# University of Tehran

**College of Engineering**

**Electrical and Computer Engineering Department**


**Internship Report**

# Minimum Throughput Maximization for Single-UAV Enabled WPCN: A Deep Reinforcement Learning Method


**Communication Department**

**Alireza Vosoughi Rad**

**810196583**

**Professor: Dr. Maryam Sabaghiyan**


**August - September 2020**

# Contents

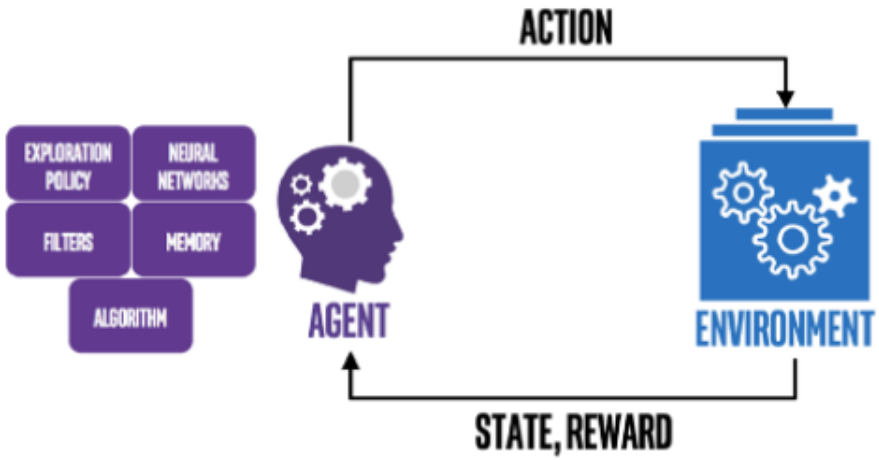
# Reinforcement Learning:

Reinforcement learning is an area of Machine Learning that deals with sequential decision-making.

Typically, a RL setup is composed of two components, an agent and an environment.

Environment refers to the object that the agent is acting on, while the agent represents the RL algorithm. The environment starts by sending a state to the agent, which then based on its knowledge to take an action in response to that state. After that, the environment send a pair of next state and reward back to the agent. The agent will update its knowledge with the reward returned by the environment to evaluate its last action. The loop keeps going on until the environment sends a terminal state, which ends to episode.

Thus, the RL agent does not require complete knowledge or control of the environment; it only needs to be able to interact with the environment and collect information. In an offline setting, the experience is acquired a priori, then it is used as a batch for learning (hence the offline setting is also called batch RL). This is in contrast to the online setting where data becomes available in a sequential order and is used to progressively update the behavior of the agent. In both cases, the core learning algorithms are essentially the same but the main difference is that in an online setting, the agent can influence how it gathers experience so that it is the most useful for learning.




There are three basic concepts in reinforcement learning: state, action, and reward.

The state describes the current situation. For a robot that is learning to walk, the state is the position of its two legs, or for maximization throughput for multi-UAV enabled WPCN [] state is position of UAV in 2-D plane.

Action is all the possible moves that the agent can take in each state. Given the state, or positions of UAV, a UAV can take steps within a certain distance. There are typically finite (or a fixed range of) actions an agent can take.

The term “reward” is an abstract concept that describes feedback from the environment. A reward can be positive or negative. When the reward is positive, it is corresponding to our normal meaning of reward. When the reward is negative, it is corresponding to what we usually call “punishment.”

RL is about taking suitable action to maximize reward in a particular situation. It is employed by various software and machines to find the best possible behavior or path it should take in a specific situation. Reinforcement learning differs from the supervised learning in a way that in supervised learning the training data has the answer key with it so the model is trained with the correct answer itself whereas in reinforcement learning, there is no answer but the reinforcement agent decides what to do to perform the given task. In the absence of a training dataset, it is bound to learn from its experience.

## Difference between Reinforcement learning Supervised learning and Unsupervised learning:

Reinforcement learning is all about making decisions sequentially. In simple words we can say that the output depends on the state of the current input and the next input depends on the output of the previous input.
In reinforcement learning the goal is to find a suitable action model that would maximize the total cumulative reward of the agent.

In Supervised learning the decision is made on the initial input or the input given at the start.

