Yr_2_DL_CV_Coursework

May 25, 2023

1 Image Based Deep Learning Techniques Coursework

1.1 Question 1

###**Breakdown of the code:**

1. Import the following libraries: **numpy, urllib, tarfile, pickle, matplotlib.pyplot**.
2. Using the specified URL, download and extract the CIFAR-10 dataset. The photos in the dataset are separated into training and test sets.
3. Create a method called 'load_data' that will load the CIFAR-10 dataset from the extracted files. The function concatenates data and labels from numerous batch files into a single dataset.
4. Create a function called **'preprocess_data'** to handle the loaded data. The function transforms the labels to one-hot encoded vectors after normalising the picture data between 0 and 1.
5. Define the activation functions "relu" and "sigmoid."
6. Define the **'forward_propagation'** function. To calculate the neural network's output, it executes matrix multiplications and uses activation functions.
7. Using cross-entropy loss, define the loss function **'compute_loss'**.
8. Define the **'backward_propagation'** backward propagation function. Using the output, labels, and intermediate activations, it computes the gradients of the weights and biases.
9. Create the training function **'train_model'**, which will be used to train the neural network using stochastic gradient descent (SGD) with learning rate decay. It sets up the weights and biases, conducts forward and backward propagation for each sample, changes the parameters, and monitors the loss and accuracy over time.
10. Use the **'load_data'** method to load the CIFAR-10 dataset.
11. Use the **'preprocess_data'** function to preprocess the CIFAR-10 dataset.
12. Flatten the picture data to create a 1D array for each image.
13. Use the **'train_model'** function to train the model with the supplied number of epochs, learning rate, step size, and decay rate.

The code prints the loss and accuracy every ten epochs and plots the loss and accuracy over epochs with matplotlib.

###**Explanation of the Math:**

1. **'relu(x)'**: The activation function of the ReLU (Rectified Linear Unit) is specified as 'np.maximum(0, x)'. It accepts 'x' as an input and returns the element-wise maximum of 'x' and 0. In other words, it substitutes negative numbers with 0 while leaving positive ones alone. This adds nonlinearity to the network and aids in the detection of complicated patterns

in data.

2. **sigmoid activation** function is defined as 1.0/(1.0 + np.exp(-x)). It takes an x as an input and applies the sigmoid function on it. The sigmoid function transfers any real integer to the range (0, 1), which can then be interpreted as a probability. It is often employed as an activation function in the output layer of binary classification problems, where it can reflect the probability of a sample belonging to a certain class.

3. **Forward propagation**: Given the input data and the current set of weights and biases, the 'forward_propagation' function computes the output of a neural network.

   - 'z2 = np.dot(W1, X) + b1': This line computes the weighted sum of the inputs 'X' by multiplying them by the first layer's weights 'W1' and then adding the biases 'b1'.
   - 'a2 = relu(z2)': This line introduces non-linearity by applying the ReLU activation function to the intermediate variables 'z2'.
   - 'z3 = np.dot(W2, a2) + b2': This line computes the weighted sum of the activations 'a2' by multiplying them by the second layer's weights 'W2' and then adding the biases 'b2'.
   - 'output = sigmoid(z3)': This line applies the sigmoid activation function to the network's final values 'z3', resulting in the network's output.

4. **Loss function**: The 'compute_loss' function computes the difference between the predicted and true labels. The binary cross-entropy loss formula is used.

   - 'epsilon = 1e-8': A minor number is supplied to prevent numerical instability while calulating the logarithm.
   - 'loss = -np.mean(y * np.log(output + epsilon) + (1 - y) * np.log(1 - output + epsilon))': The loss is calculated using the formula 'y * log(output) - (1 - y) * log(1 - output)'. It compares anticipated 'output' to genuine labels 'y' and penalises significant inaccuracies.

5. **Backpropagation**: By propagating the loss backward through the network, the 'backward_propagation' function determines the gradients of the weights and biases.

   - 'delta3 = output - y': The error at the output layer is calculated by subtracting the true labels 'y' from the anticipated output.
   - 'delta2 = np.dot(W2.T, delta3) * (a2 > 0)': The error at the hidden layer is computed by backpropagating the error 'delta3' through the weights 'W2' and the ReLU activation function.
   - 'dW2 = np.dot(delta3, a2.T)': This line computes the gradient of the weights 'W2' by taking the outer product of 'delta3' and 'a2'.
   - 'db2 = np.sum(delta3, axis=1, keepdims=True)': This line computes the gradient of the bias 'b2' by adding the error 'delta3' along the axis 1.
   - 'dW1 = np.dot(delta2, X.T)': This line computes the gradient of weight 'W1' by performing the outer product of 'delta2' and the input 'X'.
   - 'db1 = np.sum(delta2, axis=1, keepdims=True)': This line computes the gradient of bias 'b1' by summing the error 'delta2' along axis 1.

6. **Stochastic Gradient Descent (SGD) with learning rate decay**: The 'train_model' function implemets the SGD method with learning rate decay.

   - 'learning_rate = *1.0 / (1.0 + decay_rate* epoch)': Decreases the learning rate by multiplying it by the reciprocal of '(1 + decay_rate * epoch). The learning rate reduces as the epoch number increases.

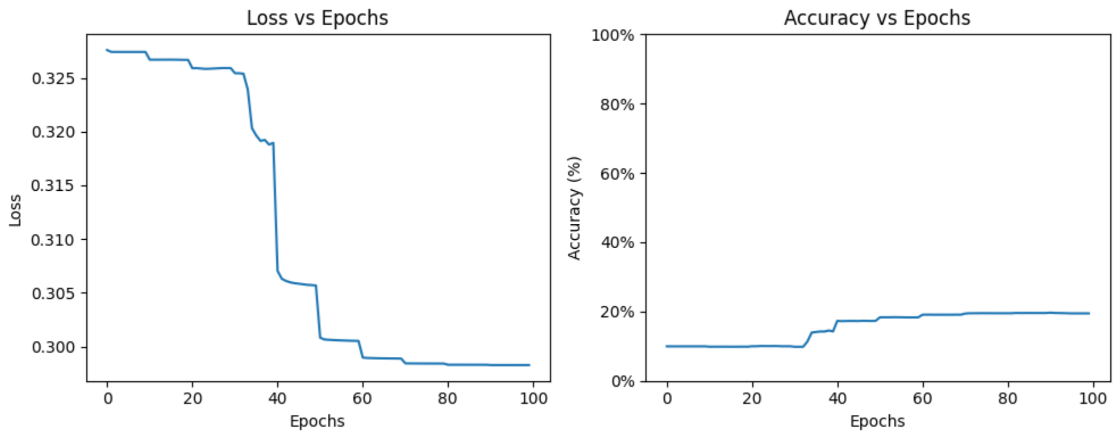
- 'W1 = W1 - learning_rate * dW1', 'b1 = b1 - learning_rate * db1', 'W2 = W2 - learning_rate * dW2', 'b2 = b2 - learning_rate * db2': These lines update the weights and biasees by multiplying them by the learning rate, and then backpropogation is used to compute gradients.

