BBM418 Fundamentals of Computer Vision Laboratuary

M. Davut Kulaksız
21946419
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
davut.kulaksiz.n7@gmail.com

Overview

The aim of this assignment is to get familiar with image classification using **Convolutional Neural Networks (CNNs)** and transfer learning with a **pre-trained ResNet-18 model**. The project is implemented using PyTorch, a deep-learning framework. The dataset consists of images of bacteria, with 8 categories:

- Amoeba
- Euglena
- Hydra
- Paramecium
- Rod Bacteria
- Spherical Bacteria
- Spiral Bacteria
- Yeast

The assignment involves the following objectives:

- Implement **two CNN models** from scratch, one **with residual connections** and **one without**. Train and evaluate the models using different **learning rates** and **batch sizes**.
  - **LR = [0.01, 0.001]**
  - **Batch Sizes = [16, 32]**
- Investigate the impact of **dropout** on the performance of the models by experimenting with various dropout rates.
  - **Dropout Rates = [0.1, 0.3, 0.6, 0.8]**
- Train and evaluate the models using transfer learning with a **pre-trained ResNet-18 model**.
  - One model with freezing all the layers except the **Fully Connected** layer
  - One model with freezing all the layers except the **Fully Connected** layer and last two **Convolutional** layers.
- Analyze the results obtained from the CNN models and transfer learning. Compare the performance of the models and discuss the findings.

The project contains several **Jupyter Notebook** files. Each notebook implements a different model or set of parameters for the CNN:

Notebook	Description
first_attempt.ipynb	Playground notebook for exploring PyTorch. This notebook has detailed comments and observations on each implementation.
image_classification[1-4].ipynb	Implementations of a CNN without residual connections, with varying learning rates (0.001 or 0.01) and batch sizes (16 or 32).
image_classification[5-8].ipynb	Implementations of a CNN with residual connections, with varying learning rates (0.001 or 0.01) and batch sizes (16 or 32).
image_classification_dropout[1-4].ipynb	Implementations of the CNN from image_classification.ipynb with dropout rates from 0.1 to 0.8.
image_classification_7_dropout[1-4].ipynb	Implementations of the CNN from image_classification7.ipynb with dropout rates from 0.1 to 0.8.
part2/resnet.ipynb	Implementation of transfer learning with the ResNet-18 model, training only the FC layer.
part2/resnet2.ipynb	Implementation of transfer learning with the ResNet-18 model, training the last two convolutional layers and the FC layer.

Table 1: Overview of the Jupyter Notebook files in the project

1 Dataset

The dataset used for this assignment consists of bacteria images in both **.jpg** and **.png** formats. These images are categorized into **8 different classes**, representing various types of bacteria. The class distribution and the number of images in each class are as follows:

- Amoeba: 72 images
- Euglena: 168 images
- Hydra: 76 images
- Paramecium: 152 images
- Rod bacteria: 85 images
- Spherical bacteria: 86 images
- Spiral bacteria: 75 images
- Yeast: 75 images

1.1 Splitting the Dataset

Firstly, I needed to load the data as tensors to work with them in **PyTorch**. The following section explains the process of loading the data and transforming the images. Afterwards, I discuss how the dataset was split and how the number of images in each class was counted.

1.1.1 Loading the Data

To load the data as tensors, I imported the **ImageFolder** function from the **torchvision** library. Since the images have different dimensions, it was necessary to **resize** them. Additionally, some **Data Augmentation** techniques were applied, which will be discussed later in more detail.

```
import os
import matplotlib
import matplotlib.pyplot as plt

%matplotlib inline

dataset_path = './Micro_Organism'

from torchvision.datasets import ImageFolder
from torchvision.transforms import ToTensor, Resize, Compose, Normalize, RandomHorizontalFlip,
    RandomRotation, RandomVerticalFlip, ColorJitter

transform = Compose([
    Resize((TENSOR_SIZE, TENSOR_SIZE)),
    ToTensor(),
    RandomRotation(degrees=30), # new here
    RandomHorizontalFlip(), # new here
    RandomVerticalFlip(), # new here
    ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0.1), # new here
    Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

dataset = ImageFolder(dataset_path, transform=transform)
```

1.1.2 Splitting the Dataset into Train, Validation, and Test Sets

To divide the dataset into **Train**, **Validation**, and **Test** sets, I used the random_split function from **torch** after specifying the lengths for each set.

- The training dataset has 400 images.
- The validation dataset has 160 images.
- The test dataset has 229 images.

I set the **manual_seed** value to **42**. While this number has no particular importance, it is a reference to "The Hitchhiker's Guide to the Galaxy," and I decided to keep it as 42 for a bit of fun.

```
import torch
from torch.utils.data import random_split

torch.manual_seed(42)
train_size = 400
val_size = 160
test_size = len(dataset) - (train_size + val_size)

train_dataset, validation_dataset, test_dataset = random_split(dataset,
                                                               [train_size, val_size, test_size])
```

1.1.3 Calculating the Class Distribution

To ensure that the **Validation** and **Test** sets contain at least 10 images per class, I created a function named *count_labels*. This function calculates the class distribution for train, validation, and test datasets by counting the number of each class in these sets.

The function takes three arguments:

- *dataset_subset*: The subset of the dataset for which the class distribution needs to be calculated (train, validation, or test).
- *dataset_targets*: The list of target labels for the entire dataset.
- *num_classes*: The total number of classes in the dataset.

```
def count_labels(dataset_subset, dataset_targets, num_classes):
indices = dataset_subset.indices
labels = [dataset_targets[i] for i in indices]
counts = [labels.count(i) for i in range(num_classes)]
return counts

num_classes = 8

labels = dataset.targets

train_counts = count_labels(train_dataset, labels, num_classes)
val_counts = count_labels(validation_dataset, labels, num_classes)
test_counts = count_labels(test_dataset, labels, num_classes)

print("Training set class distribution:", train_counts)
print("Validation set class distribution:", val_counts)
print("Test set class distribution:", test_counts)
```

The function works as follows:

1. Retrieves the indices of the samples in the dataset subset.
2. Creates a list of labels corresponding to the indices.
3. Counts the occurrences of each class in the list of labels and returns the counts.

