ECE-6524 / CS-6524 Deep Learning

Assignment 2 [80 pts]

In this assignment, **you need to complete the following sections**:

1. PyTorch Basics
    - Toy example with PyTorch
2. Image Classification with PyTorch
    - Implement a simple MLP network for image classification
    - Implement a convolutional network for image classification
    - Experiment with different numbers of layers and optimizers
    - Push the performance of your CNN

This assignment is inspired and adopted from the official PyTorch tutorial.

Submission guideline for the coding part (Jupyter Notebook)

1. Click the Save button at the top of the Jupyter Notebook
2. Please make sure to have entered your Virginia Tech PID below
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of cells)
4. Select Cell -> Run All. This will run all the cells in order
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX
6. Look at the PDF file and make sure all your solutions are displayed correctly there
7. Zip all the files along with this notebook (Please don't include the data). Name it as Assignment_2_Code_[YOUR PID NUMBER].zip
8. Name your PDF file as Assignment_2_NB_[YOUR PID NUMBER].pdf
9. **Submit your zipped file and the PDF SEPARATELY**

Note: if facing issues with step 5 refer: https://pypi.org/project/notebook-as-pdf/

Submission guideline for the coding part (Google Colab)

1. Click the Save button at the top of the Notebook
2. Please make sure to have entered your Virginia Tech PID below
3. Follow last two cells in this notebook for guidelines to download pdf file of this notebook
4. Look at the PDF file and make sure all your solutions are displayed correctly there
5. Zip all the files along with this notebook (Please don't include the data). Name it as Assignment_2_Code_[YOUR PID NUMBER].zip
6. Name your PDF file as Assignment_2_NB_[YOUR PID NUMBER].pdf
7. **Submit your zipped file and the PDF SEPARATELY**

While you are encouraged to discuss with your peers, all work submitted is expected to be your own. If you use any information from other resources (e.g. online materials), you are required to cite it below you VT PID. Any violation will result in a 0 mark for the assignment.

Please Write Your VT PID Here: 906213559

Reference (if any):

https://medium.com/analytics-vidhya/simple-neural-network-with-bceloss-for-binary-classification-for-a-custom-dataset-8d5c69ffffee

In this homework, you would need to use **Python 3.6+** along with the following packages:

```
1. pytorch 1.2
2. torchvision
3. numpy
4. matplotlib
```

To install pytorch, please follow the instructions on the Official website. In addition, the official document could be very helpful when you want to find certain functionalities.

You can also consider to use Google Colab, where PyTorch has been installed.

Section 1. PyTorch Basics [10 pts]

Simply put, PyTorch is a **Tensor** library like Numpy. These two libraries similarly provide useful and efficient APIs for you to deal with your tensor data. What really differentiate PyTorch from Numpy are the following two features:

1. Numerical operations that can **run on GPUs** (more than 10x speedup)
2. Automatic differentiation for building and training neural networks

In this section, we will walk through some simple example, and see how the automatic differentiation functionality can make your life much easier.

To select GPU in Google Colab:

- go to **Edit -> Notebook settings -> Hardware accelerator -> GPU**

```
import torch # import pytorch.
import torch.nn as nn
```

```
dtype = torch.float
device = torch.device("cpu")
#device = torch.device("cuda:0") # Uncomment this to run on GPU
#print(torch.cuda.get_device_name(0)) # Check GPU Device name
```

1.1. Automatic Differentiation

Gradient descent is the driving force of the deep learning field. In the lectures and assignment 1, we learned how to derive the gradient for a given function, and implement methods for calculating and performing gradient descents. We also see how we can manually implement the backward and forward functions for the simple NN example. While implementing these functions may not be a big deal for a small network, it may get very nasty when we want to build something with tens of hundreds of layers.

In PyTorch (as well as other major deep learning libraries), we can use autograd (automatic differentiation) to handle the tedious computation of backward passes. When doing forward passes with autograd, we are essentially defining a **computational graph**, while the nodes in the graph are **tensors**, the edges are the functions that produce output tensors (e.g. ReLU, Linear, Convolutional Layer) given the input tensors. To do backpropagation, we can simply backtrack through this graph to compute gradients.

This may sound a little bit abstract, so let's take a look at the example:

```
target = 10.

# create a matrix of size 2x2. Each with value draws from standard normal distributic
x = torch.randn(2, 2, requires_grad=True)
y = torch.randn(2, 2, requires_grad=True)

a = x + y
b = a.sum()
loss = b - target

# print out each tensor:
print(x)
print(y)
print(a)
print(b)
print(loss)

print("-----gradient-----")
print(x.grad)
print(y.grad)
```

learnable parameters. The nn package also defines a set of useful loss functions that are commonly used when training neural networks.

