CSL7670 : Fundamentals of Machine Learning

Lab Report

Name:	**Ganesh Kumar Nagal**
Roll Number:	**M23MEA004**
Program:	**M.Tech Adavance Manufacturing And Design**

Chapter 1

Lab-5 and 6 (CNN)

1.1 Objective

Objective of this assignment is to gain familiarity with convolutional neural networks.

1.2 Problem-1

1. (Simple CNN) Go through the following tutorial to understand how to train a CNN classifier: https://pytorch.org/tutorials/beginner/blitz/cifar10 tutorial.html Now,

1. (A) Understand the code completely and run it. perceptron,input/output/hidden layers.
2. (B) (b) Explain the following Pytorch Functions: (i) conv2D (ii) MaxPool2d (iii) Linear (iv) Relu (v) linear.
3. (C) (c) Plot the loss function.
4. (D) (d) Edit the code to modify the CNN architecture in the following four steps (call it myCNN): (i) Instead of 6 activation maps in conv1, use 5 activation maps, (ii) instead of maxpool use average pool, (iii) Instead of 16 activation maps in conv1, use 10 activation maps, and (iv) Remove fc2 and change fc1 so that it projects to 100 dimensions instead of 120 currently. Rerun the experiment and compare CNN (original code) and myCNN (this code).

Solution 1:

```
# -*- coding: utf-8 -*-
"""Untitled3.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1f-
    ↪ CdBABYVKgJKcWzd2XfX2u1p52pcEVb

1. Load and normalize CIFAR10
"""

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

```python
import matplotlib.pyplot as plt

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

# functions 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()


# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)

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

"""2. Define a Convolutional Neural Network"""

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()

"""3. Define a Loss function and optimizer"""

import torch.optim as optim

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

"""4. Train the network"""

# Lists to store training loss values
train_losses = []



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
        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}')
            train_losses.append(running_loss / 2000)
```

```
            running_loss = 0.0


print('Finished Training')

"""Save trained model"""

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

"""5. Test the network on the test data"""

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

# let us see what the neural network thinks these examples above are:
outputs = net(images)

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

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

# Let us look at how the network performs on the whole dataset.
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} %')

# the classes that performed well, and the classes that did not perform
    well:

# 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} %')



"""Explaination of following Pytorch Functions:


(i) `conv2d`:
    - `conv2d` stands for "convolutional 2D." It is a function in PyTorch used for 2D convolution operations, which are fundamental in deep learning for tasks like image processing and computer vision. Convolution involves applying a filter (also known as a kernel) to an input image to produce a feature map. The filter slides over the input, and at each position, it computes a weighted sum of the input values within its receptive field. This operation is used to extract features from the input data.

(ii) `MaxPool2d`:
    - `MaxPool2d` is short for "Max Pooling 2D." It is a pooling operation in PyTorch used primarily in convolutional neural networks (CNNs) for downsampling and reducing the spatial dimensions of feature maps. Max pooling works by dividing the input into non-overlapping regions (typically 2x2 or 3x3), and for each region, it takes the maximum value. This reduces the size of the feature maps while retaining the most important information, helping to reduce computational complexity and prevent overfitting.

(iii) `Linear`:
    - `Linear` is a PyTorch module that represents a fully connected layer or a linear transformation. In a neural network, this layer performs a linear mapping of the input data to a set of output neurons, where each output neuron is connected to every input neuron. This layer is also known as a dense layer or a fully connected layer. The linear transformation is typically followed by an activation function to introduce non-linearity into the
```

```
network.

(iv) `ReLU`:
    - `ReLU` stands for "Rectified Linear Unit." It is an activation function commonly used in neural networks. The ReLU activation function introduces non-linearity by replacing all negative values in the input with zero and leaving positive values unchanged. Mathematically, it is defined as `f(x) = max(0, x)`. ReLU helps the network learn complex patterns and is computationally efficient.

(v) `linear`:
    - `linear` is a common term used in the context of linear regression. However, in PyTorch, the term "linear" is often used to refer to the fully connected layer or linear transformation discussed in (iii) above. It represents a linear mapping from the input to the output, where each input neuron is connected to each output neuron with learned weights.

