

ECE 601: Homework 5

Wei Xu

Email: xu1639@purdue.edu

Due date: 11:59 PM, Feb. 15, 2024
(Spring 2024)

1 Building My Deep Convolutional Neural Network

Note: The code for this section is borrowed from the DLStudio package (<https://engineering.purdue.edu/kak/distDLS/>).

The SkipBlock class is directly copied from DLStudio, and there is no any modification. The deep neural network class HW5Net is shown below, and this is modified based on BMEnet from DLStudio. In HW5Net, the first is the initial convolutional layer following by batch normalization, ReLU activation, and max pooling. Then it has four skip block arrays, and each one has specific fixed channel numbers. There are also three downsampling layers with skip connections and each one is between two skip block arrays and doubles the number of channels at the same time. There are also two linear layers to take care of the final prediction from the features. In this network, there are $5 + 1 + 5 + 1 + 5 + 1 + 5 = 23$ SkipBlock instances. And the number of layers counted by `len(list(net.parameters()))` is 250, because it counts weights and bias separately and it also counts the layers such as batch normalization.

```
1 ...
2 # skip connection block, which is from DLStudio
3 class SkipBlock(nn.Module):
4     """
5     Class Path: DLStudio -> SkipConnections -> SkipBlock
6     """
7     def __init__(self, in_ch, out_ch, downsample=False,
8         ↪ skip_connections=True):
9         super().__init__()
10        self.downsample = downsample
11        self.skip_connections = skip_connections
12        self.in_ch = in_ch
13        self.out_ch = out_ch
14        self.conv1 = nn.Conv2d(in_ch, in_ch, 3, stride=1, padding=1)
15        self.conv2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
16        self.bn1 = nn.BatchNorm2d(in_ch)
17        self.bn2 = nn.BatchNorm2d(out_ch)
18
19        self.in2out = nn.Conv2d(in_ch, out_ch, 1) ### <<<<< from
20        ↪ Cheng-Hao Chen
21
22        if downsample:
23            ## Setting stride to 2 and kernel_size to 1 amounts to
24            ↪ retaining every
25            ## other pixel in the image --- which halves the size of
26            ↪ the image:
27            self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, stride=2)
28            self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1, stride=2)
```

```

26     def forward(self, x):
27         identity = x
28         out = self.conv1(x)
29         out = self.bn1(out)
30         out = nn.functional.relu(out)
31         out = self.conv2(out)
32         out = self.bn2(out)
33         out = nn.functional.relu(out)
34         if self.downsample:
35             identity = self.downsampler1(identity)
36             out = self.downsampler2(out)
37         if self.skip_connections:
38             if (self.in_ch == self.out_ch) and (self.downsample is
39                 ↪ False):
40                 out = out + identity
41             elif (self.in_ch != self.out_ch) and (self.downsample is
42                 ↪ False):
43
44                 identity = self.in2out( identity ) ### <<<< from Cheng-
45                 ↪ Hao Chen
46
47                 out = out + identity
48             elif (self.in_ch != self.out_ch) and (self.downsample is
49                 ↪ True):
50                 out = out + torch.cat((identity, identity), dim=1)
51         return out
52
53 class HW5Net(nn.Module):
54     def __init__(self, cifar=False):
55         super().__init__()
56         # initial convolutional layer
57         self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding
58             ↪ =1)
59         # batch normalization
60         self.bn = nn.BatchNorm2d(64)
61         # ReLU activation
62         self.relu = nn.ReLU(inplace=True)
63         # max pooling and downsizing
64         self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
65         # convolutional layers with skip connection
66         self.skip64_arr = nn.ModuleList()
67         for i in range(5):
68             self.skip64_arr.append(SkipBlock(64, 64, skip_connections=
69                 ↪ True))
70         # downsampling convolutional layers with skip connection
71         self.ds1 = SkipBlock(64, 128, downsample=True, skip_connections
72             ↪ =True)
73         # convolutional layers with skip connection
74         self.skip128_arr = nn.ModuleList()
75         for i in range(5):
76             self.skip128_arr.append(SkipBlock(128, 128,
77                 ↪ skip_connections=True))
78         # downsampling convolutional layers with skip connection
79         self.ds2 = SkipBlock(128, 256, downsample=True,

```

```

72         ↪ skip_connections=True)
73     # convolutional layers with skip connection
74     self.skip256_arr = nn.ModuleList()
75     for i in range(5):
76         self.skip256_arr.append(SkipBlock(256, 256,
77             ↪ skip_connections=True))
78     # downsampling convolutional layers with skip connection
79     self.ds3 = SkipBlock(256, 512, downsample=True,
80         ↪ skip_connections=True)
81     # convolutional layers with skip connection
82     self.skip512_arr = nn.ModuleList()
83     for i in range(5):
84         self.skip512_arr.append(SkipBlock(512, 512,
85             ↪ skip_connections=True))
86     # adaptive average pooling over height and width
87     self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
88     # flatten to a vector for each sample
89     self.flatten = nn.Flatten()
90     # dropout layer to avoid over-fitting
91     self.dropout = nn.Dropout(p=0.3)
92     # linear layers
93     self.fc1 = nn.Linear(512, 100)
94     if cifar:
95         self.fc2 = nn.Linear(100, 10)
96     else:
97         self.fc2 = nn.Linear(100, 5)
98
99 def forward(self, x):
100     # feature extraction
101     x = self.conv1(x)
102     x = self.bn(x)
103     x = self.relu(x)
104     x = self.maxpool(x)
105     for skip64 in self.skip64_arr:
106         x = skip64(x)
107     x = self.ds1(x)
108     for i, skip128 in enumerate(self.skip128_arr):
109         x = skip128(x)
110     x = self.ds2(x)
111     for i, skip256 in enumerate(self.skip256_arr):
112         x = skip256(x)
113     x = self.ds3(x)
114     for i, skip512 in enumerate(self.skip512_arr):
115         x = skip512(x)
116     # classification
117     x = self.avgpool(x)
118     x = self.flatten(x)
119     x = self.dropout(x)
120     x = self.fc1(x)
121     x = self.relu(x)
122     x = self.dropout(x)
123     x = self.fc2(x)
124     return x

