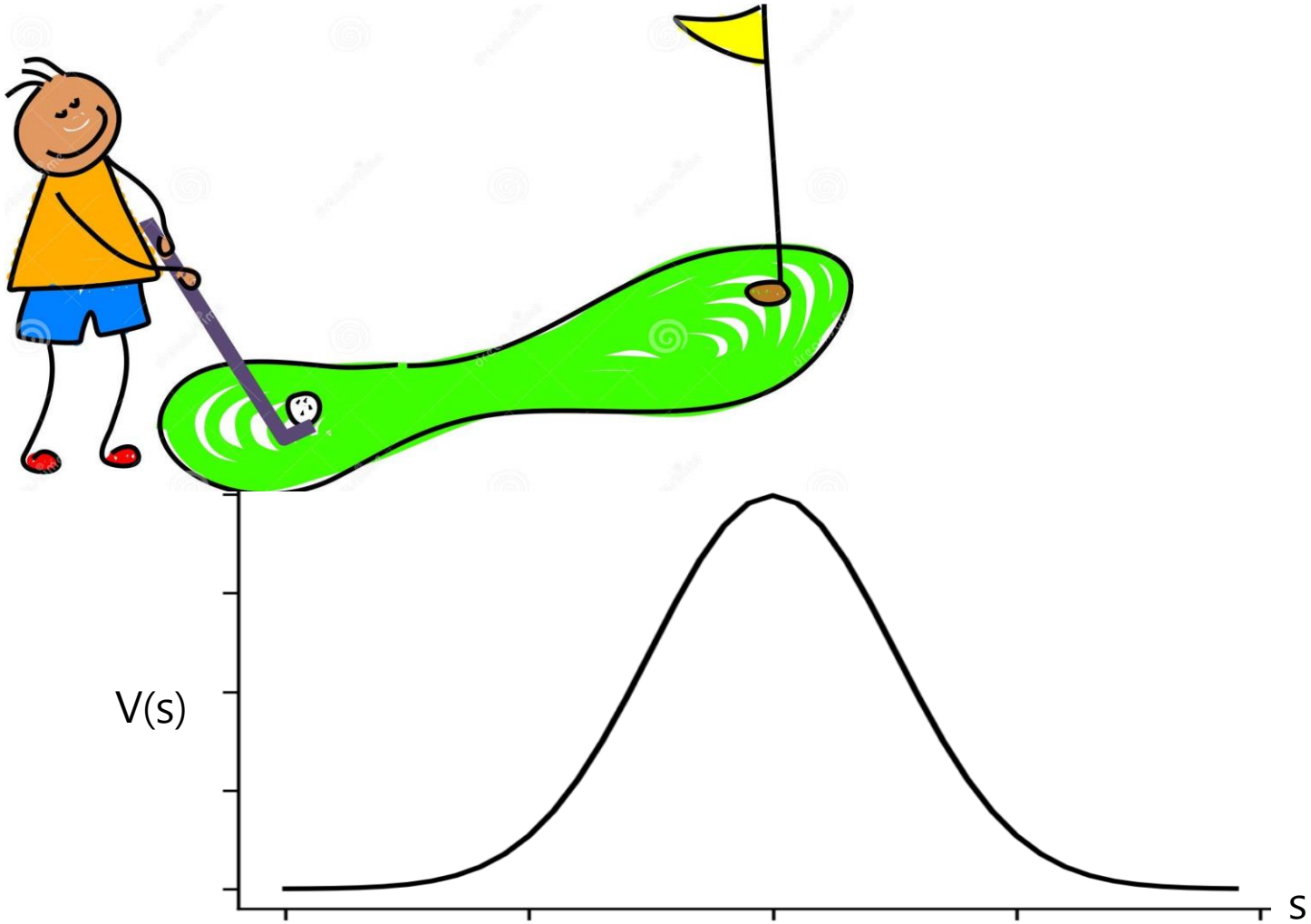
6.강 Deep Learning

Contents

- Continuous State Space
- Neural Network
- Pytorch

Continuous state spaces



Continuous state spaces

Tabular methods

Algorithm 1 SARSA

Input: α learning rate, ϵ random action probability, γ discount factor
$\pi \leftarrow \epsilon$-greedy policy w.r.t $Q(s,a)$
Initialize $Q(s,a)$ arbitrarily, with $Q(terminal,\cdot) = 0$
for episode $\in 1..N$ **do**
    Reset the environment and observe S_0
    $A_0 \sim \pi(S_0)$
    for $t \in 0..T-1$ **do**
        Execute A_t in the environment and observe S_{t+1}, R_{t+1}
        $A_{t+1} \sim \pi(S_{t+1})$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$
    end for
end for
Output: Near optimal policy π and action values $Q(s,a)$

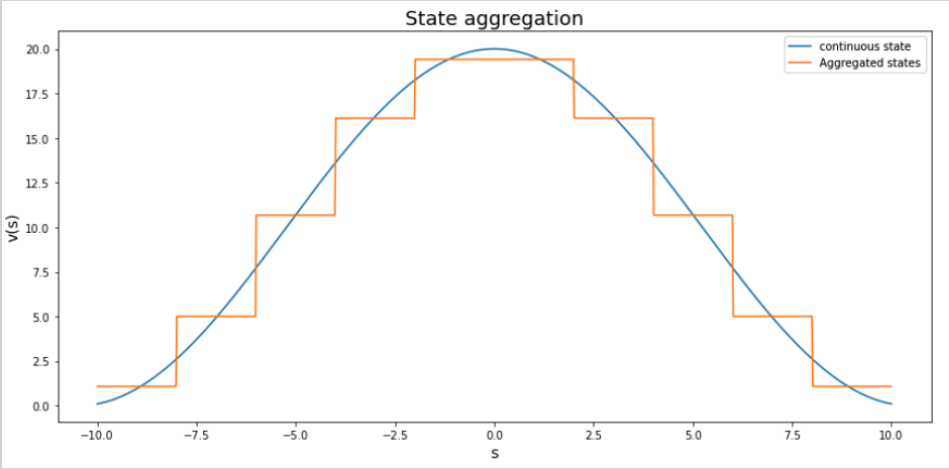
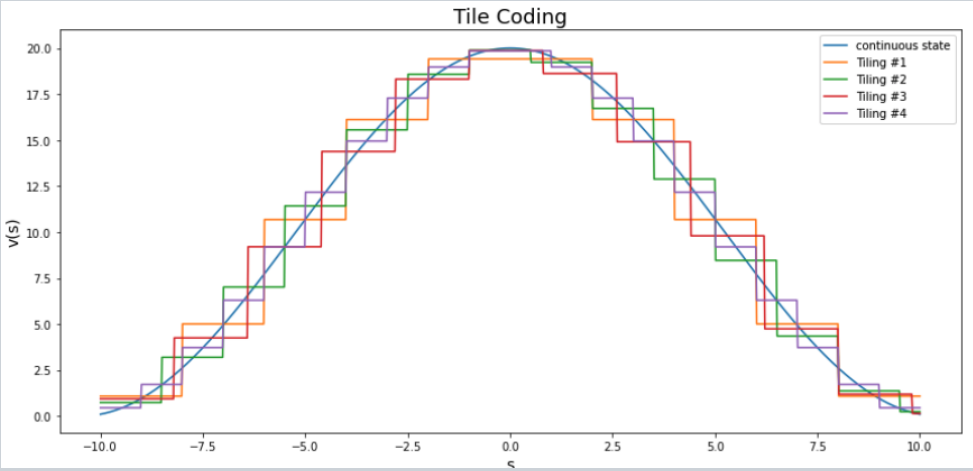
Infinite states: [-10,10]

