BBM 418 - Computer Vision Laboratory

Assignment 2 Report

Ahmet İkbal Kayapınar
2210356043

PART 1

Architecture

Introduction: The architecture consists of two main parts. The first part is the features block, which primarily handles feature extraction and contains convolution layers. The second part is the classifier block, which mainly performs classification tasks and includes linear neural network layers. The parametric details of each layer and the connections between layers can be seen in Figure 1.

```
# channel widths for each conv
C1, C2, C3, C4, C5 = 32, 64, 128, 256, 256

self.features = nn.Sequential(
    # — Conv 1 — 256x256 → 256x256
    nn.Conv2d(3,  C1, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(C1),       # ← single BatchNorm
    nn.ReLU(),

    # — Conv 2 — 256x256 → 128x128
    nn.Conv2d(C1, C2, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),

    # — Conv 3 — 128x128 → 128x128
    nn.Conv2d(C2, C3, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),

    # — Conv 4 — 128x128 → 128x128
    nn.Conv2d(C3, C4, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),

    # — Conv 5 — 128x128 → 128x128
    nn.Conv2d(C4, C5, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),

    # — Single Pool — 128x128 → 16x16
    nn.MaxPool2d(kernel_size=8),
)
self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(C5 * 16 * 16, 256),
    nn.ReLU(),
    nn.Linear(256, num_classes),
)
```

(a) Layers

```
def forward(self, x):
    x = self.features(x)
    x = self.classifier(x)
    return x
```

(b) Connections

Figure 1: The parametric details of each layer and the connections between layers

Features Block

The features block consists of five convolution layers in sequence, followed by activation functions, a batch normalization layer, and a max pooling layer.

Convolution Layers: The first convolution layer uses a kernel size of 5 to operate on larger regions and make the work of the subsequent layers easier. The remaining layers use a kernel size of 3. The second layer uses a stride value of 2 in order to control computational complexity and reduce dimensions early on, effectively halving the dimensions from this point onward. In line with common practice, the number of filters increases throughout the convolution layers, aiming to produce more accurate values by the final stage.

Batch Normalization Layer: Right after the first convolution layer, a batch normalization layer is placed at the beginning of the block to normalize abnormal values and to enhance the impact of normalization on subsequent computations.

Max Pooling Layer: There is only one pooling layer used, with a kernel size of 8, placed at the very end of the block

to reduce the large dimensions before passing to the final block. This placement minimizes the negative impact on features.

Classifier Block

This block includes a flatten layer followed by two linear neural network layers.

Flatten: Reduces the input from the previous block to a one-dimensional array.

Linear: These are fully connected neural network layers. The input feature size of the first layer matches the output from the previous block. The output feature size is set to 256 neurons, considering the overall model size. The output size of the second layer equals the number of classes in the dataset.

Functions

The optimizer and loss functions used in the code can be seen in Figure 2.

```
# Define the loss function
criterion = nn.CrossEntropyLoss()
# Define the optimizer
optimizer = optim.AdamW(model.parameters(), lr=learning_rate, weight_decay=1e-6)
```

Figure 2: The optimizer and loss functions

Activation Function: The ReLU activation function is used. There are two main reasons for this choice. First, it is as simple as possible in terms of computational complexity. Second, compared to the sigmoid function, it handles the vanishing gradient problem more effectively.

Loss Function: The commonly used and well-proven loss function for classification tasks, cross-entropy loss, is employed. It offers smoother gradients than other loss functions, which leads to better convergence during training.

Optimizer: The AdamW algorithm is used as the optimizer. Adam is known for its ability to adjust the learning rate for each parameter individually, and its bias-correction mechanism contributes to faster convergence and more stable optimization. AdamW is an improved version of Adam, as it decouples weight decay from the gradient update step. Instead of adding the weight decay to the loss function, it applies it directly during the parameter update, resulting in more consistent regularization and better generalization.

Residual Connection

The version of the model with a residual connection fundamentally contains the same layers. It includes an additional residual connection at the last convolution layer, taking advantage of the fact that the input and output dimensions of this layer are equal. The model details can be seen in Figure 3.

```python
C1, C2, C3, C4, C5 = 32, 64, 128, 256, 256

self.first_block = nn.Sequential(
    # — Conv 1 — 256x256 → 256x256
    nn.Conv2d(3,  C1, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(C1),       # ← single BatchNorm
    nn.ReLU(),

    # — Conv 2 — 256x256 → 128x128
    nn.Conv2d(C1, C2, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),

    # — Conv 3 — 128x128 → 128x128
    nn.Conv2d(C2, C3, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),

    # — Conv 4 — 128x128 → 128x128
    nn.Conv2d(C3, C4, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
)
self.second_block = nn.Sequential(
    # — Conv 5 — 128x128 → 128x128
    nn.Conv2d(C4, C5, kernel_size=3, stride=1, padding=1),
)
self.third_block = nn.Sequential(
    nn.ReLU(),

    # — Single Pool — 128x128 → 16x16
    nn.MaxPool2d(kernel_size=8),
)
self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(C5 * 16 * 16, 256),
    nn.ReLU(),
    nn.Linear(256, num_classes),
```

