STAT 479: Deep Learning (Spring 2019)
Instructor: Sebastian Raschka (sraschka@wisc.edu)
Course website: http://pages.stat.wisc.edu/~sraschka/teaching/stat479-ss2019/
GitHub repository: https://github.com/rasbt/stat479-deep-learning-ss19

Homework 4: Implementing a Convolutional Neural Network

In this 4th homework, your task is to implement a convolutional neural network for classifying ima
(https://www.cs.toronto.edu/~kriz/cifar.html).

Dataset Overview

- The CIFAR-10 dataset contains 60,000 color images with pixel dimensions 32x32.
- There are 50,000 training images and 10,000 test images
- Shown below is a snapshot showing a random selection for the 10 different object classes (
  https://www.cs.toronto.edu/~kriz/cifar.html):

The CIFAR-10 dataset is already made accessible via the PyTorch API as it is a common dataset fo
you do not have to download the dataset manually -- it will be downloaded automatically when you

```
train_dataset = datasets.CIFAR10(root='data',
                                 train=True,
                                 transform=transforms.ToTensor(),
                                 download=True)
```

in the provided code cells below for the first time. Thus, keep in mind that calling this function for t
your internet connection. On a conventional internet connection, it should be downloaded in a mat

Note that we are **not** using a separate validation dataset in this homework for tuning this network.
simplicity. However, in a real-world application, you are highly advised to use a validation dataset t
as discussed in class.

Your Tasks

Your main task is to implement a simple convolutional neural network that is loosely inspired by th
competition in 2012:

- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolu
  information processing systems (pp. 1097-1105).

Then, you will make several simple modifications to this network architecture to improve its perfor

Note that in this homework, as explained above, you will NOT be working with ImageNet but CIFAR for you to be able to train the network in a timely manner.

In particular, you will be asked to first implement a basic convolutional neural network based on Al to optimize the performance and reduce overfitting. These "improvements" include Dropout, Batch serve as a good exercise for familiarizing yourself with "Deep Learning Tricks" as well as convoluti

Note that the homework is relatively easy and straightforward, but the training of the network in ea on a GPU. On a CPU, it will probably be much longer. Because training on the CPU will take much l your computer to overheat, I **highly recommend running this homework on a cloud server**, for exa a GPU for free). Since you don't have to download the dataset manually, it should be relatively strai Colab and then download the solution for submission via Canvas. Please let me know if you have computing refresher, please see: https://github.com/rasbt/stat479-deep-learning-ss19/tree/maste

The due date for this homework is Friday, 11 April 11:59 pm. Please start as soon as possible, be straightforward, you have to factor in the computation time (~30 min runtime for the complete not

Imports

```
################################
### NO NEED TO CHANGE THIS CELL
################################

import os
import time

import numpy as np
import pandas as pd

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader

from torchvision import datasets
from torchvision import transforms

import matplotlib.pyplot as plt
from PIL import Image


if torch.cuda.is_available():
    torch.backends.cudnn.deterministic = True


!pip install watermark
%load_ext watermark
%watermark -a 'Sebastian Raschka' -ud -iv
```

```
Collecting watermark
  Downloading https://files.pythonhosted.org/packages/60/fe/3ed83b6122e70dce6f
Requirement already satisfied: ipython in /usr/local/lib/python3.6/dist-packag
Requirement already satisfied: pygments in /usr/local/lib/python3.6/dist-packa
Requirement already satisfied: pickleshare in /usr/local/lib/python3.6/dist-pa
Requirement already satisfied: prompt-toolkit<2.0.0,>=1.0.4 in /usr/local/lib/
Requirement already satisfied: pexpect; sys_platform != "win32" in /usr/local/
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.6/di
Requirement already satisfied: simplegeneric>0.8 in /usr/local/lib/python3.6/d
Requirement already satisfied: decorator in /usr/local/lib/python3.6/dist-pack
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.6/dist
Requirement already satisfied: wcwidth in /usr/local/lib/python3.6/dist-packag
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.6/dist-pac
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.6/dis
Requirement already satisfied: ipython-genutils in /usr/local/lib/python3.6/di
Installing collected packages: watermark
Successfully installed watermark-2.0.2
Sebastian Raschka
last updated: 2019-11-23
```

Model Settings

```
##################################
### NO NEED TO CHANGE THIS CELL
##################################


#--------------------------
### SETTINGS
#--------------------------

# Hyperparameters
RANDOM_SEED = 1
LEARNING_RATE = 0.001
BATCH_SIZE = 256
NUM_EPOCHS = 20

# Architecture
NUM_FEATURES = 32*32
NUM_CLASSES = 10

# Other
DEVICE = "cuda:0"
```

The following code cell that implements the ResNet-34 architecture is a derivative of the code pro
https://pytorch.org/docs/0.4.0/_modules/torchvision/models/resnet.html.

Dataset

```
##############################
### NO NEED TO CHANGE THIS CELL
##############################

#--------------------------
### CIFAR-10 Dataset
#--------------------------


# Note transforms.ToTensor() scales input images
# to 0-1 range
train_dataset = datasets.CIFAR10(root='data',
                                 train=True,
                                 transform=transforms.ToTensor(),
                                 download=True)

test_dataset = datasets.CIFAR10(root='data',
                                train=False,
                                transform=transforms.ToTensor())


train_loader = DataLoader(dataset=train_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=8,
                          shuffle=True)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=BATCH_SIZE,
                         num_workers=8,
                         shuffle=False)

# Checking the dataset
for images, labels in train_loader:
    print('Image batch dimensions:', images.shape)
    print('Image label dimensions:', labels.shape)
    break

# Checking the dataset
for images, labels in train_loader:
    print('Image batch dimensions:', images.shape)
    print('Image label dimensions:', labels.shape)
    break
```

```
  0%|          | 0/170498071 [00:00<?, ?it/s]Downloading https://www.cs.toront
170500096it [00:05, 33795332.13it/s]
Extracting data/cifar-10-python.tar.gz to data
Image batch dimensions: torch.Size([256, 3, 32, 32])
Image label dimensions: torch.Size([256])
Image batch dimensions: torch.Size([256, 3, 32, 32])
Image label dimensions: torch.Size([256])
```

```
##############################
### NO NEED TO CHANGE THIS CELL
##############################
```

```
##################################

def compute_epoch_loss(model, data_loader):
    model.eval()
    curr_loss, num_examples = 0., 0
    with torch.no_grad():
        for features, targets in data_loader:
            features = features.to(DEVICE)
            targets = targets.to(DEVICE)
            logits, probas = model(features)
            loss = F.cross_entropy(logits, targets, reduction='sum')
            num_examples += targets.size(0)
            curr_loss += loss

        curr_loss = curr_loss / num_examples
        return curr_loss


def compute_accuracy(model, data_loader, device):
    model.eval()
    correct_pred, num_examples = 0, 0
    for i, (features, targets) in enumerate(data_loader):

        features = features.to(device)
        targets = targets.to(device)

        logits, probas = model(features)
        _, predicted_labels = torch.max(probas, 1)
        num_examples += targets.size(0)
        correct_pred += (predicted_labels == targets).sum()
    return correct_pred.float()/num_examples * 100
```

1) Implement a Convolutional Neural Network

In this part, you will be implementing the AlexNet-variant that you will be using and modifying throu
be a bit more "hands-off" than usual, so that you get a chance to practice implementing neural net
short descriptions (which is a useful real-world skill as it is quite common to reimplement architec
and compare those architectures to your own methods).

The architecture is as follows:

Note that I made this network based on AlexNet, as mentioned in the introduction, but there are sc
hidden layers in total: 5 convolutional layers and 2 fully-connected layers. There is one output laye
classes. For this network,

- all hidden layers are connected via ReLU activation functions
- the output layer uses a softmax activation function

- make sure you return the logits and the softmax output; the logits are used for computing th
  softmax outputs) for numerical stability reasons as discussed in earlier lectures.

