

ml-tutorial (/github/Varal7/ml-tutorial/tree/master)

/ Part2.ipynb (/github/Varal7/ml-tutorial/tree/master/Part2.ipynb)

6.86x - Introduction to ML Packages (Part 2)

This tutorial is designed to provide a short introduction to deep learning with PyTorch.

You can start studying this tutorial as you work through unit 3 of the course. For more resources, check out the PyTorch tutorials (<https://pytorch.org/tutorials/>)! There are many more in-depth examples available there.

Source code for this notebook hosted at: <https://github.com/varal7/ml-tutorial> (<https://github.com/varal7/ml-tutorial>)

PyTorch

PyTorch (<https://pytorch.org>) is a flexible scientific computing package targetted towards gradient-based deep learning. Its low-level API closely follows NumPy (<https://www.numpy.org/>). However, there are a several key additions:

- GPU support!
- Automatic differentiation!
- Deep learning modules!
- Data loading!
- And other generally useful goodies.

If you don't have GPU enabled hardware, don't worry. Like NumPy, PyTorch runs pre-compiled, highly efficient C code to handle all intensive backend functions.

Go to pytorch.org to download the correct package for your computing environment.

In [2]:

```
# Start by importing torch  
import torch
```

Tensors

Tensors are PyTorch's equivalent of NumPy ndarrays.

In [3]:

```
# Construct a bunch of ones
some_ones = torch.ones(2, 2)
print(some_ones)

# Construct a bunch of zeros
some_zeros = torch.zeros(2, 2)
print(some_zeros)

# Construct some normally distributed values
some_normals = torch.randn(2, 2)
print(some_normals)

tensor([[1., 1.],
        [1., 1.]])
tensor([[0., 0.],
        [0., 0.]])
tensor([[ -2.4514, -0.6150],
        [ 0.9997,  0.4635]])
```

PyTorch tensors and NumPy ndarrays even share the same memory handles, so you can switch between the two types essentially for free:

In [4]:

```
torch_tensor = torch.randn(5, 5)
numpy_ndarray = torch_tensor.numpy()
back_to_torch = torch.from_numpy(numpy_ndarray)
```

Like NumPy, there are a zillion different operations you can do with tensors. Best thing to do is to go to <https://pytorch.org/docs/stable/tensors.html> (<https://pytorch.org/docs/stable/tensors.html>) if you know you want to do something to a tensor but don't know how!

We can cover a few major ones here:

In the Numpy tutorial, we have covered the basics of Numpy, numpy arrays, element-wise operations, matrices operations and generating random matrices. In this section, we'll cover indexing, slicing and broadcasting, which are useful concepts that will be reused in Pandas and PyTorch .

In [5]:

```
# Create two tensors
a = torch.randn(5, 5)
b = torch.randn(5, 5)
print(a)
print(b)

tensor([[ -2.4792,  0.7683, -0.8724, -1.0555, -1.3677],
        [ 0.2659,  0.3905,  0.4132,  1.0330,  1.3572],
        [-0.3723, -0.8348, -1.1457, -1.4766, -1.0380],
        [ 1.7401,  1.5151, -0.6725, -0.8755,  0.2736],
        [ 0.9129,  0.9838, -0.8510, -0.2960, -0.3731]])
tensor([[ 0.7255,  0.7353, -0.5352,  1.4629, -0.4881],
        [-1.2316,  0.7042, -1.3126,  0.8110, -1.3477],
        [-2.4669,  0.0770,  0.9740,  0.4297, -0.5245],
        [-1.0458, -1.2261,  0.6324,  0.8264, -1.3746],
        [-2.2290,  0.1202, -0.4826, -1.9797, -0.0879]])
```

In [7]:

```
# Indexing by i,j
another_tensor = a[2, 2]
print(another_tensor)

# The above returns a tensor type! To get the python value:
python_value = a[2, 2].item()
print(python_value)

tensor(-1.1457)
-1.1456899642944336
```

