Assignment 2 Report

Süleyman Yolcu
2210765016

1 Introduction

Image classification with convolutional neural networks (CNNs) can be approached either by training a model from scratch or by fine-tuning a pretrained model. In this assignment, we explore both approaches on a food image dataset with 11 classes (e.g. apple pie, sushi, etc.). **Part 1** covers the design and training of two CNN models from scratch: a plain 5-layer CNN and a modified version with residual connections. **Part 2** applies transfer learning using MobileNetV2, comparing two fine-tuning strategies. We report model architectures, training procedures, hyperparameter experiments, and results including accuracy/loss curves, confusion matrices, and analysis. Transfer learning is particularly useful when the dataset is limited and we expect the pretrained MobileNetV2 to outperform the scratch models. All results are evaluated in terms of classification accuracy on validation and test sets, and we discuss overfitting, regularization (dropout), and the impact of residual connections and pretrained features.

2 Part 1: CNN Classifier Trained from Scratch

2.1 Model Architecture

We implemented two CNN architectures in PyTorch: (1) a *plain CNN* with five convolutional blocks, and (2) a *residual CNN* that integrates residual connections in the final convolutional stages. Both models include standard components: each convolution is followed by batch normalization and a ReLU activation; max pooling is used for downsampling after specific layers; and a fully connected classifier produces the output. Dropout is used for regularization before the final classification layer, with a tunable probability p.

For the **Plain CNN**, we used a sequential block-based design. Each block consists of a 3×3 convolution, batch normalization, ReLU activation, and optional 2×2 max pooling. We progressively increase the channel size as follows: $3 \to 32 \to 64 \to 128 \to 256 \to 512$. After five convolutional blocks, we flatten the output and apply two fully connected layers with dropout in between. This structure results in 5 convolutional layers followed by a two FC layer.

Table 1: Layer–wise configuration and parameter counts for the **PlainCNN5**. All convolutions use a 3×3 kernel, **stride 1**, and padding 1; "MP" denotes 2×2 max-pooling. Input size is $3 \times 224 \times 224$.

Stage / Layer	**Output size**	**Kernel / OP**	**#Params**	**ΣParams**
Conv1 + BN + ReLU + MP	$32 \times 112 \times 112$	3×3 / 32	928	928
Conv2 + BN + ReLU + MP	$64 \times 56 \times 56$	3×3 / 64	18 560	19 488
Conv3 + BN + ReLU + MP	$128 \times 28 \times 28$	3×3 / 128	73 984	93 472
Conv4 + BN + ReLU + MP	$256 \times 14 \times 14$	3×3 / 256	295 424	388 896
Conv5 + BN + ReLU + MP	$512 \times 7 \times 7$	3×3 / 512	1 180 672	1 569 568
Flatten	25 088	–	0	1 569 568
FC_1 + ReLU + Dropout	256	$25\,088 \times 256$	6 422 784	7 992 352
FC_2 (Logits)	11	256×11	2 827	7 995 179
Total	–	–	**7.99 M**	

A simplified code version of the architecture is shown below: (For *ConvBNReLU* implementation see Appendix A.)

```
class PlainCNN5(nn.Module):
    def __init__(self, n_classes=11, p_drop=0.3):
        super().__init__()
        self.features = nn.Sequential(
            ConvBNReLU(3, 32, pool=True),    # 224 → 112
            ConvBNReLU(32, 64, pool=True),   # 112 → 56
            ConvBNReLU(64, 128, pool=True), # 56 → 28
            ConvBNReLU(128, 256, pool=True),# 28 → 14
            ConvBNReLU(256, 512, pool=True) # 14 → 7
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512 * 7 * 7, 256),  # 25088 input features
            nn.ReLU(inplace=True),
            nn.Dropout(p_drop),
            nn.Linear(256, n_classes),
        )

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

The **Residual CNN** maintains the first three convolutional blocks as in the plain model, then replaces the final two conv layers with residual blocks. Each residual block consists of two 3×3 convolutions (with batch norm and ReLU), and includes an identity or projection shortcut connection. This design enables learning of residual functions and improves gradient flow in deeper architectures. Max pooling is applied after each residual block to maintain spatial downsampling. The final fully connected layers are kept the same.

Table 2: Layer–wise configuration and parameter counts for the **ResCNN5**. Residual blocks (RB) contain two 3×3 convolutions; "Proj 1×1" is the skip-projection when input/output channels differ.

Stage / Layer	**Output size**	**Kernel / OP**	**#Params**	**ΣParams**
Conv1 + BN + ReLU + MP	$32 \times 112 \times 112$	3×3 / 32	928	928
Conv2 + BN + ReLU + MP	$64 \times 56 \times 56$	3×3 / 64	18 560	19 488
Conv3 + BN + ReLU + MP	$128 \times 28 \times 28$	3×3 / 128	73 984	93 472
Residual Block 1 (in 128, out 256)				
Conv1 + BN + ReLU	$256 \times 28 \times 28$	3×3 / 256	295 424	388 896
Conv2 + BN	$256 \times 28 \times 28$	3×3 / 256	590 336	979 232
Proj 1×1	$256 \times 28 \times 28$	1×1 / –	32 768	1 012 000
MP (downsample)	$256 \times 14 \times 14$	2×2	0	1 012 000
Residual Block 2 (in 256, out 512)				
Conv1 + BN + ReLU	$512 \times 14 \times 14$	3×3 / 512	1 180 672	2 192 672
Conv2 + BN	$512 \times 14 \times 14$	3×3 / 512	2 360 320	4 553 -———
Proj 1×1	$512 \times 14 \times 14$	1×1 / –	131 072	4 684 064
MP (downsample)	$512 \times 7 \times 7$	2×2	0	4 684 064
Flatten	25 088	–	0	4 684 064
FC_1 + ReLU + Dropout	256	$25\,088 \times 256$	6 422 784	11 106 848
FC_2 (Logits)	11	256×11	2 827	11 109 675
Total	–	–	**11.11 M**	

Below is a simplified code version of the architecture: (For *ResidualBlock* implementation see Appendix B.)

```
class ResCNN5(nn.Module):
    def __init__(self, n_classes=11, p_drop=0.3):
        super().__init__()
        self.features = nn.Sequential(
            ConvBNReLU(3, 32, pool=True),      # 224 → 112
            ConvBNReLU(32, 64, pool=True),     # 112 → 56
            ConvBNReLU(64, 128, pool=True),    # 56 → 28
            ResidualBlock(128, 256),
            nn.MaxPool2d(2),                   # 28 → 14
            ResidualBlock(256, 512),
            nn.MaxPool2d(2),                   # 14 → 7
        )
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(512 * 7 * 7, 256),
            nn.ReLU(inplace=True),
            nn.Dropout(p_drop),
            nn.Linear(256, n_classes),
        )

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

In both models, the number of trainable parameters is approximately 8M for the plain CNN and 11M for the residual CNN. The residual connections help improve training stability and generalization, especially in deeper architectures, while the plain CNN serves as a strong baseline for comparison.

