Lab 7: Self-Attention

This lab covers the following topics:

- Gain insight into the self-attention operation using the sequential MNIST example from before.
- Gain insight into positional encodings

0 Initialization

Run the code cell below to download the MNIST digits dataset:

In [1]:

```
!wget -O MNIST.tar.gz https://activeeon-public.s3.eu-west-2.amazonaws.
!tar -zxvf MNIST.tar.gz

import torchvision
import torch
import torchvision.transforms as transforms
from torch import nn
import torch.nn.functional as F

from torch.utils.data import Subset

dataset = torchvision.datasets.MNIST('./', download=True , transform=t
train_indices = torch.arange(0, 10000)
train_dataset = Subset(dataset, train_indices)

dataset=torchvision.datasets.MNIST('./', download=True , transform=tra
test_indices = torch.arange(0, 10000)
test_dataset = Subset(dataset, test_indices)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=6
                                          shuffle=True, num_workers=0)

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=16,
                                          shuffle=False, num_workers=0
```

```
--2023-03-21 13:37:59--  https://activeeon-public.s3.eu-west-2.amazon
aws.com/datasets/MNIST.new.tar.gz (https://activeeon-public.s3.eu-wes
t-2.amazonaws.com/datasets/MNIST.new.tar.gz)
Resolving activeeon-public.s3.eu-west-2.amazonaws.com (activeeon-publ
ic.s3.eu-west-2.amazonaws.com)... 52.95.148.6
Connecting to activeeon-public.s3.eu-west-2.amazonaws.com (activeeon-
public.s3.eu-west-2.amazonaws.com)|52.95.148.6|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 34812527 (33M) [application/x-gzip]
Saving to: ‘MNIST.tar.gz’

MNIST.tar.gz        100%[===================>]  33.20M   717KB/s    i
n 47s

2023-03-21 13:38:46 (726 KB/s) - ‘MNIST.tar.gz’ saved [34812527/34812
527]

MNIST/
MNIST/raw/
MNIST/raw/train-labels-idx1-ubyte.gz
MNIST/raw/t10k-images-idx3-ubyte
MNIST/raw/train-images-idx3-ubyte
MNIST/raw/t10k-labels-idx1-ubyte.gz
MNIST/raw/train-images-idx3-ubyte.gz
MNIST/raw/t10k-images-idx3-ubyte.gz
MNIST/raw/train-labels-idx1-ubyte
MNIST/raw/t10k-labels-idx1-ubyte
MNIST/processed/
MNIST/processed/test.pt
MNIST/processed/training.pt

/home/ziruiqiu/anaconda3/envs/DL2/lib/python3.9/site-packages/tqdm/au
to.py:22: TqdmWarning: IProgress not found. Please update jupyter and
ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_inst
all.html (https://ipywidgets.readthedocs.io/en/stable/user_install.ht
ml)
  from .autonotebook import tqdm as notebook_tqdm
```

Exercise 1: Self-Attention without Positional Encoding

In this section, will implement a very simple model based on self-attention without positional encoding. The model you will implement will consider the input image as a sequence of 28 rows. You may use PyTorch's `nn.MultiheadAttention`

(https://pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html) for this part.
Implement a model with the following architecture:

- **Input**: Input image of shape `(batch_size, sequence_length, input_size)`, where $\text{sequence\_length} = \text{image\_height}$ and $\text{input\_size} = \text{image\_width}$.
- **Linear 1**: Linear layer which converts input of shape `(sequence_length*batch_size, input_size)` to input of shape `(sequence_length*batch_size, embed_dim)`, where `embed_dim` is the embedding dimension.
- **Attention 1**: `nn.MultiheadAttention` layer with 8 heads which takes an input of shape `(sequence_length, batch_size, embed_dim)` and outputs a tensor of shape `(sequence_length, batch_size, embed_dim)`.
- **ReLU**: ReLU activation layer.
- **Linear 2**: Linear layer which converts input of shape `(sequence_length*batch_size, embed_dim)` to input of shape `(sequence_length*batch_size, embed_dim)`.
- **ReLU**: ReLU activation layer.
- **Attention 2**: `nn.MultiheadAttention` layer with 8 heads which takes an input of shape `(sequence_length, batch_size, embed_dim)` and outputs a tensor of shape `(sequence_length, batch_size, embed_dim)`.
- **ReLU**: ReLU activation layer.
- **AvgPool**: Average along the sequence dimension from `(batch_size, sequence_length, embed_dim)` to `(batch_size, embed_dim)`
- **Linear 3**: Linear layer which takes an input of shape `(batch_size, embed_dim)` and outputs the class logits of shape `(batch_size, 10)`.