```

2 Training and Evaluating My Trained Network

Note: The code for this section is borrowed from the DLStudio package (<https://engineering.purdue.edu/kak/distDLS/>).

The training and evaluation pipelines are modified based on my implementation from HW4. The code is shown below. The `COCODataset` class is the same as that from HW4. The `load_cifar_10_dataset` function is mainly the same as that from DLStudio, and only the dataset path is modified and the numbers of samples in the training set and test set are returned. In `net_training`, the `net_list` contains a list of tuples consisting of the networks and different learning rates. And the networks are trained in a for loop. In addition, in `net_training`, `train`, `plot_learning_curve`, `net_evaluation`, a bool variable `cifar` is applied to indicate whether the CIFAR-10 dataset is used (MS-COCO dataset is used by default). `classFromTensor` is the same as that from HW4.

```
1 ...
2 # MS-COCO dataset class
3 class COCODataset(Dataset):
4     def __init__(self, path, device, train=True):
5         super().__init__()
6         self.device = device
7         # load the image and category files
8         if train:
9             self.imgs = np.load('{}coco_train.npy'.format(path))
10            self.dict = np.load('{}coco_train_dict.npy'.format(path),
11                                ↪ allow_pickle=True).item()
12        else:
13            self.imgs = np.load('{}coco_val.npy'.format(path))
14            self.dict = np.load('{}coco_val_dict.npy'.format(path),
15                                ↪ allow_pickle=True).item()
16        # transform the array to tensor and normalize
17        self.xform = tvf.Compose([tvf.ToTensor(),
18                                   tvf.Normalize([0.5,0.5,0.5],\
19                                                  [0.5,0.5,0.5])])
19    def __len__(self):
20        return self.imgs.shape[0]
21    def __getitem__(self, index):
22        # one-hot encoding
23        class_enc = torch.zeros(5, device=self.device)
24        class_enc[self.dict[index][2]] = 1
25        return self.xform(self.imgs[index, :, :, :]).to(self.device),
26        ↪ class_enc
27
28 # CIFAR-10 data loader, which is from DLStudio
29 def load_cifar_10_dataset(batch_size):
30     ## But then the call to Normalize() changes the range to -1.0-1.0
31     ↪ float vals.
32     transform = tvf.Compose([tvf.ToTensor(),
33                               tvf.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
34                               ↪ 0.5))])
35
36     ## Define where the training and the test datasets are located:
37     train_data_loc = torchvision.datasets.CIFAR10(root='./dataset/CIFAR
38     ↪ -10/', train=True, download=True, transform=transform)
39     test_data_loc = torchvision.datasets.CIFAR10(root='./dataset/CIFAR
40     ↪ -10/', train=False, download=True, transform=transform)
41     ## Now create the data loaders:
```

```

35     train_data_loader = DataLoader(train_data_loc, batch_size=
        ↪ batch_size, shuffle=True, num_workers=2)
36     test_data_loader = DataLoader(test_data_loc, batch_size=batch_size,
        ↪ shuffle=False, num_workers=2)
37     return train_data_loader, test_data_loader, len(train_data_loc),
        ↪ len(test_data_loc)
38 ...
39 # plot the learning curves
40 def plot_learning_curve(loss_list, save_every, net_list, cifar):
41     # calculate x-axis
42     x = len(loss_list[0])
43     x = [i*save_every for i in range(x)]
44     plt.figure(figsize=(8, 6))
45     # plot each curve
46     for i, loss in enumerate(loss_list):
47         plt.plot(x, loss, label='learning rate = {:.0e}'.format(
            ↪ net_list[i][1]))
48     plt.xlabel('Iterations')
49     plt.ylabel('losses')
50     plt.legend()
51     cf = plt.gcf()
52     cf.savefig('loss_c_{}.jpg'.format(cifar), bbox_inches='tight', \
53               format='jpg', dpi=1000)
54
55 # plot confusion matrix
56 def plot_confusion_matrix(pred, gt, cat_list, title, train):
57     plt.figure(figsize=(4,4))
58     ConfusionMatrixDisplay.from_predictions(gt, pred,\
59         display_labels=cat_list, cmap='Blues')
60     cf = plt.gcf()
61     cf.savefig('CM_HW5Net{}_{}.jpg'.format(title, train), bbox_inches='
        ↪ tight', format='jpg')
62
63 # train a networking
64 def train(net, train_data_loader, epochs, lr, save_every, device, cifar
        ↪ ):
65     net = net.to(device)
66     # use cross entropy loss
67     criterion = nn.CrossEntropyLoss()
68     # use Adam optimizer
69     optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas
        ↪ =(0.9,0.99))
70     j = 1
71     loss_all = []
72     running_loss_save = 0.0
73     for epoch in range(epochs):
74         running_loss = 0.0
75         for i, data in enumerate(train_data_loader):
76             inputs, labels = data
77             if cifar: # put to desired device if from CIFAR-10
78                 inputs = inputs.to(device)
79                 labels = labels.to(device)
80             optimizer.zero_grad()
81             # predict