In [8]:

```
# Getting a whole row or column or range
first_row = a[0, :]
first_column = a[:, 0]
combo = a[2:4, 2:4]
print(combo)

tensor([[ -1.1457, -1.4766],
        [-0.6725, -0.8755]])
```

In [9]:

```
# Addition
c = a + b

# Elementwise multiplication: c_ij = a_ij * b_ij
c = a * b

# Matrix multiplication: c_ik = a_ij * b_jk
c = a.mm(b)

# Matrix vector multiplication
c = a.matmul(b[:, 0])
```

```
In [11]: a = torch.randn(5, 5)
print(a.size())

vec = a[:, 0]
print(vec.size())

# Matrix multiple 5x5 * 5x5 --> 5x5
aa = a.mm(a)

# matrix vector 5x5 * 5 --> 5
v1 = a.matmul(vec)
print(v1)

vec_as_matrix = vec.view(5, 1)
v2 = a.mm(vec_as_matrix)
print(v2)

torch.Size([5, 5])
torch.Size([5])
tensor([ 3.8873,  2.8224,  0.5655, -1.8550,  3.2441])
tensor([[ 3.8873],
        [ 2.8224],
        [ 0.5655],
        [-1.8550],
        [ 3.2441]])
```

In-place operations exist to, generally denoted by a trailing '_' (e.g. `my_tensor.my_inplacefunction`).

```
In [12]: # Add one to all elements
a.add_(1)

# Divide all elements by 2
a.div_(2)

# Set all elements to 0
a.zero_()

Out[12]: tensor([[0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.]])
```

Manipulate dimensions...

```
In [13]: # Add a dummy dimension, e.g. (n, m) --> (n, m, 1)
a = torch.randn(10, 10)

# At the end
print(a.unsqueeze(-1).size())

# At the beginning
print(a.unsqueeze(0).size())

# In the middle
print(a.unsqueeze(1).size())

# What you give you can take away
print(a.unsqueeze(0).squeeze(0).size())

# View things differently, i.e. flat
print(a.view(100, 1).size())

# Or not flat
print(a.view(50, 2).size())

# Copy data across a new dummy dimension!
a = torch.randn(2)
a = a.unsqueeze(-1)
print(a)
print(a.expand(2, 3))
```

```
torch.Size([10, 10, 1])
torch.Size([1, 10, 10])
torch.Size([10, 1, 10])
torch.Size([10, 10])
torch.Size([100, 1])
torch.Size([50, 2])
tensor([[ -0.1561,
          0.1588]])
tensor([[ -0.1561, -0.1561, -0.1561],
        [ 0.1588,  0.1588,  0.1588]])
```

If you have a GPU...

```
In [14]: # Check if you have it
do_i_have_cuda = torch.cuda.is_available()

if do_i_have_cuda:
    print('Using fancy GPUs')
    # One way
    a = a.cuda()
    a = a.cpu()

    # Another way
    device = torch.device('cuda')
    a = a.to(device)

    device = torch.device('cpu')
    a = a.to(device)
else:
    print('CPU it is!')
```

CPU it is!

And many more!

A Quick Note about Batching

In most ML applications we do mini-batch stochastic gradient descent instead of pure stochastic gradient descent.

Mini-batch SGD is a step between full gradient descent and stochastic gradient descent by computing the average gradient over a small number of examples.

In a nutshell, given n examples:

- **Full GD:** dL/dw = average over all n examples. One step per n examples.
- **SGD:** dL/dw = point estimate over a single example. n steps per n examples.
- **Mini-batch SGD:** dL/dw = average over $m \ll n$ examples. n / m steps per n examples.

Advantages of mini-batch SGD include a more stable gradient estimate and computational efficiency on modern hardware (exploiting parallelism gives sub-linear to constant time complexity, especially on GPU).

In PyTorch, batched tensors are represented as just another dimension. Most of the deep learning modules assume batched tensors as input (even if the batch size is just 1).

```
In [15]: # Batched matrix multiply
a = torch.randn(10, 5, 5)
b = torch.randn(10, 5, 5)

# The same as for i in 1 ... 10, c_i = a[i].mm(b[i])
c = a.bmm(b)

print(c.size())
```

```
torch.Size([10, 5, 5])
```

Autograd: Automatic Differentiation!

