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.

```
...
# skip connection block, which is from DLStudio
class SkipBlock(nn.Module):
    """
    Class Path: DLStudio -> SkipConnections -> SkipBlock
    """
    def __init__(self, in_ch, out_ch, downsample=False,
        ↪ skip_connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, in_ch, 3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(in_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)

        self.in2out  =  nn.Conv2d(in_ch, out_ch, 1) ### <<<<< from
            ↪ Cheng-Hao Chen

        if downsample:
            ## Setting stride to 2 and kernel_size to 1 amounts to
                ↪ retaining every
            ## other pixel in the image --- which halves the size of
                ↪ the image:
            self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, stride=2)
            self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1, stride=2)

```

```
26      def forward(self, x):
27          identity = x
28          out = self.convo1(x)
29          out = self.bn1(out)
30          out = nn.functional.relu(out)
31          out = self.convo2(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
                    ↪ False):
39                  out = out + identity
40              elif (self.in_ch != self.out_ch) and (self.downsample is
                    ↪ False):
41
42                  identity = self.in2out( identity ) ### <<<< from Cheng-
                        ↪ Hao Chen
43
44                  out = out + identity
45              elif (self.in_ch != self.out_ch) and (self.downsample is
                    ↪ True):
46                  out = out + torch.cat((identity, identity), dim=1)
47          return out
48
49  class HW5Net(nn.Module):
50      def __init__(self, cifar=False):
51          super().__init__()
52          # initial convolutional layer
53          self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding
                ↪ =1)
54          # batch normalization
55          self.bn = nn.BatchNorm2d(64)
56          # ReLU activation
57          self.relu = nn.ReLU(inplace=True)
58          # max pooling and downsizing
59          self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
60          # convolutional layers with skip connection
61          self.skip64_arr = nn.ModuleList()
62          for i in range(5):
63              self.skip64_arr.append(SkipBlock(64, 64, skip_connections=
                    ↪ True))
64          # downsampling convolutional layers with skip connection
65          self.ds1 = SkipBlock(64, 128, downsample=True, skip_connections
                ↪ =True)
66          # convolutional layers with skip connection
67          self.skip128_arr = nn.ModuleList()
68          for i in range(5):
69              self.skip128_arr.append(SkipBlock(128, 128,
                    ↪ skip_connections=True))
70          # downsampling convolutional layers with skip connection
71          self.ds2 = SkipBlock(128, 256, downsample=True,
```

```
            ↪ skip_connections=True)
        # convolutional layers with skip connection
        self.skip256_arr = nn.ModuleList()
        for i in range(5):
            self.skip256_arr.append(SkipBlock(256, 256,
                ↪ skip_connections=True))
        # downsampling convolutional layers with skip connection
        self.ds3 = SkipBlock(256, 512, downsample=True,
            ↪ skip_connections=True)
        # convolutional layers with skip connection
        self.skip512_arr = nn.ModuleList()
        for i in range(5):
            self.skip512_arr.append(SkipBlock(512, 512,
                ↪ skip_connections=True))
        # adaptive average pooling over height and width
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        # flatten to a vector for each sample
        self.flatten = nn.Flatten()
        # dropout layer to avoid over-fitting
        self.dropout = nn.Dropout(p=0.3)
        # linear layers
        self.fc1 =  nn.Linear(512, 100)
        if cifar:
            self.fc2 =  nn.Linear(100, 10)
        else:
            self.fc2 =  nn.Linear(100, 5)

    def forward(self, x):
        # feature extraction
        x = self.conv1(x)
        x = self.bn(x)
        x = self.relu(x)
        x = self.maxpool(x)
        for skip64 in self.skip64_arr:
            x = skip64(x)
        x = self.ds1(x)
        for i, skip128 in enumerate(self.skip128_arr):
            x = skip128(x)
        x = self.ds2(x)
        for i, skip256 in enumerate(self.skip256_arr):
            x = skip256(x)
        x = self.ds3(x)
        for i, skip512 in enumerate(self.skip512_arr):
            x = skip512(x)
        # classification
        x = self.avgpool(x)
        x = self.flatten(x)
        x = self.dropout(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        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.

