AIMS 5702 Artificial Intelligence in Practice (Fall 2025-26)

AIMS 5702 Assignment 2 (in-class). Basic Training

Please complete this colab and:

- Download it as **both** .ipynb file and .pdf file (click on file -> download -> download as .ipynb file)
- Submit **both** .pdf and .ipynb file to blackboard

No GPU is required. Just use CPU host time is enough.

Deadline of submission: 2025/10/02 23:59pm (Thursday)

Coding style requirement

Please follow the following coding style rule

- Do not have more than 80 chars in a line.
- Add type hint for all functions, including utility functions.
- Add docstring for main function (e.g. interface function)

Machine

Most of these question shall be solvable using CPU (and using GPU did not accelerate too much), except Question A.2.

Setup

First, let us import all necessary libraries for this one

```
# PyTorch and numpy
import torch
import numpy as np
# Building neural network layers
import torch.nn as nn
# Optimization algorithms for training neural networks
import torch.optim as optim
# DataLoader for creating data loading iterators
from torch.utils.data import DataLoader
# Computer vision datasets and transforms
import torchvision
# Transforms for data preprocessing and augmentation
import torchvision.transforms as transforms
# A popular library for plotting and visualization
import matplotlib.pyplot as plt
# Library for matrix operation
import numpy as np
# Scipy signal library
import scipy.signal
```

```
# Typing
from typing import Optional
```

```
#@title Utilities

def is_same_tensor(result: torch.Tensor,
                   ref: torch.Tensor,
                   tol: Optional[float]=None) -> bool:
  """
  Check if two tensors are the same.

  Args:
    result: Results by your code.
    ref: Ground truth result.

  Return:
    Whether result and ref are the same.
  """
  if (not isinstance(result, torch.Tensor) or
      not isinstance(ref, torch.Tensor)):
    return False
  if result.dtype != ref.dtype:
    result = result.to(ref.dtype)
  if tol is not None:
    return torch.allclose(result, ref, rtol=0, atol=tol)
  else:
    return torch.equal(result, ref)
```

Problem A. Type conversion and precision

1. Let us create a matrix and check its storage.

Question A.1. Create matrix in CPU.

Create 3 matrices with different shape and types.

- a: dimension 3x4, float32. The value should range from 1 to 12.
- b: dimension dimension 32x64x4, float16. The value should range from 1 ... 8192.
- c: Slicing of b. Only keep the first 32 columns. Other dimensions are unchanged.

```
######################
# Your code goes here
######################
a = torch.reshape(torch.linspace(1, 12, 12), (3, 4))
b = torch.reshape(torch.linspace(1, 8192, 8192, dtype=torch.float16),(32,64,4))
c = b[:,:32,:]
```

Print the memory cost of each matrix above (in Bytes)

```
####################
# Your code goes here
####################

print(f'Tensor a consumes {a.element_size() * a.nelement()} bytes')
print(f'Tensor b consumes {b.element_size() * b.nelement()} bytes')
print(f'Tensor c consumes {c.element_size() * c.nelement()} bytes')
```

```
Tensor a consumes 48 bytes
Tensor b consumes 16384 bytes
Tensor c consumes 8192 bytes
```

Question A.2 Create matrix in GPU

Create 2 matrices in GPU.

- d: dimension 32x64x4, float16. The value should range from 1 ... 8192.
- e: Slicing of d. Only keep the first 32 columns. Other dimensions are unchanged.

```
####################
# Your code goes here
####################

d = torch.reshape(torch.linspace(1, 8192, 8192, dtype=torch.float16,
device='cuda'),(32,64,4))
e = d[:,:32,:]
```

Print the memory cost of d and e.

Also, try using actual memory cost of d and e together using torch.cuda.memory_allocated.

Note that this value is different from the same of mem cost of d and e itself. Why?

Hint: Sometimes, the calculate of GPU mem is not precise. Consider to use torch.cuda.empty_cache() to remove uncleaned mem.

```
####################
# Your code goes here
####################
d_mem_cost = d.element_size() * d.nelement()
e_mem_cost = e.element_size() * e.nelement()
mem_cost_using_memory_allocated = torch.cuda.memory_allocated(device=d.device)

print(f'Tensor d consumes {d_mem_cost} bytes')
```

```
print(f'Tensor e consumes {e_mem_cost} bytes')
print(f'Using memory_allocated, the actual memory consumption of d and e is '
      f'{mem_cost_using_memory_allocated}, this is different from sum d and '
      f'e consumptions, which is {d_mem_cost + e_mem_cost} bytes')
```

```
Tensor d consumes 16384 bytes
Tensor e consumes 8192 bytes
Using memory_allocated, the actual memory consumption of d and e is 16384, this is
different from sum d and e consumptions, which is 24576 bytes
```

```
print('This is because slicing operations in PyTorch, like '
'e = d[:,:32,:], create a tensor view, not a copy. A tensor object '
'consists of two parts: a metadata header (containing information like '
'shape, stride, and data type) and a pointer to an underlying torch.'
'Storage object, which is a contiguous block of memory holding the '
'actual numerical data.'
'When creating the view e, a new tensor object with its own metadata '
'is created, but it shares the same underlying Storage with the '
'original tensor d. Therefore, torch.cuda.memory_allocated() shows '
'minimal to no increase in memory usage, as no new memory for the '
'bulk data is allocated on the GPU.')
```

```
This is because slicing operations in PyTorch, like e = d[:,:32,:], create a
tensor view, not a copy. A tensor object consists of two parts: a metadata header
(containing information like shape, stride, and data type) and a pointer to an
underlying torch.Storage object, which is a contiguous block of memory holding the
actual numerical data.When creating the view e, a new tensor object with its own
metadata is created, but it shares the same underlying Storage with the original
tensor d. Therefore, torch.cuda.memory_allocated() shows minimal to no increase in
memory usage, as no new memory for the bulk data is allocated on the GPU.
```

Floating-point calculation accuracy.

In this question, let us check how accuracy affects the calculation results.

We will calculate two problems: summation and softmax, using float64, float32, and float16. Then, we will show that float16 has relatively low accuracy. And at the end, we will show a trick to improve accuracy.

Question A.3 Summation

Let us first create a random matrix using float16

```
torch.manual_seed(1234)
a = torch.rand(1000, dtype=torch.float16)
```

Calculate the ground truth summation using float64.

```
a_f64 = a.to(torch.float64)
a_gt = torch.sum(a_f64)

print(f'summation if the array using float64 is {a_gt}')
```

```
summation if the array using float64 is 513.48291015625
```

Next, let us try to calculate summation using float16 and float32.

