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

AIMS 5702 Assignment 3

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

Accelerator:. Although this training can be runnable on CPU, GPU will be much faster, so we would suggest to use T4.

Deadline of submission: 23:59, October. 20th (Monday), 2025

First, let us import all necessary libraries for this one

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal
import math
import time
from typing import List, Union, Dict
```

Problem A. Back propagation

In this problem, let us try to implement the back propagation by ourselves.

Requirement:

- You cannot reuse the any pytorch backward function
- You cannot use any auto-grad library
- You can use pytorch forward function, or numpy utility functions, unless we explicitly mention that certain functions cannot be used.

Utility functions for verifying the gradient results

First, let us create a base class, which you will need to create your inheritted class later. Do not change it.

```
#@title Base class for all your own implementation.

class CustomizedLayer(object):
```

```
    def set_params(self, param_dict: Dict[str, torch.Tensor]) -> None:
        """
        Set the parameters of the layer, like weight and bias of linear layer.

        Args:
            param_dict: A dictionary of parameters.
        """
        raise NotImplementedError

    def forward(self, inputs: List[np.ndarray]) -> np.ndarray:
        """
        Forward pass of the layer.

        Args:
            inputs: A list of input tensors.

        Returns:
            The output of the layer.
        """
        raise NotImplementedError

    def input_gradients(
            self,
            inputs: List[np.ndarray],
            outputs: np.ndarray,
            output_gradient: np.ndarray
    ) -> Union[np.ndarray, List[np.ndarray]]:
        """
        Calculate the input gradients of the layer.
        Args:
            inputs: A list of input tensors.
            outputs: The output of the layer.
            output_gradient: The gradient of the output.
        """
        raise NotImplementedError

    def param_gradients(
            self,
            inputs: List[np.ndarray],
            outputs: np.ndarray,
            output_gradient: np.ndarray
    ) -> Dict[str, np.ndarray]:
        """
        Calculate the parameter gradients of the layer.
        Args:
            inputs: A list of input tensors.
            outputs: The output of the layer.
            output_gradient: The gradient of the output.
        """
        raise NotImplementedError
```

Then, define a few utility functions for testing.

```
#@title Utility functions

def verify_output(
    actual: Union[np.ndarray, List[np.ndarray]],
    reference: Union[np.ndarray, List[np.ndarray]],
    name: Union[str, List[str]],
    atol: float = 1e-6,
    rtol: float = 1e-6
) -> None:
  """
  Verify if the actual output is close to the reference output.

  Args:
    actual: The actual output.
    reference: The reference output.
    atol: The absolute tolerance.
    rtol: The relative tolerance.
  """
  if isinstance(actual, np.ndarray):
    close = np.allclose(actual, reference, atol=atol, rtol=rtol)
    if not close:
      print(f"{name} is not close to the reference output.")
    else:
      print(f"{name} is close to the reference output.")
  elif isinstance(actual, List):
    close = all(verify_output(a, b, name[i], atol=atol, rtol=rtol)
                for i, (a, b) in enumerate(zip(actual, reference)))
  return close

def compare_with_actual_layer(
    official_layer: nn.modules,
    customized_layer: CustomizedLayer,
    input_np: List[np.ndarray],
    output_grad_np: np.ndarray,
    parameter_names: List[str],
    param_np: List[np.ndarray],
    a_tol: float = 1e-6,
    r_tol: float = 1e-6,
) -> bool:
  """
  Compare forward/backward of your customized layer with the official layer.

  Args:
    official_layer: The official layer.
    customized_layer: The customized layer.
    input_np: The input numpy array.
    output_grad_np: The output gradient numpy array.
    parameter_names: The names of the parameters.
    param_np: The numpy array of the parameters.
    a_tol: The absolute tolerance.
    r_tol: The relative tolerance.
```

```
    Returns:
      True if the output of the customized layer is close to the output of
      the official layer, False otherwise.
    """
    success = True
    input_torch = [
        torch.tensor(x, requires_grad=True) for x in input_np
    ]
    output_grad_torch = torch.tensor(output_grad_np, requires_grad=True)
    for i, name in enumerate(parameter_names):
      setattr(official_layer, name,
              nn.Parameter(torch.tensor(param_np[i], requires_grad=True)))

    # Run forward and backward on official layers.
    output_official_torch = official_layer(*input_torch)
    output_official_torch.backward(output_grad_torch)
    output_official_np = output_official_torch.detach().numpy()

    # Check output
    output_customized_np = customized_layer.forward(input_np)
    success = verify_output(output_customized_np, output_official_np,
                            "Output", atol=a_tol, rtol=r_tol)
    if not success:
      return success

    # Check input gradient
    input_grad_customized = customized_layer.input_gradients(
        input_np, output_official_np, output_grad_np)
    input_grad_official = [x.grad.numpy() for x in input_torch]
    success = verify_output(
        input_grad_customized,
        input_grad_official,
        [f'Input Gradient {i}' for i in range(len(input_grad_customized))]
    )
    if not success:
      return success

    # Check parameter gradient
    parameter_grad_customized = customized_layer.param_gradients(
        input_np, output_official_np, output_grad_np)
    for name in parameter_names:
      success = verify_output(parameter_grad_customized[name],
                              getattr(official_layer, name).grad.numpy(),
                              f"Parameter Gradient {name}")
      if not success:
        return success

    print('All tests passed')
    return success
```

Question A.1 Back-propagation of ADD

Complete the implementation of the following class below, then run the test below.

- We have provided the implementation of forward pass
- Complete the backward one `input_gradients`.
- Note that Add did not have parameters, so `param_gradients` just return an empty result.

```
#@title CustomizedAdd

class CustomizedAdd(CustomizedLayer):
  def __init__(self):
    pass

  def set_params(self, param_dict: Dict[str, torch.Tensor]) -> None:
    pass

  def forward(self, inputs: List[np.ndarray]) -> np.ndarray:
    x, y = inputs
    return x + y

  def input_gradients(
    self,
    inputs: List[np.ndarray],
    outputs: np.ndarray,
    output_gradient: np.ndarray
  ) -> Union[np.ndarray, List[np.ndarray]]:
    ######################
    # Your code goes here
    ######################
    grad_x = output_gradient
    grad_y = output_gradient
    return [grad_x, grad_y]

  def param_gradients(
      self,
      inputs: List[np.ndarray],
      outputs: np.ndarray,
      output_gradient: np.ndarray
  ) -> Dict[str, np.ndarray]:
    return {}
```

Run the test below. Do not change it.

```
#@title Run test

def verify_add():
  for width, height in [(3, 3), (10, 1), (100, 300)]:
    print(f'=== Testing with width={width} and height={height} ===')
    if width == 1:
      x1 = np.random.randn(height)
      x2 = np.random.randn(height)
```

```
    else:
      x1 = np.random.randn(height, width)
      x2 = np.random.randn(height, width)
    output_gradient = np.random.randn(height, width)
    customized_add = CustomizedAdd()
    customized_add.set_params({})
    class TorchAdd(nn.Module):
      def __init__(self):
        super(TorchAdd, self).__init__()
      def forward(self, x1, x2):
        return x1 + x2
    torch_add = TorchAdd()

    compare_with_actual_layer(
        torch_add,
        customized_add,
        [x1, x2],
        output_gradient,
        param_np=[],
        parameter_names=[],
    )

verify_add()
```

```
=== Testing with width=3 and height=3 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Input Gradient 1 is close to the reference output.
All tests passed
=== Testing with width=10 and height=1 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Input Gradient 1 is close to the reference output.
All tests passed
=== Testing with width=100 and height=300 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Input Gradient 1 is close to the reference output.
All tests passed
```

Question A.2 Back-propagation of ReLU

Similarly, complete the implementation of the following class below, then run the test below.

- We have provided the implementation of forward pass
- Complete the backward one `input_gradients`.
- Note that Add did not have parameters, so `param_gradients` just return an empty result.

Hint: for x=0, you can assume gradient of relu is 0.

```
#@title CustomizedReLU

class CustomizedReLU(CustomizedLayer):
  def __init__(self):
    pass

  def set_params(self, param_dict: Dict[str, torch.Tensor]) -> None:
    pass

  def forward(self, inputs: List[np.ndarray]) -> np.ndarray:
    x = inputs[0]
    return np.maximum(x, 0)

  def input_gradients(
    self,
    inputs: List[np.ndarray],
    outputs: np.ndarray,
    output_gradient: np.ndarray
  ) -> Union[np.ndarray, List[np.ndarray]]:
    ######################
    # Your code goes here
    ######################
    x = inputs[0]
    grad_x = output_gradient * (x > 0).astype(x.dtype)
    return [grad_x]

  def param_gradients(
      self,
      inputs: List[np.ndarray],
      outputs: np.ndarray,
      output_gradient: np.ndarray
  ) -> Dict[str, torch.Tensor]:
    return {}
```

Run the test below. Do not change it.

```
#@title Run test

def verify_relu():
  for width, height in [(3, 3), (10, 1), (100, 300)]:
    print(f'=== Testing with width={width} and height={height} ===')
    if width == 1:
      x = np.random.randn(height)
    else:
      x = np.random.randn(height, width)
    output_gradient = np.random.randn(height, width)
    customized_relu = CustomizedReLU()
    customized_relu.set_params({})
    torch_relu = nn.ReLU()
```

```
    compare_with_actual_layer(
        torch_relu,
        customized_relu,
        [x],
        output_gradient,
        param_np=[],
        parameter_names=[],
    )

verify_relu()
```

```
=== Testing with width=3 and height=3 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
All tests passed
=== Testing with width=10 and height=1 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
All tests passed
=== Testing with width=100 and height=300 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
All tests passed
```

Question A.3 Back-propagation of Linear

Similarly, complete the implementation of the following class below, then run the test below.

- This time, we also ask you to implement the forward pass.
- And complete the backward pass `input_gradients` and `param_gradients`.

```
#@title CustomizedLinear

class CustomizedLinear(CustomizedLayer):
  def __init__(self, in_features, out_features):
    self.in_features = in_features
    self.out_features = out_features
    self.weight = np.zeros((in_features,out_features))
    self.bias = np.zeros(out_features)

  def set_params(self, param_dict: Dict[str, torch.Tensor]):
    self.weight = param_dict['weight']
    self.bias = param_dict['bias']

  def forward(self, inputs: List[np.ndarray]) -> np.ndarray:
    ######################
    # Your code goes here
    ######################
```

```
    x = inputs[0]
    return np.dot(x,self.weight.T) + self.bias

  def input_gradients(
    self,
    inputs: List[np.ndarray],
    outputs: np.ndarray,
    output_gradient: np.ndarray
  ) -> Union[np.ndarray, List[np.ndarray]]:
    ######################
    # Your code goes here
    ######################
    grad_x = np.dot(output_gradient, self.weight)
    return [grad_x]

  def param_gradients(
      self,
      inputs: List[np.ndarray],
      outputs: np.ndarray,
      output_gradient: np.ndarray
  ) -> Dict[str, torch.Tensor]:
    ######################
    # Your code goes here
    ######################
    x = inputs[0]
    grad_w = np.dot(output_gradient.T,x)
    grad_b = np.sum(output_gradient, axis=0)
    return {'weight': grad_w,
            'bias': grad_b}
```

Run the test below. Do not change it.

