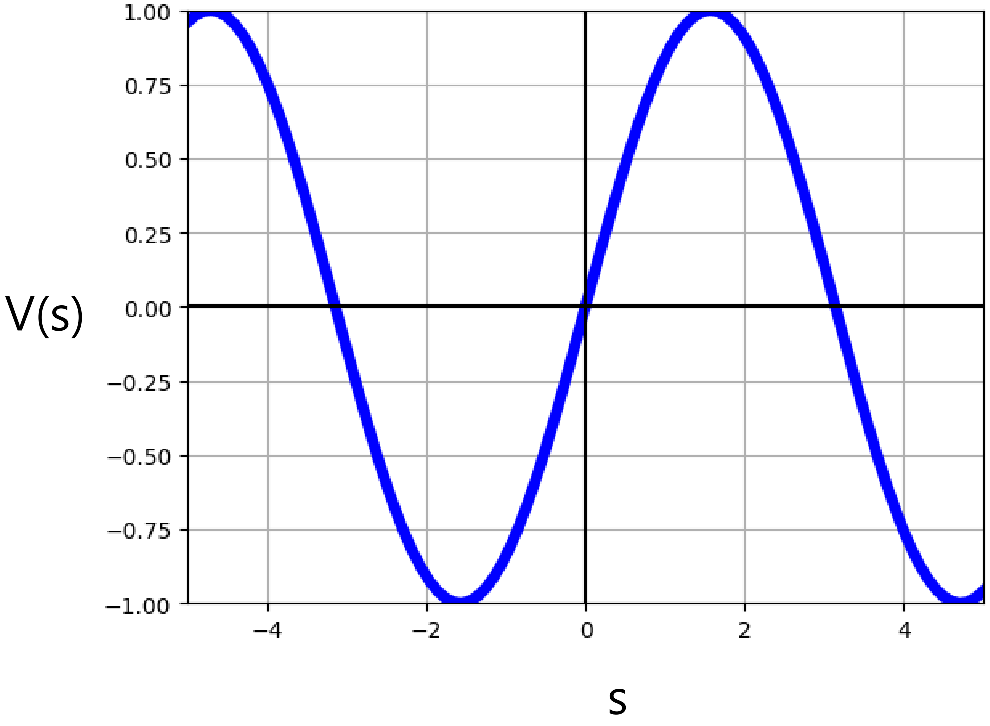
7.강 Deep Q-learning

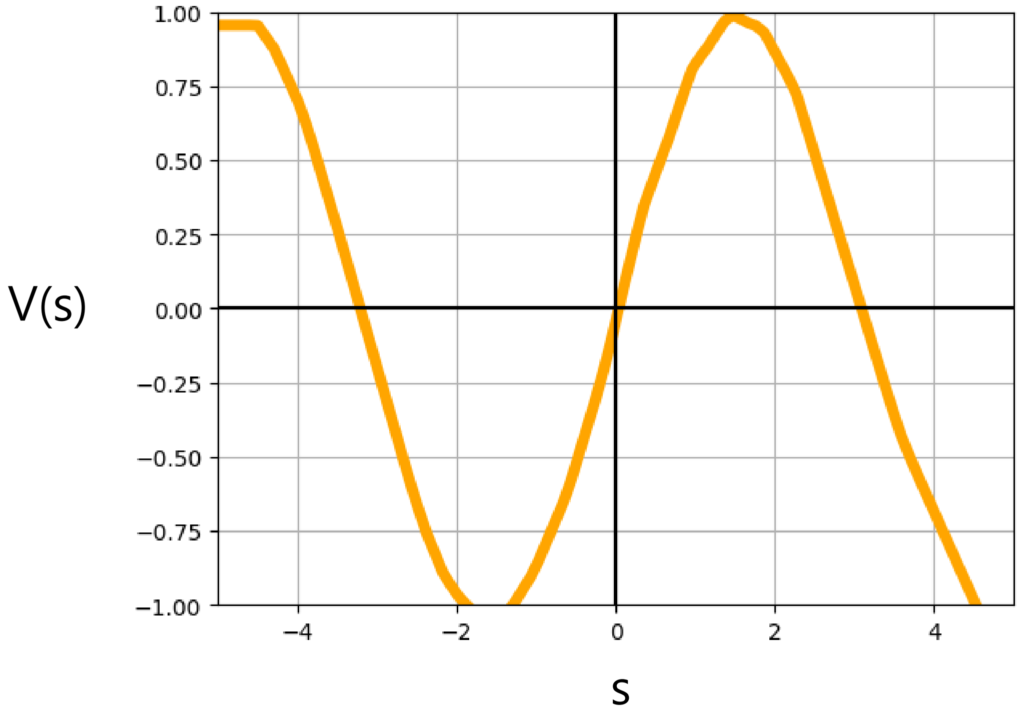
Contents

- Neural Network
- Pytorch
- Deep SARSA
- Code Exercise

Neural Network

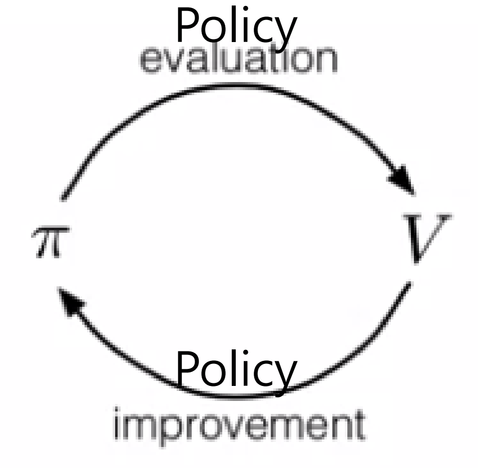
Function approximators

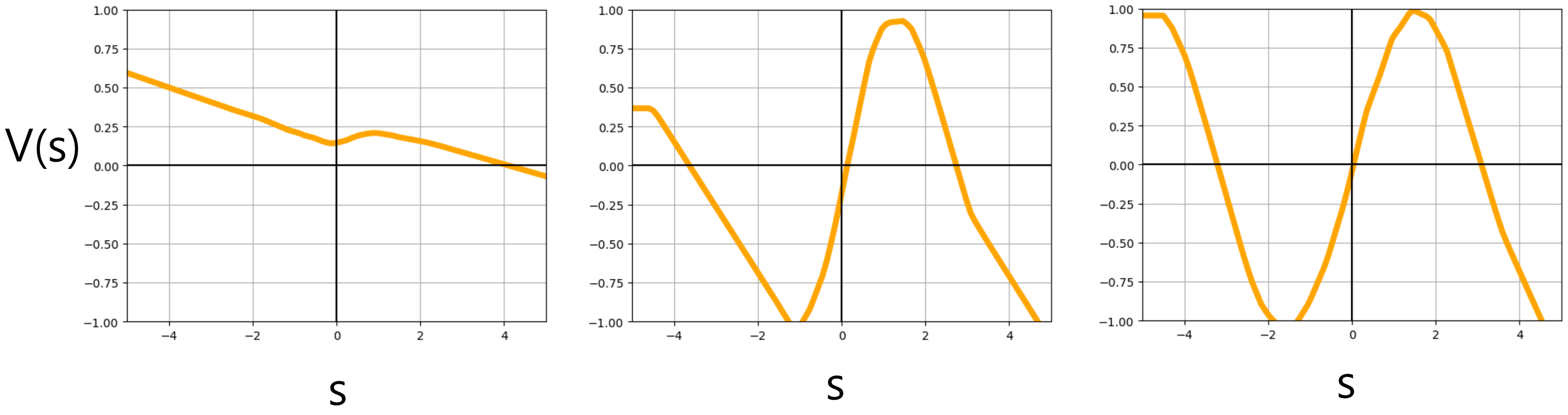




Function approximators

