# Ass1-Exercise3

November 16, 2022

[1]:
```
import numpy as np
import math
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
from torchvision import datasets
import matplotlib.pyplot as plt
```

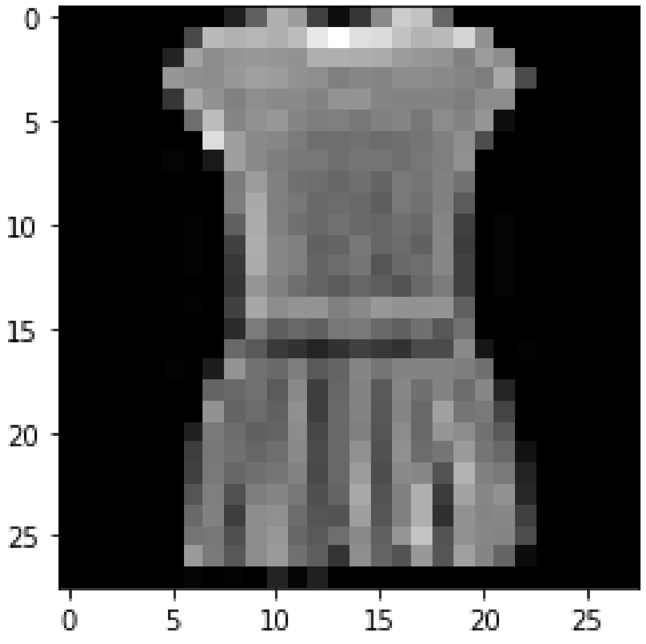
[2]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# HypeParameters
input=28*28
input_size = input
num_classes = 10
num_epochs = 10
hidden_size = 500
batch_size = 128
learning_rate = 0.0001
```

[3]:
```
#Data-loading
train_dataset = torchvision.datasets.FashionMNIST(root='./data',train=True,
transform=transforms.ToTensor(),download=True)
test_dataset = torchvision.datasets.FashionMNIST(root='./data',train=False,
transform=transforms.ToTensor(),download=True)
train_loader = torch.utils.data.DataLoader(dataset=(train_dataset),
batch_size=batch_size,shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=(test_dataset) ,
batch_size=batch_size,shuffle=False)
###########################################
def create_datasets(batch_size):
# percentage of training set to use as validation
  valid_size = 0.15
# convert data to torch.FloatTensor
  transform = transforms.ToTensor()
# choose the training and test datasets
  train_data = datasets.FashionMNIST(root='data',train=True,download=True,
                              transform=transform)
  test_data = datasets.FashionMNIST(root='data',train=False,download=True,
                            transform=transform)
  # obtain training indices that will be used for validation
  num_train = len(train_data)
  indices = list(range(num_train))
  np.random.shuffle(indices)
  split = int(np.floor(valid_size * num_train))
  train_idx, valid_idx = indices[split:], indices[:split]
  # define samplers for obtaining training and validation batches
  train_sampler = SubsetRandomSampler(train_idx)
  valid_sampler = SubsetRandomSampler(valid_idx)
  # load training data in batches
  train_loader = torch.utils.data.DataLoader(train_data,batch_size=batch_size,
  sampler=train_sampler,num_workers=0)
  # load validation data in batches
  valid_loader = torch.utils.data.DataLoader(train_data,batch_size=batch_size,
  sampler=valid_sampler,num_workers=0)
  # load test data in batches
  test_loader = torch.utils.data.DataLoader(test_data,batch_size=batch_size,
  num_workers=0)
  return train_loader, test_loader, valid_loader
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/26421880 [00:00<?, ?it/s]

Extracting ./data/FashionMNIST/raw/train-images-idx3-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/29515 [00:00<?, ?it/s]

Extracting ./data/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/4422102 [00:00<?, ?it/s]

Extracting ./data/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
```

```
./data/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

  0%|          | 0/5148 [00:00<?, ?it/s]

Extracting ./data/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
./data/FashionMNIST/raw
```

[4]:
```
# example = iter(train_loader)
# example
# example_data, example_targ = example.next()
# for i in range(6):
#   plt.subplot(2,3,i+1)
#   plt.imshow(example_data[i][0])
#   plt.show()
#pick a sample to plot

sample = 3
image = train_dataset[3][0][0]
# plot the sample
fig = plt.figure
plt.imshow(image, cmap='gray')
plt.show()
```



[4]:

```

```

[5]:

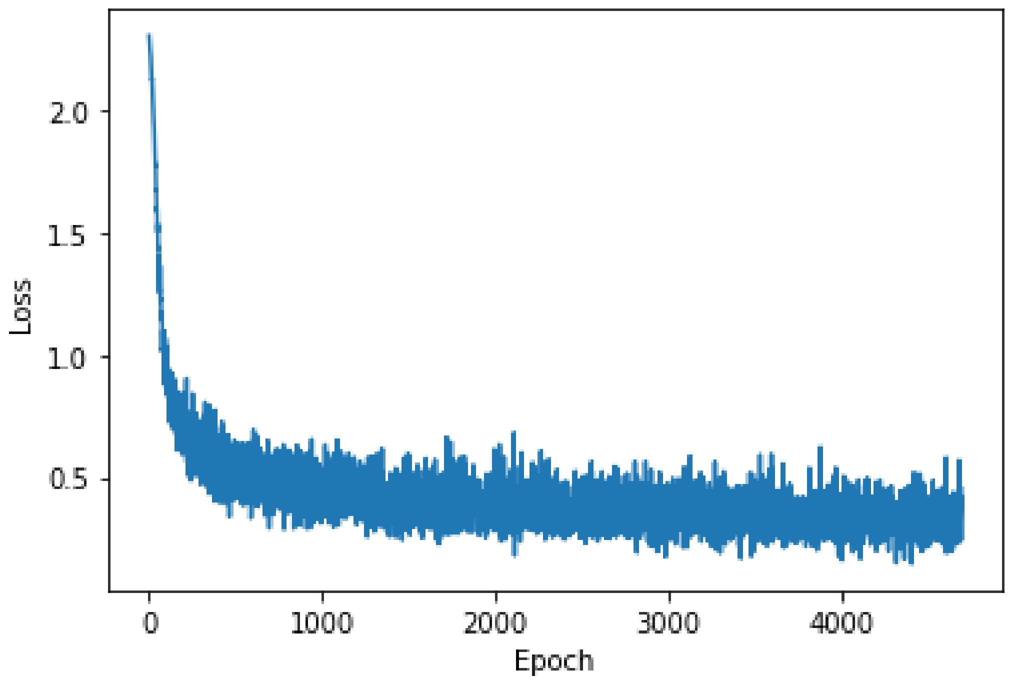
```
class NeuralNet(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(NeuralNet, self).__init__()
    self.input_size = input_size
    self.l1 = nn.Linear(input_size, hidden_size)
    self.relu = nn.ReLU()
    self.l2 = nn.Linear(hidden_size, math.floor(hidden_size/2))
    self.l3 = nn.Linear(math.floor(hidden_size/2), num_classes)
    self.N=nn.Softmax()
  def forward(self, x):
    out = self.l1(x)
    out = self.relu(out)
    out = self.l2(out)
    out = self.relu(out)
    out = self.l3(out)
    # no activation and no softmax at the end
    return out
model = NeuralNet(input_size, hidden_size, num_classes).to(device)
print(model)
```

```
NeuralNet(
  (l1): Linear(in_features=784, out_features=500, bias=True)
  (relu): ReLU()
  (l2): Linear(in_features=500, out_features=250, bias=True)
  (l3): Linear(in_features=250, out_features=10, bias=True)
  (N): Softmax(dim=None)
)
```

[6]:

```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,␣
 ↪weight_decay=0.00001)
Loss=[]
n_total_steps = len(train_loader)
for epoch in range(num_epochs):
  for i, (images, labels) in enumerate(train_loader):
    # origin shape: [100, 1, 28, 28]
    # resized: [100, 784]
    images = images.reshape(-1, 28*28).to(device)
    labels = labels.to(device)
    # Forward pass
    outputs = model(images)
    loss = criterion(outputs, labels)
    # Backward and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    Loss.append(loss.item())
    if (i+1) % 100 == 0:
      print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/{n_total_steps}],␣
 ↪Loss: {loss.item()}')

