Carnegie Mellon University

Introduction to PyTorch

10-301/10-601 Introduction to Machine learning

Nov 10, 2023

(Credit to 16-824 Spring ‘22, Fall ‘23 TAs - modified by ML Fall ‘23 TAs)

Configure Colab with PyTorch

- Covered at the end of HW6 recitation (recording in canvas) (colab link)

```
[ ] import torch
```

The following command will return `True` if we have a device that supports CUDA, which in our case is the T4 GPU.

```
[ ] torch.cuda.is_available()

    True
```

This command tells us how many GPUs we have available.

```
[ ] torch.cuda.device_count()

    1
```

This command tells us the name of the GPU that we are using.

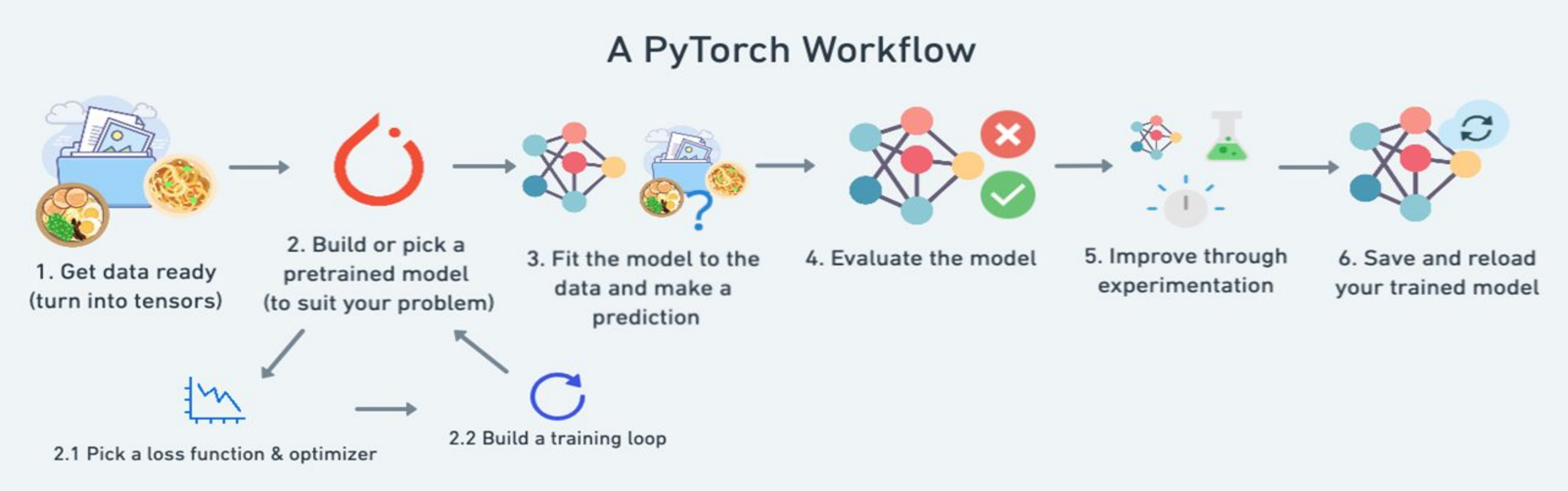
```
[ ] torch.cuda.get_device_name(0)

    'Tesla T4'
```

