CSL7670 : Fundamentals of Machine Learning

Lab Report

Name:	**SOUHITYA KUNDU**
Roll Number:	**M20PH209**
Program:	**MSc-MTech(Physics & Materials Sc.)**

Chapter 1

Lab-5 and 6

1.1 Objective

The main objective of this assignment is to learn about CNN(Convolutional Neural Networks). The code is generated and based on the given question modified to a certain extent.

1.2 Problem-1

The main objective is to learn CNN. A reference code was used for the following tasks:

- First task is to understand the code fully and run it
- Then certain terms were to be explained,like, Conv2D,MaxPool2D,ReLu,Linear.
- Thirdly, generate the Loss function graph
- Fourth,optimize the code according to given requirements and compare accordingly

Solution 1(a,b,c):

```
#!/usr/bin/env python
# coding: utf-8

# # QUESTION 1:
# # (a) Understand the code completely and run it.
# # (b) Explain the following Pytorch Functions: (i) conv2D (ii) MaxPool2d (iii)
# # Linear (iv) Relu (v) linear.
# # (c) Plot the loss function.

# In[ ]:


# 1 b)
# i)Conv2D:
# The conv2D is basically a function in the Pytorch.
# It performs a 2D convolution on a given input tensor.
# So basically it takes a tensor as an input to perform the image processing tasks.
# It contains a filter which when applied on the input tensor, throws a feature map.
# This filter is applied based on the strides, and filter, given the application is possible on the tensor.
```

```
# Finally, it results in a feature map as output.


# ii)MaxPool2d:
# MaxPool2d is a function in the Pytorch which is used to reduce the
    ↪ number of layers in the given input tensor.
# From the spatial part it takes the maximum value of the given part and
    ↪ ultimately returns a concise tensor with
# reduced layers. The computational complexity is hence reduced in the CNN
    ↪ .

# iii)Linear:
# The linear activation function mainly works for the data which has
    ↪ linear type of relation amongst themselves.
# But for non-linear data it fails. The Linear activation function is f(x)
    ↪  = x basically returns just the one layer.

# iv)ReLU:
# ReLU is called as Rectified Linear Activation Function. This gives
    ↪ better performance.
# The vanishing gradient problem is solved by using ReLU. ReLU is not
    ↪ differentiable at all x, except at x = 0.
# The slope for positive value is taken as 1 and that of negatives are
    ↪ taken as 0.
# The behaviour is close to linear hence quite useful as well.


# In[1]:


# Question 1a), 1c) Done together. Full code and the plot
import torch
import torchvision
import torchvision.transforms as transforms


# In[2]:


# Transforming to tensors
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

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')


# In[3]:


import matplotlib.pyplot as plt
import numpy as np

#THE FUNCTION TO SHOW AN IMAGE


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# RANDOM TRAINING IMAGE
dataiter = iter(trainloader)
images, labels = next(dataiter)

# SHOWING THE IMAGES
imshow(torchvision.utils.make_grid(images))
# print labels
print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))


# In[4]:


import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        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 = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
```

```
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()


# In[5]:


# OPTIMIZER


# In[7]:


import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)


# # THE TRAINING

# In[24]:


import matplotlib.pyplot as plt


loss_val = []

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        loss_val.append(loss.item())
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
```

```
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')

            running_loss = 0.0


plt.plot(loss_val, label='Training Loss Plot')
plt.xlabel('Iterations')
plt.ylabel('The Loss Value')
plt.legend()
plt.show()
print('Finished Training')


# In[9]:


PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)


# In[10]:


dataiter = iter(testloader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))


# In[12]:


net = Net()
net.load_state_dict(torch.load(PATH))


# In[13]:


outputs = net(images)


# In[14]:


_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))

```

```

# # THE WHOLE DATASET IS NOW TAKEN FOR PREDICTION!!

# In[15]:


correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')


# In[16]:


# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')


# In[18]:


```

```
# # IS GPU THERE?
# device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# # Assuming that we are on a CUDA machine, this should print a CUDA
    ↪ device:

# print(device)


# In[ ]:
```

[1,	2000]	loss:	1.267
[1,	4000]	loss:	1.234
[1,	6000]	loss:	1.236
[1,	8000]	loss:	1.235
[1,	10000]	loss:	1.244
[1,	12000]	loss:	1.248
[2,	2000]	loss:	1.260
[2,	4000]	loss:	1.230
[2,	6000]	loss:	1.241
[2,	8000]	loss:	1.262
[2,	10000]	loss:	1.232
[2,	12000]	loss:	1.245

Accuracy	for	class:	plane	is	71.1	%
Accuracy	for	class:	car	is	66.3	%
Accuracy	for	class:	bird	is	56.0	%
Accuracy	for	class:	cat	is	31.8	%
Accuracy	for	class:	deer	is	44.8	%
Accuracy	for	class:	dog	is	31.9	%
Accuracy	for	class:	frog	is	66.8	%
Accuracy	for	class:	horse	is	69.1	%
Accuracy	for	class:	ship	is	41.5	%
Accuracy	for	class:	truck	is	64.5	%

