Fashion-MNIST Classification using Neural Network

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

In [1]:
```
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, transfor
test_ds = datasets.FashionMNIST('F_MNIST_data', download=True, train=False, transfor
```

```
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 F_MNIST_data\FashionMNIST\raw\train-images-idx3-ubyte.gz

Extracting F_MNIST_data\FashionMNIST\raw\train-images-idx3-ubyte.gz to F_MNIST_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 F_MNIST_data\FashionMNIST\raw\train-labels-idx1-ubyte.gz

Extracting F_MNIST_data\FashionMNIST\raw\train-labels-idx1-ubyte.gz to F_MNIST_data
\FashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-i
dx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-images-i
dx3-ubyte.gz to F_MNIST_data\FashionMNIST\raw\t10k-images-idx3-ubyte.gz

Extracting F_MNIST_data\FashionMNIST\raw\t10k-images-idx3-ubyte.gz to F_MNIST_data\F
ashionMNIST\raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-i
dx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/t10k-labels-i
dx1-ubyte.gz to F_MNIST_data\FashionMNIST\raw\t10k-labels-idx1-ubyte.gz

Extracting F_MNIST_data\FashionMNIST\raw\t10k-labels-idx1-ubyte.gz to F_MNIST_data\F
ashionMNIST\raw


C:\Users\Kushal Gupta\anaconda3\lib\site-packages\torchvision\datasets\mnist.py:498:
UserWarning: The given NumPy array is not writeable, and PyTorch does not support no
n-writeable tensors. This means you can write to the underlying (supposedly non-writ
eable) NumPy array using the tensor. You may want to copy the array to protect its d
ata or make it writeable before converting it to a tensor. This type of warning will
be suppressed for the rest of this program. (Triggered internally at  ..\torch\csrc
\utils\tensor_numpy.cpp:180.)
  return torch.from_numpy(parsed.astype(m[2], copy=False)).view(*s)
```

In [2]:
```
# split train set into training (80%) and validation set (20%)
train_num = len(train_ds)
```

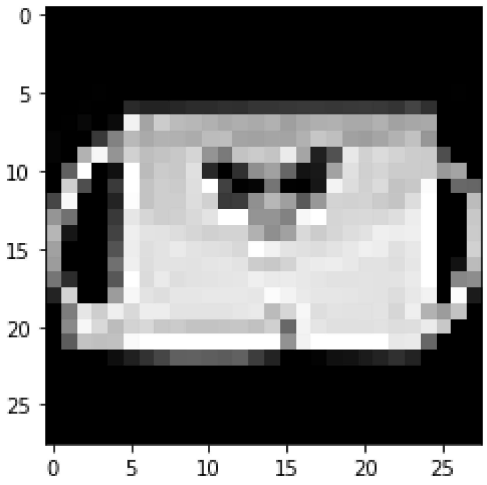
```
indices = list(range(train_num))
np.random.shuffle(indices)
split = int(np.floor(0.2 * train_num))
val_idx, train_idx = indices[:split], indices[split:]
len(val_idx), len(train_idx)
```

Out[2]: (12000, 48000)

In [3]:
```
# prepare dataloaders
train_sampler = torch.utils.data.sampler.SubsetRandomSampler(train_idx)
train_dl = torch.utils.data.DataLoader(train_ds, batch_size=64, sampler=train_sample
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)
```

In [4]:
```
image, label = next(iter(train_dl))
print(image[0].shape, label.shape)
desc = ['T-shirt/top','Trouser','Pullover','Dress','Coat','Sandal','Shirt','Sneaker'
print(desc[label[0].item()])
plt.imshow(image[0].numpy().squeeze(), cmap='gray');
```

```
torch.Size([1, 28, 28]) torch.Size([64])
Bag
```



Build the network

In [5]:
```
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
```

In [6]:
```
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

In [7]:
```
def train_validate(model, loss_fn, optimizer, trainloader, testloader, device, n_epo
    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 p
                    proba = torch.exp(test_outputs)
                    _, pred_labels = proba.topk(1, dim=1)

