BBM 418 - Computer Vision Laboratory

Assignment 2 Report

Ömer Kayra Çetin
2210356060

PART 1 - Modeling and Training a CNN classifier from Scratch

Data Augmentation

Various data augmentation techniques are applied to improve generalization and reduce overfitting. These include:

- **RandomResizedCrop**: Randomly crops the image and resizes it to 128×128. It generates spatial invariance.
- **RandomHorizontalFlip**: Randomly flips the images left-right for variation.
- **RandomAffine**: Applies small affine translations to better capture camera perspective changes.
- **ColorJitter**: Randomly changes brightness, contrast, saturation, and hue to introduce color variation.
- **Normalization**: Standardizes input to zero mean and unit variance with `mean=[0.5]*3` and `std=[0.5]*3`.

```python
train_transform = transforms.Compose([
    transforms.RandomResizedCrop(128, scale=(0.8, 1.0), ratio=(0.9, 1.1)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomAffine(degrees=10, translate=(0.1, 0.1), scale=(0.9, 1.1)),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue=0.02),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5]*3, std=[0.5]*3)
])

val_test_transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5]*3, std=[0.5]*3)
])
```

Figure 1: Data augmentation.

SimpleCNN and ResidualCNN Implementation

Two CNN models are implemented from scratch using pytorch: a simple cnn and a residual cnn. They both use convolutional layers to extract features from input images and fully connected layers for classification. The second one also uses residual connections to enhance the performance.

Convolutional Layers

Convolutional layers are implemented by using "nn.Conv2d". Each convolutional layer is followed with Batch Normalization and ReLU activation in both of the models. Batch Normalization is used for normalizing the output of a layer to stabilize it. ReLU is used because it is fast, it introduces non-linearity into the model and helps avoiding the vanishing gradient problem. Max pooling with kernel 2 and stride 2 is used after certain layers to gradually reduce the size of an image ($128 \rightarrow 64 \rightarrow 32 \rightarrow 16 \rightarrow 8$).

Convolutional layers are chosen with kernel size 3, stride 1 and padding 1. Kernel size 3 is chosen because it captures local patterns effectively while also being computationally efficient. Stride 1 and padding 1 is chosen to keep the spatial size same. This way, downsampling is controlled explicitly by max pooling. Input and output chanels are chosen with a $3 \rightarrow 32 \rightarrow 64 \rightarrow 128 \rightarrow 256 \rightarrow 256$ approach. This follows a standard CNN design pattern. As the spatial resolution decreases due to max pooling, channel depth increases to keep computational cost reasonable while increasing feature richness.

```python
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=11, use_dropout=False, dropout_rate=0.5):
        super(SimpleCNN, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 128 → 64

            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 64 → 32

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 32 → 16

            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 16 → 8

            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU()
        )
```

Figure 2: SimpleCNN convolutional layers.

Fully-Connected Layers

After the convolutional feature extraction, the 3D features are flattened to 1D vectors. Then these vectors are given to two fully connected layers for classification. There is an optional dropout layer before the final fully connected layer to reduce overfitting. The input to the first layer is the output of the last convolutional layer, in this case 256 filters with 8x8 sizes. The final output is the number of the classes, in this case it's 11.

```python
# Classifier with optional dropout
classifier_layers = [
    nn.Flatten(),
    nn.Linear(256 * 8 * 8, 512),
    nn.ReLU()
]

if use_dropout:
    classifier_layers.append(nn.Dropout(dropout_rate))
classifier_layers.append(nn.Linear(512, num_classes))

self.classifier = nn.Sequential(*classifier_layers)
```

Figure 3: SimpleCNN fully-connected layers.

Residual Connections

The ResidualCNN model includes residual blocks in addition to SimpleCNN. The input is allowed to bypass one or more layers using a shortcut connection thanks to residual blocks that enables the network to learn residual functions. This is useful for training deeper networks effectively by preventing the vanishing gradient problem.

In my implementation, each residual block does two convolution operations and adds the original input back before getting through the activation function. Each convolution operation is still followed with batch normalization and ReLU activation layer as it is in SimpleCNN. There is also a option to downsample the image which makes the stride of the first convolution operation 2, effectively downsampling the by half.

```
def forward(self, x):
    identity = self.shortcut(x)
    out = self.relu(self.bn1(self.conv1(x)))
    out = self.bn2(self.conv2(out))
    out += identity
    out = self.relu(out)
    return out
```

Figure 4: Residual block operations.