2.2 Training Setup

- **Optimization:** We trained both CNNs from scratch using the Adam optimizer (with default momentum terms $\beta_1 = 0.9$, $\beta_2 = 0.999$). Adam was chosen for its efficient convergence properties on a variety of network architectures. The loss function is multi-class cross-entropy loss (categorical cross-entropy), appropriate for our classification task with one-hot labels. We monitored the classification accuracy as our primary metric. After each epoch, accuracy is computed as the number of correct predictions divided by the total, for both training and validation sets.

- **Learning Rate Schedule:** Rather than a fixed learning rate, we employed a cosine annealing learning rate scheduler (`torch.optim.lr_scheduler.CosineAnnealingLR`). This scheduler starts with an initial learning rate and then gradually **anneals** it toward a minimum value following a cosine curve over the course of training. The idea is that the learning rate will decrease smoothly to help the model converge as training progresses, possibly avoiding oscillations and local minima issues by taking progressively smaller steps. We set the scheduler's $T_{\max}$ to the total number of epochs (50), so that the learning rate starts at its initial value and decays to a small value by epoch 50. No restarts were used in the scheduler (i.e., we used a single cosine decay cycle without resets).

- **Data Augmentation:** To improve generalization, we applied basic data augmentations to the training images. These included random horizontal flips, random rotations, and random crops. Augmentation increases the effective size and diversity of the training dataset, helping the models become more robust to variations. For example, random flips make the model invariant to image mirroring (which is plausible for food images), and random slight rotations/crops simulate different camera angles or zooms. The validation and test data, however, were not augmented (only standard resizing/preprocessing), to evaluate performance on true unseen data.

- **Batch Size:** We experimented with two batch sizes, 32 and 64, in training. Smaller batches can introduce more noise in gradient updates but might generalize better, while larger batches can offer more stable gradients at the cost of potential generalization drop. We wanted to see if batch size had a significant impact on training dynamics or final accuracy in our setting.

- **Training Procedure:** Each model was trained for 50 epochs for each hyperparameter configuration. At each epoch, we computed training loss and accuracy (averaged over batches) and also evaluated the model on the

validation set to obtain validation loss and accuracy. The model parameters were saved for the best validation accuracy achieved. We also employed dropout (when enabled) during training; at test time, dropout is disabled (using the trained weights as-is). The entire training was implemented in PyTorch with careful tracking of metrics for analysis.

The complete augmentation pipeline and data-loader utility are shown in Listing 1. Listings 2–3 provide the core training loop, including the accuracy computation and the `Adam`+CosineAnnealingLR optimization used in all experiments.

```
IMAGENET_MEAN = (0.485, 0.456, 0.406)
IMAGENET_STD  = (0.229, 0.224, 0.225)

train_tf = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize(IMAGENET_MEAN, IMAGENET_STD),
])

eval_tf = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(IMAGENET_MEAN, IMAGENET_STD),
])

def get_loaders(batch_size: int = 32, num_workers: int = 4, pin=True):
    root = Path('/content/drive/MyDrive/food11')
    train_ds = datasets.ImageFolder(root/'train',      transform=train_tf)
    val_ds   = datasets.ImageFolder(root/'validation', transform=eval_tf)
    test_ds  = datasets.ImageFolder(root/'test',       transform=eval_tf)
    return {
        'train': DataLoader(train_ds, batch_size, shuffle=True,
                            num_workers=num_workers, pin_memory=pin),
        'val'  : DataLoader(val_ds,   batch_size, shuffle=False,
                            num_workers=num_workers, pin_memory=pin),
        'test' : DataLoader(test_ds,  batch_size, shuffle=False,
                            num_workers=num_workers, pin_memory=pin),
    }
```

Listing 1: Dataset loaders and augmentation pipelines (train/val/test).

```python
def epoch_loop(model, loader, criterion, optimiser=None):
    train_mode = optimiser is not None          # flag: train or eval
    model.train(train_mode)

    total_loss, correct, total = 0.0, 0, 0
    for x, y in loader:
        x, y = x.to(DEVICE), y.to(DEVICE)

        if train_mode:
            optimiser.zero_grad()

        with torch.set_grad_enabled(train_mode):
            logits = model(x)
            loss   = criterion(logits, y)
            if train_mode:
                loss.backward()
                optimiser.step()

        total_loss += loss.item() * x.size(0)
        correct    += (logits.argmax(1) == y).sum().item()
        total      += x.size(0)

    avg_loss = total_loss / total
    acc      = correct / total                  # ← accuracy metric
    return avg_loss, acc
```

Listing 2: Core training/validation loop with accuracy calculation.

```python
def fit(model, loaders, *, epochs=50, lr=1e-3, tag='?', bs=0):
    criterion = nn.CrossEntropyLoss()
    optimiser = torch.optim.Adam(model.parameters(), lr=lr)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimiser, T_max=epochs)

    best_state, best_val = None, 0.0
    history = {'train': [], 'val': []}

    for ep in range(1, epochs + 1):
        tr_loss, tr_acc = epoch_loop(model, loaders['train'],
                                     criterion, optimiser)
        v_loss,  v_acc  = epoch_loop(model, loaders['val'],
                                     criterion)

        scheduler.step()                # cosine LR decay
        history['train'].append({'loss': tr_loss, 'acc': tr_acc})
        history['val'  ].append({'loss': v_loss, 'acc': v_acc})

        if v_acc > best_val:            # checkpoint best weights
            best_state, best_val = model.state_dict(), v_acc

        if ep % 10 == 0 or ep == epochs:
            print(f"[{tag}] ep={ep:02}/{epochs} "
                  f"| train {tr_acc:.2%}/{tr_loss:.3f} "
                  f"| val {v_acc:.2%}/{v_loss:.3f}")

    model.load_state_dict(best_state)    # restore best epoch
    return history, best_val
```

Listing 3: Training wrapper (`fit`) using Adam and CosineAnnealingLR.

2.3 Hyperparameter Experiments

We performed a grid search over learning rates and batch sizes for each model. In particular, we tried **three learning rates:** 1×10^{-3}, 3×10^{-4}, and 1×10^{-4}, and **two batch sizes:** 32 and 64. This resulted in $3 \times 2 = 6$ configurations for the plain CNN, and the same 6 for the residual CNN. All other settings (optimizer, scheduler, epochs=50, etc.) were kept constant.

