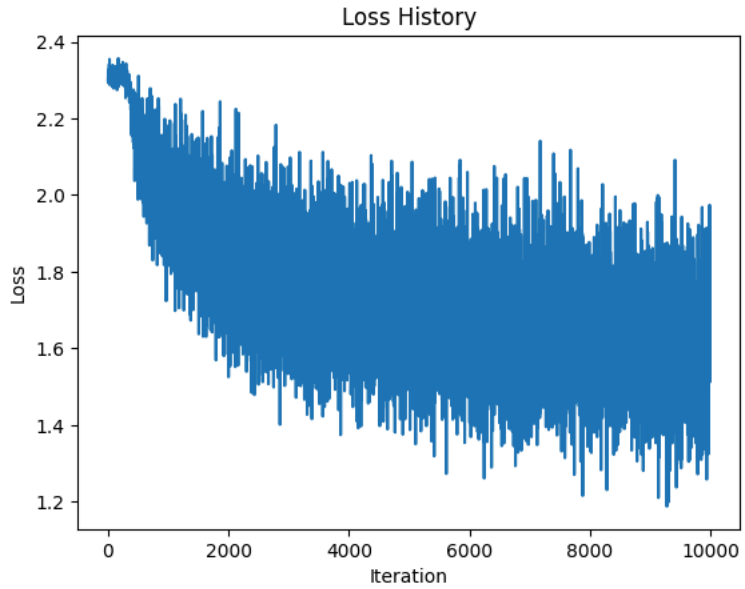
Github Link : https://github.com/kavishanGT/Image_assignment_03.git

Question 01

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

# 1. Dataloading
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
batch_size = 50
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=2)
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=2)
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

# 2. Define Network Parameters
Din = 3 * 32 * 32  # Input size (flattened CIFAR-10 image size)
Hidden = 100             # Hidden layer size
K = 10              # Output size (number of classes in CIFAR-10)
std = 1e-5

# Initialize weights and biases
w1 = torch.randn(Din, Hidden) * std  # Weights for input to hidden layer
b1 = torch.zeros(Hidden)             # Bias for hidden layer
w2 = torch.randn(Hidden, K) * std     # Weights for hidden to output layer
b2 = torch.zeros(K)              # Bias for output layer

# Hyperparameters
iterations = 10
lr = 1e-3  # Learning rate
lr_decay = 0.9  # Learning rate decay
loss_history = []

# 3. Training Loop
for epoch in range(iterations):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # Get inputs and labels
        inputs, labels = data
        Ntr = inputs.shape[0]  # Batch size
        x_train = inputs.view(Ntr, -1)  # Flatten input to (Ntr, Din)
        y_train_onehot = nn.functional.one_hot( labels , K).float() # Convert lab el s to one=h

        # Forward pass
        hidden = torch.sigmoid(x_train.mm(w1) + b1)
        y_pred = hidden.mm(w2) + b2  # Output layer activation

        # Loss calculation (Cross-Entropy Loss)
        loss = nn.functional.cross_entropy(y_pred, labels)
        loss_history.append(loss.item())
        running_loss += loss.item()

        # Backpropagation
        dy_pred = torch.softmax(y_pred, dim=1) - nn.functional.one_hot(labels, K).float()
        dw2 = hidden.t().mm(dy_pred)
        db2 = dy_pred.sum(dim=0)

        d_hidden = dy_pred.mm(w2.t()) * hidden * (1 - hidden)  # Sigmoid derivative
        dw1 = x_train.t().mm(d_hidden)
        db1 = d_hidden.sum(dim=0)

        # Parameter update
        w2 -= lr * dw2
        b2 -= lr * db2
        w1 -= lr * dw1
```

```
        b1 -= lr * db1

    # Print loss for every epoch
    print(f"Epoch {epoch + 1}/{iterations}, Loss: {running_loss / len(trainloader)}")
    lr *= lr_decay  # Learning rate decay

