Deep Reinforcement Learning

Part 3

Outlines

1. Value-based Agent
    a. Learning Value Network
    b. Deep Q-Learning
2. Policy-based Agent
3. Actor-Critic

Function Approximation in Deep RL

Function approximation in Deep RL can be broadly divided into two main approaches:

Value Function Approximation: In this approach, **the value function, either $v_\pi(s)$ or $q_\pi(s, a)$, is represented using neural networks**. These networks aim to approximate the values associated with states or state-action pairs. This approach is commonly used in algorithms like Deep Q-Networks (DQN), where the goal is to learn the optimal action-value function.

Policy Function Approximation: This approach focuses on **directly representing the policy function, $\pi(a|s)$, using a neural network**. The policy network learns to output the probability distribution over actions for a given state. Methods like Trust Region Policy Optimization (TRPO) and Proximal Policy Optimization (PPO) are examples of algorithms that fall into this category, as they aim to find an optimal policy directly.

Reinforcement Learning

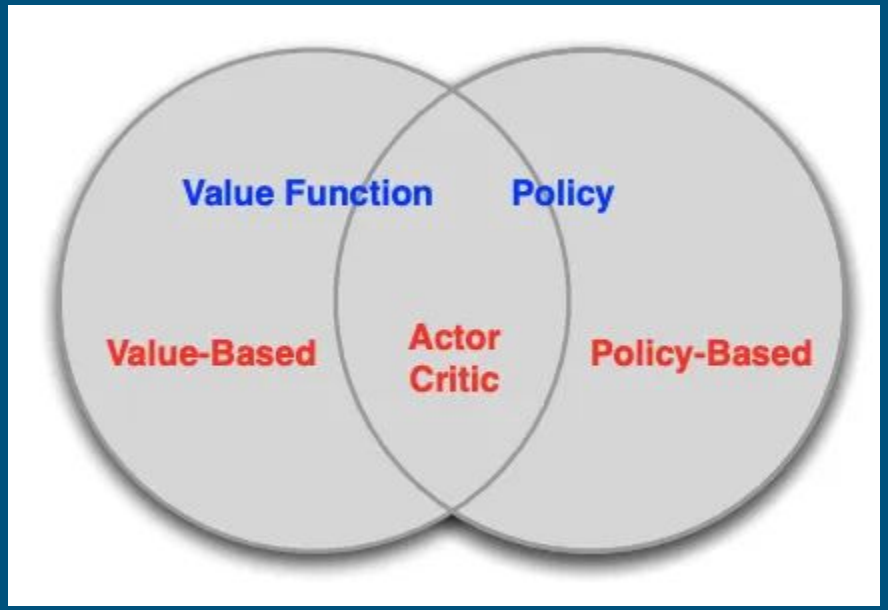
Indeed, in the realm of reinforcement learning, we can categorize agents into three main types:

Value-Based Agent: This type of agent operates with a value function. It assesses the value of different states or state-action pairs and **makes decisions based on these value estimates**. Common algorithms include Q-learning and Deep Q-Networks (DQN).

Policy-Based Agent: Policy-based agents **make decisions directly based on a policy function**. These agents learn a policy that defines the probability of taking different actions in various states. Methods like REINFORCE and Proximal Policy Optimization (PPO) fall under this category.

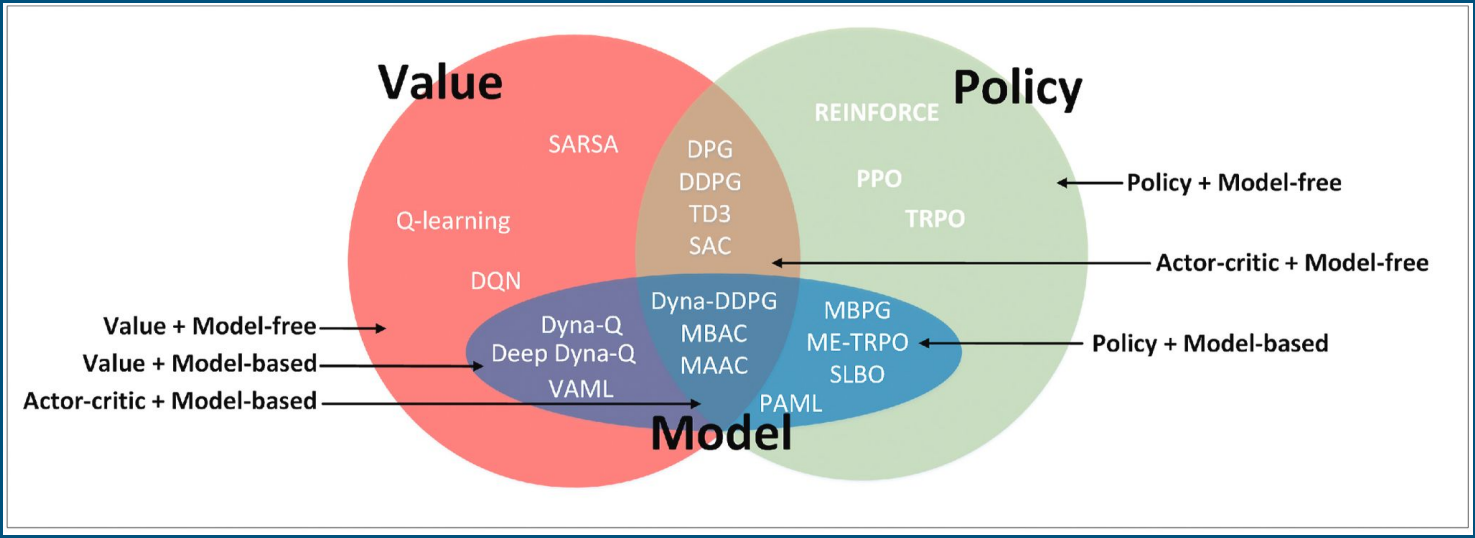
Actor-Critic Agent: Actor-critic agents **combine aspects of both value-based and policy-based methods**. They employ an actor (policy) network to decide actions and a critic (value) network to evaluate the expected rewards. The actor tries to maximize expected rewards, guided by the critic's feedback. Examples include Advantage Actor-Critic (A2C) and Advantage Actor-Critic with Generalized Advantage Estimation (A3C with GAE).

