BBM 418 - Computer Vision Laboratory

Assignment 2 Report

Emirhan Utku
2210765029

1 Introduction

In recent years, deep learning approaches—especially Convolutional Neural Networks (CNNs)—have revolutionized the field of computer vision. One of the most fundamental tasks in this domain is image classification, where the goal is to assign an input image to one of several predefined categories. CNNs have demonstrated remarkable success on such tasks by automatically learning hierarchical representations of input images, starting from low-level features such as edges to more complex, abstract patterns.

In this assignment, we focus on building and training CNN models for image classification from scratch and via transfer learning. Our objective is to familiarize ourselves with key deep learning concepts, such as convolutional layers, pooling, batch normalization, residual connections, and transfer learning strategies. We implement two custom CNN models—one with and one without residual connections—and analyze their performance under various hyperparameter settings. Additionally, we fine-tune a pre-trained MobileNetV2 model on the Food-11 dataset to explore the benefits of transfer learning.

Throughout this project, we aim to investigate how different design choices (e.g., architecture, learning rates, batch sizes, dropout probabilities) influence the model's generalization ability. Furthermore, we analyze confusion matrices to understand class-specific performance and provide insights into strengths and weaknesses of different approaches.

In the following sections, we describe the dataset in detail, explain our model architectures and training methodology, present experimental results, and conclude with a discussion of findings and potential future improvements.

2 Dataset

The dataset used in this assignment is the Food-11 dataset, a medium-sized image classification dataset containing approximately 2,750 food images divided into 11 different categories: apple pie, cheesecake, chicken curry, french fries, fried rice, hamburger, hot dog, ice cream, omelette, pizza, and sushi. The dataset is organized into three main folders:

- `train/`: Contains the majority of the images and is used for training the models.
- `validation/`: Contains a smaller number of images used for tuning hyperparameters and monitoring model performance during training.
- `test/`: Contains unseen images used for evaluating the final model performance.

Each category has its own subfolder under `train/`, `validation/`, and `test/`, with the corresponding images. The dataset exhibits moderate intra-class variation and realistic noise (e.g., variations in lighting, background clutter), making it a suitable benchmark for evaluating the robustness of image classification models.

All images were resized to a uniform size of 150×150 pixels during preprocessing. Data augmentation techniques, such as random affine transformations and horizontal flips, were applied during training to improve generalization by exposing the models to a wider range of variations.

In the next sections, we describe the model architectures, training procedures, and experimental evaluations in detail.

3 Model Architectures and Training Setup

In this section, we present the two convolutional neural networks (CNNs) we built for the classification task: **a Plain CNN** and **a Residual CNN**. Both models are trained from scratch using the Food-11 dataset.

We explain the layer structures and parameter choices such as the number of input and output channels, kernel sizes, strides, and activation functions. In addition, we describe the loss function, optimization algorithm, and training strategy used for both models.

Throughout this part, we include relevant PyTorch code snippets to illustrate how the models were implemented.

3.1 Plain Convolutional Neural Network (PlainCNN)

The PlainCNN model is a simple convolutional neural network designed for the image classification task. It consists of five convolutional layers followed by two fully connected (dense) layers. In each convolutional block, we also include Batch Normalization, ReLU activation, and Max Pooling layers to stabilize and improve training.

Parametric Details

Each convolutional layer in the PlainCNN has the following key parameters:

- **in_channels**: Number of channels (depth) in the input feature map.
- **out_channels**: Number of channels (depth) in the output feature map after the convolution.
- **kernel_size**: Size of the filter (we use 5×5 or 3×3 filters).
- **stride**: Step size of the convolution (set to 1 in our model to preserve more spatial information).

We started with a higher number of channels (128) at the first layer because the early layers need to capture a wide range of low-level features like edges, textures, and simple shapes. As we move deeper into the network, we gradually reduce the number of channels ($64 \rightarrow 32 \rightarrow 16$) because deeper layers focus on combining earlier features into more abstract concepts, and fewer channels are sufficient for that.

Max Pooling layers are added after each convolution to reduce the spatial size of the feature maps. This helps in two ways:

- It makes the model more efficient by reducing the number of computations.
- It introduces a small amount of translation invariance, which improves generalization.

Batch Normalization is used after each convolutional layer to stabilize training and allow the use of higher learning rates by reducing internal covariate shift.

The table below summarizes the structure:

Layer	Parameters
Conv2d	in_channels=3, out_channels=128, kernel_size=5, stride=1
BatchNorm2d	num_features=128
MaxPool2d	kernel_size=2, stride=2
Conv2d	in_channels=128, out_channels=64, kernel_size=3, stride=1
BatchNorm2d	num_features=64
MaxPool2d	kernel_size=2, stride=2
Conv2d	in_channels=64, out_channels=32, kernel_size=3, stride=1
BatchNorm2d	num_features=32
MaxPool2d	kernel_size=2, stride=2
Conv2d	in_channels=32, out_channels=32, kernel_size=3, stride=1
BatchNorm2d	num_features=32
MaxPool2d	kernel_size=2, stride=2
Conv2d	in_channels=32, out_channels=16, kernel_size=3, stride=1
BatchNorm2d	num_features=16
MaxPool2d	kernel_size=2, stride=2
Flatten	-
Linear	in_features=flattened_size, out_features=256
ReLU	-
Dropout (optional)	p=0.3 or p=0.5
Linear	in_features=256, out_features=11 (number of classes)

Table 1: Layer-wise configuration of the PlainCNN model.

PyTorch Code

The PlainCNN is implemented in PyTorch as follows:

```python
class PlainCNN(nn.Module):
    def __init__(self, num_classes=NUM_CLASSES, img_size=IMG_SIZE, dropout_prob=0.0):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3,128,5), nn.BatchNorm2d(128), nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(128,64,3), nn.BatchNorm2d(64),  nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(64,32,3),  nn.BatchNorm2d(32),  nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(32,32,3),  nn.BatchNorm2d(32),  nn.ReLU(), nn.MaxPool2d(2),
            nn.Conv2d(32,16,3),  nn.BatchNorm2d(16),  nn.ReLU(), nn.MaxPool2d(2),
        )
        with torch.no_grad():
            dummy = torch.zeros(1, 3, img_size, img_size)
            n_flat = self.features(dummy).view(1, -1).size(1)

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(n_flat, 256),
            nn.ReLU(),
            nn.Dropout(dropout_prob),
            nn.Linear(256, num_classes)
        )

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

Figure 1: PyTorch implementation of the PlainCNN model.

3.2 Residual Convolutional Neural Network (ResCNN)

The ResCNN model is an enhanced version of the PlainCNN architecture. It follows a similar overall structure but introduces a residual block to improve the flow of information through the network and help it learn better.

Residual connections are a powerful technique that allows the model to learn identity mappings easily. This helps avoid the "vanishing gradient" problem that often makes deeper networks harder to train.

Parametric Details

Similar to the PlainCNN, the ResCNN model uses five convolutional layers. The key parameters for each convolutional layer are:

- **in_channels**: Number of channels (feature maps) that the layer receives as input.
- **out_channels**: Number of channels the layer outputs after the convolution.
- **kernel_size**: Size of the filter (either 5×5 or 3×3) applied to the input.
- **stride**: Step size of the filter when moving across the input (set to 1 for all layers to keep maximum feature information).

The first three convolutional layers extract general features from the images, with a high number of output channels ($128 \rightarrow 64 \rightarrow 32$). After that, instead of stacking more convolutional layers directly, we add a residual block to learn more complex patterns without hurting the training process.

The residual block consists of:

- Two convolutional layers with `in_channels=32` and `out_channels=32`, both with a 3×3 kernel size and `padding=1`.
- A shortcut connection that adds the input of the block directly to its output.

This design allows the model to either learn a transformation or simply pass through the important features unchanged if needed.

Finally, a Max Pooling layer reduces the spatial size, and two more convolutional layers are used to further refine the learned features before passing to the classifier.

The table below shows the full architecture:

Layer	Parameters
Conv2d	in_channels=3, out_channels=128, kernel_size=5, stride=1
BatchNorm2d	num_features=128
MaxPool2d	kernel_size=2, stride=2
Conv2d	in_channels=128, out_channels=64, kernel_size=3, stride=1
BatchNorm2d	num_features=64
MaxPool2d	kernel_size=2, stride=2
Conv2d	in_channels=64, out_channels=32, kernel_size=3, stride=1
BatchNorm2d	num_features=32
MaxPool2d	kernel_size=2, stride=2
Conv2d	in_channels=32, out_channels=32, kernel_size=3, stride=1, padding=1
BatchNorm2d	num_features=32
Residual Block	2 × (Conv2d + BatchNorm2d + ReLU) with 32 channels
MaxPool2d	kernel_size=2, stride=2
Conv2d	in_channels=32, out_channels=16, kernel_size=3, stride=1
BatchNorm2d	num_features=16
MaxPool2d	kernel_size=2, stride=2
Flatten	-
Linear	in_features=flattened_size, out_features=256
ReLU	-
Dropout (optional)	p=0.3 or p=0.5
Linear	in_features=256, out_features=11 (number of classes)

Table 2: Layer-wise configuration of the ResCNN model.

Why Add Residual Connections?

As neural networks become deeper, they become harder to train. Residual connections solve this by allowing information to skip layers and flow directly through the network if needed. If a layer is not helpful, the network can easily learn to "ignore" it by setting its transformation close to zero, thanks to the residual shortcut.

This makes training faster, more stable, and improves generalization, especially when adding more layers.

In our design, we added the residual block after the fourth convolution to allow the model to better learn mid-level abstract features before final pooling and classification.

PyTorch Code

The Residual Block is implemented as:

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

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

Figure 2: PyTorch implementation of the Residual Block.

And the complete ResCNN model is:

```
class ResCNN(nn.Module):
    def __init__(self, num_classes=NUM_CLASSES, img_size=IMG_SIZE, dropout_prob=0.0):
        super().__init__()
        self.block1 = nn.Sequential(nn.Conv2d(3,128,5), nn.BatchNorm2d(128),
                                    nn.ReLU(), nn.MaxPool2d(2))
        self.block2 = nn.Sequential(nn.Conv2d(128,64,3), nn.BatchNorm2d(64),
                                    nn.ReLU(), nn.MaxPool2d(2))
        self.block3 = nn.Sequential(nn.Conv2d(64,32,3),  nn.BatchNorm2d(32),
                                    nn.ReLU(), nn.MaxPool2d(2))
        self.conv4  = nn.Sequential(nn.Conv2d(32,32,3,padding=1),
                                    nn.BatchNorm2d(32), nn.ReLU())
        self.res4   = ResBlock(32)
        self.pool4  = nn.MaxPool2d(2)
        self.block5 = nn.Sequential(nn.Conv2d(32,16,3), nn.BatchNorm2d(16),
                                    nn.ReLU(), nn.MaxPool2d(2))

        with torch.no_grad():
            dummy = torch.zeros(1,3,img_size,img_size)
            x = self.block1(dummy)
            x = self.block2(x)
            x = self.block3(x)
            x = self.pool4(self.res4(self.conv4(x)))
            x = self.block5(x)
            n_flat = x.view(1, -1).size(1)

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(n_flat, 256),
            nn.ReLU(),
            nn.Dropout(dropout_prob),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.pool4(self.res4(self.conv4(x)))
        x = self.block5(x)
        return self.classifier(x)
```

Figure 3: PyTorch implementation of the ResCNN model.

Short Summary of Design Choices:

- Residual connections allow easier learning of identity mappings, which improves training for deeper networks.
- Same initial approach as PlainCNN: start with more channels (128) and reduce as the network deepens ($64 \rightarrow 32 \rightarrow 16$).
- Batch Normalization and ReLU after every convolutional layer stabilize and accelerate learning.
- Max Pooling helps to reduce the feature size and prevent overfitting.
- Dropout is later added in experiments to regularize and improve generalization.

3.3 Activation Functions, Loss Function, and Optimization Algorithm

In this section, we explain the choice of activation functions, loss function, and optimization algorithm used during model training. Each of these components plays a critical role in the success of deep learning models. We selected commonly used and well-proven techniques to ensure stable and efficient learning.

3.3.1 Activation Function: ReLU (Rectified Linear Unit)

After every convolutional and fully connected (linear) layer—except the final output layer—we apply the **ReLU (Rectified Linear Unit)** activation function.

The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

This means it outputs the input directly if it is positive, otherwise, it outputs zero.

We chose ReLU because:

- **Avoids vanishing gradients**: Unlike traditional activation functions like sigmoid or tanh, ReLU does not saturate in the positive region, helping gradients to flow effectively through the network during backpropagation.
- **Computationally efficient**: ReLU is extremely simple and fast to compute, which speeds up the training process.
- **Sparse activations**: ReLU introduces sparsity in the network by zeroing out negative values. Sparse representations often lead to better generalization and less overfitting.

Thus, ReLU helps the network converge faster and learn more complex patterns without unnecessary computational overhead.

3.3.2 Loss Function: Cross-Entropy Loss

For training both PlainCNN and ResCNN models, we use **Cross-Entropy Loss**, implemented in PyTorch as:

$$\text{criterion} = \text{nn.CrossEntropyLoss()}$$

Cross-Entropy Loss is commonly used for *multi-class classification tasks*, where each input belongs to exactly one class. It measures the difference between the predicted probability distribution (softmax output) and the true label distribution (one-hot encoding).

The formula for Cross-Entropy Loss for a single sample is:

$$\text{Loss} = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

where:

- C is the number of classes (in our case, 11 food categories),
- y_i is the ground truth label (1 for the correct class, 0 otherwise),
- $\hat{y}_i$ is the predicted probability for class i.

We chose Cross-Entropy Loss because:

- **Natural fit for classification**: It directly penalizes incorrect predictions more heavily and rewards confident correct predictions.
- **Works well with softmax outputs**: Our final linear layer outputs raw logits, and PyTorch's CrossEntropyLoss automatically applies a softmax function internally.
- **Stable and efficient**: It leads to stable gradients and well-behaved optimization for classification tasks.

By minimizing Cross-Entropy Loss, the models learn to assign higher probabilities to the correct classes over time.

3.3.3 Optimization Algorithm: Adam Optimizer

For updating the model weights during training, we use the **Adam (Adaptive Moment Estimation)** optimizer. It is implemented as:

```
optimizer = optim.Adam(model.parameters(), lr=learning_rate, weight_decay=5e-5)
```

We chose Adam for several reasons:

- **Adaptive learning rates**: Adam computes individual learning rates for each parameter, adjusting them based on estimates of first and second moments of the gradients. This helps in faster convergence and better stability across different layers.
- **Combines advantages of AdaGrad and RMSProp**: Adam inherits the benefits of both AdaGrad (good for sparse gradients) and RMSProp (good for non-stationary objectives).
- **Requires less tuning**: Adam usually works well with default hyperparameters (learning rate = 0.001), making it a very practical and efficient choice.

Additionally, we apply a **small weight decay (L2 regularization)** to prevent the models from overfitting by penalizing excessively large weight values.

Summary of Choices:

Component	Choice	Reason
Activation Function	ReLU	Prevents vanishing gradients, computationally fast, introduces sparsity
Loss Function	Cross-Entropy Loss	Suitable for multi-class classification, stable gradients
Optimizer	Adam	Adaptive, efficient, less sensitive to hyperparameters

Table 3: Summary of key component choices and their motivations.

4 Implementation of Convolution, Fully Connected Layers, and Residual Connections

In this section, we explain how the basic building blocks of our CNN models—Convolutional Layers, Fully Connected Layers, and Residual Connections—were implemented using PyTorch. We also provide code snippets to illustrate the implementations.

4.1 Convolutional Layers

Convolutional layers are the fundamental components that allow CNNs to extract spatial features from images, such as edges, textures, and patterns.

We implement convolutional layers in PyTorch using the **nn.Conv2d** module. Each convolutional layer is usually followed by a Batch Normalization layer, a ReLU activation function, and a Max Pooling operation.

Example implementation of a convolutional block:

```
nn.Conv2d(in_channels=3, out_channels=128, kernel_size=5, stride=1),
nn.BatchNorm2d(128),
nn.ReLU(),
nn.MaxPool2d(kernel_size=2, stride=2)
```

Figure 4: Convolutional block implementation in PyTorch.

Explanation of key parameters:

- **in_channels**: Number of input feature channels (e.g., 3 for RGB images).
- **out_channels**: Number of output feature channels produced by the convolution.
- **kernel_size**: Size of the convolutional filter (5×5 or 3×3 in our case).
- **stride**: Step size while sliding the filter (set to 1).
- **padding**: In some layers, we add padding (e.g., `padding=1`) to preserve spatial dimensions.

Max Pooling layers (`nn.MaxPool2d`) are used after convolutions to downsample feature maps, reducing computational cost and making the model more robust to spatial variations.

4.2 Fully Connected Layers

Fully connected (dense) layers are used at the end of the feature extraction part to map the learned features to the output classes.

We flatten the feature maps after the final convolutional layer and pass them through two fully connected layers:

```
nn.Flatten(),
nn.Linear(in_features=n_flat, out_features=256),
nn.ReLU(),
nn.Dropout(p=dropout_prob),
nn.Linear(in_features=256, out_features=num_classes)
```

Figure 5: Fully connected (dense) layers implementation in PyTorch.

Explanation:

- **Flatten**: Converts the 3D feature maps into a 1D vector for the dense layers.
- **First Linear Layer**: Maps the high-dimensional feature vector to 256 hidden units.
- **ReLU**: Adds non-linearity to allow learning complex decision boundaries.
- **Dropout**: Randomly drops units during training (with a probability p) to prevent overfitting.
- **Second Linear Layer**: Maps 256 units to the final number of classes (11 in our dataset).

By using two fully connected layers, the network has enough capacity to model complex relationships between extracted features and class labels.

4.3 Residual Connections

Residual connections are used in the ResCNN model to allow gradients to flow more easily through the network and help the model learn identity mappings when necessary.

We implement the residual connection through a custom `ResBlock` class:

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

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

Figure 6: Implementation of Residual Block (ResBlock) in PyTorch.

Explanation:

- Two convolutional layers are applied sequentially to the input.
- After the second convolution, the input (identity) is added to the output.
- Finally, a ReLU activation is applied to the summed result.

This simple addition allows the network to easily learn to pass through important features or modify them only slightly if needed, making deeper networks easier to optimize.

5 Data Augmentation

Data augmentation is a widely used technique in deep learning to improve model generalization by artificially expanding the training dataset. Instead of collecting new data, we apply random transformations to existing images, creating slightly modified versions of the original samples.

In our project, we applied data augmentation to the training images to help the models learn more robust and invariant features.

5.1 How We Applied Augmentation

We used the following augmentation techniques on the training data:

```
train_tf = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.RandomAffine(
        degrees=0,
        translate=(0.2, 0.2),
        scale=(0.8, 1.0),
        fill=0
    ),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize([0.485,0.456,0.406],[0.229,0.224,0.225]),
])
```

Figure 7: Data augmentation transformations

Explanation of each step:

- **RandomAffine**: Applies random translation and scaling.
  - `translate=(0.2, 0.2)` allows shifting the image by up to 20% of its width and height.
  - `scale=(0.8, 1.0)` randomly scales the image between 80% and 100% of its size.

  This simulates different camera positions or zoom levels.
- **RandomHorizontalFlip**: Randomly flips the image horizontally with a 50% chance. This helps the model become invariant to left-right orientation changes.
- **Resize**: All images are resized to a uniform size of 150×150 pixels for consistency.
- **Normalization**: The pixel values are normalized using the mean and standard deviation of the ImageNet dataset. This helps in faster convergence.

For the validation and test datasets, we only applied `Resize`, `ToTensor`, and `Normalization` — no random augmentations — to ensure that evaluations are performed on the original data without artificial alterations.

5.2 Why We Apply Augmentation

There are several important reasons why data augmentation is beneficial:

- **Prevents overfitting**: Without augmentation, the model might memorize specific patterns in the training images. By presenting slightly different versions of the images at each epoch, we force the model to learn more general and meaningful features.
- **Improves generalization**: Augmented data helps the model perform better on unseen data because it learns to recognize objects despite small changes in position, scale, or orientation.
- **Simulates real-world variations**: In real-world scenarios, images are rarely perfectly aligned. Augmentation mimics natural variations in how images are captured.
- **Expands the training set size**: Although we are not adding new images, augmentation virtually increases the diversity of the dataset without requiring additional data collection.

5.3 Benefits Observed in Our Experiments

Applying data augmentation contributed to better validation and test accuracies across almost all configurations. Models trained with augmented data learned more robust features, leading to improved generalization and reduced overfitting, especially when using lower learning rates or smaller batch sizes.

Without augmentation, we observed that the models tended to overfit quickly to the training data, especially in early epochs.

Thus, even simple augmentations like affine transformations and horizontal flips provided significant improvements in model performance.

6 Training and Validation Results: Loss and Accuracy Curves

In this section, we analyze the training and validation behavior of both the PlainCNN and ResCNN models over 50 epochs under different learning rates (0.0010, 0.0005, 0.0001) and batch sizes (32 and 64).

We present graphs for:

- Training Loss
- Validation Loss
- Validation Accuracy

and discuss the trends, interpret the changes over time, and compare the two models.

6.1 Training Loss Curves

Looking at the training loss curves:

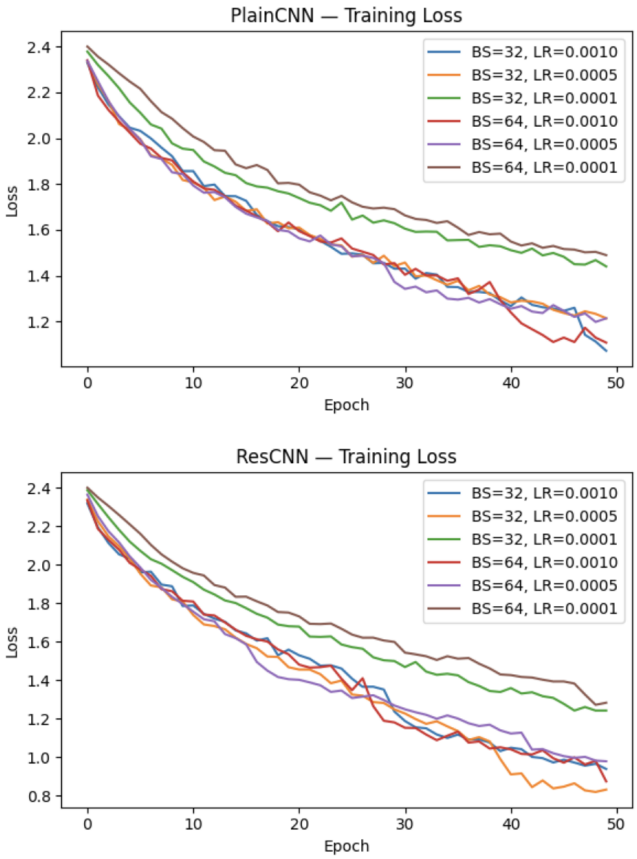

Figure 8: Training loss curves for PlainCNN and ResCNN models.

General Behavior: For both PlainCNN and ResCNN, the training loss consistently decreases over time, indicating that the models are successfully learning from the training data. Early in training (first 10-20 epochs), losses drop rapidly, while the decline becomes slower in later epochs, which is typical as models converge.

Effect of Learning Rate:

- Higher learning rates (0.0010) lead to faster initial decrease but sometimes noisier convergence.
- Lower learning rates (0.0001) show smoother but slower convergence.
- A learning rate of 0.0005 often achieves a good balance: steady convergence without too much noise.

Effect of Batch Size:

- Smaller batch size (32) sometimes results in slightly more fluctuation but often achieves better minima eventually compared to batch size 64.
- Larger batch size (64) produces smoother loss curves but may converge to slightly higher final loss.

Comparison (PlainCNN vs ResCNN): The ResCNN model consistently reaches lower training loss than the PlainCNN model for almost all learning rates and batch sizes. This shows that residual connections help the model fit the training data better, making optimization easier.

6.2 Validation Loss Curves

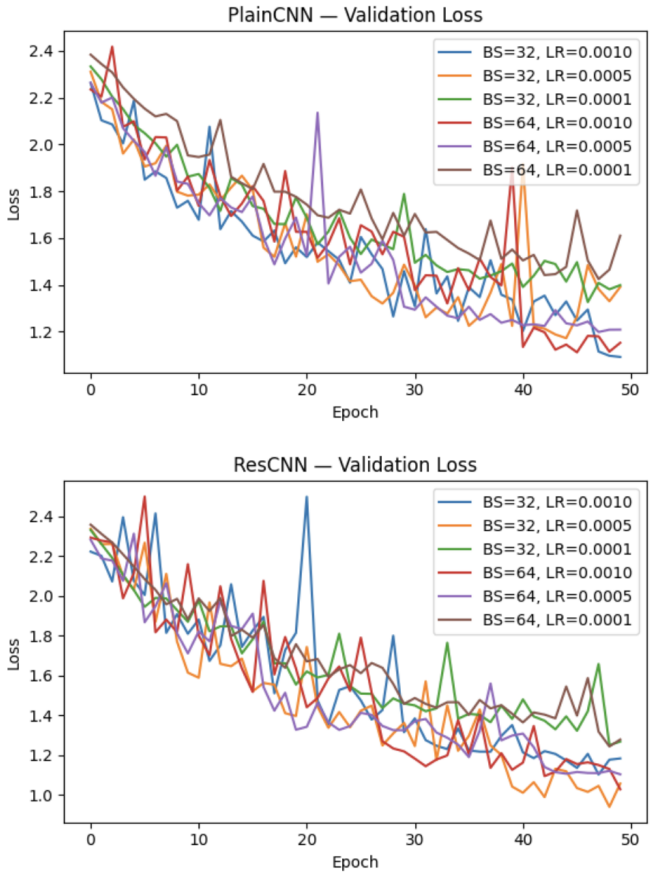

Figure 9: Validation loss curves for PlainCNN and ResCNN models.

General Behavior: Validation loss decreases for both models in the early stages but then tends to fluctuate more heavily compared to training loss. This is normal because validation loss depends on new, unseen samples and data augmentations may cause variation.

Effect of Learning Rate:

- A moderate learning rate (0.0005) typically results in the lowest validation losses.
- Very high learning rate (0.0010) can sometimes cause instability (sudden jumps in the curve).
- Very low learning rate (0.0001) results in slower learning and sometimes higher validation loss at the end.

