



# Introduction to PyTorch

CPEN 455 Tutorial 3

Jan 27, 2025

Qi Yan

Acknowledgment: Muchen Li



# Outline

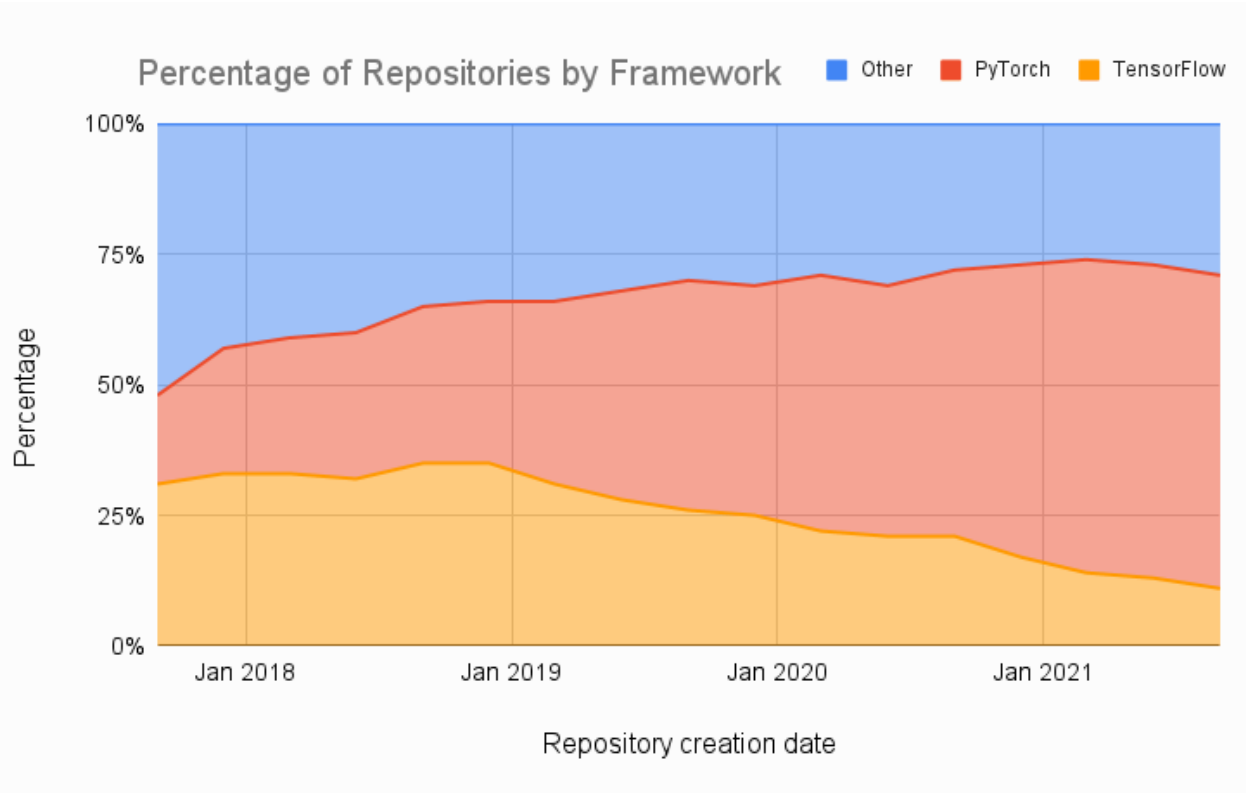
- Introduction
- Tensor & Tensor Operations
- Dataset & Dataloaders
- Build the model
- Model Optimization



