Introduction to PyTorch

Khatam University
Winter 1399

وزارت علوم، تحقیقات و فناوری
دانشگاه خاتم
Khatam University

Outline

- What is PyTorch?
- Trivia about PyTorch
- PyTorch vs TensorFlow
- Core Concepts
- Try PyTorch in Colab!

🔥

Warm up

What is PyTorch?

PyTorch is a machine learning framework which provides us ...

- Tensor computing (like NumPy) with strong acceleration via graphics processing units (GPU)
- Deep neural networks built on a tape-based automatic differentiation system

Trivia

📃 Trivia

PyTorch is ...

- An open source machine learning framework
- Based on Torch library (lua language)
- Redesigned and implemented in Python, keeping the core C libraries as the backbone
- Developed by Facebook's AI Research (FAIR)
- Used by Tesla Autopilot, Uber's Pyro, ...

📜 Trivia

Facebook has also developed ...

- Convolutional Architecture for Fast Feature Embedding (Caffe) which has no compatibility with PyTorch!
- the Open Neural Network Exchange (ONNX) in collaboration with Microsoft. This one is natively supported by PyTorch.

The tough rival

In common

Both TensorFlow and PyTorch are...

- Deep learning frameworks
- Actively developed by technology giants
- Mainly used by Python interface
- Compatible with CUDA
- ...

Pros

In comparison with TensorFlow, PyTorch ...

- Is more pythonic! 🐍 and has easier interface
- Uses dynamic computation graph,
  which leads to more flexibility in model design.
  In fact, this is the main pythonic feature!
- Is easier to debug.
- Better for prototyping, deep learning research 🎓
- Has excellent official docs, accessible without VPN! 😃

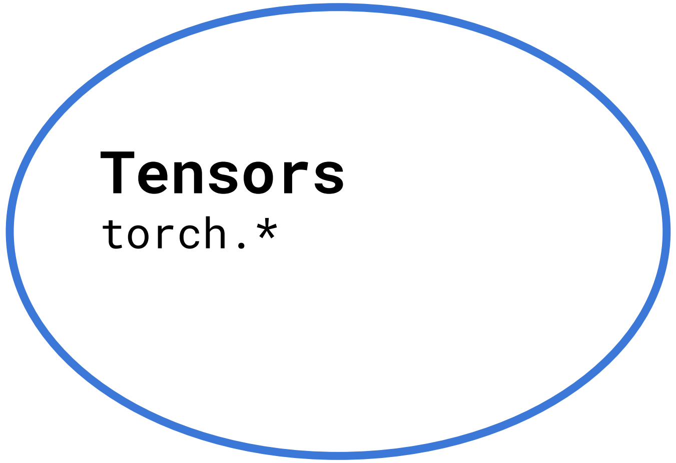
Cons

In comparison with PyTorch, TensorFlow ...

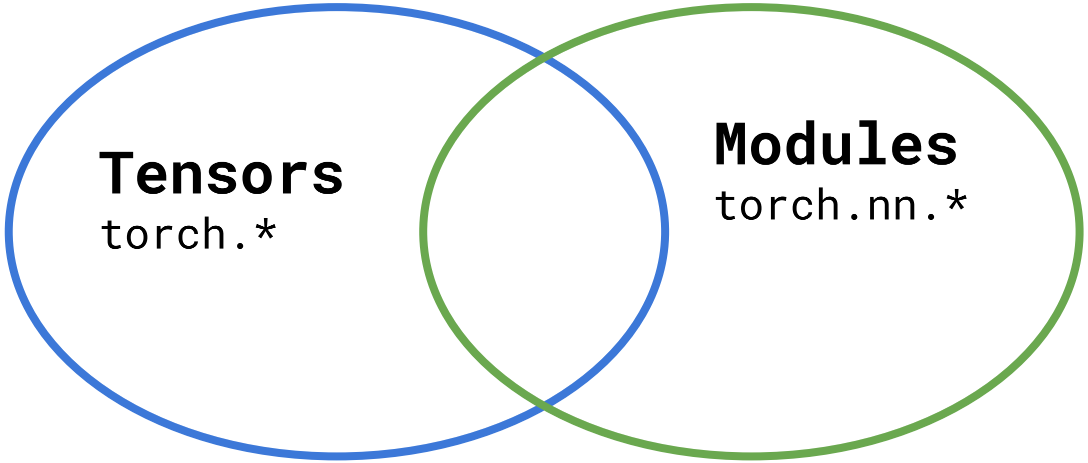
- Is more C++/Java-ic! So you get errors more in compilation and less in runtime!
- Needs less [boilerplate] code for simple use-cases.
- Provides better visualization tools
- Integrates with Google TPUs easily
- Is better for production and deployment