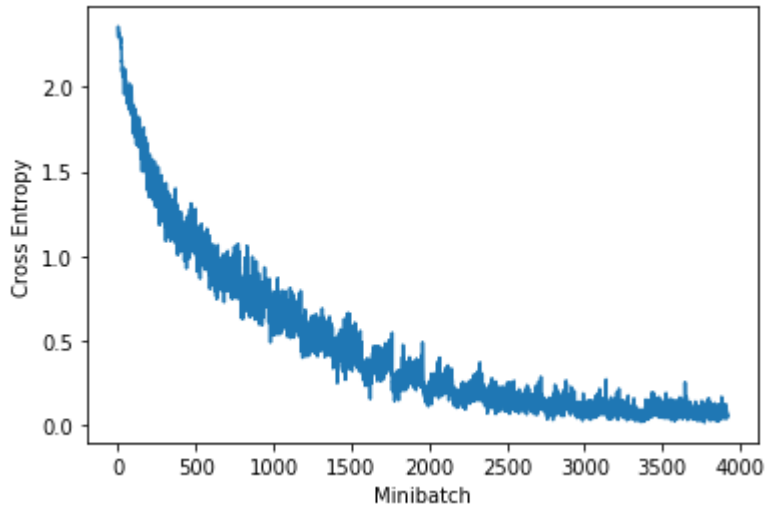
```
##########################
### MODEL
##########################

class ConvNet1(nn.Module):

    def __init__(self, num_classes=10):
        super(ConvNet1, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=5, stride=1, padding=2)
        self.mp1 = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(64, 192, kernel_size=5, stride=1, padding=2)
        self.mp2 = nn.MaxPool2d(2, 2)

        self.conv3 = nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1)
        self.mp3 = nn.MaxPool2d(2, 2)

        self.conv4 = nn.Conv2d(384, 256, kernel_size=5, stride=1, padding=2)
        self.mp4 = nn.MaxPool2d(2, 2)

        self.conv5 = nn.Conv2d(256, 256, kernel_size=5, stride=1, padding=2)
        self.mp5 = nn.MaxPool2d(2, 2)

        self.linear1 = nn.Linear(256, 4096)
        self.linear2 = nn.Linear(4096, 4096)
        self.linear3 = nn.Linear(4096, num_classes)

    def forward(self, x):

        # ... IMPLEMENT FORWARD PASS ...
        out = self.conv1(x)
        out = self.mp1(out)
        out = F.relu(out)

        out = self.conv2(out)
        out = self.mp2(out)
        out = F.relu(out)

        out = self.conv3(out)
        out = self.mp3(out)
        out = F.relu(out)

        out = self.conv4(out)
        out = self.mp4(out)
        out = F.relu(out)

        out = self.conv5(out)
        out = self.mp5(out)
        out = F.relu(out)
```

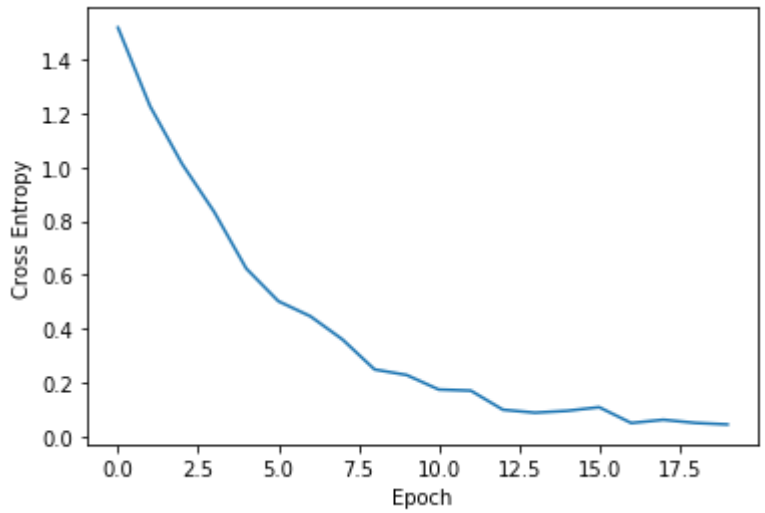
```
        out = self.linear1(out.view(-1, 256))
        out = F.relu(out)

        out = self.linear2(out)
        out = F.relu(out)

        logits = self.linear3(out)
        probas = F.softmax(logits, dim=1)
        return logits, probas


torch.manual_seed(RANDOM_SEED)

model1 = ConvNet1(NUM_CLASSES)
model1.to(DEVICE)

optimizer = torch.optim.Adam(model1.parameters(), lr=LEARNING_RATE)


##############################
### NO NEED TO CHANGE THIS CELL
##############################

def train(model, train_loader, test_loader):

    minibatch_cost, epoch_cost = [], []
    start_time = time.time()
    for epoch in range(NUM_EPOCHS):

        model.train()
        for batch_idx, (features, targets) in enumerate(train_loader):

            features = features.to(DEVICE)
            targets = targets.to(DEVICE)

            ### FORWARD AND BACK PROP
            logits, probas = model(features)
            cost = F.cross_entropy(logits, targets)
            optimizer.zero_grad()

            cost.backward()
            minibatch_cost.append(cost)

            ### UPDATE MODEL PARAMETERS
            optimizer.step()

            ### LOGGING
            if not batch_idx % 150:
                print ('Epoch: %03d/%03d | Batch %04d/%04d | Cost: %.4f'
                       %(epoch+1, NUM_EPOCHS, batch_idx,
                         len(train_loader), cost))


        with torch.set_grad_enabled(False): # save memory during inference
            print('Epoch: %03d/%03d | Train: %.3f%%' % (
```

```
                  epoch+1, NUM_EPOCHS,
                  compute_accuracy(model, train_loader, device=DEVICE)))

            cost = compute_epoch_loss(model, train_loader)
            epoch_cost.append(cost)

        print('Time elapsed: %.2f min' % ((time.time() - start_time)/60))

    print('Total Training Time: %.2f min' % ((time.time() - start_time)/60))


    with torch.set_grad_enabled(False): # save memory during inference
        print('Test accuracy: %.2f%%' % (compute_accuracy(model, test_loader, devic

    print('Total Time: %.2f min' % ((time.time() - start_time)/60))

    return minibatch_cost, epoch_cost


minibatch_cost, epoch_cost = train(model1, train_loader, test_loader)


plt.plot(range(len(minibatch_cost)), minibatch_cost)
plt.ylabel('Cross Entropy')
plt.xlabel('Minibatch')
plt.show()

plt.plot(range(len(epoch_cost)), epoch_cost)
plt.ylabel('Cross Entropy')
plt.xlabel('Epoch')
plt.show()
```

```
Epoch: 001/020 | Batch 0000/0196 | Cost: 2.3027
Epoch: 001/020 | Batch 0150/0196 | Cost: 1.6785
Epoch: 001/020 | Train: 42.008%
Time elapsed: 1.24 min
Epoch: 002/020 | Batch 0000/0196 | Cost: 1.5909
Epoch: 002/020 | Batch 0150/0196 | Cost: 1.2254
Epoch: 002/020 | Train: 55.926%
Time elapsed: 2.49 min
Epoch: 003/020 | Batch 0000/0196 | Cost: 1.2151
Epoch: 003/020 | Batch 0150/0196 | Cost: 0.9628
Epoch: 003/020 | Train: 63.860%
Time elapsed: 3.75 min
Epoch: 004/020 | Batch 0000/0196 | Cost: 0.9971
Epoch: 004/020 | Batch 0150/0196 | Cost: 0.7793
Epoch: 004/020 | Train: 70.854%
Time elapsed: 5.00 min
Epoch: 005/020 | Batch 0000/0196 | Cost: 0.9405
Epoch: 005/020 | Batch 0150/0196 | Cost: 0.8329
Epoch: 005/020 | Train: 78.940%
Time elapsed: 6.25 min
Epoch: 006/020 | Batch 0000/0196 | Cost: 0.4898
Epoch: 006/020 | Batch 0150/0196 | Cost: 0.5188
Epoch: 006/020 | Train: 82.872%
Time elapsed: 7.50 min
Epoch: 007/020 | Batch 0000/0196 | Cost: 0.5261
Epoch: 007/020 | Batch 0150/0196 | Cost: 0.4790
Epoch: 007/020 | Train: 84.206%
Time elapsed: 8.75 min
Epoch: 008/020 | Batch 0000/0196 | Cost: 0.5149
Epoch: 008/020 | Batch 0150/0196 | Cost: 0.4902
Epoch: 008/020 | Train: 87.634%
Time elapsed: 10.00 min
Epoch: 009/020 | Batch 0000/0196 | Cost: 0.4279
Epoch: 009/020 | Batch 0150/0196 | Cost: 0.4846
Epoch: 009/020 | Train: 92.074%
Time elapsed: 11.25 min
Epoch: 010/020 | Batch 0000/0196 | Cost: 0.2051
Epoch: 010/020 | Batch 0150/0196 | Cost: 0.2524
Epoch: 010/020 | Train: 92.306%
Time elapsed: 12.50 min
Epoch: 011/020 | Batch 0000/0196 | Cost: 0.2470
Epoch: 011/020 | Batch 0150/0196 | Cost: 0.2353
Epoch: 011/020 | Train: 94.086%
Time elapsed: 13.76 min
Epoch: 012/020 | Batch 0000/0196 | Cost: 0.2061
Epoch: 012/020 | Batch 0150/0196 | Cost: 0.2429
Epoch: 012/020 | Train: 94.148%
Time elapsed: 15.00 min
Epoch: 013/020 | Batch 0000/0196 | Cost: 0.2141
Epoch: 013/020 | Batch 0150/0196 | Cost: 0.2450
Epoch: 013/020 | Train: 96.876%
Time elapsed: 16.25 min
Epoch: 014/020 | Batch 0000/0196 | Cost: 0.1032
Epoch: 014/020 | Batch 0150/0196 | Cost: 0.1210
Epoch: 014/020 | Train: 97.052%
Time elapsed: 17.50 min
Epoch: 015/020 | Batch 0000/0196 | Cost: 0.0971
Epoch: 015/020 | Batch 0150/0196 | Cost: 0.1026
Epoch: 015/020 | Train: 96.848%
Time elapsed: 18.75 min
Epoch: 016/020 | Batch 0000/0196 | Cost: 0.0886
```

```
Epoch: 016/020 | Batch 0150/0196 | Cost: 0.0639
Epoch: 016/020 | Train: 96.396%
Time elapsed: 20.00 min
Epoch: 017/020 | Batch 0000/0196 | Cost: 0.0926
Epoch: 017/020 | Batch 0150/0196 | Cost: 0.0776
Epoch: 017/020 | Train: 98.384%
Time elapsed: 21.24 min
Epoch: 018/020 | Batch 0000/0196 | Cost: 0.0786
Epoch: 018/020 | Batch 0150/0196 | Cost: 0.0761
Epoch: 018/020 | Train: 97.942%
Time elapsed: 22.49 min
Epoch: 019/020 | Batch 0000/0196 | Cost: 0.0819
Epoch: 019/020 | Batch 0150/0196 | Cost: 0.1381
Epoch: 019/020 | Train: 98.300%
Time elapsed: 23.73 min
Epoch: 020/020 | Batch 0000/0196 | Cost: 0.0547
Epoch: 020/020 | Batch 0150/0196 | Cost: 0.0443
Epoch: 020/020 | Train: 98.536%
Time elapsed: 24.97 min
Total Training Time: 24.97 min
Test accuracy: 71.85%
Total Time: 25.03 min
```






```
del model1  # to save memory if you don't use it anymore
```