(a) Layers

```python
def forward(self, x):
    x = self.first_block(x)
    out = self.second_block(x)
    out = out + x
    out = self.third_block(out)
    out = self.classifier(out)
    return out
```

(b) Connections

Figure 3: Model with residual connection

Data Augmentation

Due to the small size of the dataset and the observed accuracy values, extensive transformations were applied to improve prediction performance. The details can be seen in Figure 4.

```python
transform_train = transforms.Compose([
    # Geometric transforms (PIL-based)
    transforms.RandomResizedCrop(256, scale=(0.7, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(p=0.3),
    transforms.RandomRotation(30),

    # Color transforms (PIL-based)
    transforms.ColorJitter(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.1),

    # Conversion to tensor
    transforms.ToTensor(),

    # Tensor-based operations
    transforms.Normalize(mean=mean, std=std),
])
```

Figure 4: Data transformations

Tests and Tuning

Accuracy Calculation

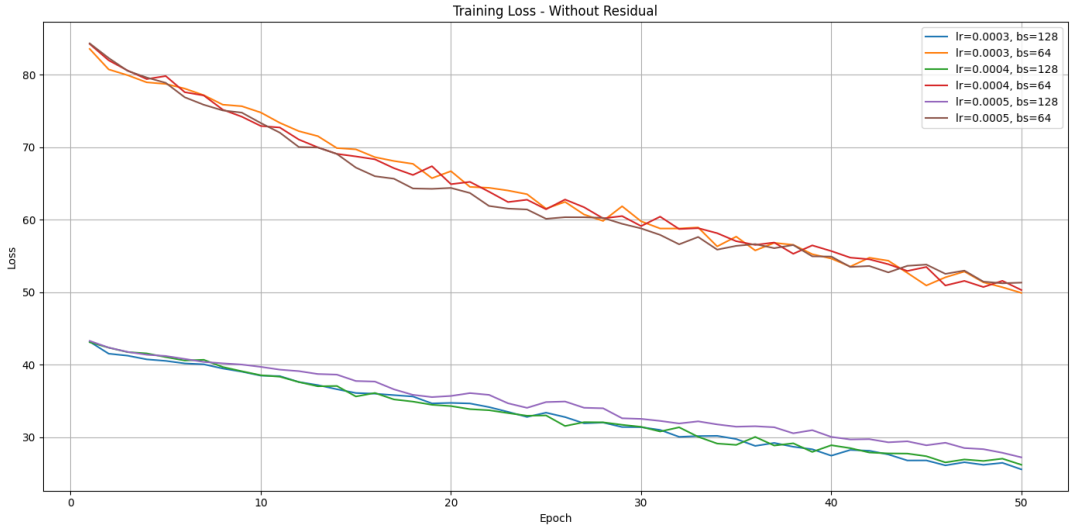
During model training, after the training step is completed in each epoch, the model is switched to evaluation mode and the entire validation dataset is used for prediction. These predictions are then compared with the true labels to calculate accuracy. In some tests, the same process was also applied to the test dataset alongside validation. The corresponding code snippet can be seen in Figure 5.

```python
# Validation
model.eval()
val_correct, val_total, test_correct, test_total = 0, 0, 0, 0
with torch.no_grad():
    for images, labels in val_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)

        _, predicted = torch.max(outputs, 1)
        val_total += labels.size(0)
        val_correct += (predicted == labels).sum().item()

    val_accs.append(100 * val_correct / val_total)
```

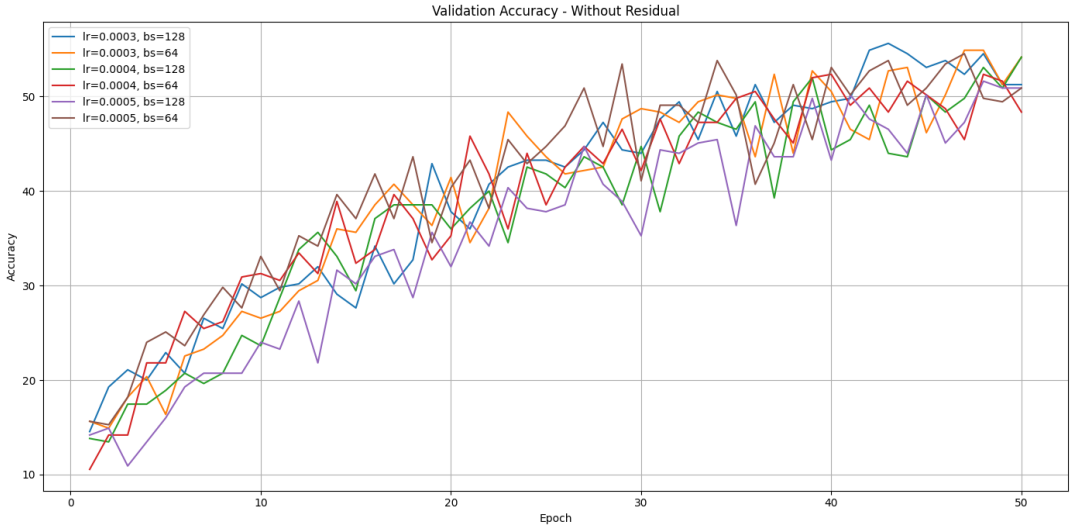
Figure 5: Accuracy calculation

Parameter Testing

Based on the results in Figures 6 and 7, the optimal parameters were determined to be a batch size of 64 and a learning rate of 0.0003 for the model without a residual connection, and a batch size of 64 with a learning rate of 0.0004 for the residual model. The test accuracies of the best-performing models are shown in Figure 8.