- 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

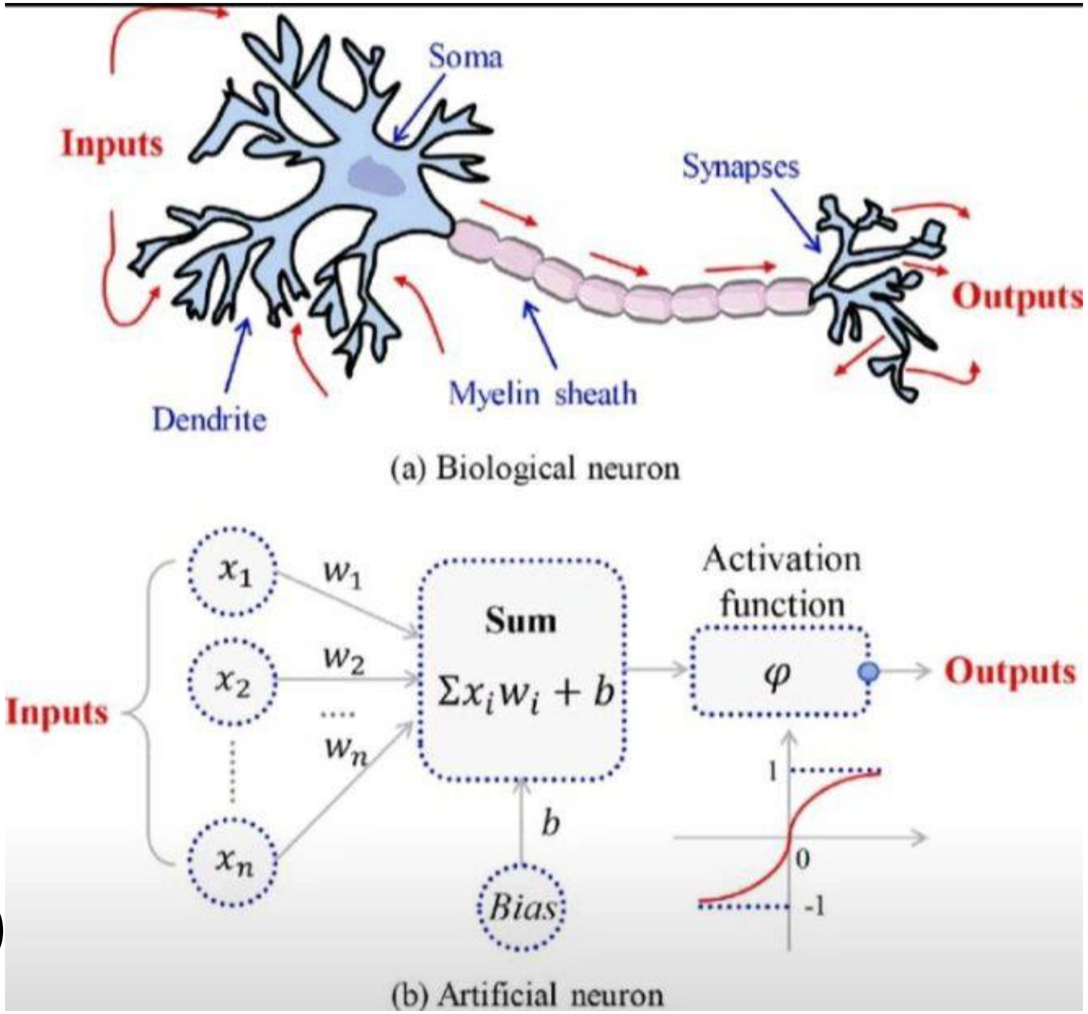




$$f_1(s|w) \rightarrow f_2(s|w) \rightarrow \cdots . \rightarrow f_n(s|w) \approx v^*(s)$$

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 $\text{Output} = \varphi(\sum_{i=1}^{n} w_i x_i + b)$

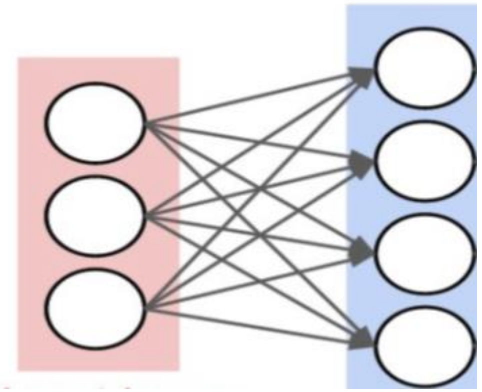


Neural networks

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

- Connection matrix:

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



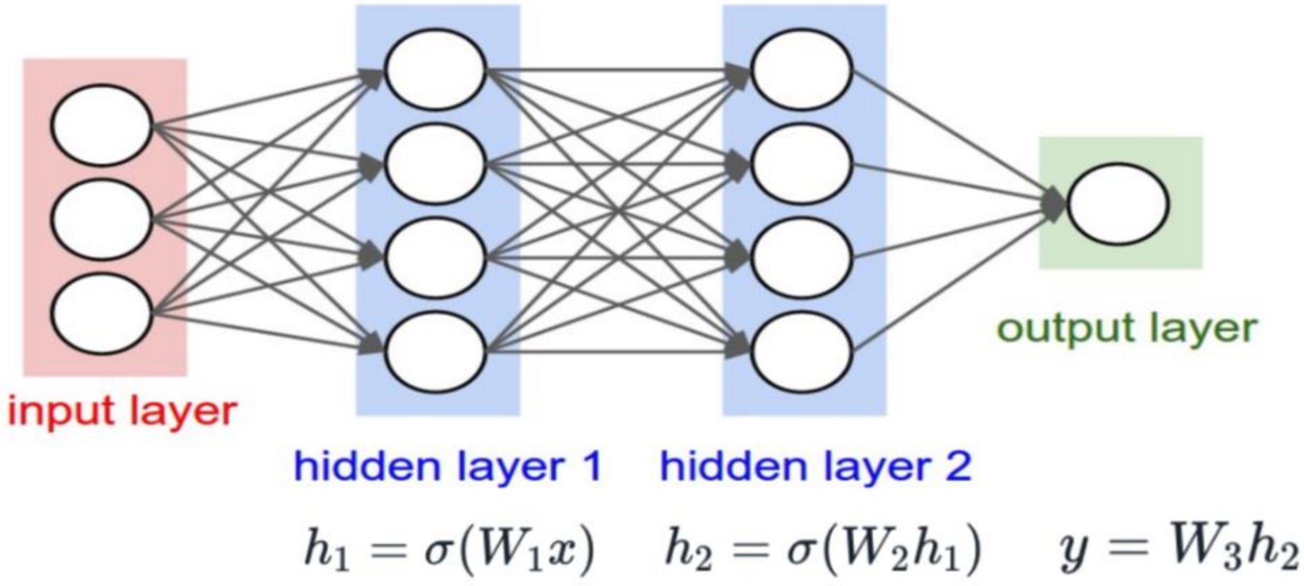
By changing its parameters W1, we can modify it to approximate the function we are interested in

- Output vector:

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

Neural Networks

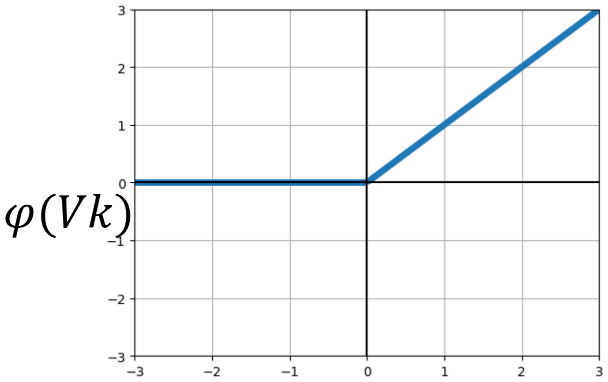
- Networks that do not have cycles are known as feedforward NN. Signals always propagate forward
- The neuron receives inputs, process & aggregate those inputs, and either inhibits or amplifies before passing the signal to the next layer



Neural networks

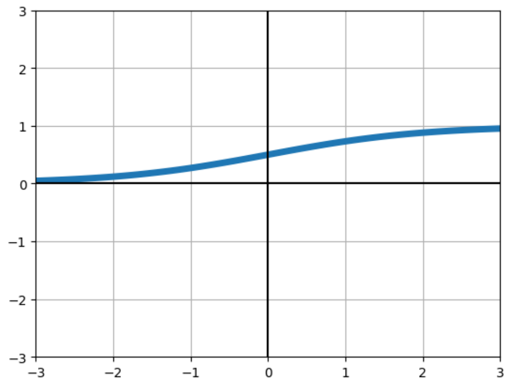
- Activation functions

Relu()



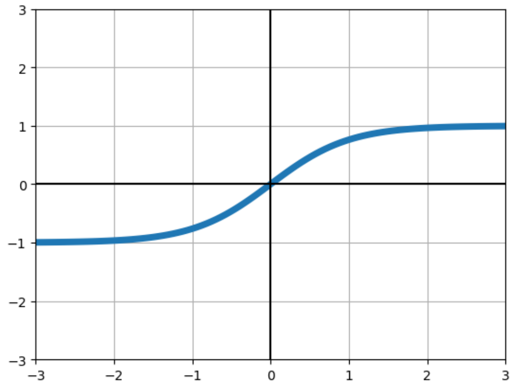
V_k

Sigmoid()



V_k

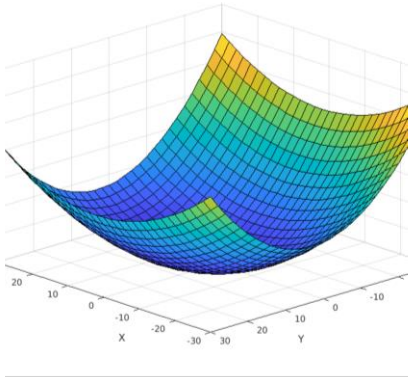
Tanh()

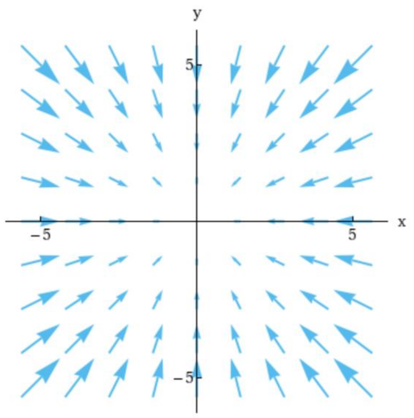


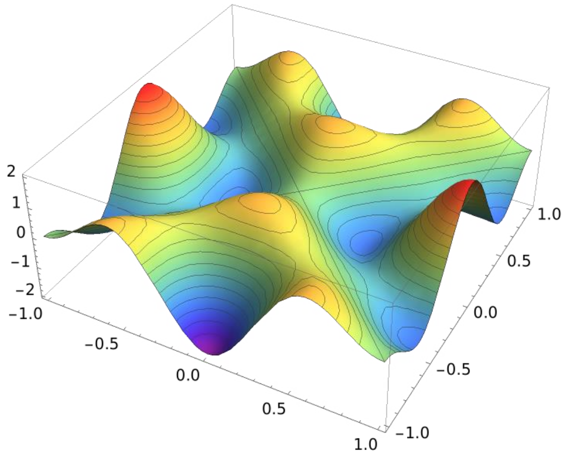
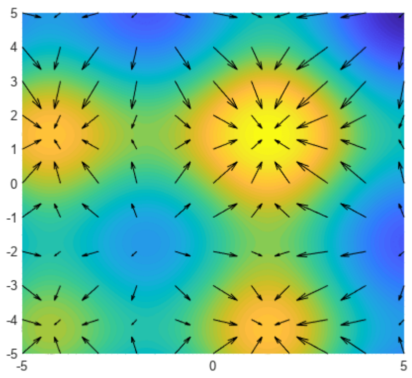
V_k