```
...
# MS-COCO dataset class
class COCODataset(Dataset):
    def __init__(self, path, device, train=True):
        super().__init__()
        self.device = device
        # load the image and category files
        if train:
            self.imgs = np.load('{}/coco_train.npy'.format(path))
            self.dict = np.load('{}/coco_train_dict.npy'.format(path),
               ↪ allow_pickle=True).item()
        else:
            self.imgs = np.load('{}/coco_val.npy'.format(path))
            self.dict = np.load('{}/coco_val_dict.npy'.format(path),
               ↪ allow_pickle=True).item()
        # transform the array to tensor and normalize
        self.xform = tvt.Compose([tvt.ToTensor(),
                                  tvt.Normalize([0.5,0.5,0.5],\
                                                [0.5,0.5,0.5])])
    def __len__(self):
        return self.imgs.shape[0]
    def __getitem__(self, index):
        # one-hot encoding
        class_enc = torch.zeros(5, device=self.device)
        class_enc[self.dict[index][2]] = 1
        return self.xform(self.imgs[index, :, :, :]).to(self.device),
           ↪ class_enc

# CIFAR-10 data loader, which is from DLStudio
def load_cifar_10_dataset(batch_size):
    ## But then the call to Normalize() changes the range to -1.0-1.0
       ↪ float vals.
    transform = tvt.Compose([tvt.ToTensor(),
                             tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
                                ↪ 0.5))])
    ## Define where the training and the test datasets are located:
    train_data_loc = torchvision.datasets.CIFAR10(root='./dataset/CIFAR
       ↪ -10/', train=True, download=True, transform=transform)
    test_data_loc = torchvision.datasets.CIFAR10(root='./dataset/CIFAR
       ↪ -10/', train=False, download=True, transform=transform)
    ## 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
```

```
            outputs = net(inputs)
            loss = criterion(outputs, labels)
            if j % save_every == 0:
                loss_all.append(running_loss_save/save_every)
                running_loss_save = 0.0
            loss.backward()
            # update weights
            optimizer.step()
            running_loss_save += loss.item()
            running_loss += loss.item()
            if (i+1) % 100 == 0:
                print('[epoch: %d, batch: %5d] loss: %.3f'%(epoch+1, i
                    ↪ +1, running_loss/100))
                running_loss = 0.0
            j += 1
    return net, loss_all

# train the three networks
def net_training(batch_size, save_every, epochs, device, cifar=False):
    if cifar:
        # create data loader
        train_data_loader, _, _, _ = load_cifar_10_dataset(batch_size)
        net_list = [(HW5Net(True), 1e-5), (HW5Net(True), 1e-3)]
    else:
        # create the dataset instance
        dataset_train = COCODataset('./dataset', device, train=True)
        # define teh dataloader
        train_data_loader = DataLoader(dataset_train, batch_size=
            ↪ batch_size, shuffle=True, drop_last=True)
        net_list = [(HW5Net(), 1e-5), (HW5Net(), 1e-3)]
    # train three networks in a loop
    loss_list = []
    for i, (net, lr) in enumerate(net_list):
        print('Training HW5Net with learning rate of %s...'%(str(lr)))
        num_layers = len(list(net.parameters()))
        print('Number of learnable layers: ', num_layers)
        net_trained, loss = train(net, train_data_loader, epochs, lr,
            ↪ save_every, device, cifar)
        # save the trained network
        torch.save(net_trained, './net{}.pth'.format(i+1+cifar*2))
        loss_list.append(loss)
    # plot learning curve
    plot_learning_curve(loss_list, save_every, net_list, cifar)
    np.save('loss.npy', loss_list)

# get class index from the one-hot encoding
def classFromTensor(tensor):
    # find out the class index
    index = torch.max(tensor, -1)[1]
    return index.detach().cpu().tolist()

