```python
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim

from datetime import datetime

import torchvision
import torchvision.transforms as transforms

from torchvision import datasets, transforms
import matplotlib.pyplot as plt
%matplotlib inline

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

from PIL import Image

input_size = 28*28
num_classes = 10
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

def accuracy(outputs, labels):
    _, preds = torch.max(outputs, dim = 1)
    return(torch.tensor(torch.sum(preds == labels).item()/
len(preds)))

# We put all of the above:
class MnistModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, xb):
        xb = xb.reshape(-1, 784)
        out = self.linear(xb)
        return(out)

    # We add extra methods
    def training_step(self, batch):
        # when training, we compute the cross entropy, which help us
update weights
        images, labels = batch
        images, labels = images.to(device), labels.to(device)
        out = self(images) ## Generate predictions
        loss = F.cross_entropy(out, labels) ## Calculate the loss
        return(loss)
```

```
    def validation_step(self, batch):
        images, labels = batch
        images, labels = images.to(device), labels.to(device)
        out = self(images) ## Generate predictions
        loss = F.cross_entropy(out, labels) ## Calculate the loss
        # in validation, we want to also look at the accuracy
        # idealy, we would like to save the model when the accuracy is
the highest.
        acc = accuracy(out, labels) ## calculate metrics/accuracy
        return({'val_loss':loss, 'val_acc': acc})

    def validation_epoch_end(self, outputs):
        # at the end of epoch (after running through all the batches)
        batch_losses = [x['val_loss'] for x in outputs]
        epoch_loss = torch.stack(batch_losses).mean()
        batch_accs = [x['val_acc'] for x in outputs]
        epoch_acc = torch.stack(batch_accs).mean()
        return({'val_loss': epoch_loss.item(), 'val_acc' :
epoch_acc.item()})

    def epoch_end(self, epoch,result):
        # log epoch, loss, metrics
        print("Epoch [{}], val_loss: {:.4f}, val_acc:
{:.4f}".format(epoch, result['val_loss'], result['val_acc']))


# a simple helper function to evaluate
def evaluate(model, data_loader):
    # for batch in data_loader, run validation_step
    outputs = [model.validation_step(batch) for batch in data_loader]
    return(model.validation_epoch_end(outputs))


# actually training
def fit(epochs, lr, model, train_loader, val_loader, opt_func =
torch.optim.SGD):
    history = []
    optimizer = opt_func(model.parameters(), lr)
    for epoch in range(epochs):
        ## Training Phase
        for batch in train_loader:
            loss = model.training_step(batch)
            loss.backward() ## backpropagation starts at the loss and
goes through all layers to model inputs
            optimizer.step() ## the optimizer iterate over all
parameters (tensors); use their stored grad to update their values
            optimizer.zero_grad() ## reset gradients

        ## Validation phase
        result = evaluate(model, val_loader)
```

```
        model.epoch_end(epoch, result)
        history.append(result)
    return(history)

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(
                in_channels=1,
                out_channels=16,
                kernel_size=5,
                stride=1,
                padding=2,
            ),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, 5, 1, 2),
            nn.ReLU(),
            nn.MaxPool2d(2),
        )
        # fully connected layer, output 10 classes
        self.out = nn.Linear(32 * 7 * 7, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # flatten the output of conv2 to (batch_size, 32 * 7 * 7)
        x = x.view(x.size(0), -1)
        output = self.out(x)
        return output, x    # return x for visualization

def train(num_epochs, cnn, loaders):
    cnn.train()
    optimizer = optim.Adam(cnn.parameters(), lr = 0.01)
    loss_func = nn.CrossEntropyLoss()
    # Train the model
    total_step = len(loaders)

    for epoch in range(num_epochs):
        epoch_loss = 0
        for i, (images, labels) in enumerate(loaders):

            # gives batch data, normalize x when iterate train_loader
            b_x = images.to(device)   # batch x
            b_y = labels.to(device)   # batch y
            output = cnn(b_x)[0]
            loss = loss_func(output, b_y)
```

```
            epoch_loss += loss.item()

            # clear gradients for this training step
            optimizer.zero_grad()

            # backpropagation, compute gradients
            loss.backward()
            # apply gradients
            optimizer.step()

        print (f'Epoch [{epoch + 1}/{num_epochs}], Loss:
{epoch_loss/total_step}')
```

HOMEWORK 1

Build a classifier for fashion MNIST.

