CSC 561: Neural Networks and Deep Learning

Vectorized Backpropagation / Pytorch

Marco Alvarez

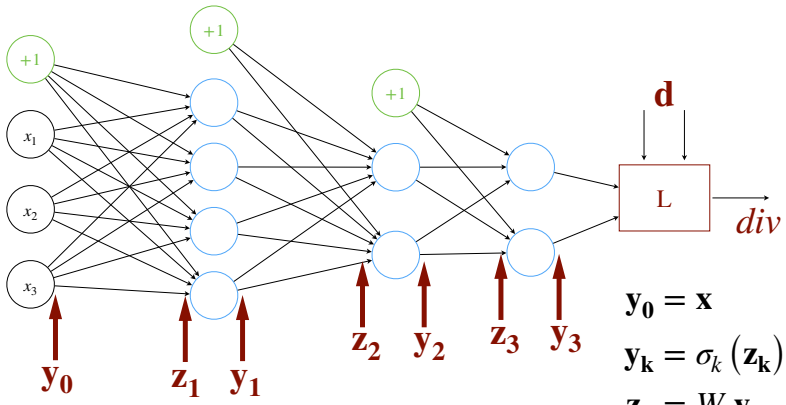
Department of Computer Science and Statistics
University of Rhode Island

Spring 2024

THINK BIG WE DO℠

Vectorized operations

Forward pass



$$\mathbf{y_0} = \mathbf{x}$$
$$\mathbf{y_k} = \sigma_k(\mathbf{z_k})$$
$$\mathbf{z_k} = W_k\mathbf{y_{k-1}} + \mathbf{b_k}$$

$$div = L\left(\sigma_3\left(W_3\sigma_2\left(W_2\sigma_1\left(W_1\mathbf{y_0} + \mathbf{b_1}\right) + \mathbf{b_2}\right) + \mathbf{b_3}\right), \mathbf{d}\right)$$

Vector calculus (derivatives)

- Scalar function of a vector argument
  - $y = f(\mathbf{z})$ $\quad \Delta y = \nabla_{\mathbf{z}} y \Delta \mathbf{z}$ (row vector)
  - transpose of the derivative is the gradient of y w.r.t. $\mathbf{z}$
- Vector function of a vector argument
  - $\mathbf{y} = f(\mathbf{z})$ $\quad \Delta \mathbf{y} = \nabla_{\mathbf{z}} y \Delta \mathbf{z}$ (matrix $m \times n$)
  - derivative is called the Jacobian of y w.r.t. $\mathbf{z}$

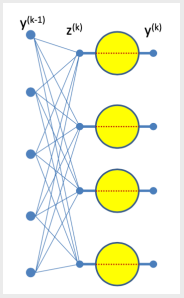
$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = f\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}\right)$$

Jacobian matrix

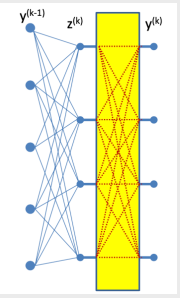
$$\mathbf{y} = f(\mathbf{z})$$

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix} = f\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}\right) \qquad J_{\mathbf{y}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \frac{\partial y_m}{\partial z_2} & \cdots & \frac{\partial y_m}{\partial z_n} \end{bmatrix}$$

Jacobian matrices for activations



$$J_{\mathbf{y}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & 0 & \cdots & 0 \\ 0 & \frac{\partial y_2}{\partial z_2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial y_m}{\partial z_n} \end{bmatrix}$$

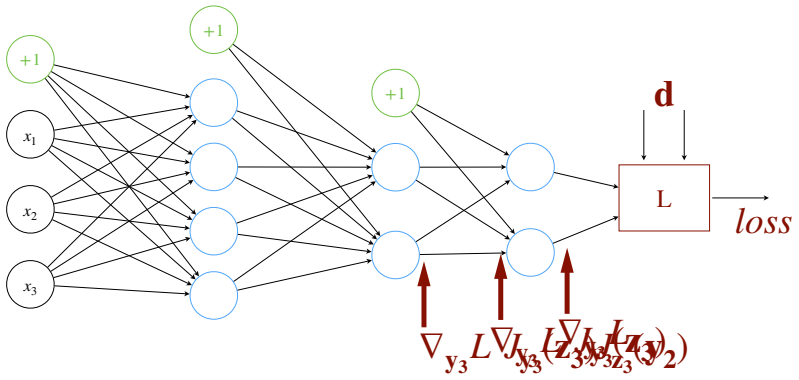


$$J_{\mathbf{y}}(\mathbf{z}) = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \frac{\partial y_m}{\partial z_2} & \cdots & \frac{\partial y_m}{\partial z_n} \end{bmatrix}$$