Now, let's see how our simple NN could be implemented using the nn module.

```
import torch.nn as nn
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Tensors for its weight and bias.
model = nn.Sequential(
        nn.Linear(D_in, H),
        nn.ReLU(),
        nn.Linear(H, D_out),
)

# The nn package also contains definitions of popular loss functions; in this
# case we will use Mean Squared Error (MSE) as our loss function.
loss_fn = nn.MSELoss(reduction='sum')

learning_rate = 1e-4
for t in range(500):
        # Forward pass: compute predicted y by passing x to the model. Module objects
        # override the __call__ operator so you can call them like functions. When
        # doing so you pass a Tensor of input data to the Module and it produces
        # a Tensor of output data.
        y_pred = model(x)

        # Compute and print loss. We pass Tensors containing the predicted and true
        # values of y, and the loss function returns a Tensor containing the
        # loss.
        loss = loss_fn(y_pred, y)
        if t % 100 == 99:
                print(f'iteration {t}: {loss.item()}')

        # Zero the gradients before running the backward pass.
        model.zero_grad()

        # Backward pass: compute gradient of the loss with respect to all the learnab
        # parameters of the model. Internally, the parameters of each Module are store
        # in Tensors with requires_grad=True, so this call will compute gradients for
        # all learnable parameters in the model.
        loss.backward()

        # Update the weights using gradient descent. Each parameter is a Tensor, so
```

```
tensor([[0.8357, 0.8022],
        [1.2792, 0.6401]], requires_grad=True)
tensor([[-0.1575,  0.0758],
        [ 1.3740,  0.9312]], requires_grad=True)
tensor([[0.6782, 0.8780],
        [2.6533, 1.5713]], grad_fn=<AddBackward0>)
```

In the above example, we have seen a few things:

1. `requires_grad` flag: If false, we can safely exclude this tensor (and its subgraph) from gradient computation and therefore increase efficiency.
2. `grad_fn` : we can see that once an operation is done to a tensor, the output tensor is bound to a backward function associated to the operation. In this case, we have Add, Sum, and Sub.

However, even if we set `requires_grad=True` , we still don't have gradient for `x` and `y` . This is because that we haven't performed the backpropagation yet. So let's do it:

```
# perform backpropagation from this "node"
loss.backward()
print('-----gradient-----')
print(x.grad)
print(y.grad)
```

```
-----gradient-----
tensor([[1., 1.],
        [1., 1.]])
tensor([[1., 1.],
        [1., 1.]])
```

Great, seems like we can perform gradient descent without writing backwards function! Now, let's see a simple toy example on how we can fit some weights `w1` and `w2` with random input `x` and target `y` :

```
# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold input and outputs.
# Setting requires_grad=False indicates that we do not need to compute gradients
# with respect to these Tensors during the backward pass.
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)

# Create random Tensors for weights.
# Setting requires_grad=True indicates that we want to compute gradients with
# respect to these Tensors during the backward pass.
w1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
```

```
    # Forward pass: compute predicted y using operations on Tensors; these
    # are exactly the same operations we used to compute the forward pass using
    # Tensors, but we do not need to keep references to intermediate values since
    # we are not implementing the backward pass by hand.
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(f'iteration {t}: {loss.item()}')
    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call w1.grad and w2.grad will be Tensors holding the gradient
    # of the loss with respect to w1 and w2 respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.  (because we don't need the gradient for the operation
    # learning_rate * w1.grad)
    # An alternative way is to operate on weight.data and weight.grad.data.
    # Recall that tensor.data gives a tensor that shares the storage with
    # tensor, but doesn't track history.
    # You can also use torch.optim.SGD to achieve this.
    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad

        # Manually zero the gradients after updating weights
        w1.grad.zero_()
        w2.grad.zero_()
```

```
iteration 99: 254.9320526123047
iteration 199: 0.4827786087989807
iteration 299: 0.001796723110601306
iteration 399: 7.213609933387488e-05
iteration 499: 1.901478572108317e-05
```

1.2. nn Module

Computational graphs and autograd are a very powerful paradigm for defining complex operators and automatically taking derivatives; however for large neural networks raw autograd can be a bit too low-level.

When building neural networks we frequently think of arranging the computation into layers, some of which have learnable parameters which will be optimized during learning.

In PyTorch, the nn package serves this purpose. The nn package defines a set of Modules, which are roughly equivalent to neural network layers. A Module receives input Tensors and computes output Tensors, but may also hold internal state such as Tensors containing

```
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
```

```
iteration 99: 2.890277624130249
iteration 199: 0.05156201869249344
iteration 299: 0.001801187638193369
iteration 399: 8.995144162327051e-05
iteration 499: 5.438630068965722e-06
```

So far, we have been updating the model parameters manually with `torch.no_grad()` . However, if we want to use optimization algorithms other than SGD, it might get a bit nasty to do it manually. Instead of manually doing this, we can use `optim` pacakge to help optimize our model:

```
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Use the nn package to define our model and loss function.
model = nn.Sequential(
    nn.Linear(D_in, H),
    nn.ReLU(),
    nn.Linear(H, D_out),
)
loss_fn = torch.nn.MSELoss(reduction='sum')

# Use the optim package to define an Optimizer that will update the weights of
# the model for us.
learning_rate = 1e-4
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
for t in range(500):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(x)

    # Compute and print loss.
    loss = loss_fn(y_pred, y)
    if t % 100 == 99:
        print(f'iteration {t}: {loss.item()}')

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

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

```
        # Calling the step function on an Optimizer makes an update to its
        # parameters
        optimizer.step()
```

