Self-Attention

In this lab, we will try to gain insight into the self-attention operation using the sequential MNIST example from before.

0 Initialization

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

In [1]:

```
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
from six.moves import urllib
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
urllib.request.install_opener(opener)
```

In [2]:

```
### Hotfix for very recent MNIST download issue https://github.com/pytorch/vision/issue
s/1938
from six.moves import urllib
opener = urllib.request.build_opener()
opener.addheaders = [('User-agent', 'Mozilla/5.0')]
urllib.request.install_opener(opener)
###

dataset = torchvision.datasets.MNIST('./', download=True, transform=transforms.Compose
([transforms.ToTensor()]), train=True)
train_indices = torch.arange(0, 10000)
train_dataset = Subset(dataset, train_indices)

dataset=torchvision.datasets.MNIST('./', download=True, transform=transforms.Compose([t
ransforms.ToTensor()]), train=False)
test_indices = torch.arange(0, 10000)
test_dataset = Subset(dataset, test_indices)


train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
                                          shuffle=True, num_workers=0)

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

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)](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, features_dim)` to `(batch_size, features_dim)`
- **Linear 3**: Linear layer which takes an input of shape `(batch_size, sequence_length*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 [118]:

```
class MultiHead_Attn(nn.Module):
    def __init__(self, embed_dim ,num_head ):
        super().__init__()

        self.embed_dim = embed_dim ##1024
        self.num_head = num_head ##8
        self.head_dim = self.embed_dim // self.num_head


        assert self.embed_dim%self.num_head == 0

        self.q = nn.Linear(self.embed_dim , self.embed_dim )
        self.k = nn.Linear(self.embed_dim , self.embed_dim )
        self.v = nn.Linear(self.embed_dim , self.embed_dim)

        self.f_linear = nn.Linear(self.embed_dim, self.embed_dim)
        self.dropout = nn.Dropout(.35)
        self.scale = torch.sqrt(torch.FloatTensor([self.head_dim])).to(device)

    def forward(self, x ):
        batch_size = x.shape[0]
        src_len    = x.shape[1]
#         print('src_len=',src_len)

        w_k = self.k(x)
        w_q = self.q(x)
        w_v = self.v(x)

        w_q = w_q.view(batch_size,-1,self.head_dim)
        w_k = w_k.view(batch_size,-1,self.head_dim)
        w_v = w_v.view(batch_size,-1,self.head_dim)

        energy = torch.matmul( w_k.permute(0,2,1) ,w_q )
        energy = energy/self.scale
        energy = torch.softmax(energy,-1)


        f_energy = torch.matmul( self.dropout(energy) , w_v.permute(0,2,1))
        f_energy = f_energy.permute(0, 2, 1)
        f_energy = f_energy.reshape(batch_size,-1)
        out = self.f_linear(f_energy)

        return out
```

In [119]:

```
# 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.attention1=MultiHead_Attn(embed_dim, 8)
        self.relu=nn.ReLU()
        self.linear2 = nn.Linear(embed_dim, embed_dim)
        self.attention2=MultiHead_Attn(embed_dim, 8)
        self.linear3 = nn.Linear(embed_dim*seq_length, 10)
        self.avgpool=nn.AvgPool2d((seq_length, 1), stride=(2, 1))

    def forward(self,x):
        # TODO: Implement myModel forward pass
        batch_size, sequence_length, input_size = x.shape
        input=x.reshape(batch_size*sequence_length, -1)
        l1_out=self.linear1(input)
        a1_out=self.attention1(l1_out)
        relu1_out=self.relu(a1_out)
#         print(type(relu1_out))
        l2_out=self.linear2(relu1_out)
#         print(l2_out.shape)
        relu2_out=self.relu(l2_out)
        a2_out=self.attention2(relu2_out)
#         print(a2_out.shape)
        relu3_out=self.relu(a2_out)
#         print(relu3_out.shape)
        relu3_out=relu3_out.reshape(batch_size,sequence_length, -1)
#         print(relu3_out.shape)
        avgpool_out=self.avgpool(relu3_out)
        avgpool_out=avgpool_out.reshape(batch_size, -1)
#         print(avgpool_out.shape)
        l3_out=self.linear2(avgpool_out)
        return l3_out
```

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

In [120]:

```
# 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
batch_size = 100
num_epochs = 8
learning_rate = 0.005

# Initialize model
model = myModel(input_size=input_size, embed_dim=hidden_size, seq_length=sequence_lengt
h)
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(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
#         print(labels.shape)
#         print(outputs.shape)
        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.item()))


# 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(device)
        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: {} %'.format(100 * corr
ect / total))
```

```
Epoch [1/8], Step [10/157], Loss: 2.5169
Epoch [1/8], Step [20/157], Loss: 2.3786
Epoch [1/8], Step [30/157], Loss: 2.3539
Epoch [1/8], Step [40/157], Loss: 2.3043
Epoch [1/8], Step [50/157], Loss: 2.3285
Epoch [1/8], Step [60/157], Loss: 2.2978
Epoch [1/8], Step [70/157], Loss: 2.0004
Epoch [1/8], Step [80/157], Loss: 1.9677
Epoch [1/8], Step [90/157], Loss: 2.1236
Epoch [1/8], Step [100/157], Loss: 1.7003
Epoch [1/8], Step [110/157], Loss: 1.9901
Epoch [1/8], Step [120/157], Loss: 1.6480
Epoch [1/8], Step [130/157], Loss: 1.8560
Epoch [1/8], Step [140/157], Loss: 1.8463
Epoch [1/8], Step [150/157], Loss: 1.6737
Epoch [2/8], Step [10/157], Loss: 1.3322
Epoch [2/8], Step [20/157], Loss: 1.6528
Epoch [2/8], Step [30/157], Loss: 1.6002
Epoch [2/8], Step [40/157], Loss: 1.7638
Epoch [2/8], Step [50/157], Loss: 1.5760
Epoch [2/8], Step [60/157], Loss: 1.5664
Epoch [2/8], Step [70/157], Loss: 1.4938
Epoch [2/8], Step [80/157], Loss: 1.5565
Epoch [2/8], Step [90/157], Loss: 1.5405
Epoch [2/8], Step [100/157], Loss: 1.5736
Epoch [2/8], Step [110/157], Loss: 1.5054
Epoch [2/8], Step [120/157], Loss: 1.2671
Epoch [2/8], Step [130/157], Loss: 1.4646
Epoch [2/8], Step [140/157], Loss: 1.2533
Epoch [2/8], Step [150/157], Loss: 1.5371
Epoch [3/8], Step [10/157], Loss: 1.4050
Epoch [3/8], Step [20/157], Loss: 1.0369
Epoch [3/8], Step [30/157], Loss: 1.2893
Epoch [3/8], Step [40/157], Loss: 1.4952
Epoch [3/8], Step [50/157], Loss: 1.5757
Epoch [3/8], Step [60/157], Loss: 1.5483
Epoch [3/8], Step [70/157], Loss: 1.2638
Epoch [3/8], Step [80/157], Loss: 1.1697
Epoch [3/8], Step [90/157], Loss: 1.2990
Epoch [3/8], Step [100/157], Loss: 0.8806
Epoch [3/8], Step [110/157], Loss: 0.9845
Epoch [3/8], Step [120/157], Loss: 1.0830
Epoch [3/8], Step [130/157], Loss: 1.1220
Epoch [3/8], Step [140/157], Loss: 1.1152
Epoch [3/8], Step [150/157], Loss: 1.1880
Epoch [4/8], Step [10/157], Loss: 1.3257
Epoch [4/8], Step [20/157], Loss: 1.0064
Epoch [4/8], Step [30/157], Loss: 1.1558
Epoch [4/8], Step [40/157], Loss: 1.3290
Epoch [4/8], Step [50/157], Loss: 1.1791
Epoch [4/8], Step [60/157], Loss: 1.0900
Epoch [4/8], Step [70/157], Loss: 1.0315
Epoch [4/8], Step [80/157], Loss: 1.1494
Epoch [4/8], Step [90/157], Loss: 1.3993
Epoch [4/8], Step [100/157], Loss: 1.3090
Epoch [4/8], Step [110/157], Loss: 1.0155
Epoch [4/8], Step [120/157], Loss: 1.0457
Epoch [4/8], Step [130/157], Loss: 0.9468
Epoch [4/8], Step [140/157], Loss: 1.3770
Epoch [4/8], Step [150/157], Loss: 1.0364
Epoch [5/8], Step [10/157], Loss: 1.3091
```