Supervised learning the decisions are independent of each other so labels are given to each decision.

Goal in unsupervised learning is to find similarities and differences between data points and make them ready for analyzing.

## The reinforcement learning setting:

The general RL problem is formalized as a discrete time stochastic control process where an agent interacts with its environment in the following way: the agent starts, in a given state within its environment $s_0 \in S$, by gathering an initial observation $\omega_0 \in \Omega$. At each time step t, the agent has to take an action $a \in A$.

## The Markov property:

The Markov property means that the future of the process only depends on the current observation, and the agent has no interest in looking at the full history. A Markov Decision Process (MDP) (Bellman, 1957a) is a discrete time stochastic control process defined as follows:

An MDP is a 5-tuple $(S, A, T, R, \gamma)$ where:

• $S$ is the state space,

• $A$ is the action space,

• $T : S \times A \times S \rightarrow [0, 1]$ is the transition function (set of conditional transition probabilities between states),

• $R : S \times A \times S \rightarrow R$ is the reward function, where $R$ is a continuous set of possible rewards in a range $R_{max} \in R^+$ (e.g. $[0, R_{max}]$),

• $\gamma \in [0, 1)$ is the discount factor. The system is fully observable in an MDP, which means that the observation is the same as the state of the environment: $\omega_t = s_t$. At each time step t, the probability of moving to $s_{t+1}$ is given by the state transition function $T(s_t, a_t, s_{t+1})$ and the reward is given by a bounded reward function $R(s_t, a_t, s_{t+1}) \in R$.

Deep Reinforcement Learning is the combination of Reinforcement Learning and Deep Learning and it is also the most trending type of Machine Learning at this moment because it is being able to solve a wide range of complex decision-making tasks that were previously out of reach for a machine to solve real-world problems with human-like intelligence.

# Q-Learning:

Q-learning is the most commonly used reinforcement learning method, where Q stands for the long-term value of an action. Q-learning is about learning Q-values through observations.

The procedure for Q-learning is:

In the beginning, the agent initializes Q-values to 0 for every state-action pair. More precisely, $Q(s, a) = 0$ for all states *s* and actions $a$. This is essentially saying we have no information on long-term reward for each state-action pair.

After the agent starts learning, it takes an action a in state *s* and receives reward *r*. It also observes that the state has changed to a new state s'. The agent will update $Q(s, a)$ with this formula:

$$\mathrm{Q(s, a)} = (1 - \mathrm{learning\_rate})\mathrm{Q(s, a)} + \mathrm{learning\_rate}\ (\mathrm{r} + \mathrm{discount\_rate}\ \mathrm{max\_a}\ \mathrm{Q(s', a)})$$

The learning rate is a number between 0 and 1. It is a weight given to the new information versus the old information. The new long-term reward is the current reward, *r*, plus all future rewards in the next state, *s'*, and later states, assuming this agent always takes its best actions in the future. The future rewards are discounted by a discount rate between 0 and 1, meaning future rewards are not as valuable as the reward now.

In this updating method, Q carries memory from the past and takes into account all future steps. Note that we use the maximized Q-value for the new state, assuming we always follow the optimal path afterward. As the agent visits all the states and tries different actions, it eventually learns the optimal Q-values for all possible state-action pairs. Then it can derive the action in every state that is optimal for the long term.

Q-learning requires the agent try many times to visit all possible state-action pairs. Only then does the agent have a complete picture of the world. Q-values represent the optimal values when taking the best sequence of actions. This sequence of actions is also called "policy."

A fundamental question we face is: is it possible for an agent to learn all the Q-values when it explores the actions possible in a given environment? In other words, is such learning feasible? The answer is yes if we assume the world responds to actions. In other words, the state changes based on an action. This assumption is called the Markov Decision Process (MDP). It assumes that the next state depends on the previous state and the action taken. Based on this assumption, all possible states can eventually be visited, and the long-term value (Q-value) of every state-action pair can be determined.

Imagine that we live in a random universe where our actions have no impact on what happens next, then reinforcement learning (Q-learning) would break down. After many times of trying, we'd have to throw in the towel. Fortunately, our universe is much more predictable. When a Go player puts down a piece on the board, the board position is clear in the next round. Our agent interacts with the environment and shapes it through its actions. The exact impact of our agent's action on the state is typically straightforward. The new state is immediately observable. The robot can tell where it ends up.

