In [1]:

```
%matplotlib inline

import torch
import torchvision
import torchvision.transforms as transforms

import matplotlib.pyplot as plt
import numpy as np

import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import random
import numpy as np

import os

from shutil import copy2
```

In [2]:

```
in_submission = os.path.exists('/flags/isgrader.flag')
perform_computation = not in_submission

if in_submission:
    assert os.path.exists('./cifar_net.pth'), 'The trained network for CIFAR was
not stored properly. ' + \
                                              'Please read and follow the instru
ctions/important notes.'

    assert os.path.exists('./mnist_net.pth'), 'The trained network for MNIST was
not stored properly. ' + \
                                              'Please read and follow the instru
ctions/important notes.'

    copy2('./cifar_net.pth', './cifar_net_submitted.pth')
    copy2('./mnist_net.pth', './mnist_net_submitted.pth')
```

*Assignment Summary

Go through the CIFAR-10 tutorial at https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html), and ensure you can run the code. Modify the architecture that is offered in the CIFAR-10 tutorial to get the best accuracy you can. Anything better than about 93.5% will be comparable with current research.

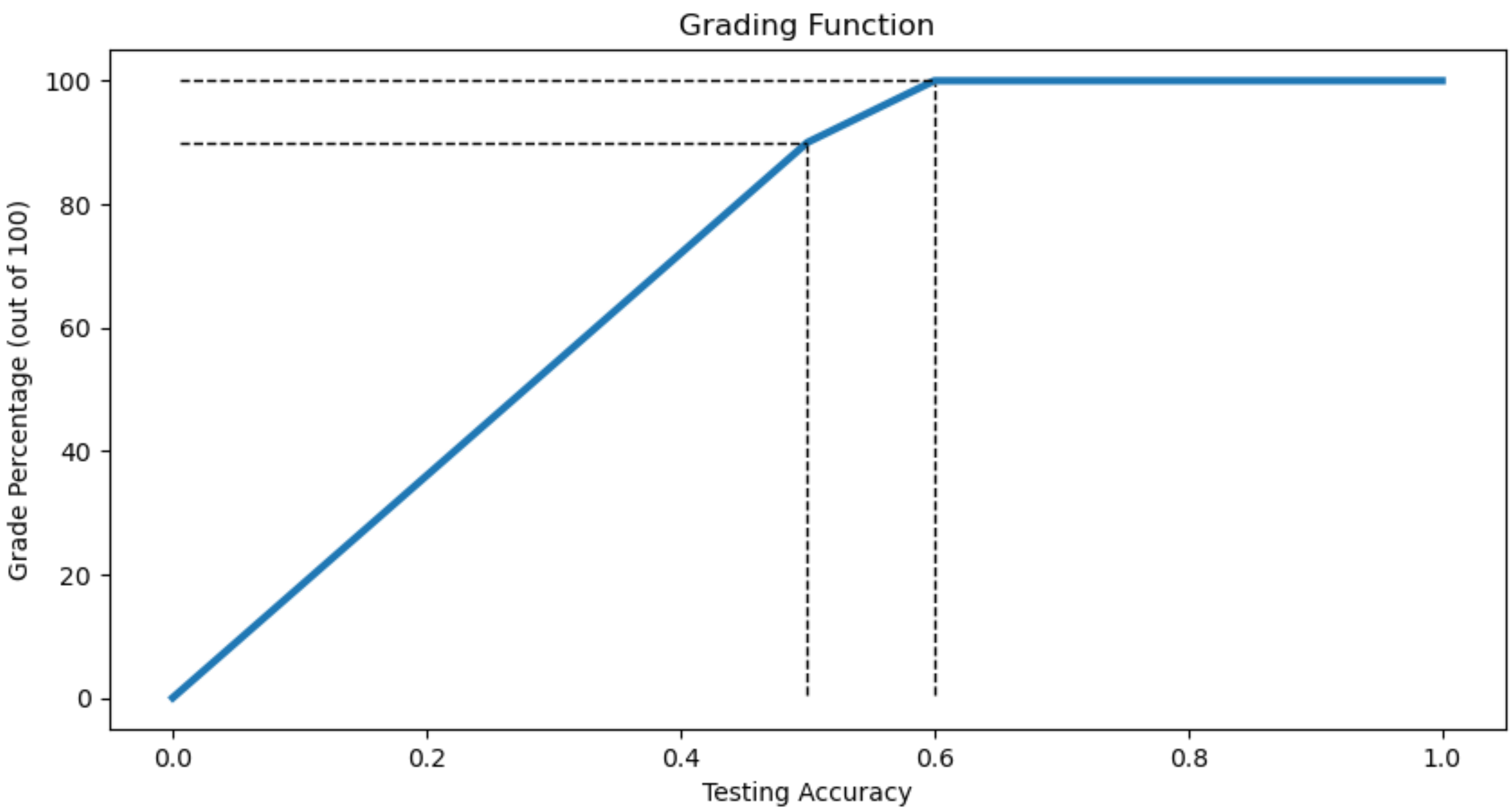
Redo the same efforts for the MNIST digit data set.

