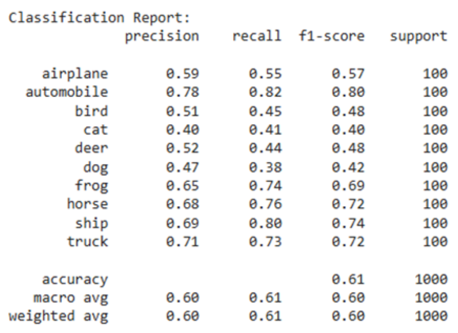
Results:

- training accuracy: **99.96%**
- validation accuracy: **61.6%**.
- Confusion matrix (Figure 6.2)
- Classification report (Figure 6.3)



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	55	4	6	3	2	4	5	2	12	7
automobile	1	82	0	1	1	0	1	0	4	10
bird	12	1	45	7	9	14	5	5	2	0
cat	5	3	7	41	9	14	14	4	2	1
deer	5	1	11	9	44	4	8	10	6	2
dog	1	0	13	24	5	38	5	11	1	2
frog	2	0	5	10	5	1	74	1	1	1
horse	1	0	1	4	7	5	0	76	2	4
ship	8	5	0	1	3	0	0	0	80	3
truck	4	9	1	3	0	0	2	2	6	73

Figure 6.2. VGG11 confusion matrix



```
Classification Report:
              precision    recall  f1-score   support

    airplane       0.59      0.55      0.57       100
  automobile       0.78      0.82      0.80       100
        bird       0.51      0.45      0.48       100
         cat       0.40      0.41      0.40       100
        deer       0.52      0.44      0.48       100
         dog       0.47      0.38      0.42       100
        frog       0.65      0.74      0.69       100
       horse       0.68      0.76      0.72       100
        ship       0.69      0.80      0.74       100
       truck       0.71      0.73      0.72       100

    accuracy                           0.61      1000
   macro avg       0.60      0.61      0.60      1000
weighted avg       0.60      0.61      0.60      1000
```

Figure 6.3. VG11 Classification Report

This model has the same strong classes automobile and ship and weaker classes cat, bird and deer. The overfitting signifies that it memorizes data but struggles to generalize. IT performs worse than the NB and mlp did on the features extracted with ResNet-18 and PCA.

6.2 Shallow CNN

Next step was to try a smaller model with only 4 conv layers. This shallow model has 690 000 trainable parameters, much less than the original VGG11 (See appendix section ii.). Following the hyperparameter search best batch: 256, and best learning rate were: 0.01. The model was then trained, and it reached optimal performance after 10 epochs.
This shallower network did a worse job and also overfits (Figure 6.4).

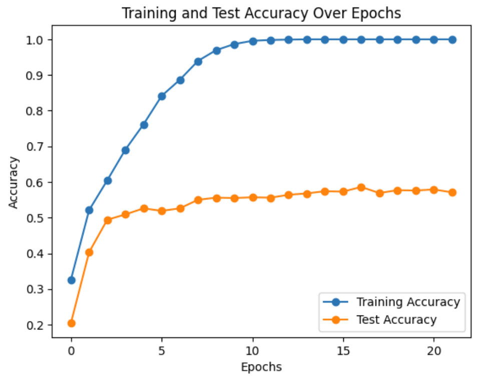


Figure 6.4. Train and validation curves for shallow CNN

Results:

- training accuracy: **98.58%**
- test accuracy**: 55.80%**
- Confusion Matrix (Figure 6.5)

- Classification report (Figure 6.6)

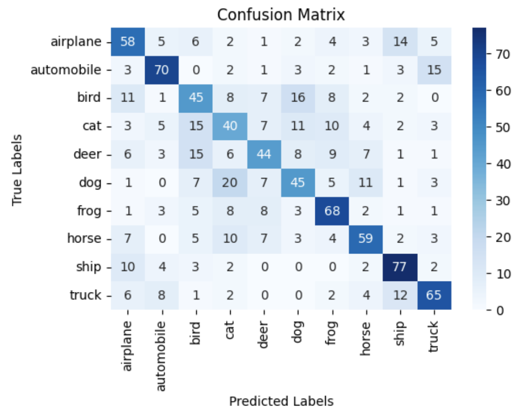
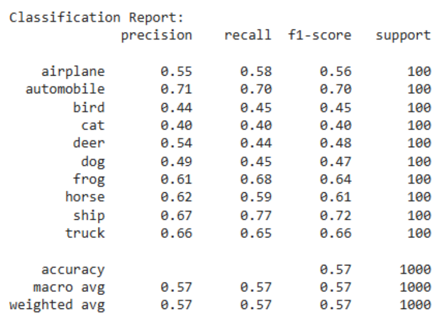


Figure 6.5. Shallow CNN Confusion Matrix



```
Classification Report:
              precision    recall  f1-score   support

    airplane       0.55      0.58      0.56       100
  automobile       0.71      0.70      0.70       100
        bird       0.44      0.45      0.45       100
         cat       0.40      0.40      0.40       100
        deer       0.54      0.44      0.48       100
         dog       0.49      0.45      0.47       100
        frog       0.61      0.68      0.64       100
       horse       0.62      0.59      0.61       100
        ship       0.67      0.77      0.72       100
       truck       0.66      0.65      0.66       100

    accuracy                           0.57      1000
   macro avg       0.57      0.57      0.57      1000
weighted avg       0.57      0.57      0.57      1000
```

Figure 6.6. Shallow CNN Classification report

Best classes again are Ship and automobile, and weaker classes cat and bird. The reduced parameter count (690,000) makes this model computationally efficient, but it has significantly more misclassification, and performance is much lower than VGG11.