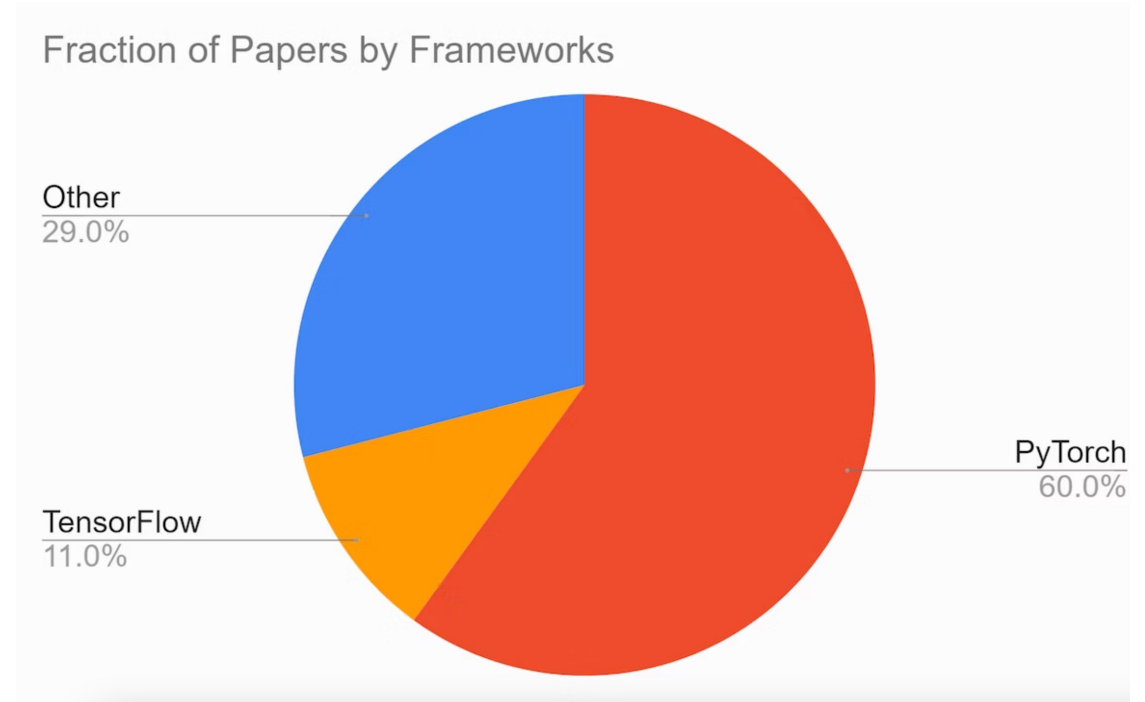
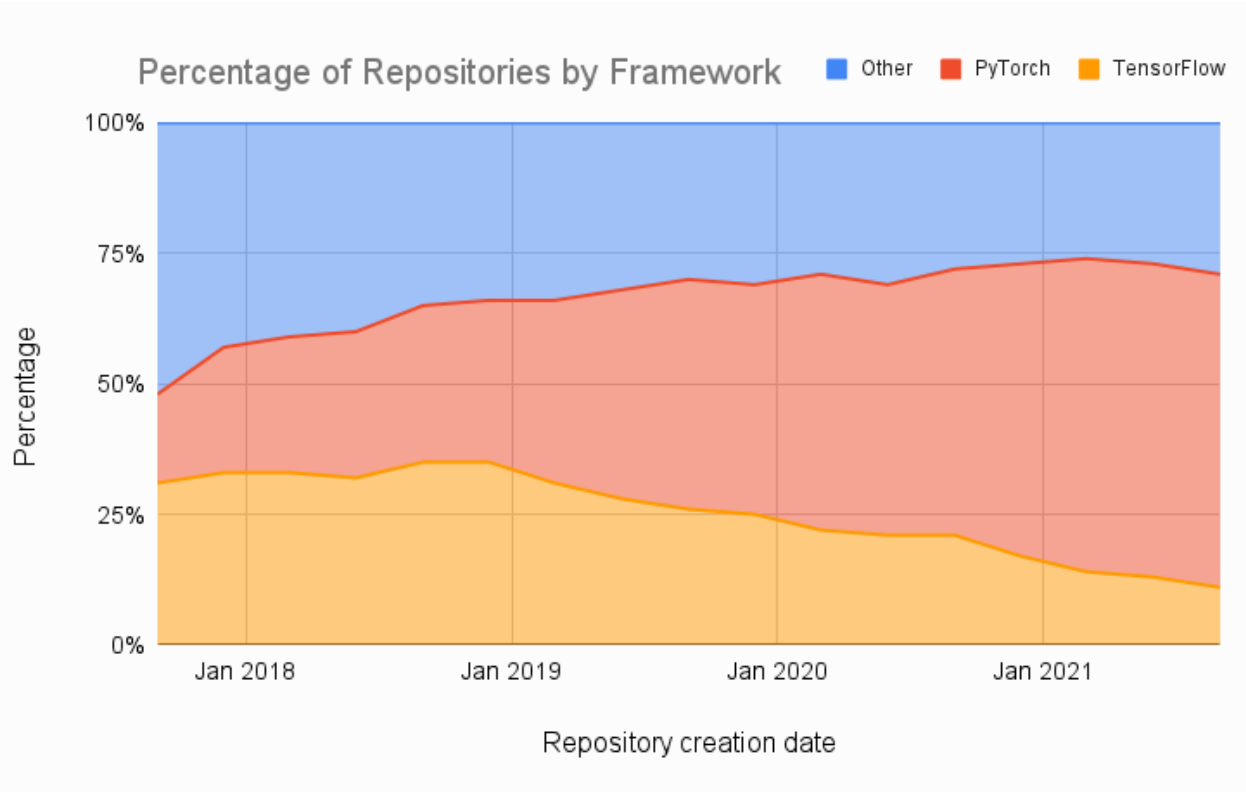
PyTorch [1] is a deep learning framework (free and open-sourced under the modified BSD license) based on the Torch library, originally developed by Meta AI and now part of the Linux Foundation umbrella.

Many pieces of deep learning software are built on top of PyTorch, including Tesla Autopilot, Uber's Pyro, Hugging Face's Transformers, PyTorch Lightning, and Catalyst.