Please:

- Convert tensor to float16/32
- Sum it using torch.sum

```
######################
# Your code goes here
######################

a_f32_sum = torch.sum(a.to(torch.float32))

print(f'summation if the array using float32 is {a_f32_sum}, '
      f'this is different from float64 result (a_gt)')

a_f16_sum = torch.sum(a.to(torch.float16))

print(f'summation if the array using float16 is {a_f16_sum}, '
      f'this is different from float64 result (a_gt)')
```

```
summation if the array using float32 is 513.48291015625, this is different from
float64 result (a_gt)
summation if the array using float16 is 513.5, this is different from float64
result (a_gt)
```

Let us try to improve accuracy.

Here you just use a forloop to explicitly enumerate each element (for i in range(1000) and sum them together. You can try to use float64 as the type for accumulator.

```
acc64 = torch.tensor(0.0, dtype=torch.float64)
for i in range(1000):
```

```
    acc64 += a[i].to(torch.float64)

print(f'forloop sum with float64 = {acc64}')
```

```
forloop sum with float64 = 513.48291015625
```

Question A.4 Softmax

Softmax of an array is very common in deep learning. Torch provide it a function for calculating it.

Check this link for more details of Softmax:

https://en.wikipedia.org/wiki/Softmax_function

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

First, let us calculate ground truth softmax result using float32.

```
torch.manual_seed(1234)
a = torch.rand(100, dtype=torch.float32) * 10
softmax_gt = torch.softmax(a, dim=0)

a_f16 = a.to(torch.float16)
```

Let us try to implement ourselves. Do not use torch.softmax or any equivalent function. Also, do not use forloop.

```
def cal_simple_softmax(x: torch.Tensor) -> torch.Tensor:
  ######################
  # Your code goes here
  # Please also add docstring
  ######################
  """
    Args:
      x (torch.Tensor): 1-D logits tensor.

    Returns:
      torch.Tensor: 1-D probabilities with the same dtype as x.
  """
  x_exp = torch.exp(x)
  denom = torch.sum(x_exp)
  return x_exp / denom
```

```
def cal_simple_softmax(x: torch.Tensor) -> torch.Tensor:
  x_exp = torch.exp(x)
  return x_exp / sum(x_exp)
```

Testing

Please do run this block before submission, but do not change it. If you change it, you will get no mark for this question.

```
softmax_myresult = cal_simple_softmax(a)
assert is_same_tensor(softmax_gt, softmax_myresult, 1e-6)
```

But if you try that on float16, it is not correct.

```
softmax_simple = cal_simple_softmax(a_f16)
err_softmax_simple = torch.norm(softmax_simple - softmax_gt)
print(f'The error of simple softmax using f16 is {err_softmax_simple}')
```

```
The error of simple softmax using f16 is 0.2395230531692505
```

The error above should be normally larger than 0.1 (but should be less than 0.4, check your implementation if it is larger than 0.4).

It is even worse when a has large values. Try the following block.

```
torch.manual_seed(1234)
a2 = (torch.rand(100, dtype=torch.float32) * 30).to(torch.float16)
softmax2_simple = cal_simple_softmax(a2)
print(f'Because of overflow or underflow, there are '
      f'{torch.count_nonzero(softmax2_simple.isnan())} elements'
       ' are inf')
```

```
Because of overflow or underflow, there are 64 elements are inf
```

To solve this problem, let us try a trick called LogSumExp.

Please read this artical to see how to LogSumExp and try that for softmax.

https://en.wikipedia.org/wiki/LogSumExp#log-sum-exp_trick_for_log-domain_calculations

Hint, the key is to subtract max(x) from x before exp.

Note, You CANNOT convert all vectors to float32 and calculate. You still need use float16 for major calculation, but think about how to improve the precision.

You can also try to use torch.logsumexp.

```
def cal_precise_softmax(x: torch.Tensor) -> torch.Tensor:
  ######################
  # Your code goes here
  # Please also add docstring
  ######################
  """
    Args:
        x (torch.Tensor): 1-D logits.

    Returns:
        torch.Tensor: Softmax probabilities with the same dtype as x.
  """
  x_max = torch.max(x)
  x_shifted = x - x_max
  lse = torch.logsumexp(x_shifted.to(torch.float32), dim=0)
  return torch.exp(x_shifted - lse.to(x.dtype))
```

Test. **Do not change the block below.**

```
torch.manual_seed(1234)
a1 = torch.rand(100, dtype=torch.float32) * 10
a2 = torch.rand(1000, dtype=torch.float32) * 10
a3 = torch.rand(100, dtype=torch.float32) * 30

for a_f32 in [a1, a2, a3]:
  a_f16 = a_f32.to(torch.float16)
  softmax_gt = torch.softmax(a_f16, dim=0)
  softmax_precise = cal_precise_softmax(a_f16)
  assert is_same_tensor(softmax_precise, softmax_gt, 1e-3)

print('Test successs')
```

```
Test successs
```

Problem B. Deep learning training

Step 1. Load the training data

First, let us check what is the mean and variance of FashionMNIST.

Why we need to know this information?

Because majority of network prefers a normalized input. If the input is not normalized, the training may converge slower, or even failed.

Step 1.1 Estimate the mean and variance

Let us use the first few samples of the training set.

First, let us download the dataset (no question)

```
train_dataset = torchvision.datasets.FashionMNIST(
    root='./data', # The dataset will be downloaded from the webserver to the
local machine, this is where it stores
    train=True, # We will download the training set
    download=True,
)
train_data = train_dataset.train_data.to(torch.float32) / 255 # Convert it to
float32
train_labels = train_dataset.train_labels
```

```
100%|██████████| 26.4M/26.4M [00:01<00:00, 17.1MB/s]
100%|██████████| 29.5k/29.5k [00:00<00:00, 270kB/s]
100%|██████████| 4.42M/4.42M [00:00<00:00, 5.01MB/s]
100%|██████████| 5.15k/5.15k [00:00<00:00, 9.64MB/s]
/usr/local/lib/python3.12/dist-packages/torchvision/datasets/mnist.py:76:
UserWarning: train_data has been renamed data
  warnings.warn("train_data has been renamed data")
/usr/local/lib/python3.12/dist-packages/torchvision/datasets/mnist.py:66:
UserWarning: train_labels has been renamed targets
  warnings.warn("train_labels has been renamed targets")
```