```
iteration 99: 2.6702218055725098
iteration 199: 0.03655353933572769
iteration 299: 0.001012633554637432
iteration 399: 7.044992526061833e-05
iteration 499: 9.368484825245105e-06
```

Sometimes you will want to specify models that are more complex than a sequence of existing Modules; for these cases you can define your own Modules by subclassing nn.Module and defining a forward which receives input Tensors and produces output Tensors using other modules or other autograd operations on Tensors.

For example, we can implement our 2-layer simple NN as the following:

```
class TwoLayerNet(nn.Module):
        def __init__(self, D_in, H, D_out):
                """
                In the constructor we instantiate two nn.Linear modules and assign them
                member variables.
                """
                super(TwoLayerNet, self).__init__()
                self.linear1 = nn.Linear(D_in, H)
                self.linear2 = nn.Linear(H, D_out)

        def forward(self, x):
                """
                In the forward function we accept a Tensor of input data and we must
                a Tensor of output data. We can use Modules defined in the constructor
                well as arbitrary operators on Tensors.
                """
                h_relu = self.linear1(x).clamp(min=0)
                y_pred = self.linear2(h_relu)
                return y_pred


# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = 64, 1000, 100, 10

# Create random Tensors to hold inputs and outputs
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# Construct our model by instantiating the class defined above
model = TwoLayerNet(D_in, H, D_out)

# Construct our loss function and an Optimizer. The call to model.parameters()
# in the SGD constructor will contain the learnable parameters of the two
# nn.Linear modules which are members of the model.
criterion = nn.MSELoss(reduction='sum')
optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
```

```
for t in range(500):
    # Forward pass: Compute predicted y by passing x to the model
    #print(x.shape)
    y_pred = model(x)
    #print(y_pred)
    # Compute and print loss
    loss = criterion(y_pred, y)
    if t % 100 == 99:
        print(f'iteration {t}: {loss.item()}')

    # Zero gradients, perform a backward pass, and update the weights.
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

```
iteration 99: 1.9846348762512207
iteration 199: 0.026682140305638313
iteration 299: 0.0006898815045133233
iteration 399: 2.6139518013224006e-05
iteration 499: 1.362579268970876e-06
```

1.3. Warm-up: Two-moon datasets [10 pts]

Now, let's use PyTorch to solve some synthetic datasets. In previous assignment, we have to write some codes to create training batches. Again, this can also be done with PyTorch `DataLoader`. The `DataLoader` utilizes parallel workers to read and prepare batches for you, which can greatly speedup the code when your time bottleneck is on file I/O.

Here, we show a simple example that can create a dataloader from numpy data:

Setup for Google Colab (Skip for Jupyter Notebook)

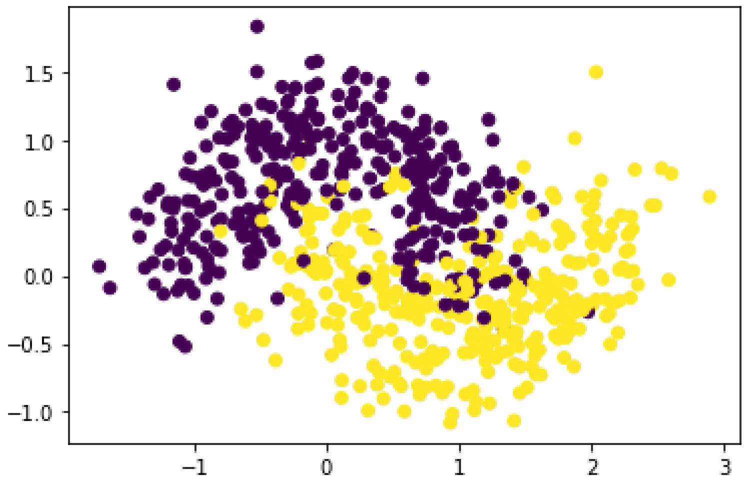
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.
```

```
# Find path to your data folder in drive and enter for "path_to_dataset"
path_to_dataset = '/content/drive/My Drive/DL_Fall_2020/HW2/data'
# For Jupyter notebook give path from your local PC
```

```
import numpy as np
import matplotlib.pyplot as plt

X_train = np.loadtxt(path_to_dataset + '/X1_train.csv', delimiter=',')
X_test = np.loadtxt(path_to_dataset + '/X1_test.csv', delimiter=',')
y_train = np.loadtxt(path_to_dataset + '/y1_train.csv', delimiter=',')
y_test = np.loadtxt(path_to_dataset + '/y1_test.csv', delimiter=',')
```

```
# Plot it to see why is it called two-moon dataset
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train);
```



Now, let's create a PyTorch `DataLoader`:

```
from torch.utils.data import TensorDataset, DataLoader
batch_size = 64 # mini-batch size
num_workers = 4 # how many parallel workers are we gonna use for reading data
shuffle = True # shuffle the dataset

# Convert numpy array import torch tensor
X_train = torch.FloatTensor(X_train)
X_test = torch.FloatTensor(X_test)
y_train = torch.LongTensor(y_train.reshape(-1, 1))
y_test = torch.LongTensor(y_test.reshape(-1, 1))

# First, create a dataset from torch tensor. A dataset define how to read data
# and process data for creating mini-batches.
train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=shuffle)


print(X_train.shape)
print(y_train.shape)
#print(y_train)
```

```
torch.Size([700, 2])
torch.Size([700, 1])
```

Below, we provide a simple example on how to train your model with this dataloader:

```
epoch = 5 # an epoch means looping through all the data in the datasets
lr = 1e-1

# create a simple model that is probably not gonna work well
model = nn.Linear(X_train.size(1), 1)
optim = torch.optim.SGD(model.parameters(), lr=lr)
```

```
for e in range(epoch):
    loss_epoch = 0
    # loop through train loader to get x and y
    for x, y in train_loader:
        optim.zero_grad()
        y_pred = model(x)
        # !!WARNING!!
        # THIS IS A CLASSIFICATION TASK, SO YOU SHOULD NOT
        # USE THIS LOSS FUNCTION.
        loss = (y_pred - y.float()).abs().mean()
        loss.backward()
        optim.step()
        loss_epoch += loss.item()
    print(f'Epcoh {e}: {loss_epoch}')
```

```
Epcoh 0: 4.294688671827316
Epcoh 1: 2.959206283092499
Epcoh 2: 2.914952725172043
Epcoh 3: 2.8958882242441177
Epcoh 4: 2.898398518562317
```

1.3.1 Your Simple NN [10 pts]

Now, it is time for you to implement your own model for this classification task. Your job here is to:

1. Complete the SimpleNN class. It should be a 2- or 3-layer NN with proper non-linearity.
2. Train your model with SGD optimizer.
3. Tune your model a bit so you can achieve at least 80% accuracy on training set. Hint: you might want to look up `nn.ReLU`, `nn.Sigmoid`, `nn.BCELoss` in the official document. You are allowed to freely pick the hyperparameters of your model.
4. **Please note this is a binary classification problem.**

```
class SimpleNN(nn.Module):

    def __init__(self):
        super().__init__()
        ################################################################################
        # TODO:
        # Construct your small feedforward NN here.
        ################################################################################
        self.linear1 = nn.Linear(X_train.size(1), 4)
        self.linear2 = nn.Linear(4, 2)
        self.linear3 = nn.Linear(2, 1)
        self.sigmoid = nn.Sigmoid()
        ################################################################################
        #                                                          END OF YOUR (
        ################################################################################

    def forward(self, x):
        ################################################################################
```

```
        # TODO:
        # feed the input to your network, and output the predictions.
        ################################################################################
        out = self.linear1(x)
        out = self.linear2(out)
        out = self.linear3(out)
        out = self.sigmoid(out)
        return out
        ################################################################################
        #                                                                   END OF YOUR (
        ################################################################################


epoch = 10 # an epoch means looping through all the data in the datasets
lr = 1e-1

# create a simple model that is probably not gonna work well


################################################################################
# TODO:
# Initialize your model and SGD optimizer here.
################################################################################
model = SimpleNN()
criterion = nn.BCELoss()
optim = torch.optim.SGD(model.parameters(), lr = lr)
################################################################################
#                                                                   END OF YOUR CODE
################################################################################

for e in range(epoch):
    loss_epoch = 0  # record accmulative loss for each epoch
    ################################################################################
    # TODO:
    # Loop through the dataloader and train your model with nn.BCELoss.
    ################################################################################
    for x, y in train_loader:
        optim.zero_grad()
        y_pred = model(x)
        loss = criterion(y_pred, y.float())
        loss.backward()
        optim.step()
        loss_epoch += loss.item()
    print(f'Epcoh {e}: {loss_epoch}')
    ################################################################################
    #                                                                   END OF YOUR CODE
    ################################################################################
```

```
Epcoh 0: 8.032418847084045
Epcoh 1: 7.818895637989044
Epcoh 2: 7.686410307884216
```

```
# helper function for computing accuracy
def get_acc(pred, y):
        pred = pred.float()
        y = y.float()
        return (y==pred).sum().float()/y.size(0)*100.
```

Evaluate your accuracy:

```
y_pred = (model(X_train) > 0.5)
train_acc = get_acc(y_pred, y_train)

y_pred = (model(X_test) > 0.5)
test_acc = get_acc(y_pred, y_test)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
```

```
Training accuracy: 80.85714721679688, Testing accuracy: 83.66666412353516
```

Section 2. Image Classification with CNN [70 pts]

Now, we are back to the image classification problem. In this section, our goal is to, again, train models on CIFAR-10 to perform image classification. Your tasks here are to:

1. Build and Train a simple feed-forward Neural Network (consists of only nn.Linear layer with activation function) for the classification task
2. Build and Train a **Convolutional** Neural Network (CNN) for the classification task
3. Try different settings for training your CNN
4. Reproduce

In the following cell, we provide the code for creating a CIFAR10 dataloader. As you can see, PyTorch's `torchvision` package actually has an interface for the CIFAR10 dataset:

```
import torchvision
import torchvision.transforms as transforms

# Preprocessing steps on the training/testing data. You can define your own data augme
# here, and PyTorch's API will do the rest for you.
transform_train = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

```
])

# This will automatically download the dataset for you if it cannot find the data in
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=tr
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=tr
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-
                                        170500096/? [00:08<00:00, 19623588.12it/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
```

2.1 Simple NN [10 pts]

Implement a simple feed-forward neural network, and train it on the CIFAR-10 training set. Here's some specific requirements:

1. The network should only consists of `nn.Linear` layers and the activation functions of your choices (e.g. `nn.Tanh`, `nn.ReLU`, `nn.Sigmoid`, etc).
2. Train your model with `torch.optim.SGD` with the hyperparameters you like the most.