An entry Q(s, a) per state

We would need infinite memory

Continuous state spaces

- The solution is, either transform the states to convert the continuous state space into a discrete state space, or use other algorithms

State aggregation	Tile coding
	

Continuous state spaces

```
class StateAggregationEnv(gym.ObservationWrapper):

    def __init__(self, env, bins, low, high):
        # low = [-1.2, -0.07], high = [0.6, 0.07], bias = [20, 20]
        super().__init__(env)
        self.buckets = [np.linspace(j,k, l-1) for j,k,l in zip(low, high, bins)]
        # [w0, w0] --> 400
        self.observation_space = gym.spaces.MultiDiscrete(nvec=bins.tolist())

    def observation(self, obs):
        # [-1.2, 0.] -> [4, 3]
        indices = tuple(np.digitize(i, b) for i,b in zip(obs, self.buckets))
        return indices
```

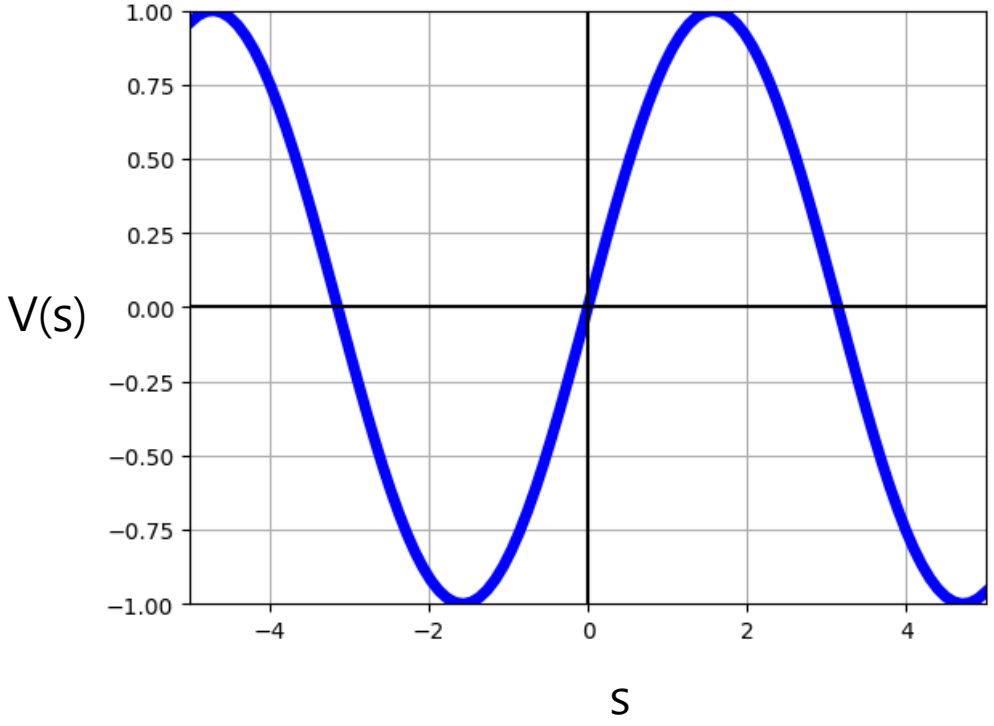
```
bins = np.array([20, 20])
low = env.observation_space.low
high = env.observation_space.high
saenv = StateAggregationEnv(env, bins=bins, low=low, high=high)
```

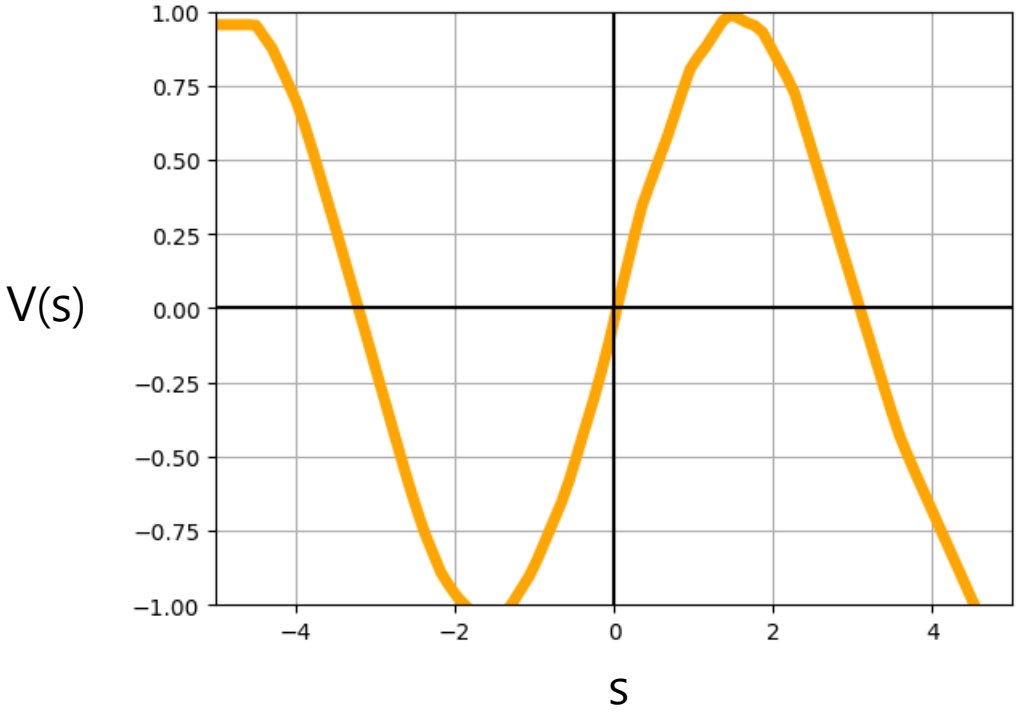
Disadvantages of transforming

- The problem is, transforming the states to convert the continuous state space into a discrete state space has limited precision and complexity of O(n^k)