Question B.1. Next, let us check how the dataset looks

```
print('train_dataset.train_data is a',
   f'{type(train_dataset.train_data)}. Its dimension is',
   f'{train_dataset.train_data.shape}, and its type is
{train_dataset.train_data.dtype}')
print()
print(f'The first dimension is number of samples, which is
{train_dataset.train_data.shape[0]}')
print()
print('The 2nd and 3rd dimensions are height and width, which means the image
resolution is '
   f'{train_dataset.train_data.shape[1]}x{train_dataset.train_data.shape[2]}')
```

```
train_dataset.train_data is a <class 'torch.Tensor'>. Its dimension is
torch.Size([60000, 28, 28]), and its type is torch.uint8

The first dimension is number of samples, which is 60000

The 2nd and 3rd dimensions are height and width, which means the image resolution
is 28x28
```

The dataset also contains labels, which is what network should output

```
# Define class names
class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

```
print('train_dataset.train_labels is',
   f'{type(train_labels)}, and its dimension is',
   f'{train_labels.shape}, and its type is {train_labels.dtype}')
print()
print('The label is only one dimension, and the first dimension is the number
samples, which is the same as train_data')
print()
```

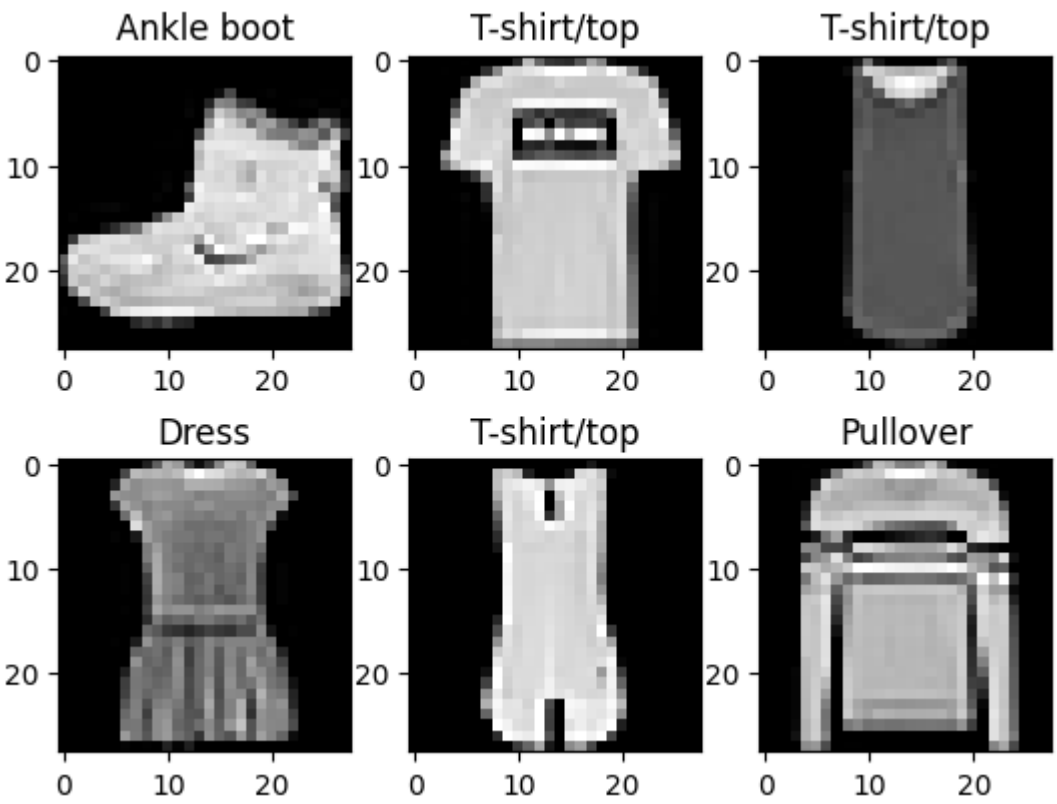
```
train_dataset.train_labels is <class 'torch.Tensor'>, and its dimension is
torch.Size([60000]), and its type is torch.int64

The label is only one dimension, and the first dimension is the number samples,
which is the same as train_data
```

```
####################
# Your code goes here
####################

sample_labels = train_labels[:10]
sample_class_names = [class_names[int(i)] for i in sample_labels]
print(f'Here are the labels for the first 10 samples: {sample_labels}')
print('Each number is an index to the class names')
print(f'So the actual class names of these 10 samples should be:
{sample_class_names}')
```

```
Here are the labels for the first 10 samples: tensor([9, 0, 0, 3, 0, 2, 7, 2, 5,
5])
Each number is an index to the class names
So the actual class names of these 10 samples should be: ['Ankle boot', 'T-
shirt/top', 'T-shirt/top', 'Dress', 'T-shirt/top', 'Pullover', 'Sneaker',
'Pullover', 'Sandal', 'Sandal']
```

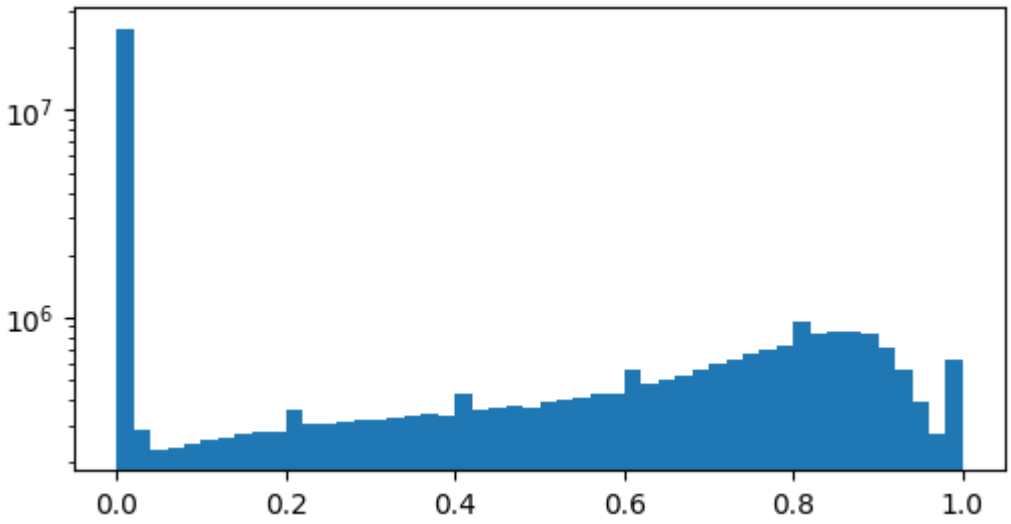
Let us visualize the images in the dataset.