plt.plot(Loss)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

```
Epoch [1/10], Step [200/469], Loss: 0.6810627579689026
Epoch [1/10], Step [400/469], Loss: 0.5205481648445129
Epoch [2/10], Step [200/469], Loss: 0.5623003244400024
Epoch [2/10], Step [400/469], Loss: 0.38035720586776733
Epoch [3/10], Step [200/469], Loss: 0.4969681203365326
Epoch [3/10], Step [400/469], Loss: 0.4654163122177124
Epoch [4/10], Step [200/469], Loss: 0.3807421624660492
Epoch [4/10], Step [400/469], Loss: 0.45237284898757935
Epoch [5/10], Step [200/469], Loss: 0.46666577458381653
Epoch [5/10], Step [400/469], Loss: 0.41578760743141174
Epoch [6/10], Step [200/469], Loss: 0.42519769072532654
Epoch [6/10], Step [400/469], Loss: 0.4667348861694336
Epoch [7/10], Step [200/469], Loss: 0.32523852586746216
Epoch [7/10], Step [400/469], Loss: 0.4016990065574646
Epoch [8/10], Step [200/469], Loss: 0.3768923282623291
Epoch [8/10], Step [400/469], Loss: 0.2854776084423065
Epoch [9/10], Step [200/469], Loss: 0.3018138110637665
Epoch [9/10], Step [400/469], Loss: 0.39742064476013184
Epoch [10/10], Step [200/469], Loss: 0.4302467703819275
Epoch [10/10], Step [400/469], Loss: 0.30491912364959717
```



[7]:
```
# Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)
with torch.no_grad():
  n_correct = 0
  n_samples = 0
  for images, labels in test_loader:
    images = images.reshape(-1, 28*28).to(device)
    labels = labels.to(device)
    outputs = model(images)
    # max returns (value ,index)
    _, predicted = torch.max(outputs.data, 1)
    n_samples += labels.size(0)
    n_correct += (predicted == labels).sum().item()
  acc = 100.0 * n_correct / n_samples
  print(f'Accuracy of the network on the 10000 test i```mages: {acc} %')
```

```
Accuracy of the network on the 10000 test i```mages: 87.16 %
```

[8]:
```
del model
class NeuralNet(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(NeuralNet, self).__init__()
    self.input_size = input_size
    self.l1 = nn.Linear(input_size, hidden_size)
    self.relu = nn.ReLU()
```

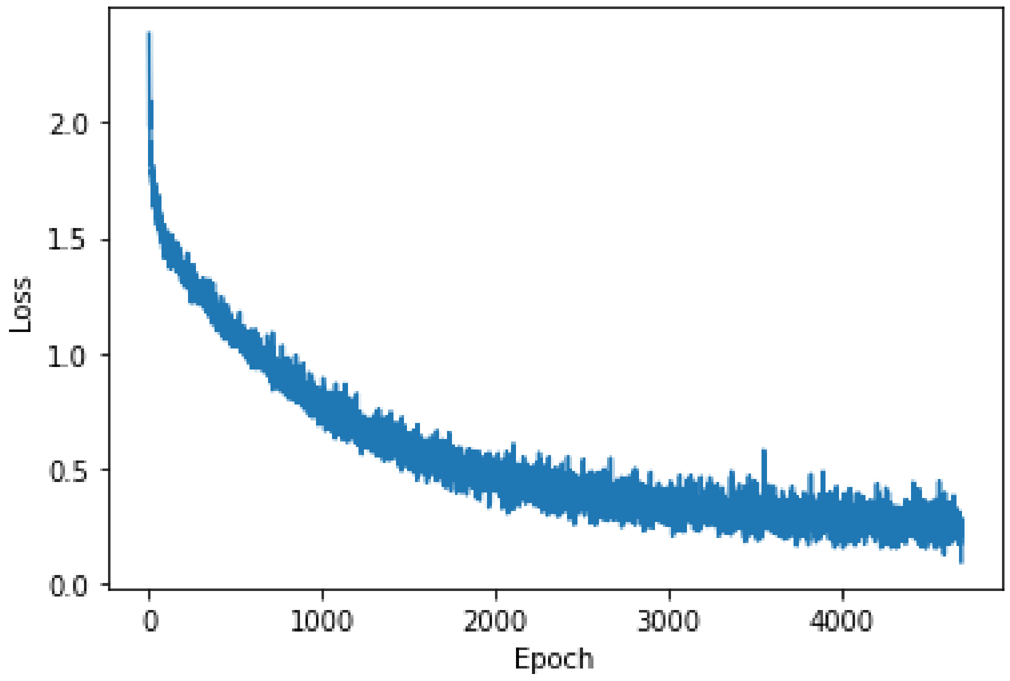
```
        self.l2 = nn.Linear(hidden_size, math.floor(hidden_size/2))
        self.BN= nn.BatchNorm1d(math.floor(hidden_size/2))
        self.D0=nn.Dropout(0.05)
        self.D1=nn.Dropout(0.15)
        self.l3 = nn.Linear(math.floor(hidden_size/2), num_classes*2)
        self.BN2= nn.BatchNorm1d(num_classes*2)
        self.l4 = nn.Linear(num_classes*2, num_classes)
        self.N=nn.Softmax()
    def forward(self, x):
        out = self.l1(x)
        out = self.relu(out)
        out= self.D0(out)
        out = self.l2(out)
        out= self.BN(out)
        out = self.relu(out)
        out= self.D1(out)
        out = self.l3(out)
        out = self.relu(out)
        out= self.BN2(out)
        out = self.relu(out)
        out = self.l4(out)
        # no activation and no softmax at the end
        return out
model = NeuralNet(input_size, hidden_size, num_classes).to(device)
print(model)
```

```
NeuralNet(
  (l1): Linear(in_features=784, out_features=500, bias=True)
  (relu): ReLU()
  (l2): Linear(in_features=500, out_features=250, bias=True)
  (BN): BatchNorm1d(250, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (D0): Dropout(p=0.05, inplace=False)
  (D1): Dropout(p=0.15, inplace=False)
  (l3): Linear(in_features=250, out_features=20, bias=True)
  (BN2): BatchNorm1d(20, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (l4): Linear(in_features=20, out_features=10, bias=True)
  (N): Softmax(dim=None)
)
```