Contents

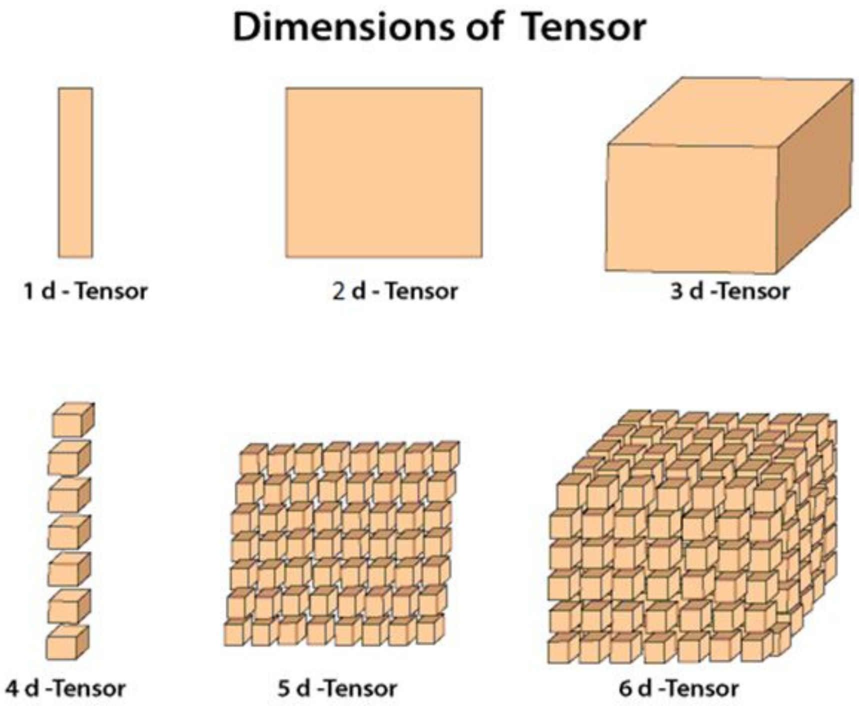
What is PyTorch?

- An open source machine learning framework that accelerates the path from research prototyping to production deployment.
- A library of deep learning modules/functions/losses/optimizers, etc.



Tensors: Backbone of PyTorch

- Multi-dimensional array, same as numpy array
- Biggest difference: Tensors can be run on CPU/GPU




```
import torch
import numpy as np
a_np = np.zeros((32,32))
a_torch = torch.from_numpy(a_np)
a_np = a_torch.numpy()
```

Basic elements: Tensor

- torch.Tensor
- Just like numpy array

Create from list

```
import torch

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

Create from numpy array

```
import torch
import numpy as np

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

Basic elements: Tensor

- Create tensor

```python
import torch

a = torch.ones(3, 3)
a = torch.zeros(3, 3)
a = torch.randn(3, 3)
```

Basic elements: Tensor

- Indexing and slicing

```
import torch

tensor = torch.ones(4, 4)
tensor[:, 1] = 0
tensor[1:2, 3:-1] = 2
```

Basic elements: Tensor

- device and type

```
import torch

a = torch.ones(3,3, dtype=torch.float32)
a.device     # cpu
a.dtype      # torch.float32
b = torch.ones_like(a)
b.to("cuda")
b.to(torch.int32)
b.to(a.device)
b.to(a.dtype)
```

Basic elements: Tensor

- Tensor operation
- Add/subtract/multiply/matrix multiply/transpose/expand …
- Support operation in batch