```
###################
# Your code goes here
###################

for i in range(6):
  plt.subplot(2, 3, i+1)
  im = train_data[i] # Load the i-th training image
  label = class_names[int(train_labels[i])] # Load the i-th training label
  plt.imshow(im, cmap='gray', vmin=0.0, vmax=1.0)
  plt.title(label)
```



And we can also show the histogram of these images

```
plt.figure(figsize=(6, 3))
result = plt.hist(train_data.flatten(), bins=50)
plt.yscale('log')
```



Question B.2 Finally, we need to get the mean and standard deviation (std).

Hint: Torch has built function to get mean and standard deviation. The mean should be around 0.28, and std should be around 0.35.

```
####################
# Your code goes here
####################

mean_train = train_data.mean().item()
std_train = train_data.std().item()
print(f'mean = {mean_train}, std = {std_train}')
```

```
mean = 0.2860405743122101, std = 0.3530242443084717
```

Step 1.2. Define the data transformation (No questions)

```
# Define transformations for the training and test sets

transform_train = transforms.Compose([
    # Convert input to a float tensor. It will automatically converts uint8
    # image to float32 image.
    transforms.ToTensor(),
    # normalize it to zero-mea, unit variance.
    transforms.Normalize((mean_train,), (std_train,))
])

# The transformation applied to the testing is the same as training
transform_test = transforms.Compose([
    transforms.ToTensor(),
```

```
    transforms.Normalize((mean_train,), (std_train,))
])
```

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

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

# Create data loaders
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

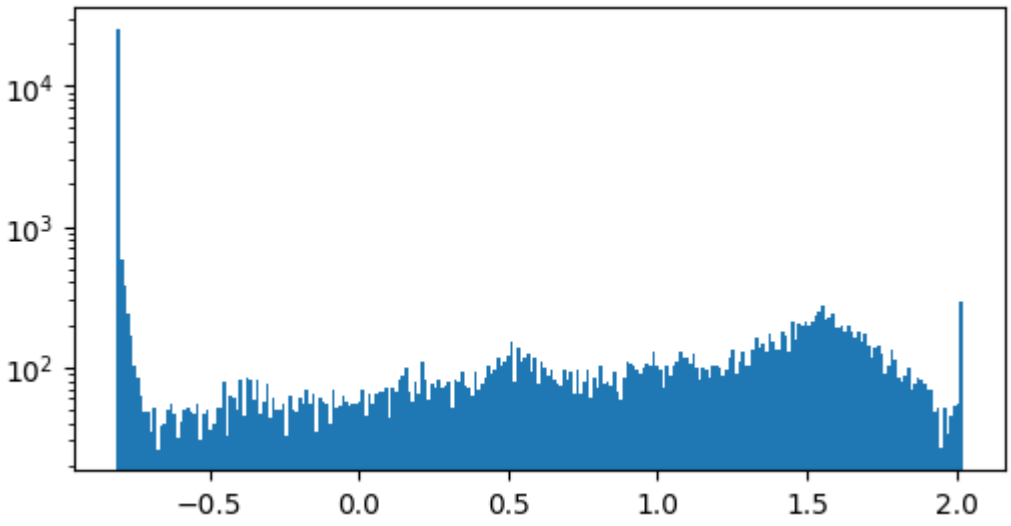
Let us visualize the loaded data again.

Note that:

- The image after loading is a floating image
- The image is normalized, where the value is distributed around 0

```
for batch_image, batch_label in train_loader:
  print(f'In each batch, the batch_image dimension is {batch_image.shape}')
  print(f'In each batch, the batch_label dimension is {batch_label.shape}')
  plt.figure(figsize=(6, 3))
  result = plt.hist(batch_image.flatten(), bins=256)
  plt.yscale('log')
  break
```

```
In each batch, the batch_image dimension is torch.Size([64, 1, 28, 28])
In each batch, the batch_label dimension is torch.Size([64])
```



Step 2. Define other training options (no questions)

For the first training project, CPU is enough.

Still, if you like to (and has enough quota), you can always run it with GPU.

```
# Check if CUDA is available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cpu
```

Also, we need to define the loss function.

```
loss_fn = nn.CrossEntropyLoss()
```

Step 3. Define the network structure

Question B.3

Let us create the first model, which is a MLP. Please use:

- 2 linear layers
- The dimension of intermediate layer is 32
- The last layer should be a linear

```
class MLPNetwork(nn.Module):
  def __init__(self):
    super().__init__()
    # Flatten the 28x28 image into a 784-element vector
```

```
    self.flatten = nn.Flatten()
    # Define a MLP model here
    self.mlp = nn.Sequential(
      ####################
      # Your code goes here
      ####################
      nn.Linear(784, 32),
      nn.ReLU(inplace=True),
      nn.Linear(32, 10)
    )
  def forward(self, x):
    # Pass the input through the layers
    x = self.flatten(x)
    logits = self.mlp(x)
    return logits

model = MLPNetwork().to(device)
print('==== This is the model we created ====')
print(model)
```

```
==== This is the model we created ====
MLPNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (mlp): Sequential(
    (0): Linear(in_features=784, out_features=32, bias=True)
    (1): ReLU(inplace=True)
    (2): Linear(in_features=32, out_features=10, bias=True)
  )
)
```

Calculate the number of trainable parameters.

For example, for a nn.Linear(10, 20), the weight matrix has 10x20=200 parameters, and the bias has 20 parameters.

Hint: you can `model.parameters()` to iterate all parameters.

```
def calc_model_params(model: nn.Module) -> int:
  ######################
  # Your code goes here
  # Add docstring
  ######################
  """
    Return the total number of trainable parameters in a PyTorch model.
  """
  return sum(p.numel() for p in model.parameters())

print(f'There are {calc_model_params(model)} parameters in this model')
```

```
There are 25450 parameters in this model
```

Step 4: Training and Evaluation Loops (Although no questions, but make it runs correctly)

First, let us define a training loop iterates over all the batches of the training dataset.

```python
def train_one_epoch(
    dataloader: torch.utils.data.DataLoader,
    model: nn.Module,
    loss_fn: nn.Module,
    optimizer: optim.Optimizer,
    loss_print_iter: int=100
  ) -> None:
  num_train_samples = len(dataloader.dataset)

  # Set the model to the training mod
  model.train()

  for batch_index, (image, label) in enumerate(dataloader):
    image, label = image.to(device), label.to(device)

    # Compute prediction error
    pred = model(image)
    loss = loss_fn(pred, label)

    # Backpropagation

    # Computes gradients of the loss with respect to all parameters
    # in the model that require gradients, storing them in the .grad
    # attribute of each parameter.
    loss.backward()

    # Updates all the model parameters using the gradients computed in the
    # previous step, moving parameters in the direction that reduces the
    # loss.
    optimizer.step()

    # Clear gradient for next iterations.
    optimizer.zero_grad()

    if batch_index % loss_print_iter == 0:
      loss, trained_samples = loss.item(), (batch_index + 1) * image.shape[0]
      print(f'loss: {loss:>7f} '
            f'[{trained_samples:>5d}/{num_train_samples:>5d}] ')

  return None
```

Also, defines an evaluation loop checks the model's performance on the test dataset.