Solution 1(d):

```
#!/usr/bin/env python
# coding: utf-8

# In[ ]:


# Question 1D)
import torch
import torchvision
import torchvision.transforms as transforms

# Transforming to tensors
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

batch_size = 4

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')

import matplotlib.pyplot as plt
import numpy as np

#THE FUNCTION TO SHOW AN IMAGE


def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()


# RANDOM TRAINING IMAGE
dataiter = iter(trainloader)
images, labels = next(dataiter)

# SHOWING THE IMAGES
imshow(torchvision.utils.make_grid(images))
# print labels
print('␣'.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))






# In[8]:


import torch.nn as nn
import torch.nn.functional as F


class myCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 5, 5) # changed activation map from 6 to
            ↪  5 in conv1
        self.pool = nn.AvgPool2d(2, 2)
        self.conv2 = nn.Conv2d(5, 10, 5) #INSTEAD OF 16 I have used 10 as
            ↪ asked, and input as 5,since in conv1 output was 5
        self.fc1 = nn.Linear(10 * 5 * 5, 100) # projected to 100
            ↪ dimensions
        self.fc3 = nn.Linear(100, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
```

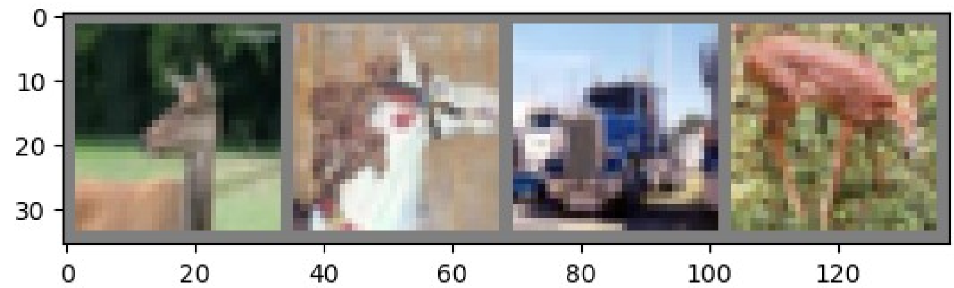
```
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = self.fc3(x)
        return x


net = myCNN()


# In[ ]:


# OPTIMIZER
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

# THE TRAINING
loss_val = []

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        loss_val.append(loss.item())
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss /
                2000:.3f}')

            running_loss = 0.0

plt.plot(loss_val,'r', label='Training Loss Plot')
plt.xlabel('Iterations')
plt.ylabel('The Loss Value')
plt.legend()
plt.show()
print('Finished Training')
```

```python

# In[11]:


PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)

dataiter = iter(testloader)
images, labels = next(dataiter)

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join(f'{classes[labels[j]]:5s}' for j in range(4)))

net = myCNN()
net.load_state_dict(torch.load(PATH))

outputs = net(images)

_, predicted = torch.max(outputs, 1)

print('Predicted: ', ' '.join(f'{classes[predicted[j]]:5s}'
                              for j in range(4)))

# THE WHOLE DATASET IS NOW TAKEN FOR PREDICTION!!

correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        # calculate outputs by running images through the network
        outputs = net(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the network on the 10000 test images: {100 * correct // total} %')

# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}

# again no gradients needed
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
```

```
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1


# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print(f'Accuracy for class: {classname:5s} is {accuracy:.1f} %')




# In[ ]:
```

[1,	2000]	loss:	1.151			
[1,	4000]	loss:	1.173			
[1,	6000]	loss:	1.185			
[1,	8000]	loss:	1.172			
[1,	10000]	loss:	1.173			
[1,	12000]	loss:	1.175			
[2,	2000]	loss:	1.115			
[2,	4000]	loss:	1.136			
[2,	6000]	loss:	1.131			
[2,	8000]	loss:	1.125			
[2,	10000]	loss:	1.108			

Accuracy	of	the	network	on	the	10000	test	images:	51	%
Accuracy	for	class:	plane	is	56.9	%				
Accuracy	for	class:	car	is	70.7	%				
Accuracy	for	class:	bird	is	29.7	%				
Accuracy	for	class:	cat	is	33.3	%				
Accuracy	for	class:	deer	is	51.2	%				
Accuracy	for	class:	dog	is	44.9	%				
Accuracy	for	class:	frog	is	64.2	%				
Accuracy	for	class:	horse	is	58.6	%				
Accuracy	for	class:	ship	is	69.3	%				
Accuracy	for	class:	truck	is	36.3	%				



deer horse truck deer

Figure 1.1: The initial random training image.

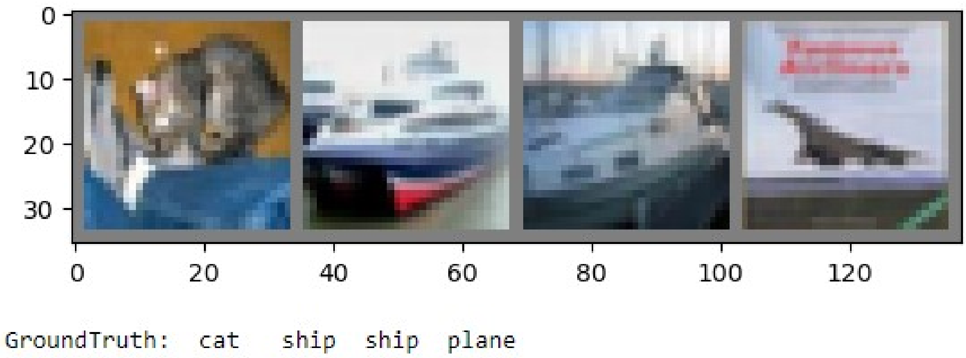


Figure 1.2: The groundtruth image.

