ECE-CS 6524 - Assignment 2

October 10, 2019

1 ECE-6524 / CS-6524 Deep Learning

2 Assignment 2

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

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 the all the files along with this notebook (Please don't include the data)
8. Name your PDF file as Assignment2_[YOUR ID NUMBER].
9. Submit your zipped file and the PDF **INDEPENDENTLY**.

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.

2.0.1 Please Write Your VT PID Here:

2.0.2 Reference (if any):

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.

3 Section 1. PyTorch Basics [30 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.

3.1 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:

[42]:
```python
import torch # import pytorch.

target = 10.

# create a matrix of size 2x2. Each with value draws from standard normal␣
 ↪distribution.
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)
```

```
tensor([[-1.0191,  0.0611],
        [-1.4309, -0.5273]], requires_grad=True)
tensor([[-0.3967,  1.0023],
        [ 0.1400, -0.5677]], requires_grad=True)
tensor([[-1.4158,  1.0633],
        [-1.2909, -1.0950]], grad_fn=<AddBackward0>)
tensor(-2.7384, grad_fn=<SumBackward0>)
tensor(-12.7384, grad_fn=<SubBackward0>)
-----gradient-----
None
None
```

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:

[43]:
```
# 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`:

[44]:
```
dtype = torch.float
#device = torch.device("cpu")
device = torch.device("cuda:0") # Uncomment this to run on GPU

# 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: 393.90301513671875
iteration 199: 0.9325202703475952
iteration 299: 0.0035673119127750397
iteration 399: 0.00010743165330495685
iteration 499: 2.4693455998203717e-05
```

3.2 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 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.

[45]:
```python
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␣
↪learnable
    # parameters of the model. Internally, the parameters of each Module are␣
↪stored
    # 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
    # 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: 1.7277593612670898
iteration 199: 0.014585996977984905
iteration 299: 0.00022753622033633292
iteration 399: 4.899233317701146e-06
iteration 499: 1.3099771933866577e-07
```

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:

[46]:
```
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)
```

```python
# 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: 1.8390264511108398
iteration 199: 0.0300820954144001
iteration 299: 0.0010718220146372914
iteration 399: 5.0371243560221046e-05
iteration 499: 2.6691325274441624e-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:

```
[47]: class TwoLayerNet(nn.Module):
          def __init__(self, D_in, H, D_out):
              """
              In the constructor we instantiate two nn.Linear modules and assign them␣
       ↪as
              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␣
       ↪return
              a Tensor of output data. We can use Modules defined in the constructor␣
       ↪as
              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
          y_pred = model(x)

          # 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()
```

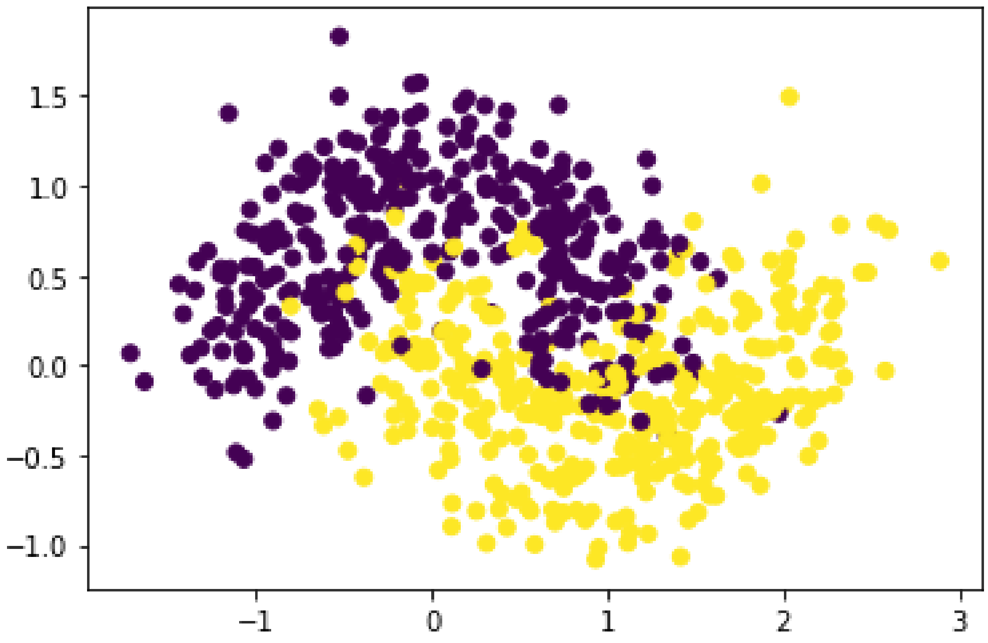
```
iteration 99: 1.7147811651229858
iteration 199: 0.014921829104423523
iteration 299: 0.00039041665149852633
iteration 399: 1.7119744370575063e-05
iteration 499: 9.435925107936782e-07
```

3.3 1.3. Warm-up: Two-moon datasets [30 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:

[48]:
```
import numpy as np
import matplotlib.pyplot as plt

X_train = np.loadtxt('data/X1_train.csv', delimiter=',')
X_test = np.loadtxt('data/X1_test.csv', delimiter=',')
y_train = np.loadtxt('data/y1_train.csv', delimiter=',')
y_test = np.loadtxt('data/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`:

```
[216]: 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)
```

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

```
[50]: 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: 6.937639683485031
Epcoh 1: 3.3677134215831757
Epcoh 2: 2.969435602426529
Epcoh 3: 2.914456009864807
Epcoh 4: 2.893051356077194
```

3.3.1 1.3.1 Your Simple NN [30 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.

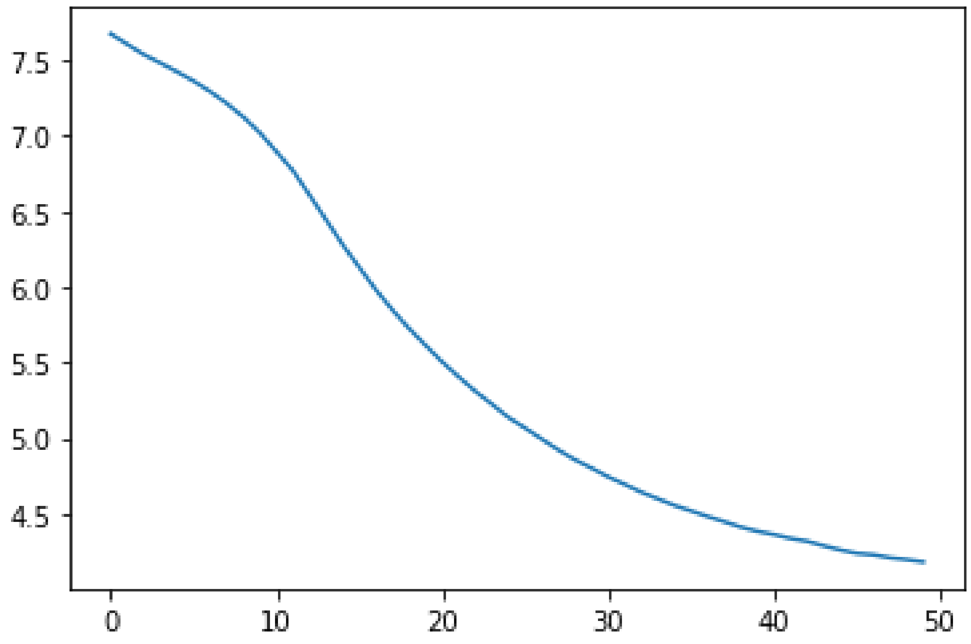
[226]:
```
class SimpleNN(nn.Module):

    def __init__(self, X_train, y_train):
        super().__init__()
        ␣
 ↪###############################################################################
        # TODO:                                                                  ␣
 ↪       #
        # Construct your small feedforward NN here.                             ␣
 ↪       #
        ␣
 ↪###############################################################################
        inputLayerWidth = X_train.size(1)
        hiddenLayerWidth = X_train.size(1)
        outputLayerWidth = y_train.size(1)
        self.linear1 = nn.Linear(inputLayerWidth, hiddenLayerWidth)
        self.linear2 = nn.Linear(hiddenLayerWidth, hiddenLayerWidth)
        self.linear3 = nn.Linear(hiddenLayerWidth, outputLayerWidth)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