```
def test_all_samples(
    dataloader: torch.utils.data.DataLoader,
    model: nn.Module,
    loss_fn: nn.Module
) -> None:
  model.eval()  # Set the model to evaluation mode
  num_testing_samples = len(dataloader.dataset)
  num_batches = len(dataloader)
  test_loss, correct = 0, 0

  # Disable gradient calculation for inference
  with torch.no_grad():
    for image, label in dataloader:
      image, label = image.to(device), label.to(device)
      pred = model(image)
      test_loss += loss_fn(pred, label).item()
      correct += (pred.argmax(1) == label).type(torch.float).sum().item()

  test_loss /= num_batches
  correct /= num_testing_samples
  print(f'Test Error: \n Accuracy: {(100*correct):>0.1f}%, '
        f'Avg loss: {test_loss:>8f} \n')
```

At last, let us run the training. In each epoch:

- We loop all training samples and update the network `train_one_epoch`

```
model = MLPNetwork().to(device)
epochs = 3   # Number of training epochs
optimizer = torch.optim.SGD(
  model.parameters(),    # All trainable parameters.
  lr=1e-4                # Learning rate
)

for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  # Run one training epoch
  train_one_epoch(train_loader, model, loss_fn, optimizer)
  # After each epoch, run testing on all samples
  test_all_samples(test_loader, model, loss_fn)

print("Training done!")
```

```
Epoch 1
-------------------------------
loss: 2.391819 [   64/60000]
loss: 2.379251 [ 6464/60000]
loss: 2.307301 [12864/60000]
```

```
loss: 2.295874 [19264/60000]
loss: 2.241051 [25664/60000]
loss: 2.249795 [32064/60000]
loss: 2.196032 [38464/60000]
loss: 2.147539 [44864/60000]
loss: 2.160627 [51264/60000]
loss: 2.095410 [57664/60000]
Test Error:
 Accuracy: 33.0%, Avg loss: 2.093733

Epoch 2
-------------------------------
loss: 2.067951 [   64/60000]
loss: 2.047303 [ 6464/60000]
loss: 2.078721 [12864/60000]
loss: 2.030339 [19264/60000]
loss: 1.987212 [25664/60000]
loss: 1.972972 [32064/60000]
loss: 1.915595 [38464/60000]
loss: 1.935348 [44864/60000]
loss: 1.915719 [51264/60000]
loss: 1.875273 [57664/60000]
Test Error:
 Accuracy: 48.7%, Avg loss: 1.879135

Epoch 3
-------------------------------
loss: 1.866139 [   64/60000]
loss: 1.818330 [ 6464/60000]
loss: 1.775717 [12864/60000]
loss: 1.771365 [19264/60000]
loss: 1.881365 [25664/60000]
loss: 1.778277 [32064/60000]
loss: 1.692431 [38464/60000]
loss: 1.713676 [44864/60000]
loss: 1.747060 [51264/60000]
loss: 1.646270 [57664/60000]
Test Error:
 Accuracy: 54.3%, Avg loss: 1.692861

Training done!
```

At last, let us test this model on real images.

```
model.eval()
with torch.no_grad():
  for image, label in test_loader:
    print(image.shape)
    image = image.to(device)
```

```
    model_output = model(image[:6, ...])
    predicted_label = [class_names[x] for x in model_output.argmax(1)]
    actual_label = [class_names[x] for x in label[:6]]
    # Display the first image and its prediction
    plt.figure(figsize=(12, 8))
    for i in range(6):
      plt.subplot(2, 3, i+1);
      plt.imshow(image[i, 0, ...], cmap='gray')
      plt.title(f'Predicted: {predicted_label[i]}, '
                f'Actual: {actual_label[i]}', fontsize=8)
      plt.axis('off')
    plt.show()
    break;
```

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

Question B.4. Visualize train / test loss curve

Write another version of training pipeline, similar to step 4, but record:

- The traininng loss and accuracy at each iterations (training one batch is one iteration): `train_acc` and `train_loss`
- Calculate the exponential moving average (EMA) of them: `train_acc_ema` and `train_loss_ema`. This is very useful in visualization, as the original train_acc is very noisy (see the plot question below)

Check this for details of EMA:

https://en.wikipedia.org/wiki/Exponential_smoothing

$$s_0 = x_0$$
$$s_t = \alpha x_t + (1-\alpha)s_{t-1}, \quad t > 0$$

```
####################
# Your code goes here
####################
def eval_on_test(model, test_loader, loss_fn, device):
    model.eval()
    correct, total = 0, 0
    loss_sum, n_batches = 0.0, 0
    with torch.no_grad():
        for x, y in test_loader:
            x, y = x.to(device), y.to(device)
            logits = model(x)
            loss_sum += loss_fn(logits, y).item()
            n_batches += 1
            pred = logits.argmax(1)
            correct += (pred == y).sum().item()
            total += y.numel()
    acc = correct / total
    avg_loss = loss_sum / n_batches
    return acc, avg_loss

def train_with_curves(model, train_loader, test_loader, loss_fn, optimizer,
device,
                      epochs=3, ema_alpha=0.2):
    train_acc, train_loss = [], []
    train_acc_ema, train_loss_ema = [], []
    test_acc, test_loss, test_iters = [], [], []

    ema_acc = None
    ema_loss = None
    global_iter = 0

    for ep in range(epochs):
        model.train()
        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)

            logits = model(images)
            loss = loss_fn(logits, labels)

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

            acc = (logits.argmax(1) == labels).float().mean().item()
            l = loss.item()
```

```
            train_acc.append(acc)
            train_loss.append(l)

