Homework 6 Report

Heng Zhang

0029109755

zhan2614

# Introduction

In this homework, we use the pretrained alexnet to train the last layer which has only 200 features instead of 1000. We use tiny imagenet dataset. It has 200 classes where each class has 500 training images, 50 validation images. In this homework, we copied the pretrained alexnet weights except the last layer.

# Setup

I use train.py and test.py to implement these two tasks. The train.py build the model for 200 classes and the test.py is capable to test it by continuously capturing image from camera and giving real-time predicted result.

# Evaluation

## Task 1 training results

The results for this task are shown in Figures 1, 2, and 3.

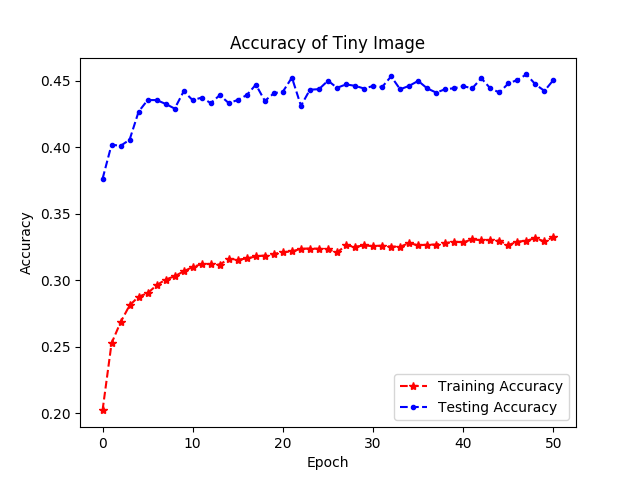
**

Figure Accuracy comparison

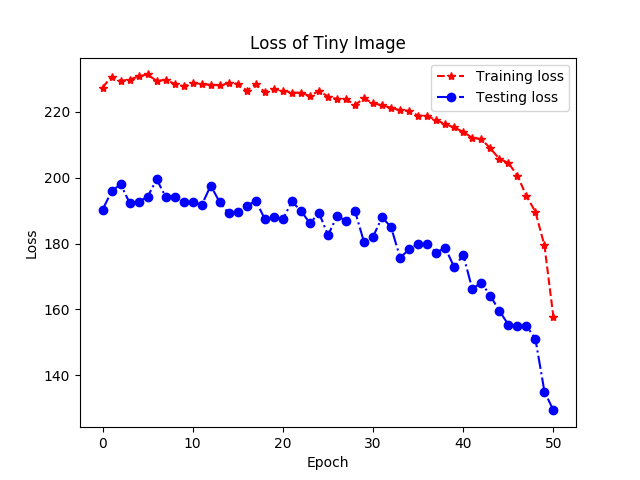


Figure Loss comparison

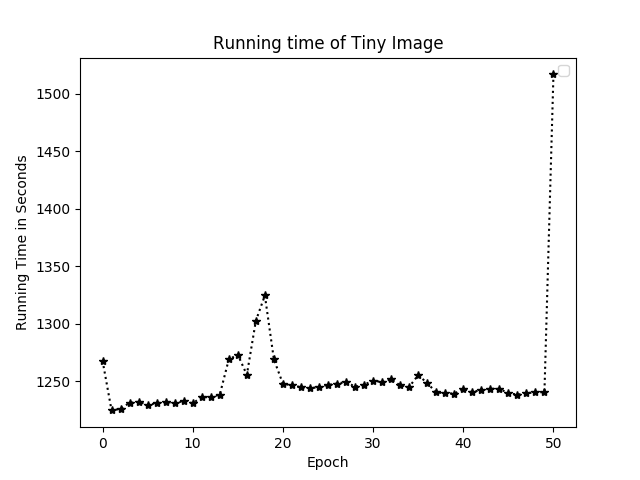


Figure Running time

The accuracy saturates at about 45%. The running time is approximately 20 minutes for one epoch depending the system status.