These functions are essential building blocks for creating and training neural networks in PyTorch, and they play a crucial role in various deep learning tasks, especially in tasks related to image processing and classification.
"""





"""(c) Plot the loss function."""

# Plot the training loss
plt.plot(train_losses, label='Training Loss')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title('Training Loss Over Iterations')
plt.legend()
plt.show()



"""(d) Edit the code to modify the CNN architecture in the following four steps
(call it myCNN):
"""

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

# Define the modified CNN architecture (MyCNN)
```

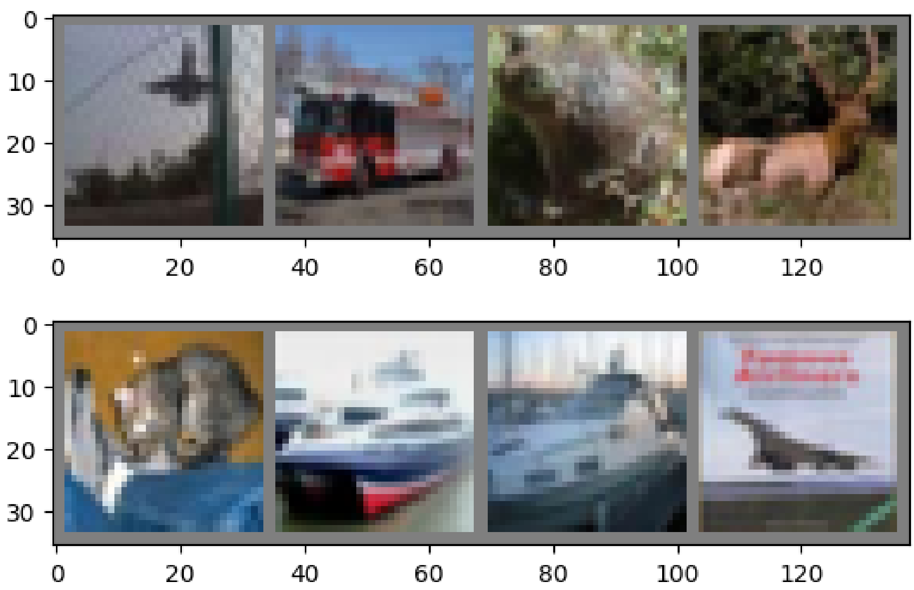
```python
class MyCNN(nn.Module):
    def __init__(self):
        super(MyCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 5, 5)  # (i) 5 activation maps instead of 6
        self.pool = nn.AvgPool2d(2, 2)  # (ii) Average pooling instead of max pooling
        self.conv2 = nn.Conv2d(5, 10, 5)  # (iii) 10 activation maps instead of 16
        self.fc1 = nn.Linear(10 * 5 * 5, 100)  # (iv) 100 dimensions instead of 120
        self.fc3 = nn.Linear(100, 10)  # Output layer

    def forward(self, x):
        x = self.pool(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = x.view(-1, 10 * 5 * 5)
        x = nn.functional.relu(self.fc1(x))
        x = self.fc3(x)
        return x

# Load CIFAR-10 dataset and create data loaders
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)

# Define the neural network (MyCNN), loss function, and optimizer
my_cnn = MyCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(my_cnn.parameters(), lr=0.001, momentum=0.9)

