

# Lab 8: Deep Reinforcement Learning

University of Washington

ECE 596

Spring 2021

# Outline

## Part 1: Agents in Reinforcement Learning

- Agent and Environment in RL
- Markov Decision Process
- Q-learning

## Part 2: Deep Q-learning

- DQN
- PyTorch implementation of DQN
- Extensions of DQN

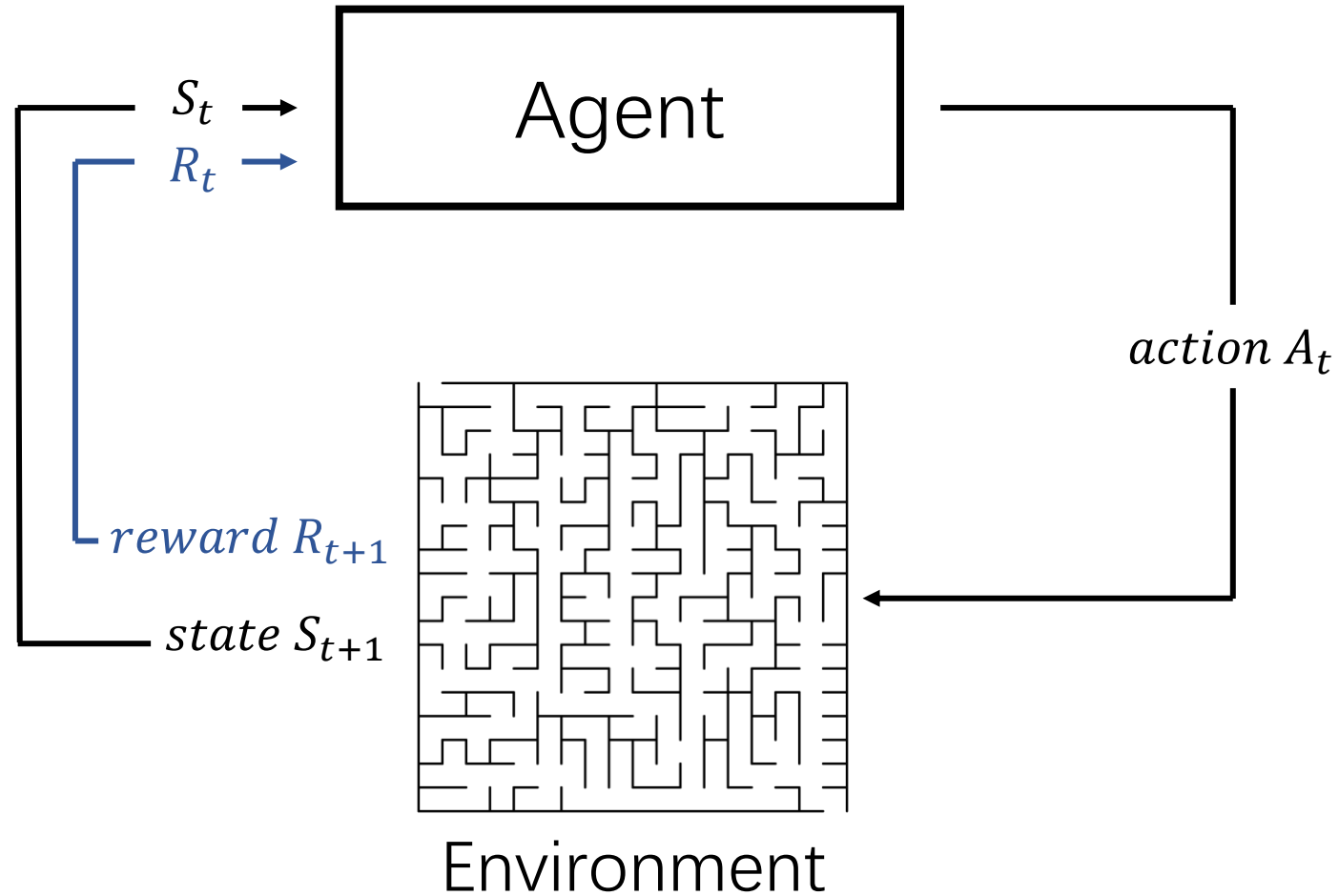
## Part 3: Policy Gradient based Methods