Along with the flexible deep learning modules (to follow) this is the best part of using a package PyTorch.

What is autograd? It *automatically* computes *gradients*. All those complicated functions you might be using for your model need gradients for back-propagation. Autograd does this automatically! (Sorry, you still need to do this by hand for homework 4.)

Let's warmup.

```
In [16]: # A tensor that will remember gradients
x = torch.randn(1, requires_grad=True)
print(x)
```

```
tensor([-0.1662], requires_grad=True)
```

At first the 'grad' parameter is None:

```
In [17]: print(x.grad)
```

```
None
```

Let's do an operation. Take $y = e^x$.

```
In [18]: y = x.exp()
```

To run the gradient computing magic, call '.backward()' on a variable.

```
In [19]: y.backward()
```

For all dependent variables $\{x_1, \dots, x_n\}$ that were used to compute y , dy/dx_i is computed and stored in the $x_i.grad$ field.

Here $dy/dx = e^x = y$. Let's see!

In [21]:

```
print(x.grad, y)

tensor([0.8469]) tensor([0.8469], grad_fn=<ExpBackward>)
```

Important! Remember to zero gradients before subsequent calls to backwards.

In [22]:

```
# Compute another thingy with x.
z = x * 2
z.backward()

# Should be 2! But it will be 2 + e^x.
print(x.grad)

tensor([2.8469])
```

In [25]:

```
x_a = torch.randn(1, requires_grad=True)
x_b = torch.randn(1, requires_grad=True)
x = x_a * x_b
x1 = x ** 2
x2 = 1 / x1
x3 = x2.exp()
x4 = 1 + x3
x5 = x4.log()
x6 = x5 ** (1/3)
x6.backward()
print(x_a.grad)
print(x_b.grad)

x = torch.randn(1, requires_grad=True)
y = torch.tanh(x)
y.backward()
print(x.grad)

tensor([-2.4034])
tensor([-31.5575])
tensor([0.8933])
```

Also important! Under the hood PyTorch stores all the stuff required to compute gradients (call stack, cached values, etc). If you want to save a variable just to keep it around (say for logging or plotting) remember to call `.item()` to get the python value and free the PyTorch machinery memory.

You can stop auto-grad from running in the background by using the `torch.no_grad()` context manager.

```
with torch.no_grad():
    do_all_my_things()
```

Manual Neural Net + Autograd SGD Example (read this while studying unit 3)

Before we move on to the full PyTorch wrapper library, let's do a simple NN SGD example by hand.

We'll train a one hidden layer feed forward NN on a toy dataset.

```
In [26]: # Set our random seeds
import random
import numpy as np

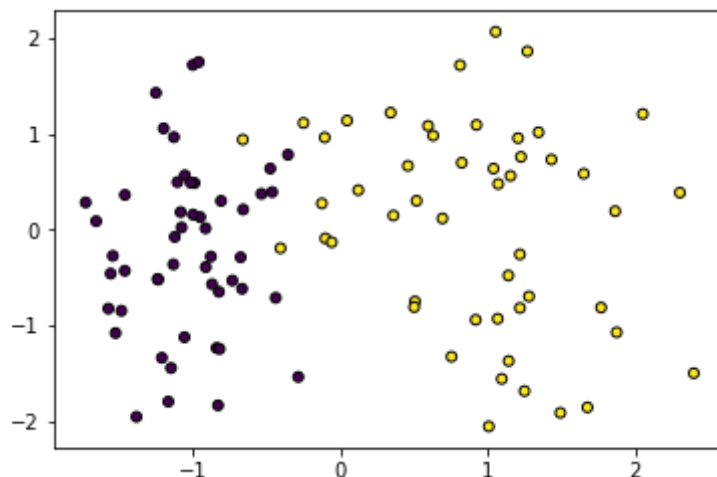
def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)

In [27]: # Get ourselves a simple dataset
from sklearn.datasets import make_classification
set_seed(7)
X, Y = make_classification(n_features=2, n_redundant=0, n_informative=2)
print('Number of examples: %d' % X.shape[0])
print('Number of features: %d' % X.shape[1])

# Take a peak
%matplotlib inline
import matplotlib
import matplotlib.pyplot as plt
plt.scatter(X[:, 0], X[:, 1], marker='o', c=Y, s=25, edgecolor='k')
plt.show()
```

