PyTorch + Gym Tutorial

Owen Oertell

CS 4/5789: Introduction to Reinforcement Learning

Outline

- What is PyTorch?
- Installing PyTorch
- Tensors, Shapes, Using the CPU vs GPU
- Gradients
- Defining and Training a NN
- Gym Environments

What is PyTorch?

Neural network

PyTorch handles everything!

$$\theta^* = \arg\min_{f \in \mathcal{F}} \sum_{(x,y) \in \mathcal{D}} \mathcal{L}(f(x), y)$$

Gradient descent

Dataset

Loss function

What is Pytorch?

- PyTorch is a framework that…
  - Lets you define neural networks
  - Automatically computes gradients
  - Handles datasets
  - Manages GPUs
  - … and more

Installing Pytorch

Go to the website: https://pytorch.org/get-started/locally/

Select your version, os, package manager, etc.

And install

PyTorch Build	Stable (2.6.0)			Preview (Nightly)	
Your OS	Linux	Mac		Windows	
Package	Conda	Pip		LibTorch	Source
Language	Python			C++ / Java	
Compute Platform	~~CUDA 11.8~~	~~CUDA 12.4~~	~~CUDA 12.6~~	~~ROCm 6.2.4~~	Default
Run this Command:	`pip3 install torch torchvision torchaudio`				

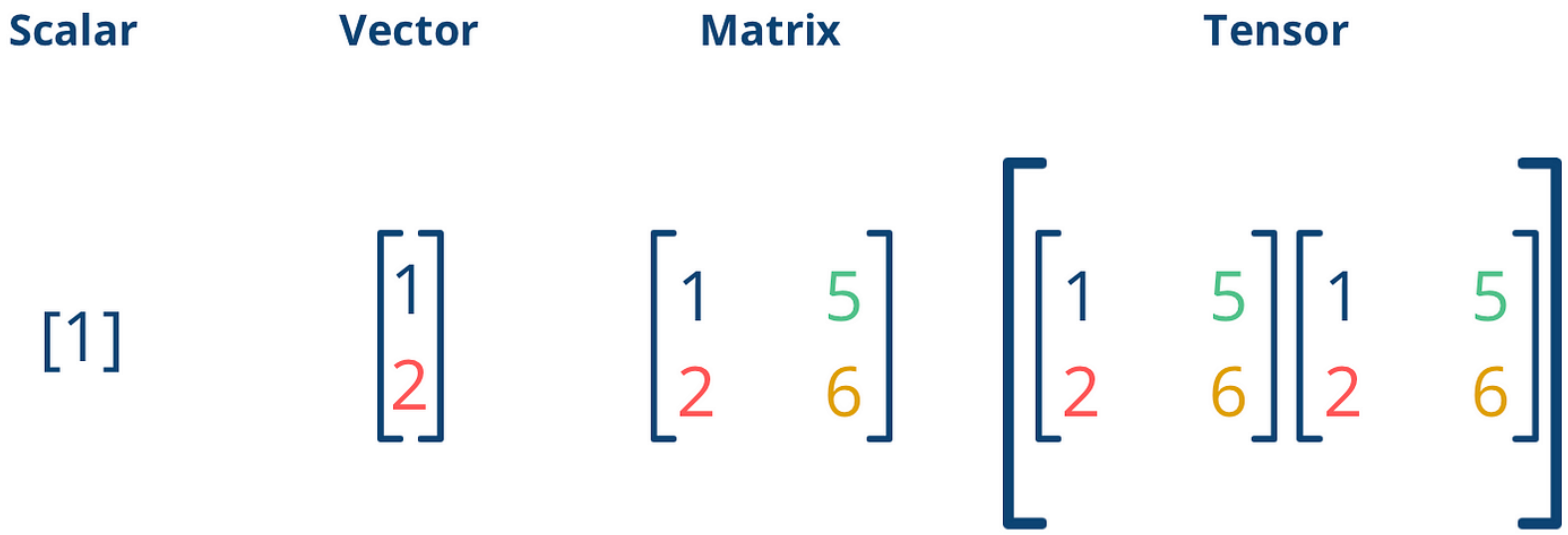
Installing Pytorch

- If you have an NVIDIA GPU, make sure that you install the right version, by checking your version of NCCL with `nvcc -V` or `nvidia-smi`

```
ojo2@computer:/path$ nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Feb__7_19:32:13_PST_2023
Cuda compilation tools, release 12.1, V12.1.66
Build cuda_12.1.r12.1/compiler.32415258_0
```

- The astute observer would realize the there is no latest torch for cu12.1, so you'd need to get an older version.

Tensors & Shapes



Tensors & Shapes

Unlike lists of lists,
tensors cannot be jagged!