```
#@title Run test

def verify_linear():
  for batch_size, in_features, out_features in [(3, 4, 8), (5, 10, 20), (2, 100,
300)]:
    print(f'=== Testing with in_features={in_features} '
          f'and out_features={out_features} ===')
    input_np = np.random.randn(batch_size, in_features)
    output_gradient = np.random.randn(batch_size, out_features)
    weight = np.random.randn(out_features, in_features)
    bias = np.random.randn(out_features)
    customized_linear = CustomizedLinear(in_features, out_features)
    customized_linear.set_params({'weight': weight, 'bias': bias})
    torch_linear = nn.Linear(out_features, in_features)

    compare_with_actual_layer(
        torch_linear,
        customized_linear,
        [input_np],
        output_gradient,
```

```
        param_np=[weight, bias],
        parameter_names=['weight', 'bias'],
    )

verify_linear()
```

```
=== Testing with in_features=4 and out_features=8 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Parameter Gradient weight is close to the reference output.
Parameter Gradient bias is close to the reference output.
All tests passed
=== Testing with in_features=10 and out_features=20 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Parameter Gradient weight is close to the reference output.
Parameter Gradient bias is close to the reference output.
All tests passed
=== Testing with in_features=100 and out_features=300 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Parameter Gradient weight is close to the reference output.
Parameter Gradient bias is close to the reference output.
All tests passed
```

Question A.4 Back-propagation of Conv

At last, let us try convolutional layer. To make your life easier, let us assume:

- Padding is always 0
- No striding
- Kernel is always squared

Similarly, complete the implementation of the following class below, then run the test below.

- For your reference, we have provided the forward pass.
- Please complete the backward pass `input_gradients` and `param_gradients`.

Also, we have additional requirement for this question:

- You cannot use any library functions that can do convolution, either from numpy, pytorch, or other libraries

```
#@title CustomizedConv2d

class CustomizedConv2d(CustomizedLayer):
  def __init__(self, in_channels, out_channels, kernel_size):
    if (kernel_size % 2 == 0):
```

```
        raise ValueError(f"Kernel size {kernel_size} must be odd")
      self.in_channels = in_channels
      self.out_channels = out_channels
      self.kernel_size = kernel_size
      self.weight = np.zeros((out_channels, in_channels, kernel_size, kernel_size))
      self.bias = np.zeros(out_channels)

    def set_params(self, param_dict: Dict[str, torch.Tensor]):
      self.weight = param_dict['weight']
      self.bias = param_dict['bias']

    def forward(self, inputs: List[np.ndarray]) -> np.ndarray:
      input = inputs[0] # [batch_size, in_channels, height, width]
      batch_size, in_channels, height, width = input.shape
      if self.in_channels != in_channels:
        raise ValueError(f'Input channels {in_channels} do not '
                         f'match expected {self.in_channels}')
      w = self.weight # [out_channels, in_channels, kernel_size, kernel_size]
      b = self.bias
      out_height = height - self.kernel_size + 1
      out_width = width - self.kernel_size + 1
      output = np.zeros((batch_size, self.out_channels, out_height, out_width),
                        np.float64)
      kf = (self.kernel_size - 1) // 2
      for dx in range(0, self.kernel_size):
        for dy in range(0, self.kernel_size):
          # channel order: [B, out_channels, in_channels, H, W]
          y_start = dy
          y_end = y_start + out_height
          x_start = dx
          x_end = x_start + out_width
          output += (input[:, np.newaxis, :, y_start:y_end, x_start:x_end] *
                     w[np.newaxis, :, :, np.newaxis, np.newaxis, dy, dx]
          ).sum(axis=2)
      output += b[np.newaxis, :, np.newaxis, np.newaxis]
      return output

    def input_gradients(
      self,
      inputs: List[np.ndarray],
      outputs: np.ndarray,
      output_gradient: np.ndarray
    ) -> Union[np.ndarray, List[np.ndarray]]:
      ######################
      # Your code goes here
      ######################
      x = inputs[0]
      batch_size, in_channels, height, width = x.shape
      _, out_channels, out_height, out_width = output_gradient.shape
      grad_x = np.zeros_like(x)
      for dx in range(self.kernel_size):
        for dy in range(self.kernel_size):
          y_start = dy
          y_end = y_start + out_height
```

```
        x_start = dx
        x_end = x_start + out_width
        for oc in range(out_channels):
          grad_x[:, :, y_start:y_end, x_start:x_end] += (
              output_gradient[:, oc:oc+1, :, :] *
              self.weight[oc, :, dy, dx][np.newaxis, :, np.newaxis, np.newaxis]
          )
    return [grad_x]

  def param_gradients(
    self,
    inputs: List[np.ndarray],
    outputs: np.ndarray,
    output_gradient: np.ndarray
  ) -> Dict[str, np.ndarray]:
    ######################
    # Your code goes here
    ######################
    x = inputs[0]
    batch_size, in_channels, height, width = x.shape
    _, out_channels, out_height, out_width = output_gradient.shape
    grad_w = np.zeros_like(self.weight)
    grad_b = np.zeros_like(self.bias)

    for dx in range(self.kernel_size):
      for dy in range(self.kernel_size):
        y_start = dy
        y_end = y_start + out_height
        x_start = dx
        x_end = x_start + out_width
        for oc in range(out_channels):
          grad_w[oc, :, dy, dx] = np.sum(
              x[:, :, y_start:y_end, x_start:x_end] *
              output_gradient[:, oc:oc+1, :, :],
              axis=(0, 2, 3)
          )
    grad_b = np.sum(output_gradient, axis=(0, 2, 3))
    return {
        'weight': grad_w,
        'bias': grad_b
    }
```

Run the test below. Do not change it.

```
#@title Run test

def verify_conv2d():
  for batch_size, in_channels, out_channels, kernel_size, height, width in [
      (1, 1, 1, 3, 5, 5),
      (1, 4, 5, 3, 11, 11),
      (1, 4, 7, 5, 12, 12),
      (3, 4, 5, 3, 20, 30),
```

```
        (1, 5, 7, 5, 256, 256),
    ]:
        print(f'=== Testing with in_channels={in_channels} '
              f'and out_channels={out_channels} and kernel_size={kernel_size} ===')
        out_height = height - kernel_size + 1
        out_width = width - kernel_size + 1
        input_np = np.random.randn(batch_size, in_channels, height, width)
        output_gradient = np.random.randn(batch_size, out_channels, out_height,
out_width)
        weight = np.random.randn(out_channels, in_channels, kernel_size, kernel_size)
        bias = np.random.randn(out_channels)
        customized_conv2d = CustomizedConv2d(in_channels, out_channels, kernel_size)
        customized_conv2d.set_params({'weight': weight, 'bias': bias})
        torch_conv2d = nn.Conv2d(in_channels, out_channels, kernel_size)

        compare_with_actual_layer(
            torch_conv2d,
            customized_conv2d,
            [input_np],
            output_gradient,
            param_np=[weight, bias],
            parameter_names=['weight', 'bias'],
        )

verify_conv2d()
```

```
=== Testing with in_channels=1 and out_channels=1 and kernel_size=3 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Parameter Gradient weight is close to the reference output.
Parameter Gradient bias is close to the reference output.
All tests passed
=== Testing with in_channels=4 and out_channels=5 and kernel_size=3 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Parameter Gradient weight is close to the reference output.
Parameter Gradient bias is close to the reference output.
All tests passed
=== Testing with in_channels=4 and out_channels=7 and kernel_size=5 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Parameter Gradient weight is close to the reference output.
Parameter Gradient bias is close to the reference output.
All tests passed
=== Testing with in_channels=4 and out_channels=5 and kernel_size=3 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Parameter Gradient weight is close to the reference output.
Parameter Gradient bias is close to the reference output.
All tests passed
```

```
=== Testing with in_channels=5 and out_channels=7 and kernel_size=5 ===
Output is close to the reference output.
Input Gradient 0 is close to the reference output.
Parameter Gradient weight is close to the reference output.
Parameter Gradient bias is close to the reference output.
All tests passed
```

Problem B. ResNet

For the ease of network design, we resize the image to 32x32.

```
#@title Load the training dataset

# Define transformations for the training and test sets

mean_train = 0.2860405743122101
std_train = 0.3530242443084717
transform_train = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((mean_train,), (std_train,))
])

# The transformation applied to the testing is the same as training
transform_test = transforms.Compose([
    # For the ease of network design, we resize the image to 32x32[
    transforms.Resize((32, 32)),
    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
)
```

```
100%|██████████| 26.4M/26.4M [00:02<00:00, 11.0MB/s]
100%|██████████| 29.5k/29.5k [00:00<00:00, 209kB/s]
100%|██████████| 4.42M/4.42M [00:01<00:00, 3.89MB/s]
100%|██████████| 5.15k/5.15k [00:00<00:00, 26.6MB/s]
```

Also, here are the same utility function as assignment 2. If necessary, feel free to change it.

```python
#@title Utility functions for training

def train_one_epoch(
    dataloader: torch.utils.data.DataLoader,
    model: nn.Module,
    loss_fn: nn.Module,
    optimizer: optim.Optimizer,
    device: str='cpu',
    loss_print_iter: int=100
  ):
  num_train_samples = len(dataloader.dataset)

  # Set the model to the training mod
  model.train()
  all_losses = []
  all_acc = []

  for batch_index, (image, label) in enumerate(dataloader):
    image, label = image.to(device), label.to(device)
    pred = model(image)
    loss = loss_fn(pred, label)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

    # Calculate the loss
    all_losses.append(loss.item())
    acc = ((pred.argmax(1) == label).type(torch.float).sum().item() /
           image.shape[0])
    all_acc.append(acc)

    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 all_losses, all_acc

def test_all_samples(
    dataloader: torch.utils.data.DataLoader,
    model: nn.Module,
    loss_fn: nn.Module,
    device: str='cpu'
```

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

And let us define the running devices.

GPU devices are suggested.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
Using device: cuda
```

In this question, let us verify why resnet structure is important for deep network.

First, let us create the data loader. Feel free to change the batch size.

```
# Create data loaders

batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Problem B.1 A simple CNN.

No coding is required. Just run the following blocks.

```
#@title An utility function to create network
```

```
def gen_double_conv(in_channels, out_channels):
  return [nn.Conv2d(in_channels, out_channels, 3, padding=1),
          nn.ReLU(),
          nn.Conv2d(out_channels, out_channels, 3, padding=1),
          nn.ReLU()]
```

First, we will define an CNN networks with 11 convolution/linear layers, and let us try it on FashineMNIST.