# evaluate the networks
def net_evaluation(batch_size, train, device, cifar=False):
    if cifar:
```

```
        # create data loader
        if train:
            val_data_loader, _, num_sample, _ = load_cifar_10_dataset(
                ↪ batch_size)
        else:
            _, val_data_loader, _, num_sample = load_cifar_10_dataset(
                ↪ batch_size)
    else:
        # create the dataset instance
        dataset_val = COCODataset('./dataset', device, train=train)
        num_sample = len(dataset_val)
        # define teh dataloader
        val_data_loader = DataLoader(dataset_val, batch_size=batch_size
            ↪ , shuffle=False, drop_last=False)
    for i in range(2):
        # load the trained network
        net = torch.load('./net{}.pth'.format(i+1+cifar*2)).to(device).
            ↪ eval()
        gt = []
        pred = []
        with torch.no_grad():
            for data in val_data_loader:
                inputs, labels = data
                if cifar:
                    inputs = inputs.to(device)
                    labels = labels.to(device)
                    gt += list(labels.detach().cpu())
                else:
                    gt += classFromTensor(labels)
                outputs = net(inputs)
                pred += classFromTensor(outputs)
        gt_array = np.array(gt)
        pred_array = np.array(pred)
        # count the number of correct prediction
        n_correct_pred = sum(gt_array == pred_array)
        # calculate the accuracy
        accuracy = n_correct_pred / num_sample
        print('Accuracy for the %dth neural network on training set (=%
            ↪ s) is %.2f%%'%(i+1+cifar*2, str(train), accuracy*100))
        # plot the confusion matrix
        if cifar:
            plot_confusion_matrix(pred, gt, ['plane','car','bird','cat'
                ↪ ,'deer','dog','frog','horse','ship','truck'], i+1+
                ↪ cifar*2, train)
        else:
            plot_confusion_matrix(pred, gt, ['boat','cake','couch','dog
                ↪ ','motorcycle'], i+1, train)
...
```

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.

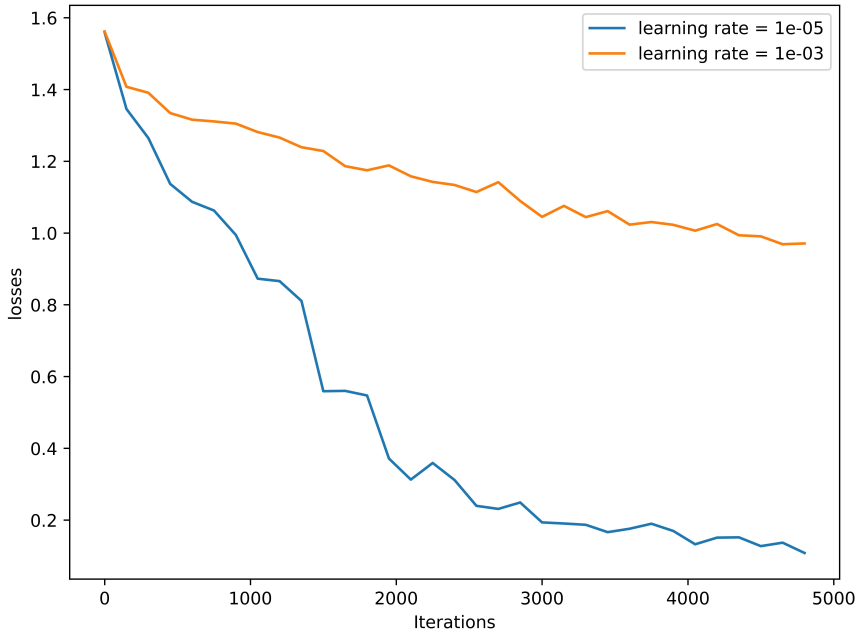
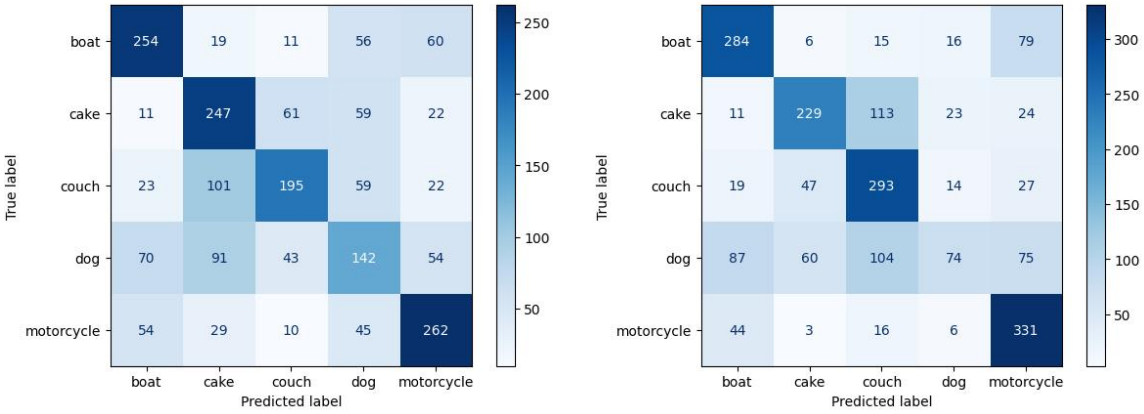


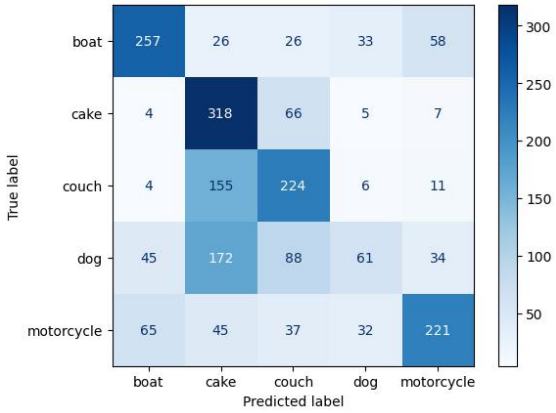
Figure 1: Learning curves over MS-COCO dataset.

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".



(a) `HW5Net` with lr= 1e-5

(b) `HW5Net` with lr= 1e-3



(c) `Net3` from HW4

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.

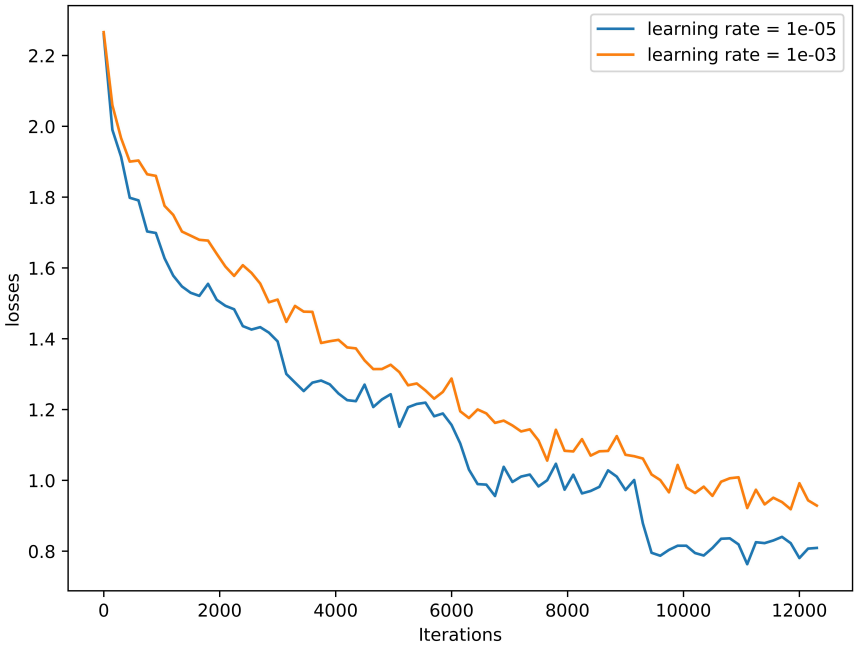
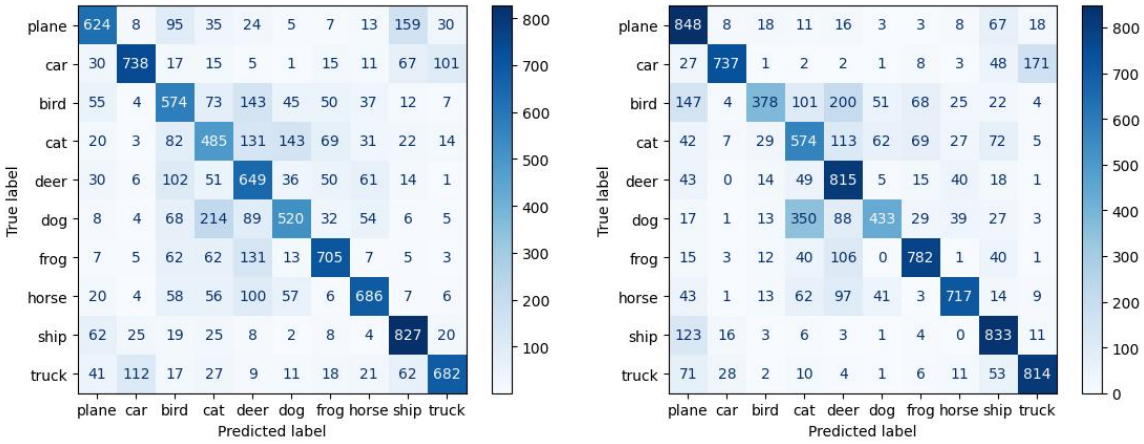


Figure 3: Learning curves over CIFAR-10 dataset.

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.



(a) `HW5Net` with lr= $1e$-5

(b) `HW5Net` with lr= $1e$-3

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