[ ]:
```
import numpy as np
import urllib.request
import tarfile
import pickle
import matplotlib.pyplot as plt

# Download and extract CIFAR-10 dataset
url = "https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
filename = "cifar-10-python.tar.gz"
urllib.request.urlretrieve(url, filename)
tar = tarfile.open(filename, "r:gz")
tar.extractall()
tar.close()

# Load CIFAR-10 dataset
def load_data():
    def unpickle(file):
        with open(file, 'rb') as fo:
            dict = pickle.load(fo, encoding='bytes')
        return dict

    data = []
    labels = []
    for i in range(1, 6):
        batch = unpickle(f'cifar-10-batches-py/data_batch_{i}')
        data.append(batch[b'data'])
        labels.extend(batch[b'labels'])
    test_batch = unpickle('cifar-10-batches-py/test_batch')
    data.append(test_batch[b'data'])
    labels.extend(test_batch[b'labels'])
    data = np.concatenate(data, axis=0).reshape(-1, 3, 32, 32).transpose(0, 2,␣
↪3, 1)
    labels = np.array(labels)
    return data, labels

# Preprocess CIFAR-10 dataset
def preprocess_data(data, labels):
    data = data.astype(np.float32) / 255.0
    num_classes = np.max(labels) + 1
    labels = np.eye(num_classes)[labels]
    return data, labels
```

```
# ReLU activation function
def relu(x):
    return np.maximum(0, x)

# Sigmoid activation function
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

# Forward propagation
def forward_propagation(X, W1, b1, W2, b2):
    z2 = np.dot(W1, X) + b1
    a2 = relu(z2)
    z3 = np.dot(W2, a2) + b2
    output = sigmoid(z3)
    return a2, output

# Loss function
def compute_loss(output, y):
    epsilon = 1e-8
    loss = -np.mean(y * np.log(output + epsilon) + (1 - y) * np.log(1 - output␣
 ↪+ epsilon))
    return loss

# Backpropagation
def backward_propagation(X, y, output, a2, W2):
    epsilon = 1e-8
    delta3 = output - y
    delta2 = np.dot(W2.T, delta3) * (a2 > 0)
    dW2 = np.dot(delta3, a2.T)
    db2 = np.sum(delta3, axis=1, keepdims=True)
    dW1 = np.dot(delta2, X.T)
    db1 = np.sum(delta2, axis=1, keepdims=True)
    return dW1, db1, dW2, db2


# Stochastic Gradient Descent (SGD) with learning rate decay
def train_model(X, y, num_epochs, learning_rate, step_size, decay_rate):
    np.random.seed(0)
    num_samples = X.shape[0]
    num_features = X.shape[1]
    num_hidden = 10
    num_classes = y.shape[1]

    # Initialize weights and biases
    W1 = np.random.randn(num_hidden, num_features) * 0.01
    b1 = np.zeros((num_hidden, 1))
    W2 = np.random.randn(num_classes, num_hidden) * 0.01
    b2 = np.zeros((num_classes, 1))

    losses = []
    accuracies = []

    for epoch in range(num_epochs):
        epoch_loss = 0
        correct_predictions = 0

        for i in range(num_samples):
            # Forward propagation
            a2, output = forward_propagation(X[i].reshape(num_features, 1), W1,␣
↪b1, W2, b2)

            # Compute loss
            loss = compute_loss(output, y[i].reshape(num_classes, 1))
            epoch_loss += loss

            # Backpropagation
            dW1, db1, dW2, db2 = backward_propagation(X[i].
↪reshape(num_features, 1), y[i].reshape(num_classes, 1), output, a2, W2)

            # Update weights and biases
            W1 = W1 - learning_rate * dW1
            b1 = b1 - learning_rate * db1
            W2 = W2 - learning_rate * dW2
            b2 = b2 - learning_rate * db2

            # Count correct predictions
            if np.argmax(output) == np.argmax(y[i]):
                correct_predictions += 1

        # Decay learning rate
        if (epoch + 1) % step_size == 0:
            learning_rate *= 1.0 / (1.0 + decay_rate * epoch)

        # Compute accuracy
        accuracy = correct_predictions / num_samples

        # Save loss and accuracy for plotting
        losses.append(epoch_loss / num_samples)
        accuracies.append(accuracy)

        # Print loss and accuracy every 10 epochs
        if (epoch + 1) % 10 == 0:
            print(f"Epoch: {epoch + 1}, Loss: {losses[-1]}, Accuracy:␣
↪{accuracies[-1]}")

    # Plot loss and accuracy
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.plot(losses)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss vs Epochs')
    plt.subplot(1, 2, 2)
    plt.plot(accuracies)
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')  # Update ylabel to indicate percentage
    plt.title('Accuracy vs Epochs')
    plt.ylim([0, 1])  # Set y-axis limits to show percentages (0% to 100%)
    plt.gca().set_yticklabels(['{:.0f}%'.format(x * 100) for x in plt.gca().
↪get_yticks()])  # Format y-axis tick labels as percentages
    plt.tight_layout()
    plt.show()

# Load CIFAR-10 dataset
data, labels = load_data()

# Preprocess CIFAR-10 dataset
data, labels = preprocess_data(data, labels)

# Flatten image data
data = data.reshape(data.shape[0], -1)

# Train the model
num_epochs = 100
learning_rate = 0.1
step_size = 10  # Specify the step size here
decay_rate = 0.05  # Specify the decay rate here
train_model(data, labels, num_epochs, learning_rate, step_size, decay_rate)
```

```
Epoch: 10, Loss: 0.3274020815054513, Accuracy: 0.09951666666666667
Epoch: 20, Loss: 0.3266673325849233, Accuracy: 0.09851666666666667
Epoch: 30, Loss: 0.32590233875021596, Accuracy: 0.09978333333333333
Epoch: 40, Loss: 0.3189476549465496, Accuracy: 0.14266666666666666
Epoch: 50, Loss: 0.30567601869215444, Accuracy: 0.17306666666666667
Epoch: 60, Loss: 0.300518077317953, Accuracy: 0.1832
Epoch: 70, Loss: 0.29888021280768406, Accuracy: 0.19086666666666666
Epoch: 80, Loss: 0.2984130758040285, Accuracy: 0.19498333333333334
Epoch: 90, Loss: 0.29830009007644864, Accuracy: 0.19576666666666667
Epoch: 100, Loss: 0.29827202597736713, Accuracy: 0.19458333333333333
```

```
<ipython-input-1-d010462e545c>:146: UserWarning: FixedFormatter should only be
used together with FixedLocator
  plt.gca().set_yticklabels(['{:.0f}%'.format(x * 100) for x in
plt.gca().get_yticks()])  # Format y-axis tick labels as percentages
```