## Task 2 test if predict the object

I intentionally find a picture of jellyfish and put in front of the camera. It successfully predicts the jellyfish as in Figure 4

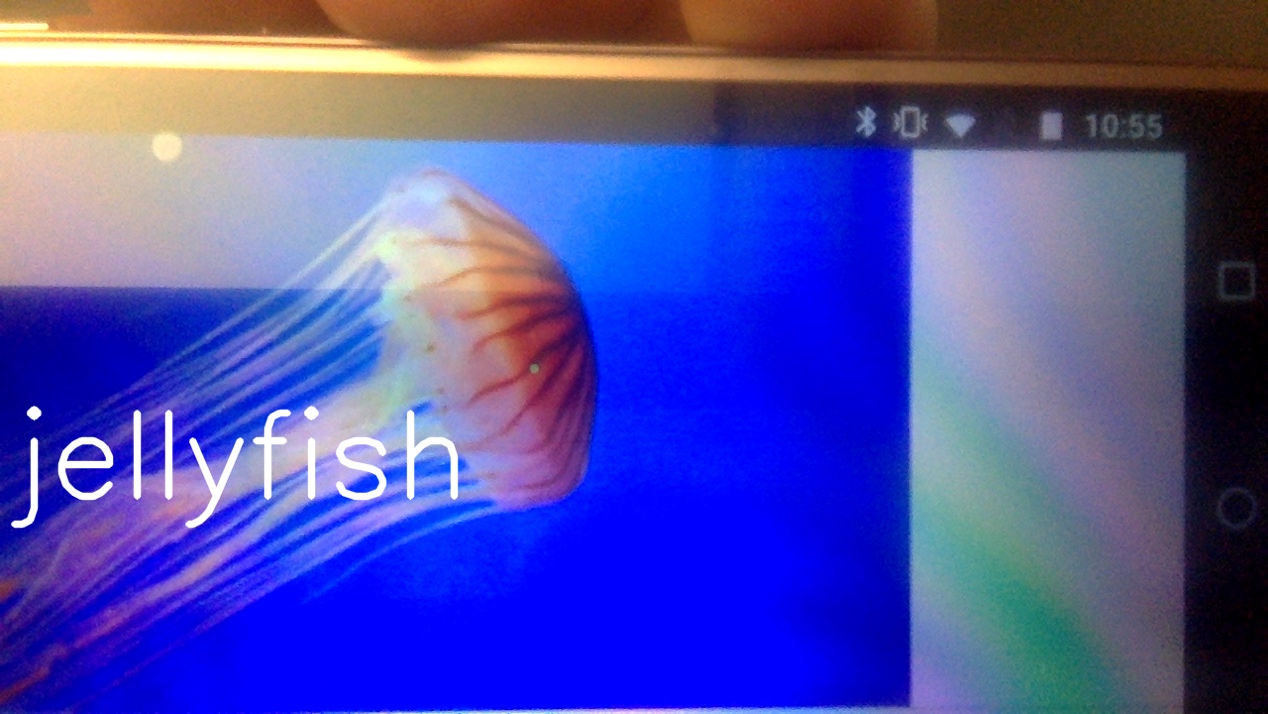


Figure Predicted Jellyfish

I also tested with my keyboard and it correctly predicts it as in Figure 5.



Figure Predicted keyboard

# Appendix

train.py

1. **import** argparse
2. **import** matplotlib.pyplot as plt
3. **import** os
4. **import** shutil
5. **import** torch
6. **import** torch.nn as nn
7. **import** torch.utils.model\_zoo as model\_zoo
8. **import** torch.nn.functional as F
9. **from** torchvision **import** datasets, models, transforms
10. **from** time **import** time