Procedural Instructions:

This assignment is less guided than the previous assignments. You are supposed to train a deep convolutional classifier, and store it in a file. The autograder will load the trained model, and test its accuracy on a hidden test data set. Your classifier's test accuracy will determine your grade for each part according to the following model.

In [3]:

```
fig, ax = plt.subplots(1, 1, figsize=(10, 5), dpi=100)
ax.plot([0., 0.5, 0.6, 1.], [0., 90., 100., 100.], lw=3)
ax.axhline(y=90, xmin=0.05, xmax=.5, lw=1, ls='--', c='black')
ax.axvline(x=0.5, ymin=0.05, ymax=.86, lw=1, ls='--', c='black')
ax.axhline(y=100, xmin=0.05, xmax=.59, lw=1, ls='--', c='black')
ax.axvline(x=0.6, ymin=0.05, ymax=.95, lw=1, ls='--', c='black')
ax.set_xlabel('Testing Accuracy')
ax.set_ylabel('Grade Percentage (out of 100)')
ax.set_title('Grading Function')
None
```



Important Notes

You **should** read these notes before starting as these notes include crucial information about what is expected from you.

1. **Use Pytorch**: The autograder will only accept pytorch models.

   - Pytorch's CIFAR-10 tutorial at [https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html)](https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html) is the best starting point for this assignment. However, we will not prohibit using or learning from any other tutorial you may find online.

1. **No Downloads**: The coursera machines are disconnected from the internet. We already have downloaded the pytorch data files, and uploaded them for you. You will need to disable downloading the files if you're using data collector APIs such as `torchvision.datasets`.

   - For the CIFAR data, you should provide the `root='/home/jovyan/work/course-lib/data_cifar', download=False` arguments to the `torchvision.datasets.CIFAR10` API.
   - For the MNIST data, you should provide the `root='/home/jovyan/work/course-lib/data_mnist', download=False` arguments to the `torchvision.datasets.MNIST` API.

1. **Store the Trained Model**: The autograder can not and will not retrain your model. You are supposed to train your model, and then store your best model with the following names:

   - The CIFAR classification model must be stored at `./cifar_net.pth`.
   - The MNIST classification model must be stored at `./mnist_net.pth`.
   - Do not place these file under any newly created directory.
   - The trained model may **not exceed 1 MB** in size.

1. **Model Class Naming**: The neural models in the pytorch library are subclasses of the `torch.nn.Module` class. While you can define any architecture as you please, your `torch.nn.Module` must be named `Net` exactly. In other words, you are supposed to have the following lines somewhere in your network definition:

   ```
   import torch.nn as nn
   class Net(nn.Module):
       ...
   ```

1. **Grading Reference Pre-processing**: We will use a specific randomized transformation for grading that can be found in the `Autograding and Final Tests` section. Before training any model for long periods of time, you need to pay attention to the existence of such a testing pre-processing.
2. **Training Rules**: You are able to make the following decisions about your model:

   - You **can** choose and change your architecture as you please.