Core concepts

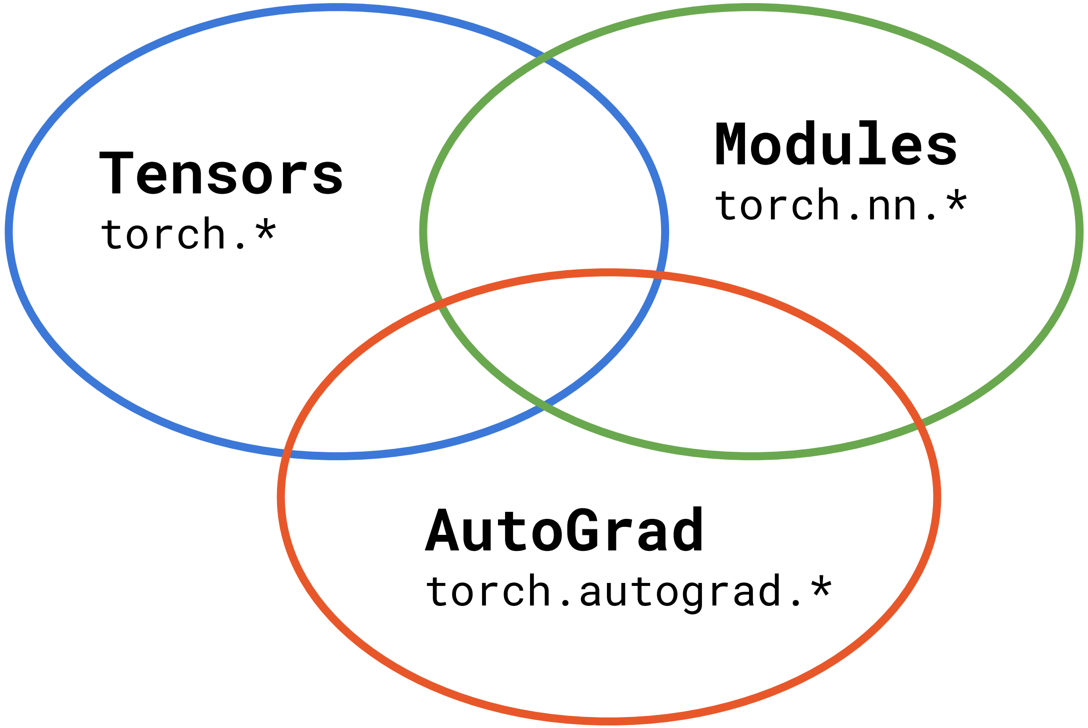
Core concepts



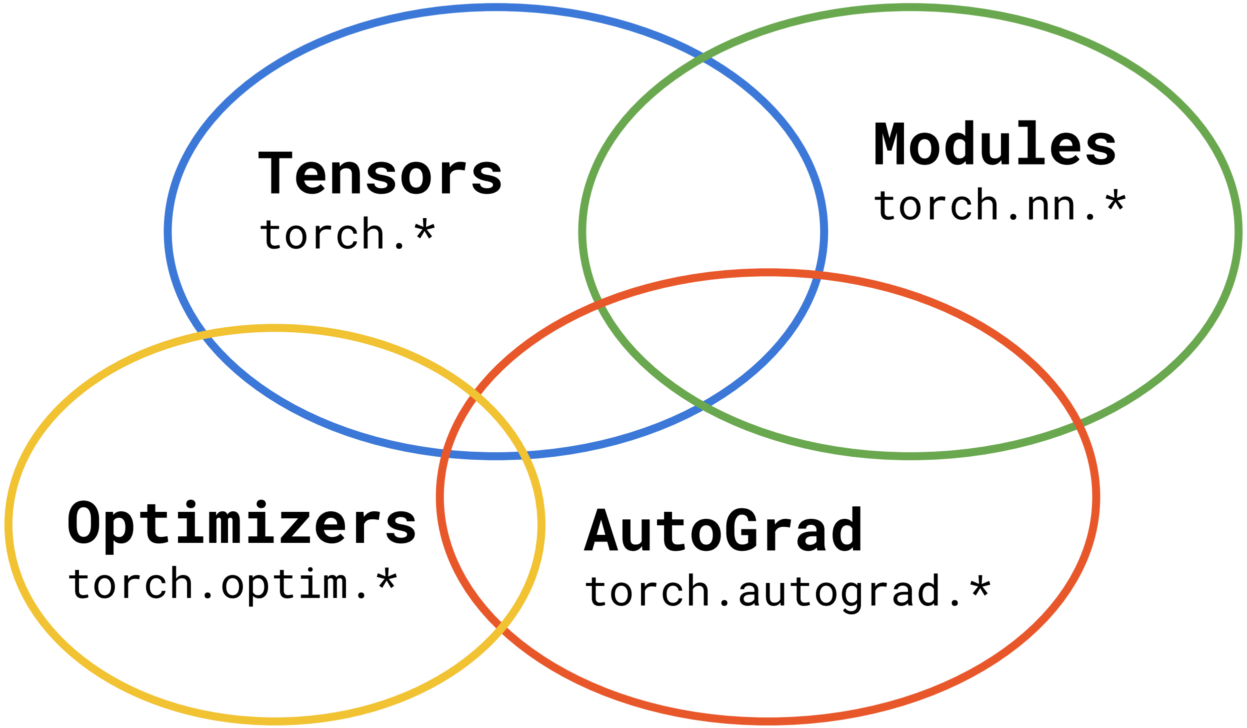
Core concepts



Core concepts



Core concepts



Tensors

```
All of deep learning
is computations on tensors,
which are generalizations of
a matrix that can be indexed in
more than 2 dimensions.
```

```
import torch

x = torch.tensor([1., 2., 3.])
y = torch.tensor([4., 5., 6.])
z = x + y
```

Tensors

```
Main tensor attributes:
* Shape
* Data Type
* Device



import torch

x = torch.tensor([1., 2., 3.])
```

Tensors

```
Main tensor attributes:



x = torch.tensor([1., 2., 3.])

x.shape # or x.size()
torch.Size([3])

x.dtype
torch.float32

x.device
device(type='cpu')
```

Modules

Modules are building blocks of neural network models.

From the simplest nn layer to a complex high-level model, they’re all called modules.

```
import torch.nn as nn


conv1 = nn.Conv2d(1, 20, 5)
conv2 = nn.Conv2d(20, 20, 5)
```

Modules

Modules can also contain other Modules, allowing to nest them in a tree structure.

```
import torch.nn as nn

class Model(nn.Module):
    ...
```