13. **class** AlexNet(nn.Module):
14. **def** \_\_init\_\_(self, num\_classes=200):
15. super(AlexNet, self).\_\_init\_\_()
16. self.features = nn.Sequential(
17. nn.Conv2d(3, 64, kernel\_size=11, stride=4, padding=2),
18. nn.ReLU(inplace=True),
19. nn.MaxPool2d(kernel\_size=3, stride=2),
20. nn.Conv2d(64, 192, kernel\_size=5, padding=2),
21. nn.ReLU(inplace=True),
22. nn.MaxPool2d(kernel\_size=3, stride=2),
23. nn.Conv2d(192, 384, kernel\_size=3, padding=1),
24. nn.ReLU(inplace=True),
25. nn.Conv2d(384, 256, kernel\_size=3, padding=1),
26. nn.ReLU(inplace=True),
27. nn.Conv2d(256, 256, kernel\_size=3, padding=1),
28. nn.ReLU(inplace=True),
29. nn.MaxPool2d(kernel\_size=3, stride=2),
30. )
31. # the fully connected layer
32. self.classifier = nn.Sequential(
33. nn.Dropout(),
34. nn.Linear(256 \* 6 \* 6, 4096),
35. nn.ReLU(inplace=True),
36. nn.Dropout(),
37. nn.Linear(4096, 4096),
38. nn.ReLU(inplace=True),
39. nn.Linear(4096, num\_classes),
40. )
42. **def** forward(self, x):
43. x = self.features(x)
44. x = x.view(x.size(0), -1)
45. x = self.classifier(x)
46. **return** x
48. **class** Model:
49. **def** \_\_init\_\_(self):
50. parser = argparse.ArgumentParser()
51. parser.add\_argument('--data', type=str, help='Directory to thee tiny image set')
52. parser.add\_argument('--save', type=str, help='Directory to save trained model after completion of training')
53. args = parser.parse\_args()

56. self.train\_batch\_size =  100
57. self.epoch = 51
58. self.rate = 0.1
59. self.val\_batch\_size = 10
60. **def** create\_val\_folder():
61. path = os.path.join(args.data, 'val/images')  # path where validation data is present now
62. filename = os.path.join(args.data, 'val/val\_annotations.txt')  # file where image2class mapping is present
63. fp = open(filename, "r")  # open file in read mode
64. data = fp.readlines()  # read line by line
66. # Create a dictionary with image names as key and corresponding classes as values
67. val\_img\_dict = {}
68. **for** line **in** data:
69. words = line.split("\t")
70. val\_img\_dict[words[0]] = words[1]
71. fp.close()
73. # Create folder if not present, and move image into proper folder
74. **for** img, folder **in** val\_img\_dict.items():
75. newpath = (os.path.join(path, folder))
76. **if** **not** os.path.exists(newpath):  # check if folder exists
77. os.makedirs(newpath)
79. **if** os.path.exists(os.path.join(path, img)):  # Check if image exists in default directory
80. os.rename(os.path.join(path, img), os.path.join(newpath, img))