# 4. Plotting the Loss History
plt.plot(loss_history)
plt.title("Loss History")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.show()

# 5. Calculate Accuracy on Training Set
correct_train = 0
total_train = 0
with torch.no_grad():
    for data in trainloader:
        inputs, labels = data
        Ntr = inputs.shape[0]
        x_train = inputs.view(Ntr, -1)
        hidden = torch.sigmoid(x_train.mm(w1) + b1)
        y_train_pred = hidden.mm(w2) + b2
        predicted_train = torch.argmax(y_train_pred, dim=1)
        total_train += labels.size(0)
        correct_train += (predicted_train == labels).sum().item()

train_acc = 100 * correct_train / total_train
print(f"Training accuracy: {train_acc:.2f}%")

# 6. Calculate Accuracy on Test Set
correct_test = 0
total_test = 0
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        Nte = inputs.shape[0]
        x_test = inputs.view(Nte, -1)
        hidden = torch.sigmoid(x_test.mm(w1) + b1)
        y_test_pred = hidden.mm(w2) + b2
        predicted_test = torch.argmax(y_test_pred, dim=1)
        total_test += labels.size(0)
        correct_test += (predicted_test == labels).sum().item()

test_acc = 100 * correct_test / total_test
print(f"Test accuracy: {test_acc:.2f}%")
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%|██████████| 170M/170M [00:05<00:00, 33.6MB/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
Epoch 1/10, Loss: 2.1572108463048933
Epoch 2/10, Loss: 1.8985418719053269
Epoch 3/10, Loss: 1.8023609819412232
Epoch 4/10, Loss: 1.7482713304758073
Epoch 5/10, Loss: 1.7098904373645782
Epoch 6/10, Loss: 1.6815169230699538
Epoch 7/10, Loss: 1.6590511175394058
Epoch 8/10, Loss: 1.6400600455999375
Epoch 9/10, Loss: 1.6241114428043366
Epoch 10/10, Loss: 1.6099350160360337
```



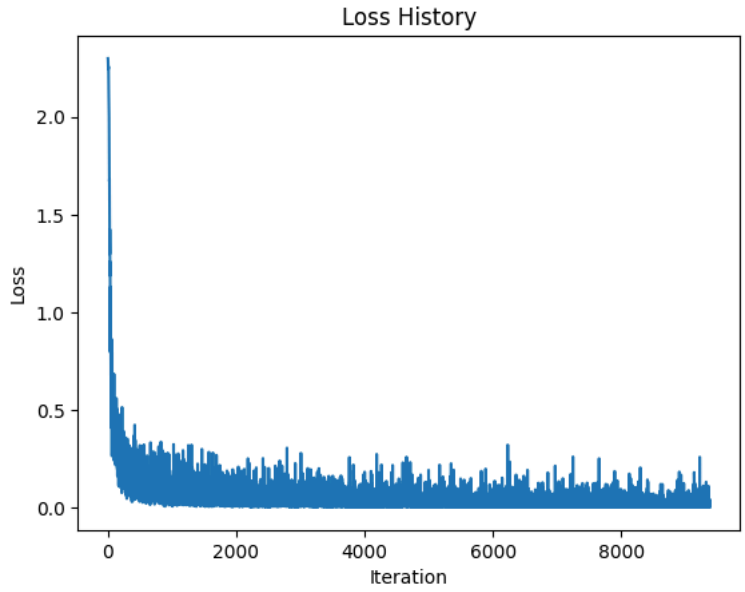