1.1.1 After hyperparameter tuning

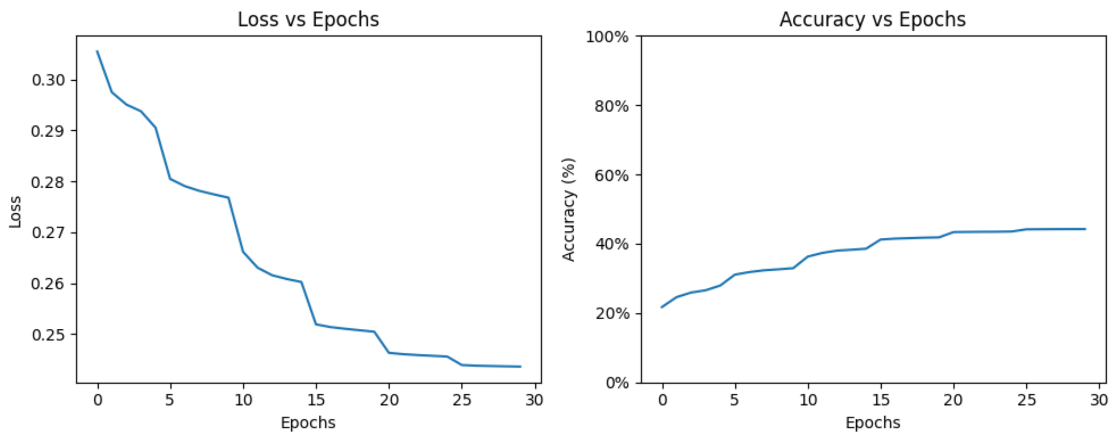
I experimented with the hyperparameters (step size, gamma, and learning rate), and after around 15 training attempts with different hyperparameters I managed to more than double the accuracy from 20% to 44%:

[ ]:
```python
import numpy as np
import urllib.request
import tarfile
import pickle
import matplotlib.pyplot as plt

# Download and extract CIFAR-10 dataset
url = "https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"
filename = "cifar-10-python.tar.gz"
urllib.request.urlretrieve(url, filename)
tar = tarfile.open(filename, "r:gz")
tar.extractall()
tar.close()

# Load CIFAR-10 dataset
def load_data():
    def unpickle(file):
        with open(file, 'rb') as fo:
            dict = pickle.load(fo, encoding='bytes')
        return dict
    data = []
    labels = []
    for i in range(1, 6):
        batch = unpickle(f'cifar-10-batches-py/data_batch_{i}')
        data.append(batch[b'data'])
        labels.extend(batch[b'labels'])
    test_batch = unpickle('cifar-10-batches-py/test_batch')
    data.append(test_batch[b'data'])
    labels.extend(test_batch[b'labels'])
    data = np.concatenate(data, axis=0).reshape(-1, 3, 32, 32).transpose(0, 2,␣
 ↪3, 1)
    labels = np.array(labels)
    return data, labels

# Preprocess CIFAR-10 dataset
def preprocess_data(data, labels):
    data = data.astype(np.float32) / 255.0
    num_classes = np.max(labels) + 1
    labels = np.eye(num_classes)[labels]
    return data, labels

# ReLU activation function
def relu(x):
    return np.maximum(0, x)

# Sigmoid activation function
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

# Forward propagation
def forward_propagation(X, W1, b1, W2, b2):
    z2 = np.dot(W1, X) + b1
    a2 = relu(z2)
    z3 = np.dot(W2, a2) + b2
    output = sigmoid(z3)
    return a2, output

# Loss function
def compute_loss(output, y):
    epsilon = 1e-8
    loss = -np.mean(y * np.log(output + epsilon) + (1 - y) * np.log(1 - output␣
 ↪+ epsilon))
    return loss

# Backpropagation
def backward_propagation(X, y, output, a2, W2):
    epsilon = 1e-8
    delta3 = output - y
    delta2 = np.dot(W2.T, delta3) * (a2 > 0)
    dW2 = np.dot(delta3, a2.T)
    db2 = np.sum(delta3, axis=1, keepdims=True)
    dW1 = np.dot(delta2, X.T)
    db1 = np.sum(delta2, axis=1, keepdims=True)
    return dW1, db1, dW2, db2


# Stochastic Gradient Descent (SGD) with learning rate decay
def train_model(X, y, num_epochs, learning_rate, step_size, decay_rate):
    np.random.seed(0)
    num_samples = X.shape[0]
    num_features = X.shape[1]
    num_hidden = 10
    num_classes = y.shape[1]

    # Initialize weights and biases
    W1 = np.random.randn(num_hidden, num_features) * 0.01
    b1 = np.zeros((num_hidden, 1))
    W2 = np.random.randn(num_classes, num_hidden) * 0.01
    b2 = np.zeros((num_classes, 1))

    losses = []
    accuracies = []

    for epoch in range(num_epochs):
        epoch_loss = 0
        correct_predictions = 0

        for i in range(num_samples):
            # Forward propagation
            a2, output = forward_propagation(X[i].reshape(num_features, 1), W1,␣
 ↪b1, W2, b2)

            # Compute loss
            loss = compute_loss(output, y[i].reshape(num_classes, 1))
            epoch_loss += loss

            # Backpropagation
            dW1, db1, dW2, db2 = backward_propagation(X[i].
 ↪reshape(num_features, 1), y[i].reshape(num_classes, 1), output, a2, W2)

            # Update weights and biases
            W1 = W1 - learning_rate * dW1
            b1 = b1 - learning_rate * db1
            W2 = W2 - learning_rate * dW2
            b2 = b2 - learning_rate * db2

            # Count correct predictions
            if np.argmax(output) == np.argmax(y[i]):
                correct_predictions += 1

        # Decay learning rate
        if (epoch + 1) % step_size == 0:
            learning_rate *= 1.0 / (1.0 + decay_rate * epoch)

        # Compute accuracy
        accuracy = correct_predictions / num_samples

        # Save loss and accuracy for plotting
        losses.append(epoch_loss / num_samples)
        accuracies.append(accuracy)

        # Print loss and accuracy every 10 epochs
        if (epoch + 1) % 5 == 0:
            print(f"Epoch: {epoch + 1}, Loss: {losses[-1]}, Accuracy:␣
↪{accuracies[-1]}")

    # Plot loss and accuracy
    plt.figure(figsize=(10, 4))
    plt.subplot(1, 2, 1)
    plt.plot(losses)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.title('Loss vs Epochs')
    plt.subplot(1, 2, 2)
    plt.plot(accuracies)
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy (%)')
    plt.title('Accuracy vs Epochs')
    plt.ylim([0, 1])  # Set y-axis limits to show percentages 0% to 100%
    plt.gca().set_yticklabels(['{:.0f}%'.format(x * 100) for x in plt.gca().
↪get_yticks()])  # Format y-axis tick labels as percentages
    plt.tight_layout()
    plt.show()

# Load CIFAR-10 dataset
data, labels = load_data()

# Preprocess CIFAR-10 dataset
data, labels = preprocess_data(data, labels)

# Flatten image data
```

```
data = data.reshape(data.shape[0], -1)

# Train the model
num_epochs = 30
learning_rate = 0.01
step_size = 5  # Specify the step size here
decay_rate = 0.1  # Specify the decay rate here
train_model(data, labels, num_epochs, learning_rate, step_size, decay_rate)
```

```
Epoch: 5, Loss: 0.2905088315503516, Accuracy: 0.27935
Epoch: 10, Loss: 0.27677132959066236, Accuracy: 0.3293
Epoch: 15, Loss: 0.260223199891374, Accuracy: 0.38563333333333333
Epoch: 20, Loss: 0.2504696788467608, Accuracy: 0.41828333333333334
Epoch: 25, Loss: 0.2455960739251015, Accuracy: 0.4353666666666667
Epoch: 30, Loss: 0.24362293661906598, Accuracy: 0.44228333333333333
```

```
<ipython-input-2-58360238b9e0>:146: UserWarning: FixedFormatter should only be
used together with FixedLocator
  plt.gca().set_yticklabels(['{:.0f}%'.format(x * 100) for x in
plt.gca().get_yticks()])  # Format y-axis tick labels as percentages
```