                    # compare actual labels and predicted labels
```

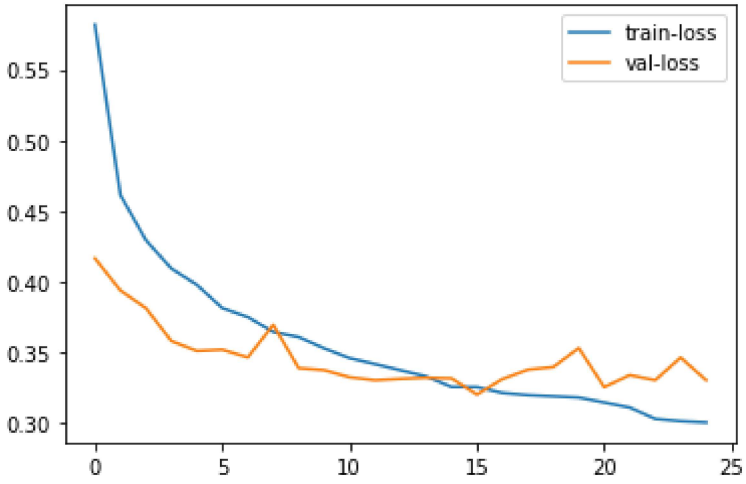
```
                    result = pred_labels == labels.view(pred_labels.shape)
                    batch_acc = torch.mean(result.type(torch.FloatTensor))
                    test_epoch_acc += batch_acc.item()
                else:
                    # One epoch of training and validation done
                    # 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}, v
                          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()
```

In [8]:

```
# Train and validate
train_validate(model, loss_fn, optimizer, train_dl, val_dl, device)
```

```
Epoch: 0 -> train_loss: 0.5820046128431956012, val_loss: 0.4161849021911621094,  val
_acc: 84.60%
Epoch: 1 -> train_loss: 0.4609627281824747458, val_loss: 0.3933480381965637207,  val
_acc: 85.46%
Epoch: 2 -> train_loss: 0.4290109670062859926, val_loss: 0.3808104693889617920,  val
_acc: 86.36%
Epoch: 3 -> train_loss: 0.4089395367503166345, val_loss: 0.3574513792991638184,  val
_acc: 86.78%
Epoch: 4 -> train_loss: 0.3974039002855618841, val_loss: 0.3504286110401153564,  val
_acc: 87.54%
Epoch: 5 -> train_loss: 0.3808735365271568307, val_loss: 0.3513041138648986816,  val
_acc: 86.95%
Epoch: 6 -> train_loss: 0.3744849762717882835, val_loss: 0.3458298742771148682,  val
_acc: 87.48%
Epoch: 7 -> train_loss: 0.3639392338792483139, val_loss: 0.3689590692520141602,  val
_acc: 86.85%
Epoch: 8 -> train_loss: 0.3602720575730005703, val_loss: 0.3381931185722351074,  val
_acc: 88.10%
Epoch: 9 -> train_loss: 0.3522783599694570000, val_loss: 0.3368195593357086182,  val
_acc: 88.01%
Epoch: 10 -> train_loss: 0.3452950304845968654, val_loss: 0.3317974507808685303,  va
l_acc: 87.99%
Epoch: 11 -> train_loss: 0.3410937736233075612, val_loss: 0.3296695351600646973,  va
l_acc: 88.10%
Epoch: 12 -> train_loss: 0.3367249659498532388, val_loss: 0.3306388258934020996,  va
l_acc: 88.14%
Epoch: 13 -> train_loss: 0.3324477933247884098, val_loss: 0.3314307928085327148,  va
l_acc: 88.55%
Epoch: 14 -> train_loss: 0.3249294434090455597, val_loss: 0.3310419321060180664,  va
l_acc: 87.65%
Epoch: 15 -> train_loss: 0.3248153601984182792, val_loss: 0.3195063769817352295,  va
l_acc: 88.58%
Epoch: 16 -> train_loss: 0.3206507736047108925, val_loss: 0.3304931819438934326,  va
l_acc: 88.32%
Epoch: 17 -> train_loss: 0.3190858544508616279, val_loss: 0.3371344506740570068,  va
l_acc: 88.11%
Epoch: 18 -> train_loss: 0.3182504490117232243, val_loss: 0.3390006124973297119,  va
l_acc: 88.37%
Epoch: 19 -> train_loss: 0.3173692301313082442, val_loss: 0.3524542450904846191,  va
l_acc: 87.64%
Epoch: 20 -> train_loss: 0.3138494770427545100, val_loss: 0.3247825801372528076,  va
l_acc: 88.60%
```