Reinforcement Learning



https://www.davidsilver.uk/wp-content/uploads/2020/03/pg.pdf

Reinforcement Learning



https://www.sciencedirect.com/science/article/pii/S1364032120309023#fig6

Value-based Agent

Value-based Agent

1. Learning Value Network
2. Deep Q-Learning

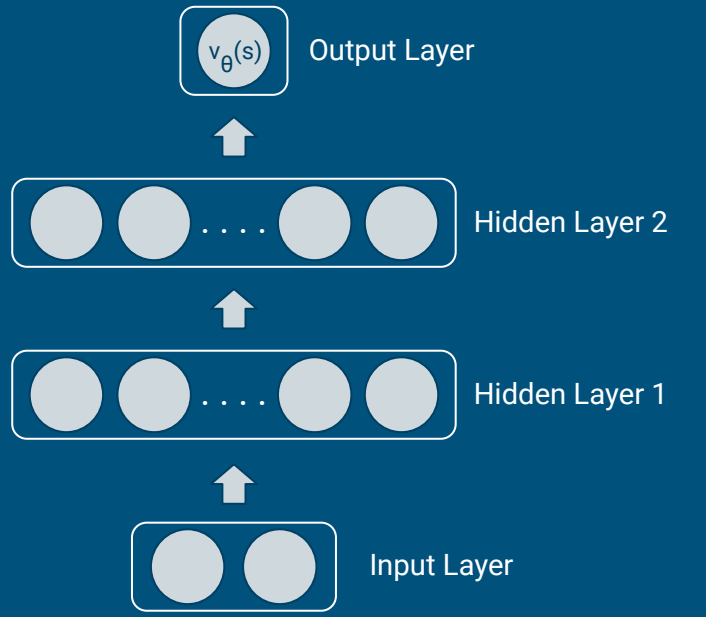
Value-based Agent

1. **Learning Value Network**
2. Deep Q-Learning

Value Network

When the policy π is fixed, let's explore how to use a neural network to learn the value function $v_{\theta,\pi}(s)$.

Here, we have a **value network**, which is a neural network represented as shown in the figure. In this neural network, θ represents the parameters. This network is designed to **return the state value for a given input state s**.



Loss function

We may not know the actual state values, but let's define them as $v_{true}(s)$. Then, we can define the loss function as follows:

$$L(\theta)=E_{\pi}[(v_{true}(s)-v(s))^2]$$

Next, we calculate the gradient of this loss function with respect to the parameters θ.

$$\nabla_{\theta}L(\theta)=-E_{\pi}[(v_{true}(s)-v_{\theta}(s))\nabla_{\theta}v_{\theta}(s)]$$

This is induced using the equation,

$$\frac{d}{dx}\{c-f(x)\}^2=-2\{c-f(x)\}*\frac{d}{dx}f(x)$$

Parameter update

Once the gradient has been computed, the next step is to **update the parameters**.

$$\theta = \theta - \alpha \nabla_{\theta} L(\theta)$$

$$\boldsymbol{\theta = \theta + \alpha(v_{true}(s) - v_{\theta}(s)) \nabla_{\theta} v_{\theta}(s)}$$

Since we cannot obtain the actual value function $v_{true}(s)$, we will approximate it using methods like **Monte Carlo (MC)** or **Temporal Difference (TD)** methods.

