Akademie věd České republiky
Ústav teorie informace a automatizace, v.v.i.

Academy of Sciences of the Czech Republic
Institute of Information Theory and Automation

RESEARCH REPORT

Ing. Milan Berka

Experiment: Cooperative Decision Making via Reinforcement Learning

No. 2380 December 29, 2018

ÚTIA AV ČR, P.O.Box 18, 182 08 Prague, Czech Republic
Tel: (+420)266052422, Fax: (+420)286890378, Url: http://www.utia.cas.cz, E-mail: utia@utia.cas.cz

This research has been partially supported by GAČR GA16-09848S.

1 Motivation

This report was inspired by [1], where authors developed a new method (within the *fully probabilistic design* framework) for solving cooperative decision making tasks. The aforementioned method assumes multiple agents with different goals, where agents can share their goals with each other. These agents are *wise* and *adaptive*, i.e. they can choose to "sacrifice" their goal and work together towards the best-possible compromise.

Authors demonstrated the method on the following simple example: two agents (heaters A, B) are in one room, heater A wants the room to have temperature T_A, heater B wants the room to have temperature T_B and each heater can choose to be either ON (trying to increase the room's temperature) or OFF (trying to decrease the room's temperature). Cooperation is vital - if both heaters act selfishly and independently, there will be an infinite clash and both will struggle to achieve their ideal temperatures.

Goal of this work is to apply the increasingly popular *reinforcement learning* methods [2],[3],[4] to the example above and observe the result.

This report consists of four main parts: Section 1, where a brief overview of reinforcement learning (both single-agent and multi-agent) is given. Section 2 is the actual experiment, where two particular reinforcement learning approaches are applied to the heaters example. Section 3 is the conclusion and Appendix contains the actual Python source code for the experiment.

2 Introduction to Reinforcement Learning

To get the intuition which type of problems reinforcement learning (RL) tackles, let's review a typical RL setting: we throw an agent (decision maker) into some unknown environment; at each step, the agent is offered actions $A_1, A_2, \ldots$, which he can take; the chosen action then takes agent from the old state s to a new state s' in the environment and agent is given some reward (reinforcement) r for making the move. This can repeat many times, creating an agent-environment interaction loop (Figure 1). The goal of reinforcement learning is to learn a *strategy* that maximizes the overall cumulative reward - in another words: given state I am in, which action/path should I take so that I collect the maximum amount of reward r along the way.

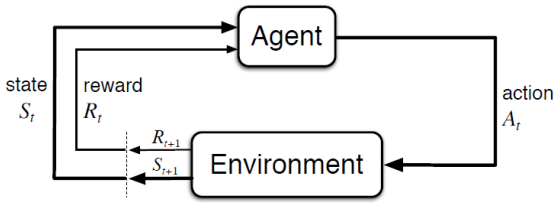

Figure 1: The agent-environment interaction in reinforcement learning. Picture taken from [2].

To make things even more concrete, imagine the popular game of Black Jack - the cards on the board and in player's hand represent *state*, at each step player can choose from two *actions* – either *HIT* (take another card from the dealer) or *STAND* (take no more cards and resolve the game); if the player wins the game, he gets *reward* $+1$; if he loses, he gets reward -1; during the game, he gets 0 reward at each step. Goal is to find *strategy* that tells player "given the current cards in your hand and on the board, the best action is to HIT (or STAND)". Another example could be a trading on a stock exchange – current price of the stock represents the state, actions could be either *BUY, SELL* or *KEEP* and reward would be how much money one made. Again, the goal is to find a trading strategy that maximizes the earnings.

Overall, games provide a great arena for reinforcement learning development as one can play them cheaply over and over again. State-of-the-art RL algorithms are able to beat humans in Atari games [5], chess, Go [6] or even DOTA 2 [7]. Other examples of successful reinforcement learning applications include robotics, autonomous vehicles, biology, portfolio management, online banner placement and more [2],[8],[9].

