# `Fashion-MNIST Classification using Neural Network

In this notebook, we'll build a neural network to classify Fashion-MNIST images

import numpy as np  
import torch  
import torch.nn as nn  
import torch.optim as optim  
from torchvision import datasets, transforms  
import matplotlib.pyplot as plt  
from collections import OrderedDict  
  
# Download training and testing data  
transform = transforms.Compose([transforms.ToTensor(),  
 transforms.Normalize((0.5,), (0.5,))])  
train\_ds = datasets.FashionMNIST('F\_MNIST\_data', download=True, train=True, transform=transform)  
test\_ds = datasets.FashionMNIST('F\_MNIST\_data', download=True, train=False, transform=transform)

# split train set into training (80%) and validation set (20%)  
train\_num = len(train\_ds)  
indices = list(range(train\_num))  
np.random.shuffle(indices)  
split = int(np.floor(0.2 \* train\_num))  
val\_idx, train\_idx = indices[:split], indices[split:]  
len(val\_idx), len(train\_idx)

(12000, 48000)

# prepare dataloaders  
train\_sampler = torch.utils.data.sampler.SubsetRandomSampler(train\_idx)  
train\_dl = torch.utils.data.DataLoader(train\_ds, batch\_size=64, sampler=train\_sampler)  
val\_sampler = torch.utils.data.sampler.SubsetRandomSampler(val\_idx)  
val\_dl = torch.utils.data.DataLoader(train\_ds, batch\_size=64, sampler=val\_sampler)  
test\_dl = torch.utils.data.DataLoader(test\_ds, batch\_size=64, shuffle=True)

image, label = next(iter(train\_dl))  
print(image[0].shape, label.shape)  
desc = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle Boot']  
print(desc[label[0].item()])  
plt.imshow(image[0].numpy().squeeze(), cmap='gray');

torch.Size([1, 28, 28]) torch.Size([64])  
Pullover



## Build the network

def network():  
 model = nn.Sequential(OrderedDict([('fc1', nn.Linear(784, 128)),  
 ('relu1', nn.ReLU()),  
 ('drop1', nn.Dropout(0.25)),   
 ('fc2', nn.Linear(128, 64)),  
 ('relu2', nn.ReLU()),  
 ('drop1', nn.Dropout(0.25)),   
 ('output', nn.Linear(64, 10)),  
 ('logsoftmax', nn.LogSoftmax(dim=1))]))  
 # Use GPU if available  
 device = 'cuda' if torch.cuda.is\_available() else 'cpu'  
 model = model.to(device)  
  
 # define the criterion and optimizer  
 loss\_fn = nn.NLLLoss()  
 optimizer = optim.Adam(model.parameters(), lr=0.003)  
  
 return model, loss\_fn, optimizer, device

model, loss\_fn, optimizer, device = network()  
print(model)

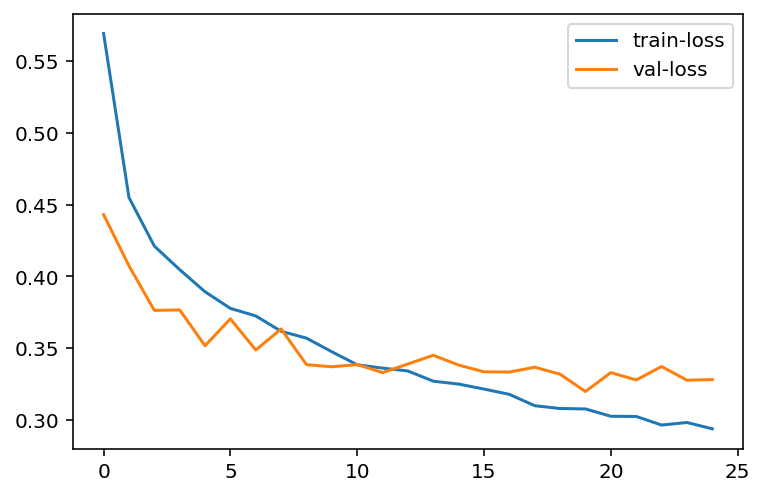
Sequential(  
 (fc1): Linear(in\_features=784, out\_features=128, bias=True)  
 (relu1): ReLU()  
 (drop1): Dropout(p=0.25, inplace=False)  
 (fc2): Linear(in\_features=128, out\_features=64, bias=True)  
 (relu2): ReLU()  
 (output): Linear(in\_features=64, out\_features=10, bias=True)  
 (logsoftmax): LogSoftmax(dim=1)  
)