NOTE: Be cautious of correctly permuting and reshaping the input between layers. E.g. if `x` is of shape `(batch_size, sequence_length, input_size)`, note that `x.reshape(batch_size*sequence_length, -1) != x.permute(1,0,2).reshape(batch_size*sequence_length, -1)`. In this example, `x.reshape(batch_size*sequence_length, -1)` has `[batch0_seq0, batch0_seq1, ..., batch1_seq0, batch1_seq1, ...]` format, while `x.permute(1,0,2).reshape(batch_size*sequence_length, -1)` has

In [83]:
```
# Self-attention without positional encoding
torch.manual_seed(691)

# Define your model here
class myModel(nn.Module):
    def __init__(self, input_size, embed_dim, seq_length,
                 num_classes=10, num_heads=8):
        super(myModel, self).__init__()
        # TODO: Initialize myModel
        self.input_size = input_size
        self.embed_dim = embed_dim
        self.seq_length = seq_length
        self.num_classes = num_classes
        self.num_heads = num_heads

        self.linear1 = nn.Linear(input_size, embed_dim)
        self.attention = nn.MultiheadAttention(embed_dim, num_heads)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(embed_dim, embed_dim)
        self.avgpool = nn.AvgPool1d(kernel_size=seq_length)
        self.linear3 = nn.Linear(embed_dim, num_classes)



    def forward(self,x):
        # TODO: Implement myModel forward pass
        batch_size, sequence_length, input_size = x.shape # 64, 28, 28
        input=x.reshape(batch_size*sequence_length, -1) # 1792, 28

        l1_out=self.linear1(input) # 1792, 64
        l1_out=l1_out.reshape(batch_size,sequence_length, -1) # 64, 28
        l1_out=l1_out.permute(1,0,2) # 28, 64, 64

        a1_out, _=self.attention(l1_out, l1_out, l1_out)
        a1_out=a1_out.permute(1,0,2) # 64, 28, 64
        a1_out=a1_out.reshape(batch_size*sequence_length, -1) # 1792,

        relu1_out=self.relu(a1_out) # 1792, 64

        l2_out=self.linear2(relu1_out)

        relu2_out=self.relu(l2_out) # 1792, 64
        relu2_out=relu2_out.reshape(batch_size, sequence_length, -1) #
        relu2_out=relu2_out.permute(1,0,2) # 28, 64, 64

        a2_out, _=self.attention(relu2_out, relu2_out, relu2_out) # 17
        a2_out=a2_out.permute(1, 0, 2) # 64, 28, 64
        a2_out=a2_out.reshape(batch_size, sequence_length, -1) # 1792,
        a2_out=a2_out.permute(0, 2, 1) # 64, 64, 28

        relu3_out=self.relu(a2_out) # 64, 64, 28

        avgpool_out=self.avgpool(relu3_out).squeeze() # 64, 64
        l3_out=self.linear3(avgpool_out) # 64, 10
        return l3_out
```

Train and evaluate your model by running the cell below. Expect to see `60-80%` test accuracy.