# Lists to store training loss values
train_losses_my_cnn = []

# Training loop for MyCNN
num_epochs = 2
for epoch in range(num_epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = my_cnn(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
```

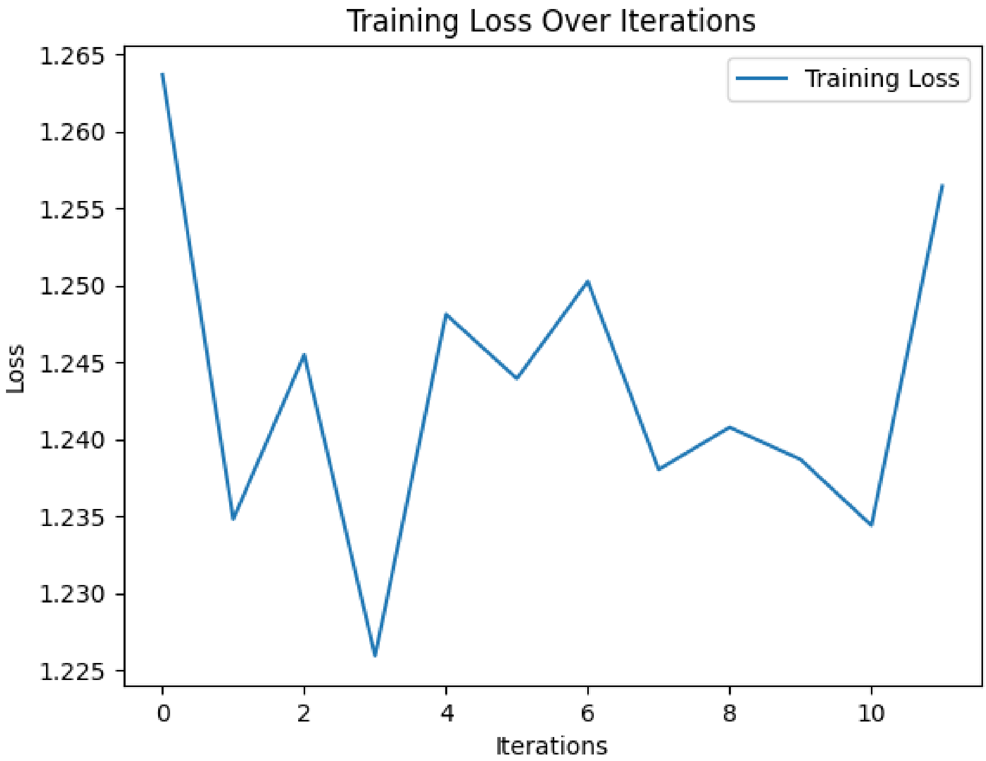
```
        if i % 2000 == 1999:  # Print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            train_losses_my_cnn.append(running_loss / 2000)
            running_loss = 0.0

print('Finished Training MyCNN')

# Plot the training loss for MyCNN
plt.plot(train_losses_my_cnn, label='MyCNN Training Loss')
plt.xlabel('Iterations (x2000)')
plt.ylabel('Loss')
plt.title('MyCNN Training Loss Over Iterations')
plt.legend()
plt.show()

"""To compare the original CNN and the modified MyCNN, we can look at several factors:

Architecture Differences:

Original CNN:
Conv1: 6 activation maps
Max pooling
Conv2: 16 activation maps
FC1: 120 dimensions
FC2: 84 dimensions
MyCNN:
Conv1: 5 activation maps
Average pooling
Conv2: 10 activation maps
FC1: 100 dimensions
FC2 removed
Training Loss:

Compare the training loss curves for both models to see how quickly they converge during training. Lower training loss indicates better convergence.
Accuracy on Test Data:

After training, evaluate both models on the test dataset and compare their  accuracy. Higher accuracy indicates better performance.
Let's add the evaluation code for the original CNN and then compare the two models:
"""