Vector calculus (chain rule)

- Nested functions
  - $\mathbf{y} = f\left(g(\mathbf{z})\right)$ $\qquad J_{\mathbf{y}}(\mathbf{z}) = J_f(g) J_g(\mathbf{z})$
  - **note**: the derivative of the outer function comes first
- Jacobians can be combined with gradients
  - e.g., scalar functions of vector inputs

Backward pass



$$\nabla_{\mathbf{y}_3} L \quad \nabla_{\mathbf{y}_3} L \, J_{\mathbf{y}_3}(\mathbf{z}_3) \quad \nabla_{\mathbf{y}_3} L \, J_{\mathbf{y}_3}(\mathbf{z}_3) \, \mathbf{y}_2)$$

$$\nabla_{W_3} L = \mathbf{y}_2 \nabla_{\mathbf{y}_3} L \, J_{\mathbf{y}_3}(\mathbf{z}_3) \qquad \nabla_{\mathbf{y}_3} L \, J_{\mathbf{y}_3}(\mathbf{z}_3) W_3$$

$$\nabla_{\mathbf{b}_3} L = \nabla_{\mathbf{y}_3} L \, J_{\mathbf{y}_3}(\mathbf{z}_3)$$

Numpy code for a 2-layer MLP

```
import numpy as np
from numpy.random import randn

N, D_in, H, D_out = 64, 1000, 100, 10
x, y = randn(N, D_in), randn(N, D_out)
w1, w2 = randn(D_in, H), randn(H, D_out)

for t in range(2000):
  h = 1 / (1 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  print(t, loss)

  grad_y_pred = 2.0 * (y_pred - y)
  grad_w2 = h.T.dot(grad_y_pred)
  grad_h = grad_y_pred.dot(w2.T)
  grad_w1 = x.T.dot(grad_h * h * (1 - h))

  w1 -= 1e-4 * grad_w1
  w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

Gradient descent

PyTorch Autograd explained

https://www.youtube.com/watch?v=MswxJw-8PvE

Show me the code

```
import torch

x = torch.tensor(-2., requires_grad=True)
y = torch.tensor(5., requires_grad=True)
z = torch.tensor(-4., requires_grad=True)

# forward pass
f = (x + y) * z

# backward pass
f.backward()

print(x.grad, y.grad, z.grad)
```

```
tensor(-4.) tensor(-4.) tensor(3.)
```


Show me the code (PyTorch)

```
import torch

x = torch.tensor([-1.,-2.], requires_grad=True)
w = torch.tensor([2.,-3.], requires_grad=True)
b = torch.tensor(-3., requires_grad=True)

# forward pass
f = 1 / (1 + torch.exp(-(w@x + b)))

# backward pass
f.backward()

print(w.grad, x.grad, b.grad)
```

```
tensor([-0.1966, -0.3932]) tensor([ 0.3932, -0.5898]) tensor(0.1966)
```

Equivalent computational graph

```
import torch

x = torch.tensor([-1.,-2.], requires_grad=True)
w = torch.tensor([2.,-3.], requires_grad=True)
b = torch.tensor(-3., requires_grad=True)

# forward pass
f = torch.sigmoid(w @ x + b)

# backward pass
f.backward()

print(w.grad, x.grad, b.grad)
```

```
tensor([-0.1966, -0.3932]) tensor([ 0.3932, -0.5898]) tensor(0.1966)
```

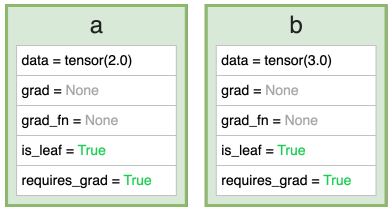

Computational graphs

- Pseudo-code for **forward** and **backward** passes

```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```


a = torch.tensor(2.0, requires_grad=True)

