2a.

Build a Convolutional Neural Network, like what we built in lectures (without skip connections), to classify the images across all 10 classes in CIFAR 10. You need to adjust the fully connected layer at the end properly with respect to the number of output classes. Train your network for 300 epochs. Report your training time, training loss, and evaluation accuracy after 300 epochs. Analyze your results in your report and compare them against a fully connected network (homework 2) on training time, achieved accuracy, and model size.

CIFAR 10 setup:

```
import torch
import numpy as np
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler

device = torch.device('cuda:0')
print(torch.cuda.is_available())
```

```
True
```

```
# number of subprocesses to use for data loading
num_workers = 0

# how many samples per batch to load
batch_size = 20

# percentage of training set to use as validation
valid_size = 0.2

# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

# choose the training and test datasets
train_data = datasets.CIFAR10('data', train=True,
                              download=True, transform=transform)
test_data = datasets.CIFAR10('data', train=False,
                             download=True, transform=transform)

# obtain training indices that will be used for validation
num_train = len(train_data)
```

```
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders (combine dataset and sampler)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=train_sampler, num_workers=num_workers)
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
    num_workers=num_workers)

# specify the image classes
classes = ['airplane','automobile','bird','cat','deer',
               'dog','frog','horse','ship','truck']
```

```
Files already downloaded and verified
Files already downloaded and verified
```

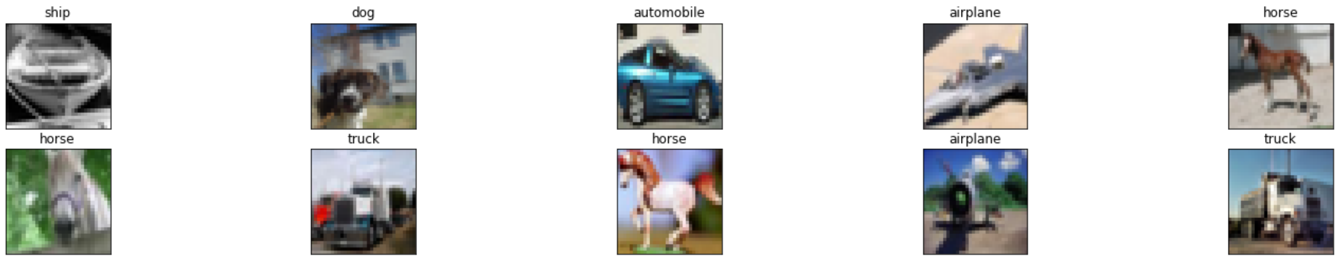
```
import matplotlib.pyplot as plt
%matplotlib inline

# helper function to un-normalize and display an image
def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image

# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))

# display 10 images
for idx in np.arange(10):
  ax = fig.add_subplot(2, 10/2, idx+1, xticks=[], yticks=[])
  imshow(images[idx])
  ax.set_title(classes[labels[idx]])
```



Model 1 Architecture:

```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net1(nn.Module):
  def __init__(self):
    super(Net1, self).__init__()
    self.conv1 = nn.Conv2d(3, 6, 5)
    self.pool = nn.MaxPool2d(2, 2)
    self.conv2 = nn.Conv2d(6, 16, 5)
    self.fc1 = nn.Linear(16 * 5 * 5, 120)
    self.fc2 = nn.Linear(120, 84)
    self.fc3 = nn.Linear(84, 10)

  def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.view(-1, 16 * 5 * 5)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x

# create a complete CNN
model1 = Net1()
if torch.cuda.is_available():
    model1.cuda()
print(model1)
```

```
Net1(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

```
import torch.optim as optim

# specify loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer = optim.SGD(model1.parameters(), lr=.01)
```

Model 1: training & testing

```
%%time
from tqdm import tqdm

# number of epochs to train the model
n_epochs = 300

#List to store loss to visualize
train_losslist = []

# track change in validation loss
valid_loss_min = np.Inf

for epoch in range(1, n_epochs+1):
  print('\nEpoch: {}'.format(epoch))
  # keep track of training and validation loss
  train_loss = 0.0
  valid_loss = 0.0

  print('training: ')
  with tqdm(train_loader, unit="batch") as tepoch:
    # train the model
    model1.train()
    for data, target in tepoch:

        # move tensors to GPU if CUDA is available
        data, target = data.to(device), target.to(device)

        # clear the gradients of all optimized variables
        optimizer.zero_grad()

        # forward pass: compute predicted outputs by passing inputs to the model
        output = model1(data)

        # calculate the batch loss
        loss = criterion(output, target)

        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()

```

```
        # perform a single optimization step (parameter update)
        optimizer.step()

        # update training loss
        train_loss += loss.item()*data.size(0)

  print('validation: ')
  with tqdm(valid_loader, unit="batch") as vepoch:
    # validate the model
    model1.eval()
    for data, target in vepoch:

        # move tensors to GPU if CUDA is available
        data, target = data.to(device), target.to(device)

        # forward pass: compute predicted outputs by passing inputs to the model
        output = model1(data)

        # calculate the batch loss
        loss = criterion(output, target)

        # update average validation loss
        valid_loss += loss.item()*data.size(0)

  # calculate average losses
  train_loss = train_loss/len(train_loader.dataset)
  valid_loss = valid_loss/len(valid_loader.dataset)
  train_losslist.append(train_loss)