Note that the hyperparameters work in previous assignment might not work the same, as the implementations of layers could be different.

2.1.1 Design and training [8 pts]

```
class SimpleNN(nn.Module):

    def __init__(self):
        super().__init__()
        ################################################################################
        # TODO:
        # Construct your small feedforward NN here.
        ################################################################################
        self.linear1 = nn.Linear(3072, 256)
        self.linear2 = nn.Linear(256, 16)
        self.linear3 = nn.Linear(16, 10)
        self.relu = nn.ReLU()
        ################################################################################
        #                                                               END OF YOUR (
        ################################################################################

    def forward(self, x):
        # note that: here, the data is of the shape (B, C, H, W)
        # where B is the batch size, C is color channels, and H
        # and W is height and width.
        # To feed it into the linear layer, we need to reshape it
        # with .view() function.
        batch_size = x.size(0)
        x = x.view(batch_size, -1) # reshape the data from (B, C, H, W) to (I
        ################################################################################
```

```
        # TODO:
        # Forward pass, output the prediction score.
        ################################################################################
        out = self.linear1(x)
        out = self.relu(out)
        out = self.linear2(out)
        out = self.relu(out)
        out = self.linear3(out)
        out = self.relu(out)
        return out
        ################################################################################
        #                                                             END OF YOUR (
        ################################################################################


epoch = 10
lr = 1e-2
n_input = 3072
n_classes = 10

train_loader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=
test_loader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_workers=

################################################################################
# TODO:
# Your training code here.
################################################################################
model = SimpleNN()
criterion = nn.CrossEntropyLoss()
optim = torch.optim.SGD(model.parameters(), lr=lr)

for e in range(epoch):
    loss_epoch = 0
    ################################################################################
    # TODO:
    # Loop through the dataloader and train your model with nn.BCELoss.
    ################################################################################
    for x, y in train_loader:
        optim.zero_grad()
        y_pred = model(x)
        loss = criterion(y_pred, y)
        loss.backward()
        optim.step()
        loss_epoch += loss.item()
    print(f'Epcoh {e}: {loss_epoch}')
################################################################################
#                                                             END OF YOUR CODE
################################################################################
```

```
Epcoh 0: 1576.7239553928375
Epcoh 1: 1418.152688384056
Epcoh 2: 1300.7838598489761
Epcoh 3: 1235.6106083393097
Epcoh 4: 1188.454677581787
```

Now evaluate your model with the helper function:

```
def get_model_acc(model, loader):
    ys = []
    y_preds = []
    for x, y in loader:
        ys.append(y)
        # set the prediction to the one that has highest value
        # Note that the the output size of model(x) is (B, 10)
        y_preds.append(torch.argmax(model(x), dim=1))
    y = torch.cat(ys, dim=0)
    y_pred = torch.cat(y_preds, dim=0)
    print((y == y_pred).sum())
    return get_acc(y_pred, y)
```

2.1.2 Evaluate NN [2 pts]

Evaluate your NN. You should get an accuracy around **50%** on training set and **49%** on testing set.

```
train_acc = get_model_acc(model, train_loader)
test_acc = get_model_acc(model, test_loader)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
```

```
tensor(29456)
tensor(5002)
Training accuracy: 58.9119987487793, Testing accuracy: 50.019996643066406
```

2.2 Convolutional Neural Network (CNN) [60 pts]

Convolutional layer has been proven to be extremely useful for vision-based task. As mentioned in the lecture, this speical layer allows the model to learn filters that capture crucial visual features.

2.2.1 Implement and Evaluate CNN [15 pts]

In this section, you will need to construct a CNN for classifying CIFAR-10 image. Specifically, you need to:

1. build a `CNNClassifier` with `nn.Conv2d`, `nn.Maxpool2d` and activation functions that you think are appropriate.
2. You would need to flatten the output of your convolutional networks with `view()`, and feed it into a `nn.Linear` layer to predict the class labels of the input.

Once you are done with your module, train it with `optim.SGD`, and evaluate it. You should get an accuracy around **55%** on training set and **53%** on testing set.

Hint: You might want to look up `nn.Conv2d`, `nn.Maxpool2d`, `nn.CrossEntropyLoss()`, `view()` and `size()`

```
class CNNClassifier(nn.Module):

    def __init__(self):
        super().__init__()
        ################################################################################
        # TODO:
        # Construct a CNN with 2 or 3 convolutional layers and 1 linear layer
        # outputing class prediction. You are free to pick the hyperparameters
        ################################################################################
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=10, kernel_size=3)
        self.maxpool = nn.MaxPool2d(kernel_size=3)
        self.conv2 = nn.Conv2d(in_channels=10, out_channels=10, kernel_size=3)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

        # caculate the input dimension of the dense layers
        input = torch.randn(1, 3, 32, 32)
        input = self.conv1(input)
        input = self.maxpool(input)
        input = self.conv2(input)
        self.input_size = input.numel()

        self.linear = nn.Linear(self.input_size, 10)
        ################################################################################
        #                                                                  END OF YOUR (
        ################################################################################


    def forward(self, x):
        ################################################################################
        # TODO:
        # Forward pass of your network. First extract feature with CNN, and pre
        # class scores with linear layer. Be careful about your input/output sha
        ################################################################################
        out = self.conv1(x)
        out = self.relu(out)
        out = self.maxpool(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.relu(out)
        out = self.linear(out.view(out.shape[0], self.input_size))
        return out
        ################################################################################
        #                                                                  END OF YOUR (
```

```
        #                                                                    END OF YOUR C
        ################################################################################

# You can tune these hyperparameters as you like.
epoch = 10
lr = 1e-1
n_input = 3072
n_classes = 10
batch_size = 64
num_workers = num_workers

train_loader = torch.utils.data.DataLoader(trainset, batch_size=64, shuffle=True, num_workers=
test_loader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_workers=

################################################################################
# TODO:
# Your training code here.
################################################################################
model = CNNClassifier()
criterion = nn.CrossEntropyLoss()
optim = torch.optim.SGD(model.parameters(), lr=lr)

for e in range(epoch):
    loss_epoch = 0
    for x, y in train_loader:
        optim.zero_grad()
        y_pred = model(x)
        loss = criterion(y_pred, y)
        loss.backward()
        optim.step()
        loss_epoch += loss.item()
    print(f'Epcoh {e}: {loss_epoch}')
################################################################################
#                                                              END OF YOUR CODE
################################################################################
```

```
Epcoh 0: 1270.396087050438
Epcoh 1: 1071.751238822937
Epcoh 2: 997.9186962842941
Epcoh 3: 963.4325615763664
Epcoh 4: 932.900406062603
Epcoh 5: 919.5405436754227
Epcoh 6: 906.511062502861
Epcoh 7: 896.6453180909157
Epcoh 8: 889.945413351059
Epcoh 9: 890.3938311338425
```

```
# turn on evaluation mode. This is crucial when you have BatchNorm in your network,
# as you want to use the running mean/std you obtain durining training time to norma
# your input data. Rememeber to call .train() function after evaluation
model.eval()
train_acc = get_model_acc(model, train_loader)
test_acc = get_model_acc(model, test_loader)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
```

```
tensor(26922)
tensor(5025)
Training accuracy: 53.843997955322266, Testing accuracy: 50.25
```

Explain your design and hyperparameter choice in three or four sentences:

In this model, there are three convolutional layers and one linear layer. Kernel size 3 by 3 for input image of size (32, 32). I didn't change the hyperparamters for training, e.g. learning rate, etc.

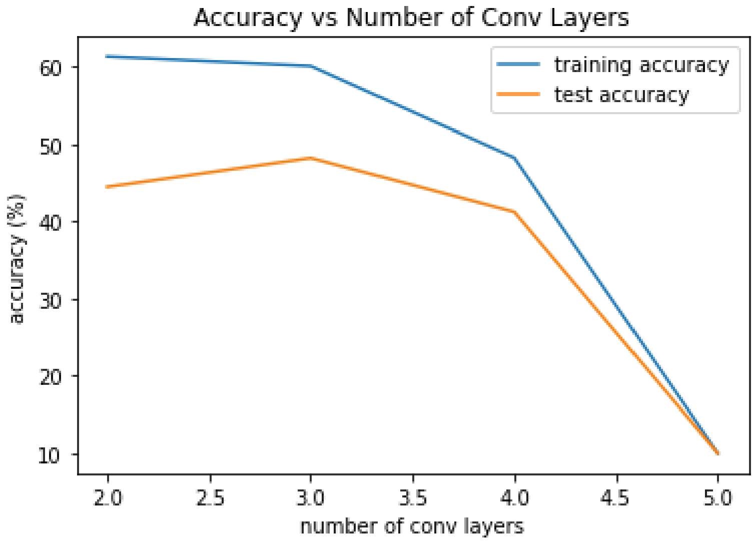
2.2.2 STACK MORE LAYERS [20 pts]