```
#@title SimpleCNN

class SimpleCNN(nn.Module):
  def __init__(
    self,
    img_size: int=32,
    in_channels: int=1,
    num_classes: int=10,
  ):
    super().__init__()
    base_channel = 16

    # Starting block. One 2x downampling.
    self.start_conv = nn.Sequential(
        *gen_double_conv(1, base_channel),
    )

    self.start_ds = nn.Sequential(
        nn.AvgPool2d(2),
    )

    # Mid block. No downsampling.
    self.mid_block = nn.Sequential(
        nn.Conv2d(base_channel, base_channel * 2, 3, padding=1),
        nn.ReLU(),
        nn.AvgPool2d(2),
        nn.Conv2d(base_channel * 2, base_channel * 2, 3, padding=1),
        nn.ReLU(),
        # We ConvTranspose2d to increase the resolution of an image by 2.
        nn.ConvTranspose2d(base_channel * 2, base_channel, 2, stride=2),
        nn.ReLU(),
    )

    # Last block. Two 2x downsampling.
    self.end_conv1 = nn.Sequential(
        *gen_double_conv(base_channel, base_channel * 2)
    )
    self.end_ds1 = nn.Sequential(
        nn.AvgPool2d(2),
    )
    self.end_conv2 = nn.Sequential(
        *gen_double_conv(base_channel * 2, base_channel * 4)
    )
```

```
    n_pooling_layer = 3
    last_layer_size = img_size // (2 ** n_pooling_layer)
    self.end_ds2_linear = nn.Sequential(
        nn.AvgPool2d(2),
        nn.Flatten(),
        nn.Linear(last_layer_size * last_layer_size *
                  base_channel * 4, base_channel * 8),
        nn.ReLU(),
        nn.Linear(base_channel * 8, num_classes)
    )

  def forward(self, x):
    after_start = self.start_ds(self.start_conv(x))
    after_mid = self.mid_block(after_start)
    output = self.end_ds2_linear(
        self.end_conv2(self.end_ds1(self.end_conv1(after_mid))))
    return output

  def get_n_conv_linear_layers(self) -> int:
    all_blocks = [self.start_conv, self.start_ds,
                  self.mid_block,
                  self.end_conv1, self.end_ds1,
                  self.end_conv2, self.end_ds2_linear]
    module_list = (nn.ConvTranspose2d, nn.Conv2d, nn.Linear)
    nlayer = 0
    for block in all_blocks:
      nlayer += len([mod for mod in block.modules()
                     if isinstance(mod, module_list)])
    return nlayer
```

```
#@title Create a model and corresponding optmizer

model = SimpleCNN()
model.to(device)
loss_fn = nn.CrossEntropyLoss()

epochs = 20   # Number of training epochs
optimizer = torch.optim.Adam(
  model.parameters(),    # All trainable parameters.
  lr=1e-4                # Learning rate
)

print(f'The model contains {model.get_n_conv_linear_layers()} conv or '
      f'linear layers')
```

```
The model contains 11 conv or linear layers
```

Running the training below.

After training, you should be avaiable to get accuracy of 89% on the testing set.

```
#@title Run training

for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  start_time =time.time()
  _, _ = train_one_epoch(train_loader, model, loss_fn, optimizer, device=device)
  print(f'One epoch takes {time.time() - start_time}')
  _, _ = test_all_samples(test_loader, model, loss_fn, device=device)

test_all_samples(test_loader, model, loss_fn, device=device)

print("Training done!")
```

```
Epoch 1
-------------------------------
loss: 2.297736 [   32/60000]
loss: 2.235576 [ 3232/60000]
loss: 0.919315 [ 6432/60000]
loss: 0.699960 [ 9632/60000]
loss: 0.623896 [12832/60000]
loss: 0.863583 [16032/60000]
loss: 0.756371 [19232/60000]
loss: 0.710544 [22432/60000]
loss: 0.800340 [25632/60000]
loss: 0.588454 [28832/60000]
loss: 0.777952 [32032/60000]
loss: 0.632524 [35232/60000]
loss: 0.659805 [38432/60000]
loss: 0.959370 [41632/60000]
loss: 0.454744 [44832/60000]
loss: 0.910513 [48032/60000]
loss: 0.625027 [51232/60000]
loss: 0.354011 [54432/60000]
loss: 0.511700 [57632/60000]
One epoch takes 27.26918077468872
Test Error:
 Accuracy: 73.8%, Avg loss: 0.662984

Epoch 2
-------------------------------
loss: 0.697680 [   32/60000]
loss: 0.509128 [ 3232/60000]
loss: 0.758403 [ 6432/60000]
loss: 0.653725 [ 9632/60000]
loss: 0.522013 [12832/60000]
loss: 0.519024 [16032/60000]
```

```
loss: 0.584618 [19232/60000]
loss: 0.593864 [22432/60000]
loss: 0.558848 [25632/60000]
loss: 0.553335 [28832/60000]
loss: 0.441373 [32032/60000]
loss: 0.456937 [35232/60000]
loss: 0.818964 [38432/60000]
loss: 0.430084 [41632/60000]
loss: 0.532991 [44832/60000]
loss: 0.466305 [48032/60000]
loss: 0.562671 [51232/60000]
loss: 0.556237 [54432/60000]
loss: 0.565536 [57632/60000]
One epoch takes 25.82933020591736
Test Error:
 Accuracy: 78.6%, Avg loss: 0.559191

Epoch 3
-------------------------------
loss: 0.812886 [   32/60000]
loss: 0.453518 [ 3232/60000]
loss: 0.533239 [ 6432/60000]
loss: 0.348498 [ 9632/60000]
loss: 0.357188 [12832/60000]
loss: 0.590914 [16032/60000]
loss: 0.520601 [19232/60000]
loss: 0.680230 [22432/60000]
loss: 0.776465 [25632/60000]
loss: 0.573567 [28832/60000]
loss: 0.362547 [32032/60000]
loss: 0.294106 [35232/60000]
loss: 0.442009 [38432/60000]
loss: 0.662747 [41632/60000]
loss: 0.403804 [44832/60000]
loss: 0.391109 [48032/60000]
loss: 0.435858 [51232/60000]
loss: 0.375418 [54432/60000]
loss: 0.542666 [57632/60000]
One epoch takes 25.639429330825806
Test Error:
 Accuracy: 82.9%, Avg loss: 0.463563

Epoch 4
-------------------------------
loss: 0.365183 [   32/60000]
loss: 0.454861 [ 3232/60000]
loss: 0.420903 [ 6432/60000]
loss: 0.434710 [ 9632/60000]
loss: 0.571609 [12832/60000]
loss: 0.297285 [16032/60000]
loss: 0.345900 [19232/60000]
```

```
loss: 0.377569 [22432/60000]
loss: 0.457717 [25632/60000]
loss: 0.354617 [28832/60000]
loss: 0.297544 [32032/60000]
loss: 0.444616 [35232/60000]
loss: 0.372396 [38432/60000]
loss: 0.623269 [41632/60000]
loss: 0.404205 [44832/60000]
loss: 0.576991 [48032/60000]
loss: 0.341540 [51232/60000]
loss: 0.481927 [54432/60000]
loss: 0.339210 [57632/60000]
One epoch takes 25.19131588935852
Test Error:
 Accuracy: 84.4%, Avg loss: 0.417481

Epoch 5
-------------------------------
loss: 0.358426 [   32/60000]
loss: 0.365550 [ 3232/60000]
loss: 0.473461 [ 6432/60000]
loss: 0.335972 [ 9632/60000]
loss: 0.484150 [12832/60000]
loss: 0.562335 [16032/60000]
loss: 0.317384 [19232/60000]
loss: 0.291181 [22432/60000]
loss: 0.308512 [25632/60000]
loss: 0.780750 [28832/60000]
loss: 0.561253 [32032/60000]
loss: 0.471801 [35232/60000]
loss: 0.239376 [38432/60000]
loss: 0.592940 [41632/60000]
loss: 0.294595 [44832/60000]
loss: 0.305832 [48032/60000]
loss: 0.306059 [51232/60000]
loss: 0.434685 [54432/60000]
loss: 0.300350 [57632/60000]
One epoch takes 25.36003017425537
Test Error:
 Accuracy: 84.5%, Avg loss: 0.415821

Epoch 6
-------------------------------
loss: 0.465660 [   32/60000]
loss: 0.129147 [ 3232/60000]
loss: 0.522631 [ 6432/60000]
loss: 0.127695 [ 9632/60000]
loss: 0.475722 [12832/60000]
loss: 0.383680 [16032/60000]
loss: 0.199294 [19232/60000]
loss: 0.419956 [22432/60000]
```

```
loss: 0.139582 [25632/60000]
loss: 0.304168 [28832/60000]
loss: 0.197557 [32032/60000]
loss: 0.349470 [35232/60000]
loss: 0.416504 [38432/60000]
loss: 0.196921 [41632/60000]
loss: 0.280208 [44832/60000]
loss: 0.323261 [48032/60000]
loss: 0.427375 [51232/60000]
loss: 0.367031 [54432/60000]
loss: 0.431781 [57632/60000]
One epoch takes 25.67291808128357
Test Error:
 Accuracy: 85.6%, Avg loss: 0.390058

Epoch 7
-------------------------------
loss: 0.409121 [   32/60000]
loss: 0.537415 [ 3232/60000]
loss: 0.448114 [ 6432/60000]
loss: 0.266243 [ 9632/60000]
loss: 0.156280 [12832/60000]
loss: 0.287372 [16032/60000]
loss: 0.321715 [19232/60000]
loss: 0.396972 [22432/60000]
loss: 0.242352 [25632/60000]
loss: 0.098967 [28832/60000]
loss: 0.318911 [32032/60000]
loss: 0.431383 [35232/60000]
loss: 0.333402 [38432/60000]
loss: 0.226108 [41632/60000]
loss: 0.448978 [44832/60000]
loss: 0.362270 [48032/60000]
loss: 0.248244 [51232/60000]
loss: 0.293990 [54432/60000]
loss: 0.425763 [57632/60000]
One epoch takes 25.55284094810486
Test Error:
 Accuracy: 86.7%, Avg loss: 0.354657

Epoch 8
-------------------------------
loss: 0.273685 [   32/60000]
loss: 0.238825 [ 3232/60000]
loss: 0.150432 [ 6432/60000]
loss: 0.396989 [ 9632/60000]
loss: 0.148318 [12832/60000]
loss: 0.380804 [16032/60000]
loss: 0.298401 [19232/60000]
loss: 0.258468 [22432/60000]
loss: 0.202153 [25632/60000]
```

```
loss: 0.204157 [28832/60000]
loss: 0.334011 [32032/60000]
loss: 0.080030 [35232/60000]
loss: 0.131482 [38432/60000]
loss: 0.366100 [41632/60000]
loss: 0.423104 [44832/60000]
loss: 0.228402 [48032/60000]
loss: 0.280715 [51232/60000]
loss: 0.182781 [54432/60000]
loss: 0.320759 [57632/60000]
One epoch takes 25.61137294769287
Test Error:
 Accuracy: 87.5%, Avg loss: 0.339301

Epoch 9
-------------------------------
loss: 0.248227 [   32/60000]
loss: 0.365013 [ 3232/60000]
loss: 0.340779 [ 6432/60000]
loss: 0.353613 [ 9632/60000]
loss: 0.228145 [12832/60000]
loss: 0.355464 [16032/60000]
loss: 0.179594 [19232/60000]
loss: 0.373484 [22432/60000]
loss: 0.419006 [25632/60000]
loss: 0.320714 [28832/60000]
loss: 0.396346 [32032/60000]
loss: 0.340802 [35232/60000]
loss: 0.263206 [38432/60000]
loss: 0.300169 [41632/60000]
loss: 0.434190 [44832/60000]
loss: 0.134992 [48032/60000]
loss: 0.140242 [51232/60000]
loss: 0.452480 [54432/60000]
loss: 0.191456 [57632/60000]
One epoch takes 26.171803951263428
Test Error:
 Accuracy: 87.7%, Avg loss: 0.328815

