

CSL7670 : Fundamentals of Machine Learning

Lab Report



॥ त्वं ज्ञानमयो विज्ञानमयोऽसि ॥

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):

```
1  #!/usr/bin/env python
2  # coding: utf-8
3
4  # # QUESTION 1:
5  # # (a) Understand the code completely and run it.
6  # # (b) Explain the following Pytorch Functions: (i) conv2D (ii) MaxPool2d
7      ↪ (iii)
8  # # Linear (iv) Relu (v) linear.
9  # # (c) Plot the loss function.
10 # In[ ]:
11
12
13 # 1 b)
14 # i)Conv2D:
15 # The conv2D is basically a function in the Pytorch.
16 # It performs a 2D convolution on a given input tensor.
17 # So basically it takes a tensor as an input to perform the image
18     ↪ processing tasks.
19 # It contains a filter which when applied on the input tensor, throws a
20     ↪ feature map.
21 # This filter is applied based on the strides, and filter, given the
22     ↪ application is possible on the tensor.
```

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

```
21
22
23 # 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
→ .

```
27
```

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.

```
31
```

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.

```
37
38
```

In[1]:

```
40
41
```

Question 1a), 1c) Done together. Full code and the plot

```
42 import torch
43 import torchvision
44 import torchvision.transforms as transforms
```

```
46
47
```

In[2]:

```
49
50
```

Transforming to tensors

```
51 transform = transforms.Compose(
52     [transforms.ToTensor(),
53      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

```
55
```

batch_size = 4

```
57
```

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)

```
61
62
```

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
download=True, transform=transform)

```
64
```

```

65 testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,
66                                         shuffle=False, num_workers=2)
67
68 classes = ('plane', 'car', 'bird', 'cat',
69            'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
70
71
72 # In[3]:
73
74
75 import matplotlib.pyplot as plt
76 import numpy as np
77
78 #THE FUNCTION TO SHOW AN IMAGE
79
80
81 def imshow(img):
82     img = img / 2 + 0.5      # unnormalize
83     npimg = img.numpy()
84     plt.imshow(np.transpose(npimg, (1, 2, 0)))
85     plt.show()
86
87
88 # RANDOM TRAINING IMAGE
89 dataiter = iter(trainloader)
90 images, labels = next(dataiter)
91
92 # SHOWING THE IMAGES
93 imshow(torchvision.utils.make_grid(images))
94 # print labels
95 print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
96
97
98 # In[4]:
99
100
101 import torch.nn as nn
102 import torch.nn.functional as F
103
104
105 class Net(nn.Module):
106     def __init__(self):
107         super().__init__()
108         self.conv1 = nn.Conv2d(3, 6, 5)
109         self.pool = nn.MaxPool2d(2, 2)
110         self.conv2 = nn.Conv2d(6, 16, 5)
111         self.fc1 = nn.Linear(16 * 5 * 5, 120)
112         self.fc2 = nn.Linear(120, 84)
113         self.fc3 = nn.Linear(84, 10)
114
115     def forward(self, x):
116         x = self.pool(F.relu(self.conv1(x)))
117         x = self.pool(F.relu(self.conv2(x)))
118         x = torch.flatten(x, 1) # flatten all dimensions except batch

```

```
119         x = F.relu(self.fc1(x))
120         x = F.relu(self.fc2(x))
121         x = self.fc3(x)
122         return x
123
124
125 net = Net()
126
127
128 # In[5]:
129
130
131 # OPTIMIZER
132
133
134 # In[7]:
135
136
137 import torch.optim as optim
138
139 criterion = nn.CrossEntropyLoss()
140 optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
141
142
143 # # THE TRAINING
144
145 # In[24]:
146
147
148 import matplotlib.pyplot as plt
149
150
151 loss_val = []
152
153 for epoch in range(2): # loop over the dataset multiple times
154
155     running_loss = 0.0
156     for i, data in enumerate(trainloader, 0):
157         # get the inputs; data is a list of [inputs, labels]
158         inputs, labels = data
159
160         # zero the parameter gradients
161         optimizer.zero_grad()
162
163         # forward + backward + optimize
164         outputs = net(inputs)
165         loss = criterion(outputs, labels)
166         loss.backward()
167         optimizer.step()
168
169         # print statistics
170         loss_val.append(loss.item())
171         running_loss += loss.item()
172         if i % 2000 == 1999: # print every 2000 mini-batches
```

```

173         print(f'[{epoch+1}],[{i+1:5d}]_loss:{running_loss/_
174             ↪ 2000:.3f}')
175
176     running_loss = 0.0
177
178     plt.plot(loss_val, label='Training_Loss_Plot')
179     plt.xlabel('Iterations')
180     plt.ylabel('The_Loss_Value')
181     plt.legend()
182     plt.show()
183     print('Finished_Training')
184
185
186 # In[9]:
187
188
189 PATH = './cifar_net.pth'
190 torch.save(net.state_dict(), PATH)
191
192
193 # In[10]:
194
195
196 dataiter = iter(testloader)
197 images, labels = next(dataiter)
198
199 # print images
200 imshow(torchvision.utils.make_grid(images))
201 print('GroundTruth:', '_'.join(f'{classes[labels[j]]:5s}' for j in range
202     ↪ (4)))
203
204 # In[12]:
205
206
207 net = Net()
208 net.load_state_dict(torch.load(PATH))
209
210
211 # In[13]:
212
213
214 outputs = net(images)
215
216 # In[14]:
217
218
219 _, predicted = torch.max(outputs, 1)
220
221 print('Predicted:', '_'.join(f'{classes[predicted[j]]:5s}'
222     for j in range(4)))
223
224