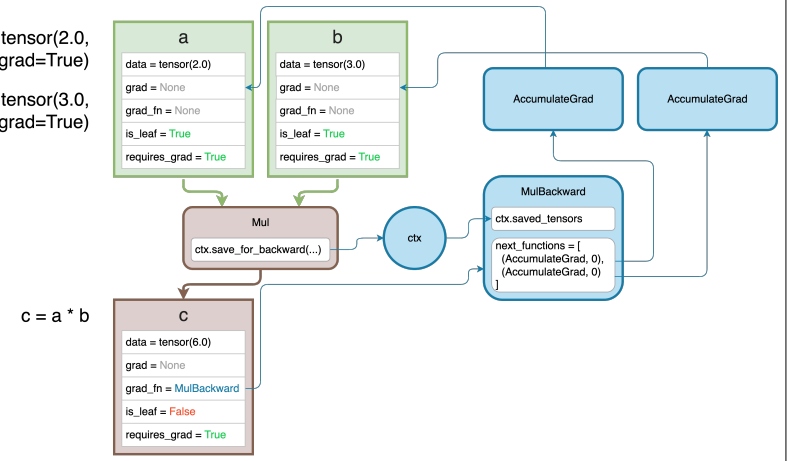
b = torch.tensor(3.0, requires_grad=True)



a = torch.tensor(2.0, requires_grad=True)

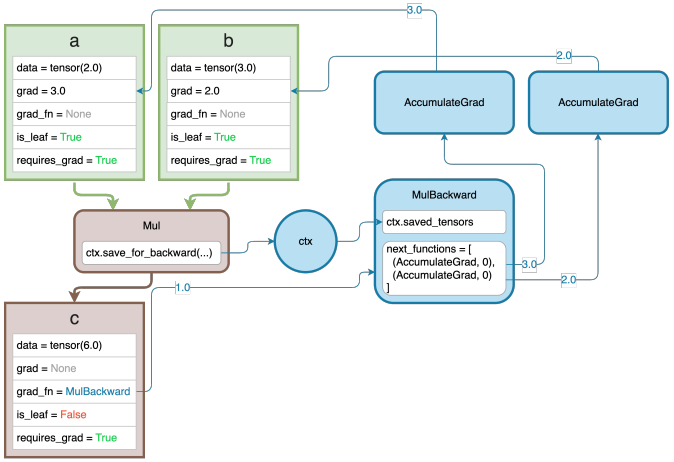
b = torch.tensor(3.0, requires_grad=True)

c = a * b



a = torch.tensor(2.0, requires_grad=True)

b = torch.tensor(3.0, requires_grad=True)

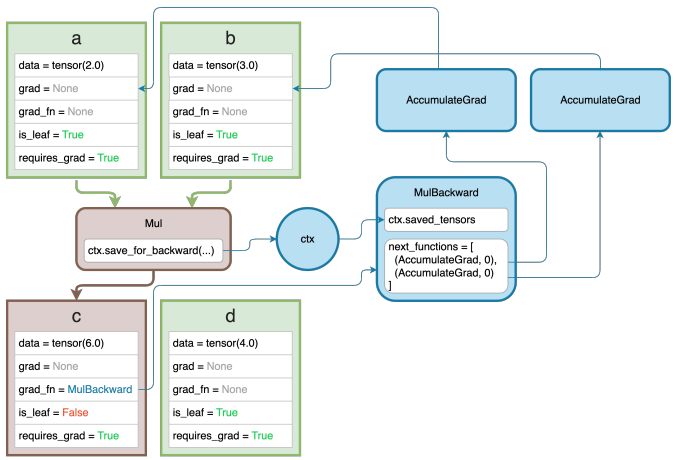


c = a * b

c.backward()

a = torch.tensor(2.0, requires_grad=True)

b = torch.tensor(3.0, requires_grad=True)

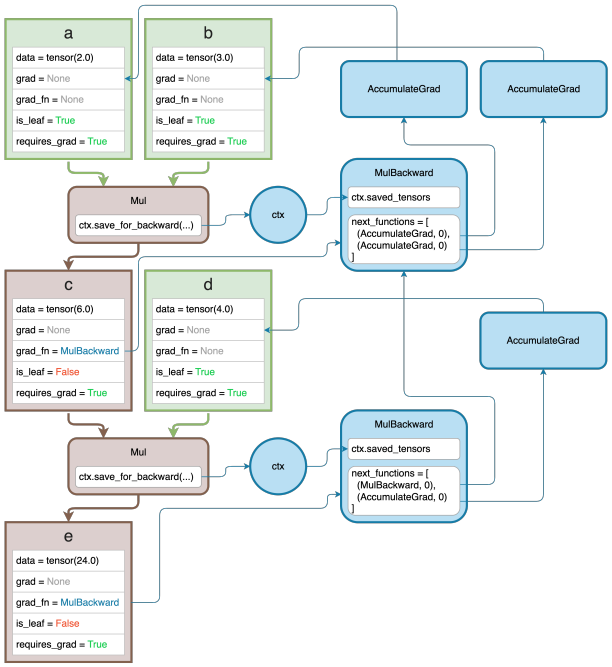


c = a * b

d = torch.tensor(4.0, requires_grad = True)

a = torch.tensor(2.0, requires_grad=True)

b = torch.tensor(3.0, requires_grad=True)



