Artificial Intelligence for Medical Data with Python

10 SAMPLE SLIDES

7th session –
Reinforcement Learning and Deep Neural Nets

UNIVERSITY OF THE AEGEAN

SCHOOL OF ENGINEERING

DEPARTMENT OF INFORMATION AND COMMUNICATION SYSTEMS ENGINEERING

Presenter: Panagiotis Symeonidis

Associate Professor

http://panagiotissymeonidis.com

psymeon@aegean.gr

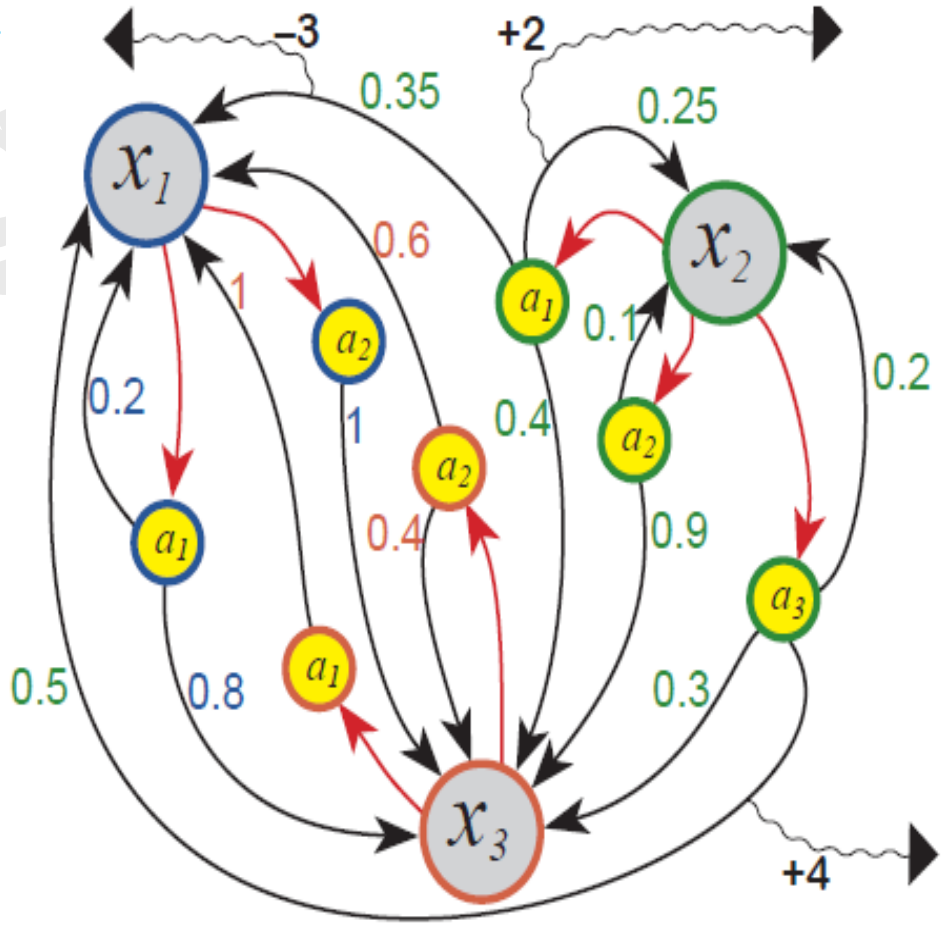
Markov Decision Processes

- The transition probability to the new state x_{t+1} depends only on the current state x_t and the selected action α.