       ␣
↪################################################################################
        #                                  END OF YOUR CODE                     ␣
↪        #
       ␣
↪################################################################################

    def forward(self, x):
       ␣
↪################################################################################
        # TODO:                                                                 ␣
↪        #
        # feed the input to your network, and output the predictions.           ␣
↪        #
       ␣
↪################################################################################

        # Create ReLU hidden layers, sigmoid output function
        x = self.relu(self.linear1(x))
        x = self.relu(self.linear2(x))
        y_pred = self.sigmoid(self.linear3(x))
        return y_pred

       ␣
↪################################################################################
        #                                  END OF YOUR CODE                     ␣
↪        #
       ␣
↪################################################################################
```

[230]:
```
epoch = 50 # an epoch means looping through all the data in the datasets
lr = 1e-2

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

################################################################################
# TODO:                                                                        #
# Initialize your model and SGD optimizer here.                                #
################################################################################
model = SimpleNN(X_train, y_train)
```

```
model.to(device)
optim = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.8)
bce_criterion = nn.BCELoss(reduction='mean')

################################################################################
#                                 END OF YOUR CODE                             #
################################################################################
loss_hist = []

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:
        x = x.to(device)
        y = y.to(device)

        y_pred = model(x)
        #import pdb; pdb.set_trace()

        #loss = (y_pred - y.float()).pow(2).mean()
        loss = bce_criterion(y_pred, y.float())

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
    loss_hist.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist)
    ␣
 ↪################################################################################
    #                                 END OF YOUR CODE                         ␣
 ↪   #
    ␣
 ↪################################################################################
```

```
Epcoh 0: 7.673741102218628
Epcoh 1: 7.60462874174118
Epcoh 2: 7.53637558221817
Epcoh 3: 7.480567097663879
Epcoh 4: 7.423857867717743
Epcoh 5: 7.362698674201965
Epcoh 6: 7.2930192947387695
Epcoh 7: 7.214723646640778
Epcoh 8: 7.124233245849609
Epcoh 9: 7.019440412521362
Epcoh 10: 6.894010543823242
Epcoh 11: 6.76827347278595
Epcoh 12: 6.6061625480651855
Epcoh 13: 6.4420445561409
Epcoh 14: 6.2784406542778015
Epcoh 15: 6.129379749298096
Epcoh 16: 5.981109082698822
Epcoh 17: 5.84803831577301
Epcoh 18: 5.72474679350853
Epcoh 19: 5.612942457199097
Epcoh 20: 5.503344237804413
Epcoh 21: 5.404290467500687
Epcoh 22: 5.309252381324768
Epcoh 23: 5.224063485860825
Epcoh 24: 5.137385725975037
Epcoh 25: 5.06739816069603
Epcoh 26: 4.994948893785477
Epcoh 27: 4.9230402410030365
Epcoh 28: 4.857765406370163
Epcoh 29: 4.802604138851166
Epcoh 30: 4.744755685329437
Epcoh 31: 4.695244073867798
Epcoh 32: 4.643418103456497
Epcoh 33: 4.599562853574753
Epcoh 34: 4.5557162165641785
Epcoh 35: 4.51969900727272
Epcoh 36: 4.482547074556351
Epcoh 37: 4.448710143566132
Epcoh 38: 4.4123846888542175
Epcoh 39: 4.384690165519714
Epcoh 40: 4.362939417362213
Epcoh 41: 4.337519019842148
Epcoh 42: 4.318515419960022
Epcoh 43: 4.288520991802216
Epcoh 44: 4.261151969432831
Epcoh 45: 4.238545387983322
Epcoh 46: 4.229094833135605
Epcoh 47: 4.209377348423004
Epcoh 48: 4.197415888309479
Epcoh 49: 4.181982487440109
```

```
[230]: [<matplotlib.lines.Line2D at 0x7f9dab550d90>]
```



```
[228]: # 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:

```
[229]: X_train = X_train.to(device)
y_train = y_train.to(device)
X_test = X_test.to(device)
y_test = y_test.to(device)
y_pred = y_pred.to(device)
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: 84.71428680419922, Testing accuracy: 84.0
```

4 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:

[55]:
```
import torchvision
import torchvision.transforms as transforms

# Preprocessing steps on the training/testing data. You can define your own␣
 ↪data augmentation
# 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 root
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,␣
 ↪download=True, transform=transform_train)
testset = torchvision.datasets.CIFAR10(root='./data', train=False,␣
 ↪download=True, transform=transform_test)

# Addon validation set. We just split the training set into two, where the␣
 ↪validation set is 0.2 * trainset
trainset_len = len(trainset)
valset_len = int(0.2*trainset_len)
valset, reduced_trainset = torch.utils.data.random_split(trainset, [valset_len,␣
 ↪trainset_len-valset_len])
```

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

4.1 2.1 Simple NN [30 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.

[231]:
```
class SimpleNN(nn.Module):

    def __init__(self, feature_size, output_size):
        super().__init__()
        ␣
↪###############################################################################
        # TODO:                                                                 ␣
↪        #
        # Construct your small feedforward NN here.                             ␣
↪        #
        ␣
↪###############################################################################
        inputLayerWidth = feature_size
        hiddenLayerWidth = 64
        outputLayerWidth = output_size

        self.linear1 = nn.Linear(inputLayerWidth, hiddenLayerWidth)
        self.linear2 = nn.Linear(hiddenLayerWidth, hiddenLayerWidth)
        self.linear3 = nn.Linear(hiddenLayerWidth, outputLayerWidth)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)


        ␣
↪###############################################################################
        #                              END OF YOUR CODE                         ␣
↪        #
        ␣
↪###############################################################################

    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 (B,␣
↪C*H*W)
        ␣
↪###############################################################################
```

```
        # TODO:                                                              ␣
↪        #
        # Forward pass, output the prediction score.                         ␣
↪        #
      ␣
↪################################################################################

        # Create ReLU hidden layers, sigmoid output function
        a1 = self.relu(self.linear1(x))
        a2 = self.relu(self.linear2(a1))
        # Note to self: Found out that CE loss function in pytorch already does␣
↪softmax
        # Also, y labels doesn't need to be manually binarized as it already␣
↪does it internally
#         y_pred = self.softmax(self.linear3(a2))
#         return y_pred
        return a2

      ␣
↪################################################################################
        #                                 END OF YOUR CODE                   ␣
↪        #
      ␣
↪################################################################################
```

[233]:
```
epoch = 30
lr = 1e-2
n_input = 3072
n_classes = 10

train_loader = torch.utils.data.DataLoader(trainset, batch_size=64,␣
↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
↪shuffle=False, num_workers=2)

################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################

model = SimpleNN(n_input, n_classes)
model.to(device)
optim = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.8)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist = []
```

```
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:
        x = x.to(device)
        y = y.to(device)
        y_pred = model(x)

        loss = cross_entropy(y_pred, y)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
    loss_hist.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist)

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
Epcoh 0: 1385.4522876739502
Epcoh 1: 1197.1521178483963
Epcoh 2: 1142.03027677536
Epcoh 3: 1103.1839476823807
Epcoh 4: 1078.9604600071907
Epcoh 5: 1050.1283797621727
Epcoh 6: 1028.8153638839722
Epcoh 7: 1010.4932276010513
Epcoh 8: 994.3197276592255
Epcoh 9: 981.4399881958961
Epcoh 10: 963.590448141098
Epcoh 11: 951.1732661724091
Epcoh 12: 942.0032137036324
Epcoh 13: 930.999405503273
Epcoh 14: 919.9791864156723
```

```
     ␣
↪---------------------------------------------------------------------------

      KeyboardInterrupt                         Traceback (most recent call␣
↪last)

      <ipython-input-233-088cd312f7e4> in <module>
       33
       34         optim.zero_grad()
   ---> 35         loss.backward()
       36         optim.step()
       37


      ~/.local/lib/python3.7/site-packages/torch/tensor.py in backward(self,␣
↪gradient, retain_graph, create_graph)
      116                 products. Defaults to ``False``.
      117         """
   --> 118         torch.autograd.backward(self, gradient, retain_graph,␣
↪create_graph)
      119
      120     def register_hook(self, hook):


      ~/.local/lib/python3.7/site-packages/torch/autograd/__init__.py in␣
↪backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables)
       91     Variable._execution_engine.run_backward(
       92         tensors, grad_tensors, retain_graph, create_graph,
   ---> 93         allow_unreachable=True)  # allow_unreachable flag
       94
       95


      KeyboardInterrupt:
```

Now evaluate your model with the helper function:

[202]:
```
def get_model_acc(model, loader):
    ys = []
    y_preds = []
    for x, y in loader:
        x = x.to(device)
        y = y.to(device)
        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)
```

4.1.1 2.1.1 Evaluate NN [30 pts]

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

[59]:
```
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(32873)
tensor(4687)
Training accuracy: 65.7459945678711, Testing accuracy: 46.869998931884766
```

4.2 2.2 Convolutional Neural Network (CNN) [40 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.