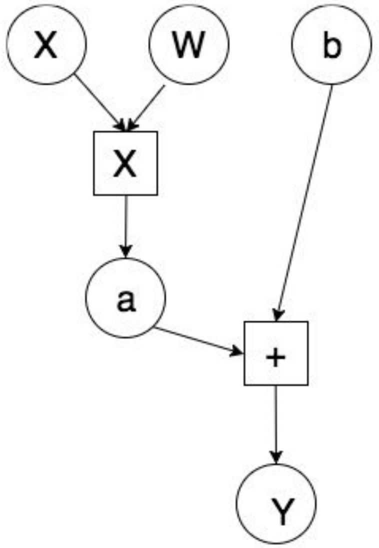
Modules

```
import torch.nn as nn


class Model(nn.Module):

  def __init__(self):

    super(Model, self).__init__()

    # submodules
    self.conv1 = nn.Conv2d(1, 20, 5)
    self.conv2 = nn.Conv2d(20, 20, 5)
```

Modules

```
import torch.nn as nn
import torch.nn.functional as F


class Model(nn.Module):

  def __init__(self):

    super(Model, self).__init__()
    self.conv1 = nn.Conv2d(1, 20, 5)
    self.conv2 = nn.Conv2d(20, 20, 5)


  def forward(self, x):

    x = F.relu(self.conv1(x))
    return F.relu(self.conv2(x))
```

AutoGrad, DCG

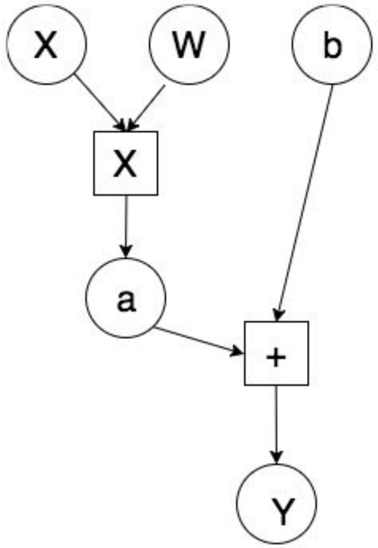


We won't write the back propagation gradients ourselves!

AutoGrad does it by creating a dynamic computation graph while we are doing forward computation/propagation.

But how?

AutoGrad, DCG

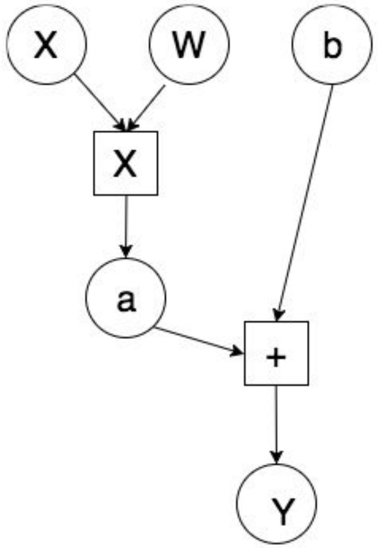


A computation graph is simply a specification of how your data is combined to give you the output.

Since the graph totally specifies what parameters were involved with which operations, it contains enough information to compute derivatives.

Do all tensors know how they are created/computed?

AutoGrad, DCG


$$\begin{tikzcd}
X \arrow[dr] & & W \arrow[dl] & b \arrow[ddd] \\
 & \boxed{\times} \arrow[d] & & \\
 & a \arrow[drr] & & \\
 & & & \boxed{+} \arrow[d] \\
 & & & Y
\end{tikzcd}$$

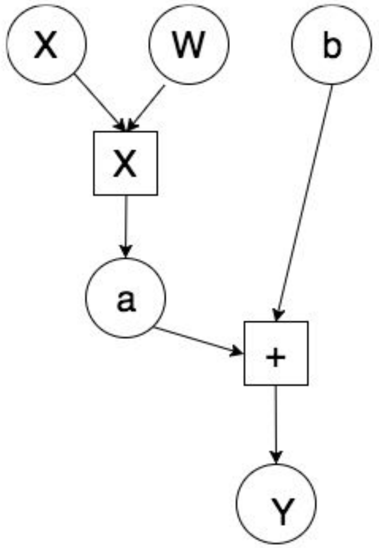
A computation graph is simply a specification of how your data is combined to give you the output.