If the input and output dimensions don't match (when downsampling), a 1x1 convolution is used in the shortcut path to ensure that the identity (shortcut) connection has the same shape as the output of the residual block, which allows element-wise addition.

```
self.downsample = downsample
if downsample or in_channels != out_channels:
    self.shortcut = nn.Sequential(
        nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=stride),
        nn.BatchNorm2d(out_channels)
    )
else:
    self.shortcut = nn.Identity()
```

Figure 5: Residual block shortcut.

In my ResidualCNN implementation, two residual blocks are used after the initial convolution layer.

```
class ResidualCNN(nn.Module):
    def __init__(self, num_classes=11, use_dropout=False, dropout_rate=0.5):
        super(ResidualCNN, self).__init__()

        self.initial = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)  # 128 → 64
        )

        self.layer1 = nn.Sequential(
            ResidualBlock(32, 64, downsample=True),  # 64 → 32
        )

        self.layer2 = nn.Sequential(
            ResidualBlock(64, 128, downsample=True),  # 32 → 16
        )

        self.layer3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)  # 16 → 8
        )

        self.layer4 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),  # No pooling
            nn.BatchNorm2d(256),
            nn.ReLU()
        )
```

Figure 6: ResidualCNN with residual blocks.

Loss Function and Optimization Algorithm

CrossEntrophyLoss is chosen as the loss function because this is a multi-classification problem. It is well-suited for such a task because it combines a softmax activation with negative log-likelihood loss. It allows direct comparison with the model's predicted probability distribution with the true class label.
Adam optimizer is chosen as the optimization algorithm because it is efficient and robust in deep learning tasks. It adjusts the learning rates for each parameter individually using estimates of first and second moments of the gradients. This leads to faster convergence and improved stability. It also needs less learning rate tuning.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=lr)
```

Figure 7: Loss function and optimization algorithm.

Evaluation of the Models with Different Batch Sizes and Learning Rates

Both of the models are evaluated with batch sizes 64 and 128 and learning rates 0.001, 0.0005 and 0.0001. Batch sizes are chosen in that way to explore the trade-off between accuracy and efficiency. Small batch sizes introduce more noise and generalize better whereas bigger ones lead to smoother gradient estimates with better GPU utilization. Learning rates are the typical ones used with the Adam optimizer. Trying multiple values can help choosing the best setup for effective learning. Both of the models are trained with 50 epochs.

```
best_model_data_simple_cnn = train_and_evaluate_model(
    model_class=SimpleCNN,
    train_dataset=train_dataset,
    val_dataset=val_dataset,
    class_names=class_names,
    batch_sizes=[64, 128],
    learning_rates=[0.001, 0.0005, 0.0001],
    num_epochs=50,
    device=device,
    title="Simple CNN"
)
```

Figure 8: SimpleCNN trained with different batch sizes and learning rates.