During training, we observed the learning curves (accuracy and loss) for both training and validation sets to understand the effect of these hyperparameters. Figures 1 and 2 show the training and validation accuracy curves, respectively, for the plain CNN under each of the 6 hyperparameter combinations. Similarly, Figures 3 and 4 show the curves for the residual CNN. In all cases, epoch 0 corresponds to the start of training and epoch 50 the end.

The exhaustive 3×2 grid is implemented in Listing 4; each configuration runs `fit` for 50 epochs with a cosine–annealed learning rate. Listing 5 shows the driver used to launch both searches and extract the highest-validation-accuracy checkpoints that are later analysed in Sections 2.4–2.6.

```python
# search space: three learning rates × two batch sizes
LR_CANDIDATES = (1e-3, 3e-4, 1e-4)
BS_CANDIDATES = (32, 64)

def run_grid(ModelCls, tag, *, epochs=50, p_drop=0.0):
    """Train `ModelCls` for all LR/BS combos; return a list of result dicts."""
    results = []
    for lr, bs in itertools.product(LR_CANDIDATES, BS_CANDIDATES):

        loaders, _ = get_loaders(batch_size=bs)       #  see Listing \ref{lst:loaders}
        model = ModelCls(p_drop=p_drop).to(DEVICE)

        # optional cosine scheduler wrapped as a lambda for clarity
        sched_fn = lambda opt, e: torch.optim.lr_scheduler.CosineAnnealingLR(opt, e)

        print(f">> {tag.upper()}  bs={bs}  lr={lr:.0e}  "
              f"params={count_params(model):.2f}M")

        history, val_acc = fit(model, loaders, epochs=epochs, lr=lr,
                               scheduler_fn=sched_fn, tag=tag, bs=bs)

        # keep all metadata for later analysis / plotting
        results.append(dict(
            model   = tag,          # plain | res
            bs      = bs,
            lr      = lr,
            val_acc = val_acc,
            history = history,
            state   = copy.deepcopy(model.state_dict())
        ))

        # tidy up GPU memory between runs
        del model, loaders
        torch.cuda.empty_cache(); gc.collect()

    return results
```

Listing 4: Grid–search driver that trains every {`lr`, `bs`} pair for a given model class. Validation accuracy is returned for model selection.

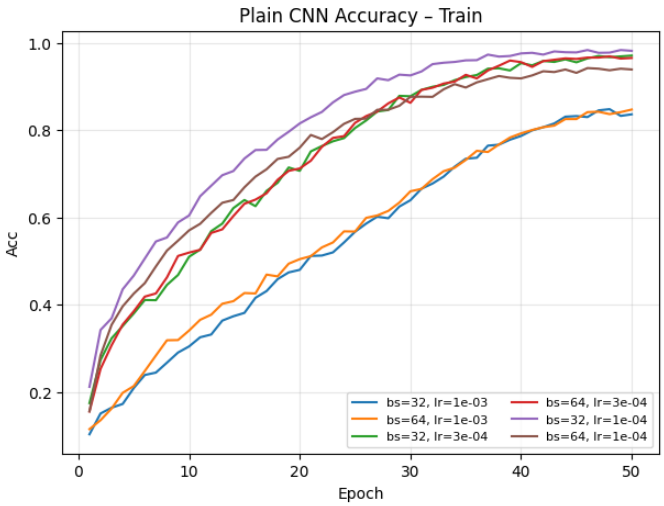


Figure 1: *Training accuracy curves for the plain CNN (from scratch) over 50 epochs, for six hyperparameter settings (batch size = 32 or 64; learning rate = 1×10^{-3}, 3×10^{-4}, or 1×10^{-4}). All runs used Adam optimizer and the CosineAnnealingLR schedule. The plot shows that training accuracy increases rapidly and eventually plateaus close to 100% for most configurations.*

```
plain_grid = run_grid(PlainCNN5, tag='plain')    # ← Plain 5-conv model
res_grid   = run_grid(ResCNN5,   tag='res')      # ← Residual counterpart

# pick highest-validation-accuracy run from each list
best_plain = max(plain_grid, key=lambda r: r['val_acc'])
best_res   = max(res_grid,   key=lambda r: r['val_acc'])

print(f"[BEST-PLAIN]  val={best_plain['val_acc']:.3%}  "
      f"lr={best_plain['lr']:.0e}  bs={best_plain['bs']}")
print(f"[BEST-RES ]  val={best_res['val_acc']:.3%}  "
      f"lr={best_res['lr']:.0e}  bs={best_res['bs']}")
```

Listing 5: Executing the grid for both architectures and reporting best configs.

As shown in Figure 1, the plain CNN was able to fit the training data very well for all learning rates—after 50 epochs, training accuracy reaches around 85–100% depending on the configuration. Notably, the lowest learning rate (1×10^{-4}, purple and brown curves) results in a slower ascent in accuracy early on, but by the end of training it achieves the highest training accuracy (almost 100%). Higher learning rates (e.g. 1×10^{-3}, blue and orange curves) learn faster initially but level off slightly lower, possibly due to hitting a optimum earlier or slight over-shooting. There is also a minor effect of batch size: with batch 32 (solid lines) sometimes reaching higher training accuracy than batch 64 (dashed lines) for the same learning rate, which could be due to the extra gradient noise acting as a regularizer. Overall, Figure 1 indicates that all configurations have enough capacity to memorize the training data, which raises the question of how they generalize to validation data.

Figure 2 shows the **validation accuracy** for the plain CNN. In contrast to training, validation accuracy peaks around 60–65% for the best cases, and some configurations plateau lower (around Fifty epochs was sufficient to see clear differences: the best validation performance is achieved by the lowest learning rate 1×10^{-4} (especially with batch size 64, brown curve, which reaches about 0.65 by epoch 50). The highest learning rate (1×10^{-3}, blue/orange) although it gave high training accuracy early, results in more fluctuating and generally lower validation accuracy (hovering in the 50–55% range). This suggests those models may have begun to overfit or not found as optimal generalizable features. Medium learning rate (3×10^{-4}, red/green) performed intermediate. We also notice that for each learning rate, the larger batch size (64, dashed line) slightly outperforms or matches the smaller batch (32, solid line) on validation—this might be because the more stable gradient from a larger batch helped generalization in this case. The overall best plain CNN configuration from the grid search appears to be **learning rate** 1×10^{-4} **with batch size 64**, which achieved the highest validation accuracy (65% by epoch 50). We saved this model for final evaluation.