```
Epoch [5/8], Step [20/157], Loss: 1.1745
Epoch [5/8], Step [30/157], Loss: 1.3945
Epoch [5/8], Step [40/157], Loss: 1.1823
Epoch [5/8], Step [50/157], Loss: 1.2852
Epoch [5/8], Step [60/157], Loss: 1.2442
Epoch [5/8], Step [70/157], Loss: 0.9090
Epoch [5/8], Step [80/157], Loss: 0.8563
Epoch [5/8], Step [90/157], Loss: 0.9839
Epoch [5/8], Step [100/157], Loss: 0.8785
Epoch [5/8], Step [110/157], Loss: 0.8066
Epoch [5/8], Step [120/157], Loss: 1.5008
Epoch [5/8], Step [130/157], Loss: 0.9967
Epoch [5/8], Step [140/157], Loss: 1.0386
Epoch [5/8], Step [150/157], Loss: 1.0976
Epoch [6/8], Step [10/157], Loss: 0.9835
Epoch [6/8], Step [20/157], Loss: 1.0819
Epoch [6/8], Step [30/157], Loss: 0.9429
Epoch [6/8], Step [40/157], Loss: 1.1362
Epoch [6/8], Step [50/157], Loss: 1.0435
Epoch [6/8], Step [60/157], Loss: 1.0566
Epoch [6/8], Step [70/157], Loss: 0.9276
Epoch [6/8], Step [80/157], Loss: 0.8511
Epoch [6/8], Step [90/157], Loss: 0.8713
Epoch [6/8], Step [100/157], Loss: 0.9633
Epoch [6/8], Step [110/157], Loss: 1.0507
Epoch [6/8], Step [120/157], Loss: 0.9260
Epoch [6/8], Step [130/157], Loss: 0.9487
Epoch [6/8], Step [140/157], Loss: 1.0235
Epoch [6/8], Step [150/157], Loss: 0.9021
Epoch [7/8], Step [10/157], Loss: 0.9614
Epoch [7/8], Step [20/157], Loss: 0.9864
Epoch [7/8], Step [30/157], Loss: 0.8781
Epoch [7/8], Step [40/157], Loss: 1.0833
Epoch [7/8], Step [50/157], Loss: 0.9997
Epoch [7/8], Step [60/157], Loss: 0.8864
Epoch [7/8], Step [70/157], Loss: 0.9337
Epoch [7/8], Step [80/157], Loss: 0.9632
Epoch [7/8], Step [90/157], Loss: 0.9593
Epoch [7/8], Step [100/157], Loss: 0.7020
Epoch [7/8], Step [110/157], Loss: 1.0222
Epoch [7/8], Step [120/157], Loss: 1.0243
Epoch [7/8], Step [130/157], Loss: 1.0990
Epoch [7/8], Step [140/157], Loss: 0.6862
Epoch [7/8], Step [150/157], Loss: 0.9897
Epoch [8/8], Step [10/157], Loss: 0.9542
Epoch [8/8], Step [20/157], Loss: 0.9306
Epoch [8/8], Step [30/157], Loss: 0.8751
Epoch [8/8], Step [40/157], Loss: 0.9715
Epoch [8/8], Step [50/157], Loss: 0.7755
Epoch [8/8], Step [60/157], Loss: 0.8226
Epoch [8/8], Step [70/157], Loss: 0.8874
Epoch [8/8], Step [80/157], Loss: 0.8327
Epoch [8/8], Step [90/157], Loss: 0.7186
Epoch [8/8], Step [100/157], Loss: 0.8437
Epoch [8/8], Step [110/157], Loss: 1.1434
Epoch [8/8], Step [120/157], Loss: 0.9664
Epoch [8/8], Step [130/157], Loss: 0.7147
Epoch [8/8], Step [140/157], Loss: 0.9429
Epoch [8/8], Step [150/157], Loss: 0.7495
Test Accuracy of the model on the 10000 test images: 71.25 %
```

2 Self-Attention with Positional Encoding

Implement a similar model to part (1), except this time your embedded input should be concatenated 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 enconding 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, embed_dim)` to input of shape `(sequence_length*batch_size, embed_dim)` .

- **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)`

In [179]:

```
# Self-attention without positional encoding
torch.manual_seed(691)

# Define your model here
class myModel_pos(nn.Module):
    def __init__(self, input_size, embed_dim, seq_length,
                 num_classes=10, num_heads=8):
        super(myModel_pos, 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_length, self.input_
size))
        self.linear1 = nn.Linear(input_size, embed_dim)
        self.attention1=MultiHead_Attn(embed_dim, 8)
        self.relu=nn.ReLU()
        self.linear2 = nn.Linear(embed_dim, embed_dim)
        self.attention2=MultiHead_Attn(embed_dim, 8)
        self.linear3 = nn.Linear(embed_dim*seq_length, 10)
        self.avgpool=nn.AvgPool2d((seq_length, 1), stride=(2, 1))

    def forward(self,x):
        # TODO: Implement myModel forward pass
        batch_size, sequence_length, input_size = x.shape
        for i in range(batch_size):
            x[i]=x[i]+self.positional_encoding


        input=x.reshape(batch_size*sequence_length, -1)
        l1_out=self.linear1(input)
        a1_out=self.attention1(l1_out)
        relu1_out=self.relu(a1_out)
#         print(type(relu1_out))
        l2_out=self.linear2(relu1_out)
#         print(l2_out.shape)
        relu2_out=self.relu(l2_out)
        a2_out=self.attention2(relu2_out)
#         print(a2_out.shape)
        relu3_out=self.relu(a2_out)
#         print(relu3_out.shape)
        relu3_out=relu3_out.reshape(batch_size,sequence_length, -1)
#         print(relu3_out.shape)
        avgpool_out=self.avgpool(relu3_out)
        avgpool_out=avgpool_out.reshape(batch_size, -1)
#         print(avgpool_out.shape)
        l3_out=self.linear2(avgpool_out)
        return l3_out
```

In [180]:

```
p=nn.Parameter(torch.rand(28, 64))
print(p.shape)
```

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

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 [182]:

```
# 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
batch_size = 100
num_epochs = 15
learning_rate = 0.005

# Initialize model
model = myModel_pos(input_size=input_size, embed_dim=hidden_size, seq_length=sequence_l
ength)
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(device)
        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.item()))


# 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(device)
```

```
        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: {} %'.format(100 * corr
ect / total))
```

```
Epoch [1/15], Step [10/157], Loss: 2.3809
Epoch [1/15], Step [20/157], Loss: 2.3217
Epoch [1/15], Step [30/157], Loss: 2.3399
Epoch [1/15], Step [40/157], Loss: 2.3338
Epoch [1/15], Step [50/157], Loss: 2.3359
Epoch [1/15], Step [60/157], Loss: 2.3004
Epoch [1/15], Step [70/157], Loss: 2.3477
Epoch [1/15], Step [80/157], Loss: 2.3101
Epoch [1/15], Step [90/157], Loss: 2.3444
Epoch [1/15], Step [100/157], Loss: 2.3145
Epoch [1/15], Step [110/157], Loss: 2.3092
Epoch [1/15], Step [120/157], Loss: 2.3262
Epoch [1/15], Step [130/157], Loss: 2.3195
Epoch [1/15], Step [140/157], Loss: 2.2571
Epoch [1/15], Step [150/157], Loss: 2.0634
Epoch [2/15], Step [10/157], Loss: 2.1390
Epoch [2/15], Step [20/157], Loss: 1.8546
Epoch [2/15], Step [30/157], Loss: 1.9561
Epoch [2/15], Step [40/157], Loss: 1.5295
Epoch [2/15], Step [50/157], Loss: 1.6447
Epoch [2/15], Step [60/157], Loss: 1.6268
Epoch [2/15], Step [70/157], Loss: 1.4775
Epoch [2/15], Step [80/157], Loss: 1.6287
Epoch [2/15], Step [90/157], Loss: 1.5356
Epoch [2/15], Step [100/157], Loss: 1.2293
Epoch [2/15], Step [110/157], Loss: 1.3573
Epoch [2/15], Step [120/157], Loss: 1.2911
Epoch [2/15], Step [130/157], Loss: 1.2015
Epoch [2/15], Step [140/157], Loss: 1.2681
Epoch [2/15], Step [150/157], Loss: 1.2519
Epoch [3/15], Step [10/157], Loss: 1.2392
Epoch [3/15], Step [20/157], Loss: 1.1675
Epoch [3/15], Step [30/157], Loss: 1.2775
Epoch [3/15], Step [40/157], Loss: 1.1163
Epoch [3/15], Step [50/157], Loss: 1.1770
Epoch [3/15], Step [60/157], Loss: 1.2627
Epoch [3/15], Step [70/157], Loss: 0.9154
Epoch [3/15], Step [80/157], Loss: 0.9214
Epoch [3/15], Step [90/157], Loss: 1.0988
Epoch [3/15], Step [100/157], Loss: 0.7856
Epoch [3/15], Step [110/157], Loss: 0.7391
Epoch [3/15], Step [120/157], Loss: 1.0325
Epoch [3/15], Step [130/157], Loss: 0.9927
Epoch [3/15], Step [140/157], Loss: 0.8329
Epoch [3/15], Step [150/157], Loss: 0.7294
Epoch [4/15], Step [10/157], Loss: 1.0562
Epoch [4/15], Step [20/157], Loss: 0.7436
Epoch [4/15], Step [30/157], Loss: 0.6303
Epoch [4/15], Step [40/157], Loss: 0.5647
Epoch [4/15], Step [50/157], Loss: 0.7945
Epoch [4/15], Step [60/157], Loss: 0.7355
Epoch [4/15], Step [70/157], Loss: 0.6854
Epoch [4/15], Step [80/157], Loss: 0.7833
Epoch [4/15], Step [90/157], Loss: 0.6526
Epoch [4/15], Step [100/157], Loss: 0.8945
Epoch [4/15], Step [110/157], Loss: 0.5157
Epoch [4/15], Step [120/157], Loss: 0.6335
Epoch [4/15], Step [130/157], Loss: 0.6782
Epoch [4/15], Step [140/157], Loss: 0.6885
Epoch [4/15], Step [150/157], Loss: 0.7086
Epoch [5/15], Step [10/157], Loss: 0.6421
```

