Assignment 2 Report

Ufuk Cefaker
2220356171

Introduction

In this assignment, we aimed to perform image classification using Convolutional Neural Networks (CNNs). Two main tasks were completed:

- **Part 1:** Implement two CNN classifiers from scratch (SimpleCNN and ResidualCNN).
- **Part 2:** Apply transfer learning using MobileNetV2.

Dataset

The dataset used in this assignment is the **Food-11 dataset**, which contains approximately 2750 medium-quality food images categorized into 11 classes. The dataset is organized into three main directories: `train`, `validation`, and `test`, allowing for standard supervised learning and evaluation.

The 11 food categories are:

Apple Pie – Cheesecake – Chicken Curry – French Fries
Fried Rice – Hamburger – Hot Dog – Ice Cream
Omelette – Pizza – Sushi

The dataset contains the following distribution:

- **Training set:** 2200 images
- **Validation set:** 275 images
- **Test set:** 275 images

The dataset can be downloaded from the following link: **Food-11 Dataset**.

Model Training Function

The main function used for training and validating the models is `train_model`. This function performs the full training loop, including:

- Training the model for a specified number of epochs.
- Calculating and storing **loss** and **accuracy** for both training and validation datasets.
- Tracking and saving the **best model** based on validation accuracy.

Below is the full implementation of the `train_model` function:

Listing 1: Full training function

```
def train_model(model, train_loader, val_loader, optimizer, criterion, num_epochs=50):
    model.to(device)
    history = {'train_loss': [], 'train_acc': [], 'val_loss': [], 'val_acc': []}
    best_val_acc = 0.0
    best_weights = None

    for epoch in range(1, num_epochs + 1):
        # --- Training phase ---
        model.train()
        running_loss = 0.0
        running_corrects = 0
        total_samples = 0

        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)

            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            preds = outputs.argmax(dim=1)
            running_loss += loss.item() * inputs.size(0)
            running_corrects += (preds == labels).sum().item()
            total_samples += inputs.size(0)

        epoch_train_loss = running_loss / total_samples
        epoch_train_acc = running_corrects / total_samples

        # --- Validation phase ---
        model.eval()
        val_loss = 0.0
        val_corrects = 0
        val_samples = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = model(inputs)
                loss = criterion(outputs, labels)

                preds = outputs.argmax(dim=1)
                val_loss += loss.item() * inputs.size(0)
                val_corrects += (preds == labels).sum().item()
                val_samples += inputs.size(0)

        epoch_val_loss = val_loss / val_samples
        epoch_val_acc = val_corrects / val_samples

        history['train_loss'].append(epoch_train_loss)
        history['train_acc'].append(epoch_train_acc)
        history['val_loss'].append(epoch_val_loss)
        history['val_acc'].append(epoch_val_acc)

        if epoch_val_acc > best_val_acc:
            best_val_acc = epoch_val_acc
            best_weights = model.state_dict().copy()

    model.load_state_dict(best_weights)
```

Part 1: CNN Classifier from Scratch

Data Augmentation and DataLoader

Transformations code block:

Listing 2: Data augmentation transforms

```
train_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2, contrast=0.2, saturation=0.2, hue
    =0.1),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # [-1,1]
    range
])

val_test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
])
```

For the training phase, various data augmentation techniques were applied to improve generalization and prevent overfitting. The following transformations were used for the `train` set:

- **Resize:** Resized all images to `224x224` pixels.
- **Random Horizontal Flip:** Applied with a probability of 0.5.
- **Random Rotation:** Randomly rotated images within a range of ±15 degrees.
- **Color Jitter:** Adjusted brightness, contrast, saturation, and hue with parameters: `brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1`.
- **Normalization:** Normalized images to have mean `[0.5, 0.5, 0.5]` and standard deviation `[0.5, 0.5, 0.5]`, scaling them into the range `[-1, 1]`.

For the `validation` and `test` sets, only resizing and normalization were applied to ensure a consistent evaluation setup.

The datasets were loaded using `torchvision.datasets.ImageFolder`, and wrapped in PyTorch `DataLoader` objects as follows:

- **Batch Size:** 64
- **Shuffle:** `True` for training, `False` for validation and test.
- **Workers:** 2 worker threads for faster data loading.
- **Pin Memory:** Enabled to speed up data transfer to GPU.

SimpleCNN Architecture

The **SimpleCNN** architecture is designed to progressively extract deeper and more abstract features from the input images through a stack of convolutional layers. It consists of 5 convolutional blocks followed by 2 fully connected (FC) layers.

Layer	Input Channels	Output Channels	Kernel Size / Stride / Padding	Other Components
Conv1	3	32	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
Conv2	32	64	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
Conv3	64	128	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
Conv4	128	256	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
Conv5	256	512	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
FC1	–	256	Linear(512*7*7 → 256)	ReLU, Dropout
FC2	–	11	Linear(256 → 11)	–

Table 1: SimpleCNN architecture: convolutional and fully connected layers overview.

Listing 3: SimpleCNN architecture snippet

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=11, dropout_rate=0):
        super(SimpleCNN, self).__init__()
        self.features = nn.Sequential(
            # Conv1 block
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Conv2 block
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Conv3 block
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Conv4 block
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            # Conv5 block
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
```

Architecture rationale:

The network follows a typical deep convolutional design where each subsequent layer increases the number of output channels. This allows the network to:

- Capture simple patterns (edges, textures) in early layers.
- Learn increasingly abstract and high-level representations (shapes, objects) in deeper layers.

Detailed breakdown:

- **Conv1:** `(in_channels=3, out_channels=32)` – Captures low-level features like edges and basic color regions.
- **Conv2:** `(in_channels=32, out_channels=64)` – Builds on earlier features to detect patterns and textures.
- **Conv3:** `(in_channels=64, out_channels=128)` – Learns more abstract visual concepts and mid-level features.
- **Conv4:** `(in_channels=128, out_channels=256)` – Focuses on more detailed structures and object parts.
- **Conv5:** `(in_channels=256, out_channels=512)` – Captures highly abstract and class-specific patterns crucial for classification.

Fully Connected Layers:
Once the feature extraction is complete, the 512-channel output (with spatial size 7x7) is flattened and passed through two FC layers:

- **FC1:** Reduces the flattened vector from size `512*7*7` to 256 dimensions.
- **FC2:** Maps the 256-dimensional feature vector to 11 output classes.

This structure allows the model to convert rich visual features into final class scores.

Code snippet of the classifier block:

Listing 4: Classifier block in SimpleCNN