4.2.1 2.2.1 Implement and Evaluate CNN [10 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()`.

[75]:
```
class CNNClassifier(nn.Module):

    def __init__(self, n_input, n_classes):
        super().__init__()
       ␣
 ↪#################################################################################
        # TODO:                                                                   ␣
 ↪        #
        # Construct a CNN with 2 or 3 convolutional layers and 1 linear layer␣
 ↪for      #
        # outputing class prediction. You are free to pick the hyperparameters ␣
 ↪        #
```

```
    ␣
↪###########################################################################
        hidden_layer_size = 64

        # In channels = 3 because of RGB,
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=128, kernel_size=3,␣
↪padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=128, out_channels=128,␣
↪kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=128, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.conv4 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=1)


        self.fc1 = nn.Linear(256 * 6 * 6, hidden_layer_size)
        self.fc2 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.fc3 = nn.Linear(hidden_layer_size, n_classes)


        self.relu = nn.ReLU()

    ␣
↪###########################################################################
        #                                  END OF YOUR CODE                  ␣
↪        #
    ␣
↪###########################################################################


    def forward(self, x):
    ␣
↪###########################################################################
        # TODO:                                                              ␣
↪        #
        # Forward pass of your network. First extract feature with CNN, and␣
↪predict    #
        # class scores with linear layer. Be careful about your input/output␣
↪shape.    #
```

```
        ␣
↪################################################################################
#         import pdb; pdb.set_trace()
        x = self.pool1(self.relu(self.conv1(x)))
        x = self.pool2(self.relu(self.conv2(x)))
        x = self.pool3(self.relu(self.conv3(x)))
        x = self.pool4(self.relu(self.conv4(x)))
        x = x.view(-1, 256 * 6 * 6)
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
        ␣
↪################################################################################
        #                                 END OF YOUR CODE                              ␣
↪        #
        ␣
↪################################################################################
```

[235]:
```
# You can tune these hyperparameters as you like.
epoch = 8
lr = 1e-2
n_input = 3072
n_classes = 10
batch_size = 64
num_workers = num_workers

train_loader = torch.utils.data.DataLoader(reduced_trainset, batch_size=64,␣
↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
↪shuffle=False, num_workers=4)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True,␣
↪num_workers=4)

################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################
model = CNNClassifier(n_input, n_classes)
model.to(device)

optim = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.8)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist_train = []
loss_hist_val = []
```

```
val_loader_iterator = iter(val_loader)

for e in range(epoch):
    loss_epoch = 0
    loss_val_epoch = 0

    for x, y in train_loader:
        x = x.to(device)
        y = y.to(device)
        y_pred = y_pred.to(device)
        y_pred = model(x)

        loss = cross_entropy(y_pred, y)

        # This section is just to test against validation set. We don't need to␣
↪keep track of gradients here.
        with torch.no_grad():
            try:
                x_val, y_val = next(val_loader_iterator)
            except StopIteration:
                # Validation set is smaller than training set so we'll run out␣
↪of batches faster, reinstantiate
                # if we do.
                val_loader_iterator = iter(val_loader)
                x_val, y_val = next(val_loader_iterator)

            # Just make a prediction with the validation x and y and get the␣
↪loss
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            y_val_pred = model(x_val)
            loss_val = cross_entropy(y_val_pred, y_val)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
        loss_val_epoch += loss_val.item()

    loss_hist_val.append(loss_val_epoch)
    loss_hist_train.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist_train)
plt.plot(loss_hist_val)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper left')
```

```
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
Epcoh 0: 1162.905691742897


      ␣
 ↪---------------------------------------------------------------------------

        KeyboardInterrupt                         Traceback (most recent call␣
 ↪last)

        <ipython-input-235-cdd3af3eaeed> in <module>
         57         optim.step()
         58
    ---> 59         loss_epoch += loss.item()
         60         loss_val_epoch += loss_val.item()
         61


        KeyboardInterrupt:
```

[117]:
```
# 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␣
 ↪normalize
# your input data. Rememeber to call .train() function after evaluation

# Get the testing accuracy and plot for each number of layers:
test_acc_depth = [0] * 3

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}')

# For plot later
test_acc_depth[1] = test_acc
```

```
tensor(32203)
tensor(7449)
Training accuracy: 80.50749969482422, Testing accuracy: 74.48999786376953
```

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

Answer:

I already played with several different number of layers/padding/kernel size/depth at this point, before realizing the rest of the assignment IS to try these parameters out. So, thank you for letting us try out all these different configurations, it's really interesting.

At this point in the submission I'll just use this architecture, then refine it in the next sections. Currently I have 4 layers of Conv2d followed by pooling after each conv layer. Tried using smaller kernels but more output channels to extract more features early on, maybe if we increase the layers even more that might increase accuracy. Added padding so corner of the images also gets some weight. I follow that with 3 fc layers. Also split the training set into validation set (20%) and training set (80%) then plotted it out so it's easier to do hyperparameter tuning. Using momentum SGD with 0.8 momentum.

4.2.2 2.2.2 STACK MORE LAYERS [10 pts]

Now, **try at least 4 network architectures with different numbers of convolutional layers**. Train these settings with `optim.SGD`, plot the accuracy as a fuction of convolutional layers and describe what you have observed (running time, performance, etc).

[236]:
```
# For 6 layer conv network, I also tried adding dropout regularization because␣
↪as you can see from the
# previous 4 layer CNN, after epoch 5 validation error goes up while training␣
↪error goes down - we're overfitting.
# After we plot the 2 layer CNN, we'll try adding some other regularization,␣
↪and maybe some batch norm layers.
class CNNClassifier_6(nn.Module):

    def __init__(self, n_input, n_classes):
        super().__init__()
        ␣
↪#############################################################################
        # TODO:                                                                  ␣
↪        #
        # Construct a CNN with 2 or 3 convolutional layers and 1 linear layer␣
↪for      #
        # outputing class prediction. You are free to pick the hyperparameters ␣
↪        #
        ␣
↪#############################################################################
        hidden_layer_size = 64

        # In channels = 3 because of RGB,
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=128, kernel_size=3,␣
↪padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=128, out_channels=128,␣
↪kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv3 = nn.Conv2d(in_channels=128, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.conv4 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.conv5 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.conv6 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.pool6 = nn.MaxPool2d(kernel_size=2, stride=1)


        self.fc1 = nn.Linear(256 * 4 * 4, hidden_layer_size)
        self.fc2 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.fc3 = nn.Linear(hidden_layer_size, n_classes)

        self.dropout = nn.Dropout(p=0.2)
        self.relu = nn.ReLU()

        ␣
↪################################################################################
        #                                  END OF YOUR CODE                       ␣
↪        #
        ␣
↪################################################################################


    def forward(self, x):
        ␣
↪################################################################################
        # TODO:                                                                   ␣
↪        #
        # Forward pass of your network. First extract feature with CNN, and␣
↪predict    #
        # class scores with linear layer. Be careful about your input/output␣
↪shape.    #
        ␣
↪################################################################################
#          import pdb; pdb.set_trace()
        x = self.pool1(self.relu(self.conv1(x)))
        x = self.pool2(self.relu(self.conv2(x)))
        x = self.pool3(self.relu(self.conv3(x)))
        x = self.pool4(self.relu(self.conv4(x)))
        x = self.pool5(self.relu(self.conv5(x)))
        x = self.pool6(self.relu(self.conv6(x)))
        x = x.view(-1, 256 * 4 * 4)
        x = self.relu(self.dropout(self.fc1(x)))
        x = self.relu(self.dropout(self.fc2(x)))
        x = self.dropout(self.fc3(x))
        return x
       ␣
 ↪################################################################################
        #                                  END OF YOUR CODE                     ␣
 ↪        #
       ␣
 ↪################################################################################
```