6.3 Deeper CNN

Next, a deeper network is tried with 14 convolutional layers that has 33 million hyperparameters (see appendix iii. for model summary). The best batch size was 512, learning rate: 0.01 and the model achieved optimal results after 34 epochs. There is still an issue with overfitting (curve depicted in Figure 6.7).

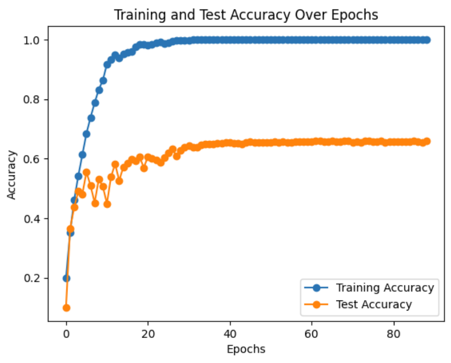


Figure 6.7. Deep CNN training and validation curve

Results:

- training accuracy: **100%**
- test accuracy: **65.9%**
- Confusion Matrix (Figure 6.8)
- Classification report (Figure 6.9)

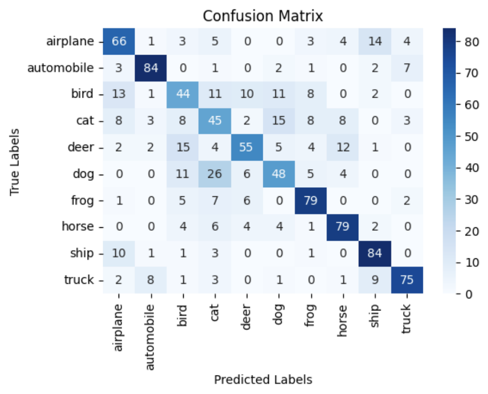
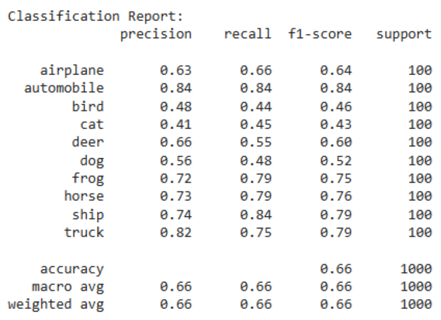


Figure 6.8. Deep CNN Confusion matrix



```
Classification Report:
              precision    recall  f1-score   support

    airplane       0.63      0.66      0.64       100
  automobile       0.84      0.84      0.84       100
        bird       0.48      0.44      0.46       100
         cat       0.41      0.45      0.43       100
        deer       0.66      0.55      0.60       100
         dog       0.56      0.48      0.52       100
        frog       0.72      0.79      0.75       100
       horse       0.73      0.79      0.76       100
        ship       0.74      0.84      0.79       100
       truck       0.82      0.75      0.79       100

    accuracy                           0.66      1000
   macro avg       0.66      0.66      0.66      1000
weighted avg       0.66      0.66      0.66      1000
```

Figure 6.9. Deep CNN Classification report

The deeper CNN achieves the highest test accuracy so far, but at the cost of increased computational resources and still significant overfitting. The deeper architecture allows the model to learn more detailed patterns, which slightly improves test accuracy compared to the shallow and base models. Strong classes are again automobile and ship and weaker bird and cat. The slight improvement in test accuracy compared to VGG-11 (65.9% vs. 61.6%) shows that additional depth provides marginal benefits but does not solve the overfitting issue.

6.4 Different kernel sizes

CIFAR-10 images are of size 32×32, which restricts the use of large kernel sizes, as larger kernels would reduce the spatial dimensions of feature maps too early in the network. To address this, images were upscaled to 64×64, allowing for larger kernel sizes while maintaining spatial resolution. Scaling images to 224×224 (commonly used in deep networks) was not feasible with larger batch sizes due to memory limitations. Prioritizing batch size over image scaling was preferred, as larger batch sizes generally improved training stability and generalization in previous steps. The model architecture was modified to include two convolutional layers with kernel sizes 5 and 7 at the beginning to capture broader spatial features, a deeper layer with kernel size 2 for finer feature extraction.

This model has a total of 8 convolutional layers, with approximately 32 million parameters (see appendix iv for model summary). Model was trained with best batch size of 256 and learning rate of 0.01 for 46 epochs.