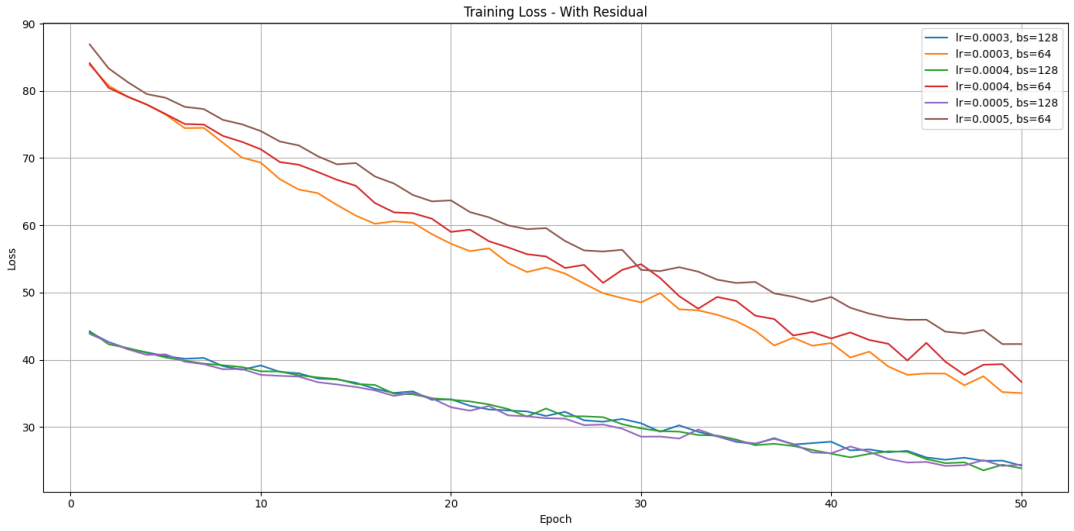


(a) Training Losses

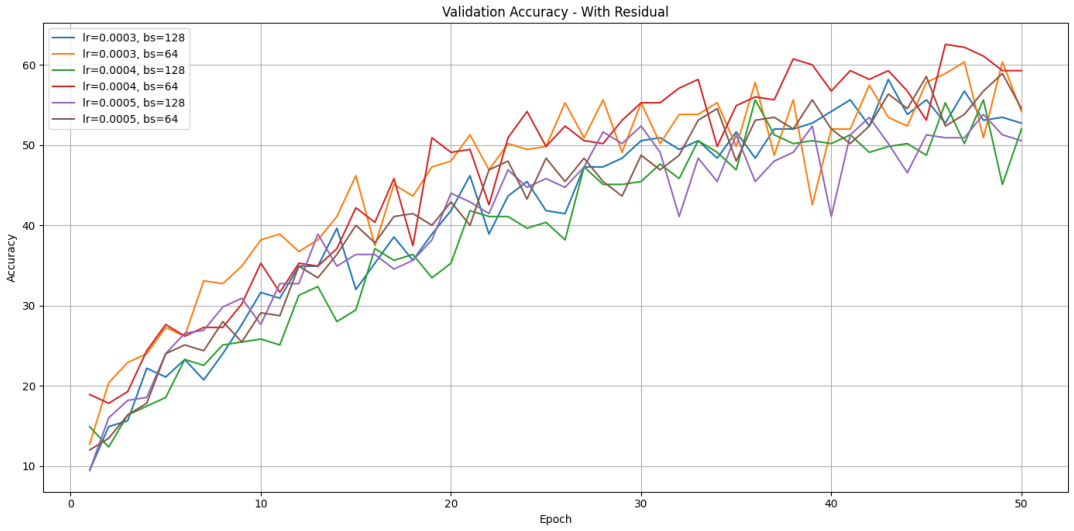


(b) Validation Accuracies

Figure 6: Graphs of the model without residual connection



(a) Training Losses



(b) Validation Accuracies

Figure 7: Graphs of the model with residual connection

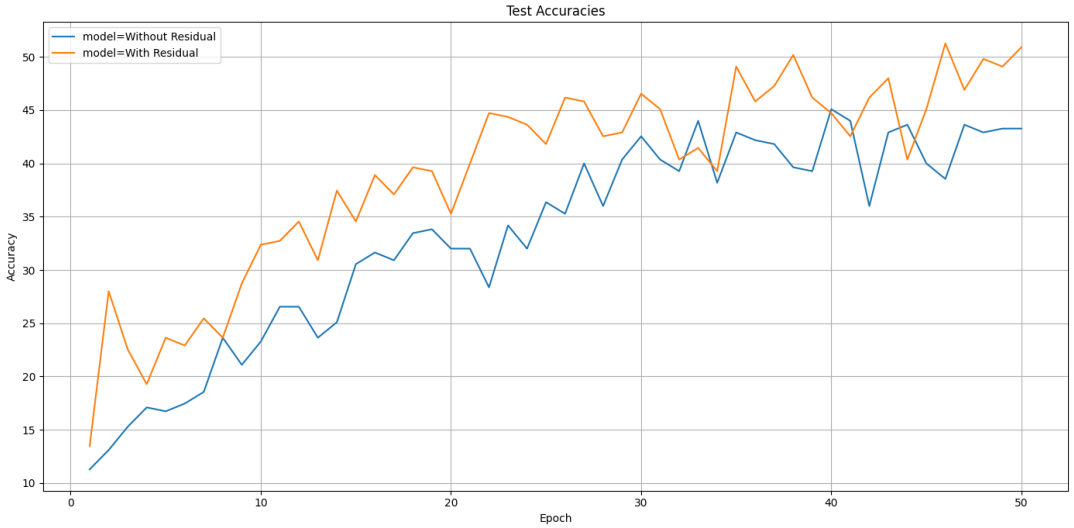


Figure 8: Test accuracies of the best models

Dropout Integration

Architecture

Due to the limited size of the dataset and the model's initial tendency to overfit, the primary purpose of using dropout was to achieve regularization and prevent overfitting. Therefore, the dropout layer was placed right before the final linear layer. This placement was chosen to maximize the regularization benefit. The models with dropout added can be seen in Figure 9.

```
self.features = nn.Sequential(
    # — Conv 1 — 256x256 → 256x256
    nn.Conv2d(3,  C1, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(C1),       # ← single BatchNorm
    nn.ReLU(),

    # — Conv 2 — 256x256 → 128x128
    nn.Conv2d(C1, C2, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),

    # — Conv 3 — 128x128 → 128x128
    nn.Conv2d(C2, C3, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),

    # — Conv 4 — 128x128 → 128x128
    nn.Conv2d(C3, C4, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),