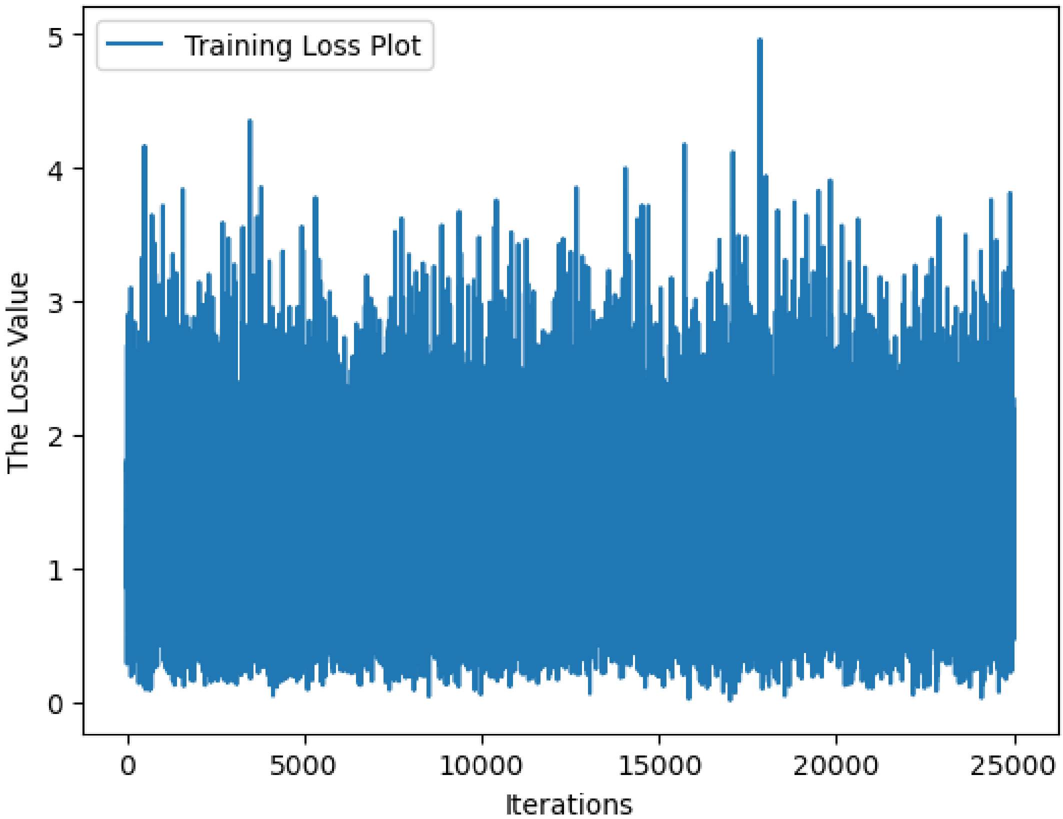


Figure 1.3: The loss function plot(question 1c).

1.3 Problem-2

- Use the given code based on CNN
- The dataset is of two category bird and horse
- Train this dataset as provided and extract the accuracy out of the code

Solution 2:

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:


import torch, torchvision
from torchvision import datasets, models, transforms
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import time
from torchsummary import summary