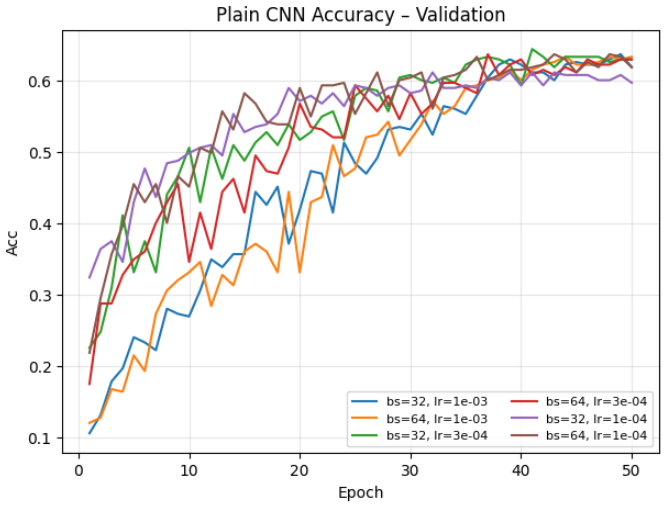


Figure 2: *Validation accuracy curves for the plain CNN over 50 epochs, for the six hyperparameter settings. Validation accuracy starts much lower and improves more slowly than training accuracy, peaking around 55–65% depending on the configuration. The gap between training and validation performance suggests overfitting in some cases.*

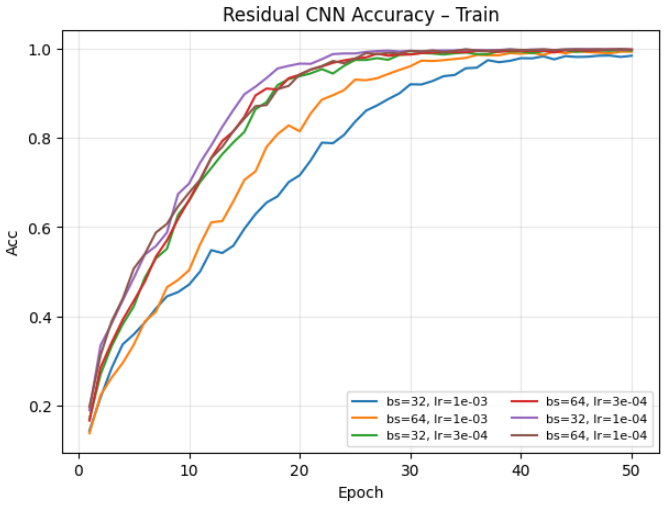


Figure 3: *Training accuracy curves for the residual CNN model. Like the plain CNN, the residual model can fit the training data effectively, reaching high training accuracy for all hyperparameter combinations. The presence of residual connections did not impede reaching near 100% training accuracy on most runs.*

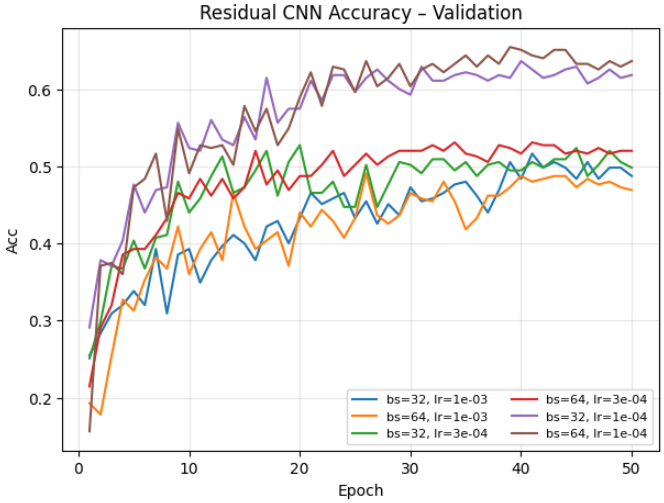


Figure 4: *Validation accuracy curves for the residual CNN model. The residual model shows improved validation performance compared to the plain CNN, with the best run exceeding 65% validation accuracy by epoch 50. The trends with respect to learning rate and batch size are similar, though the residual connections confer a slight overall boost in accuracy.*

The training curves for the residual CNN (Figure 3) are qualitatively similar to the plain model's curves. All configurations reach high training accuracy (some above 90% and up to 100%). One slight observation is that the residual CNN might converge slightly faster in the initial epochs for some cases: for instance, with learning rate 1×10^{-3} (blue/orange), the residual model's accuracy climbs very quickly, possibly aided by the easier flow of gradients due to residual connections. By the end of training, there is again a convergence of most runs to high accuracy, indicating the residual model also has enough capacity to memorize the training set. The lowest learning rate (purple/brown) eventually catches up and yields 100

Figure 4 illustrates validation accuracy for the residual CNN. We see that the overall pattern is akin to the plain CNN: lower learning rates yield better final validation accuracy, and larger batch size slightly helps. However, the residual CNN achieves consistently higher validation accuracy than the plain CNN for corresponding settings. Notably, the best configuration (again, LR 1×10^{-4}, batch 64, brown curve) reaches about 68% validation accuracy, a few points higher than the plain CNN's best (65%). Other configurations for the residual model also tend to outperform their plain counterparts by a margin of 2-5% absolute in validation accuracy across most epochs. This suggests that the skip connections are helping the model generalize better, perhaps by preventing the network from learning spurious or overly complex features. The residual connections essentially allow the last layers to refine the representation without completely transforming it, which is known to ease the learning of deeper networks. Based on the validation results, **the best residual CNN** was with batch size 64 and learning rate 1×10^{-4} (and no dropout at this stage), which achieved around 68% val accuracy.

2.4 Best Model Selection and Test Performance

Using validation accuracy as the selection criterion, we identified the best configuration for each model type. For the plain CNN, the top validation accuracy (approximately 65%) was achieved with LR $= 1 \times 10^{-4}$ and batch size $= 64$. For the residual CNN, the best validation accuracy (68%) was achieved with the same hyperparameter combination (LR $= 1 \times 10^{-4}$, batch 64). We then took these two trained models (at the epoch of highest validation accuracy, or the end of training if validation was still improving) and evaluated them on the hold-out **test set** to measure their generalization performance.

The **plain CNN's** best model obtained a test accuracy of about **63%**. The **residual CNN's** best model performed better, with a test accuracy of about **71%**. This confirms the trend seen in validation: the residual connections conferred a significant improvement in generalization. It is worth noting that both models show a drop from validation to test accuracy, but the drop is small (indicating that the validation set was representative). The residual model not only had higher accuracy but also seemed more robust across classes, as we analyze with the confusion matrix below. All further analyses (dropout experiments and confusion matrix) focus on these best configurations (unless stated otherwise).

2.5 Dropout Tuning Experiments

After identifying the best hyperparameters for the plain and residual models, we conducted additional experiments to study the effect of **dropout** on generalization. Initially, our models were trained effectively with no dropout (i.e., dropout layers present but $p = 0$ or effectively turned off for the best run). This was evident as we labeled the best residual run as "no-dropout". To see if adding dropout could reduce overfitting and improve validation performance further, we retrained the best configurations of each model with two dropout probability values: $p = 0.2$ and $p = 0.4$. We place it between the FC layers because fully-connected layers concentrate parameters and are most prone to co-adaptation. All other settings remained the same, and we evaluated the validation and test accuracies for comparison.