```
self.classifier = nn.Sequential(
    nn.Linear(512 * 7 * 7, 256),
    nn.ReLU(),
    nn.Dropout(p=dropout_rate),
    nn.Linear(256, num_classes)
)
```

Explanation:

- `nn.Linear(512 * 7 * 7, 256):` Flattens and maps the convolutional output to a 256D vector.
- `nn.ReLU():` Applies a non-linear activation function.
- `nn.Dropout(p=dropout_rate):` Regularizes the network by randomly deactivating neurons. Initially set to `0` (no dropout).
- `nn.Linear(256, num_classes):` Outputs raw class scores for the 11 categories.

Dropout:
To prevent overfitting, a `Dropout` layer is integrated into the classifier block. While the base model uses `dropout_rate = 0` (no dropout applied), further experiments introduce dropout rates of 0.3 and 0.5 to investigate the effect of regularization on performance.

ResidualCNN Architecture

The **ResidualCNN** architecture builds upon the SimpleCNN structure by integrating **Residual Blocks**, which are designed to allow deeper networks to train more effectively by enabling gradient flow through shortcut connections. It follows the same 5 convolutional blocks and 2 fully connected (FC) layers, but introduces `ResidualBlock` modules after Conv2 and Conv4.

Layer	Input Channels	Output Channels	Kernel / Stride / Padding	Other Components
Conv1	3	32	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
Conv2	32	64	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
Residual Block	64	64	3x3 / 1 / 1 (x2)	Two Conv layers with BatchNorm, ReLU, skip connection
Conv3	64	128	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
Conv4	128	256	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
Residual Block	256	256	3x3 / 1 / 1 (x2)	Two Conv layers with BatchNorm, ReLU, skip connection
Conv5	256	512	3x3 / 1 / 1	BatchNorm, ReLU, MaxPool(2x2)
FC1	–	256	Linear(512*7*7 → 256)	ReLU, Dropout
FC2	–	11	Linear(256 → 11)	–

Table 2: ResidualCNN architecture: convolutional, residual, and fully connected layers overview.

Architecture rationale:
ResidualCNN retains the core logic of SimpleCNN but enhances it with two key residual connections:

- **After Conv2:** A Residual Block with 64 channels helps the network refine mid-level feature representations without losing gradient strength.
- **After Conv4:** Another Residual Block with 256 channels focuses on deeper features, improving the model's ability to capture complex patterns.

The residual connections address potential problems like vanishing gradients and enable the model to learn identity mappings, which makes deeper networks more stable.

Fully Connected Layers:
As in SimpleCNN:

- **FC1:** Maps the flattened 512-channel (7x7) output to a 256-dimensional vector.
- **FC2:** Maps the 256D vector to 11 output classes.

Code snippet of the ResidualBlock:

Listing 5: Residual Block used in ResidualCNN

```
class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(channels, channels, kernel_size=3, stride=1,
    padding=1)
        self.bn1 = nn.BatchNorm2d(channels)
        self.relu = nn.ReLU()
        self.conv2 = nn.Conv2d(channels, channels, kernel_size=3, stride=1,
    padding=1)
        self.bn2 = nn.BatchNorm2d(channels)

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

Explanation:

- Each Residual Block contains two convolutional layers with the same input and output channels, followed by BatchNorm and ReLU.
- The input (`residual`) is added back to the output after the two convolutions (skip connection).
- This enables the block to learn residual mappings ($F(x) + x$), which helps in deeper network optimization.

Classifier block:
The classifier block remains identical to the SimpleCNN:

Listing 6: Classifier block in ResidualCNN

```
self.classifier = nn.Sequential(
    nn.Linear(512 * 7 * 7, 256),
    nn.ReLU(),
    nn.Dropout(p=dropout_rate),
    nn.Linear(256, num_classes)
)
```

Dropout:
Just like in SimpleCNN, a `Dropout` layer is integrated into the classifier block to prevent overfitting. The default `dropout_rate` is 0, but experiments with 0.3 and 0.5 were conducted to observe regularization effects.

Training Settings

Both **SimpleCNN** and **ResidualCNN** models were trained for **50 epochs** under different configurations to examine the effects of learning rate and batch size.

Hyperparameters:

- **Learning rates:** `0.0001, 0.001, 0.005`
- **Batch sizes:** `32, 64`

Optimizer: `Adam`, known for its adaptive learning, was chosen for faster and more stable convergence.
Loss Function: `CrossEntropyLoss`, the standard loss for multi-class classification tasks, was used to calculate error.

Training setup:
For each configuration, both models were trained from scratch using their respective architectures. The different combinations of learning rate and batch size were defined in a list, and iteratively tested. The training pipeline included dynamically adjusting the `DataLoader` for the current batch size and initializing a fresh model instance each time.

Tracking results:
Rather than printing all code details, it's worth mentioning that:

- I maintained two dictionaries: `results_with_residual` and `results_without_residual`.
- Each dictionary recorded:
  - Learning rate and batch size of the run,
  - Best validation accuracy,
  - Training history (loss and accuracy per epoch).
- These results were later compared to observe the performance differences between **SimpleCNN** and **ResidualCNN** across all configurations.

This structured approach allowed for an effective side-by-side evaluation of both architectures and helped in identifying the best-performing hyperparameter settings.

Results

We present the results in two parts: **SimpleCNN** and **ResidualCNN**. Each part shows 6 training runs with different learning rates and batch sizes, followed by an overall comparison plot.