  # print training/validation statistics
  print('Training Loss: {:.6f} \tValidation Loss: {:.6f}'.format(train_loss, valid_loss))

  # save model if validation loss has decreased
  if valid_loss <= valid_loss_min:
      print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
      valid_loss_min,
      valid_loss))
      torch.save(model1.state_dict(), 'model1_cifar.pt')
      valid_loss_min = valid_loss

# plt.plot(n_epochs, train_losslist)
# plt.xlabel("Epoch")
# plt.ylabel("Loss")
# plt.title("Performance of Model 1")
# plt.show()
```

```
Epoch: 1
training:
100%|██████████| 2000/2000 [00:11<00:00, 181.50batch/s]
validation:
100%|██████████| 500/500 [00:02<00:00, 223.27batch/s]
```

```
Training Loss: 1.757462         Validation Loss: 0.389919
Validation loss decreased (inf --> 0.389919).  Saving model ...

Epoch: 2
training:
100%|██████████| 2000/2000 [00:11<00:00, 176.81batch/s]
validation:
100%|██████████| 500/500 [00:02<00:00, 231.08batch/s]
Training Loss: 1.421384         Validation Loss: 0.321831
Validation loss decreased (0.389919 --> 0.321831).  Saving model ...

Epoch: 3
training:
100%|██████████| 2000/2000 [00:10<00:00, 182.56batch/s]
validation:
100%|██████████| 500/500 [00:02<00:00, 215.26batch/s]
Training Loss: 1.234788         Validation Loss: 0.288718
Validation loss decreased (0.321831 --> 0.288718).  Saving model ...

Epoch: 4
training:
100%|██████████| 2000/2000 [00:10<00:00, 181.92batch/s]
validation:
100%|██████████| 500/500 [00:02<00:00, 230.61batch/s]
Training Loss: 1.134695         Validation Loss: 0.270984
Validation loss decreased (0.288718 --> 0.270984).  Saving model ...

Epoch: 5
training:
100%|██████████| 2000/2000 [00:11<00:00, 181.71batch/s]
validation:
100%|██████████| 500/500 [00:02<00:00, 234.99batch/s]
Training Loss: 1.058859         Validation Loss: 0.255602
Validation loss decreased (0.270984 --> 0.255602).  Saving model ...

Epoch: 6
training:
100%|██████████| 2000/2000 [00:10<00:00, 184.02batch/s]
validation:
100%|██████████| 500/500 [00:02<00:00, 230.67batch/s]
Training Loss: 0.993789         Validation Loss: 0.243864
Validation loss decreased (0.255602 --> 0.243864).  Saving model ...

Epoch: 7
training:
100%|██████████| 2000/2000 [00:10<00:00, 182.31batch/s]
validation:
100%|██████████| 500/500 [00:02<00:00, 231.69batch/s]
Training Loss: 0.941940         Validation Loss: 0.231812
Validation loss decreased (0.243864 --> 0.231812).  Saving model ...
```

```
1 model1.load_state_dict(torch.load('model1_cifar.pt'))
```

```
<All keys matched successfully>
```

```
# track test loss
test_loss = 0.0
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))

model1.eval()
# iterate over test data
for data, target in test_loader:
    # move tensors to GPU if CUDA is available
    data, target = data.to(device), target.to(device)
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model1(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # update test loss
    test_loss += loss.item()*data.size(0)
    # convert output probabilities to predicted class
    _, pred = torch.max(output, 1)
    # compare predictions to true label
    correct_tensor = pred.eq(target.data.view_as(pred))
    correct = np.squeeze(correct_tensor.cpu().numpy())
    # calculate test accuracy for each object class
    for i in range(batch_size):
        label = target.data[i]
        class_correct[label] += correct[i].item()
        class_total[label] += 1

# average test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))

for i in range(10):
    if class_total[i] > 0:
        print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
            classes[i], 100 * class_correct[i] / class_total[i],
            np.sum(class_correct[i]), np.sum(class_total[i])))
    else:
        print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))

print('\nTest Accuracy (Overall): %2d%% (%2d/%2d)' % (
    100. * np.sum(class_correct) / np.sum(class_total),
    np.sum(class_correct), np.sum(class_total)))
```

```
Test Loss: 1.030446

Test Accuracy of airplane: 73% (732/1000)
Test Accuracy of automobile: 75% (752/1000)
Test Accuracy of  bird: 45% (457/1000)
Test Accuracy of   cat: 40% (402/1000)
Test Accuracy of  deer: 60% (601/1000)
```

```
Test Accuracy of   dog: 67% (670/1000)
Test Accuracy of  frog: 65% (656/1000)
Test Accuracy of horse: 75% (754/1000)
Test Accuracy of  ship: 76% (767/1000)
Test Accuracy of truck: 71% (712/1000)

Test Accuracy (Overall): 65% (6503/10000)
```

2b.

Extend your CNN by adding one more additional convolution layer followed by an activation function and pooling function. You also need to adjust your fully connected layer properly with respect to intermediate feature dimensions. Train your network for 300 epochs. Report your training time, loss, and evaluation accuracy after 300 epochs. Analyze your results in your report and compare your model size and accuracy over the baseline implementation in Problem1.a. Do you see any over-fitting? Make sure to submit your code by providing the GitHub URL of your course repository for this course.

Model 2 Architecture:

[ ] ↳ *2 cells hidden*

Model 2: training & testing

[ ] ↳ *3 cells hidden*