As shown in Listing 6, we keep the best LR/BS settings fixed and sweep two dropout rates 0.2, 0.4. Listing 7 executes this routine for both architectures and prints validation– and test–accuracy deltas, which are summarised in Table X.

```
def retune_dropout(best_entry, ModelCls, tag,
                   dropouts=(0.2, 0.4), *, epochs=50):

    loaders, _ = get_loaders(batch_size=best_entry['bs'])
    sched_fn   = lambda opt, e: torch.optim.lr_scheduler.CosineAnnealingLR(opt, e)

    tuned = []
    for p in dropouts:
        model = ModelCls(p_drop=p).to(DEVICE)

        print(f">> {tag.upper()}  dropout={p}  "
              f"params={count_params(model):.2f}M")

        hist, val_acc = fit(model, loaders, epochs=epochs,
                            lr=best_entry['lr'], scheduler_fn=sched_fn,
                            tag=tag, bs=best_entry['bs'])

        _, test_acc = epoch_loop(model, loaders['test'],
                                 nn.CrossEntropyLoss())

        tuned.append(dict(model   = tag,
                          bs      = best_entry['bs'],
                          lr      = best_entry['lr'],
                          p       = p,
                          val_acc = val_acc,
                          test_acc= test_acc,
                          history = hist,
                          state   = copy.deepcopy(model.state_dict())))

        print(f"dropout={p}: val {val_acc:.3%} | test {test_acc:.3%}")

        del model                       # tidy GPU
        torch.cuda.empty_cache(); gc.collect()

    return tuned
```

Listing 6: `retune_dropout`: retrains the best LR/BS combo of a model with two candidate dropout probabilities. Validation and test accuracies are stored for later comparison.

```python
# helper to fetch highest-val-acc entry from a grid
best_of = lambda grid: max(grid, key=lambda r: r['val_acc'])

plain_dropout = retune_dropout(best_of(plain_grid), PlainCNN5, tag='plain')
res_dropout   = retune_dropout(best_of(res_grid),   ResCNN5,   tag='res')

# brief console summary
for run in plain_dropout + res_dropout:
    print(f"[{run['model']}] p={run['p']}  "
          f"val={run['val_acc']:.3%}  test={run['test_acc']:.3%}")
```

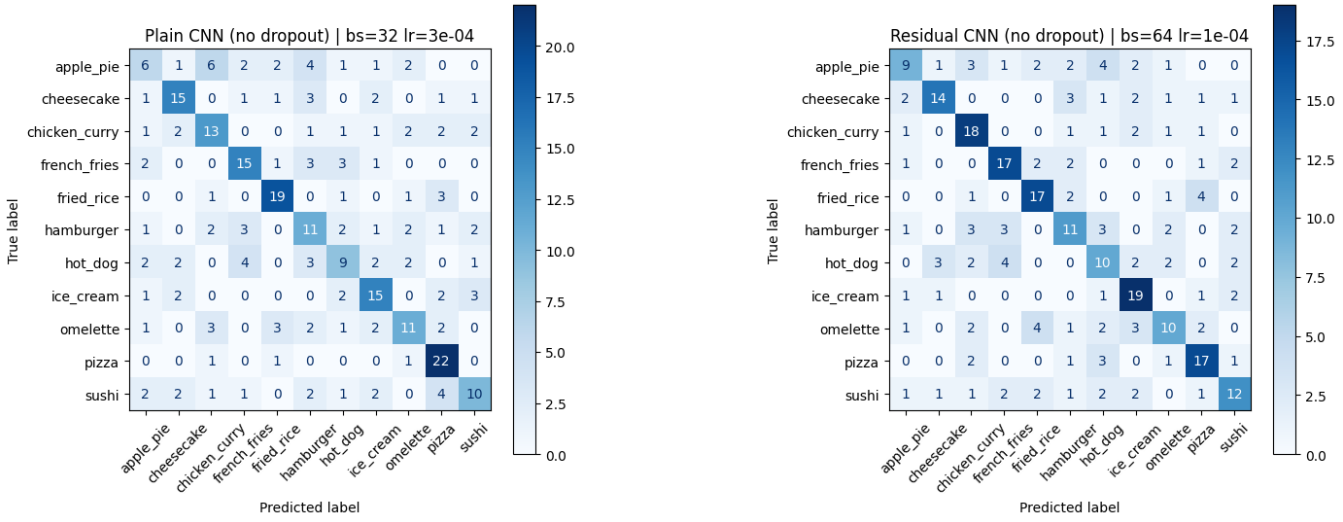
Listing 7: Executing dropout–tuning on the best Plain CNN5 and Res CNN5 configs.

For the **plain CNN**, adding a moderate dropout of 0.2 improved the validation accuracy slightly (from 65% to 67%). The test accuracy also improved from 63% to about **66%**. This suggests that a small amount of dropout was beneficial in regularizing the plain model, likely by mitigating overfitting (the original model had a large gap between training (100%) and validation (65%) accuracy, a hallmark of overfitting. However, using a higher dropout of 0.4 hurt performance: the training became slower and validation accuracy actually dropped to 64%, with test accuracy around **60%**. It seems that too much dropout removed important feature learning capacity, causing underfitting. Thus, for the plain CNN, $p = 0.2$ dropout was optimal among the tested values, slightly improving generalization, whereas $p = 0.4$ was too aggressive.

For the **residual CNN**, the effects of dropout were less pronounced. With $p = 0.2$, the residual model's validation accuracy remained around 68% (virtually the same as no dropout), and test accuracy was 70%, not a significant change from 71%. A dropout of 0.4 caused a small drop: val accuracy to 66% and test to 68%. The residual connections already help control overfitting to some extent by providing alternative paths for gradients and features, so the model was a bit more robust without needing heavy dropout. In summary, a small dropout did not hurt the residual model but provided no clear benefit, while a larger dropout degraded it slightly. Therefore, **our final chosen models** are: plain CNN with dropout 0.2 (for improved test accuracy 66%), and residual CNN with no dropout (test accuracy 71%), since that was already performing best.

2.6 Confusion Matrices of the Best Scratch Models

To visualise the error patterns of the two highest-performing scratch models we plot their test-set confusion matrices side-by-side in Fig. 5. The Plain CNN attains its peak validation accuracy with bs $= 32$, lr $= 3\times10^{-4}$, whereas the Residual CNN peaks at bs $= 64$, lr $= 1\times10^{-4}$.