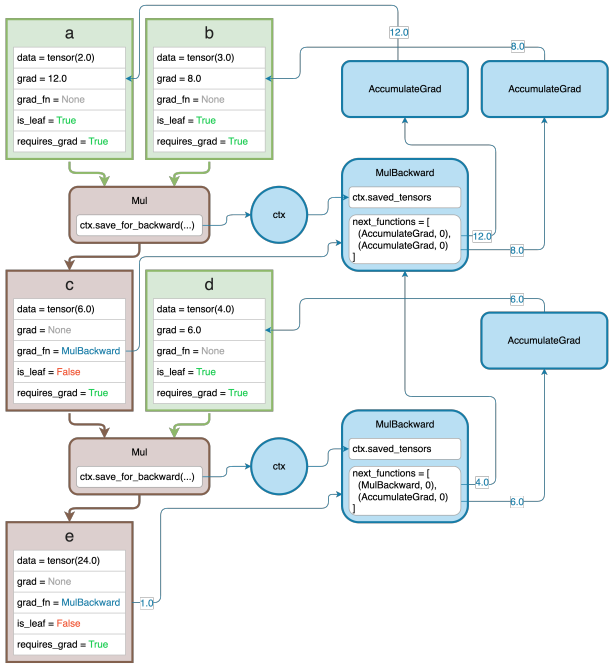
c = a * b

d = torch.tensor(4.0, requires_grad=True)

e = c * d

a = torch.tensor(2.0, requires_grad=True)

b = torch.tensor(3.0, requires_grad=True)



c = a * b

d = torch.tensor(4.0, requires_grad=True)

e = c * d

e.backward()

Defining new functions

```
class LegendrePolynomial3(torch.autograd.Function):
    '''
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    '''

    @staticmethod
    def forward(ctx, input):
        '''
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        '''
        ctx.save_for_backward(input)
        return 0.5 * (5 * input ** 3 - 3 * input)

    @staticmethod
    def backward(ctx, grad_output):
        '''
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        '''
        input, = ctx.saved_tensors
        return grad_output * 1.5 * (5 * input ** 2 - 1)
```

https://pytorch.org/tutorials/beginner/examples_autograd/two_layer_net_custom_function.html