```

```

82         outputs = net(inputs)
83         loss = criterion(outputs, labels)
84         if j % save_every == 0:
85             loss_all.append(running_loss_save/save_every)
86             running_loss_save = 0.0
87         loss.backward()
88         # update weights
89         optimizer.step()
90         running_loss_save += loss.item()
91         running_loss += loss.item()
92         if (i+1) % 100 == 0:
93             print('[epoch: %d, batch: %5d] loss: %.3f'%(epoch+1, i
94                 ↪ +1, running_loss/100))
95             running_loss = 0.0
96             j += 1
97         return net, loss_all
98
99 # train the three networks
100 def net_training(batch_size, save_every, epochs, device, cifar=False):
101     if cifar:
102         # create data loader
103         train_data_loader, _, _ = load_cifar_10_dataset(batch_size)
104         net_list = [(HW5Net(True), 1e-5), (HW5Net(True), 1e-3)]
105     else:
106         # create the dataset instance
107         dataset_train = COCODataset('./dataset', device, train=True)
108         # define the dataloader
109         train_data_loader = DataLoader(dataset_train, batch_size=
110             ↪ batch_size, shuffle=True, drop_last=True)
111         net_list = [(HW5Net(), 1e-5), (HW5Net(), 1e-3)]
112     # train three networks in a loop
113     loss_list = []
114     for i, (net, lr) in enumerate(net_list):
115         print('Training HW5Net with learning rate of %s...'%(str(lr)))
116         num_layers = len(list(net.parameters()))
117         print('Number of learnable layers: ', num_layers)
118         net_trained, loss = train(net, train_data_loader, epochs, lr,
119             ↪ save_every, device, cifar)
120         # save the trained network
121         torch.save(net_trained, './net{}.pth'.format(i+1+cifar*2))
122         loss_list.append(loss)
123     # plot learning curve
124     plot_learning_curve(loss_list, save_every, net_list, cifar)
125     np.save('loss.npy', loss_list)
126
127 # get class index from the one-hot encoding
128 def classFromTensor(tensor):
129     # find out the class index
130     index = torch.max(tensor, -1)[1]
131     return index.detach().cpu().tolist()
132
133 # evaluate the networks
134 def net_evaluation(batch_size, train, device, cifar=False):
135     if cifar:

```

```

133     # create data loader
134     if train:
135         val_data_loader, _, num_sample, _ = load_cifar_10_dataset(
136             ↪ batch_size)
137     else:
138         _, val_data_loader, _, num_sample = load_cifar_10_dataset(
139             ↪ batch_size)
140 else:
141     # create the dataset instance
142     dataset_val = COCODataset('./dataset', device, train=train)
143     num_sample = len(dataset_val)
144     # define the dataloader
145     val_data_loader = DataLoader(dataset_val, batch_size=batch_size
146         ↪ , shuffle=False, drop_last=False)
147 for i in range(2):
148     # load the trained network
149     net = torch.load('./net{}.pth'.format(i+1+cifar*2)).to(device).
150     ↪ eval()
151 gt = []
152 pred = []
153 with torch.no_grad():
154     for data in val_data_loader:
155         inputs, labels = data
156         if cifar:
157             inputs = inputs.to(device)
158             labels = labels.to(device)
159             gt += list(labels.detach().cpu())
160         else:
161             gt += classFromTensor(labels)
162             outputs = net(inputs)
163             pred += classFromTensor(outputs)
164 gt_array = np.array(gt)
165 pred_array = np.array(pred)
166 # count the number of correct prediction
167 n_correct_pred = sum(gt_array == pred_array)
168 # calculate the accuracy
169 accuracy = n_correct_pred / num_sample
170 print('Accuracy for the %dth neural network on training set (=%
171     ↪ s) is %.2f%%'%(i+1+cifar*2, str(train), accuracy*100))
172 # plot the confusion matrix
173 if cifar:
174     plot_confusion_matrix(pred, gt, ['plane', 'car', 'bird', 'cat',
175         ↪ , 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'], i+1+
176         ↪ cifar*2, train)
177 else:
178     plot_confusion_matrix(pred, gt, ['boat', 'cake', 'couch', 'dog
179         ↪ ', 'motorcycle'], i+1, train)
180 ...

```

2.1 Training and evaluation over MS-COCO dataset

The batch size is 16, and the number of epochs is 10. The training loss curves for two different learning rates can be found in Fig. 1. When the learning rate is larger, it is harder to converge,

but when the learning curve is smaller, it converges much faster.