81. create\_val\_folder()
83. # load data: from https://github.com/pytorch/examples/blob/master/imagenet/main.py
85. traindir = os.path.join(args.data, 'train')
86. **if** **not** os.path.exists(traindir):
87. os.makedirs(traindir)
88. valdir = os.path.join(args.data, 'val/images')
90. **if** **not** os.path.exists(valdir):
91. os.makedirs(valdir)
93. normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
94. train\_dataset = datasets.ImageFolder(
95. traindir,
96. transforms.Compose([
97. transforms.RandomResizedCrop(224),
98. transforms.RandomHorizontalFlip(),
99. transforms.ToTensor(),
100. normalize,
101. ]))
103. self.train\_loader = torch.utils.data.DataLoader(train\_dataset, batch\_size = self.train\_batch\_size, shuffle=True, num\_workers = 5)
105. val\_dataset = datasets.ImageFolder(
106. valdir,
107. transforms.Compose([
108. transforms.Resize(256),
109. transforms.CenterCrop(224),
110. transforms.ToTensor(),
111. normalize,
112. ]))
114. self.val\_loader = torch.utils.data.DataLoader(
115. val\_dataset, batch\_size = self.val\_batch\_size, shuffle=True, num\_workers = 5)
116. **def** get\_tiny\_classes(class\_list):
117. fp = open(os.path.join(args.data, 'words.txt'))
118. whole\_class\_dict = {}
119. **for** line **in** fp.readlines():
120. fields = line.split("\t")
121. super\_label = fields[1].split(',')
122. whole\_class\_dict[fields[0]] = super\_label[0].rstrip()
123. fp.close()
125. tiny\_class = {}
126. **for** lab **in** class\_list:
127. **for** k,v **in** whole\_class\_dict.items():
128. **if** lab == k:
129. tiny\_class[k] = v
130. **continue**
131. **return** tiny\_class
133. self.classes = train\_dataset.classes
134. self.tiny\_classes = get\_tiny\_classes(self.classes)
136. pretrained\_model = models.alexnet(pretrained=True)
137. torch.manual\_seed(1)
138. self.model = AlexNet()
139. # To copy parameters
140. **for** i, j **in** zip(self.model.modules(), pretrained\_model.    modules()):  # iterate over both models
141. **if** **not** list(i.children()):
142. **if** len(i.state\_dict()) > 0:  # copy weights only     for the convolution and linear layers
143. **if** i.weight.size() == j.weight.size():  # this helps to prevent copying of weights of last layer
144. i.weight.data = j.weight.data
145. i.bias.data = j.bias.data
146. **for** p **in** self.model.parameters():
147. p.requires\_grad = False
148. **for** p **in** self.model.classifier[6].parameters():
149. p.requires\_grad = True