            # EMA: s_t = α x_t + (1-α) s_{t-1}, s_0 = x_0
            ema_acc  = acc if ema_acc  is None else ema_alpha * acc + (1 -
ema_alpha) * ema_acc
            ema_loss = l   if ema_loss is None else ema_alpha * l   + (1 -
ema_alpha) * ema_loss
            train_acc_ema.append(ema_acc)
            train_loss_ema.append(ema_loss)

            global_iter += 1

        acc_val, loss_val = eval_on_test(model, test_loader, loss_fn, device)
        test_acc.append(acc_val)
        test_loss.append(loss_val)
        test_iters.append(global_iter - 1)

    return (train_acc, train_loss, train_acc_ema, train_loss_ema,
            test_acc, test_loss, test_iters)

train_acc, train_loss, train_acc_ema, train_loss_ema, test_acc, test_loss,
test_iters = \
    train_with_curves(model, train_loader, test_loader, loss_fn, optimizer,
device,
                      epochs=3, ema_alpha=0.2)
```

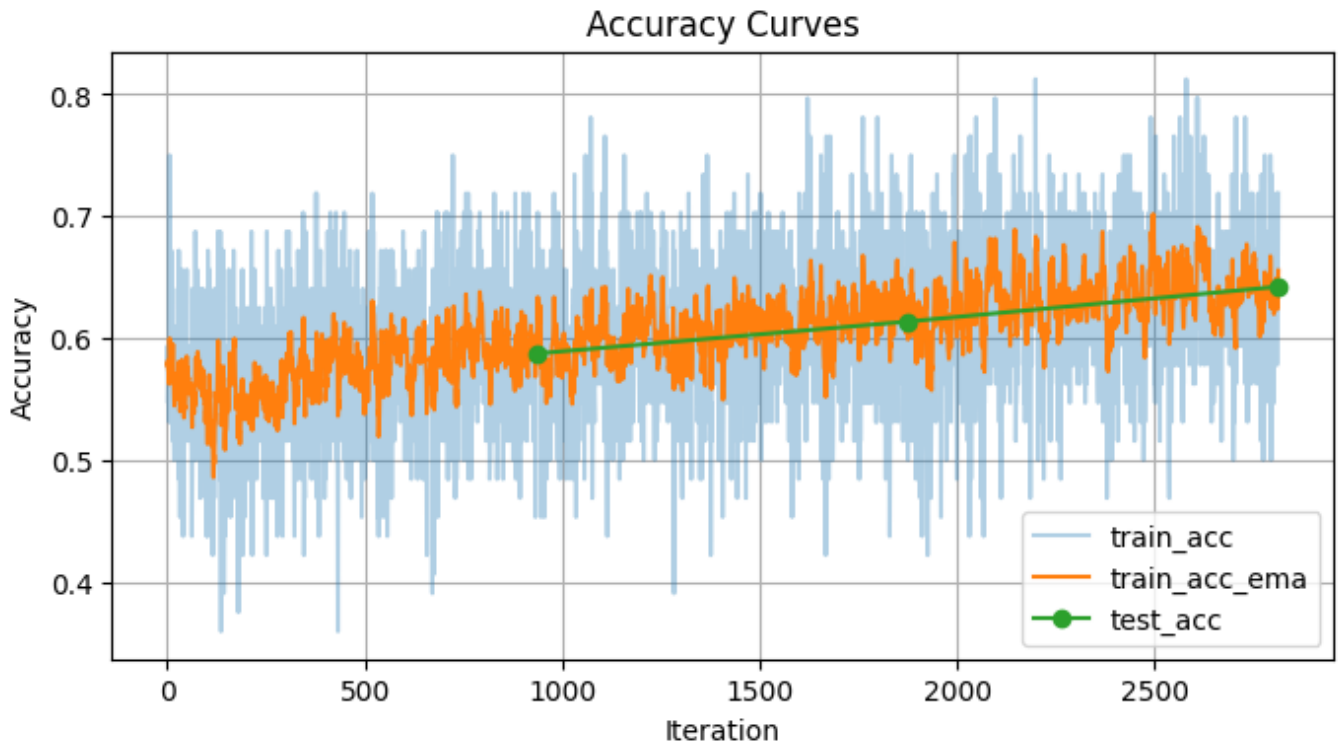
In the next, plot the following 3 curves in the single plot:

- `train_acc`
- `train_acc_ema`
- `test_acc` Note that `train_acc` is very noisy. Tune the parameter of ema to make the `train_acc_ema`

x-axis should be iteration. Note that `test_acc` has much less samples compared with `train_acc`.

Hint: When plot a curve, you can use `plt.plot(x, y)`

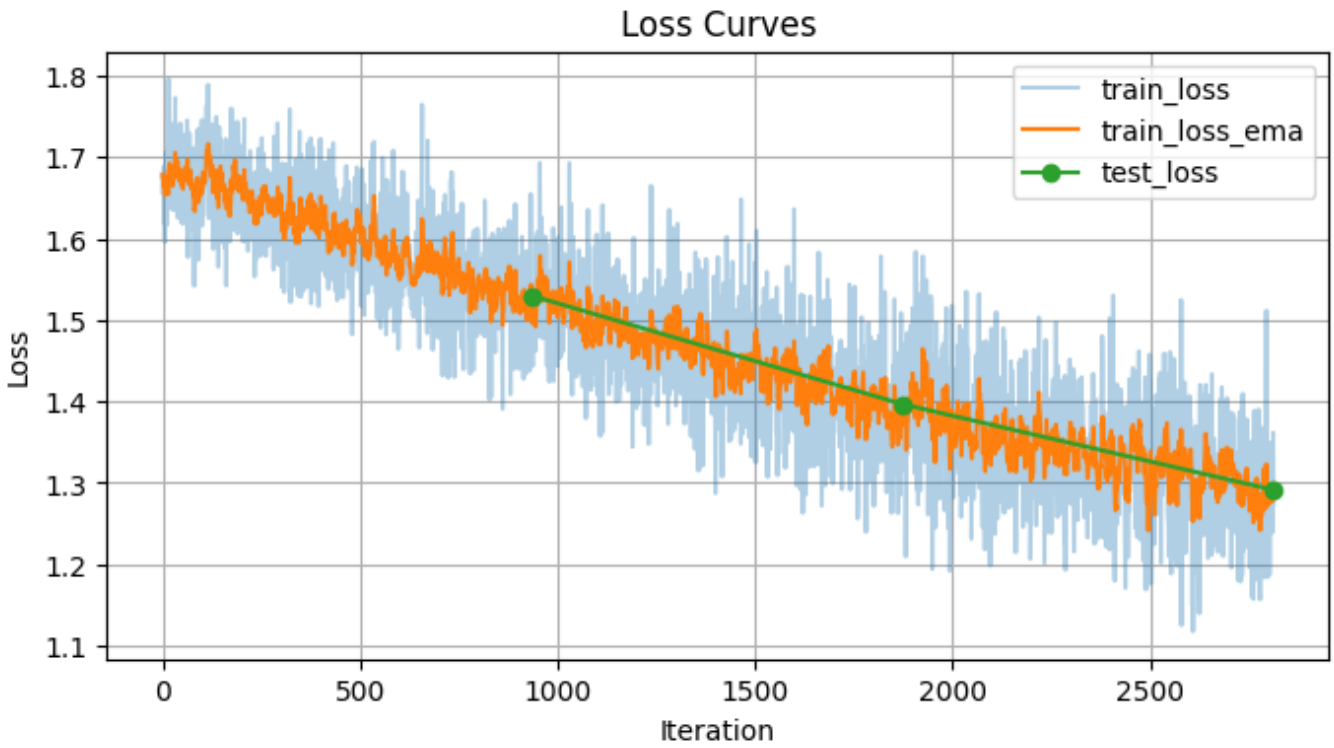
```
######################
# Your code goes here
######################
iters_train = np.arange(len(train_acc))