import numpy as np
import matplotlib.pyplot as plt
import os

from PIL import Image


# In[2]:


# pip install torchsummary


# In[3]:


# pip install torchvision


# In[2]:


# Applying Transforms to the Data
image_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(size=256, scale=(0.8, 1.0)),
        transforms.RandomRotation(degrees=15),
        transforms.RandomHorizontalFlip(),
        transforms.CenterCrop(size=224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
    ]),
    'valid': transforms.Compose([
        transforms.Resize(size=256),
        transforms.CenterCrop(size=224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406],
                             [0.229, 0.224, 0.225])
```

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


# In[3]:


# Load the Data

# Set train and valid directory paths

dataset = "C:/Users/user/FML + IMAGE_PROCESSING+AI_B/FML_Assignments/A_5/
    ↪ data/" #CALL DATA FROM LOCAL DIRECTORY

train_directory = os.path.join(dataset, 'train')
valid_directory = os.path.join(dataset, 'valid')

# Batch size
bs = 32

# Number of classes
num_classes = len(os.listdir(valid_directory))  #10#2#257
print(num_classes)

# Load Data from folders
data = {
    'train': datasets.ImageFolder(root=train_directory, transform=
        ↪ image_transforms['train']),
    'valid': datasets.ImageFolder(root=valid_directory, transform=
        ↪ image_transforms['valid'])
}

# Get a mapping of the indices to the class names, in order to see the
    ↪ output classes of the test images.
idx_to_class = {v: k for k, v in data['train'].class_to_idx.items()}
print(idx_to_class)

# Size of Data, to be used for calculating Average Loss and Accuracy
train_data_size = len(data['train'])
valid_data_size = len(data['valid'])

# Create iterators for the Data loaded using DataLoader module
train_data_loader = DataLoader(data['train'], batch_size=bs, shuffle=True)
valid_data_loader = DataLoader(data['valid'], batch_size=bs, shuffle=True)


# In[4]:
```

```
train_data_size, valid_data_size


# In[5]:


alexnet = models.alexnet(pretrained=True)
alexnet


# In[6]:


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


# In[7]:


# Change the final layer of AlexNet Model for Transfer Learning
alexnet.classifier[6] = nn.Linear(4096, num_classes)
alexnet.classifier.add_module("7", nn.LogSoftmax(dim = 1))
alexnet


# In[8]:


summary(alexnet, (3, 224, 224))


# In[9]:


# Define Optimizer and Loss Function
loss_func = nn.NLLLoss()
optimizer = optim.Adam(alexnet.parameters())
optimizer


# In[10]:


def train_and_validate(model, loss_criterion, optimizer, epochs=25):
    '''
    Function to train and validate
    Parameters
        :param model: Model to train and validate
        :param loss_criterion: Loss Criterion to minimize
        :param optimizer: Optimizer for computing gradients
```

```
        :param epochs: Number of epochs (default=25)

    Returns
        model: Trained Model with best validation accuracy
        history: (dict object): Having training loss, accuracy and validation loss, accuracy
    '''

    start = time.time()
    history = []
    best_acc = 0.0

    for epoch in range(epochs):
        epoch_start = time.time()
        print("Epoch: {}/{}".format(epoch+1, epochs))

        # Training
        model.train()

        # Loss and Accuracy for the epochs
        train_loss = 0.0
        train_acc = 0.0

        valid_loss = 0.0
        valid_acc = 0.0

        for i, (inputs, labels) in enumerate(train_data_loader):

            inputs = inputs.to(device)
            labels = labels.to(device)

            # optimizing the gradient to zero
            optimizer.zero_grad()

            # Forward pass - compute outputs on input data using the model
            outputs = model(inputs)

            # Compute the loss
            loss = loss_criterion(outputs, labels)

            #Gradient Backpropagation
            loss.backward()

            # Parameters update
            optimizer.step()

            # Compute the total loss for the batch and add it to train_loss
            train_loss += loss.item() * inputs.size(0)

            # Accuracy computation
            ret, predictions = torch.max(outputs.data, 1)
            correct_counts = predictions.eq(labels.data.view_as(predictions))
```

```

            # Convert correct_counts to float and then compute the mean
            acc = torch.mean(correct_counts.type(torch.FloatTensor))

            # Compute total accuracy in the whole batch and add to train_acc
            train_acc += acc.item() * inputs.size(0)

            #print("Batch number: {:03d}, Training: Loss: {:.4f}, Accuracy: {:.4f}".format(i, loss.item(), acc.item()))


        # Validation - No gradient tracking needed
        with torch.no_grad():

            # Set to evaluation mode
            model.eval()

            # Validation loop
            for j, (inputs, labels) in enumerate(valid_data_loader):
                inputs = inputs.to(device)
                labels = labels.to(device)

                # Forward pass - compute outputs on input data using the model
                outputs = model(inputs)

                # Compute loss
                loss = loss_criterion(outputs, labels)

                # Compute the total loss for the batch and add it to valid_loss
                valid_loss += loss.item() * inputs.size(0)

                # Calculate validation accuracy
                ret, predictions = torch.max(outputs.data, 1)
                correct_counts = predictions.eq(labels.data.view_as(predictions))

                # Convert correct_counts to float and then compute the mean
                acc = torch.mean(correct_counts.type(torch.FloatTensor))

                # Compute total accuracy in the whole batch and add to valid_acc
                valid_acc += acc.item() * inputs.size(0)

                #print("Validation Batch number: {:03d}, Validation: Loss: {:.4f}, Accuracy: {:.4f}".format(j, loss.item(), acc.item()))

        # Find average training loss and training accuracy
        avg_train_loss = train_loss/train_data_size
        avg_train_acc = train_acc/train_data_size
```

```

        # Find average training loss and training accuracy
        avg_valid_loss = valid_loss/valid_data_size
        avg_valid_acc = valid_acc/valid_data_size

        history.append([avg_train_loss, avg_valid_loss, avg_train_acc, avg_valid_acc])

        epoch_end = time.time()

        print("Epoch␣:␣{:03d},␣Training:␣Loss:␣{:.4f},␣Accuracy:␣{:.4f}%,␣\n\t\tValidation␣:␣Loss␣:␣{:.4f},␣Accuracy:␣{:.4f}%,␣Time:␣{:.4f}s".format(epoch+1, avg_train_loss, avg_train_acc*100, avg_valid_loss, avg_valid_acc*100, epoch_end-epoch_start))

        # Save if the model has best accuracy till now
        #torch.save(model, dataset+'_model_'+str(epoch)+'.pt')

    return model, history


# In[11]:


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

num_epochs = 5
trained_model, history = train_and_validate(alexnet, loss_func, optimizer, num_epochs)

torch.save(history, dataset+'_history.pt')


# In[14]:


history = np.array(history)
plt.plot(history[:,0:2])
plt.legend(['Training_Loss_Value', 'Loss_Validation_Value'])
plt.xlabel('Epoch␣Number')
plt.ylabel('Loss')
plt.ylim(0,1)
plt.savefig(dataset+'_loss_curve.png')
plt.show()


# In[15]:


plt.plot(history[:,2:4])
plt.legend(['Training_Accuracy', 'Validation_Accuracy'])
plt.xlabel('Epoch␣Number')
plt.ylabel('Accuracy')
plt.ylim(0,1)
```

```
plt.savefig(dataset+'_accuracy_curve.png')
plt.show()


# In[25]:


def prediction_model(model, test_image_name):
    '''
    This function will predict the class in which a single image belongs

    '''

    transform_save = image_transforms['test']

    test_image = Image.open(test_image_name)
    plt.imshow(test_image)

    test_image_tensor = transform_save(test_image)

    if torch.cuda.is_available():
        test_image_tensor = test_image_tensor.view(1, 3, 224, 224).cuda()
    else:
        test_image_tensor = test_image_tensor.view(1, 3, 224, 224)

    with torch.no_grad():
        model.eval()
        # Model outputs log probabilities
        out = model(test_image_tensor)
        ps = torch.exp(out)

        # Check the number of elements along dimension 1
        num_elements = ps.shape[1]

        k = min(3, num_elements)  # Setting the value of 'k'

        topk, topclass = ps.topk(k, dim=1)
        for i in range(k):
            print("The Prediction", i + 1, "is", idx_to_class[topclass.
                ↪ numpy()[0][i]], ", Score:", topk.numpy()[0][i])


# In[26]:


prediction_model(trained_model, 'C:\\Users\\user\FML + IMAGE_PROCESSING+
    ↪ AI_B\\FML_Assignments\\A_5\\data\\test\\bird\\32731.png')


# In[27]:


prediction_model(trained_model, 'C:\\Users\\user\\FML + IMAGE_PROCESSING+
    ↪ AI_B\\FML_Assignments\\A_5\\data\\test\\bird\\32793.png')
```

```


# In[28]:


prediction_model(trained_model, 'C:/Users/user/FML + IMAGE_PROCESSING+AI_B/FML_Assignments/A_5/data/test/horse/31469.png')


# In[29]:


prediction_model(trained_model, 'C:/Users/user/FML + IMAGE_PROCESSING+AI_B/FML_Assignments/A_5/data/test/horse/33845.png')


# In[ ]:
```