2) Adding Dropout

In this second part, your task is now to add dropout layers to reduce overfitting. You can copy&pas
appropriate modifications. In particular,

- place a Dropout2d (this is also referred to as "spatial dropout"; will be explained in the lecture
probability p=0.2,
- place a regular dropout after each fully connected layer with probability p=0.5, except for the

The architecture is as follows (changes, compared to the previous section, are highlighted in red):

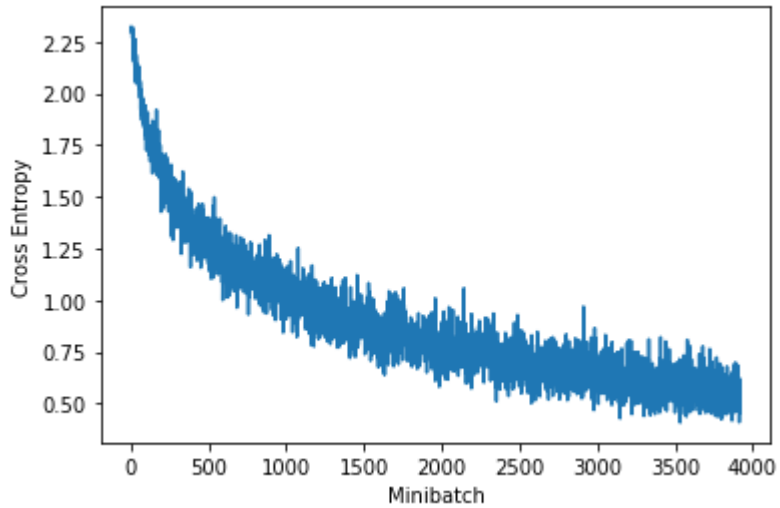
```
#########################
### MODEL
#########################

class ConvNet2(nn.Module):

    def __init__(self, num_classes=10):
        super(ConvNet2, self).__init__()

        self.conv1 =  nn.Conv2d(3, 64,   kernel_size=5, stride=1, padding=2)
        self.dp1=nn.Dropout2d(p=0.2)
        self.mp1=nn.MaxPool2d(2, 2)

        self.conv2 =  nn.Conv2d(64, 192,    kernel_size=5, stride=1, padding=2)
        self.dp2=nn.Dropout2d(p=0.2)
        self.mp2=nn.MaxPool2d(2, 2)

        self.conv3 =  nn.Conv2d(192, 384,    kernel_size=3, stride=1, padding=1)
        self.dp3=nn.Dropout2d(p=0.2)
        self.mp3=nn.MaxPool2d(2, 2)

        self.conv4 =  nn.Conv2d(384, 256,    kernel_size=5, stride=1, padding=2)
        self.dp4=nn.Dropout2d(p=0.2)
        self.mp4=nn.MaxPool2d(2, 2)

        self.conv5 =  nn.Conv2d(256, 256,    kernel_size=5, stride=1, padding=2)
        self.dp5=nn.Dropout2d(p=0.2)
        self.mp5=nn.MaxPool2d(2, 2)

        self.linear1 = nn.Linear(256, 4096)
        self.dp6=nn.Dropout(p=0.5)
        self.linear2 = nn.Linear(4096, 4096)
        self.dp7=nn.Dropout(p=0.5)

        self.linear3 = nn.Linear(4096, num_classes)


    def forward(self, x):
```