[237]:
```
# Run 6 layer CNN
epoch = 8
lr = 1e-2
n_input = 3072
n_classes = 10
batch_size = 64
num_workers = num_workers

train_loader = torch.utils.data.DataLoader(reduced_trainset, batch_size=64,␣
 ↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
 ↪shuffle=False, num_workers=4)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True,␣
 ↪num_workers=4)

################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################
model6 = CNNClassifier_6(n_input, n_classes)
model6.to(device)

# Initialize weights with Kaiming, just curious to try. It's not very deep so␣
 ↪shouldn't have impact.:
def init_weights_kaiming(m):
    if type(m) == nn.Linear:
        torch.nn.init.kaiming_normal_(m.weight)

model6.apply(init_weights_kaiming)
```

```
optim = torch.optim.SGD(model6.parameters(), lr=lr, momentum=0.8)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist_train = []
loss_hist_val = []
val_loader_iterator = iter(val_loader)

for e in range(epoch):
    loss_epoch = 0
    loss_val_epoch = 0

    for x, y in train_loader:
        x = x.to(device)
        y = y.to(device)
        y_pred = model6(x)

        loss = cross_entropy(y_pred, y)

        # This section is just to test against validation set. We don't need to␣
 ↪keep track of gradients here.
        with torch.no_grad():
            try:
                x_val, y_val = next(val_loader_iterator)
            except StopIteration:
                # Validation set is smaller than training set so we'll run out␣
 ↪of batches faster, reinstantiate
                # if we do.
                val_loader_iterator = iter(val_loader)
                x_val, y_val = next(val_loader_iterator)

            # Just make a prediction with the validation x and y and get the␣
 ↪loss
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            y_val_pred = model6(x_val)
            loss_val = cross_entropy(y_val_pred, y_val)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
        loss_val_epoch += loss_val.item()

    loss_hist_val.append(loss_val_epoch)
    loss_hist_train.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist_train)
plt.plot(loss_hist_val)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper left')

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
Epcoh 0: 1261.7796112298965


       ␣
 ↪---------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
 ↪last)

       <ipython-input-237-88d7644d56bd> in <module>
        63         optim.step()
        64
   ---> 65         loss_epoch += loss.item()
        66         loss_val_epoch += loss_val.item()
        67


       KeyboardInterrupt:
```

[118]:
```
# Evaluate 6 layer CNN
model6.eval()
train_acc = get_model_acc(model6, train_loader)
test_acc = get_model_acc(model6, test_loader)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
# For plot later
test_acc_depth[2] = test_acc
```

```
tensor(34082)
tensor(7661)
Training accuracy: 85.20500183105469, Testing accuracy: 76.61000061035156
```

[238]:
```
# Just do a 2 layer network for the simple plot with relation to depth of CNN␣
 ↪layers.
class CNNClassifier_2(nn.Module):

    def __init__(self, n_input, n_classes):
        super().__init__()
```

```
      ␣
↪################################################################################
        # TODO:                                                                   ␣
↪        #
        # Construct a CNN with 2 or 3 convolutional layers and 1 linear layer␣
↪for       #
        # outputing class prediction. You are free to pick the hyperparameters ␣
↪        #
      ␣
↪################################################################################
        hidden_layer_size = 64

        # In channels = 3 because of RGB,
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=128, kernel_size=3,␣
↪padding=1)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=128, out_channels=128,␣
↪kernel_size=3, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(128 * 8 * 8, hidden_layer_size)
        self.fc2 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.fc3 = nn.Linear(hidden_layer_size, n_classes)

        self.dropout = nn.Dropout(p=0.2)
        self.relu = nn.ReLU()


      ␣
↪################################################################################
        #                                 END OF YOUR CODE                        ␣
↪        #
      ␣
↪################################################################################


    def forward(self, x):
      ␣
↪################################################################################
        # TODO:                                                                   ␣
↪        #
        # Forward pass of your network. First extract feature with CNN, and␣
↪predict     #
        # class scores with linear layer. Be careful about your input/output␣
↪shape.     #
```

```
        ␣
↪##################################################################################
#         import pdb; pdb.set_trace()
        x = self.pool1(self.relu(self.conv1(x)))
        x = self.pool2(self.relu(self.conv2(x)))
        x = x.view(-1, 128 * 8 * 8)
        x = self.relu(self.dropout(self.fc1(x)))
        x = self.relu(self.dropout(self.fc2(x)))
        x = self.dropout(self.fc3(x))
        return x
        ␣
↪##################################################################################
        #                                END OF YOUR CODE                                ␣
↪        #
        ␣
↪##################################################################################
```

[239]:
```
# Run 2 layer CNN
epoch = 8
lr = 1e-2
n_input = 3072
n_classes = 10
batch_size = 64
num_workers = num_workers

train_loader = torch.utils.data.DataLoader(reduced_trainset, batch_size=64,␣
↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
↪shuffle=False, num_workers=4)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True,␣
↪num_workers=4)

################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################
model2 = CNNClassifier_2(n_input, n_classes)
model2.to(device)

optim = torch.optim.SGD(model2.parameters(), lr=lr, momentum=0.8)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist_train = []
loss_hist_val = []
val_loader_iterator = iter(val_loader)
```

```
for e in range(epoch):
    loss_epoch = 0
    loss_val_epoch = 0

    for x, y in train_loader:
        x = x.to(device)
        y = y.to(device)
        y_pred = model2(x)

        loss = cross_entropy(y_pred, y)

        # This section is just to test against validation set. We don't need to␣
 ↪keep track of gradients here.
        with torch.no_grad():
            try:
                x_val, y_val = next(val_loader_iterator)
            except StopIteration:
                # Validation set is smaller than training set so we'll run out␣
 ↪of batches faster, reinstantiate
                # if we do.
                val_loader_iterator = iter(val_loader)
                x_val, y_val = next(val_loader_iterator)

            # Just make a prediction with the validation x and y and get the␣
 ↪loss
            y_val = y_val.to(device)
            x_val = x_val.to(device)
            y_val_pred = model2(x_val)
            loss_val = cross_entropy(y_val_pred, y_val)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
        loss_val_epoch += loss_val.item()

    loss_hist_val.append(loss_val_epoch)
    loss_hist_train.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist_train)
plt.plot(loss_hist_val)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper left')

################################################################################
#                                 END OF YOUR CODE                             #
```

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

```
Epcoh 0: 1175.075127005577
```

```
      ␣
 ↪---------------------------------------------------------------------------

        KeyboardInterrupt                         Traceback (most recent call␣
 ↪last)

        <ipython-input-239-6e8f503a3b63> in <module>
         56         optim.step()
         57
    ---> 58         loss_epoch += loss.item()
         59         loss_val_epoch += loss_val.item()
         60


        KeyboardInterrupt:
```

[126]:
```
# Evaluate 2 layer CNN (2 conv layers)
model2.eval()
train_acc = get_model_acc(model2, train_loader)
test_acc = get_model_acc(model2, test_loader)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
# For plot later
test_acc_depth[2] = test_acc
```

```
tensor(32817)
tensor(7153)
Training accuracy: 82.04249572753906, Testing accuracy: 71.52999877929688
```