```
import sys
import numpy as np
import argparse
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import torchvision
import torchvision.transforms as tvt
from sklearn.metrics import ConfusionMatrixDisplay

# MS-COCO dataset class
class COCODataset(Dataset):
    def __init__(self, path, device, train=True):
        super().__init__()
        self.device = device
        # load the image and category files
        if train:
            self.imgs = np.load('{}/coco_train.npy'.format(path))
            self.dict = np.load('{}/coco_train_dict.npy'.format(path),
                ↪ allow_pickle=True).item()
        else:
            self.imgs = np.load('{}/coco_val.npy'.format(path))
            self.dict = np.load('{}/coco_val_dict.npy'.format(path),
                ↪ allow_pickle=True).item()
        # transform the array to tensor and normalize
        self.xform = tvt.Compose([tvt.ToTensor(),
                                  tvt.Normalize([0.5,0.5,0.5],\
                                                [0.5,0.5,0.5])])
    def __len__(self):
        return self.imgs.shape[0]
    def __getitem__(self, index):
        # one-hot encoding
        class_enc = torch.zeros(5, device=self.device)
```

```
        class_enc[self.dict[index][2]] = 1
        return self.xform(self.imgs[index, :, :, :]).to(self.device),
            ↪ class_enc

# CIFAR-10 data loader, which is from DLStudio
def load_cifar_10_dataset(batch_size):
    ## But then the call to Normalize() changes the range to -1.0-1.0
        ↪ float vals.
    transform = tvt.Compose([tvt.ToTensor(),
                             tvt.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
                                ↪ 0.5))])
    ## Define where the training and the test datasets are located:
    train_data_loc = torchvision.datasets.CIFAR10(root='./dataset/CIFAR
        ↪ -10/', train=True, download=True, transform=transform)
    test_data_loc = torchvision.datasets.CIFAR10(root='./dataset/CIFAR
        ↪ -10/', train=False, download=True, transform=transform)
    ## Now create the data loaders:
    train_data_loader = DataLoader(train_data_loc, batch_size=
        ↪ batch_size, shuffle=True, num_workers=2)
    test_data_loader = DataLoader(test_data_loc, batch_size=batch_size,
        ↪  shuffle=False, num_workers=2)
    return train_data_loader, test_data_loader, len(train_data_loc),
        ↪ len(test_data_loc)

# skip connection block, which is from DLStudio
class SkipBlock(nn.Module):
    """
    Class Path: DLStudio -> SkipConnections -> SkipBlock
    """
    def __init__(self, in_ch, out_ch, downsample=False,
        ↪ skip_connections=True):
        super().__init__()
        self.downsample = downsample
        self.skip_connections = skip_connections
        self.in_ch = in_ch
        self.out_ch = out_ch
        self.convo1 = nn.Conv2d(in_ch, in_ch, 3, stride=1, padding=1)
        self.convo2 = nn.Conv2d(in_ch, out_ch, 3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(in_ch)
        self.bn2 = nn.BatchNorm2d(out_ch)

        self.in2out  =  nn.Conv2d(in_ch, out_ch, 1) ### <<<<< from
            ↪ Cheng-Hao Chen

        if downsample:
            ## Setting stride to 2 and kernel_size to 1 amounts to
                ↪ retaining every
            ## other pixel in the image --- which halves the size of
                ↪ the image:
            self.downsampler1 = nn.Conv2d(in_ch, in_ch, 1, stride=2)
            self.downsampler2 = nn.Conv2d(out_ch, out_ch, 1, stride=2)

    def forward(self, x):
        identity = x
```

```
        out = self.convo1(x)
        out = self.bn1(out)
        out = nn.functional.relu(out)
        out = self.convo2(out)
        out = self.bn2(out)
        out = nn.functional.relu(out)
        if self.downsample:
            identity = self.downsampler1(identity)
            out = self.downsampler2(out)
        if self.skip_connections:
            if (self.in_ch == self.out_ch) and (self.downsample is
               ↪ False):
                out = out + identity
            elif (self.in_ch != self.out_ch) and (self.downsample is
               ↪ False):

                identity = self.in2out( identity ) ### <<<< from Cheng-
                   ↪ Hao Chen

                out = out + identity
            elif (self.in_ch != self.out_ch) and (self.downsample is
               ↪ True):
                out = out + torch.cat((identity, identity), dim=1)
        return out

# deep neural network with skip connections
class HW5Net(nn.Module):
    def __init__(self, cifar=False):
        super().__init__()
        # initial convolutional layer
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding
           ↪ =1)
        # batch normalization
        self.bn = nn.BatchNorm2d(64)
        # ReLU activation
        self.relu = nn.ReLU(inplace=True)
        # max pooling and downsizing
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        # convolutional layers with skip connection
        self.skip64_arr = nn.ModuleList()
        for i in range(5):
            self.skip64_arr.append(SkipBlock(64, 64, skip_connections=
               ↪ True))
        # downsampling convolutional layers with skip connection
        self.ds1 = SkipBlock(64, 128, downsample=True, skip_connections
           ↪ =True)
        # convolutional layers with skip connection
        self.skip128_arr = nn.ModuleList()
        for i in range(5):
            self.skip128_arr.append(SkipBlock(128, 128,
               ↪ skip_connections=True))
        # downsampling convolutional layers with skip connection
        self.ds2 = SkipBlock(128, 256, downsample=True,
           ↪ skip_connections=True)
```

```
        # convolutional layers with skip connection
        self.skip256_arr = nn.ModuleList()
        for i in range(5):
            self.skip256_arr.append(SkipBlock(256, 256,
               ↪ skip_connections=True))
        # downsampling convolutional layers with skip connection
        self.ds3 = SkipBlock(256, 512, downsample=True,
           ↪ skip_connections=True)
        # convolutional layers with skip connection
        self.skip512_arr = nn.ModuleList()
        for i in range(5):
            self.skip512_arr.append(SkipBlock(512, 512,
               ↪ skip_connections=True))
        # adaptive average pooling over height and width
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        # flatten to a vector for each sample
        self.flatten = nn.Flatten()
        # dropout layer to avoid over-fitting
        self.dropout = nn.Dropout(p=0.3)
        # linear layers
        self.fc1 =  nn.Linear(512, 100)
        if cifar:
            self.fc2 =  nn.Linear(100, 10)
        else:
            self.fc2 =  nn.Linear(100, 5)

    def forward(self, x):
        # feature extraction
        x = self.conv1(x)
        x = self.bn(x)
        x = self.relu(x)
        x = self.maxpool(x)
        for skip64 in self.skip64_arr:
            x = skip64(x)
        x = self.ds1(x)
        for i, skip128 in enumerate(self.skip128_arr):
            x = skip128(x)
        x = self.ds2(x)
        for i, skip256 in enumerate(self.skip256_arr):
            x = skip256(x)
        x = self.ds3(x)
        for i, skip512 in enumerate(self.skip512_arr):
            x = skip512(x)
        # classification
        x = self.avgpool(x)
        x = self.flatten(x)
        x = self.dropout(x)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        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(
            ↪ net_list[i][1]))
179     plt.xlabel('Iterations')
180     plt.ylabel('losses')
181     plt.legend()
182     cf = plt.gcf()
183     cf.savefig('loss_c_{}.jpg'.format(cifar), bbox_inches='tight', \
184                 format='jpg', dpi=1000)
185 
186 # plot confusion matrix
187 def plot_confusion_matrix(pred, gt, cat_list, title, train):
188     plt.figure(figsize=(4,4))
189     ConfusionMatrixDisplay.from_predictions(gt, pred,\
190         display_labels=cat_list, cmap='Blues')
191     cf = plt.gcf()
192     cf.savefig('CM_HW5Net{}_{}.jpg'.format(title, train), bbox_inches='
        ↪ tight', format='jpg')
193 
194 # train a networking
195 def train(net, train_data_loader, epochs, lr, save_every, device, cifar
    ↪ ):
196     net = net.to(device)
197     # use cross entropy loss
198     criterion = nn.CrossEntropyLoss()
199     # use Adam optimizer
200     optimizer = torch.optim.Adam(net.parameters(), lr=lr, betas
        ↪ =(0.9,0.99))
201     j = 1
202     loss_all = []
203     running_loss_save = 0.0
204     for epoch in range(epochs):
205         running_loss = 0.0
206         for i, data in enumerate(train_data_loader):
207             inputs, labels = data
208             if cifar: # put to desired device if from CIFAR-10
209                 inputs = inputs.to(device)
210                 labels = labels.to(device)
211             optimizer.zero_grad()
212             # predict
213             outputs = net(inputs)
214             loss = criterion(outputs, labels)
215             if j % save_every == 0:
216                 loss_all.append(running_loss_save/save_every)
217                 running_loss_save = 0.0
218             loss.backward()
219             # update weights
220             optimizer.step()
```

```
            running_loss_save += loss.item()
            running_loss += loss.item()
            if (i+1) % 100 == 0:
                print('[epoch: %d, batch: %5d] loss: %.3f'%(epoch+1, i
                    ↪ +1, running_loss/100))
                running_loss = 0.0
            j += 1
    return net, loss_all

# train the three networks
def net_training(batch_size, save_every, epochs, device, cifar=False):
    if cifar:
        # create data loader
        train_data_loader, _, _, _ = load_cifar_10_dataset(batch_size)
        net_list = [(HW5Net(True), 1e-5), (HW5Net(True), 1e-3)]
    else:
        # create the dataset instance
        dataset_train = COCODataset('./dataset', device, train=True)
        # define teh dataloader
        train_data_loader = DataLoader(dataset_train, batch_size=
            ↪ batch_size, shuffle=True, drop_last=True)
        net_list = [(HW5Net(), 1e-5), (HW5Net(), 1e-3)]
    # train three networks in a loop
    loss_list = []
    for i, (net, lr) in enumerate(net_list):
        print('Training HW5Net with learning rate of %s...'%(str(lr)))
        num_layers = len(list(net.parameters()))
        print('Number of learnable layers: ', num_layers)
        net_trained, loss = train(net, train_data_loader, epochs, lr,
            ↪ save_every, device, cifar)
        # save the trained network
        torch.save(net_trained, './net{}.pth'.format(i+1+cifar*2))
        loss_list.append(loss)
    # plot learning curve
    plot_learning_curve(loss_list, save_every, net_list, cifar)
    np.save('loss.npy', loss_list)

# get class index from the one-hot encoding
def classFromTensor(tensor):
    # find out the class index
    index = torch.max(tensor, -1)[1]
    return index.detach().cpu().tolist()

# evaluate the networks
def net_evaluation(batch_size, train, device, cifar=False):
    if cifar:
        # create data loader
        if train:
            val_data_loader, _, num_sample, _ = load_cifar_10_dataset(
                ↪ batch_size)
        else:
            _, val_data_loader, _, num_sample = load_cifar_10_dataset(
                ↪ batch_size)
    else:
```

```
270         # create the dataset instance
271         dataset_val = COCODataset('./dataset', device, train=train)
272         num_sample = len(dataset_val)
273         # define teh 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)
```