## Train the network

def train\_validate(model, loss\_fn, optimizer, trainloader, testloader, device, n\_epochs=25):  
 train\_losses = []  
 test\_losses = []  
 for epoch in range(n\_epochs):  
 # Set mode to training - Dropouts will be used here  
 model.train()  
 train\_epoch\_loss = 0  
 for images, labels in trainloader:  
 images, labels = images.to(device), labels.to(device)  
 # flatten the images to batch\_size x 784  
 images = images.view(images.shape[0], -1)  
 # forward pass  
 outputs = model(images)  
 # backpropogation  
 train\_batch\_loss = loss\_fn(outputs, labels)  
 optimizer.zero\_grad()  
 train\_batch\_loss.backward()  
 # Weight updates  
 optimizer.step()  
 train\_epoch\_loss += train\_batch\_loss.item()  
 else:  
 # One epoch of training complete  
 # calculate average training epoch loss  
 train\_epoch\_loss = train\_epoch\_loss/len(trainloader)  
  
 # Now Validate on testset  
 with torch.no\_grad():  
 test\_epoch\_acc = 0  
 test\_epoch\_loss = 0  
 # Set mode to eval - Dropouts will NOT be used here  
 model.eval()  
 for images, labels in testloader:  
 images, labels = images.to(device), labels.to(device)   
 # flatten images to batch\_size x 784  
 images = images.view(images.shape[0], -1)  
 # make predictions   
 test\_outputs = model(images)  
 # calculate test loss  
 test\_batch\_loss = loss\_fn(test\_outputs, labels)  
 test\_epoch\_loss += test\_batch\_loss  
   
 # get probabilities, extract the class associated with highest probability  
 proba = torch.exp(test\_outputs)  
 \_, pred\_labels = proba.topk(1, dim=1)  
   
 # compare actual labels and predicted labels  
 result = pred\_labels == labels.view(pred\_labels.shape)  
 batch\_acc = torch.mean(result.type(torch.FloatTensor))  
 test\_epoch\_acc += batch\_acc.item()  
 else:  
 # One epoch of training and validation done  
 # calculate average testing epoch loss  
 test\_epoch\_loss = test\_epoch\_loss/len(testloader)  
 # calculate accuracy as correct\_pred/total\_samples  
 test\_epoch\_acc = test\_epoch\_acc/len(testloader)  
 # save epoch losses for plotting  
 train\_losses.append(train\_epoch\_loss)  
 test\_losses.append(test\_epoch\_loss)  
 # print stats for this epoch  
 print(f'Epoch: {epoch} -> train\_loss: {train\_epoch\_loss:.19f}, val\_loss: {test\_epoch\_loss:.19f}, ',  
 f'val\_acc: {test\_epoch\_acc\*100:.2f}%')  
 # Finally plot losses  
 plt.plot(train\_losses, label='train-loss')  
 plt.plot(test\_losses, label='val-loss')  
 plt.legend()  
 plt.show()