```

```

225
226 # # THE WHOLE DATASET IS NOW TAKEN FOR PREDICTION!!
227
228 # In[15]:
229
230
231 correct = 0
232 total = 0
233 # since we're not training, we don't need to calculate the gradients for
    ↪ our outputs
234 with torch.no_grad():
235     for data in testloader:
236         images, labels = data
237         # calculate outputs by running images through the network
238         outputs = net(images)
239         # the class with the highest energy is what we choose as
    ↪ prediction
240         _, predicted = torch.max(outputs.data, 1)
241         total += labels.size(0)
242         correct += (predicted == labels).sum().item()
243
244 print(f'Accuracy of the network on the 10000 test images: {100 * correct /
    ↪ total}%')
245
246
247 # In[16]:
248
249
250 # prepare to count predictions for each class
251 correct_pred = {classname: 0 for classname in classes}
252 total_pred = {classname: 0 for classname in classes}
253
254 # again no gradients needed
255 with torch.no_grad():
256     for data in testloader:
257         images, labels = data
258         outputs = net(images)
259         _, predictions = torch.max(outputs, 1)
260         # collect the correct predictions for each class
261         for label, prediction in zip(labels, predictions):
262             if label == prediction:
263                 correct_pred[classes[label]] += 1
264                 total_pred[classes[label]] += 1
265
266
267 # print accuracy for each class
268 for classname, correct_count in correct_pred.items():
269     accuracy = 100 * float(correct_count) / total_pred[classname]
270     print(f'Accuracy for class: {classname:5s} is {accuracy:.1f}%')
271
272
273 # In[18]:
274
275

```



```
276 # # IS GPU THERE?
277 # device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
278
279 # # Assuming that we are on a CUDA machine, this should print a CUDA
    ↪ device:
280
281 # print(device)
282
283
284 # 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):

```

1  #!/usr/bin/env python
2  # coding: utf-8
3
4  # In[ ]:
5
6
7  # Question 1D)
8  import torch
9  import torchvision
10 import torchvision.transforms as transforms
11
12 # Transforming to tensors
13 transform = transforms.Compose(
14     [transforms.ToTensor(),
15      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
16
17 batch_size = 4
18
19 trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
20                                         download=True, transform=transform
21                                         ↪ )
22 trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
23                                           shuffle=True, num_workers=2)
24
25 testset = torchvision.datasets.CIFAR10(root='./data', train=False,
26                                         download=True, transform=transform)
27 testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size,

```

```

27 shuffle=False, num_workers=2)
28
29 classes = ('plane', 'car', 'bird', 'cat',
30            'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
31
32 import matplotlib.pyplot as plt
33 import numpy as np
34
35 #THE FUNCTION TO SHOW AN IMAGE
36
37
38 def imshow(img):
39     img = img / 2 + 0.5      # unnormalize
40     npimg = img.numpy()
41     plt.imshow(np.transpose(npimg, (1, 2, 0)))
42     plt.show()
43
44
45 # RANDOM TRAINING IMAGE
46 dataiter = iter(trainloader)
47 images, labels = next(dataiter)
48
49 # SHOWING THE IMAGES
50 imshow(torchvision.utils.make_grid(images))
51 # print labels
52 print(' '.join(f'{classes[labels[j]]:5s}' for j in range(batch_size)))
53
54
55
56
57
58
59 # In[8]:
60
61
62 import torch.nn as nn
63 import torch.nn.functional as F
64
65
66 class myCNN(nn.Module):
67     def __init__(self):
68         super().__init__()
69         self.conv1 = nn.Conv2d(3, 5, 5) # changed activation map from 6 to
70             → 5 in conv1
71         self.pool = nn.AvgPool2d(2, 2)
72         self.conv2 = nn.Conv2d(5, 10, 5) #INSTEAD OF 16 I have used 10 as
73             → asked, and input as 5, since in conv1 output was 5
74         self.fc1 = nn.Linear(10 * 5 * 5, 100) # projected to 100
75             → dimensions
76         self.fc3 = nn.Linear(100, 10)
77
78     def forward(self, x):
79         x = self.pool(F.relu(self.conv1(x)))
80         x = self.pool(F.relu(self.conv2(x)))

```

```

78         x = torch.flatten(x, 1) # flatten all dimensions except batch
79         x = F.relu(self.fc1(x))
80         x = self.fc3(x)
81         return x
82
83
84 net = myCNN()
85
86
87 # In[ ]:
88
89
90 # OPTIMIZER
91 import torch.optim as optim
92
93 criterion = nn.CrossEntropyLoss()
94 optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
95
96 # THE TRAINING
97 loss_val = []
98
99 for epoch in range(2): # loop over the dataset multiple times
100
101     running_loss = 0.0
102     for i, data in enumerate(trainloader, 0):
103         # get the inputs; data is a list of [inputs, labels]
104         inputs, labels = data
105
106         # zero the parameter gradients
107         optimizer.zero_grad()
108
109         # forward + backward + optimize
110         outputs = net(inputs)
111         loss = criterion(outputs, labels)
112         loss.backward()
113         optimizer.step()
114
115         # print statistics
116         loss_val.append(loss.item())
117         running_loss += loss.item()
118         if i % 2000 == 1999: # print every 2000 mini-batches
119             print(f'[{epoch+1}], [{i+1:5d}] loss: {running_loss/2000:.3f}')
120
121     running_loss = 0.0
122
123 plt.plot(loss_val, 'r', label='Training Loss Plot')
124 plt.xlabel('Iterations')
125 plt.ylabel('The Loss Value')
126 plt.legend()
127 plt.show()
128 print('Finished Training')
129
130

```