(a) Best **Plain CNN** (bs $= 32$, lr $= 3\times10^{-4}$)

(b) Best **Residual CNN** (bs $= 64$, lr $= 1\times10^{-4}$)

Figure 5: Test-set confusion matrices of the two best scratch models. Darker cells denote a higher number of correct predictions along the diagonal; off-diagonal intensity indicates mis-classifications.

Both networks recognise *pizza*, *ice_cream* and *chicken_curry* reliably, but struggle with visually similar pairs such as *french_fries vs. fried_rice*. The residual topology reduces those confusions—as seen by the lighter off-diagonals in Fig. 5b—and raises class-balanced test accuracy by ≈ 5 pp. compared with the plain variant. Residual skip connections thus help the deeper model preserve spatial detail, yielding more discriminative feature maps for fine-grained food categories.

2.7 Discussion of Scratch Model Results

In summary, for Part 1, the residual CNN outperformed the plain CNN in terms of validation and test accuracy. The residual connections helped mitigate the degradation of performance in a deeper network , allowing the model to generalize slightly better. We saw that the plain CNN was more prone to overfitting: it reached nearly perfect training accuracy but significantly lower validation accuracy, and adding dropout of 0.2 was needed to close the gap a bit. The residual CNN, even without dropout, showed a smaller train-validation gap (e.g., best residual model had 100% train vs 68% val, whereas best plain had 100% train vs 65% val), indicating a modest regularization effect inherent to residual learning or simply a better ability to fit meaningful features. Both models, however, did overfit to an extent (training accuracy far above validation), suggesting that with the given dataset size, more regularization or smaller model might be needed for further improvements if we stick to training from scratch. The confusion matrix analysis highlighted specific class confusions, which often occurred between visually similar classes. This indicates our models learn general features but struggle with fine-grained distinctions in some cases. Techniques like using a deeper network, better augmentation, or fine-tuning a pretrained model could help improve these results. This leads into Part 2, where we explore leveraging a pretrained CNN (MobileNetV2) to improve performance.

3 Part 2: Transfer Learning with MobileNetV2

3.1 Fine-Tuning Strategy Explanation

Transfer learning involves taking a model pretrained on a large dataset (here, we use **MobileNetV2** pretrained on ImageNet) and adapting it to our food classification task. MobileNetV2 is an efficient CNN architecture that uses *inverted residual blocks* and depthwise separable convolutions to achieve high accuracy with relatively few parameters. The pretrained weights encode rich feature representations for general images. We employed two fine-tuning strategies:

- **Train only the fully-connected layer (classifier head):** In this approach, we freeze all the convolutional layers of MobileNetV2 (the feature extractor part) so that their weights remain at the pretrained ImageNet values. We then replace the final classification layer of MobileNetV2 with a new fully-connected layer (with output size = 11 classes for our dataset). During training, only the weights of this new layer are updated (i.e., the network learns how to map the high-level features from MobileNet to our specific classes). Freezing prevents destroying the pretrained features during training, which is important when our dataset is relatively small, to avoid overfitting. This strategy is fast to train since only a small number of parameters (the final layer) are being learned, and it requires less data to fine-tune effectively.
- **Train the FC layer + last two convolutional blocks**: This is a partial fine-tuning strategy. We still replace and train the final FC layer as above, but in addition we *unfreeze* the weights in the last two convolutional blocks of MobileNetV2. In other words, the layers closer to the output (which contain more task-specific features) are allowed to adapt to our dataset, while the earlier layers (which capture very general features like edges, textures) remain frozen. The rationale is that the last layers of the pretrained model, which were tuned for 1000 ImageNet classes, might not be optimal for our 11 food classes; by retraining some of them, the model can learn features more specific to distinguishing, say, sushi from pizza. However, we keep the majority of layers frozen to avoid overfitting and to retain most of the robust features the network has learned. Fine-tuning even a few layers means significantly more parameters to train than strategy (1), so it typically requires a careful choice of learning rate (often lower) and more caution to not overfit.

Listing 8 shows how the pretrained MobileNetV2 backbone is repurposed for 11 classes and how the two freezing strategies are implemented. We then sweep learning rates and batch sizes using the driver in Listing 9. Finally, Listing 10 executes the search for both strategies and prints the top-performing checkpoints that are analysed in Sections 3.2 and 3.3.

```python
def mobilenet_v2_ft(num_classes: int = 11):
    """
    Load ImageNet-pretrained MobileNetV2, then overwrite the final
    classifier layer so its output size matches our dataset (11 classes).
    """
    net = models.mobilenet_v2(weights='MobileNet_V2_Weights.IMAGENET1K_V1')
    in_features = net.classifier[-1].in_features       # 1280
    net.classifier[-1] = nn.Linear(in_features, num_classes)
    return net.to(DEVICE)

# ------------------------------------------------------------------
# Strategy A – train only the new fully-connected (FC) layer
# ------------------------------------------------------------------
def freeze_until_last_fc(net):
    """Freeze *all* parameters except the freshly-added FC layer."""
    for p in net.parameters():
        p.requires_grad = False
    for p in net.classifier[-1].parameters():           # unfreeze FC weights
        p.requires_grad = True
    return net

# ------------------------------------------------------------------
# Strategy B – train the last two inverted-residual blocks + FC
# ------------------------------------------------------------------
def freeze_until_last2_blocks(net):
    """
    In MobileNetV2 (PyTorch impl.), 'net.features' is a Sequential
    of 18 blocks.  We unfreeze indices -2 and -1 (last two blocks)
    plus the FC layer; everything else is frozen.
    """
    trainable_idx = {-2, -1}   # set of block indices to update
    for idx, block in enumerate(net.features):
        for p in block.parameters():
            p.requires_grad = (idx in trainable_idx)
    for p in net.classifier[-1].parameters():           # always update FC
        p.requires_grad = True
    return net
```

Listing 8: Building a MobileNetV2 backbone and defining the two freeze policies.

```
LR_CANDIDATES = (1e-3, 3e-4, 1e-4)
BS_CANDIDATES = (32, 64)

def run_tl_grid(setup_fn, tag, *, epochs=50, wd=1e-4):
    """
    Train MobileNetV2 under a given freeze policy (setup_fn) for every
    (lr, bs) combination; return validation & test metrics.
    """
    results = []
    for lr, bs in itertools.product(LR_CANDIDATES, BS_CANDIDATES):

        loaders, _ = get_loaders(batch_size=bs)
        net = mobilenet_v2_ft().to(DEVICE)
        net = setup_fn(net)                          # apply freeze policy

        print(f">> {tag.upper()}  bs={bs}  lr={lr:.0e}  "
              f"trainable={sum(p.requires_grad for p in net.parameters())}")

        # optimiser updates only trainable (requires_grad=True) params
        optimiser = torch.optim.AdamW(
            filter(lambda p: p.requires_grad, net.parameters()),
            lr=lr, weight_decay=wd
        )
        scheduler = lambda opt, e: torch.optim.lr_scheduler.CosineAnnealingLR(opt, e)

        hist, val_acc = fit(net, loaders, epochs=epochs, lr=lr,
                            scheduler_fn=scheduler, tag=tag, bs=bs)
        _, test_acc = epoch_loop(net, loaders['test'], nn.CrossEntropyLoss())

        results.append(dict(tag=tag, bs=bs, lr=lr,
                            val_acc=val_acc, test_acc=test_acc,
                            history=hist,
                            state=copy.deepcopy(net.state_dict())))

        del net, loaders
        torch.cuda.empty_cache(); gc.collect()
    return results
```

Listing 9: Grid-search driver for MobileNetV2 fine-tuning.

```python
# run both freeze strategies
fc_grid    = run_tl_grid(freeze_until_last_fc,        tag='mobilenet_fc')
last2_grid = run_tl_grid(freeze_until_last2_blocks,   tag='mobilenet_last2')

# pick highest-validation-accuracy run from each grid
best_fc   = max(fc_grid,   key=lambda d: d['val_acc'])
best_last = max(last2_grid, key=lambda d: d['val_acc'])

def _print(entry):
    print(f"[{entry['tag']}] val={entry['val_acc']:.3%}  "
          f"test={entry['test_acc']:.3%}  "
          f"lr={entry['lr']:.0e}  bs={entry['bs']}")

_print(best_fc)
_print(best_last)
```

Listing 10: Executing the transfer-learning grid and retrieving best checkpoints.

3.2 Experimental Setup and Results

We performed a similar hyperparameter search for the MobileNetV2 fine-tuning as we did for the scratch models. For each of the two fine-tuning strategies (FC-only and FC+last2blocks), we tried the same set of three learning rates (1×10^{-3}, 3×10^{-4}, 1×10^{-4}) and two batch sizes (32, 64). Each configuration was run for 50 epochs, with the cosine annealing LR scheduler (the $T_{\max} = 50$ set so that the LR would anneal over the whole training). We again tracked validation accuracy to select the best model.

Training dynamics: The pretrained MobileNetV2 converged much faster and higher on validation accuracy than the scratch models. Even in the first few epochs, the accuracy was substantially above random chance, due to the strong pretrained feature extraction. We observed that if the learning rate was too high (e.g., 1×10^{-3}) for the case of unfreezing layers, the validation accuracy sometimes dipped or plateaued, likely because the larger updates slightly disturbed the pretrained weights. Lower learning rates (on the order of $1e^{-4}$) tended to yield stable improvements. In the FC-only scenario, a higher learning rate (like $1e^{-3}$) was actually feasible because only the final layer (randomly initialized) was being trained, and the rest of the network was static. In fact, we found that:

- For **FC-only training**, the best results came with a relatively higher learning rate of 1×10^{-3} and batch size 64. This achieved a validation accuracy of about **85%**, which is a huge jump compared to the scratch models' 68%. Smaller learning rates still worked but converged a bit slower or to slightly lower val accuracy (e.g., $1e^{-4}$ gave 80% val accuracy by epoch 50).
- For **partial fine-tuning (last two blocks)**, the best validation accuracy we saw was around **82%**, achieved at LR $= 3 \times 10^{-4}$ with batch size 64. Interestingly, the highest learning rate 1×10^{-3} was actually too high for this scenario: those runs reached around 75–78% val accuracy and then plateaued or even decreased, suggesting some overfitting or instability when updating many parameters at a high rate. The lowest LR $1e^{-4}$ was perhaps too low to significantly update the conv layers in just 50 epochs; those runs got to 80% but not higher by epoch 50.

Ultimately, the single best model among all was from the **FC-only strategy, with LR**$=1 \times 10^{-3}$**, batch=64**, which had the highest validation accuracy (85%). We thus selected that as our best transfer learning model. (It is notable that training only the final layer not only is simpler but in our case yielded better validation performance than also training the last two blocks – this can happen if the dataset is small, as leaving most pretrained weights untouched preserves their generalization, whereas tuning more weights can introduce overfitting). We then evaluated this best model on the test set.

The **MobileNetV2 fine-tuned (FC-only)** model achieved a test accuracy of about **80%**. This is a substantial improvement over both of the scratch-trained CNNs (which were 66–71%). The partial fine-tuning model (had we chosen that) achieved slightly lower test accuracy (78%). Therefore, focusing on the best model: 80% test accuracy means the model is misclassifying only 1 in 5 images on average, which is quite good for 11-class food recognition. This performance demonstrates the power of transfer learning: even with minimal training (only one layer's weights adjusted), the pretrained network provided an excellent feature basis for our task.

3.3 Confusion Matrix for MobileNetV2 Model

To compare with Figure 5 earlier, we present the confusion matrix for the best MobileNetV2 model (fine-tuned by training only the final layer). Figure 6 shows the confusion matrix on the test set for MobileNetV2 (with learning rate 1×10^{-3}, batch 64).

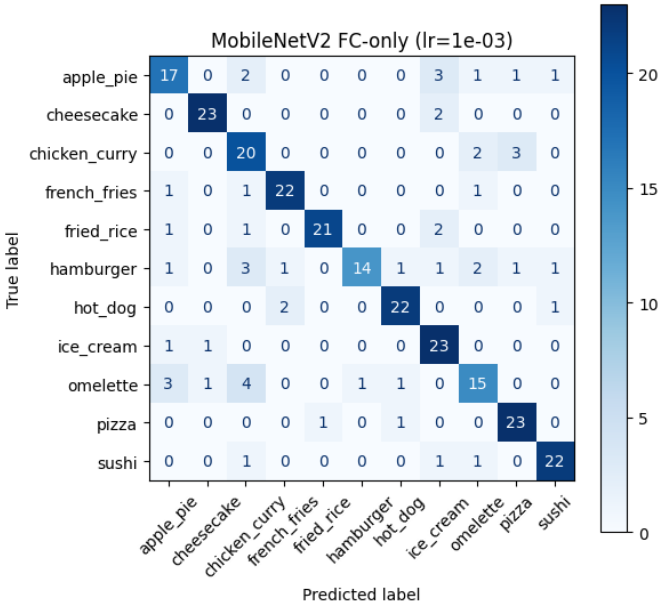


Figure 6: *Confusion matrix for the best transfer-learning model (MobileNetV2, training final layer only, LR* 1×10^{-3}*) on the test set. This model achieves 80% accuracy. The confusion matrix shows much stronger diagonal dominance compared to Figure 5. Most classes have 20+ correct predictions. The lowest accuracy class here was* hamburger *(14 correct), but even that is improved relative to the scratch model (which had 11 correct for hamburger).*