- DDPG
- A3C
- DQN vs Policy Gradient

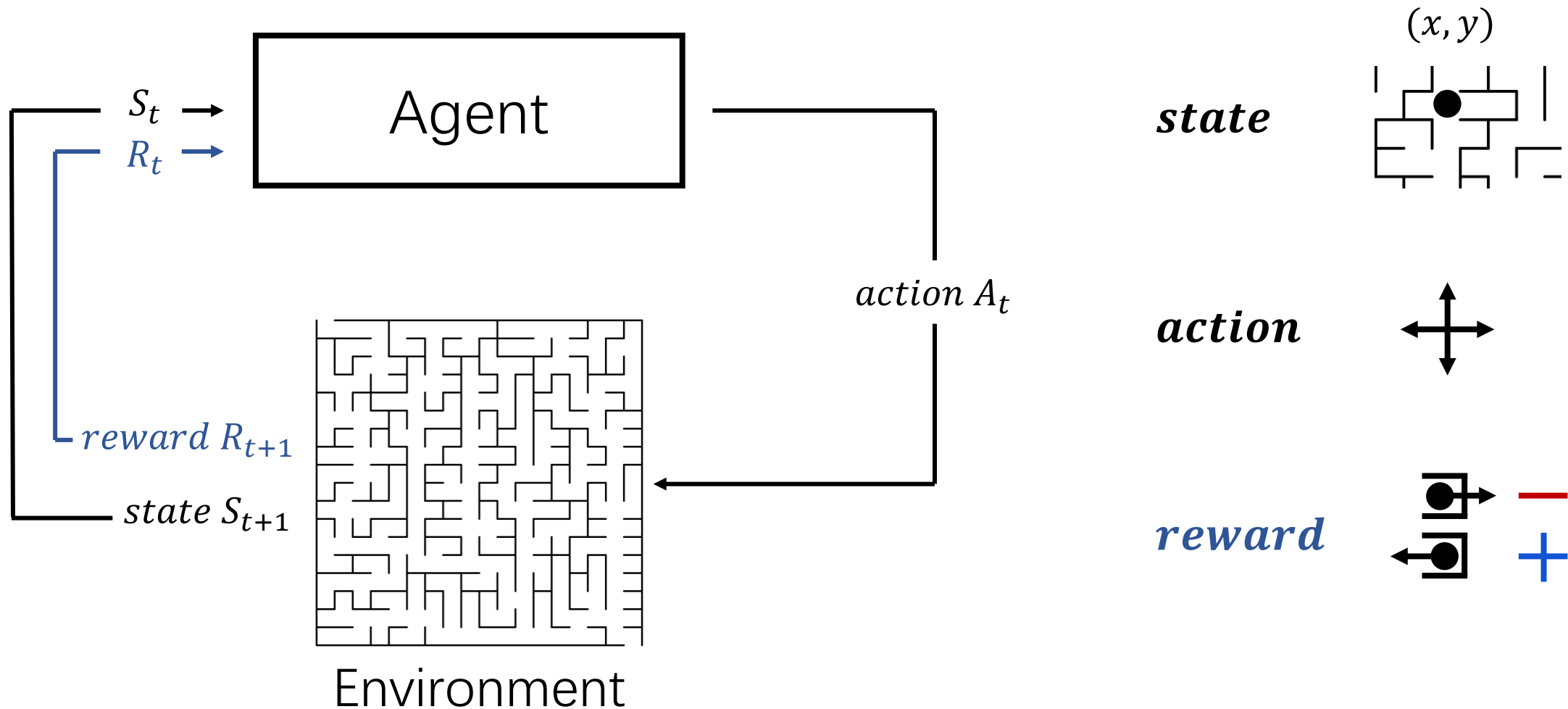
## Lab Assignment

# Part 1: Agents in Reinforcement Learning

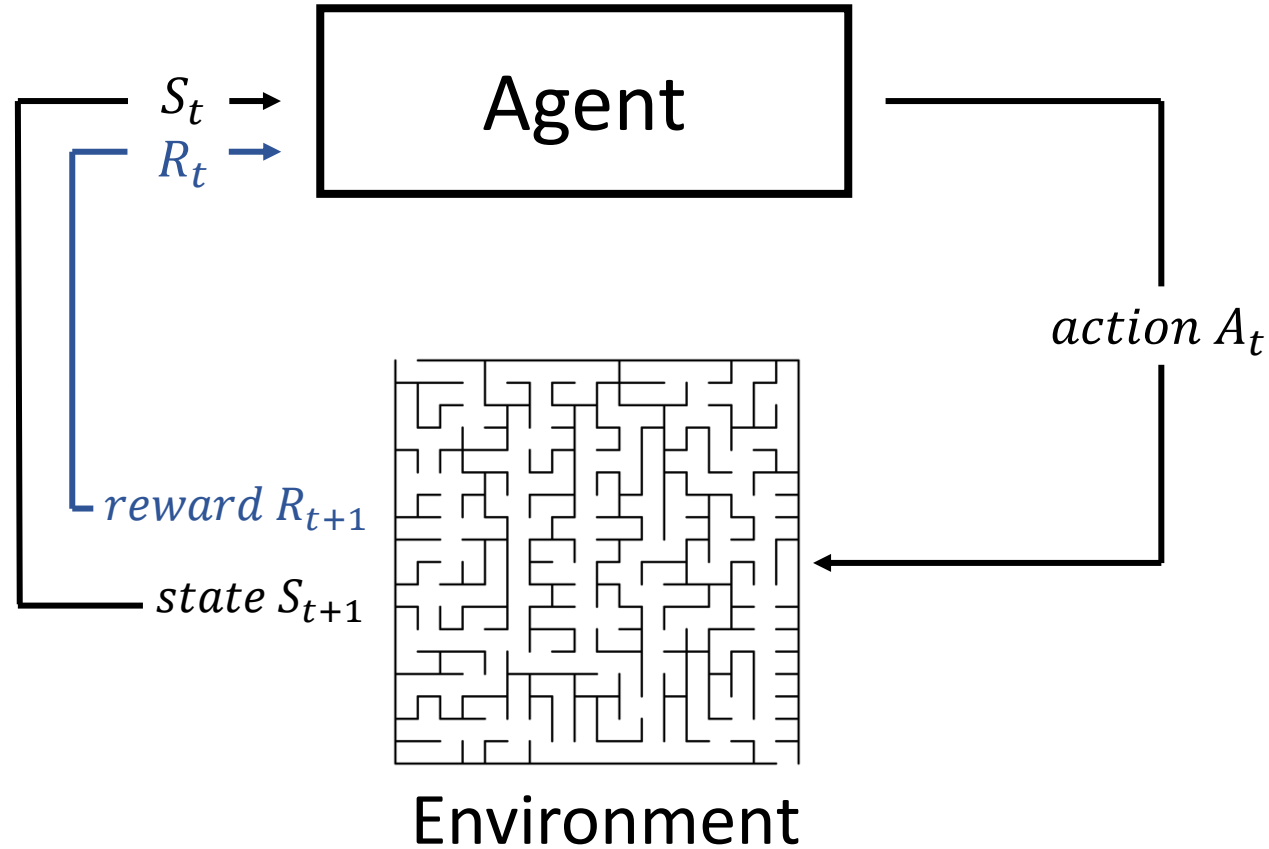
# Agent and Environment in RL



# Agent and Environment in RL



# Markov Decision Process (MDP)



## MDP Tuple

$$[S, A, T, r, \gamma]$$

## State transition function

$$T(s, a, s') = P[S_{t+1} = s' | S_t = s, A_t = a]$$

$\pi(s, a)$  Mapping of state to action

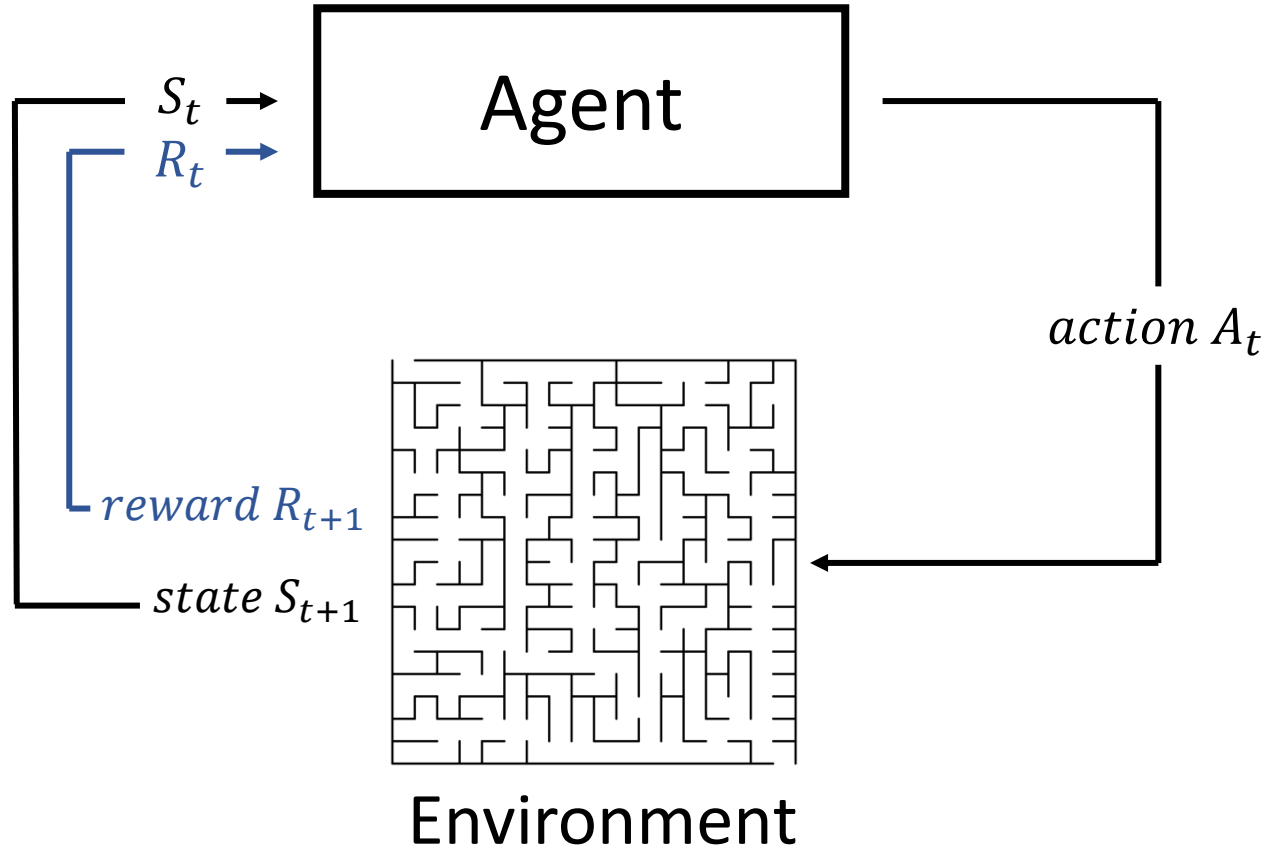
## Reward function

$$R_t = \sum_{k=t}^T \gamma^{k-t} r_k(s_k, a_k)$$

$$\pi^*(s, a) = \operatorname{argmax}(R_t) \text{ for } \forall t$$

Optimal Policy

# Q-Learning



## Markov Decision Process

$$[S, A, T, r, \gamma]$$

$$T(s, a, s') = P[S_{t+1} = s' | S_t = s, A_t = a]$$