```
best_model_data_residual_cnn = train_and_evaluate_model(
    model_class=ResidualCNN,
    train_dataset=train_dataset,
    val_dataset=val_dataset,
    class_names=class_names,
    batch_sizes=[64, 128],
    learning_rates=[0.001, 0.0005, 0.0001],
    num_epochs=50,
    device=device,
    title="Residual CNN"
)
```

Figure 9: ResidualCNN trained with different batch sizes and learning rates.

Validation Accuracies of the Models

Validation accuracies for the two models after training 50 epochs with different batch sizes and learning rates are shown below. Batch sizes are in the rows and learning rates are in the columns. Train loss and accuracy change graphs of these models can be found in the jupyter notebook, it is decided to only show the best model's graphs in the report to make it easier to read.

	0.001	**0.0005**	**0.0001**
64	0.5964	0.6182	0.5818
128	0.5273	0.5818	0.5782

Table 1: Validation accuracies for SimpleCNN.

	0.001	0.0005	0.0001
64	0.6073	0.6073	0.5673
128	0.5673	0.5709	0.5782

Table 2: Validation accuracies for ResidualCNN.

Selecting the Best Models

Looking at the results of the valudation accuracy results, we can see that the best model for SimpleCNN is with 64 batch size and 0.0005 learning rate value achieving 0.6182 accuracy. When we look at ResidualCNN, we can see that the best model is with 64 batch size and 0.001 learning rate achieving 0.6073 accuracy. Their loss and accuracy graphs is shown below.

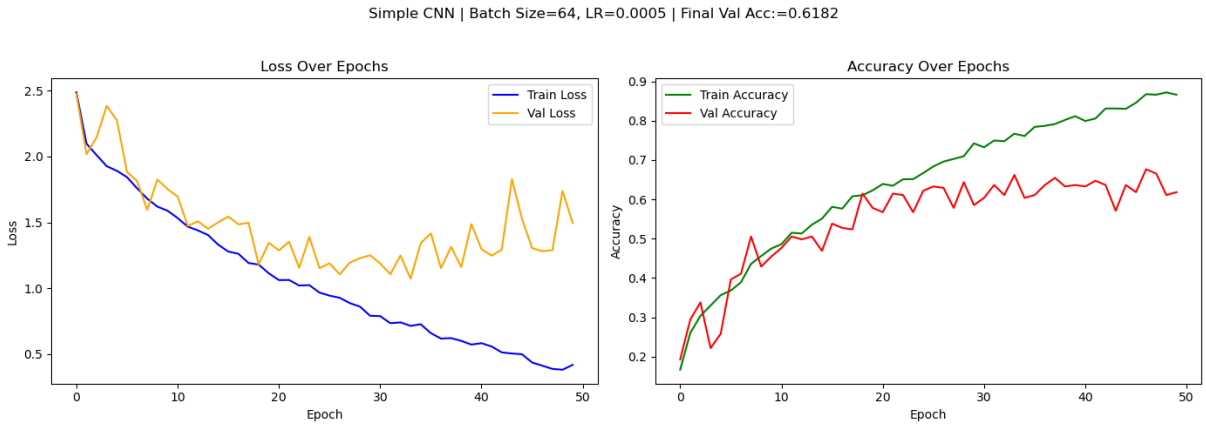

Figure 10: Loss-Accuracy Graph for best SimpleCNN.

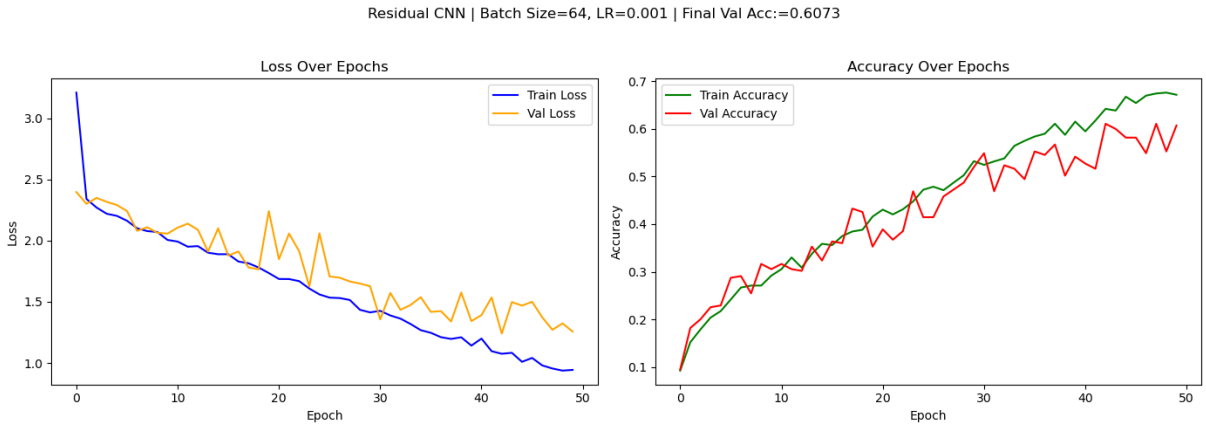

Figure 11: Loss-Accuracy Graph for best ResidualCNN.

Test accuracy results for the best models are:

- SimpleCNN : 0.5527