    # — Conv 5 — 128x128 → 128x128
    nn.Conv2d(C4, C5, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),

    # — Single Pool — 128x128 → 16x16
    nn.MaxPool2d(kernel_size=8),
)
self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(C5 * 16 * 16, 256),
    nn.ReLU(),
    nn.Dropout(p),
    nn.Linear(256, num_classes),
)
```

(a) Model without residual connection

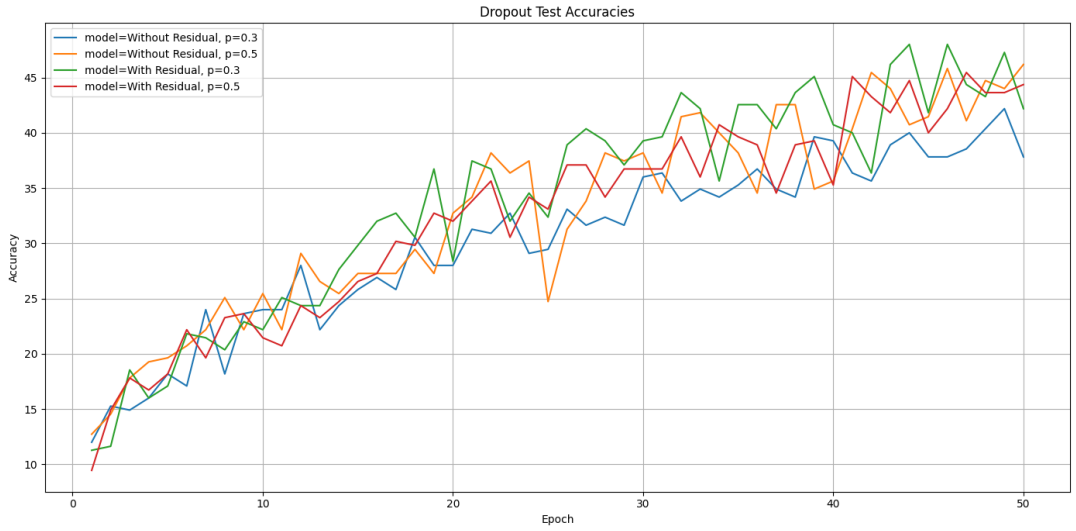
```
self.first_block = nn.Sequential(
    # — Conv 1 — 256x256 → 256x256
    nn.Conv2d(3,  C1, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(C1),       # ← single BatchNorm
    nn.ReLU(),

    # — Conv 2 — 256x256 → 128x128
    nn.Conv2d(C1, C2, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),

    # — Conv 3 — 128x128 → 128x128
    nn.Conv2d(C2, C3, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),

    # — Conv 4 — 128x128 → 128x128
    nn.Conv2d(C3, C4, kernel_size=3, stride=1, padding=1),
    nn.ReLU(),
)
self.second_block = nn.Sequential(
    # — Conv 5 — 128x128 → 128x128
    nn.Conv2d(C4, C5, kernel_size=3, stride=1, padding=1),
)
self.third_block = nn.Sequential(
    nn.ReLU(),

    # — Single Pool — 128x128 → 16x16
    nn.MaxPool2d(kernel_size=8),
)
self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(C5 * 16 * 16, 256),
    nn.ReLU(),
    nn.Dropout(p),
    nn.Linear(256, num_classes),
)
```