```
Training accuracy: 44.80%
Test accuracy: 43.51%
```

Question 02

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

#Initialize the LeNET5 model
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5, stride=1)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.max_pool2d(x, kernel_size=2, stride=2)
        x = torch.relu(self.conv2(x))
        x = torch.max_pool2d(x, kernel_size=2, stride=2)
        x = x.view(-1, 16 * 5 * 5)  # Flatten
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# Dataloading
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
batch_size = 64
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
```

```
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True)
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False)

#Define network parameters
model = LeNet5()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

#Train the model
epochs = 10
loss_array =[]
for epoch in range(epochs):
  running_loss = 0.0
  for inputs, labels in trainloader:
        optimizer.zero_grad()
        output = model(inputs)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        loss_array.append(loss.item())
        running_loss += loss.item()

  print(f"Epoch [{epoch + 1}/{epochs}], Loss: {running_loss / len(trainloader):.4f}")

#Calculate Training Accuracy
correct_train = 0
total_train = 0
with torch.no_grad():
    for inputs, labels in trainloader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        total_train += labels.size(0)
        correct_train += (predicted == labels).sum().item()

train_acc = 100 * correct_train / total_train
print(f"Training Accuracy: {train_acc:.2f}%")


#Calculate Test Accuracy
correct_test = 0
total_test = 0
with torch.no_grad():
    for inputs, labels in trainloader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        total_test += labels.size(0)
        correct_test += (predicted == labels).sum().item()

test_acc = 100 * correct_test / total_test
print(f"Test Accuracy: {test_acc:.2f}%")

# 4. Plotting the Loss History
plt.plot(loss_array)
plt.title("Loss History")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.show()
```

```
Epoch [1/10], Loss: 0.2461
Epoch [2/10], Loss: 0.0700
Epoch [3/10], Loss: 0.0479
Epoch [4/10], Loss: 0.0370
Epoch [5/10], Loss: 0.0324
Epoch [6/10], Loss: 0.0268
Epoch [7/10], Loss: 0.0224
Epoch [8/10], Loss: 0.0195
Epoch [9/10], Loss: 0.0177
Epoch [10/10], Loss: 0.0145
Training Accuracy: 99.73%
Test Accuracy: 99.73%
```