Number of examples: 100

Number of features: 2



```
In [28]: # Convert data to PyTorch
X, Y = torch.from_numpy(X), torch.from_numpy(Y)

# Gotcha: "Expected object of scalar type Float but got scalar type Double"
# If you see this it's because numpy defaults to Doubles whereas pytorch defaults to Floats
X, Y = X.float(), Y.float()
```

We'll train a one layer neural net to classify this dataset. Let's define the parameter sizes:

```
In [29]: # Define dimensions
num_feats = 2
hidden_size = 100
num_outputs = 1

# Learning rate
eta = 0.1
num_steps = 1000
```

And now run a few steps of SGD!

```

In [30]: # Input to hidden weights
W1 = torch.randn(hidden_size, num_feats, requires_grad=True)
b1 = torch.zeros(hidden_size, requires_grad=True)

# Hidden to output
W2 = torch.randn(num_outputs, hidden_size, requires_grad=True)
b2 = torch.zeros(num_outputs, requires_grad=True)

# Group parameters
parameters = [W1, b1, W2, b2]

# Get random order
indices = torch.randperm(X.size(0))

# Keep running average losses for a learning curve?
avg_loss = []

# Run!
for step in range(num_steps):
    # Get example
    i = indices[step % indices.size(0)]
    x_i, y_i = X[i], Y[i]

    # Run example
    hidden = torch.relu(W1.matmul(x_i) + b1)
    y_hat = torch.sigmoid(W2.matmul(hidden) + b2)

    # Compute loss binary cross entropy:  $-(y_i * \log(y_{\text{hat}}) + (1 - y_i) * \log(1 - y_{\text{hat}}))$ 
    # Epsilon for numerical stability
    eps = 1e-6
    loss = -(y_i * (y_hat + eps).log() + (1 - y_i) * (1 - y_hat + eps).log())

    # Add to our running average learning curve. Don't forget .item()!
    if step == 0:
        avg_loss.append(loss.item())
    else:
        old_avg = avg_loss[-1]
        new_avg = (loss.item() + old_avg * len(avg_loss)) / (len(avg_loss) + 1)
        avg_loss.append(new_avg)

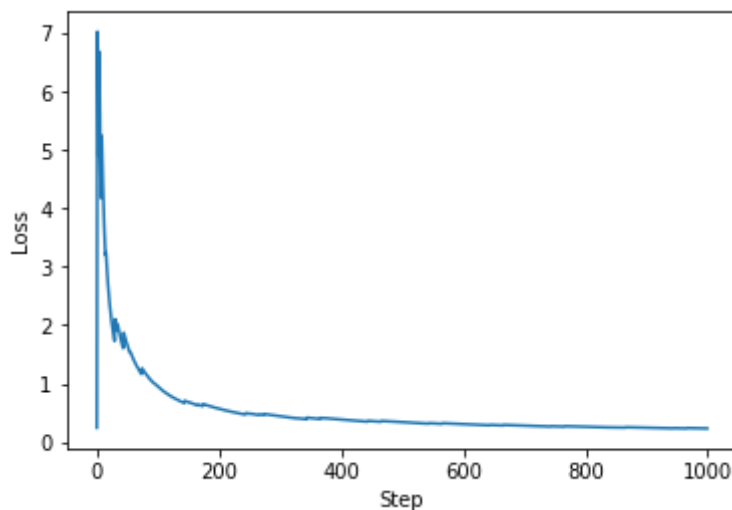
    # Zero out all previous gradients
    for param in parameters:
        # It might start out as None
        if param.grad is not None:
            # In place
            param.grad.zero_()

    # Backward pass
    loss.backward()

    # Update parameters
    for param in parameters:
        # In place!
        param.data = param.data - eta * param.grad

```

```
plt.plot(range(num_steps), avg_loss)
plt.ylabel('Loss')
plt.xlabel('Step')
plt.show()
```



torch.nn

The `nn` package is where all of the cool neural network stuff is. Layers, loss functions, etc.