In [84]:

```
# Same training code

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms


# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
sequence_length = 28
input_size = 28
hidden_size = 64
num_layers = 2
num_classes = 10
num_epochs = 8
learning_rate = 0.005

# Initialize model
model = myModel(input_size=input_size, embed_dim=hidden_size, seq_leng
model = model.to(device)
# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, sequence_length, input_size).to(de
        labels = labels.to(device)

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

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()

        optimizer.step()

        if (i+1) % 10 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.


# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, sequence_length, input_size).to(de
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.
```

```
Epoch [1/8], Step [10/157], Loss: 2.2329
Epoch [1/8], Step [20/157], Loss: 2.2536
Epoch [1/8], Step [30/157], Loss: 2.0334
Epoch [1/8], Step [40/157], Loss: 2.1703
Epoch [1/8], Step [50/157], Loss: 2.0529
Epoch [1/8], Step [60/157], Loss: 1.9635
Epoch [1/8], Step [70/157], Loss: 1.9589
Epoch [1/8], Step [80/157], Loss: 1.5411
Epoch [1/8], Step [90/157], Loss: 1.9601
Epoch [1/8], Step [100/157], Loss: 1.6902
Epoch [1/8], Step [110/157], Loss: 1.5693
Epoch [1/8], Step [120/157], Loss: 1.4755
Epoch [1/8], Step [130/157], Loss: 1.4865
Epoch [1/8], Step [140/157], Loss: 1.3998
Epoch [1/8], Step [150/157], Loss: 1.3154
Epoch [2/8], Step [10/157], Loss: 1.5555
Epoch [2/8], Step [20/157], Loss: 1.4212
Epoch [2/8], Step [30/157], Loss: 1.1543
Epoch [2/8], Step [40/157], Loss: 1.3650
Epoch [2/8], Step [50/157], Loss: 1.3075
Epoch [2/8], Step [60/157], Loss: 1.3857
Epoch [2/8], Step [70/157], Loss: 1.5852
Epoch [2/8], Step [80/157], Loss: 1.4893
Epoch [2/8], Step [90/157], Loss: 1.4679
Epoch [2/8], Step [100/157], Loss: 1.2682
Epoch [2/8], Step [110/157], Loss: 1.3742
Epoch [2/8], Step [120/157], Loss: 0.8979
Epoch [2/8], Step [130/157], Loss: 1.2355
Epoch [2/8], Step [140/157], Loss: 1.0357
Epoch [2/8], Step [150/157], Loss: 1.0891
Epoch [3/8], Step [10/157], Loss: 1.0933
Epoch [3/8], Step [20/157], Loss: 0.9731
Epoch [3/8], Step [30/157], Loss: 0.8808
Epoch [3/8], Step [40/157], Loss: 1.1650
Epoch [3/8], Step [50/157], Loss: 1.0106
Epoch [3/8], Step [60/157], Loss: 0.6729
Epoch [3/8], Step [70/157], Loss: 0.7851
Epoch [3/8], Step [80/157], Loss: 0.9127
Epoch [3/8], Step [90/157], Loss: 0.9715
Epoch [3/8], Step [100/157], Loss: 0.7178
Epoch [3/8], Step [110/157], Loss: 0.7671
Epoch [3/8], Step [120/157], Loss: 0.8968
Epoch [3/8], Step [130/157], Loss: 0.9808
Epoch [3/8], Step [140/157], Loss: 0.9363
Epoch [3/8], Step [150/157], Loss: 0.8001
Epoch [4/8], Step [10/157], Loss: 0.6939
Epoch [4/8], Step [20/157], Loss: 0.5057
Epoch [4/8], Step [30/157], Loss: 0.8507
Epoch [4/8], Step [40/157], Loss: 0.8800
Epoch [4/8], Step [50/157], Loss: 0.7528