Now, **try at least 4 network architectures with different numbers of convolutional layers**. Train these settings with `optim.SGD`, plot the training/testing accuracy as a fuction of convolutional layers and describe what you have observed (running time, performance, etc). **Please make sure your figures are with clear legends and labels**.

```
#################################################################################
# TODO:
# Your training code here.
#################################################################################
class CNN(torch.nn.Module):
    def __init__(self, input_channel, hidden_channel, input_size, output_size, kernel_siz
        super(CNN, self).__init__()
        self.depth = depth
        self.input_layer = torch.nn.Conv2d(in_channels=input_channel, out_channels=hidde
        self.conv_layers = nn.ModuleList([torch.nn.Conv2d(in_channels=hidden_channel, ou

        self.activation = activation()
        self.output_activation = 'Linear'

        # caculate the input dimension of the dense layers
        input = torch.randn(1, input_channel, *input_size)
        input = self.input_layer(input)

        # hidden layers
        for h_layer in self.conv_layers:
            input = h_layer(input)
        self.input_size = input.numel()

        # linear layer
        self.output_layer  = nn.Linear(self.input_size, output_size)

    def forward(self, x):
        # input layer
        out = self.input_layer(x)
        # hidden layers
        for h_layer in self.conv_layers:
            out = h_layer(out)
            out = self.activation(out)
```

```
            # flatten
            out = out.view(out.shape[0], self.input_size)
            # output layer
            out = self.output_layer(out)
            return out

# training different convolutional layers
train_acc_list = list()
test_acc_list = list()

import time
train_time_list = list()

for d in range(2, 6):
    model = CNN(input_channel=3, hidden_channel=10, input_size=(32, 32), output_size=10,
    criterion = nn.CrossEntropyLoss()
    optim = torch.optim.SGD(model.parameters(), lr=lr)

    st_time = time.time()
    print('Training CNN with %d Convolutional Layers' % (d+1))

    for e in range(epoch):
        loss_epoch = 0
        for x, y in train_loader:
            optim.zero_grad()
            y_pred = model(x)
            loss = criterion(y_pred, y)
            loss.backward()
            optim.step()
            loss_epoch += loss.item()

    ed_time = time.time()
    train_time_list.append((ed_time-st_time))

    # evaluate
    model.eval()
    train_acc = get_model_acc(model, train_loader)
    test_acc = get_model_acc(model, test_loader)
    train_acc_list.append(train_acc)
    test_acc_list.append(test_acc)
    print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')

fig = plt.figure()
plt.plot(range(2, 6), train_acc_list, label='training accuracy')
plt.plot(range(2, 6), test_acc_list,  label='test accuracy')
plt.title('Accuracy vs Number of Conv Layers')
plt.legend()
plt.xlabel('number of conv layers')
plt.ylabel('accuracy (%)')
plt.show()
################################################################################
#                                                                    END OF YOUR CODE
################################################################################
```

```
Training CNN with 3 Convolutional Layers
tensor(30653)
tensor(4448)
Training accuracy: 61.305999755859375, Testing accuracy: 44.47999954223633
Training CNN with 4 Convolutional Layers
tensor(30039)
tensor(4820)
Training accuracy: 60.0780029296875, Testing accuracy: 48.20000076293945
Training CNN with 5 Convolutional Layers
tensor(24114)
tensor(4123)
Training accuracy: 48.227996826171875, Testing accuracy: 41.22999954223633
Training CNN with 6 Convolutional Layers
tensor(5000)
tensor(1000)
Training accuracy: 10.0, Testing accuracy: 10.0
```



Briefly explain what you have observed in three or four sentences. Does stacking layers always give you better results? How about the computational time?:

1.Stacking layers NOT always give you better results. Sometimes it might decrease the accurancy, e.g. 4,5 layers vs 3 layers.

2.Computational costs are higher for more layers of input.

2.2.3 Optimizer? Optimizer! [15 pts]

So far, we only use SGD as our optimizer. Now, pick two other optimizers, train your favorite CNN models, and compare the performance you get. What did you see?

```
#################################################################################
# TODO:
# Your training code here.
#################################################################################
epoch = 10
lr = 0.01

# Optimizer
```

```python
optimizers = [torch.optim.SGD, torch.optim.Adam, torch.optim.Adadelta, torch.optim.Adamax]

for optim in optimizers:
    print('Training with optimizer', optim)

    model = CNN(input_channel=3, hidden_channel=10, input_size=(32, 32), output_size=10,
    optim = optim(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()

    for e in range(epoch):
        loss_epoch = 0
        for x, y in train_loader:
            optim.zero_grad()
            y_pred = model(x)
            loss = criterion(y_pred, y)
            loss.backward()
            optim.step()
            loss_epoch += loss.item()

    # evaluate
    model.eval()
    train_acc = get_model_acc(model, train_loader)
    test_acc = get_model_acc(model, test_loader)
    print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
################################################################################
#                                                         END OF YOUR CODE
################################################################################
```

```
Training with optimizer <class 'torch.optim.sgd.SGD'>
tensor(30162)
tensor(5648)
Training accuracy: 60.32400131225586, Testing accuracy: 56.480003356933594
Training with optimizer <class 'torch.optim.adam.Adam'>
tensor(5000)
tensor(1000)
Training accuracy: 10.0, Testing accuracy: 10.0
Training with optimizer <class 'torch.optim.adadelta.Adadelta'>
tensor(23223)
tensor(4575)
Training accuracy: 46.44599914550781, Testing accuracy: 45.75
Training with optimizer <class 'torch.optim.adamax.Adamax'>
tensor(38839)
tensor(5731)
Training accuracy: 77.6780014038086, Testing accuracy: 57.30999755859375
```

What did you see? Which optimizer is your favorite? Describe:

My optimizer candidates are Adadelta, Adam, SGD and Adamax.

SDG is very good, but it can't compete with advanced optimizers.

Adamax is the best optimizer in this model and it's my favorite optimizer.

2.2.4 Improve Your Model [10 pts]

Again, we want you to play with your model a bit harder, and improve it. You are free to use everything you can find in the documents (`BatchNorm`, `SeLU`, etc), as long as it is not a **predefined network architectures in PyTorch package**. You can also implement some famous network architectures to push the performance.