Parameter update

MC:

$$L(\theta) = E_\pi[(G_t - v_\theta(s_t))^2]$$
$$\theta = \theta + \alpha(G_t - v_\theta(s_t))\nabla_\theta v_\theta(s_t)$$

TD:

$$L(\theta) = E_\pi[(r_{t+1} + \gamma v_\theta(s_{t+1}) - v_\theta(s_t))^2]$$
$$\theta = \theta + \alpha(r_{t+1} + \gamma v_\theta(s_{t+1}) - v_\theta(s_t))\nabla_\theta v_\theta(s_t)$$

Value-based Agent

1. Learning Value Network
2. **Deep Q-Learning**

Deep Q-Learning

A **value-based agent** operates without explicit policies. There is **no π**.

Instead, it employs **q(s, a)** as if it were a policy—a **implicit policy**.

In Q-learning, tables were used, but Deep Q-learning utilizes **neural networks**.

In Q-learning, the **Bellman Optimality Equation** was used. Review the equations!

$$q_*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \, max_{a'} \, q_*(s',a')\right]$$

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma max_{A'} Q(S',A') - Q(S,A))$$

Loss function & Parameter update

Loss function

$$L(\theta) = E_{\pi}[(r + \gamma \max_{a'} Q_{\theta}(s', a') - Q_{\theta}(s, a))^2]$$

Parameter update

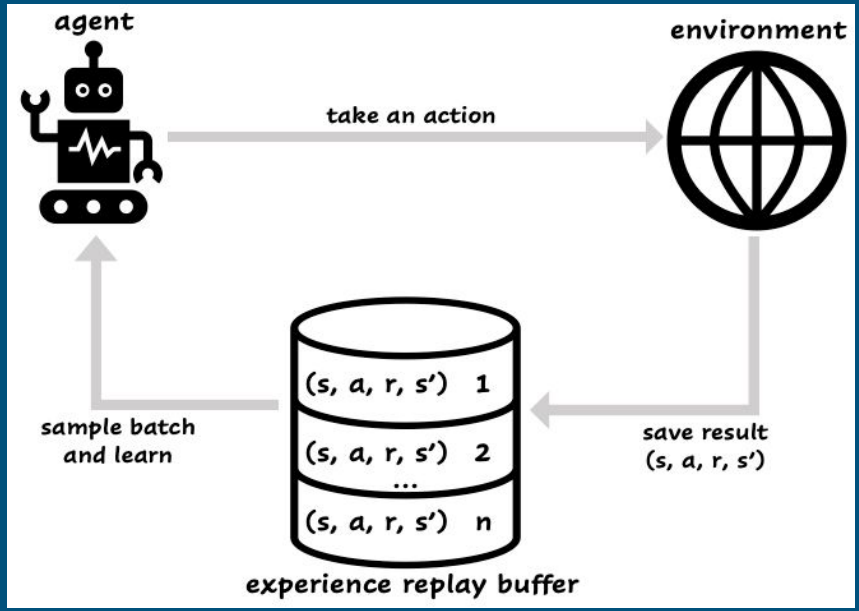
$$\theta = \theta + \alpha(r_{t+1} + \gamma \max_{a'} Q_{\theta}(s', a') - Q_{\theta}(s, a)) \nabla_{\theta} Q_{\theta}(s, a)$$

Mini-batch

When using the update equation to update parameters in reinforcement learning, **a mini-batch is a collection of data**. A mini-batch can consist of just one transition (s, a, r, s') sample, or it can include thousands or even tens of thousands of samples. The size of this mini-batch is defined by the user.

The mini-batch size is a user-defined parameter. It can have an impact on the performance and efficiency of the learning algorithm. A small mini-batch updates parameters more frequently but can introduce noise, while a larger mini-batch provides smoother updates but may slow down learning.

Replay Memory



Target Network

$$L(\theta) = E_{\pi}[(r + \gamma \max_{a'} Q_{\theta}(s', a') - Q_{\theta}(s, a))^2]$$

Target Network (Target Policy) Q Network (Behavior Policy)

Deep Q-learning pseudo code

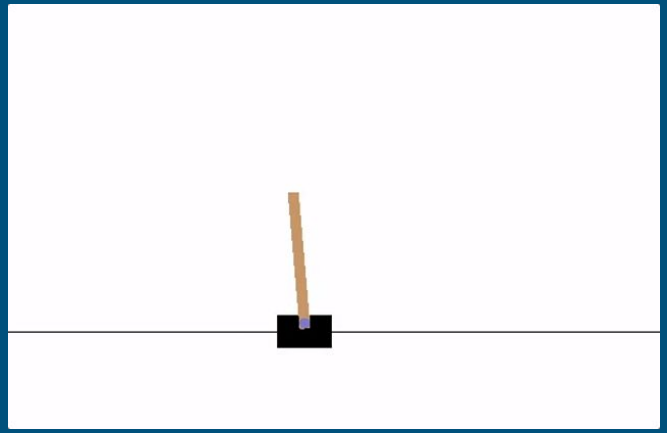
```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
    select an action a
        with probability ε select a random action
        otherwise select a = argmaxa'Q(s,a')
    carry out action a
    observe reward r and new state s'
    store experience <s, a, r, s'> in replay memory D

    sample random transitions <ss, aa, rr, ss'> from replay memory D
    calculate target for each minibatch transition
        if ss' is terminal state then tt = rr
        otherwise tt = rr + γmaxa'Q(ss', aa')
    train the Q network using (tt - Q(ss, aa))^2 as loss

    s = s'
until terminated
```

Implementation of DQN

The problem we want to solve is **CartPole**. CartPole is one of the environments included in the OpenAI Gym library, provided by the company OpenAI. Among these environments, it is one of the simplest.