The Output table:

- Contains the Epoch and Accuracy values
- validation and training loss

Epoch:	1/5							
Epoch	:	001,	Training:	Loss:	0.3165,	Accuracy:	86.2798%,	
Validation	:	Loss	:	0.2043,	Accuracy:	89.5455%,	Time:	119.1857s
Epoch:	2/5							
Epoch	:	002,	Training:	Loss:	0.2572,	Accuracy:	88.6905%,	
Validation	:	Loss	:	0.2311,	Accuracy:	90.0000%,	Time:	145.6974s
Epoch:	3/5							
Epoch	:	003,	Training:	Loss:	0.2349,	Accuracy:	90.0893%,	
Validation	:	Loss	:	0.0912,	Accuracy:	97.7273%,	Time:	137.9283s
Epoch:	4/5							
Epoch	:	004,	Training:	Loss:	0.2337,	Accuracy:	90.0893%,	
Validation	:	Loss	:	0.1127,	Accuracy:	95.0000%,	Time:	154.4473s
Epoch:	5/5							
Epoch	:	005,	Training:	Loss:	0.2159,	Accuracy:	91.1310%,	
Validation	:	Loss	:	0.1606,	Accuracy:	92.7273%,	Time:	146.0598s

The Loss and Accuracy Graphs

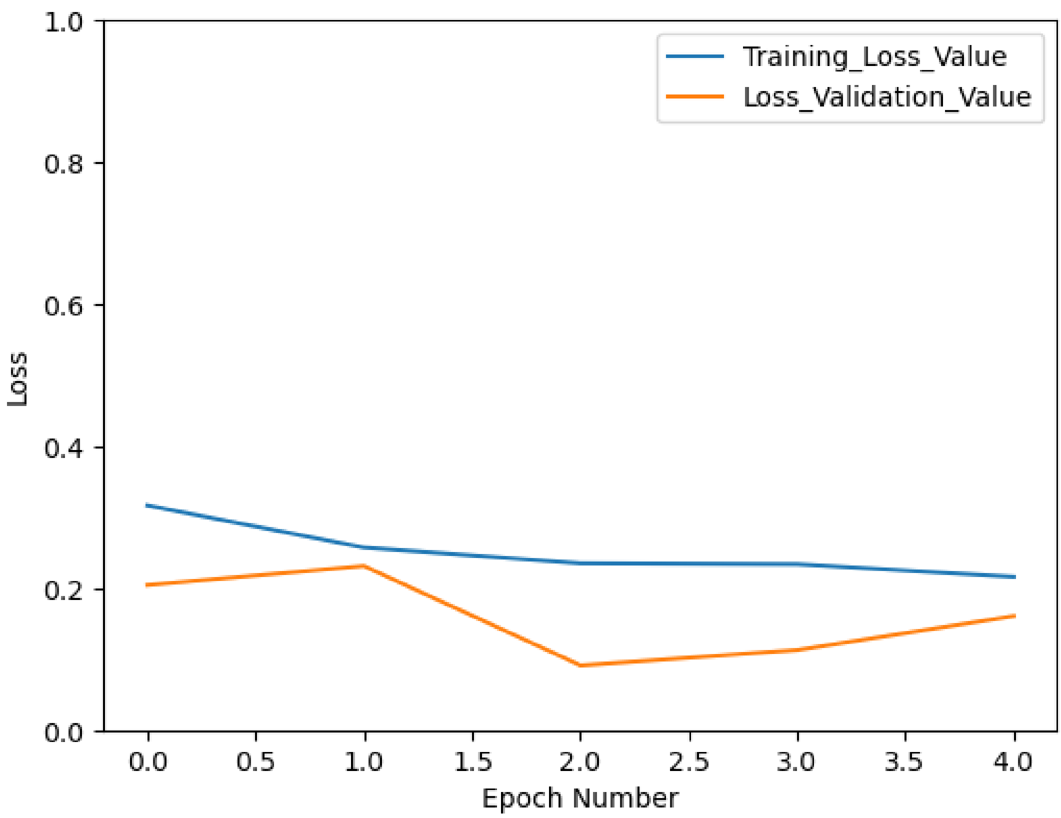


Figure 1.4: The Loss vs Epoch number.