As such, reinforcement learning is considered a branch of machine learning. It bears a resemblance to the supervised learning as both are trying to find the best response given current state (features). The difference is that in supervised learning, the agent is a priory *given the right answer* (target), which he can use for training, whereas in reinforcement learning, the agent has to *discover the right answer* through a smart interaction with the environment.

This brings up two distinct problems of RL [2]. First is the problem of *credit assignment* – let us assume we played a game, made hundreds of decisions (actions) along the way and won; the problem now is how to decide which actions were actually responsible for the win, i.e. how much credit should be assigned to each action that was taken. Second problem is the *exploration vs exploitation dilemma* – it is necessary for the agent to explore the unknown environment, but after some time he wants to start taking actions that lead to a high cumulative reward. Problem is that if the agent starts acting greedily (taking actions that previously led to a high reward), he might miss some hidden paths that could lead to even higher reward. Good analogue in real world could be choosing a meal in a restaurant. One could always go for his favourite meal (exploitation) which gives him a high satisfaction (reward), but he might also risk and try other meals (exploration) and maybe he will find a meal even better than his currently-favourite one.

2.1 Single-Agent Reinforcement Learning

Formally, we use *Markov Decision Process* (MDP) to model the reinforcement learning problems.

Markov Decision Process consists of:

- $\mathcal{S}$ – set of states.
- $\mathcal{A}$ – set of actions agent can take.
- $P_a(s, s') = \Pr(S_{t+1} = s' | S_t = s, A_t = a)$ – transitional probability from state s to state s'.
- $R_a(s, s')$ – reward function.

General MDPs can have uncountable states and actions.

Agent's behaviour is described by his *strategy* (policy) $\pi(a_t|s_t)$ which specifies how the agent chooses his actions given the state. Strategy can be deterministic or stochastic.

The agent's ultimate goal is to find strategy that maximizes the *expected reward*

$$G_t = E\left[\sum_{j=0}^{\infty} \gamma^j r_{t+j+1}\right], \tag{1}$$

where r_t is a reward at time t and $\gamma \in [0, 1]$ is a discount factor. Idea behind discount factor is to appreciate the imminent rewards more than the future (therefore unsure) rewards. Also it provides a mechanism to bound the sum which might otherwise grow infinitely.

Next, we introduce two major concepts - *value function* $V^\pi(s)$ and *action-value function* $Q^\pi(s, a)$:

$$V^\pi(s) = E\left[\sum_{j=0}^{\infty} \gamma^j r_{t+j+1} | S_t = s\right], \tag{2}$$

$$Q^\pi(s, a) = E\left[\sum_{j=0}^{\infty} \gamma^j r_{t+j+1} | S_t = s, A_t = a\right]. \tag{3}$$

Idea behind value function $V^\pi(s)$ is following: given I am in state s, how much total reward will I accumulate if I follow the policy π from now onward. Idea behind action-value function $Q^\pi(s,a)$ is similar, but now I am asking myself: given I am in state s, how much total reward will I accumulate if I now choose action a and follow the policy π afterwards.

We define *optimal* value functions as

$$V^*(s) = \max_\pi V^\pi(s), \tag{4}$$

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \tag{5}$$

The following equality holds from the definition of value function:

$$V^*(s) = \max_a Q^*(s,a) \tag{6}$$

and $Q^*(s,a)$ satisfies the recursive *Bellman optimality equation*:

$$Q^*(s,a) = \sum_{s' \in \mathcal{S}} P_a(s,s')[R_a(s,s') + \gamma \max_{a'} Q^*(s',a')]. \tag{7}$$

If we know the transitional probabilities $P_a(s,s')$ and reward function $R_a(s,s')$, the Bellman optimality equation (7) can be solved using dynamic programming methods [10].