Epoch 10
-------------------------------
loss: 0.466868 [   32/60000]
loss: 0.254635 [ 3232/60000]
loss: 0.285901 [ 6432/60000]
loss: 0.078331 [ 9632/60000]
loss: 0.337318 [12832/60000]
loss: 0.360626 [16032/60000]
loss: 0.200829 [19232/60000]
loss: 0.256140 [22432/60000]
loss: 0.339342 [25632/60000]
loss: 0.089705 [28832/60000]
```

```
loss: 0.180096 [32032/60000]
loss: 0.153492 [35232/60000]
loss: 0.486314 [38432/60000]
loss: 0.169022 [41632/60000]
loss: 0.308449 [44832/60000]
loss: 0.550638 [48032/60000]
loss: 0.227430 [51232/60000]
loss: 0.079632 [54432/60000]
loss: 0.121422 [57632/60000]
One epoch takes 26.7770516872406
Test Error:
 Accuracy: 88.0%, Avg loss: 0.324617

Epoch 11
-------------------------------
loss: 0.268395 [   32/60000]
loss: 0.289901 [ 3232/60000]
loss: 0.473514 [ 6432/60000]
loss: 0.230974 [ 9632/60000]
loss: 0.231742 [12832/60000]
loss: 0.095060 [16032/60000]
loss: 0.125420 [19232/60000]
loss: 0.281799 [22432/60000]
loss: 0.334580 [25632/60000]
loss: 0.241873 [28832/60000]
loss: 0.090076 [32032/60000]
loss: 0.233693 [35232/60000]
loss: 0.464635 [38432/60000]
loss: 0.062476 [41632/60000]
loss: 0.268905 [44832/60000]
loss: 0.205996 [48032/60000]
loss: 0.189565 [51232/60000]
loss: 0.442159 [54432/60000]
loss: 0.235319 [57632/60000]
One epoch takes 27.06607413291931
Test Error:
 Accuracy: 88.3%, Avg loss: 0.313170

Epoch 12
-------------------------------
loss: 0.156778 [   32/60000]
loss: 0.482919 [ 3232/60000]
loss: 0.364379 [ 6432/60000]
loss: 0.288835 [ 9632/60000]
loss: 0.302254 [12832/60000]
loss: 0.403462 [16032/60000]
loss: 0.164105 [19232/60000]
loss: 0.220942 [22432/60000]
loss: 0.179306 [25632/60000]
loss: 0.224240 [28832/60000]
loss: 0.174547 [32032/60000]
```

```
loss: 0.374494 [35232/60000]
loss: 0.284665 [38432/60000]
loss: 0.277219 [41632/60000]
loss: 0.204552 [44832/60000]
loss: 0.477161 [48032/60000]
loss: 0.256204 [51232/60000]
loss: 0.431683 [54432/60000]
loss: 0.296712 [57632/60000]
One epoch takes 26.741021633148193
Test Error:
 Accuracy: 89.1%, Avg loss: 0.301131

Epoch 13
-------------------------------
loss: 0.159364 [   32/60000]
loss: 0.305010 [ 3232/60000]
loss: 0.158494 [ 6432/60000]
loss: 0.357527 [ 9632/60000]
loss: 0.380310 [12832/60000]
loss: 0.189881 [16032/60000]
loss: 0.231745 [19232/60000]
loss: 0.428348 [22432/60000]
loss: 0.519370 [25632/60000]
loss: 0.141978 [28832/60000]
loss: 0.329399 [32032/60000]
loss: 0.388044 [35232/60000]
loss: 0.271593 [38432/60000]
loss: 0.222212 [41632/60000]
loss: 0.347879 [44832/60000]
loss: 0.133572 [48032/60000]
loss: 0.230810 [51232/60000]
loss: 0.208944 [54432/60000]
loss: 0.185791 [57632/60000]
One epoch takes 27.448895931243896
Test Error:
 Accuracy: 88.8%, Avg loss: 0.303119

Epoch 14
-------------------------------
loss: 0.153709 [   32/60000]
loss: 0.271682 [ 3232/60000]
loss: 0.494952 [ 6432/60000]
loss: 0.073443 [ 9632/60000]
loss: 0.158582 [12832/60000]
loss: 0.331521 [16032/60000]
loss: 0.435311 [19232/60000]
loss: 0.146683 [22432/60000]
loss: 0.386781 [25632/60000]
loss: 0.377032 [28832/60000]
loss: 0.255849 [32032/60000]
loss: 0.345101 [35232/60000]
```

```
loss: 0.175766 [38432/60000]
loss: 0.315401 [41632/60000]
loss: 0.253217 [44832/60000]
loss: 0.262259 [48032/60000]
loss: 0.162946 [51232/60000]
loss: 0.130304 [54432/60000]
loss: 0.151975 [57632/60000]
One epoch takes 27.15833568572998
Test Error:
 Accuracy: 89.1%, Avg loss: 0.288083

Epoch 15
-------------------------------
loss: 0.387041 [   32/60000]
loss: 0.244723 [ 3232/60000]
loss: 0.320544 [ 6432/60000]
loss: 0.235483 [ 9632/60000]
loss: 0.139709 [12832/60000]
loss: 0.608851 [16032/60000]
loss: 0.195895 [19232/60000]
loss: 0.339447 [22432/60000]
loss: 0.070748 [25632/60000]
loss: 0.185028 [28832/60000]
loss: 0.134043 [32032/60000]
loss: 0.278618 [35232/60000]
loss: 0.228988 [38432/60000]
loss: 0.129275 [41632/60000]
loss: 0.285978 [44832/60000]
loss: 0.165174 [48032/60000]
loss: 0.442344 [51232/60000]
loss: 0.193648 [54432/60000]
loss: 0.259667 [57632/60000]
One epoch takes 26.822678804397583
Test Error:
 Accuracy: 89.2%, Avg loss: 0.298052

Epoch 16
-------------------------------
loss: 0.385520 [   32/60000]
loss: 0.124100 [ 3232/60000]
loss: 0.528379 [ 6432/60000]
loss: 0.244626 [ 9632/60000]
loss: 0.307356 [12832/60000]
loss: 0.239283 [16032/60000]
loss: 0.237338 [19232/60000]
loss: 0.231597 [22432/60000]
loss: 0.307159 [25632/60000]
loss: 0.365381 [28832/60000]
loss: 0.284822 [32032/60000]
loss: 0.168508 [35232/60000]
loss: 0.132055 [38432/60000]
```

```
loss: 0.282825 [41632/60000]
loss: 0.204207 [44832/60000]
loss: 0.377513 [48032/60000]
loss: 0.394354 [51232/60000]
loss: 0.207179 [54432/60000]
loss: 0.095653 [57632/60000]
One epoch takes 26.270687580108643
Test Error:
 Accuracy: 89.9%, Avg loss: 0.279611

Epoch 17
-------------------------------
loss: 0.199888 [   32/60000]
loss: 0.282926 [ 3232/60000]
loss: 0.306031 [ 6432/60000]
loss: 0.310444 [ 9632/60000]
loss: 0.092469 [12832/60000]
loss: 0.184238 [16032/60000]
loss: 0.277865 [19232/60000]
loss: 0.384646 [22432/60000]
loss: 0.100709 [25632/60000]
loss: 0.126286 [28832/60000]
loss: 0.292777 [32032/60000]
loss: 0.242591 [35232/60000]
loss: 0.180557 [38432/60000]
loss: 0.158250 [41632/60000]
loss: 0.459267 [44832/60000]
loss: 0.326155 [48032/60000]
loss: 0.162747 [51232/60000]
loss: 0.096437 [54432/60000]
loss: 0.099594 [57632/60000]
One epoch takes 26.355380058288574
Test Error:
 Accuracy: 89.7%, Avg loss: 0.280891

Epoch 18
-------------------------------
loss: 0.354213 [   32/60000]
loss: 0.363784 [ 3232/60000]
loss: 0.108140 [ 6432/60000]
loss: 0.281843 [ 9632/60000]
loss: 0.200095 [12832/60000]
loss: 0.254014 [16032/60000]
loss: 0.266998 [19232/60000]
loss: 0.170889 [22432/60000]
loss: 0.156812 [25632/60000]
loss: 0.306701 [28832/60000]
loss: 0.141223 [32032/60000]
loss: 0.126887 [35232/60000]
loss: 0.209207 [38432/60000]
loss: 0.183537 [41632/60000]
```

```
loss: 0.219431 [44832/60000]
loss: 0.324837 [48032/60000]
loss: 0.212245 [51232/60000]
loss: 0.248877 [54432/60000]
loss: 0.398835 [57632/60000]
One epoch takes 26.15299153327942
Test Error:
 Accuracy: 90.0%, Avg loss: 0.275450

Epoch 19
-------------------------------
loss: 0.093985 [   32/60000]
loss: 0.551365 [ 3232/60000]
loss: 0.058557 [ 6432/60000]
loss: 0.137703 [ 9632/60000]
loss: 0.190026 [12832/60000]
loss: 0.117383 [16032/60000]
loss: 0.129716 [19232/60000]
loss: 0.234115 [22432/60000]
loss: 0.163683 [25632/60000]
loss: 0.283108 [28832/60000]
loss: 0.151511 [32032/60000]
loss: 0.257342 [35232/60000]
loss: 0.103038 [38432/60000]
loss: 0.370758 [41632/60000]
loss: 0.130640 [44832/60000]
loss: 0.271714 [48032/60000]
loss: 0.144983 [51232/60000]
loss: 0.267600 [54432/60000]
loss: 0.258806 [57632/60000]
One epoch takes 26.001039028167725
Test Error:
 Accuracy: 90.1%, Avg loss: 0.274933

Epoch 20
-------------------------------
loss: 0.322409 [   32/60000]
loss: 0.066484 [ 3232/60000]
loss: 0.087295 [ 6432/60000]
loss: 0.341265 [ 9632/60000]
loss: 0.271896 [12832/60000]
loss: 0.269165 [16032/60000]
loss: 0.171736 [19232/60000]
loss: 0.144848 [22432/60000]
loss: 0.213355 [25632/60000]
loss: 0.279024 [28832/60000]
loss: 0.466890 [32032/60000]
loss: 0.171449 [35232/60000]
loss: 0.199738 [38432/60000]
loss: 0.355121 [41632/60000]
loss: 0.137161 [44832/60000]
```

```
loss: 0.188522 [48032/60000]
loss: 0.312974 [51232/60000]
loss: 0.228070 [54432/60000]
loss: 0.187547 [57632/60000]
One epoch takes 26.018440008163452
Test Error:
 Accuracy: 90.4%, Avg loss: 0.271083

Test Error:
 Accuracy: 90.4%, Avg loss: 0.271083

Training done!
```

Problem B.2. Deeper CNN

Now, please change the network we have provided, by duplicating some of conv parts. For example, you can make a multiple copy of self.mid_block. The network you have created should have at least 30 conv/linear layers.