Epoch [4/8], Step [60/157], Loss: 0.6637
Epoch [4/8], Step [70/157], Loss: 0.7703
Epoch [4/8], Step [80/157], Loss: 0.7926
Epoch [4/8], Step [90/157], Loss: 0.7631
Epoch [4/8], Step [100/157], Loss: 0.7017
Epoch [4/8], Step [110/157], Loss: 0.7269
Epoch [4/8], Step [120/157], Loss: 0.7160
Epoch [4/8], Step [130/157], Loss: 0.5749
Epoch [4/8], Step [140/157], Loss: 0.6693
Epoch [4/8], Step [150/157], Loss: 0.6612
Epoch [5/8], Step [10/157], Loss: 0.9460
Epoch [5/8], Step [20/157], Loss: 0.6572
Epoch [5/8], Step [30/157], Loss: 0.7643
Epoch [5/8], Step [40/157], Loss: 0.8802
Epoch [5/8], Step [50/157], Loss: 0.9343
Epoch [5/8], Step [60/157], Loss: 0.4237
Epoch [5/8], Step [70/157], Loss: 0.6545
Epoch [5/8], Step [80/157], Loss: 0.5541
Epoch [5/8], Step [90/157], Loss: 0.6303
Epoch [5/8], Step [100/157], Loss: 0.6406
Epoch [5/8], Step [110/157], Loss: 0.8309
Epoch [5/8], Step [120/157], Loss: 0.3618
Epoch [5/8], Step [130/157], Loss: 0.4042
Epoch [5/8], Step [140/157], Loss: 0.4704
```

```
Epoch [5/8], Step [150/157], Loss: 0.6671
Epoch [6/8], Step [10/157], Loss: 0.4238
Epoch [6/8], Step [20/157], Loss: 0.4956
Epoch [6/8], Step [30/157], Loss: 0.7377
Epoch [6/8], Step [40/157], Loss: 0.5700
Epoch [6/8], Step [50/157], Loss: 0.4547
Epoch [6/8], Step [60/157], Loss: 0.9616
Epoch [6/8], Step [70/157], Loss: 0.8709
Epoch [6/8], Step [80/157], Loss: 0.4768
Epoch [6/8], Step [90/157], Loss: 0.5356
Epoch [6/8], Step [100/157], Loss: 0.6317
Epoch [6/8], Step [110/157], Loss: 0.6358
Epoch [6/8], Step [120/157], Loss: 0.6565
Epoch [6/8], Step [130/157], Loss: 0.8560
Epoch [6/8], Step [140/157], Loss: 0.4965
Epoch [6/8], Step [150/157], Loss: 0.6847
Epoch [7/8], Step [10/157], Loss: 0.6178
Epoch [7/8], Step [20/157], Loss: 0.4543
Epoch [7/8], Step [30/157], Loss: 0.5569
Epoch [7/8], Step [40/157], Loss: 0.4994
Epoch [7/8], Step [50/157], Loss: 0.5710
Epoch [7/8], Step [60/157], Loss: 0.5431
Epoch [7/8], Step [70/157], Loss: 0.7153
Epoch [7/8], Step [80/157], Loss: 0.4249
Epoch [7/8], Step [90/157], Loss: 0.4682
Epoch [7/8], Step [100/157], Loss: 0.5195
Epoch [7/8], Step [110/157], Loss: 0.5253
Epoch [7/8], Step [120/157], Loss: 0.5488
Epoch [7/8], Step [130/157], Loss: 0.4088
Epoch [7/8], Step [140/157], Loss: 0.5550
Epoch [7/8], Step [150/157], Loss: 0.4822
Epoch [8/8], Step [10/157], Loss: 0.4190
Epoch [8/8], Step [20/157], Loss: 0.5732
Epoch [8/8], Step [30/157], Loss: 0.5904
Epoch [8/8], Step [40/157], Loss: 0.6656
Epoch [8/8], Step [50/157], Loss: 0.9752
Epoch [8/8], Step [60/157], Loss: 0.3643
Epoch [8/8], Step [70/157], Loss: 0.4897
Epoch [8/8], Step [80/157], Loss: 0.5713
Epoch [8/8], Step [90/157], Loss: 0.4094
Epoch [8/8], Step [100/157], Loss: 0.6001
Epoch [8/8], Step [110/157], Loss: 0.6457
Epoch [8/8], Step [120/157], Loss: 0.6060
Epoch [8/8], Step [130/157], Loss: 0.6569
Epoch [8/8], Step [140/157], Loss: 0.4777
Epoch [8/8], Step [150/157], Loss: 0.6663
Test Accuracy of the model on the 10000 test images: 77.54 %
```