Notice that once we know the optimal state-value function $Q^*(s,a)$ for each state s, we can construct the *optimal strategy* simply as

$$\pi^*(a|s) = \operatorname*{argmax}_a Q^*(s,a). \tag{8}$$

When the transitional probabilities $P_a(s,s')$ and reward function $R_a(s,s')$ are apriori unknown, we cannot solve the Bellman equation right from the get-go, we have to first obtain *samples* from the environment and proceed with them. This is the typical RL setting, in which we put our agent into unknown environment and he has to explore it and learn from the samples of states and rewards he observes.

There are many types of single-agent RL algorithms, e.g. *model-based* methods which aim to estimate $P_a(s,s')$, $R_a(s,s')$ from the samples and then apply dynamic programming approach; *model-free* methods, which aim to learn $V^*(s), Q^*(s)$ from the samples and then derive optimal strategy $\pi^*(a|s)$ or optimize the strategy directly (policy gradients); or *combinations of both*. If the state-action space is too large, one could use *approximators* such as neural networks, which leads to *deep reinforcement learning*. [2]

2.1.1 Q-learning

Perhaps the most well-known RL algorithm is the *Q-learning*, which turns the Bellman equation (7) into an iterative approximation procedure:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s,a') - Q(s,a)], \tag{9}$$

where $\alpha \in (0,1]$ is the learning parameter and can vary in time. Q-learning is model-free method as it does not require the knowledge of $P_a(s,s')$ or $R_a(s,s')$. Full algorithm will be shown in Section 2.

$Q(s,a)$ from the Q-learning algorithm provably converges to the real $Q^*(s,a)$ under following conditions [11]:

- $\sum_t \alpha_t = \infty$.
- $\sum_t \alpha_t^2 < \infty$.
- Agent keeps visiting all possible state-action pairs with non-zero probability.

The optimal strategy can be again derived greedily from (8). However, acting always greedily would kill the exploration (and the third convergence condition would be violated). To remedy this, one can introduce a random element to the strategy, e.g. by choosing the greedy action with probability $\epsilon \in (0, 1)$ and random action with probability $(1 - \epsilon)$. This approach is called *ϵ-greedy*. Another option is to use *soft-max* (Boltzmann) exploration, in which agents selects the action a in state s with probability

$$\pi(a|s) = \frac{\exp^{Q(s,a)/\tau}}{\sum_{a'} \exp^{Q(s,a')/\tau}}, \tag{10}$$

where τ controls the randomness (0 = greedy, $\to \infty$ = completely random). More sophisticated approaches exist, such as "optimisms in the face of uncertainty" or "information state-space" methods. Perhaps surprisingly, *ϵ-greedy* and *soft-max* approaches tend to work very well in the practice [2].

2.2 Multi-Agent Reinforcement Learning

When the environment is occupied by two or more agents who learn, we talk about *multi-agent reinforcement learning*, often abbreviated as MARL [9].

We extend the Markov decision process to *Markov stochastic game* (MSG), which consists of:

- $\mathcal{S}$ – set of states.
- $\mathcal{A}^1, \ldots, \mathcal{A}^n$ – set of actions, specific for each of the n agents.
- $P_{a^1,\ldots,a^n}(s, s') = \Pr(S_{t+1} = s' | S_t = s, A_t^1 = a^1, \ldots, A_t^n = a^n)$ – transitional probability from state s to state s' given agents' actions $a^1, \ldots, a^n$.
- $R^1_{a^1,\ldots,a^n}(s, s'), \ldots, R^n_{a^1,\ldots,a^n}(s, s')$ – reward function, specific for each of the n agents. Implicitly depends on the actions of other agents.

If $R^1_{\mathbf{a}} = \ldots = R^n_{\mathbf{a}}$, all agents have the same goal and the game is *fully cooperative*. On the other hand, if $R^1_{\mathbf{a}} = -R^2_{\mathbf{a}}$ (here for $n = 2$), the game is *fully competitive*. If the game is neither fully cooperative nor fully competitive, we call it a *mixed game*.