Neural Network

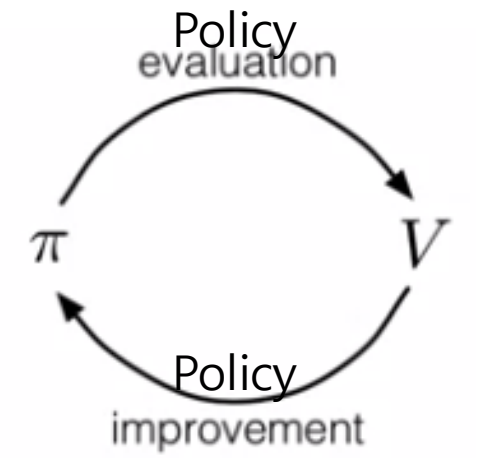
Function approximators

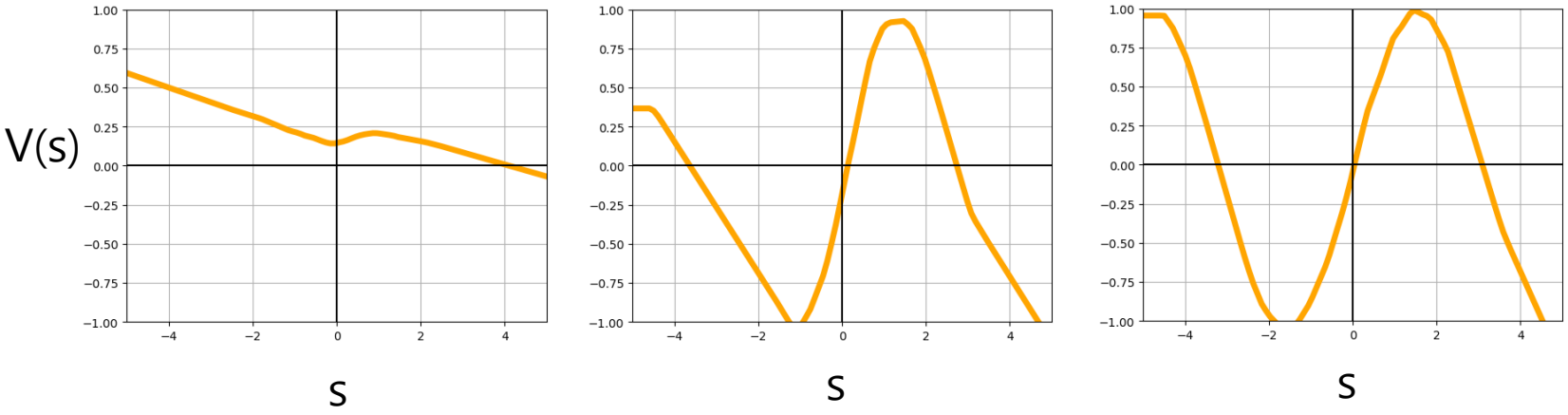




Function approximators

- How do we observe the value function?
  - The agent learns based on experience.
  - The functions v*(s) and q*(s ,a) are not know in advance

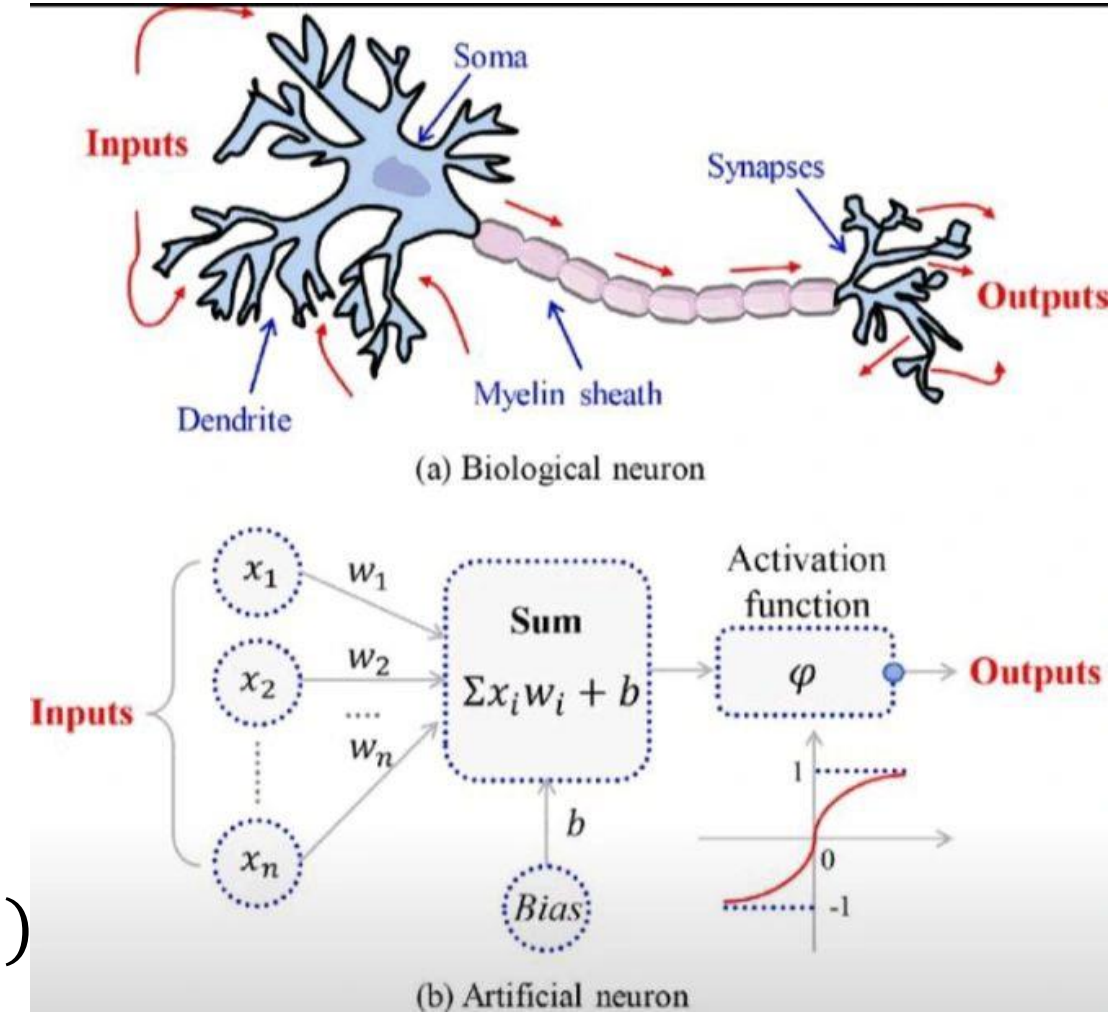




$$f_1(s|w) \rightarrow f_2(s|w) \rightarrow \cdots . \rightarrow f_n(s|w) \approx v^*(s)$$

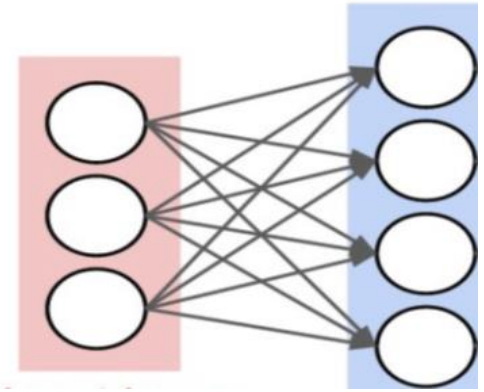
Neural Networks

- Computing system inspired by the biological neural networks that constitute our brain
- They server multiple purposes, including function approximation: $\hat{y} = f(x|w)$
- Mathematical function typically consisting of a weighted sum of inputs and a activation/transfer function $\text{Output} = \varphi(\sum_{i=1}^{n} w_i x_i + b)$



Neural networks

- Input vector x = [x1, x2, x3]



- Connection matrix:

$$W1 = \begin{bmatrix} w11 & w12 & w13 & w14 \\ w21 & w22 & w23 & w24 \\ w31 & w32 & w33 & w34 \end{bmatrix}$$

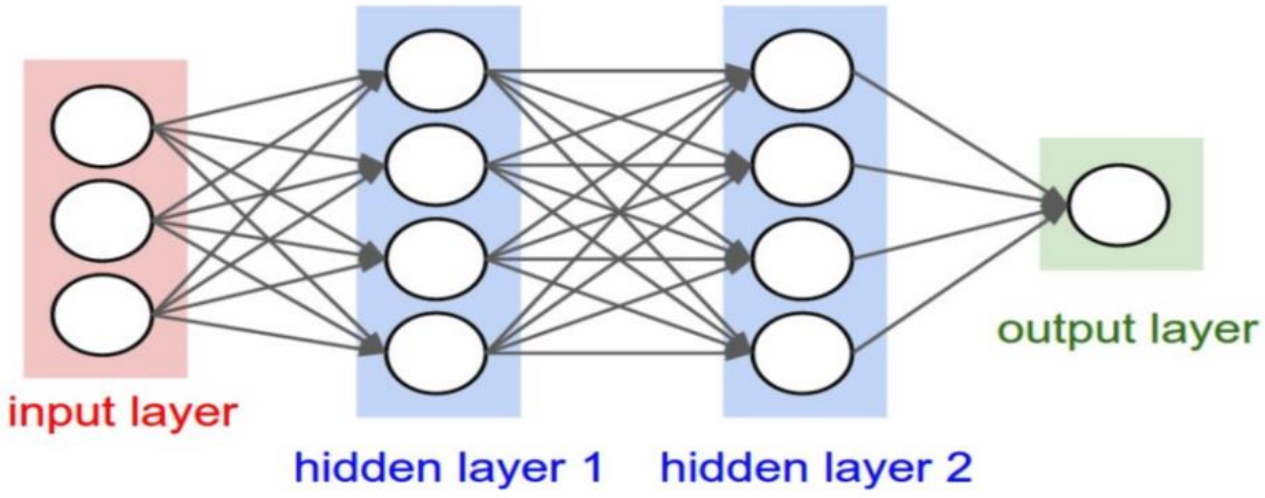
By changing its parameters W1, we can modify it to approximate the function we are interested in