```

131
132 # In[11]:
133
134
135 PATH = './cifar_net.pth'
136 torch.save(net.state_dict(), PATH)
137
138 dataiter = iter(testloader)
139 images, labels = next(dataiter)
140
141 # print images
142 imshow(torchvision.utils.make_grid(images))
143 print('GroundTruth: ', ' '.join('%5s' % classes[labels[j]] for j in range
    ↪ (4)))
144
145 net = myCNN()
146 net.load_state_dict(torch.load(PATH))
147
148 outputs = net(images)
149
150 _, predicted = torch.max(outputs, 1)
151
152 print('Predicted: ', ' '.join('%5s' % classes[predicted[j]]
    ↪ for j in range(4)))
153
154 # THE WHOLE DATASET IS NOW TAKEN FOR PREDICTION!!
155
156 correct = 0
157 total = 0
158 # since we're not training, we don't need to calculate the gradients for
    ↪ our outputs
159 with torch.no_grad():
160     for data in testloader:
161         images, labels = data
162         # calculate outputs by running images through the network
163         outputs = net(images)
164         # the class with the highest energy is what we choose as
    ↪ prediction
165         _, predicted = torch.max(outputs.data, 1)
166         total += labels.size(0)
167         correct += (predicted == labels).sum().item()
168
169
170 print('Accuracy of the network on the 10000 test images: %d%%' %
    ↪ (100 * correct // total))
171
172 # prepare to count predictions for each class
173 correct_pred = {classname: 0 for classname in classes}
174 total_pred = {classname: 0 for classname in classes}
175
176 # again no gradients needed
177 with torch.no_grad():
178     for data in testloader:
179         images, labels = data
180         outputs = net(images)

```

```
181     _, predictions = torch.max(outputs, 1)
182     # collect the correct predictions for each class
183     for label, prediction in zip(labels, predictions):
184         if label == prediction:
185             correct_pred[classes[label]] += 1
186             total_pred[classes[label]] += 1
187
188
189     # print accuracy for each class
190     for classname, correct_count in correct_pred.items():
191         accuracy = 100 * float(correct_count) / total_pred[classname]
192         print(f'Accuracy for class: {classname:5s} is {accuracy:.1f}%')
193
194
195
196
197 # 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	%				

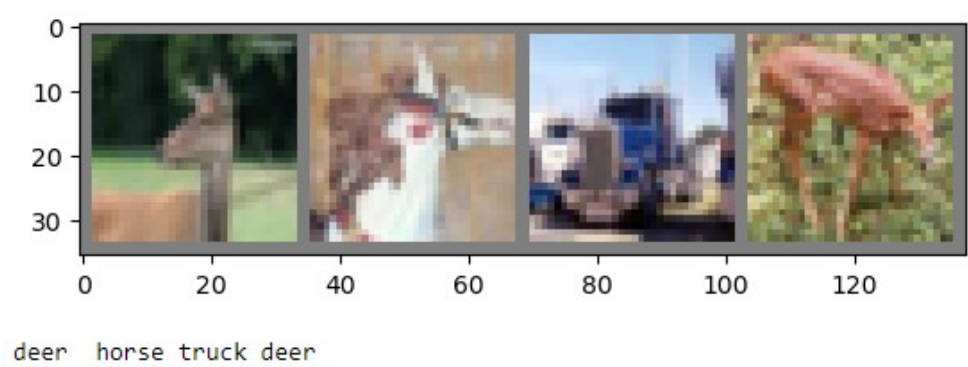


Figure 1.1: The initial random training image.



Figure 1.2: The groundtruth image.

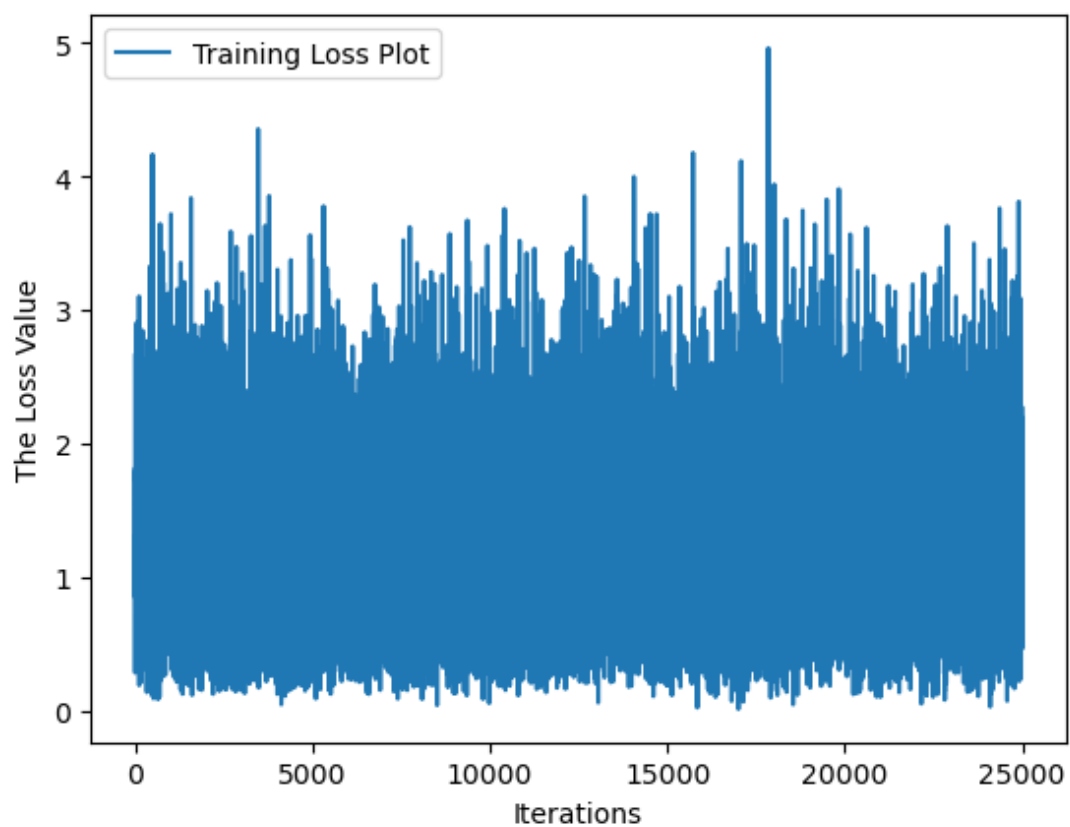


Figure 1.3: The loss function plot(question 1c).