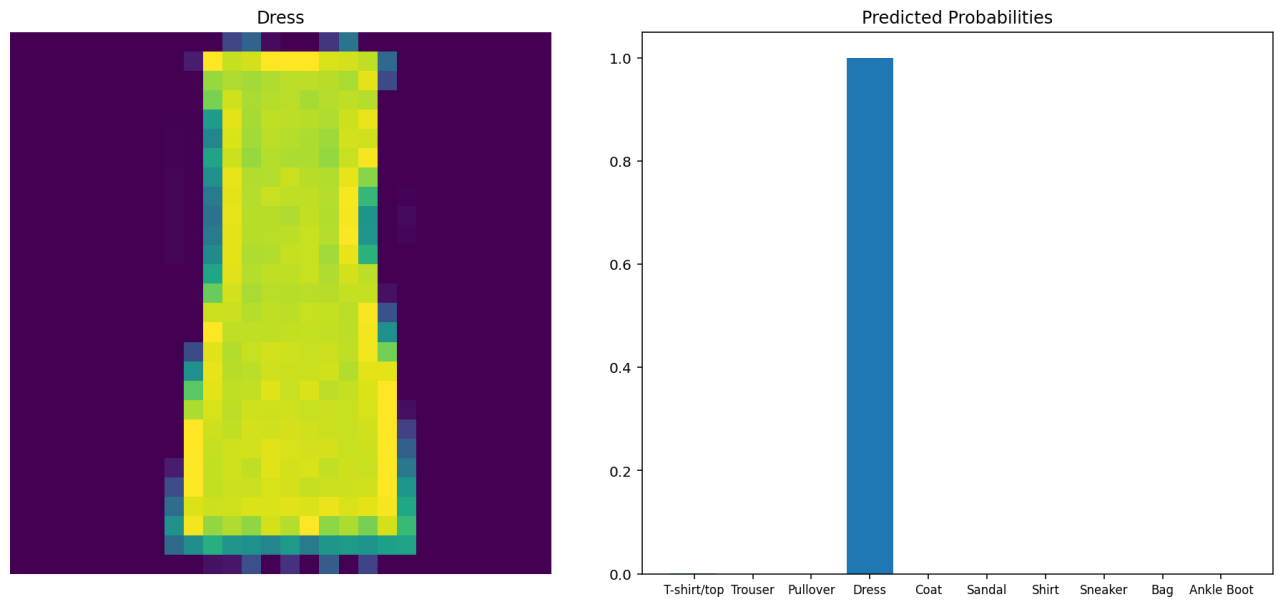
# Train and validate  
train\_validate(model, loss\_fn, optimizer, train\_dl, val\_dl, device)

Epoch: 0 -> train\_loss: 0.5692988239328066191, val\_loss: 0.4430967867374420166, val\_acc: 83.27%  
Epoch: 1 -> train\_loss: 0.4549953033725420704, val\_loss: 0.4073663055896759033, val\_acc: 84.97%  
Epoch: 2 -> train\_loss: 0.4211076561212539482, val\_loss: 0.3762904107570648193, val\_acc: 86.11%  
Epoch: 3 -> train\_loss: 0.4046927829583485958, val\_loss: 0.3766101896762847900, val\_acc: 86.46%  
Epoch: 4 -> train\_loss: 0.3892695648074150361, val\_loss: 0.3516367673873901367, val\_acc: 87.34%  
Epoch: 5 -> train\_loss: 0.3776730005542437429, val\_loss: 0.3703932464122772217, val\_acc: 86.71%  
Epoch: 6 -> train\_loss: 0.3724320774773756670, val\_loss: 0.3486533761024475098, val\_acc: 87.37%  
Epoch: 7 -> train\_loss: 0.3617128521899382054, val\_loss: 0.3633341491222381592, val\_acc: 87.08%  
Epoch: 8 -> train\_loss: 0.3569285793701807430, val\_loss: 0.3384532332420349121, val\_acc: 87.58%  
Epoch: 9 -> train\_loss: 0.3474347092111905178, val\_loss: 0.3369818329811096191, val\_acc: 87.88%  
Epoch: 10 -> train\_loss: 0.3383618565599123551, val\_loss: 0.3385397493839263916, val\_acc: 87.97%  
Epoch: 11 -> train\_loss: 0.3360619519948959133, val\_loss: 0.3329190611839294434, val\_acc: 87.97%  
Epoch: 12 -> train\_loss: 0.3340372469524542365, val\_loss: 0.3389228284358978271, val\_acc: 88.00%  
Epoch: 13 -> train\_loss: 0.3269288233717282388, val\_loss: 0.3449897170066833496, val\_acc: 87.77%  
Epoch: 14 -> train\_loss: 0.3249198505381742930, val\_loss: 0.3381777107715606689, val\_acc: 87.75%  
Epoch: 15 -> train\_loss: 0.3214456404149532154, val\_loss: 0.3334679007530212402, val\_acc: 87.97%  
Epoch: 16 -> train\_loss: 0.3177473722100258025, val\_loss: 0.3333184421062469482, val\_acc: 87.75%  
Epoch: 17 -> train\_loss: 0.3098344492912292747, val\_loss: 0.3366726040840148926, val\_acc: 88.12%  
Epoch: 18 -> train\_loss: 0.3078885127405325828, val\_loss: 0.3318108916282653809, val\_acc: 88.41%  
Epoch: 19 -> train\_loss: 0.3075984100500742668, val\_loss: 0.3197179138660430908, val\_acc: 88.56%  
Epoch: 20 -> train\_loss: 0.3024848873019218565, val\_loss: 0.3329302072525024414, val\_acc: 88.16%  
Epoch: 21 -> train\_loss: 0.3023494395713011151, val\_loss: 0.3277964591979980469, val\_acc: 88.35%  
Epoch: 22 -> train\_loss: 0.2963562680284182460, val\_loss: 0.3371585309505462646, val\_acc: 88.25%  
Epoch: 23 -> train\_loss: 0.2981447652975718343, val\_loss: 0.3276244103908538818, val\_acc: 88.55%  
Epoch: 24 -> train\_loss: 0.2937244445780913260, val\_loss: 0.3280715346336364746, val\_acc: 88.67%



