Hacettepe University
BBM418 Assignment 3

Name and Surname: Pınar Konuk
Identity Number: 21827613

1) Explain how you arrange your dataset into train, validation, and test and write how many images are there in the sets

1) Dataset Organization:
   - The Animals-10 dataset was used.
   - The dataset consists of a folder named "dataset/raw-image" containing subfolders for each class of animals.
   - Each class folder contains multiple image files representing samples of that particular animal.
   - Resized 128x128

2) Train-Validation-Test Split:

   The dataset was divided into three subsets: train, validation, and test.

   - Training Set: The training set contains 500 images, with at least 50 images per class.
   - Validation Set: The validation set contains 100 images in total, with 10 images per class.
   - Test Set: The test set also contains 100 images, with 10 images per class.

PART 1 - Modeling and Training a CNN classifier from Scratch

1) **Give parametric details with relevant Pytorch code snippets; number of in channels, out channels, stride, etc. Specify your architecture in detail. Write your choice of activation functions, loss functions and optimization algorithms with relevant code snippets.**

Model 1: CNN Model without Dropout

Model Name: AnimalClassifierWithouthDropout

```
class AnimalClassifierWithouthDropout(nn.Module):
    def __init__(self, input_size=(128, 128)):
        super(AnimalClassifierWithouthDropout, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(128)
        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn5 = nn.BatchNorm2d(256)
        self.conv6 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn6 = nn.BatchNorm2d(256)
        self.pool = nn.MaxPool2d(2, 2)
        self.relu = nn.ReLU()


        out_size = self.calc_output_size(input_size[0])
        self.fc1 = nn.Linear(256 * out_size * out_size, 512)
        self.fc2 = nn.Linear(512, 10)

    def calc_output_size(self, size):
        for _ in range(2):
            size = (size - 3 + 2 * 1) // 1 + 1
            size = (size - 2) // 2 + 1
        return size

    def forward(self, x):
        x = self.pool(self.relu(self.bn1(self.conv1(x))))
        x = self.relu(self.bn2(self.conv2(x)))
        x = self.relu(self.bn3(self.conv3(x)))
        x = self.pool(self.relu(self.bn4(self.conv4(x))))
        x = self.relu(self.bn5(self.conv5(x)))
        x = self.relu(self.bn6(self.conv6(x)))
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc1(x))
```

```
        x = self.fc2(x)
        return x
```

Architecture:

- Number of convolutional layers: 6
  conv1 (3 input channels to 16 output channels)
  conv2 (16 input channels to 32 output channels)
  conv3 (32 input channels to 64 output channels)
  conv4 (64 input channels to 128 output channels)
  conv5 (128 input channels to 256 output channels)
  conv6 (256 input channels to 256 output channels)
- Activation function: ReLU
- Pooling layer: MaxPool2d is used after conv1 and conv4 layers.
- Fully connected layer: Linear
- Output layer: Linear with the number of output classes (10)

Model 2: CNN Model with Dropout

Model Name: AnimalClassifierWithDropout

```
class AnimalClassifierWithDropout(nn.Module):
    def __init__(self, dropout_rate=0.5, input_size=(128, 128)):
        super(AnimalClassifierWithDropout, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(16)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(32)
        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.bn3 = nn.BatchNorm2d(64)
        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.bn4 = nn.BatchNorm2d(128)
        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.bn5 = nn.BatchNorm2d(256)
        self.conv6 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bn6 = nn.BatchNorm2d(256)
        self.pool = nn.MaxPool2d(2, 2)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)


        out_size = self.calc_output_size(input_size[0])
        self.fc1 = nn.Linear(256 * out_size * out_size, 512)
        self.fc2 = nn.Linear(512, 10)

    def calc_output_size(self, size):
        for _ in range(3):
            size = (size - 3 + 2 * 1) // 1 + 1
```

```
            size = (size - 2) // 2 + 1
        return size

    def forward(self, x):
        x = self.pool(self.relu(self.bn1(self.conv1(x))))
        x = self.dropout(x)
        x = self.relu(self.bn2(self.conv2(x)))
        x = self.pool(self.relu(self.bn3(self.conv3(x))))
        x = self.dropout(x)
        x = self.relu(self.bn4(self.conv4(x)))
        x = self.pool(self.relu(self.bn5(self.conv5(x))))