plt.figure(figsize=(8, 4))
plt.plot(iters_train, train_acc,     label='train_acc', alpha=0.35)
plt.plot(iters_train, train_acc_ema, label='train_acc_ema')
plt.plot(test_iters,  test_acc,      label='test_acc', marker='o')
plt.xlabel('Iteration')
plt.ylabel('Accuracy')
plt.title('Accuracy Curves')
plt.legend()
plt.grid(True)
plt.show()
```



Similarly, plot the following 3 curves :

- `train_loss`
- `train_loss_ema`
- `test_loss`

```
######################
# Your code goes here
######################
iters_train = np.arange(len(train_loss))

plt.figure(figsize=(8, 4))
plt.plot(iters_train, train_loss,     label='train_loss', alpha=0.35)
plt.plot(iters_train, train_loss_ema, label='train_loss_ema')
plt.plot(test_iters,  test_loss,      label='test_loss', marker='o')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Loss Curves')
plt.legend()
plt.grid(True)
plt.show()
```



Question B.5. Improve the model accuracy.

This model is definitely not the best. Can you try to make some changes to improve the accuracy. You could try:

- Add more layers
- Change intermediate channel number
- Change epochs
- Change learning rate
- ...

With some tunings, you may be able to achieve about 87% accuracy (with more careful design, it could reach 88% or higher).

Still, if you cannot achieve 87%, you can stead record 2-3 models you have tried (including the print output of each each training). If you achieved 87%, just keep the model that achieves it.

You shall achieve a reasonable result with less than 7 min training on CPU.

```
class MLPNetwork2(nn.Module):
  ######################
  # Your code goes here
  ######################
    """
      The hidden layer uses ReLU
      After each layer, BatchNorm1d stable training is connected
      Dropout boosts generalization
    """
    def __init__(self, in_dim=784, num_classes=10, p_dropout=0.2):
        super().__init__()
```

```
        self.flatten = nn.Flatten()
        self.net = nn.Sequential(
            nn.Linear(in_dim, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(inplace=True),
            nn.Dropout(p_dropout),

            nn.Linear(256, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(inplace=True),
            nn.Dropout(p_dropout),

            nn.Linear(128, 64),
            nn.BatchNorm1d(64),
            nn.ReLU(inplace=True),
            nn.Dropout(p_dropout),

            nn.Linear(64, num_classes)
        )

        self.apply(self._init_weights)

    @staticmethod
    def _init_weights(m):
        if isinstance(m, nn.Linear):
            nn.init.kaiming_normal_(m.weight, nonlinearity='relu')
            if m.bias is not None:
                nn.init.zeros_(m.bias)
        elif isinstance(m, nn.BatchNorm1d):
            nn.init.ones_(m.weight)
            nn.init.zeros_(m.bias)

    def forward(self, x):
        x = self.flatten(x)
        logits = self.net(x)
        return logits

model_v2 = MLPNetwork2()
```

And rerun your training and test below.

```
######################
# Your code goes here
######################
model_v2 = MLPNetwork2().to(device)

optimizer_v2 = optim.Adam(model_v2.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

epochs = 6
print('==== Training MLPNetwork2 ====')
for t in range(epochs):
```

```
    print(f'Epoch {t+1}/{epochs}')
    train_one_epoch(train_loader, model_v2, loss_fn, optimizer_v2,
loss_print_iter=100)
    test_all_samples(test_loader, model_v2, loss_fn)

print('Done. Final evaluation on test set:')
test_all_samples(test_loader, model_v2, loss_fn)
```

```
==== Training MLPNetwork2 ====
Epoch 1/6
loss: 2.777508 [   64/60000]
loss: 0.619867 [ 6464/60000]
loss: 0.734239 [12864/60000]
loss: 0.454678 [19264/60000]
loss: 0.457658 [25664/60000]
loss: 0.431952 [32064/60000]
loss: 0.441264 [38464/60000]
loss: 0.392109 [44864/60000]
loss: 0.399027 [51264/60000]
loss: 0.480801 [57664/60000]
Test Error:
 Accuracy: 85.7%, Avg loss: 0.396832

Epoch 2/6
loss: 0.292000 [   64/60000]
loss: 0.382639 [ 6464/60000]
loss: 0.259327 [12864/60000]
loss: 0.521330 [19264/60000]
loss: 0.251380 [25664/60000]
loss: 0.385965 [32064/60000]
loss: 0.448488 [38464/60000]
loss: 0.296443 [44864/60000]
loss: 0.493529 [51264/60000]
loss: 0.391150 [57664/60000]
Test Error:
 Accuracy: 86.4%, Avg loss: 0.374405

Epoch 3/6
loss: 0.223269 [   64/60000]
loss: 0.223458 [ 6464/60000]
loss: 0.457729 [12864/60000]
loss: 0.318554 [19264/60000]
loss: 0.564702 [25664/60000]
loss: 0.624877 [32064/60000]
loss: 0.238512 [38464/60000]
loss: 0.375950 [44864/60000]
loss: 0.202133 [51264/60000]
loss: 0.427017 [57664/60000]
Test Error:
```

```
 Accuracy: 86.8%, Avg loss: 0.357779

Epoch 4/6
loss: 0.340582 [   64/60000]
loss: 0.436064 [ 6464/60000]
loss: 0.378212 [12864/60000]
loss: 0.320826 [19264/60000]
loss: 0.168757 [25664/60000]
loss: 0.318055 [32064/60000]
loss: 0.443034 [38464/60000]
loss: 0.297918 [44864/60000]
loss: 0.353749 [51264/60000]
loss: 0.310714 [57664/60000]
Test Error:
 Accuracy: 87.7%, Avg loss: 0.338344

Epoch 5/6
loss: 0.262848 [   64/60000]
loss: 0.315444 [ 6464/60000]
loss: 0.350869 [12864/60000]
loss: 0.348496 [19264/60000]
loss: 0.300961 [25664/60000]
loss: 0.279562 [32064/60000]
loss: 0.467133 [38464/60000]
loss: 0.387992 [44864/60000]
loss: 0.386008 [51264/60000]
loss: 0.336491 [57664/60000]
Test Error:
 Accuracy: 88.2%, Avg loss: 0.327223

Epoch 6/6
loss: 0.453888 [   64/60000]
loss: 0.262533 [ 6464/60000]
loss: 0.519644 [12864/60000]
loss: 0.265709 [19264/60000]
loss: 0.257184 [25664/60000]
loss: 0.343327 [32064/60000]
loss: 0.367825 [38464/60000]
loss: 0.383714 [44864/60000]
loss: 0.285730 [51264/60000]
loss: 0.416593 [57664/60000]
Test Error:
 Accuracy: 87.8%, Avg loss: 0.329152

Done. Final evaluation on test set:
Test Error:
 Accuracy: 87.8%, Avg loss: 0.329152
```

After training, run the test below. Do not change it.