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:

```
1  #!/usr/bin/env python
2  # coding: utf-8
3
4  # In[1]:
5
6
7  import torch, torchvision
8  from torchvision import datasets, models, transforms
9  import torch.nn as nn
10 import torch.optim as optim
11 from torch.utils.data import DataLoader
12 import time
13 from torchsummary import summary
14
15 import numpy as np
16 import matplotlib.pyplot as plt
17 import os
18
19 from PIL import Image
20
21
22 # In[2]:
23
24
25 # pip install torchsummary
26
27
28 # In[3]:
29
30
31 # pip install torchvision
32
33
34 # In[2]:
35
36
37 # Applying Transforms to the Data
38 image_transforms = {
39     'train': transforms.Compose([
40         transforms.RandomResizedCrop(size=256, scale=(0.8, 1.0)),
41         transforms.RandomRotation(degrees=15),
42         transforms.RandomHorizontalFlip(),
43         transforms.CenterCrop(size=224),
44         transforms.ToTensor(),
45         transforms.Normalize([0.485, 0.456, 0.406],
46                               [0.229, 0.224, 0.225])
47     ]),
48     'valid': transforms.Compose([
49         transforms.Resize(size=256),
50         transforms.CenterCrop(size=224),
51         transforms.ToTensor(),
52         transforms.Normalize([0.485, 0.456, 0.406],
53                               [0.229, 0.224, 0.225])
```

```

54     ]),
55     'test': transforms.Compose([
56         transforms.Resize(size=256),
57         transforms.CenterCrop(size=224),
58         transforms.ToTensor(),
59         transforms.Normalize([0.485, 0.456, 0.406],
60                               [0.229, 0.224, 0.225])
61     ])
62 }
63
64
65 # In[3]:
66
67
68 # Load the Data
69
70 # Set train and valid directory paths
71
72 dataset = "C:/Users/user/FML\+IMAGE_PROCESSING+AI_B/FML_Assignments/A_5/
    ↪ data/" #CALL DATA FROM LOCAL DIRECTORY
73
74 train_directory = os.path.join(dataset, 'train')
75 valid_directory = os.path.join(dataset, 'valid')
76
77 # Batch size
78 bs = 32
79
80 # Number of classes
81 num_classes = len(os.listdir(valid_directory)) #10#2#257
82 print(num_classes)
83
84 # Load Data from folders
85 data = {
86     'train': datasets.ImageFolder(root=train_directory, transform=
    ↪ image_transforms['train']),
87     'valid': datasets.ImageFolder(root=valid_directory, transform=
    ↪ image_transforms['valid'])
88 }
89
90 # Get a mapping of the indices to the class names, in order to see the
    ↪ output classes of the test images.
91 idx_to_class = {v: k for k, v in data['train'].class_to_idx.items()}
92 print(idx_to_class)
93
94 # Size of Data, to be used for calculating Average Loss and Accuracy
95 train_data_size = len(data['train'])
96 valid_data_size = len(data['valid'])
97
98 # Create iterators for the Data loaded using DataLoader module
99 train_data_loader = DataLoader(data['train'], batch_size=bs, shuffle=True)
100 valid_data_loader = DataLoader(data['valid'], batch_size=bs, shuffle=True)
101
102
103 # In[4]:

```

```

104
105
106 train_data_size, valid_data_size
107
108
109 # In[5]:
110
111
112 alexnet = models.alexnet(pretrained=True)
113 alexnet
114
115
116 # In[6]:
117
118
119 # Freeze model parameters
120 for param in alexnet.parameters():
121     param.requires_grad = False
122
123
124 # In[7]:
125
126
127 # Change the final layer of AlexNet Model for Transfer Learning
128 alexnet.classifier[6] = nn.Linear(4096, num_classes)
129 alexnet.classifier.add_module("7", nn.LogSoftmax(dim = 1))
130 alexnet
131
132
133 # In[8]:
134
135
136 summary(alexnet, (3, 224, 224))
137
138
139 # In[9]:
140
141
142 # Define Optimizer and Loss Function
143 loss_func = nn.NLLLoss()
144 optimizer = optim.Adam(alexnet.parameters())
145 optimizer
146
147
148 # In[10]:
149
150
151 def train_and_validate(model, loss_criterion, optimizer, epochs=25):
152     '''
153     Function to train and validate
154     Parameters
155         :param model: Model to train and validate
156         :param loss_criterion: Loss Criterion to minimize
157         :param optimizer: Optimizer for computing gradients

```

```

158         :param epochs: Number of epochs (default=25)
159
160     Returns
161         model: Trained Model with best validation accuracy
162         history: (dict object): Having training loss, accuracy and
163             ↪ validation loss, accuracy
164     '''
165
166     start = time.time()
167     history = []
168     best_acc = 0.0
169
170     for epoch in range(epochs):
171         epoch_start = time.time()
172         print("Epoch: {} / {}".format(epoch+1, epochs))
173
174         # Training
175         model.train()
176
177         # Loss and Accuracy for the epochs
178         train_loss = 0.0
179         train_acc = 0.0
180
181         valid_loss = 0.0
182         valid_acc = 0.0
183
184         for i, (inputs, labels) in enumerate(train_data_loader):
185
186             inputs = inputs.to(device)
187             labels = labels.to(device)
188
189             # optimizing the gradient to zero
190             optimizer.zero_grad()
191
192             # Forward pass - compute outputs on input data using the model
193             outputs = model(inputs)
194
195             # Compute the loss
196             loss = loss_criterion(outputs, labels)
197
198             # Gradient Backpropagation
199             loss.backward()
200
201             # Parameters update
202             optimizer.step()
203
204             # Compute the total loss for the batch and add it to
205             ↪ train_loss
206             train_loss += loss.item() * inputs.size(0)
207
208             # Accuracy computation
209             ret, predictions = torch.max(outputs.data, 1)
210             correct_counts = predictions.eq(labels.data.view_as(
211                 ↪ predictions))