If $\mathcal{S} = \emptyset$, we call the game *static (stateless)*. Without the state signal, the rewards depend only on the joint actions of the agents. Example of such a game could be Rock-Paper-Scissors. When agents play the same static game multiple-times, we call it a *repeated game*. Difference between static and repeated game is that in repeated game, agents may use the gathered information from previous runs to adjust their strategies. When the state information is present, we call the game *dynamic*.

Compared to the single-agent, multi-agent setup is significantly more challenging. As the number of agents increases, the (discrete) state-action space grows exponentially. This induces a severe *curse of dimensionality*. Another problem is the *nonstationarity* of the environment. Learning agents change their strategies which may result in (from other agents' point of view) unexpected influence on the environment. This creates a moving-target learning problem for each agent to which he must react (therefore moving the target for others, again – creating a cyclic problem). As a result, the convergence properties from single-agent RL do not generally translate to multi-agent settings. *Exploration vs exploitation* must be also carefully balanced. If agents' exploration is excessive, i.e. there is huge random effect, the learning process of other agents may be destabilized.

On the other hand, there are multiple benefits of MARL – computational speed-up thanks to parallel computation; robust learning thanks to experience sharing; or teaching (knowledge transfer) between agents. One might also argue that MARL is closer to the artificial general intelligence concept, for which the interaction with other agents is crucial.

Table 1 provides an overview of various MARL algorithms for various game types. These can be further broken down, e.g. by criteria of: *homogeneity* (whether the algorithm works only if all agents uses it), *model-free vs model-based* (whether agents build model of their opponents and the environment or not), *agent-tracking vs agent-aware vs agent-independent* (whether agents need to track other agents' actions or just know that other agents exist or neither) and more [9].

Fully cooperative		Fully competitive		Mixed	
Static	**Dynamic**	**Static**	**Dynamic**	**Static**	**Dynamic**
JAL	Team-Q	Minimax-Q		Fictitious Play	Single-agent RL
FMQ	Distributed-Q			MetaStrategy	Nash-Q
	OAL			IGA	CE-Q
				WoLF-IGA	Asymmetric-Q
				GIGA	NSCP
				GIGA-WoLF	WoLF-PHC
				AWESOME	PD-WoLF
				Hyper-Q	EXORL

Table 1: Overview of MARL algorithms divided by game types as outlined in [9].

3 Experiment

In this section, we take two MARL approaches and apply them to the "Two Heaters in One Room" example from [1]. First, both agents will employ the single-agent Q-learning. Second, both agents will employ the WoLF-PHC algorithm.

In this section, we first outline both algorithms (from i-th player perspective), then describe the experiment setup and finally discuss the results.

3.1 Q-learning Algorithm

We will proceed with a standard Q-learning algorithm with ϵ-greedy strategy (Algorithm 1). Because both states and actions will be discrete, we can store the $Q(s, a)$ values in a table (there is no need for function approximations).

Algorithm 1 Single-agent Q-learning with ϵ-greedy strategy

Initiate the matrix Q with zeros. Q is of shape $(|\mathcal{S}|, |\mathcal{A}^i|)$, where $|\mathcal{S}|$ is the number of states and $|\mathcal{A}^i|$ is the number of available actions for i-th player.
Set the number of episodes T.
Set the number of steps in one episode K.
Set the learning rate $\alpha \in (0, 1]$ (can be possibly time-varying) .
Set the discount factor $\gamma \in [0, 1]$.
Set decaying schedule for ϵ, for example $\epsilon_t = \min(1, 0.01T/t + 1)$.

For $t = 1, 2, \ldots, T$ do:

1. Set ϵ_t according to the decaying schedule.
2. Observe the first state s.
3. For $k = 1, 2, \ldots, K$:
   (a) Sample u from $Uniform(0, 1)$.
   (b) If $\epsilon_t < u$ take greedy action
   $$a = \underset{a'}{\operatorname{argmax}}\, Q(s, a'),$$
   else take an action from $\mathcal{A}^i$ at random.
   (c) Observe reward r and new state s'.
   (d) Update the $Q(s, a)$ according to:
   $$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s, a') - Q(s, a)].$$
   (e) Set $s \leftarrow s'$.