Since the graph totally specifies what parameters were involved with which operations, it contains enough information to compute derivatives.

Do all tensors know how they are created/computed?

No, not all of them!

AutoGrad, DCG

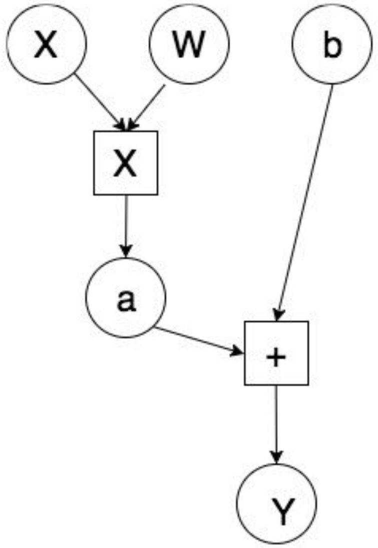


Tensors have a fundamental flag called requires_grad.

If requires_grad=True, the tensor object keeps track of how it was created.

```
X = torch.tensor([1., 2])
W = torch.tensor([3., 4], requires_grad=True)
b = torch.tensor([5.], requires_grad=True)

a = X * W

print(a)
# tensor([3., 8.], grad_fn=<MulBackward0>)
```

AutoGrad, DCG

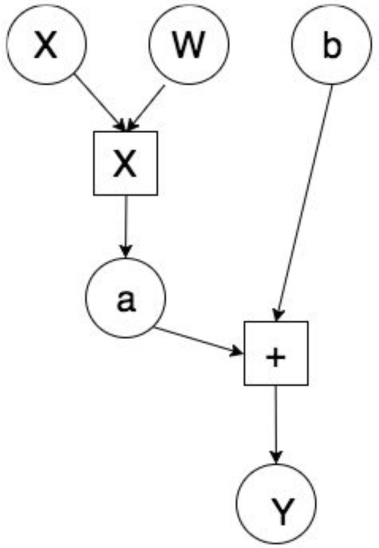


Tensors have a fundamental flag called requires_grad.

If requires_grad=True, the tensor object keeps track of how it was created.

```
X = torch.tensor([1., 2])
W = torch.tensor([3., 4], requires_grad=True)
b = torch.tensor([5.], requires_grad=True)

a = X * W

Y = a + b

print(Y)
# tensor([8., 13.], grad_fn=<AddBackward0>)
```

AutoGrad, DCG

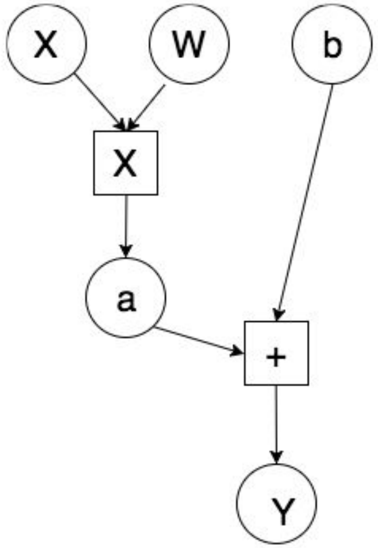


By calling backward() method on a tensor t, autograd calculates gradient of t w.r.t. Its computation graph leaves.

Those who were requiring grad, will have gradients accumulated within their grad attributes!

```
loss = Y.sum()

loss.backward()

print(W.grad)
# tensor([1., 2.])
```

Optimizers



Optimizers use the *grad* accumulated in tensors to change their values.

```
optimizer = torch.optim.SGD([W, b], lr=0.01)

optimizer.step()


print(W)
# tensor([2.9900, 3.9800], requires_grad=True)
```

**“Talk is cheap.
Show me the code.”**

- Linus Torvalds

Let’s see PyTorch
In action.