```

```

209
210     # Convert correct_counts to float and then compute the mean
211     acc = torch.mean(correct_counts.type(torch.FloatTensor))
212
213     # Compute total accuracy in the whole batch and add to
214     ↪ train_acc
215     train_acc += acc.item() * inputs.size(0)
216
217     #print("Batch number: {:03d}, Training: Loss: {:.4f}, Accuracy
218     ↪ : {:.4f}".format(i, loss.item(), acc.item()))
219
220 # Validation - No gradient tracking needed
221 with torch.no_grad():
222
223     # Set to evaluation mode
224     model.eval()
225
226     # Validation loop
227     for j, (inputs, labels) in enumerate(valid_data_loader):
228         inputs = inputs.to(device)
229         labels = labels.to(device)
230
231         # Forward pass - compute outputs on input data using the
232         ↪ model
233         outputs = model(inputs)
234
235         # Compute loss
236         loss = loss_criterion(outputs, labels)
237
238         # Compute the total loss for the batch and add it to
239         ↪ valid_loss
240         valid_loss += loss.item() * inputs.size(0)
241
242         # Calculate validation accuracy
243         ret, predictions = torch.max(outputs.data, 1)
244         correct_counts = predictions.eq(labels.data.view_as(
245             ↪ predictions))
246
247         # Convert correct_counts to float and then compute the
248         ↪ mean
249         acc = torch.mean(correct_counts.type(torch.FloatTensor))
250
251         # Compute total accuracy in the whole batch and add to
252         ↪ valid_acc
253         valid_acc += acc.item() * inputs.size(0)
254
255         #print("Validation Batch number: {:03d}, Validation: Loss:
256         ↪ {:.4f}, Accuracy: {:.4f}".format(j, loss.item(),
257         ↪ acc.item()))
258
259 # Find average training loss and training accuracy
260 avg_train_loss = train_loss/train_data_size
261 avg_train_acc = train_acc/train_data_size

```

```

254
255     # Find average training loss and training accuracy
256     avg_valid_loss = valid_loss/valid_data_size
257     avg_valid_acc = valid_acc/valid_data_size
258
259     history.append([avg_train_loss, avg_valid_loss, avg_train_acc,
260                    ↪ avg_valid_acc])
261
262     epoch_end = time.time()
263
264     print("Epoch:_{:03d}, Training: Loss:_{:.4f}, Accuracy:_{:.4f}%,
265           ↪ \n\t\tValidation: Loss:_{:.4f}, Accuracy:_{:.4f}%, Time:
266           ↪ {:.4f}s".format(epoch+1, avg_train_loss, avg_train_acc*100,
267           ↪ avg_valid_loss, avg_valid_acc*100, epoch_end-epoch_start))
268
269     # Save if the model has best accuracy till now
270     #torch.save(model, dataset+'_model_'+str(epoch)+'.pt')
271
272     return model, history
273
274 # In[11]:
275
276 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
277
278 num_epochs = 5
279 trained_model, history = train_and_validate(alexnet, loss_func, optimizer,
280 ↪ num_epochs)
281
282 torch.save(history, dataset+'_history.pt')
283
284 # In[14]:
285
286 history = np.array(history)
287 plt.plot(history[:,0:2])
288 plt.legend(['Training_Loss_Value', 'Loss_Validation_Value'])
289 plt.xlabel('Epoch Number')
290 plt.ylabel('Loss')
291 plt.ylim(0,1)
292 plt.savefig(dataset+'_loss_curve.png')
293 plt.show()
294
295 # In[15]:
296
297 plt.plot(history[:,2:4])
298 plt.legend(['Training_Accuracy', 'Validation_Accuracy'])
299 plt.xlabel('Epoch Number')
300 plt.ylabel('Accuracy')
301 plt.ylim(0,1)
302

```

```

303 plt.savefig(dataset+'_accuracy_curve.png')
304 plt.show()
305
306
307 # In[25]:
308
309
310 def prediction_model(model, test_image_name):
311     '''
312     This function will predict the class in which a single image belongs
313     '''
314
315     transform_save = image_transforms['test']
316
317     test_image = Image.open(test_image_name)
318     plt.imshow(test_image)
319
320     test_image_tensor = transform_save(test_image)
321
322     if torch.cuda.is_available():
323         test_image_tensor = test_image_tensor.view(1, 3, 224, 224).cuda()
324     else:
325         test_image_tensor = test_image_tensor.view(1, 3, 224, 224)
326
327     with torch.no_grad():
328         model.eval()
329         # Model outputs log probabilities
330         out = model(test_image_tensor)
331         ps = torch.exp(out)
332
333         # Check the number of elements along dimension 1
334         num_elements = ps.shape[1]
335
336         k = min(3, num_elements) # Setting the value of 'k'
337
338         topk, topclass = ps.topk(k, dim=1)
339         for i in range(k):
340             print("The Prediction", i + 1, "is", idx_to_class[topclass.
341                 ↪ numpy()[0][i]], ", Score:", topk.numpy()[0][i])
342
343
344 # In[26]:
345
346
347 prediction_model(trained_model, 'C:\\Users\\user\\FML\\IMAGE_PROCESSING+
348     ↪ AI_B\\FML_Assignments\\A_5\\data\\test\\bird\\32731.png')
349
350 # In[27]:
351
352
353 prediction_model(trained_model, 'C:\\Users\\user\\FML\\IMAGE_PROCESSING+
354     ↪ AI_B\\FML_Assignments\\A_5\\data\\test\\bird\\32793.png')