$$P\left(X_{t+1} \mid X_t, a\right)$$

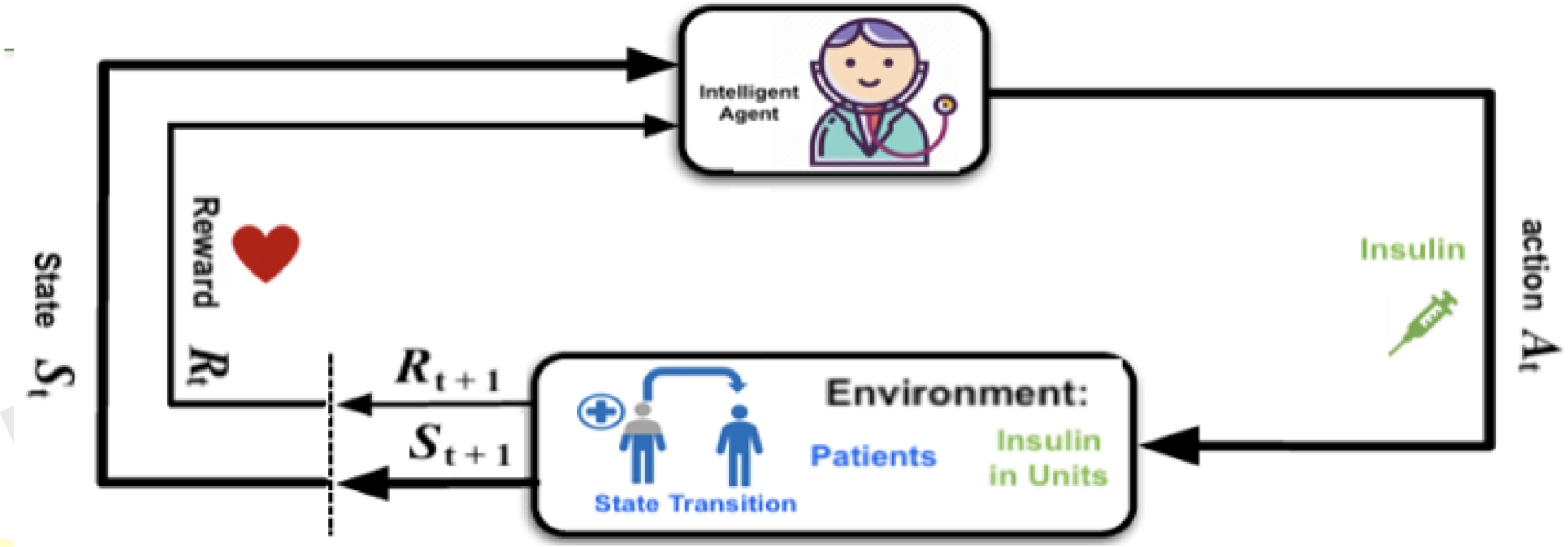
- Each action α provides a reward R, and the goal is to maximize the reward.

$$\max_a \sum_{t=0}^{\infty} R(x_t, a, x_{t+1})$$



Reinforcement Learning

Trains a learning model through the interaction of an agent (intelligent software) with the environment of patients and drugs



Algorithm Deep Q-Network (DQN)

- The Bellman equation is central to Q-learning, describing the relationship between current and future rewards.

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s',a') \middle| s,a \right]$$

- In DQN, the Bellman equation is used to iteratively update the Q-network, driving it towards optimal action-value functions.
- A Q-network can be trained by minimizing a sequence of loss functions Li(θi) that changes at each iteration i:

$$L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right]$$

Toy Example

We are given the patient-drug interactions matrix, where U_0–U_5 are patients, and I_0–I_5 are the drugs they have interacted with along with the number of these interactions.

	I_0	I_1	I_2	I_3	I_4	I_5
U_0	5	1	1	4	0	0
U_1	1	5	4	0	0	4
U_2	2	1	5	0	0	4
U_3	1	2	1	5	0	0
U_4	5	1	2	0	1	0
U_5	1	5	4	0	4	0

Exercise: Implement the Deep Q-Network (DQN) with experience replay algorithm with Python code and tensorflow.

Environment Setup: Import libraries and standardize data

```
import os
import numpy as np
import random
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras import backend as K
from collections import deque
import pandas as pd
import matplotlib.pyplot as plt

# Clear TensorFlow session
K.clear_session()
# Adjust pandas display settings
pd.set_option('display.max_columns', None)  # Show all columns
pd.set_option('display.max_colwidth', 20)  # Set maximum column width
SEED_NUMBER = 42

os.environ['CUDA_VISIBLE_DEVICES'] = "-1"
os.environ['TF_CUDNN_USE_AUTOTUNE'] = "0"
os.environ['OMP_NUM_THREADS'] = "1"
os.environ['PYTHONHASHSEED'] = str(SEED_NUMBER)
os.environ['TF_DETERMINISTIC_OPS'] = "1"
os.environ['TF_CUDNN_DETERMINISTIC'] = "1"

random.seed(SEED_NUMBER)
np.random.seed(SEED_NUMBER)
tf.random.set_seed(SEED_NUMBER)

config = tf.compat.v1.ConfigProto(intra_op_parallelism_threads=1,
inter_op_parallelism_threads=4, allow_soft_placement=True,
device_count={'CPU': 5})
sess = tf.compat.v1.Session(graph=tf.compat.v1.get_default_graph(),
config=config)
```

```
# Data
items = ['I_0', 'I_1', 'I_2', 'I_3', 'I_4', 'I_5']
data = [{items[0]: 5, items[1]: 1, items[2]: 1, items[3]: 4, items[4]: 0, items[5]: 0},
        {items[0]: 1, items[1]: 5, items[2]: 4, items[3]: 0, items[4]: 0, items[5]: 4},
        {items[0]: 2, items[1]: 1, items[2]: 5, items[3]: 0, items[4]: 0, items[5]: 4},
        {items[0]: 1, items[1]: 2, items[2]: 1, items[3]: 5, items[4]: 0, items[5]: 0},
        {items[0]: 5, items[1]: 1, items[2]: 2, items[3]: 0, items[4]: 1, items[5]: 0},
        {items[0]: 1, items[1]: 5, items[2]: 4, items[3]: 0, items[4]: 4, items[5]: 0}]

users = ['U_0', 'U_1', 'U_2', 'U_3', 'U_4', 'U_5']

# Create the DataFrame directly with the index parameter
df = pd.DataFrame(data, index=users)
for col in df.columns:
    df[col] = df[col].astype(float)

print("Data")
display(df)

state_matrix = df.to_numpy()
```