Effect of Batch Size: Similar to training loss, batch size 32 performs slightly better in validation loss compared to batch size 64 in many cases.

Comparison (PlainCNN vs ResCNN): ResCNN generally achieves lower validation losses than PlainCNN, especially after the middle of training (after 20-30 epochs). This suggests that ResCNN generalizes better to unseen data, thanks to the stability brought by residual connections.

6.3 Validation Accuracy Curves

Analyzing validation accuracy curves:

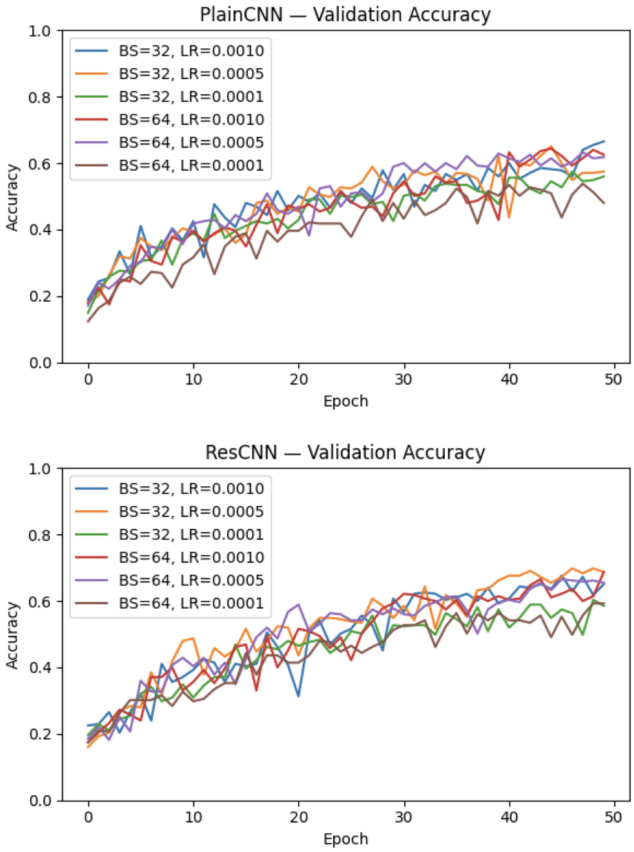

Figure 10: Validation accuracy curves for PlainCNN and ResCNN models.

General Behavior: Validation accuracy steadily increases during training for both models, although with more fluctuation compared to loss curves. Both models show a clear trend of improving accuracy over epochs.

Effect of Learning Rate:

- Learning rate 0.0005 generally yields the highest validation accuracies for both models.
- Learning rate 0.0010 also performs reasonably well but shows more unstable jumps.
- Learning rate 0.0001 improves slower and reaches lower maximum accuracy.

Effect of Batch Size: Batch size 32 consistently provides slightly better final validation accuracy than batch size 64. This is likely because smaller batches introduce more noise during gradient computation, helping the model escape shallow local minima.

Comparison (PlainCNN vs ResCNN): ResCNN consistently achieves slightly higher validation accuracy than PlainCNN, especially toward the later epochs. This demonstrates that adding residual connections not only improves fitting but also enhances generalization.

6.4 Overall Observations and Conclusions

- Models improve both loss and accuracy consistently over time, validating the effectiveness of the architectures and training strategy.
- Learning rate = 0.0005 and batch size = 32 combinations often lead to the best overall performance.
- Residual connections in ResCNN improve both training behavior and validation performance compared to PlainCNN.
- Validation loss and accuracy fluctuations are expected due to random augmentations and the small size of the dataset.
- Even though both models perform similarly at the beginning of training, ResCNN shows a clear advantage after around 20–30 epochs, reaching higher stability and final accuracy.

Thus, based on training dynamics, validation loss stabilization, and final validation accuracy, ResCNN is the stronger architecture among the two.

7 Best Model Selection and Test Accuracy

After training both the PlainCNN and ResCNN models across different hyperparameter settings, we selected the best models based on the highest validation accuracy achieved.

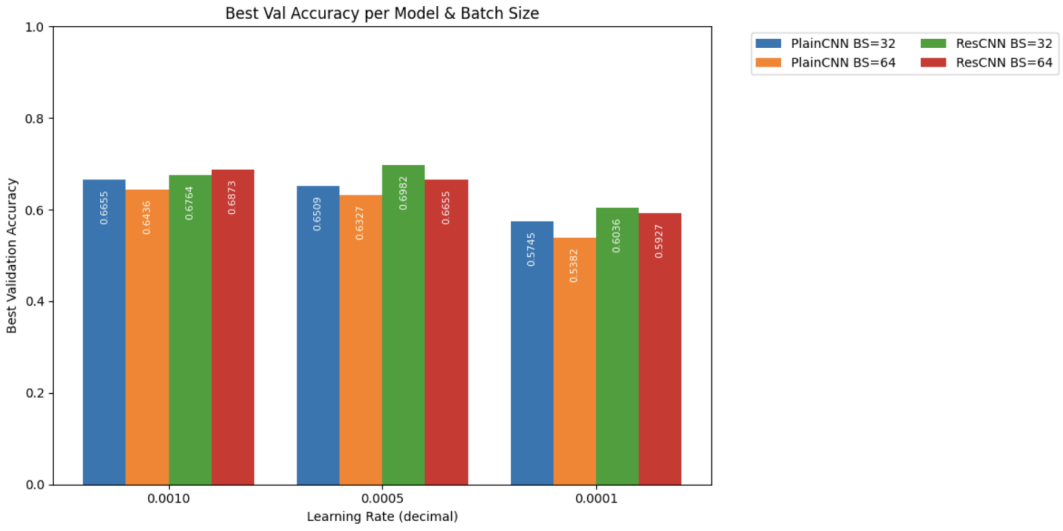

Figure 11: Best validation accuracies per model and batch size.

7.1 Best Model for PlainCNN

For the PlainCNN architecture:

- The best validation accuracy achieved was 66.55% when using a learning rate of 0.0010 and a batch size of 32.
- After selecting this model and evaluating it on the test set, we obtained a test accuracy of 53.82%.

Metric	Value
Best Validation Accuracy	66.55%
Test Accuracy	53.82%

Table 4: PlainCNN best validation and test accuracies.

Summary:
This result shows that while the PlainCNN model performs reasonably well on validation data, it struggles slightly more on completely unseen test data, likely due to overfitting or limited model capacity.

7.2 Best Model for ResCNN

For the ResCNN architecture:

- The best validation accuracy achieved was 69.82%, using a learning rate of 0.0005 and a batch size of 32.
- On the test set, this model achieved a test accuracy of 58.18%.

Metric	Value
Best Validation Accuracy	69.82%
Test Accuracy	58.18%

Table 5: ResCNN best validation and test accuracies.

Summary: Compared to PlainCNN, ResCNN achieves both higher validation and test accuracies. The addition of residual connections allows the ResCNN model to better generalize to unseen data, confirming the advantage of deeper and more stable network architectures.

7.3 Overall Observations

- In both models, using a batch size of 32 led to better results compared to batch size 64, confirming that smaller batches can introduce helpful noise into training and improve generalization.
- The ResCNN model not only fits the training and validation data better but also transfers this performance to the unseen test set more successfully.
- Residual connections significantly enhance the model's ability to learn deeper representations without causing optimization difficulties.
- Validation and test accuracies are lower than training accuracies, which is normal and expected in real-world datasets with moderate variability and relatively small training sizes.

Thus, the ResCNN model with learning rate = 0.0005 and batch size = 32 is selected as the overall best-performing model in this study.

8 Dropout Integration and Analysis

To improve the generalization ability of our best models, we integrated Dropout layers and retrained them. Dropout is a regularization technique that helps prevent overfitting by randomly "dropping" (setting to zero) a subset of neurons during training.

8.1 Where Dropout Was Added

We added Dropout after the first fully connected layer (256 units) and before the final classification layer in both models.

Example implementation in PyTorch:

```
self.classifier = nn.Sequential(
    nn.Flatten(),
    nn.Linear(n_flat, 256),
    nn.ReLU(),
    nn.Dropout(dropout_prob),
    nn.Linear(256, num_classes)
)
```

Figure 12: Dropout integration

Why Add Dropout at This Point?

- Fully connected layers typically have a large number of parameters and are more prone to overfitting compared to convolutional layers.
- Adding dropout after the dense layer forces the model to rely on distributed feature representations rather than memorizing specific neuron activations.
- This improves the robustness of the model and enhances its ability to generalize to unseen data.

Thus, inserting dropout after the dense 256-unit layer is an effective point to apply regularization without harming early feature extraction.

8.2 Experimental Results

We tested two different dropout probabilities: 0.3 and 0.5 for both PlainCNN and ResCNN.

PlainCNN with Dropout

Dropout Rate	Best Validation Accuracy	Test Accuracy
0.3	52.36%	44.00%
0.5	49.09%	46.18%

Table 6: PlainCNN results with dropout.

Observations:

- Dropout slightly reduced the validation accuracy compared to the model without dropout (66.55% $\rightarrow$ 52% or 49%).
- Test accuracy also decreased compared to the no-dropout model (53.82% $\rightarrow$ 44-46%).
- Higher dropout (0.5) led to slightly worse validation performance but slightly better test accuracy, suggesting stronger regularization at the cost of learning speed.

ResCNN with Dropout

Dropout Rate	Best Validation Accuracy	Test Accuracy
0.3	54.18%	48.00%
0.5	44.00%	42.18%

Table 7: ResCNN results with dropout.

Observations:

- For ResCNN as well, introducing dropout caused validation and test accuracies to drop compared to the no-dropout version (69.82% $\rightarrow$ 54% or 44%).
- Using a higher dropout rate (0.5) significantly harmed validation accuracy compared to 0.3.
- The model with dropout 0.3 still performed better than the model with dropout 0.5.

8.3 Overall Analysis

- In our experiments, adding dropout did not improve performance compared to the no-dropout models.
- This suggests that the original architectures, combined with data augmentation, already provided sufficient regularization for the Food-11 dataset.
- Applying dropout introduced too much randomness, possibly harming feature learning, especially since the dataset is not extremely large.
- ResCNN remains the better model even after dropout, but its margin of superiority decreases with stronger regularization.

Thus, although dropout is a powerful regularization tool, it must be carefully tuned depending on the dataset size and the baseline model performance.

9 Confusion Matrix Analysis

A confusion matrix provides a detailed breakdown of a model's performance by showing the number of correct and incorrect predictions for each class.
It allows us to analyze which classes are most confused with each other and how well the model distinguishes between different categories.
Below, we discuss the confusion matrices for the best-performing PlainCNN and ResCNN models.

9.1 PlainCNN Confusion Matrix

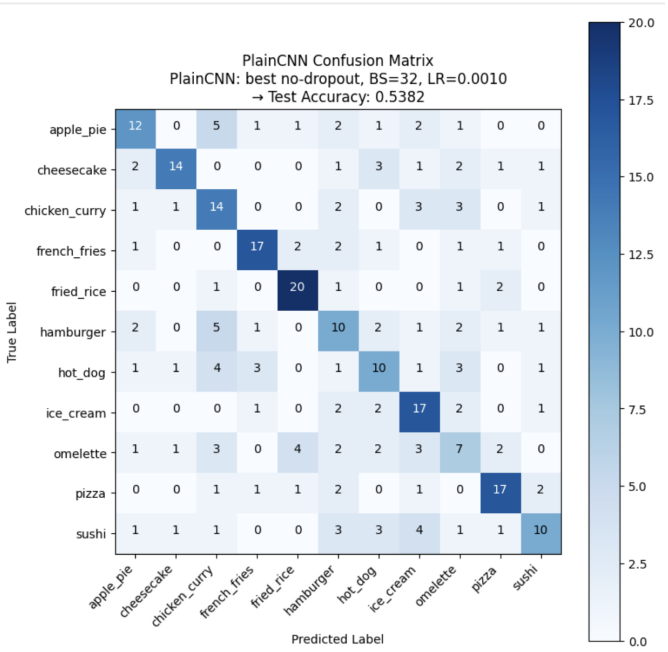

Figure 13: PlainCNN confusion matrix (BS=32, LR=0.0010, Test Accuracy=53.82%).

The confusion matrix for the best PlainCNN model (validation accuracy: 66.55%, test accuracy: 53.82%) shows the following:

Strengths:

- The model performs relatively well on classes like *fried_rice*, *ice_cream*, *pizza*, and *french_fries*.
- For example, 20 correct predictions for *fried_rice*, 17 for *ice_cream*, 17 for *pizza*, and 17 for *french_fries*.
- These classes have distinctive visual patterns that may make them easier to classify.

Weaknesses:

- Significant confusion is observed between similar food items such as:
  - *hamburger* vs. *hot_dog*,
  - *omelette* vs. *chicken_curry*,
  - *sushi* vs. *fried_rice*.
- For example, *hamburger* is often misclassified as *hot_dog* or *fried_rice*.

General Observations:

- The model has difficulty differentiating between classes that have visually similar textures, colors, or shapes.
- Misclassifications also occur for less distinctive foods like *omelette*, where the class is predicted inconsistently.

9.2 ResCNN Confusion Matrix

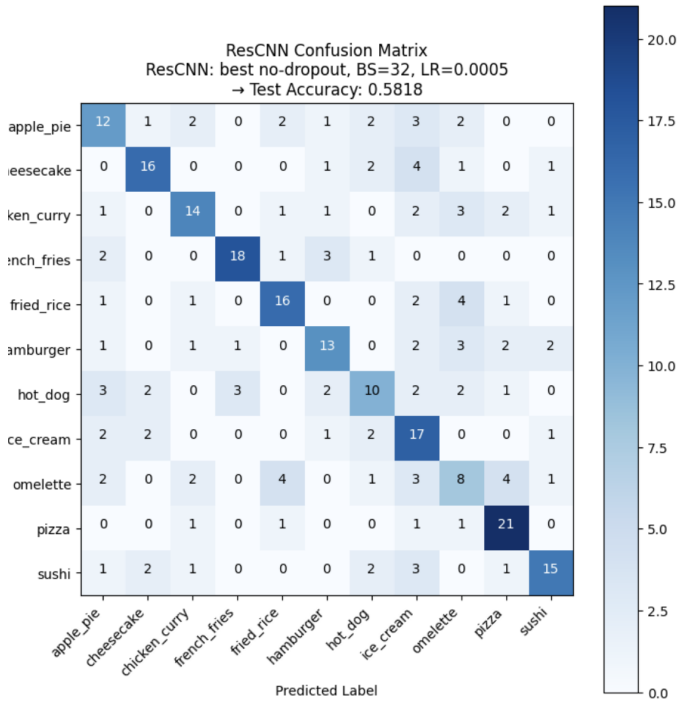

Figure 14: ResCNN confusion matrix (BS=32, LR=0.0005, Test Accuracy=58.18%).

The confusion matrix for the best ResCNN model (validation accuracy: 69.82%, test accuracy: 58.18%) shows notable improvements:

Strengths:

- The ResCNN model correctly classifies even more samples compared to PlainCNN.
- *french_fries*, *ice_cream*, and *pizza* again have high true positive counts:
  - *french_fries*: 18 correct,
  - *ice_cream*: 17 correct,
  - *pizza*: 21 correct (higher than PlainCNN).
- *Cheesecake* and *chicken_curry* classes also show improved prediction counts.

Weaknesses:

- Although reduced, some confusion still exists between similar-looking classes:
  - *hamburger* and *hot_dog*,
  - *sushi* and *fried_rice*.
- However, the misclassification rate is lower compared to the PlainCNN model.

General Observations: The addition of residual connections allows the ResCNN model to learn deeper and more discriminative features, resulting in a better separation between classes in the feature space.

9.3 Overall Analysis

Aspect	PlainCNN	ResCNN
Strength on distinctive classes	Moderate	High
Confusion between similar classes	High	Reduced
Overall prediction consistency	Moderate	Improved
Test Accuracy	53.82%	58.18%

Table 8: Comparison between PlainCNN and ResCNN based on confusion matrices.

The confusion matrix analysis confirms our earlier findings:

- ResCNN provides better generalization and class separation compared to PlainCNN.
- Residual connections help in minimizing confusion among similar classes.
- However, there is still room for improvement, particularly for fine-grained food categories where visual similarities are significant.

Thus, the ResCNN model is more reliable and performs better across almost all classes compared to PlainCNN.

10 Transfer Learning and Fine-Tuning

In this section, we explore the concept of transfer learning and describe how we fine-tuned a pre-trained model (MobileNetV2) to adapt it to the Food-11 dataset.

10.1 What is Fine-Tuning?

Fine-tuning is a common technique in deep learning where a model that has already been trained on a large dataset (such as ImageNet) is further trained on a smaller, target dataset for a new, often related task.

Instead of training the entire model from scratch, we use the pre-trained weights as a strong starting point.

This leverages the feature extraction capabilities the model has already learned, allowing faster convergence and often better performance, especially when the target dataset is small.

Fine-tuning can involve:

- Training only the newly added layers,
- Training a subset of layers,
- Or retraining the entire model with a smaller learning rate.

10.2 Why Should We Do Fine-Tuning?

We apply fine-tuning because:

- **Better feature representations:** Models pre-trained on massive datasets (like ImageNet) have learned generic features such as edges, textures, and patterns that are useful across different vision tasks.
- **Reduces the need for large datasets:** Collecting and labeling large datasets is expensive and time-consuming. Fine-tuning allows good performance even with smaller datasets.
- **Speeds up training:** Since the model already has useful weights, we can reach high accuracy much faster than training from scratch.
- **Prevents overfitting:** By starting with meaningful features, the model focuses on adapting to the new classes rather than trying to learn from random initialization, which is especially important when data is limited.

Thus, fine-tuning gives us a strong advantage in both accuracy and training efficiency.

10.3 Why Do We Freeze the Pre-trained Layers and Train Only FC Layers?

When adapting a pre-trained model to a new task (especially if the new dataset is small), it is common to freeze the earlier layers and train only the fully connected (FC) layers at first.

The reasons are:

- **Generic features are transferable:** Early layers of CNNs typically learn very general patterns (e.g., edges, corners, textures) that are useful for many tasks. Therefore, there is no need to re-learn them.
- **Avoids overfitting:** If we update all layers with a small dataset, the model could easily overfit. Freezing the feature extractor part prevents this.
- **Reduces computation and training time:** Training fewer parameters (only FC layers) makes the process much faster and less memory-intensive.

Thus, initially training only the FC layers allows the model to specialize in the new classification task without disturbing the strong generic representations already learned.

10.4 Relevant Code Snippet

The following PyTorch code demonstrates how we froze all pre-trained layers and modified the classifier head for Food-11:

```python
# 1) Load pre-trained MobileNetV2
weights = models.MobileNet_V2_Weights.DEFAULT
model_fc = models.mobilenet_v2(weights=weights).to(device)

# 2) Freeze all params
for p in model_fc.parameters():
    p.requires_grad = False

# 3) Re-shape the existing FC head to output NUM_CLASSES
in_feats               = model_fc.classifier[1].in_features
model_fc.classifier[1] = nn.Linear(in_feats, NUM_CLASSES).to(device)

# 4) Ensure FC head is trainable
for p in model_fc.classifier[1].parameters():
    p.requires_grad = True

# 5) Optimizer over only that layer
optimizer_fc = optim.Adam(
    model_fc.classifier[1].parameters(),
    lr=LRS['fc_only'],
    weight_decay=WEIGHT_DECAY
)
```

Figure 15: Transfer learning setup: Freezing MobileNetV2 and training the final classifier layer.

Explanation:

- We load a MobileNetV2 model pre-trained on ImageNet.
- We freeze all layers (`requires_grad=False`).
- We replace the final classifier layer to match the number of Food-11 classes (11 outputs).
- We define an optimizer to update only the new classifier parameters.

This setup allows the model to retain useful low- and mid-level features and adapt only the final decision layer to the new dataset.

11 Transfer Learning Experiments: Fine-Tuning Strategies

In this section, we explore two different fine-tuning strategies using the pre-trained MobileNetV2 model on the Food-11 dataset:

- Training only the fully connected (FC) layer, freezing all other layers.
- Training the last two convolutional layers along with the FC layer, freezing the rest of the network.

We compare their performance based on validation and test accuracies and analyze the results.

11.1 Case 1: Train Only FC Layer

In the first strategy, we freeze all the layers of the MobileNetV2 feature extractor and train only the final classification layer, which we replace with a new fully connected layer to output 11 classes.

Relevant PyTorch code:

```python
# 1) Load pre-trained MobileNetV2
weights = models.MobileNet_V2_Weights.DEFAULT
model_fc = models.mobilenet_v2(weights=weights).to(device)

# 2) Freeze all params
for p in model_fc.parameters():
    p.requires_grad = False

# 3) Re-shape the existing FC head to output NUM_CLASSES
in_feats               = model_fc.classifier[1].in_features
model_fc.classifier[1] = nn.Linear(in_feats, NUM_CLASSES).to(device)

# 4) Ensure FC head is trainable
for p in model_fc.classifier[1].parameters():
    p.requires_grad = True

# 5) Optimizer over only that layer
optimizer_fc = optim.Adam(
    model_fc.classifier[1].parameters(),
    lr=LRS['fc_only'],
    weight_decay=WEIGHT_DECAY
)

best_val_fc, best_w_fc = 0.0, None
for ep in range(1, EPOCHS+1):
    tr_loss, tr_acc = train_epoch(model_fc, train_loader, criterion, optimizer_fc)
    va_loss, va_acc = eval_epoch(model_fc,   val_loader,   criterion)
    if va_acc > best_val_fc:
        best_val_fc, best_w_fc = va_acc, copy.deepcopy(model_fc.state_dict())
    if ep % 10 == 0:
        print(f"[FC-only] Ep{ep:02d} — tr_acc={tr_acc:.4f}, val_acc={va_acc:.4f}")

# 7) Evaluate on test
model_fc.load_state_dict(best_w_fc)
te_loss_fc, te_acc_fc = eval_epoch(model_fc, test_loader, criterion)
print(f"\n→ FC-only: best_val_acc={best_val_fc:.4f}, test_acc={te_acc_fc:.4f}\n")
```

Figure 16: Training only the fully connected layer (FC-only).

Results for FC-only training:

Metric	Value
Best Validation Accuracy	82.91%
Test Accuracy	73.82%

Table 9: Performance of FC-only training.

11.2 Case 2: Train Last Two Convolutional Layers + FC Layer

In the second strategy, we unfreeze the last two convolutional blocks of MobileNetV2 in addition to the final fully connected layer. This allows the model to fine-tune more high-level features that are closer to the classification task.

Relevant PyTorch code:

```python
# 1) Reload fresh pre-trained model
model_ft = models.mobilenet_v2(weights=weights).to(device)
for p in model_ft.parameters():
    p.requires_grad = False

# 2) Unfreeze the last two convolutional blocks
#    (MobileNetV2.features is a Sequential)
for block in model_ft.features[-2:]:
    for p in block.parameters():
        p.requires_grad = True

# 3) Re-shape the existing FC head to output NUM_CLASSES
in_feats               = model_ft.classifier[1].in_features
model_ft.classifier[1]  = nn.Linear(in_feats, NUM_CLASSES).to(device)
for p in model_ft.classifier[1].parameters():
    p.requires_grad = True

# 4) Optimizer over all params needing grad
optimizer_ft = optim.Adam(
    filter(lambda p: p.requires_grad, model_ft.parameters()),
    lr=LRS['last2+fc'],
    weight_decay=WEIGHT_DECAY
)

# 5) Training loop
best_val_ft, best_w_ft = 0.0, None
for ep in range(1, EPOCHS+1):
    tr_loss, tr_acc = train_epoch(model_ft, train_loader, criterion, optimizer_ft)
    va_loss, va_acc = eval_epoch(model_ft,   val_loader,   criterion)
    if va_acc > best_val_ft:
        best_val_ft, best_w_ft = va_acc, copy.deepcopy(model_ft.state_dict())
    if ep % 10 == 0:
        print(f"[last2+FC] Ep{ep:02d} - tr_acc={tr_acc:.4f}, val_acc={va_acc:.4f}")