- ResidualCNN: 0.5018

Integrating Dropout to the Best Models

In both of the models, dropout is added after the first layer of the fully connected layer, before the final classification layer. The reason for that is dropout is most effective in reducing overfitting in dense layers. In these kind of layers, model has high number of parameters and has a tendency to memorize the training data. By applying dropout in that place, we aim to reduce the overfitting by introducing generalization through reducing the reliance on specific neuron activations.

The best models are tried with two different dropout parameters 0.3 and 0.5. The validation accuracy table is given below. The first column is for SimpleCNN and the second is for ResidualCNN. The results will be discussed in "Results and Findings" section.

Dropout Rate	SimpleCNN	ResidualCNN
0.3	0.6036	0.4109
0.5	0.5891	0.2400

Table 3: Validation accuracy for different dropout rates in SimpleCNN and ResidualCNN.

Loss and accuracy graphs of the best dropout models can be seen below.

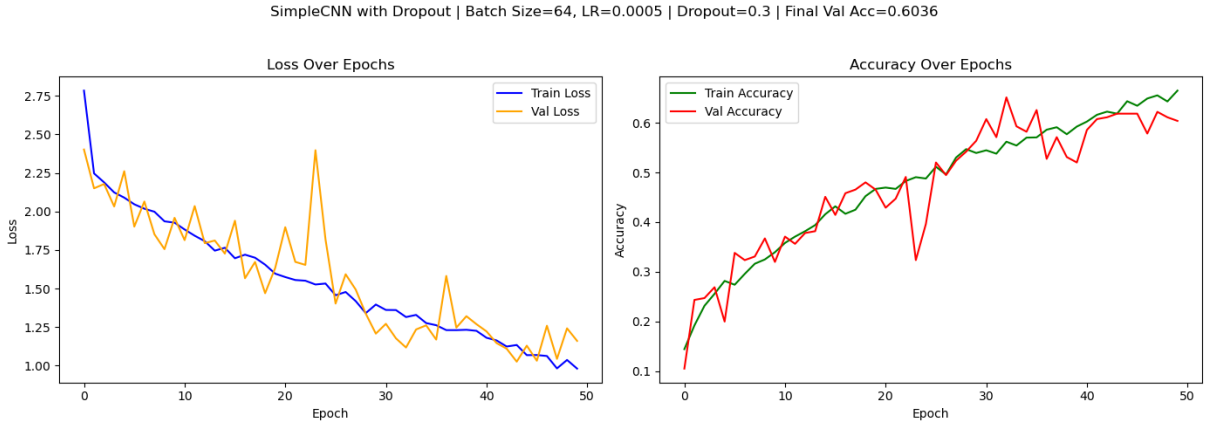

Figure 12: Loss-Accuracy Graph for best dropout SimpleCNN.

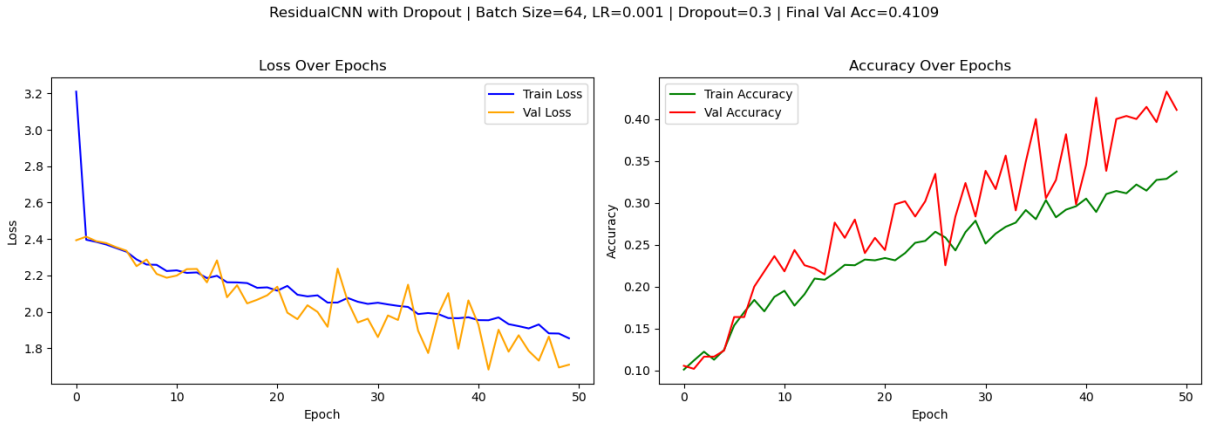

Figure 13: Loss-Accuracy Graph for best dropout ResidualCNN.

Test accuracy results for the best dropout models are:

- SimpleCNN with dropout 0.3: 0.4909
- ResidualCNN with dropout 0.3: 0.3782

Confusion Matrices for Best Dropout Models

Confusion matrices of best dropout models can be seen below.

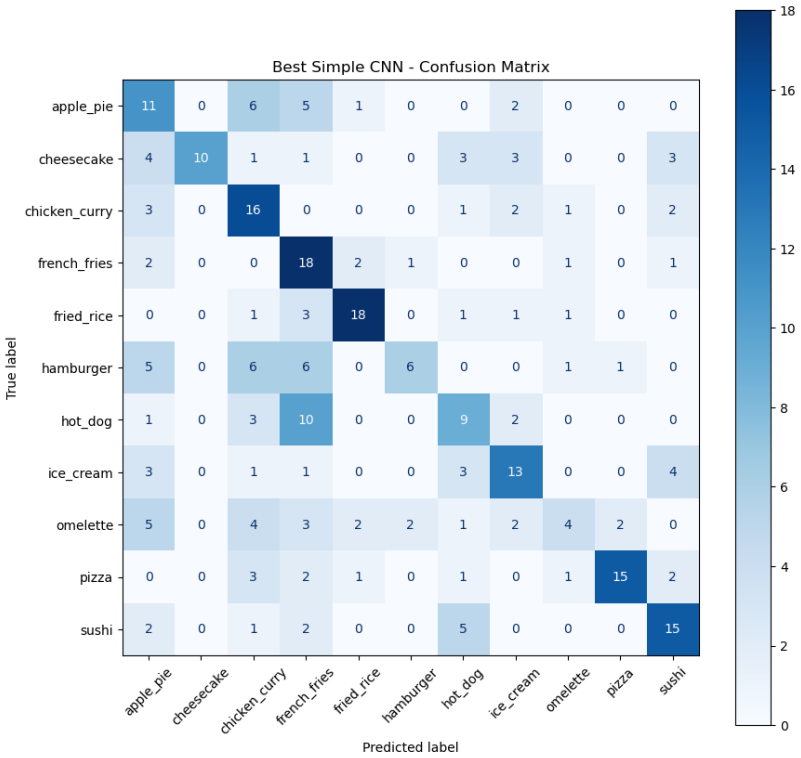

Figure 14: Confusion Matrix for best dropout SimpleCNN.

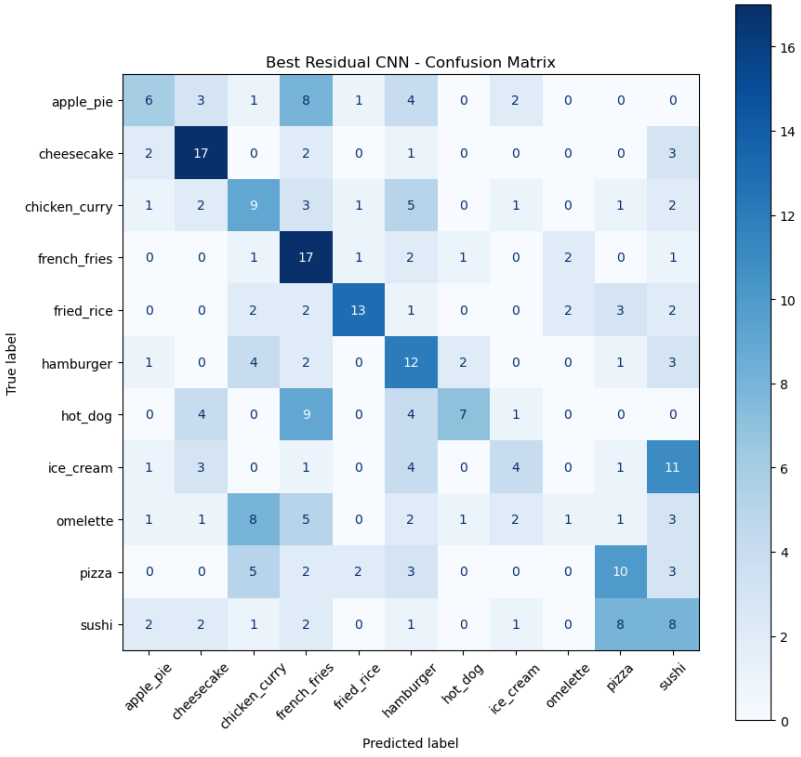

Figure 15: Confusion Matrix for best dropout ResidualCNN.

Results and Findings