(b) Model with residual connection

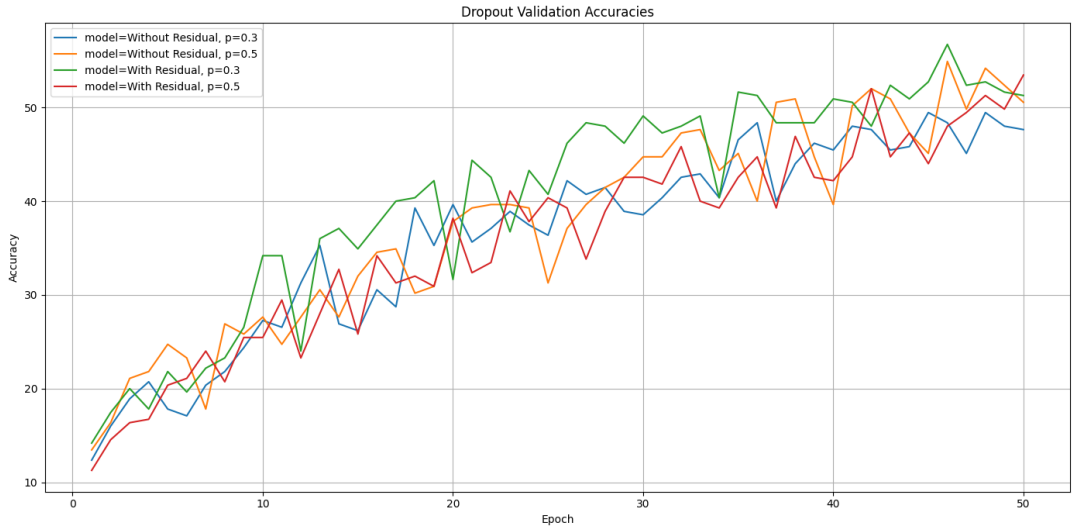
Figure 9: Layers of the models with dropout layer

Testing

The models with a dropout layer were tested with p values of 0.3 and 0.5. Overall, instead of improvements, there was a decline in accuracy. The details can be seen in Figure 10.



(a) Test Accuracies



(b) Validation Accuracies

Figure 10: Test results with dropout

Analysis

Under the optimal hyperparameters—batch size of 64 with a learning rate of 0.0003 for the baseline model and batch size of 64 with a learning rate of 0.0004 for the residual variant—the model with the residual connection consistently outperformed the plain convolutional network. As shown in Figures 6 and 7, the residual architecture converged faster (lower training losses) and achieved higher validation accuracies across epochs . This uplift in performance carried through to the test set, where the residual model exhibited a modest but meaningful gain in generalization (Figure 8).

Data augmentation (Figure 4) proved essential for combating overfitting given the limited dataset size. Geometric transformations (random rotations, flips) and photometric adjustments (intensity jitter, color shifts) yielded smoother validation curves and reduced variance between runs, demonstrating improved robustness . By contrast, introducing a dropout layer before the final classifier (Figure 9) did not confer additional benefits: both p=0.3 and p=0.5 settings led to a drop in validation and test accuracies (Figure 10), indicating that the combination of data augmentation and residual connections was sufficient to regulate model capacity without dropout.

The confusion matrix of the best-performing residual model (Figure 11) highlights strong per-class performance, with the vast majority of predictions falling along the diagonal. A handful of off-diagonal entries point to residual confusion between visually similar classes, suggesting that targeted augmentations or customized loss functions could help further disambiguate these challenging cases.