1.2 Question 2

```
[ ]: import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
import torchvision
from torchvision import datasets
from torchvision import transforms
from sklearn.model_selection import train_test_split
from torch.optim import lr_scheduler
from torch.optim.lr_scheduler import StepLR
!pip install optuna
import optuna
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting optuna
  Downloading optuna-3.1.1-py3-none-any.whl (365 kB)
                           365.7/365.7

kB 8.4 MB/s eta 0:00:00
Collecting alembic>=1.5.0 (from optuna)
  Downloading alembic-1.11.1-py3-none-any.whl (224 kB)
                          224.5/224.5 kB
25.8 MB/s eta 0:00:00
Collecting cmaes>=0.9.1 (from optuna)
  Downloading cmaes-0.9.1-py3-none-any.whl (21 kB)
Collecting colorlog (from optuna)
  Downloading colorlog-6.7.0-py2.py3-none-any.whl (11 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(from optuna) (1.22.4)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from optuna) (23.1)
Requirement already satisfied: sqlalchemy>=1.3.0 in
/usr/local/lib/python3.10/dist-packages (from optuna) (2.0.10)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages
(from optuna) (4.65.0)
Requirement already satisfied: PyYAML in /usr/local/lib/python3.10/dist-packages
(from optuna) (6.0)
Collecting Mako (from alembic>=1.5.0->optuna)
  Downloading Mako-1.2.4-py3-none-any.whl (78 kB)
                           78.7/78.7 kB
9.3 MB/s eta 0:00:00
Requirement already satisfied: typing-extensions>=4 in
/usr/local/lib/python3.10/dist-packages (from alembic>=1.5.0->optuna) (4.5.0)
Requirement already satisfied: greenlet!=0.4.17 in
/usr/local/lib/python3.10/dist-packages (from sqlalchemy>=1.3.0->optuna) (2.0.2)
Requirement already satisfied: MarkupSafe>=0.9.2 in
/usr/local/lib/python3.10/dist-packages (from Mako->alembic>=1.5.0->optuna)
(2.1.2)
Installing collected packages: Mako, colorlog, cmaes, alembic, optuna
Successfully installed Mako-1.2.4 alembic-1.11.1 cmaes-0.9.1 colorlog-6.7.0
optuna-3.1.1
```

Vectorisation using the transforms library from torchvision, can also add other transformations like flipping the image and rotating it to improve robustness and generalisation abilities of the model.

```
[ ]: transforms = transforms.Compose([
         transforms.ToTensor(),
     ])
```

Download CIFAR10 from torchvision, first by downloading the training dataset by applying train=True to a variable called dataset and then apply the transforms from before, finally storing it into the dataset variable.

Then using train_test_split from sklearn we can split the training data into a train (80%) and validation dataset (20%).

Finally we download the test dataset by assinging train as false.

```
[ ]: dataset = torchvision.datasets.CIFAR10(root="./data", train=True,␣
      ↪transform=transforms, download=True)
     train_dataset, val_dataset = train_test_split(dataset, test_size=0.2,␣
      ↪random_state=42)
     test_dataset = torchvision.datasets.CIFAR10(root="./data", train=False,␣
      ↪transform=transforms, download=True)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz

100%|      | 170498071/170498071 [00:01<00:00, 85752267.10it/s]

Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

create dataloaders to load data into the neural network, set batch_size to 32 and the data gets shuffled.

```
[ ]: train_loader = torch.utils.data.DataLoader(train_dataset,
                                                batch_size=32,
                                                shuffle=True)

     val_loader = torch.utils.data.DataLoader(val_dataset,
                                                batch_size=32,
                                                shuffle=True)

     test_loader = torch.utils.data.DataLoader(test_dataset,
                                                batch_size=32,
                                                shuffle=True)
```

Within the CNN class there are two function the **init** or initialisation function that initialises the layesr of the neural network: 2 convolutional layers, one pooling layer, and two fully connected linear layers.

Then there is the forward function that defines the computation graph or the sequence of the forward propogation.

[ ]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(32 * 8 * 8, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = nn.functional.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = nn.functional.relu(x)
        x = self.pool(x)
        x = x.view(-1, 32 * 8 * 8)
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.fc2(x)
        return x
```

This code allows the model to train on NVIDIA GPU using CUDA if it available or a cpu if it is not.