Axis 1

A

Axis 0

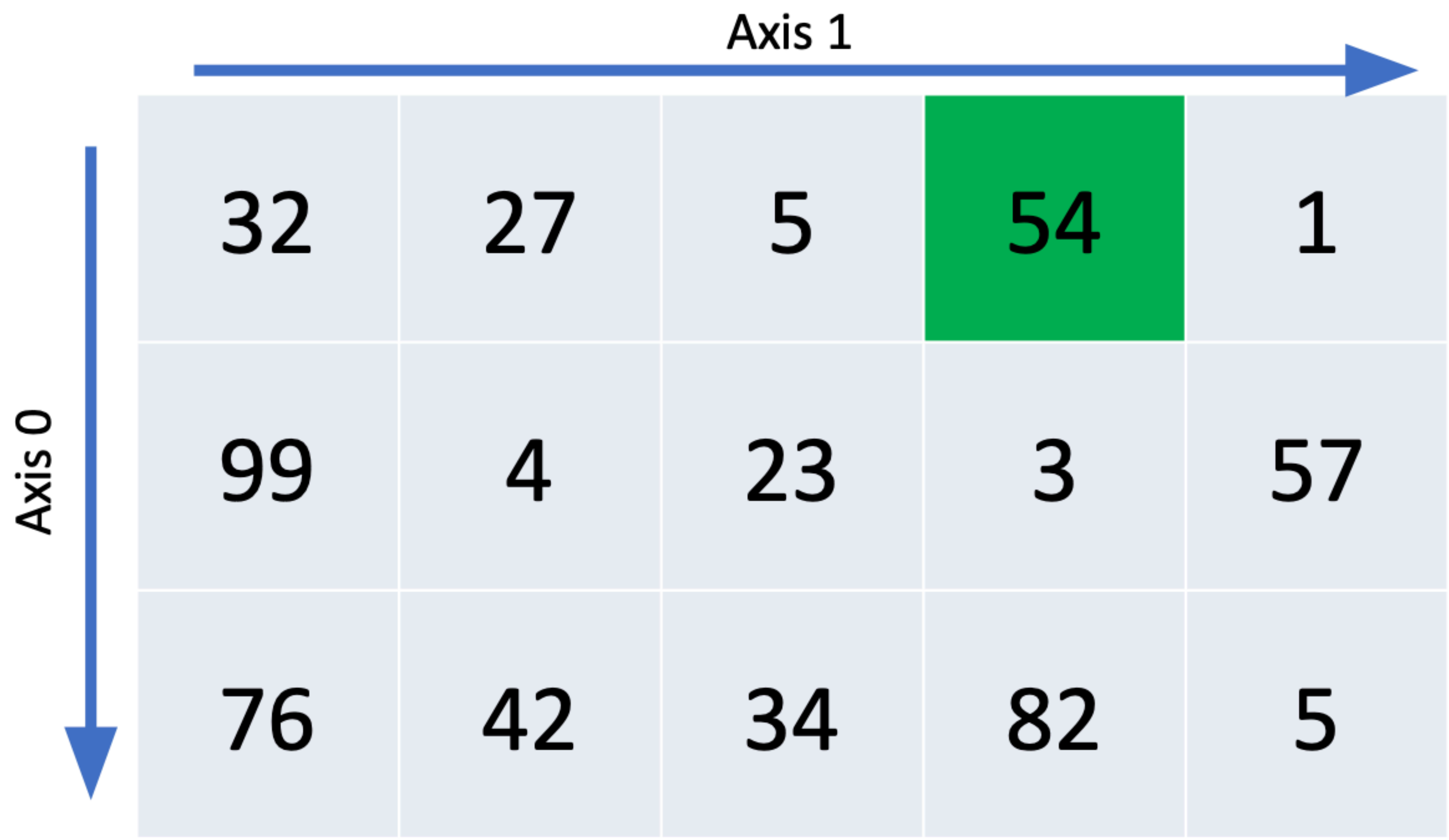
32	27	5	54	1
99	4	23	3	57
76	42	34	82	5

`A.shape == (3, 5)`

* slide credit, Stachowicz, CS285

Tensors & Shapes

Unlike lists of lists,
tensors cannot be jagged!