Let's dive in.

Layers

Before we manually defined our linear layers. PyTorch has them for you as sub-classes of `nn.Module`.

In [31]:

```
import torch.nn as nn

# Linear layer: in_features, out_features
linear = nn.Linear(10, 10)
print(linear)

# Convolution layer: in_channels, out_channels, kernel_size, stride
conv = nn.Conv2d(1, 20, 5, 1)
print(conv)

# RNN: num_inputs, num_hidden, num_layers
rnn = nn.RNN(10, 10, 1)
print(rnn)

Linear(in_features=10, out_features=10, bias=True)
Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
RNN(10, 10)
```

```
In [33]: print(linear.weight)
print([k for k,v in conv.named_parameters()]])
```

Parameter containing:

```
tensor([[-0.2087,  0.0624,  0.0927,  0.2812,  0.0016,  0.2136, -0.1054,
          -0.0307,  0.1642],
        [-0.1235, -0.2677, -0.1926,  0.0560,  0.3015,  0.0175, -0.2549,
          0.1605, -0.0995],
        [-0.0427,  0.2353,  0.1162,  0.1936,  0.2839, -0.1041,  0.0458,
          0.3143, -0.2120],
        [ 0.3006,  0.2895,  0.0688, -0.2734, -0.0102, -0.1303,  0.0969,
          0.1761,  0.1016],
        [-0.2423, -0.2660,  0.0934, -0.0694,  0.1478,  0.3073,  0.0955,
          -0.0913,  0.1948],
        [ 0.0300,  0.2156, -0.3031, -0.0390, -0.1542,  0.2403,  0.1383,
          -0.2934, -0.0373],
        [ 0.2564, -0.0085, -0.0131, -0.2924,  0.2504,  0.2616, -0.2541,
          0.0153, -0.1809],
        [-0.2588,  0.0992, -0.0820,  0.1096,  0.1257,  0.2816,  0.1879,
          -0.2548,  0.2535],
        [-0.2687,  0.1933, -0.1927,  0.2537,  0.1788, -0.2183, -0.2614,
          -0.1446, -0.1795],
        [ 0.2228,  0.0777, -0.0397, -0.0215,  0.1316,  0.0324, -0.0392,
          0.2182,  0.0222]], requires_grad=True)
['weight', 'bias']
```

```
In [34]: # Make our own model!

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 input channel to 20 feature maps of 5x5 kernel. Stride 1.
        self.conv1 = nn.Conv2d(1, 20, 5, 1)

        # 20 input channels to 50 feature maps of 5x5 kernel. Stride 1.
        self.conv2 = nn.Conv2d(20, 50, 5, 1)

        # Full connected of final 4x4 image to 500 features
        self.fc1 = nn.Linear(4*4*50, 500)

        # From 500 to 10 classes
        self.fc2 = nn.Linear(500, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 2, 2)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 4*4*50)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

# Initialize it
model = Net()
```

A note on convolution sizes:

Running a kernel over the image reduces the image height/length by $\text{kernel_size} - 1$.

Running a max pooling over the image reduces the image height/length by a factor of the kernel size.

So starting from a 28 x 28 image:

- Run 5x5 conv --> 24 x 24
- Apply 2x2 max pool --> 12 x 12
- Run 5x5 conv --> 8 x 8
- Apply 2x2 max pool --> 4 x 4

Optimizers

PyTorch handles all the optimizing too. There are several algorithms you can learn about later. Here's SGD:

In [35]:

```
import torch.optim as optim

# Initialize with model parameters
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

Updating is now as easy as:

```
loss = loss_fn()
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Full train and test loops

Let's look at a full train loop now.