### Predict a single image

%matplotlib inline  
%config InlineBackend.figure\_format = 'retina'  
  
# Test out the network!  
dataiter = iter(test\_dl)  
images, labels = dataiter.next()  
images, labels = images.to(device), labels.to(device)  
index = 49  
img, label = images[index], labels[index]  
# Convert 2D image to 1D vector  
img = img.view(img.shape[0], -1)  
  
# Calculate the class probabilities (softmax) for img  
proba = torch.exp(model(img))  
  
# Plot the image and probabilities  
desc = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker','Bag','Ankle Boot']  
fig, (ax1, ax2) = plt.subplots(figsize=(13, 6), nrows=1, ncols=2)  
ax1.axis('off')  
ax1.imshow(images[index].cpu().numpy().squeeze())  
ax1.set\_title(desc[label.item()])  
ax2.bar(range(10), proba.detach().cpu().numpy().squeeze())  
ax2.set\_xticks(range(10))  
ax2.set\_xticklabels(desc, size='small')  
ax2.set\_title('Predicted Probabilities')  
plt.tight\_layout()



## Validate on test set

# Validate  
with torch.no\_grad():  
 batch\_acc = []  
 model.eval()  
 for images, labels in test\_dl:  
 images, labels = images.to(device), labels.to(device)  
 # flatten images to batch\_size x 784  
 images = images.view(images.shape[0], -1)  
 # make predictions and get probabilities  
 proba = torch.exp(model(images))  
 # extract the class associted with highest probability  
 \_, pred\_labels = proba.topk(1, dim=1)  
 # compare actual labels and predicted labels  
 result = pred\_labels == labels.view(pred\_labels.shape)  
 acc = torch.mean(result.type(torch.FloatTensor))  
 batch\_acc.append(acc.item())  
 else:  
 print(f'Test Accuracy: {torch.mean(torch.tensor(batch\_acc))\*100:.2f}%')