SimpleCNN Results

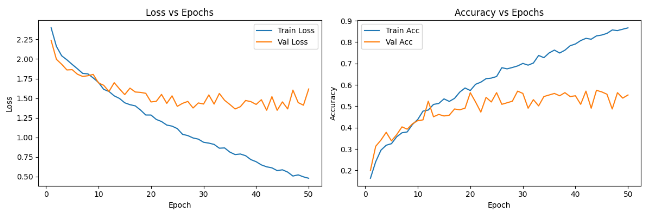


Figure 1: Run 1: lr=0.0001, batch_size=64

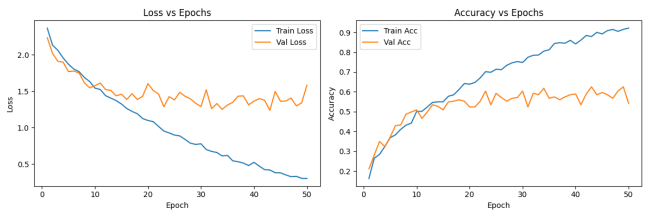


Figure 2: Run 2: lr=0.0001, batch_size=32

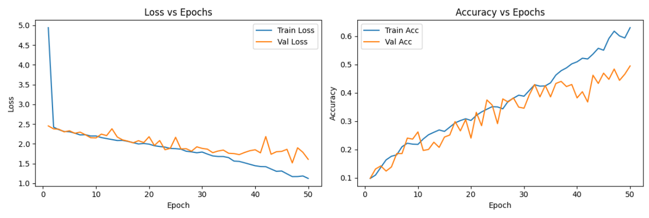


Figure 3: Run 3: lr=0.001, batch_size=64

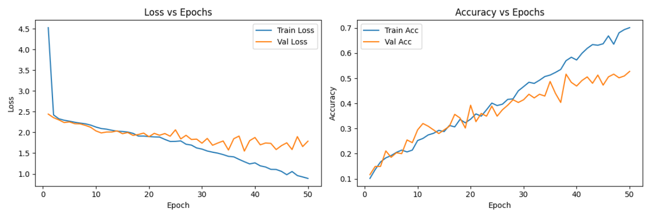


Figure 4: Run 4: lr=0.001, batch_size=32

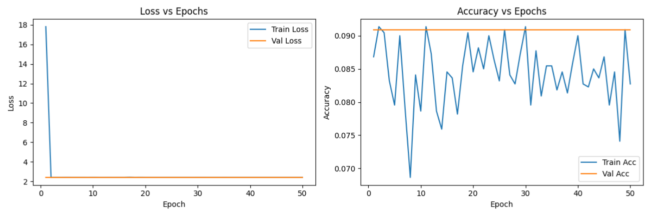


Figure 5: Run 5: lr=0.005, batch_size=64

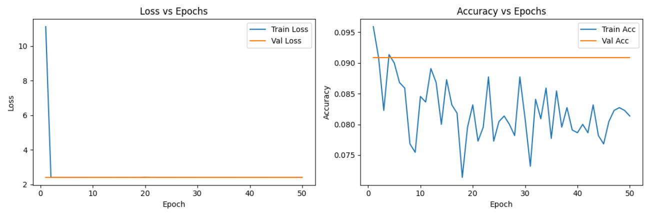


Figure 6: Run 6: lr=0.005, batch_size=32

SimpleCNN Overall

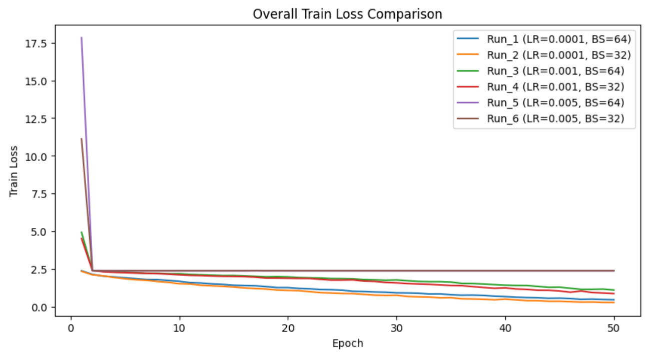


Figure 7: SimpleCNN: Overall Train Loss Comparison

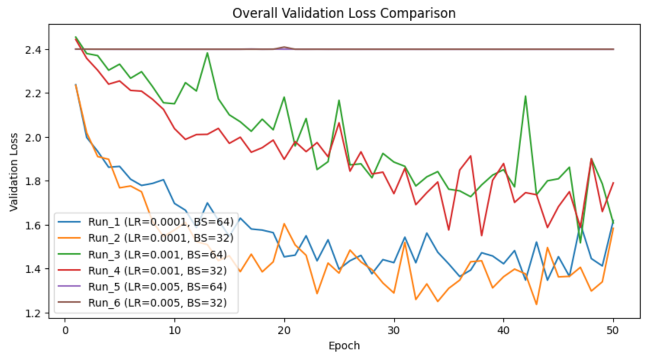


Figure 8: SimpleCNN: Overall Validation Loss Comparison

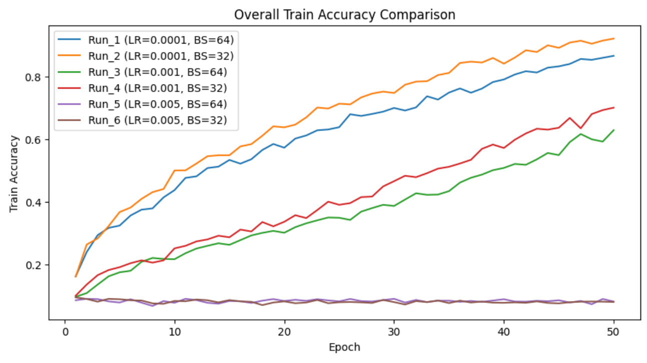


Figure 9: SimpleCNN: Overall Train Accuracy Comparison

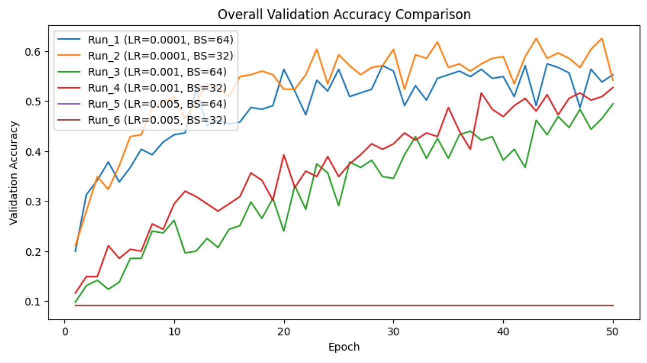


Figure 10: SimpleCNN: Overall Validation Accuracy Comparison

ResidualCNN Results

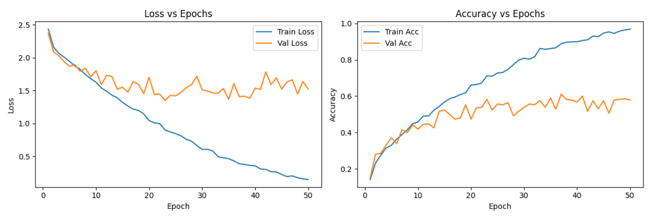


Figure 11: Run 1: lr=0.0001, batch_size=64

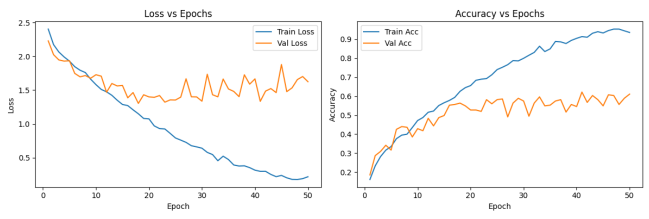


Figure 12: Run 2: lr=0.0001, batch_size=32

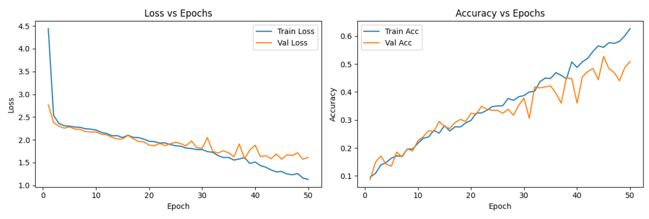