The essential technique of reinforcement learning is exploration versus exploitation. An agent learns about the value of Q(s,a) in state *s* for every action *a*. Since the agent needs to get a high reward, it can choose the action that leads to the highest reward based on current information (exploitation), or keep trying new actions, hoping it brings even higher reward (exploration). When an agent is learning online (in real time), the balance of these two strategies is very important. That's because learning in real time means the agent has to maintain its own survival (when exploring a cave or fighting in a combat) versus finding the best move. It is less of a concern when an agent is learning offline (meaning not in real time). In machine learning terms, offline learning means an agent processes information without interacting with the world. In such cases, the price to pay for failing (like hitting a wall, or being defeated in a game) is little when it can experiment with (explore) many different actions without worrying about the consequences.

The performance of Q-learning depends on visiting all state-action pairs in order to learn the correct Q-values. This can be easily achieved with a small number of states. In the real world, however, the number of states can be very large, particularly when there are multiple agents in the system. For example, in a maze game, a robot has at most 1,000 states (locations); this grows to 1,000,000 when it is in a game against another robot, where the state represents the joint location of two robots (1,000 x 1,000).

When the state space is large, it is not efficient to wait until we visit all state-actions pairs. A faster way to learn is called the Monte Carlo method. In statistics, the Monte Carlo method derives an average through repeated sampling. In reinforcement learning, the Monte Carlo method is used to derive Q-values after repeatedly seeing the same state-action pair. It sets the Q-value, Q(s,a), as the average reward after many visits to the same state-action pair (s, a). This method removes the need for using a learning rate or a discount rate. It depends only on large

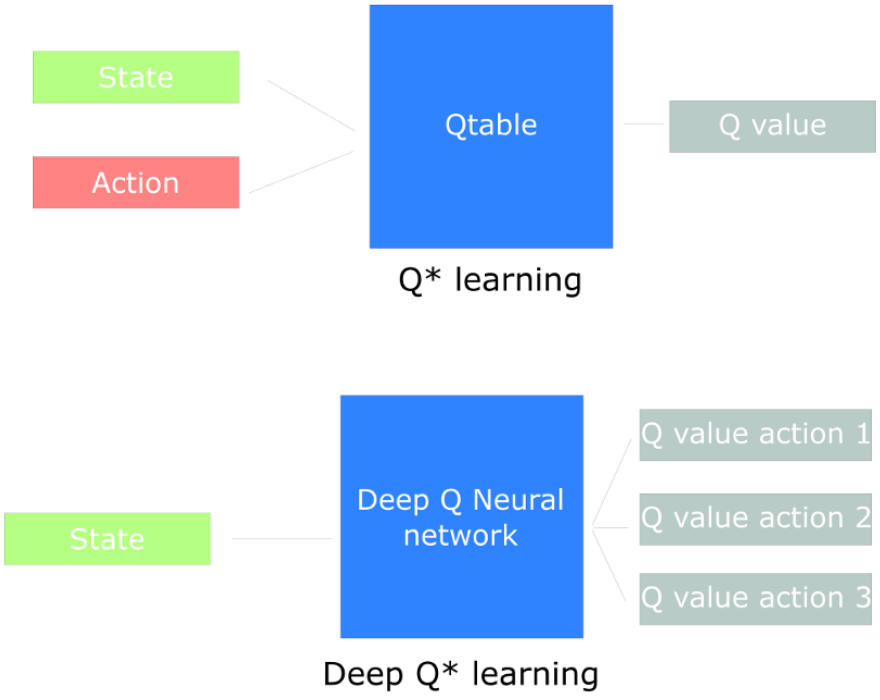
numbers of simulations. Due to its simplicity, this method has become very popular. It has been used by AlphaGo after playing many games against itself to learn about the best moves.

Another way to reduce the number of states is by using a neural network, where the inputs are states and outputs are actions, or Q-values associated with each action. A deep neural network has the power to dramatically simplify the representation of states through hidden layers. In this Nature paper on deep reinforcement learning used with Atari games, the whole game board is mapped by a convolutional neural network to determine Q-values.