[9]:
```
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,␣
 ↪weight_decay=0.00001)
Loss2=[]
n_total_steps = len(train_loader)
```

```
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):

        # origin shape: [100, 1, 28, 28]
        # resized: [100, 784]
        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        outputsval = model(images)
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        Loss2.append(loss.item())
        if (i+1) % 100 == 0:

            print (f'Epoch [{epoch+1}/{num_epochs}], Step [{i+1}/
 ↪{n_total_steps}], Loss: {loss.item()}')
plt.plot(Loss2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```

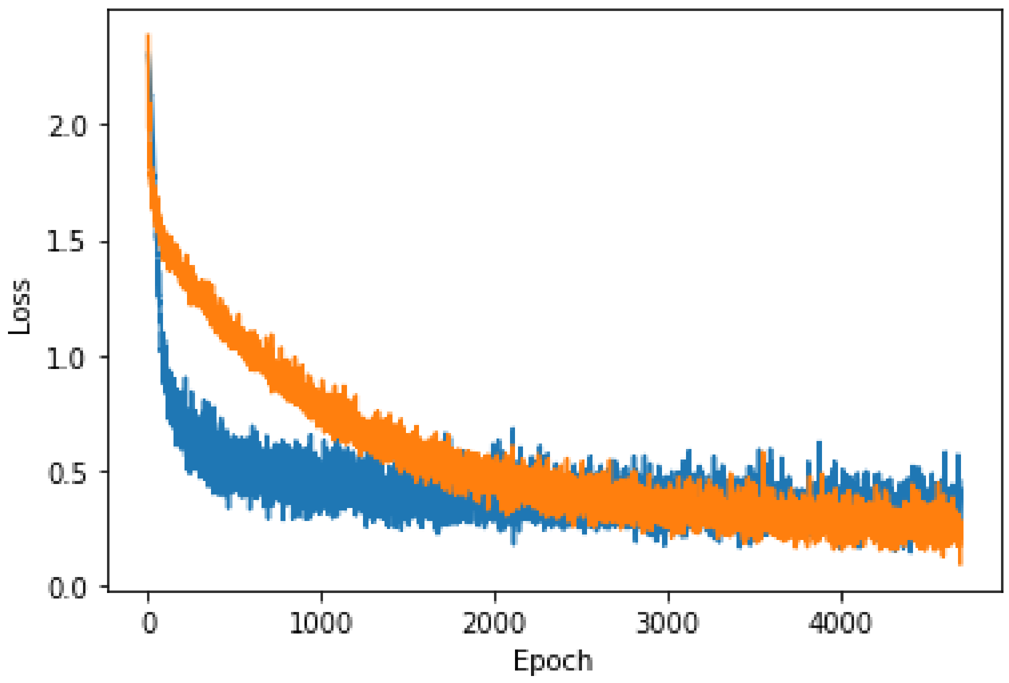
```
Epoch [1/10], Step [200/469], Loss: 1.3604509830474854
Epoch [1/10], Step [400/469], Loss: 1.184468150138855
Epoch [2/10], Step [200/469], Loss: 0.9321385622024536
Epoch [2/10], Step [400/469], Loss: 0.7840135097503662
Epoch [3/10], Step [200/469], Loss: 0.7352109551429749
Epoch [3/10], Step [400/469], Loss: 0.6467800140380859
Epoch [4/10], Step [200/469], Loss: 0.5735831260681152
Epoch [4/10], Step [400/469], Loss: 0.4175354540348053
Epoch [5/10], Step [200/469], Loss: 0.3801096975803375
Epoch [5/10], Step [400/469], Loss: 0.4717716872692108
Epoch [6/10], Step [200/469], Loss: 0.3688111901283264
Epoch [6/10], Step [400/469], Loss: 0.27271077036857605
Epoch [7/10], Step [200/469], Loss: 0.35826289653778076
Epoch [7/10], Step [400/469], Loss: 0.36484241485595703
Epoch [8/10], Step [200/469], Loss: 0.29707223176956177
Epoch [8/10], Step [400/469], Loss: 0.21347501873970032
Epoch [9/10], Step [200/469], Loss: 0.2890476882457733
Epoch [9/10], Step [400/469], Loss: 0.25596883893013
Epoch [10/10], Step [200/469], Loss: 0.21785162389278412
Epoch [10/10], Step [400/469], Loss: 0.22564862668514252
```