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 $\nabla \hat{L}(w)$





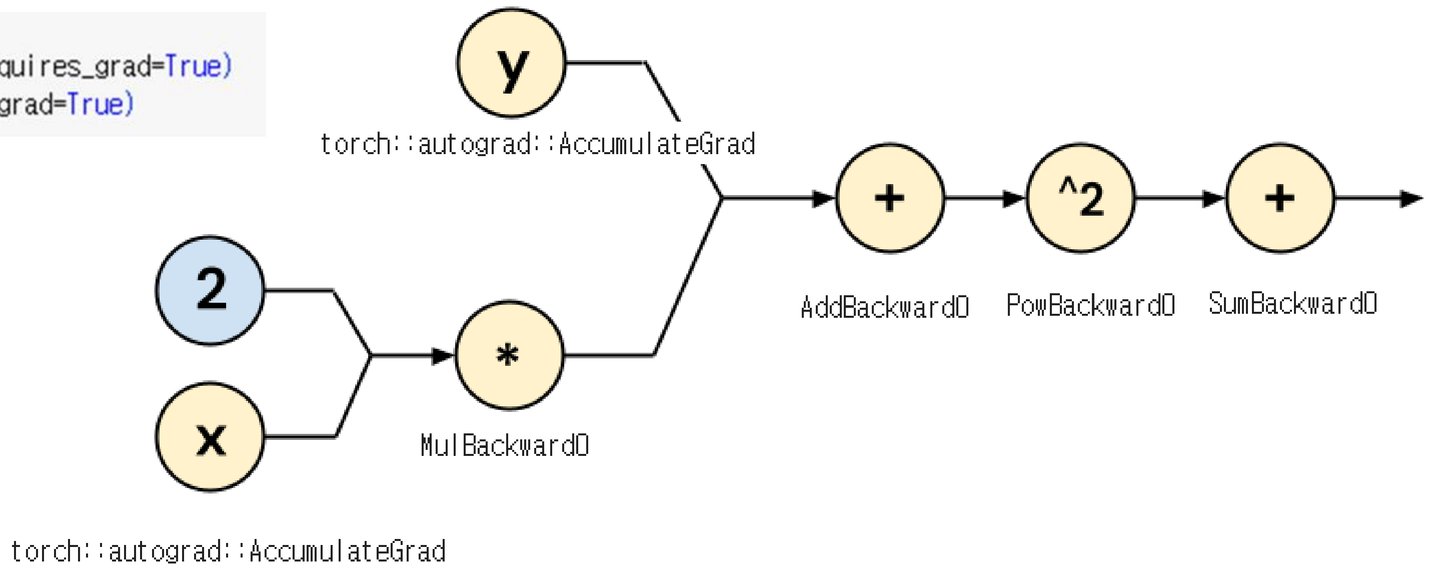
Backpropagation

- 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

Numpy & PyTorch

NumPy

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

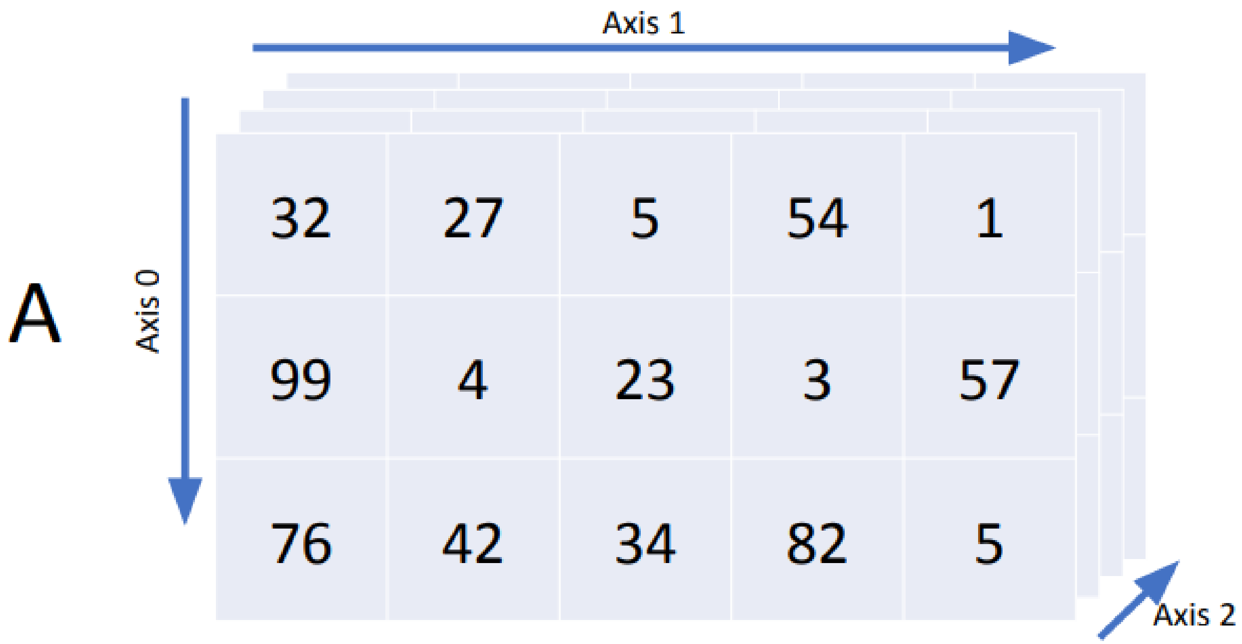
PyTorch

- Fast CPU implementations
- **Allows GPU**
- **Supports autodiff**
- **Imperative**