In CartPole, the objective is to balance a pole on a moving cart by moving the cart left or right. There are only **two possible actions**: move the cart left or move the cart right. A **reward of +1** is given at each time step as long as the pole remains upright. The state is represented as a vector of length 4:

s = (cart position, cart velocity, pole angle, pole angular velocity)

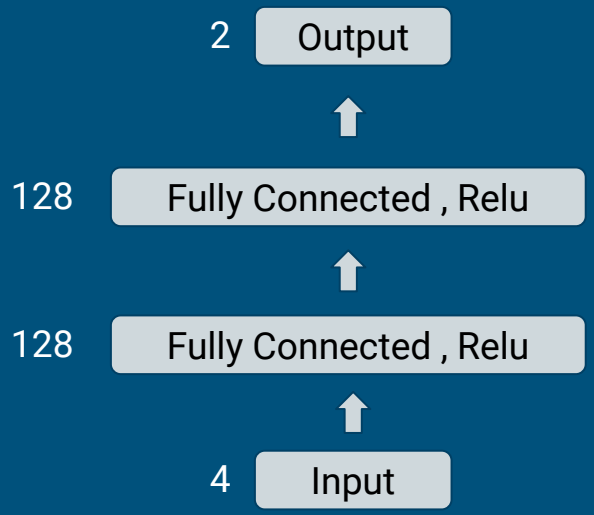


Library import and Hyper-parameters

```
import gym
import collections
import random

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

learning_rate = 0.0005
gamma         = 0.98
buffer_limit  = 50000
batch_size    = 32
```

```
class ReplayBuffer():
    def __init__(self):
        self.buffer = collections.deque(maxlen=buffer_limit)

    def put(self, transition):
        self.buffer.append(transition)

    def sample(self, n):
        mini_batch = random.sample(self.buffer, n) # random sample
        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst = [], [], [], [], []

        for transition in mini_batch:
            s, a, r, s_prime, done_mask = transition
            s_lst.append(s)
            a_lst.append([a])
            r_lst.append([r])
            s_prime_lst.append(s_prime)
            done_mask_lst.append([done_mask])

        return torch.tensor(s_lst, dtype=torch.float), torch.tensor(a_lst), \
               torch.tensor(r_lst), torch.tensor(s_prime_lst, dtype=torch.float), \
               torch.tensor(done_mask_lst)

    def size(self):
        return len(self.buffer)
```

```python
class Qnet(nn.Module):
    def __init__(self):
        super(Qnet, self).__init__()
        self.fc1 = nn.Linear(4, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def sample_action(self, obs, epsilon):
        out = self.forward(obs)
        coin = random.random()
        if coin < epsilon:
            return random.randint(0,1)
        else :
            return out.argmax().item()
```



Train function

```
def train(q, q_target, memory, optimizer):
    for i in range(10):
        s,a,r,s_prime,done_mask = memory.sample(batch_size)

        q_out = q(s)
        q_a = q_out.gather(1,a)
        max_q_prime = q_target(s_prime).max(1)[0].unsqueeze(1)
        target = r + gamma * max_q_prime * done_mask
        loss = F.smooth_l1_loss(q_a, target)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

```
def main():
    env = gym.make('CartPole-v1')
    q = Qnet()
    q_target = Qnet()
    q_target.load_state_dict(q.state_dict())
    memory = ReplayBuffer()

    print_interval = 20
    score = 0.0
    optimizer = optim.Adam( q.parameters(), lr=learning_rate)

    for n_epi in range(10000):
        epsilon = max(0.01, 0.08 - 0.01*(n_epi/200)) #Linear annealing from 8% to 1%
        s = env.reset()
        done = False

        while not done:
            a = q.sample_action(torch.from_numpy(s).float(), epsilon)
            s_prime, r, done, info = env.step(a)
            done_mask = 0.0 if done else 1.0
            memory.put((s,a,r/100.0,s_prime, done_mask))
            s = s_prime

            score += r
            if done:
                break

        if memory.size()>2000:
            train(q, q_target, memory, optimizer)

        if n_epi%print_interval==0 and n_epi!=0:
            q_target.load_state_dict(q.state_dict())
            print("n_episode :{}, score : {:.1f}, n_buffer : {}, \
                  eps : {:.1f}%".format(n_epi, score/print_interval, memory.size(), epsilon*100))
            score = 0.0
    env.close()
```

Results