The CNN class is then moved to the device (either gpu or cpu)

[ ]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else ("cpu"))

model = CNN().to(device=device)
```

The criterion is the type of loss used in the neural network, since it is a multiclass classification problem we use cross entropy loss. Then we set the otpimizer as stochastic gradient descent with a learning rate of 0.01. Then we define a scheduler that provides a step_size which sets how often (after how many epochs) the learning rate decay is applied. The gamma represents the decrease in learning rate the learning rate decay applies in this case it is 0.1 so it will decrease by 10%.

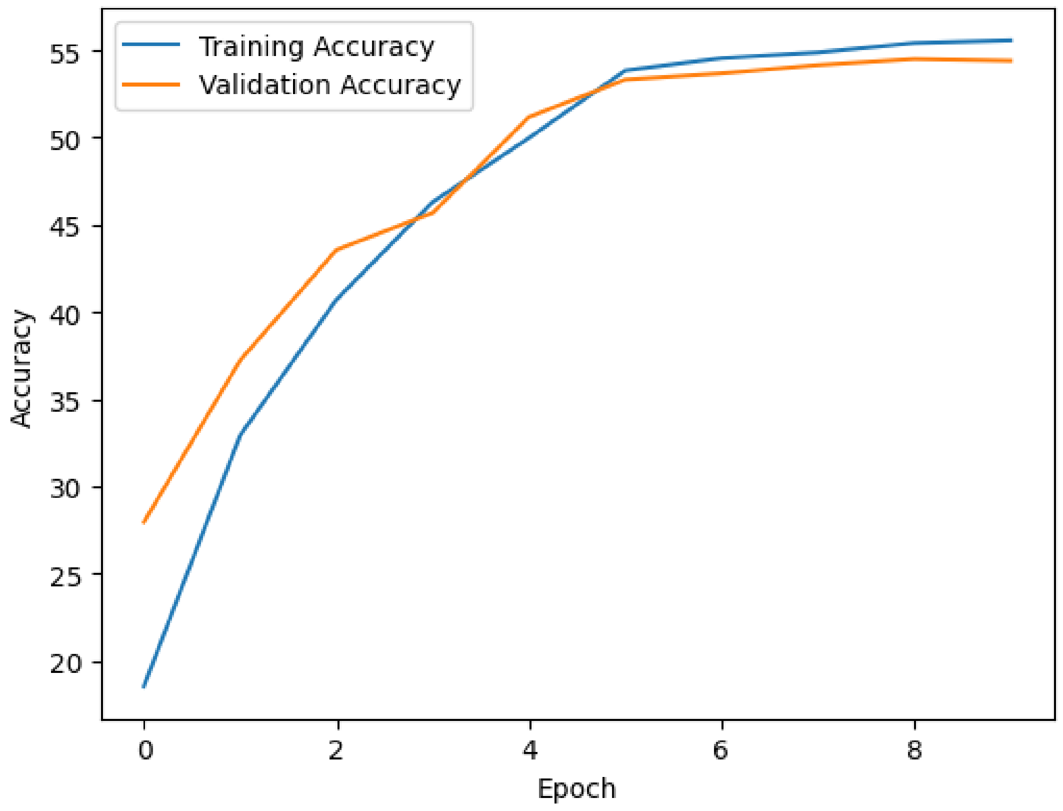
[ ]:
```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
scheduler = StepLR(optimizer, step_size=5, gamma=0.1)
```

The training process that also prints the training and validation loss alongside the graph that plots the accuracy.

[ ]:
```
# Train the model and track loss and accuracy
train_loss = []
train_acc = []
valid_loss = []
```

```python
valid_acc = []
num_epochs = 10

for epoch in range(num_epochs):
    # Train the model
    model.train()
    correct = 0
    total = 0
    for i, (inputs, labels) in enumerate(train_loader):
        inputs = inputs.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(inputs)

        # Compute loss
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Save training loss
        train_loss.append(loss.item())

        # Calculate training accuracy
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    train_acc.append(100 * correct / total)

    # Update the learning rate
    scheduler.step()

    # Evaluate the model on the validation set
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(inputs)
            # Compute loss
            loss = criterion(outputs, labels)

            # Save validation loss
            valid_loss.append(loss.item())

            # Calculate validation accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    valid_acc.append(100 * correct / total)

    # Print epoch and losses
    print('Epoch [{}/{}], Training Loss: {:.4f}, Training Acc: {:.4f}      ␣
 ↪Val Loss: {:.4f}, Validation Acc: {:.4f}'
          .format(epoch+1, num_epochs, train_loss[-1], train_acc[-1],␣
 ↪valid_loss[-1], valid_acc[-1]))

# Plot the loss and accuracy over epochs
plt.plot(train_acc, label='Training Accuracy')
plt.plot(valid_acc, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
Epoch [1/10], Training Loss: 1.9784, Training Acc: 18.5225        Val Loss:
2.0325, Validation Acc: 27.9400
Epoch [2/10], Training Loss: 1.7184, Training Acc: 32.9200        Val Loss:
1.6965, Validation Acc: 37.2200
Epoch [3/10], Training Loss: 1.6318, Training Acc: 40.7025        Val Loss:
1.4596, Validation Acc: 43.5600
Epoch [4/10], Training Loss: 1.5069, Training Acc: 46.3025        Val Loss:
1.4865, Validation Acc: 45.6800
Epoch [5/10], Training Loss: 1.1219, Training Acc: 49.9800        Val Loss:
1.3502, Validation Acc: 51.1800
Epoch [6/10], Training Loss: 1.3370, Training Acc: 53.8450        Val Loss:
1.1676, Validation Acc: 53.3200
Epoch [7/10], Training Loss: 1.4543, Training Acc: 54.5500        Val Loss:
1.6389, Validation Acc: 53.6800
Epoch [8/10], Training Loss: 0.9964, Training Acc: 54.8775        Val Loss:
1.6324, Validation Acc: 54.1500
Epoch [9/10], Training Loss: 1.4287, Training Acc: 55.4175        Val Loss:
1.4687, Validation Acc: 54.5000
Epoch [10/10], Training Loss: 1.3425, Training Acc: 55.5750        Val Loss:
1.4639, Validation Acc: 54.4000
```



###**Hyperparameter tuning**

Using an open source hyperparameter tuning library called optuna we can provide a search space for the hyperparameters, in this case the learning rate, step size, and gamma, and then it will find the optimal hyperprameters by running a set amount of trials with at first random combinations of hyperparameter values inbetween the values provided in .suggest and then slowly homing in on the hyperparameters that provide the highest accuracy.