```

```
354
355
356 # In[28]:
357
358
359 prediction_model(trained_model, 'C:/Users/user/FML+IMAGE_PROCESSING+AI_B
    ↪ /FML_Assignments/A_5/data/test/horse/31469.png')
360
361
362 # In[29]:
363
364
365 prediction_model(trained_model, 'C:/Users/user/FML+IMAGE_PROCESSING+AI_B
    ↪ /FML_Assignments/A_5/data/test/horse/33845.png')
366
367
368 # 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

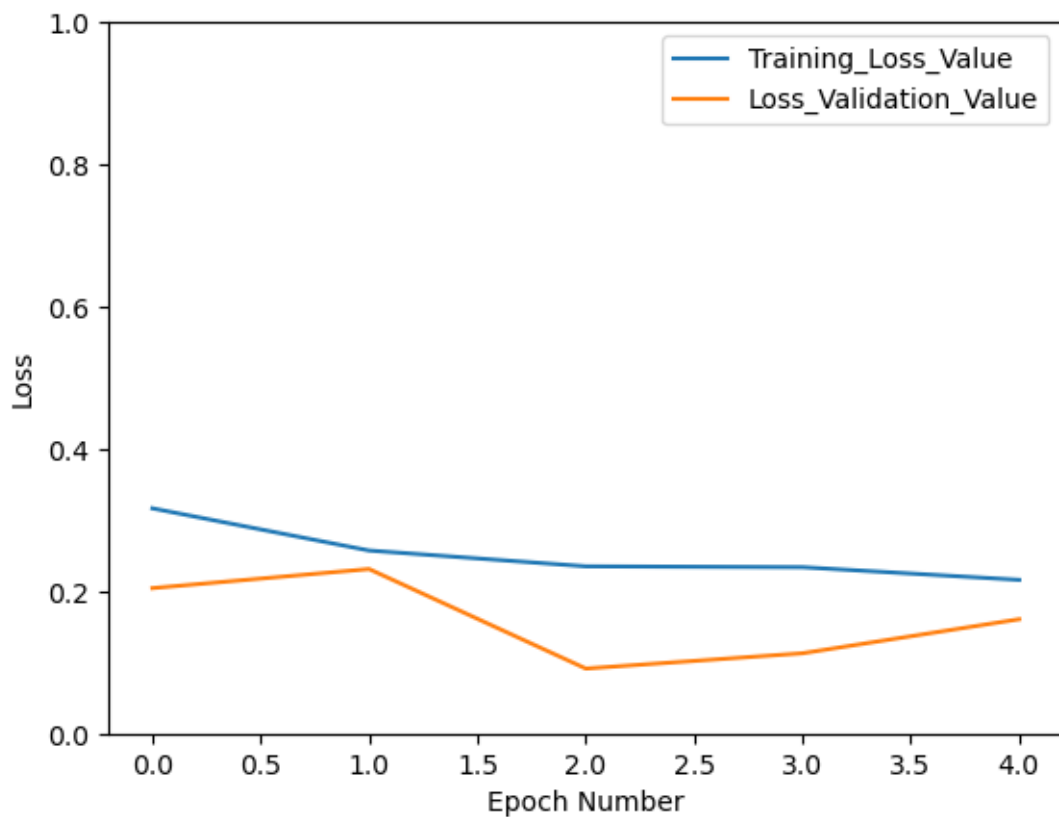
The Loss and Accuracy Graphs

Figure 1.4: The Loss vs Epoch number.

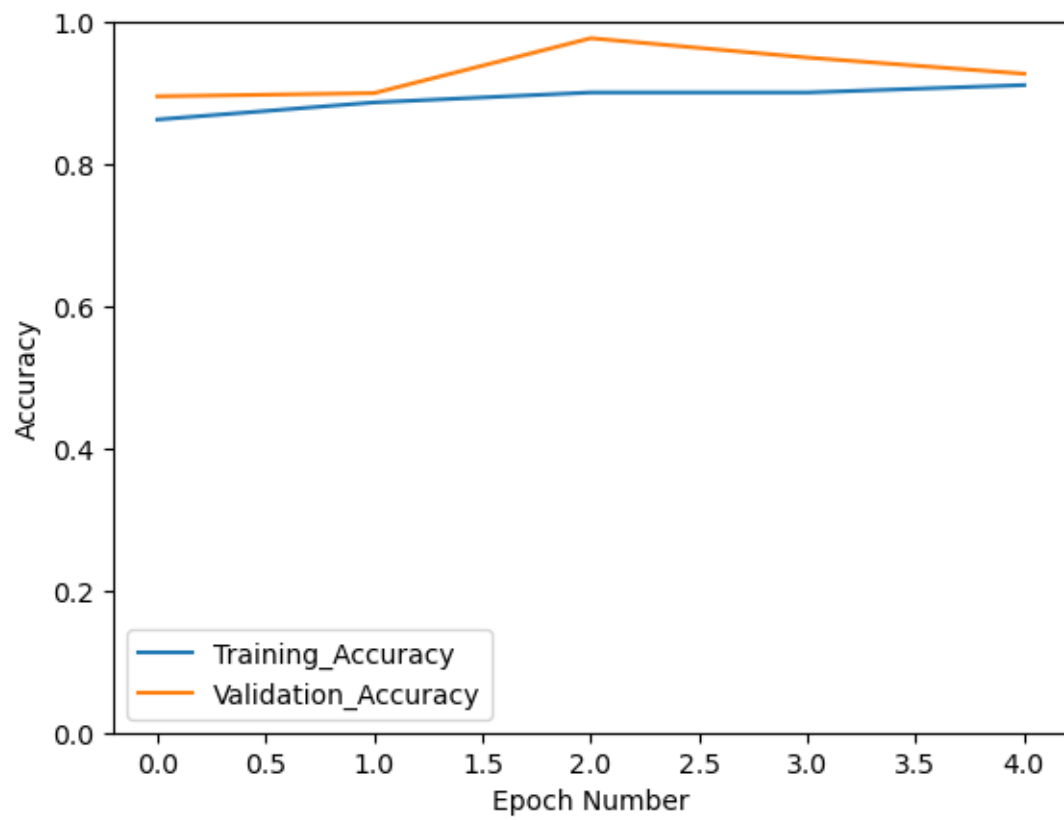


Figure 1.5: The Accuracy vs Epoch number.

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
```