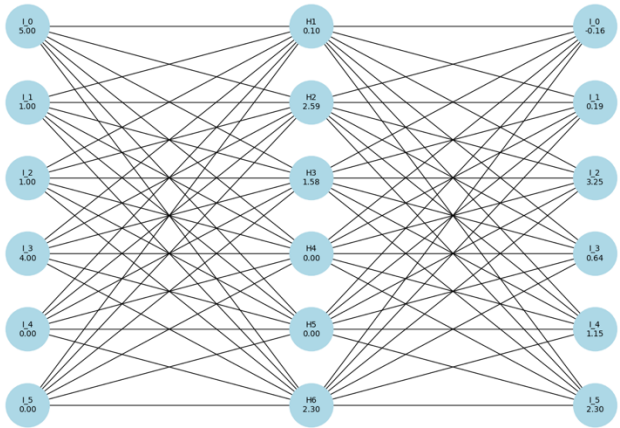
	I_0	I_1	I_2	I_3	I_4	I_5
U_0	5.0	1.0	1.0	4.0	0.0	0.0
U_1	1.0	5.0	4.0	0.0	0.0	4.0
U_2	2.0	1.0	5.0	0.0	0.0	4.0
U_3	1.0	2.0	1.0	5.0	0.0	0.0
U_4	5.0	1.0	2.0	0.0	1.0	0.0
U_5	1.0	5.0	4.0	0.0	4.0	0.0

DQN initialization

```
1.  # Define hyperparameters
2.  EPISODES = 500
3.  GAMMA = 0.75
4.  LEARNING_RATE = 0.001
5.  BATCH_SIZE = 64
6.  MEMORY_SIZE = 10000
7.  TARGET_UPDATE = 10  # Frequency to update the target network


8.  # Define the Q-Network
9.  # The Q-Network approximates the Q-function using a neural network,
    allowing us to handle large state spaces.
10. def build_q_network(input_dim, output_dim, learning_rate,
    hidden_size=6):
11.     # Input layer to handle the state representation as input.
12.     inputs = tf.keras.Input(shape=(input_dim,))
13.     # Hidden layer to learn features from the input state, using ReLU
    activation to introduce non-linearity.
14.     x = tf.keras.layers.Dense(hidden_size, activation='relu')(inputs)
15.     # Output layer to produce Q-values for each possible action,
    representing the expected future rewards.
16.     outputs = tf.keras.layers.Dense(output_dim)(x)
17.     # Create the sequential model
18.     model = tf.keras.Model(inputs=inputs, outputs=outputs)
19.     # The network is trained using Mean Squared Error loss and Adam
    optimizer, aligning with the Q-learning update rule.
20.     model.compile(optimizer=optimizers.Adam(learning_rate=learning_rate
    ), loss='mse')
21.     return model
```