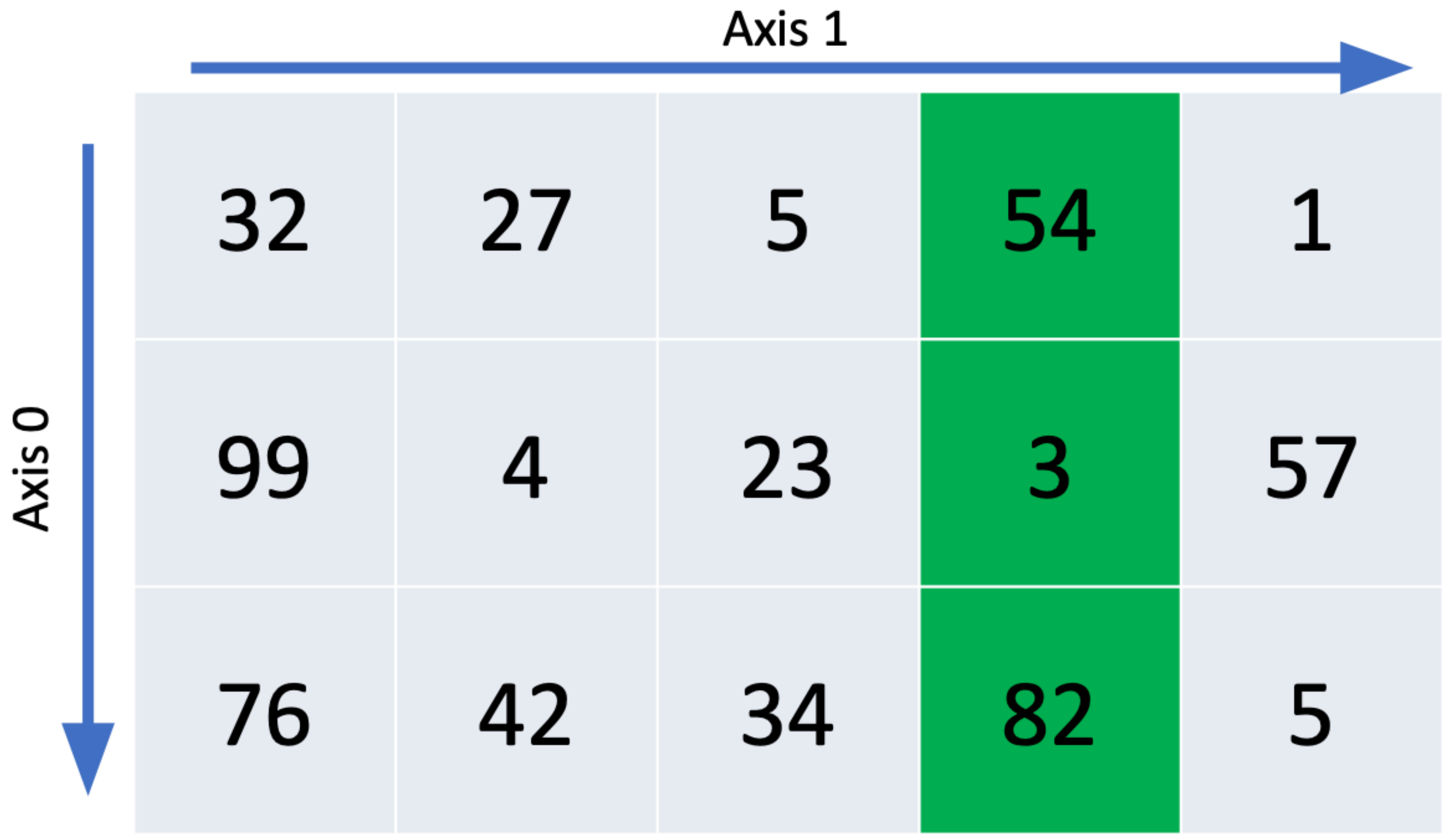



```
A[0, 3]
```

* slide credit, Stachowicz, CS285

Tensors & Shapes

Unlike lists of lists,
tensors cannot be jagged!



`A[:, 3]`

* slide credit, Stachowicz, CS285

Tensors & Shapes

Unlike lists of lists,
tensors cannot be jagged!

Axis 1

Axis 0

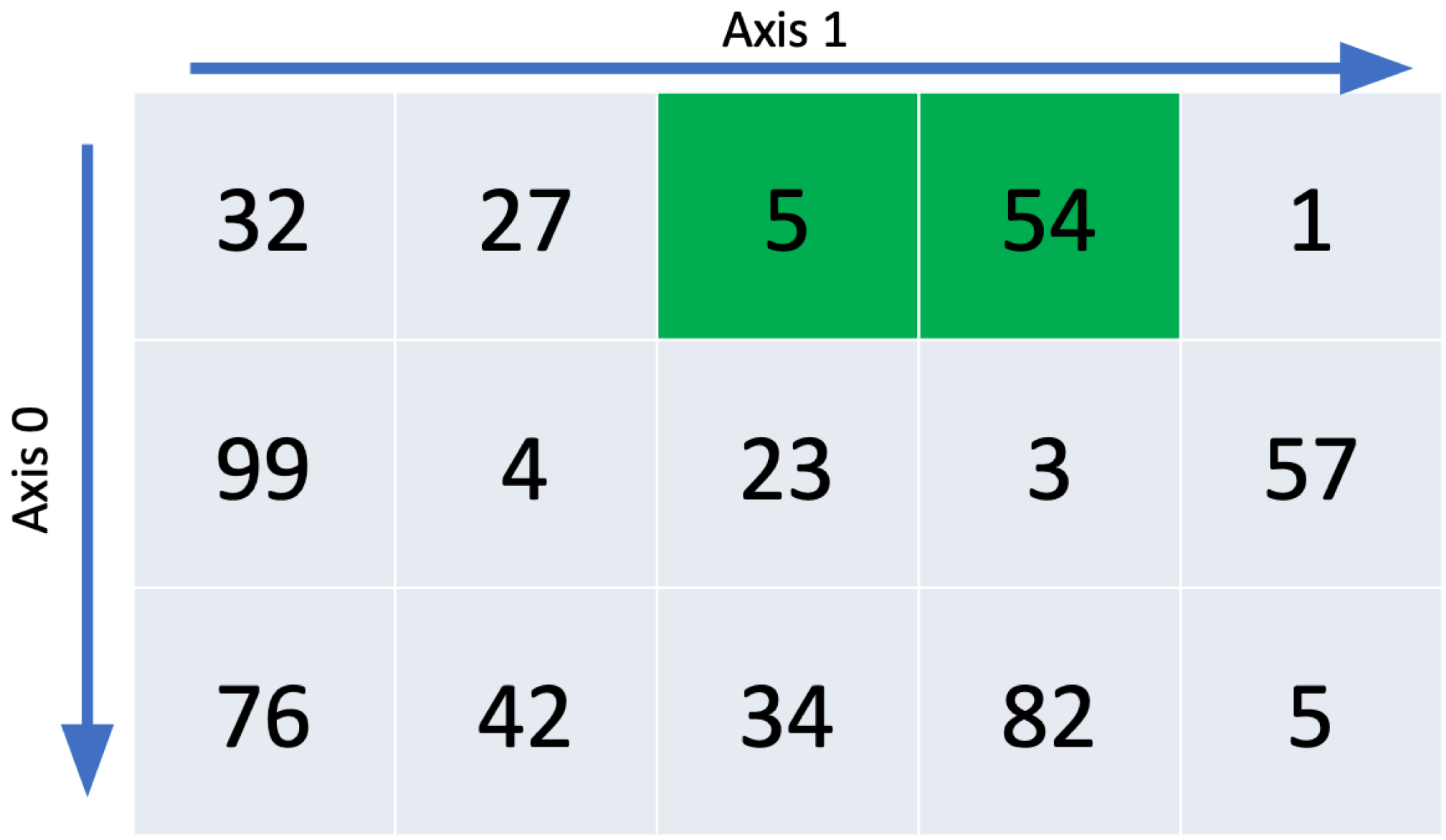
32	27	5	54	1
99	4	23	3	57
76	42	34	82	5

```
A[0, :]
```

* slide credit, Stachowicz, CS285

Tensors & Shapes

Unlike lists of lists,
tensors cannot be jagged!