- Output vector:

$$H = [\varphi(\sum_{i=1}^{n} w_i x_i + b), \ldots, \varphi(\sum_{i=1}^{n} w_i x_i + b)]$$

Neural Networks

- Networks that do not have cycles are known as feedforward NN. Signals always propagate forward
- The neuron receives inputs, process & aggregate those inputs, and either inhibits or amplifies before passing the signal to the next layer

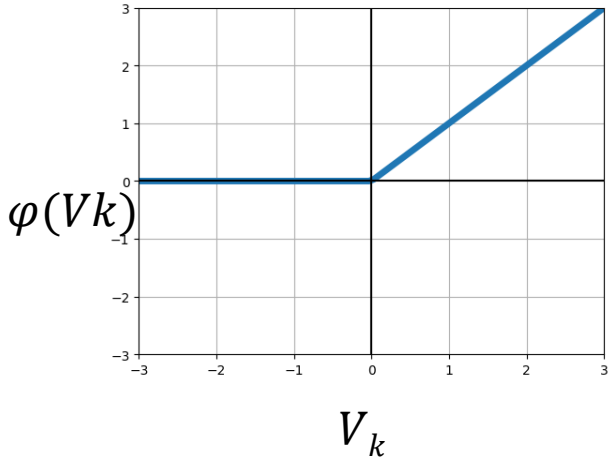


$$h_1 = \sigma(W_1 x) \quad h_2 = \sigma(W_2 h_1) \quad y = W_3 h_2$$

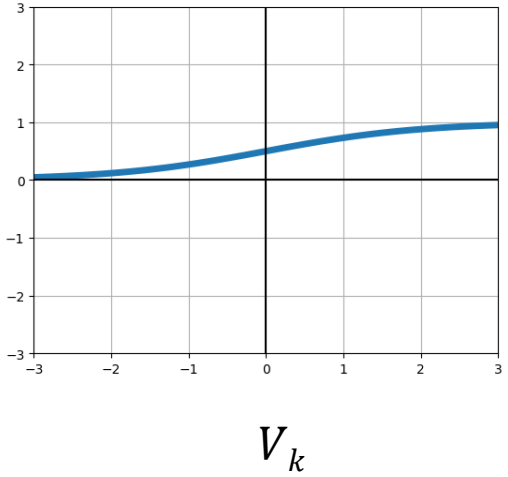
Neural networks

- Activation functions

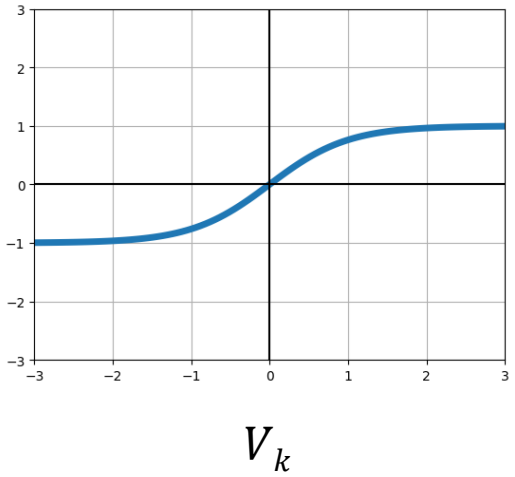
Relu()



Sigmoid()

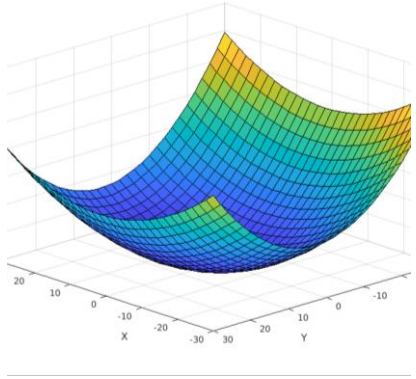


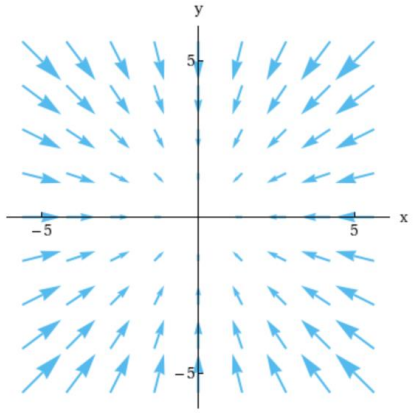
Tanh()

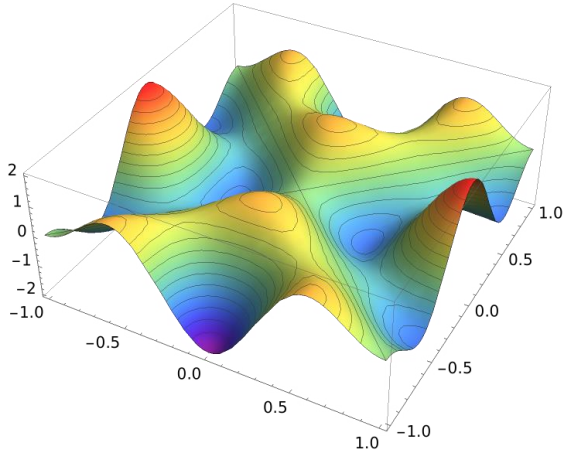


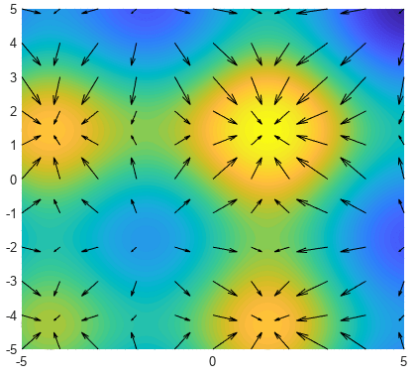
Gradient Descent

- Given some loss function: $L(\vec{x}, \vec{y}) = \|2\vec{x} + 2\vec{y}\|$
- Update rules for the parameters: $w_{t+1} = w_t - \alpha \nabla \hat{L}(w)$
- Gradient vector: $\nabla \hat{L}(w) = [\frac{\partial L}{\partial w1}, \frac{\partial L}{\partial w2}, \ldots, \frac{\partial L}{\partial wn}]$
- Computed using the backpropagation algorithm
- $\nabla \hat{L}(w)$ points to the direction of maximum growth of $\nabla \hat{L}(w)$
- α is the size of the step we take in the opposite direction to $\nabla \hat{L}(w)$









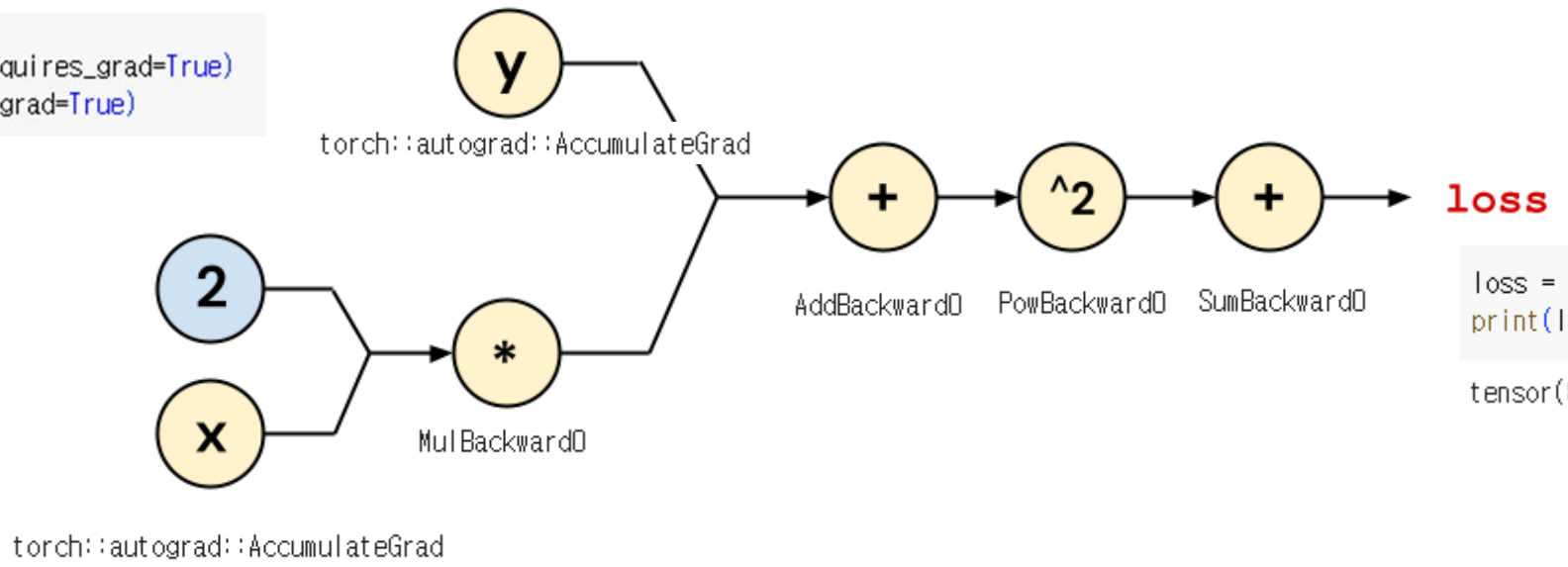
Backpropagation

- Computed using the backpropagation algorithm
- We want to evaluate partial derivative: $\frac{\partial L}{\partial \vec{x}}$ and $\frac{\partial L}{\partial \vec{y}}$

```
shape = (3, )
x = torch.tensor([1., 2, 3], requires_grad=True)
y = torch.ones(shape, requires_grad=True)
```

```
loss = ((2 * x + y)**2).sum()
loss.backward()
print(x.grad)
print(y.grad)
```

```
tensor([24., 40., 56.])
tensor([12., 20., 28.])
```




```
loss = ((2 * x + y)**2).sum()
print(loss)
```

```
tensor(83., grad_fn=<SumBackward0>)
```

Cost function

- Mean squared error:

$$L(w) = \frac{1}{N}\sum_{i=0}^{N}[y - \hat{y}]^2$$

- For our neural network to estimate q(s, a) as well as possible, we will minimize the observed squared errors

$$\hat{L}(w) = \frac{1}{N}\sum_{i=0}^{N}[R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}|w) - \hat{q}(S_t, A_t|w)]^2$$

- Target value:

$$R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}|w)$$

Estimated value:

$$\hat{y} = \hat{q}(S_t, A_t|w)$$

Neural network optimization

- For our neural network to estimate q(s, a) as well as possible, we will minimize the observed squared errors

$$\hat{\mathrm{L}}(\theta) = \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=0}^{\mathrm{N}} [\mathrm{R}_i + \gamma \hat{\mathrm{q}}(\mathrm{S}_i', \mathrm{A}_i' | \theta_{targ}) - \hat{\mathrm{q}}(\mathrm{S}_i, \mathrm{A}_i | \theta)]^2$$

- Target value: a value towards which we want to push the estimates

$$\mathrm{R}_i + \gamma \hat{\mathrm{q}}(\mathrm{S}_i', \mathrm{A}_i' | \theta_{targ})$$

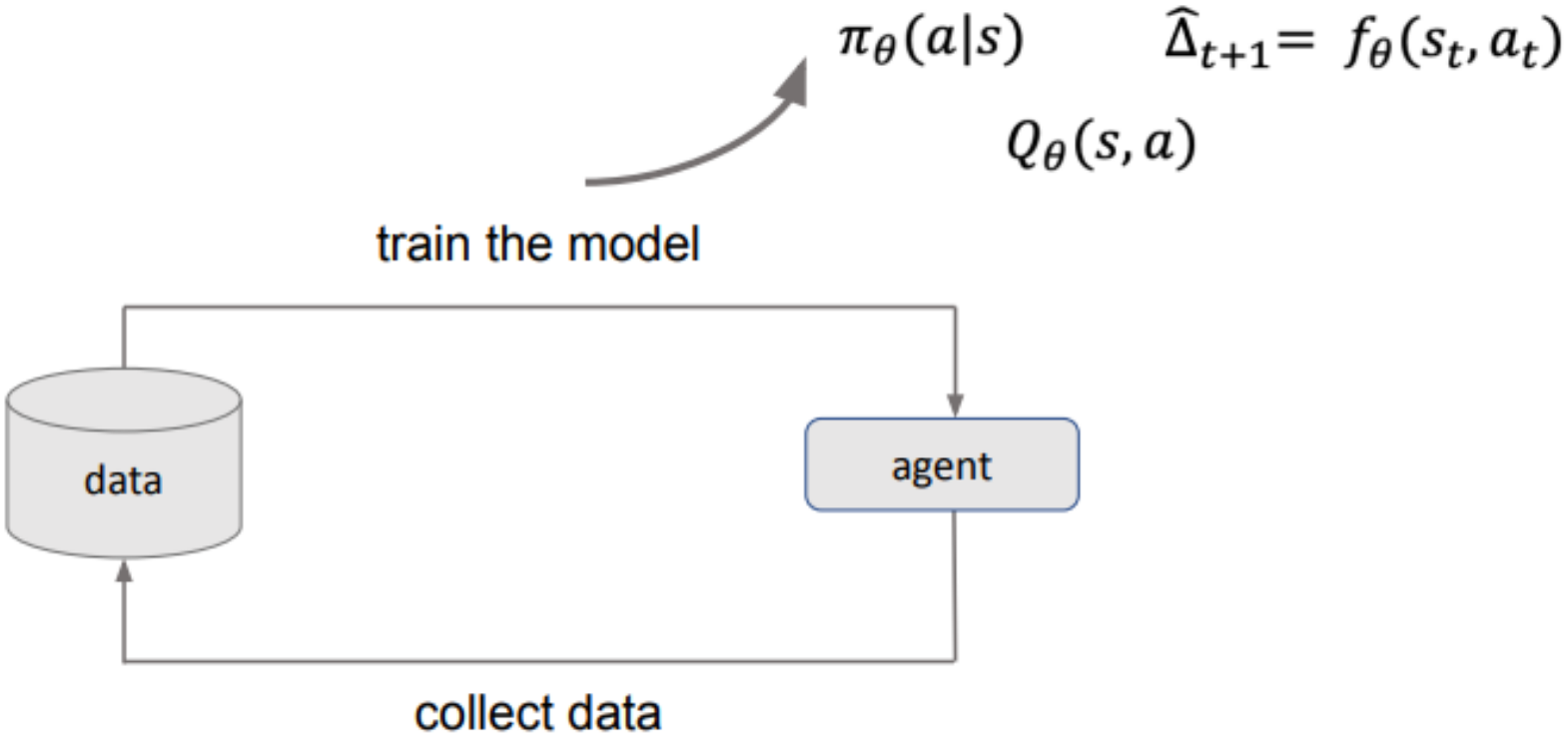
- Estimate of the q-value of a state-action pair