Exercise 2: Self-Attention with Positional Encoding

Implement a similar model to exercise 1, except this time your embedded input should be added with the positional encoding. For the purpose of this lab, we will use a learned positional encoding, which will be a trainable embedding. Your positional encodings will be added to the initial transformation of the input.

- **Input**: Input image of shape `(batch_size, sequence_length, input_size)` , where $\text{sequence\_length} = \text{image\_height}$ and $\text{input\_size} = \text{image\_width}$.
- **Linear 1**: Linear layer which converts input of shape `(batch_size*sequence_length, input_size)` to input of shape `(batch_size*sequence_length, embed_dim)` , where `embed_dim` is the embedding dimension.
- **Add Positional Encoding**: Add a learnable positional encoding of shape `(sequence_length, batch_size, embed_dim)` to input of shape `(sequence_length, batch_size, embed_dim)` , where `pos_embed` is the positional embedding size. The output will be of shape `(sequence_length, batch_size, embed_dim)` .
- **Attention 1**: `nn.MultiheadAttention` layer with 8 heads which takes an input of shape `(sequence_length, batch_size, embed_dim)` and outputs a tensor of shape `(sequence_length, batch_size, embed_dim)` .

- **ReLU**: ReLU activation layer.
- **Linear 2**: Linear layer which converts input of shape `(sequence_length*batch_size, features_dim)` to input of shape `(sequence_length*batch_size, features_dim)` .
- **ReLU**: ReLU activation layer.
- **Attention 2**: `nn.MultiheadAttention` layer with 8 heads which takes an input of shape `(sequence_length, batch_size, features_dim)` and outputs a tensor of shape `(sequence_length, batch_size, features_dim)` .
- **ReLU**: ReLU activation layer.
- **AvgPool**: Average along the sequence dimension from `(batch_size, sequence_length, features_dim)` to `(batch_size, features_dim)`
- **Linear 3**: Linear layer which takes an input of shape `(batch_size, sequence_length*features_dim)` and outputs the class logits of shape `(batch_size, 10)` .

In [88]:
```
# Self-attention with positional encoding
torch.manual_seed(691)

# Define your model here
class myModel(nn.Module):
    def __init__(self, input_size, embed_dim, seq_length,
                 num_classes=10, num_heads=8):
        super(myModel, self).__init__()
        # TODO: Initialize myModel
        self.input_size = input_size
        self.embed_dim = embed_dim
        self.seq_length = seq_length
        self.num_classes = num_classes
        self.num_heads = num_heads

        self.positional_encoding = nn.Parameter(torch.rand(self.seq_le
        self.linear1 = nn.Linear(input_size, embed_dim)
        self.attention = nn.MultiheadAttention(embed_dim, num_heads)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(embed_dim, embed_dim)
        self.avgpool = nn.AvgPool1d(kernel_size=seq_length)
        self.linear3 = nn.Linear(embed_dim, num_classes)

    def forward(self,x):
        # TODO: Implement myModel forward pass
        batch_size, sequence_length, input_size = x.shape # 64, 28, 28
        for i in range(batch_size):
            x[i]=x[i]+self.positional_encoding
        input=x.reshape(batch_size*sequence_length, -1) # 1792, 28

        l1_out=self.linear1(input) # 1792, 64
        l1_out=l1_out.reshape(batch_size,sequence_length, -1) # 64, 28
        l1_out=l1_out.permute(1,0,2) # 28, 64, 64

        #pe_out = self.positional_encoding.unsqueeze(1).repeat(1, batc

        a1_out, _=self.attention(l1_out, l1_out, l1_out)
        a1_out=a1_out.permute(1,0,2) # 64, 28, 64
        a1_out=a1_out.reshape(batch_size*sequence_length, -1) # 1792,

        relu1_out=self.relu(a1_out) # 1792, 64

        l2_out=self.linear2(relu1_out)

        relu2_out=self.relu(l2_out) # 1792, 64
        relu2_out=relu2_out.reshape(batch_size, sequence_length, -1) #
        relu2_out=relu2_out.permute(1,0,2) # 28, 64, 64

        a2_out, _=self.attention(relu2_out, relu2_out, relu2_out) # 17
        a2_out=a2_out.permute(1, 0, 2) # 64, 28, 64
        a2_out=a2_out.reshape(batch_size, sequence_length, -1) # 1792,
        a2_out=a2_out.permute(0, 2, 1) # 64, 64, 28

        relu3_out=self.relu(a2_out) # 64, 64, 28

        avgpool_out=self.avgpool(relu3_out).squeeze() # 64, 64
        l3_out=self.linear3(avgpool_out) # 64, 10
        return l3_out
```

Use the same training code as the one from part 1 to train your model. You may copy the training loop here. Expect to see close to `~90+%` test accuracy.