1. Use exactly the same architectures (both densely connected layers and from convolutional layers) as the above MNIST e.g., replace the dataset. Save the Jupyter Notebook in its original format and output a PDF file after training, testing, and validation. Make sure to write down how do they perform (training accuracny, testing accuracy).
2. Improve the architecture. Experiment with different numbers of layers, size of layers, number of filters, size of filters. You are required to make those adjustment to get the highest accuracy. Watch out for overfitting -- we want the highest testing accuracy! Please provide a PDF file of the result, the best test accuracy and the architecture (different numbers of layers, size of layers, number of filters, size of filters)

```
from torchvision import datasets, transforms

# Define a transform to normalize the data
transform = transforms.Compose([
    transforms.ToTensor(),  # to tensor + scale to [0, 1]
    transforms.Normalize((0.5,), (0.5,))  # normalize to [-1, 1]
])

transform_train = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.3, contrast=0.3,
saturation=0.3, hue=0.1),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])
```

```python
# Load the datasets
train_dataset = datasets.FashionMNIST(
    root='data',
    train=True,
    download=True,
    transform=transform_train
)

partitions = [int(len(train_dataset)*0.9),
int(len(train_dataset)*0.1)]
train_dataset, val_dataset = random_split(train_dataset, partitions)

test_dataset = datasets.FashionMNIST(
    root='data',
    train=False,
    download=True,
    transform=transform_test
)

# Create data loaders
fashion_train_loader = DataLoader(train_dataset, batch_size=64,
shuffle=True)
fashion_validation_loader = DataLoader(val_dataset, batch_size=64,
shuffle=True)
fashion_test_loader = DataLoader(test_dataset, batch_size=64,
shuffle=False)

# linear net
linear_model = MnistModel().to(device)

train_metrics_linear = fit(5, 0.001, linear_model,
fashion_train_loader, fashion_validation_loader)
train_metrics_linear
```

```
Epoch [0], val_loss: 1.1157, val_acc: 0.6642
Epoch [1], val_loss: 0.9603, val_acc: 0.6943
Epoch [2], val_loss: 0.8840, val_acc: 0.7138
Epoch [3], val_loss: 0.8505, val_acc: 0.7180
Epoch [4], val_loss: 0.8261, val_acc: 0.7256
```

```
[{'val_loss': 1.1157022714614868, 'val_acc': 0.6642287373542786},
 {'val_loss': 0.9603449106216431, 'val_acc': 0.6943151354789734},
 {'val_loss': 0.8840186595916748, 'val_acc': 0.7138187289237976},
 {'val_loss': 0.8504688739776611, 'val_acc': 0.7180296778678894},
 {'val_loss': 0.8261046409606934, 'val_acc': 0.7256205677986145}]
```

```python
test_metrics_linear = evaluate(linear_model, fashion_test_loader)
test_metrics_linear
```

```
{'val_loss': 0.7531313896179199, 'val_acc': 0.7481091022491455}
```

```
# CNN net
cnn_model = CNN().to(device)
train(num_epochs=5, cnn=cnn_model, loaders=fashion_train_loader)
```

```
Epoch [1/5], Loss: 0.7304294432438381
Epoch [2/5], Loss: 0.6249355729934164
Epoch [3/5], Loss: 0.6130553808469343
Epoch [4/5], Loss: 0.5978283848948953
Epoch [5/5], Loss: 0.5913082581913867
```

```
cnn_model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in fashion_test_loader:
        images, labels = images.to(device), labels.to(device)
        test_output, last_layer = cnn_model(images)
        pred_y = torch.max(test_output, 1)[1].data.squeeze()
        acc = (pred_y == labels).sum().item() / float(labels.size(0))
        pass
print('Test Accuracy of the model on the 10000 test images: %.2f' %
acc)
```

```
Test Accuracy of the model on the 10000 test images: 0.81
```

```
class FashionClassifier(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv_block = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),

            nn.Conv2d(16, 32, kernel_size=5, stride=2, padding=2),
            nn.BatchNorm2d(32),
            nn.ReLU(),

            nn.Conv2d(32, 64, kernel_size=7, stride=1, padding=3),
            nn.ReLU(),

            nn.MaxPool2d(2, 2)  # Output: 7x7
        )

        self.flatten = nn.Flatten()

        self.linear_block = nn.Sequential(
            nn.Linear(7 * 7 * 64, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, 128),
```

```
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(128, 10)
        )

    def forward(self, x):
        x = self.conv_block(x)
        x = self.flatten(x)
        x = self.linear_block(x)
        return x

def train_model(model, train_loader, val_loader, device, lr=0.001,
num_epochs=10):
    model.to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)

    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        running_acc = 0.0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            # Forward
            outputs = model(images)
            loss = criterion(outputs, labels)
            acc = accuracy(outputs, labels)

            # Backward
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            running_acc += acc

        avg_loss = running_loss / len(train_loader)
        avg_acc = running_acc / len(train_loader)

        # Validation
        model.eval()
        val_loss = 0.0
        val_acc = 0.0
        with torch.no_grad():
            for images, labels in val_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss = criterion(outputs, labels)
```

```
                acc = accuracy(outputs, labels)
                val_loss += loss.item()
                val_acc += acc

        val_loss /= len(val_loader)
        val_acc /= len(val_loader)

        print(f"Epoch [{epoch+1}/{num_epochs}] "
              f"Train Loss: {avg_loss:.4f}, Acc: {avg_acc:.4f} | "
              f"Val Loss: {val_loss:.4f}, Acc: {val_acc:.4f}")

custom_net = FashionClassifier()
train_model(custom_net, fashion_train_loader,
fashion_validation_loader, device, lr=0.001, num_epochs=5)
```

```
Epoch [1/5] Train Loss: 0.6327, Acc: 0.7716 | Val Loss: 0.3908, Acc:
0.8568
Epoch [2/5] Train Loss: 0.4147, Acc: 0.8525 | Val Loss: 0.3368, Acc:
0.8737
Epoch [3/5] Train Loss: 0.3644, Acc: 0.8699 | Val Loss: 0.3205, Acc:
0.8877
Epoch [4/5] Train Loss: 0.3370, Acc: 0.8795 | Val Loss: 0.3001, Acc:
0.8892
Epoch [5/5] Train Loss: 0.3170, Acc: 0.8872 | Val Loss: 0.2940, Acc:
0.8982
```

```
def evaluate_model(model, test_loader, device):
    model.eval()
    model.to(device)
    criterion = nn.CrossEntropyLoss()

    test_loss = 0.0
    test_acc = 0.0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
            loss = criterion(outputs, labels)
            acc = accuracy(outputs, labels)
            test_loss += loss.item()
            test_acc += acc

    test_loss /= len(test_loader)
    test_acc /= len(test_loader)
    print(f"Test Loss: {test_loss:.4f}, Accuracy: {test_acc:.4f}")

evaluate_model(custom_net, fashion_test_loader, device)
```

```
Test Loss: 0.2775, Accuracy: 0.9002
```