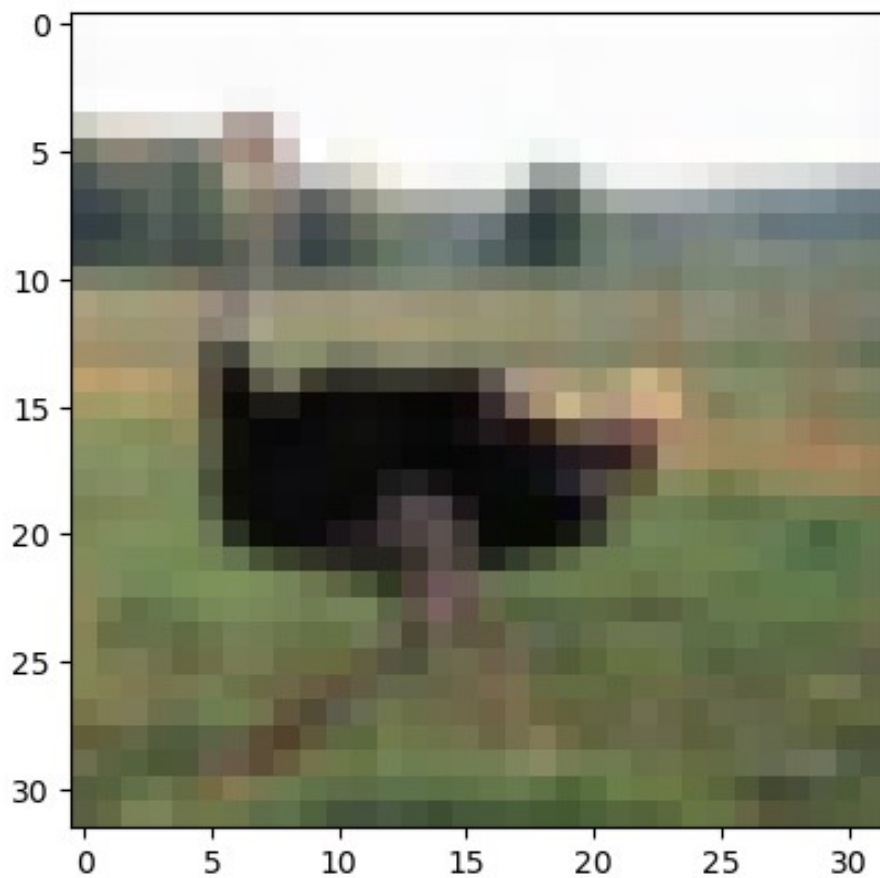


Figure 1.6: Prediction model result-1

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

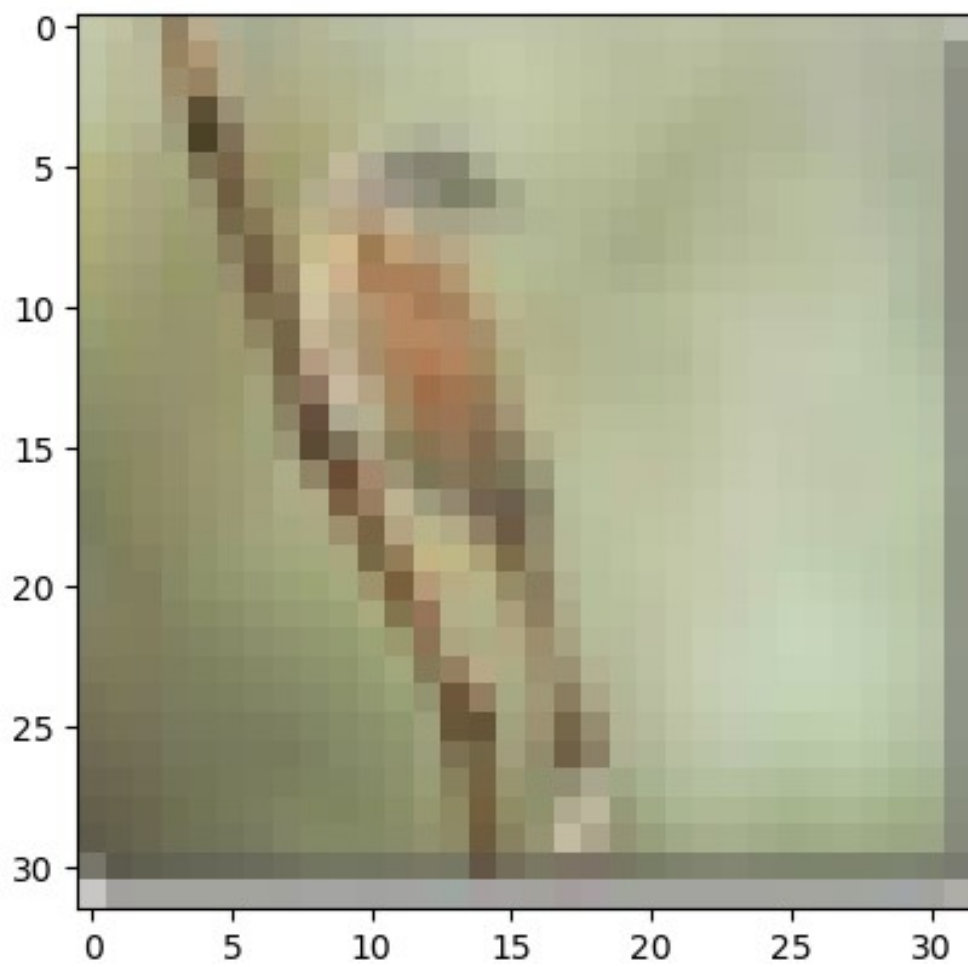


Figure 1.7: Prediction model result-2