$$\hat{\mathrm{q}}(\mathrm{S}_i, \mathrm{A}_i | \theta)$$

Pytorch

Pictures from Stanford's CS231n
Pictures from Berkeley CS285

Train an agent to perform useful tasks



How do train a model

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

gradient descent

dataset

loss

neural network

PyTorch does all of these!

What is Pytorch

Python library for:

- Defining neural networks
- Automating computing gradients
- And more! (datasets, optimizers, GPUs, etc.)

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

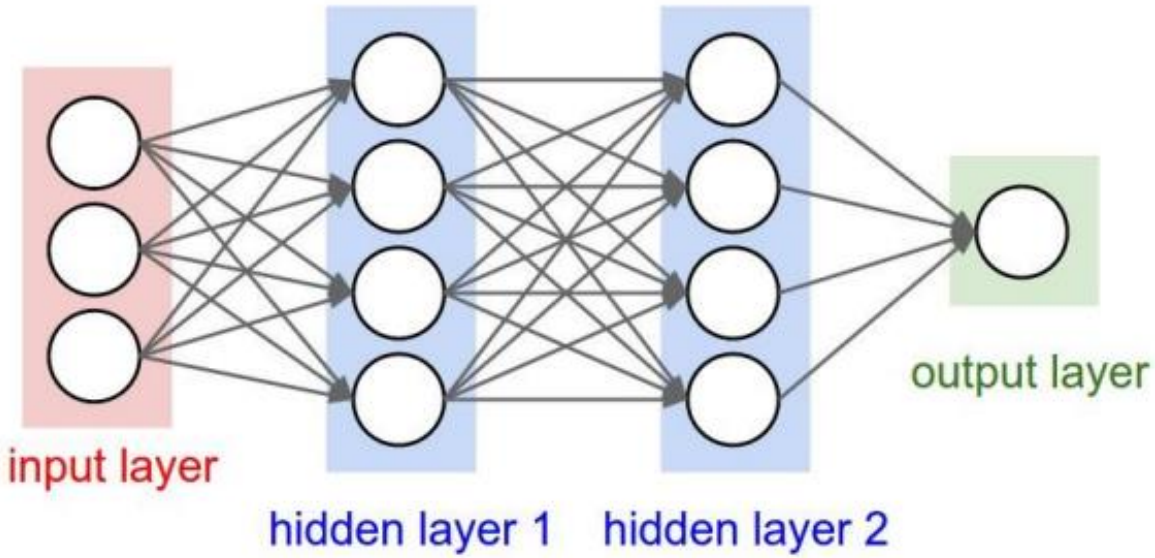
gradient descent

dataset

loss

neural network

How does Pytorch works?



You define:	$h_1 = \sigma(W_1 x)$	$h_2 = \sigma(W_2 h_1)$	$y = \sigma(W_3 h_2)$
PyTorch computes:	$\frac{\partial y}{\partial W_1} = \frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W_1}$	$\frac{\partial y}{\partial W_2} = \frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial W_1}$	$\frac{\partial y}{\partial W_3}$

Numpy & PyTorch

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

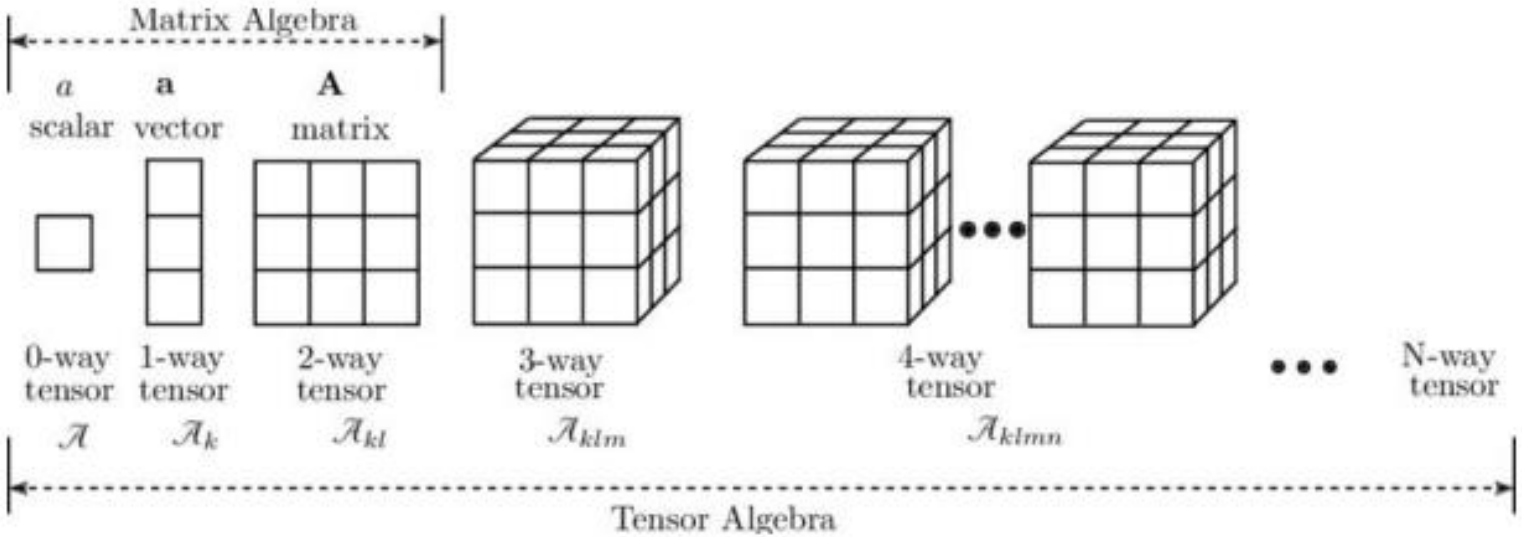
```python
import numpy as np

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

# Element-wise operations
arr_sum = a + b
arr_prod = a * b
arr_doubled = 2 * a

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

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

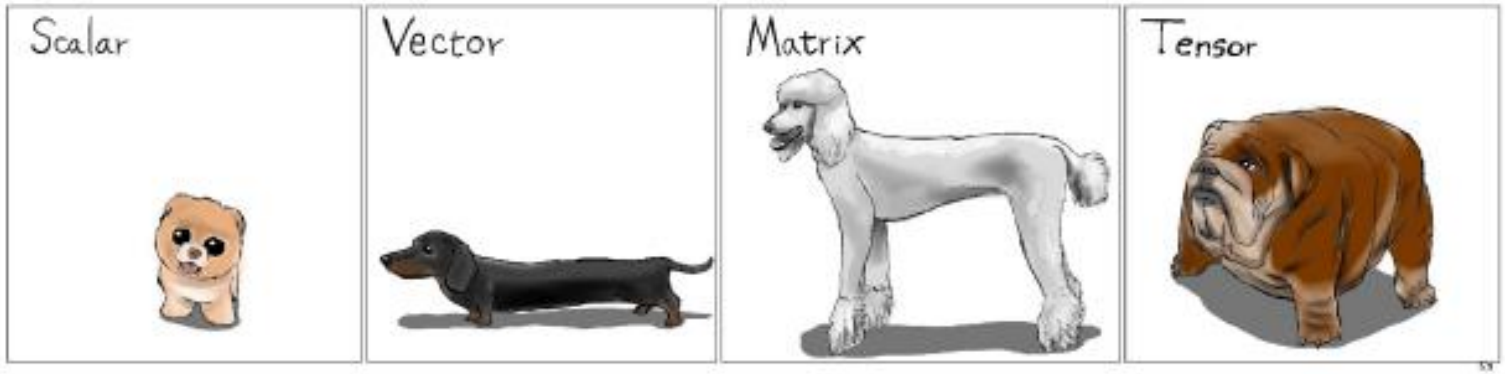
PyTorch