        x = self.dropout(x)
        x = self.relu(self.bn6(self.conv6(x)))
        x = self.dropout(x)
        x = x.view(x.size(0), -1)
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

Architecture:

- Number of convolutional layers: 6
  - conv1 (3 input channels to 16 output channels)
  - conv2 (16 input channels to 32 output channels)
  - conv3 (32 input channels to 64 output channels)
  - conv4 (64 input channels to 128 output channels)
  - conv5 (128 input channels to 256 output channels)
  - conv6 (256 input channels to 256 output channels)

- Activation function: ReLU is used as the activation function after each convolutional layer and fully connected layer.
- Pooling layer: MaxPool2d is used after the following layers: conv1,conv3,conv5
- Pooling reduces the dimensions of the feature maps by a factor of 2.
- Fully connected layer: Linear
- Output layer: Linear with the number of output classes (10 in this case)
- Dropout Layers:
  - After `pool` layer following `conv1`
  - After `pool` layer following `conv3`
  - After `pool` layer following `conv5`
  - After `fc1`

Choice of Activation Function: ReLU

- ReLU is a commonly used activation function in CNNs as it introduces non-linearity and helps with feature extraction.

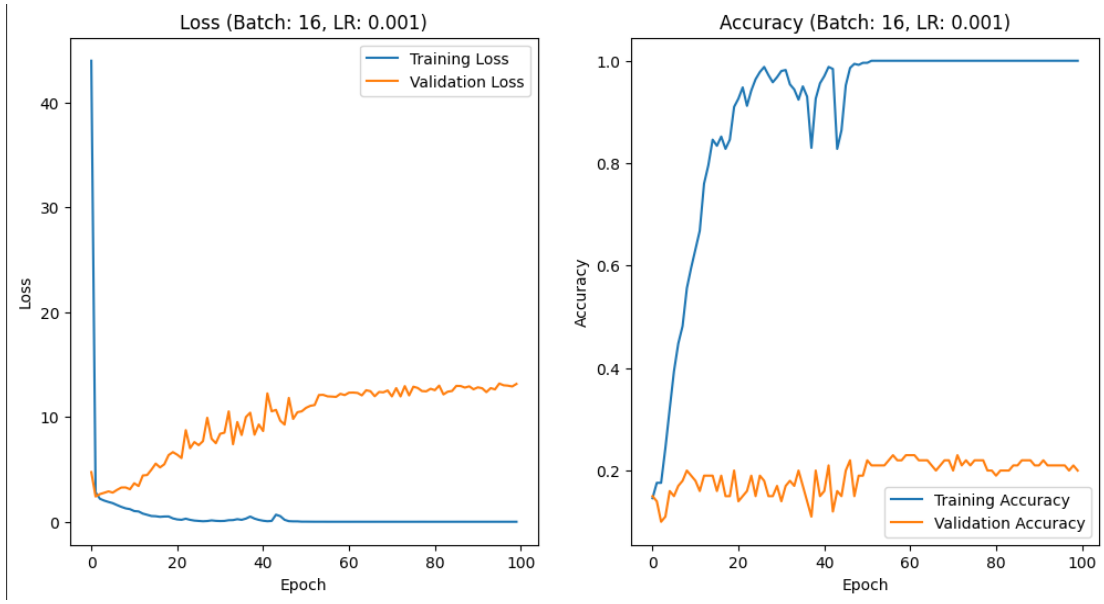
Choice of Loss Function: CrossEntropyLoss

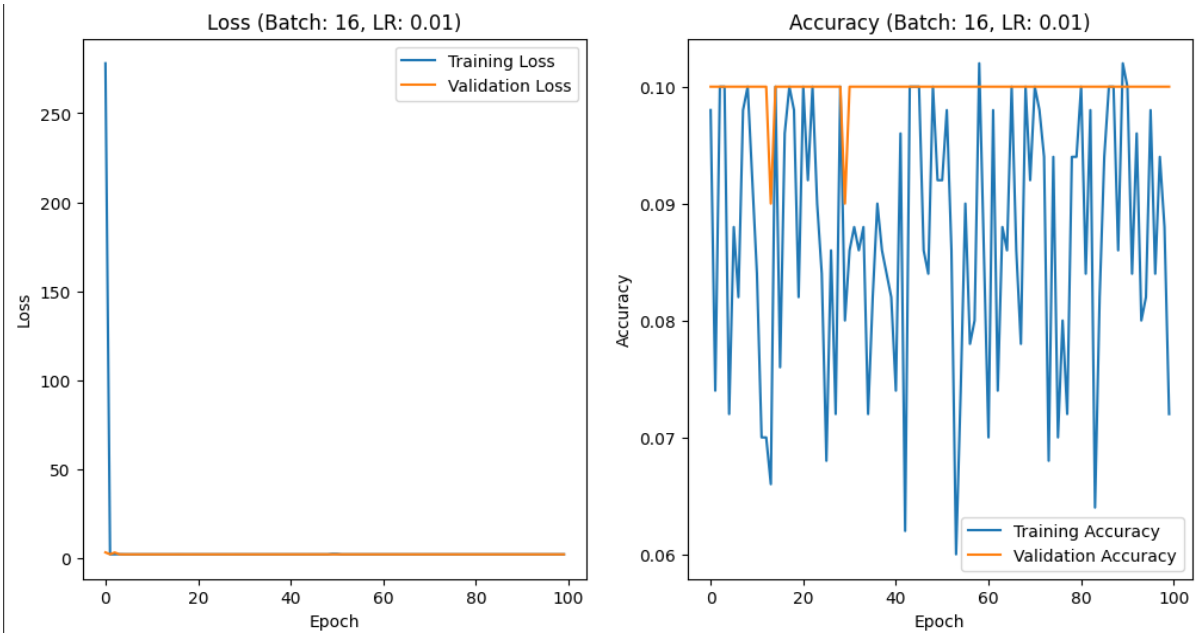
- CrossEntropyLoss is suitable for multi-class classification tasks as it calculates the loss between the predicted probabilities and the target class labels.

