Agenda

• Introduction PyTorch
• Installation
• Examples
What is PyTorch

- Open source machine learning library
- Developed by Facebook's AI Research lab
- It leverages the power of GPUs
- Automatic computation of gradients
- Makes it easier to test and develop new ideas.
Popular Deep Learning Frameworks
Dynamic or Static?

Popular Deep Learning Frameworks

**Imperative:** Imperative-style programs perform computation as you run them

```python
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

**Symbolic:** define the function first, then compile them

```python
A = Variable('A')
B = Variable('B')
C = B * A
D = C + Constant(1)
# compiles the function
f = compile(D)
d = f(A=np.ones(10), B=np.ones(10)*2)
```
Popular Deep Learning Frameworks

- imperative
- symbolic

- theano
- Caffe
- Chainer
- PyTorch
- mxnet
- K
- CNTK
- TensorFlow
- Caffe2

before 2012 2013 2014 2015 2016 2017
Popular Deep Learning Frameworks

- Caffe (UC Berkeley)
- Caffe2 (Facebook)
- Torch (NYU / Facebook)
- PyTorch (Facebook)
- Theano (U Montreal)
- TensorFlow (Google)
- Paddle (Baidu)
- CNTK (Microsoft)
- MXNet (Amazon)

And others...
CPU vs GPU

- VGG-16: 66x
- VGG-19: 67x
- ResNet-18: 71x
- ResNet-50: 64x
- ResNet-200: 76x

Example code:

```python
import numpy as np
import torch

d = 3000

# Task: compute matrix multiplication C = AB
A = np.random.rand(d, d).astype(np.float32)
B = np.random.rand(d, d).astype(np.float32)
C = A.dot(B)

# using numpy

# using torch with gpu
A = torch.rand(d, d).cuda()
B = torch.rand(d, d).cuda()
C = torch.mm(A, B)
```

350 ms vs 0.1 ms
Why PyTorch

- It is pythonic - concise, close to Python conventions
- Strong GPU support
- Autograd - automatic differentiation
- Many algorithms and components are already implemented
- Similar to NumPy
Why PyTorch

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 tensorflow as tf
N, D = 3, 4
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)
```

```
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.randn((N, D), requires_grad=True)
y = torch.randn((N, D), requires_grad=True)
z = torch.randn((N, D), requires_grad=True)
a = x * y
b = a + z
c = torch.sum(b)
c.backward()
```

```
PyTorch In Academia

PyTorch vs TensorFlow: % Unique Mentions of PyTorch
Levels of Abstraction

**Tensor**: Imperative ndarray, but runs on GPU

**Variable**: Node in a computational graph; stores data and gradient

**Module**: A neural network layer; may store state or learnable weights
Tensors

Attributes of a tensor 't':

- \( t = \text{torch.randn(1)} \)

- **requires_grad**: making a trainable parameter
  - By default False
  - Turn on:
    - \( t.\text{requires_grad}() \)
    - \( t = \text{torch.randn(1, requires_grad=True)} \)

- Accessing tensor value:
  - \( t.\text{data} \)

- Accessing tensor gradient
  - \( t.\text{grad} \)

- **grad_fn**: history of operations for autograd
  - \( t.\text{grad}._fn \)
Loading Data, Devices and CUDA

Numpy arrays to PyTorch tensors

- `torch.from_numpy(x_train)`
- Returns a cpu tensor!

PyTorch tensor to numpy

- `t.numpy()`

Using GPU acceleration

- `t.to()`
- Sends to whatever device (cuda or cpu)

Fallback to cpu if gpu is unavailable:

- `torch.cuda.is_available()`

Check cpu/gpu tensor OR numpyarray?

- `type(t)` or `t.type()` returns
  - `numpy.ndarray`
  - `torch.Tensor`
    - CPU - `torch.cpu.FloatTensor`
    - GPU - `torch.cuda.FloatTensor`
Autograd

- Automatic Differentiation Package
- Don’t need to worry about partial differentiation, chain rule etc.
  - backward() does that
- Gradients are accumulated for each step by default:
  - Need to zero out gradients after each update
  - tensor.grad_zero()

```python
# Create tensors.
x = torch.tensor(1., requires_grad=True)
w = torch.tensor(2., requires_grad=True)
b = torch.tensor(3., requires_grad=True)

# Build a computational graph.
y = w * x + b  # y = 2 * x + 3

# Compute gradients.
y.backward()

# Print out the gradients.
print(x.grad)   # x.grad = 2
print(w.grad)   # w.grad = 1
print(b.grad)   # b.grad = 1
```
Optimizer and Loss

Optimizer

- Adam, SGD etc.
- An optimizer takes the parameters we want to update, the learning rate we want to use along with other hyper-parameters and performs the updates

Loss

- Various predefined loss functions to choose from
- L1, MSE, Cross Entropy
Model

In PyTorch, a model is represented by a regular Python class that inherits from the Module class.

- Two components
  - \_\_init\_\_(self): it defines the parts that make up the model- in our case, two parameters, a and b
  - forward(self, x): it performs the actual computation, that is, it outputs a prediction, given the input

```python
class ManualLinearRegression(nn.Module):
    def \_\_init\_\_(self):
        super().\_\_init\_\_()
        # To make "a" and "b" real parameters of the model, we need to wrap them with nn.Parameter
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.a + self.b * x
```
Install PyTorch

- [https://pytorch.org/get-started/locally/](https://pytorch.org/get-started/locally/)

```bash
conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch
```
Examples

• https://github.com/shib0li/CS6350-PyTorch-Tutorial