```
test_all_samples(test_loader, model_v2, loss_fn)
```

```
Test Error:
 Accuracy: 87.8%, Avg loss: 0.329152
```

Question B.6. Bonus. CNN.

Although we have not talk about CNN yet, but you can try to train a multi-layer CNN.

It is not easy to achieve much higher accuracy using CNN on FashionMNIST (compared to B5), but you shall reduce parameters.

Hint:

- You can try to use nn.Conv2d to create conv layer and nn.MaxPool2d to create a pooling layer.

```
class MLPNetwork3(nn.Module):
  def __init__(self):
    super().__init__()
    # Define a sequential model with dense layers
    self.mlp = nn.Sequential(
      nn.Conv2d(1, 8, 3),
      nn.ReLU(),
      nn.MaxPool2d((2, 2), (2, 2)),
      nn.Conv2d(8, 8, 3, stride=1),
      nn.ReLU(),
      nn.MaxPool2d((2, 2), (2, 2)),
      nn.Flatten(),
      nn.Linear(5 * 5 * 8, 10),
    )
  def forward(self, x):
    # Pass the input through the layers
    logits = self.mlp(x)
    return logits



model_v3 = MLPNetwork3()
```

```
print(f'There are {calc_model_params(model_v3)} parameters in this model')
```

```
There are 2674 parameters in this model
```

And rerun your training.

```
####################
# Your code goes here
####################
model_v3 = MLPNetwork3().to(device)

loss_fn = nn.CrossEntropyLoss()
optimizer_v3 = optim.Adam(model_v3.parameters(), lr=1e-2)

epochs = 6
print('==== Training model_v3 (tiny CNN) ====')
for ep in range(1, epochs + 1):
    print(f'Epoch {ep}/{epochs}')
    train_one_epoch(train_loader, model_v3, loss_fn, optimizer_v3,
loss_print_iter=100)
    test_all_samples(test_loader, model_v3, loss_fn)
```

```
==== Training model_v3 (tiny CNN) ====
Epoch 1/6
loss: 2.296422 [   64/60000]
loss: 0.468836 [ 6464/60000]
loss: 0.589826 [12864/60000]
loss: 0.394269 [19264/60000]
loss: 0.461370 [25664/60000]
loss: 0.294580 [32064/60000]
loss: 0.386580 [38464/60000]
loss: 0.464018 [44864/60000]
loss: 0.393796 [51264/60000]
loss: 0.432493 [57664/60000]
Test Error:
 Accuracy: 84.8%, Avg loss: 0.417140

Epoch 2/6
loss: 0.393371 [   64/60000]
loss: 0.413733 [ 6464/60000]
loss: 0.253810 [12864/60000]
loss: 0.574092 [19264/60000]
loss: 0.319418 [25664/60000]
loss: 0.485133 [32064/60000]
loss: 0.399259 [38464/60000]
loss: 0.276067 [44864/60000]
loss: 0.247233 [51264/60000]
loss: 0.401385 [57664/60000]
Test Error:
 Accuracy: 83.7%, Avg loss: 0.431089

Epoch 3/6
```

```
loss: 0.502833 [   64/60000]
loss: 0.438376 [ 6464/60000]
loss: 0.468687 [12864/60000]
loss: 0.322762 [19264/60000]
loss: 0.312897 [25664/60000]
loss: 0.508363 [32064/60000]
loss: 0.391909 [38464/60000]
loss: 0.318909 [44864/60000]
loss: 0.388322 [51264/60000]
loss: 0.316501 [57664/60000]
Test Error:
 Accuracy: 84.4%, Avg loss: 0.417699

Epoch 4/6
loss: 0.617799 [   64/60000]
loss: 0.474196 [ 6464/60000]
loss: 0.210379 [12864/60000]
loss: 0.461336 [19264/60000]
loss: 0.499912 [25664/60000]
loss: 0.385403 [32064/60000]
loss: 0.397351 [38464/60000]
loss: 0.326066 [44864/60000]
loss: 0.170937 [51264/60000]
loss: 0.433694 [57664/60000]
Test Error:
 Accuracy: 84.9%, Avg loss: 0.414011

Epoch 5/6
loss: 0.274021 [   64/60000]
loss: 0.140456 [ 6464/60000]
loss: 0.630454 [12864/60000]
loss: 0.524391 [19264/60000]
loss: 0.454967 [25664/60000]
loss: 0.375920 [32064/60000]
loss: 0.361971 [38464/60000]
loss: 0.549707 [44864/60000]
loss: 0.636604 [51264/60000]
loss: 0.257208 [57664/60000]
Test Error:
 Accuracy: 85.6%, Avg loss: 0.400000

Epoch 6/6
loss: 0.284339 [   64/60000]
loss: 0.213502 [ 6464/60000]
loss: 0.585063 [12864/60000]
loss: 0.335552 [19264/60000]
loss: 0.204357 [25664/60000]
loss: 0.351918 [32064/60000]
loss: 0.430657 [38464/60000]
loss: 0.433020 [44864/60000]
loss: 0.397622 [51264/60000]
```

```
loss: 0.478108 [57664/60000]
Test Error:
 Accuracy: 85.7%, Avg loss: 0.385911
```

After training, run the test below. Do not change it.

```
test_all_samples(test_loader, model_v3, loss_fn)
```

```
Test Error:
 Accuracy: 85.7%, Avg loss: 0.385911
```

After training, print the network parameters, and run the test below.

```
print(f'There are {calc_model_params(model_v3)} parameters in this model')

test_all_samples(test_loader, model_v3, loss_fn)
```

```
There are 2674 parameters in this model
Test Error:
 Accuracy: 85.7%, Avg loss: 0.385911
```