In [90]:
```
# Same training code

import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms


# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Hyper-parameters
sequence_length = 28
input_size = 28
hidden_size = 64
num_layers = 2
num_classes = 10
num_epochs = 8
learning_rate = 0.005

# Initialize model
model = myModel(input_size=input_size, embed_dim=hidden_size, seq_leng
model = model.to(device)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
total_step = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, sequence_length, input_size).to(de
        labels = labels.to(device)

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

        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()

        optimizer.step()

        if (i+1) % 10 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                   .format(epoch+1, num_epochs, i+1, total_step, loss.


# Test the model
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.reshape(-1, sequence_length, input_size).to(de
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.
```

```
Epoch [1/8], Step [10/157], Loss: 2.3080
Epoch [1/8], Step [20/157], Loss: 2.2362
Epoch [1/8], Step [30/157], Loss: 2.1281
Epoch [1/8], Step [40/157], Loss: 2.0270
Epoch [1/8], Step [50/157], Loss: 2.0658
Epoch [1/8], Step [60/157], Loss: 1.9943
Epoch [1/8], Step [70/157], Loss: 1.8028
Epoch [1/8], Step [80/157], Loss: 1.7964
Epoch [1/8], Step [90/157], Loss: 1.6519
Epoch [1/8], Step [100/157], Loss: 1.6729
Epoch [1/8], Step [110/157], Loss: 1.5191
Epoch [1/8], Step [120/157], Loss: 1.1571
Epoch [1/8], Step [130/157], Loss: 1.1786
Epoch [1/8], Step [140/157], Loss: 1.3048
Epoch [1/8], Step [150/157], Loss: 1.1641
Epoch [2/8], Step [10/157], Loss: 1.3100
Epoch [2/8], Step [20/157], Loss: 1.1102
Epoch [2/8], Step [30/157], Loss: 0.8163
Epoch [2/8], Step [40/157], Loss: 0.7992
Epoch [2/8], Step [50/157], Loss: 0.7793
Epoch [2/8], Step [60/157], Loss: 0.7339
Epoch [2/8], Step [70/157], Loss: 0.8414
Epoch [2/8], Step [80/157], Loss: 0.7955
Epoch [2/8], Step [90/157], Loss: 0.6519
Epoch [2/8], Step [100/157], Loss: 0.8286
Epoch [2/8], Step [110/157], Loss: 0.4260
Epoch [2/8], Step [120/157], Loss: 0.5824
Epoch [2/8], Step [130/157], Loss: 0.7125
Epoch [2/8], Step [140/157], Loss: 0.5897
Epoch [2/8], Step [150/157], Loss: 0.6990
Epoch [3/8], Step [10/157], Loss: 0.4409
Epoch [3/8], Step [20/157], Loss: 0.5969
Epoch [3/8], Step [30/157], Loss: 0.3652
Epoch [3/8], Step [40/157], Loss: 0.5446
Epoch [3/8], Step [50/157], Loss: 0.5357
Epoch [3/8], Step [60/157], Loss: 0.2991
Epoch [3/8], Step [70/157], Loss: 0.5821
Epoch [3/8], Step [80/157], Loss: 0.4854
Epoch [3/8], Step [90/157], Loss: 0.3104
Epoch [3/8], Step [100/157], Loss: 0.2107
Epoch [3/8], Step [110/157], Loss: 0.5561
Epoch [3/8], Step [120/157], Loss: 0.5118
Epoch [3/8], Step [130/157], Loss: 0.5580
Epoch [3/8], Step [140/157], Loss: 0.2601
Epoch [3/8], Step [150/157], Loss: 0.1975
Epoch [4/8], Step [10/157], Loss: 0.2935
Epoch [4/8], Step [20/157], Loss: 0.3384
Epoch [4/8], Step [30/157], Loss: 0.4056