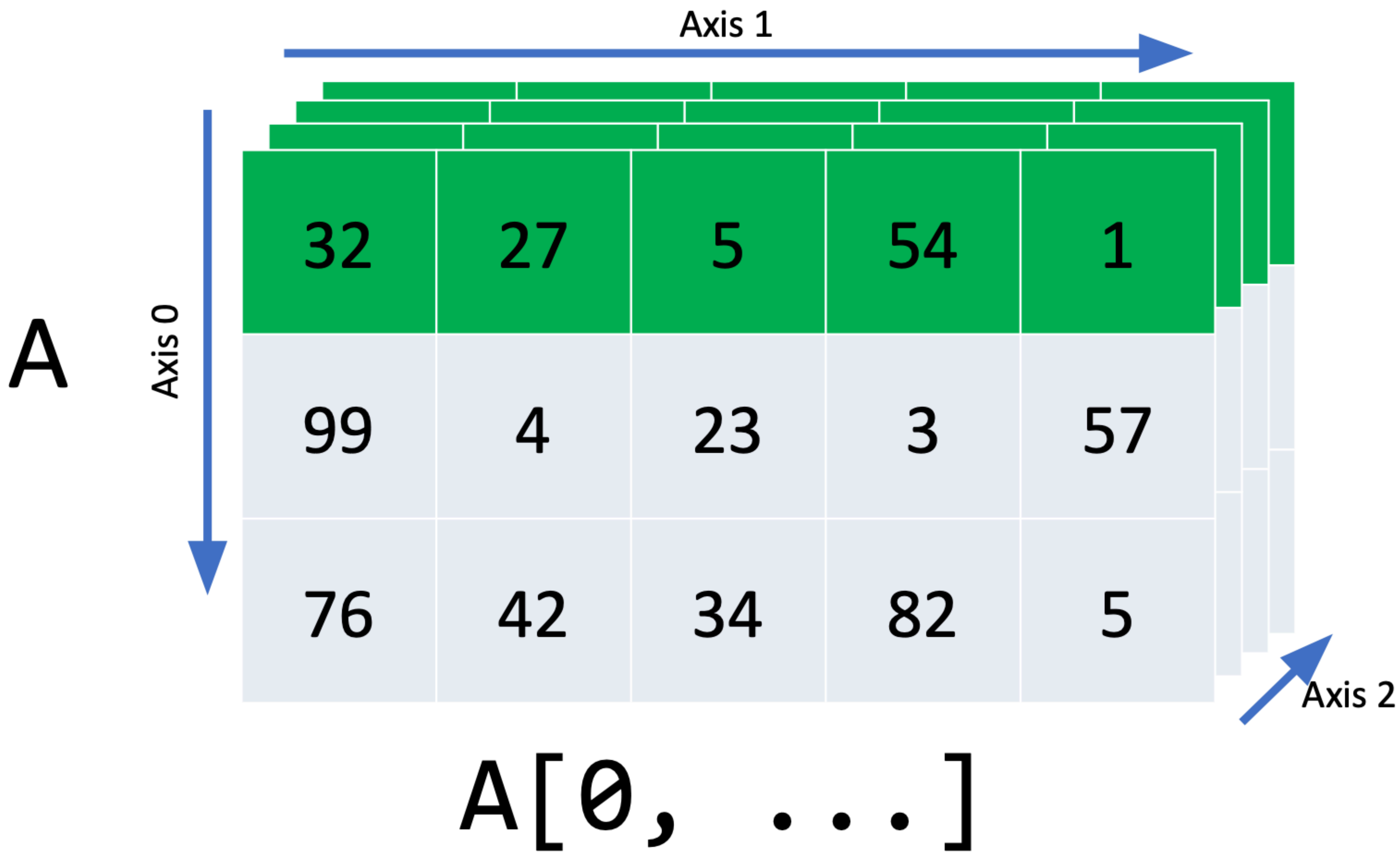



```
A[0, 2:4]
```

* slide credit, Stachowicz, CS285

Tensors & Shapes

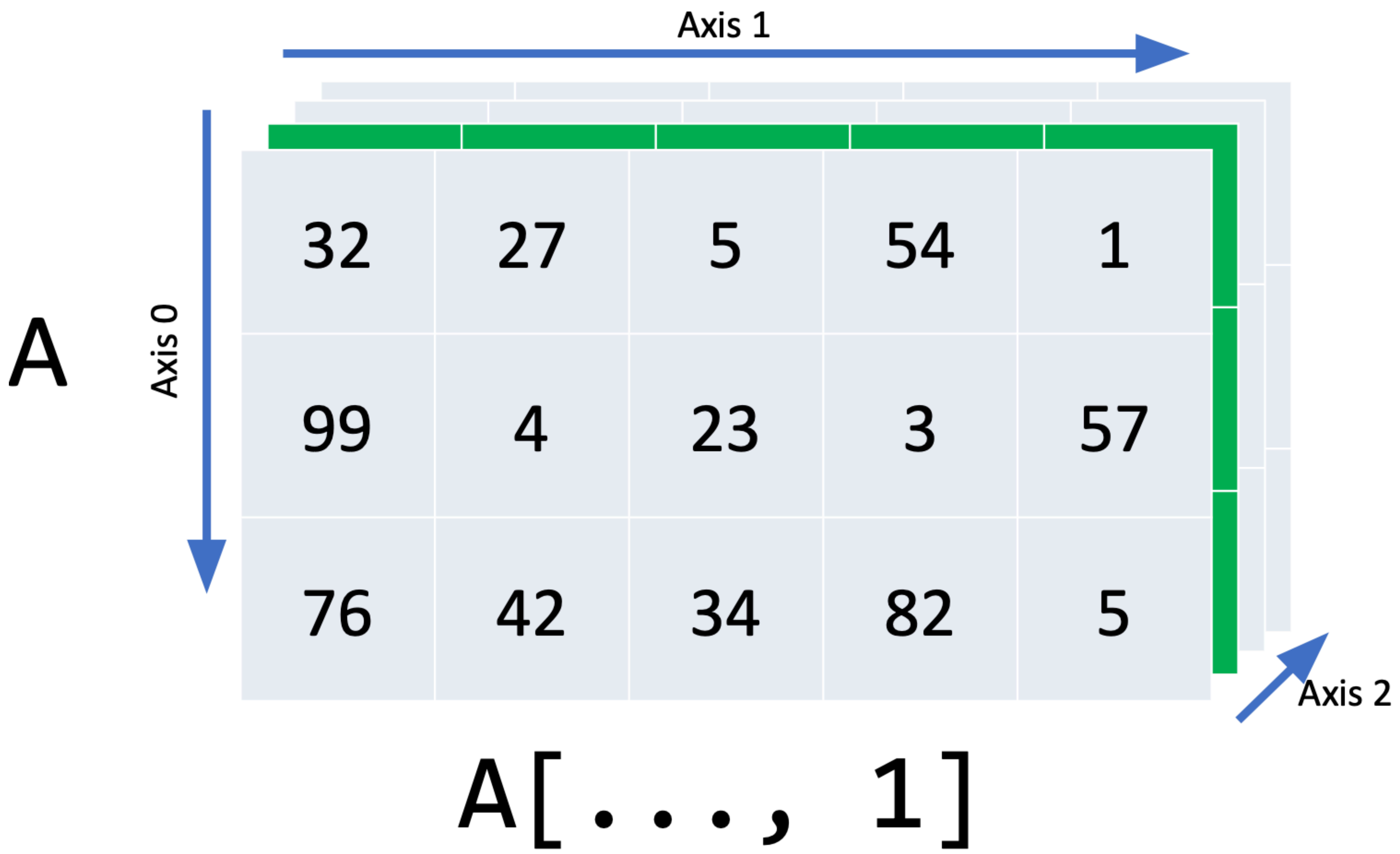
Unlike lists of lists,
tensors cannot be jagged!