(A simple network with 5-6 `nn.Conv2d` can give you at least 70% accuracy on testing set).

```
################################################################################
# TODO:
# Your training code here.
################################################################################
class ImproveCNN(nn.Module):
        def __init__(self):
                super(ImproveCNN, self).__init__()
                self.conv1 = nn.Conv2d(3, 9, 5)
                self.conv2 = nn.Conv2d(9, 16, 5)
                self.elu = nn.ELU()

                # drop-out
                self.drop = nn.Dropout(p=0.1)

                # pooling layers
                self.maxpool = nn.MaxPool2d(2)

                # fully-connected layers
                self.linear1 = nn.Linear(self.input_dim(), 120)
                self.linear2 = nn.Linear(120, 84)
                self.linear3 = nn.Linear(84, 10)

        def input_dim(self):
                input = torch.randn(1, 3, 32, 32)
                out = self.conv_forward(input)# convolutions
                return out.numel()

        def conv_forward(self, x):
                # convolution 1
                out = self.conv1(x)
                out = self.elu(out)
                out = self.maxpool(out)
                out = self.drop(out)

                # convolution 2
                out = self.conv2(out)
                out = self.elu(out)
                out = self.maxpool(out)
                out = self.drop(out)

                return out

        def forward(self, x):
                out = self.conv_forward(x)
                out = out.view(out.shape[0], -1)# flattening
```

```
        # fully-connected 1
        out = self.linear1(out)
        out = self.elu(out)

        # fully-connected 2
        out = self.linear2(out)
        out = self.elu(out)

        out = self.drop(out)
        out = self.linear3(out)
        return out

model = ImproveCNN()
criterion = nn.CrossEntropyLoss()
optim = torch.optim.Adamax(model.parameters())

epoch = 100
for e in range(epoch):
    loss_epoch = 0
    for x, y in train_loader:
        optim.zero_grad()
        y_pred = model(x)
        loss = criterion(y_pred, y)
        loss.backward()
        optim.step()
        loss_epoch += loss.item()
    if e % 5 == 0:
        print(f'Epcoh {e}: {loss_epoch}')

model.eval()
train_acc = get_model_acc(model, train_loader)
test_acc = get_model_acc(model, test_loader)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
################################################################################
#                                                                END OF YOUR CODE
################################################################################
```

```
Epcoh 0: 1248.6097791194916
Epcoh 5: 848.8630584478378
Epcoh 10: 730.3264741301537
Epcoh 15: 660.2822466492653
Epcoh 20: 614.5830645859241
```

Guidelines for Downloading PDF in Google Colab

- Run below cells only in Google Colab, Comment out in case of Jupyter notebook

```
Epcoh 50: 489.37844651937485
```

```
#Run below two lines (in google colab), installation steps to get .pdf of the notebo

!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc

# After installation, comment above two lines and run again to remove installation co

# Find path to your notebook file in drive and enter in below line

!jupyter nbconvert --to PDF "/content/drive/My Drive/DL_Fall_2020/HW2/Assignment_2_Code_90621

#Example: "/content/drive/My Drive/DL_Fall_2020/HW2/DL_Assignment_2.ipynb"
```

```
[NbConvertApp] WARNING | pattern u'/content/drive/My Drive/DL_Fall_2020/HW2/Assignme
This application is used to convert notebook files (*.ipynb) to various other
formats.

WARNING: THE COMMANDLINE INTERFACE MAY CHANGE IN FUTURE RELEASES.

Options
-------

Arguments that take values are actually convenience aliases to full
Configurables, whose aliases are listed on the help line. For more information
on full configurables, see '--help-all'.

--execute
    Execute the notebook prior to export.
--allow-errors
    Continue notebook execution even if one of the cells throws an error and include
--no-input
    Exclude input cells and output prompts from converted document.
    This mode is ideal for generating code-free reports.
--stdout
    Write notebook output to stdout instead of files.
--stdin
    read a single notebook file from stdin. Write the resulting notebook with defaul
--inplace
    Run nbconvert in place, overwriting the existing notebook (only
    relevant when converting to notebook format)
-y
    Answer yes to any questions instead of prompting.
--clear-output
    Clear output of current file and save in place,
    overwriting the existing notebook.
--debug
    set log level to logging.DEBUG (maximize logging output)
--no-prompt
    Exclude input and output prompts from converted document.
--generate-config
    generate default config file
--nbformat=<Enum> (NotebookExporter.nbformat_version)
    Default: 4
    Choices: [1, 2, 3, 4]
    The nbformat version to write. Use this to downgrade notebooks.
--output-dir=<Unicode> (FilesWriter.build_directory)
    Default: ''
    Directory to write output(s) to. Defaults to output to the directory of each
    notebook. To recover previous default behaviour (outputting to the current
    working directory) use . as the flag value.
--writer=<DottedObjectName> (NbConvertApp.writer_class)
    Default: 'FilesWriter'
    Writer class used to write the  results of the conversion
--log-level=<Enum> (Application.log_level)
    Default: 30
    Choices: (0, 10, 20, 30, 40, 50, 'DEBUG', 'INFO', 'WARN', 'ERROR', 'CRITICAL')
    Set the log level by value or name.
--reveal-prefix=<Unicode> (SlidesExporter.reveal_url_prefix)
    Default: u''
    The URL prefix for reveal.js (version 3.x). This defaults to the reveal CDN,
    but can be any url pointing to a copy  of reveal.js.
    For speaker notes to work, this must be a relative path to a local  copy of
    reveal.js: e.g., "reveal.js".
    If a relative path is given, it must be a subdirectory of the current
```