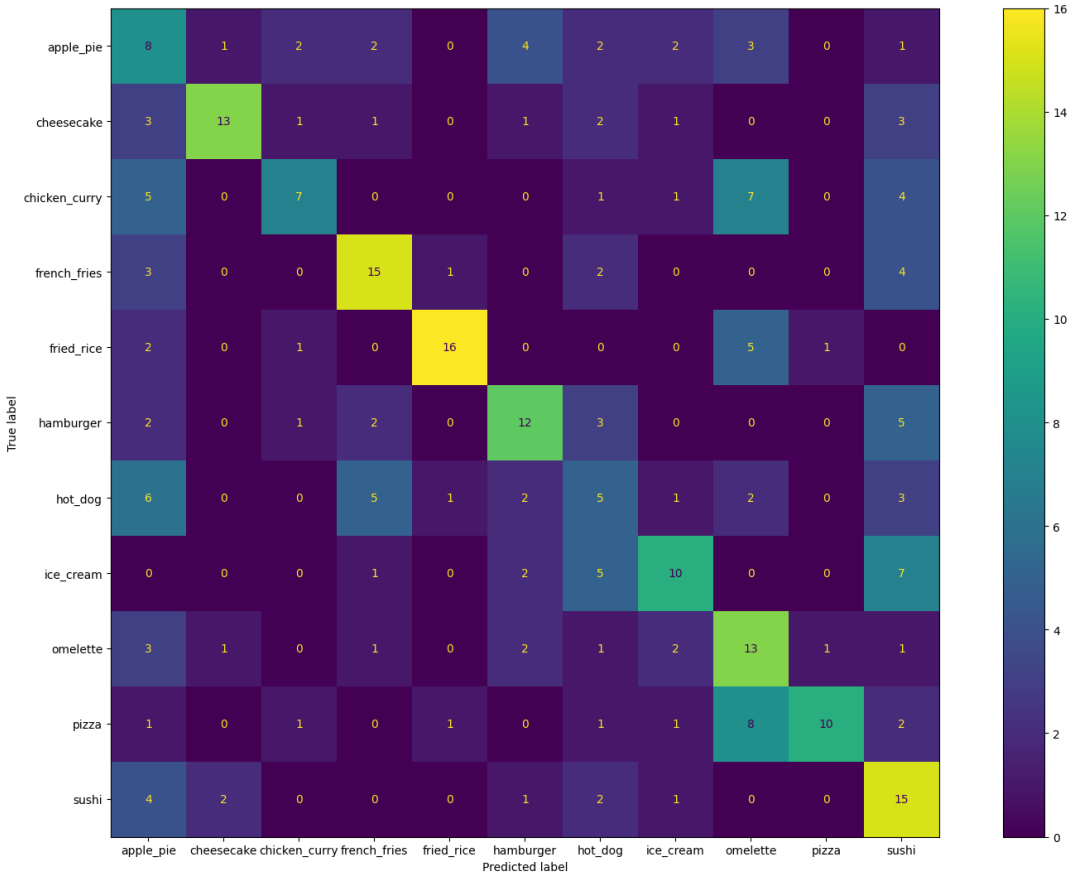


Figure 11: Confusion matrix of the best model

PART 2

Explanation of Fine-Tuning

Fine-tuning is the process of adapting a pre-designed and pre-trained model to a new dataset by retraining some of its layers.

If the dataset to be used is not large enough, training deep and large models from scratch would not be efficient. In such cases, large models that have been trained on similar datasets can be used through fine-tuning. This also helps save computational resources.

In fine-tuning, typically the model is frozen except for the fully connected layer and, optionally, some of the final convolution layers, which are retrained. As CNN models pass data from deeper to shallower layers, they transition from extracting local features to more global features. Therefore, the deeper layers, which extract local features, can be reused thanks to the universal nature of such features. The fully connected layer, however, is responsible for classification and must be retrained according to the classes in the current dataset.

Implementation

In this project, the MobileNetV3 model was used for fine-tuning. In Figure 12, you can see the case where all layers except the fully connected layer are frozen. In Figure 13, the case where the fully connected layer along with the last two convolution blocks are also unfrozen is shown.

```python
def get_mobilenetv3_case1(num_classes):
    """
    Loads MobileNetV3 and freezes all layers except the final fully-connected layer.
    """
    # Load pretrained model
    model = models.mobilenet_v3_small(weights=models.MobileNet_V3_Small_Weights.DEFAULT)

    # Freeze all parameters
    for param in model.parameters():
        param.requires_grad = False

    # Replace the classifier's final Linear layer to match num_classes
    # The classifier is nn.Sequential([... , Linear(in_features, 1000)])
    in_features = model.classifier[3].in_features
    model.classifier[3] = nn.Linear(in_features, num_classes)

    # Unfreeze only the new FC layer
    for param in model.classifier.parameters():
        param.requires_grad = True

    return model
```