Figure 13: Run 3: lr=0.001, batch_size=64

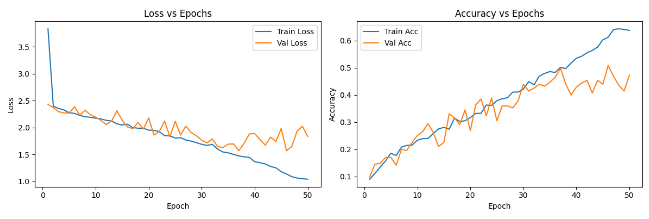


Figure 14: Run 4: lr=0.001, batch_size=32

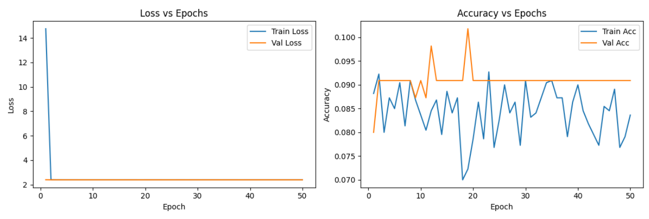


Figure 15: Run 5: lr=0.005, batch_size=64

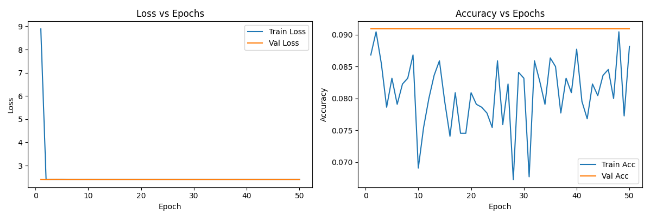


Figure 16: Run 6: lr=0.005, batch_size=32

ResidualCNN Overall

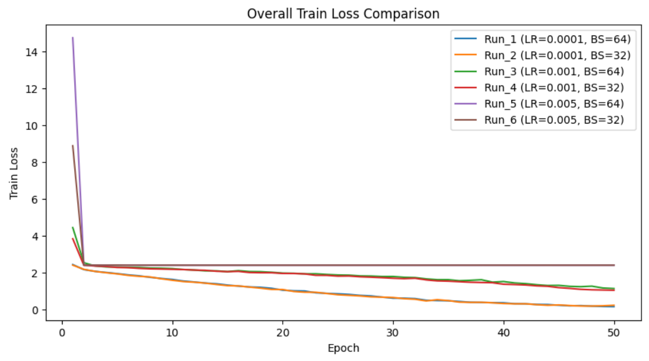


Figure 17: ResidualCNN: Overall Train Loss Comparison

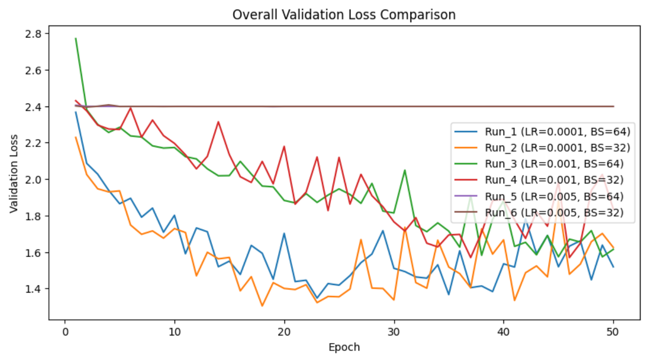


Figure 18: ResidualCNN: Overall Validation Loss Comparison

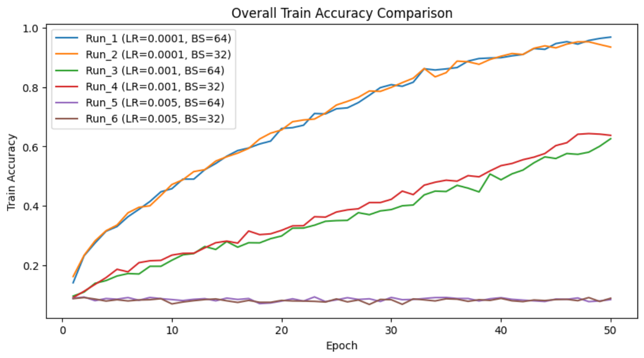


Figure 19: ResidualCNN: Overall Train Accuracy Comparison

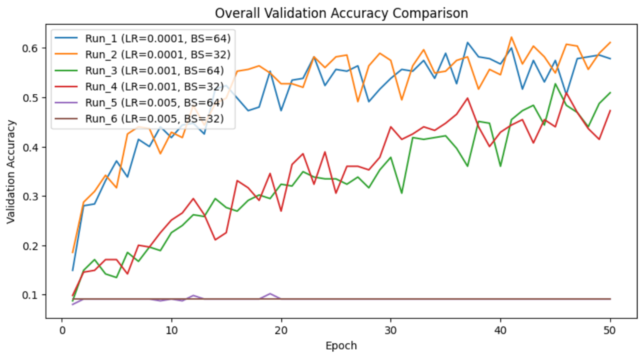


Figure 20: ResidualCNN: Overall Validation Accuracy Comparison

Best Model

The best performing configuration for both **SimpleCNN** and **ResidualCNN** was achieved with a learning rate of `0.0001` and batch size of `32`.

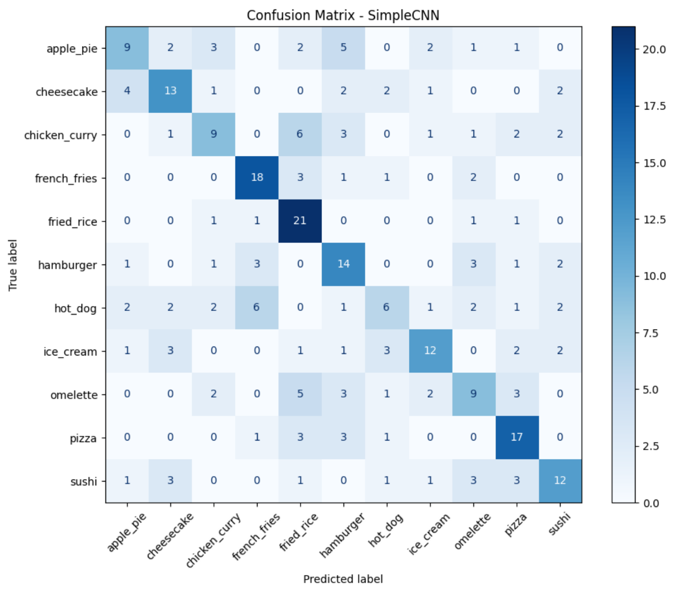
Model	Run	LR	Batch Size	Best Val Accuracy	Test Accuracy
SimpleCNN	Run 2	0.0001	32	0.5891	0.5091
ResidualCNN	Run 2	0.0001	32	0.6327	0.5018

Table 3: Best model configurations and their corresponding validation and test accuracies.