```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 Array





Multidimensional Indexing

Axis 1

A

Axis 0

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

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

Multidimensional Indexing

Axis 1

Axis 0

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

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

Axis 1

Axis 0

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

```
A[:, 3]
```

Multidimensional Indexing

Axis 1

Axis 0

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

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

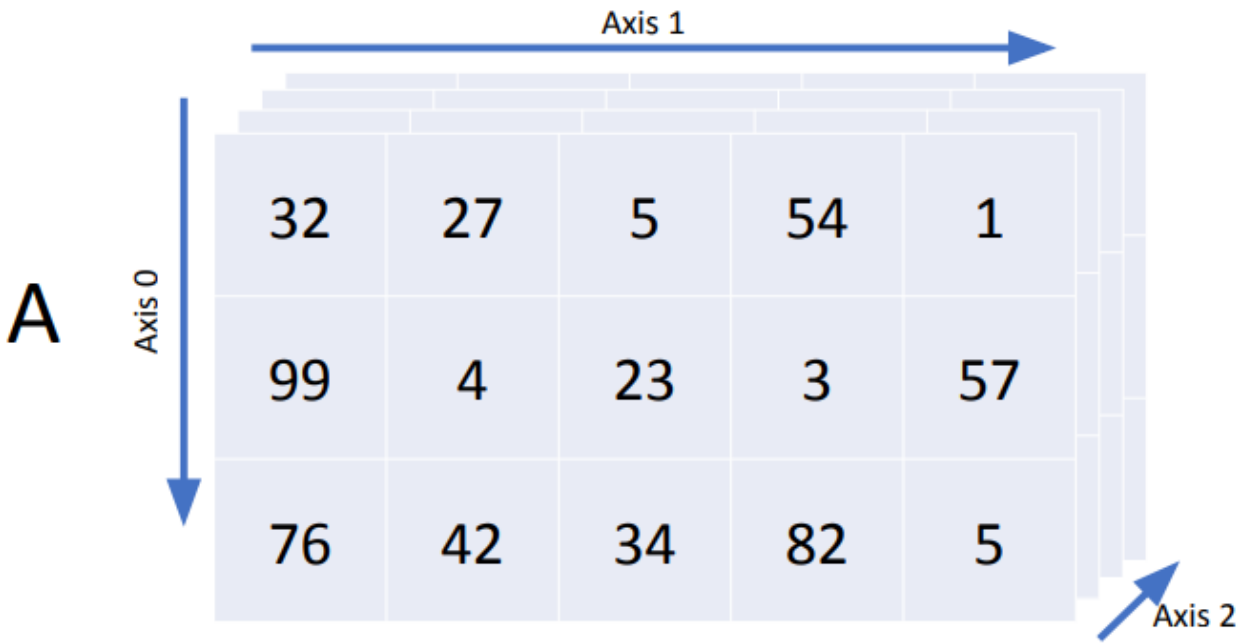
Axis 1

Axis 0

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

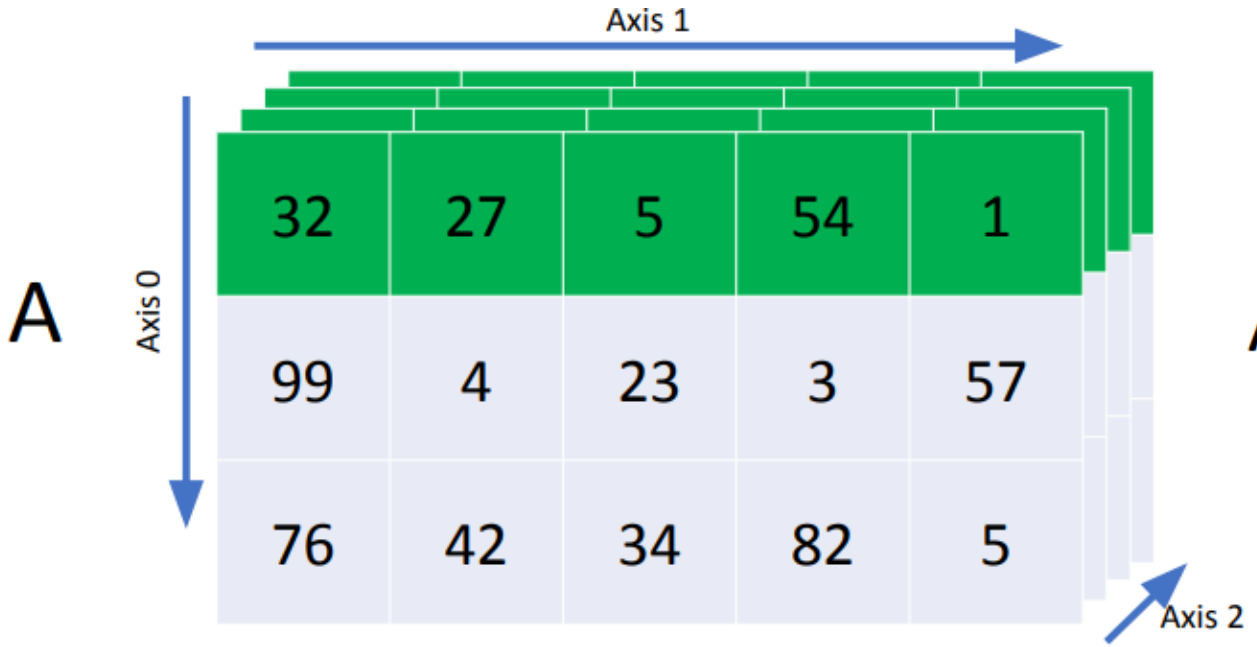
```
A[0, 2:4]
```

Multidimensional Indexing

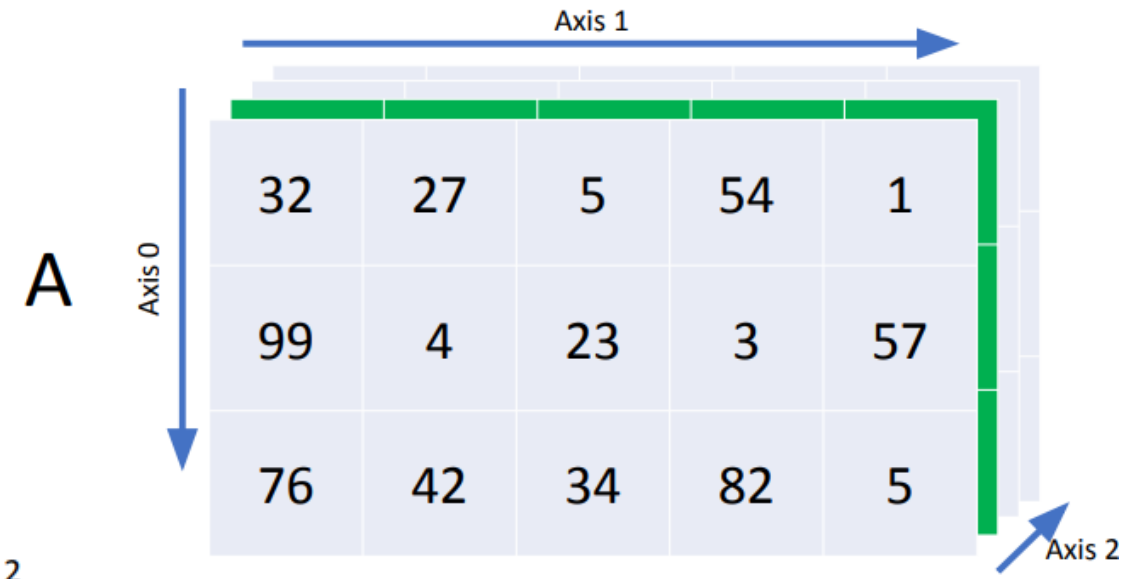



```
A.shape == (3, 5, 4)
```

Multidimensional Indexing



A[0, ...]



A[..., 1]

Shape Operations

NumPy

```
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

```
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!

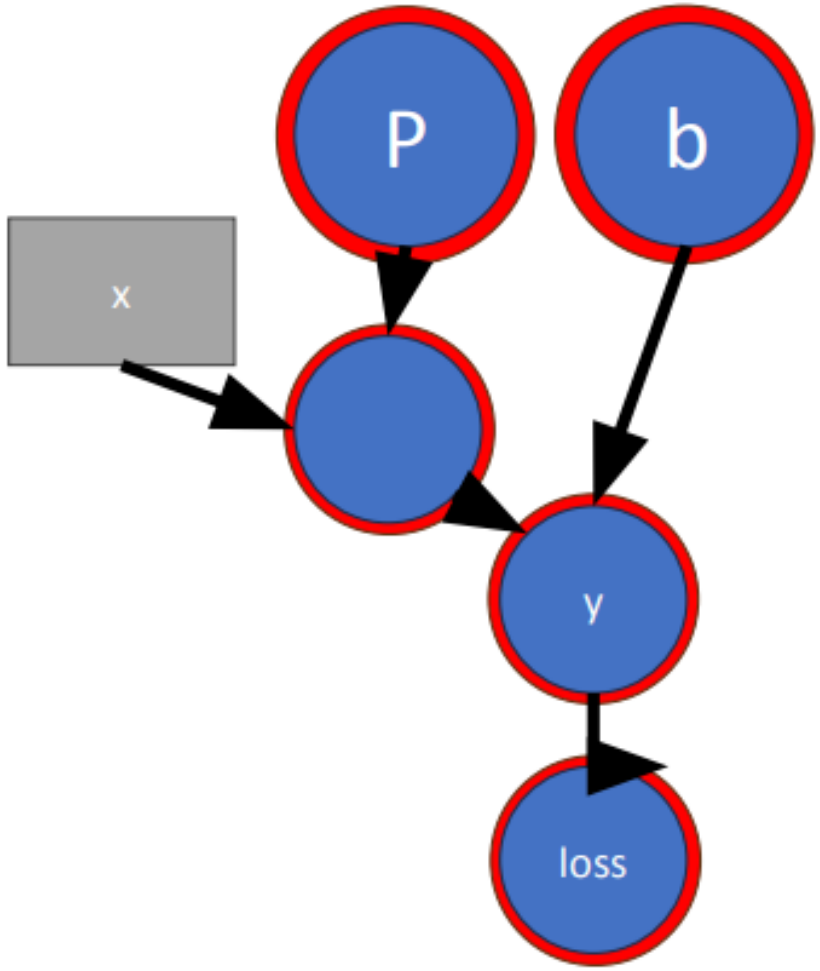
Device Management

```
[ ] device = torch.device("cuda")
    x = torch.zeros((2, 3))