Here's an example of the output for the *count_labels* function in the *first_attempt.ipynb* notebook:

- Training set class distribution: [33, 82, 33, 78, 47, 46, 42, 39]
- Validation set class distribution: [15, 32, 10, 35, 17, 17, 18, 16]
- Test set class distribution: [24, 54, 33, 39, 21, 23, 15, 20]

The numbers in each list represent the counts of samples per class for the corresponding dataset subset. These distributions show that there are at least 10 samples for each class in the **Validation** and **Test** sets.

2 PART 1 - Modeling and Training a CNN classifier from Scratch

In this part of the assignment, I designed and trained **CNN classifiers**. Each model consists of six **Convolutional** layers followed by a **Fully Connected** layer. After the first **Convolutional** layer, a **Max Pooling** layer is added. Half of the models are designed without **Residual Connections**, while the other half has Residual Connections.

2.1 CNN Architecture

2.1.1 CNN Model without Residual Connections

The implemented CNN model without residual connections is composed of six Convolutional layers and a Fully Connected layer. The detailed architecture is as follows:

```
class CNNModel(ImageClassificationModel):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),

            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),

            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),

            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),

            nn.Flatten(),
            nn.Linear(128 * int(TENSOR_SIZE / 2) * int(TENSOR_SIZE / 2), 8))

    def forward(self, xb):
        return self.network(xb)
```

In the implemented CNN model, the parametric details are as follows:

- **Input Channels:** Since our dataset has **RGB** images, the first layer starts with **3 input channels**. For grayscale images, this number would be 1.
- **Output Channels:** I experimented with different output channel configurations while considering memory constraints with CUDA and feature extraction capabilities. The tested configurations were, each element is a number of the output channels of a conv layer sequentially:
  - 32, 64, 128, 128, 256, 256, 8
  - 64, 128, 256, 256, 512, 512, 8
  - 16, 32, 64, 64, 128, 128, 8

  The optimal configuration was the third option: **[16, 32, 64, 64, 128, 128, 8]**.
- **Stride:** The stride was set to **1** for all convolutional layers to move the kernel one step at a time, ensuring that all available information is processed.

- **Padding:** A padding of **1** was applied to all convolutional layers to maintain the input tensor dimensions after convolution.
- **Flatten Layer:** The Flatten layer reshapes the multidimensional tensor into a one-dimensional tensor, enabling the transition from convolutional layers to fully connected layers.
- **ReLU (Rectified Linear Unit):** ReLU serves as the activation function, replacing all negative input values with 0 and introducing non-linearity to the model. It is defined as $ReLU(x) = max(0, x)$. This non-linearity is essential for the model to learn complex tasks like image classification effectively.
- **Kernel Size:** The kernel size was set to 3 for all convolutional layers, which is a common practice for achieving a balance between local feature extraction and information compression before a pooling layer.

2.1.2 CNN Model with Residual Connections

The implemented CNN model with one residual connection is composed of six Convolutional layers and a Fully Connected layer. The detailed architecture is as follows:

```
class CNNModelResidual(ImageClassificationModel):
    def __init__(self):
        super(CNNModelResidual, self).__init__()

        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.relu1 = nn.ReLU()
        self.pool = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()

        self.conv3 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.relu3 = nn.ReLU()

        self.conv4 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)
        self.relu4 = nn.ReLU()

        self.conv5 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
        self.relu5 = nn.ReLU()

        self.conv6 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1)
        self.relu6 = nn.ReLU()

        self.flatten = nn.Flatten()
        self.connected1 = nn.Linear(128 * int(TENSOR_SIZE / 2) * int(TENSOR_SIZE / 2), 8)

    def forward(self, value):
        # First Conv
        out = self.conv1(value)
        out = self.relu1(out)
        out = self.pool(out)

        # Second Conv
        out = self.conv2(out)
        out = self.relu2(out)

        # Third Conv
        out = self.conv3(out)
        out = self.relu3(out)

        # Fourth Conv
        res4 = out
```

```
        out = self.conv4(out)
        out = self.relu4(out)
        out = out + res4  # Residual connection

        # Fifth Conv
        out = self.conv5(out)
        out = self.relu5(out)

        # Sixth Conv
        out = self.conv6(out)
        out = self.relu6(out)

        ## Fully Connected Layer
        out = self.flatten(out)
        out = self.connected1(out)

        return out
```

In terms of parametric details, this model is the same as the one without any residual connections. However, in this model, a residual connection is added in the **fourth Convolutional layer**. The residual connection is implemented by first storing the output of the **third Convolutional layer** in a variable called **res4**. The model then processes the output using the **fourth Convolutional layer (conv4)** and its **activation function (relu4)**.

After applying the convolution and activation, the residual connection is created by adding the stored output from the third convolutional layer (res4) back to the output of the fourth Convolutional layer. This addition operation creates a shortcut connection, allowing the model to learn more efficiently by propagating gradients directly from the output of the fourth convolutional layer to the input of the same layer.

Here is the relevant code snippet for the **Residual Connection**:

```
...
# Fourth Conv
res4 = out # Store the output of the third convolutional layer
out = self.conv4(out) # Apply the fourth convolutional layer
out = self.relu4(out) # Apply the ReLU activation function
out = out + res4  # Add the stored output (res4) to create the residual connection
...
```

This residual connection helps improve the model's performance by enabling it to learn more complex features.