```
Epoch [5/15], Step [20/157], Loss: 0.6250
Epoch [5/15], Step [30/157], Loss: 0.5396
Epoch [5/15], Step [40/157], Loss: 0.7412
Epoch [5/15], Step [50/157], Loss: 0.7954
Epoch [5/15], Step [60/157], Loss: 0.8396
Epoch [5/15], Step [70/157], Loss: 0.7070
Epoch [5/15], Step [80/157], Loss: 0.6495
Epoch [5/15], Step [90/157], Loss: 0.3598
Epoch [5/15], Step [100/157], Loss: 0.5121
Epoch [5/15], Step [110/157], Loss: 0.3943
Epoch [5/15], Step [120/157], Loss: 0.3910
Epoch [5/15], Step [130/157], Loss: 0.4647
Epoch [5/15], Step [140/157], Loss: 0.3422
Epoch [5/15], Step [150/157], Loss: 0.5517
Epoch [6/15], Step [10/157], Loss: 0.4880
Epoch [6/15], Step [20/157], Loss: 0.5752
Epoch [6/15], Step [30/157], Loss: 0.5535
Epoch [6/15], Step [40/157], Loss: 0.4481
Epoch [6/15], Step [50/157], Loss: 0.7284
Epoch [6/15], Step [60/157], Loss: 0.5501
Epoch [6/15], Step [70/157], Loss: 0.4748
Epoch [6/15], Step [80/157], Loss: 0.4822
Epoch [6/15], Step [90/157], Loss: 0.4227
Epoch [6/15], Step [100/157], Loss: 0.7819
Epoch [6/15], Step [110/157], Loss: 0.5505
Epoch [6/15], Step [120/157], Loss: 0.4995
Epoch [6/15], Step [130/157], Loss: 0.5384
Epoch [6/15], Step [140/157], Loss: 0.4378
Epoch [6/15], Step [150/157], Loss: 0.5206
Epoch [7/15], Step [10/157], Loss: 0.3289
Epoch [7/15], Step [20/157], Loss: 0.6379
Epoch [7/15], Step [30/157], Loss: 0.3283
Epoch [7/15], Step [40/157], Loss: 0.4271
Epoch [7/15], Step [50/157], Loss: 0.4683
Epoch [7/15], Step [60/157], Loss: 0.5638
Epoch [7/15], Step [70/157], Loss: 0.4205
Epoch [7/15], Step [80/157], Loss: 0.6184
Epoch [7/15], Step [90/157], Loss: 0.4659
Epoch [7/15], Step [100/157], Loss: 0.6414
Epoch [7/15], Step [110/157], Loss: 0.3406
Epoch [7/15], Step [120/157], Loss: 0.4259
Epoch [7/15], Step [130/157], Loss: 0.3278
Epoch [7/15], Step [140/157], Loss: 0.4668
Epoch [7/15], Step [150/157], Loss: 0.5011
Epoch [8/15], Step [10/157], Loss: 0.5002
Epoch [8/15], Step [20/157], Loss: 0.4879
Epoch [8/15], Step [30/157], Loss: 0.6285
Epoch [8/15], Step [40/157], Loss: 0.6124
Epoch [8/15], Step [50/157], Loss: 0.4505
Epoch [8/15], Step [60/157], Loss: 0.5732
Epoch [8/15], Step [70/157], Loss: 0.3607
Epoch [8/15], Step [80/157], Loss: 0.3387
Epoch [8/15], Step [90/157], Loss: 0.6701
Epoch [8/15], Step [100/157], Loss: 0.6387
Epoch [8/15], Step [110/157], Loss: 0.5065
Epoch [8/15], Step [120/157], Loss: 0.3656
Epoch [8/15], Step [130/157], Loss: 0.8395
Epoch [8/15], Step [140/157], Loss: 0.4246
Epoch [8/15], Step [150/157], Loss: 0.5607
Epoch [9/15], Step [10/157], Loss: 0.3730
Epoch [9/15], Step [20/157], Loss: 0.5083
```