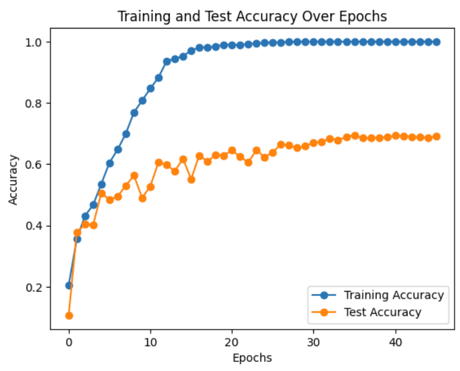
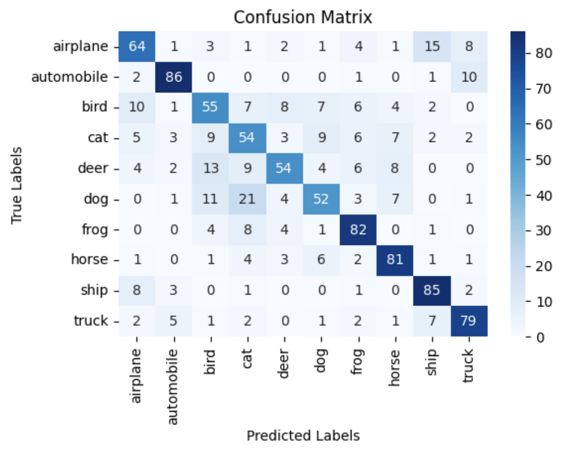


Figure 6.10. training and validation curve per epochs for a model with various kernel sizes CNN

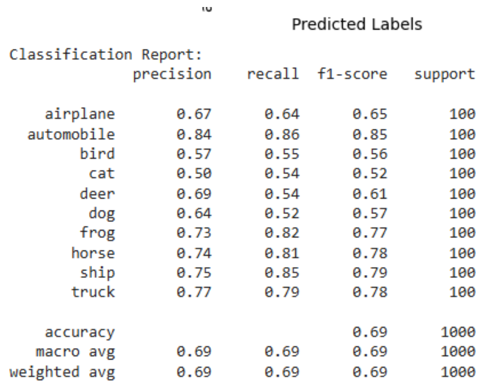
Results:

- training accuracy: **100.00%,**
- test accuracy: **69.5%**
- confusion matrix (Figure 6.11)
- classification report (Figure 6.12)



	airplane	automobile	bird	cat	deer	dog	frog	horse	ship	truck
airplane	64	1	3	1	2	1	4	1	15	8
automobile	2	86	0	0	0	0	1	0	1	10
bird	10	1	55	7	8	7	6	4	2	0
cat	5	3	9	54	3	9	6	7	2	2
deer	4	2	13	9	54	4	6	8	0	0
dog	0	1	11	21	4	52	3	7	0	1
frog	0	0	4	8	4	1	82	0	1	0
horse	1	0	1	4	3	6	2	81	1	1
ship	8	3	0	1	0	0	1	0	85	2
truck	2	5	1	2	0	1	2	1	7	79

Figure 6.11. Confusion Matrix for model with various kernel sizes CNN



```
Classification Report:
              precision    recall  f1-score   support

    airplane       0.67      0.64      0.65       100
  automobile       0.84      0.86      0.85       100
        bird       0.57      0.55      0.56       100
         cat       0.50      0.54      0.52       100
        deer       0.69      0.54      0.61       100
         dog       0.64      0.52      0.57       100
        frog       0.73      0.82      0.77       100
       horse       0.74      0.81      0.78       100
        ship       0.75      0.85      0.79       100
       truck       0.77      0.79      0.78       100

    accuracy                           0.69      1000
   macro avg       0.69      0.69      0.69      1000
weighted avg       0.69      0.69      0.69      1000
```

Figure 6.12. Classification report for model with various kernel sizes CNN

Despite overfitting (evident from 100% training accuracy and figure 6.10), the modifications demonstrate that larger kernels improve generalization to unseen data. Larger kernels in the initial layers (5 and 7) help capture broader spatial patterns, improving performance for classes with distinct and large features (e.g., Automobiles, Trucks).

The smaller kernel size (2) in deeper layers ensures finer details are captured, aiding class differentiation. It struggles still with Cat, Bird categories.

6.5 Grid Search and Hyperparameter Insights

The grid search for CNN hyperparameters tested batch sizes [128,256,512] and learning rates [0.01,0.001,0.0001] with most frequent **batch size 256** and higher and **learning rate 0.01** yielding the best performance. Larger batch sizes worked better due to the small image size (32×32 or 64×64), as they provided smoother gradient updates and improved feature distinction. Smaller batch sizes introduced more noise, leading to less stable training. The learning rate of 0.01 balanced convergence speed and stability, outperforming lower rates that slowed training. These results underscore the importance of tuning hyperparameters to align with dataset characteristics and resource constraints.