2.2 Training and Evaluating the Model

The following table summarizes the parameters and results of each **Jupyter notebook** used for **training** and **evaluating the models**.

Notebook	Residual	Learning Rate	Batch Size	Dropout Rate	Accuracy (%)
image_classification.ipynb	No	0.001	16	-	33
image_classification2.ipynb	No	0.01	16	-	20
image_classification3.ipynb	No	0.001	32	-	33
image_classification4.ipynb	No	0.01	32	-	20
image_classification5.ipynb	Yes	0.001	16	-	34
image_classification6.ipynb	Yes	0.01	16	-	20
image_classification7.ipynb	Yes	0.001	32	-	35
image_classification8.ipynb	Yes	0.01	32	-	20
image_classification_dropout.ipynb	No	0.001	16	0.1	33
image_classification_dropout2.ipynb	No	0.001	16	0.3	38
image_classification_dropout3.ipynb	No	0.001	16	0.6	32
image_classification_dropout4.ipynb	No	0.001	16	0.8	20
image_classification_7_dropout.ipynb	Yes	0.001	32	0.1	33
image_classification_7_dropout2.ipynb	Yes	0.001	32	0.3	37
image_classification_7_dropout3.ipynb	Yes	0.001	32	0.6	20
image_classification_7_dropout4.ipynb	Yes	0.001	32	0.8	20

- The best model **without residual connection** is **image_classification.ipynb** with:
  - **Batch size:** 16
  - **Learning rate:** 0.001
  - **Validation accuracy:** 33%
- The best model **with residual connection** is **image_classification7.ipynb** with:
  - **Batch size:** 32
  - **Learning rate:** 0.001
  - **Validation accuracy:** 35%

When dropout rates are introduced:

Model	Dropout Rate	Validation Acc.	Test Acc.
image_classification_droput.ipynb	0.1	33%	25%
image_classification_droput2.ipynb	0.3	38%	29%
image_classification_droput3.ipynb	0.6	32%	21%
image_classification_droput4.ipynb	0.8	20% (stopped)	23%
image_classification_7_dropout.ipynb	0.1	33%	33%
image_classification_7_dropout2.ipynb	0.3	37%	30%
image_classification_7_dropout3.ipynb	0.6	20% (stopped)	22%
image_classification_7_dropout4.ipynb	0.8	20% (stopped)	18%

Table 2: Results of the models with varying dropout rates.

- **(Without residual connection)** The highest test accuracy (29%) is achieved with **image_classification_droput2.ipynb**, and:
  - **Dropout rate:** 0.3
  - **Batch Size:** 16
  - **Learning Rate:** 0.001
  - **Validation Accuracy:** 38%
  - **Test Accuracy:** 29%

- **(With a residual connection)** The highest test accuracy (33%) is achieved with **image_classification_7_2dropout.ipynb**, and:
  - **Dropout rate:** 0.3
  - **Batch size:** 32
  - **Learning Rate:** 0.001
  - **Validation Accuracy:** 32%
  - **Test Accuracy:** 37%

It is worth noting that in some cases, **the models stopped learning** when the learning rate was set to 0.01, and when both the learning rate and dropout rate were too high. ***This suggests that these hyperparameters need to be carefully tuned to achieve better performance.***

In conclusion, **the models with residual connections generally show slightly better performance compared to the models without residual connections**. Furthermore, **introducing dropout helps improve the model's performance**, with an optimal **dropout rate of around 0.1 to 0.3**.

2.2.1 Accuracy

The accuracy function calculates the accuracy of the model's predictions for a given batch of images.

```
1 def accuracy(outputs, labels):
2   _, preds = torch.max(outputs, dim=1)
3   return torch.tensor(torch.sum(preds == labels).item() / len(preds))
```

This function takes two arguments:

- **outputs:** the predicted class scores from the model
- **labels:** the true class labels

It calculates the accuracy of the predictions by performing the following steps:

- It computes the maximum value along columns of the outputs tensor using torch.max. This function returns two values below and stores the indices in **preds**.
  - the maximum values
  - the corresponding indices of the maximum values
- It compares the predicted class indices (preds) with the true class labels. This results in boolean values, with True representing a correct prediction and False representing a false prediction.
- It calculates the total number of correct predictions by summing the boolean tensor with torch.sum and converting it to a Python scalar using the item() method.
- It divides the total number of correct predictions by the total number of predictions (length of preds) to obtain accuracy.

```python
class ImageClassificationModel(nn.Module):
  def training_step(self, batch):
    images, labels = batch
    # Generate predictions
    out = self(images)
    # Calculate loss
    loss = F.cross_entropy(out, labels)
    return loss

  def validation_step(self, batch):
    images, labels = batch
    out = self(images)
    loss = F.cross_entropy(out, labels)
    # Calculate accuracy
    acc = accuracy(out, labels)
    return {'validation_loss': loss.detach(), 'validation_accuracy': acc}

  def validation_epoch_end(self, outputs):
      batch_losses = [x['validation_loss'] for x in outputs]
      # Combine losses
      epoch_loss = torch.stack(batch_losses).mean()
      batch_accs = [x['validation_accuracy'] for x in outputs]
      # Combine accuracies
      epoch_acc = torch.stack(batch_accs).mean()
      return {'validation_loss': epoch_loss.item(), 'validation_accuracy': epoch_acc.item()}

  def epoch_end(self, epoch, result):
      train_loss, val_loss, val_acc = result['train_loss'],
      result['validation_loss'], result['validation_accuracy']
      print(f"Epoch [{epoch}/{EPOCH_NUMBER}], Training Loss: {train_loss:.4f},
      Validation Loss: {val_loss:.4f}, Validation Accuracy: {val_acc:.4f}")
```

ImageClassificationModel extends nn.Module and it gets extended by the **CNN Models** later on. It provides **training**, **validation**, and **epoch progress** methods for handling them.