```
[10]:  # Test the model
# In test phase, we don't need to compute gradients (for memory efficiency)
with torch.no_grad():
    n_correct = 0
    n_samples= 0
    for images, labels in test_loader:

        images = images.reshape(-1, 28*28).to(device)
        labels = labels.to(device)
        outputs = model(images)
    # max returns (value ,index)
        _, predicted = torch.max(outputs.data, 1)
        n_samples += labels.size(0)
        n_correct += (predicted == labels).sum().item()
    acc= 100.0 * n_correct / n_samples
    print(f'Accuracy of the network on the 10000 test images: {acc} %')
    #89.18
```

```
Accuracy of the network on the 10000 test images: 88.22 %
```

```
[11]: plt.plot(Loss)
plt.plot(Loss2)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



[18]:
```
del model
model = NeuralNet(input_size, hidden_size, num_classes).to(device)
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate,␣
 ↪weight_decay=0.00001)
```

[13]:
```
class EarlyStopping:
    """Early stops the training if validation loss doesn't improve after a given␣
 ↪patience."""
    def __init__(self, patience=7, verbose=False, delta=0, path='checkpoint.pt',␣
 ↪trace_func=print):
        """
        Args:
            patience (int): How long to wait after last time validation loss␣
 ↪improved.
                            Default: 7
            verbose (bool): If True, prints a message for each validation loss␣
 ↪improvement.
                            Default: False
            delta (float): Minimum change in the monitored quantity to qualify␣
 ↪as an improvement.
                            Default: 0
            path (str): Path for the checkpoint to be saved to.
```

```
                                Default: 'checkpoint.pt'
                trace_func (function): trace print function.
                                Default: print
        """
        self.patience = patience
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.val_loss_min = np.Inf
        self.delta = delta
        self.path = path
        self.trace_func = trace_func
    def __call__(self, val_loss, model):

        score = -val_loss

        if self.best_score is None:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
        elif score < self.best_score + self.delta:
            self.counter += 1
            self.trace_func(f'EarlyStopping counter: {self.counter} out of {self.
↪patience}')
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_score = score
            self.save_checkpoint(val_loss, model)
            self.counter = 0

    def save_checkpoint(self, val_loss, model):
        '''Saves model when validation loss decrease.'''
        if self.verbose:
            self.trace_func(f'Validation loss decreased ({self.val_loss_min:.6f}␣
↪--> {val_loss:.6f}).  Saving model ...')
        torch.save(model.state_dict(), self.path)
        self.val_loss_min = val_loss
```