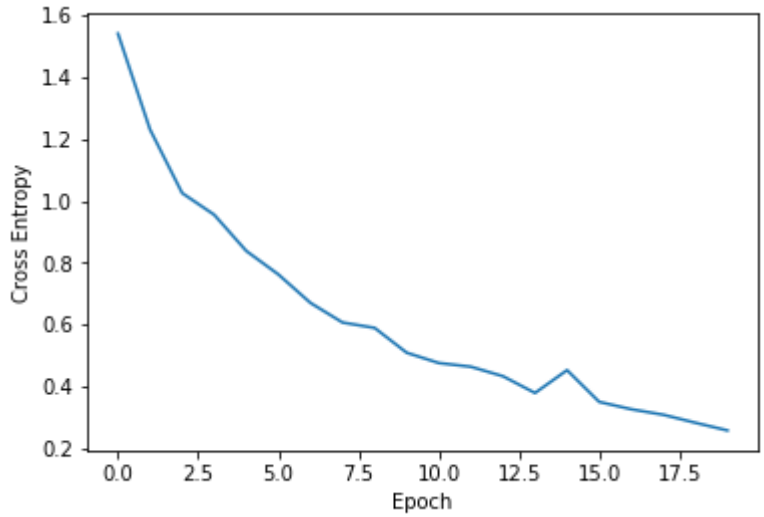
```
        out = self.conv1(x)
        out = self.dp1(out)
        out = self.mp1(out)
        out = F.relu(out)

        out = self.conv2(out)
        out = self.dp2(out)
        out = self.mp2(out)
        out = F.relu(out)

        out = self.conv3(out)
        out = self.dp3(out)
        out = self.mp3(out)
        out = F.relu(out)

        out = self.conv4(out)
        out = self.dp4(out)
        out = self.mp4(out)
        out = F.relu(out)

        out = self.conv5(out)
        out = self.dp5(out)
        out = self.mp5(out)
        out = F.relu(out)

        out = self.linear1(out.view(-1, 256))
        out = self.dp6(out)
        out = F.relu(out)

        out = self.linear2(out)
        out = self.dp7(out)
        out = F.relu(out)

        logits = self.linear3(out)
        probas = F.softmax(logits, dim=1)
        return logits, probas


torch.manual_seed(RANDOM_SEED)

model2 = ConvNet2(NUM_CLASSES)
model2.to(DEVICE)

optimizer = torch.optim.Adam(model2.parameters(), lr=LEARNING_RATE)

minibatch_cost, epoch_cost = train(model2, train_loader, test_loader)


plt.plot(range(len(minibatch_cost)), minibatch_cost)
plt.ylabel('Cross Entropy')
plt.xlabel('Minibatch')
plt.show()

plt.plot(range(len(epoch_cost)), epoch_cost)
plt.ylabel('Cross Entropy')
```

```
plt.xlabel('Epoch')
plt.show()
```

```
Epoch: 001/020 | Batch 0000/0196 | Cost: 2.3029
Epoch: 001/020 | Batch 0150/0196 | Cost: 1.7172
Epoch: 001/020 | Train: 40.548%
Time elapsed: 1.26 min
Epoch: 002/020 | Batch 0000/0196 | Cost: 1.6736
Epoch: 002/020 | Batch 0150/0196 | Cost: 1.4108
Epoch: 002/020 | Train: 54.966%
Time elapsed: 2.52 min
Epoch: 003/020 | Batch 0000/0196 | Cost: 1.3379
Epoch: 003/020 | Batch 0150/0196 | Cost: 1.2590
Epoch: 003/020 | Train: 63.066%
Time elapsed: 3.78 min
Epoch: 004/020 | Batch 0000/0196 | Cost: 1.0861
Epoch: 004/020 | Batch 0150/0196 | Cost: 1.0414
Epoch: 004/020 | Train: 66.288%
Time elapsed: 5.05 min
Epoch: 005/020 | Batch 0000/0196 | Cost: 1.1922
Epoch: 005/020 | Batch 0150/0196 | Cost: 1.0304
Epoch: 005/020 | Train: 70.844%
Time elapsed: 6.31 min
Epoch: 006/020 | Batch 0000/0196 | Cost: 0.8433
Epoch: 006/020 | Batch 0150/0196 | Cost: 0.9639
Epoch: 006/020 | Train: 73.444%
Time elapsed: 7.57 min
Epoch: 007/020 | Batch 0000/0196 | Cost: 1.0072
Epoch: 007/020 | Batch 0150/0196 | Cost: 0.8604
Epoch: 007/020 | Train: 76.868%
Time elapsed: 8.84 min
Epoch: 008/020 | Batch 0000/0196 | Cost: 0.9673
Epoch: 008/020 | Batch 0150/0196 | Cost: 0.8145
Epoch: 008/020 | Train: 79.522%
Time elapsed: 10.10 min
Epoch: 009/020 | Batch 0000/0196 | Cost: 0.9576
Epoch: 009/020 | Batch 0150/0196 | Cost: 0.8055
Epoch: 009/020 | Train: 80.014%
Time elapsed: 11.36 min
Epoch: 010/020 | Batch 0000/0196 | Cost: 0.7813
Epoch: 010/020 | Batch 0150/0196 | Cost: 0.7934
Epoch: 010/020 | Train: 82.626%
Time elapsed: 12.63 min
Epoch: 011/020 | Batch 0000/0196 | Cost: 0.8335
Epoch: 011/020 | Batch 0150/0196 | Cost: 0.7808
Epoch: 011/020 | Train: 84.088%
Time elapsed: 13.89 min
Epoch: 012/020 | Batch 0000/0196 | Cost: 0.8884
Epoch: 012/020 | Batch 0150/0196 | Cost: 0.8172
Epoch: 012/020 | Train: 83.978%
Time elapsed: 15.15 min
Epoch: 013/020 | Batch 0000/0196 | Cost: 0.6461
Epoch: 013/020 | Batch 0150/0196 | Cost: 0.7928
Epoch: 013/020 | Train: 85.252%
Time elapsed: 16.41 min
Epoch: 014/020 | Batch 0000/0196 | Cost: 0.6717
Epoch: 014/020 | Batch 0150/0196 | Cost: 0.7067
Epoch: 014/020 | Train: 87.346%
Time elapsed: 17.68 min
Epoch: 015/020 | Batch 0000/0196 | Cost: 0.6962
Epoch: 015/020 | Batch 0150/0196 | Cost: 0.6285
Epoch: 015/020 | Train: 84.450%
Time elapsed: 18.94 min
Epoch: 016/020 | Batch 0000/0196 | Cost: 0.6454
```

```
Epoch: 016/020 | Batch 0150/0196 | Cost: 0.5211
Epoch: 016/020 | Train: 89.042%
Time elapsed: 20.20 min
Epoch: 017/020 | Batch 0000/0196 | Cost: 0.6229
Epoch: 017/020 | Batch 0150/0196 | Cost: 0.5803
Epoch: 017/020 | Train: 89.394%
Time elapsed: 21.46 min
Epoch: 018/020 | Batch 0000/0196 | Cost: 0.6462
Epoch: 018/020 | Batch 0150/0196 | Cost: 0.5123
Epoch: 018/020 | Train: 89.984%
Time elapsed: 22.73 min
Epoch: 019/020 | Batch 0000/0196 | Cost: 0.7163
Epoch: 019/020 | Batch 0150/0196 | Cost: 0.7771
Epoch: 019/020 | Train: 91.192%
Time elapsed: 23.99 min
Epoch: 020/020 | Batch 0000/0196 | Cost: 0.4892
Epoch: 020/020 | Batch 0150/0196 | Cost: 0.5834
Epoch: 020/020 | Train: 91.830%
Time elapsed: 25.25 min
Total Training Time: 25.25 min
Test accuracy: 76.20%
Total Time: 25.31 min
```






```
del model2  # to save memory if you don't use it anymore
```

3) Add BatchNorm

In this 3rd part, you are now going to add BatchNorm layers to further improve the performance of convolutional layers and BatchNorm1D for the fully connected layers.

The architecture is as follows (changes, compared to the previous section, are highlighted in red):

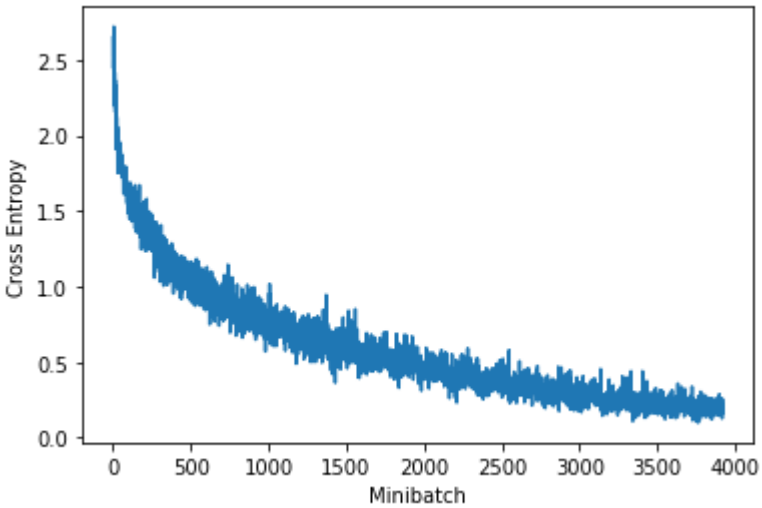
```
#########################
### MODEL
#########################

class ConvNet3(nn.Module):

    def __init__(self, num_classes=10):
      super(ConvNet3, self).__init__()

      self.conv1 =  nn.Conv2d(3, 64,   kernel_size=5, stride=1, padding=2)
      self.conv1_bn=torch.nn.BatchNorm2d(64)
      self.dp1=nn.Dropout2d(p=0.2)
      self.mp1=nn.MaxPool2d(2, 2)

      self.conv2 =  nn.Conv2d(64, 192,    kernel_size=5, stride=1, padding=2)
      self.conv2_bn=torch.nn.BatchNorm2d(192)
      self.dp2=nn.Dropout2d(p=0.2)
      self.mp2=nn.MaxPool2d(2, 2)

      self.conv3 =  nn.Conv2d(192, 384,    kernel_size=3, stride=1, padding=1)
      self.conv3_bn=torch.nn.BatchNorm2d(384)
      self.dp3=nn.Dropout2d(p=0.2)
      self.mp3=nn.MaxPool2d(2, 2)

      self.conv4 =  nn.Conv2d(384, 256,    kernel_size=5, stride=1, padding=2)
      self.conv4_bn=torch.nn.BatchNorm2d(256)
      self.dp4=nn.Dropout2d(p=0.2)
      self.mp4=nn.MaxPool2d(2, 2)

      self.conv5 =  nn.Conv2d(256, 256,    kernel_size=5, stride=1, padding=2)
      self.conv5_bn=torch.nn.BatchNorm2d(256)
      self.dp5=nn.Dropout2d(p=0.2)
      self.mp5=nn.MaxPool2d(2, 2)

      self.linear1 = nn.Linear(256, 4096)
      self.linear1_bn=torch.nn.BatchNorm1d(4096)
      self.dp6=nn.Dropout(p=0.5)

      self.linear2 = nn.Linear(4096, 4096)
      self.linear2_bn=torch.nn.BatchNorm1d(4096)
      self.dp7=nn.Dropout(p=0.5)

      self.linear3 = nn.Linear(4096, num_classes)
```