  - You can have shallow networks, or deep ones.
  - You can customize the number of neural units in each layer and the depth of the network.
  - You are free to use convolutional, and non-convolutional layers.
  - You can employ batch normalization if you would like to.
  - You can use any type of non-linear layers as you please. `Tanh` , `Sigmoid` , and `ReLU` are some common activation functions.
  - You can use any kind of pooling layers you deem appropriate.
  - etc.
- You **can** initialize your network using any of the methods described in `https://pytorch.org/docs/stable/nn.init.html` .
  - Some common layer initializations include the Xavier (a.k.a. Glorot), and orthogonal initializations.
  - You may want to avoid initializing your network with all zeros (think about the symmetry of the neural units, and how identical initialization may be a bad idea considering what happens during training).
- You **can** use and customize any kind of optimization methods you deem appropriate.
  - You can use any first order stochastic methods (i.e., Stochastic Gradient Descent variants) such as Vanilla SGD, Adam, RMSProp, Adagrad, etc.
  - You are also welcome to use second order optimization methods such as newton and quasi-newton methods. However, it may be expensive and difficult to make them work for this setting.
  - Zeroth order methods (i.e., Black Box methods) are also okay (although you may not find them very effective in this setting).
  - You can specify any learning rates first order stochastic methods. In fact, you can even customize your learning rate schedules.
  - You are free to use any mini-batch sizes for stochastic gradient computation.
  - etc.
- You **can** use any kind of loss function you deem effective.
  - You can add any kind of regularization to your loss.
  - You can pick any kind of classification loss functions such as the cross-entropy and the mean squared loss.
- You **cannot** warm-start your network (i.e., you **cannot** use a pre-trained network).
- You **may** use any kind of image pre-processing and transformations during training. However, for the same transformations to persist at grading time, you may need to apply such transformations within the neural network's `forward` function definition.
  - In other words, we will drop any `DataLoader` or transformations that your network may rely on to have good performance, and we will only load and use your neural network for grading.

1. Object Classification Using the CIFAR Data

1.1 Loading the Data

In [232]:

```
message = 'You can implement the pre-processing transformations, data sets, data loaders, etc. in this cell. \n'
message = message + '**Important Note**: Read the "Grading Reference Pre-processing" bullet above, and look at the'
message = message + ' test pre-processing transformations in the "Autograding and Final Tests" section before'
message = message + ' training models for long periods of time.'
print(message)

transformation_list = [transforms.RandomAffine(degrees=30, translate=(0.01, 0.01), scale=(0.9, 1.1),
                                               shear=None, resample=0, fillcolor=0),
                       transforms.ToTensor(),
                       transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]

transform = transforms.Compose(transformation_list)

BATCH_SIZE = 4
dataset = torchvision.datasets.CIFAR10(root='/home/jovyan/work/course-lib/data_cifar', download=False, transform=transform)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

# your code here
# raise NotImplementedError
```

```
You can implement the pre-processing transformations, data sets, data loaders, etc. in this cell.
**Important Note**: Read the "Grading Reference Pre-processing" bullet above, and look at the test pre-processing transformations in the "Autograding and Final Tests" section before training models for long periods of time.
```

In [233]:

```
import matplotlib.pyplot as plt
# message = 'You can visualize some of the pre-processed images here (This is optional and only for your own reference).'
# print(message)

# your code here
image, label = next(iter(train_loader))
# print(len(train_loader))
print(label)
# plt.imshow(image.squeeze(0).permute(1,2,0))
# plt.show()
# raise NotImplementedError
```

```
tensor([4, 1, 3, 2])
```

1.2 Defining the Model

Important Note: As mentioned above, make sure you name the neural module class as `Net`. In other words, you are supposed to have the following lines somewhere in your network definition:

```
import torch.nn as nn
class Net(nn.Module):
    ...
```