[240]:
```
# Trying to optimize with batch norm in the 6 layer network.

class CNNClassifier_6_batchnorm(nn.Module):

    def __init__(self, n_input, n_classes):
        super().__init__()
       ␣
 ↪################################################################################
        # TODO:                                                                  ␣
 ↪         #
        # Construct a CNN with 2 or 3 convolutional layers and 1 linear layer␣
 ↪for      #
```

```
        # outputing class prediction. You are free to pick the hyperparameters ␣
↪        #
      ␣
↪################################################################################
        hidden_layer_size = 512

        # In channels = 3 because of RGB,
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=128, kernel_size=3,␣
↪padding=1)
        self.bn1   = nn.BatchNorm2d(128)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=128, out_channels=128,␣
↪kernel_size=3, padding=1)
        self.bn2   = nn.BatchNorm2d(128)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv3 = nn.Conv2d(in_channels=128, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn3   = nn.BatchNorm2d(256)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.conv4 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn4   = nn.BatchNorm2d(256)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=1)


        self.conv5 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn5   = nn.BatchNorm2d(256)
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.conv6 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn6   = nn.BatchNorm2d(256)
        self.pool6 = nn.MaxPool2d(kernel_size=2, stride=1)


        self.fc1 = nn.Linear(256 * 4 * 4, hidden_layer_size)
        self.fc2 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.fc3 = nn.Linear(hidden_layer_size, n_classes)

        self.dropout = nn.Dropout(p=0.2)
        self.relu = nn.ReLU()
        ␣
↪###########################################################################
        #                                 END OF YOUR CODE                                  ␣
↪        #
        ␣
↪###########################################################################


    def forward(self, x):
        ␣
↪###########################################################################
        # TODO:                                                                             ␣
↪        #
        # Forward pass of your network. First extract feature with CNN, and␣
↪predict    #
        # class scores with linear layer. Be careful about your input/output␣
↪shape.    #
        ␣
↪###########################################################################
#          import pdb; pdb.set_trace()
        x = self.pool1(self.relu(self.bn1(self.conv1(x))))
        x = self.pool2(self.relu(self.bn2(self.conv2(x))))
        x = self.pool3(self.relu(self.bn3(self.conv3(x))))
        x = self.pool4(self.relu(self.bn4(self.conv4(x))))
        x = self.pool5(self.relu(self.bn5(self.conv5(x))))
        x = self.pool6(self.relu(self.bn6(self.conv6(x))))
        x = x.view(-1, 256 * 4 * 4)
        x = self.relu(self.dropout(self.fc1(x)))
        x = self.relu(self.dropout(self.fc2(x)))
        x = self.dropout(self.fc3(x))
        return x
        ␣
↪###########################################################################
        #                                 END OF YOUR CODE                                  ␣
↪        #
        ␣
↪###########################################################################
```

[241]:
```
# Run 6 layer CNN
epoch = 12
lr = 1e-2
n_input = 3072
n_classes = 10
batch_size = 64
num_workers = num_workers
```

```
train_loader = torch.utils.data.DataLoader(reduced_trainset, batch_size=64,␣
↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
↪shuffle=False, num_workers=4)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True,␣
↪num_workers=4)

################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################
model6n = CNNClassifier_6_batchnorm(n_input, n_classes)
model6n.to(device)

# Initialize weights with Kaiming, just curious to try. It's not very deep so␣
↪shouldn't have impact.:
def init_weights_kaiming(m):
    if type(m) == nn.Linear:
        torch.nn.init.kaiming_normal_(m.weight)

model6n.apply(init_weights_kaiming)

optim = torch.optim.SGD(model6n.parameters(), lr=lr, momentum=0.8)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist_train = []
loss_hist_val = []
val_loader_iterator = iter(val_loader)

for e in range(epoch):
    loss_epoch = 0
    loss_val_epoch = 0

    for x, y in train_loader:
        x = x.to(device)
        y = y.to(device)
        y_pred = model6n(x)

        loss = cross_entropy(y_pred, y)

        # This section is just to test against validation set. We don't need to␣
↪keep track of gradients here.
        with torch.no_grad():
            try:
                x_val, y_val = next(val_loader_iterator)
            except StopIteration:
                # Validation set is smaller than training set so we'll run out␣
↪of batches faster, reinstantiate
                # if we do.
                val_loader_iterator = iter(val_loader)
                x_val, y_val = next(val_loader_iterator)

            # Just make a prediction with the validation x and y and get the␣
↪loss
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            y_val_pred = model6n(x_val)
            loss_val = cross_entropy(y_val_pred, y_val)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
        loss_val_epoch += loss_val.item()

    loss_hist_val.append(loss_val_epoch)
    loss_hist_train.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist_train)
plt.plot(loss_hist_val)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper right')

################################################################################
#                                 END OF YOUR CODE                             #
################################################################################
```

```
      ␣
↪---------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
↪last)

       <ipython-input-241-5d1f2430238f> in <module>
        63         optim.step()
        64
   ---> 65         loss_epoch += loss.item()
        66         loss_val_epoch += loss_val.item()
        67
        KeyboardInterrupt:
```

```
[145]: # Evaluate 6 layer modified CNN, this is the one with batch Norm
       model6n.eval()
       train_acc = get_model_acc(model6n, train_loader)
       test_acc = get_model_acc(model6n, test_loader)
       print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
```

```
tensor(37665)
tensor(8155)
Training accuracy: 94.1624984741211, Testing accuracy: 81.55000305175781
```

```
[ ]: # What if we follow known architectures and stack a few Conv layers before we␣
      ↪actually do any max pooling?
     # Add on to the architecture that already has batch norm and dropout with 6␣
      ↪conv layers.

     # WARNING: THIS MODEL TOOK THE WHOLE NIGHT TO RUN EVEN WITH CUDA (granted, only␣
      ↪using GTX 960M)

     class CNNClassifier_6_modified(nn.Module):

         def __init__(self, n_input, n_classes):
             super().__init__()
            ␣
      ↪##############################################################################
             # TODO:                                                                  ␣
      ↪        #
             # Construct a CNN with 2 or 3 convolutional layers and 1 linear layer␣
      ↪for       #
             # outputing class prediction. You are free to pick the hyperparameters ␣
      ↪        #
            ␣
      ↪##############################################################################
             hidden_layer_size = 256

             # In channels = 3 because of RGB,
             self.conv1 = nn.Conv2d(in_channels=3, out_channels=128, kernel_size=3,␣
      ↪padding=1)
             self.bn1   = nn.BatchNorm2d(128)

             self.conv2 = nn.Conv2d(in_channels=128, out_channels=128,␣
      ↪kernel_size=3, padding=1)
             self.bn2   = nn.BatchNorm2d(128)
```

```
        self.conv3 = nn.Conv2d(in_channels=128, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn3   = nn.BatchNorm2d(256)

        self.conv4 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn4   = nn.BatchNorm2d(256)

        self.conv5 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn5   = nn.BatchNorm2d(256)

        self.conv6 = nn.Conv2d(in_channels=256, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn6   = nn.BatchNorm2d(256)

        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.fc1 = nn.Linear(256 * 8 * 8, hidden_layer_size)
        self.fc2 = nn.Linear(hidden_layer_size, hidden_layer_size)
        self.fc3 = nn.Linear(hidden_layer_size, n_classes)

        self.dropout = nn.Dropout(p=0.2)
        self.relu = nn.ReLU()

      ␣
↪################################################################################
        #                                 END OF YOUR CODE                          ␣
↪        #
      ␣
↪################################################################################


    def forward(self, x):
      ␣
↪################################################################################
        # TODO:                                                                     ␣
↪        #
        # Forward pass of your network. First extract feature with CNN, and␣
↪predict    #
        # class scores with linear layer. Be careful about your input/output␣
↪shape.    #
      ␣
↪################################################################################
#         import pdb; pdb.set_trace()
```

