PyTorch and Neural Nets

CS285 Deep RL
Instructor: Kyle Stachowicz

[Adapted from Marwa Abdulhai’s CS285 Fa22 Slides]
PyTorch Tutorial (Colab)

https://colab.research.google.com/drive/12nQiv6aZHXXNuCfAAuTjJenDWKQblt2Mz

Goal of this course

Train an agent to perform useful tasks

\[ \pi_\theta(a|s), \quad \tilde{\Delta}_{t+1} = f_\theta(s_t, a_t) \]

\[ Q_\theta(s, a) \]

train the model

data

agent

collect data
Goal of this course

Train an agent to perform useful tasks

$\pi_\theta(a|s)$

$\hat{\Delta}_{t+1} = f_\theta(s_t, a_t)$

$Q_\theta(s, a)$

focus for today's lecture!
How do train a model?

\[ \theta^* = \text{arg min}_\theta \sum_{(x,y) \in D} L(f_\theta(x), y) \]

PyTorch does all of these!
What is PyTorch?

Python library for:
• Defining neural networks
• Automating computing gradients
• And more! (datasets, optimizers, GPUs, etc.)
How does PyTorch work?

You define:

\[ h_1 = \sigma(W_1 x) \quad h_2 = \sigma(W_2 h_1) \quad y = \sigma(W_3 h_2) \]

PyTorch computes:

\[ \frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial W_1} \quad \frac{\partial y}{\partial W_2} = \frac{\partial y}{\partial h_2} \frac{\partial h_2}{\partial W_1} \quad \frac{\partial y}{\partial W_3} \]
NumPy

• Fast CPU implementations
• CPU-only
• No autodiff
• Imperative

PyTorch

• Fast CPU implementations
• Allows GPU
• Supports autodiff
• Imperative

Other features include:
• Datasets and dataloading
• Common neural network operations
• Built-in optimizers (Adam, SGD, …)
The Basics

**Python**

```python
arr_a = [1, 3, 4, 5, 9]
arr_b = [9, 5, 7, 2, 5]

# Element-wise operations
list_sum = [a + b for a, b in zip(list_a, list_b)]
list_prod = [a * b for a, b in zip(list_a, list_b)]
list_doubled = [2 * a for a in list_a]

# Indexing
value = list_a[3]
list_slice = list_a[2:3]

arr_idx = [3, 2, 1]
arr_indexed = [arr_a[i] for i in arr_idx]
```

**NumPy**

```python
import numpy as np

arr_a = np.array([1, 3, 4, 5, 9])
arr_b = np.array([9, 5, 7, 2, 5])

# Element-wise operations
arr_sum = arr_a + arr_b
arr_prod = arr_a * arr_b
arr_doubled = 2 * arr_a

# Indexing
value = arr_a[3]
list_slice = arr_a[2:3]

arr_idx = np.array([3, 2, 1])
arr_indexed = arr_a[arr_idx]
```

**PyTorch**

```python
import torch

tensor_a = torch.tensor([1, 3, 4, 5, 9])
tensor_b = torch.tensor([9, 5, 7, 2, 5])

# Element-wise operations
tensor_sum = tensor_a + tensor_b
tensor_prod = tensor_a * tensor_b
tensor_doubled = 2 * tensor_a

# Indexing
value = tensor_a[3]
tensor_slice = tensor_a[2:3]

tensor_idx = torch.tensor([3, 2, 1])
tensor_indexed = tensor_a[tensor_idx]
```

100x faster!
Multidimensional Arrays
**Multidimensional Indexing**

The table `A` has a shape of `(3, 5)`. The elements are:

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>27</th>
<th>5</th>
<th>54</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99</td>
<td>4</td>
<td>23</td>
<td>3</td>
<td>57</td>
</tr>
<tr>
<td>1</td>
<td>76</td>
<td>42</td>
<td>34</td>
<td>82</td>
<td>5</td>
</tr>
</tbody>
</table>

A.shape == (3, 5)
Multidimensional Indexing

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>27</th>
<th>5</th>
<th>54</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>4</td>
<td>23</td>
<td>3</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>42</td>
<td>34</td>
<td>82</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

$A[0, 3]$
# Multidimensional Indexing

![Image of multidimensional indexing with PyTorch and NumPy logos]

Consider a 2D array `A`:

\[
A = \begin{bmatrix}
32 & 27 & 5 & 54 & 1 \\
99 & 4 & 23 & 3 & 57 \\
76 & 42 & 34 & 82 & 5
\end{bmatrix}
\]

To extract the elements at axis 1, we can use `A[:, 3]`:

\[
\begin{bmatrix}
54 \\
3 \\
82
\end{bmatrix}
\]
# Multidimensional Indexing

<table>
<thead>
<tr>
<th>Axis 0</th>
<th>Axis 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>99</td>
<td>4</td>
</tr>
<tr>
<td>76</td>
<td>42</td>
</tr>
</tbody>
</table>

\[
A[0, :] 
\]
### Multidimensional Indexing

```python
A[0, 2:4]
```

<table>
<thead>
<tr>
<th></th>
<th>32</th>
<th>27</th>
<th>5</th>
<th>54</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>99</td>
<td>4</td>
<td>23</td>
<td>3</td>
<td>57</td>
</tr>
<tr>
<td>1</td>
<td>76</td>
<td>42</td>
<td>34</td>
<td>82</td>
<td>5</td>
</tr>
</tbody>
</table>