3.2 WoLF-PHC Algorithm

WoLF-PHC stands for "Win-or-Learn-Fast Policy Hill Climbing". The algorithm is outlined below.

Algorithm 2 WoLF-PHC with ϵ-greedy strategy

Initiate the matrix Q with zeros. Q is of shape $(|\mathcal{S}|, |\mathcal{A}^i|)$, where $|\mathcal{S}|$ is the number of states and $|\mathcal{A}^i|$ is the number of available actions for i-th player.
Initiate counter $C(s) \leftarrow 0 \; \forall s \in \mathcal{S}$.
Initiate T as the number of episodes.
Initiate K as the number of steps in one episode.
Set the learning rates $\alpha, \delta_l > \delta_w \in (0, 1]$ (α can be possibly time-varying).
Set the discount factor $\gamma \in [0, 1]$.
Set decaying schedule for ϵ, for example $\epsilon_t = \min(1, 0.01T/t + 1)$.
Set the strategy $\pi(a|s) = |\mathcal{A}^i|^{-1}$, where i represents i-th player.

For $t = 1, 2, \ldots, T$ do:

1. Set ϵ_t according to the decaying schedule.
2. Observe the first state s.
3. For $k = 1, 2, \ldots, K$:
   (a) Sample u from $Uniform(0, 1)$
   (b) If $\epsilon_t < u$ sample $a \sim \pi(a|s)$ else sample action at random.
   (c) Take action a, observe reward r and new state s'.
   (d) Update the $Q(s, a)$ according to:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s, a') - Q(s, a)].$$

   (e) Update the estimate of average policy, $\tilde{\pi}$,

$$C(s) \leftarrow C(s) + 1,$$
$$\forall a' \in \mathcal{A}^i : \tilde{\pi}(s, a') \leftarrow \frac{1}{C(s)} \left(\pi(s, a') - \tilde{\pi}(s, a')\right).$$

   (f) Determine δ as

$$\delta = \begin{cases} \delta_w & \text{if } \sum_{a'} \pi(a'|s) Q(s, a') > \sum_{a'} \tilde{\pi}(a'|s) Q(s, a') \\ \delta_l & \text{otherwise.} \end{cases}$$

   (g) Update strategy $\pi(a|s)$:

$$\pi(a|s) \leftarrow \pi(a|s) + \begin{cases} \delta & \text{if } a = \text{argmax}_{a'} Q(s, a') \\ -\frac{\delta}{|\mathcal{A}^i| - 1} & \text{otherwise.} \end{cases}$$

   (h) Set $s \leftarrow s'$.

It is *agent-aware* algorithm for *mixed dynamic game*. Agent-aware means it takes into account that other agents might be in the environment too, but it does not observe their actions nor communicates with them.

WoLF-PHC is based on Q-learning but with an additional strategy tuning. Rationale is that when agent is loosing, he should change his strategy more aggressively, whereas when agent is winning he should aim to stabilize his strategy (therefore two learning rates $\delta_l > \delta_w$). To determine if agent is winning, algorithm checks whether the expected Q-value under current policy $\pi(a|s)$ is

higher than under the "average" policy $\tilde{\pi}(a|s)$. The average policy is intended to represents some unknown equilibrium policy.

3.3 Experiment Setup

As in [1], we consider two agents (heaters $1, 2$) in one room. Both influence the common room temperature s. The discretized temperature is the observable state $s \in \{1, \ldots, 20\}$. Both agents can choose one of two actions $\{1, 2\} = \mathcal{A}^1 = \mathcal{A}^2$, where 1 represents *OFF* and 2 represents *ON*. Number of steps during one episode is $K = 500$.