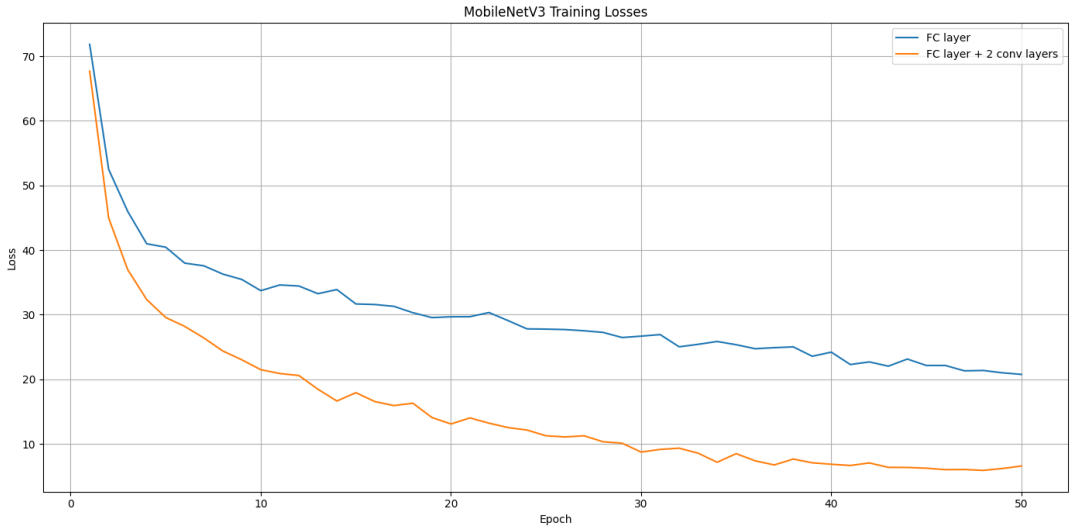
Figure 12: Only FC is trained

```python
def get_mobilenetv3_case2(num_classes):
    """
    Loads MobileNetV3 and freezes all layers except the last two convolutional blocks and the final FC layer.
    """
    # Load pretrained model
    model = models.mobilenet_v3_small(weights=models.MobileNet_V3_Small_Weights.DEFAULT)

    # Freeze all parameters
    for param in model.parameters():
        param.requires_grad = False

    # Identify and unfreeze the last two convolutional blocks
    # model.features is an nn.Sequential of blocks; we take the last two
    for block in list(model.features.children())[-2:]:
        for param in block.parameters():
            param.requires_grad = True

    # Replace the classifier's final Linear layer
    in_features = model.classifier[3].in_features
    model.classifier[3] = nn.Linear(in_features, num_classes)

    # Unfreeze classifier parameters
    for param in model.classifier.parameters():
        param.requires_grad = True

    return model
```

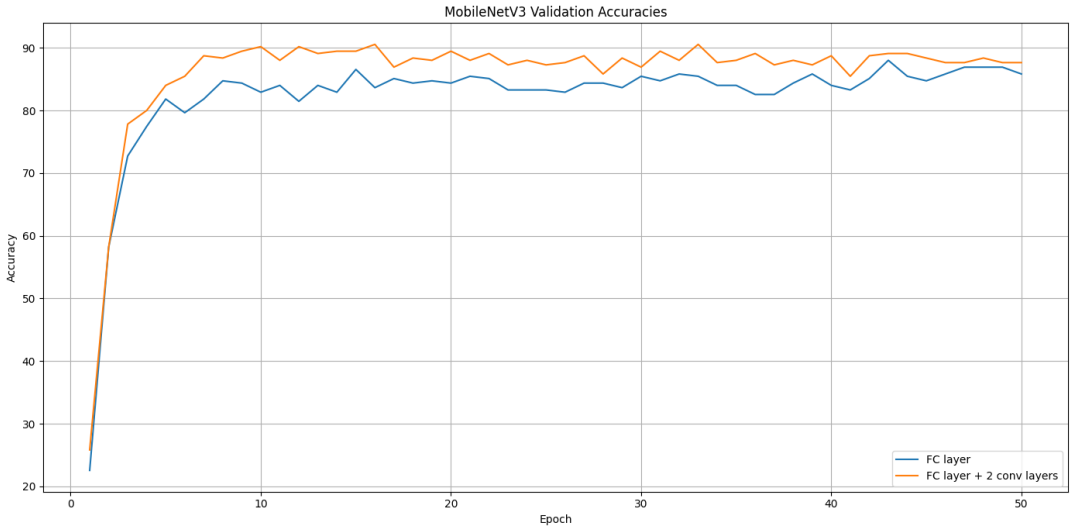
Figure 13: FC + last 2 convolutions are trained

Testing

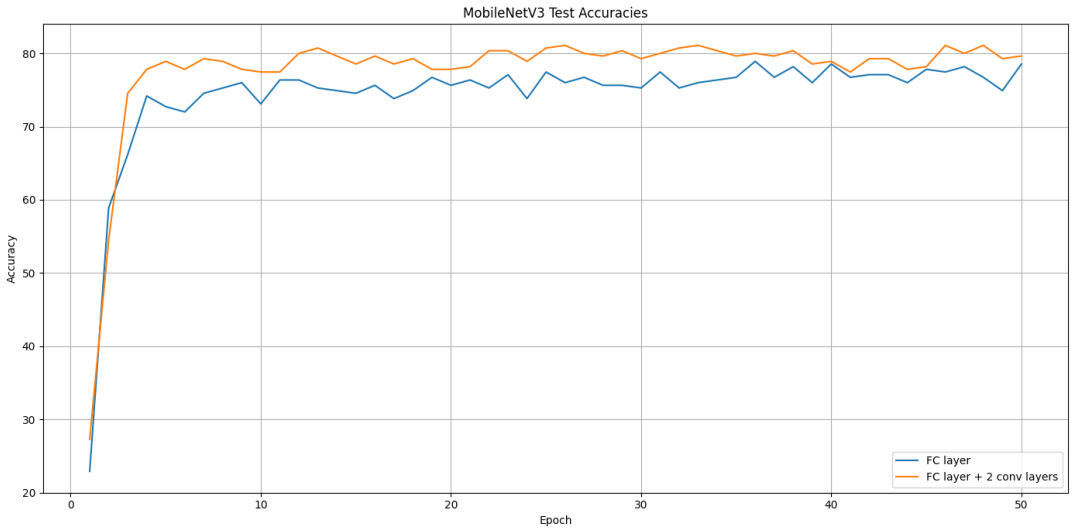
According to all results, the model in which the fully connected layer and the last two convolution blocks were retrained performed better, as shown in Figure 14. This suggests that retraining only the classifier layer yielded weaker results, likely due to the niche nature of the dataset.