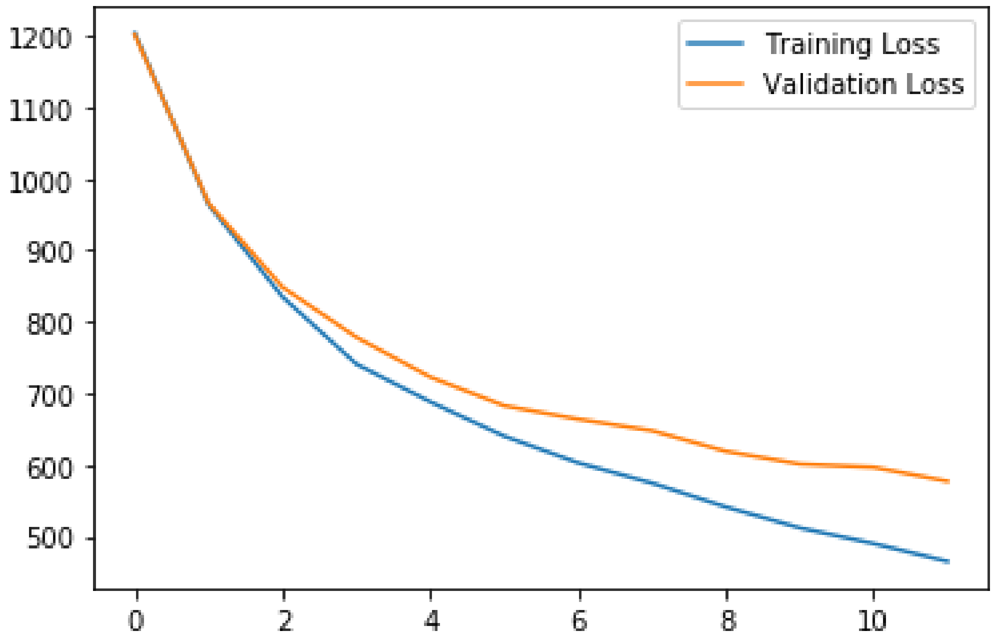
```
        x = self.relu(self.bn1(self.conv1(x)))
        x = self.relu(self.bn2(self.conv2(x)))
        x = self.pool1(self.relu(self.bn3(self.conv3(x))))
        x = self.relu(self.bn4(self.conv4(x)))
        x = self.relu(self.bn5(self.conv5(x)))
        x = self.pool2(self.relu(self.bn6(self.conv6(x))))
        x = x.view(-1, 256 * 8 * 8)
        x = self.relu(self.dropout(self.fc1(x)))
        x = self.relu(self.dropout(self.fc2(x)))
        x = self.dropout(self.fc3(x))
        return x
       ␣
 ↪###############################################################################
        #                                END OF YOUR CODE                       ␣
 ↪       #
       ␣
 ↪###############################################################################
```

[147]:
```
# Run modified 6 layer CNN. WARNING, takes a few hours to run.

epoch = 12
lr = 1e-2
n_input = 3072
n_classes = 10
batch_size = 64
num_workers = num_workers

train_loader = torch.utils.data.DataLoader(reduced_trainset, batch_size=64,␣
 ↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
 ↪shuffle=False, num_workers=4)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True,␣
 ↪num_workers=4)

################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################
model6m = CNNClassifier_6_modified(n_input, n_classes)
model6m.to(device)

# Initialize weights with Kaiming, just curious to try. It's not very deep so␣
 ↪shouldn't have impact.:
def init_weights_kaiming(m):
    if type(m) == nn.Linear:
        torch.nn.init.kaiming_normal_(m.weight)
```

```
model6m.apply(init_weights_kaiming)

optim = torch.optim.SGD(model6m.parameters(), lr=lr, momentum=0.8)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist_train = []
loss_hist_val = []
val_loader_iterator = iter(val_loader)

for e in range(epoch):
    loss_epoch = 0
    loss_val_epoch = 0

    for x, y in train_loader:
        x = x.to(device)
        y = y.to(device)
        y_pred = model6m(x)

        loss = cross_entropy(y_pred, y)

        # This section is just to test against validation set. We don't need to␣
 ↪keep track of gradients here.
        with torch.no_grad():
            try:
                x_val, y_val = next(val_loader_iterator)
            except StopIteration:
                # Validation set is smaller than training set so we'll run out␣
 ↪of batches faster, reinstantiate
                # if we do.
                val_loader_iterator = iter(val_loader)
                x_val, y_val = next(val_loader_iterator)

            # Just make a prediction with the validation x and y and get the␣
 ↪loss
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            y_val_pred = model6m(x_val)
            loss_val = cross_entropy(y_val_pred, y_val)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
        loss_val_epoch += loss_val.item()
    loss_hist_val.append(loss_val_epoch)
    loss_hist_train.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist_train)
plt.plot(loss_hist_val)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper right')

################################################################################
#                                 END OF YOUR CODE                             #
################################################################################
```

```
Epcoh 0: 1203.559067606926
Epcoh 1: 963.6857343912125
Epcoh 2: 834.79960334301
Epcoh 3: 740.9108527302742
Epcoh 4: 688.0639671087265
Epcoh 5: 640.2783396244049
Epcoh 6: 602.9625856876373
Epcoh 7: 574.5812652111053
Epcoh 8: 541.1765455007553
Epcoh 9: 511.8699654340744
Epcoh 10: 489.89281487464905
Epcoh 11: 464.91505831480026
```

```
[147]: <matplotlib.legend.Legend at 0x7f9daad327d0>
```

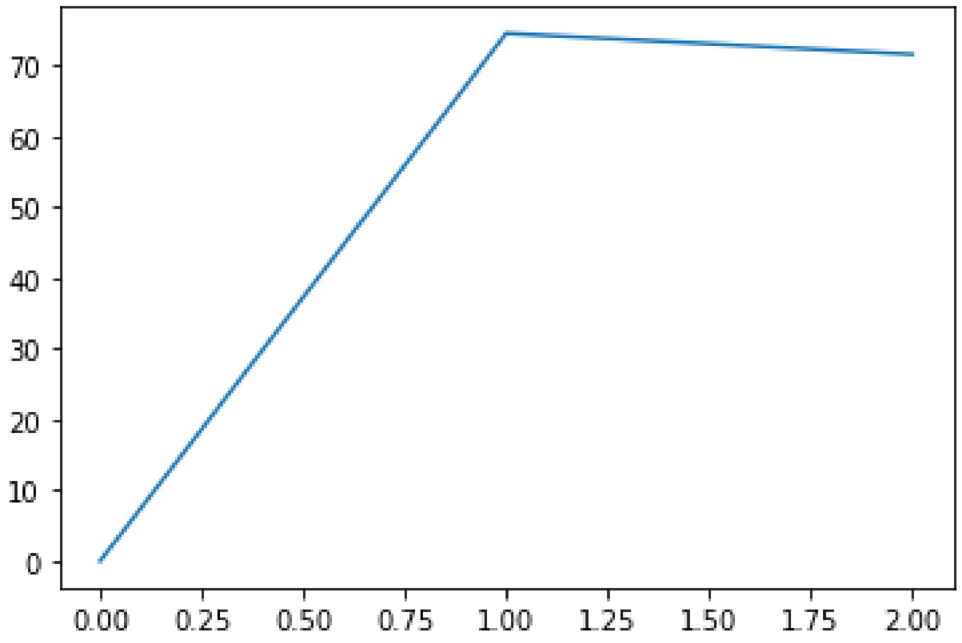


```
[148]: # Evaluate 6 layer modified CNN, with fewer pooling layers
       model6m.eval()
       train_acc = get_model_acc(model6m, train_loader)
       test_acc = get_model_acc(model6m, test_loader)
       print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
```

```
tensor(35477)
tensor(8079)
Training accuracy: 88.69249725341797, Testing accuracy: 80.79000091552734
```

```
[157]: # Ok, let's plot the test accuracy over the depth of the neural network
       plt.plot(test_acc_depth)
```

```
[157]: [<matplotlib.lines.Line2D at 0x7f9dab1a20d0>]
```



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

Answer: Stacking layers makes it take longer to train. After a certain point it doesn't necessarily give better results until we perform batch norm on the network. I also noticed that when using fewer pooling layers (2 as opposed to 6) it takes even longer to train. Currently with 6 conv layers, batch norm and dropout we're getting around 81% accuracy.