152. self.optimizer = torch.optim.Adam(self.model.classifier[6].parameters(), lr=self.rate)
153. self.loss\_function = nn.CrossEntropyLoss()
155. self.check\_point\_file = os.path.join(args.save, 'alex\_checkpoint.tar')
156. **if** **not** os.path.exists(os.path.dirname(self.check\_point\_file)):
157. **try**:
158. os.makedirs(os.path.dirname(self.check\_point\_file))
159. **except** OSError as exc: # Guard against race condition
160. **if** exc.errno != errno.EEXIST:
161. **raise**
162. **if** os.path.isfile(self.check\_point\_file):
163. cp = torch.load(self.check\_point\_file)
164. self.start = cp['epoch']
165. self.best\_acc = cp['best\_acc']
167. **print**('checkpoint found at epoch', self.start)
168. self.model.load\_state\_dict(cp['model'])
169. self.optimizer.load\_state\_dict(cp['optimizer'])
171. self.training\_loss = cp['training\_loss']
172. self.training\_acc = cp['training\_acc']
173. self.testing\_loss = cp['testing\_loss']
174. self.testing\_acc = cp['testing\_acc']
175. self.time = cp['time']
176. **else**:
177. self.start = 0
178. self.best\_acc = 0
180. self.training\_loss = []
181. self.training\_acc = []
182. self.testing\_loss = []
183. self.testing\_acc = []
184. self.time = []
185. **def** train(self, plot=False):
186. **def** save(state, better, f=self.check\_point\_file):
187. torch.save(state, f)
188. **if** better:
189. shutil.copyfile(f, os.path.join(args.save, 'alexnet\_best.tar'))
190. **def** training():
191. correct = 0
192. loss = 0
193. self.model.train() # set to training mode
194. **for** batch\_id, (data, target) **in** enumerate(self.train\_loader):
195. # data.view change the dimension of input to use forward function
196. forward\_pass\_output = self.model(data)
197. #print(onehot\_target.type())
198. cur\_loss = self.loss\_function(forward\_pass\_output, target)
199. loss += cur\_loss.data
200. self.optimizer.zero\_grad()
201. cur\_loss.backward()
202. self.optimizer.step()
204. val, position = torch.max(forward\_pass\_output.data, 1)
205. **for** i **in** range(self.train\_batch\_size):
206. **if** position[i] == target.data[i]:
207. correct += 1
208. # loss / number of batches
209. avg\_loss = loss / (len(self.train\_loader.dataset) / self.train\_batch\_size)
210. accuracy = correct / len(self.train\_loader.dataset)
212. **return** avg\_loss, accuracy
214. **def** testing():
215. self.model.eval()
216. loss = 0
217. correct = 0
218. **for** batch\_id, (data, target) **in** enumerate(self.val\_loader):
219. # data.view change the dimension of input to use forward function
220. forward\_pass\_output = self.model(data)
221. cur\_loss = self.loss\_function(forward\_pass\_output, target)
222. loss += cur\_loss.data
223. #print(forward\_pass\_output.size())
224. #print(onehot\_target.size())
225. val, position = torch.max(forward\_pass\_output.data, 1)
226. **for** i **in** range(self.val\_batch\_size):
227. #   print('prediction = {}, actual = {}'.format(int(position), target[i]))
228. **if** position[i] == target[i]:
229. correct += 1
230. # loss / number of batches
231. avg\_loss = loss / (len(self.val\_loader.dataset) / self.val\_batch\_size)
232. accuracy = correct / len(self.val\_loader.dataset)
233. **return** avg\_loss, accuracy
235. **for** i **in** range(self.start + 1, self.epoch + 1):
236. **print**('Epoch {}'.format(i))
237. s = time()
238. **print**('Training\n')
239. train\_loss, train\_accuracy = training()
240. e = time()
241. **print**('Training Done. Testing....\n')
242. test\_loss, test\_accuracy = testing()
243. self.testing\_acc.append(test\_accuracy)
244. self.training\_acc.append(train\_accuracy)
245. self.training\_loss.append(train\_loss)
246. self.testing\_loss.append(test\_loss)
247. self.time.append(e-s)
248. better = False
249. **if** test\_accuracy > self.best\_acc:
250. better = True
251. self.best\_acc = max(self.best\_acc, test\_accuracy)
252. **print**('training\_loss = {}, testing\_loss = {}, training accuracy = {}, testing accuracy = {}, current best test accuracy = {}, time = {}, better = {}'.format(train\_loss, test\_loss, train\_accuracy, test\_accuracy, self.best\_acc, e - s, better))
253. **print**('Saved checkpoint at', i)
254. state = {
255. 'epoch': i,
256. 'best\_acc': self.best\_acc,
257. 'model': self.model.state\_dict(),
258. 'optimizer': self.optimizer.state\_dict(),
259. 'training\_loss': self.training\_loss,
260. 'testing\_loss': self.testing\_loss,
261. 'testing\_acc': self.testing\_acc,
262. 'training\_acc': self.training\_acc,
263. 'time': self.time,
264. 'classes': self.classes,
265. 'tiny\_class': self.tiny\_classes
267. }
268. save(state,better)
269. **if** plot == True:
270. **return** self.time, self.training\_loss, self.testing\_loss, self.training\_acc, self.testing\_acc