* slide credit, Stachowicz, CS285

Tensors & Shapes

Unlike lists of lists,
tensors cannot be jagged!



* slide credit, Stachowicz, CS285

Tensors & Shapes

Tensors follow expected rules for operations (same for NumPy)

```
# tensor operations
z = x + y # element-wise addition
z = x - y # element-wise subtraction
z = x * y # element-wise multiplication # !! not matrix multiplication
z = x / y # element-wise division
z = x @ y # matrix multiplication or z = torch.matmul(x, y)
```

* slide credit, Stachowicz, CS285

Tensors & Shapes

NumPy

```python
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)
```

PyTorch

```python
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)
```

Note: torch also has reshape, but it modifies the underlying data structure, views don’t

* slide credit, Stachowicz, CS285

Tensors & Shapes

```
import torch

x = torch.randn(100,50,5) # [100,50,5]
index_tensor = torch.tensor([1,2,3])

print(x[index_tensor].shape)
```

Guess the shape!

Device Management

- When you have a GPU, there become 2 places tensors can live (for torch)
  - CPU: We send to cpu with `.to("cpu")`/`.cpu()`
  - GPU: We send to gpu with `.to("cuda")`/`.cuda()`
- NumPy arrays always live on the CPU

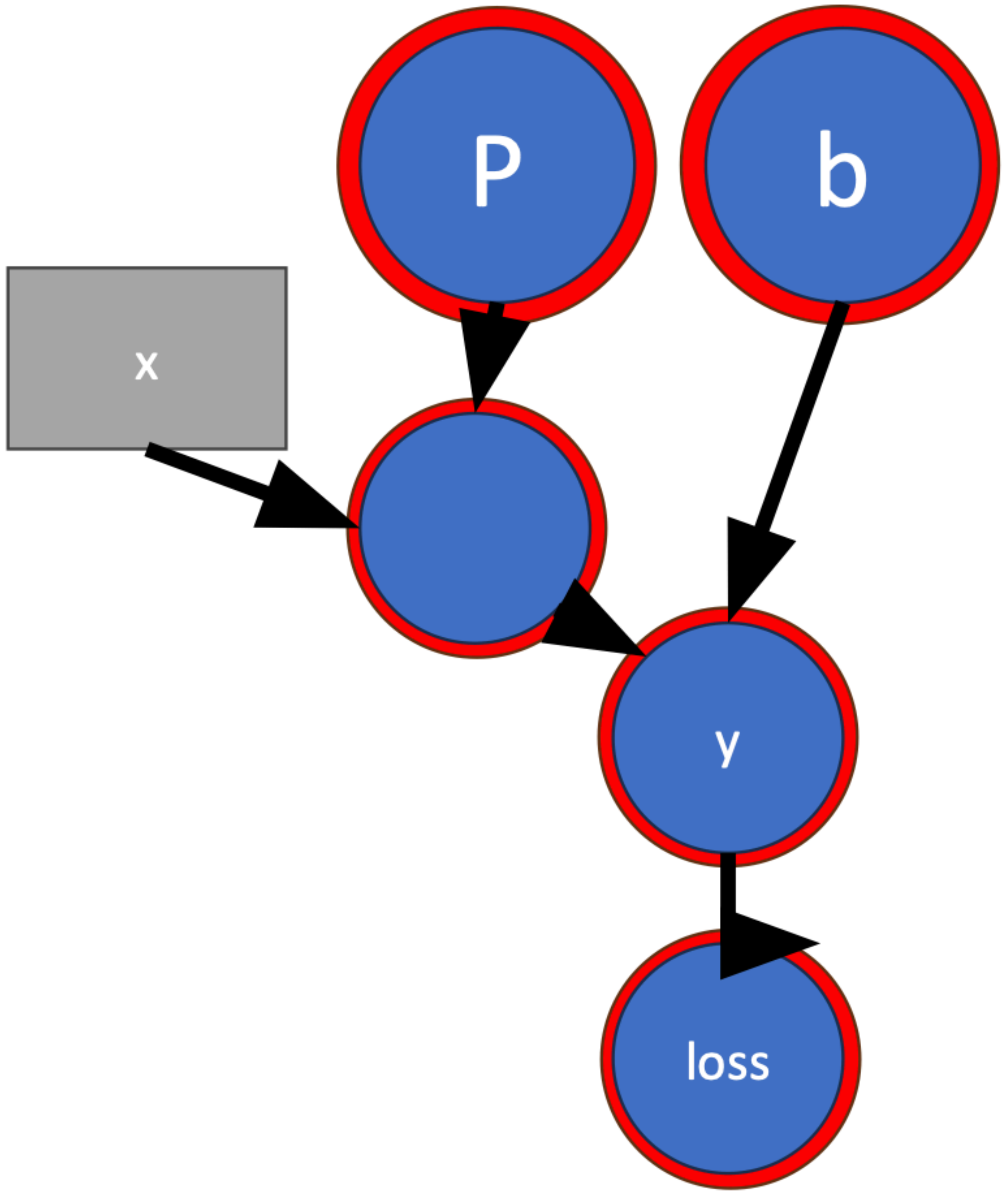
You can't perform operations between tensors on different devices!