Axis 0: 0, 1, 2, 3, 4

Axis 1: 0, 1, 2, 3, 4, 5

- 32
- 27
- 5
- 54
- 1

- 99
- 4
- 23
- 3
- 57

- 76
- 42
- 34
- 82
- 5

**PyTorch**

**NumPy**
# Multidimensional Indexing

A multidimensional array `A` with shape `(3, 5, 4)`

```
A = 
[[[32, 27, 5, 54, 1],
  [99, 4, 23, 3, 57],
  [76, 42, 34, 82, 5]],
  [[32, 27, 5, 54, 1],
  [99, 4, 23, 3, 57],
  [76, 42, 34, 82, 5]],
  [[32, 27, 5, 54, 1],
  [99, 4, 23, 3, 57],
  [76, 42, 34, 82, 5]]]
```

```
A.shape == (3, 5, 4)
```
Multidimensional Indexing

A

A[θ, ...]
Multidimensional Indexing

A

A[... , 1]
Broadcasting

TL;DR: Shape (1, 3, 2) acts like (6, 5, 4, 3, 2) when added to shape (6, 5, 4, 3, 2)

(Trailing dimensions will be matched, arrays will be repeated along matching dimensions)
Shape Operations

A = np.random.normal(size=(10, 15))

# Indexing with newaxis/None
# adds an axis with size 1
A[np.newaxis] # -> shape (1, 10, 15)

# Squeeze removes a axis with size 1
A[np.newaxis].squeeze(0) # -> shape (10, 15)

# Transpose switches out axes.
A.transpose((1, 0)) # -> shape (15, 10)

# !!! BE CAREFUL WITH RESHAPE !!!
A.reshape(15, 10) # -> shape (15, 10)
A.reshape(3, 25, -1) # -> shape (3, 25, 2)

A = torch.randn((10, 15))

# Indexing with None
# adds an axis with size 1
A[None] # -> shape (1, 10, 15)

# Squeeze removes a axis with size 1
A[None].squeeze(0) # -> shape (10, 15)

# Permute switches out axes.
A.permute((1, 0)) # -> shape (15, 10)

# !!! BE CAREFUL WITH VIEW !!!
A.view(15, 10) # -> shape (15, 10)
A.view(3, 25, -1) # -> shape (3, 25, 2)
Device Management

• Numpy: all arrays live on the CPU’s RAM
• Torch: tensors can either live on CPU or GPU memory
  • Move to GPU with .to("cuda")/.cuda()
  • Move to CPU with .to("cpu")/.cpu()

YOU CANNOT PERFORM OPERATIONS BETWEEN TENSORS ON DIFFERENT DEVICES!
device = torch.device("cuda")
x = torch.zeros((2, 3))
y = torch.ones((2, 3), device=device)
z = x + y

----> 4 z = x + y

RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!
Computing Gradients

```python
P = torch.randn((1024, 1024))
print(P.requires_grad) # -> False
P = torch.randn((1024, 1024), requires_grad=True)
b = torch.randn((1024,), requires_grad=True)
print(P.grad) # -> None
```
Computing Gradients

```python
P = torch.randn((1024, 1024))
print(P.requires_grad) # -> False
P = torch.randn((1024, 1024), requires_grad=True)
b = torch.randn((1024,), requires_grad=True)
print(P.grad) # -> None

x = torch.randn((32, 1024))
y = torch.nn.relu(x @ P + b)

target = 3
loss = torch.mean((y - target) ** 2).detach()
```
Training Loop

```python
net = (...).to("cuda")
dataset = ...
dataloader = ..
optimizer = ...
loss_fn = ..
for epoch in range(num_epochs):
    # Training..
    net.train()
    for data, target in dataloader:
        data = torch.from_numpy(data).float().cuda()
        target = torch.from_numpy(data).float().cuda()

        prediction = net(data)
        loss = loss_fn(prediction, target)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    net.eval()
    # Do evaluation..
```
Converting Numpy / PyTorch

Numpy -> PyTorch:

```
torch.from_numpy(numpy_array).float()
```

PyTorch -> Numpy:

• (If requires_grad) Get a copy without graph with .detach()
• (If on GPU) Move to CPU with .to("cpu")/.cpu()
• Convert to numpy with .numpy

All together:

```
torch_tensor.detach().cpu().numpy()
```
Custom networks

```python
import torch.nn as nn

class SingleLayerNetwork(nn.Module):
    def __init__(self, in_dim: int, out_dim: int, hidden_dim: int):
        super().__init__() # <- Don't forget this!
        self.net = nn.Sequential(
            nn.Module(in_dim, hidden_dim),
            nn.ReLU(),
            nn.Module(hidden_dim, out_dim),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)

batch_size = 256
my_net = SingleLayerNetwork(2, 32, 1).to("cuda")
output = my_net(torch.randn(size=(batch_size, 2)).cuda())
```

- Prefer `net()` over `net.forward()`
- Everything (network and its inputs) on the same device!!!
Torch Best Practices

• When in doubt, **assert** is your friend
  
  ```python
  assert x.shape == (B, N), \
  f"Expected shape ({B, N}) but got {x.shape}"
  ```

• Be extra careful with `.reshape/ .view`
  • If you use it, assert before and after
  • Only use it to collapse/expand a single dim
  • In Torch, prefer `.flatten()/.permute()/.unflatten()`

• If you do some complicated operation, test it!
  • Compare to a pure Python implementation
Torch Best Practices (continued)

• Don’t mix numpy and Torch code
  • Understand the boundaries between the two
  • Make sure to cast 64-bit numpy arrays to 32 bits
  • torch.Tensor only in nn.Module!

• Training loop will always look the same
  • Load batch, compute loss
  • .zero_grad(), .backward(), .step()