$$R_t = \sum_{k=t}^T \gamma^{k-t} r_k(s_k, a_k)$$

$$\pi^*(s, a) = \operatorname{argmax}(R_t) \text{ for } \forall t$$

Optimal Policy

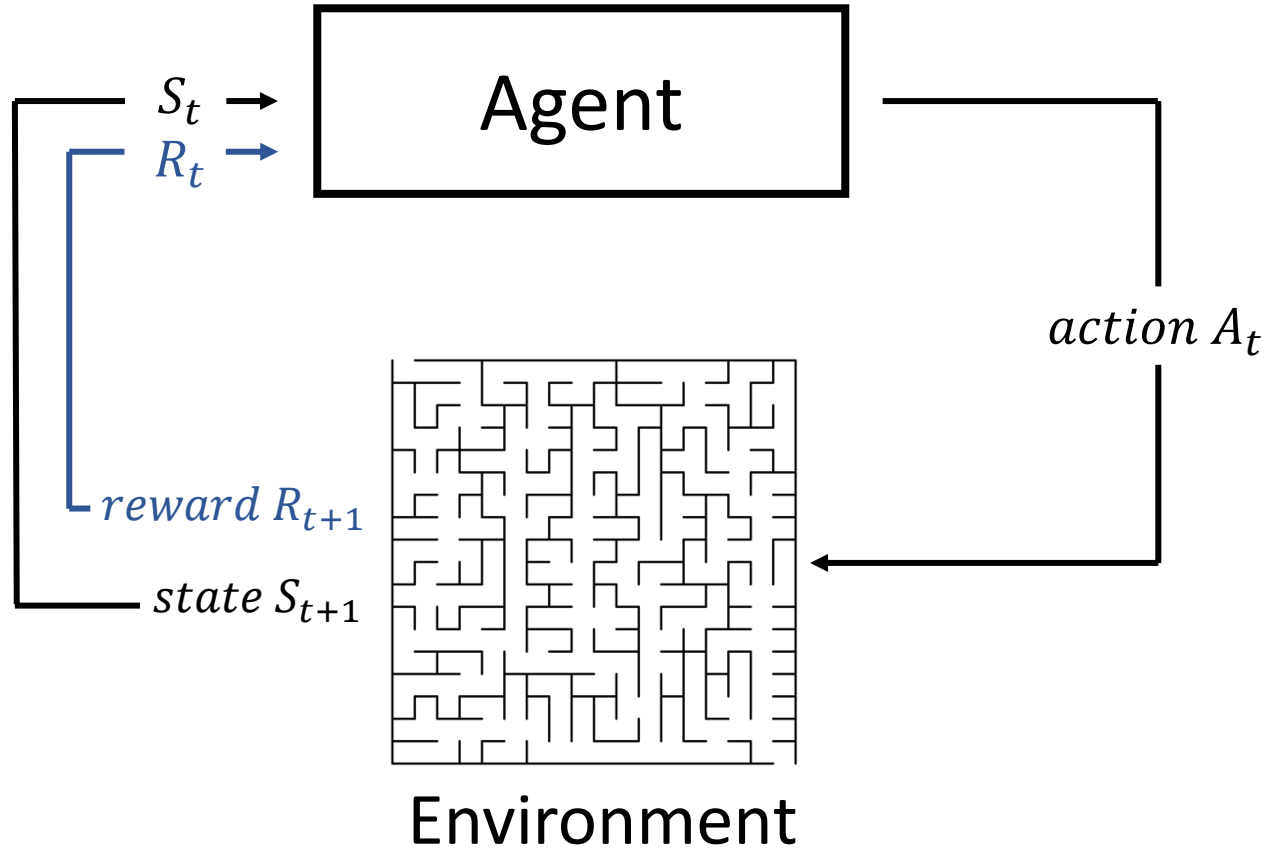
## Q-Learning

$$q^\pi(s, a) = E_\pi[R_t | S_t = s, A_t = a]$$

Policy as a function of  $R_t$  (q-values)

$$\pi^*(s, a) = \operatorname{argmax}_a q^\pi(s, a)$$

# Q-Learning



**Not efficient with large  $S_t$  and  $A_t$**

## Markov Decision Process

$$[S, A, T, r, \gamma]$$

$$T(s, a, s') = P[S_{t+1} = s' | S_t = s, A_t = a]$$

$$R_t = \sum_{k=t}^T \gamma^{k-t} r_k(s_k, a_k)$$

$$\pi^*(s, a) = \operatorname{argmax}(R_t) \text{ for } \forall t$$

Optimal Policy

## Q-Learning

$$q^\pi(s, a) = E_\pi[R_t | S_t = s, A_t = a]$$

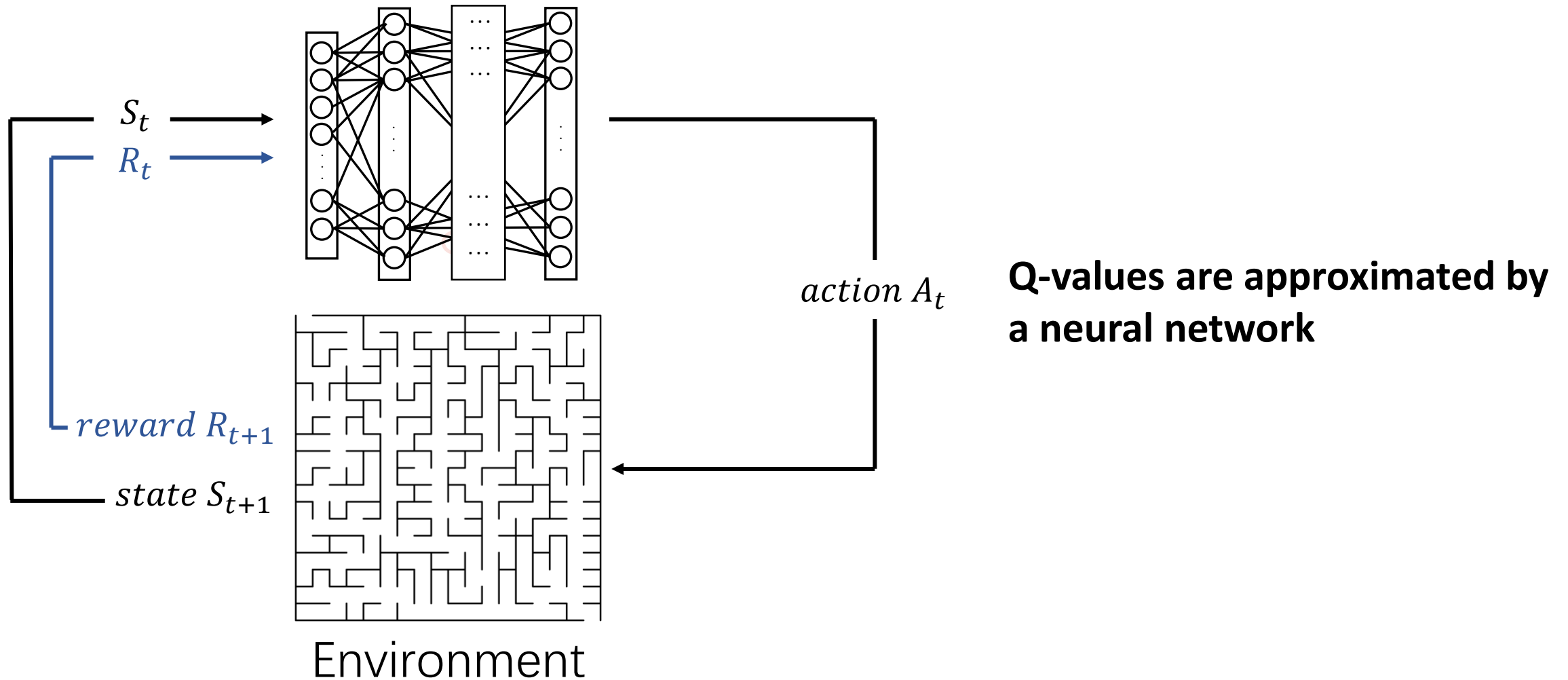
Policy as a function of  $R_t$  (q-values)