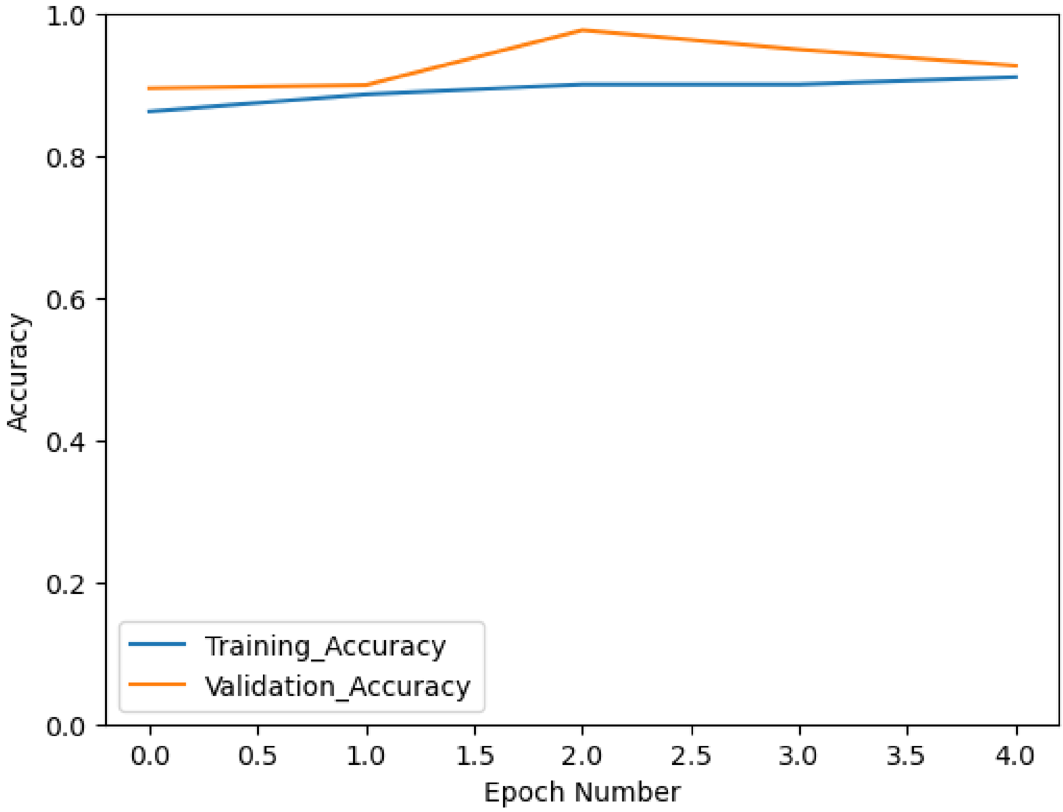


Figure 1.5: The Accuracy vs Epoch number.

The Prediction Score and Images

- Here I used the function(prediction model) to predict the score of each test data
- I have used 4 single images for both the bird and horse class and predicted the score using the model
- The model finally predicted whether the test data belongs to bird or horse category

```
The Prediction 1 is bird , Score: 0.8729614
The Prediction 2 is horse , Score: 0.12703863
```