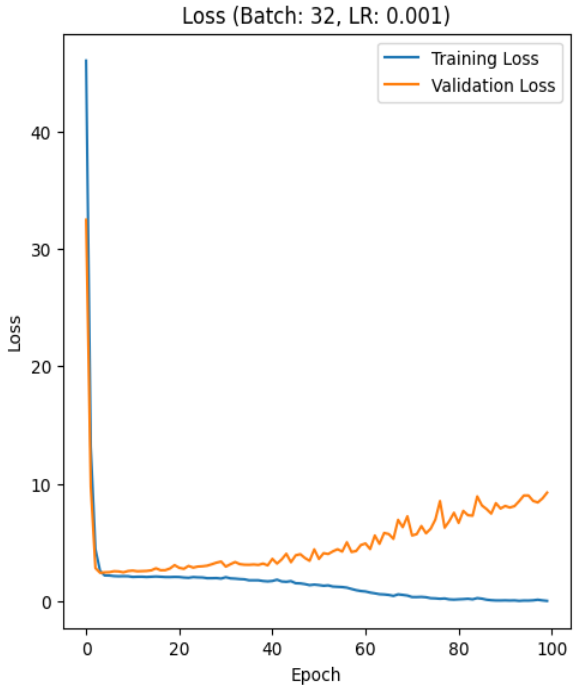
Choice of Optimization Algorithm: Adam

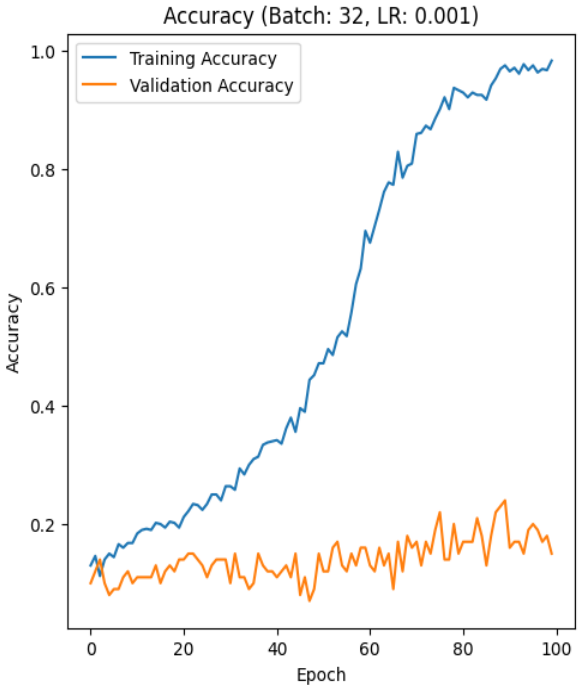
- Adam optimizer is used to optimize the model's parameters. Adam is an adaptive optimization algorithm that computes adaptive learning rates for each parameter, allowing for efficient and effective parameter updates during training

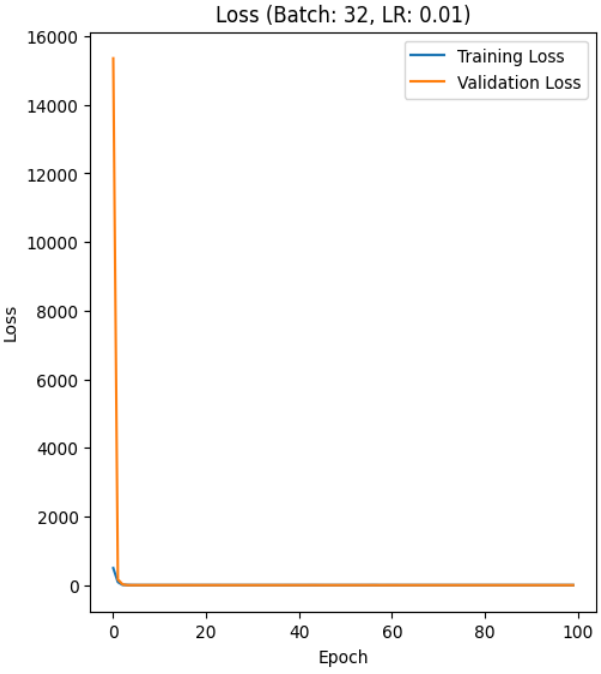
2) Draw a graph of loss and accuracy change for two different learning rates [0.001, 0.01] and three batch sizes [16, 32, 64] and 100 epoch .

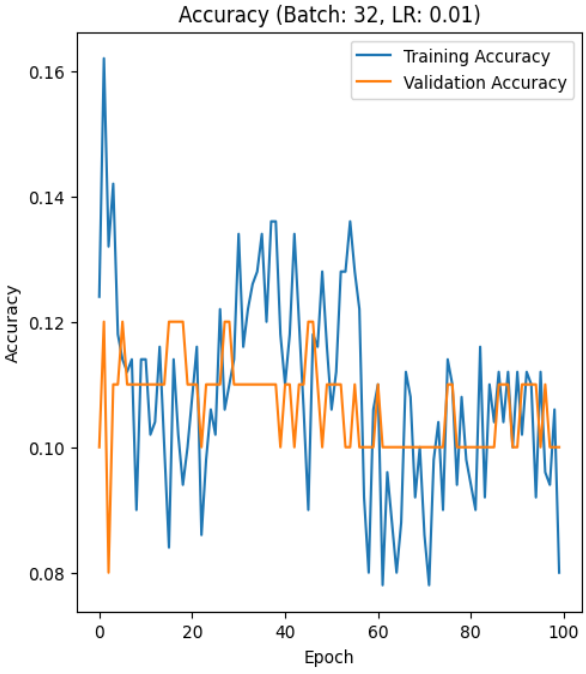


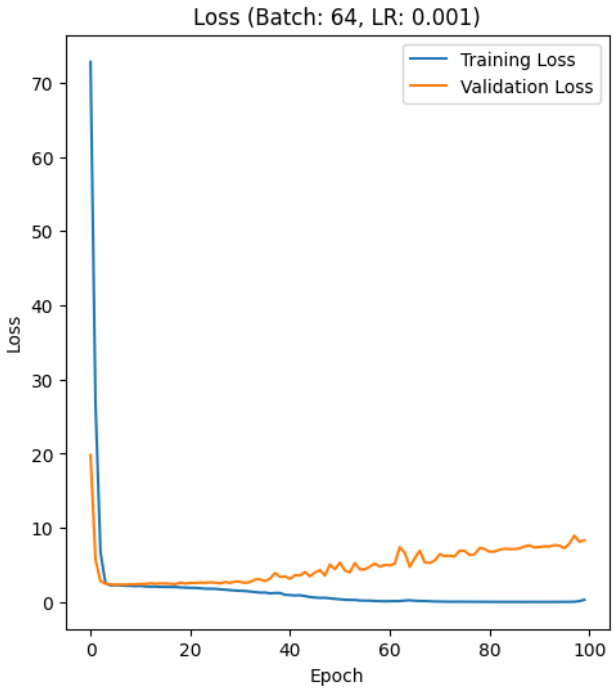


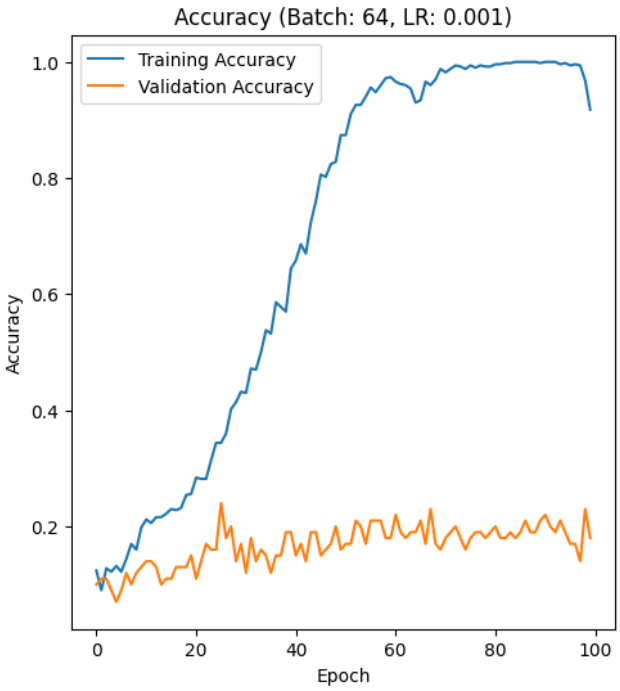

Loss (Batch: 32, LR: 0.001)
Training Loss
Validation Loss
Loss
Epoch
0
10
20
30
40
0
20
40
60
80
100

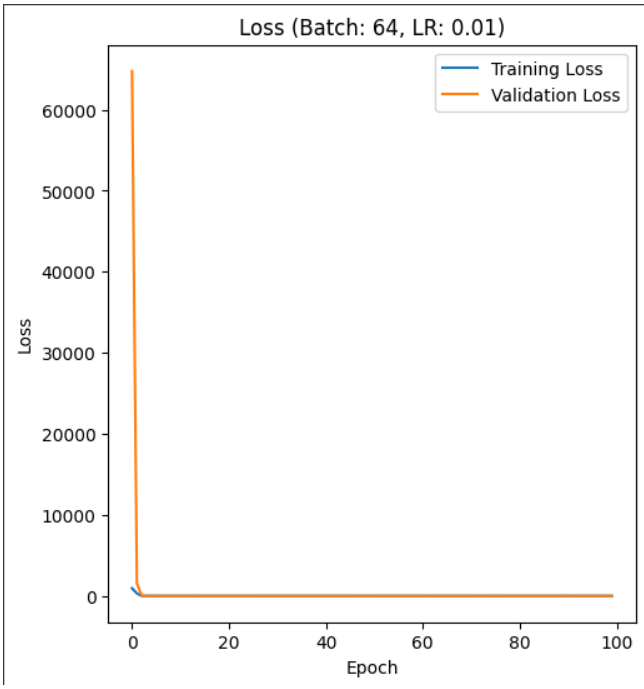

Accuracy (Batch: 32, LR: 0.001)
Training Accuracy
Validation Accuracy
Accuracy
Epoch
0.2
0.4
0.6
0.8
1.0
0
20
40
60
80
100

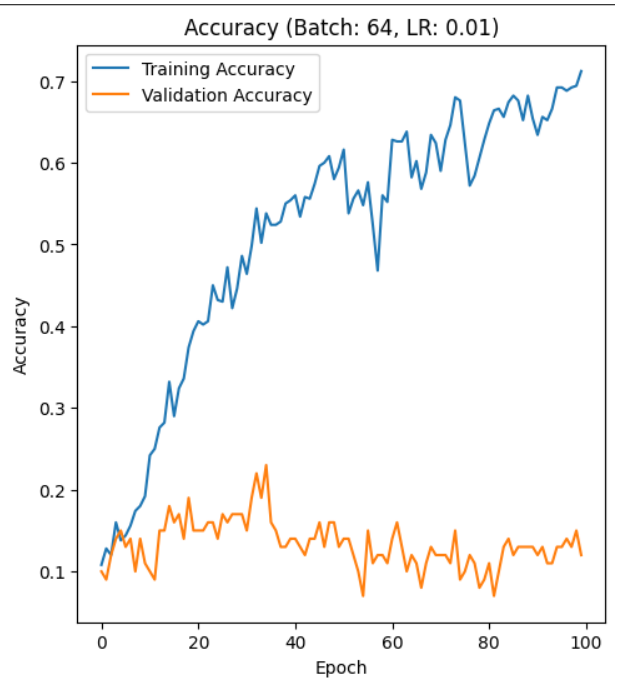

Loss (Batch: 32, LR: 0.01)
Training Loss
Validation Loss
Loss
Epoch
0
2000
4000
6000
8000
10000
12000
14000
16000
0
20
40
60
80
100


Accuracy (Batch: 32, LR: 0.01)
Training Accuracy
Validation Accuracy
Accuracy
Epoch
0.08
0.10
0.12
0.14
0.16
0
20
40
60
80
100

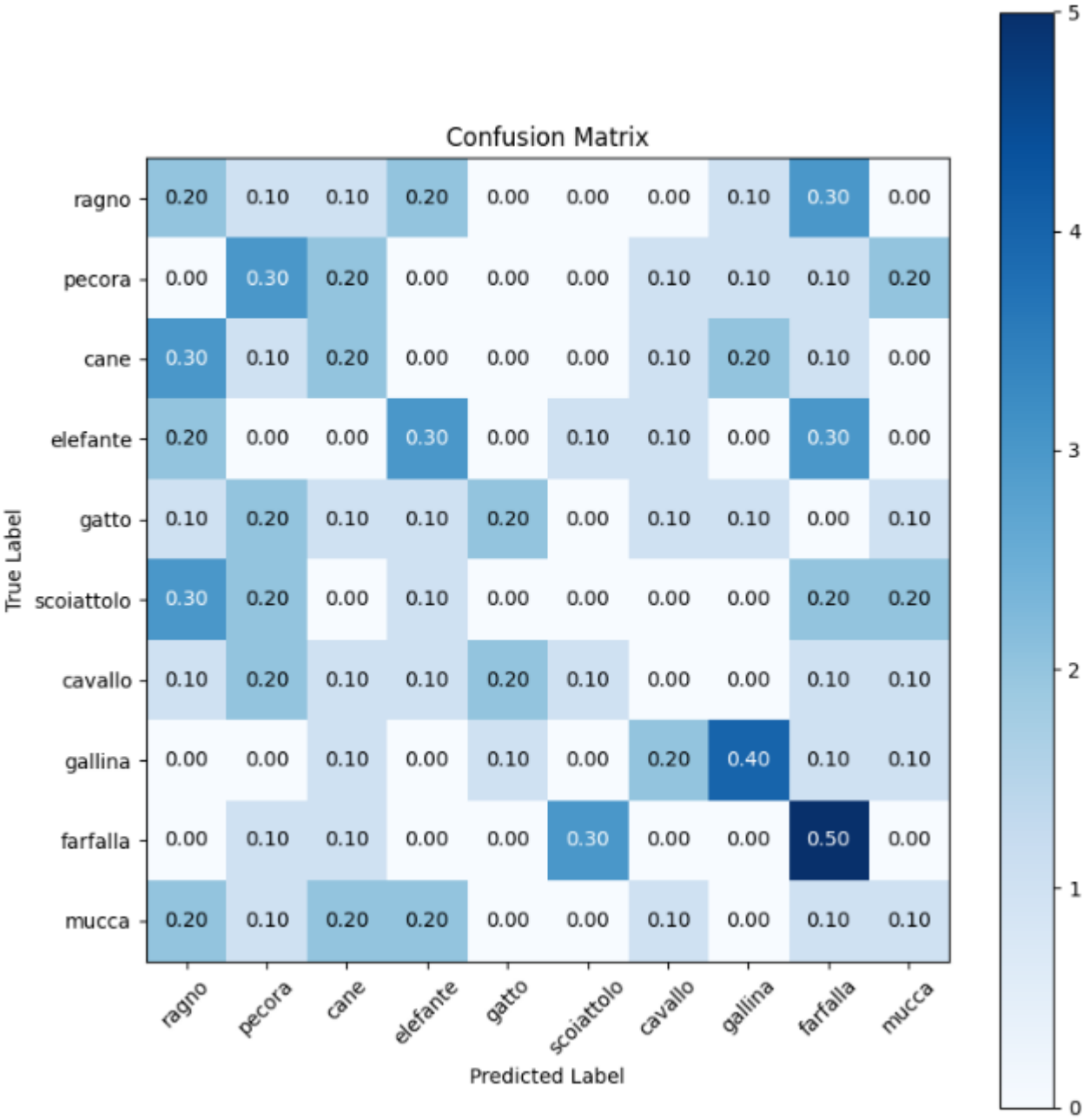




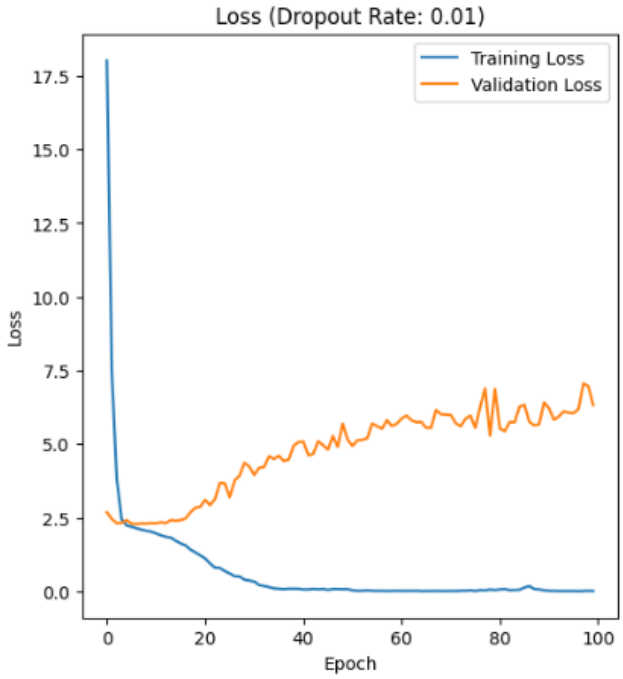


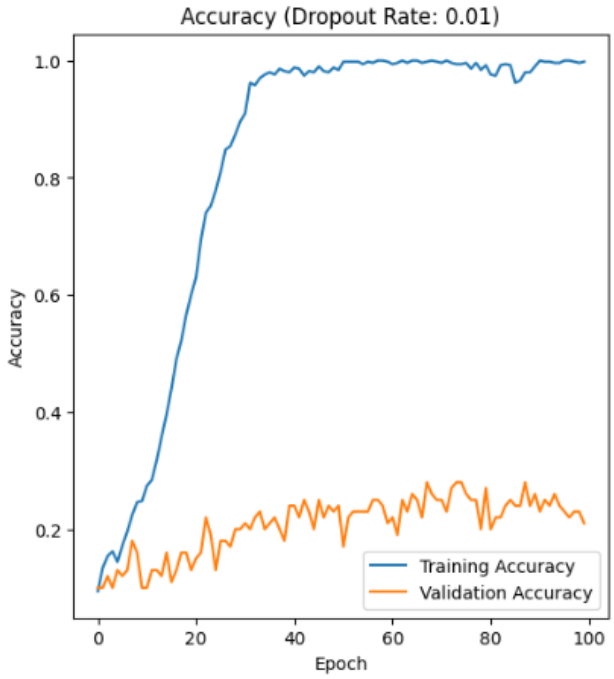


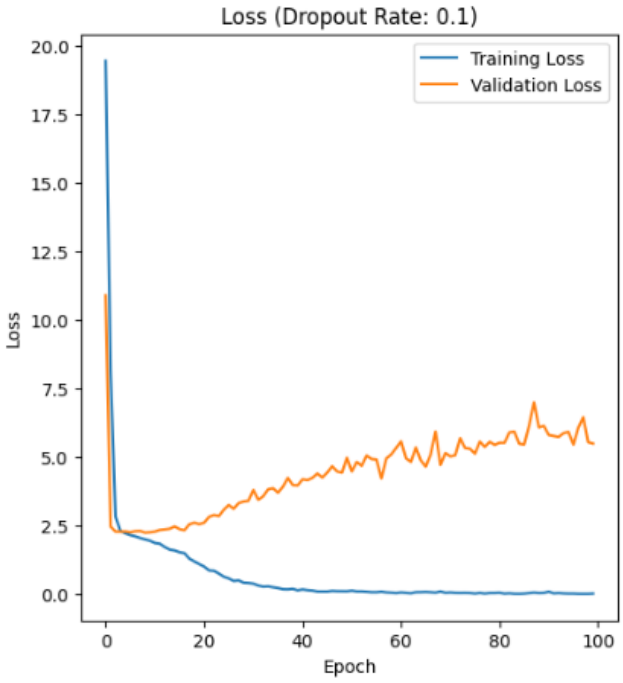