# Pytorch is getting Popular



# Pytorch is getting Popular



# DL Frameworks



Caffe



PYTORCH



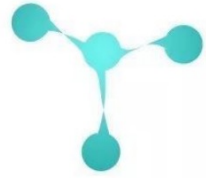
dy/net



# DL Frameworks



Caffe



PYTORCH



dy/net



Caffe  
(UC Berkeley)



Caffe2  
(Facebook)



Torch  
(NYU / Facebook)



PyTorch  
(Facebook)

Theano  
(U Montreal)



TensorFlow  
(Google)

PaddlePaddle  
(Baidu)

Chainer

MXNet  
(Amazon)  
Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK  
(Microsoft)

JAX  
(Google)

# Why Pytorch

- It's easy to use.
- It's Pythonic and flexible (easy to customize and extend).
- It's your best friend in deep learning projects.

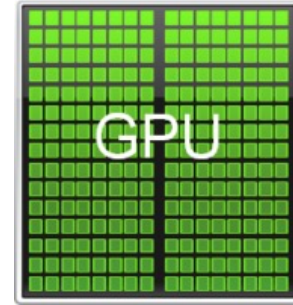


# Why Pytorch

- It's easy to use.
- It's Pythonic and flexible (easy to customize and extend).
- It's your best friend in deep learning projects.
- And you'll need it in your assignment anyway ;)