4.2.3 2.2.3 Optimizer? Optimizer! [10 pts]

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

[253]:
```
################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################

# Reuse our batch-normed 6 layer CNN class. Try different optimizers. First␣
↪let's try Adagrad

epoch = 35
lr = 1e-2
n_input = 3072
n_classes = 10
batch_size = 64
num_workers = num_workers

train_loader = torch.utils.data.DataLoader(reduced_trainset, batch_size=64,␣
↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
↪shuffle=False, num_workers=4)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True,␣
↪num_workers=4)

model6m = CNNClassifier_6_batchnorm(n_input, n_classes)
model6m.to(device)

# Initialize weights with Kaiming, just curious to try. It's not very deep so␣
↪shouldn't have impact.:
def init_weights_kaiming(m):
    if type(m) == nn.Linear:
        torch.nn.init.kaiming_normal_(m.weight)

model6m.apply(init_weights_kaiming)

optim = torch.optim.Adagrad(model6m.parameters(), lr=lr, weight_decay=0.2)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist_train = []
loss_hist_val = []
val_loader_iterator = iter(val_loader)

for e in range(epoch):
    loss_epoch = 0
    loss_val_epoch = 0

    for x, y in train_loader:
        x = x.to(device)
        y = y.to(device)
        y_pred = y_pred.to(device)
        y_pred = model6m(x)

        loss = cross_entropy(y_pred, y)

        # This section is just to test against validation set. We don't need to␣
↪keep track of gradients here.
        with torch.no_grad():
            try:
                x_val, y_val = next(val_loader_iterator)
            except StopIteration:
                # Validation set is smaller than training set so we'll run out␣
↪of batches faster, reinstantiate
                # if we do.
                val_loader_iterator = iter(val_loader)
                x_val, y_val = next(val_loader_iterator)

            # Just make a prediction with the validation x and y and get the␣
↪loss
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            y_val_pred = model6m(x_val)
            loss_val = cross_entropy(y_val_pred, y_val)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
        loss_val_epoch += loss_val.item()

    loss_hist_val.append(loss_val_epoch)
    loss_hist_train.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist_train)
plt.plot(loss_hist_val)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper right')

################################################################################
#                                  END OF YOUR CODE                            #
################################################################################
```

```
      ␣
↪---------------------------------------------------------------------------
```

```
        KeyboardInterrupt                         Traceback (most recent call␣
↪last)

        <ipython-input-253-7068e8879fc6> in <module>
         57
         58             # Just make a prediction with the validation x and y and␣
↪get the loss
    ---> 59             x_val = x_val.to(device)
         60             y_val = y_val.to(device)
         61             y_val_pred = model6m(x_val)


        KeyboardInterrupt:
```

[252]:
```
# Evaluate Adam
model6m.eval()
train_acc = get_model_acc(model6m, train_loader)
test_acc = get_model_acc(model6m, test_loader)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
```

```
tensor(17807, device='cuda:0')
tensor(4414, device='cuda:0')
Training accuracy: 44.51749801635742, Testing accuracy: 44.13999938964844
```

[205]:
```
################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################

# Reuse our modified 6 layer CNN class. Try different optimizers. Next, let's␣
 ↪try nesterov momentum!

epoch = 10
lr = 1e-2
n_input = 3072
n_classes = 10
batch_size = 64
num_workers = num_workers

train_loader = torch.utils.data.DataLoader(reduced_trainset, batch_size=64,␣
 ↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
 ↪shuffle=False, num_workers=4)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True,␣
 ↪num_workers=4)
```

```
model6_nesterov = CNNClassifier_6_batchnorm(n_input, n_classes)
model6.to(device)

# Initialize weights with Kaiming, just curious to try. It's not very deep so␣
↪shouldn't have impact.:
def init_weights_kaiming(m):
    if type(m) == nn.Linear:
        torch.nn.init.kaiming_normal_(m.weight)

model6_nesterov.apply(init_weights_kaiming)

optim = torch.optim.SGD(model6_nesterov.parameters(), lr=lr, momentum=0.8,␣
↪nesterov=True)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist_train = []
loss_hist_val = []
val_loader_iterator = iter(val_loader)

for e in range(epoch):
    loss_epoch = 0
    loss_val_epoch = 0

    for x, y in train_loader:
        x = x.to(device)
        y = y.to(device)
        y_pred = y_pred.to(device)
        y_pred = model6_nesterov(x)

        loss = cross_entropy(y_pred, y)

        # This section is just to test against validation set. We don't need to␣
↪keep track of gradients here.
        with torch.no_grad():
            try:
                x_val, y_val = next(val_loader_iterator)
            except StopIteration:
                # Validation set is smaller than training set so we'll run out␣
↪of batches faster, reinstantiate
                # if we do.
                val_loader_iterator = iter(val_loader)
                x_val, y_val = next(val_loader_iterator)

            # Just make a prediction with the validation x and y and get the␣
↪loss
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            y_val_pred = model6_nesterov(x_val)
            loss_val = cross_entropy(y_val_pred, y_val)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
        loss_val_epoch += loss_val.item()

    loss_hist_val.append(loss_val_epoch)
    loss_hist_train.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist_train)
plt.plot(loss_hist_val)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper right')

################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
Epcoh 0: 1056.286468744278


       ␣
 ↪---------------------------------------------------------------------------

       KeyboardInterrupt                         Traceback (most recent call␣
 ↪last)

       <ipython-input-205-5dad26ea8b5b> in <module>
        58
        59         optim.zero_grad()
   ---> 60         loss.backward()
        61         optim.step()
        62


       ~/.local/lib/python3.7/site-packages/torch/tensor.py in backward(self,␣
 ↪gradient, retain_graph, create_graph)
       116                 products. Defaults to ``False``.
       117         """
   --> 118         torch.autograd.backward(self, gradient, retain_graph,␣
 ↪create_graph)
       119
       120     def register_hook(self, hook):
```

```
    ~/.local/lib/python3.7/site-packages/torch/autograd/__init__.py in␣
↪backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables)
         91     Variable._execution_engine.run_backward(
         92         tensors, grad_tensors, retain_graph, create_graph,
    ---> 93         allow_unreachable=True)  # allow_unreachable flag
         94
         95


    KeyboardInterrupt:
```

[ ]:
```
# Evaluate nesterov momentum
model6_nesterov.eval()
train_acc = get_model_acc(model6_nesterov, train_loader)
test_acc = get_model_acc(model6_nesterov, test_loader)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
```

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

Answer: I tried SGD with nesterov momentum and Adagrad. Adagrad takes a very long time for loss to start decreasing while SGD with momentum decreases very fast. My favorite is SGD with nesterov momentum because it seems to yield better results in fewer training epochs.

4.2.4 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).

[244]:
```
# I wanted to try Alexnet or VGG16, then realized it takes days to train...
↪let's just reuse and tweak our
# previous 6 layer conv net.

class CNNClassifier_6_final(nn.Module):

    def __init__(self, n_input, n_classes):
        super().__init__()
       ␣
↪##############################################################################
        # TODO:                                                               ␣
↪         #
        # Construct a CNN with 2 or 3 convolutional layers and 1 linear layer␣
↪for       #
```