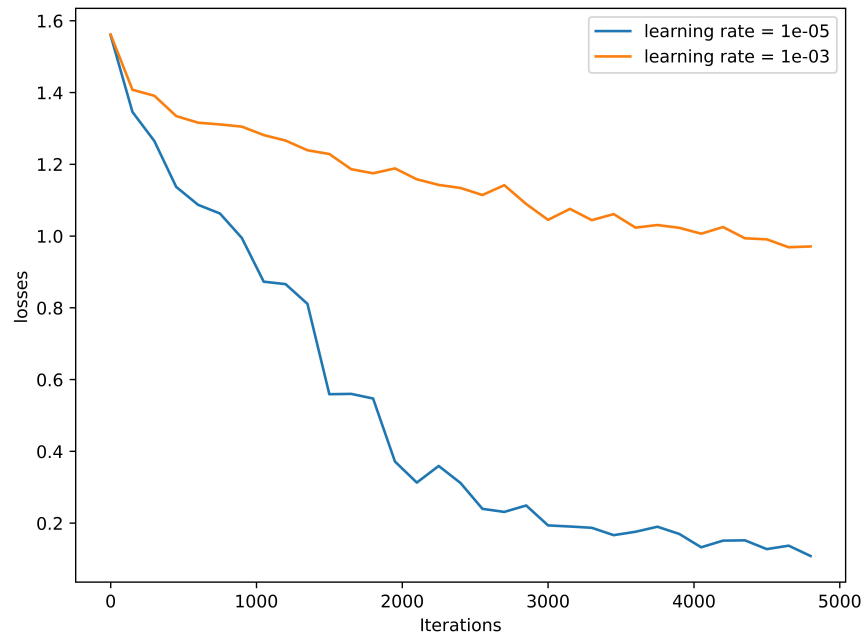


Figure 1: Learning curves over MS-COCO dataset.

The confusion matrix can be found in Fig. 2. For **HW5Net**, when the learning rate is smaller, the prediction is more "balanced", but when the learning rate is larger, it is more sensitive to some of the categories, such as "motorcycle" and "couch", while it is not sensitive to "dog". Compared with the confusion matrix by **Net3** from HW4, the matrices by **HW5Net** are better, having more correct predictions. **Net3** from HW4 has a higher probability of having the prediction of "cake".

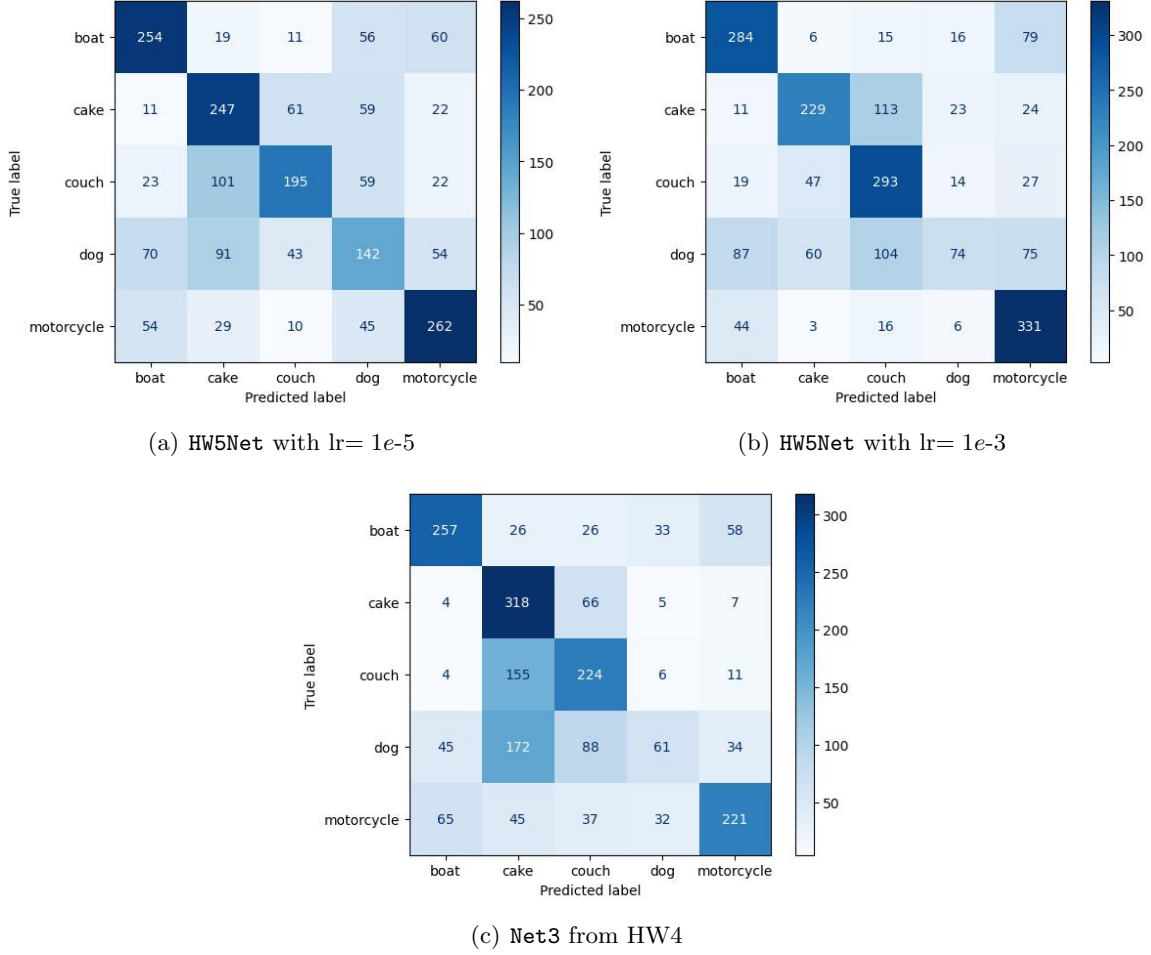


Figure 2: Confusion matrix on validation set of MS-COCO.