Best model: Best model withouth dropout info: {'batch_size': 32, 'learning_rate': 0.001, 'epoch': 89}
Best model withouth dropout validation accuracy: 0.24
Best model without dropot test accuracy:0.22

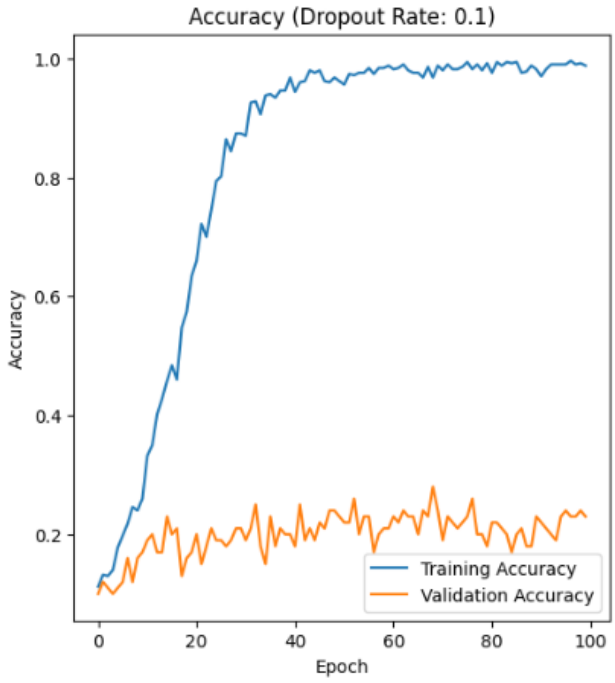


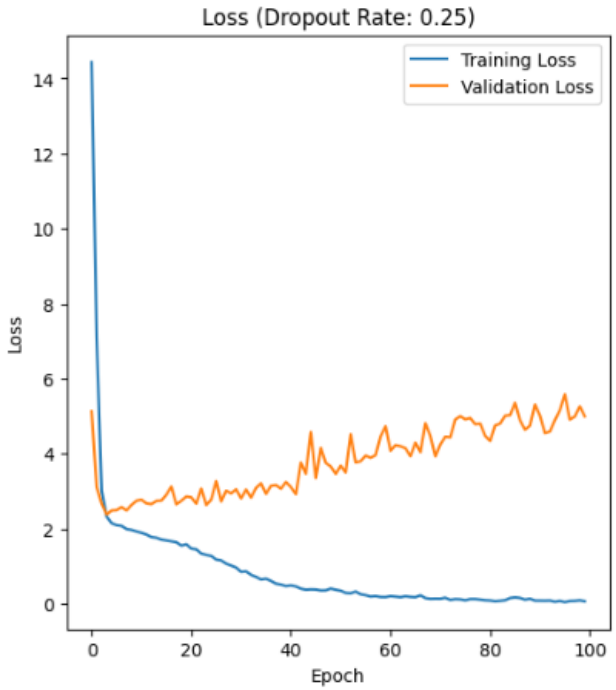
Dropout Rate [0.01, 0.1, 0.25, 0.5], batch:32 learning rate : 0.001

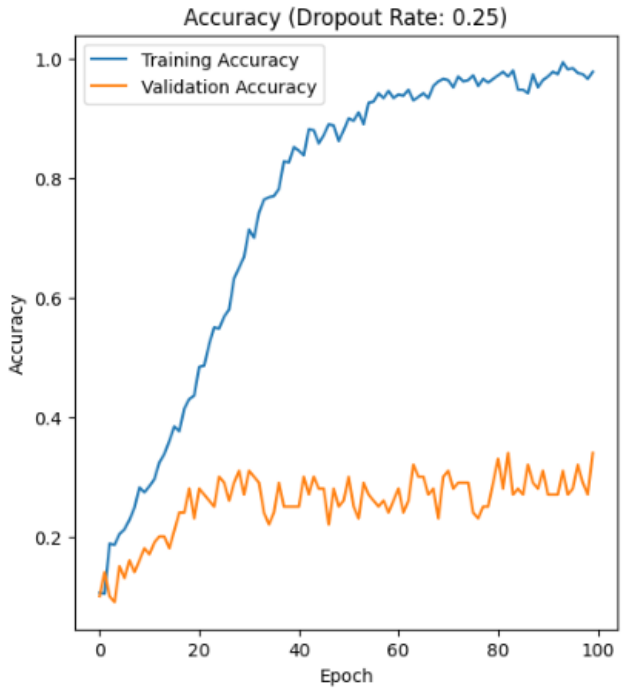

Loss (Dropout Rate: 0.01)
Training Loss
Validation Loss
Loss
Epoch

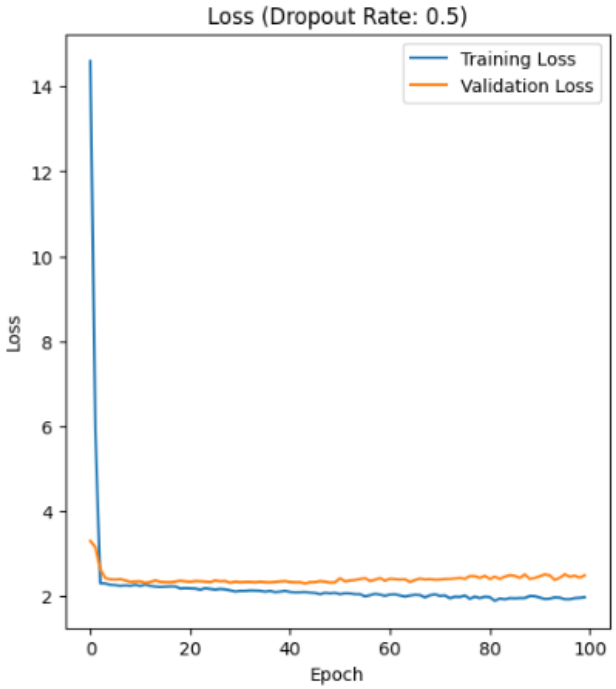

Accuracy (Dropout Rate: 0.01)
Training Accuracy
Validation Accuracy
Accuracy
Epoch

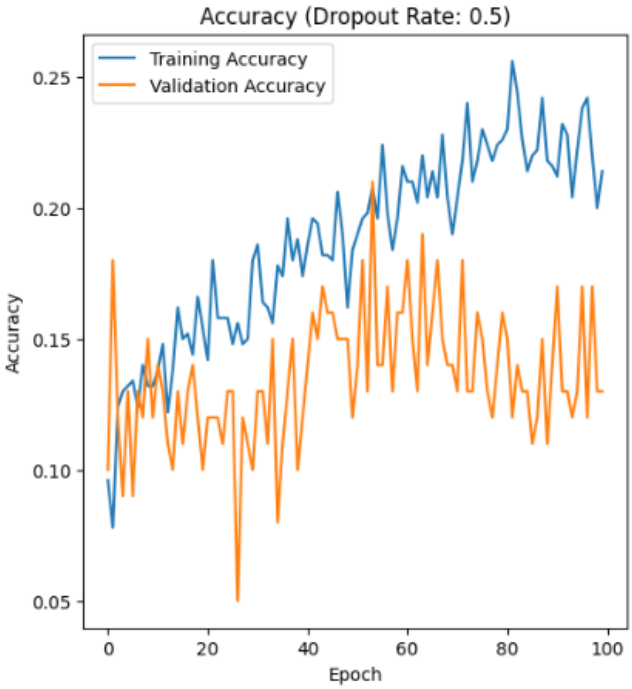

Loss (Dropout Rate: 0.1)
Training Loss
Validation Loss
Loss
Epoch


Accuracy (Dropout Rate: 0.1)
Training Accuracy
Validation Accuracy
Accuracy
Epoch









Best model dropout rate: 0.25 Best model validation accuracy: 0.34 Best model test accuracy: 0.32

3) **Select your best model with respect to validation accuracy and give test accuracy result.**

Best model dropout rate: 0.25 Best model validation accuracy: 0.34 Best model test accuracy: 0.32

The best model with respect to validation accuracy is the one trained with a dropout rate of 0.25 and learning rate of 0.001 and a batch size of 32. It achieved a validation accuracy of 0.34
The test accuracy of the best model is 0.32

PART 2 - Transfer Learning with EfficientNet

1) **What is fine-tuning? Why should we do this? Why do we freeze the rest and train only FC layers? Give your explanation in detail with relevant code snippet**

The process of training a neural network that has already been trained on a new dataset is called fine-tuning. From a sizable dataset, like EfficientNet, the pre-trained model has picked up helpful properties that it can apply to a new task. In order to adjust the model to the unique classes in the new dataset, fine-tuning entails changing and training the final layers, which are frequently the fully-connected layers.

Why should we fine-tune?

Faster Training: Pre-trained models have already learned to detect generic features like edges and textures, decreasing the quantity of data needed.
Better Performance: Using pre-trained weights generally results in higher accuracy than training from start.
Efficient Resource employ: Fine-tuning enables us to employ pre-trained models rather than starting from scratch, which saves computing resources.

Freezing Layers and Training Only FC Layers

Freezing the rest of the layers while training only the fully connected (FC) layers ensures that the core feature extraction layers (convolutional layers) retain their learned representations. This approach requires less training data and computational power.

2) **Explore training with two different cases; train only FC layer and freeze rest, train last two convolutional layers and FC layer and freeze rest. Tune your parameters accordingly and give accuracy on validation set and test set. Compare and analyze your results. Give relevant code snippet.**