Other features include:

- Datasets and dataloading
- Common neural network operations
- Built-in optimizers (Adam, SGD, …)

Multidimensional Indexing



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

Shape Operations

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)
```

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()`

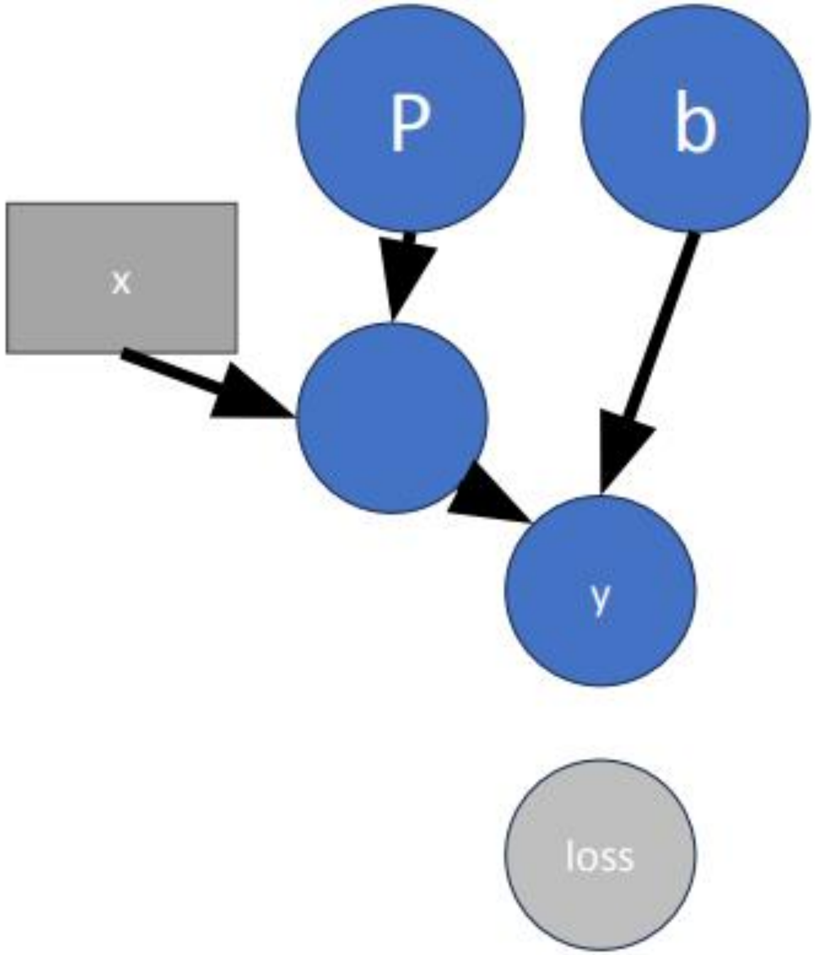
YOU CANNOT PERFORM OPERATIONS BETWEEN TENSORS ON DIFFERENT DEVICES!

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

REMEMBER THIS!