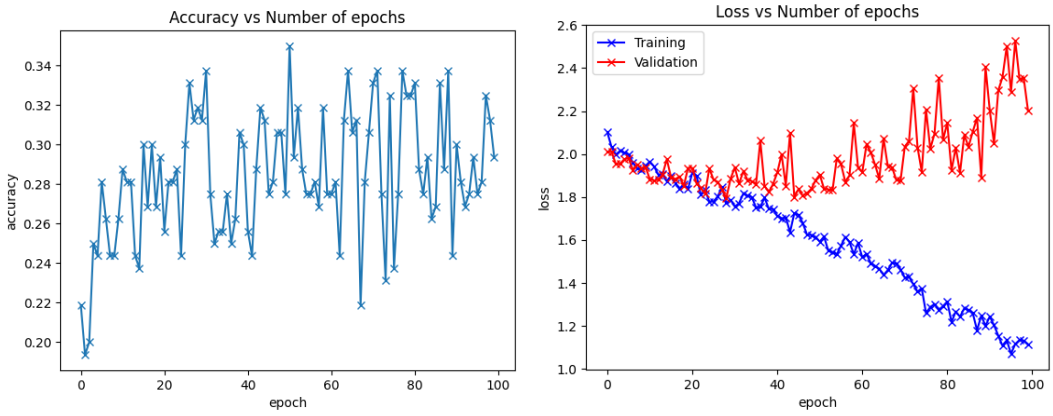
- **training_step(self, batch):** Takes a batch of input data (images and labels) and computes the loss for the training step. It generates predictions by passing the images through to the model and calculates the loss using the **cross-entropy function**.
- **validation_step(self, batch):** Takes a batch of input data (images and labels) and computes the loss and accuracy for the validation step. It generates predictions by passing the images to the model, calculates the loss using the cross-entropy function, and computes the accuracy using the accuracy function. The method returns a dictionary containing the validation loss and accuracy.
- **validation_epoch_end(self, outputs):** Takes a list of dictionaries containing the validation losses and accuracies from all the validation steps in an epoch. It combines the losses and accuracies by calculating their mean values and returns a dictionary containing the mean validation loss and accuracy.
- **epoch_end(self, epoch, result):** Gets called at the end of each epoch and prints the **epoch number**, **training loss**, **validation loss**, and **validation accuracy**.

2.2.2 Plotting Losses and Accuracies

```
def plot_accuracies(history):
    accuracies = [x['validation_accuracy'] for x in history]
    plt.plot(accuracies, '-x')
    plt.xlabel('epoch')
    plt.ylabel('accuracy')
    plt.title('Accuracy vs Number of epochs')

plot_accuracies(history)
```

```
def plot_losses(history):
    train_losses = [x.get('train_loss') for x in history]
    val_losses = [x['validation_loss'] for x in history]
    plt.plot(train_losses, '-bx')
    plt.plot(val_losses, '-rx')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.legend(['Training', 'Validation'])
    plt.title('Loss vs Number of epochs')

plot_losses(history)
```

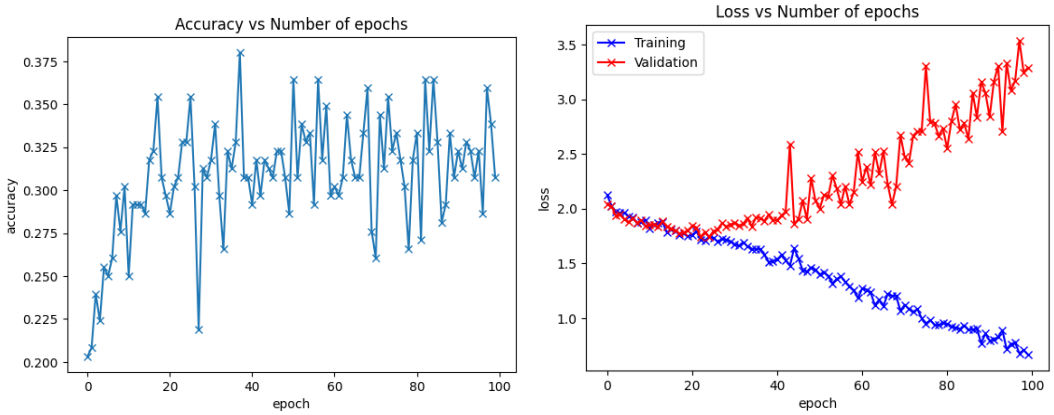
- **plot_losses function**: This function takes a list of dictionaries containing the training loss, validation loss, and validation accuracy for each epoch as an argument. Then the function first extracts the training and validation losses from this and stores them in two separate lists, **train_losses** and **val_losses**. After that, it uses Matplotlib to plot these losses against the number of epochs.
- **plot_accuracies:** This function also just like plot_loses function takes a history object as input. It extracts the validation accuracies from the history object and stores them in a list called accuracies. The function then uses Matplotlib to plot the accuracies against the number of epochs.



(a) Accuracy vs Number of Epochs

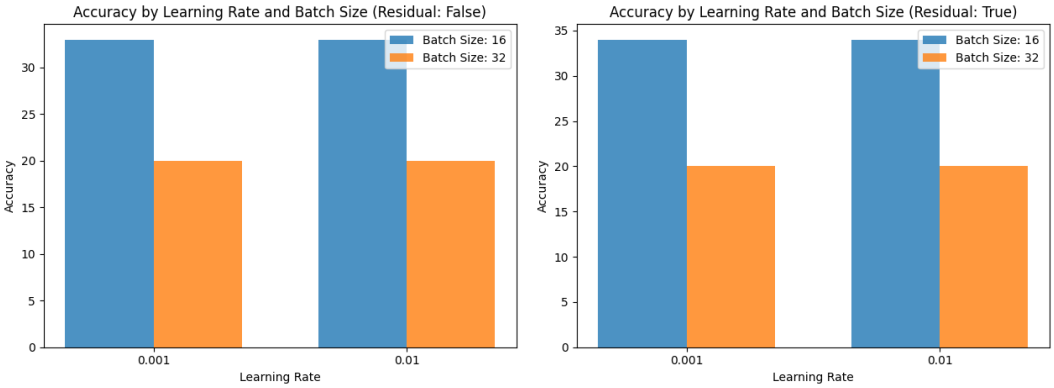
(b) Loss ve Number of Epochs

Figure 1: The plots for the best model without residual connection: **image_classification.ipynb**



(a) Accuracy vs Number of Epochs (b) Loss vs Number of Epochs

Figure 2: The plots for the best model without residual connection: **image_classification_7.ipynb**



(a) Without Residual Connection (b) With Residual Connection

Figure 3: Accuracy vs Learning Rate for Different Batch Sizes

2.2.3 Dropouts

I added dropouts to each Convolutional layer, just after the activation function. Dropouts help to prevent overfitting in models, and since I had huge problems with overfitting on the previous models, I used them in each Conv. layer.

```
class CNNModel(ImageClassificationModel):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Dropout(dropout_rate),
            nn.MaxPool2d(2, 2),

            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(dropout_rate),

            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(dropout_rate),

            nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(dropout_rate),

            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(dropout_rate),

            nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1),
            nn.ReLU(),
            nn.Dropout(dropout_rate),

            nn.Flatten(),
            nn.Linear(128 * int(TENSOR_SIZE / 2) * int(TENSOR_SIZE / 2), 8))

    def forward(self, xb):
        return self.network(xb)
```

The validation and test accuracies are varying for different dropout rates.

Model	Dropout Rate	Validation Acc.	Test Acc.
image_classification_droput.ipynb	0.1	33%	25%
image_classification_droput2.ipynb	0.3	38%	29%
image_classification_droput3.ipynb	0.6	32%	21%
image_classification_droput4.ipynb	0.8	20% (stopped)	23%
image_classification_7_dropout.ipynb	0.1	33%	33%
image_classification_7_dropout2.ipynb	0.3	37%	30%
image_classification_7_dropout3.ipynb	0.6	20% (stopped)	22%
image_classification_7_dropout4.ipynb	0.8	20% (stopped)	18%

Table 3: Results of the models with varying dropout rates.

- The models with dropout rates of 0.1 and 0.3 generally have higher validation and test accuracies than those with higher dropout rates.
- The models with dropout rates of 0.6 and 0.8 generally have lower validation and test accuracies, and some of them even stopped training before completing all epochs.

Therefore, it appears that a moderate dropout rate (e.g., 0.1 or 0.3) is effective in improving the model. While a higher rate is not effective, even more so it even can make the model stop learning.

Confusion Matrices:

- **image_classification_droput2.ipynb**

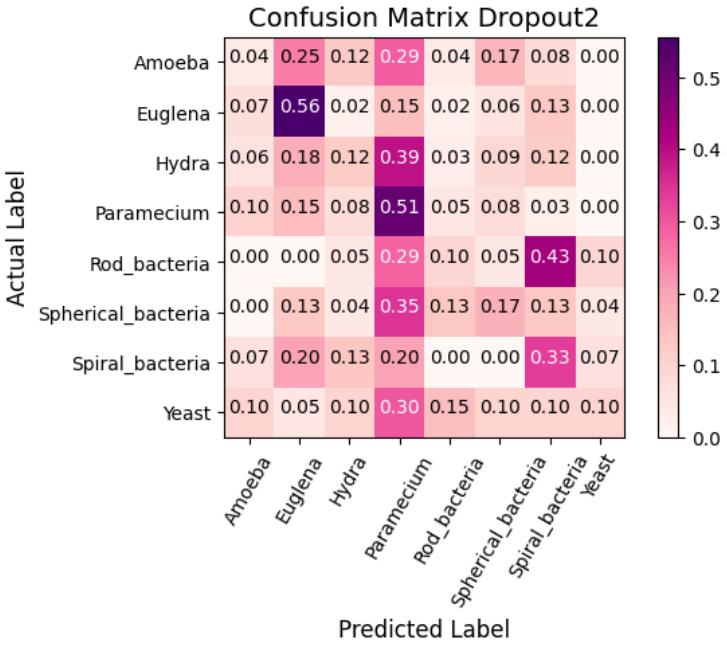


Figure 4: Dropout Rate: 0.3

- **image_classification_7_dropout2.ipynb**

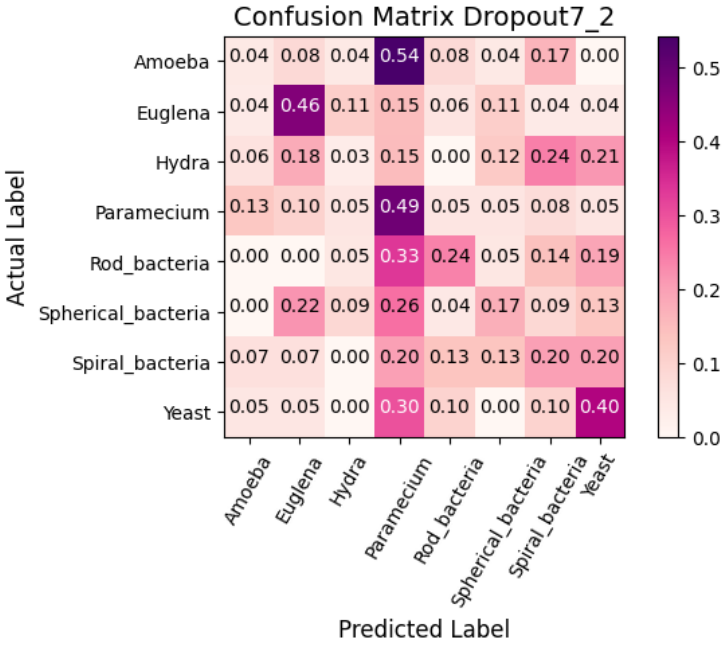


Figure 5: Dropout Rate: 0.3

By looking at the first matrix, we can say that:

- The model made a prediction that the microorganism belongs to the 'Spiral_bacteria' class in 43% of the instances where the microorganism actually belonged to the 'Rod_bacteria' class. In other words, the model confused the two classes for 43% of the cases where the true label was 'Rod_bacteria'.
- The model made a prediction that the microorganism belongs to the 'Paramecium' class in 30% of the instances where the microorganism actually belonged to the 'Yeast' class. In other words, the model confused the two classes for 30% of the cases where the true label was 'Yeast'.
- The model made a prediction that the microorganism belongs to the 'Euglena' class in 25% of the instances where the microorganism actually belonged to the 'Amoeba' class. In other words, the model confused the two classes for 25% of the cases where the true label was 'Amoeba'.
- The model made a prediction that the microorganism belongs to the 'Paramecium' class in 39% of the instances where the microorganism actually belonged to the 'Hydra' class. In other words, the model confused the two classes for 39% of the cases where the true label was 'Hydra'.

And by looking at the second matrix, we can say that:

- The model made a prediction that the microorganism belongs to the 'Paramecium' class in 54% of the instances where the microorganism actually belonged to the 'Amoeba' class. In other words, the model confused the two classes for 54% of the cases where the true label was 'Amoeba'.
- The model made a prediction that the microorganism belongs to the 'Paramecium' class in 33% of the instances where the microorganism actually belonged to the 'Rod_bacteria' class. In other words, the model confused the two classes for 33% of the cases where the true label was 'Rod_bacteria'.
- The model made a prediction that the microorganism belongs to the 'Paramecium' class in 30% of the instances where the microorganism actually belonged to the 'Yeast' class. In other words, the model confused the two classes for 30% of the cases where the true label was 'Yeast'.
- The model made a prediction that the microorganism belongs to the 'Paramecium' class in 26% of the instances where the microorganism actually belonged to the 'Spherical_bacteria' class. In other words, the model confused the two classes for 26% of the cases where the true label was 'Spherical_bacteria'.

This information can help to identify which classes are more prone to being confused by the model. Then, we can find ways to improve the accuracy of such classes.

2.3 Findings and Results

Each section, subsection, and subsubsection have its own detailed findings and results. However, in summary, we can conclude these:

- The best model without a residual connection is **image_classification.ipynb**
  - Batch Size: 16
  - LR = 0.001
- The best model with a residual connection is **image_classification7.ipynb**
  - Batch Size: 32
  - LR = 0.001
- The best model without a residual connection but with dropout: **image_classification_droput2.ipynb**
  - Batch Size: 16
  - LR = 0.001
  - Dropout Rate: 0.3
- The best model with a residual connection but with dropout: **image_classification_7_2dropout.ipynb**
  - Batch Size: 32
  - LR = 0.001
  - Dropout Rate: 0.3
- When dropout rates are introduced, the models generally show better performance, with an **optimal dropout rate of around 0.1 to 0.3**.
- Models with higher dropout rates generally have lower validation and test accuracies and some of them even stopped training before completing all epochs.
- **Confusion Matrices** can help to identify which classes are more prone to being confused by the model. Then we can use this information to improve the accuracy of these classes.

For more detailed comments and results, please check the sections and for even more comments about each step of the implementations, please check the Jupyter Notebooks.

3 PART 2 - Transfer Learning with CNNs

In this part, I fine-tuned the pre-trained ResNet-18 network which is available at PyTorch.

3.1 What is fine-tuning? Why should we do this? Why do we freeze the rest and train only FC layers?

Fine-tuning is the process of taking a pre-trained neural network and training it on a new dataset. The goal is to use the knowledge learned by the pre-trained network on a new dataset and adapt it.

We freeze the pre-trained layers of the network because they are already trained with the original dataset and they know relevant features. By freezing the layers, we protect the weights from being updated and prevent the features from being lost.

In our case, we froze all the layers of the ResNet-18 network except for the fully connected (FC) layer. This is because the FC layer is responsible for the final classification based on the features learned by the convolutional layers.

Downloading the pre-trained ResNet18 with most up to date weights:

```
from torchvision import models
from torchvision.models import ResNet18_Weights

# Load the pre-trained ResNet-18 model
resnet18 = models.resnet18(weights=ResNet18_Weights.DEFAULT)
resnet18
```

From https://pytorch.org/docs/master/notes/autograd.html:

"Setting requires_grad should be the main way you control which parts of the model are part of the gradient computation, for example, if you need to freeze parts of your pre-trained model during model fine-tuning. To freeze parts of your model, simply apply .requires_grad_(False) to the parameters that you don't want updated. And as described above, since computations that use these parameters as inputs would not be recorded in the forward pass, they won't have their .grad fields updated in the backward pass because they won't be part of the backward graph in the first place, as desired."

```
# Freeze all layers
for param in resnet18.parameters():
    param.requires_grad = False

# Unfreeze the FC layer
for param in resnet18.fc.parameters():
    param.requires_grad = True
```

Since ResNet18 has 512 input features on its last FC. We can update the last layer for our classification task with our number of classes as output channels.

```
resnet18.fc = nn.Linear(512, 8)
```

We can do these by creating a new custom **ResNet** model by extending the **ImageClassificationModel** class definition as well:

```
class CustomResNet(ImageClassificationModel):
    def __init__(self, num_classes=8):
        super().__init__()
        self.resnet = models.resnet18(weights=ResNet18_Weights.DEFAULT)
        self.resnet.fc = nn.Linear(in_features=512, out_features=num_classes)

    def forward(self, x):
        return self.resnet(x)
```

This way we can also use the methods the **ImageClassificationModel** class has.