```python
class DeeperCNN(nn.Module):
  def __init__(
    self,
    img_size: int = 32,
    in_channels: int = 1,
    num_classes: int = 10,
  ):
    super().__init__()
    base_channel = 16

    # Starting block
    self.start_conv = nn.Sequential(
        *gen_double_conv(in_channels, base_channel)
    )
    self.start_ds = nn.AvgPool2d(2)

    # Mid block
    mid_blocks = []
    in_ch = base_channel
    out_ch = base_channel * 2
    for _ in range(6):  # 6 个 mid_block
        mid_blocks.append(nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.ReLU(),
            nn.AvgPool2d(2),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(out_ch, in_ch, 2, stride=2),
            nn.ReLU()
```

```
        ))
    self.mid_blocks = nn.ModuleList(mid_blocks)

    # Last block
    self.end_conv1 = nn.Sequential(
        *gen_double_conv(base_channel, base_channel * 2)
    )
    self.end_ds1 = nn.AvgPool2d(2)
    self.end_conv2 = nn.Sequential(
        *gen_double_conv(base_channel * 2, base_channel * 4)
    )

    last_layer_size = img_size // (2 ** 3)
    self.end_ds2_linear = nn.Sequential(
        nn.AvgPool2d(2),
        nn.Flatten(),
        nn.Linear(last_layer_size * last_layer_size * base_channel * 4,
base_channel * 8),
        nn.ReLU(),
        nn.Linear(base_channel * 8, num_classes)
    )

  def forward(self, x):
    x = self.start_ds(self.start_conv(x))
    for block in self.mid_blocks:
        x = block(x)
    x = self.end_ds2_linear(self.end_conv2(self.end_ds1(self.end_conv1(x))))
    return x

  def get_n_conv_linear_layers(self) -> int:
    all_blocks = [
        self.start_conv, self.start_ds,
        *self.mid_blocks, self.end_conv1, self.end_ds1,
        self.end_conv2, self.end_ds2_linear
    ]
    module_list = (nn.Conv2d, nn.ConvTranspose2d, nn.Linear)
    nlayer = 0
    for block in all_blocks:
        nlayer += len([mod for mod in block.modules() if isinstance(mod,
module_list)])
    return nlayer
```

```
#@title Create a model and corresponding optmizer

model = DeeperCNN()
model.to(device)
loss_fn = nn.CrossEntropyLoss()

epochs = 20   # Number of training epochs
optimizer = torch.optim.Adam(
  model.parameters(),    # All trainable parameters.
```

```
  lr=1e-4                # Learning rate
)

print(f'The model contains {model.get_n_conv_linear_layers()} conv or '
      f'linear layers')
```

```
The model contains 32 conv or linear layers
```

Running the training below.

Interesting, you will find that with deeper network, the accuracy actually goes down.

```
#@title Run training

for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  start_time =time.time()
  _, _ = train_one_epoch(train_loader, model, loss_fn, optimizer, device=device)
  print(f'One epoch takes {time.time() - start_time}')
  _, _ = test_all_samples(test_loader, model, loss_fn, device=device)

test_all_samples(test_loader, model, loss_fn, device=device)

print("Training done!")
```

```
Epoch 1
-------------------------------
loss: 2.296424 [   32/60000]
loss: 2.296483 [ 3232/60000]
loss: 2.306451 [ 6432/60000]
loss: 2.291713 [ 9632/60000]
loss: 2.301537 [12832/60000]
loss: 2.302730 [16032/60000]
loss: 2.303861 [19232/60000]
loss: 2.300101 [22432/60000]
loss: 2.305724 [25632/60000]
loss: 2.301529 [28832/60000]
loss: 2.301438 [32032/60000]
loss: 2.304308 [35232/60000]
loss: 2.307059 [38432/60000]
loss: 2.290480 [41632/60000]
loss: 2.304431 [44832/60000]
loss: 2.303193 [48032/60000]
loss: 2.305325 [51232/60000]
loss: 2.300168 [54432/60000]
loss: 2.303761 [57632/60000]
```

```
One epoch takes 34.28541660308838
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302657

Epoch 2
-------------------------------
loss: 2.299863 [   32/60000]
loss: 2.302035 [ 3232/60000]
loss: 2.306422 [ 6432/60000]
loss: 2.298994 [ 9632/60000]
loss: 2.302483 [12832/60000]
loss: 2.306445 [16032/60000]
loss: 2.302811 [19232/60000]
loss: 2.304950 [22432/60000]
loss: 2.303362 [25632/60000]
loss: 2.305706 [28832/60000]
loss: 2.308852 [32032/60000]
loss: 2.302210 [35232/60000]
loss: 2.304250 [38432/60000]
loss: 2.302684 [41632/60000]
loss: 2.301793 [44832/60000]
loss: 2.301969 [48032/60000]
loss: 2.302847 [51232/60000]
loss: 2.299973 [54432/60000]
loss: 2.297915 [57632/60000]
One epoch takes 34.950209617614746
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302621

Epoch 3
-------------------------------
loss: 2.303431 [   32/60000]
loss: 2.303319 [ 3232/60000]
loss: 2.301727 [ 6432/60000]
loss: 2.304741 [ 9632/60000]
loss: 2.301761 [12832/60000]
loss: 2.302819 [16032/60000]
loss: 2.299970 [19232/60000]
loss: 2.308623 [22432/60000]
loss: 2.306213 [25632/60000]
loss: 2.302439 [28832/60000]
loss: 2.303945 [32032/60000]
loss: 2.301562 [35232/60000]
loss: 2.303973 [38432/60000]
loss: 2.301196 [41632/60000]
loss: 2.304501 [44832/60000]
loss: 2.304320 [48032/60000]
loss: 2.299997 [51232/60000]
loss: 2.301250 [54432/60000]
loss: 2.303461 [57632/60000]
One epoch takes 35.37681555747986
```

```
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302627

Epoch 4
-------------------------------
loss: 2.303912 [   32/60000]
loss: 2.304092 [ 3232/60000]
loss: 2.300921 [ 6432/60000]
loss: 2.303999 [ 9632/60000]
loss: 2.304758 [12832/60000]
loss: 2.301660 [16032/60000]
loss: 2.303245 [19232/60000]
loss: 2.301865 [22432/60000]
loss: 2.303132 [25632/60000]
loss: 2.303878 [28832/60000]
loss: 2.303364 [32032/60000]
loss: 2.302064 [35232/60000]
loss: 2.304175 [38432/60000]
loss: 2.304283 [41632/60000]
loss: 2.301147 [44832/60000]
loss: 2.304511 [48032/60000]
loss: 2.302815 [51232/60000]
loss: 2.301025 [54432/60000]
loss: 2.303916 [57632/60000]
One epoch takes 35.56762409210205
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302735

Epoch 5
-------------------------------
loss: 2.301207 [   32/60000]
loss: 2.302446 [ 3232/60000]
loss: 2.301330 [ 6432/60000]
loss: 2.301762 [ 9632/60000]
loss: 2.303312 [12832/60000]
loss: 2.302371 [16032/60000]
loss: 2.301073 [19232/60000]
loss: 2.301420 [22432/60000]
loss: 2.299967 [25632/60000]
loss: 2.302865 [28832/60000]
loss: 2.302042 [32032/60000]
loss: 2.301584 [35232/60000]
loss: 2.305849 [38432/60000]
loss: 2.302345 [41632/60000]
loss: 2.300154 [44832/60000]
loss: 2.302360 [48032/60000]
loss: 2.302931 [51232/60000]
loss: 2.306270 [54432/60000]
loss: 2.303652 [57632/60000]
One epoch takes 34.91383910179138
Test Error:
```

```
 Accuracy: 10.0%, Avg loss: 2.302623

Epoch 6
-------------------------------
loss: 2.302324 [   32/60000]
loss: 2.300538 [ 3232/60000]
loss: 2.305446 [ 6432/60000]
loss: 2.302370 [ 9632/60000]
loss: 2.300959 [12832/60000]
loss: 2.301695 [16032/60000]
loss: 2.300411 [19232/60000]
loss: 2.300962 [22432/60000]
loss: 2.306212 [25632/60000]
loss: 2.303579 [28832/60000]
loss: 2.301833 [32032/60000]
loss: 2.302967 [35232/60000]
loss: 2.300673 [38432/60000]
loss: 2.304538 [41632/60000]
loss: 2.307719 [44832/60000]
loss: 2.300217 [48032/60000]
loss: 2.302008 [51232/60000]
loss: 2.301086 [54432/60000]
loss: 2.303460 [57632/60000]
One epoch takes 35.12574625015259
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302591

Epoch 7
-------------------------------
loss: 2.301951 [   32/60000]
loss: 2.303441 [ 3232/60000]
loss: 2.302596 [ 6432/60000]
loss: 2.301467 [ 9632/60000]
loss: 2.302844 [12832/60000]
loss: 2.305110 [16032/60000]
loss: 2.302158 [19232/60000]
loss: 2.304240 [22432/60000]
loss: 2.298556 [25632/60000]
loss: 2.300628 [28832/60000]
loss: 2.303118 [32032/60000]
loss: 2.303473 [35232/60000]
loss: 2.302385 [38432/60000]
loss: 2.302019 [41632/60000]
loss: 2.302122 [44832/60000]
loss: 2.303449 [48032/60000]
loss: 2.301592 [51232/60000]
loss: 2.303139 [54432/60000]
loss: 2.303421 [57632/60000]
One epoch takes 34.98262071609497
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302589
```

```
Epoch 8
-------------------------------
loss: 2.302361 [   32/60000]
loss: 2.303741 [ 3232/60000]
loss: 2.302435 [ 6432/60000]
loss: 2.303634 [ 9632/60000]
loss: 2.302760 [12832/60000]
loss: 2.301052 [16032/60000]
loss: 2.300157 [19232/60000]
loss: 2.304319 [22432/60000]
loss: 2.302392 [25632/60000]
loss: 2.302161 [28832/60000]
loss: 2.303813 [32032/60000]
loss: 2.302701 [35232/60000]
loss: 2.303200 [38432/60000]
loss: 2.302765 [41632/60000]
loss: 2.302990 [44832/60000]
loss: 2.301208 [48032/60000]
loss: 2.302868 [51232/60000]
loss: 2.302814 [54432/60000]
loss: 2.302779 [57632/60000]
One epoch takes 35.31421399116516
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302606

Epoch 9
-------------------------------
loss: 2.303159 [   32/60000]
loss: 2.304212 [ 3232/60000]
loss: 2.303064 [ 6432/60000]
loss: 2.301250 [ 9632/60000]
loss: 2.303226 [12832/60000]
loss: 2.303298 [16032/60000]
loss: 2.302861 [19232/60000]
loss: 2.302437 [22432/60000]
loss: 2.302929 [25632/60000]
loss: 2.302486 [28832/60000]
loss: 2.303917 [32032/60000]
loss: 2.303522 [35232/60000]
loss: 2.300854 [38432/60000]
loss: 2.303269 [41632/60000]
loss: 2.302862 [44832/60000]
loss: 2.303976 [48032/60000]
loss: 2.304668 [51232/60000]
loss: 2.301903 [54432/60000]
loss: 2.302519 [57632/60000]
One epoch takes 35.23510265350342
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302599
```