7. Conclusion

This project evaluated the performance of Naïve Bayes (NB), Decision Tree (DT), Multilayer Perceptron (MLP), and Convolutional Neural Networks (CNNs) on the CIFAR-10 dataset. NB and DT, which utilized ResNet-18 and PCA for feature extraction, demonstrated computational efficiency but struggled with complex patterns, achieving test accuracies of 79.4% and 60%, respectively. The MLP, leveraging the same feature extraction pipeline, emerged as the best-performing model overall, with an accuracy of 81.9%. The best MLP architecture was relatively shallow and 32 hidden units per layer. This result highlights the strength of ResNet-18's feature extraction capabilities, as the extracted features significantly simplified the classification task for the MLP.

CNNs, trained directly on image data, showcased the advantages of hierarchical feature learning. Among them, the modified CNN with larger kernels and upscaled images achieved the highest CNN test accuracy of 69.5%, outperforming VGG-11 and deeper CNN architectures. However, overfitting remained a challenge across all CNN models, reflecting the need for improved regularization, using the entire Cifar-10 dataset with twice as many images, and perhaps data augmentation.

Overall, the results highlight the trade-offs between model complexity, interpretability, and computational cost, with MLP proving most effective for pre-processed data and the modified CNN excelling on raw images. Future improvements could also focus on transfer learning.

The use of the test set as a validation set provided an effective way to monitor model performance during training. However, it also limits the ability to evaluate true generalization on unseen data. Future work should consider the inclusion of a dedicated test set for a more robust evaluation of model performance.

Final Words on Performance Comparison

Table 1 provides a summary of the performance metrics for all models evaluated in this project. It highlights the trade-offs between the simplicity of statistical models like Naïve Bayes and Decision Trees, the flexibility and power of MLPs, and the hierarchical feature learning capabilities of CNNs.

From the table, we observe that the MLP model achieved the highest test accuracy among all implementations, demonstrating its effectiveness with pre-processed feature vectors. Among the CNN models, the modified VGG with larger kernels performed the best showing the benefits of architectural adjustments. Decision trees performed the worst because they struggle with high dimensional data such as images. Despite the feature extraction, the data remained too complex to be captured effectively.

Table 1. Results and metrics of all models. Highlighted in pink is the row that showcases the best performing model

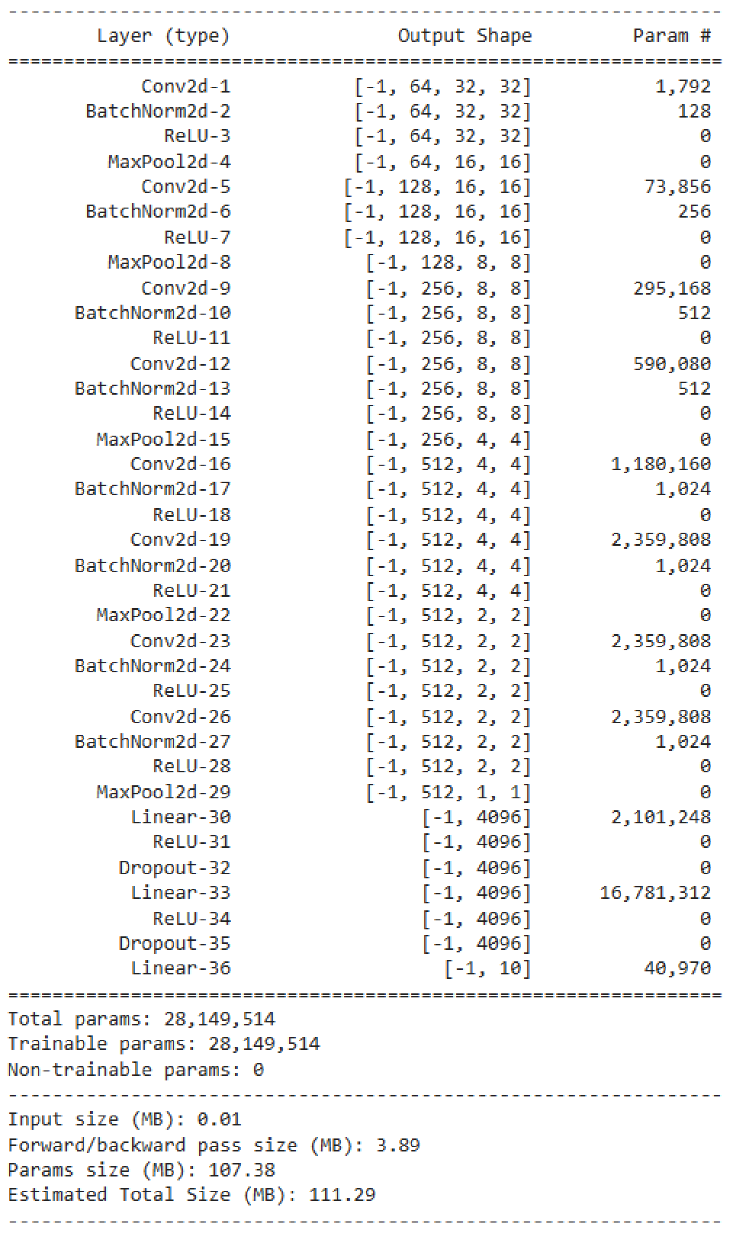
	Test accuracy	precision	recall	F1 score
Custom Naïve Bayes	79.40%	0.8	0.79	0.79
Sklearn BN	79.4%	0.8	0.79	0.79
Custom Decision Trees	60%	0.61	0.6	0.6
Sklearn decision trees	60.6%	0.61	0.6	0.6
MLP	81.9%	0.82	0.82	0.82
VGG11	61.6%	0.6	0.61	0.6
Shallow VGG	55.8%	0.57	0.57	0.57
Deep VGG	64.3%	0.64	0.64	0.64
VGG with kernel variations	69.5%	0.69	0.69	0.69

References

[1] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," presented at the International Conference on Learning Representations (ICLR), 2015.

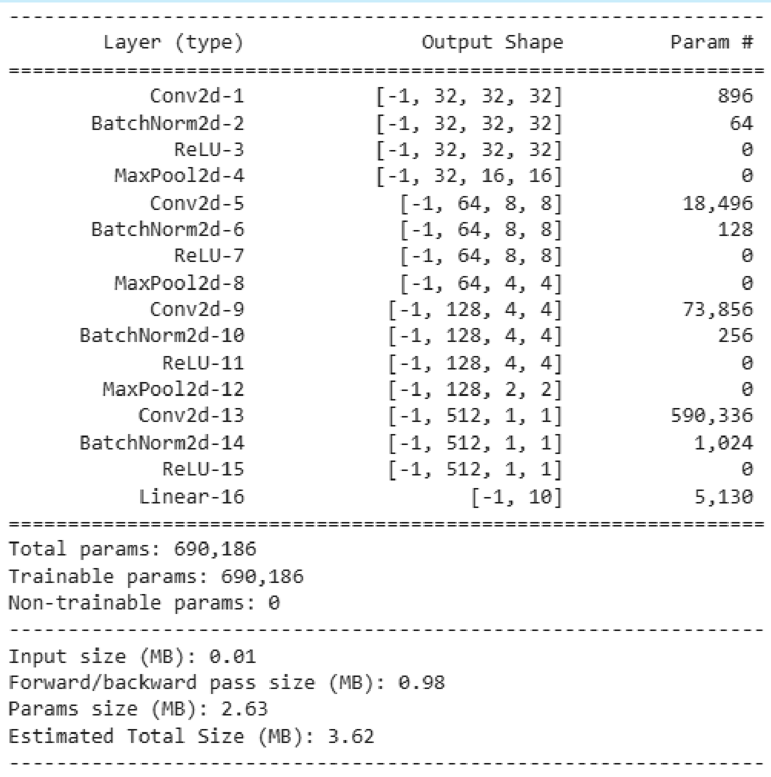
Appendix:

i. VGG11 Model summary



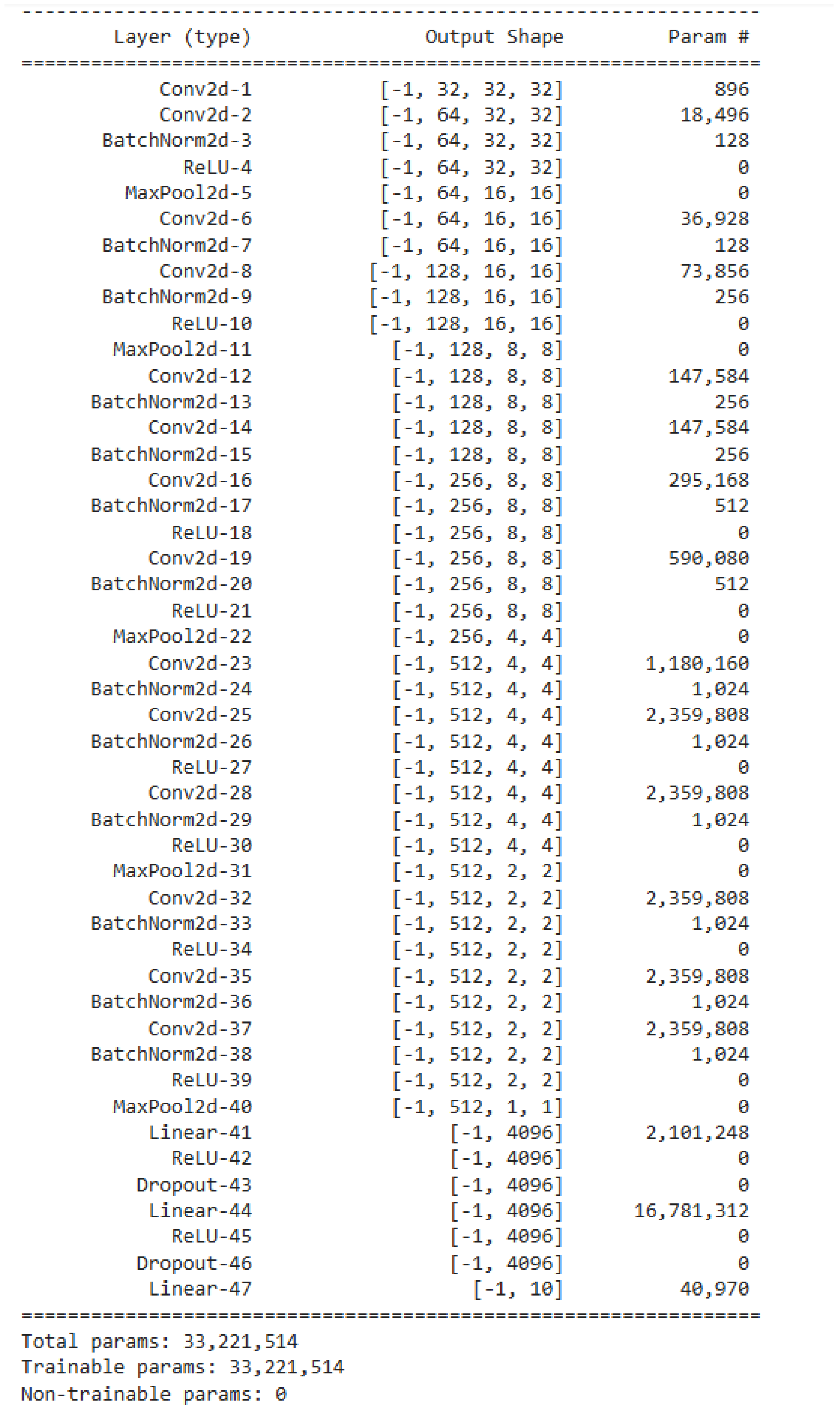
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 32, 32]           1,792
       BatchNorm2d-2           [-1, 64, 32, 32]             128
              ReLU-3           [-1, 64, 32, 32]               0
         MaxPool2d-4           [-1, 64, 16, 16]               0
            Conv2d-5          [-1, 128, 16, 16]          73,856
       BatchNorm2d-6          [-1, 128, 16, 16]             256
              ReLU-7          [-1, 128, 16, 16]               0
         MaxPool2d-8            [-1, 128, 8, 8]               0
            Conv2d-9            [-1, 256, 8, 8]         295,168
      BatchNorm2d-10            [-1, 256, 8, 8]             512
             ReLU-11            [-1, 256, 8, 8]               0
           Conv2d-12            [-1, 256, 8, 8]         590,080
      BatchNorm2d-13            [-1, 256, 8, 8]             512
             ReLU-14            [-1, 256, 8, 8]               0
        MaxPool2d-15            [-1, 256, 4, 4]               0
           Conv2d-16            [-1, 512, 4, 4]       1,180,160
      BatchNorm2d-17            [-1, 512, 4, 4]           1,024
             ReLU-18            [-1, 512, 4, 4]               0
           Conv2d-19            [-1, 512, 4, 4]       2,359,808
      BatchNorm2d-20            [-1, 512, 4, 4]           1,024
             ReLU-21            [-1, 512, 4, 4]               0
        MaxPool2d-22            [-1, 512, 2, 2]               0
           Conv2d-23            [-1, 512, 2, 2]       2,359,808
      BatchNorm2d-24            [-1, 512, 2, 2]           1,024
             ReLU-25            [-1, 512, 2, 2]               0
           Conv2d-26            [-1, 512, 2, 2]       2,359,808
      BatchNorm2d-27            [-1, 512, 2, 2]           1,024
             ReLU-28            [-1, 512, 2, 2]               0
        MaxPool2d-29            [-1, 512, 1, 1]               0
           Linear-30                 [-1, 4096]       2,101,248
             ReLU-31                 [-1, 4096]               0