The room temperature is modelled by Gaussian transition probabilities

$$s_{t+1} \sim \mathcal{N}(0.65(a_t^1 + a_t^2 - 2) + 0.96 s_t - 0.02, 0.25) \tag{11}$$

where s_{t+1} is then rounded. The initial state is $s_0 = 9$.

Next we define the *reward functions*. We define four reward functions (we will adress them as a reward modes) for each agent, see Table 2.

Reward Mode	**Cooperative**				**Mixed**			
	Hard		**Soft**		**Hard**		**Soft**	
States \Agents	**1**	**2**	**1**	**2**	**1**	**2**	**1**	**2**
9	0	0	0	0	0	0	0.05	0
10	0	0	0	0	0	0	0.2	0
11	0	0	0.05	0.05	1	0	1	0
12	0	0	0.2	0.2	0	0	0.2	0.05
13	1	1	1	1	0	0	0.05	0.2
14	0	0	0.2	0.2	0	1	0	1
15	0	0	0.05	0.05	0	0	0	0.2
16	0	0	0	0	0	0	0	0.05
Otherwise	0	0	0	0	0	0	0	0

Table 2: Different types of rewards for agent 1 and 2 given the state of the environment (temperature in the room). For example, if temperature $s = 13$, both agents will get $+1$ reward under *Cooperative Hard* reward mode. However, under *Mixed Soft* reward mode, Agent 1 will get reward $+0.05$, while Agent 2 will get reward $+0.2$ for achieving the temperature $s = 14$. In this case, the reward function is not explicitly dependent on the action or previous state.

As can be seen from the Table 2, under cooperative mode, agents share the same reward functions and we expect that agents should exhibit cooperative behaviour. Under mixed mode, we expect a slight competition between the two.

Looking at the table, we could say that the *ideal* temperature in cooperative mode is $s_i = 13$ for both agents, while in mixed mode, the ideal temperature for agent 1 is $s_i^1 = 11$ and for agent 2 $s_i^2 = 14$. *Soft reward function* awards agent for at least approaching his ideal temperature, while *hard reward function* is awarding agent only when the current room's temperature really matches the agent's ideal temperature.

We run the simulation as described in Algorithm 3 with all possible combinations of reward modes and learning algorithms. We initiate the parameters for both learning algorithms as follows: learning rate $\alpha_t = 0.9$, discount factor $\gamma = 0.9$ and $\delta_w = 0.05, \delta_l = 0.2$ (for WoLF-PHC). We use the ϵ-greedy strategy with the following decaying $\epsilon_t = \min(1, 200/t + 1)$.

3.4 Results

We run the simulations and compare accumulated rewards of both agents. Each simulation (the 5000 episodes) is run twenty times and the results are put together (e.g. by using box-plots; averaging over runs or explicitly denoting the run number).

Algorithm 3 SIMULATION

Set the number of episodes $T = 5000$.
Set the number of steps within one episode $K = 500$.
Initiate both agents $1, 2$.

For $t = 1, 2, \ldots, T$ do:

1. Set the initial observation $s = s_0 = 10$.
2. For $k = 1, 2, \ldots, K$:
   (a) Let both agents pick their actions a^1, a^2.
   (b) Sample next observation s' from the model (11).
   (c) Given s', get the rewards r^1, r^2 (according to Table 2).
   (d) Let both agents learn.
   (e) $s \leftarrow s'$.

3.4.1 Cooperative Reward Mode

First, we run the simulations in cooperative reward mode. Recall that both agents (heaters) will have the same goal - reach and keep the temperature $s_i = 13$.