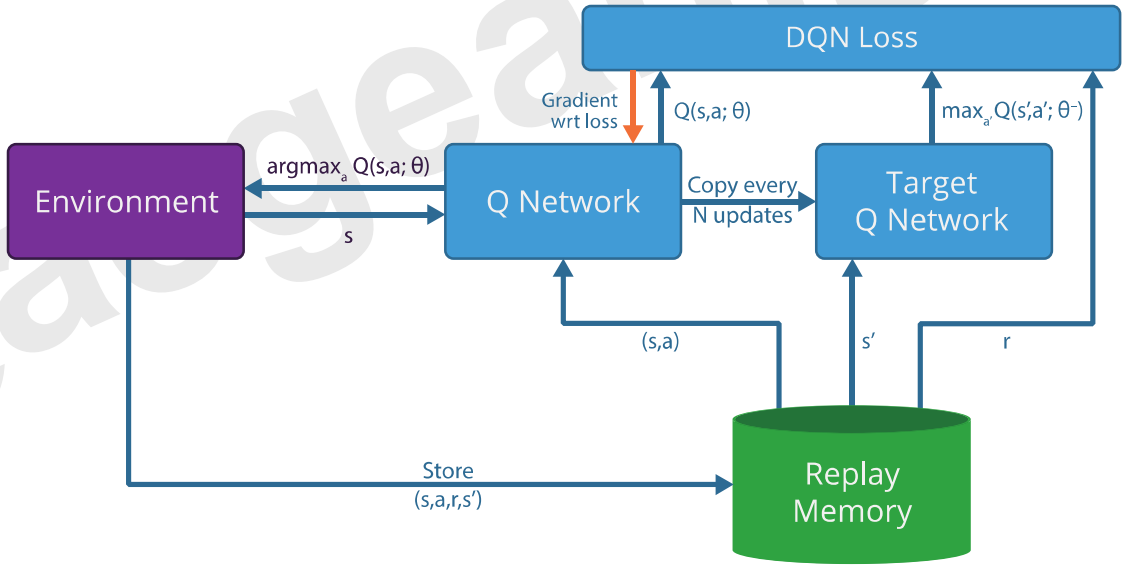
- In lines #2 to #7 we set up some initial parameter for the network
- In lines #10 to # 21 we define the function `**build_q_network**` , which initializes a neural network that approximates the Q-values for each action given a state. It creates a sequential model:
  - The input nodes represent the state of the environment, which could include features like user-item interaction histories in a recommender system. It has **input_dim** nodes; in our example it equals to the number of users: 6.
  - The hidden layer (**Dense**) which has 6 nodes in this example; it can be increased for more accuracy.
  - The output nodes represent the Q-values corresponding to each possible action (e.g., the different items that can be recommended). It has **output_dim** nodes, which in our example equals the number of items: 6 per user.



DQNAgent and select_action function

```
1.  # Agent class
2.  # The DQNAgent class encapsulates the DQN algorithm, managing the Q-Networks, epsilon-greedy
    policy, and training process.
3.  class DQNAgent:
4.      def __init__(self, state_dim, action_dim, learning_rate=0.001, gamma=0.75,
    memory_size=10000):
5.          self.state_dim = state_dim
6.          self.action_dim = action_dim
7.          self.learning_rate = learning_rate
8.          self.gamma = gamma
9.          # Initialize the primary Q-Network to predict action values based on the current state.
10.         self.q_network = build_q_network(state_dim, action_dim, learning_rate)
11.         # Initialize the target Q-Network to provide stable targets for the primary network's
    training.
12.         self.target_network = build_q_network(state_dim, action_dim, learning_rate)
13.         # Initialize the replay memory buffer to store past experiences.
14.         self.replay_memory = ReplayMemoryBuffer(memory_size)
15.         # Set initial epsilon for the epsilon-greedy policy, promoting exploration at the start
    of training.
16.         self.epsilon = 1.0  # Initial exploration rate
17.         # Define the rate at which epsilon decreases, gradually shifting from exploration to
    exploitation.
18.         self.epsilon_decay = 0.995
19.         # Set a minimum value for epsilon to ensure some exploration continues throughout
    training.
20.         self.epsilon_min = 0.01
21.
        # Select an action based on the epsilon-greedy policy.
22.     def select_action(self, state):
23.         # With probability epsilon, select a random action (exploration).
24.         if random.random() > self.epsilon:
25.             state = np.expand_dims(state, axis=0)
26.             # Predict Q-values for the current state and choose the action with the highest
    value (exploitation).
27.             q_values = self.q_network.predict(state, batch_size=1)
28.             action = np.argmax(q_values[0])
29.         else:
30.             action = random.choice(range(self.action_dim))
31.         # Decrease epsilon after each action, reducing the exploration rate over time.
32.         self.epsilon = max(self.epsilon * self.epsilon_decay, self.epsilon_min)
33.         return action
```

The **DGNAgent** class manages two Q-networks (primary and target): the use of two networks (one for selecting actions and another for computing target Q-values) is a key feature of DQN that helps stabilize training