Question 03 - Fine Tuning approach

```
import kagglehub
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
import time
import copy
import os

# Define transformations for the training and validation sets
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

# Load the dataset
path = kagglehub.dataset_download("thedatasith/hymenoptera")

# Define paths for train and validation data
data_dir = os.path.join(path, "hymenoptera")
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x]) for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=32, shuffle=True, num_workers=4) for x in ['train', 'val']}

# Load the pre-trained ResNet18 model
model = models.resnet18(pretrained=True)
```

```
num_features = model.fc.in_features

# Modify the final layer for two classes (ants and bees)
model.fc = nn.Linear(num_features, 2)

# Move model to device (GPU if available)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Set up loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

import time
import copy

num_epochs = 25  # Specify the number of epochs

# Function to train and validate the model
def train_model(model, dataloaders, criterion, optimizer, num_epochs=25):
    start_time = time.time()
    best_model_weights = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch+1}/{num_epochs}")
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data
            for inputs, labels in dataloaders[phase]:
                inputs, labels = inputs.to(device), labels.to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()

                # Forward pass
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # Backward pass + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # Statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

            print(f"{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}")

            # Deep copy the model if we get the best accuracy on validation set
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_weights = copy.deepcopy(model.state_dict())

        print()

    time_elapsed = time.time() - start_time
    print(f"Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s")
    print(f"Best val Acc: {best_acc:.4f}")

    # Load best model weights
    model.load_state_dict(best_model_weights)
```

```
    return model

# Train and evaluate the model
model = train_model(model, dataloaders, criterion, optimizer, num_epochs=num_epochs)
```

```
Epoch 15/25
----------
train Loss: 0.0865 Acc: 0.9713
val Loss: 0.1451 Acc: 0.9542

Epoch 16/25
----------
train Loss: 0.0512 Acc: 0.9959
val Loss: 0.1444 Acc: 0.9542

Epoch 17/25
----------
train Loss: 0.1305 Acc: 0.9303
val Loss: 0.1491 Acc: 0.9412

Epoch 18/25
----------
train Loss: 0.0800 Acc: 0.9795
val Loss: 0.1472 Acc: 0.9542

Epoch 19/25
----------
train Loss: 0.0795 Acc: 0.9672
val Loss: 0.1443 Acc: 0.9542

Epoch 20/25
----------
train Loss: 0.0697 Acc: 0.9795
val Loss: 0.1493 Acc: 0.9477

Epoch 21/25
----------
train Loss: 0.0896 Acc: 0.9754
val Loss: 0.1475 Acc: 0.9477

Epoch 22/25
----------
train Loss: 0.0548 Acc: 0.9877
val Loss: 0.1529 Acc: 0.9477

Epoch 23/25
----------
train Loss: 0.0880 Acc: 0.9590
val Loss: 0.1569 Acc: 0.9477

Epoch 24/25
----------
train Loss: 0.0631 Acc: 0.9877
val Loss: 0.1552 Acc: 0.9542

Epoch 25/25
----------
train Loss: 0.0803 Acc: 0.9795
val Loss: 0.1596 Acc: 0.9542

Training complete in 22m 45s
Best val Acc: 0.9542
```

```
import kagglehub
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
import time
import copy
import os

# Define transformations for the training and validation sets
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
```

```
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
}

# Load the dataset
path = kagglehub.dataset_download("thedatasith/hymenoptera")

# Define paths for train and validation data
data_dir = os.path.join(path, "hymenoptera")
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x]) for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=32, shuffle=True, num_workers=4) for x in ['train', 'val']}

# Load the pre-trained ResNet18 model
model = models.resnet18(pretrained=True)
num_features = model.fc.in_features

import time
import copy

num_epochs = 25  # Specify the number of epochs

# Freeze all layers except the last one
for param in model.parameters():
    param.requires_grad = False

# Replace and train only the final layer for two classes (ants and bees)
model.fc = nn.Linear(num_features, 2)
model = model.to(device)

# Define the criterion and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.fc.parameters(), lr=0.001, momentum=0.9)

# Function to train and validate the model
def train_model_feature_extraction(model, dataloaders, criterion, optimizer, num_epochs=25):
    start_time = time.time()
    best_model_weights = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    for epoch in range(num_epochs):
        print(f"Epoch {epoch+1}/{num_epochs}")
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data
            for inputs, labels in dataloaders[phase]:
                inputs, labels = inputs.to(device), labels.to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()

                # Forward pass
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # Backward pass + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # Statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
```

```
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

            print(f"{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}")

            # Deep copy the model if we get the best accuracy on validation set
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_weights = copy.deepcopy(model.state_dict())

        print()

    time_elapsed = time.time() - start_time
    print(f"Training complete in {time_elapsed // 60:.0f}m {time_elapsed % 60:.0f}s")
    print(f"Best val Acc: {best_acc:.4f}")

    # Load best model weights
    model.load_state_dict(best_model_weights)
    return model

# Train and evaluate the model using feature extraction
model = train_model_feature_extraction(model, dataloaders, criterion, optimizer, num_epochs=num_epochs)
```

```
Epoch 15/25
----------
train Loss: 0.2241 Acc: 0.8975
val Loss: 0.1669 Acc: 0.9542

Epoch 16/25
----------
train Loss: 0.2104 Acc: 0.9385
val Loss: 0.1757 Acc: 0.9412

Epoch 17/25
----------
train Loss: 0.1747 Acc: 0.9385
val Loss: 0.1659 Acc: 0.9608

Epoch 18/25
----------
train Loss: 0.1945 Acc: 0.9180
val Loss: 0.1638 Acc: 0.9608

Epoch 19/25
----------
train Loss: 0.1463 Acc: 0.9590
val Loss: 0.1631 Acc: 0.9477

Epoch 20/25
----------
train Loss: 0.2071 Acc: 0.9426
val Loss: 0.1629 Acc: 0.9542
```