The results show that batch size and learning rate can have significant effect on the learning process. In general and in our case as well, a smaller batch size works better as it includes more noise and generalizes the model better avoiding overfitting. Among the learning rates tested, 0.0005 gave the best result for SimpleCNN and 0.001 gave the

best result for ResidualCNN. We can understand that different models could have different optimal learning rates which they should be tuned for.

Comparing the results between SimpleCNN and ResidualCNN, we can see that SimpleCNN performed slightly better in the best model. Although, there were not that much of a meaningful difference when we look at it all together. This is most likely due to our models not being deep enough for residual connections to make a meaningful difference.

Introducing dropout didn't actually lead to a improvement with SimpleCNN, it more or less stayed the same with respect to validation accuracy. The reason for that is SimpleCNN were not overfitting in the first place thanks to data augmentations that were applied to the training data. That's why trying to reduce overfitting with dropouts didn't change much. Suprisingly, dropout had a significant negative effect on ResidualCNN. This indicates that the residual connections already introduced a layer of regularization and further trying to regularize the learning process led to a disruption in the feature learning all together. That means using dropouts are only meaningful when overfitting really applies to the model.

Test accuracies for both models were generally lower than their corresponding validation accuracies. This is because hyperparameters were tuned using the validation data and could lead to an optimistic estimation of generalization. It also highlights the challenges of achieving consistent generalization on unseen data.

PART 2 - Transfer Learning with CNNs

MobileNetV3

MobileNetV3 is chosen out of the three models that were given. The model offers a great balance between performance and computational efficiency. It outperforms its previous model MobileNetV3 in both speed and accuracy on ImageNet. These characteristics makes it a good source to apply transfer learning.

Fine-Tuning Explanation

What is Fine-Tuning?

Fine-tuning is a transfer learning method where a pretrained model on a large dataset (in this case "ImageNet") is taken and adapted to a new, typically a smaller dataset (in our case "food11"). Instead of training from scratch, we use the learned features of the pretrained model and only change some parts of it (usually the final layers) to make it viable for the new classification task.

Why Should We Do This?

A large dataset is usually needed to train a model from scratch and it uses significant computational resources. Pretrained models are already fairly effective at capturing features such as textures and edges. Using fine-tuning we can adapt to a task faster and using much smaller datasets.

Why Freeze the Rest and Train Only FC Layers?

Layers before the FC layers contain generic visual features useful for most vision tasks. Also by freezing the rest, we reduce the number of trainable parameters which prevents overfitting on a small dataset. We train the FC layer because that's where the classification happens. Only the final layer is specific to the task, so we can replace and train only that layer.