# 6) Evaluate on test
model_ft.load_state_dict(best_w_ft)
te_loss_ft, te_acc_ft = eval_epoch(model_ft, test_loader, criterion)
print(f"\n→ last2+FC: best_val_acc={best_val_ft:.4f}, test_acc={te_acc_ft:.4f}\n")
```

Figure 17: Training the last two convolutional layers and the fully connected layer (Last2+FC).

Results for Last2+FC training:

Metric	Value
Best Validation Accuracy	88.36%
Test Accuracy	74.55%

Table 10: Performance of Last2+FC training.

11.3 Comparison and Analysis

Strategy	Best Validation Accuracy	Test Accuracy	Observations
FC-only	82.91%	73.82%	Faster to train, good results, limited feature adaptation
Last2+FC	88.36%	74.55%	Higher accuracy, better feature adaptation

Table 11: Comparison of fine-tuning strategies.

Key Observations:

- Training only the FC layer achieved strong performance quickly.
  Since early and mid-level features (edges, textures) are transferable, the model performed well by adapting only the classifier.
- Training last two convolutional layers + FC further improved both validation and test accuracies.
  Fine-tuning high-level feature representations allowed the model to better adapt to the specific characteristics of food images in the Food-11 dataset.
- Validation and test performance both improved when unfreezing more layers, but the improvement was more notable on validation data (around 5% higher validation accuracy).
- Training time for the Last2+FC strategy was longer than FC-only training, but the performance gains justified the additional cost.

Thus, unfreezing the last two convolutional blocks in addition to the FC layer proved to be the most effective fine-tuning strategy for our task.

12 Confusion Matrix Analysis for Best Transfer Learning Model

After fine-tuning the last two convolutional layers and the fully connected layer of the pre-trained MobileNetV2 model, we achieved the best results in transfer learning.

The best model achieved:

- Validation Accuracy: 88.36%
- Test Accuracy: 74.55%

The corresponding confusion matrix for the test set predictions is shown below.

12.1 Analysis of Confusion Matrix

The confusion matrix shows how well the model classified each of the 11 food categories.

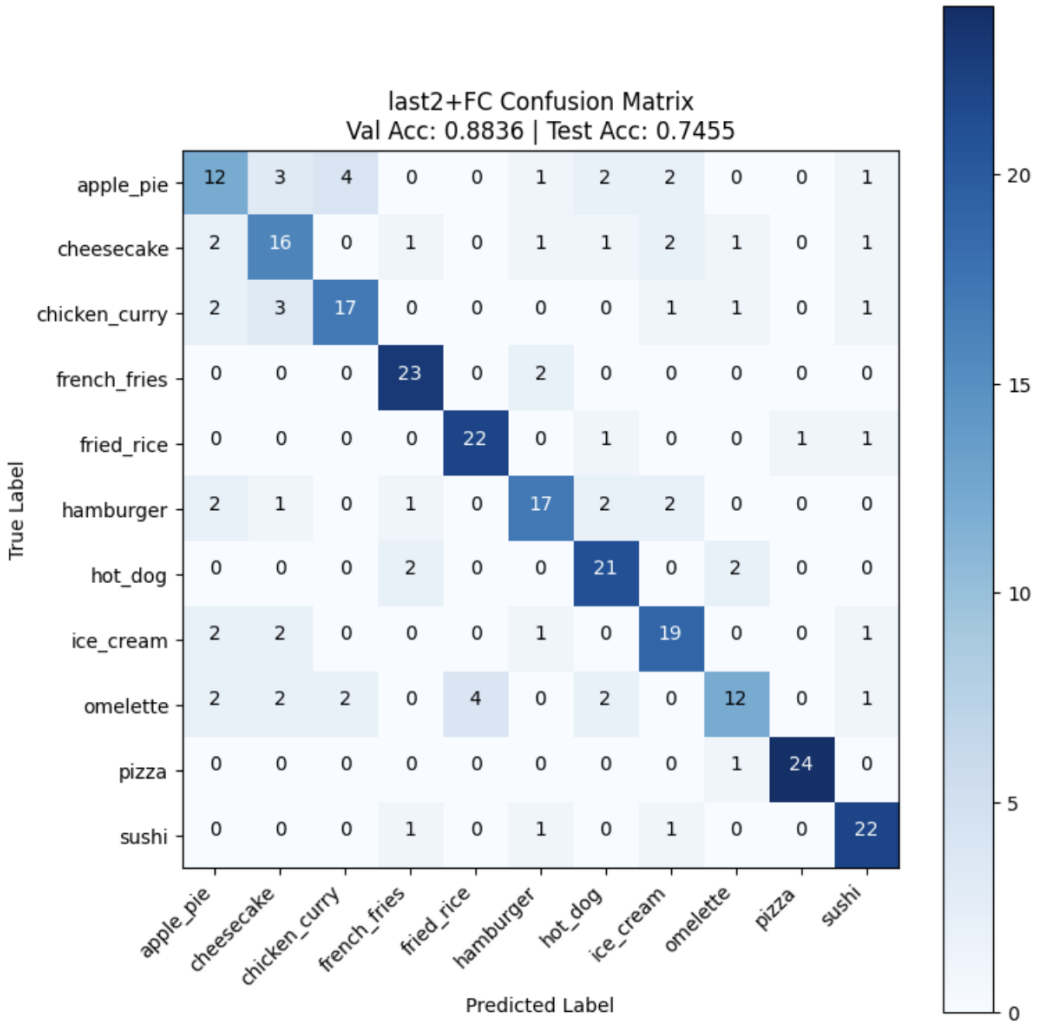

Figure 18: Confusion matrix for fine-tuned MobileNetV2 (Last2+FC).

We can derive the following observations:

- **High Correct Classification:** The model shows strong diagonal dominance, meaning most predictions are correct. Especially high correct counts are observed for:

  - french_fries (23 correct),
  - fried_rice (22 correct),
  - pizza (24 correct),
  - sushi (22 correct),
  - hot_dog (21 correct),
  - ice_cream (19 correct).

  These categories have distinctive features (color, structure) that the model can recognize reliably.

- **Some Class Confusion:** Minor confusion is observed among certain visually similar classes:
  - apple_pie is sometimes misclassified as chicken_curry or ice_cream.
  - hamburger has some misclassifications as hot_dog and french_fries.
  - omelette is harder to classify correctly and is confused with chicken_curry or ice_cream.
- **Comparison to Part 1 Models:** Compared to the best models from Part 1 (PlainCNN and ResCNN trained from scratch), this fine-tuned model demonstrates:
  - Higher correct counts across almost all classes,
  - Fewer misclassifications, especially among similar-looking foods,
  - Overall better balance between different food categories.

12.2 Overall Observations

- The fine-tuned MobileNetV2 model successfully adapts to the Food-11 dataset by leveraging pre-trained features and selectively retraining deeper layers.
- Transfer learning allows the model to achieve significantly higher performance than training from scratch, especially in recognizing fine-grained food categories.
- Some confusion remains for foods with similar visual textures or blended backgrounds, which is a common limitation even for advanced CNNs.

Thus, fine-tuning the last two convolutional blocks and the FC layer results in strong and reliable classification performance for the Food-11 image classification task.

13 Comparison and Analysis of Results Between Part 1 and Part 2

In this section, we compare the performances of models trained from scratch (PlainCNN and ResCNN) with the models obtained through transfer learning using a pre-trained MobileNetV2 architecture.

13.1 Overall Performance Comparison

Model	**Best Validation Accuracy**	**Best Test Accuracy**
PlainCNN (from scratch)	66.55%	53.82%
ResCNN (from scratch)	69.82%	58.18%
MobileNetV2 (train only FC)	82.91%	73.82%
MobileNetV2 (train last2+FC)	88.36%	74.55%

Table 12: Overall performance comparison between Part 1 and Part 2 models.

13.2 Key Observations

- **Transfer learning significantly outperforms training from scratch:**
  - Both MobileNetV2-based models achieved much higher validation and test accuracies compared to PlainCNN and ResCNN trained from scratch.
  - This demonstrates the strength of using pre-trained models that have already learned rich and generalizable features from large-scale datasets like ImageNet.
- **Fine-tuning more layers leads to better results:**
  - Training only the FC layer already provided strong performance.
  - Additionally fine-tuning the last two convolutional blocks improved validation and test accuracies even further (88.36% and 74.55%, respectively).
  - This shows that adapting higher-level features specifically for the Food-11 dataset is beneficial.
- **Training from scratch requires much more effort:**
  - Both PlainCNN and ResCNN required 50 epochs to reach moderate accuracies ($\sim$53–58% test accuracy).
  - In contrast, the fine-tuned models achieved higher performance in fewer epochs and with fewer trainable parameters.
- **Residual connections helped when training from scratch:**
  - The ResCNN model performed better than PlainCNN in Part 1.
  - Residual connections helped deeper architectures to train effectively even from random initialization, but the overall performance was still limited compared to transfer learning.
- **Generalization gap:**
  - All models showed a small drop from validation accuracy to test accuracy, which is normal.
  - However, transfer learning models had a smaller generalization gap, indicating better robustness to unseen data.

13.3 Final Conclusion

Based on the experiments:

- Transfer learning is a far more effective strategy than training CNNs from scratch when the dataset size is moderate and the task is visually similar to the source task (ImageNet).
- Fine-tuning the last two convolutional blocks, in addition to the classifier, produced the best-performing model for the Food-11 classification task.
- Pre-trained models not only converge faster but also achieve higher final performance with less risk of overfitting.

Thus, transfer learning, especially with careful layer unfreezing and fine-tuning, is the most practical and powerful approach for image classification tasks like Food-11.