$$\pi^*(s, a) = \operatorname{argmax}_a q^\pi(s, a)$$

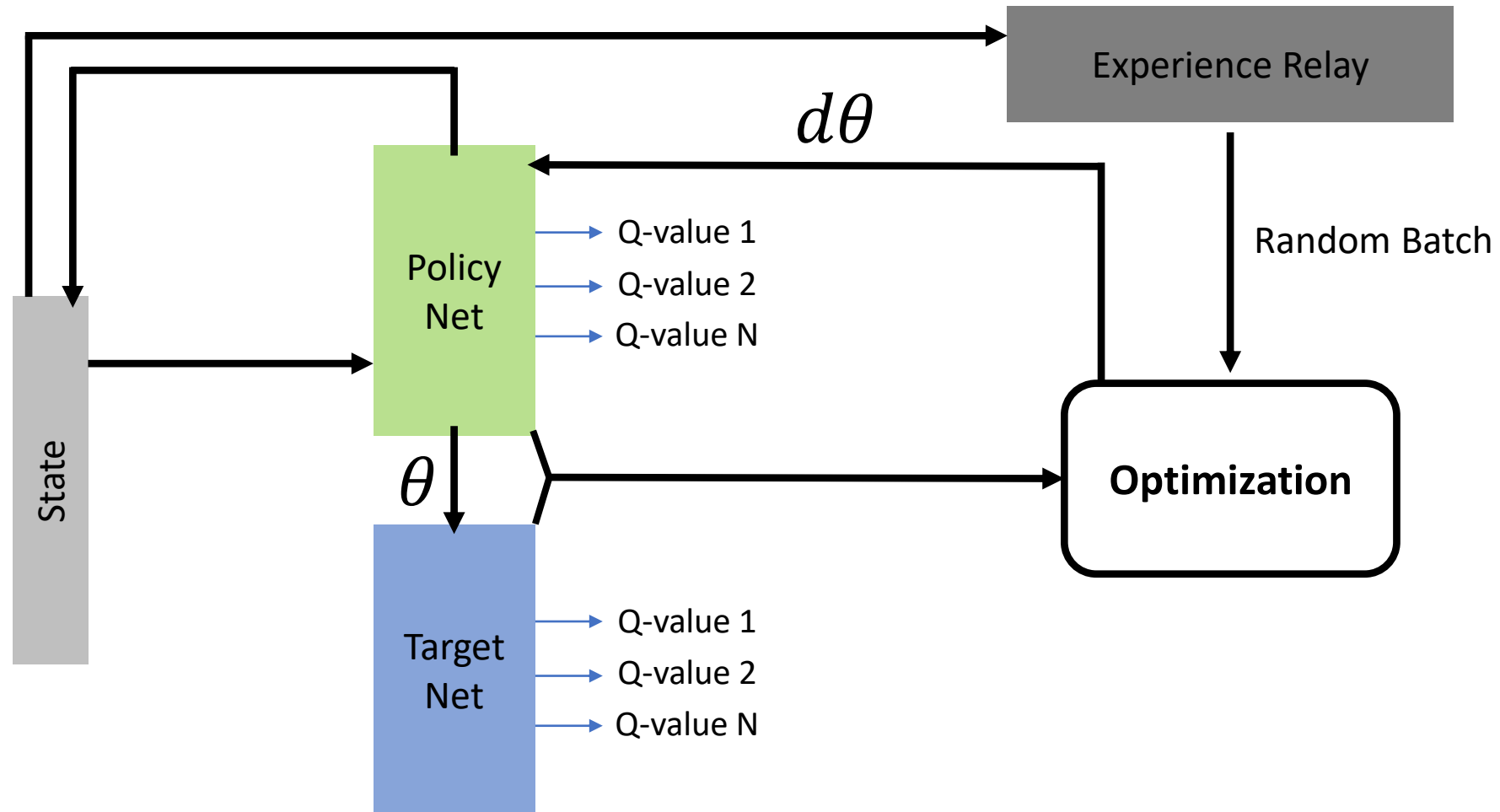


# Part 2: Deep Q-Learning

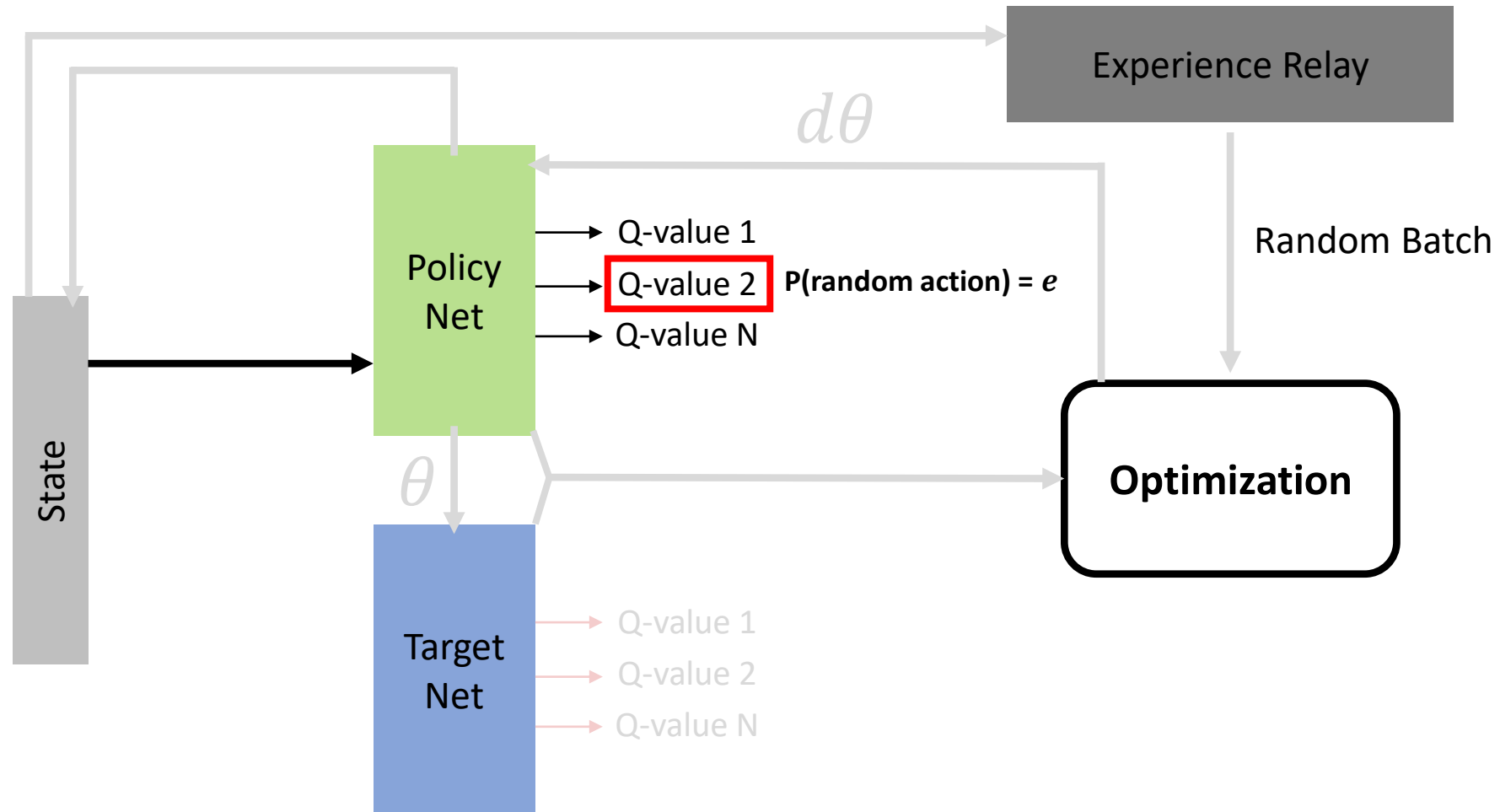
# Deep Q-Network (Mnih et al., 2014)