```
n_episode :20, score : 18.6, n_buffer : 373, eps : 7.9%
n_episode :40, score : 17.2, n_buffer : 718, eps : 7.8%
n_episode :60, score : 17.9, n_buffer : 1075, eps : 7.7%
n_episode :80, score : 17.9, n_buffer : 1432, eps : 7.6%
n_episode :100, score : 18.9, n_buffer : 1809, eps : 7.5%
n_episode :120, score : 28.6, n_buffer : 2382, eps : 7.4%
n_episode :140, score : 10.3, n_buffer : 2589, eps : 7.3%
n_episode :160, score : 10.3, n_buffer : 2795, eps : 7.2%
n_episode :180, score : 11.6, n_buffer : 3027, eps : 7.1%
n_episode :200, score : 16.6, n_buffer : 3358, eps : 7.0%
n_episode :220, score : 52.2, n_buffer : 4402, eps : 6.9%
n_episode :240, score : 149.3, n_buffer : 7389, eps : 6.8%
n_episode :260, score : 138.2, n_buffer : 10152, eps : 6.7%
n_episode :280, score : 133.9, n_buffer : 12831, eps : 6.6%
n_episode :300, score : 168.9, n_buffer : 16209, eps : 6.5%
```

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [11, 22, 16] and speech recognition [6, 7]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

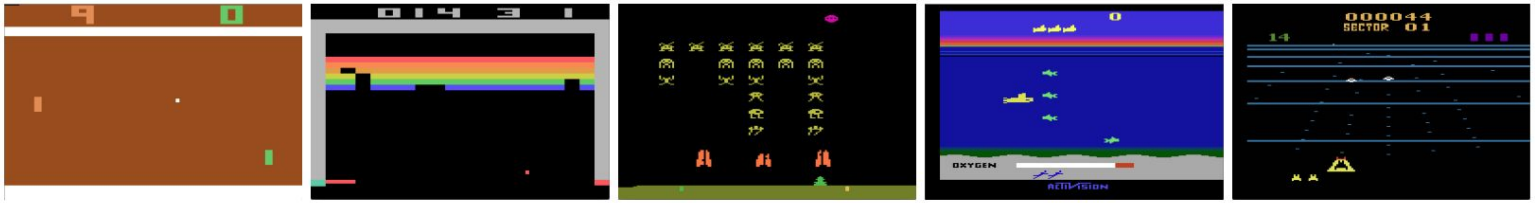


Figure 1: Screen shots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider

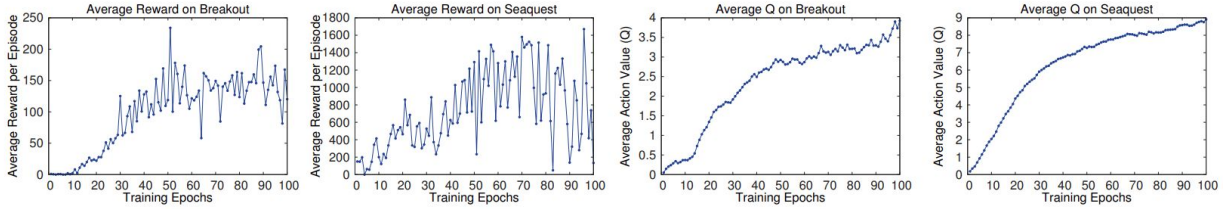


Figure 2: The two plots on the left show average reward per episode on Breakout and Seaquest respectively during training. The statistics were computed by running an ϵ-greedy policy with $\epsilon = 0.05$ for 10000 steps. The two plots on the right show the average maximum predicted action-value of a held out set of states on Breakout and Seaquest respectively. One epoch corresponds to 50000 minibatch weight updates or roughly 30 minutes of training time.

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Policy-based Agent

Value-based Agent vs Policy-based Agent

Value-Based Agent:

When faced with various available actions for a given state, these agents opt for the action 'a' associated with the **highest Q(s, a) value**.

The Q(s, a) value signifies the anticipated reward achievable by selecting action 'a' within state 's'. The agent endeavors to learn the optimal policy by consistently choosing the action with the maximum Q-value.

The choice mechanism of value-based agents is **deterministic**. In other words, once the learning process concludes, the action with the highest Q(s, a) value is always chosen

Value-based Agent vs Policy-based Agent

Policy-Based Agent:

Policy-based agents utilize **policy functions** to make action selections.

A policy delineates a **probability distribution for each action in a given state 's,'** determining the likelihood of taking a specific action in that state.

The selection process of policy-based agents is **stochastic**. Consequently, even when an action is chosen repeatedly within the same state, it may not always be the same action selected.

Q(s, a) vs π(s)

Let's examine a scenario in which the action space is **continuous**.

For example, one can choose any real value **between 0 and 1** as an action.

In value-based agents, the challenge lies in identifying an 'a' that maximizes the Q(s, a) value across the entire range of real values.

Conversely, a policy-based agent faces no such issue, as it can **promptly make decisions** regarding an action when equipped with the **policy function π(s)**.

Policy-based Agent

Our current objective is to focus on enhancing the **policy function**

The problem at hand remains unchanged from our previous context.

It remains a **model-free** problem, and due to the large scale of the environment, which cannot be accommodated in a table, we will employ a **policy network** to represent the policy functions.

Policy Network

Let's redefine the policy network as **$\pi_\theta(s, a)$**, where **θ** represents the network's parameters.

Our objective is to enhance/train the policy $\pi_\theta(s, a)$ as the agent accumulates experience in its environment.

To achieve this, we require a **loss function** to adjust the parameters θ. How can we formulate the loss function for $\pi_\theta(s, a)$?

Objective function

The loss function typically quantifies the difference between a neural network's predicted value and the true value. While we can ascertain the accurate value function through methods like Monte Carlo (MC) or Temporal Difference (TD), **directly determining the true policy function is not possible.**

Instead, we introduce a function to evaluate the policy function as follows:

$$J(\theta)$$

In this expression, the policy π is represented by the parameter θ. The function J(θ) serves as a measure of the **quality of the policy $\pi_\theta(s, a)$**. Consequently, our objective is to enhance J(θ), which signifies the desirability of the policy. To achieve this, we employ **gradient ascent** to adjust the parameter θ.

J(θ)

J(θ) represents the expected value of summation of rewards. We calculate the expected value because, even when the policy π remains constant, the rewards can vary. The definition of J(θ) is as follows:

$$J(\theta) = E_{\pi_\theta}[\sum_t r_t]$$

J(θ)

Certainly, J(θ) is indeed a form of a value function, specifically one that quantifies the expected value of rewards.

If we consider the initial state as s_0, J(θ) can be formulated as:

$$J(\theta) = E_{\pi_\theta}[\sum_t r_t] = v_{\pi_\theta}(s_0)$$

J(θ)

However, it's important to note that episodes do not necessarily commence from s_0 in every instance. Consequently, we need to establish a probability distribution of starting states, denoted as **d(s)**.

$$J(\theta) = \sum_{s \in S} d(s) * v_{\pi_\theta}(s)$$

J(θ)

We now have a function J(θ) capable of assessing $\pi_\theta(s, a)$. Consequently, you have the ability to adjust θ through the process of gradient ascent.

$$\theta \leftarrow \theta + \alpha * \nabla_\theta J(\theta)$$

1-step MDP

Nonetheless, determining $\nabla_\theta J(\theta)$ is a non-trivial task.

To simplify the problem, let's begin by examining how to compute $\nabla_\theta J(\theta)$ in the context of a **1-step MDP**. A 1-step MDP is defined as an instance in which an action **a** is chosen in the initial state $\mathbf{s_0}$, resulting in the receipt of a reward $\mathbf{R_{s,a}}$, after which the episode concludes.

$$J(\theta) = \sum_{s \in S} d(s) * v_{\pi_\theta}(s)$$
$$= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s, a) * R_{s,a}$$

1-step MDP

Now, let's calculate the **gradient**.

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s, a) * R_{s,a}$$

But we cannot calculate it because **we do not know $R_{s,a}$**.

1-step MDP

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a) * R_{s,a}$$
$$= \sum_{s \in S} d(s) \sum_{a \in A} \nabla_\theta \pi_\theta(s,a) * R_{s,a}$$
$$= \sum_{s \in S} d(s) \sum_{a \in A} \frac{\pi_\theta(s,a)}{\pi_\theta(s,a)} \nabla_\theta \pi_\theta(s,a) * R_{s,a}$$
$$= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} * R_{s,a}$$
$$= \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s,a) \nabla_\theta log \pi_\theta(s,a) * R_{s,a}$$

$$\frac{d(\ln x)}{dx} = \frac{1}{x}$$
$$d(\ln x) = \frac{dx}{x}$$

1-step MDP

$$\nabla_\theta J(\theta) = \sum_{s \in S} d(s) \sum_{a \in A} \pi_\theta(s, a) \nabla_\theta log \pi_\theta(s, a) * R_{s,a}$$

The reason for this transformation is to utilize the expectation value.

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s, a) * R_{s,a}]$$

By using the expectation value (E), we can employ a "sampling-based methodology." Since it's an expectation over $\pi_\theta(s, a)$, we can place an agent in the environment following $\pi_\theta(s, a)$, and then collect the values of $\nabla_\theta \log \pi_\theta(s, a) * R_{s,a}$ multiple times.

MDP - Policy Gradient Theorem

1-step MDP: $\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) * R_{s,a}]$

MDP: $\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) * Q_{\pi_\theta}(s,a)]$

In the case of a general MDP, we use the **Q(s, a)** instead of $R_{s,a}$. This concept involves considering rewards accumulated beyond a single step, as the MDP does not terminate immediately after one step but continues for multiple steps into the future.

REINFORCE

REINFORCE is a modification of the Policy Gradient Theorem

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) * G_t]$$

REINFORCE

Initialize policy π with random parameters θ

for each episode do:

Generate a trajectory $\tau = [(s_1, a_1, r_1), (s_2, a_2, r_2), ..., (s_T, a_T, r_T)]$ using π

G = 0 # Initialize the return

for t = T to 1 do:

$G = \gamma * G + r_t$ # Calculate the discounted return

$\theta = \theta + \alpha * \nabla_\theta \log(\pi(a_t|s_t, \theta)) * G$ # Update policy parameters

end for

REINFORCE

Before and after derivative

Gradient

$$G_t * \nabla_\theta log \pi_\theta(s, a)$$

Objective function

$G_t * log \pi_\theta(s, a)$ to maximize

$-G_t * log \pi_\theta(s, a) * G_t$ to minimize