```
Epoch: 21 -> train_loss: 0.3103037677804629246, val_loss: 0.3332910537719726562,  va
l_acc: 88.55%
Epoch: 22 -> train_loss: 0.3022127579450607504, val_loss: 0.3296991884708404541,  va
l_acc: 87.95%
Epoch: 23 -> train_loss: 0.3005808478891849322, val_loss: 0.3457268774509429932,  va
l_acc: 87.95%
Epoch: 24 -> train_loss: 0.2996912881930668959, val_loss: 0.3297417163848876953,  va
l_acc: 88.69%
```



Predict a single image

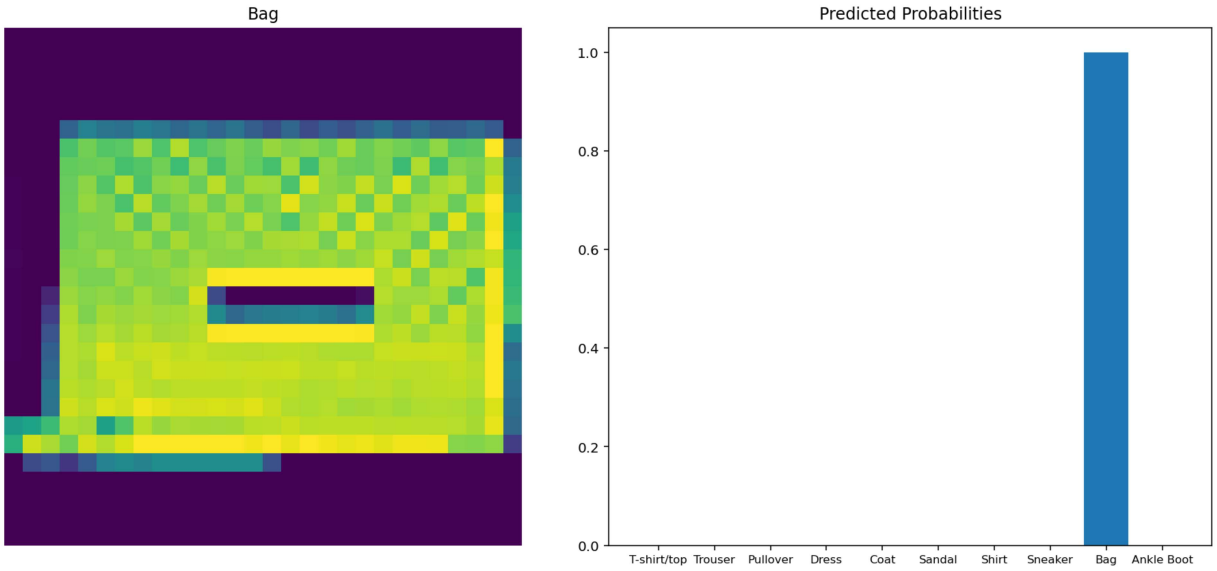
In [9]:

```
%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'
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

In [10]:

```
# 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: 87.68%
```

More powerful model

In [11]:

```
# 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
```

In [12]:

```
model, loss_fn, optimizer, device = network()
model
```

Out[12]:

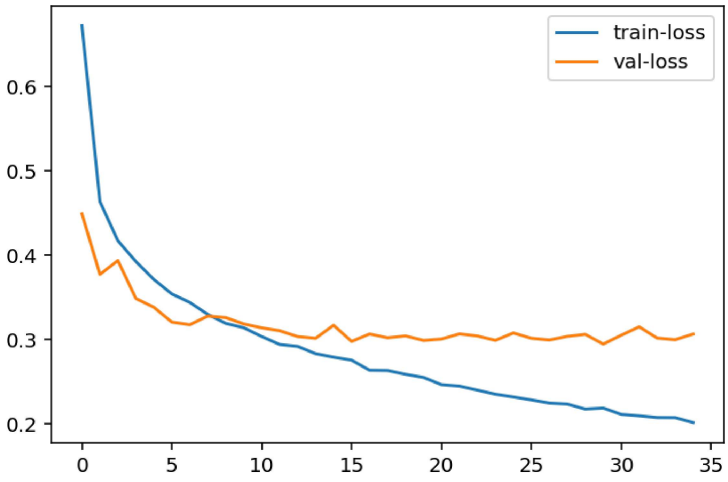
```
Sequential(
  (fc1): Linear(in_features=784, out_features=392, bias=True)
  (relu1): ReLU()
  (drop1): Dropout(p=0.25, inplace=False)
  (fc12): Linear(in_features=392, out_features=196, bias=True)
  (relu2): ReLU()
  (drop2): Dropout(p=0.25, inplace=False)
  (fc3): Linear(in_features=196, out_features=98, 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)
)
```