Test Accuracy: 87.42%

## More powerful model

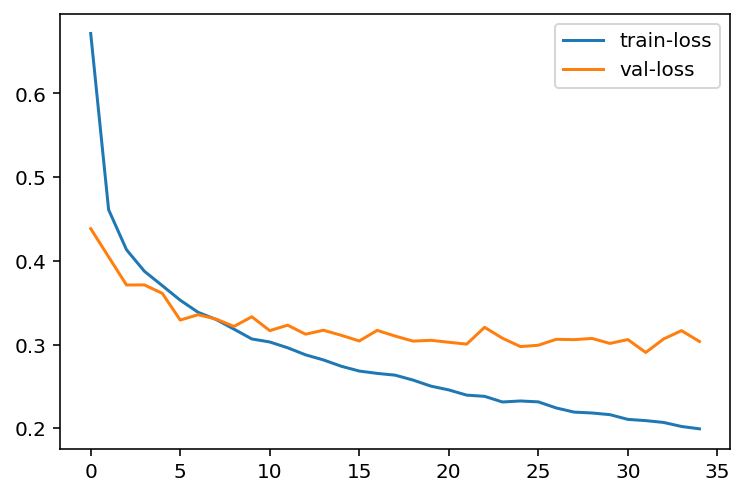
# Redefine network with dropout layers in between  
def network():  
 model = nn.Sequential(OrderedDict([('fc1', nn.Linear(784, 392)),  
 ('relu1', nn.ReLU()),  
 ('drop1', nn.Dropout(0.25)),  
 ('fc12', nn.Linear(392, 196)),  
 ('relu2', nn.ReLU()),  
 ('drop2', nn.Dropout(0.25)),  
 ('fc3', nn.Linear(196, 98)),  
 ('relu3', nn.ReLU()),  
 ('drop3', nn.Dropout(0.25)),   
 ('fc4', nn.Linear(98, 49)),  
 ('relu4', nn.ReLU()),  
 ('output', nn.Linear(49, 10)),  
 ('logsoftmax', nn.LogSoftmax(dim=1))]))  
   
 # Use GPU if available  
 device = 'cuda' if torch.cuda.is\_available() else 'cpu'  
 model = model.to(device)  
  
 # define the criterion and optimizer  
 loss\_fn = nn.NLLLoss()  
 optimizer = optim.Adam(model.parameters(), lr=0.0007)  
  
 return model, loss\_fn, optimizer, device

model, loss\_fn, optimizer, device = network()  
model

Sequential(  
 (fc1): Linear(in\_features=784, out\_features=392, bias=True)  
 (relu1): ReLU()  
 (drop1): Dropout(p=0.25, inplace=False)  
 (fc12): Linear(in\_features=392, out\_features=196, bias=True)  
 (relu2): ReLU()  
 (drop2): Dropout(p=0.25, inplace=False)  
 (fc3): Linear(in\_features=196, out\_features=98, bias=True)  
 (relu3): ReLU()  
 (drop3): Dropout(p=0.25, inplace=False)  
 (fc4): Linear(in\_features=98, out\_features=49, bias=True)  
 (relu4): ReLU()  
 (output): Linear(in\_features=49, out\_features=10, bias=True)  
 (logsoftmax): LogSoftmax(dim=1)  
)

# Train and validate again with new architecture  
train\_validate(model, loss\_fn, optimizer, train\_dl, val\_dl, device, n\_epochs=35)