```
[ ]: def objective(trial):
         model = CNN().to(device=device)

         criterion = nn.CrossEntropyLoss()

         # Define the hyperparameters to tune
         learning_rate = trial.suggest_loguniform('learning_rate', 0.01, 0.1)
         optimizer = optim.SGD(model.parameters(), lr=learning_rate)

         step_size = trial.suggest_int('step_size', 1, 9)
         gamma = trial.suggest_uniform('gamma', 0.01, 0.99)
         scheduler = StepLR(optimizer, step_size=step_size, gamma=gamma)

         for epoch in range(num_epochs):
```

```
        model.train()
        correct = 0
        total = 0
        for i, (inputs, labels) in enumerate(train_loader):
            inputs = inputs.to(device)
            labels = labels.to(device)

            optimizer.zero_grad()

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

            train_loss.append(loss.item())

            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_acc.append(100 * correct / total)

        scheduler.step()

        model.eval()
        correct = 0
        total = 0
        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs = inputs.to(device)
                labels = labels.to(device)

                outputs = model(inputs)
                loss = criterion(outputs, labels)

                valid_loss.append(loss.item())

                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        valid_acc.append(100 * correct / total)

    return valid_acc[-1]  # Return the last validation accuracy
```

We set n_trials in study.optimize to 50 meaning it will run 50 experiments with different hyper-parameters to find the ideal ones which will then be stored in the optuna study.

```
[ ]: device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

     # Define the lists to store the losses and accuracies
     train_loss = []
     train_acc = []
     valid_loss = []
     valid_acc = []
     num_epochs = 10

     # Create an Optuna study and optimize the objective function
     study = optuna.create_study(direction='maximize')
     study.optimize(objective, n_trials=50)

     # Print epoch and losses
     print('Epoch [{}/{}], Training Loss: {:.4f}, Training Acc: {:.4f}, Val Loss: {:.
      ↪4f}, Validation Acc: {:.4f}'
               .format(epoch + 1, num_epochs, train_loss[-1], train_acc[-1],␣
      ↪valid_loss[-1], valid_acc[-1]))
```

```
[I 2023-05-22 18:02:30,857] A new study created in memory with name:
no-name-b4811f8d-2374-4de9-b980-ad4eecf16037
<ipython-input-9-86e96d8da5df>:7: FutureWarning: suggest_loguniform has been
deprecated in v3.0.0. This feature will be removed in v6.0.0. See
https://github.com/optuna/optuna/releases/tag/v3.0.0. Use
:func:`~optuna.trial.Trial.suggest_float` instead.
  learning_rate = trial.suggest_loguniform('learning_rate', 0.01, 0.1)
<ipython-input-9-86e96d8da5df>:11: FutureWarning: suggest_uniform has been
deprecated in v3.0.0. This feature will be removed in v6.0.0. See
https://github.com/optuna/optuna/releases/tag/v3.0.0. Use
:func:`~optuna.trial.Trial.suggest_float` instead.
  gamma = trial.suggest_uniform('gamma', 0.01, 0.99)
[I 2023-05-22 18:03:00,970] Trial 0 finished with value: 68.23 and
parameters: {'learning_rate': 0.02637802560542686, 'step_size': 9, 'gamma':
0.24712315124208378}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:03:30,608] Trial 1 finished with value: 54.76 and
parameters: {'learning_rate': 0.010338735667478907, 'step_size': 2, 'gamma':
0.8061229377137479}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:03:59,339] Trial 2 finished with value: 64.67 and
parameters: {'learning_rate': 0.06216431808990478, 'step_size': 9, 'gamma':
0.8467216476904583}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:04:28,402] Trial 3 finished with value: 39.47 and
parameters: {'learning_rate': 0.020641678446490886, 'step_size': 1, 'gamma':
0.18457914851643062}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:04:57,100] Trial 4 finished with value: 45.75 and
parameters: {'learning_rate': 0.010584020317857837, 'step_size': 1, 'gamma':
0.5942431166556005}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:05:25,845] Trial 5 finished with value: 66.69 and
```

```
parameters: {'learning_rate': 0.08387310523409777, 'step_size': 6, 'gamma':
0.40629849267523976}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:05:55,095] Trial 6 finished with value: 65.75 and
parameters: {'learning_rate': 0.07678473063997548, 'step_size': 9, 'gamma':
0.8041350852644271}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:06:23,745] Trial 7 finished with value: 58.32 and
parameters: {'learning_rate': 0.0326970845395735, 'step_size': 2, 'gamma':
0.22174312903534477}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:06:52,379] Trial 8 finished with value: 66.5 and
parameters: {'learning_rate': 0.03290905953820757, 'step_size': 6, 'gamma':
0.7430560379982618}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:07:21,321] Trial 9 finished with value: 68.04 and
parameters: {'learning_rate': 0.035849659173632614, 'step_size': 8, 'gamma':
0.4411007695209298}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:07:50,777] Trial 10 finished with value: 58.78 and
parameters: {'learning_rate': 0.02059611501132636, 'step_size': 5, 'gamma':
0.02141712378881741}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:08:19,467] Trial 11 finished with value: 67.25 and
parameters: {'learning_rate': 0.045395750676216304, 'step_size': 8, 'gamma':
0.4341306679108693}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:08:48,503] Trial 12 finished with value: 67.99 and
parameters: {'learning_rate': 0.0540292617111037, 'step_size': 7, 'gamma':
0.5639490136637896}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:09:17,494] Trial 13 finished with value: 67.41 and
parameters: {'learning_rate': 0.04049295699468209, 'step_size': 4, 'gamma':
0.3295626690610961}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:09:46,096] Trial 14 finished with value: 65.61 and
parameters: {'learning_rate': 0.024135361824882434, 'step_size': 8, 'gamma':
0.9880538469599285}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:10:15,047] Trial 15 finished with value: 67.08 and
parameters: {'learning_rate': 0.02715968799415044, 'step_size': 8, 'gamma':
0.2833387082394926}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:10:44,130] Trial 16 finished with value: 63.8 and
parameters: {'learning_rate': 0.016233973802564174, 'step_size': 9, 'gamma':
0.4916931304845869}. Best is trial 0 with value: 68.23.
[I 2023-05-22 18:11:12,919] Trial 17 finished with value: 68.27 and
parameters: {'learning_rate': 0.039419807402341196, 'step_size': 7, 'gamma':
0.14288790361955045}. Best is trial 17 with value: 68.27.
[I 2023-05-22 18:11:42,160] Trial 18 finished with value: 66.38 and
parameters: {'learning_rate': 0.05081712821259726, 'step_size': 6, 'gamma':
0.05074403359602625}. Best is trial 17 with value: 68.27.
[I 2023-05-22 18:12:11,486] Trial 19 finished with value: 68.18 and
parameters: {'learning_rate': 0.06414284678128976, 'step_size': 4, 'gamma':
0.14241158533214646}. Best is trial 17 with value: 68.27.
[I 2023-05-22 18:12:40,426] Trial 20 finished with value: 68.03 and
parameters: {'learning_rate': 0.03729626962782863, 'step_size': 7, 'gamma':
0.12218620260963578}. Best is trial 17 with value: 68.27.
[I 2023-05-22 18:13:09,465] Trial 21 finished with value: 69.62 and
```