```
def efficientnet(num_classes, train_fc_only=True,
train_last_two_blocks=False):
    model = models.efficientnet_b0(pretrained=True)


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

    if train_last_two_blocks:

        for layer in [model.features[-1], model.features[-2]]:
            for param in layer.parameters():
                param.requires_grad = True
    elif train_fc_only:

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


    model.classifier = nn.Sequential(
        nn.Dropout(p=0.2, inplace=True),
        nn.Linear(model.classifier[1].in_features, num_classes)
    )

    return model
```

Case 1: Train Only the FC Layer (Freeze Rest)

```
model_fc = efficientnet(len(classes), train_fc_only=True)
model_fc = model_fc.to(device)
optimizer_fc = optim.Adam(model_fc.classifier.parameters(), lr=0.001)

model_fc, val_acc_fc, train_losses_fc, train_accuracies_fc,
valid_losses_fc, valid_accuracies_fc = train_efficientnet(
    model_fc, criterion, optimizer_fc, train_loader, valid_loader,
num_epochs=100
)

test_acc_fc = evaluate_efficientnet(model_fc, test_loader)
print(f"Validation Accuracy (Train FC Only): {val_acc_fc:.4f}")
print(f"Test Accuracy (Train FC Only): {test_acc_fc:.4f}")
```

Validation Accuracy (Train FC Only): 0.7800

Test Accuracy (Train FC Only): 0.7900

Case 2: Train Last Two Convolutional Layers and FC Layer (Freeze Rest)