Epoch: 0 -> train\_loss: 0.6715717662970225321, val\_loss: 0.4384306371212005615, val\_acc: 83.58%  
Epoch: 1 -> train\_loss: 0.4610257453123728366, val\_loss: 0.4046924412250518799, val\_acc: 84.90%  
Epoch: 2 -> train\_loss: 0.4132076732317606638, val\_loss: 0.3710882067680358887, val\_acc: 86.29%  
Epoch: 3 -> train\_loss: 0.3874667741258939202, val\_loss: 0.3711968660354614258, val\_acc: 86.47%  
Epoch: 4 -> train\_loss: 0.3702941476901372431, val\_loss: 0.3610889613628387451, val\_acc: 87.08%  
Epoch: 5 -> train\_loss: 0.3529314926067987912, val\_loss: 0.3292031884193420410, val\_acc: 88.07%  
Epoch: 6 -> train\_loss: 0.3384442613820234924, val\_loss: 0.3356113731861114502, val\_acc: 87.88%  
Epoch: 7 -> train\_loss: 0.3299360174735387341, val\_loss: 0.3304152488708496094, val\_acc: 88.25%  
Epoch: 8 -> train\_loss: 0.3183699831465879870, val\_loss: 0.3215932250022888184, val\_acc: 88.32%  
Epoch: 9 -> train\_loss: 0.3066120248536268744, val\_loss: 0.3331522047519683838, val\_acc: 88.21%  
Epoch: 10 -> train\_loss: 0.3030157501995563440, val\_loss: 0.3165770173072814941, val\_acc: 88.86%  
Epoch: 11 -> train\_loss: 0.2961013345122337492, val\_loss: 0.3231025636196136475, val\_acc: 88.11%  
Epoch: 12 -> train\_loss: 0.2876731425921122409, val\_loss: 0.3123144209384918213, val\_acc: 88.69%  
Epoch: 13 -> train\_loss: 0.2815732069909572810, val\_loss: 0.3170706033706665039, val\_acc: 88.46%  
Epoch: 14 -> train\_loss: 0.2739904375871022313, val\_loss: 0.3109590411186218262, val\_acc: 88.96%  
Epoch: 15 -> train\_loss: 0.2682814720571041200, val\_loss: 0.3042424619197845459, val\_acc: 89.29%  
Epoch: 16 -> train\_loss: 0.2654960344235102188, val\_loss: 0.3168744444847106934, val\_acc: 89.11%  
Epoch: 17 -> train\_loss: 0.2633762069841226183, val\_loss: 0.3101049065589904785, val\_acc: 88.79%  
Epoch: 18 -> train\_loss: 0.2575348454515139496, val\_loss: 0.3040629029273986816, val\_acc: 89.22%  
Epoch: 19 -> train\_loss: 0.2502811358968416999, val\_loss: 0.3050162196159362793, val\_acc: 89.39%  
Epoch: 20 -> train\_loss: 0.2457498750587304437, val\_loss: 0.3026749193668365479, val\_acc: 89.45%  
Epoch: 21 -> train\_loss: 0.2395246799389521286, val\_loss: 0.3004771769046783447, val\_acc: 89.69%  
Epoch: 22 -> train\_loss: 0.2380762482732534380, val\_loss: 0.3205380141735076904, val\_acc: 89.53%  
Epoch: 23 -> train\_loss: 0.2313262286484241426, val\_loss: 0.3075141012668609619, val\_acc: 89.69%  
Epoch: 24 -> train\_loss: 0.2325393498738606846, val\_loss: 0.2975164949893951416, val\_acc: 89.84%  
Epoch: 25 -> train\_loss: 0.2314718961914380424, val\_loss: 0.2991222441196441650, val\_acc: 89.60%  
Epoch: 26 -> train\_loss: 0.2241996667136748611, val\_loss: 0.3062023818492889404, val\_acc: 89.69%  
Epoch: 27 -> train\_loss: 0.2192408299843470187, val\_loss: 0.3058206140995025635, val\_acc: 89.56%  
Epoch: 28 -> train\_loss: 0.2181850755562385058, val\_loss: 0.3072552680969238281, val\_acc: 89.46%  
Epoch: 29 -> train\_loss: 0.2161911162187655777, val\_loss: 0.3013016283512115479, val\_acc: 90.00%  
Epoch: 30 -> train\_loss: 0.2105504951179027473, val\_loss: 0.3058974742889404297, val\_acc: 89.90%  
Epoch: 31 -> train\_loss: 0.2090706681162118885, val\_loss: 0.2904951870441436768, val\_acc: 89.88%  
Epoch: 32 -> train\_loss: 0.2068803765972455355, val\_loss: 0.3066459596157073975, val\_acc: 89.62%  
Epoch: 33 -> train\_loss: 0.2020631300210952719, val\_loss: 0.3165600001811981201, val\_acc: 89.64%  
Epoch: 34 -> train\_loss: 0.1992565853993097935, val\_loss: 0.3035983443260192871, val\_acc: 89.69%



## Validate on test set

# Validate  
with torch.no\_grad():  
 model.eval()  
 batch\_acc = []  
 for images, labels in test\_dl:  
 images, labels = images.to(device), labels.to(device)  
 # flatten images to batch\_size x 784  
 images = images.view(images.shape[0], -1)  
 # make predictions and get probabilities  
 proba = torch.exp(model(images))  
 # extract the class associted with highest probability  
 \_, pred\_labels = proba.topk(1, dim=1)  
 # compare actual labels and predicted labels  
 result = pred\_labels == labels.view(pred\_labels.shape)  
 acc = torch.mean(result.type(torch.FloatTensor))  
 batch\_acc.append(acc.item())  
 else:  
 print(f'Accuracy: {torch.mean(torch.tensor(batch\_acc))\*100:.2f}%')

Accuracy: 88.65%