The accuracy can be found in Tab. 1. For HW5Net, when the learning rate is smaller, it is overfitting to the training set, because the accuracy on the training set is more than 99%, but the accuracy on the validation set is not such high (only 55%); but when the learning rate is higher, the performance is much better, although the accuracy on the training set is not too high (just about 66%), the accuracy on the validation set is the highest (more than 60%). Net3 from HW4 performs worst in terms of accuracy on both training and validation sets, which is because of the gradient vanishing. Overall, the skip connection helps the network overcome the gradient vanishing, even if the network is much deeper.

Table 1: Accuracy on MS-COCO.

Network	Accuracy (training set) (%)	Accuracy (validation set) (%)
HW5Net with $lr= 1e-5$	99.74	55.00
HW5Net with $lr= 1e-3$	65.74	60.55
Net3 from HW4	53.99	54.05

2.2 Optional: Training and evaluation over CIFAR-10 dataset

The batch size is 16, and the number of epochs is 4. The training loss curves for two different learning rates over the CIFAR-10 dataset can be found in Fig. 3. The same trend is observed, but it is not as obvious as that for MS-COCO dataset. When the learning rate is larger, it also converges, but just slower than the situation when the learning curve is smaller.

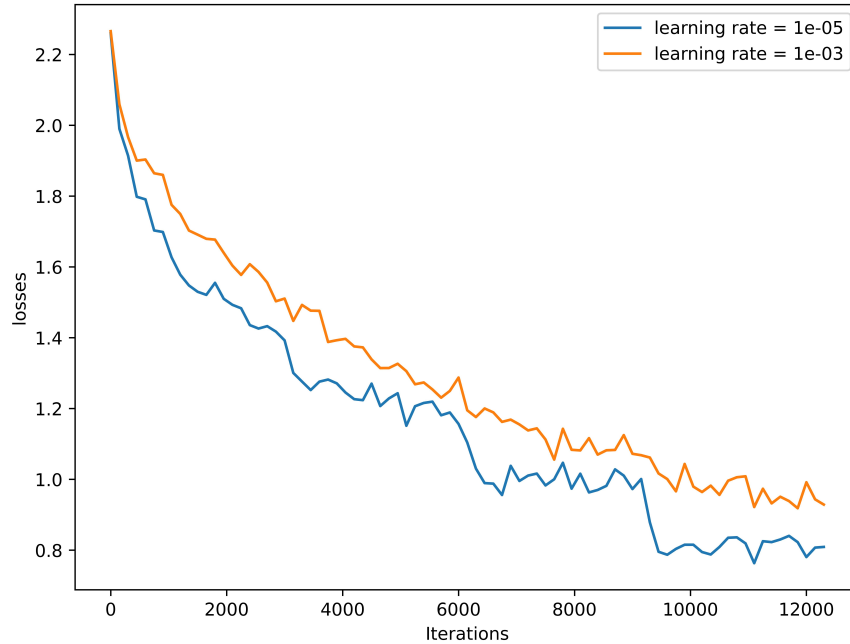


Figure 3: Learning curves over CIFAR-10 dataset.

The confusion matrix can be found in Fig. 4. Overall, HW5Net performs better on CIFAR-10 than on MS-COCO, because the prediction is more concentrated on the diagonal. And when the learning rate is larger, this is more obvious.

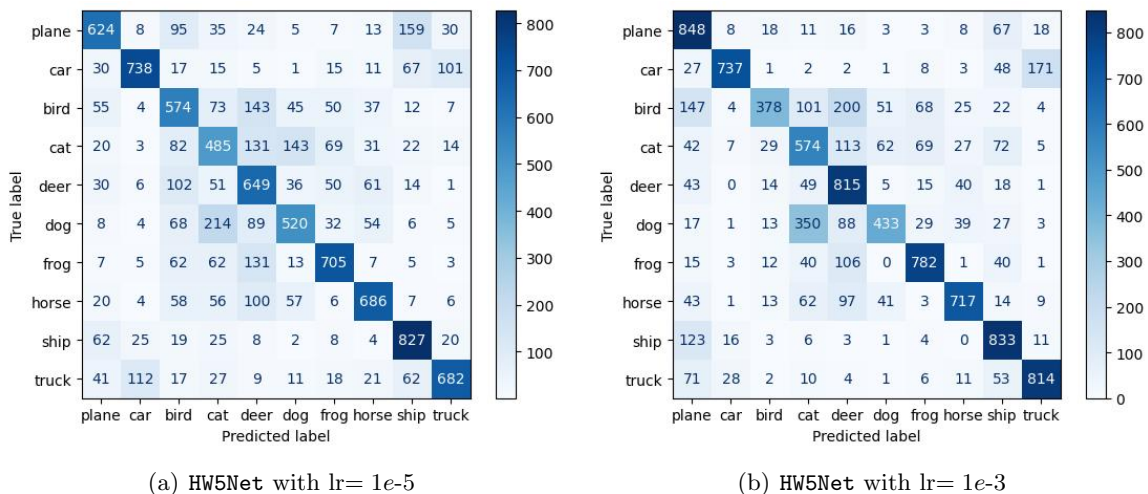


Figure 4: Confusion matrix on validation set of CIFAR-10.