SimpleCNN Detailed Results

Metric	Value
Train Loss (Epoch 50)	0.4216
Train Accuracy (Epoch 50)	0.8850
Validation Loss (Epoch 50)	1.4478
Validation Accuracy (Epoch 50)	0.5745
Best Validation Accuracy	0.5891
Test Accuracy	0.5091

Table 4: Detailed results of the best SimpleCNN model (Run 2).



True label	apple_pie	cheesecake	chicken_curry	french_fries	fried_rice	hamburger	hot_dog	ice_cream	omelette	pizza	sushi
apple_pie	9	2	3	0	2	5	0	2	1	1	0
cheesecake	4	13	1	0	0	2	2	1	0	0	2
chicken_curry	0	1	9	0	6	3	0	1	1	2	2
french_fries	0	0	0	18	3	1	1	0	2	0	0
fried_rice	0	0	1	1	21	0	0	0	1	1	0
hamburger	1	0	1	3	0	14	0	0	3	1	2
hot_dog	2	2	2	6	0	1	6	1	2	1	2
ice_cream	1	3	0	0	1	1	3	12	0	2	2
omelette	0	0	2	0	5	3	1	2	9	3	0
pizza	0	0	0	1	3	3	1	0	0	17	0
sushi	1	3	0	0	1	0	1	1	3	3	12

Figure 21: Confusion matrix of the best SimpleCNN model on the test set.

ResidualCNN Detailed Results

Metric	Value
Train Loss (Epoch 50)	0.1533
Train Accuracy (Epoch 50)	0.9614
Validation Loss (Epoch 50)	1.8360
Validation Accuracy (Epoch 50)	0.6291
Best Validation Accuracy	0.6327
Test Accuracy	0.5018

Table 5: Detailed results of the best ResidualCNN model (Run 2).

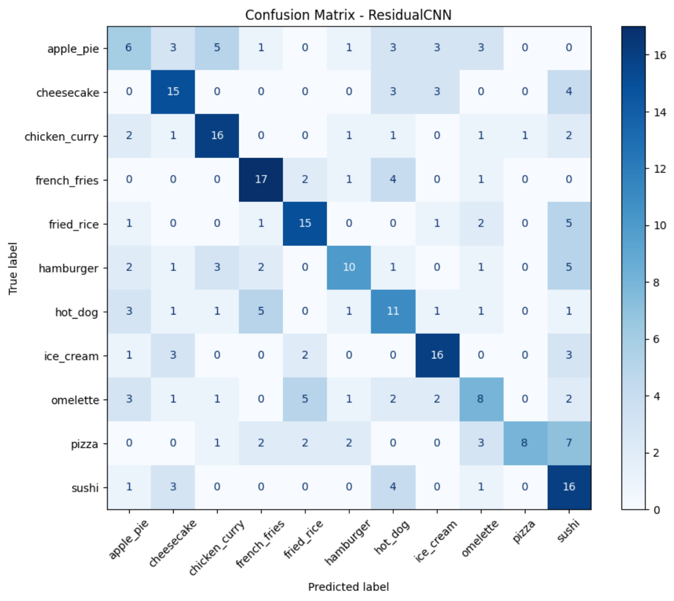


Figure 22: Confusion matrix of the best ResidualCNN model on the test set.

Although **ResidualCNN** achieved a higher peak validation accuracy, the **SimpleCNN** slightly outperformed it on the test set, indicating better generalization in this experiment.

Dropout Integration

We integrated `Dropout` into the classifier section with values of `0.3` and `0.5`, placing it after the first fully connected layer to prevent overfitting.

These specific values were chosen because they are widely accepted as effective regularization rates in deep learning literature: `0.5` is a standard benchmark dropout rate, often used in fully connected layers to provide strong regularization, while `0.3` offers a more moderate regularization effect. By testing both, we aimed to observe the trade-off between underfitting (high dropout rate) and overfitting (low/no dropout). Additionally, since our models have a moderate depth (5 convolutional layers followed by 2 fully connected layers), we expected `0.3` to provide sufficient regularization without severely hindering learning, while `0.5` served as a stronger baseline for comparison.

SimpleCNN Dropout Results

Dropout Rate	Train Loss (Epoch 50)	Train Acc	Val Loss	Best Val Acc	Test Accuracy
0.3	1.0923	0.6382	1.2153	0.6182	0.5018
0.5	1.6676	0.4023	1.5560	0.5673	0.4327

Table 6: SimpleCNN performance with different dropout rates.

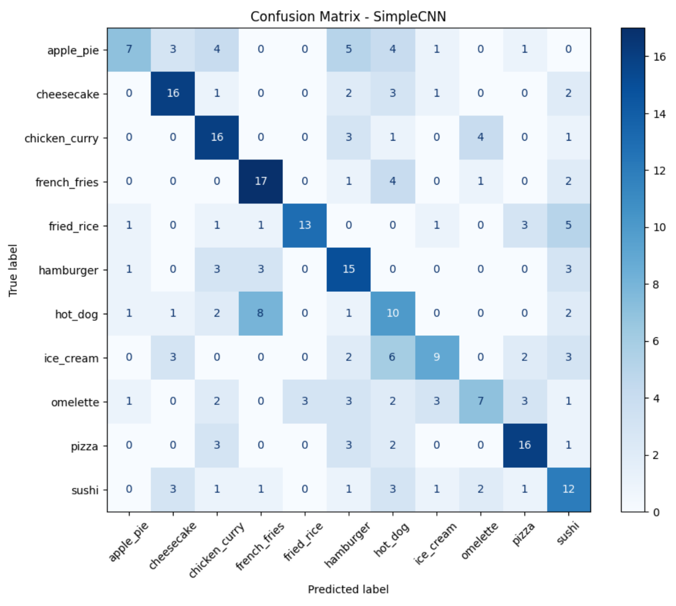


Figure 23: Confusion matrix of SimpleCNN with dropout 0.3.