Pytorch example

```
# define constants (hyperparameters)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
b_size = 64
l_rate = 0.0005
n_epochs = 5
```

```
# define the transformations
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

# load the MNIST data
train_data = datasets.MNIST('~/.cache/', train=True, download=True, transform=transform)
test_data = datasets.MNIST('~/.cache/', train=False, download=True, transform=transform)

# create data loaders
train_loader = DataLoader(train_data, batch_size=b_size, shuffle=True)
test_loader = DataLoader(test_data, batch_size=len(test_data), shuffle=False)
```



```
class MLP(torch.nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = Linear(28*28, 512)
        self.d1 = Dropout(0.4)
        self.fc2 = Linear(512, 512)
        self.d2 = Dropout(0.4)
        self.fc3 = Linear(512, 10)

    def forward(self, x):
        x = x.view(-1, 28*28)
        x = relu(self.fc1(x))
        x = self.d1(x)
        x = relu(self.fc2(x))
        x = self.d2(x)
        # note softmax is not used explicitly here
        x = self.fc3(x)
        return x
```

```python
def train(model, device, loader, opt, epoch, log_every=1):
    model.train()

    for batch_idx, (data, target) in enumerate(loader):
        data, target = data.to(device), target.to(device)
        opt.zero_grad()
        output = model(data)
        # cross entropy loss between input logits and target
        # note predictions are not a probability distribution
        loss = cross_entropy(output, target)

        loss.backward()
        opt.step()

        if batch_idx % log_every == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(loader.dataset),
                100. * batch_idx / len(loader), loss.item()))
```



```python
def test(model, device, loader):
    model.eval()
    test_loss = 0
    correct = 0

    with torch.no_grad():
        for data, target in loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += cross_entropy(output, target, reduction='sum').item()

            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(loader.dataset),
        100. * correct / len(loader.dataset)))
```



```python
model = MLP().to(device)
optimizer = AdamW(model.parameters(), lr=l_rate)

for epoch in range(n_epochs):
    train(model, device, train_loader, optimizer, epoch, log_every=100)
    test(model, device, test_loader)
```

```
Train Epoch: 0 [0/60000 (0%)]    Loss: 2.330892
Train Epoch: 0 [6400/60000 (11%)]    Loss: 0.301402
Train Epoch: 0 [12800/60000 (21%)]    Loss: 0.398661
Train Epoch: 0 [19200/60000 (32%)]    Loss: 0.162983
Train Epoch: 0 [25600/60000 (43%)]    Loss: 0.218468
Train Epoch: 0 [32000/60000 (53%)]    Loss: 0.168886
Train Epoch: 0 [38400/60000 (64%)]    Loss: 0.204717
Train Epoch: 0 [44800/60000 (75%)]    Loss: 0.111907
Train Epoch: 0 [51200/60000 (85%)]    Loss: 0.347752
Train Epoch: 0 [57600/60000 (96%)]    Loss: 0.232287

Test set: Average loss: 0.1187, Accuracy: 9632/10000 (96%)

Train Epoch: 1 [0/60000 (0%)]    Loss: 0.157370
Train Epoch: 1 [6400/60000 (11%)]    Loss: 0.125208
Train Epoch: 1 [12800/60000 (21%)]    Loss: 0.093008
Train Epoch: 1 [19200/60000 (32%)]    Loss: 0.061850
Train Epoch: 1 [25600/60000 (43%)]    Loss: 0.051277
Train Epoch: 1 [32000/60000 (53%)]    Loss: 0.132460
Train Epoch: 1 [38400/60000 (64%)]    Loss: 0.053607
Train Epoch: 1 [44800/60000 (75%)]    Loss: 0.041962
Train Epoch: 1 [51200/60000 (85%)]    Loss: 0.129311
Train Epoch: 1 [57600/60000 (96%)]    Loss: 0.136489

Test set: Average loss: 0.0928, Accuracy: 9697/10000 (97%)

Train Epoch: 2 [0/60000 (0%)]    Loss: 0.214178
Train Epoch: 2 [6400/60000 (11%)]    Loss: 0.094175
Train Epoch: 2 [12800/60000 (21%)]    Loss: 0.170777
Train Epoch: 2 [19200/60000 (32%)]    Loss: 0.083292
Train Epoch: 2 [25600/60000 (43%)]    Loss: 0.047731
Train Epoch: 2 [32000/60000 (53%)]    Loss: 0.073823
Train Epoch: 2 [38400/60000 (64%)]    Loss: 0.123365
Train Epoch: 2 [44800/60000 (75%)]    Loss: 0.078627
Train Epoch: 2 [51200/60000 (85%)]    Loss: 0.111702
Train Epoch: 2 [57600/60000 (96%)]    Loss: 0.093963

Test set: Average loss: 0.0772, Accuracy: 9761/10000 (98%)

Train Epoch: 3 [0/60000 (0%)]    Loss: 0.027528
Train Epoch: 3 [6400/60000 (11%)]    Loss: 0.069929
Train Epoch: 3 [12800/60000 (21%)]    Loss: 0.118869
Train Epoch: 3 [19200/60000 (32%)]    Loss: 0.097563
Train Epoch: 3 [25600/60000 (43%)]    Loss: 0.052691
Train Epoch: 3 [32000/60000 (53%)]    Loss: 0.027609
Train Epoch: 3 [38400/60000 (64%)]    Loss: 0.183863
Train Epoch: 3 [44800/60000 (75%)]    Loss: 0.154557
Train Epoch: 3 [51200/60000 (85%)]    Loss: 0.050854
Train Epoch: 3 [57600/60000 (96%)]    Loss: 0.052494

Test set: Average loss: 0.0694, Accuracy: 9797/10000 (98%)

Train Epoch: 4 [0/60000 (0%)]    Loss: 0.081912
Train Epoch: 4 [6400/60000 (11%)]    Loss: 0.045535
Train Epoch: 4 [12800/60000 (21%)]    Loss: 0.030686
Train Epoch: 4 [19200/60000 (32%)]    Loss: 0.054139
Train Epoch: 4 [25600/60000 (43%)]    Loss: 0.077830
Train Epoch: 4 [32000/60000 (53%)]    Loss: 0.019547
Train Epoch: 4 [38400/60000 (64%)]    Loss: 0.029897
Train Epoch: 4 [44800/60000 (75%)]    Loss: 0.104272
Train Epoch: 4 [51200/60000 (85%)]    Loss: 0.244031
Train Epoch: 4 [57600/60000 (96%)]    Loss: 0.063373

Test set: Average loss: 0.0683, Accuracy: 9795/10000 (98%)
```