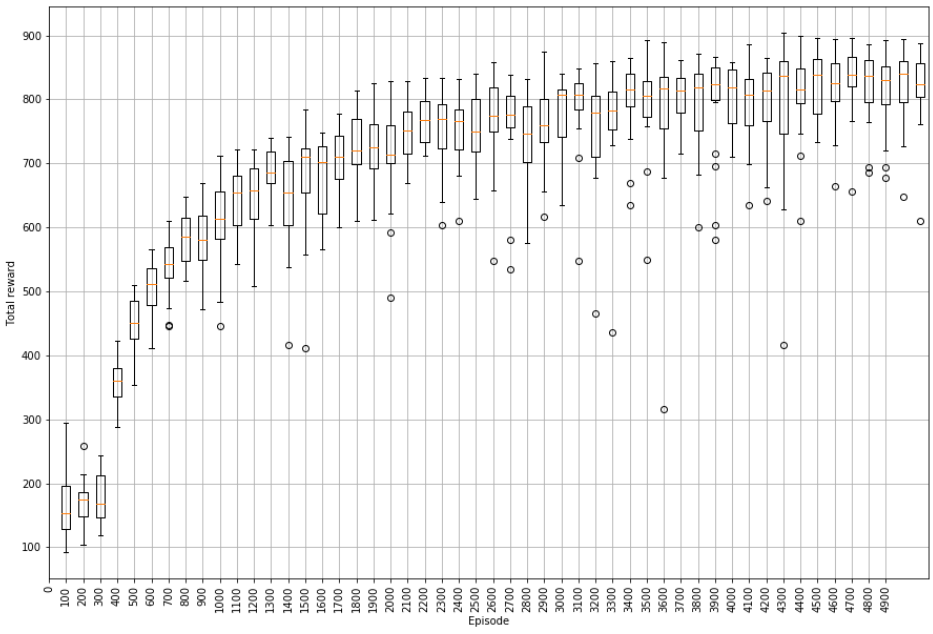

Figure 2: Box-plot of sum of both agent's total reward after each episode using Q-learning with hard reward function.

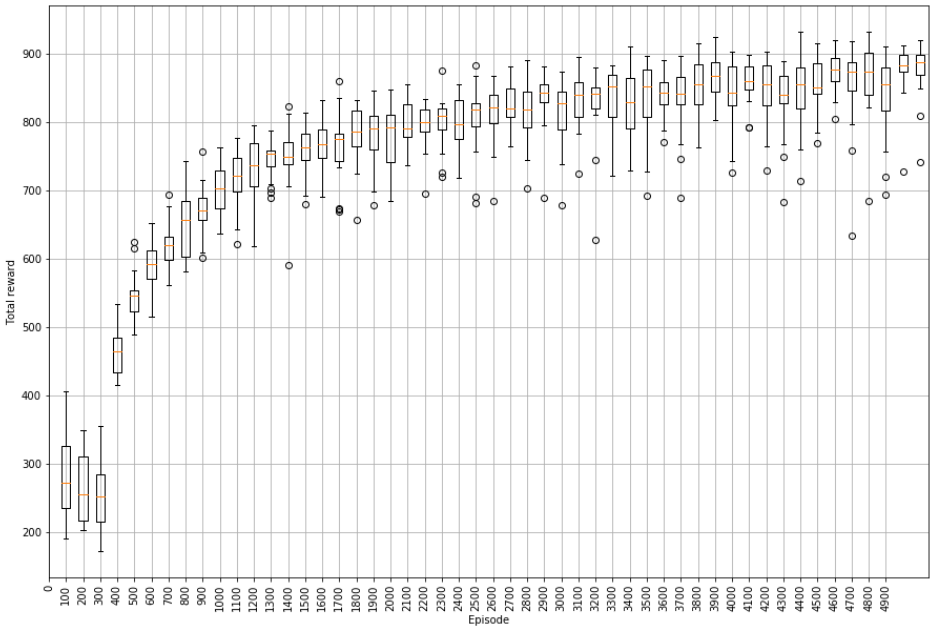

Figure 3: Box-plot of sum of both agent's total reward after each episode using Q-learning with soft reward function.