[14]:
```
def train_model(model, batch_size, patience, n_epochs):

      # to track the training loss as the model trains
    train_losses = []
  # to track the validation loss as the model trains
    valid_losses = []
  # to track the average training loss per epoch as the model trains
    avg_train_losses = []
```

```
 # to track the average validation loss per epoch as the model trains
   avg_valid_losses = []
 # initialize the early_stopping object
   early_stopping = EarlyStopping(patience=patience, verbose=True)
   for epoch in range( n_epochs ):
   ###################
   # train the model #
   ###################
       model.train() # prep model for training
   #for i, (images, labels) in enumerate(train_loader):
       for batch, (data, target) in enumerate(train_loader,1):
     # clear the gradients of all optimized variables
           data = data.reshape(-1, 28*28).to(device)
           target = target.to(device)
           optimizer.zero_grad()
     # forward pass: compute predicted outputs by passing inputs to the model
           output = model(data)
     # calculate the loss
           loss = criterion(output, target)
     # backward pass: compute gradient of the loss with respect to model␣
↪parameters
           loss.backward()
     # perform a single optimization step (parameter update)
           optimizer.step()
     # record training loss
           train_losses.append(loss.item())
   ######################
   # validate the model #
   ######################
       model.eval() # prep model for evaluation
       for data, target in valid_loader:
           data = data.reshape(-1, 28*28).to(device)
           target = target.to(device)
         # forward pass: compute predicted outputs by passing inputs to the␣
↪model
           output = model(data)
         # calculate the loss
           loss = criterion(output, target)
         # record validation loss
           valid_losses.append(loss.item())
   # print training/validation statistics
   # calculate average loss over an epoch
       train_loss = np.average(train_losses)
       valid_loss = np.average(valid_losses)
       avg_train_losses.append(train_loss)
       avg_valid_losses.append(valid_loss)
       epoch_len = len(str(n_epochs))
```

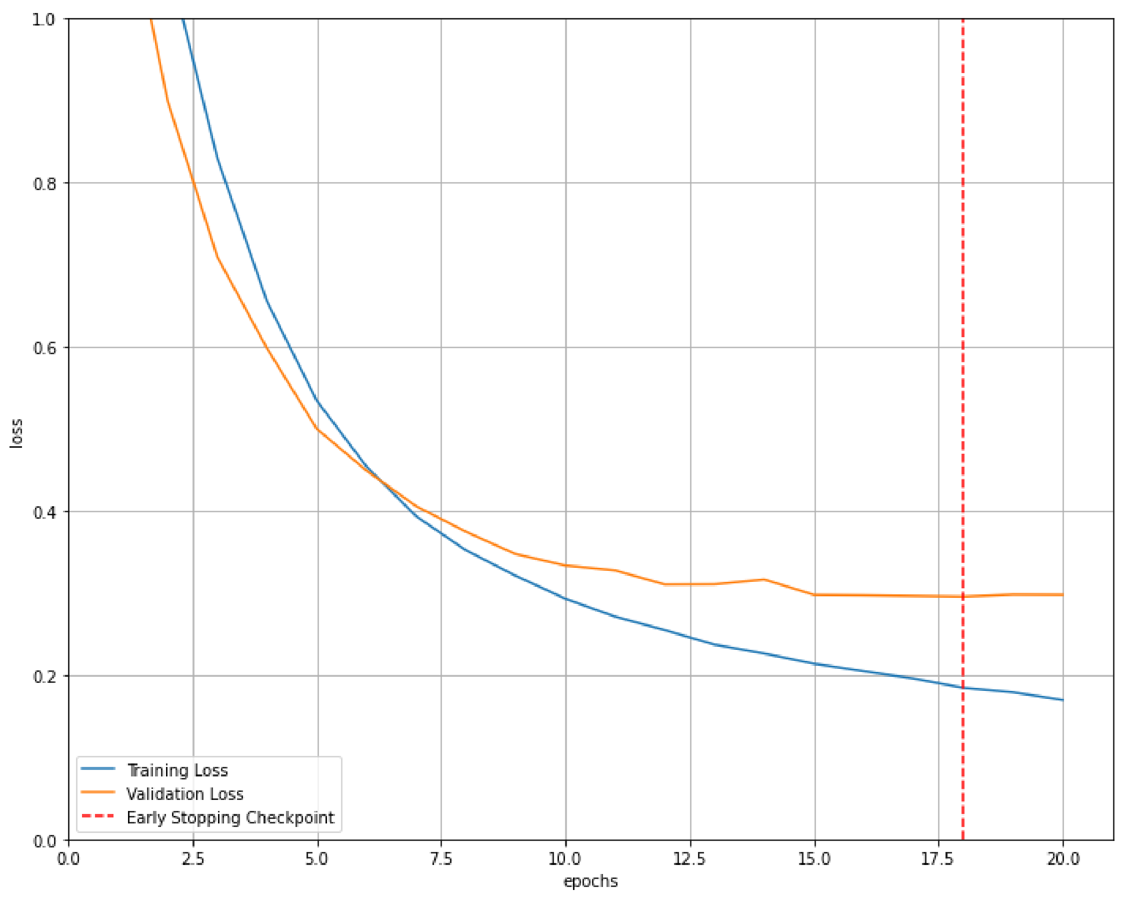
```
        print_msg = (f'[{epoch:>{epoch_len+1}}/{n_epochs:>{epoch_len}}] ' +
        f'train_loss: {train_loss:.5f} ' +
        f'valid_loss: {valid_loss:.5f}')
        print(print_msg)
        # clear lists to track next epoch
        train_losses = []
        valid_losses = []
        # early_stopping needs the validation loss to check if it has decresed,
        # and if it has, it will make a checkpoint of the current model
        early_stopping(valid_loss, model)
        if early_stopping.early_stop:
            print("Early stopping")
            break
  # load the last checkpoint with the best model
    model.load_state_dict(torch.load('checkpoint.pt'))
    return model, avg_train_losses, avg_valid_losses
```