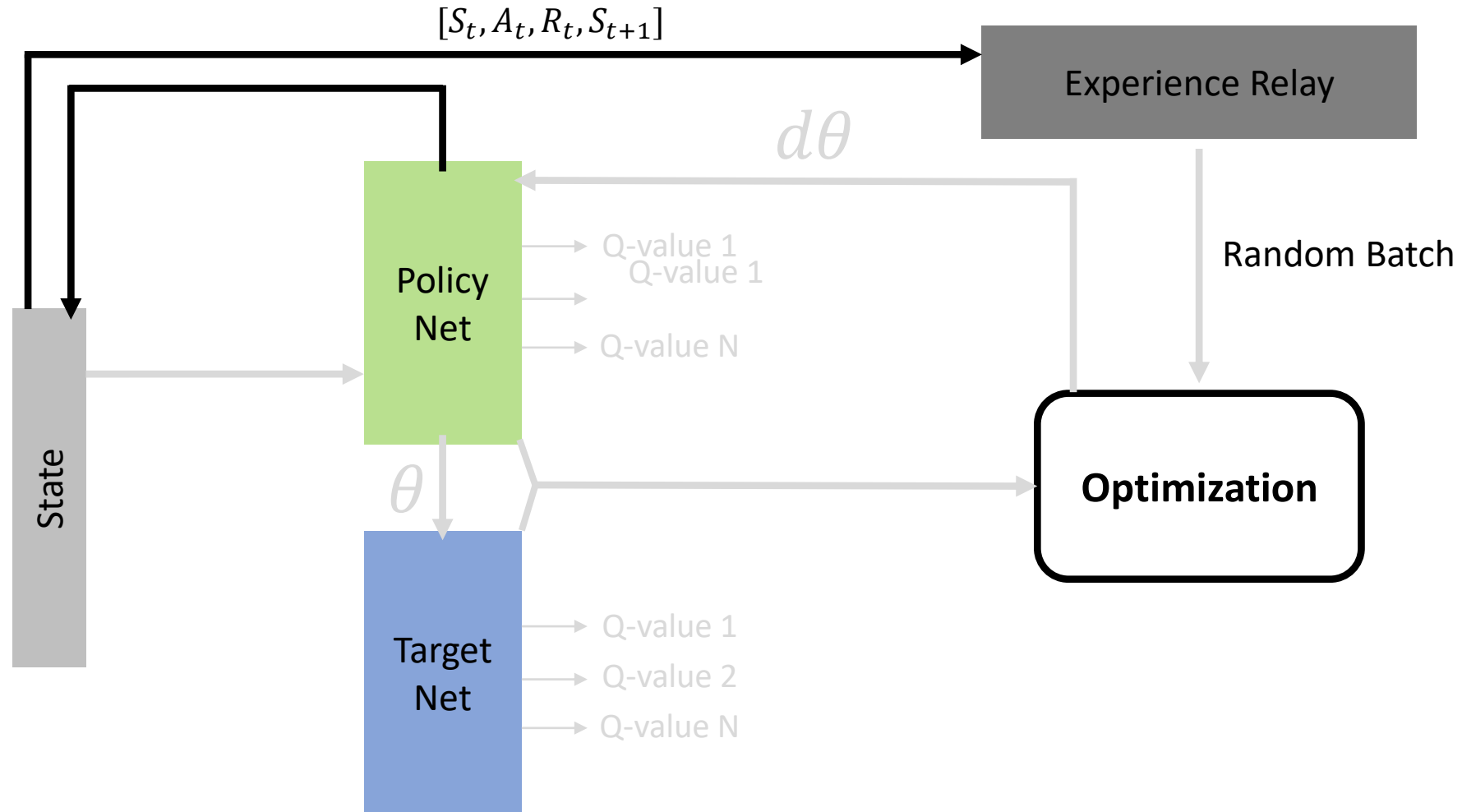
# DQN Architecture



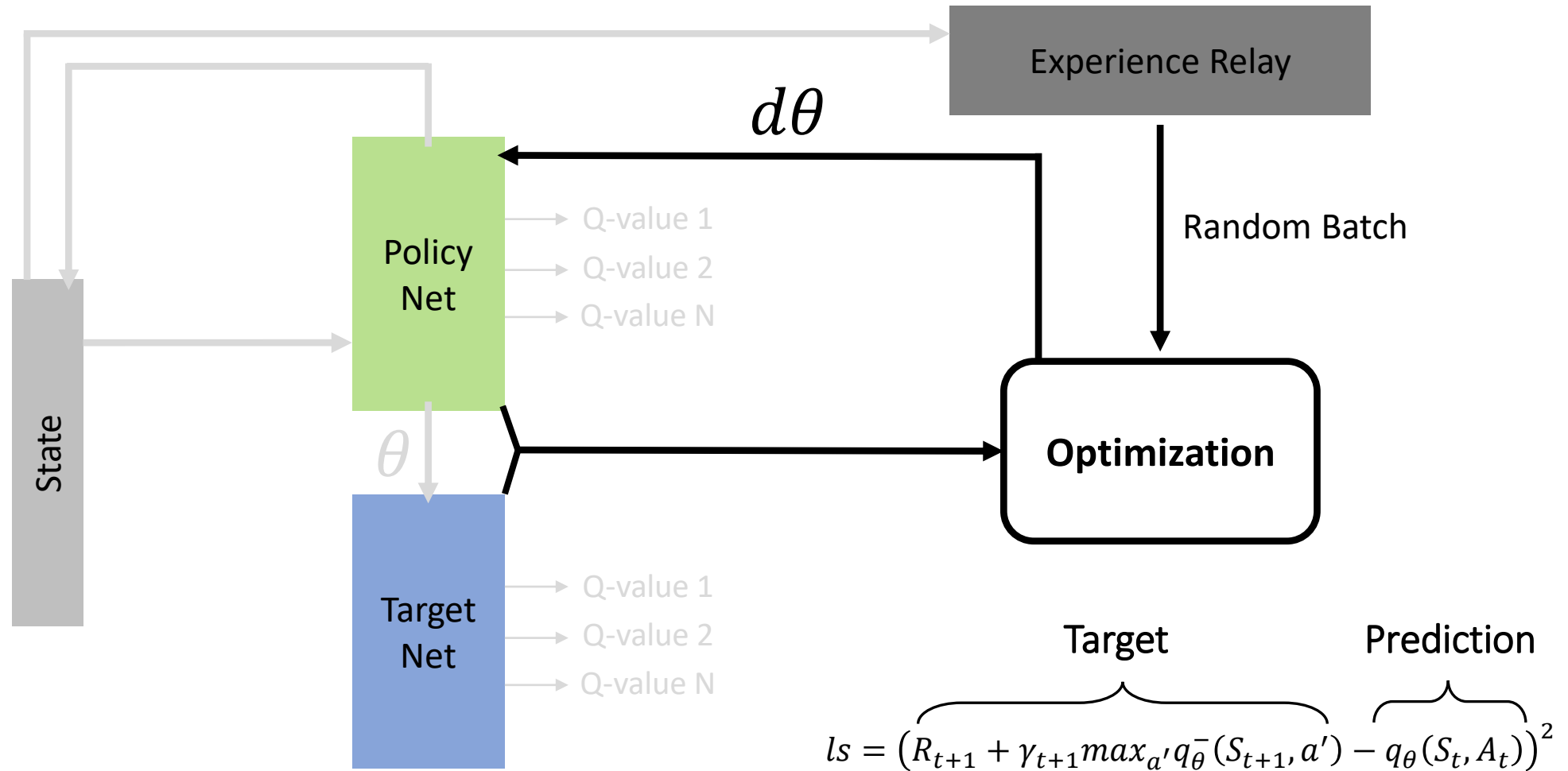
# DQN Architecture



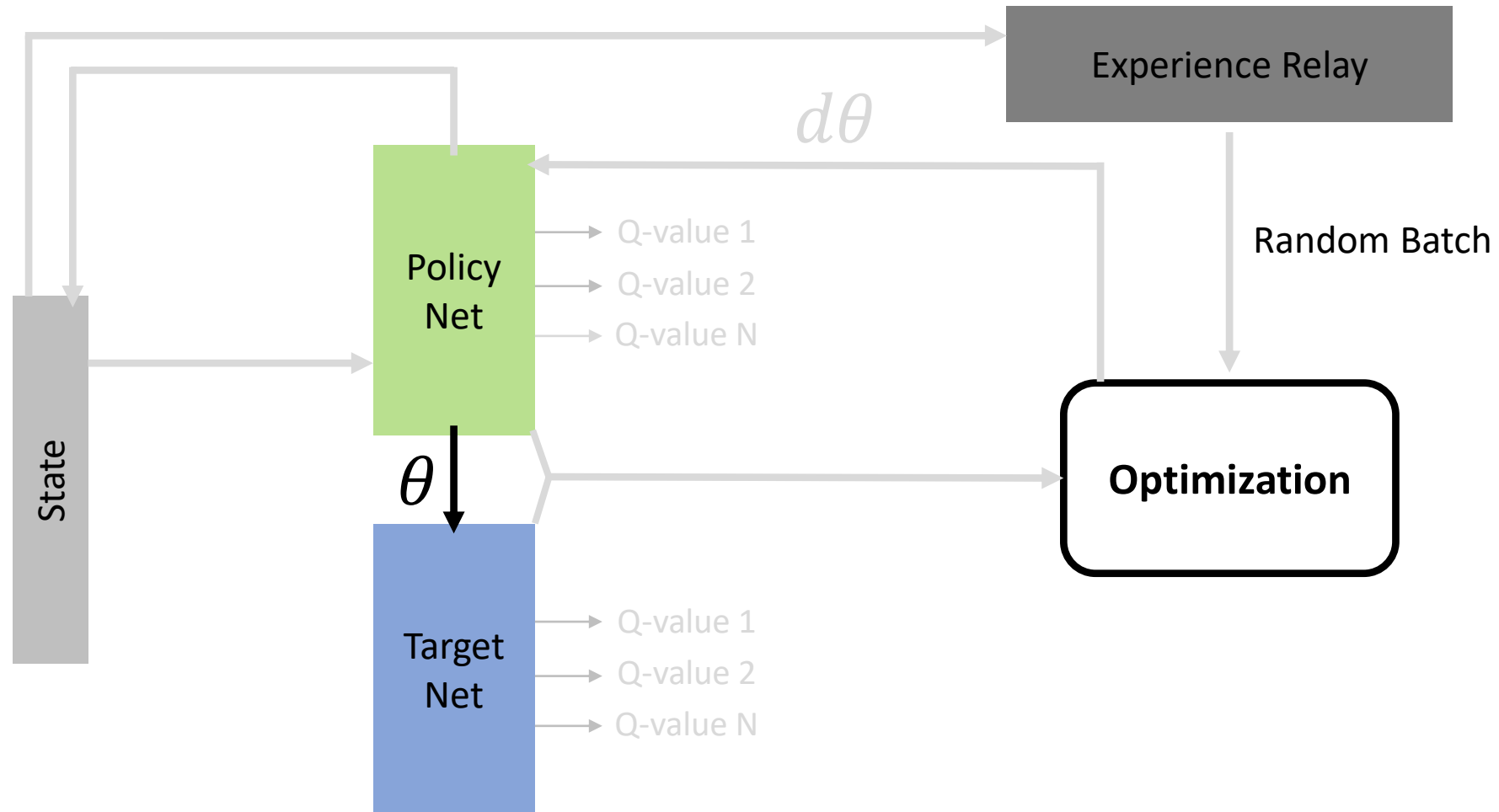
# DQN Architecture



# DQN Architecture



# DQN Architecture



# PyTorch Implementation: Policy/Target networks

```
class Q_network(nn.Module):  
  
    def __init__(self, state_size, action_size, seed, fc1_unit,  
                 fc2_unit):  
  
        super(QNetwork, self).__init__()  
        self.seed = torch.manual_seed(seed)  
        self.fc1 = nn.Linear(state_size, fc1_unit)  
        self.fc2 = nn.Linear(fc1_unit, fc2_unit)  
        self.fc3 = nn.Linear(fc2_unit, action_size)  
  
    def forward(self, x):  
  
        x = F.relu(self.fc1(x))  
        x = F.relu(self.fc2(x))  
        return self.fc3(x)
```

State\_size : Dimension of each state

Action\_size : Dimension of each action

Seed : random seed

Fc1\_unit : # of neurons in first hidden layer

Fc2\_unit : # of neurons in second hidden layer



# PyTorch Implementation: Agent (initialization)

```
class Agent():  
  
    def __init__(self, state_size, action_size, seed):  
  
        self.state_size = state_size  
        self.action_size = action_size  
        self.seed = random.seed(seed)  
  
        #Q- Network  
        self.qnetwork_policy = Q_network(state_size, action_size, seed).to(device)  
        self.qnetwork_target = Q_network(state_size, action_size, seed).to(device)  
  
        self.optimizer = optim.Adam(self.qnetwork_policy.parameters(),lr=LR)  
  
        # Replay memory  
        self.memory = ExperienceRelay(action_size, BUFFER_SIZE,BATCH_SIZE,seed)  
        # Initialize time step (for updating every UPDATE_EVERY steps)  
        self.t_step = 0
```

Network initializations

Define optimizer

Memory initializations

# PyTorch Implementation: Agent (ENV interactions + Memory Retrieval)

```
def step(self, state, action, reward, next_step, done):  
    # Save experience in replay memory  
    self.memory.add(state, action, reward, next_step, done)
```

Store [s, a, r, s'] to memory

```
    # Learn every UPDATE_EVERY time steps.  
    self.t_step = (self.t_step+1)% UPDATE_EVERY  
    if self.t_step == 0:  
        # If enough samples are available in memory, get radom subset and Learn
```

```
        if len(self.memory)>BATCH_SIZE:  
            experience = self.memory.sample()  
            self.learn(experience, GAMMA)
```

Retrieve memory and learn every learning steps

```
def act(self, state, eps = 0):  
  
    state = torch.from_numpy(state).float().unsqueeze(0).to(device)  
    self.qnetwork_policy.eval()  
    with torch.no_grad():  
        action_values = self.qnetwork_policy(state)  