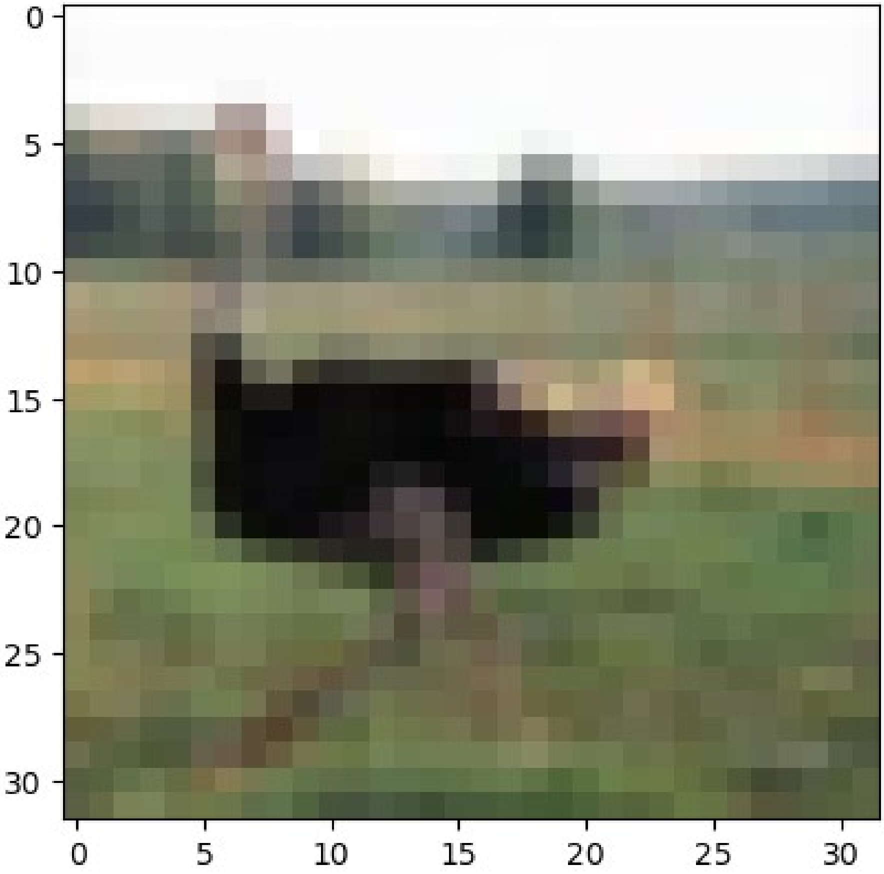


Figure 1.6: Prediction model result-1

```
The Prediction 1 is bird , Score: 0.9990458
The Prediction 2 is horse , Score: 0.00095422706
```

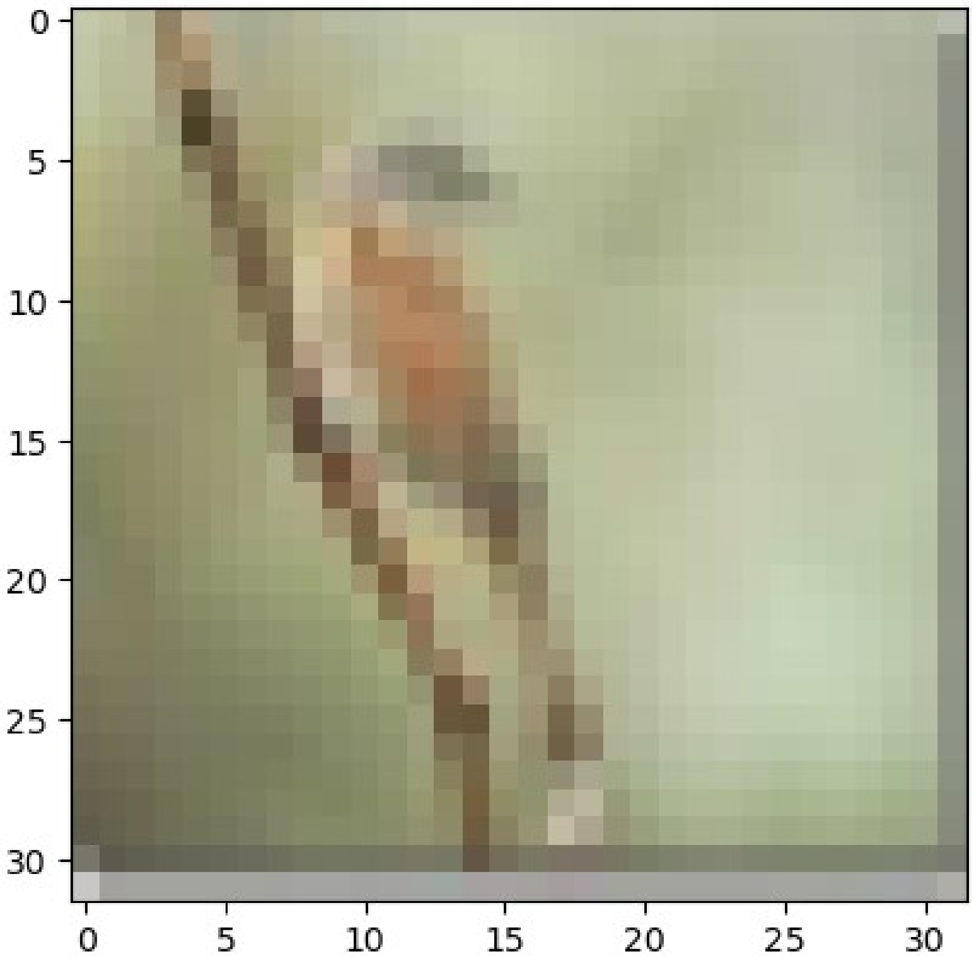


Figure 1.7: Prediction model result-2

```
The Prediction 1 is horse , Score: 0.99243915
The Prediction 2 is bird , Score: 0.0075608236
```