In [234]:

```
# message = 'You can define the neural architecture and instantiate it in this c
ell.'
# print(message)

# # your code here
# class Net(nn.Module):
#     def __init__(self):
#         super(Net, self).__init__()
#         self.conv1 = ConvLayer(3, 32, 5, 1)
#         self.conv2 = ConvLayer(32, 64, 5, 1)
#         self.conv3 = ConvLayer(64, 128, 3, 1)
#         self.fc1 = nn.Linear(128 * 1 * 1, 50)
#         self.fc2 = nn.Linear(50, 10)

#     def forward(self, x):
#         x = F.max_pool2d(self.conv1(x), 2, 2)
#         x = F.max_pool2d(self.conv2(x), 2, 2)
#         x = F.max_pool2d(self.conv3(x), 2, 2)
#         x = x.view(-1, 128 * 1 * 1)
#         x = F.relu(self.fc1(x))
#         return self.fc2(x)

# class ConvLayer(nn.Module):
#     def __init__(self, in_ch, out_ch, ksize, stride):
#         super(ConvLayer, self).__init__()
#         self.conv = nn.Conv2d(in_ch, out_ch, ksize, stride)
# #         self.pad = nn.ReflectionPad2d(ksize // 2)

#     def forward(self, x):
#         return F.relu(self.conv(x))
# # raise NotImplementedError
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 60, 5)
        self.fc1 = nn.Linear(60 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 60 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

1.3 Initializing the Neural Model

It may be a better idea to fully control the initialization process of the neural weights rather than leaving it to the default procedure chosen by pytorch.

Here is pytorch's documentation about different initialization methods:
[https://pytorch.org/docs/stable/nn.init.html (https://pytorch.org/docs/stable/nn.init.html)](https://pytorch.org/docs/stable/nn.init.html)

Some common layer initializations include the Xavier (a.k.a. Glorot), and orthogonal initializations.

In [235]:

```
message = 'You can initialize the neural weights here, and not leave it to the l
ibrary default (this is optional).'
print(message)

net = Net()

# Reference: https://stackoverflow.com/questions/49433936/how-to-initialize-weig
hts-in-pytorch
def init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform(m.weight)
        m.bias.data.fill_(0.01)

net.apply(init_weights)
# your code here
# raise NotImplementedError
```

```
You can initialize the neural weights here, and not leave it to the
library default (this is optional).
```

Out[235]:

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(32, 60, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=1500, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

In [236]:

```
def save_model(path):
    net.eval()
    torch.save(net.state_dict(), path)

save_model('lala.pth')
```

In [237]:

```
# !ls -sh
```

```
total 5.0M
948K cifar_net.pth  2.3M CNN.ipynb  948K lala.pth  832K mnist_net.pt
h
```

In [238]:

```
# !rm lala.pth
```

1.4 Defining The Loss Function and The Optimizer

In [240]:

```
message = 'You can define the loss function and the optimizer of interest here.'
print(message)
optimizer = optim.SGD(net.parameters(), lr=1e-3, momentum=0.9)
optimizer.zero_grad()

criterion = nn.CrossEntropyLoss()
# your code here
# raise NotImplementedError
```

```
You can define the loss function and the optimizer of interest here.
```

1.5 Training the Model

Important Note: In order for the autograder not to time-out due to training during grading, please make sure you wrap your training code within the following conditional statement:

```
if perform_computation:
   # Place any computationally intensive training/optimization code here
```

In [242]:

```
epoch = 10
```

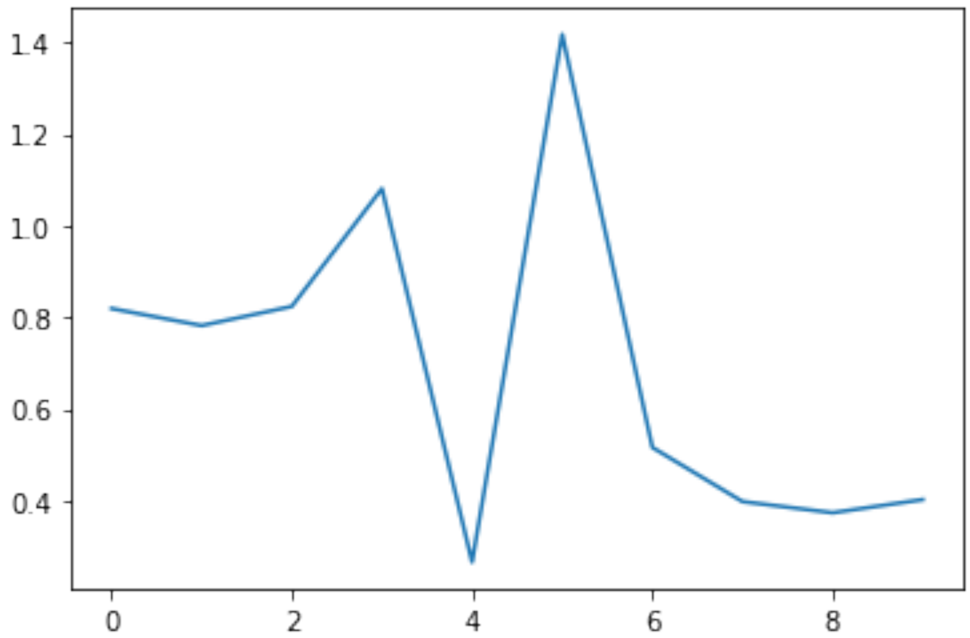
In [243]:

```
import matplotlib.pyplot as plt
```

In [244]:

```
if perform_computation:
    message = 'You can define the training loop and forward-backward propagation here.'
    print(message)
    loss_history = []
    for i in range(epoch):
        for batch_i, (image, label) in enumerate(train_loader):
            optimizer.zero_grad()

            out = net(image)
            loss = criterion(out, label)

            loss.backward()
            optimizer.step()

        loss_history.append(loss)
        save_model(f'model_batch{i + 1}.pth')


    plt.plot(loss_history)
    # your code here
#     raise NotImplementedError
```

```
You can define the training loop and forward-backward propagation he
re.
```



1.6 Storing the Model

In [245]:

```
# !ls
```

```
cifar_net.pth       model_batch1.pth  model_batch5.pth  model_batch9.
pth
CNN.ipynb           model_batch2.pth  model_batch6.pth
mnist_net.pth       model_batch3.pth  model_batch7.pth
model_batch10.pth   model_batch4.pth  model_batch8.pth
```

In [246]:

```
# !mv model_batch10.pth cifar_net.pth
```

In [97]:

```
# !rm model*
```

In [102]:

```
message = 'Here you should store the model at "./cifar_net.pth" .'
print(message)

# your code here
# save_model("./cifar_net.pth")
# raise NotImplementedError
```

```
Here you should store the model at "./cifar_net.pth" .
```

1.7 Evaluating the Trained Model

In [99]:

```
message = 'Here you can visualize a bunch of examples and print the prediction o
f the trained classifier (this is optional).'
print(message)

# your code here
# raise NotImplementedError
```

```
Here you can visualize a bunch of examples and print the prediction
of the trained classifier (this is optional).
```

In [100]:

```
message = 'Here you can evaluate the overall accuracy of the trained classifier
(this is optional).'
print(message)

# your code here
# raise NotImplementedError
```

```
Here you can evaluate the overall accuracy of the trained classifier
(this is optional).
```

In [101]:

```
message = 'Here you can evaluate the per-class accuracy of the trained classifie
r (this is optional).'
print(message)

# your code here
# raise NotImplementedError
```

```
Here you can evaluate the per-class accuracy of the trained classifi
er (this is optional).
```

1.8 Autograding and Final Tests

In [46]:

```
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

In [247]:

```
assert 'Net' in globals().keys(), 'The Net class was not defined earlier. ' + \
                                  'Make sure you read and follow the instruction
s provided as Important Notes' + \
                                  '(especially, the "Model Class Naming" part).'

cifar_net_path = './cifar_net_submitted.pth' if in_submission else './cifar_net.
pth'

assert os.path.exists(cifar_net_path), 'You have not stored the trained model pr
operly. '+  \
                                       'Make sure you read and follow the instru
ctions provided as Important Notes.'

assert os.path.getsize(cifar_net_path) < 1000000, 'The size of your trained mode
l exceeds 1 MB.'



if 'net' in globals():
```

```
    del net

net = Net()
net.load_state_dict(torch.load(cifar_net_path))
net = net.eval()

# Disclaimer: Most of the following code was adopted from Pytorch's Documentatio
n and Examples
# https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

transformation_list = [transforms.RandomAffine(degrees=30, translate=(0.01, 0.01
), scale=(0.9, 1.1),
                                               shear=None, resample=0, fillcolor
=0),
                       transforms.ToTensor(),
                       transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]

test_pre_tranformation = transforms.Compose(transformation_list)

cifar_root = '/home/jovyan/work/course-lib/data_cifar'
testset = torchvision.datasets.CIFAR10(root=cifar_root, train=False,
                                       download=False, transform=test_pre_tranfo
rmation)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=1)


class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
print('----------')
print(f'Overall Testing Accuracy: {100. * sum(class_correct) / sum(class_total)}
%%')
```

```
Accuracy of  zero : 64 %
Accuracy of   one : 86 %
Accuracy of   two : 52 %
Accuracy of three : 52 %
Accuracy of  four : 61 %
Accuracy of  five : 58 %
Accuracy of   six : 78 %
Accuracy of seven : 66 %
Accuracy of eight : 82 %
Accuracy of  nine : 65 %
----------
Overall Testing Accuracy: 66.83 %%
```

In [ ]:

```
# "Object Classification Test: Checking the accuracy on the CIFAR Images"
```

2. Digit Recognition Using the MNIST Data

2.1 Loading the Data

In [216]:

```
message = 'You can implement the pre-processing transformations, data sets, data loaders, etc. in this cell. \n'
message = message + '**Important Note**: Read the "Grading Reference Pre-processing" bullet, and look at the'
message = message + ' test pre-processing transformations in the "Autograding and Final Tests" section before'
message = message + ' training models for long periods of time.'
print(message)

# root='/home/jovyan/work/course-lib/data_mnist', download=False

transformation_list = [transforms.RandomAffine(degrees=60, translate=(0.2, 0.2), scale=(0.5, 2.),
                                               shear=None, resample=0, fillcolor=0),
                       transforms.ToTensor(),
                       transforms.Normalize((0.5,), (0.5,))]

transform = transforms.Compose(transformation_list)

BATCH_SIZE = 64
dataset = torchvision.datasets.MNIST(root='/home/jovyan/work/course-lib/data_mnist', download=False, transform=transform)
train_loader = torch.utils.data.DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)

# your code here
# raise NotImplementedError
```

```
You can implement the pre-processing transformations, data sets, data loaders, etc. in this cell.
**Important Note**: Read the "Grading Reference Pre-processing" bullet, and look at the test pre-processing transformations in the "Autograding and Final Tests" section before training models for long periods of time.
```

In [217]:

```
message = 'You can visualize some of the pre-processed images here (This is opti
onal and only for your own reference).'
print(message)

# your code here
image, label = next(iter(train_loader))
print(len(train_loader))
# print(label)
# plt.imshow(image[0].squeeze(0))
# print(image.size())
# plt.show()
# raise NotImplementedError
```

```
You can visualize some of the pre-processed images here (This is opt
ional and only for your own reference).
938
```

2.2 Defining the Model

Important Note: As mentioned above, make sure you name the neural module class as `Net`. In other words, you are supposed to have the following lines somewhere in your network definition:

```
import torch.nn as nn
class Net(nn.Module):
    ...
```

In [218]:

```
message = 'You can define the neural architecture and instantiate it in this cel
l.'
print(message)

# your code here
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 32, 5)
        self.fc1 = nn.Linear(4 * 4 * 32, 300)
        self.fc2 = nn.Linear(300, 100)
        self.fc3 = nn.Linear(100, 10)
    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 32 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# raise NotImplementedError
```

```
You can define the neural architecture and instantiate it in this ce
ll.
```

2.3 Initializing the Neural Model

It may be a better idea to fully control the initialization process of the neural weights rather than leaving it to the default procedure chosen by pytorch.