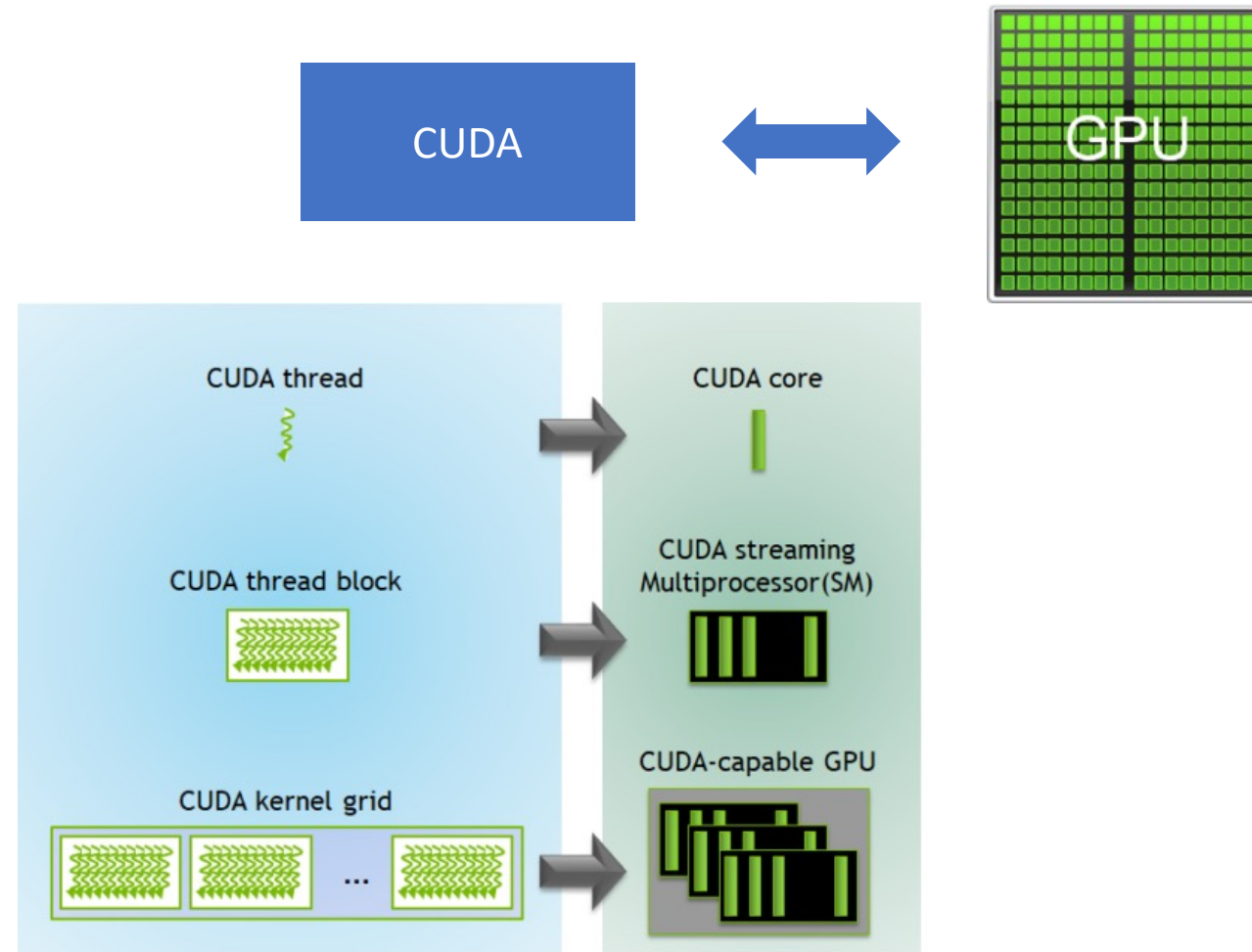
# DL Frameworks Bring Peace of Mind

- GPU Acceleration



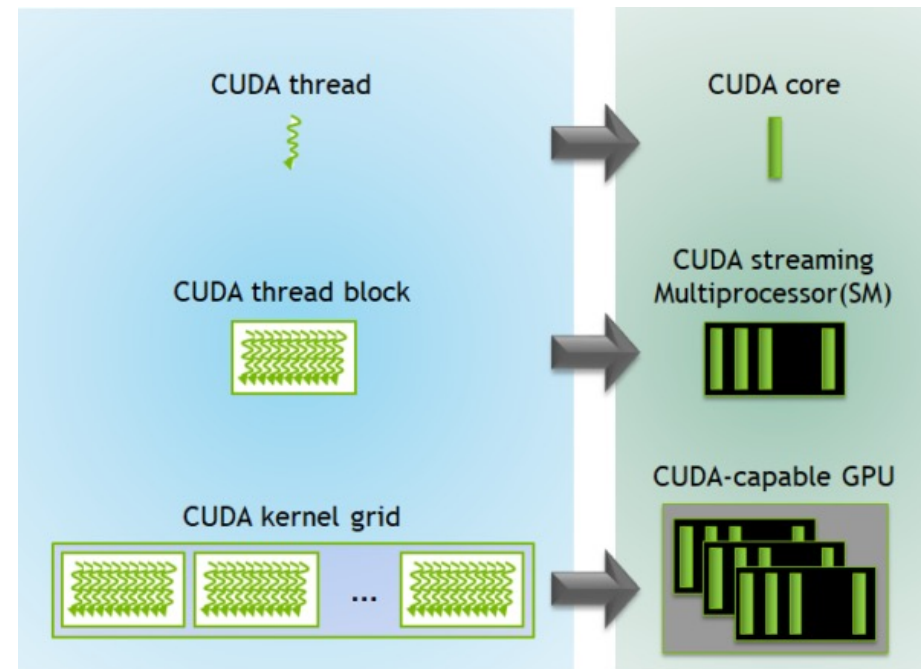
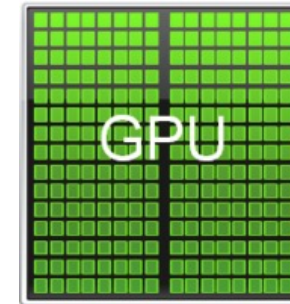
# DL Frameworks Bring Peace of Mind

- GPU Acceleration



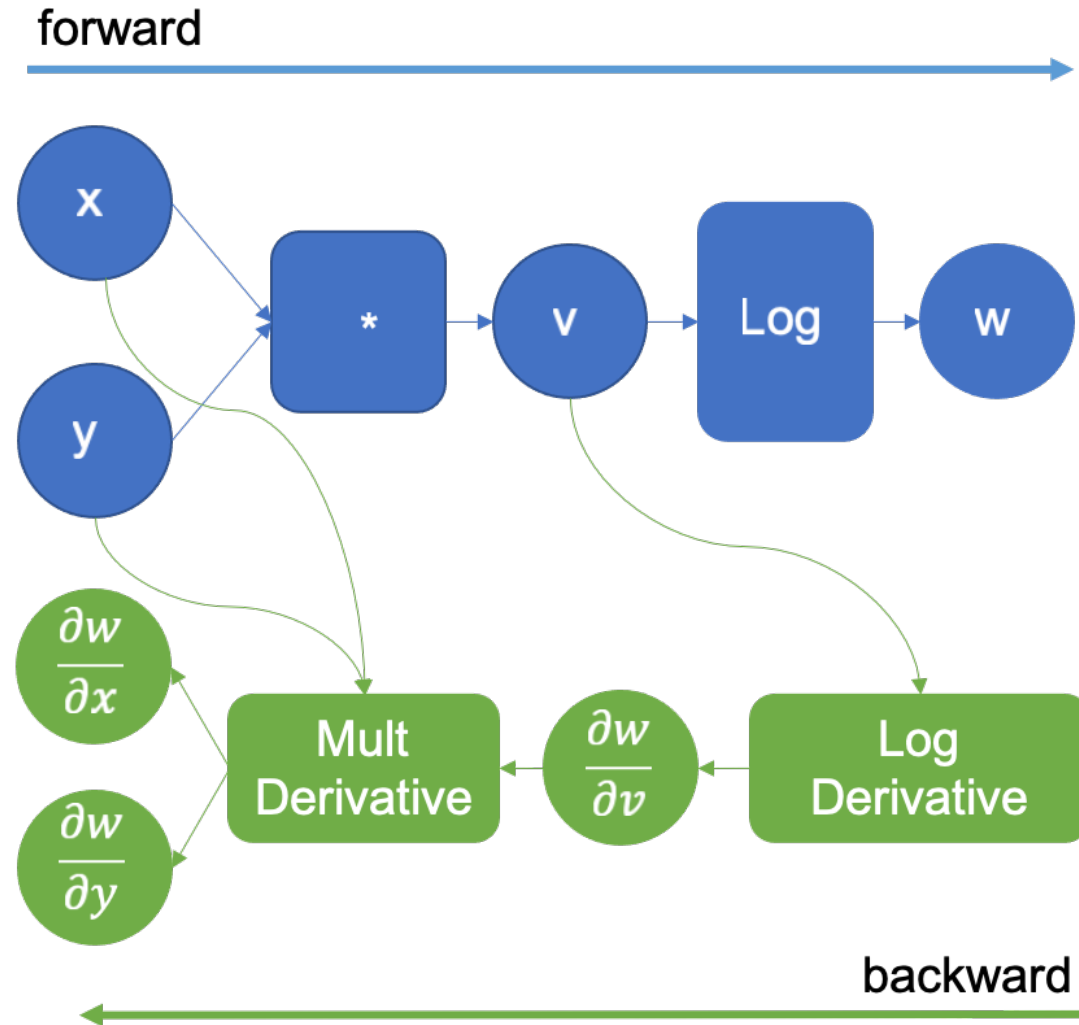
# DL Frameworks Bring Peace of Mind

- GPU Acceleration



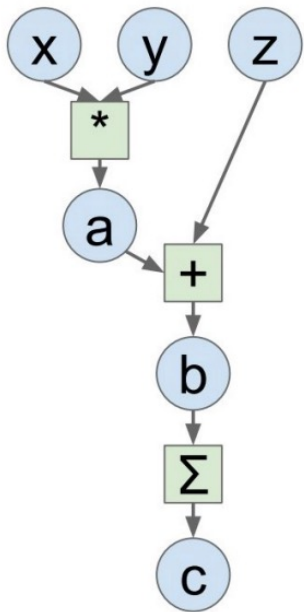
# DL Frameworks Bring Peace of Mind

- Autograd: A reverse automatic differentiation system.



# DL Frameworks

## Computation Graph



## Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

## Tensorflow

```
import numpy as np
np.random.seed(0)
import tensorflow as tf

N, D = 3, 4

with tf.device('/gpu:0'):
    x = tf.placeholder(tf.float32)
    y = tf.placeholder(tf.float32)
    z = tf.placeholder(tf.float32)

    a = x * y
    b = a + z
    c = tf.reduce_sum(b)

grad_x, grad_y, grad_z = tf.gradients(c, [x, y, z])

with tf.Session() as sess:
    values = {
        x: np.random.randn(N, D),
        y: np.random.randn(N, D),
        z: np.random.randn(N, D),
    }
    out = sess.run([c, grad_x, grad_y, grad_z],
                    feed_dict=values)
    c_val, grad_x_val, grad_y_val, grad_z_val = out
```

## PyTorch

```
import torch

N, D = 3, 4

x = torch.rand((N, D), requires_grad=True)
y = torch.rand((N, D), requires_grad=True)
z = torch.rand((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
```

# Installing Pytorch

- On your own device:
  - Use Anaconda/MiniConda to manage
  - *conda install pytorch -c pytorch*
  - <https://docs.conda.io/projects/miniconda/en/latest/>
- Google Colab:
  - Jupyter notebook Management
  - *!pip3 install torch*
  - <https://colab.google/>

# Write Code

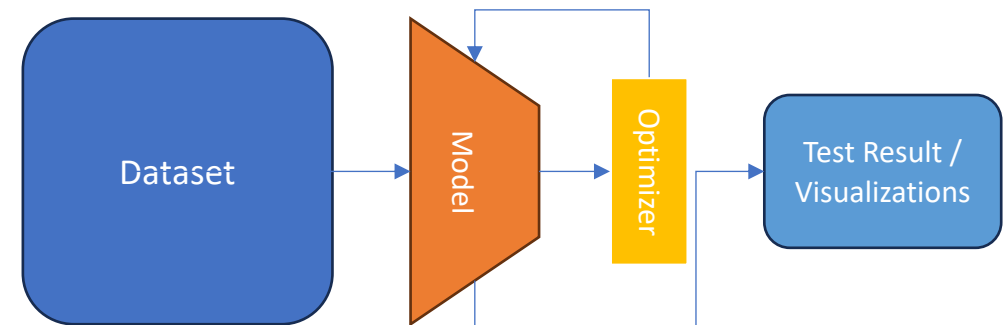
- Use Whatever Editor you like
- Jupyter Lab/Jupyter Notebook
  - Interactive coding session
  - Install: `pip/conda install jupyter`
  - Run: `jupyter notebook --port 8888`
  - Visit at `localhost:8888`
- Debug pytorch code just like debugging any python code



# Overview

## Fundamental Concepts of PyTorch:

- Tensors
- Autograd
- Modular structure
  - Models / Layers
  - Datasets & Dataloader



# 1. Tensors

- Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameter
- Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators.

# 1. Tensors

## 1.1 Creating a Tensor

## 1.2 Attribute of Tensors

- Shape, Device and datatype

## 1.3 Operation on Tensors

- Move between devices
- Indexing & Slicing
- Stack & Concatenation
- Algorithmic operations (Tensor products)

# 1. Creating a Tensor

- Directly from Data

```
data = [[1, 2],[3, 4]]  
x_data = torch.tensor(data)
```

```
np_array = np.array(data)  
x_np = torch.from_numpy(np_array)
```

# 1. Creating a Tensor

- Directly from Data

```
data = [[1, 2],[3, 4]]  
x_data = torch.tensor(data)
```

```
np_array = np.array(data)  
x_np = torch.from_numpy(np_array)
```

- With Random / Constant Value

```
shape = (2,3,)  
rand_tensor = torch.rand(shape)  
ones_tensor = torch.ones(shape)  
zeros_tensor = torch.zeros(shape)  
  
print(f"Random Tensor: \n {rand_tensor} \n")  
print(f"Ones Tensor: \n {ones_tensor} \n")  
print(f"Zeros Tensor: \n {zeros_tensor}")
```

```
Random Tensor:  
tensor([[0.5771, 0.5705, 0.3618],  
        [0.9053, 0.1620, 0.2274]])  
  
Ones Tensor:  
tensor([[1., 1., 1.],  
        [1., 1., 1.]])  
  
Zeros Tensor:  
tensor([[0., 0., 0.],  
        [0., 0., 0.]])
```

# 1.2 Attribute of Tensors

- Data type, Shape and Device

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

# 1.2 Attribute of Tensors

- Data type, Shape and Device

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

**requires\_grad**: a flag for whether the tensor is a parameter to be updated

# 1.3 Operations

- There is over 100 tensor operations, including arithmetic, linear algebra (e.g. matrix manipulation transposing, indexing, slicing), sampling).



# 1.3 Operations: Move between devices

- By default tensors are created on CPU, one can move tensor explicitly to GPU for future operations:

```
# We move our tensor to the GPU if available  
if torch.cuda.is_available():  
    tensor = tensor.to("cuda")
```

- In the case of multiple GPUs, you should specify the CUDA id like “cuda:1”, otherwise, “cuda” default to GPU 0.


# 1.3 Operations: Index & Slicing

- Pytorch support a lot of APIs available in Numpy. E.g. You can index and slice tensor exactly like numpy arrays:


```
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f>Last column: {tensor[:, -1]}")
tensor[:,1] = 0
print(tensor)
```

```
First row: tensor([1., 1., 1., 1.])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

# Tips 1.1 Mastering Indexing

- Basic Indexing
  - Single element indexing
  - Dimensional indexing tools: ':', '...'
  - Slicing and striding: `x[start:end:stride]`
- Advance Indexing
  - Integer array indexing
  - Boolean array indexing
- Demo 

# Tips 1.2: Master the Shape of your Tensor

- Mastering the shape of your tensor can be the most crucial to get your code right. In practice, we need to get tensor into right shape before we can apply certain functions on it.
- `.view()` `.reshape()` `.contiguous()`
- `.permute()` `.transpose()` `.repeat()`
- `.squeeze()` `.unsqueeze()`
- Demo 

# 1.3 Operations: Concatenation

- You can use `torch.cat` to concatenate a sequence of tensors along a given dimension. See also `torch.stack`

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

```
In [3]: t1 = torch.cat([tensor, tensor, tensor], dim=1)
...: print(t1)
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

# 1.3 Operations: Arithmetic operations

```
# This computes the matrix multiplication between two tensors. y1, y2, y3  
will have the same value
```

```
# ``tensor.T`` returns the transpose of a tensor
```

```
y1 = tensor @ tensor.T
```

```
y2 = tensor.matmul(tensor.T)
```

```
y3 = torch.rand_like(y1)
```

```
torch.matmul(tensor, tensor.T, out=y3)
```

```
# This computes the element-wise product. z1, z2, z3 will have the same  
value
```

```
z1 = tensor * tensor
```

```
z2 = tensor.mul(tensor)
```

```
z3 = torch.rand_like(tensor)
```

```
torch.mul(tensor, tensor, out=z3)
```

```
tensor([[3., 3., 3., 3.],  
        [3., 3., 3., 3.],  
        [3., 3., 3., 3.],  
        [3., 3., 3., 3.]])
```

# 1.3 Operations: Arithmetic operations

```
# This computes the matrix multiplication between two tensors. y1, y2, y3  
will have the same value  
# ``tensor.T`` returns the transpose of a tensor
```

```
y1 = tensor @ tensor.T  
y2 = tensor.matmul(tensor.T)
```

```
y3 = torch.rand_like(y1)  
torch.matmul(tensor, tensor.T, out=y3)
```

```
# This computes the element-wise product. z1, z2, z3 will have the same  
value
```

```
z1 = tensor * tensor  
z2 = tensor.mul(tensor)
```

```
z3 = torch.rand_like(tensor)  
torch.mul(tensor, tensor, out=z3)
```

```
tensor([[1., 0., 1., 1.],  
        [1., 0., 1., 1.],  
        [1., 0., 1., 1.],  
        [1., 0., 1., 1.]])
```

# Tips 1.3 Avoid Loops

- Loops can be easy to implement but is very inefficient.
- You should avoid using loops as much as you can and try to use pytorch provided API to get around. Let the optimized backend library plays its role.



# Tips 1.3 Avoid Loops

- Loops can be easy to implement but is very inefficient.
- You should avoid using loops as much as you can and try to use pytorch provided API to get around. Let the optimized backend library plays its role.

- Vectorization:

Example 1: To calculate a dot product between Vector X and Vector Y

Use `torch.dot(X, Y)` instead of a for loop.

Example 2: Batch Matrix Multiplication A:  $k*m*n$ . B:  $k*n*1$

Use `torch.bmm(A, B)` instead of doing `torch.mm()` k times.

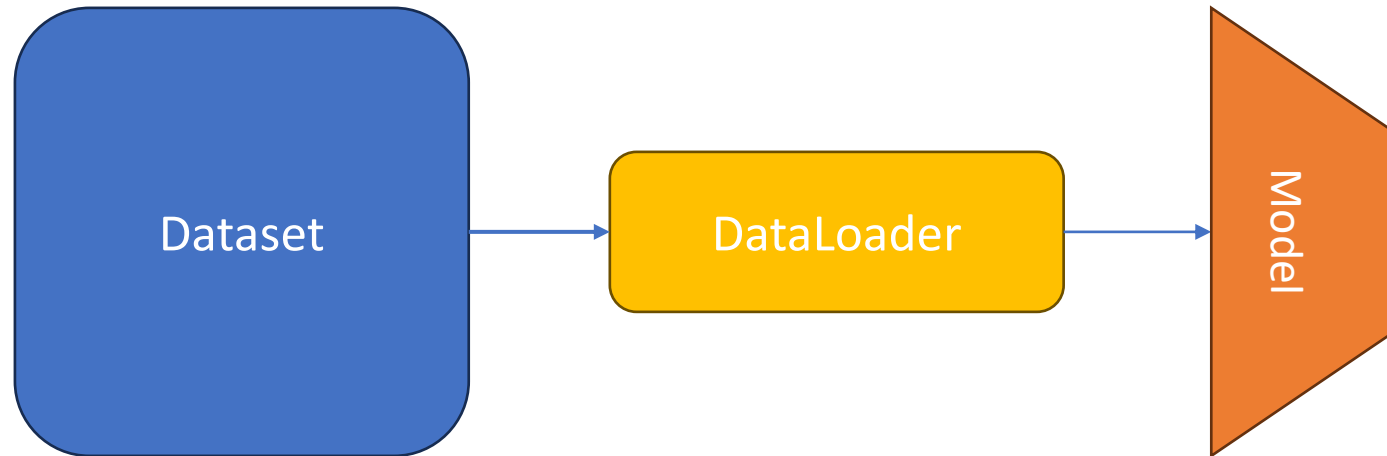
# 1.3 Operations

Check out more operations at

<https://pytorch.org/docs/stable/torch.html>

## 2. Dataset & Dataloaders

- Data is of crucial importance in deep learning. We want to handle it well and reduce the overhead of the dl system.
  - **torch.utils.data.DataLoader**
  - **torch.utils.data.Dataset**



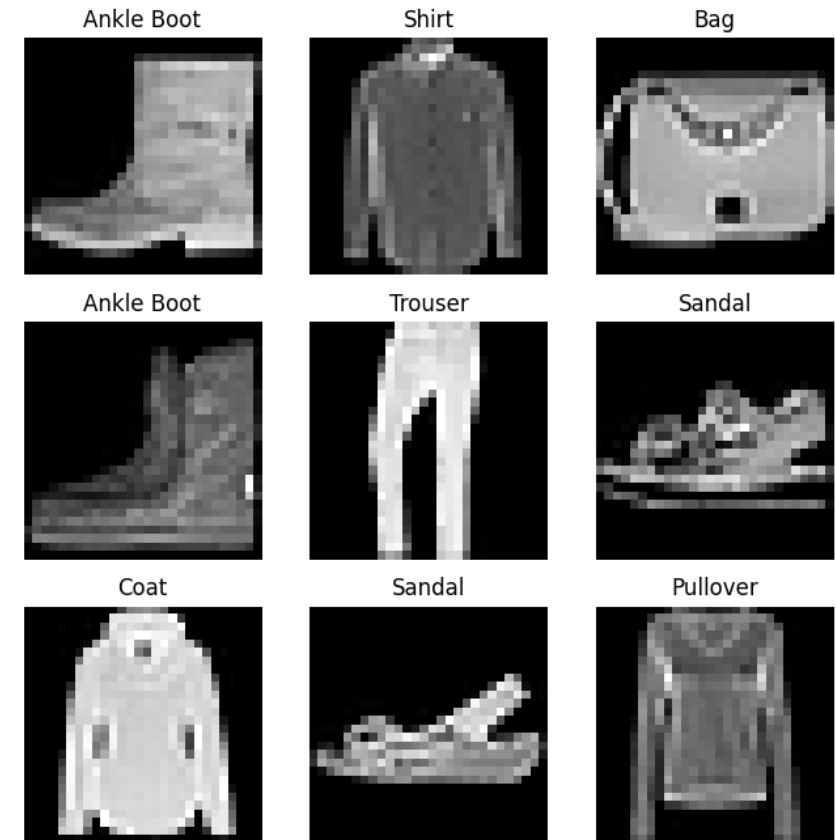
# 2.1 Dataset: A Example

- Image Dataset: FashionMNIST

annotations\_file

```
tshirt1.jpg, 0  
tshirt2.jpg, 0  
.....  
ankleboot999.jpg, 9
```

```
labels_map = {  
    0: "T-Shirt",  
    1: "Trouser",  
    2: "Pullover",  
    3: "Dress",  
    4: "Coat",  
    5: "Sandal",  
    6: "Shirt",  
    7: "Sneaker",  
    8: "Bag",  
    9: "Ankle Boot",  
}
```



# 2.1 Dataset: A Example

```
from torch.utils.data import Dataset
```

```
class CustomImageDataset(Dataset):  
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):  
        self.img_labels = pd.read_csv(annotations_file)  
        self.img_dir = img_dir  
        self.transform = transform  
        self.target_transform = target_transform  
  
    def __len__(self):  
        return len(self.img_labels)  
  
    def __getitem__(self, idx):  
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])  
        image = read_image(img_path)  
        label = self.img_labels.iloc[idx, 1]  
        if self.transform:  
            image = self.transform(image)  
        if self.target_transform:  
            label = self.target_transform(label)  
        return image, label
```

Store data  
Meta data as  
attributed

# 2.1 Dataset: A Example

```
class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Returns  
number of  
samples

# 2.1 Dataset: A Example

```
class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Get the idx th  
sample

Read Image  
data and label

# 2.1 Dataset: A Example

```
class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Apply  
transform to  
data. E.g.  
Normalization/  
Augmentation



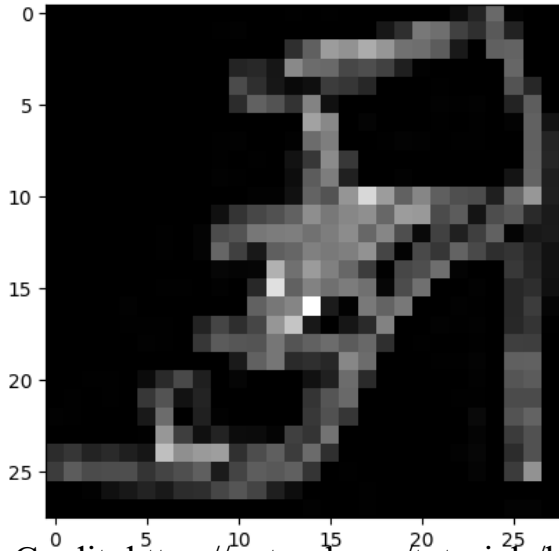
## 2.1 Dataset: A Example

- The Dataset retrieves our dataset's features and labels one sample at a time.
- While training a model, we typically want to pass samples in “minibatches”, reshuffle the data at every epoch to reduce model overfitting, and use Python's multiprocessing to speed up data retrieval.
- DataLoader is an iterable that abstracts this complexity for us in an easy API.

```
from torch.utils.data import DataLoader  
  
train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)  
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

## 2.1 Dataset: A Example

```
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```



```
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
Label: 5
```

# Tips 2.1 What's happening in dataloader

Wait, what exactly do dataloader do?

- Data Loading Order:

- Determine the order to sample data. (e.g. shuffling given a particular seed.) In the multi-gpu distributed training case. It also coordinate to avoid repeat samples between different process.

- Parallel Sampling:

- Reading data sequentially can be a bad idea. Dataloader fork `num_worker` of subprocesses to enable parallel sampling

```
CLASS torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=None, sampler=None,  
batch_sampler=None, num_workers=0, collate_fn=None, pin_memory=False, drop_last=False,  
timeout=0, worker_init_fn=None, multiprocessing_context=None, generator=None, *,  
prefetch_factor=None, persistent_workers=False, pin_memory_device='') [SOURCE]
```

# Tips 2.1 What's happening in dataloader

- Batching data:
  - This is done by the `collate_fn` (collate function), which 1. collate tensor memory for sampled data and 2. concatenate the tensors to be a single batch data. You can create your own `collate_fn` to handle different structure of data.
- Accelerate data loading:
  - This can be done by `pin_memory` and `prefetch_factor`

```
CLASS torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=None, sampler=None,  
batch_sampler=None, num_workers=0, collate_fn=None, pin_memory=False, drop_last=False,  
timeout=0, worker_init_fn=None, multiprocessing_context=None, generator=None, *,  
prefetch_factor=None, persistent_workers=False, pin_memory_device='') [SOURCE]
```

# Recap

- Tensor operations
  - Index, Slice, Stride, Concatenate, reshape
- Dataset and DataLoader Module
- Building Model
- Model Optimization

# Next Time

- More Model Optimization
- **Automatic Gradient**
- Build Training and Testing Loop