(a) Training Losses



(b) Validation Accuracies



(c) Test Accuracies

Figure 14: Test results of MobileNetV3

Analysis

he fine-tuning process yielded highly successful results, largely due to the model being trained on a very large dataset and having a very deep architecture. It achieved particularly strong performance, reaching around 90% accuracy on the validation dataset and around 80% on the test dataset—values that are difficult to achieve in typical computer vision tasks. The confusion matrix of the best-performing model can be seen in Figure 15.

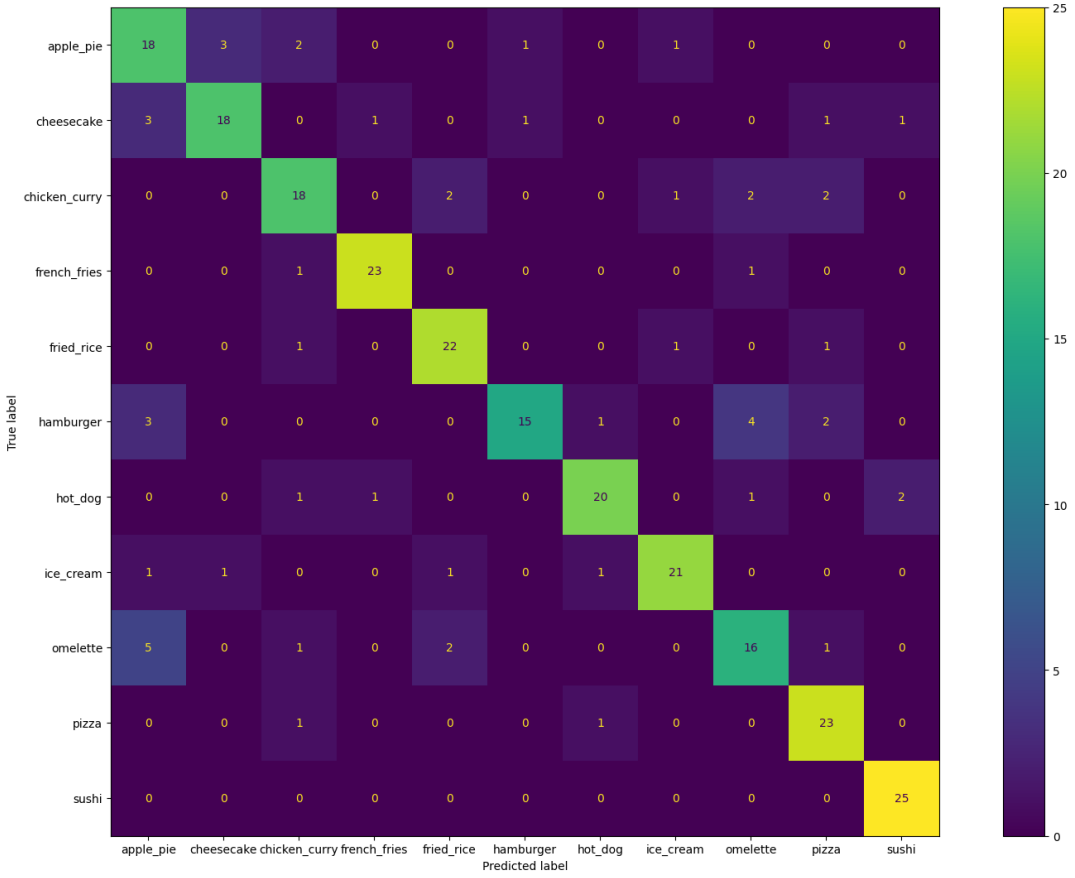


Figure 15: Confusion matrix of the MobileNetV3

Conclusion

In Part 1, our custom CNN architecture—enhanced with a residual connection—demonstrated clear benefits in both convergence speed and final accuracy under properly tuned hyperparameters. Extensive data augmentation was critical for generalization, while dropout proved unnecessary when combined with residual links and augmentations. Confusion-matrix analysis identified specific class pairs that remain challenging, guiding future efforts toward class-specific augmentation or loss designs.

Part 2's fine-tuning of MobileNetV3 further underscored the power of transfer learning: by retraining the fully connected layer and the last two convolutional blocks, we achieved approximately 90% validation accuracy and 80% test accuracy on our niche dataset. Future work could explore deeper residual blocks, advanced augmentation strategies (e.g., mixup, CutMix), ensemble techniques, and specialized regularization approaches to continue improving performance on this task.