273. **def** graph(time, train\_loss, test\_loss, train\_accuracy, test\_accuracy, name):
275. plt.plot(time, 'k\*:')
276. plt.ylabel('Running Time in Seconds')
277. plt.legend()
278. plt.xlabel('Epoch')
279. plt.title("Running time of " + name)
280. plt.savefig(name+'\_time.png')
281. plt.clf()
283. plt.plot(range(len(train\_accuracy)), train\_accuracy, 'r\*--', label='Training Accuracy')
284. plt.plot(range(len(test\_accuracy)), test\_accuracy, 'b.--', label='Testing Accuracy')
285. plt.xlabel('Epoch')
286. plt.ylabel('Accuracy')
287. plt.title("Accuracy of " + name)
288. plt.legend()
289. plt.savefig(name + '\_acc.png')
290. plt.clf()
291. plt.plot(range(len(train\_loss)), train\_loss, 'r\*--', label='Training loss')
292. plt.plot(range(len(test\_loss)), test\_loss, 'bo-.', label='Testing loss')
293. plt.xlabel('Epoch')
294. plt.legend()
295. plt.title("Loss of "+name)
296. plt.ylabel('Loss')
297. plt.savefig(name+'\_loss.png')
298. plt.clf()

301. **if** \_\_name\_\_ == "\_\_main\_\_":
302. m = Model()
303. time, train\_loss, test\_loss, train\_accuracy, test\_accuracy = m.train(True)
304. graph(time, train\_loss, test\_loss, train\_accuracy, test\_accuracy,  'Tiny Image')

test.py

1. **import** argparse
2. **import** cv2
3. **import** os
4. **import** torch
5. **from** torchvision **import** transforms
7. parser = argparse.ArgumentParser()
8. parser.add\_argument('--model', type=str, help='Directory to the saved model')
10. args = parser.parse\_args()
12. **from** train **import** AlexNet
13. **class** TestClass:
14. **def** \_\_init\_\_(self):
15. self.model = AlexNet()
16. # load from model
17. self.check\_point\_file = os.path.join(args.model, 'alex\_checkpoint.tar')
18. **if** **not** os.path.exists(os.path.dirname(self.check\_point\_file)):
19. **try**:
20. os.makedirs(os.path.dirname(self.check\_point\_file))
21. **except** OSError as exc: # Guard against race condition
22. **if** exc.errno != errno.EEXIST:
23. **raise**
24. **if** os.path.isfile(self.check\_point\_file):
25. cp = torch.load(self.check\_point\_file)
26. self.start = cp['epoch']
27. self.best\_acc = cp['best\_acc']
29. **print**('checkpoint found at epoch', self.start)
30. self.model.load\_state\_dict(cp['model'])
31. self.tiny\_class = cp['tiny\_class']
32. self.classes = cp['classes']
33. **else**:
34. **print**('No model found. Exit!!!')
35. exit()
37. **def** forward(self, img):
38. \_4d = torch.unsqueeze(img.type(torch.FloatTensor), 0)
39. self.model.eval()
40. output = self.model(\_4d)
41. \_, result = torch.max(output, 1)
43. **print**('predicted', result)
44. label = self.tiny\_class[self.classes[result.data[0]]]
45. **return** label

48. **def** cam(self, idx = 0):
50. **def** prepare(img\_origin):
51. # Convert to Tensor and Normalize
52. transformer = transforms.Compose([
53. transforms.ToPILImage(),
54. transforms.Scale(256),
55. transforms.CenterCrop(224),
56. transforms.ToTensor(),
57. transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])])
59. **return** transformer(img\_origin)
61. cam = cv2.VideoCapture(idx)
62. cam.set(3, 1280)
63. cam.set(4, 720)
64. cv2.namedWindow("test")
65. img\_counter = 0
66. **print**('Press e/E to exit, c/C to capture a picture\n')
67. **while** True:
68. ret, frame = cam.read()
69. **if** **not** ret:
70. **break**
71. norm\_img\_tensor = prepare(frame)
72. predicted\_category = self.forward(norm\_img\_tensor)
73. **print**(predicted\_category)
74. cv2.putText(frame, predicted\_category, (10,500), cv2.FONT\_HERSHEY\_SIMPLEX, 4, (255, 255, 255), 5, cv2.LINE\_AA)
75. cv2.imshow('Capturing', frame)
77. k = cv2.waitKey(1) & 0xFF
78. **if** k == ord('e'):
79. # e pressed
80. **print**("E hit, closing...")
81. **break**
82. **elif** k == ord('c'):
83. # c pressed
84. img\_name = "opencv\_frame\_{}.png".format(img\_counter)
85. cv2.imwrite(img\_name, frame)
86. **print**("{} written!".format(img\_name))
87. img\_counter += 1
89. cam.release()
90. cv2.destroyAllWindows()
92. **if** \_\_name\_\_ == "\_\_main\_\_":
93. md = TestClass()
94. md.cam()