```
class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.data = []

        self.fc1 = nn.Linear(4, 128)
        self.fc2 = nn.Linear(128, 2)
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.softmax(self.fc2(x), dim=0)
        return x

    def put_data(self, item):
        self.data.append(item)

    def train_net(self):
        R = 0
        self.optimizer.zero_grad()
        for r, prob in self.data[::-1]:
            R = r + gamma * R
            loss = -torch.log(prob) * R
            loss.backward()
        self.optimizer.step()
        self.data = []
```

```
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical

#Hyperparameters
learning_rate = 0.0002
gamma         = 0.98
```

```
def main():
    env = gym.make('CartPole-v1')
    pi = Policy()
    score = 0.0
    print_interval = 20

    for n_epi in range(10000):
        s = env.reset()
        done = False

        while not done: # CartPole-v1 forced to terminates at 500 step.
            prob = pi(torch.from_numpy(s).float())
            m = Categorical(prob)
            a = m.sample()
            s_prime, r, done, info = env.step(a.item())
            pi.put_data((r,prob[a]))
            s = s_prime
            score += r

        pi.train_net()

        if n_epi%print_interval==0 and n_epi!=0:
            print("# of episode :{}, avg score : {}".format(n_epi, score/print_interval))
            score = 0.0
    env.close()
```

Actor-Critic

Actor-Critic

AC (Actor-Critic) is a reinforcement learning paradigm that involves learning both a **policy network (actor)** and a **value function network (critic)** simultaneously. There are three common variants of AC, as you mentioned:

1. Q AC (Actor-Critic with Q-value)
2. Advantage AC (A2C or A3C - Advantage Actor-Critic)
3. TD AC (Temporal Difference Actor-Critic)

Q Actor-Critic

Q AC employs the original Policy Gradient formula without any modifications.

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) * Q_{\pi_\theta}(s,a)]$$

Since we cannot directly access the Q function, we need to train a value network Q_w with parameters w. In other words, we will simultaneously reinforce two networks: the **policy network π_θ**, which acts as the **actor** responsible for determining actions a, and the **value network Q_w**, which serves as the **critic** responsible for evaluating the value of actions.

Initialize Policy network parameters θ and Action-value network parameters w

Initialize state s

Sampling action a from $\pi_\theta(a_t, s_t)$

While loop

reward r and next stat s' by taking action a

$\theta = \theta + \alpha * \nabla_\theta \log(\pi_\theta(a_t,s_t)) * Q_w(s, a)$ # Update policy net parameters

Sampling action a' from $\pi_\theta(a_t, s_t)$

$w = w + \beta\ (r+\gamma Q_w(s', a') - Q_w(s, a)) \nabla_w Q_w(s, a)$ # Update action-value net parameters

a = a'

s = s'

Advantage Actor-Critic

The **advantage**, denoted as A(s, a), represents **how much additional value** or benefit is gained **by taking action 'a' in state 's'** compared to the expected value of being in that state, which is represented by v(s).

Mathematically, it can be expressed as:

A(s, a) = Q(s, a) - v(s)

In other words, the advantage tells us whether taking a specific action in a given state is better or worse than the average or expected value associated with that state. Positive advantage indicates that the action is better than average, while negative advantage suggests that the action is worse than average. This concept is commonly used in policy gradient methods and actor-critic algorithms to guide the learning process.

Advantage Actor-Critic

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s,a) * \{Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s)\}]$$

Initialize Policy network parameters θ, Action-value network parameters w, and State-value network parameters ϕ.

Initialize state s

Sampling action a from $\pi_\theta(a_t, s_t)$

While loop

reward r and next state s' by taking action a

$\theta = \theta + \alpha_1 * \nabla_\theta \log(\pi_\theta(a_t, s_t)) * \{Q_w(s, a) - V_\phi(s)\}$ # Update policy net parameters

Sampling action a' from $\pi_\theta(a_t, s_t)$

$w = w + \alpha_2(r+\gamma Q_w(s', a') - Q_w(s, a))\nabla_w Q_w(s, a)$ # Update action-value net parameters

$\phi = \phi + \alpha_3(r+\gamma V_\phi(s') - V_\phi(s))\nabla_\phi V_\phi(s)$ # Update state-value net parameters

a = a'

s = s'

TD Actor-Critic

Advantage Actor-Critic (Advantage AC) typically involves three networks, while TD Actor-Critic (TD AC) manages to achieve similar goals with just two networks. The TD error (delta) and Policy Gradient components are as follows:

$$\delta = r + \gamma V(s') - V(s)$$

$$\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta log \pi_\theta(s, a) * \delta]$$

Initialize Policy network parameters θ and State-value network parameters ϕ.

Initialize state s

Sampling action a from $\pi_\theta(a_t, s_t)$

While loop

reward r and next state s′ by taking action a

$\delta = r+\gamma V_\phi(s') - V_\phi(s)$

$\theta = \theta + \alpha_1 * \nabla_\theta \log(\pi_\theta(a_t,s_t)) * \delta$ # Update policy net parameters

$\phi = \phi + \alpha_2 \delta \nabla_\phi V_\phi(s)$ # Update state-value net parameters

a = a′

s = s′