[15]:
```
batch_size = batch_size
n_epochs = num_epochs+10
train_loader, test_loader, valid_loader = create_datasets(batch_size)
# early stopping patience; how long to wait after last time validation loss␣
 ↪improved.
patience = 20
model, train_loss, valid_loss = train_model(model, batch_size, patience,␣
 ↪n_epochs)
```

```
[  0/20] train_loss: 1.44969 valid_loss: 1.19880
Validation loss decreased (inf --> 1.198800).  Saving model ...
[  1/20] train_loss: 1.07484 valid_loss: 0.89989
Validation loss decreased (1.198800 --> 0.899888).  Saving model ...
[  2/20] train_loss: 0.83012 valid_loss: 0.70959
Validation loss decreased (0.899888 --> 0.709591).  Saving model ...
[  3/20] train_loss: 0.65517 valid_loss: 0.59805
Validation loss decreased (0.709591 --> 0.598048).  Saving model ...
[  4/20] train_loss: 0.53436 valid_loss: 0.50000
Validation loss decreased (0.598048 --> 0.500004).  Saving model ...
[  5/20] train_loss: 0.45405 valid_loss: 0.44903
Validation loss decreased (0.500004 --> 0.449034).  Saving model ...
[  6/20] train_loss: 0.39336 valid_loss: 0.40562
Validation loss decreased (0.449034 --> 0.405619).  Saving model ...
[  7/20] train_loss: 0.35202 valid_loss: 0.37461
Validation loss decreased (0.405619 --> 0.374605).  Saving model ...
[  8/20] train_loss: 0.32096 valid_loss: 0.34736
Validation loss decreased (0.374605 --> 0.347361).  Saving model ...
[  9/20] train_loss: 0.29297 valid_loss: 0.33345
Validation loss decreased (0.347361 --> 0.333451).  Saving model ...
[ 10/20] train_loss: 0.27111 valid_loss: 0.32750
Validation loss decreased (0.333451 --> 0.327500).  Saving model ...
```

```
[ 11/20] train_loss: 0.25476 valid_loss: 0.31034
Validation loss decreased (0.327500 --> 0.310338).  Saving model ...
[ 12/20] train_loss: 0.23693 valid_loss: 0.31060
EarlyStopping counter: 1 out of 20
[ 13/20] train_loss: 0.22614 valid_loss: 0.31637
EarlyStopping counter: 2 out of 20
[ 14/20] train_loss: 0.21378 valid_loss: 0.29769
Validation loss decreased (0.310338 --> 0.297687).  Saving model ...
[ 15/20] train_loss: 0.20466 valid_loss: 0.29721
Validation loss decreased (0.297687 --> 0.297210).  Saving model ...
[ 16/20] train_loss: 0.19564 valid_loss: 0.29633
Validation loss decreased (0.297210 --> 0.296327).  Saving model ...
[ 17/20] train_loss: 0.18444 valid_loss: 0.29548
Validation loss decreased (0.296327 --> 0.295476).  Saving model ...
[ 18/20] train_loss: 0.17899 valid_loss: 0.29810
EarlyStopping counter: 1 out of 20
[ 19/20] train_loss: 0.16966 valid_loss: 0.29785
EarlyStopping counter: 2 out of 20
```

[16]:
```
# visualize the loss as the network trained
fig = plt.figure(figsize=(10,8))
plt.plot(range(1,len(train_loss)+1),train_loss, label='Training Loss')
plt.plot(range(1,len(valid_loss)+1),valid_loss,label='Validation Loss')
# find position of lowest validation loss
minposs = valid_loss.index(min(valid_loss))+1
plt.axvline(minposs, linestyle='--', color='r',label='Early Stopping Checkpoint')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.ylim(0, 1) # consistent scale
plt.xlim(0, len(train_loss)+1) # consistent scale
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
fig.savefig('loss_plot.png', bbox_inches='tight')
```