```
Epoch [9/15], Step [30/157], Loss: 0.3999
Epoch [9/15], Step [40/157], Loss: 0.4559
Epoch [9/15], Step [50/157], Loss: 0.6846
Epoch [9/15], Step [60/157], Loss: 0.6348
Epoch [9/15], Step [70/157], Loss: 0.6705
Epoch [9/15], Step [80/157], Loss: 0.3618
Epoch [9/15], Step [90/157], Loss: 0.3790
Epoch [9/15], Step [100/157], Loss: 0.3853
Epoch [9/15], Step [110/157], Loss: 0.7393
Epoch [9/15], Step [120/157], Loss: 0.3958
Epoch [9/15], Step [130/157], Loss: 0.4558
Epoch [9/15], Step [140/157], Loss: 0.3020
Epoch [9/15], Step [150/157], Loss: 0.7041
Epoch [10/15], Step [10/157], Loss: 0.3006
Epoch [10/15], Step [20/157], Loss: 0.2955
Epoch [10/15], Step [30/157], Loss: 0.5178
Epoch [10/15], Step [40/157], Loss: 0.4833
Epoch [10/15], Step [50/157], Loss: 1.2236
Epoch [10/15], Step [60/157], Loss: 0.6087
Epoch [10/15], Step [70/157], Loss: 0.7508
Epoch [10/15], Step [80/157], Loss: 0.5057
Epoch [10/15], Step [90/157], Loss: 0.5199
Epoch [10/15], Step [100/157], Loss: 0.5405
Epoch [10/15], Step [110/157], Loss: 0.6092
Epoch [10/15], Step [120/157], Loss: 0.3723
Epoch [10/15], Step [130/157], Loss: 0.4067
Epoch [10/15], Step [140/157], Loss: 0.2451
Epoch [10/15], Step [150/157], Loss: 0.5543
Epoch [11/15], Step [10/157], Loss: 0.5108
Epoch [11/15], Step [20/157], Loss: 0.2667
Epoch [11/15], Step [30/157], Loss: 0.5179
Epoch [11/15], Step [40/157], Loss: 0.3251
Epoch [11/15], Step [50/157], Loss: 0.1705
Epoch [11/15], Step [60/157], Loss: 0.5168
Epoch [11/15], Step [70/157], Loss: 0.3248
Epoch [11/15], Step [80/157], Loss: 0.2697
Epoch [11/15], Step [90/157], Loss: 0.4176
Epoch [11/15], Step [100/157], Loss: 0.4951
Epoch [11/15], Step [110/157], Loss: 0.3623
Epoch [11/15], Step [120/157], Loss: 0.3715
Epoch [11/15], Step [130/157], Loss: 0.1256
Epoch [11/15], Step [140/157], Loss: 0.3484
Epoch [11/15], Step [150/157], Loss: 0.4650
Epoch [12/15], Step [10/157], Loss: 0.3336
Epoch [12/15], Step [20/157], Loss: 0.5885
Epoch [12/15], Step [30/157], Loss: 0.3363
Epoch [12/15], Step [40/157], Loss: 0.3494
Epoch [12/15], Step [50/157], Loss: 0.7879
Epoch [12/15], Step [60/157], Loss: 0.4695
Epoch [12/15], Step [70/157], Loss: 0.4466
Epoch [12/15], Step [80/157], Loss: 0.3855
Epoch [12/15], Step [90/157], Loss: 0.2557
Epoch [12/15], Step [100/157], Loss: 0.5303
Epoch [12/15], Step [110/157], Loss: 0.6876
Epoch [12/15], Step [120/157], Loss: 0.3864
Epoch [12/15], Step [130/157], Loss: 0.3905
Epoch [12/15], Step [140/157], Loss: 0.2623
Epoch [12/15], Step [150/157], Loss: 0.2597
Epoch [13/15], Step [10/157], Loss: 0.3164
Epoch [13/15], Step [20/157], Loss: 0.3642
Epoch [13/15], Step [30/157], Loss: 0.3067
```