Epoch [4/8], Step [40/157], Loss: 0.3852
Epoch [4/8], Step [50/157], Loss: 0.4882
Epoch [4/8], Step [60/157], Loss: 0.3175
Epoch [4/8], Step [70/157], Loss: 0.1462
Epoch [4/8], Step [80/157], Loss: 0.1619
Epoch [4/8], Step [90/157], Loss: 0.2607
Epoch [4/8], Step [100/157], Loss: 0.1872
Epoch [4/8], Step [110/157], Loss: 0.5545
Epoch [4/8], Step [120/157], Loss: 0.2168
Epoch [4/8], Step [130/157], Loss: 0.4085
Epoch [4/8], Step [140/157], Loss: 0.4577
Epoch [4/8], Step [150/157], Loss: 0.5314
Epoch [5/8], Step [10/157], Loss: 0.0768
Epoch [5/8], Step [20/157], Loss: 0.2784
Epoch [5/8], Step [30/157], Loss: 0.2626
Epoch [5/8], Step [40/157], Loss: 0.2730
Epoch [5/8], Step [50/157], Loss: 0.1499
Epoch [5/8], Step [60/157], Loss: 0.2604
Epoch [5/8], Step [70/157], Loss: 0.1520
Epoch [5/8], Step [80/157], Loss: 0.1323
Epoch [5/8], Step [90/157], Loss: 0.1853
Epoch [5/8], Step [100/157], Loss: 0.1651
Epoch [5/8], Step [110/157], Loss: 0.1566
Epoch [5/8], Step [120/157], Loss: 0.3483
Epoch [5/8], Step [130/157], Loss: 0.0964
Epoch [5/8], Step [140/157], Loss: 0.1651
```

```
Epoch [5/8], Step [150/157], Loss: 0.2590
Epoch [6/8], Step [10/157], Loss: 0.2550
Epoch [6/8], Step [20/157], Loss: 0.4580
Epoch [6/8], Step [30/157], Loss: 0.1782
Epoch [6/8], Step [40/157], Loss: 0.1885
Epoch [6/8], Step [50/157], Loss: 0.2361
Epoch [6/8], Step [60/157], Loss: 0.2653
Epoch [6/8], Step [70/157], Loss: 0.3885
Epoch [6/8], Step [80/157], Loss: 0.1829
Epoch [6/8], Step [90/157], Loss: 0.1460
Epoch [6/8], Step [100/157], Loss: 0.2599
Epoch [6/8], Step [110/157], Loss: 0.1111
Epoch [6/8], Step [120/157], Loss: 0.2736
Epoch [6/8], Step [130/157], Loss: 0.2615
Epoch [6/8], Step [140/157], Loss: 0.1536
Epoch [6/8], Step [150/157], Loss: 0.2370
Epoch [7/8], Step [10/157], Loss: 0.0294
Epoch [7/8], Step [20/157], Loss: 0.3864
Epoch [7/8], Step [30/157], Loss: 0.1680
Epoch [7/8], Step [40/157], Loss: 0.2653
Epoch [7/8], Step [50/157], Loss: 0.1296
Epoch [7/8], Step [60/157], Loss: 0.1477
Epoch [7/8], Step [70/157], Loss: 0.1475
Epoch [7/8], Step [80/157], Loss: 0.1824
Epoch [7/8], Step [90/157], Loss: 0.0887
Epoch [7/8], Step [100/157], Loss: 0.1949
Epoch [7/8], Step [110/157], Loss: 0.1048
Epoch [7/8], Step [120/157], Loss: 0.3778
Epoch [7/8], Step [130/157], Loss: 0.0570
Epoch [7/8], Step [140/157], Loss: 0.1201
Epoch [7/8], Step [150/157], Loss: 0.1044
Epoch [8/8], Step [10/157], Loss: 0.2169
Epoch [8/8], Step [20/157], Loss: 0.1166
Epoch [8/8], Step [30/157], Loss: 0.0509
Epoch [8/8], Step [40/157], Loss: 0.1508
Epoch [8/8], Step [50/157], Loss: 0.1538
Epoch [8/8], Step [60/157], Loss: 0.3041
Epoch [8/8], Step [70/157], Loss: 0.0435
Epoch [8/8], Step [80/157], Loss: 0.0855
Epoch [8/8], Step [90/157], Loss: 0.1400
Epoch [8/8], Step [100/157], Loss: 0.1639
Epoch [8/8], Step [110/157], Loss: 0.1380
Epoch [8/8], Step [120/157], Loss: 0.1583
Epoch [8/8], Step [130/157], Loss: 0.0890
Epoch [8/8], Step [140/157], Loss: 0.1910
Epoch [8/8], Step [150/157], Loss: 0.1221
Test Accuracy of the model on the 10000 test images: 90.87 %
```