[17]:
```
# initialize lists to monitor test loss and accuracy
test_loss = 0.0
class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
model.eval() # prep model for evaluation
for data, target in test_loader:
  if len(target.data) != batch_size:
    break
  data = data.reshape(-1, 28*28).to(device)
  target = target.to(device)
  # forward pass: compute predicted outputs by passing inputs to the model
  output = model(data)
  # calculate the loss
  loss = criterion(output, target)
  # update test loss
  test_loss += loss.item()*data.size(0)
  # convert output probabilities to predicted class
  _, pred = torch.max(output, 1)
  # compare predictions to true label
  correct = np.squeeze(pred.eq(target.data.view_as(pred)))
  # calculate test accuracy for each object class
  for i in range(batch_size):
    label = target.data[i]
    class_correct[label] += correct[i].item()
    class_total[label] += 1
# calculate and print avg test loss
test_loss = test_loss/len(test_loader.dataset)
print('Test Loss: {:.6f}\n'.format(test_loss))
for i in range(10):
  if class_total[i] > 0:
    print('Test Accuracy of %5s: %2d%% (%2d/%2d)' % (
    str(i), 100 * class_correct[i] / class_total[i],
    np.sum(class_correct[i]), np.sum(class_total[i])))
  else:
    print('Test Accuracy of %5s: N/A (no training examples)' % (classes[i]))
print('\nTest Accuracy (Overall): %5d%% (%5d/%5d)' % (100. * np.
 ↪sum(class_correct) / np.sum(class_total),
                                                   np.sum(class_correct), np.
 ↪sum(class_total)))
```

```
Test Loss: 0.322246

Test Accuracy of     0: 87% (875/1000)
Test Accuracy of     1: 97% (973/997)
Test Accuracy of     2: 83% (831/999)
Test Accuracy of     3: 89% (896/999)
Test Accuracy of     4: 80% (809/999)
Test Accuracy of     5: 96% (960/997)
Test Accuracy of     6: 67% (677/999)
Test Accuracy of     7: 97% (974/999)
Test Accuracy of     8: 97% (971/997)
Test Accuracy of     9: 94% (947/998)

Test Accuracy (Overall):    89% ( 8913/ 9984)
```

## Exercise 3

In this part, you will implement a simple multi-layer perceptron neural network using PyTorch to solve a clothing classification problem. You have to work with the Fashion MNIST dataset, which consists of 10 classes with 60,000 examples in the training set and 10,000 examples in the test set.

- Report the depth effect of different hidden layers.
- Analyze the dropout technique and report its results.
- Use early stopping criteria.
- Become familiar with batch normalization and report its effects.

- The model should be tested for L1 and L2 regularization.
- Add a regularization term for the weight parameter

At first I should mention about regularization part(last part that I used regularization in a different way)

- As we know that this a simple neaural net code and in this problem the depth effect is not very senseble but we can say that more depth will make our model complexity higher which is not good for calculating and timing.
- At the first i should mention that it is a good option to escape from overfitting. I make three types of neural networks: the first nn without dropout, the sec- ond neural network with dropout, and the third neural network dropout with early stopping. At the end my result is that the speed of decreasing the loss function value reduces later.
- We need validation set for this part and i use stopping criteria in the training of the third neural network. I devided the data set at the first part of my code.As i say in previous question it is a good option for reducing our time and be far from overfitting and complexity that we do not need.
- In this part I used batch normalization in the second and third part of my code.the result was good cause that the speed is faster and the accuracy improve.At last the benefits of using this part are: 1.Makes weights easier to initialize; 2.Network train faster,training iteration will actually be slower because of the extra calculations during the forward pass and the additional hyperparameters to train during back propagation. However, it converge much more quickly, so training is faster;3.it is easier to build and faster to train deeper neural networks when using batch normalization.;4.give better results at the end
- My deduction in this part is that L2 is better and I used L2 for my nn model(I used both and choose l2).I should mention that L1 regularization gives output in binary weights from 0 to 1 for the model's features and is adopted for decreasing the number of features in a huge dimensional dataset. L2 regularization disperse the error terms in all the weights that leads to more accurate customized final models and case of that L2 result is better for this dataset.