The accuracy can be found in Tab. 2. HW5Net is also overfitting to the training set when the learning rate is smaller (about 85% for the training set and about 65% for the validation set). This is a similar trend when training on MS-COCO. However, it is not that obvious. When the learning rate is larger, although the accuracy on the training set is lower (about 73%), it performs better on the validation set (about 69%). Overall, HW5Net has better performance on CIFAR-10 than on MS-COCO. The reason is CIFAR-10 has more training samples, so that HW5Net obtains a better generalization ability with fewer epochs. And MS-COCO is also a multi-label dataset, so some samples with multiple labels can disturb the model learning from the dataset.

Table 2: Accuracy on CIFAR-10.

Network	Accuracy (training set) (%)	Accuracy (validation set) (%)
HW5Net with lr= 1e-5	84.95	64.90
HW5Net with lr= 1e-3	72.70	69.31

3 Source Code

```

1 import sys
2 import numpy as np
3 import argparse
4 import matplotlib.pyplot as plt
5 import torch
6 import torch.nn as nn
7 from torch.utils.data import Dataset, DataLoader
8 import torchvision
9 import torchvision.transforms as tvt
10 from sklearn.metrics import ConfusionMatrixDisplay
11
12 # MS-COCO dataset class
13 class COCODataset(Dataset):
14     def __init__(self, path, device, train=True):
15         super().__init__()
16         self.device = device
17         # load the image and category files
18         if train:
19             self.imgs = np.load('{}coco_train.npy'.format(path))
20             self.dict = np.load('{}coco_train_dict.npy'.format(path),
21                               ↪ allow_pickle=True).item()
22         else:
23             self.imgs = np.load('{}coco_val.npy'.format(path))
24             self.dict = np.load('{}coco_val_dict.npy'.format(path),
25                               ↪ allow_pickle=True).item()
26         # transform the array to tensor and normalize
27         self.xform = tvt.Compose([tvt.ToTensor(),
28                                   ↪ tvt.Normalize([0.5,0.5,0.5],\
29                                                     [0.5,0.5,0.5])])
28     def __len__(self):
29         return self.imgs.shape[0]
30     def __getitem__(self, index):
31         # one-hot encoding
32         class_enc = torch.zeros(5, device=self.device)

```

```

33         class_enc[self.dict[index][2]] = 1
34         return self.xform(self.imgs[index, :, :, :]).to(self.device),
           ↪ class_enc
35
36 # CIFAR-10 data loader, which is from DLStudio
37 def load_cifar_10_dataset(batch_size):
38     ## But then the call to Normalize() changes the range to -1.0-1.0
           ↪ float vals.
39     transform = tvn.Compose([tvn.ToTensor(),
40                             tvn.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
           ↪ 0.5))])
41     ## Define where the training and the test datasets are located:
42     train_data_loc = torchvision.datasets.CIFAR10(root='./dataset/CIFAR
           ↪ -10/', train=True, download=True, transform=transform)
43     test_data_loc = torchvision.datasets.CIFAR10(root='./dataset/CIFAR
           ↪ -10/', train=False, download=True, transform=transform)
44     ## Now create the data loaders:
45     train_data_loader = DataLoader(train_data_loc, batch_size=
           ↪ batch_size, shuffle=True, num_workers=2)
46     test_data_loader = DataLoader(test_data_loc, batch_size=batch_size,
           ↪ shuffle=False, num_workers=2)
47     return train_data_loader, test_data_loader, len(train_data_loc),
           ↪ len(test_data_loc)
48
49 # skip connection block, which is from DLStudio
50 class SkipBlock(nn.Module):
51     """
52     Class Path: DLStudio -> SkipConnections -> SkipBlock
53     """
54     def __init__(self, in_ch, out_ch, downsample=False,
           ↪ skip_connections=True):
55         super().__init__()
56         self.downsample = downsample
57         self.skip_connections = skip_connections
58         self.in_ch = in_ch
59         self.out_ch = out_ch
60         self.conv1 = nn.Conv2d(in_ch, in_ch, 3, stride=1, padding=1)
61         self.conv2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
62         self.bn1 = nn.BatchNorm2d(in_ch)
63         self.bn2 = nn.BatchNorm2d(out_ch)
64
65         self.in2out = nn.Conv2d(in_ch, out_ch, 1) ### <<<< from
           ↪ Cheng-Hao Chen
66
67         if downsample:
68             ## Setting stride to 2 and kernel_size to 1 amounts to
           ↪ retaining every
69             ## other pixel in the image --- which halves the size of
           ↪ the image:
70             self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, stride=2)
71             self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1, stride=2)
72
73     def forward(self, x):
74         identity = x