## Deep Q-Learning:

As mentioned above Q-Learning is an algorithm which produces a Q-table that an agent uses to find the best action to take given a state.

But as we'll see, producing and updating a Q-table can become ineffective in big state space environments. In this case instead of using a Q-table, we'll implement a Neural Network that takes a state and approximates Q-values for each action based on that state.



# Code Explanation

## Simulation

All hyperparameters are defined by argument parser packages and all constants were stored in a variable. The whole process is performed in parallel on GPU. in order to break the process to parallel threads, CUDA packages from tensorflow were used.
“Designer” Class were defined to do the training and testing process. This class contains two methods. The first one is the constructor and the second one is “simulate“ method. When we execute “main.py”, “simulate” method starts to run. This method runs over for loop with episode number which we defined with argument parser. There is also an inner for loop which has all steps that an UAV agent does in a T period. In the inner loop, we select an action according to the current state and then by “step” method of “environment” class, reward and new state are calculated. afterward the whole situation will be observed and saved in memory to training. neural network do training process in each N observation.

## Uniform Experience Memory

It is a store of K number of transitions to be sampled from later for the agent to learn from.
The reason for it existing is that if you were to sample transitions from the online RL agent as they are being experienced, there would be a strong temporal/chronological relationship between them.
Stochastic gradient descent also works best with i.i.d (identically and independently distributed) samples where linearly sampled online experiences aren’t i.i.d.
So this is the type of experience replay where k experiences are stored in memory and sampled from a uniform distribution.
This breaks the chronological relationship between experiences being sampled from but it results in useful experiences with high temporal difference errors being wasted as new appended experiences remove old ones from the list as the memory capacity is limited.

## Prioritized Experience Memory

authors in [1] recommend a methodology that assigns higher probability to useful samples and sampling is not uniform anymore.
Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In prior work, experience transitions were uniformly sampled from a replay memory. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance.
a framework for prioritizing experience was developed, so as to replay important transitions more frequently, and therefore learn more efficiently.

# UAV Agent

UAV agent is a part of the environment which has a brain(neural network) and learns to do a goal in T time steps. The problem's goal is to maximize the minimum throughput of all devices.

agent uses a greedy epsilon in order to explore in state space and discover new states in initial steps. we choose a random value between zero and one then we compare it with the epsilon value. if it was greater than epsilon then we take an action according to the output of the neural network. but if it is smaller than epsilon then we choose a random action. This helps the agent to explore the environment and find ways to achieve positive rewards and learn the environment. in the beginning epsilon value is 1 so in initial steps most of the moves are randomly and agent is in exploration mode. In each step we decrease epsilon value and after a while, epsilon value will be near zero. So after that we took a lot of steps and neural network parameters were trained, we will not take actions randomly and most of the actions will be based on the learned network.

"find target" method calculates actions probability according to the reward value and then returns the current state and actions value for training process. actions probability is calculated by new state prediction with online and target models.
There are two ways to calculate actions probability: Double DQN(DDQN) and regular DQN. in the following we will explain Double DQN networks.
in regular DQN networks action probability is calculated by following formula[2]:

$$Q(s,a) = r(s,a) + \gamma \max_{a} Q(s',a)$$

# Double Neural Network

authors in [3] recommend solution for overestimation problems. The popular Q-learning algorithm is known to overestimate action values under certain conditions. It was not previously known whether, in practice, such overestimations are common, whether they harm performance, and whether they can generally be prevented. In this paper, an answer was given to all these questions affirmatively.
new action probability will be:

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q\left(S_{t+1}, \operatorname*{argmax}_a Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-\right)$$