```
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())
```

- 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()`

Neural network optimization

- Mean squared error:

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

- We want to minimize the square of the errors of the neural network estimates

Neural network optimization

- We calculate the gradient vector of the cost function with respect to the θ parameters:

$$\nabla \mathrm{L}(\theta) = \left[\frac{\partial L}{\partial \theta 1}, \frac{\partial L}{\partial \theta 2}, \ldots, \frac{\partial L}{\partial \theta \mathrm{n}}\right]$$

- With the gradient vector, we will make a SGD step:

$$\theta \leftarrow \theta - \alpha \nabla \hat{\mathrm{L}}(\theta)$$

Neural Net. Architecture for V, Q

- St vector input → NN → V scalar output

- St, At vector input → NN → Q scalar output
  - Continuous case

- St vector input → Q vector output
  - Discrete action space only
  - Output size is $|A|$

Neural Net. Architecture for policy, π

- St input → NN → vector output
  - Discrete action space case
  - Output size is $|A|$
  - SOFTMAX turns the output into prob. (sum of prob. is 1)

- St input → NN → $\mu_\theta(S_t), \delta_\theta(St)$ output
  - Continuous action space case
  - Represented with Gaussian Distribution

Deep SARSA

Neural network optimization

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

- Target is the value towards which we want to push the estimates.

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

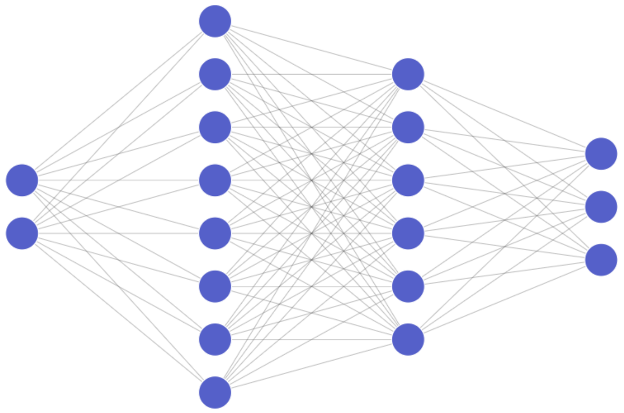
- Estimate is the estimate of the q-value of a state-action pair

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

Target network

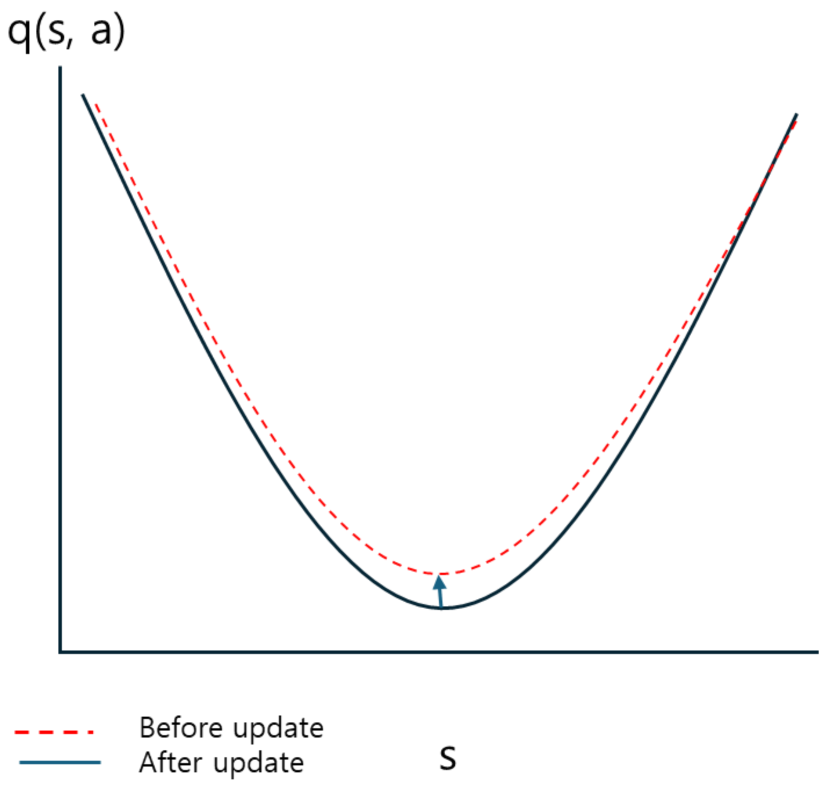
Bootstrapping + Function approximator

$$y_i = R_i + \gamma \hat{q}(S_i', A_i' | \theta_{targ})$$



Target network

- When a value is changed, nearby values will also be affected.
- By modifying a $\hat{q}(S_i, Ai|\theta)$ estimate we also modify its $\hat{q}(S_i', \ A_i'|\theta_{targ})$ target
- For the learning process to be stable, the target must also be stable
- power of neural networks



Target network

- We make a copy of the neural network to calculate the targets.

$$\theta_{targ} \leftarrow \theta$$

- This neural network does not change with SGD. Its θ parameters remain the same
- The estimated value of $\mathrm{S}_i{}'{}_, \mathrm{A}_i{}'$ is calculated with the target network:

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

Neural network optimization

Algorithm 1 Deep SARSA

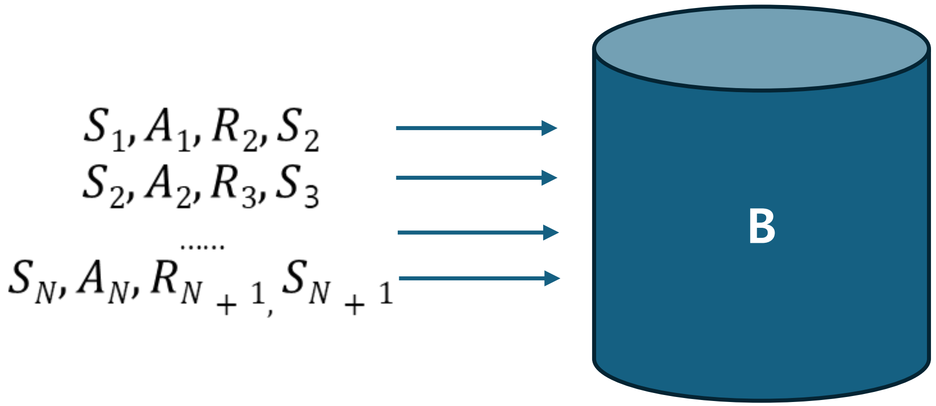
Input: α learning rate, ϵ random action probability,
γ discount factor,
Initialize q-value parameters θ and $\boxed{\text{target parameters } \theta_{targ} \leftarrow \theta}$
$\pi \leftarrow \epsilon$-greedy policy w.r.t $\hat{q}(s, a|\theta)$
Initialize replay buffer B