```
Epoch [13/15], Step [40/157], Loss: 0.4684
Epoch [13/15], Step [50/157], Loss: 0.4583
Epoch [13/15], Step [60/157], Loss: 0.3656
Epoch [13/15], Step [70/157], Loss: 0.4527
Epoch [13/15], Step [80/157], Loss: 0.4932
Epoch [13/15], Step [90/157], Loss: 0.4289
Epoch [13/15], Step [100/157], Loss: 0.4127
Epoch [13/15], Step [110/157], Loss: 0.5347
Epoch [13/15], Step [120/157], Loss: 0.5154
Epoch [13/15], Step [130/157], Loss: 0.4715
Epoch [13/15], Step [140/157], Loss: 0.5203
Epoch [13/15], Step [150/157], Loss: 0.6086
Epoch [14/15], Step [10/157], Loss: 0.4260
Epoch [14/15], Step [20/157], Loss: 0.3986
Epoch [14/15], Step [30/157], Loss: 0.3195
Epoch [14/15], Step [40/157], Loss: 0.8566
Epoch [14/15], Step [50/157], Loss: 0.3116
Epoch [14/15], Step [60/157], Loss: 0.4197
Epoch [14/15], Step [70/157], Loss: 0.8163
Epoch [14/15], Step [80/157], Loss: 0.4062
Epoch [14/15], Step [90/157], Loss: 0.6478
Epoch [14/15], Step [100/157], Loss: 0.4344
Epoch [14/15], Step [110/157], Loss: 0.4650
Epoch [14/15], Step [120/157], Loss: 0.4384
Epoch [14/15], Step [130/157], Loss: 0.5317
Epoch [14/15], Step [140/157], Loss: 0.3790
Epoch [14/15], Step [150/157], Loss: 0.4541
Epoch [15/15], Step [10/157], Loss: 0.3259
Epoch [15/15], Step [20/157], Loss: 0.6518
Epoch [15/15], Step [30/157], Loss: 0.5384
Epoch [15/15], Step [40/157], Loss: 0.3236
Epoch [15/15], Step [50/157], Loss: 0.3991
Epoch [15/15], Step [60/157], Loss: 0.4147
Epoch [15/15], Step [70/157], Loss: 0.3977
Epoch [15/15], Step [80/157], Loss: 0.4348
Epoch [15/15], Step [90/157], Loss: 0.3023
Epoch [15/15], Step [100/157], Loss: 0.7657
Epoch [15/15], Step [110/157], Loss: 0.3700
Epoch [15/15], Step [120/157], Loss: 0.2918
Epoch [15/15], Step [130/157], Loss: 0.2377
Epoch [15/15], Step [140/157], Loss: 0.3527
Epoch [15/15], Step [150/157], Loss: 0.3052
Test Accuracy of the model on the 10000 test images: 89.97 %
```

In [ ]:

```

```