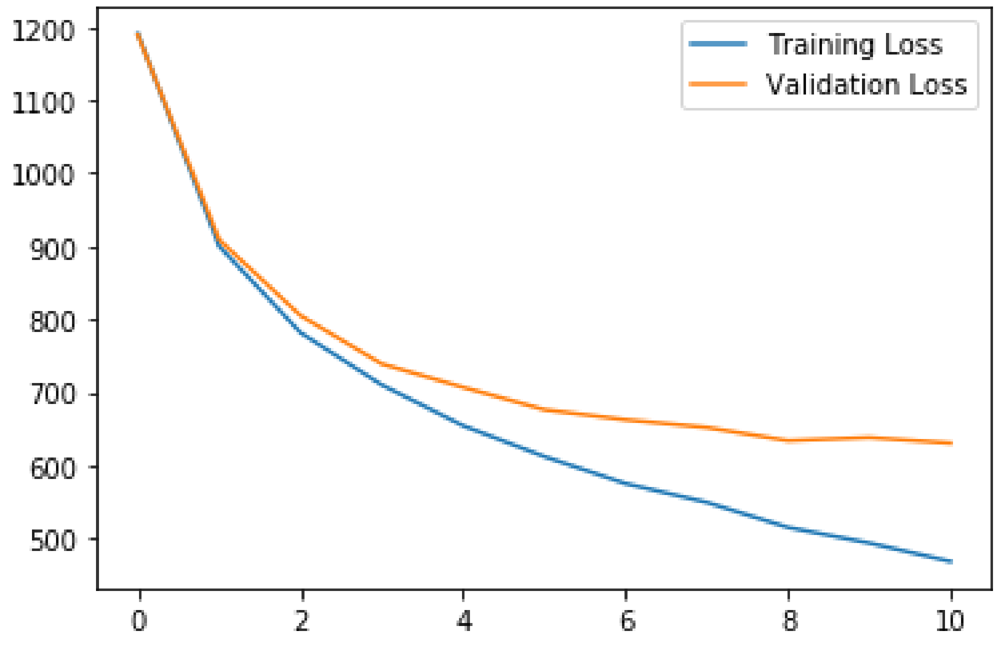
```
        # outputing class prediction. You are free to pick the hyperparameters ␣
↪        #
      ␣
↪################################################################################
        hidden_layer_size = 1024

        # In channels = 3 because of RGB, let's try similar to VGG and Alex␣
↪where we use fewer filters first.

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3,␣
↪padding=1)
        self.bn1   = nn.BatchNorm2d(64)
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv2 = nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3,␣
↪padding=1)
        self.bn2   = nn.BatchNorm2d(64)
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,␣
↪padding=1)
        self.bn3   = nn.BatchNorm2d(128)
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv4 = nn.Conv2d(in_channels=128, out_channels=128,␣
↪kernel_size=3, padding=1)
        self.bn4   = nn.BatchNorm2d(128)
        self.pool4 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.conv5 = nn.Conv2d(in_channels=128, out_channels=256,␣
↪kernel_size=3, padding=1)
        self.bn5   = nn.BatchNorm2d(256)
        self.pool5 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.conv6 = nn.Conv2d(in_channels=256, out_channels=512,␣
↪kernel_size=3, padding=1)
        self.bn6   = nn.BatchNorm2d(512)
        self.pool6 = nn.MaxPool2d(kernel_size=2, stride=1)

        self.fc1 = nn.Linear(512 * 2 * 2, hidden_layer_size)
        self.fc2 = nn.Linear(hidden_layer_size, hidden_layer_size//2)
        self.fc3 = nn.Linear(hidden_layer_size//2, n_classes)

        self.dropout = nn.Dropout(p=0.2)
        self.relu = nn.ReLU()
        ␣
↪##################################################################################
        #                                  END OF YOUR CODE                        ␣
↪        #
        ␣
↪##################################################################################


    def forward(self, x):
        ␣
↪##################################################################################
        # TODO:                                                                    ␣
↪        #
        # Forward pass of your network. First extract feature with CNN, and␣
↪predict     #
        # class scores with linear layer. Be careful about your input/output␣
↪shape.     #
        ␣
↪##################################################################################
#         import pdb; pdb.set_trace()
        x = self.pool1(self.relu(self.bn1(self.conv1(x))))
        x = self.pool2(self.relu(self.bn2(self.conv2(x))))
        x = self.pool3(self.relu(self.bn3(self.conv3(x))))
        x = self.pool4(self.relu(self.bn4(self.conv4(x))))
        x = self.pool5(self.relu(self.bn5(self.conv5(x))))
        x = self.pool6(self.relu(self.bn6(self.conv6(x))))

#         x = self.pool6(self.relu(self.bn6(self.conv6(x))))

        x = x.view(-1, 512 * 2 * 2)
        x = self.relu(self.dropout(self.fc1(x)))
        x = self.relu(self.dropout(self.fc2(x)))
        x = self.dropout(self.fc3(x))
        return x
        ␣
↪##################################################################################
        #                                  END OF YOUR CODE                        ␣
↪        #
        ␣
↪##################################################################################
```

[245]:
```
# Run our finalized 6 layer CNN. Might take a long time to run.

epoch = 20
lr = 1e-2
n_input = 3072
```

```
n_classes = 10
batch_size = 64
num_workers = num_workers

train_loader = torch.utils.data.DataLoader(reduced_trainset, batch_size=64,␣
 ↪shuffle=True, num_workers=4)
test_loader = torch.utils.data.DataLoader(testset, batch_size=100,␣
 ↪shuffle=False, num_workers=4)
val_loader = torch.utils.data.DataLoader(valset, batch_size=64, shuffle=True,␣
 ↪num_workers=4)

################################################################################
# TODO:                                                                        #
# Your training code here.                                                     #
################################################################################
model6f = CNNClassifier_6_final(n_input, n_classes)
model6f.cuda()

# Initialize weights with Kaiming, just curious to try. It's not very deep so␣
 ↪shouldn't have impact.:
def init_weights_kaiming(m):
    if type(m) == nn.Linear:
        torch.nn.init.kaiming_normal_(m.weight)

model6f.apply(init_weights_kaiming)

optim = torch.optim.SGD(model6f.parameters(), lr=lr, momentum=0.8,␣
 ↪nesterov=True)
cross_entropy = nn.CrossEntropyLoss(reduction='mean')

loss_hist_train = []
loss_hist_val = []
val_loader_iterator = iter(val_loader)

for e in range(epoch):
    loss_epoch = 0
    loss_val_epoch = 0

    for x, y in train_loader:
        x = x.to(device)
        y = y.to(device)
        y_pred = y_pred.to(device)

        y_pred = model6f(x)

        loss = cross_entropy(y_pred, y)
        # This section is just to test against validation set. We don't need to␣
↪keep track of gradients here.
        with torch.no_grad():
            try:
                x_val, y_val = next(val_loader_iterator)
            except StopIteration:
                # Validation set is smaller than training set so we'll run out␣
↪of batches faster, reinstantiate
                # if we do.
                val_loader_iterator = iter(val_loader)
                x_val, y_val = next(val_loader_iterator)

            # Just make a prediction with the validation x and y and get the␣
↪loss
            x_val = x_val.to(device)
            y_val = y_val.to(device)
            y_val_pred = model6f(x_val)
            loss_val = cross_entropy(y_val_pred, y_val)

        optim.zero_grad()
        loss.backward()
        optim.step()

        loss_epoch += loss.item()
        loss_val_epoch += loss_val.item()

    loss_hist_val.append(loss_val_epoch)
    loss_hist_train.append(loss_epoch)
    print(f'Epcoh {e}: {loss_epoch}')

plt.plot(loss_hist_train)
plt.plot(loss_hist_val)
plt.legend(['Training Loss', 'Validation Loss'], loc='upper right')

################################################################################
#                                 END OF YOUR CODE                             #
################################################################################
```

```
Epcoh 0: 1190.6327819824219
Epcoh 1: 900.2163406610489
Epcoh 2: 781.8112269043922
Epcoh 3: 710.259522497654
Epcoh 4: 654.5348156690598
Epcoh 5: 612.2905205488205
Epcoh 6: 575.232914686203
Epcoh 7: 549.4927686452866
Epcoh 8: 515.1297890543938
Epcoh 9: 493.7614367902279
Epcoh 10: 468.4654134809971
```

[245]: <matplotlib.legend.Legend at 0x7f9dab70b9d0>



[246]:
```
# Evaluate final
model6f.eval()
train_acc = get_model_acc(model6f, train_loader)
test_acc = get_model_acc(model6f, test_loader)
print(f'Training accuracy: {train_acc}, Testing accuracy: {test_acc}')
```

```
tensor(35465, device='cuda:0')
tensor(7850, device='cuda:0')
Training accuracy: 88.6624984741211, Testing accuracy: 78.5
```

[ ]:
```

```