In [36]:

```

import tqdm
import torch.nn.functional as F

def train(model, train_loader, optimizer, epoch):
    # For things like dropout
    model.train()

    # Avg loss
    total_loss = 0

    # Iterate through dataset
    for data, target in tqdm.tqdm(train_loader):
        # Zero grad
        optimizer.zero_grad()

        # Forward pass
        output = model(data)

        # Negative log likelihood loss function
        loss = F.nll_loss(output, target)

        # Backward pass
        loss.backward()
        total_loss += loss.item()

        # Update
        optimizer.step()

    # Print average loss
    print("Train Epoch: {} \t Loss: {:.6f}".format(epoch, total_loss / ))

```

Testing loops are similar.

In [37]:

```

def test(model, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()
            pred = output.argmax(dim=1, keepdim=True) # get the index of the prediction
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)'
          '\n\t test_loss, correct, len(test_loader.dataset),\n\t 100. * correct / len(test_loader.dataset))')

```

MNIST

Just going to run mnist!

In [38]:

```

from torchvision import datasets, transforms

# See the torch DataLoader for more details.
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True, download=True,
        transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])),
    batch_size=32, shuffle=True)

test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False,
        transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))
        ])),
    batch_size=32, shuffle=True)

for epoch in range(1, 10 + 1):
    train(model, train_loader, optimizer, epoch)
    test(model, test_loader)

```

100%|██████████| 1875/1875 [00:57<00:00, 32.43it/s]

Train Epoch: 1 Loss: 0.314105

0%| | 3/1875 [00:00<01:03, 29.64it/s]

Test set: Average loss: 0.0967, Accuracy: 9701/10000 (97%)

100%|██████████| 1875/1875 [00:55<00:00, 33.76it/s]

Train Epoch: 2 Loss: 0.086927

0%| | 4/1875 [00:00<00:58, 31.88it/s]

Test set: Average loss: 0.0590, Accuracy: 9818/10000 (98%)

100%|██████████| 1875/1875 [00:55<00:00, 33.53it/s]

Train Epoch: 3 Loss: 0.061660

0%| | 4/1875 [00:00<00:58, 32.16it/s]

Test set: Average loss: 0.0493, Accuracy: 9839/10000 (98%)

100%|██████████| 1875/1875 [00:54<00:00, 34.44it/s]

Train Epoch: 4 Loss: 0.048638

0%| | 4/1875 [00:00<00:57, 32.47it/s]

Test set: Average loss: 0.0403, Accuracy: 9874/10000 (99%)

```
100%|██████████| 1875/1875 [00:54<00:00, 34.63it/s]
Train Epoch: 5    Loss: 0.040329
  0%|          | 4/1875 [00:00<00:55, 33.79it/s]
Test set: Average loss: 0.0337, Accuracy: 9885/10000 (99%)

100%|██████████| 1875/1875 [00:54<00:00, 34.14it/s]
Train Epoch: 6    Loss: 0.034306
  0%|          | 4/1875 [00:00<00:58, 31.85it/s]
Test set: Average loss: 0.0372, Accuracy: 9876/10000 (99%)

100%|██████████| 1875/1875 [00:53<00:00, 34.75it/s]
Train Epoch: 7    Loss: 0.029963
  0%|          | 4/1875 [00:00<00:56, 33.26it/s]
Test set: Average loss: 0.0337, Accuracy: 9889/10000 (99%)

100%|██████████| 1875/1875 [00:56<00:00, 33.23it/s]
Train Epoch: 8    Loss: 0.026079
  0%|          | 4/1875 [00:00<00:56, 33.39it/s]
Test set: Average loss: 0.0349, Accuracy: 9887/10000 (99%)

100%|██████████| 1875/1875 [00:53<00:00, 34.85it/s]
Train Epoch: 9    Loss: 0.022861
  0%|          | 4/1875 [00:00<00:56, 32.90it/s]
Test set: Average loss: 0.0374, Accuracy: 9877/10000 (99%)

100%|██████████| 1875/1875 [00:54<00:00, 34.36it/s]
Train Epoch: 10   Loss: 0.019654
Test set: Average loss: 0.0319, Accuracy: 9905/10000 (99%)
```

In []: