

# PyTorch Crash Course: 0 to MNIST in 1 Hour

Mark Chilenski  
(markchil)

October 23, 2021

You can download the code to follow along at:  
<https://github.com/markchil/pytorch-lecture>

About me:

- Graduated 2016 with PhD from Course 22
- Currently a research scientist working on a variety of machine learning applications

Goals for today:

- Cover the core classes/philosophy of PyTorch
- Give you enough vocab to confidently google stuff/read the docs

Prerequisites:

- Basic familiarity with Python

You can download the code to follow along at:

<https://github.com/markchil/pytorch-lecture>

# PyTorch Crash Course: 0 to MNIST in 1 Hour

- 1 Background: Machine Learning, Neural Networks, and PyTorch
- 2 PyTorch Fundamentals: The Tensor Class
- 3 Higher Abstractions: The `torch.nn` Module
- 4 Data Handling: The Dataset and DataLoader Classes
- 5 Putting It All Together to Solve MNIST: The “hello, world” of Machine Learning
- 6 Summary and Resources

# PyTorch Crash Course: 0 to MNIST in 1 Hour

- 1 Background: Machine Learning, Neural Networks, and PyTorch
- 2 PyTorch Fundamentals: The Tensor Class
- 3 Higher Abstractions: The `torch.nn` Module
- 4 Data Handling: The Dataset and DataLoader Classes
- 5 Putting It All Together to Solve MNIST: The “hello, world” of Machine Learning
- 6 Summary and Resources

# What Is PyTorch?

PyTorch is an open source Python **machine learning** package geared towards building **neural networks**, which provides the following key features:<sup>1</sup>

- Automatic differentiation
- GPU acceleration
- Many standard neural network building blocks
  - And nice abstractions which make building novel/non-standard ones easy
- Rich ecosystem of pre-trained models and open source building blocks

---

<sup>1</sup>We'll unpack some of this jargon on the following slides

# What Can You Do With PyTorch:

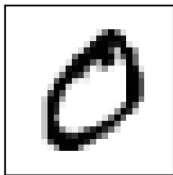
## Machine Learning Crash Course

- **Machine learning** refers to algorithms which improve automatically through experience/data
- There are various types of machine learning (supervised, unsupervised, reinforcement, etc.)
- This crash course focuses on **supervised learning**:
  - The data takes the form of (input, output) pairs
  - Example: given a picture of a handwritten digit, identify which number was written

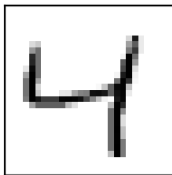
$y = 5$



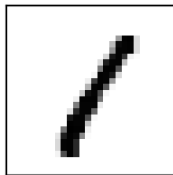
$y = 0$



$y = 4$



$y = 1$



# PyTorch's Flavor of Machine Learning: Neural Network Crash Course

- PyTorch is geared to a specific family of machine learning approaches: **(deep) neural networks**
- Neural networks are a broad family of models which started out as biologically-inspired data transformations based on networks of “artificial neurons”
  - Modern practice has diverged from these biologically-inspired roots, but some of the terminology remains

# Basic Neural Network Architecture: Multi-Layer Perceptron (MLP)

Most neural network approaches can be seen as an alternating sequence of linear transformations and (elementwise) non-linear functions:

$$x_i = \sigma(W_i x_{i-1} + b_i)$$

- $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  is a nonlinear **activation function** which is applied separately to each element of its (vector) input
- $W_i \in \mathbb{R}^{n_{\text{out}} \times n_{\text{in}}}$  and  $b_i \in \mathbb{R}^{n_{\text{out}}}$  are **learnable parameters**

$$x_i = \sigma \left( W_i \times x_{i-1} + b_i \right)$$

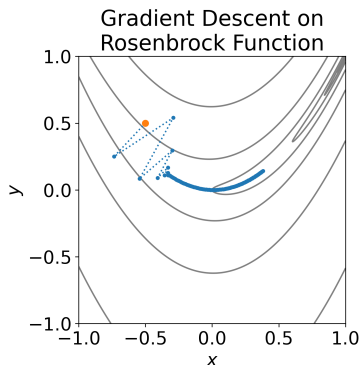


# Nuts and Bolts of Neural Network Training:

## Gradient Descent

- In order to **train** a neural network with  $L$  layers, we need to find values for the parameters  $\theta = \{W_i, b_i | 1 \leq i \leq L\}$
- We quantify how good specific parameter values are using a **loss function**  $\mathcal{L} : \Theta \rightarrow \mathbb{R}$  which indicates how well the network matches its training data for the given parameters
  - Goal is then to find  $\hat{\theta} = \arg \min_{\theta} \mathcal{L}(\theta)$
  - Can seek a (local) minimum using **gradient descent**: update parameters according to

$$\theta_{i+1} = \theta_i - \alpha \nabla \mathcal{L}(\theta_i)$$



# Why Not Just Write It in NumPy?

## Backpropagation and Automatic Differentiation

- Gradient descent requires computing the **gradient**  $\nabla \mathcal{L}(\theta_i)$
- The **backpropagation** algorithm provides an efficient way of doing this, *but needing to write explicit expressions for the gradient of your neural network would be exceedingly tedious and error-prone*
- The heart of PyTorch is the ability to perform **automatic differentiation**: you simply define the loss function computation, and PyTorch automatically computes the gradients for you

# PyTorch Crash Course: 0 to MNIST in 1 Hour

- 1 Background: Machine Learning, Neural Networks, and PyTorch
- 2 PyTorch Fundamentals: The Tensor Class**
- 3 Higher Abstractions: The `torch.nn` Module
- 4 Data Handling: The `Dataset` and `DataLoader` Classes
- 5 Putting It All Together to Solve MNIST: The “hello, world” of Machine Learning
- 6 Summary and Resources

# PyTorch Fundamentals: What Is a Tensor?

A tensor  $\mathbf{H}$  of valence  $\begin{Bmatrix} f \\ v \end{Bmatrix}$  at a point  $p$  is a multilinear, real-valued function of  $f$  1-forms, and  $v$  vectors, such that the value of  $\mathbf{H}$  at  $p$  only depends on the values of the 1-forms and vectors at  $p$ .<sup>2</sup>

---

<sup>2</sup>*Visual Differential Geometry and Forms*, T. Needham (2021)

# PyTorch Fundamentals: What Is a Tensor?

~~A tensor  $\mathbf{H}$  of valence  $\begin{Bmatrix} f \\ n \end{Bmatrix}$  at a point  $p$  is a multilinear, real-valued function of  $f$  1-forms, and  $n$  vectors, such that the value of  $\mathbf{H}$  at  $p$  only depends on the values of the 1-forms and vectors at  $p$ .<sup>2</sup>~~

**lol, nope!**

---

<sup>2</sup>*Visual Differential Geometry and Forms*, T. Needham (2021)

# PyTorch Fundamentals:

## What Does PyTorch Think a Tensor Is?

- In machine learning, it is common to abuse the word “tensor” to refer to any  $N$ -dimensional array of data
- PyTorch’s `Tensor` class is very similar to the `ndarray` class in NumPy, but with some extra machinery attached to:
  1. Keep track of gradients
  2. Easily move between CPU and GPU

# Tensor Basics

A given Tensor `x` consists of several key elements:

- A chunk of memory
- A datatype which indicates what type of value is stored in the memory (`x.dtype`)
- A shape which indicates how the elements are arranged in an  $N$ -dimensional grid (`x.shape`)

# Basics of Tensor Indexing

- You can index into a Tensor just like a Python list, with the added twist that there can be as many indices as there are dimensions:

```
1 import torch
2 x = torch.rand(3, 4, 5)  # Make a random
   Tensor
3 print(x.shape)  # torch.Size([3, 4, 5])
4 print(x[1, 0, 2])  # tensor(0.3494)
5 print(x[:, 0, 0])
6 # tensor([0.9643, 0.2863, 0.1553])
```

- There are lots of powerful things you can do by using bool and int Tensors to index into other Tensors: the examples here just barely scratch the surface



# Conventional Tensor Shapes

There are a few conventions in use for specific meanings of various dimensions for various types of data:

- Generic data (like for an MLP): shape is (samples, features)
- Image data (like for a CNN): shape is (samples, channels, height, width)
- Sequence data (like for an RNN):
  - Defaults to (steps, samples, features) for RNN/LSTM/GRU
  - But, I prefer to use the optional (samples, steps, features) form so that the samples dimension is first

## Tensor Superpowers: Autograd Example

```
1 x = torch.tensor(1.0, requires_grad=True)
2 y = torch.tensor(1.0, requires_grad=True)
3 z = x ** 2 + x + y
4 z.backward() # Compute the gradients
5 print(x.grad) # tensor(3.)
6 print(y.grad) # tensor(1.)
```

- Symbolically, you would have to find  $dz/dx = 2x + 1$ ,  $dz/dy = 1$ : *but PyTorch does it automatically for you*
- The call to `z.backward()` stores  $dz/dx$  into `x.grad` and  $dz/dy$  into `y.grad`

## More Details on Autograd

- Use the `requires_grad` keyword to construct a Tensor which will have gradients computed for it
- Use the `z.backward()` method to compute gradients of `z` with respect to all Tensors which were involved in its computation
  - Gradients of `z` with respect to `x` are stored in `x.grad`
  - Gradients are *accumulated* with subsequent calls to `backward()`: often need to manually zero out gradients
- Tracking the computation graph can be expensive: when you do not need gradients, use the `no_grad` context manager:

```
1 with torch.no_grad():  
2     y = x ** 2
```

# Tensor Superpowers: Using a GPU

- Can easily move Tensors between GPU and CPU:
  - Can create Tensor on GPU using the device keyword:  
`x = torch.tensor(1.0, device=torch.device('cuda'))`
  - Can copy existing Tensor to GPU using the `to()` method:  
`x = x.to(torch.device('cuda'))`
  - Can copy existing Tensor to CPU using the `cpu()` method:  
`x = x.cpu()`
- Best practice: don't hard-code the device keyword. Instead, make your code fail back to CPU if GPU is unavailable:

```
1 if torch.cuda.is_available():  
2     device = torch.device('cuda')  
3 else:  
4     device = torch.device('cpu')  
5  
6 x = torch.tensor(1.0, device=device)
```

# PyTorch Crash Course: 0 to MNIST in 1 Hour

- 1 Background: Machine Learning, Neural Networks, and PyTorch
- 2 PyTorch Fundamentals: The Tensor Class
- 3 Higher Abstractions: The `torch.nn` Module**
- 4 Data Handling: The Dataset and DataLoader Classes
- 5 Putting It All Together to Solve MNIST: The “hello, world” of Machine Learning
- 6 Summary and Resources

# Neural Network Building Blocks: The `torch.nn` Module

The `torch.nn` module has a wide array of standard neural network building blocks, including:

- Linear transformations
- Various activation functions
- Convolutional layers
- Recurrent layers

## Defining an MLP Using torch.nn

```
1 import torch.nn as nn
2
3 net = nn.Sequential(
4     nn.Linear(16, 128), # 16 in, 128 out
5     nn.ReLU(), # "ReLU" activation function
6     nn.Linear(128, 4) # 128 in, 4 out
7 )
8
9 x = torch.rand(1, 16)
10 out = net(x)
11 print(out.shape) # torch.Size([1, 4])
```

# The Module Class

- All of the building blocks in `torch.nn` inherit from the `torch.nn.Module` class
- A given Module has:
  - Parameters: Tensors which the optimizer should update during training
  - Buffers: Tensors which should be included when saving/restoring the Module, but which should *not* be updated by the optimizer
  - Submodules: other Modules which are contained within the Module
  - A `forward()` method which defines the actual operation performed by the Module
    - Important: never call `forward()` directly: Module provides `__call__()` which wraps `forward()` with extra steps!



## Some Additional Key Module Methods

- `to(device)`: Move to the given device
  - A Module and the Tensors it operates on must be on the same device
- `train()`: Put into training mode
- `eval()`: Put into eval mode
- `parameters()`: Get an iterator over the trainable parameters

# Implementing a Custom Module Class

```
1 class ScaledLinear(nn.Module):
2     def __init__(self, in_features, out_features,
3         scale):
4         super().__init__()
5         self.weight = nn.Parameter(torch.empty(
6             out_features, in_features))
7         self.bias = nn.Parameter(torch.empty(
8             out_features))
9         self.register_buffer('scale', torch.
10             as_tensor(scale))
11
12     def forward(self, x):
13         return self.scale * torch.nn.functional.
14             linear(
15                 x, self.weight, self.bias
16             )
```

## Using the Custom Module

```
1 layer = ScaledLinear(16, 4, 2.0)
2 x = torch.rand(1, 16)
3 out = layer(x)
4 print(out.shape)    # torch.Size([1, 4])
5 print(layer.state_dict().keys())
6 # OrderedDict keys(['weight', 'bias', 'scale'])
```

# Saving/Loading Modules

- The `state_dict()` method returns a dict which has the values of all of the parameters and buffers associated with a given Module
- This can be saved to disk, and restored using the `load_state_dict()` method

```
1 layer = ScaledLinear(16, 4, 2.0)
2 state_dict = layer.state_dict()
3 torch.save(state_dict, 'model.pt')
4
5 state_dict = torch.load('model.pt')
6 layer.load_state_dict(state_dict)
7 # <All keys matched successfully>
```

# PyTorch Crash Course: 0 to MNIST in 1 Hour

- 1 Background: Machine Learning, Neural Networks, and PyTorch
- 2 PyTorch Fundamentals: The Tensor Class
- 3 Higher Abstractions: The `torch.nn` Module
- 4 Data Handling: The Dataset and DataLoader Classes**
- 5 Putting It All Together to Solve MNIST: The “hello, world” of Machine Learning
- 6 Summary and Resources

# Two-Stage Data Handling: Dataset and DataLoader

PyTorch breaks down data handling into two steps:

1. A Dataset provides a wrapper to access single (input, output) pairs at a time
2. A DataLoader combines multiple samples from a Dataset into batches

Both of these classes are defined in `torch.utils.data`

## Dataset: Single-Sample Access

To get your data into PyTorch's format, you simply need to subclass `Dataset` and implement two methods:

- `__len__()`: returns the number of samples in the `Dataset`
  - It is usually safe to change the length over time: a `Dataset` can grow/shrink
- `__getitem__(idx)`: returns the sample at a given index
  - Can do arbitrary processing here: e.g., load image from disk, crop, scale, convert to `Tensor`, etc.
  - Can return fairly arbitrary output, as long as the result at each index has the same form
  - Do not always need to return the same value for a given index: a `Dataset` can perform random data augmentations or even data generation

## TensorDataset Example

If your data are already stored in Tensors, can simply wrap them with a TensorDataset:

```
1 x = torch.rand(500, 10, dtype=torch.float)
2 y = torch.randint(low=0, high=5, size=(500,),
    dtype=torch.long)
3 dataset = TensorDataset(x, y)
4
5 x_0, y_0 = dataset[0]
6 print(x_0.shape)  # torch.Size([10])
7 print(y_0)  # tensor(1)
```



## DataLoader: Sampling and Batching

- Typically train using **mini-batch stochastic gradient descent**: update the trainable parameters using gradients computed from a random subset of the training data
- DataLoader wraps a Dataset and enables iteration over batches

```
1 x = torch.rand(500, 10, dtype=torch.float)
2 y = torch.randint(low=0, high=5, size=(500,), dtype
   =torch.long)
3 dataset = TensorDataset(x, y)
4 loader = DataLoader(dataset, batch_size=5, shuffle=
   True)
5
6 x_0, y_0 = next(iter(loader))
7 print(x_0.shape)    # torch.Size([5, 10])
8 print(y_0.shape)    # torch.Size([5])
```

# PyTorch Crash Course: 0 to MNIST in 1 Hour

- 1 Background: Machine Learning, Neural Networks, and PyTorch
- 2 PyTorch Fundamentals: The Tensor Class
- 3 Higher Abstractions: The `torch.nn` Module
- 4 Data Handling: The Dataset and DataLoader Classes
- 5 Putting It All Together to Solve MNIST: The “hello, world” of Machine Learning**
- 6 Summary and Resources

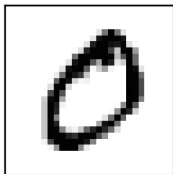
# MNIST: The “hello, world” of Machine Learning

- One of the standard image processing benchmark datasets for many years
- Consists of  $28 \times 28$  grayscale images of handwritten digits: 60 000 training images and 10 000 test images
  - Data are very clean: only one digit per image, nicely centered/scaled, etc.
- Fairly small/easy by today's standards, but good for education because you can successfully train models on the CPU

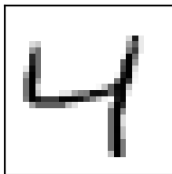
$y = 5$



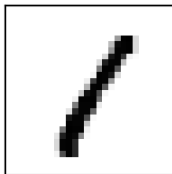
$y = 0$



$y = 4$



$y = 1$



# Solution Script (Walk Through Full Code in Editor)

```
1  train_loader, val_loader, test_loader = get_data(
2      drop_last=True, batch_size=128
3  )
4
5  num_class = 10
6  layer_widths = [256, 128, 64]
7  model = MLPNet(train_loader.dataset[0][0].shape, num_class,
8      layer_widths)
9  model.to(device)
10
11  optimizer = optim.Adam(model.parameters())
12
13  loss_fn = nn.CrossEntropyLoss(reduction='sum')
14
15  num_epoch = 10
16  run_training_loop(
17      train_loader, val_loader, model, loss_fn, device,
18      optimizer, num_epoch
19  )
20  torch.save(model.state_dict(), 'model.pt')
21
22  test_loss, test_acc = epoch(test_loader, model, loss_fn,
23      device)
24  print('\n\t\tLoss\tAcc.')
25  print(f'Test:\t\t{test_loss:.3f}\t{test_acc:.3f}')
```

## Example Output (Last Few Lines)

```
1 Epoch 7:      Loss      Acc.
2 Train:        0.027      0.992
3 Val:          0.102      0.974
4
5 Epoch 8:      Loss      Acc.
6 Train:        0.021      0.993
7 Val:          0.108      0.970
8
9 Epoch 9:      Loss      Acc.
10 Train:       0.019      0.994
11 Val:         0.090      0.979
12
13              Loss      Acc.
14 Test:        0.081      0.979
```

# PyTorch Crash Course: 0 to MNIST in 1 Hour

- 1 Background: Machine Learning, Neural Networks, and PyTorch
- 2 PyTorch Fundamentals: The Tensor Class
- 3 Higher Abstractions: The `torch.nn` Module
- 4 Data Handling: The Dataset and DataLoader Classes
- 5 Putting It All Together to Solve MNIST: The “hello, world” of Machine Learning
- 6 Summary and Resources**

# Summary and Additional Resources

PyTorch-specific resources:

- PyTorch documentation:  
<https://pytorch.org/docs/stable/index.html>
- PyTorch tutorials: <https://pytorch.org/tutorials/>
- PyTorch book: <https://pytorch.org/assets/deep-learning/Deep-Learning-with-PyTorch.pdf>

General machine learning/deep learning resources:

- Introduction to Statistical Learning (R-focused, but very popular for learning the fundamentals):  
<https://www.statlearning.com/>
- Probabilistic Machine Learning (book series):  
<https://probml.github.io/pml-book/>
- Deep Learning: <https://www.deeplearningbook.org/>