Gradients

The gradients for backpropagation are organized in a graph of the operations. You can see the edge in the graph by printing tensors with require_grad=True

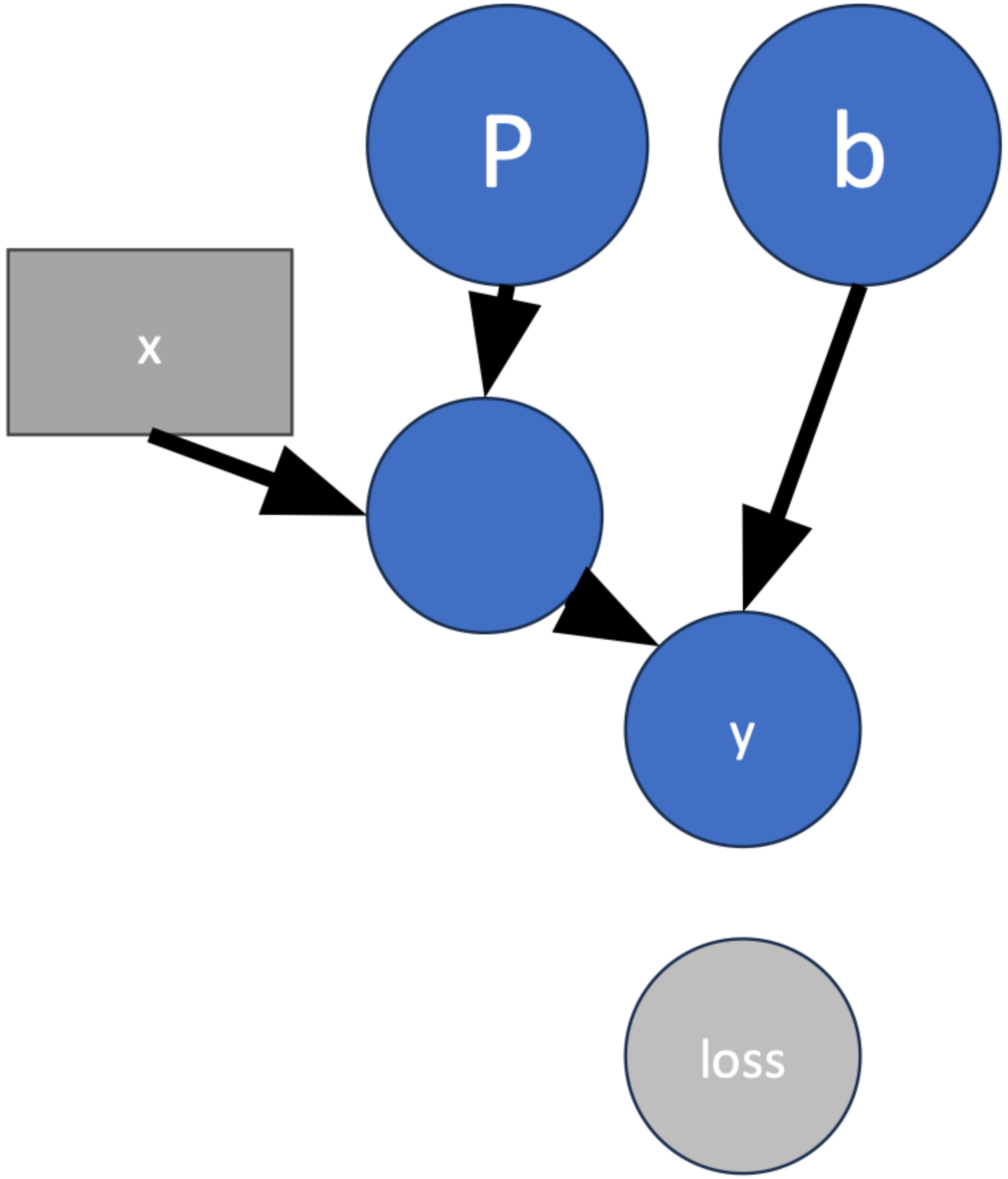
```
x = torch.randn(10,50, requires_grad=True)
x = x.sum()
print(x)
```

```
tensor(-20.7075, grad_fn=<SumBackward0>)
```



Gradients

You can disrupt the graph by using `.detach()`



Training Pipeline

(1) Define NN

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

class Model(nn.Module):

    def __init__(self):
        super().__init__()

        self.layer1 = nn.Linear(10, 10)
        self.layer2 = nn.Linear(10, 10)

    def forward(self, x):
        x = self.layer1(x)
        x = F.relu(x)
        x = self.layer2(x)
        return x
```

define layers

Define forward pass

Many other prebuilt types of layers. Check out https://pytorch.org/docs/stable/index.html

(2) Define a Dataset

Subclass torch dataset class

Override length and getitem (required)

Define a collate function (needed for data loaders)

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

class Dataset(torch.utils.data.Dataset):
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

    def collate_fn(self, batch):
        x = torch.stack([item[0] for item in batch])
        y = torch.stack([item[1] for item in batch])
        return x, y
```

(3) Putting it together!

```python
x = torch.randn(100, 10) # [batch_size, dim]
y = torch.randn(100,1) # [batch_size, pred_dim]
dataset = Dataset(x, y)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=10, shuffle=True)

model = Model()
# put the model to the GPU (if available)
model = model.to("cuda")

# define a loss function (could be manual)
criterion = nn.MSELoss()

# define an optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# train the model
for epoch in range(100):
    for batch in dataset:
        x, y = batch
        x = x.to("cuda")
        y = y.to("cuda")

        optimizer.zero_grad()
        pred = model(x)
        loss = criterion(pred, y)
        loss.backward()

        # print the loss
        print(f"Epoch {epoch}, Loss: {loss.item()}")

        # update the model parameters
        optimizer.step()
```

Data loader batches your dataset and makes it iterable

Define an optimizer (to do gradient descent)

Put your data to GPU! Important!