3.2 Two Different Cases

- **Train Only FC Layer and Freeze Rest:**

```
class CustomResNet(ImageClassificationModel):
    def __init__(self, num_classes=8):
        super().__init__()
        self.resnet = models.resnet18(weights=ResNet18_Weights.DEFAULT)
        self.resnet.fc = nn.Linear(in_features=512, out_features=num_classes)

    def forward(self, x):
        return self.resnet(x)
```

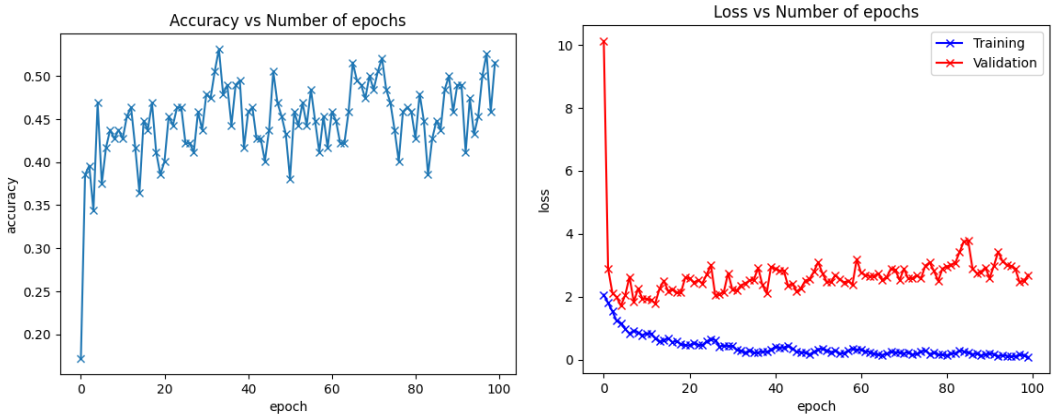
- **Train Last Two Convolutional Layers and FC Layer and Freeze Rest:**

```
class CustomResNet(ImageClassificationModel):
    def __init__(self, num_classes=8):
        super().__init__()
        self.resnet = models.resnet18(weights=ResNet18_Weights.DEFAULT)

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

        # Unfreeze the last two convolutional layers in layer4
        for param in self.resnet.layer4[1].parameters():
            param.requires_grad = True

        # Replace and unfreeze the FC layer
        self.resnet.fc = nn.Linear(in_features=512, out_features=num_classes)
        for param in self.resnet.fc.parameters():
            param.requires_grad = True

    def forward(self, x):
        return self.resnet(x)
```

Fine-tuning Case	Validation Accuracy	Test Accuracy
Train only FC layer	47%	51%
Train last two convolutional layers and FC layer	59%	55%

Table 4: Comparison of validation and test accuracies for two different cases.

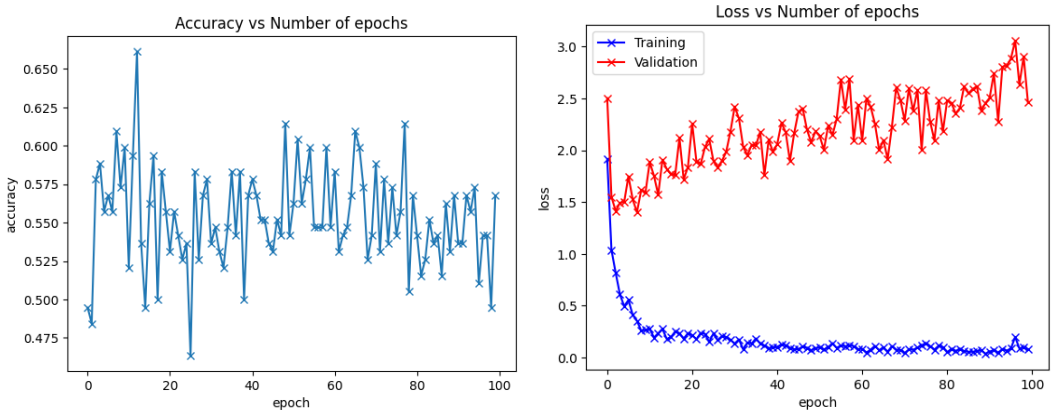
By looking at the table, we can say that it is clear that fine-tuning the last two convolutional layers and the FC layer results in higher validation and test accuracies compared to only fine-tuning the FC layer. The model fine-tuned with the last two convolutional layers and the FC layer achieved a validation accuracy of 59% and a test accuracy of 55%, while the model fine-tuned only with the FC layer achieved a validation accuracy of 47% and a test accuracy of 51%. Therefore, we can conclude that fine-tuning more layers in the pre-trained model can lead to better performance.