```
Epoch 10
-------------------------------
loss: 2.303286 [   32/60000]
loss: 2.302398 [ 3232/60000]
loss: 2.302032 [ 6432/60000]
loss: 2.302620 [ 9632/60000]
loss: 2.303607 [12832/60000]
loss: 2.304225 [16032/60000]
loss: 2.302118 [19232/60000]
loss: 2.300117 [22432/60000]
loss: 2.305063 [25632/60000]
loss: 2.302801 [28832/60000]
loss: 2.300320 [32032/60000]
loss: 2.303346 [35232/60000]
loss: 2.302724 [38432/60000]
loss: 2.303707 [41632/60000]
loss: 2.302192 [44832/60000]
loss: 2.303327 [48032/60000]
loss: 2.306211 [51232/60000]
loss: 2.305924 [54432/60000]
loss: 2.303082 [57632/60000]
One epoch takes 35.39675259590149
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302610

Epoch 11
-------------------------------
loss: 2.303247 [   32/60000]
loss: 2.305560 [ 3232/60000]
loss: 2.302023 [ 6432/60000]
loss: 2.308245 [ 9632/60000]
loss: 2.303564 [12832/60000]
loss: 2.304309 [16032/60000]
loss: 2.301647 [19232/60000]
loss: 2.303321 [22432/60000]
loss: 2.302688 [25632/60000]
loss: 2.298375 [28832/60000]
loss: 2.302873 [32032/60000]
loss: 2.302605 [35232/60000]
loss: 2.304474 [38432/60000]
loss: 2.303205 [41632/60000]
loss: 2.302356 [44832/60000]
loss: 2.304126 [48032/60000]
loss: 2.300950 [51232/60000]
loss: 2.303629 [54432/60000]
loss: 2.301976 [57632/60000]
One epoch takes 35.137104749679565
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302598

Epoch 12
```

```
-------------------------------
loss: 2.302936 [   32/60000]
loss: 2.303466 [ 3232/60000]
loss: 2.302789 [ 6432/60000]
loss: 2.300859 [ 9632/60000]
loss: 2.302408 [12832/60000]
loss: 2.302813 [16032/60000]
loss: 2.303998 [19232/60000]
loss: 2.303270 [22432/60000]
loss: 2.302959 [25632/60000]
loss: 2.300706 [28832/60000]
loss: 2.301570 [32032/60000]
loss: 2.301296 [35232/60000]
loss: 2.302804 [38432/60000]
loss: 2.305844 [41632/60000]
loss: 2.299922 [44832/60000]
loss: 2.305164 [48032/60000]
loss: 2.300935 [51232/60000]
loss: 2.301165 [54432/60000]
loss: 2.302577 [57632/60000]
One epoch takes 34.82882833480835
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302593

Epoch 13
-------------------------------
loss: 2.303214 [   32/60000]
loss: 2.303481 [ 3232/60000]
loss: 2.300579 [ 6432/60000]
loss: 2.302870 [ 9632/60000]
loss: 2.303085 [12832/60000]
loss: 2.304533 [16032/60000]
loss: 2.303060 [19232/60000]
loss: 2.302496 [22432/60000]
loss: 2.301526 [25632/60000]
loss: 2.300729 [28832/60000]
loss: 2.302177 [32032/60000]
loss: 2.305545 [35232/60000]
loss: 2.303893 [38432/60000]
loss: 2.302416 [41632/60000]
loss: 2.304213 [44832/60000]
loss: 2.303270 [48032/60000]
loss: 2.303692 [51232/60000]
loss: 2.301795 [54432/60000]
loss: 2.302369 [57632/60000]
One epoch takes 34.94845461845398
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302609

Epoch 14
-------------------------------
```

```
loss: 2.300775 [   32/60000]
loss: 2.303624 [ 3232/60000]
loss: 2.302406 [ 6432/60000]
loss: 2.301701 [ 9632/60000]
loss: 2.300453 [12832/60000]
loss: 2.303254 [16032/60000]
loss: 2.301875 [19232/60000]
loss: 2.304429 [22432/60000]
loss: 2.302963 [25632/60000]
loss: 2.303370 [28832/60000]
loss: 2.302708 [32032/60000]
loss: 2.301954 [35232/60000]
loss: 2.303348 [38432/60000]
loss: 2.301405 [41632/60000]
loss: 2.305667 [44832/60000]
loss: 2.303343 [48032/60000]
loss: 2.303401 [51232/60000]
loss: 2.303736 [54432/60000]
loss: 2.302490 [57632/60000]
One epoch takes 33.89854598045349
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302598

Epoch 15
-------------------------------
loss: 2.301241 [   32/60000]
loss: 2.303568 [ 3232/60000]
loss: 2.301755 [ 6432/60000]
loss: 2.303954 [ 9632/60000]
loss: 2.301723 [12832/60000]
loss: 2.302430 [16032/60000]
loss: 2.302421 [19232/60000]
loss: 2.303782 [22432/60000]
loss: 2.302391 [25632/60000]
loss: 2.303189 [28832/60000]
loss: 2.305250 [32032/60000]
loss: 2.302119 [35232/60000]
loss: 2.304263 [38432/60000]
loss: 2.303191 [41632/60000]
loss: 2.302223 [44832/60000]
loss: 2.302263 [48032/60000]
loss: 2.303116 [51232/60000]
loss: 2.302548 [54432/60000]
loss: 2.302226 [57632/60000]
One epoch takes 34.47557210922241
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302588

Epoch 16
-------------------------------
loss: 2.302150 [   32/60000]
```

```
loss: 2.302095 [ 3232/60000]
loss: 2.302248 [ 6432/60000]
loss: 2.303350 [ 9632/60000]
loss: 2.302938 [12832/60000]
loss: 2.302687 [16032/60000]
loss: 2.297339 [19232/60000]
loss: 2.302158 [22432/60000]
loss: 2.301195 [25632/60000]
loss: 2.302453 [28832/60000]
loss: 2.301317 [32032/60000]
loss: 2.303012 [35232/60000]
loss: 2.301855 [38432/60000]
loss: 2.303631 [41632/60000]
loss: 2.302799 [44832/60000]
loss: 2.302648 [48032/60000]
loss: 2.303461 [51232/60000]
loss: 2.304523 [54432/60000]
loss: 2.302916 [57632/60000]
One epoch takes 34.21368741989136
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302590

Epoch 17
-------------------------------
loss: 2.303058 [   32/60000]
loss: 2.302370 [ 3232/60000]
loss: 2.301296 [ 6432/60000]
loss: 2.302496 [ 9632/60000]
loss: 2.300916 [12832/60000]
loss: 2.302174 [16032/60000]
loss: 2.302836 [19232/60000]
loss: 2.302543 [22432/60000]
loss: 2.301849 [25632/60000]
loss: 2.303438 [28832/60000]
loss: 2.303639 [32032/60000]
loss: 2.303268 [35232/60000]
loss: 2.304234 [38432/60000]
loss: 2.302433 [41632/60000]
loss: 2.304506 [44832/60000]
loss: 2.302112 [48032/60000]
loss: 2.303253 [51232/60000]
loss: 2.303604 [54432/60000]
loss: 2.302362 [57632/60000]
One epoch takes 34.17753291130066
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302587

Epoch 18
-------------------------------
loss: 2.302772 [   32/60000]
loss: 2.302945 [ 3232/60000]
```

```
loss: 2.302818 [ 6432/60000]
loss: 2.302856 [ 9632/60000]
loss: 2.304205 [12832/60000]
loss: 2.301339 [16032/60000]
loss: 2.301776 [19232/60000]
loss: 2.302753 [22432/60000]
loss: 2.301951 [25632/60000]
loss: 2.302040 [28832/60000]
loss: 2.301673 [32032/60000]
loss: 2.303577 [35232/60000]
loss: 2.301380 [38432/60000]
loss: 2.303691 [41632/60000]
loss: 2.301654 [44832/60000]
loss: 2.301485 [48032/60000]
loss: 2.302763 [51232/60000]
loss: 2.302883 [54432/60000]
loss: 2.302874 [57632/60000]
One epoch takes 34.227880239486694
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302590

Epoch 19
-------------------------------
loss: 2.303687 [   32/60000]
loss: 2.302030 [ 3232/60000]
loss: 2.302207 [ 6432/60000]
loss: 2.300397 [ 9632/60000]
loss: 2.300360 [12832/60000]
loss: 2.302026 [16032/60000]
loss: 2.302525 [19232/60000]
loss: 2.303919 [22432/60000]
loss: 2.302069 [25632/60000]
loss: 2.305605 [28832/60000]
loss: 2.302112 [32032/60000]
loss: 2.302807 [35232/60000]
loss: 2.302417 [38432/60000]
loss: 2.304413 [41632/60000]
loss: 2.303056 [44832/60000]
loss: 2.302280 [48032/60000]
loss: 2.302255 [51232/60000]
loss: 2.302548 [54432/60000]
loss: 2.301150 [57632/60000]
One epoch takes 34.5910108089447
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302593

Epoch 20
-------------------------------
loss: 2.302632 [   32/60000]
loss: 2.301889 [ 3232/60000]
loss: 2.303509 [ 6432/60000]
```

```
loss: 2.302067 [ 9632/60000]
loss: 2.302664 [12832/60000]
loss: 2.303119 [16032/60000]
loss: 2.301185 [19232/60000]
loss: 2.302928 [22432/60000]
loss: 2.303850 [25632/60000]
loss: 2.303701 [28832/60000]
loss: 2.302035 [32032/60000]
loss: 2.302542 [35232/60000]
loss: 2.303391 [38432/60000]
loss: 2.302254 [41632/60000]
loss: 2.305679 [44832/60000]
loss: 2.302518 [48032/60000]
loss: 2.302196 [51232/60000]
loss: 2.303823 [54432/60000]
loss: 2.302960 [57632/60000]
One epoch takes 35.54717946052551
Test Error:
 Accuracy: 10.0%, Avg loss: 2.302593

Test Error:
 Accuracy: 10.0%, Avg loss: 2.302593

Training done!
```

Problem B.3. ResNet

Now, keep the general structure in B.2 almost no change, but add some skip connection (resnet structure). And try to achieve 89% accuracy again.

Hint, to define a resnet, when you run the module in the `forward`, instead call it as:

```
y = net1(x)
```

Call it as:

```
y = net1(x) + x
```

The only requirement is that the input and output of `net1` should be same dimension. For example, you can `self.mid_block` in the SimpleCNN can add a skip connection. Try to find whereelse you can add this skip connection.

```
class ResCNN(nn.Module):
  def __init__(
    self,
```

```
    img_size: int = 32,
    in_channels: int = 1,
    num_classes: int = 10,
  ):
    super().__init__()
    base_channel = 16

    self.start_conv = nn.Sequential(*gen_double_conv(in_channels, base_channel))
    self.start_ds = nn.AvgPool2d(2)

    self.mid_blocks = nn.ModuleList()
    in_ch = base_channel
    out_ch = base_channel * 2
    for _ in range(6):
        self.mid_blocks.append(nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(),
            nn.AvgPool2d(2),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(),
            nn.ConvTranspose2d(out_ch, in_ch, 2, stride=2),
            nn.BatchNorm2d(in_ch),
            nn.ReLU()
        ))

    self.end_conv1 = nn.Sequential(*gen_double_conv(base_channel, base_channel *
2))
    self.end_ds1 = nn.AvgPool2d(2)
    self.end_conv2 = nn.Sequential(*gen_double_conv(base_channel * 2, base_channel
* 4))

    last_layer_size = img_size // (2 ** 3)
    self.end_ds2_linear = nn.Sequential(
        nn.AvgPool2d(2),
        nn.Flatten(),
        nn.Linear(last_layer_size * last_layer_size * base_channel * 4,
base_channel * 8),
        nn.ReLU(),
        nn.Linear(base_channel * 8, num_classes)
    )
  def forward(self, x):
    ######################
    # Your code goes here
    ######################
    x = self.start_ds(self.start_conv(x))
    for block in self.mid_blocks:
        identity = x
        out = block(x)
        x = out + identity
```

```
    x = self.end_ds2_linear(self.end_conv2(self.end_ds1(self.end_conv1(x))))
    return x

  def get_n_conv_linear_layers(self) -> int:
    ######################
    # Your code goes here
    ######################
    all_blocks = [self.start_conv, self.start_ds, *self.mid_blocks,
                  self.end_conv1, self.end_ds1, self.end_conv2,
self.end_ds2_linear]
    module_list = (nn.Conv2d, nn.ConvTranspose2d, nn.Linear)
    nlayer = 0
    for block in all_blocks:
        nlayer += len([mod for mod in block.modules() if isinstance(mod,
module_list)])
    return nlayer
```

```
#@title Create a model and corresponding optmizer

model = ResCNN()
model.to(device)
loss_fn = nn.CrossEntropyLoss()

epochs = 20   # Number of training epochs
optimizer = torch.optim.Adam(
  model.parameters(),    # All trainable parameters.
  lr=1e-4                # Learning rate
)

print(f'The model contains {model.get_n_conv_linear_layers()} conv or '
      f'linear layers')
```

```
The model contains 32 conv or linear layers
```

Running the training below.

Now, with resnet design, you will should reach 89% again.

```
#@title Run training

for t in range(epochs):
  print(f"Epoch {t+1}\n-------------------------------")
  start_time =time.time()
  _, _ = train_one_epoch(train_loader, model, loss_fn, optimizer, device=device)
  print(f'One epoch takes {time.time() - start_time}')
  _, _ = test_all_samples(test_loader, model, loss_fn, device=device)
```

```
test_all_samples(test_loader, model, loss_fn, device=device)

print("Training done!")
```

```
Epoch 1
-------------------------------
loss: 2.302576 [   32/60000]
loss: 1.879880 [ 3232/60000]
loss: 0.676733 [ 6432/60000]
loss: 0.279154 [ 9632/60000]
loss: 0.572534 [12832/60000]
loss: 0.713031 [16032/60000]
loss: 0.758974 [19232/60000]
loss: 0.352091 [22432/60000]
loss: 0.437863 [25632/60000]
loss: 0.449429 [28832/60000]
loss: 0.537505 [32032/60000]
loss: 0.329161 [35232/60000]
loss: 0.440145 [38432/60000]
loss: 0.665753 [41632/60000]
loss: 0.416098 [44832/60000]
loss: 0.383447 [48032/60000]
loss: 0.241573 [51232/60000]
loss: 0.346672 [54432/60000]
loss: 0.511998 [57632/60000]
One epoch takes 40.286428928375244
Test Error:
 Accuracy: 85.0%, Avg loss: 0.410524

Epoch 2
-------------------------------
loss: 0.387058 [   32/60000]
loss: 0.193786 [ 3232/60000]
loss: 0.354211 [ 6432/60000]
loss: 0.325126 [ 9632/60000]
loss: 0.279157 [12832/60000]
loss: 0.324918 [16032/60000]
loss: 0.284702 [19232/60000]
loss: 0.412460 [22432/60000]
loss: 0.370874 [25632/60000]
loss: 0.345568 [28832/60000]
loss: 0.296172 [32032/60000]
loss: 0.240046 [35232/60000]
loss: 0.277664 [38432/60000]
loss: 0.248565 [41632/60000]
loss: 0.438888 [44832/60000]
loss: 0.470165 [48032/60000]
loss: 0.222928 [51232/60000]
loss: 0.330368 [54432/60000]
```

```
loss: 0.281798 [57632/60000]
One epoch takes 38.92794895172119
Test Error:
 Accuracy: 87.0%, Avg loss: 0.357001

Epoch 3
-------------------------------
loss: 0.369888 [   32/60000]
loss: 0.321013 [ 3232/60000]
loss: 0.298609 [ 6432/60000]
loss: 0.354391 [ 9632/60000]
loss: 0.162646 [12832/60000]
loss: 0.371958 [16032/60000]
loss: 0.239617 [19232/60000]
loss: 0.229594 [22432/60000]
loss: 0.426445 [25632/60000]
loss: 0.172232 [28832/60000]
loss: 0.317605 [32032/60000]
loss: 0.291384 [35232/60000]
loss: 0.275296 [38432/60000]
loss: 0.066866 [41632/60000]
loss: 0.191649 [44832/60000]
loss: 0.319704 [48032/60000]
loss: 0.555975 [51232/60000]
loss: 0.485543 [54432/60000]
loss: 0.370988 [57632/60000]
One epoch takes 38.43545436859131
Test Error:
 Accuracy: 88.7%, Avg loss: 0.321109

Epoch 4
-------------------------------
loss: 0.326409 [   32/60000]
loss: 0.181529 [ 3232/60000]
loss: 0.252616 [ 6432/60000]
loss: 0.308008 [ 9632/60000]
loss: 0.299473 [12832/60000]
loss: 0.398400 [16032/60000]
loss: 0.323566 [19232/60000]
loss: 0.271992 [22432/60000]
loss: 0.127219 [25632/60000]
loss: 0.258092 [28832/60000]
loss: 0.225360 [32032/60000]
loss: 0.277808 [35232/60000]
loss: 0.785960 [38432/60000]
loss: 0.376720 [41632/60000]
loss: 0.278582 [44832/60000]
loss: 0.256500 [48032/60000]
loss: 0.190989 [51232/60000]
loss: 0.194323 [54432/60000]
loss: 0.381494 [57632/60000]
```

```
One epoch takes 38.243602991104126
Test Error:
 Accuracy: 89.2%, Avg loss: 0.300486

Epoch 5
-------------------------------
loss: 0.447798 [   32/60000]
loss: 0.170050 [ 3232/60000]
loss: 0.736225 [ 6432/60000]
loss: 0.259750 [ 9632/60000]
loss: 0.186553 [12832/60000]
loss: 0.315357 [16032/60000]
loss: 0.240027 [19232/60000]
loss: 0.241695 [22432/60000]
loss: 0.196319 [25632/60000]
loss: 0.274848 [28832/60000]
loss: 0.323924 [32032/60000]
loss: 0.242317 [35232/60000]
loss: 0.105366 [38432/60000]
loss: 0.265895 [41632/60000]
loss: 0.119465 [44832/60000]
loss: 0.094557 [48032/60000]
loss: 0.430046 [51232/60000]
loss: 0.204534 [54432/60000]
loss: 0.098627 [57632/60000]
One epoch takes 38.64936089515686
Test Error:
 Accuracy: 89.4%, Avg loss: 0.292542

Epoch 6
-------------------------------
loss: 0.184902 [   32/60000]
loss: 0.273193 [ 3232/60000]
loss: 0.072524 [ 6432/60000]
loss: 0.215064 [ 9632/60000]
loss: 0.052944 [12832/60000]
loss: 0.162439 [16032/60000]
loss: 0.363335 [19232/60000]
loss: 0.308276 [22432/60000]
loss: 0.257321 [25632/60000]
loss: 0.196382 [28832/60000]
loss: 0.045898 [32032/60000]
loss: 0.286786 [35232/60000]
loss: 0.206444 [38432/60000]
loss: 0.319061 [41632/60000]
loss: 0.288593 [44832/60000]
loss: 0.517131 [48032/60000]
loss: 0.320436 [51232/60000]
loss: 0.367009 [54432/60000]
loss: 0.160107 [57632/60000]
One epoch takes 38.55434823036194
```

```
Test Error:
 Accuracy: 89.9%, Avg loss: 0.272077

Epoch 7
-------------------------------
loss: 0.185801 [   32/60000]
loss: 0.111316 [ 3232/60000]
loss: 0.170387 [ 6432/60000]
loss: 0.247587 [ 9632/60000]
loss: 0.249227 [12832/60000]
loss: 0.168499 [16032/60000]
loss: 0.102632 [19232/60000]
loss: 0.225619 [22432/60000]
loss: 0.381634 [25632/60000]
loss: 0.081675 [28832/60000]
loss: 0.475979 [32032/60000]
loss: 0.132458 [35232/60000]
loss: 0.298947 [38432/60000]
loss: 0.068656 [41632/60000]
loss: 0.089927 [44832/60000]
loss: 0.138021 [48032/60000]
loss: 0.230856 [51232/60000]
loss: 0.289062 [54432/60000]
loss: 0.309822 [57632/60000]
One epoch takes 38.995328187942505
Test Error:
 Accuracy: 90.1%, Avg loss: 0.275071

Epoch 8
-------------------------------
loss: 0.117346 [   32/60000]
loss: 0.311382 [ 3232/60000]
loss: 0.131752 [ 6432/60000]
loss: 0.294970 [ 9632/60000]
loss: 0.240446 [12832/60000]
loss: 0.178309 [16032/60000]
loss: 0.333263 [19232/60000]
loss: 0.082399 [22432/60000]
loss: 0.090035 [25632/60000]
loss: 0.365583 [28832/60000]
loss: 0.208437 [32032/60000]
loss: 0.287551 [35232/60000]
loss: 0.163464 [38432/60000]
loss: 0.205018 [41632/60000]
loss: 0.249266 [44832/60000]
loss: 0.113666 [48032/60000]
loss: 0.248609 [51232/60000]
loss: 0.266743 [54432/60000]
loss: 0.065812 [57632/60000]
One epoch takes 38.856069803237915
Test Error:
```

```
 Accuracy: 91.1%, Avg loss: 0.258449

Epoch 9
-------------------------------
loss: 0.368835 [   32/60000]
loss: 0.185376 [ 3232/60000]
loss: 0.098838 [ 6432/60000]
loss: 0.081133 [ 9632/60000]
loss: 0.169032 [12832/60000]
loss: 0.051861 [16032/60000]
loss: 0.057448 [19232/60000]
loss: 0.164309 [22432/60000]
loss: 0.323889 [25632/60000]
loss: 0.153914 [28832/60000]
loss: 0.294677 [32032/60000]
loss: 0.194580 [35232/60000]
loss: 0.158467 [38432/60000]
loss: 0.235944 [41632/60000]
loss: 0.115848 [44832/60000]
loss: 0.234421 [48032/60000]
loss: 0.234890 [51232/60000]
loss: 0.248120 [54432/60000]
loss: 0.189492 [57632/60000]
One epoch takes 38.47069001197815
Test Error:
 Accuracy: 90.5%, Avg loss: 0.265344