```
parameters: {'learning_rate': 0.09573671919685632, 'step_size': 4, 'gamma':
0.157894070420464}. Best is trial 21 with value: 69.62.
[I 2023-05-22 18:13:38,825] Trial 22 finished with value: 69.91 and
parameters: {'learning_rate': 0.09769826486956877, 'step_size': 4, 'gamma':
0.25889295300307336}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:14:07,724] Trial 23 finished with value: 68.54 and
parameters: {'learning_rate': 0.09816433447274901, 'step_size': 4, 'gamma':
0.10668570293161808}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:14:36,770] Trial 24 finished with value: 69.04 and
parameters: {'learning_rate': 0.0996137995668609, 'step_size': 4, 'gamma':
0.3239944239989587}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:15:05,945] Trial 25 finished with value: 69.28 and
parameters: {'learning_rate': 0.09131501583098692, 'step_size': 3, 'gamma':
0.3321264305832037}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:15:35,021] Trial 26 finished with value: 67.78 and
parameters: {'learning_rate': 0.07771571347601941, 'step_size': 3, 'gamma':
0.23262887087641543}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:16:04,633] Trial 27 finished with value: 67.53 and
parameters: {'learning_rate': 0.06778430522894435, 'step_size': 3, 'gamma':
0.31449924167435406}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:16:33,789] Trial 28 finished with value: 68.66 and
parameters: {'learning_rate': 0.08922655959782472, 'step_size': 3, 'gamma':
0.19854704407128757}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:17:02,473] Trial 29 finished with value: 68.62 and
parameters: {'learning_rate': 0.07590052210171148, 'step_size': 5, 'gamma':
0.252028430478366}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:17:31,378] Trial 30 finished with value: 67.18 and
parameters: {'learning_rate': 0.08934146882348089, 'step_size': 2, 'gamma':
0.3445225529283441}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:18:00,615] Trial 31 finished with value: 66.87 and
parameters: {'learning_rate': 0.09550970912579612, 'step_size': 5, 'gamma':
0.3555313502572203}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:18:29,398] Trial 32 finished with value: 69.26 and
parameters: {'learning_rate': 0.09780581339760144, 'step_size': 4, 'gamma':
0.26787296846980274}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:18:58,042] Trial 33 finished with value: 67.13 and
parameters: {'learning_rate': 0.06962124934030817, 'step_size': 3, 'gamma':
0.24342675763985008}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:19:27,397] Trial 34 finished with value: 68.74 and
parameters: {'learning_rate': 0.08246177366749759, 'step_size': 4, 'gamma':
0.27989383857647193}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:19:56,223] Trial 35 finished with value: 66.19 and
parameters: {'learning_rate': 0.05933234700445936, 'step_size': 3, 'gamma':
0.18889362998194742}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:20:25,417] Trial 36 finished with value: 68.95 and
parameters: {'learning_rate': 0.09957023455688073, 'step_size': 5, 'gamma':
0.09282394670727676}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:20:54,692] Trial 37 finished with value: 59.87 and
```

```
parameters: {'learning_rate': 0.07240981327824787, 'step_size': 1, 'gamma':
0.38619971580635964}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:21:23,664] Trial 38 finished with value: 65.9 and
parameters: {'learning_rate': 0.08388688399943811, 'step_size': 2, 'gamma':
0.28755308203222774}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:21:52,370] Trial 39 finished with value: 68.22 and
parameters: {'learning_rate': 0.06072104664084511, 'step_size': 4, 'gamma':
0.18729299911478692}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:22:21,814] Trial 40 finished with value: 67.38 and
parameters: {'learning_rate': 0.08471290472554313, 'step_size': 3, 'gamma':
0.1631637508213923}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:22:50,477] Trial 41 finished with value: 68.11 and
parameters: {'learning_rate': 0.0970269979500783, 'step_size': 4, 'gamma':
0.3581222274570047}. Best is trial 22 with value: 69.91.
[I 2023-05-22 18:23:19,143] Trial 42 finished with value: 70.02 and
parameters: {'learning_rate': 0.0754019314729869, 'step_size': 4, 'gamma':
0.2336911103165824}. Best is trial 42 with value: 70.02.
[I 2023-05-22 18:23:48,462] Trial 43 finished with value: 68.68 and
parameters: {'learning_rate': 0.0748116198894323, 'step_size': 5, 'gamma':
0.23027109025214}. Best is trial 42 with value: 70.02.
[I 2023-05-22 18:24:17,644] Trial 44 finished with value: 66.69 and
parameters: {'learning_rate': 0.08316868278005425, 'step_size': 2, 'gamma':
0.2594257496758295}. Best is trial 42 with value: 70.02.
[I 2023-05-22 18:24:46,434] Trial 45 finished with value: 68.96 and
parameters: {'learning_rate': 0.08925362540806929, 'step_size': 6, 'gamma':
0.2018203144508588}. Best is trial 42 with value: 70.02.
[I 2023-05-22 18:25:15,679] Trial 46 finished with value: 68.13 and
parameters: {'learning_rate': 0.06540449510659277, 'step_size': 4, 'gamma':
0.1654900779501907}. Best is trial 42 with value: 70.02.
[I 2023-05-22 18:25:44,431] Trial 47 finished with value: 68.77 and
parameters: {'learning_rate': 0.07237077761736511, 'step_size': 3, 'gamma':
0.4001449079254695}. Best is trial 42 with value: 70.02.
[I 2023-05-22 18:26:13,455] Trial 48 finished with value: 70.06 and
parameters: {'learning_rate': 0.07861810339506409, 'step_size': 5, 'gamma':
0.06876893199313328}. Best is trial 48 with value: 70.06.
[I 2023-05-22 18:26:42,772] Trial 49 finished with value: 68.58 and
parameters: {'learning_rate': 0.05905826668105741, 'step_size': 5, 'gamma':
0.07257926635170636}. Best is trial 48 with value: 70.06.
```