```
# Load pre-trained MobileNetV3
model = models.mobilenet_v3_large(pretrained=True)

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

Figure 16: Load and freeze all layers of the model.

```python
# Enable training for FC layer
in_features = model.classifier[3].in_features
model.classifier[3] = nn.Linear(in_features, len(class_names))

# Optionally unfreeze last two conv blocks
if mode == 'last_two_blocks':
    for layer in list(model.features.children())[-2:]:
        for param in layer.parameters():
            param.requires_grad = True
```

Figure 17: Freeze the FC layer and optinionally freeze the last two conv. layers.

Evaluation Results

Two different models are trained with two different fine-tuning approaches. The first model's only the FC layer is trained and the rest is freezed. The second model's FC layer with the last two convolutional layers are trained and the rest is freezed. While training, batch size of 64, learning rate of 0.001 and epoch value of 30 is used in both of the models. While training, the best state of the model is hold according to validation accuracy and used to evaluate the test accuracy.

Metric	**FC Only**	**FC + 2 Conv Layers**
Last Validation Accuracy	0.6909	0.7564
Best Validation Accuracy	0.7673	0.8073
Test Accuracy	0.6182	0.6545

Table 4: Validation and Test Accuracies for Different Fine-Tuning Strategies

When we analyze the results, we can see that training additional convolutional layers from the last layers of the model leads to an improved performance. It performs better than FC only in both validation accuracy and test accuracy. However, we can see that in both of the models, test accuracy is notable lower than validation accuracy. This indicates that there is some degree of overfitting to the validation set when tuning hyperparameters. It highlights the importance of the generalization techniques.

```python
results_fc_only = train_mobilenetv3_transfer(
    train_dataset=train_dataset,
    val_dataset=val_dataset,
    test_dataset=test_dataset,
    class_names=class_names,
    mode='fc_only',
    batch_size=64,
    learning_rate=0.001,
    num_epochs=30,
    device=device
)
```

Figure 18: Fine-tune FC-only model.

```python
results_last_two = train_mobilenetv3_transfer(
    train_dataset=train_dataset,
    val_dataset=val_dataset,
    test_dataset=test_dataset,
    class_names=class_names,
    mode='last_two_blocks',
    batch_size=64,
    learning_rate=0.001,
    num_epochs=30,
    device=device
)
```

Figure 19: Fine-tune FC + 2 Conv. model.

Confusion Matrix of FC + Last 2 Conv. Model

When we analyze the confusion matrix, we can see that model makes the most mistakes with apple pie, chicken curry and omelette. These types of classes are probably confused with similar-looking dishes, most likely due to colors and textures overlap. This highlights the model's difficulty in being capable of separating foods with similar appearances. The model might need more diversity or dense data for such classes.

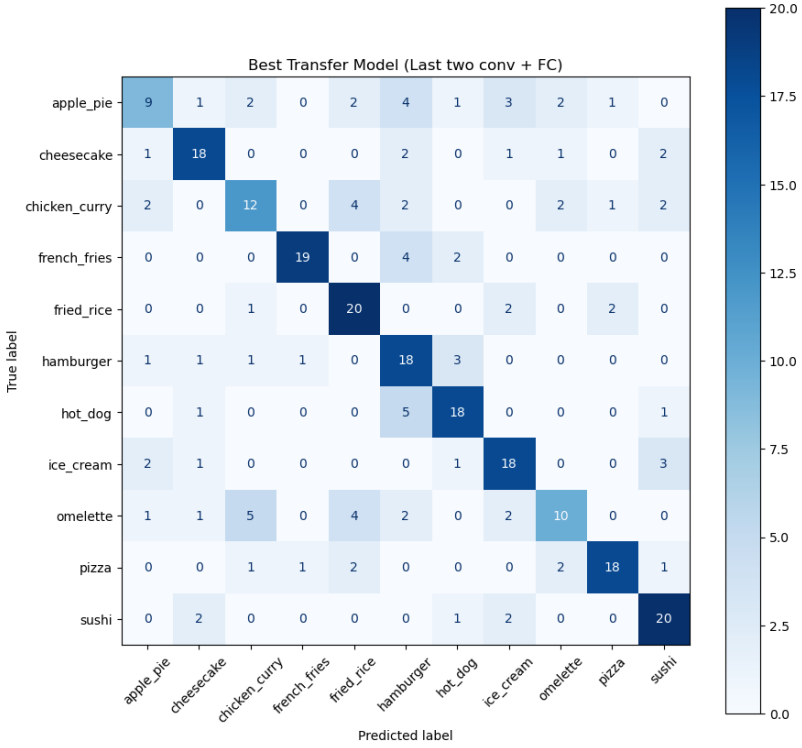

Figure 20: Confusion Matrix of FC + Last 2 Conv. Model.

Comparing and Analyzing Results in Part 1 and Part 2

Both of the models trained with transfer learning (FC only and FC + 2 Conv.) is better than all of the models trained in Part 1 in both validation accuracies and test accuracies. This proves that fine-tuning pretrained models usually lead to better results when using a small dataset such as food11. The models in Part 1 is not deep enough to capture all of the important features and there is not enough data to let them learn how to capture them effectively. The convolutional layers that are used from MobileNetV3 made a huge impact as it is trained on a large dataset (ImageNet) and it is fairly successful on capturing features such as textures, edges and complex patterns that are

common across many visual categories, which helps the model generalize better even on a relatively small and diverse dataset like food11.