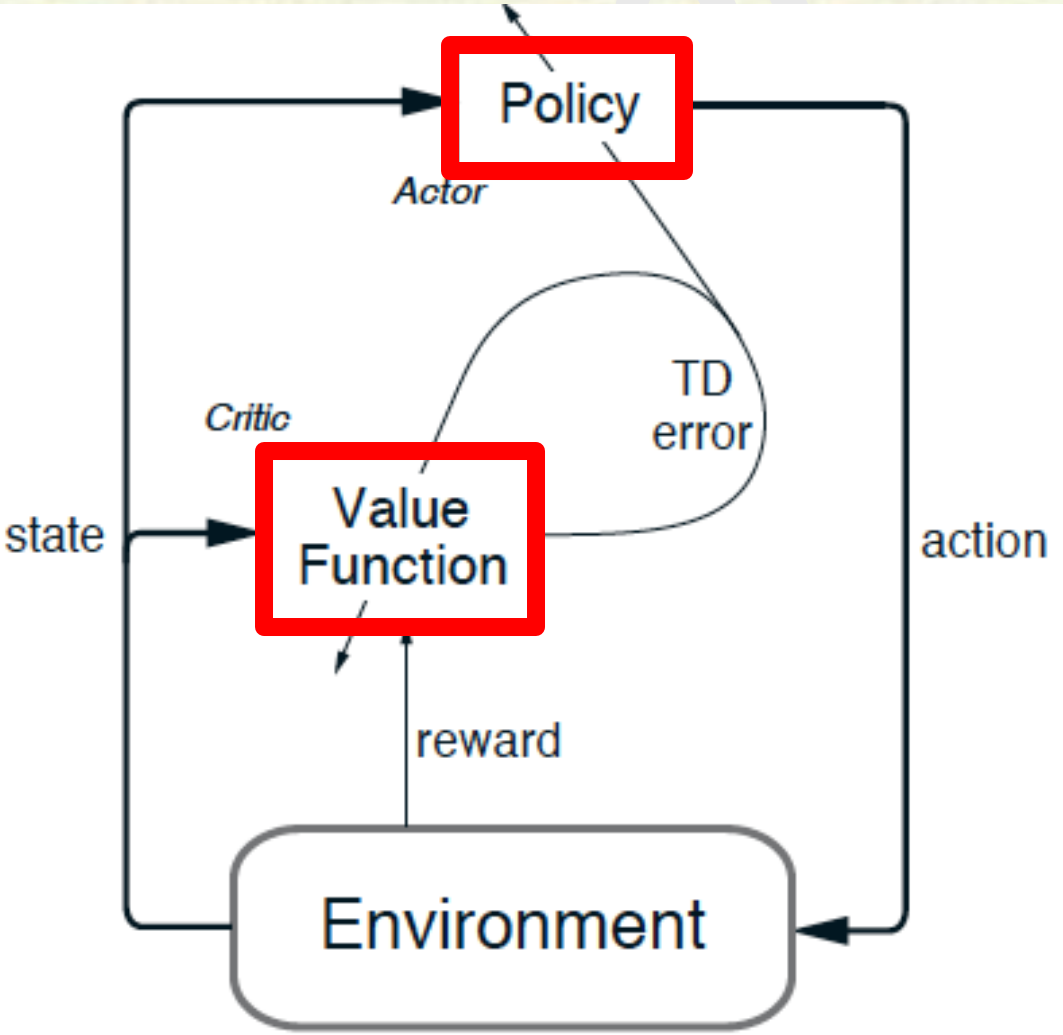
select_action function: the epsilon-greedy policy balances exploration and exploitation, gradually shifting from random exploration to exploiting the learned knowledge.
The epsilon value decays over time (code line #33), gradually shifting the agent from exploration to exploitation.

Advantage Actor Critic (A2C)