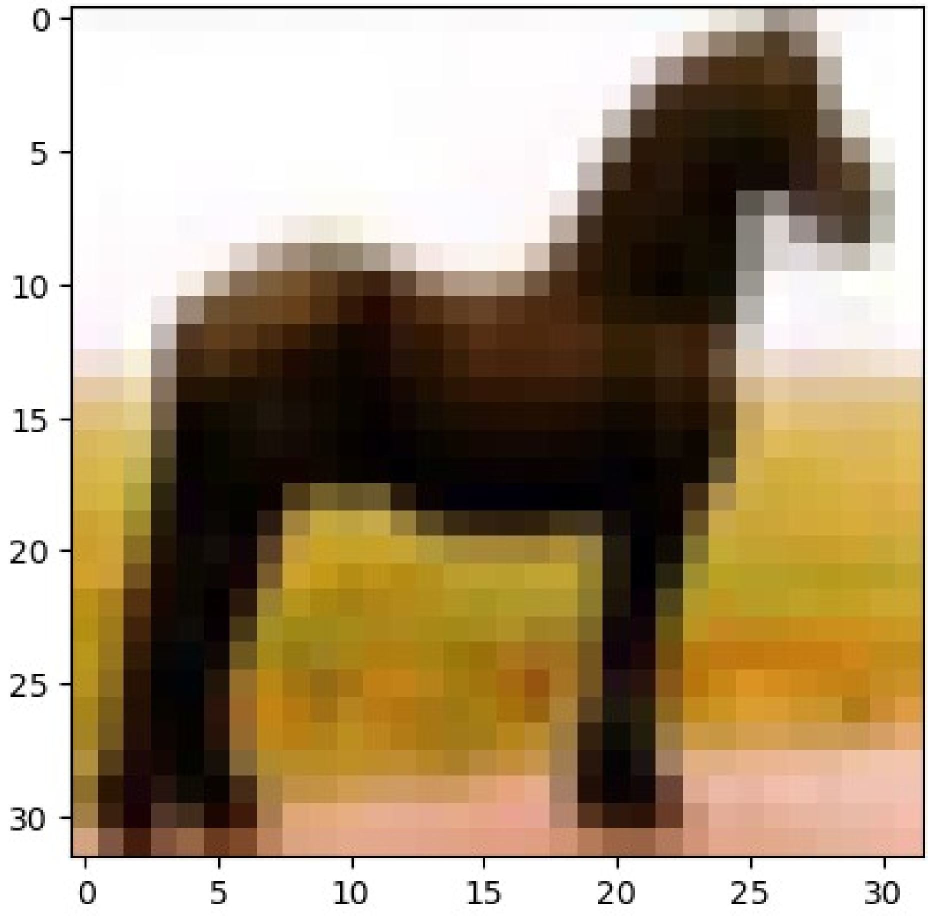


Figure 1.8: Prediction model result-3

```
The Prediction 1 is horse , Score: 0.9999434
The Prediction 2 is bird , Score: 5.659172e-05
```

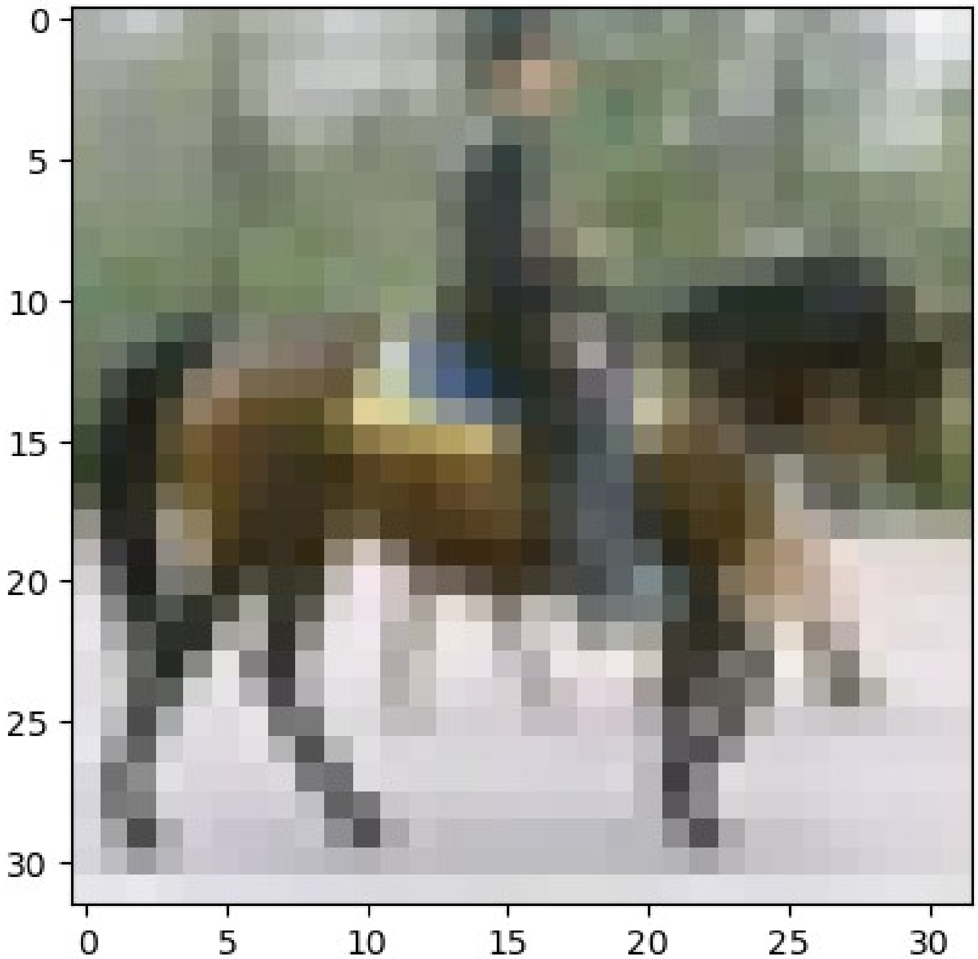


Figure 1.9: Prediction model result-4