Remove stored gradients from the model

Forward pass

Compute loss

Compute gradients, but don't do update yet!

Update with gradient descent

(4) Save the Model

```
# save model
torch.save(model.state_dict(), "model.pth")

# load model
model = Model()
model.load_state_dict(torch.load("model.pth"))
```

Parameters

- To access the parameters of a model (which you will need to do in PA2), you can iterate over them as follows
- This will give you both the weights and biases for each param group (tensor)

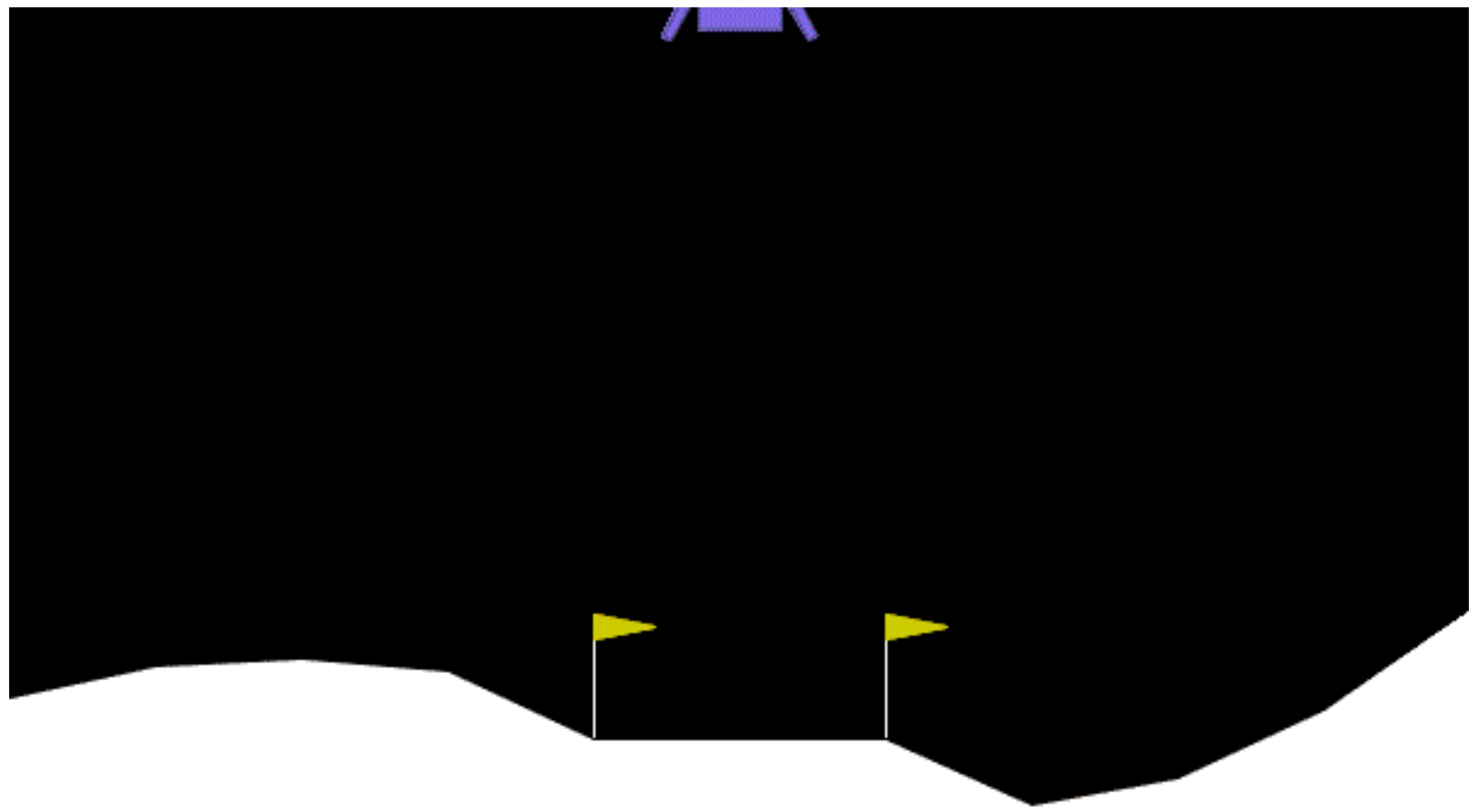
```
model = Model()
for param in model.parameters():
    print(param.data)
```

More Resources

- There are many resources on PyTorch:
  - The docs (https://pytorch.org/docs/stable/index.html)
    - Don’t just assume something does what you think because of the function name, read the description!
  - Tutorials (https://pytorch.org/tutorials/)
  - For a good comprehensive tutorial (https://colab.research.google.com/drive/12nQiv6aZHXNuCfAAuTjJenDWKQbIt2Mz)