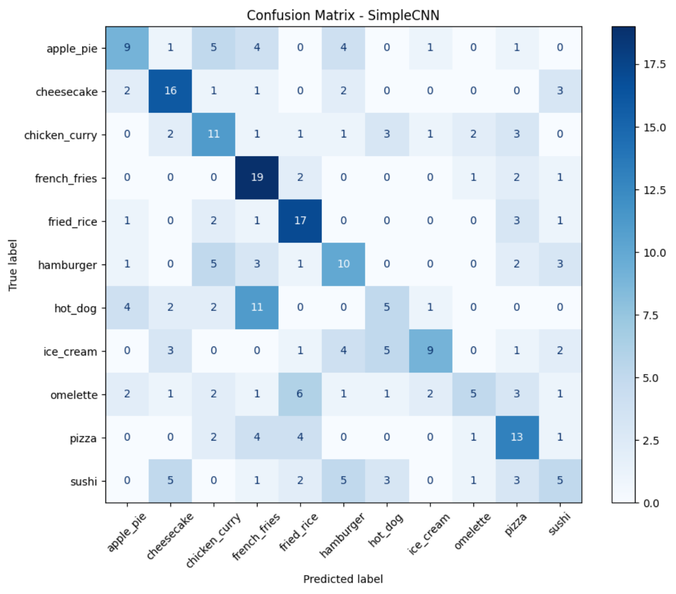


Figure 24: Confusion matrix of SimpleCNN with dropout 0.5.

ResidualCNN Dropout Results

Dropout Rate	Train Loss (Epoch 50)	Train Acc	Val Loss	Best Val Acc	Test Accuracy
0.3	0.7805	0.7409	1.2036	0.6364	0.5200
0.5	1.3986	0.5141	1.4892	0.5855	0.4727

Table 7: ResidualCNN performance with different dropout rates.

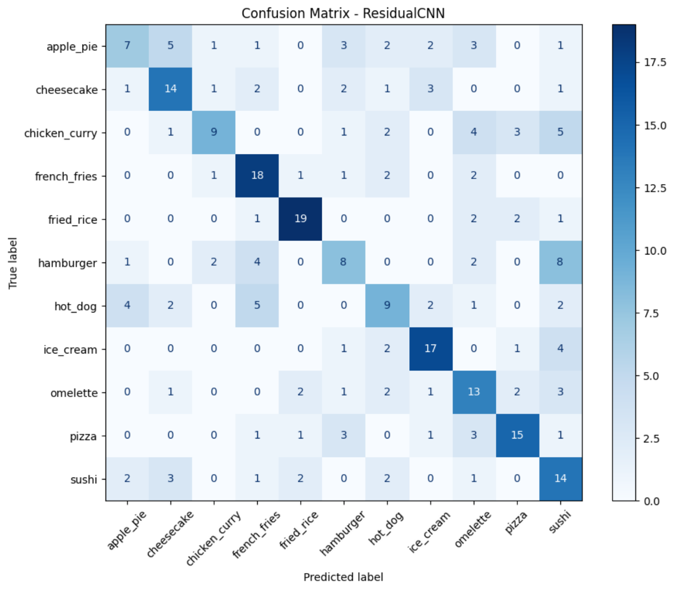


Figure 25: Confusion matrix of ResidualCNN with dropout 0.3.

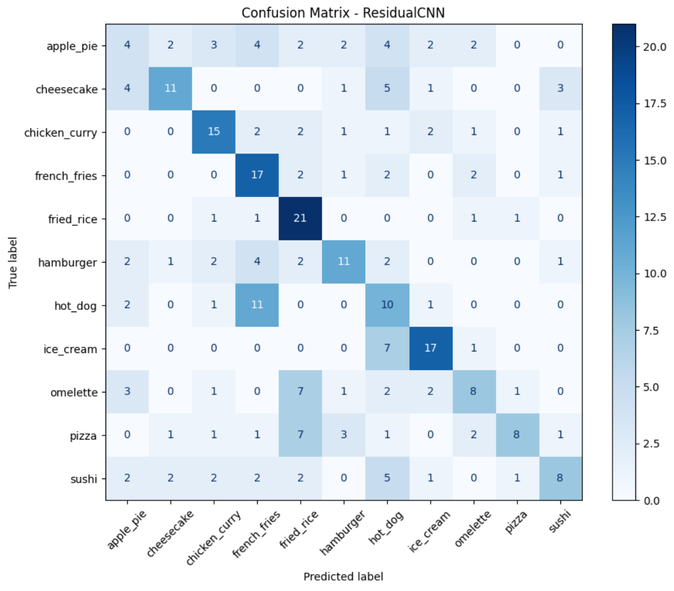


Figure 26: Confusion matrix of ResidualCNN with dropout 0.5.

Explanation

In **SimpleCNN**, adding dropout `0.3` yielded a best validation accuracy of `61.82%` and a test accuracy of `50.18%`, while `0.5` led to a drop in performance.

In **ResidualCNN**, dropout `0.3` similarly improved validation and test results (best val: `63.64%`, test: `52.00%`), but `0.5` again resulted in lower metrics.

Conclusion: A moderate dropout rate of `0.3` was the most effective for both architectures, slightly boosting generalization without sacrificing too much training performance.

Analysis

Overall Performance: Both **SimpleCNN** and **ResidualCNN** architectures demonstrated clear learning across all configurations. ResidualCNN consistently achieved higher validation accuracy than SimpleCNN, peaking at `63.27%` validation accuracy (Run 2), whereas SimpleCNN reached a peak of `58.91%`. Despite this, SimpleCNN showed slightly better generalization to the test set in some configurations, likely due to its simpler structure, which may reduce overfitting when data is limited.

Dropout Impact: Introducing dropout with a rate of `0.3` led to noticeable improvements in both architectures. For example:

- **SimpleCNN:** Validation accuracy improved from `58.91%` to `61.82%`, and test accuracy increased to `50.18%`.
- **ResidualCNN:** Validation accuracy improved from `63.27%` to `63.64%`, and test accuracy rose to `52.00%`.

However, a higher dropout rate (`0.5`) negatively impacted both models, suggesting that too much regularization harms learning capacity in this dataset.

Class-wise Performance: Examining the confusion matrices, some classes such as `Cheesecake`, `Pizza`, and `French Fries` were consistently predicted with higher accuracy. These classes likely have more distinctive visual features, making them easier for the models to classify. In contrast, classes like `Hot Dog`, `Hamburger`, and `Omelette` exhibited more confusion, which can be attributed to visual similarities (e.g., overlapping colors and textures).

Training Stability: Across multiple runs, both models exhibited a clear trend:

- Low learning rates (`0.0001`) led to more stable convergence and higher final accuracies.
- Higher learning rates (`0.005`) caused unstable training, with both training and validation metrics suffering from fluctuations and underperformance.

Residual Block Impact: The addition of residual blocks in ResidualCNN helped the model maintain high training accuracy (`96.14%` at best) and improved validation performance. This confirms that residual connections help deeper models retain gradient flow and optimize more effectively.

Generalization: Although ResidualCNN had stronger training and validation metrics, the test accuracy gains were modest. This suggests that while deeper architectures can better fit the training data, their generalization is still constrained by the size and variability of the dataset.

Challenges: The main challenge observed was class imbalance and high intra-class similarity. Even with data augmentation, certain classes remained difficult to distinguish, highlighting the need for additional data or more sophisticated augmentation strategies for further improvements.