          Dropout-32                 [-1, 4096]               0
           Linear-33                 [-1, 4096]      16,781,312
             ReLU-34                 [-1, 4096]               0
          Dropout-35                 [-1, 4096]               0
           Linear-36                   [-1, 10]          40,970
================================================================
Total params: 28,149,514
Trainable params: 28,149,514
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 3.89
Params size (MB): 107.38
Estimated Total Size (MB): 111.29
----------------------------------------------------------------
```

ii. Shallow CNN model summary (VGG_small in the notebook)



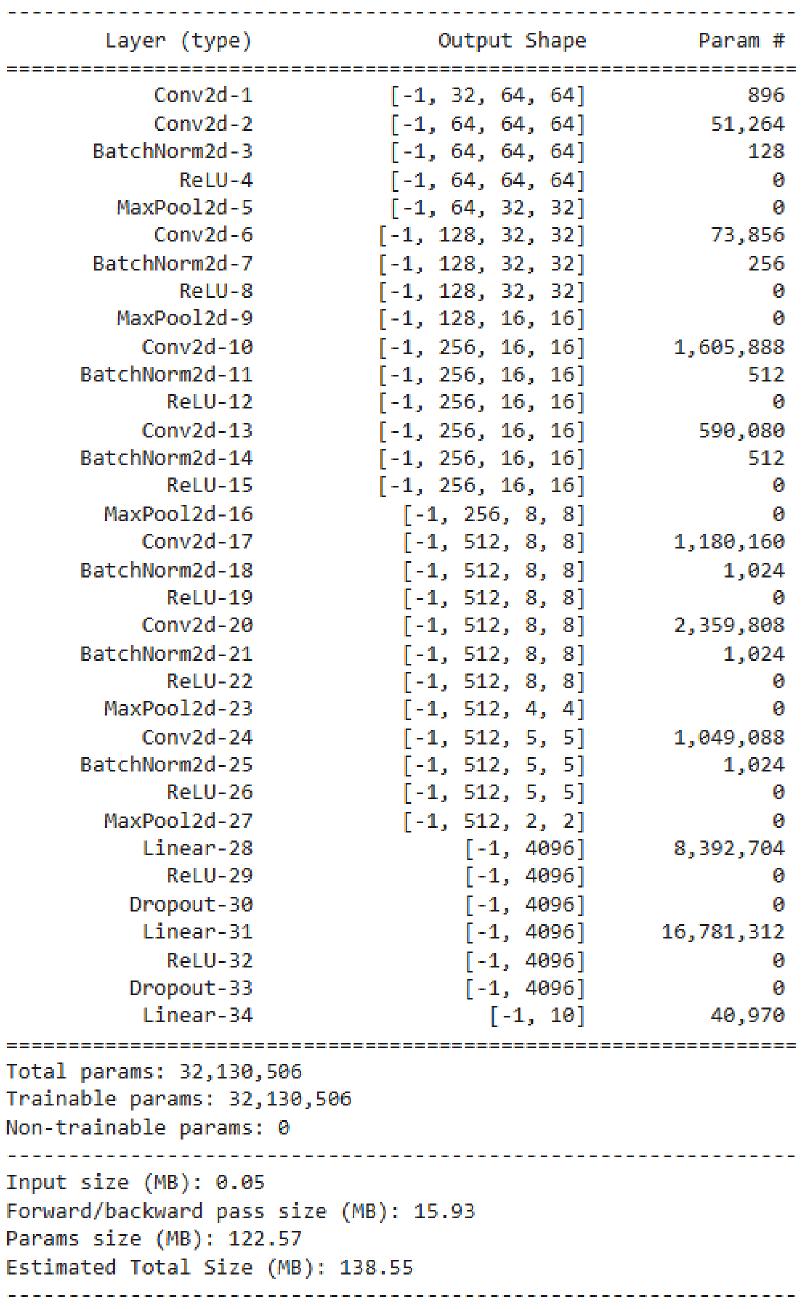
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             896
       BatchNorm2d-2           [-1, 32, 32, 32]              64
              ReLU-3           [-1, 32, 32, 32]               0
         MaxPool2d-4           [-1, 32, 16, 16]               0
            Conv2d-5             [-1, 64, 8, 8]          18,496
       BatchNorm2d-6             [-1, 64, 8, 8]             128
              ReLU-7             [-1, 64, 8, 8]               0
         MaxPool2d-8             [-1, 64, 4, 4]               0
            Conv2d-9            [-1, 128, 4, 4]          73,856
      BatchNorm2d-10            [-1, 128, 4, 4]             256
             ReLU-11            [-1, 128, 4, 4]               0
        MaxPool2d-12            [-1, 128, 2, 2]               0
           Conv2d-13            [-1, 512, 1, 1]         590,336
      BatchNorm2d-14            [-1, 512, 1, 1]           1,024
             ReLU-15            [-1, 512, 1, 1]               0
           Linear-16                   [-1, 10]           5,130
================================================================
Total params: 690,186
Trainable params: 690,186
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 0.98
Params size (MB): 2.63
Estimated Total Size (MB): 3.62
----------------------------------------------------------------
```