```

```

75         out = self.conv1(x)
76         out = self.bn1(out)
77         out = nn.functional.relu(out)
78         out = self.conv2(out)
79         out = self.bn2(out)
80         out = nn.functional.relu(out)
81         if self.downsample:
82             identity = self.downsampler1(identity)
83             out = self.downsampler2(out)
84         if self.skip_connections:
85             if (self.in_ch == self.out_ch) and (self.downsample is
86                 ↪ False):
87                 out = out + identity
88             elif (self.in_ch != self.out_ch) and (self.downsample is
89                 ↪ False):
90
91                 identity = self.in2out( identity ) ### <<<< from Cheng-
92                 ↪ Hao Chen
93
94                 out = out + identity
95             elif (self.in_ch != self.out_ch) and (self.downsample is
96                 ↪ True):
97                 out = out + torch.cat((identity, identity), dim=1)
98         return out
99
100 # deep neural network with skip connections
101 class HW5Net(nn.Module):
102     def __init__(self, cifar=False):
103         super().__init__()
104         # initial convolutional layer
105         self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding
106             ↪ =1)
107         # batch normalization
108         self.bn = nn.BatchNorm2d(64)
109         # ReLU activation
110         self.relu = nn.ReLU(inplace=True)
111         # max pooling and downsizing
112         self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
113         # convolutional layers with skip connection
114         self.skip64_arr = nn.ModuleList()
115         for i in range(5):
116             self.skip64_arr.append(SkipBlock(64, 64, skip_connections=
117                 ↪ True))
118         # downsampling convolutional layers with skip connection
119         self.ds1 = SkipBlock(64, 128, downsample=True, skip_connections
120             ↪ =True)
121         # convolutional layers with skip connection
122         self.skip128_arr = nn.ModuleList()
123         for i in range(5):
124             self.skip128_arr.append(SkipBlock(128, 128,
125                 ↪ skip_connections=True))
126         # downsampling convolutional layers with skip connection
127         self.ds2 = SkipBlock(128, 256, downsample=True,
128             ↪ skip_connections=True)

```

```

120     # convolutional layers with skip connection
121     self.skip256_arr = nn.ModuleList()
122     for i in range(5):
123         self.skip256_arr.append(SkipBlock(256, 256,
124             ↪ skip_connections=True))
125     # downsampling convolutional layers with skip connection
126     self.ds3 = SkipBlock(256, 512, downsample=True,
127         ↪ skip_connections=True)
128     # convolutional layers with skip connection
129     self.skip512_arr = nn.ModuleList()
130     for i in range(5):
131         self.skip512_arr.append(SkipBlock(512, 512,
132             ↪ skip_connections=True))
133     # adaptive average pooling over height and width
134     self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
135     # flatten to a vector for each sample
136     self.flatten = nn.Flatten()
137     # dropout layer to avoid over-fitting
138     self.dropout = nn.Dropout(p=0.3)
139     # linear layers
140     self.fc1 = nn.Linear(512, 100)
141     if cifar:
142         self.fc2 = nn.Linear(100, 10)
143     else:
144         self.fc2 = nn.Linear(100, 5)
145
146 def forward(self, x):
147     # feature extraction
148     x = self.conv1(x)
149     x = self.bn(x)
150     x = self.relu(x)
151     x = self.maxpool(x)
152     for skip64 in self.skip64_arr:
153         x = skip64(x)
154     x = self.ds1(x)
155     for i, skip128 in enumerate(self.skip128_arr):
156         x = skip128(x)
157     x = self.ds2(x)
158     for i, skip256 in enumerate(self.skip256_arr):
159         x = skip256(x)
160     x = self.ds3(x)
161     for i, skip512 in enumerate(self.skip512_arr):
162         x = skip512(x)
163     # classification
164     x = self.avgpool(x)
165     x = self.flatten(x)
166     x = self.dropout(x)
167     x = self.fc1(x)
168     x = self.relu(x)
169     x = self.dropout(x)
170     x = self.fc2(x)
171     return x

```

plot the learning curves

```

171 def plot_learning_curve(loss_list, save_every, net_list, cifar):
172     # calculate x-axis
173     x = len(loss_list[0])
174     x = [i*save_every for i in range(x)]
175     plt.figure(figsize=(8, 6))
176     # plot each curve
177     for i, loss in enumerate(loss_list):
178         plt.plot(x, loss, label='learning rate = {:.0e}'.format(
179             ↪ net_list[i][1]))
180     plt.xlabel('Iterations')
181     plt.ylabel('losses')
182     plt.legend()
183     cf = plt.gcf()
184     cf.savefig('loss_c_{}.jpg'.format(cifar), bbox_inches='tight', \
185         format='jpg', dpi=1000)
186
187 # plot confusion matrix
188 def plot_confusion_matrix(pred, gt, cat_list, title, train):
189     plt.figure(figsize=(4,4))
190     ConfusionMatrixDisplay.from_predictions(gt, pred,\
191         display_labels=cat_list, cmap='Blues')
192     cf = plt.gcf()
193     cf.savefig('CM_HW5Net{}_{}.jpg'.format(title, train), bbox_inches='
194         ↪ tight', format='jpg')
195
196 # train a networking
197 def train(net, train_data_loader, epochs, lr, save_every, device, cifar
198     ↪ ):
199     net = net.to(device)
200     # use cross entropy loss
201     criterion = nn.CrossEntropyLoss()
202     # use Adam optimizer
203     optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas
204         ↪ =(0.9,0.99))
205     j = 1
206     loss_all = []
207     running_loss_save = 0.0
208     for epoch in range(epochs):
209         running_loss = 0.0
210         for i, data in enumerate(train_data_loader):
211             inputs, labels = data
212             if cifar: # put to desired device if from CIFAR-10
213                 inputs = inputs.to(device)
214                 labels = labels.to(device)
215             optimizer.zero_grad()
216             # predict
217             outputs = net(inputs)
218             loss = criterion(outputs, labels)
219             if j % save_every == 0:
220                 loss_all.append(running_loss_save/save_every)
221                 running_loss_save = 0.0
222             loss.backward()
223             # update weights
224             optimizer.step()