```
Epoch [10/10], Training Loss: 0.9766, Training Acc: 77.8250, Val Loss: 1.6081,
Validation Acc: 68.5800
```

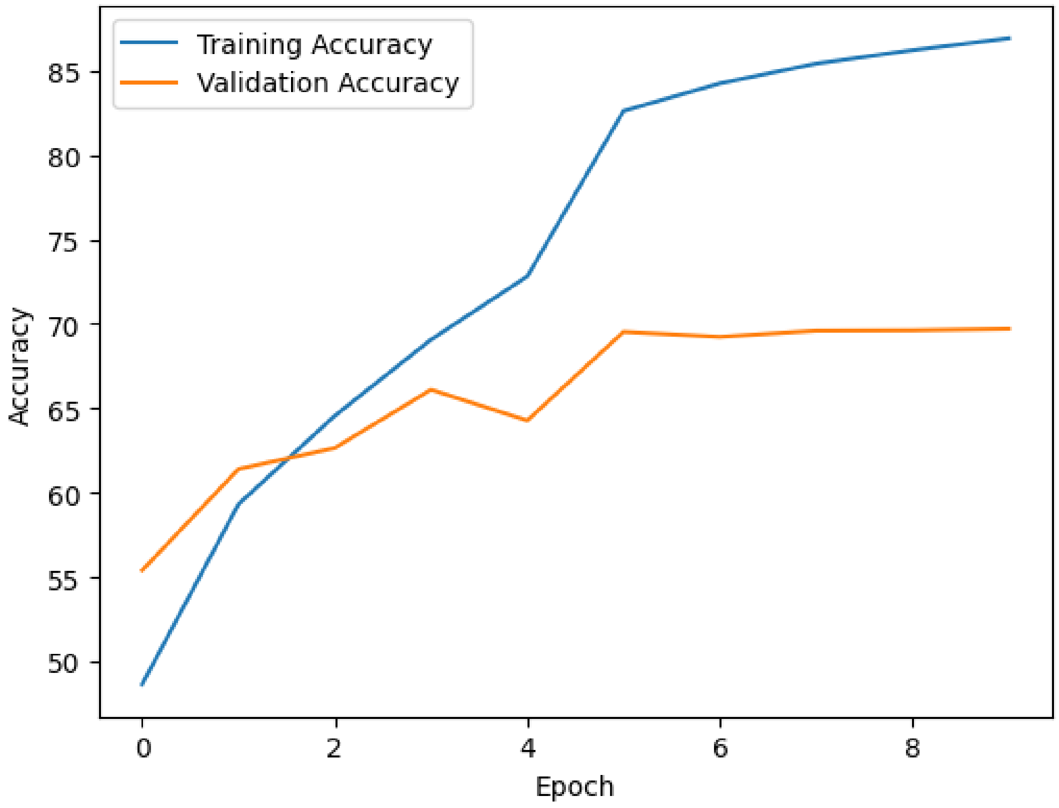
After 50 trials the hyperparameters with the highest validation accuracy is then saved and input into the model again and finally training on the test dataset which has not seen in the training process, we do not use the validation dataset as the final process of benchmarking the model because there could have been data leakage during the hyperparameter process that fits the model to specific qualities found in the validation dataset. We train it lastly on the test dataset which is unseen to ensure the model is able to generalise and is robust when met with new data.

```
[ ]: best_params = study.best_params
     learning_rate = best_params['learning_rate']
     step_size = best_params['step_size']
     gamma = best_params['gamma']

     criterion = nn.CrossEntropyLoss()
     optimizer = optim.SGD(model.parameters(), lr=learning_rate)
     scheduler = StepLR(optimizer, step_size=step_size, gamma=gamma)

     # Train the model and track loss and accuracy
     train_loss = []
     train_acc = []
     valid_loss = []
     valid_acc = []
     num_epochs = 10

     for epoch in range(num_epochs):
         # Train the model
         model.train()
         correct = 0
         total = 0
         for i, (inputs, labels) in enumerate(train_loader):
             inputs = inputs.to(device)
             labels = labels.to(device)

             # Forward pass
             outputs = model(inputs)

             # Compute loss
             loss = criterion(outputs, labels)

             # Backward pass and optimization
             optimizer.zero_grad()
             loss.backward()
             optimizer.step()

             # Save training loss
             train_loss.append(loss.item())

             # Calculate training accuracy
             _, predicted = torch.max(outputs.data, 1)
             total += labels.size(0)
             correct += (predicted == labels).sum().item()
         train_acc.append(100 * correct / total)

         # Update the learning rate
         scheduler.step()
```

```
    # Evaluate the model on the validation set
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Forward pass
            outputs = model(inputs)

            # Compute loss
            loss = criterion(outputs, labels)

            # Save validation loss
            valid_loss.append(loss.item())

            # Calculate validation accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    valid_acc.append(100 * correct / total)

    # Print epoch and losses
    print('Epoch [{}/{}], Training Loss: {:.4f}, Training Acc: {:.4f}      ␣
 ↪Val Loss: {:.4f}, Validation Acc: {:.4f}'
           .format(epoch+1, num_epochs, train_loss[-1], train_acc[-1],␣
 ↪valid_loss[-1], valid_acc[-1]))

# Plot the loss and accuracy over epochs
plt.plot(train_acc, label='Training Accuracy')
plt.plot(valid_acc, label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
Epoch [1/10], Training Loss: 1.4277, Training Acc: 48.6550        Val Loss:
1.0837, Validation Acc: 55.4000
Epoch [2/10], Training Loss: 1.2090, Training Acc: 59.3175        Val Loss:
0.9291, Validation Acc: 61.4000
Epoch [3/10], Training Loss: 0.9345, Training Acc: 64.5550        Val Loss:
1.2085, Validation Acc: 62.6500
Epoch [4/10], Training Loss: 0.6275, Training Acc: 69.0675        Val Loss:
0.9559, Validation Acc: 66.1000
```

```
Epoch [5/10], Training Loss: 0.6425, Training Acc: 72.8125        Val Loss:
1.1409, Validation Acc: 64.2800
Epoch [6/10], Training Loss: 0.7491, Training Acc: 82.6150        Val Loss:
0.7049, Validation Acc: 69.5300
Epoch [7/10], Training Loss: 0.4275, Training Acc: 84.2425        Val Loss:
0.9990, Validation Acc: 69.2300
Epoch [8/10], Training Loss: 0.4110, Training Acc: 85.4150        Val Loss:
0.6663, Validation Acc: 69.6000
Epoch [9/10], Training Loss: 0.4471, Training Acc: 86.2075        Val Loss:
1.1428, Validation Acc: 69.6200
Epoch [10/10], Training Loss: 0.4902, Training Acc: 86.9150        Val Loss:
1.2986, Validation Acc: 69.7100
```



After hyperparameter tuning the accuracy has gone from 54% to 70% which is quite a respectable increase considering the underlying architecture wasn't changed and only 10 epochs were used.