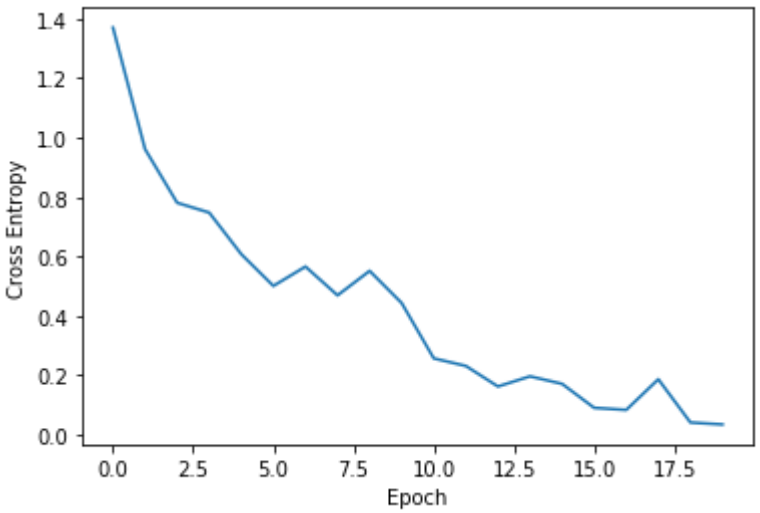
```
    def forward(self, x):
      out = self.conv1(x)
      out = self.conv1_bn(out)
      out=self.dp1(out)
      out=  self.mp1(out)
      out = F.relu(out)

      out = self.conv2(out)
      out = self.conv2_bn(out)
      out=self.dp2(out)
      out=  self.mp2(out)
      out = F.relu(out)

      out = self.conv3(out)
      out = self.conv3_bn(out)
      out=self.dp3(out)
      out=  self.mp3(out)
      out = F.relu(out)

      out = self.conv4(out)
      out = self.conv4_bn(out)
      out=self.dp4(out)
      out=  self.mp4(out)
      out = F.relu(out)

      out = self.conv5(out)
      out = self.conv5_bn(out)
      out=self.dp5(out)
      out=  self.mp5(out)
      out = F.relu(out)

      out = self.linear1(out.view(-1, 256))
      out = self.linear1_bn(out)
      out=self.dp6(out)
      out = F.relu(out)

      out = self.linear2(out)
      out = self.linear2_bn(out)
      out=self.dp7(out)
      out = F.relu(out)

      logits = self.linear3(out)
      probas = F.softmax(logits, dim=1)
      return logits, probas

torch.manual_seed(RANDOM_SEED)

model3 = ConvNet3(NUM_CLASSES)
model3.to(DEVICE)

optimizer = torch.optim.Adam(model3.parameters(), lr=LEARNING_RATE)

minibatch_cost, epoch_cost = train(model3, train_loader, test_loader)
```

```
plt.plot(range(len(minibatch_cost)), minibatch_cost)
plt.ylabel('Cross Entropy')
plt.xlabel('Minibatch')
plt.show()

plt.plot(range(len(epoch_cost)), epoch_cost)
plt.ylabel('Cross Entropy')
plt.xlabel('Epoch')
plt.show()
```

```
Epoch: 001/020 | Batch 0000/0196 | Cost: 2.4537
Epoch: 001/020 | Batch 0150/0196 | Cost: 1.4927
Epoch: 001/020 | Train: 49.538%
Time elapsed: 1.35 min
Epoch: 002/020 | Batch 0000/0196 | Cost: 1.5140
Epoch: 002/020 | Batch 0150/0196 | Cost: 1.2249
Epoch: 002/020 | Train: 65.364%
Time elapsed: 2.71 min
Epoch: 003/020 | Batch 0000/0196 | Cost: 1.0317
Epoch: 003/020 | Batch 0150/0196 | Cost: 1.0087
Epoch: 003/020 | Train: 72.594%
Time elapsed: 4.07 min
Epoch: 004/020 | Batch 0000/0196 | Cost: 0.8566
Epoch: 004/020 | Batch 0150/0196 | Cost: 0.8130
Epoch: 004/020 | Train: 73.796%
Time elapsed: 5.42 min
Epoch: 005/020 | Batch 0000/0196 | Cost: 0.9007
Epoch: 005/020 | Batch 0150/0196 | Cost: 0.7540
Epoch: 005/020 | Train: 78.860%
Time elapsed: 6.78 min
Epoch: 006/020 | Batch 0000/0196 | Cost: 0.5702
Epoch: 006/020 | Batch 0150/0196 | Cost: 0.7506
Epoch: 006/020 | Train: 82.386%
Time elapsed: 8.14 min
Epoch: 007/020 | Batch 0000/0196 | Cost: 0.7508
Epoch: 007/020 | Batch 0150/0196 | Cost: 0.6259
Epoch: 007/020 | Train: 80.240%
Time elapsed: 9.49 min
Epoch: 008/020 | Batch 0000/0196 | Cost: 0.6062
Epoch: 008/020 | Batch 0150/0196 | Cost: 0.6511
Epoch: 008/020 | Train: 83.858%
Time elapsed: 10.85 min
Epoch: 009/020 | Batch 0000/0196 | Cost: 0.5492
Epoch: 009/020 | Batch 0150/0196 | Cost: 0.5179
Epoch: 009/020 | Train: 81.552%
Time elapsed: 12.20 min
Epoch: 010/020 | Batch 0000/0196 | Cost: 0.4616
Epoch: 010/020 | Batch 0150/0196 | Cost: 0.4843
Epoch: 010/020 | Train: 84.964%
Time elapsed: 13.56 min
Epoch: 011/020 | Batch 0000/0196 | Cost: 0.4386
Epoch: 011/020 | Batch 0150/0196 | Cost: 0.5528
Epoch: 011/020 | Train: 91.404%
Time elapsed: 14.91 min
Epoch: 012/020 | Batch 0000/0196 | Cost: 0.4094
Epoch: 012/020 | Batch 0150/0196 | Cost: 0.4339
Epoch: 012/020 | Train: 92.206%
Time elapsed: 16.26 min
Epoch: 013/020 | Batch 0000/0196 | Cost: 0.3134
Epoch: 013/020 | Batch 0150/0196 | Cost: 0.4594
Epoch: 013/020 | Train: 94.866%
Time elapsed: 17.61 min
Epoch: 014/020 | Batch 0000/0196 | Cost: 0.3690
Epoch: 014/020 | Batch 0150/0196 | Cost: 0.3366
Epoch: 014/020 | Train: 93.302%
Time elapsed: 18.97 min
Epoch: 015/020 | Batch 0000/0196 | Cost: 0.4009
Epoch: 015/020 | Batch 0150/0196 | Cost: 0.2406
Epoch: 015/020 | Train: 94.152%
Time elapsed: 20.32 min
Epoch: 016/020 | Batch 0000/0196 | Cost: 0.2839
```

```
Epoch: 016/020 | Batch 0150/0196 | Cost: 0.2550
Epoch: 016/020 | Train: 97.224%
Time elapsed: 21.67 min
Epoch: 017/020 | Batch 0000/0196 | Cost: 0.1787
Epoch: 017/020 | Batch 0150/0196 | Cost: 0.2258
Epoch: 017/020 | Train: 97.310%
Time elapsed: 23.02 min
Epoch: 018/020 | Batch 0000/0196 | Cost: 0.1833
Epoch: 018/020 | Batch 0150/0196 | Cost: 0.2449
Epoch: 018/020 | Train: 93.760%
Time elapsed: 24.38 min
Epoch: 019/020 | Batch 0000/0196 | Cost: 0.1848
Epoch: 019/020 | Batch 0150/0196 | Cost: 0.3225
Epoch: 019/020 | Train: 98.830%
Time elapsed: 25.74 min
Epoch: 020/020 | Batch 0000/0196 | Cost: 0.1645
Epoch: 020/020 | Batch 0150/0196 | Cost: 0.2144
Epoch: 020/020 | Train: 99.068%
Time elapsed: 27.09 min
Total Training Time: 27.09 min
Test accuracy: 82.48%
Total Time: 27.15 min
```






```
del model3  # to save memory if you don't use it anymore
```

4) Going All-Convolutional

In this 4th part, your task is to remove all maxpooling layers and replace the fully-connected layers
number of elements of the activation tensors in the hidden layers should not change. I.e., when yo
increase the stride of the convolutional layers from 1 to 2 to achieve the same scaling. Furthermor
conmvolutional layer by a convolutional layer using stride=1 and a kernel with height and width eq

The new architecture is as follows (changes, compared to the previous section, are highlighted in

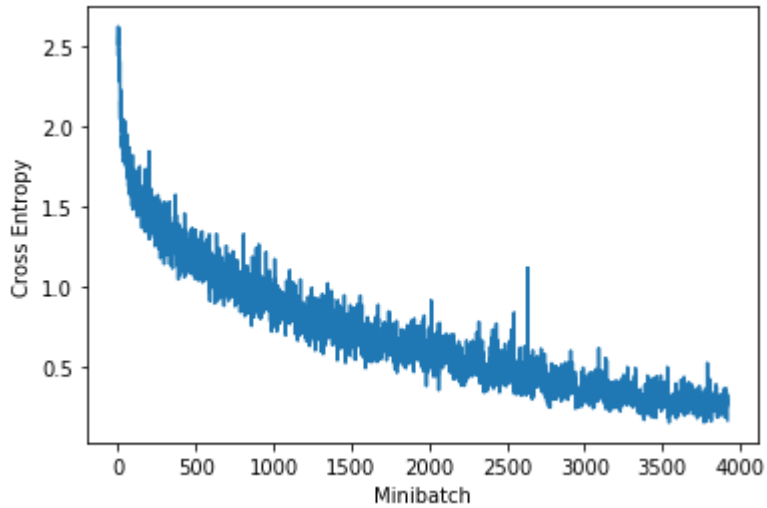
```
##########################
### MODEL
##########################

class ConvNet4(nn.Module):

    def __init__(self, num_classes=10):
        super(ConvNet4, self).__init__()

        self.conv1 =  nn.Conv2d(3, 64,   kernel_size=5, stride=2, padding=2)
        self.conv1_bn=torch.nn.BatchNorm2d(64)
        self.dp1=nn.Dropout2d(p=0.2)

        self.conv2 =  nn.Conv2d(64, 192,    kernel_size=5, stride=2, padding=2)
        self.conv2_bn=torch.nn.BatchNorm2d(192)
        self.dp2=nn.Dropout2d(p=0.2)

        self.conv3 =  nn.Conv2d(192, 384,    kernel_size=3, stride=2, padding=1)
        self.conv3_bn=torch.nn.BatchNorm2d(384)
        self.dp3=nn.Dropout2d(p=0.2)

        self.conv4 =  nn.Conv2d(384, 256,    kernel_size=5, stride=2, padding=2)
        self.conv4_bn=torch.nn.BatchNorm2d(256)
        self.dp4=nn.Dropout2d(p=0.2)

        self.conv5 =  nn.Conv2d(256, 256,    kernel_size=5, stride=2, padding=2)
        self.conv5_bn=torch.nn.BatchNorm2d(256)
        self.dp5=nn.Dropout2d(p=0.2)

        self.conv6 = nn.Conv2d(256, 4096, kernel_size=1, stride=1)
        self.conv6_bn=torch.nn.BatchNorm2d(4096)
        self.dp6=nn.Dropout2d(p=0.5)

        self.conv7 = nn.Conv2d(4096, 4096, kernel_size=1, stride=1)
        self.conv7_bn=torch.nn.BatchNorm2d(4096)
        self.dp7=nn.Dropout2d(p=0.5)

        self.conv8 = nn.Conv2d(4096, num_classes, kernel_size=1, stride=1)


    def forward(self, x):
```

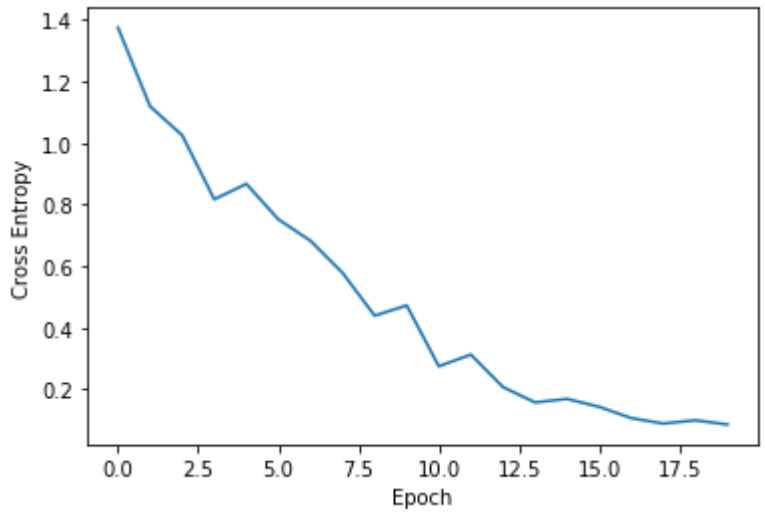
```
    def forward(self, x):

        out = self.conv1(x)
        out = self.conv1_bn(out)
        out=  self.dp1(out)
        out = F.relu(out)

        out = self.conv2(out)
        out = self.conv2_bn(out)
        out=  self.dp2(out)
        out = F.relu(out)

        out = self.conv3(out)
        out = self.conv3_bn(out)
        out=  self.dp3(out)
        out = F.relu(out)

        out = self.conv4(out)
        out = self.conv4_bn(out)
        out=  self.dp4(out)
        out = F.relu(out)

        out = self.conv5(out)
        out = self.conv5_bn(out)
        out=  self.dp5(out)
        out = F.relu(out)

        out = self.conv6(out)
        out = self.conv6_bn(out)
        out=  self.dp6(out)
        out = F.relu(out)

        out = self.conv7(out)
        out = self.conv7_bn(out)
        out=  self.dp7(out)
        out = F.relu(out)


        out = self.conv8(out)
        logits = out.view(out.size(0), NUM_CLASSES)
        probas = F.softmax(logits, dim=1)
        return logits, probas


torch.manual_seed(RANDOM_SEED)

model4 = ConvNet4(NUM_CLASSES)
model4.to(DEVICE)

optimizer = torch.optim.Adam(model4.parameters(), lr=LEARNING_RATE)

minibatch_cost, epoch_cost = train(model4, train_loader, test_loader)


plt.plot(range(len(minibatch_cost)), minibatch_cost)
plt.ylabel('Cross Entropy')
```

```
plt.ylabel('Cross Entropy')
plt.xlabel('Minibatch')
plt.show()

plt.plot(range(len(epoch_cost)), epoch_cost)
plt.ylabel('Cross Entropy')
plt.xlabel('Epoch')
plt.show()
```

```
Epoch: 001/020 | Batch 0000/0196 | Cost: 2.4499
Epoch: 001/020 | Batch 0150/0196 | Cost: 1.5547
Epoch: 001/020 | Train: 51.248%
Time elapsed: 1.22 min
Epoch: 002/020 | Batch 0000/0196 | Cost: 1.4272
Epoch: 002/020 | Batch 0150/0196 | Cost: 1.2327
Epoch: 002/020 | Train: 59.990%
Time elapsed: 2.44 min
Epoch: 003/020 | Batch 0000/0196 | Cost: 1.0700
Epoch: 003/020 | Batch 0150/0196 | Cost: 1.1092
Epoch: 003/020 | Train: 63.282%
Time elapsed: 3.65 min
Epoch: 004/020 | Batch 0000/0196 | Cost: 0.9672
Epoch: 004/020 | Batch 0150/0196 | Cost: 0.9417
Epoch: 004/020 | Train: 71.514%
Time elapsed: 4.87 min
Epoch: 005/020 | Batch 0000/0196 | Cost: 1.0373
Epoch: 005/020 | Batch 0150/0196 | Cost: 0.9450
Epoch: 005/020 | Train: 70.020%
Time elapsed: 6.09 min
Epoch: 006/020 | Batch 0000/0196 | Cost: 0.7822
Epoch: 006/020 | Batch 0150/0196 | Cost: 0.9206
Epoch: 006/020 | Train: 73.870%
Time elapsed: 7.30 min
Epoch: 007/020 | Batch 0000/0196 | Cost: 0.8268
Epoch: 007/020 | Batch 0150/0196 | Cost: 0.7287
Epoch: 007/020 | Train: 75.880%
Time elapsed: 8.52 min
Epoch: 008/020 | Batch 0000/0196 | Cost: 0.7882
Epoch: 008/020 | Batch 0150/0196 | Cost: 0.7552
Epoch: 008/020 | Train: 79.940%
Time elapsed: 9.73 min
Epoch: 009/020 | Batch 0000/0196 | Cost: 0.7944
Epoch: 009/020 | Batch 0150/0196 | Cost: 0.6225
Epoch: 009/020 | Train: 85.330%
Time elapsed: 10.95 min
Epoch: 010/020 | Batch 0000/0196 | Cost: 0.5655
Epoch: 010/020 | Batch 0150/0196 | Cost: 0.5967
Epoch: 010/020 | Train: 83.604%
Time elapsed: 12.16 min
Epoch: 011/020 | Batch 0000/0196 | Cost: 0.5521
Epoch: 011/020 | Batch 0150/0196 | Cost: 0.6055
Epoch: 011/020 | Train: 91.120%
Time elapsed: 13.37 min
Epoch: 012/020 | Batch 0000/0196 | Cost: 0.4996
Epoch: 012/020 | Batch 0150/0196 | Cost: 0.7116
Epoch: 012/020 | Train: 89.406%
Time elapsed: 14.59 min
Epoch: 013/020 | Batch 0000/0196 | Cost: 0.5135
Epoch: 013/020 | Batch 0150/0196 | Cost: 0.5775
Epoch: 013/020 | Train: 93.682%
Time elapsed: 15.80 min
Epoch: 014/020 | Batch 0000/0196 | Cost: 0.4795
Epoch: 014/020 | Batch 0150/0196 | Cost: 0.4924
Epoch: 014/020 | Train: 95.206%
Time elapsed: 17.02 min
Epoch: 015/020 | Batch 0000/0196 | Cost: 0.3639
Epoch: 015/020 | Batch 0150/0196 | Cost: 0.2904
Epoch: 015/020 | Train: 94.642%
Time elapsed: 18.25 min
Epoch: 016/020 | Batch 0000/0196 | Cost: 0.2849
```

```
Epoch: 016/020 | Batch 0150/0196 | Cost: 0.3116
Epoch: 016/020 | Train: 95.428%
Time elapsed: 19.46 min
Epoch: 017/020 | Batch 0000/0196 | Cost: 0.3027
Epoch: 017/020 | Batch 0150/0196 | Cost: 0.3501
Epoch: 017/020 | Train: 96.658%
Time elapsed: 20.68 min
Epoch: 018/020 | Batch 0000/0196 | Cost: 0.3033
Epoch: 018/020 | Batch 0150/0196 | Cost: 0.2465
Epoch: 018/020 | Train: 97.382%
Time elapsed: 21.89 min
Epoch: 019/020 | Batch 0000/0196 | Cost: 0.3597
Epoch: 019/020 | Batch 0150/0196 | Cost: 0.3582
Epoch: 019/020 | Train: 96.774%
Time elapsed: 23.11 min
Epoch: 020/020 | Batch 0000/0196 | Cost: 0.1746
Epoch: 020/020 | Batch 0150/0196 | Cost: 0.2512
Epoch: 020/020 | Train: 97.306%
Time elapsed: 24.33 min
Total Training Time: 24.33 min
Test accuracy: 74.07%
Total Time: 24.36 min
```






```
del model4
```

5) Add Image Augmentation

In this last section, you should use the architecture from the previous section (section 4) but use a
to improve the generalization performance.

In particular, you should modify the `train_transform = transforms.Compose([...)` function s

- performs a random horizontal flip with propbability 50%
- resizes the image from 32x32 to 40x40
- performs a 32x32 random crop from the 40x40 images
- normalizes the pixel intensities such that they are within the range [-1, 1]

The `test_transform = transforms.Compose([...` function should be modified accordingly, su

- resizes the image from 32x32 to 40x40
- performs a 32x32 **center** crop from the 40x40 images
- normalizes the pixel intensities such that they are within the range [-1, 1]

```
train_transform = transforms.Compose([
                                      transforms.RandomHorizontalFlip(p=0.5),
                                      transforms.Resize(40),
                                      transforms.RandomCrop(32),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=(0.5,0.5,0.5),std=(
])

test_transform = transforms.Compose([
                                     transforms.Resize(40),
                                     transforms.CenterCrop(32),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=(0.5,0.5,0.5),std=(0
])


train_dataset = datasets.CIFAR10(root='data',
                                 train=True,
                                 transform=train_transform,
                                 download=True)

test_dataset = datasets.CIFAR10(root='data',
                                train=False,
                                transform=test_transform)


train_loader = DataLoader(dataset=train_dataset,
                          batch_size=BATCH_SIZE,
                          num_workers=8,
                          shuffle=True)

test_loader = DataLoader(dataset=test_dataset,
                         batch_size=BATCH_SIZE,
```

```
                          num_workers=8,
                          shuffle=False)


torch.manual_seed(RANDOM_SEED)

model4 = ConvNet4(NUM_CLASSES)
model4.to(DEVICE)

optimizer = torch.optim.Adam(model4.parameters(), lr=LEARNING_RATE)

minibatch_cost, epoch_cost = train(model4, train_loader, test_loader)
```

```
Files already downloaded and verified
Epoch: 001/020 | Batch 0000/0196 | Cost: 2.4654
Epoch: 001/020 | Batch 0150/0196 | Cost: 1.6223
Epoch: 001/020 | Train: 43.374%
Time elapsed: 1.39 min
Epoch: 002/020 | Batch 0000/0196 | Cost: 1.5755
Epoch: 002/020 | Batch 0150/0196 | Cost: 1.4694
Epoch: 002/020 | Train: 52.796%
Time elapsed: 2.79 min
Epoch: 003/020 | Batch 0000/0196 | Cost: 1.3369
Epoch: 003/020 | Batch 0150/0196 | Cost: 1.2341
Epoch: 003/020 | Train: 57.522%
Time elapsed: 4.18 min
Epoch: 004/020 | Batch 0000/0196 | Cost: 1.2230
Epoch: 004/020 | Batch 0150/0196 | Cost: 1.2450
Epoch: 004/020 | Train: 60.718%
Time elapsed: 5.58 min
Epoch: 005/020 | Batch 0000/0196 | Cost: 1.4044
Epoch: 005/020 | Batch 0150/0196 | Cost: 1.1437
Epoch: 005/020 | Train: 63.638%
Time elapsed: 6.95 min
Epoch: 006/020 | Batch 0000/0196 | Cost: 1.1510
Epoch: 006/020 | Batch 0150/0196 | Cost: 1.1963
Epoch: 006/020 | Train: 66.526%
Time elapsed: 8.33 min
Epoch: 007/020 | Batch 0000/0196 | Cost: 1.2626
Epoch: 007/020 | Batch 0150/0196 | Cost: 0.9992
Epoch: 007/020 | Train: 67.188%
Time elapsed: 9.72 min
Epoch: 008/020 | Batch 0000/0196 | Cost: 1.1418
Epoch: 008/020 | Batch 0150/0196 | Cost: 0.9962
Epoch: 008/020 | Train: 68.264%
Time elapsed: 11.10 min
Epoch: 009/020 | Batch 0000/0196 | Cost: 1.0982
Epoch: 009/020 | Batch 0150/0196 | Cost: 1.1281
Epoch: 009/020 | Train: 71.084%
Time elapsed: 12.60 min
Epoch: 010/020 | Batch 0000/0196 | Cost: 0.8773
Epoch: 010/020 | Batch 0150/0196 | Cost: 0.9896
Epoch: 010/020 | Train: 72.054%
Time elapsed: 14.03 min
Epoch: 011/020 | Batch 0000/0196 | Cost: 0.9977
Epoch: 011/020 | Batch 0150/0196 | Cost: 1.0000
Epoch: 011/020 | Train: 73.356%
Time elapsed: 15.43 min
Epoch: 012/020 | Batch 0000/0196 | Cost: 0.9253
Epoch: 012/020 | Batch 0150/0196 | Cost: 0.9700
Epoch: 012/020 | Train: 73.658%
Time elapsed: 16.83 min
Epoch: 013/020 | Batch 0000/0196 | Cost: 0.9884
Epoch: 013/020 | Batch 0150/0196 | Cost: 0.9204
Epoch: 013/020 | Train: 73.928%
Time elapsed: 18.22 min
Epoch: 014/020 | Batch 0000/0196 | Cost: 0.8267
Epoch: 014/020 | Batch 0150/0196 | Cost: 0.9924
Epoch: 014/020 | Train: 75.528%
Time elapsed: 19.63 min
Epoch: 015/020 | Batch 0000/0196 | Cost: 0.9035
Epoch: 015/020 | Batch 0150/0196 | Cost: 1.0182
Epoch: 015/020 | Train: 77.018%
Time elapsed: 21.04 min
```

```
Epoch: 016/020 | Batch 0000/0196 | Cost: 0.7787
Epoch: 016/020 | Batch 0150/0196 | Cost: 0.8349
Epoch: 016/020 | Train: 77.130%
Time elapsed: 22.44 min
Epoch: 017/020 | Batch 0000/0196 | Cost: 0.9516
Epoch: 017/020 | Batch 0150/0196 | Cost: 0.8016
Epoch: 017/020 | Train: 77.478%
Time elapsed: 23.85 min
Epoch: 018/020 | Batch 0000/0196 | Cost: 0.7538
Epoch: 018/020 | Batch 0150/0196 | Cost: 0.8238
Epoch: 018/020 | Train: 78.332%
Time elapsed: 25.27 min
Epoch: 019/020 | Batch 0000/0196 | Cost: 0.8282
Epoch: 019/020 | Batch 0150/0196 | Cost: 0.8974
Epoch: 019/020 | Train: 79.074%
Time elapsed: 26.69 min
Epoch: 020/020 | Batch 0000/0196 | Cost: 0.6144
Epoch: 020/020 | Batch 0150/0196 | Cost: 0.7197
Epoch: 020/020 | Train: 79.412%
Time elapsed: 28.10 min
Total Training Time: 28.10 min
Test accuracy: 78.85%
Total Time: 28.16 min
```

6) Optional: Training the network for 200 epochs

In this optional section, train the network from the previous part for 200 epochs to see how it perfo take about 50 minutes and is optional (you will not receive a penalty if you don't run this section).

```
NUM_EPOCHS = 200

torch.manual_seed(RANDOM_SEED)

model4 = ConvNet4(NUM_CLASSES)
model4.to(DEVICE)

optimizer = torch.optim.Adam(model4.parameters(), lr=LEARNING_RATE)

minibatch_cost, epoch_cost = train(model4, train_loader, test_loader)


plt.plot(range(len(minibatch_cost)), minibatch_cost)
plt.ylabel('Cross Entropy')
plt.xlabel('Minibatch')
plt.show()

plt.plot(range(len(epoch_cost)), epoch_cost)
plt.ylabel('Cross Entropy')
plt.xlabel('Epoch')
plt.show()
```

Conclusions (Your Answers Required)

Now that you implemented the AlexNet-like architecture and made several modifications to it, plea
for each model, i.e., the weights and biases and batchnorm parameters, etc. (excluding the param
paste the training and test set accuracies below.

- **Model from section 1)**
    - Train accuracy: 98.536%
    - Test accuracy: 71.85%

Number of learnable parameters: (include your computation to receive partial points if the final an

input * kernel_size * output + output

- Conv2d (1) 3 * 5 * 5 * 64 + 64 = 4 864
- Conv2d (2) 64 * 5 * 5 * 192 + 192 = 307 392
- Conv2d (3) 192 * 3 * 3 * 384 + 384 = 663 936
- Conv2d (4) 384 * 5 * 5 * 256 + 256 = 2 457 856
- Conv2d (5) 256 * 5 * 5 * 256 + 256 = 1 638 656

prev_layer_size * layer_size + layer_size

- FC (1) 256 * 4096 + 4096 = 1 052 672
- FC (2) 4096 * 4096 + 4096 = 16 781 312
- FC (3) 4096 * 10 + 10 = 40970
- Total number of parameters: 22 947 658

- **Model from section 2)**
    - Train accuracy: 91.830%
    - Test accuracy: 76.20%

Number of learnable parameters: (include your computation to receive partial points if the final an

input * kernel_size * output + output

- Conv2d (1) 5 * 5 * 3 * 64 + 64 = 4 864
- Conv2d (2) 5 * 5 * 64 * 192 + 192 = 307 392
- Conv2d (3) 3 * 3 * 192 * 384 + 384 = 663 936
- Conv2d (4) 5 * 5 * 384 * 256 + 256 = 2 457 856
- Conv2d (5) 5 * 5 * 256 * 256 + 256 = 1 638 656

prev_layer_size * layer_size + layer_size

- FC (1) 256 * 4096 + 4096 = 1 052 672
- FC (2) 4096 * 4096 + 4096 = 16 781 312
- FC (3) 4096 * 10 + 10 = 40970
- Total number of parameters: 22 947 658

- **Model from section 3)**
    - Train accuracy: 99.068%
    - Test accuracy: 82.48%

Number of learnable parameters: (include your computation to receive partial points if the final an

input * kernel_size * output + output + batchnorm

- Conv2d (1) 5 * 5 * 3 * 64 + 64 + 128 = 4 992
- Conv2d (2) 5 * 5 * 64 * 192 + 192 + 384 = 307 776
- Conv2d (3) 3 * 3 * 192 * 384 + 384 + 768 = 664 704
- Conv2d (4) 5 * 5 * 384 * 256 + 256 + 512 = 2 458 368
- Conv2d (5) 5 * 5 * 256 * 256 + 256 + 512 = 1 639 168

prev_layer_size * layer_size + layer_size + batchnorm

- FC (1) 256 * 4096 + 4096 + 8192 = 1 060 864
- FC (2) 4096 * 4096 + 4096 + 8192 = 16 789 504
- FC (3) 4096 * 10 + 10 = 40970
- Total number of parameters: 22 966 346

- **Model from section 4)**
    - Train accuracy: 97.306%
    - Test accuracy: 74.07%

Number of learnable parameters: (include your computation to receive partial points if the final an

input * kernel_size * output + output + batchnorm

- Conv2d (1) 5 * 5 * 3 * 64 + 64 + 128 = 4 992
- Conv2d (2) 5 * 5 * 64 * 192 + 192 + 384 = 307 776
- Conv2d (3) 3 * 3 * 192 * 384 + 384 + 768 = 664 704
- Conv2d (4) 5 * 5 * 384 * 256 + 256 + 512 = 2 458 368
- Conv2d (5) 5 * 5 * 256 * 256 + 256 + 512 = 1 639 168

prev_layer_size * layer_size + layer_size + batchnorm

- FC (1) 256 * 4096 + 4096 + 8192 = 1 060 864
- FC (2) 4096 * 4096 + 4096 + 8192 = 16 789 504
- FC (3) 4096 * 10 + 10 = 40970
- Total number of parameters: 22 966 346

- **Model from section 5)**
    - Train accuracy: 79.412%
    - Test accuracy: 78.85%

Number of learnable parameters: (include your computation to receive partial points if the final an

input * kernel_size * output + output + batchnorm

- Conv2d (1) 5 * 5 * 3 * 64 + 64 + 128 = 4 992
- Conv2d (2) 5 * 5 * 64 * 192 + 192 + 384 = 307 776
- Conv2d (3) 3 * 3 * 192 * 384 + 384 + 768 = 664 704
- Conv2d (4) 5 * 5 * 384 * 256 + 256 + 512 = 2 458 368
- Conv2d (5) 5 * 5 * 256 * 256 + 256 + 512 = 1 639 168

prev_layer_size * layer_size + layer_size + batchnorm

- FC (1) 256 * 4096 + 4096 + 8192 = 1 060 864
- FC (2) 4096 * 4096 + 4096 + 8192 = 16 789 504
- FC (3) 4096 * 10 + 10 = 40970
- Total number of parameters: 22 966 346

- **Model from section 6) [optional]**
    - Train accuracy: ???%
    - Test accuracy: ???%