```

```

221         running_loss_save += loss.item()
222         running_loss += loss.item()
223         if (i+1) % 100 == 0:
224             print('[epoch: %d, batch: %5d] loss: %.3f'%(epoch+1, i
                ↪ +1, running_loss/100))
225             running_loss = 0.0
226             j += 1
227     return net, loss_all
228
229 # train the three networks
230 def net_training(batch_size, save_every, epochs, device, cifar=False):
231     if cifar:
232         # create data loader
233         train_data_loader, _, _ = load_cifar_10_dataset(batch_size)
234         net_list = [(HW5Net(True), 1e-5), (HW5Net(True), 1e-3)]
235     else:
236         # create the dataset instance
237         dataset_train = COCODataset('./dataset', device, train=True)
238         # define the dataloader
239         train_data_loader = DataLoader(dataset_train, batch_size=
            ↪ batch_size, shuffle=True, drop_last=True)
240         net_list = [(HW5Net(), 1e-5), (HW5Net(), 1e-3)]
241     # train three networks in a loop
242     loss_list = []
243     for i, (net, lr) in enumerate(net_list):
244         print('Training HW5Net with learning rate of %s...'%(str(lr)))
245         num_layers = len(list(net.parameters()))
246         print('Number of learnable layers: ', num_layers)
247         net_trained, loss = train(net, train_data_loader, epochs, lr,
            ↪ save_every, device, cifar)
248         # save the trained network
249         torch.save(net_trained, './net{}.pth'.format(i+1+cifar*2))
250         loss_list.append(loss)
251     # plot learning curve
252     plot_learning_curve(loss_list, save_every, net_list, cifar)
253     np.save('loss.npy', loss_list)
254
255 # get class index from the one-hot encoding
256 def classFromTensor(tensor):
257     # find out the class index
258     index = torch.max(tensor, -1)[1]
259     return index.detach().cpu().tolist()
260
261 # evaluate the networks
262 def net_evaluation(batch_size, train, device, cifar=False):
263     if cifar:
264         # create data loader
265         if train:
266             val_data_loader, _, num_sample, _ = load_cifar_10_dataset(
                ↪ batch_size)
267         else:
268             _, val_data_loader, _, num_sample = load_cifar_10_dataset(
                ↪ batch_size)
269     else:

```



```

270         # create the dataset instance
271         dataset_val = COCODataset('./dataset', device, train=train)
272         num_sample = len(dataset_val)
273         # define the dataloader
274         val_data_loader = DataLoader(dataset_val, batch_size=batch_size
        ↪ , shuffle=False, drop_last=False)
275     for i in range(2):
276         # load the trained network
277         net = torch.load('./net{}.pth'.format(i+1+cifar*2)).to(device).
        ↪ eval()
278         gt = []
279         pred = []
280         with torch.no_grad():
281             for data in val_data_loader:
282                 inputs, labels = data
283                 if cifar:
284                     inputs = inputs.to(device)
285                     labels = labels.to(device)
286                     gt += list(labels.detach().cpu())
287                 else:
288                     gt += classFromTensor(labels)
289                 outputs = net(inputs)
290                 pred += classFromTensor(outputs)
291         gt_array = np.array(gt)
292         pred_array = np.array(pred)
293         # count the number of correct prediction
294         n_correct_pred = sum(gt_array == pred_array)
295         # calculate the accuracy
296         accuracy = n_correct_pred / num_sample
297         print('Accuracy for the %dth neural network on training set (=%
        ↪ s) is %.2f%%'%(i+1+cifar*2, str(train), accuracy*100))
298         # plot the confusion matrix
299         if cifar:
300             plot_confusion_matrix(pred, gt, ['plane', 'car', 'bird', 'cat'
        ↪ , 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'], i+1+
        ↪ cifar*2, train)
301         else:
302             plot_confusion_matrix(pred, gt, ['boat', 'cake', 'couch', 'dog
        ↪ ', 'motorcycle'], i+1, train)
303
304     if __name__ == '__main__':
305         # '1' -- train and test on my MS-COCO-based dataset
306         # '2' -- train and test on CIFAR-10 dataset
307         parser = argparse.ArgumentParser()
308         parser.add_argument('-t', '--task', type=str, default='1',\
309             help='choose a task', choices=['1', '2'])
310         parser.add_argument('--cuda', type=str, default='cuda:0', help='
        ↪ Enable cuda')
311         args = parser.parse_args()
312
313         device = torch.device(args.cuda)
314
315         if args.task == '1':
316             net_training(16, 150, 10, device)

```

```
317         net_evaluation(32, False, device)
318         net_evaluation(32, True, device)
319
320     if args.task == '2':
321         net_training(16, 150, 4, device, True)
322         net_evaluation(32, False, device, True)
323         net_evaluation(32, True, device, True)
```