Key Insight: The best overall configuration was ResidualCNN with `dropout=0.3`, which achieved the highest test accuracy of `52.00%`. This indicates that a combination of deeper architecture and moderate regularization yields the best performance for this task.

Part 2: Transfer Learning with MobileNetV2

What is Fine-Tuning?

Fine-tuning is a transfer learning strategy where a pre-trained network is adapted to a new task by retraining certain layers while keeping the others fixed. This allows us to leverage the powerful feature extraction capabilities that were learned from large-scale datasets (such as ImageNet), even if our target dataset is much smaller. Instead of training a model from scratch, which would require significant data and computational resources, fine-tuning enables faster convergence and typically yields better results.

In this project, I chose MobileNetV2 because it is a lightweight, efficient architecture specifically designed for mobile and embedded vision applications. It balances accuracy and speed by using depthwise separable convolutions, making it an excellent candidate for transfer learning when computational efficiency is also a concern.

We freeze most of the network's layers to retain the generalized features learned from ImageNet. These early layers typically extract low-level features (edges, textures), which are useful across many tasks. We then retrain the final fully connected (FC) layer, which is task-specific, to adjust the model to our dataset's unique classes (11 food categories in the Food-11 dataset). This technique ensures that we reuse the most informative features while adapting only the necessary part of the network, thus reducing the risk of overfitting and optimizing training time.

Why do we freeze layers? The main reason is to prevent overwriting the valuable knowledge gained from pre-training. If we were to train the entire network on a small dataset, it could lead to overfitting and the model might "forget" the general features. By freezing the layers and training only the classifier, we fine-tune the decision-making part without affecting the proven feature extraction backbone.

Code snippet (Scenario 1 - Only FC layer is trained):

```python
# Load pretrained MobileNetV2
mobilenet_v2 = models.mobilenet_v2(pretrained=True)

# Freeze all layers
for param in mobilenet_v2.parameters():
    param.requires_grad = False

# Replace the classifier layer
mobilenet_v2.classifier[1] = nn.Linear(mobilenet_v2.last_channel, 11)

# Enable training only for the new classifier layer
for param in mobilenet_v2.classifier[1].parameters():
    param.requires_grad = True

# Move model to device
mobilenet_v2 = mobilenet_v2.to(device)
```

Training Scenarios

We explored two different fine-tuning strategies:

- **Scenario 1:** Only the FC layer was trained; all other layers were frozen.
- **Scenario 2:** The last two convolutional blocks (`features[17] and features[18]`) and the FC layer were trained; the rest was frozen.

Code snippet (Scenario 2 - Last 2 conv layers + FC layer are trained):

```
# Load pretrained MobileNetV2
mobilenet_v2_case2 = models.mobilenet_v2(pretrained=True)

# Freeze all layers
for param in mobilenet_v2_case2.parameters():
    param.requires_grad = False

# Unfreeze last two conv blocks
for i in [-1, -2]:
    for param in mobilenet_v2_case2.features[i].parameters():
        param.requires_grad = True

# Replace classifier
mobilenet_v2_case2.classifier[1] = nn.Linear(mobilenet_v2_case2.last_channel, 11)

# Ensure classifier is trainable
for param in mobilenet_v2_case2.classifier.parameters():
    param.requires_grad = True

# Move model to device
mobilenet_v2_case2 = mobilenet_v2_case2.to(device)
```

Optimization: Both scenarios used `CrossEntropyLoss` and the `Adam` optimizer, but with different learning rates:

- Scenario 1: `lr=1e-3`
- Scenario 2: `lr=1e-4`

Results

Scenario	Best Validation Accuracy	Test Accuracy
Scenario 1	0.8800	0.7491
Scenario 2	0.9018	0.7927

Table 8: Transfer learning results for two scenarios.

Scenario 1 Confusion Matrix:

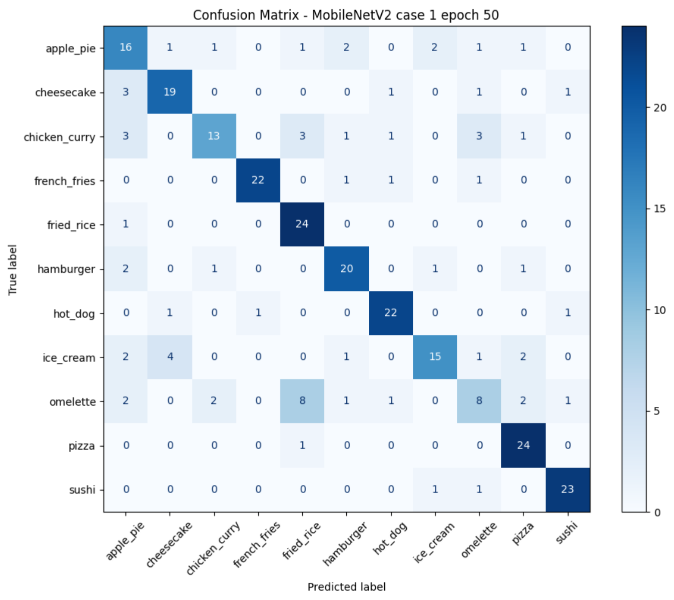


Figure 27: Confusion matrix of MobileNetV2 (Scenario 1: only FC layer trained) on the test set.

Scenario 1 Train and Validation Curves:

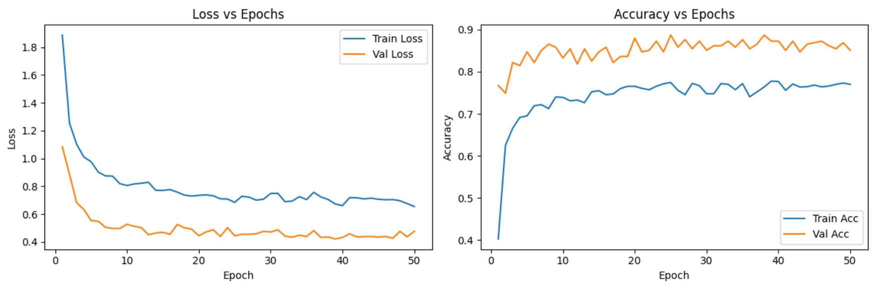


Figure 28: Scenario 1: Loss and Accuracy vs Epochs.