for episode $\in 1..N$ **do**
  Restart environment and observe the initial state S_0
  for $t \in 0..T-1$ **do**
    Select action $A_t \sim \pi(S_t)$
    Execute action A_t and observe S_{t+1}, R_{t+1}
    Insert transition $(S_t, A_t, R_{t+1}, S_{t+1})$ into the buffer B
    $K = (S, A, R, S') \sim B$
    Select actions $A' \sim \pi(S')$
    Compute loss function over the batch of experiences:

$$\boxed{L(\theta) = \frac{1}{|K|} \sum_{i=1}^{|K|} [R_i + \boxed{\gamma \hat{q}(S'_i, A'_i|\theta_{targ})} - \hat{q}(S_i, A_i|\theta)]^2} \quad (1)$$

  end for
  $\boxed{\text{Every } k \text{ episodes synchronize } \theta_{targ} \leftarrow \theta}$
end for
Output: Near optimal policy π and q-value approximations $\hat{q}(s, a|\theta)$

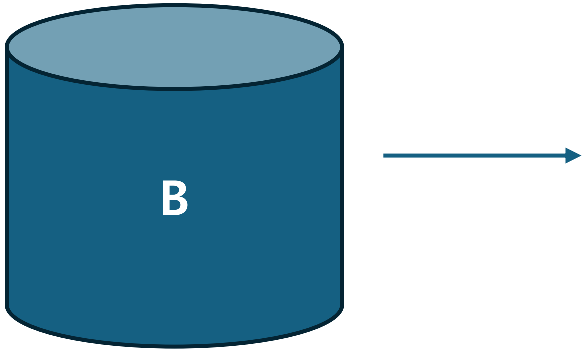
Experience Replay

- Memory that stores the state transition that the agent experiences
- The memory has a limited size and when it fills up, it replaces old transitions with new ones



Experience Replay

- To update the neural network, we randomly chose a batch of transitions from the memory



$$K = (S, A, R, S')\sim B$$

- The batch of transitions obtained from the memory is used to calculate the cost function and update the θ parameters
- $L(\theta) = \frac{1}{|K|}\sum_{\mathrm{i=0}}^{|K|}[\mathrm{R}_i + \gamma\hat{\mathrm{q}}(\mathrm{S}_i', \mathrm{A}_i'|\theta_{targ}) - \hat{\mathrm{q}}(\mathrm{S}_i, \mathrm{A}_i|\theta)]^2$

Neural network optimization

Algorithm 1 Deep SARSA

Input: α learning rate, ϵ random action probability,
γ discount factor,
Initialize q-value parameters θ and target parameters $\theta_{targ} \leftarrow \theta$
$\pi \leftarrow \epsilon$-greedy policy w.r.t $\hat{q}(s, a|\theta)$
Initialize replay buffer B
for episode $\in 1..N$ **do**
    Restart environment and observe the initial state S_0
    for $t \in 0..T-1$ **do**
        Select action $A_t \sim \pi(S_t)$
        Execute action A_t and observe S_{t+1}, R_{t+1}
        Insert transition $(S_t, A_t, R_{t+1}, S_{t+1})$ into the buffer B
        $K = (S, A, R, S') \sim B$
        Select actions $A' \sim \pi(S')$
        Compute loss function over the batch of experiences:

$$L(\theta) = \frac{1}{|K|} \sum_{i=1}^{|K|} \left[R_i + \gamma \hat{q}(S'_i, A'_i|\theta_{targ}) - \hat{q}(S_i, A_i|\theta)\right]^2 \tag{1}$$

    end for
    Every k episodes synchronize $\theta_{targ} \leftarrow \theta$
end for
Output: Near optimal policy π and q-value approximations $\hat{q}(s, a|\theta)$

Code Ex.

- MountainCar: Reach the goal from the bottom of the valley

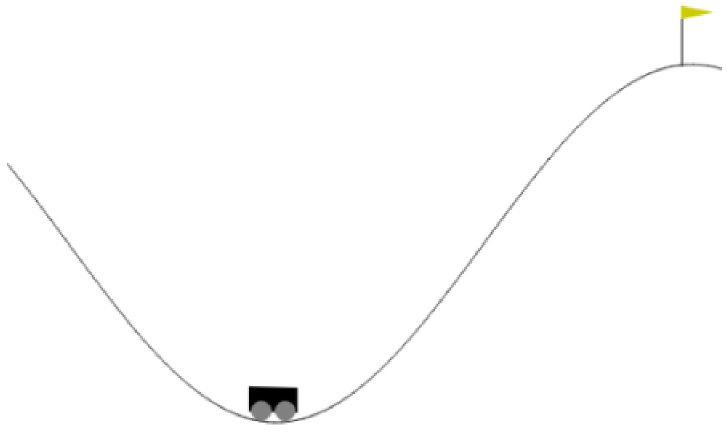
The state

The observation space consists of the car position $\in [-1.2, 0.6]$ and car velocity $\in [-0.07, 0.07]$

The actions available

The actions available three:

```
0    Accelerate to the left.
1    Don't accelerate.
2    Accelerate to the right.
```



Code Ex.

- deep_sarsa_colab.ipynb