Here is pytorch's documentation about different initialization methods: [https://pytorch.org/docs/stable/nn.init.html (https://pytorch.org/docs/stable/nn.init.html)](https://pytorch.org/docs/stable/nn.init.html)

Some common layer initializations include the Xavier (a.k.a. Glorot), and orthogonal initializations.

In [219]:

```
message = 'You can initialize the neural weights here, and not leave it to the l
ibrary default (this is optional).'
print(message)

# your code here
net = Net()

# Reference: https://stackoverflow.com/questions/49433936/how-to-initialize-weig
hts-in-pytorch
def init_weights(m):
    if type(m) == nn.Linear:
        torch.nn.init.xavier_uniform(m.weight)
        m.bias.data.fill_(0.01)

net.apply(init_weights)
# raise NotImplementedError
```

```
You can initialize the neural weights here, and not leave it to the
library default (this is optional).
```

Out[219]:

```
Net(
  (conv1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (conv2): Conv2d(32, 32, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=512, out_features=300, bias=True)
  (fc2): Linear(in_features=300, out_features=100, bias=True)
  (fc3): Linear(in_features=100, out_features=10, bias=True)
)
```

In [220]:

```
def save_model(path):
    net.eval()
    torch.save(net.state_dict(), path)

save_model('lala.pth')
```

In [231]:

```
!ls -sh
```

```
total 4.0M
948K cifar_net.pth  2.3M CNN.ipynb  832K mnist_net.pth
```

In [222]:

```
!rm lala.pth
```

In [230]:

```
!rm model*
```

2.4 Defining The Loss Function and The Optimizer

In [224]:

```
message = 'You can define the loss function and the optimizer of interest here.'
print(message)

# your code here
optimizer = optim.SGD(net.parameters(), lr=1e-2, momentum=0.9)
optimizer.zero_grad()

criterion = nn.CrossEntropyLoss()
# raise NotImplementedError
```

```
You can define the loss function and the optimizer of interest here.
```

2.5 Training the Model

Important Note: In order for the autograder not to time-out due to training during grading, please make sure you wrap your training code within the following conditional statement:

```
if perform_computation:
   # Place any computationally intensive training/optimization code here
```