Scenario 2 Confusion Matrix:

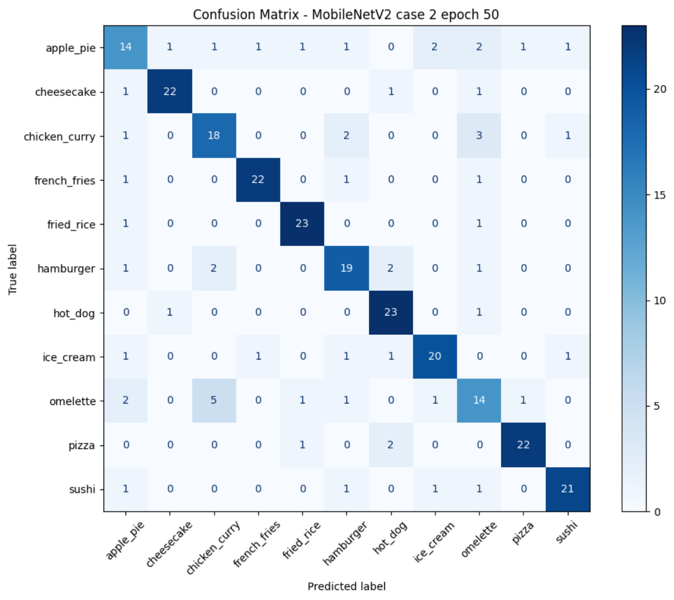


Figure 29: Confusion matrix of MobileNetV2 (Scenario 2: last 2 conv layers + FC trained) on the test set.

Scenario 2 Train and Validation Curves:

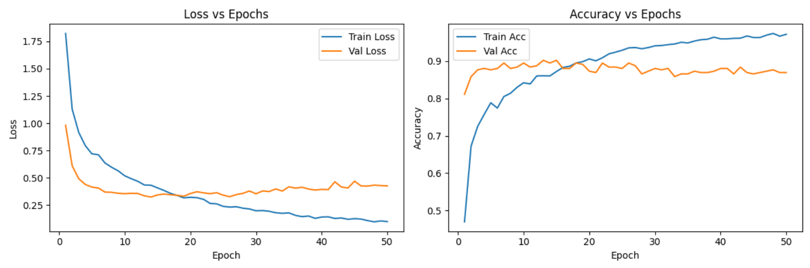


Figure 30: Scenario 2: Loss and Accuracy vs Epochs.

Analysis of Transfer Learning Results

Scenario 1 (Only FC Layer Trained): In this scenario, only the fully connected (FC) layer was trained, while all other layers were frozen. From the loss and accuracy graphs, we observe that the validation accuracy quickly increased and stabilized around 85-88% after a few epochs. The training accuracy remained lower than the validation accuracy, indicating that the pre-trained features were strong, but training only the FC layer limited the model's capacity to fully adapt to our dataset. The confusion matrix shows that while certain classes (e.g., *fried rice*, *pizza*, *sushi*) were classified accurately, there were noticeable confusions in others, particularly *omelette* and *apple pie*. The test accuracy was recorded as **0.7491**.

Scenario 2 (Last 2 Conv Layers + FC Layer Trained): In this scenario, the last two convolutional layers were unfrozen and trained along with the FC layer. The loss and accuracy plots reveal a clear improvement: training accuracy reached around 97%, and validation accuracy was more stable and higher, peaking at **90.18%**. The confusion matrix indicates that misclassifications were reduced compared to Scenario 1, and the separation between classes became clearer. The model achieved a test accuracy of **0.7927**, outperforming Scenario 1.

Comparative Findings:

- Scenario 2 achieved a significant improvement in both validation and test accuracy over Scenario 1.
- Training additional convolutional layers allowed the model to better adapt to the specific characteristics of the dataset, leading to stronger overall performance.
- While Scenario 1 provided quick convergence and stable validation accuracy, its ability to handle complex class distinctions was limited.
- Both scenarios performed well overall, but Scenario 2 particularly excelled in refining class boundaries and reducing confusion between similar classes.

In summary, while training only the FC layer is a fast and simple method, unlocking a few additional convolutional layers can substantially enhance model adaptability and accuracy when applying transfer learning.

Comparison with Part 1

The results clearly highlight the advantage of transfer learning over training CNN models from scratch:

- The best model in Part 1 (ResidualCNN with dropout) achieved a test accuracy of `52.00%`, while MobileNetV2 (Scenario 2) achieved `79.27%`.
- Transfer learning models achieved higher validation accuracy (`88.00%` and `90.18%`) compared to Part 1 models.
- The convergence speed was much faster in transfer learning, and the models generalized better even though the Food-11 dataset is relatively small.
- Fine-tuning the last two convolutional layers (Scenario 2) further improved the performance, showing that partial unfreezing of the feature extractor layers can help better adapt to the new dataset.

Finally: Transfer learning with MobileNetV2 was significantly more effective than the from-scratch CNN models. The results demonstrate the power of using pre-trained models, especially when working with a limited dataset.

Conclusion

In this assignment, we explored two distinct approaches to image classification on the Food-11 dataset: building CNN classifiers from scratch and applying transfer learning using MobileNetV2.

Part 1: CNNs from Scratch We implemented two architectures: *SimpleCNN* and *ResidualCNN*. While ResidualCNN consistently achieved higher validation accuracy (peaking at 63.27%) thanks to its deeper architecture and residual connections, SimpleCNN demonstrated slightly better test generalization in some scenarios, likely due to its simpler design reducing overfitting. Experiments with different dropout rates revealed that moderate regularization (dropout = 0.3) improved generalization for both models, but excessive regularization (dropout = 0.5) hindered performance.

Despite these efforts, the models trained from scratch were limited by the relatively small size of the dataset, and the best test accuracy plateaued around 52.00%.

Part 2: Transfer Learning Applying transfer learning with MobileNetV2 brought substantial performance gains. In Scenario 1 (only FC layer trained), the model quickly reached 88.00% validation accuracy and 74.91% test accuracy. Scenario 2, where we unfroze the last two convolutional blocks alongside the FC layer, further improved performance to 90.18% validation accuracy and 79.27% test accuracy. This showed that fine-tuning a portion of the backbone allowed the model to adapt more effectively to the specific characteristics of the Food-11 dataset.

Key Insights:

- Transfer learning outperformed from-scratch training by a large margin, both in accuracy and convergence speed.
- Fine-tuning deeper layers (Scenario 2) yielded noticeable improvements over training only the FC layer (Scenario 1).
- While deeper networks (ResidualCNN) benefited from residual connections, their generalization was still constrained by data limitations, highlighting the challenges of training deep models without sufficient data.
- Moderate dropout improved generalization, confirming that regularization is essential, especially when training from scratch.

Challenges: Some classes (e.g., *Omelette*, *Apple Pie*) consistently showed confusion across models, indicating that intra-class similarity and class imbalance remain significant hurdles. Despite using data augmentation, certain visual overlaps between classes proved difficult to disentangle.

Final Takeaway: The project clearly demonstrates that transfer learning is a powerful tool when working with limited data, enabling high performance without requiring extensive computational resources. Fine-tuning selective layers provides a practical balance between reusing learned features and adapting to new tasks. For future work, collecting more data or exploring advanced augmentation techniques could further close the gap in challenging classes.