Epoch 10
-------------------------------
loss: 0.111264 [   32/60000]
loss: 0.063982 [ 3232/60000]
loss: 0.245581 [ 6432/60000]
loss: 0.301783 [ 9632/60000]
loss: 0.371512 [12832/60000]
loss: 0.236100 [16032/60000]
loss: 0.139215 [19232/60000]
loss: 0.270188 [22432/60000]
loss: 0.137593 [25632/60000]
loss: 0.249083 [28832/60000]
loss: 0.148477 [32032/60000]
loss: 0.069563 [35232/60000]
loss: 0.320823 [38432/60000]
loss: 0.096328 [41632/60000]
loss: 0.206370 [44832/60000]
loss: 0.460481 [48032/60000]
loss: 0.108141 [51232/60000]
loss: 0.289773 [54432/60000]
loss: 0.022405 [57632/60000]
One epoch takes 38.53307747840881
Test Error:
 Accuracy: 90.7%, Avg loss: 0.260948
```

```
Epoch 11
-------------------------------
loss: 0.072224 [   32/60000]
loss: 0.083972 [ 3232/60000]
loss: 0.093337 [ 6432/60000]
loss: 0.111139 [ 9632/60000]
loss: 0.064867 [12832/60000]
loss: 0.061868 [16032/60000]
loss: 0.202572 [19232/60000]
loss: 0.106809 [22432/60000]
loss: 0.206293 [25632/60000]
loss: 0.106199 [28832/60000]
loss: 0.176656 [32032/60000]
loss: 0.392270 [35232/60000]
loss: 0.091210 [38432/60000]
loss: 0.064258 [41632/60000]
loss: 0.196043 [44832/60000]
loss: 0.103166 [48032/60000]
loss: 0.192224 [51232/60000]
loss: 0.138343 [54432/60000]
loss: 0.195617 [57632/60000]
One epoch takes 38.32692575454712
Test Error:
 Accuracy: 90.7%, Avg loss: 0.279950

Epoch 12
-------------------------------
loss: 0.182403 [   32/60000]
loss: 0.289051 [ 3232/60000]
loss: 0.063505 [ 6432/60000]
loss: 0.340229 [ 9632/60000]
loss: 0.144874 [12832/60000]
loss: 0.071269 [16032/60000]
loss: 0.296780 [19232/60000]
loss: 0.102901 [22432/60000]
loss: 0.128734 [25632/60000]
loss: 0.051519 [28832/60000]
loss: 0.031728 [32032/60000]
loss: 0.142838 [35232/60000]
loss: 0.146098 [38432/60000]
loss: 0.323504 [41632/60000]
loss: 0.024570 [44832/60000]
loss: 0.145673 [48032/60000]
loss: 0.048318 [51232/60000]
loss: 0.107978 [54432/60000]
loss: 0.135828 [57632/60000]
One epoch takes 39.463372468948364
Test Error:
 Accuracy: 90.9%, Avg loss: 0.260451
```

```
Epoch 13
-------------------------------
loss: 0.196475 [   32/60000]
loss: 0.147787 [ 3232/60000]
loss: 0.057655 [ 6432/60000]
loss: 0.227095 [ 9632/60000]
loss: 0.082048 [12832/60000]
loss: 0.064401 [16032/60000]
loss: 0.065829 [19232/60000]
loss: 0.159946 [22432/60000]
loss: 0.231071 [25632/60000]
loss: 0.125372 [28832/60000]
loss: 0.062577 [32032/60000]
loss: 0.245029 [35232/60000]
loss: 0.191586 [38432/60000]
loss: 0.215470 [41632/60000]
loss: 0.019036 [44832/60000]
loss: 0.190817 [48032/60000]
loss: 0.144570 [51232/60000]
loss: 0.173030 [54432/60000]
loss: 0.195711 [57632/60000]
One epoch takes 38.29414987564087
Test Error:
 Accuracy: 90.5%, Avg loss: 0.281447

Epoch 14
-------------------------------
loss: 0.019553 [   32/60000]
loss: 0.037009 [ 3232/60000]
loss: 0.073957 [ 6432/60000]
loss: 0.036156 [ 9632/60000]
loss: 0.096198 [12832/60000]
loss: 0.116061 [16032/60000]
loss: 0.190938 [19232/60000]
loss: 0.026868 [22432/60000]
loss: 0.051345 [25632/60000]
loss: 0.308442 [28832/60000]
loss: 0.163582 [32032/60000]
loss: 0.142452 [35232/60000]
loss: 0.118187 [38432/60000]
loss: 0.402054 [41632/60000]
loss: 0.131165 [44832/60000]
loss: 0.260687 [48032/60000]
loss: 0.112173 [51232/60000]
loss: 0.247726 [54432/60000]
loss: 0.140644 [57632/60000]
One epoch takes 38.54285764694214
Test Error:
 Accuracy: 90.8%, Avg loss: 0.288098

Epoch 15
```

```
-------------------------------
loss: 0.169161 [   32/60000]
loss: 0.038851 [ 3232/60000]
loss: 0.120287 [ 6432/60000]
loss: 0.101818 [ 9632/60000]
loss: 0.123213 [12832/60000]
loss: 0.078516 [16032/60000]
loss: 0.173378 [19232/60000]
loss: 0.170900 [22432/60000]
loss: 0.093380 [25632/60000]
loss: 0.062620 [28832/60000]
loss: 0.080839 [32032/60000]
loss: 0.133243 [35232/60000]
loss: 0.306153 [38432/60000]
loss: 0.090226 [41632/60000]
loss: 0.042840 [44832/60000]
loss: 0.139833 [48032/60000]
loss: 0.127943 [51232/60000]
loss: 0.367797 [54432/60000]
loss: 0.169641 [57632/60000]
One epoch takes 38.46945858001709
Test Error:
 Accuracy: 91.1%, Avg loss: 0.255905

Epoch 16
-------------------------------
loss: 0.049657 [   32/60000]
loss: 0.013965 [ 3232/60000]
loss: 0.380212 [ 6432/60000]
loss: 0.059443 [ 9632/60000]
loss: 0.244282 [12832/60000]
loss: 0.089714 [16032/60000]
loss: 0.183161 [19232/60000]
loss: 0.210742 [22432/60000]
loss: 0.209317 [25632/60000]
loss: 0.196510 [28832/60000]
loss: 0.117246 [32032/60000]
loss: 0.070588 [35232/60000]
loss: 0.066875 [38432/60000]
loss: 0.091504 [41632/60000]
loss: 0.114566 [44832/60000]
loss: 0.062761 [48032/60000]
loss: 0.046567 [51232/60000]
loss: 0.073854 [54432/60000]
loss: 0.070181 [57632/60000]
One epoch takes 38.357627630233765
Test Error:
 Accuracy: 90.8%, Avg loss: 0.261701

Epoch 17
-------------------------------
```

```
loss: 0.029921 [   32/60000]
loss: 0.139748 [ 3232/60000]
loss: 0.092550 [ 6432/60000]
loss: 0.139396 [ 9632/60000]
loss: 0.059180 [12832/60000]
loss: 0.122942 [16032/60000]
loss: 0.012841 [19232/60000]
loss: 0.110313 [22432/60000]
loss: 0.047343 [25632/60000]
loss: 0.066432 [28832/60000]
loss: 0.043965 [32032/60000]
loss: 0.234798 [35232/60000]
loss: 0.187890 [38432/60000]
loss: 0.186070 [41632/60000]
loss: 0.008383 [44832/60000]
loss: 0.131414 [48032/60000]
loss: 0.138266 [51232/60000]
loss: 0.102570 [54432/60000]
loss: 0.110474 [57632/60000]
One epoch takes 38.651397943496704
Test Error:
 Accuracy: 91.3%, Avg loss: 0.272987

Epoch 18
-------------------------------
loss: 0.198144 [   32/60000]
loss: 0.041794 [ 3232/60000]
loss: 0.162168 [ 6432/60000]
loss: 0.087300 [ 9632/60000]
loss: 0.045945 [12832/60000]
loss: 0.074093 [16032/60000]
loss: 0.091525 [19232/60000]
loss: 0.030007 [22432/60000]
loss: 0.060270 [25632/60000]
loss: 0.087143 [28832/60000]
loss: 0.158603 [32032/60000]
loss: 0.057029 [35232/60000]
loss: 0.071028 [38432/60000]
loss: 0.072317 [41632/60000]
loss: 0.064695 [44832/60000]
loss: 0.087272 [48032/60000]
loss: 0.036130 [51232/60000]
loss: 0.119940 [54432/60000]
loss: 0.165871 [57632/60000]
One epoch takes 38.38172483444214
Test Error:
 Accuracy: 91.0%, Avg loss: 0.278428

Epoch 19
-------------------------------
loss: 0.023451 [   32/60000]
```

```
loss: 0.074458 [ 3232/60000]
loss: 0.207987 [ 6432/60000]
loss: 0.058441 [ 9632/60000]
loss: 0.157721 [12832/60000]
loss: 0.049589 [16032/60000]
loss: 0.038191 [19232/60000]
loss: 0.008414 [22432/60000]
loss: 0.030379 [25632/60000]
loss: 0.018547 [28832/60000]
loss: 0.014193 [32032/60000]
loss: 0.081297 [35232/60000]
loss: 0.080592 [38432/60000]
loss: 0.032356 [41632/60000]
loss: 0.033402 [44832/60000]
loss: 0.102054 [48032/60000]
loss: 0.147055 [51232/60000]
loss: 0.026286 [54432/60000]
loss: 0.078949 [57632/60000]
One epoch takes 38.76998972892761
Test Error:
 Accuracy: 90.8%, Avg loss: 0.291397

Epoch 20
-------------------------------
loss: 0.034904 [   32/60000]
loss: 0.106157 [ 3232/60000]
loss: 0.070740 [ 6432/60000]
loss: 0.164774 [ 9632/60000]
loss: 0.110703 [12832/60000]
loss: 0.142151 [16032/60000]
loss: 0.046527 [19232/60000]
loss: 0.075264 [22432/60000]
loss: 0.043689 [25632/60000]
loss: 0.124718 [28832/60000]
loss: 0.231704 [32032/60000]
loss: 0.029422 [35232/60000]
loss: 0.041614 [38432/60000]
loss: 0.105649 [41632/60000]
loss: 0.279895 [44832/60000]
loss: 0.141850 [48032/60000]
loss: 0.150411 [51232/60000]
loss: 0.109856 [54432/60000]
loss: 0.027549 [57632/60000]
One epoch takes 38.36458373069763
Test Error:
 Accuracy: 90.7%, Avg loss: 0.313427

Test Error:
 Accuracy: 90.7%, Avg loss: 0.313427

Training done!
```