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

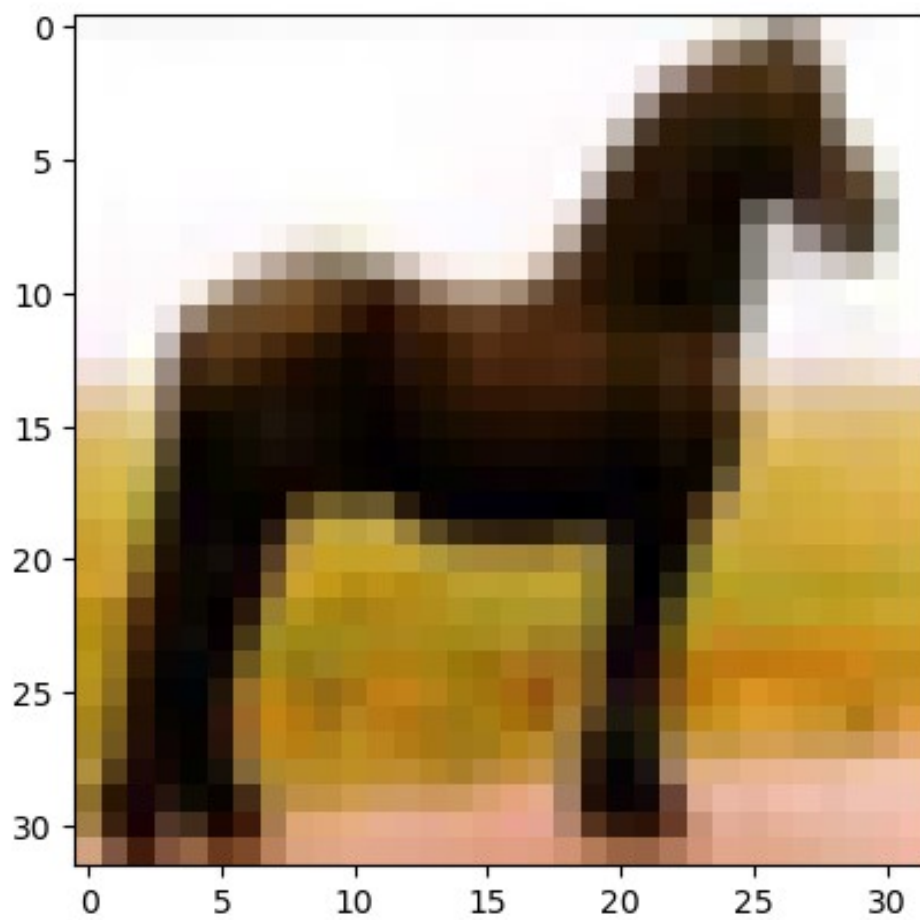


Figure 1.8: Prediction model result-3

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

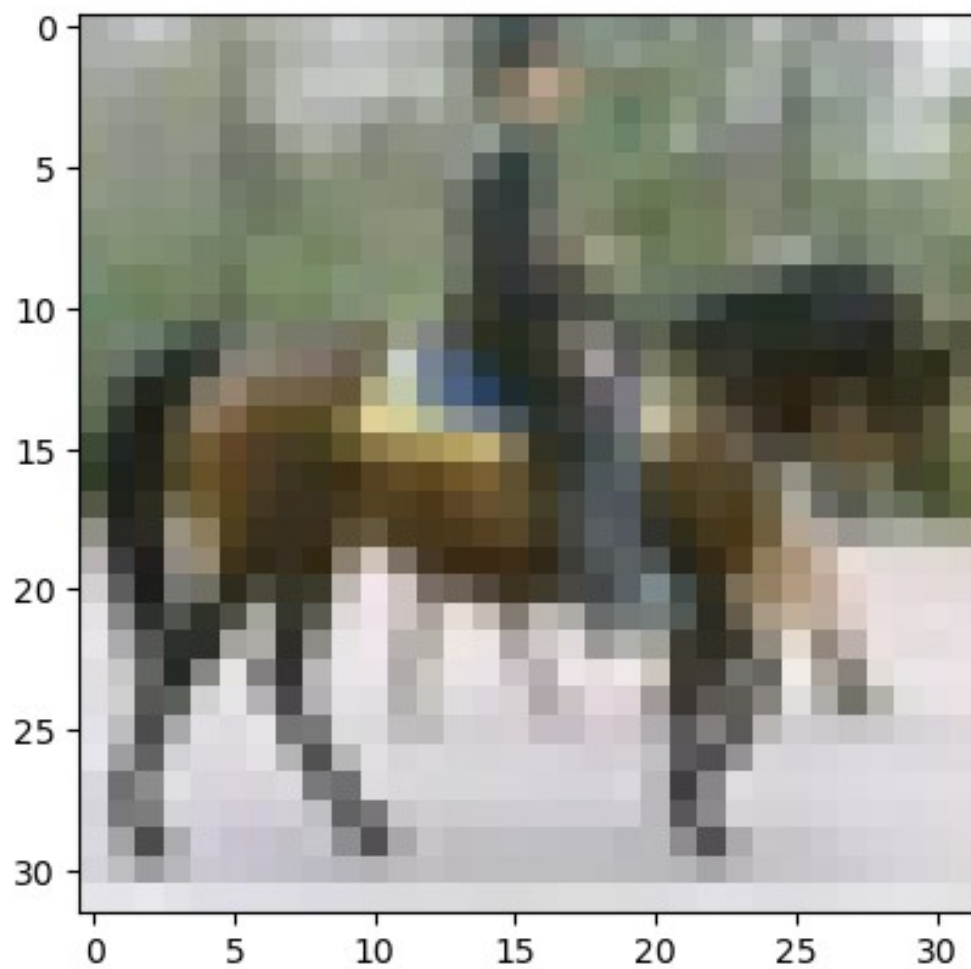


Figure 1.9: Prediction model result-4