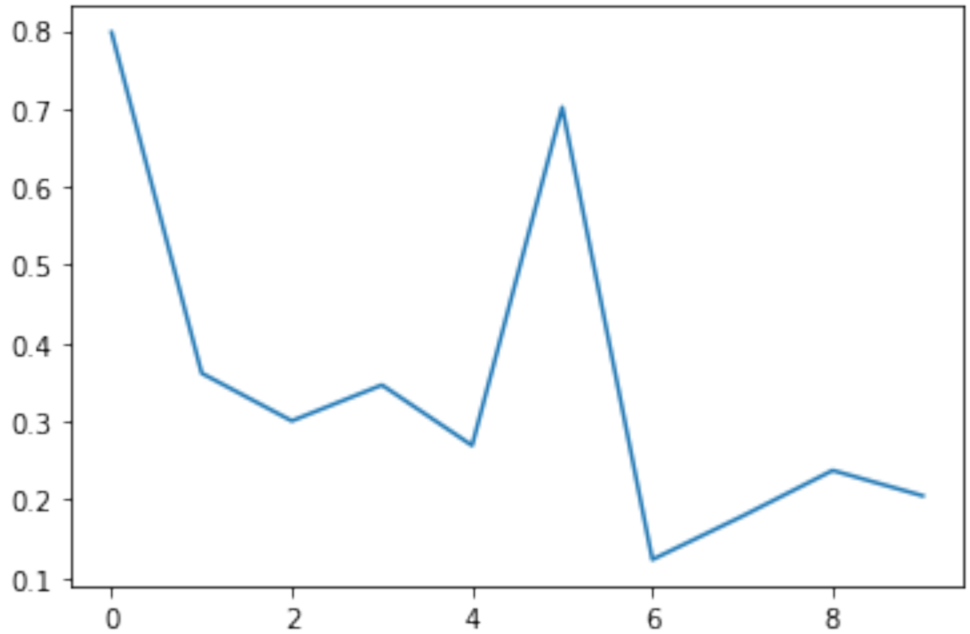
In [225]:

```
epoch = 10
```

In [226]:

```
if perform_computation:
    message = 'You can define the training loop and forward-backward propagation
here.'
    print(message)
    loss_history = []
    for i in range(epoch):
        for batch_i, (image, label) in enumerate(train_loader):
            optimizer.zero_grad()

            out = net(image)
            loss = criterion(out, label)

            loss.backward()
            optimizer.step()

        loss_history.append(loss)
        save_model(f'model_epoch{i + 1}.pth')

    plt.plot(loss_history)
    # your code here
#     raise NotImplementedError
```

```
You can define the training loop and forward-backward propagation he
re.
```



2.6 Storing the Model

In [227]:

```
!ls
```

```
cifar_net.pth       model_epoch1.pth  model_epoch5.pth  model_epoch9.
pth
CNN.ipynb           model_epoch2.pth  model_epoch6.pth
mnist_net.pth       model_epoch3.pth  model_epoch7.pth
model_epoch10.pth   model_epoch4.pth  model_epoch8.pth
```

In [228]:

```
!mv model_epoch10.pth mnist_net.pth
```

In [ ]:

```
message = 'Here you should store the model at "./mnist_net.pth" .'
print(message)

# your code here
raise NotImplementedError
```

2.7 Evaluating the Trained Model

In [ ]:

```
message = 'Here you can visualize a bunch of examples and print the prediction o
f the trained classifier (this is optional).'
print(message)

# your code here
# raise NotImplementedError
```

In [ ]:

```
message = 'Here you can evaluate the overall accuracy of the trained classifier
(this is optional).'
print(message)

# your code here
# raise NotImplementedError
```

In [ ]:

```
message = 'Here you can evaluate the per-class accuracy of the trained classifie
r (this is optional).'
print(message)

# your code here
# raise NotImplementedError
```

2.8 Autograding and Final Tests

In [154]:

```
classes = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight
', 'nine']
```

In [229]:

```
assert 'Net' in globals().keys(), 'The Net class was not defined earlier. ' + \
                                  'Make sure you read and follow the instruction
s provided as Important Notes' + \
                                  '(especially, the "Model Class Naming" part).'

mnist_net_path = './mnist_net_submitted.pth' if in_submission else './mnist_net.
pth'

assert os.path.exists(mnist_net_path), 'You have not stored the trained model pr
operly. '+  \
                                       'Make sure you read and follow the instru
ctions provided as Important Notes.'

assert os.path.getsize(mnist_net_path) < 1000000, 'The size of your trained mode
l exceeds 1 MB.'

if 'net' in globals():
    del net
net = Net()
net.load_state_dict(torch.load(mnist_net_path))
net = net.eval()

# Disclaimer: Most of the following code was adopted from Pytorch's Documentatio
n and Examples
# https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

transformation_list = [transforms.RandomAffine(degrees=60, translate=(0.2, 0.2),
scale=(0.5, 2.),
                                               shear=None, resample=0, fillcolor
=0),
                       transforms.ToTensor(),
                       transforms.Normalize((0.5,), (0.5,))]

test_pre_tranformation = transforms.Compose(transformation_list)

mnist_root = '/home/jovyan/work/course-lib/data_mnist'
testset = torchvision.datasets.MNIST(root=mnist_root, train=False,
                                     download=False, transform=test_pre_tranform
ation)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=1)
```

```python
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
print('----------')
print(f'Overall Testing Accuracy: {100. * sum(class_correct) / sum(class_total)}
%%')
```

```
Accuracy of  zero : 94 %
Accuracy of   one : 98 %
Accuracy of   two : 89 %
Accuracy of three : 91 %
Accuracy of  four : 94 %
Accuracy of  five : 89 %
Accuracy of   six : 97 %
Accuracy of seven : 86 %
Accuracy of eight : 83 %
Accuracy of  nine : 90 %
----------
Overall Testing Accuracy: 91.67 %%
```

In [ ]:

```python
# "Digit Recognition Test: Checking the accuracy on the MNIST Images"
```

In [ ]:

```python

```