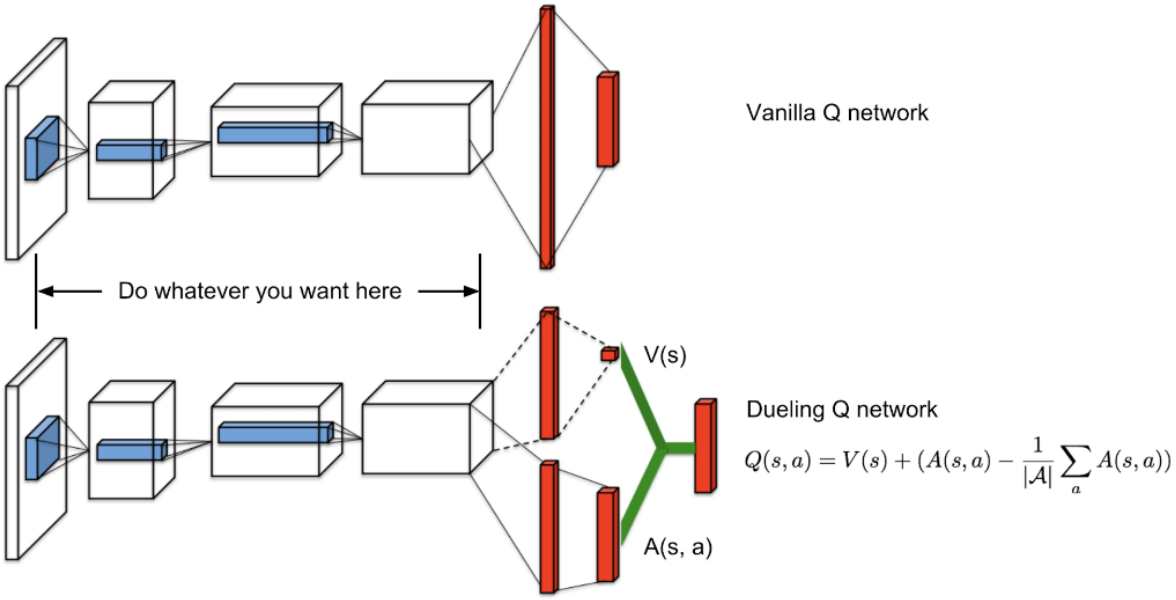
# Neural Network

Neural network is brain of agent. in "Brain" class we created a neural network to estimate the new action of the agent. This network was implemented in two ways: the regular one and the Dueling method. tensorflow and keras packages were used for implementation of the network.
in regular method, the network gets state space as input and calculates action space probability. "build model" method of Brain class creates a model. There are an online and a target model in the deep Q networks. online model update in each iteration but target model equals to online model in each N iterations.
a regular network consists of two hidden layers which each layer has K nodes. K can be determined by users and its default value is 256.

Also we can determine the optimizer. we can use “Adam”, “RMSProp”, SGD”, etc. default optimizer is “Adam”.
In the following we will discuss the Dueling method and the reason we use Dueling.

## Dueling Neural Network

Dueling neural network [4] architecture was designed for model-free reinforcement learning. Dueling network represents two separate estimators: one for the state value function and one for the state-dependent action advantage function. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm. results show that this architecture leads to better policy evaluation in the presence of many similar-valued actions.




## Environment

Previous parts were common between all single agent neural networks. Each problem has its own environment and its own reward policies. state space, action space, reward calculation, and position update are done in the environment.
“Environment” class was defined to achieve this goal. Every environment must have a “reset” method in order to initialize all parameters and values in the first of each episode. “reset” method must return the initial state. Of course, we also return positions of IOT devices.
Another important method is “step”. This method gets the UAV agent’s action and returns a new state and reward value. all policies related to calculation of reward value should be implemented in “step” method. reward policies in UAV and IOT device problem is:
.
.
There is another method for Environment class which calculates the throughput. We define continuous channel capacity as throughput. channel capacity is:
R = B log2(1+SINR)
B is bandwidth and SINR is signal to noise plus interference ratio. In this problem we assume that there is no interference, so we can consider SNR as SINR. ultimate fromulate of throughput will be[5]:

$$R_{k_l}[n] = B_{k_l} \log_2 \left( 1 + \frac{P_{k_l}^U[n] g_{l,k_l}[n]}{I_{k_l}[n] + \sigma^2} \right)$$

P is power and g define as:

$g_{l,k_l}[n] = [P_{LoS}\mu_1 + P_{NLoS}\mu_2]^{-1}(K_0 d_{l,k_l}[n])^{-\alpha}$

all these parameter's numeric value is on [5].

# References


[1] https://arxiv.org/abs/1511.05952
[2] https://www.nature.com/articles/nature14236
[3] https://arxiv.org/abs/1509.06461
[4] https://arxiv.org/abs/1511.06581
[5] https://doi.org/10.1109/ACCESS.2020.2964042