# Evaluate the original CNN on the test dataset
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
```

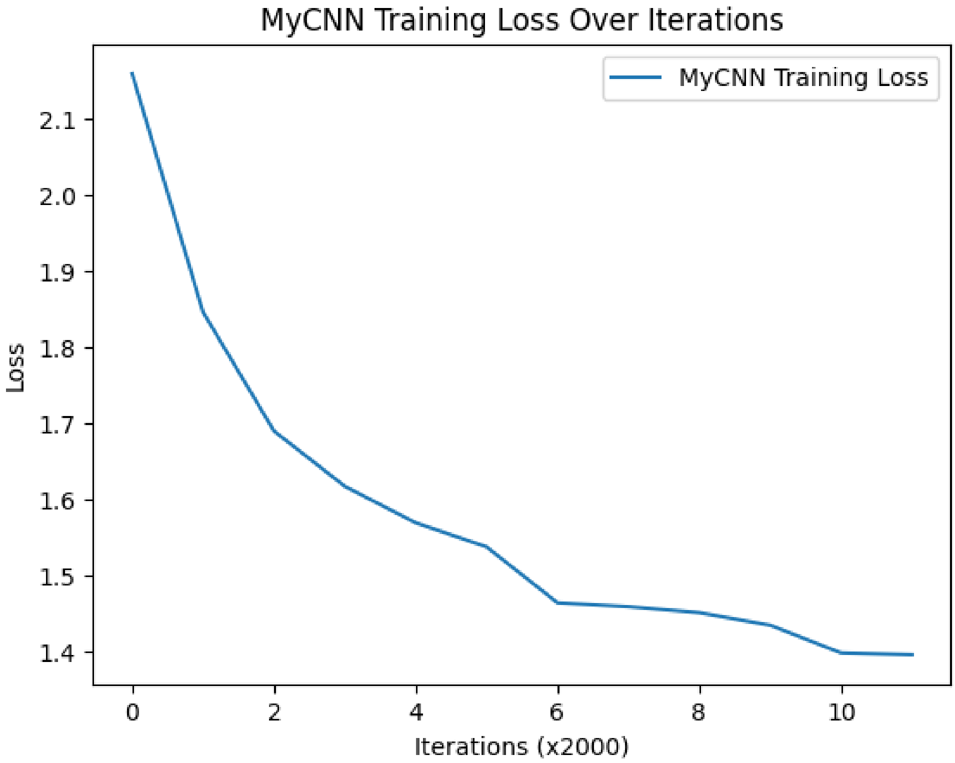
```
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the original CNN on the test images: {100 * correct / total:.2f}%')

# Evaluate MyCNN on the test dataset
correct_my_cnn = 0
total_my_cnn = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = my_cnn(images)
        _, predicted = torch.max(outputs.data, 1)
        total_my_cnn += labels.size(0)
        correct_my_cnn += (predicted == labels).sum().item()

print(f'Accuracy of MyCNN on the test images: {100 * correct_my_cnn / total_my_cnn:.2f}%')

"""Accuracy of the original CNN on the test images: 53.47%
Accuracy of MyCNN on the test images: 52.56%
We can see that accuracy is less than in MyCNN compare to Original CNN on test images.

"""
```

```
[1,  2000] loss: 1.264
[1,  4000] loss: 1.235
[1,  6000] loss: 1.245
[1,  8000] loss: 1.226
[1, 10000] loss: 1.248
[1, 12000] loss: 1.244
[2,  2000] loss: 1.250
[2,  4000] loss: 1.238
[2,  6000] loss: 1.241
[2,  8000] loss: 1.239
[2, 10000] loss: 1.234
[2, 12000] loss: 1.256
Finished Training

GroundTruth:  cat   ship  ship  plane

<All keys matched successfully>

Predicted:  cat   ship  ship  ship

Accuracy of the network on the 10000 test images: 53 %

Accuracy for class: plane is 48.0 %
Accuracy for class: car   is 78.7 %
Accuracy for class: bird  is 45.5 %
Accuracy for class: cat   is 36.7 %
Accuracy for class: deer  is 32.4 %
Accuracy for class: dog   is 54.2 %
```

```
Accuracy for class: frog  is 69.9 %
Accuracy for class: horse is 52.7 %
Accuracy for class: ship  is 70.6 %
Accuracy for class: truck is 46.0 %


Files already downloaded and verified
[1,  2000] loss: 2.159
[1,  4000] loss: 1.846
[1,  6000] loss: 1.690
[1,  8000] loss: 1.617
[1, 10000] loss: 1.569
[1, 12000] loss: 1.538
[2,  2000] loss: 1.464
[2,  4000] loss: 1.459
[2,  6000] loss: 1.451
[2,  8000] loss: 1.435
[2, 10000] loss: 1.399
[2, 12000] loss: 1.396
Finished Training MyCNN

Accuracy of the original CNN on the test images: 53.47%
Accuracy of MyCNN on the test images: 52.56%
```







1.3 Problem-2

Understand how to use pretrained CNN for extracting features and fine- tuning using the following video tutorials and associated codes: Code link:

1. (A) https://www.youtube.com/watch?v=15zlr2vJqKc
2. (B) https://www.youtube.com/watch?v=8etkVC93yU4
3. (c) https://github.com/madsendennis/notebooks/tree/master/pytorch

Solution 2:

```
# -*- coding: utf-8 -*-
"""PyTorch_Transfer_learning.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1
    ↪ d7YhrpOHC19s2tJonqehckJiqVgzR9-p

CNN Model For Image Recognization

Modified Code for CNN Imagae Classification
"""

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')

# Install the gdown library
!pip install gdown

# Define the file ID and output directory
file_id = '1fPqPl3X63XqoSWJBVQoCptPszFibjYb3'
output_dir = '/content/dataset'

# Download the file
!gdown --id $file_id -O /content/dataset.zip

# Unzip the dataset
!unzip /content/dataset.zip -d $output_dir

# Update the dataset path
dataset = '/content/dataset'

import torch
import 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
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
from torchsummary import summary

# 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])
    ])
}

# Set train and valid directory paths
dataset = '/content/dataset/data/'
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))

# 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
idx_to_class = {v: k for k, v in data['train'].class_to_idx.items()}

# Size of Data
train_data_size = len(data['train'])
valid_data_size = len(data['valid'])

# Create data loaders
train_data_loader = DataLoader(data['train'], batch_size=bs, shuffle=True)
```

```python
valid_data_loader = DataLoader(data['valid'], batch_size=bs, shuffle=True)

# Load pre-trained AlexNet model
alexnet = models.alexnet(pretrained=True)

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

# Modify 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))

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

# Define the device (GPU or CPU)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Step 7: Train and Validate the Model

# Define function to train and validate
def train_and_validate(model, loss_criterion, optimizer, epochs=5):
    start = time.time()
    history = []
    best_acc = 0.0

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

        # Set to training mode
        model.train()

        # Loss and Accuracy within the epoch
        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)

            # Clean existing gradients
            optimizer.zero_grad()

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

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

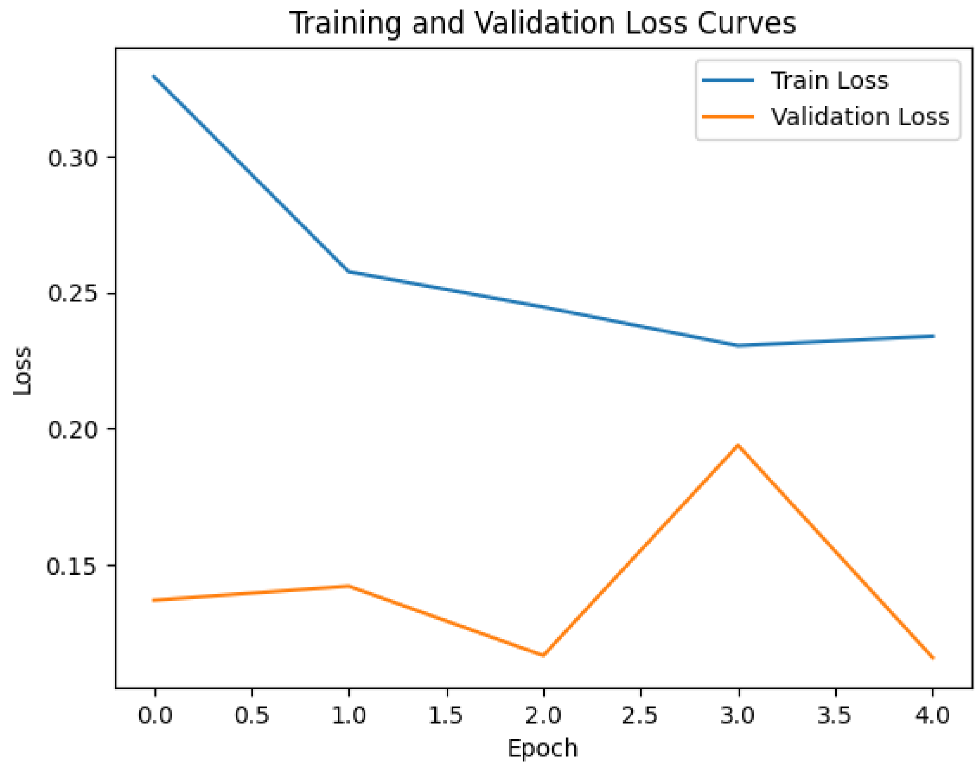
```

            # Backpropagate the gradients
            loss.backward()

            # Update the parameters
            optimizer.step()

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

            # Compute the 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 train_acc
            train_acc += acc.item() * inputs.size(0)

        # 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))