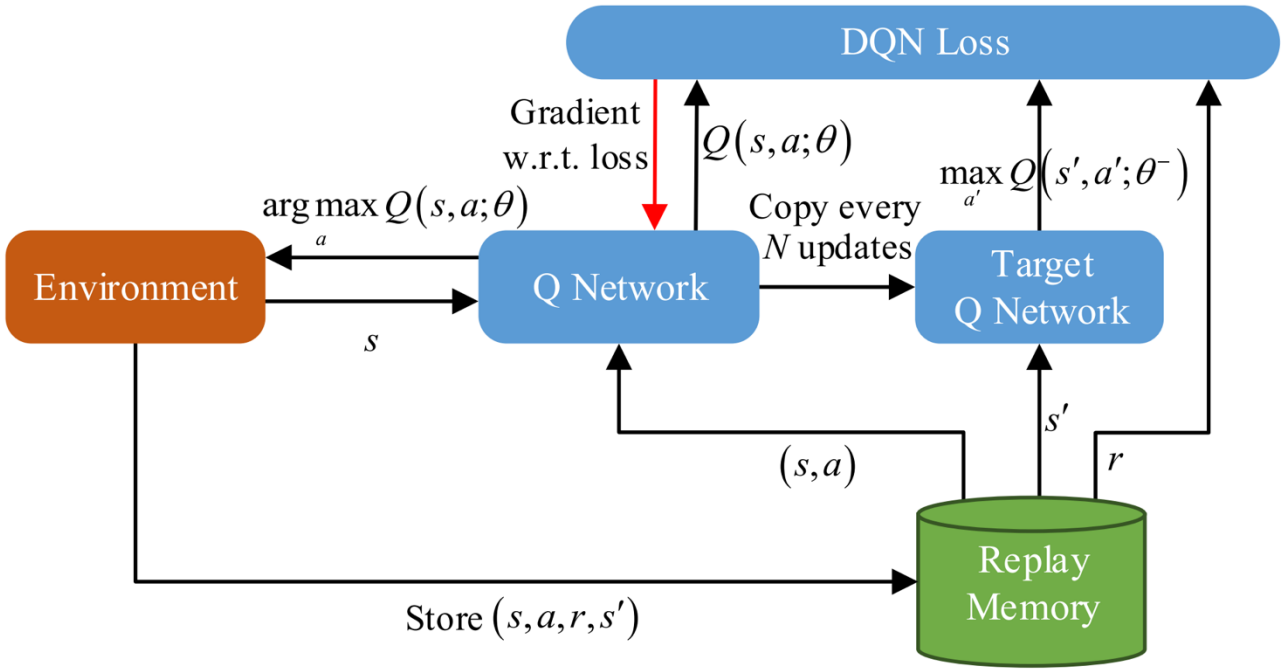
A2C consists of 2 neural nets:

1. gradient policy π function (actor),
2. the state value function (critic)

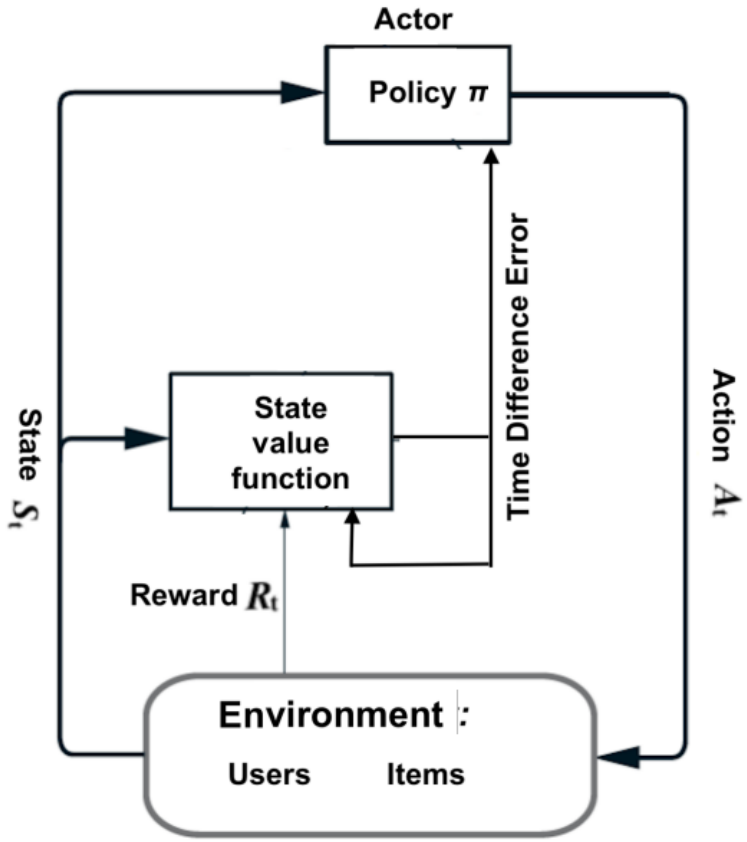
A2C combines the advantages of both off-policy RL methods, which are value-based, and on-policy methods, which directly optimize the current policy.



DQN with Replay versus Advantage Actor-Critic (A2C) algorithm



DQN with Replay



A2C

https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html