    self.qnetwork_policy.train()  
  
    #Epsilon -greedy action selction  
    if random.random() > eps:  
        return np.argmax(action_values.cpu().data.numpy())  
    else:  
        return random.choice(np.arange(self.action_size))
```

Choose action based on e-greedy action selection

# PyTorch Implementation: Agent (Learning + update Target network)

```
def learn(self, experiences, gamma):

    states, actions, rewards, next_state, dones = experiences
    criterion = torch.nn.MSELoss()
    self.qnetwork_policy.train()
    self.qnetwork_target.eval()
    predicted_targets = self.qnetwork_policy(states).gather(1,actions)

    with torch.no_grad():
        labels_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze(1)

    labels = rewards + (gamma* labels_next*(1-dones))

    loss = criterion(predicted_targets,labels).to(device)
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()

    # ----- update target network ----- #
    self.soft_update(self.qnetwork_policy,self.qnetwork_target,TAU)

def soft_update(self, local_model, target_model, tau):

    for target_param, local_param in zip(target_model.parameters(),
                                         local_model.parameters()):
        target_param.data.copy_(tau*local_param.data + (1-tau)*target_param.data)
```

Set policy network to train mode and target network to eval mode

Compute Loss (target - predicted)

Backpropagate gradients

Soft transfer weights to target network

# PyTorch Implementation: Experience Relay

```
class ExperienceRelay:

    def __init__(self, action_size, buffer_size, batch_size, seed):

        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experiences = namedtuple("Experience", field_names=["state",
                                                                "action",
                                                                "reward",
                                                                "next_state",
                                                                "done"])

        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        e = self.experiences(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not None])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not None]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        return len(self.memory)
```

Store [s, a, r, s'] as tuple

Uniform sample the batch

# Extensions of DQN

Double DQN

Prioritized Experience Relay

Dueling DQN

# Extensions of DQN – Double DQN

Q-values and action chosen by target network

$$ls = \left( R_{t+1} + \gamma_{t+1} \overbrace{\max_{a'} q_{\theta^-}(S_{t+1}, a')} - q_{\theta}(S_t, A_t) \right)^2$$

Action evaluated by target network

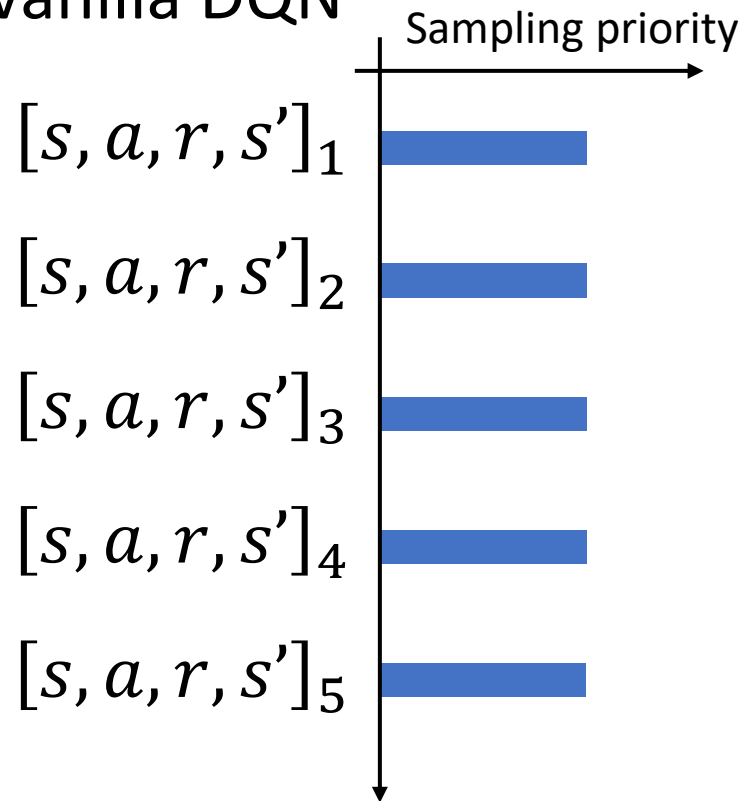
$$ls\_double = \left( R_{t+1} + \gamma_{t+1} \overbrace{q_{\theta^-}(s', \underbrace{\arg\max_{a'} q_{\theta}(S_{t+1}, a')}_{\text{Action chosen by policy network}})) - q_{\theta}(S_t, A_t) \right)^2$$

Action chosen by policy network

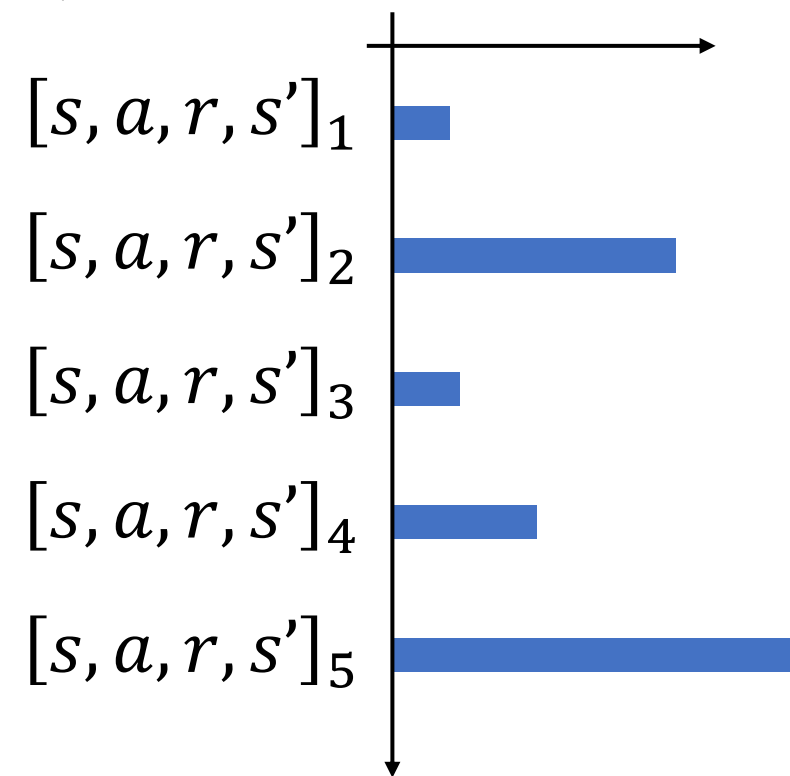
# Extensions of DQN – Prioritized Experience Replay

$$p_t \propto \left| R_{t+1} + \gamma \max_{a'} q_{\bar{\theta}}(S_{t+1}, a') - q_{\theta}(S_t, A_t) \right|^{\omega}$$