```
model_last_two_conv = initialize_efficientnet(len(classes),
train_last_two_blocks=True)
model_last_two_conv = model_last_two_conv.to(device)
optimizer_last_two_conv = optim.Adam(
    list(model_last_two_conv.classifier.parameters()) +
list(model_last_two_conv.features[-1].parameters()) +
list(model_last_two_conv.features[-2].parameters()),
    lr=0.001
)

model_last_two_conv, val_acc_last_two_conv, train_losses_last_two_conv,
train_accuracies_last_two_conv, valid_losses_last_two_conv,
valid_accuracies_last_two_conv = train_efficientnet(
    model_last_two_conv, criterion, optimizer_last_two_conv, train_loader,
valid_loader, num_epochs=100
)

test_acc_last_two_conv = evaluate_efficientnet(model_last_two_conv,
test_loader)
print(f"Validation Accuracy (Train Last Two Conv & FC):
{val_acc_last_two_conv:.4f}")
print(f"Test Accuracy (Train Last Two Conv & FC):
{test_acc_last_two_conv:.4f}")
```

Validation Accuracy (Train Last Two Conv & FC): 0.8200
Test Accuracy (Train Last Two Conv & FC): 0.8000

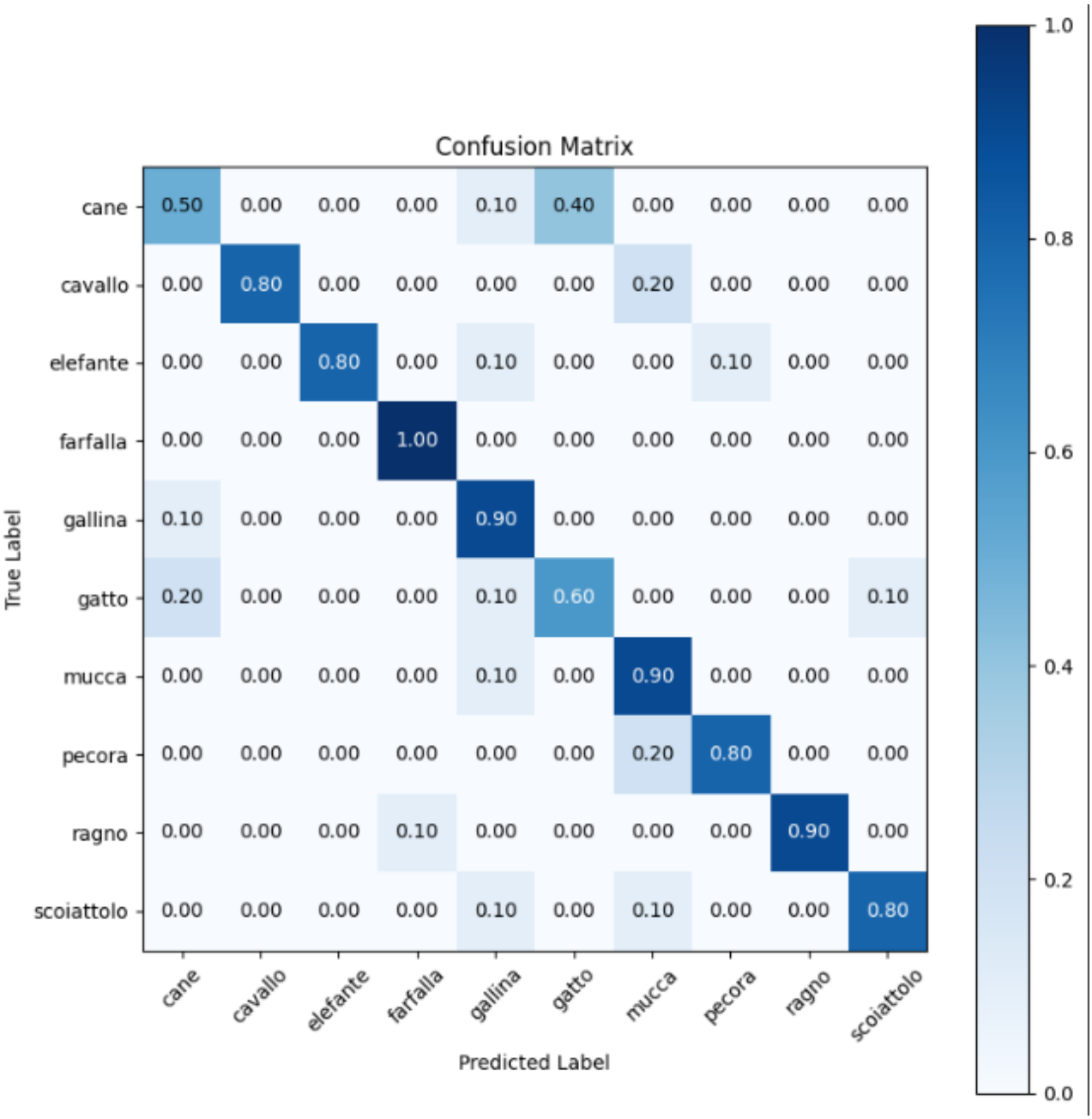
Analysis

Case 2 (Train Last Two Conv & FC) shows better performance compared to **Case 1** (Train FC Only)

In scenarios where accuracy is crucial and resources are available, training the last two convolutional layers and the fully connected layer (Case 2) is recommended for better adaptation and performance.

However, if training speed or computational resources are limited, training only the fully connected layer (Case 1) is a reasonable compromise with decent performance.

3) Plot confusion matrix for your best model and analyze results.



Compare and analyze your results in Part-1 and Part-2.

Comparison and Analysis:

Fine-tuning outperformed training from scratch in terms of accuracy and training efficiency.
Fine-tuning utilized the learned features of the pre-trained network, enabling better generalization and faster convergence.
Part-2 models achieved higher accuracy on the validation and test sets compared to the best model in Part-1