In [13]:

```
# 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: 0.6722108484506607029, val_loss: 0.4491010308265686035,  val
_acc: 82.74%
Epoch: 1 -> train_loss: 0.4634778929352760257, val_loss: 0.3772047758102416992,  val
_acc: 86.14%
Epoch: 2 -> train_loss: 0.4171390691598256262, val_loss: 0.3936618566513061523,  val
_acc: 85.47%
Epoch: 3 -> train_loss: 0.3925043706099192176, val_loss: 0.3486233651638031006,  val
_acc: 87.63%
Epoch: 4 -> train_loss: 0.3712360138694445477, val_loss: 0.3383010029792785645,  val
_acc: 87.87%
Epoch: 5 -> train_loss: 0.3540800571640332350, val_loss: 0.3206551671028137207,  val
_acc: 88.44%
Epoch: 6 -> train_loss: 0.3440062716106573881, val_loss: 0.3175850808620452881,  val
_acc: 88.67%
Epoch: 7 -> train_loss: 0.3298225763340791317, val_loss: 0.3278072476387023926,  val
_acc: 88.32%
Epoch: 8 -> train_loss: 0.3191555908620357651, val_loss: 0.3260203897953033447,  val
_acc: 88.54%
Epoch: 9 -> train_loss: 0.3139979259272416279, val_loss: 0.3184978067874908447,  val
_acc: 88.92%
Epoch: 10 -> train_loss: 0.3036111173331737523, val_loss: 0.3139036297798156738,  va
l_acc: 89.04%
Epoch: 11 -> train_loss: 0.2941199940641721078, val_loss: 0.3104389607906341553,  va
l_acc: 89.13%
Epoch: 12 -> train_loss: 0.2918689980705579390, val_loss: 0.3035447299480438232,  va
l_acc: 89.39%
Epoch: 13 -> train_loss: 0.2831989902059237063, val_loss: 0.3014743924140930176,  va
l_acc: 89.17%
Epoch: 14 -> train_loss: 0.2790738928020000675, val_loss: 0.3169613778591156006,  va
l_acc: 88.60%
Epoch: 15 -> train_loss: 0.2755716178218523549, val_loss: 0.2977825701236724854,  va
l_acc: 89.57%
Epoch: 16 -> train_loss: 0.2636287015279134005, val_loss: 0.3065706491470336914,  va
l_acc: 89.42%
```

```
Epoch: 17 -> train_loss: 0.2633225894769032904, val_loss: 0.3021140694618225098,  va
l_acc: 89.77%
Epoch: 18 -> train_loss: 0.2588147721936305268, val_loss: 0.3042859435081481934,  va
l_acc: 89.15%
Epoch: 19 -> train_loss: 0.2550887158811092625, val_loss: 0.2989240288734436035,  va
l_acc: 89.76%
Epoch: 20 -> train_loss: 0.2464823014338811125, val_loss: 0.3005456626415252686,  va
l_acc: 89.84%
Epoch: 21 -> train_loss: 0.2447312127848466357, val_loss: 0.3067142367362976074,  va
l_acc: 89.60%
Epoch: 22 -> train_loss: 0.2400031385223070912, val_loss: 0.3041906654834747314,  va
l_acc: 89.45%
Epoch: 23 -> train_loss: 0.2351681134849786803, val_loss: 0.2990351617336273193,  va
l_acc: 89.92%
Epoch: 24 -> train_loss: 0.2319054315139849887, val_loss: 0.3078336715698242188,  va
l_acc: 89.90%
Epoch: 25 -> train_loss: 0.2284384021411339505, val_loss: 0.3014802038669586182,  va
l_acc: 89.44%
Epoch: 26 -> train_loss: 0.2246469232439994856, val_loss: 0.2994275093078613281,  va
l_acc: 90.15%
Epoch: 27 -> train_loss: 0.2234746333658695350, val_loss: 0.3037487566471099854,  va
l_acc: 89.63%
Epoch: 28 -> train_loss: 0.2175256675183772914, val_loss: 0.3061659932136535645,  va
l_acc: 90.07%
Epoch: 29 -> train_loss: 0.2187462772876024308, val_loss: 0.2945981025695800781,  va
l_acc: 90.13%
Epoch: 30 -> train_loss: 0.2112338943332433827, val_loss: 0.3051328063011169434,  va
l_acc: 90.01%
Epoch: 31 -> train_loss: 0.2095508428364992182, val_loss: 0.3150895833969116211,  va
l_acc: 89.75%
Epoch: 32 -> train_loss: 0.2074856148908535702, val_loss: 0.3016926348209381104,  va
l_acc: 90.23%
Epoch: 33 -> train_loss: 0.2073508629798889169, val_loss: 0.2997615933418273926,  va
l_acc: 90.08%
Epoch: 34 -> train_loss: 0.2015710917909940003, val_loss: 0.3065292239189147949,  va
l_acc: 89.92%
```



Validate on test set

In [14]:

```
# 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: 89.11%
```