```

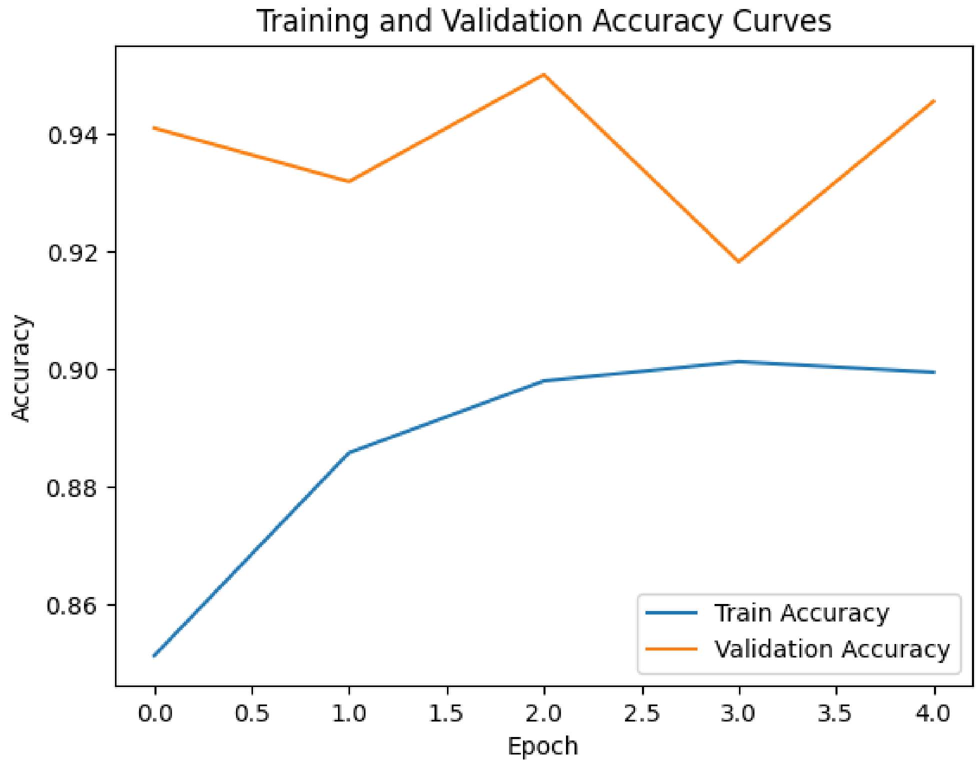
```
                # Compute total accuracy in the whole batch and add to valid_acc
                valid_acc += acc.item() * inputs.size(0)

        # 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 the best validation accuracy till now
        if avg_valid_acc > best_acc:
            best_acc = avg_valid_acc
            torch.save(model, dataset + '_model.pt')

    return model, history

# Specify the number of epochs
num_epochs = 5

# Train and validate the model
trained_model, history = train_and_validate(alexnet, loss_func, optimizer, num_epochs)

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

# Print training and validation curves
history = np.array(history)
plt.plot(history[:, 0], label='Train Loss')
plt.plot(history[:, 1], label='Validation Loss')
plt.legend()
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Validation Loss Curves')
plt.show()

plt.plot(history[:, 2], label='Train Accuracy')
plt.plot(history[:, 3], label='Validation Accuracy')
plt.legend()
plt.xlabel('Epoch')
```