```
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import
Categorical

#Hyperparameters
learning_rate = 0.0002
gamma         = 0.98
n_rollout     = 10
```

```
class ActorCritic(nn.Module):
    def __init__(self):
        super(ActorCritic, self).__init__()
        self.data = []

        self.fc1 = nn.Linear(4,256)
        self.fc_pi = nn.Linear(256,2)
        self.fc_v = nn.Linear(256,1)
        self.optimizer = optim.Adam(self.parameters(), lr=learning_rate)

    def pi(self, x, softmax_dim = 0):
        x = F.relu(self.fc1(x))
        x = self.fc_pi(x)
        prob = F.softmax(x, dim=softmax_dim)
        return prob

    def v(self, x):
        x = F.relu(self.fc1(x))
        v = self.fc_v(x)
        return v

    def put_data(self, transition):
        self.data.append(transition)
```

```
    def make_batch(self):
        s_lst, a_lst, r_lst, s_prime_lst, done_lst = [], [], [], [], []
        for transition in self.data:
            s,a,r,s_prime,done = transition
            s_lst.append(s)
            a_lst.append([a])
            r_lst.append([r/100.0])
            s_prime_lst.append(s_prime)
            done_mask = 0.0 if done else 1.0
            done_lst.append([done_mask])

        s_batch, a_batch, r_batch, s_prime_batch, done_batch = torch.tensor(s_lst, dtype=torch.float), torch.tensor(a_lst), \
                                                               torch.tensor(r_lst, dtype=torch.float), torch.tensor(s_prime_lst,
dtype=torch.float), \
                                                               torch.tensor(done_lst, dtype=torch.float)
        self.data = []
        return s_batch, a_batch, r_batch, s_prime_batch, done_batch

    def train_net(self):
        s, a, r, s_prime, done = self.make_batch()
        td_target = r + gamma * self.v(s_prime) * done
        delta = td_target - self.v(s)

        pi = self.pi(s, softmax_dim=1)
        pi_a = pi.gather(1,a)
        loss = -torch.log(pi_a) * delta.detach() + F.smooth_l1_loss(self.v(s), td_target.detach())

        self.optimizer.zero_grad()
        loss.mean().backward()
        self.optimizer.step()
```

```
def main():
    env = gym.make('CartPole-v1')
    model = ActorCritic()
    print_interval = 20
    score = 0.0

    for n_epi in range(10000):
        done = False
        s = env.reset()
        while not done:
            for t in range(n_rollout):
                prob = model.pi(torch.from_numpy(s).float())
                m = Categorical(prob)
                a = m.sample().item()
                s_prime, r, done, info = env.step(a)
                model.put_data((s,a,r,s_prime,done))

                s = s_prime
                score += r

                if done:
                    break

            model.train_net()

        if n_epi%print_interval==0 and n_epi!=0:
            print("# of episode :{}, avg score : {:.1f}".format(n_epi, score/print_interval))
            score = 0.0
    env.close()

if __name__ == '__main__':
    main()
```

More topics

PPO

DDPG

World Model

Dreamer3

And Homework 3

New Research Topics in MI@UTRGV

Durg Design

ADCNet: a unified framework for predicting the activity of antibody-drug conjugates

https://arxiv.org/abs/2401.09176

Deep reinforcement learning for de novo drug design

https://www.science.org/doi/10.1126/sciadv.aap7885

Plan to get a grant soon

CRL and Deepmimic

Causal Reinforcement Learning - Sponsored by Google ($100K)

https://crl.causalai.net/

https://arxiv.org/abs/2302.05209

DeepMimic: Example-Guided Deep Reinforcement Learning of Physics-Based Character Skills

https://xbpeng.github.io/projects/DeepMimic/index.html

HEA

Deep learning-based phase prediction of high-entropy alloys: Optimization, generation, and explanation

https://www.sciencedirect.com/science/article/pii/S0264127520307954

Sponsored by DoD ($4M)