    y = torch.ones((2, 3), device=device)
    z = x + y
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-71-565d7b7035e6> in <module>
      2 x = torch.zeros((2, 3))
      3 y = torch.ones((2, 3), device=device)
----> 4 z = x + y

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

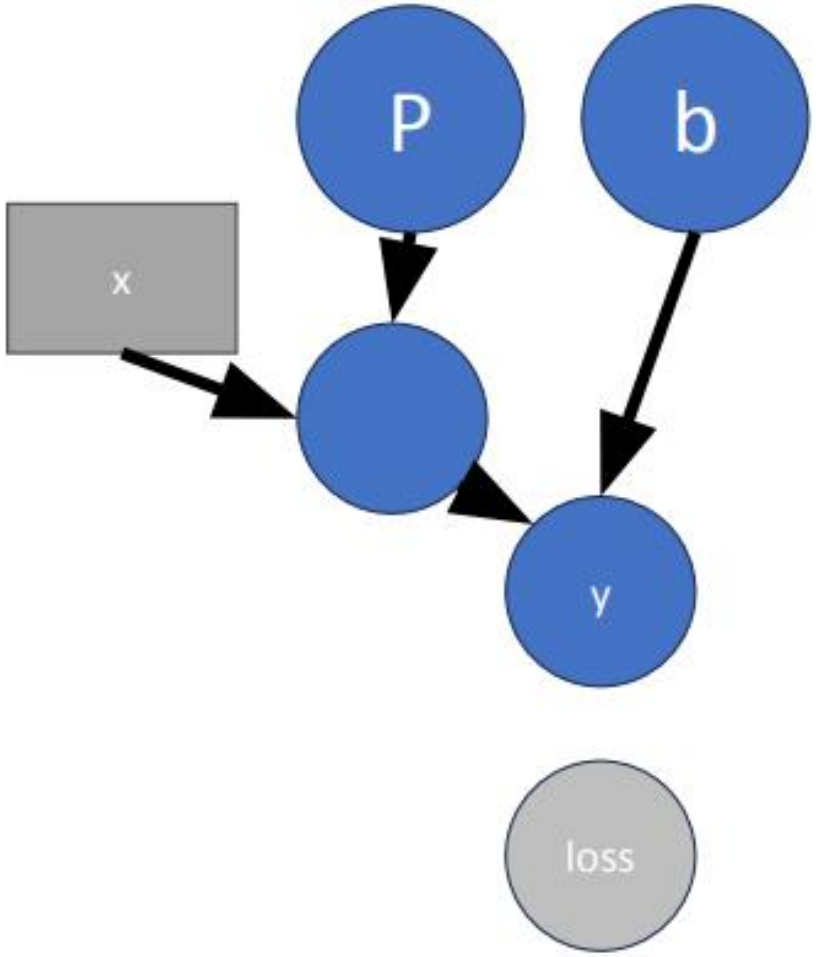
Computing Gradients

```
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

```
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

```
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..
```

REMEMBER THIS!

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())
```

- nn.Module represents the building blocks of a computation graph.
  - For example, in typical pytorch code, each convolution block is its own module, each fully connected block is a module, and the whole network itself is also a module.
- Modules can contain modules within them. All the classes inside of `torch.nn` are instances `nn.Modules`.

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

```
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

- 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()`

Let’s play with the code

- http://bit.ly/cs285-pytorch-2023