## iii.	Deep CNN (VGG_big in notebook) model summary



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]             896
            Conv2d-2           [-1, 64, 32, 32]          18,496
       BatchNorm2d-3           [-1, 64, 32, 32]             128
              ReLU-4           [-1, 64, 32, 32]               0
         MaxPool2d-5           [-1, 64, 16, 16]               0
            Conv2d-6           [-1, 64, 16, 16]          36,928
       BatchNorm2d-7           [-1, 64, 16, 16]             128
            Conv2d-8          [-1, 128, 16, 16]          73,856
       BatchNorm2d-9          [-1, 128, 16, 16]             256
             ReLU-10          [-1, 128, 16, 16]               0
        MaxPool2d-11            [-1, 128, 8, 8]               0
           Conv2d-12            [-1, 128, 8, 8]         147,584
      BatchNorm2d-13            [-1, 128, 8, 8]             256
           Conv2d-14            [-1, 128, 8, 8]         147,584
      BatchNorm2d-15            [-1, 128, 8, 8]             256
           Conv2d-16            [-1, 256, 8, 8]         295,168
      BatchNorm2d-17            [-1, 256, 8, 8]             512
             ReLU-18            [-1, 256, 8, 8]               0
           Conv2d-19            [-1, 256, 8, 8]         590,080
      BatchNorm2d-20            [-1, 256, 8, 8]             512
             ReLU-21            [-1, 256, 8, 8]               0
        MaxPool2d-22            [-1, 256, 4, 4]               0
           Conv2d-23            [-1, 512, 4, 4]       1,180,160
      BatchNorm2d-24            [-1, 512, 4, 4]           1,024
           Conv2d-25            [-1, 512, 4, 4]       2,359,808
      BatchNorm2d-26            [-1, 512, 4, 4]           1,024
             ReLU-27            [-1, 512, 4, 4]               0
           Conv2d-28            [-1, 512, 4, 4]       2,359,808
      BatchNorm2d-29            [-1, 512, 4, 4]           1,024
             ReLU-30            [-1, 512, 4, 4]               0
        MaxPool2d-31            [-1, 512, 2, 2]               0
           Conv2d-32            [-1, 512, 2, 2]       2,359,808
      BatchNorm2d-33            [-1, 512, 2, 2]           1,024
             ReLU-34            [-1, 512, 2, 2]               0
           Conv2d-35            [-1, 512, 2, 2]       2,359,808
      BatchNorm2d-36            [-1, 512, 2, 2]           1,024
           Conv2d-37            [-1, 512, 2, 2]       2,359,808
      BatchNorm2d-38            [-1, 512, 2, 2]           1,024
             ReLU-39            [-1, 512, 2, 2]               0
        MaxPool2d-40            [-1, 512, 1, 1]               0
           Linear-41                 [-1, 4096]       2,101,248
             ReLU-42                 [-1, 4096]               0
          Dropout-43                 [-1, 4096]               0
           Linear-44                 [-1, 4096]      16,781,312
             ReLU-45                 [-1, 4096]               0
          Dropout-46                 [-1, 4096]               0
           Linear-47                   [-1, 10]          40,970
================================================================
Total params: 33,221,514
Trainable params: 33,221,514
Non-trainable params: 0
```

iv. Kernel size variations CNN (VGGnet_kernel_var in notebook) model summary



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 64, 64]             896
            Conv2d-2           [-1, 64, 64, 64]          51,264
       BatchNorm2d-3           [-1, 64, 64, 64]             128
              ReLU-4           [-1, 64, 64, 64]               0
         MaxPool2d-5           [-1, 64, 32, 32]               0
            Conv2d-6          [-1, 128, 32, 32]          73,856
       BatchNorm2d-7          [-1, 128, 32, 32]             256
              ReLU-8          [-1, 128, 32, 32]               0
         MaxPool2d-9          [-1, 128, 16, 16]               0
           Conv2d-10          [-1, 256, 16, 16]       1,605,888
      BatchNorm2d-11          [-1, 256, 16, 16]             512
             ReLU-12          [-1, 256, 16, 16]               0
           Conv2d-13          [-1, 256, 16, 16]         590,080
      BatchNorm2d-14          [-1, 256, 16, 16]             512
             ReLU-15          [-1, 256, 16, 16]               0
        MaxPool2d-16            [-1, 256, 8, 8]               0
           Conv2d-17            [-1, 512, 8, 8]       1,180,160
      BatchNorm2d-18            [-1, 512, 8, 8]           1,024
             ReLU-19            [-1, 512, 8, 8]               0
           Conv2d-20            [-1, 512, 8, 8]       2,359,808
      BatchNorm2d-21            [-1, 512, 8, 8]           1,024
             ReLU-22            [-1, 512, 8, 8]               0
        MaxPool2d-23            [-1, 512, 4, 4]               0
           Conv2d-24            [-1, 512, 5, 5]       1,049,088
      BatchNorm2d-25            [-1, 512, 5, 5]           1,024
             ReLU-26            [-1, 512, 5, 5]               0
        MaxPool2d-27            [-1, 512, 2, 2]               0
           Linear-28                 [-1, 4096]       8,392,704
             ReLU-29                 [-1, 4096]               0
          Dropout-30                 [-1, 4096]               0
           Linear-31                 [-1, 4096]      16,781,312
             ReLU-32                 [-1, 4096]               0
          Dropout-33                 [-1, 4096]               0
           Linear-34                   [-1, 10]          40,970
================================================================
Total params: 32,130,506
Trainable params: 32,130,506
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 15.93
Params size (MB): 122.57
Estimated Total Size (MB): 138.55
----------------------------------------------------------------
```