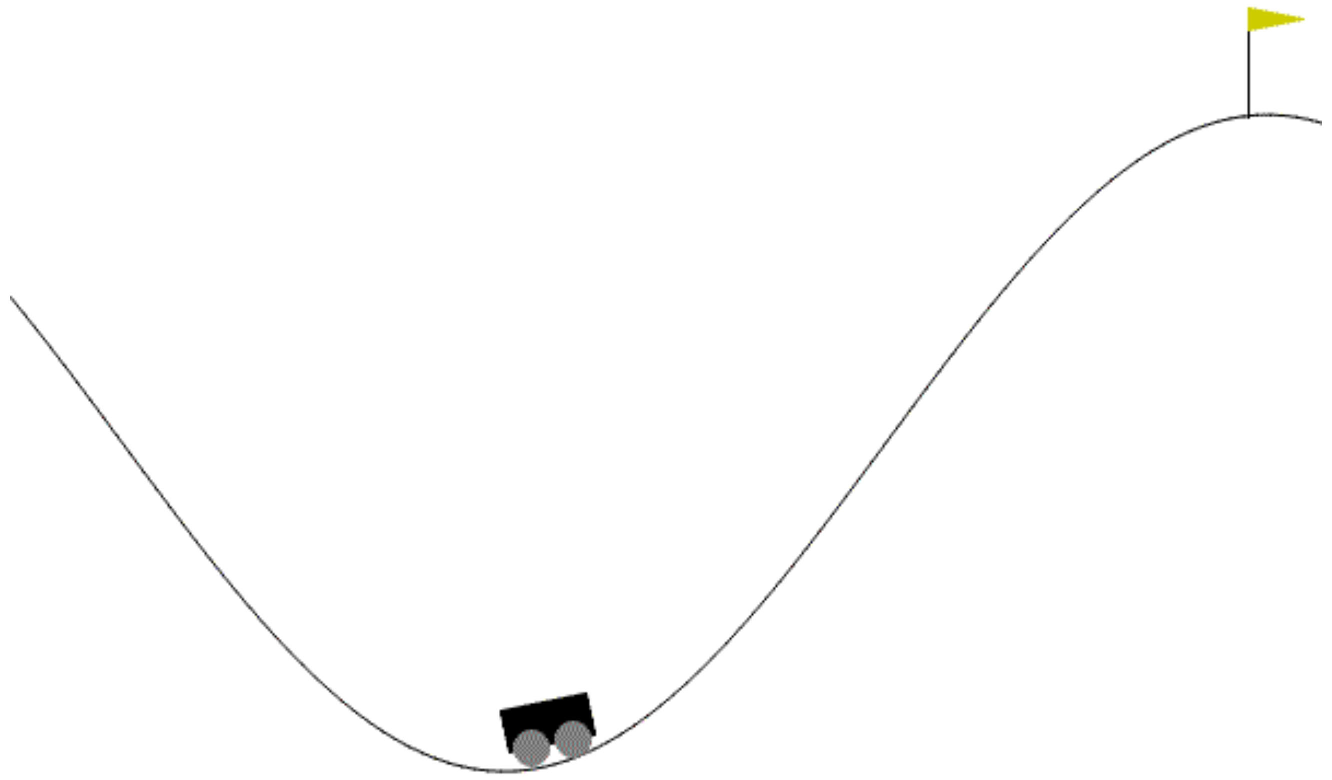
Gym Environments

The Gym interface is a standardized package capable of representing general RL problems



```python
import gym
env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(1000):
   action = policy(observation)  # User-defined policy function
   observation, reward, terminated, truncated, info = env.step(action)

   if terminated or truncated:
      observation, info = env.reset()
env.close()
```



Gym Environments

The Gym interface is a standardized package capable of representing general RL problems

```
import gym
env = gym.make("LunarLander-v2", render_mode="human")
observation, info = env.reset(seed=42)
for _ in range(1000):
    action = policy(observation)  # User-defined policy function
    observation, reward, terminated, param not in our version info = env.step(action)

    if terminated or truncated:
        observation, info = env.reset()
env.close()
```

Initialize gym environment

Reset to start state

Query the policy based on the state

Call env.step()

Reset if it is terminated (finished trajectory)

Gym Environments (step)

- State is maintained within the gym environment.
- Whenever you call step, 4 things are returned:
  - New observation (state)
  - Reward
  - Done
  - Info

This varies based on version and is specific to the one we use in PA2

Vectorized Environments

```
>>> envs = gym.vector.make("CartPole-v1", num_envs=3)
>>> envs.reset()
>>> actions = np.array([1, 0, 1])
>>> observations, rewards, dones, infos = envs.step(actions)

>>> observations
array([[ 0.00122802,  0.16228443,  0.02521779, -0.23700266],
        [ 0.00788269, -0.17490888,  0.03393489,  0.31735462],
        [ 0.04918966,  0.19421194,  0.02938497, -0.29495203]],
        dtype=float32)
>>> rewards
array([1., 1., 1.])
>>> dones
array([False, False, False])
>>> infos
{}
```

Gym Wrappers

```
>>> import gym
>>> from gym.wrappers import RescaleAction
>>> base_env = gym.make("BipedalWalker-v3")
>>> base_env.action_space
Box([-1. -1. -1. -1.], [1. 1. 1. 1.], (4,), float32)
>>> wrapped_env = RescaleAction(base_env, min_action=0, max_action=1)
>>> wrapped_env.action_space
Box([0. 0. 0. 0.], [1. 1. 1. 1.], (4,), float32)
```

Wrappers are very helpful ways to changing behavior of environments without needing to change the underlying code

You can use them to view the output of the environments in PA2

Other PyTorch and Gym Resources

https://pytorch.org/blog/flexattention/

http://blog.ezyang.com/2024/11/ways-to-use-torch-compile/

https://www.gymlibrary.dev/

Advanced-ish Pytorch

(You most likely won’t need these for your projects)

torch.compile()

```python
def foo(x, y):
    a = torch.sin(x)
    b = torch.cos(y)
    return a + b
opt_foo1 = torch.compile(foo)
print(opt_foo1(torch.randn(10, 10), torch.randn(10, 10)))
```

torch.compile() makes PyTorch code run faster by JIT-compiling PyTorch code into optimized kernels, while requiring minimal code changes.

Similar to @jax.jit for those of you familiar with JAX

Like Jax, these functions are harder to debug (since they get mapped to CUDA kernels), so only compile if you're certain that it will work!

torch.vmap()

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

torch.dot                                  # [D], [D] -> []
batched_dot = torch.func.vmap(torch.dot)  # [N, D], [N, D] -> [N]
x, y = torch.randn(2, 5), torch.randn(2, 5)
batched_dot(x, y)
```

vmap, which stands for vectorized map, vectorizes the operations more effectively than the corresponding python native version (similar to jax.vmap())