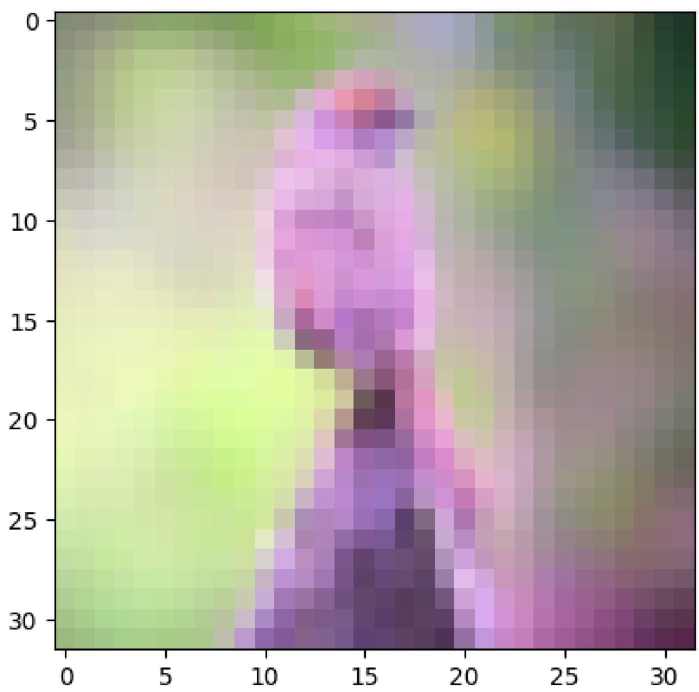
```
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracy Curves')
plt.show()

# Define a function to predict the class of a single test image
def predict(model, test_image_name, topk=3):
    transform = image_transforms['test']
    test_image = Image.open(test_image_name)
    plt.imshow(test_image)
    test_image_tensor = transform(test_image).unsqueeze(0)

    if torch.cuda.is_available():
        test_image_tensor = test_image_tensor.cuda()

    with torch.no_grad():
        model.eval()
        out = model(test_image_tensor)
        ps = torch.exp(out)

        # Check the number of available classes
        num_classes = ps.shape[1]

        # Adjust topk if there are fewer classes than requested
        topk = min(topk, num_classes)

        topk_values, topk_indices = ps.topk(topk, dim=1)
        predictions = []

        for i in range(topk):
            class_index = topk_indices[0][i].item()
            class_name = idx_to_class[class_index]
            score = topk_values[0][i].item()
            predictions.append((class_name, score))

        return predictions

# Example usage of the predict function
test_image_path = '/content/dataset/data/test/bird/38457.png'  # Replace with the path to your test image
predictions = predict(trained_model, test_image_path)

print("Predictions for", test_image_path)
for i, (class_name, score) in enumerate(predictions, start=1):
    print(f"Prediction {i}: Class: {class_name}, Score: {score:.4f}")
```

```
Epoch: 1/5
Epoch : 001, Training: Loss: 0.3292, Accuracy: 85.1190%,
        Validation : Loss : 0.1369, Accuracy: 94.0909%, Time: 137.5592s
Epoch: 2/5
Epoch : 002, Training: Loss: 0.2576, Accuracy: 88.5714%,
        Validation : Loss : 0.1421, Accuracy: 93.1818%, Time: 134.5587s
Epoch: 3/5
Epoch : 003, Training: Loss: 0.2446, Accuracy: 89.7917%,
        Validation : Loss : 0.1166, Accuracy: 95.0000%, Time: 136.8005s
Epoch: 4/5
```

```
Epoch : 004, Training: Loss: 0.2304, Accuracy: 90.1190%,
        Validation : Loss : 0.1938, Accuracy: 91.8182%, Time: 134.1123s
Epoch: 5/5
Epoch : 005, Training: Loss: 0.2339, Accuracy: 89.9405%,
        Validation : Loss : 0.1158, Accuracy: 94.5455%, Time: 139.8286s


Predictions for /content/dataset/data/test/bird/38457.png
Prediction 1: Class: bird, Score: 0.9914
Prediction 2: Class: horse, Score: 0.0086
```




Training and Validation Accuracy Curves
Accuracy
0.94
0.92
0.90
0.88
0.86
Train Accuracy
Validation Accuracy
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epoch


0
5
10
15
20
25
30