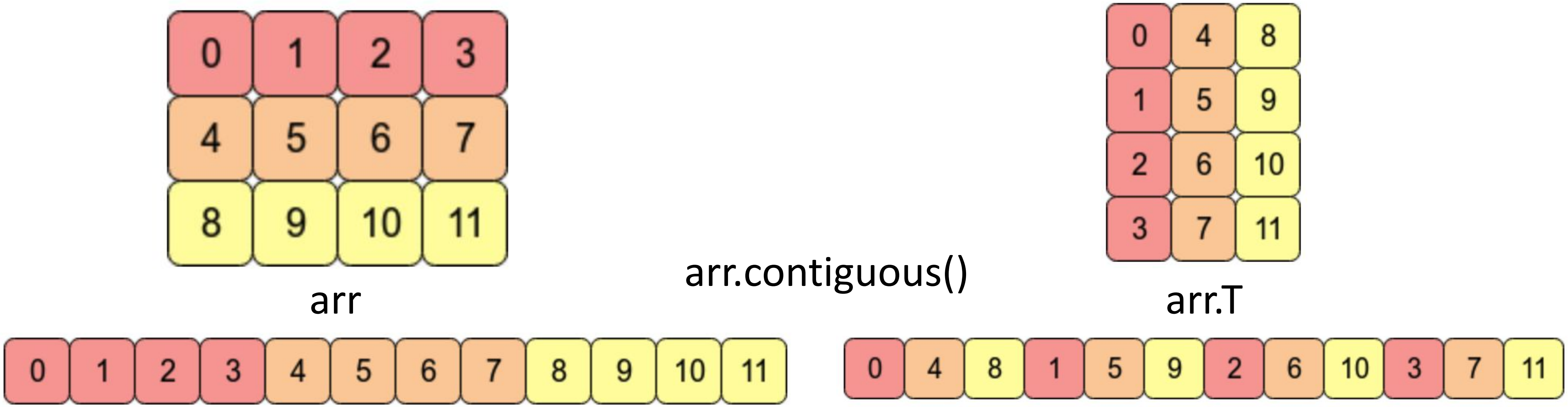
```python
import torch

a = torch.ones(2, 3, 1)
b = 2*torch.ones(2, 3, 1)
c1 = a + b  # [[[3], [3], [3]], [[3], [3], [3]]]
c2 = a - b  # [[[-1], [-1], [-1]], [[-1], [-1], [-1]]]
d = a * b   # [[[2], [2], [2]], [[2], [2], [2]]]
e = a.transpose(1, 2)   # e.size(): [2, 1, 3]
f = b @ e   # 2*torch.ones(2, 3, 3)
g = a.expand(-1, -1, 3)     # torch.ones(2, 3, 3)
```

Basic elements: Tensor

- Tensor.Contiguous()

```
arr = np.arange(12).reshape(3,4)
```



Credit:https://stackoverflow.com/questions/26998223/what-is-the-difference-between-contiguous-and-non-contiguous-arrays/26999092#26999092

Model Definition: nn.Module

- Defining a MLP
- Define basic operations
- Define forward functions
- A nn.Module.parameters() returns trainable parameters

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

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.linear(128, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)

        return x

net = Net()
print(net)
```

Model Definition: nn.Module

- Customize your layer(network)
- Declare param
- Initialize params
- Define operations

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

class MyLinear(nn.Module):

    def __init__(self, in_ch, out_ch):
        super(MyLinear, self).__init__()
        self.W = nn.Parameter(torch.zeros(in_ch, out_ch), requires_grad=True)
        self.b = nn.Parameter(torch.zeros(out_ch), requires_grad=True)

        ## initialize weights
        nn.init.xavier_uniform_(self.W)
        nn.init.zeros_(self.b)

    def forward(self, x):
        return (x[:, :, None] @ self.W[None, :, :])[..., 0] + self.b[None]
```

Training Loop

```
running_loss = 0.0
for i, data in enumerate(trainloader, 0):
    # get the inputs; data is a list of [inputs, labels]
    inputs, labels = data

    # zero the parameter gradients
    optimizer.zero_grad()

    # forward + backward + optimize
    outputs = net(inputs)
    loss = criterion(outputs, labels)
    loss.backward()
    optimizer.step()
```

Validation

- On the test dataset periodically run to look at accuracy/loss
- Set model.eval() to deactivate all the layers from updating
- Run with torch.no_grad to deactivate gradients

```python
@torch.no_grad()
def add_ab(a,b):
    return a+b
```

Key metrics to track

- Train loss / accuracy
- Validation loss / accuracy

<u>General rule of thumb:</u>

- Train loss low, Validation loss low: things are working
- Train loss low, Validation loss high: overfitting
- Train loss high, Validation loss high: underfitting
- Train loss high, Validation loss low: some bug in evaluation

Summary of training a NN

1. Load data
   - DataLoader for batching, shuffling
2. Define Forward (of the neural network)
   - Implement nn.Module
3. Define loss
   - Pytorch provides a lot of these if needed
4. Define Backward
   - PyTorch automatically computes gradients
5. Optimizer to update the given parameters
   - torch.optim
6. Track key metrics