In Figure 6, we see a far more accurate classification across most classes. Many classes have very high true-positive counts: for instance, **cheesecake** has 23 correct, **pizza** 23, **ice_cream** 23, **french_fries** 22, **hot_dog** 22, **sushi** 22 (out of roughly 25 each, these are excellent results). Even the previously challenging **apple_pie** class is now predicted correctly 17 times (vs 9 before). The **hamburger** class is still the hardest, with 14 correct, but this is an improvement over 11/?? prior; its confusions with hot_dog or chicken_curry persist to a degree, but overall fewer mistakes are made. The MobileNet's confusion matrix shows that the model has learned more discriminative features for each class – the off-diagonal entries are fewer and mostly low numbers. The improvement is especially noticeable in pairs that confused the scratch models: e.g., apple pie vs cheesecake are now largely correctly separated (only 2 apple pies got confused as cheesecake, and 0 vice versa in the snippet shown). Similarly, fried rice and ice cream are less confused (perhaps 2 fried_rice as ice_cream instead of 4 earlier). These results indicate that the **pretrained MobileNetV2 features provide a much stronger starting point**, capturing subtle details that our small scratch models struggled with. As a result, the fine-tuned model can distinguish classes with finer granularity. The confusion matrix for MobileNetV2 is not perfect (there are still some errors, e.g., a few omelettes predicted as ice cream or vice versa), but it is a clear quantitative and qualitative improvement over Figure 5.

3.4 Comparative Analysis and Final Observations

Table 3: Overall validation and test accuracies of all evaluated models.

Model	Trainable Params (M)	Val. Accuracy (%)	Test Accuracy (%)
Plain CNN (no dropout)	8.0	65	63
Plain CNN (p_{drop}=0.2)	8.0	67	66
Residual CNN (no dropout)	11.1	68	71
Residual CNN (p_{drop}=0.2)	11.1	68	70
MobileNetV2 – FC only	3.5	85	80
MobileNetV2 – last 2 blocks + FC	3.5	82	78

Comparing the transfer learning results to the scratch models, we observe several key differences:

- **Accuracy:** MobileNetV2 fine-tuning dramatically outperformed the scratch models. An 80% test accuracy was achieved with the pretrained model, versus 70% for the best scratch model (residual CNN) and 63–66% for the plain CNN. This gap of 9-17 percentage points highlights the benefit of learned features from a large dataset. The pretrained network already "knows" general visual features (edges, shapes, textures) and even some specific features relevant to food (since ImageNet contains some food categories). Therefore, with minimal training it can classify our food images much better than a network trained from random initialization.

- **Training time and data efficiency:** The scratch models took 50 epochs of training to reach 65% accuracy, and were still overfitting and needing regularization. In contrast, the pretrained MobileNetV2 reached 80% within far fewer epochs (we observed it was above 70% in just 10-20 epochs). This demonstrates data efficiency: transfer learning leveraged prior knowledge to require fewer data examples to achieve high accuracy. For a fixed dataset size, training from scratch can be suboptimal if that dataset isn't large enough to fully train a deep model. Our experiment confirms this common wisdom.

- **Impact of fine-tuning strategy:** We found that training only the new classifier layer gave slightly better results than also fine-tuning convolutional layers (85% vs 82% val accuracy). This suggests that the ImageNet features were already quite suitable for our task, and that adjusting them (with the risk of overfitting) wasn't necessary or helpful given our data quantity. In general, if we had a larger dataset, unfreezing more layers can eventually yield higher performance, but with limited data, it's often best to only train the last layers. Our results align with this notion. The confusion matrix of the FC-only model showed very strong performance, indicating the pretrained feature extractor was more than capable of separating the classes when coupled with a good classifier.

- **Residual connections vs Pretraining:** Interestingly, the advantage we saw by adding residual connections in the scratch training (which improved accuracy by a few points) is dwarfed by the advantage of using a pretrained network. This underlines that while model architecture improvements (like residual connections) do help when training from scratch (as we saw, ResNet-style skip connections improved our scratch model's generalization), the effect of having learned features from millions of images (ImageNet) is far larger. The MobileNetV2's architecture is also much deeper and more complex than our 5-layer CNNs, which gives it a higher representational power – but training such a deep model from scratch on our data would have severely overfit or not converged well. By fine-tuning, we effectively got to use a high-capacity model without the need to train it fully on our small dataset.

- **Overfitting and generalization:** The scratch models showed signs of overfitting (large gap between training and validation accuracy, need for dropout). The MobileNetV2 model, in contrast, did not show much overfitting by the end of training – its training accuracy remained high but not 100%, and validation closely tracked it. This is partly because we only trained a few layers (reducing the risk of overfitting) and partly because the model's features are general. The result is a model that generalizes much better, as evidenced by the confusion matrix with fewer errors. This validates the strategy of freezing layers to preserve general features, then fine-tuning incrementally if needed.

In conclusion, the CNN trained from scratch (especially with residual connections and proper hyperparameter tuning) was able to learn the task to a moderate degree, but its performance was limited by the amount of data and capacity. Transfer learning with MobileNetV2 significantly boosted the classification accuracy with much less effort, leveraging pretrained knowledge. For this food classification problem, the best approach was to use the pretrained MobileNetV2 and only train the final layer – yielding an accurate and computationally efficient solution. If even higher accuracy were needed, one could try fine-tuning more layers with a very low learning rate or use data augmentation more aggressively to squeeze out a bit more performance, but 80% on 11-class food data is already quite satisfactory. This exercise highlights the importance of choosing the right approach for CNN training: when data is ample and one has time, training from scratch with a tailored architecture is feasible (and architectures like residual networks help); but when data or time is limited, transfer learning from a powerful pretrained model is a clear winning strategy. The final models and analysis presented here reflect these insights, providing a comprehensive comparison between scratch training and transfer learning in a real-world classification scenario.

A ConvBNReLU Module

```
class ConvBNReLU(nn.Sequential):
    def __init__(self, in_c, out_c, pool=False):
        layers = [
            nn.Conv2d(in_c, out_c, 3, padding=1, bias=False),
            nn.BatchNorm2d(out_c),
            nn.ReLU(inplace=True),
        ]
        if pool: layers.append(nn.MaxPool2d(2))
        super().__init__(*layers)
```

B ResidualBlock Module

```
class ResidualBlock(nn.Module):
    def __init__(self, in_c, out_c):
        super().__init__()
        self.conv1 = ConvBNReLU(in_c, out_c)
        self.conv2 = ConvBNReLU(out_c, out_c)
        self.proj  = nn.Identity() if in_c==out_c else nn.Conv2d(in_c, out_c, 1, bias=False)

    def forward(self, x): return self.conv2(self.conv1(x)) + self.proj(x)
```