(a) Accuracy vs Number of Epochs (b) Loss ve Number of Epochs

Figure 6: Train only FC layer

This model seems to be overfitting as the training loss is getting lower but the validation loss is getting higher, however, the results are much better here. The accuracy is consistently over 40% and on some points, it's over 50%. However if we take the point 'Epoch [4/100], Training Loss: 1.1564, Validation Loss: 1.7106, Validation Accuracy: 0.4688' as the point where validation loss goes higher while training loss drops by looking at Loss vs Number of Epochs plot, then our accuracy can be considered as '47%'



(a) Accuracy vs Number of Epochs (b) Loss ve Number of Epochs

Figure 7: Train last two convolutional layers and FC layer

This model also seems to be overfitting as the training loss is getting lower but the validation loss is getting higher, however, the results are much better here. The accuracy is consistently over 40% and on some points, it's over 50%. However if we take the point 'Epoch [3/100], Training Loss: 0.6082, Validation Loss: 1.4915, Validation Accuracy: 0.5885' as the point where validation loss goes higher while training loss drops by looking at Loss vs Number of Epochs plot, then our accuracy can be considered as '59%'

3.3 Confusion Matrices

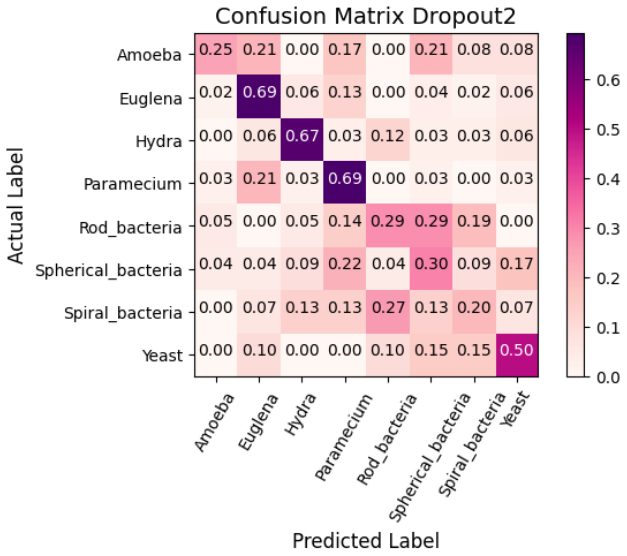


Figure 8: Train only FC layer - 'Confusion Matrix Dropout2' is the outdated title

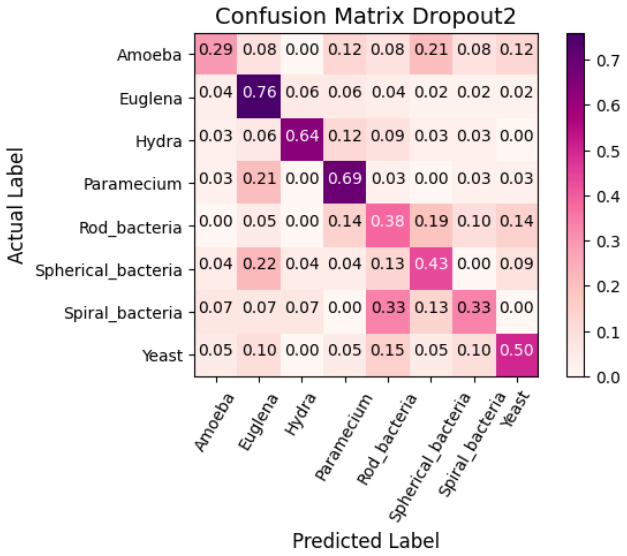


Figure 9: Train last two convolutional layers and FC layer - 'Confusion Matrix Dropout2' is the outdated title

By looking at the first matrix, we can say that:

- The model made a prediction that the microorganism belongs to the 'Spherical_bacteria' class in 29% of the instances where the microorganism actually belonged to the 'Rod_bacteria' class. In other words, the model confused the two classes for 29% of the cases where the true label was 'Rod_bacteria'.
- The model made a prediction that the microorganism belongs to the 'Rod_bacteria' class in 27% of the instances where the microorganism actually belonged to the 'Spiral_bacteria' class. In other words, the model confused the two classes for 27% of the cases where the true label was 'Spiral_bacteria'.

And by looking at the second matrix, we can say that:

- The model made a prediction that the microorganism belongs to the 'Rod_bacteria' class in 33% of the instances where the microorganism actually belonged to the 'Spiral_bacteria' class. In other words, the model confused the two classes for 33% of the cases where the true label was 'Amoeba'.
- The model made a prediction that the microorganism belongs to the 'Euglena' class in 22% of the instances where the microorganism actually belonged to the 'Spherical_bacteria' class. In other words, the model confused the two classes for 22% of the cases where the true label was 'Spherical_bacteria'.

This information can help to identify which classes are more prone to being confused by the model. Then, we can find ways to improve the accuracy of such classes.

3.4 Part 1 vs Part 2

In the first part:

- **The best model without a residual connection** had a validation accuracy of 33% and it has a **learning rate of 0.001** and a **batch size of 16**.
- **The best model with a residual connection** had a validation accuracy of 35% and it has a **learning rate of 0.001** and **a batch size of 32**.

Later, introducing dropout generally helped improve the model's performance with an optimal dropout rate of around 0.1 to 0.3. However, the changes were not dramatic.

- **(Without residual connection)** The highest test accuracy (29%) is achieved with **image_classification_droput2.ipynb**, and:
  - **Dropout rate:** 0.3
  - **Batch Size:** 16
  - **Learning Rate:** 0.001
  - **Validation Accuracy:** 38%
  - **Test Accuracy:** 29%
- **(With a residual connection)** The highest test accuracy (33%) is achieved with **image_classification_7_2dropout.ipynb**, and:
  - **Dropout rate:** 0.3
  - **Batch size:** 32
  - **Learning Rate:** 0.001
  - **Validation Accuracy:** 32%
  - **Test Accuracy:** 37%

In the second part:

Fine-tuning layers in the pre-trained model led to better performance.

- The model with only the FC layer fine-tuned had a validation accuracy of **47%** and a test accuracy of **51%**.
- The model with the last two convolutional layers and the FC layer fine-tuned had a validation accuracy of **59%** and a test accuracy of **55%**.

Observing these results, it can be concluded that fine-tuning more layers in the pre-trained model can lead to better performance. However, tuning the hyperparameters such as learning rate, batch size, and dropout rate is also essential for achieving better results.

Additionally, we can say that models with residual connections generally show slightly better performance compared to models without residual connections.