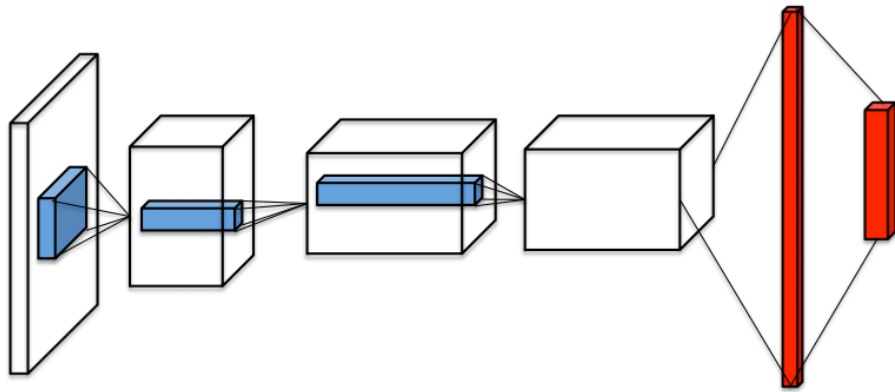
Vanilla DQN



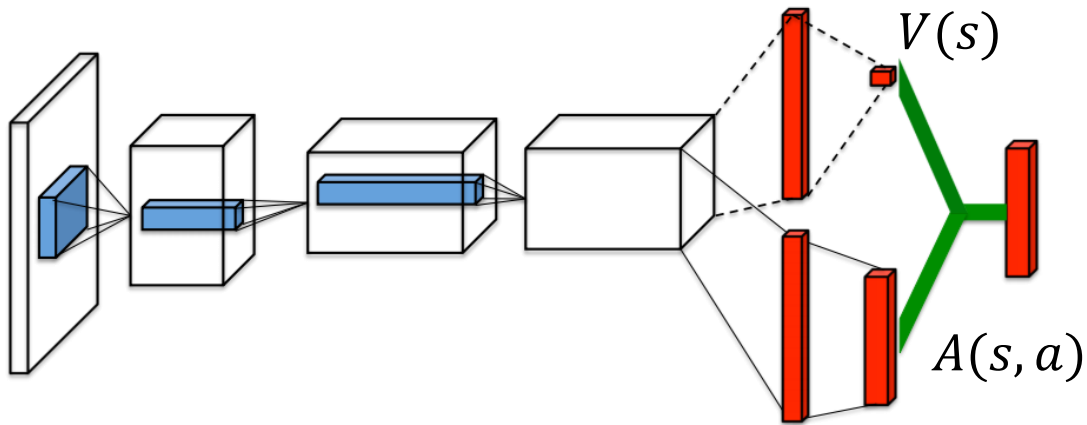
DQN with PER



# Extensions of DQN – Dueling DQN



Vanilla Deep Q-Network



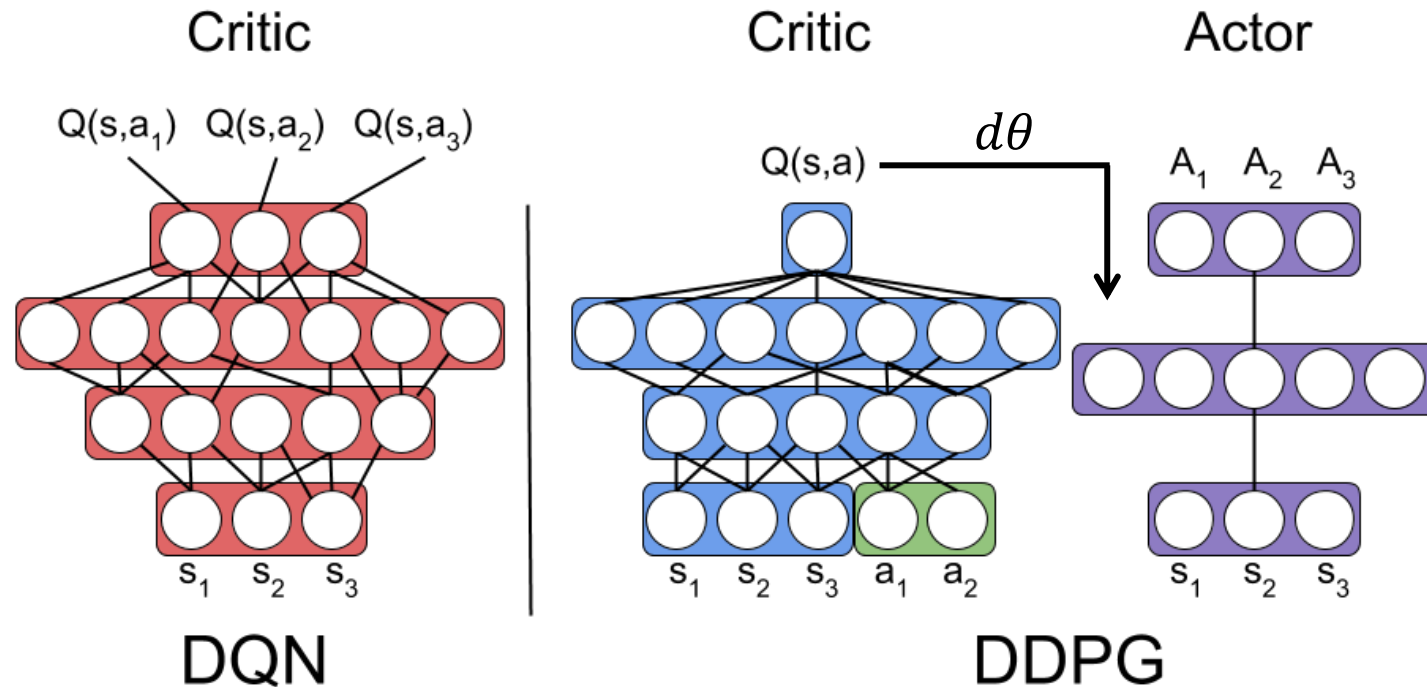
Dueling Deep Q-Network

$$Q(s, a) = V(s) + \left( A(s, a) - \frac{1}{|A|} \sum_a A(s, a) \right)$$



# Part 3: Policy Gradient based Methods

# Deep Deterministic Policy Gradient (DDPG)



**DQN** – Network

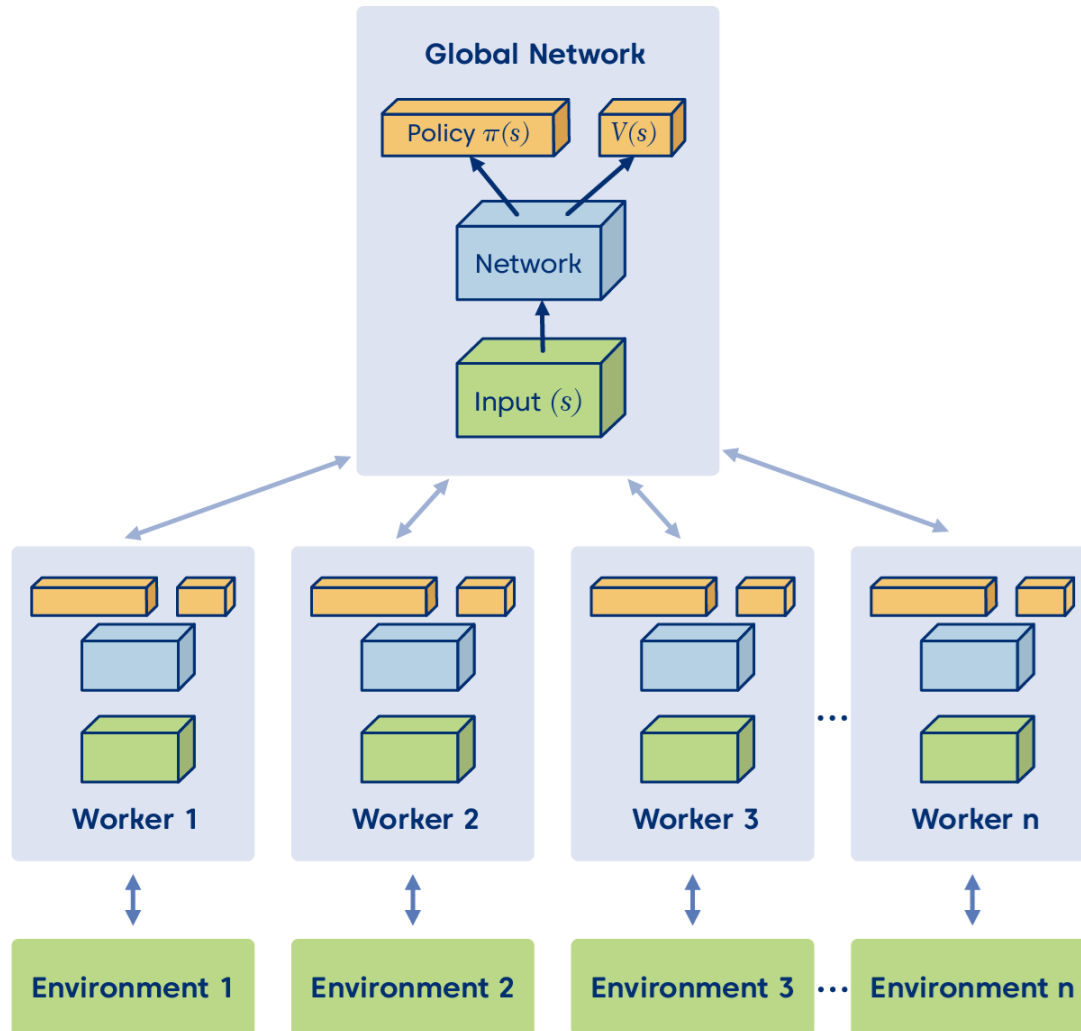
approximates **Value** function

**DDPG** – Network

approximates **Policy** function

Implementation in PyTorch: <https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b>

# Asynchronous Advantage Actor-Critic (A3C)



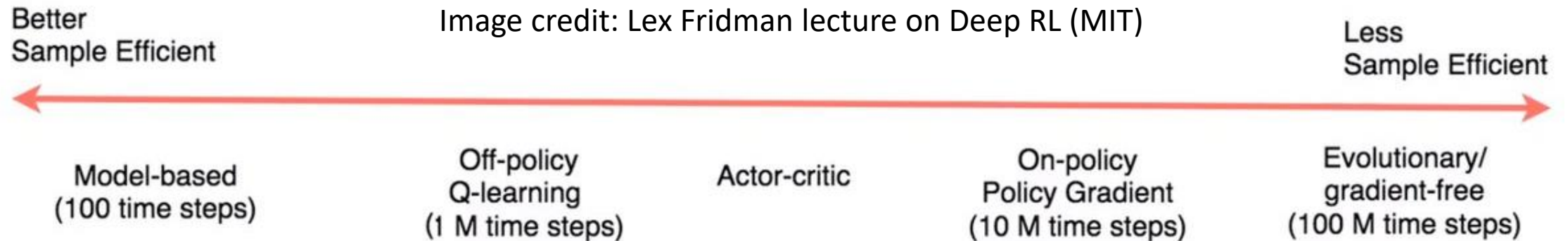
**Asynchronous:** Utilizes multiple agents each with unique experiences

**Actor-Critic:** Agent uses the value –  $V(s)$  estimate (critic) to update –  $\pi(s)$  policy (actor)

**Advantage:** discounted rewards –  $V(s)$

Implementation in PyTorch:  
<https://github.com/ikostrikov/pytorch-a3c>

# DQN vs Policy Gradient



## + vs DQN

- Better with env where Q-function is difficult to learn
- Can be applied on continuous action space
- Can learn stochastic policies
- Relatively faster to convergence

## — vs DQN

- Relatively sample inefficient
- Less stable during training
- Tendency to converge to local minima
- Poor handling of delayed rewards

## Lab Assignment:

Solve OpenAI cartpole problem using Deep RL

# OpenAI Gym - Cartpole\_v1



Cartpole documentation:

<https://gym.openai.com/envs/CartPole-v1/>

## State space:

- Position of a cart
- Velocity of a cart
- Angle of pole
- Rotation rate of pole

## Action space

- Move left (-1)
- Move right (+1)

## Performance evaluation:

Attain average rewards (frames lasted) of >475 over 100 consecutive trials.

Generate a cartpole rendering controlled by trained agent

# OpenAI Gym - Cartpole\_v1



\*Training loop code template + rendering function  
included in the lab8\_template.ipynb

## State space:

- Position of a cart
- Velocity of a cart
- Angle of pole
- Rotation rate of pole

## Action space

- Move left (-1)
- Move right (+1)

## Performance evaluation:

Attain average rewards (frames lasted) of >475  
over 100 consecutive trials.

Generate a cartpole rendering controlled by  
trained agent