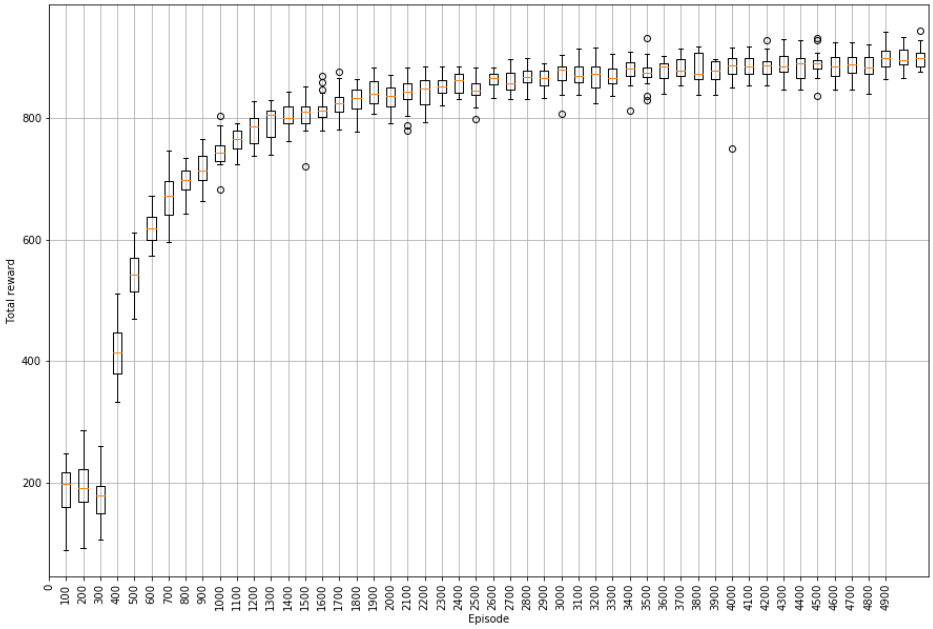

Figure 4: Box-plot of sum of both agent's total reward after each episode using WoLF-PHC with hard reward function.

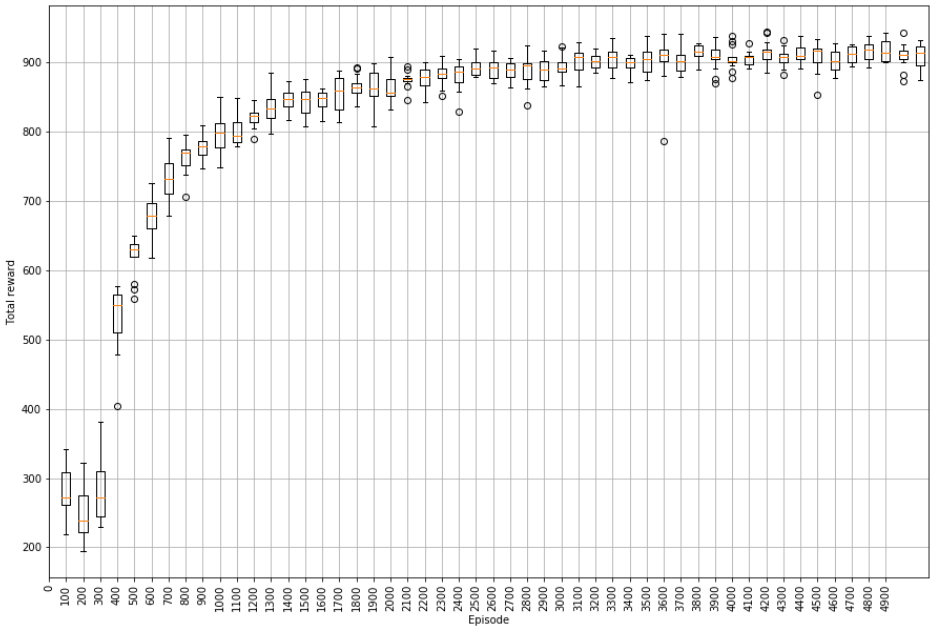

Figure 5: Box-plot of sum of both agent's total reward after each episode using WoLF-PHC with soft reward function.

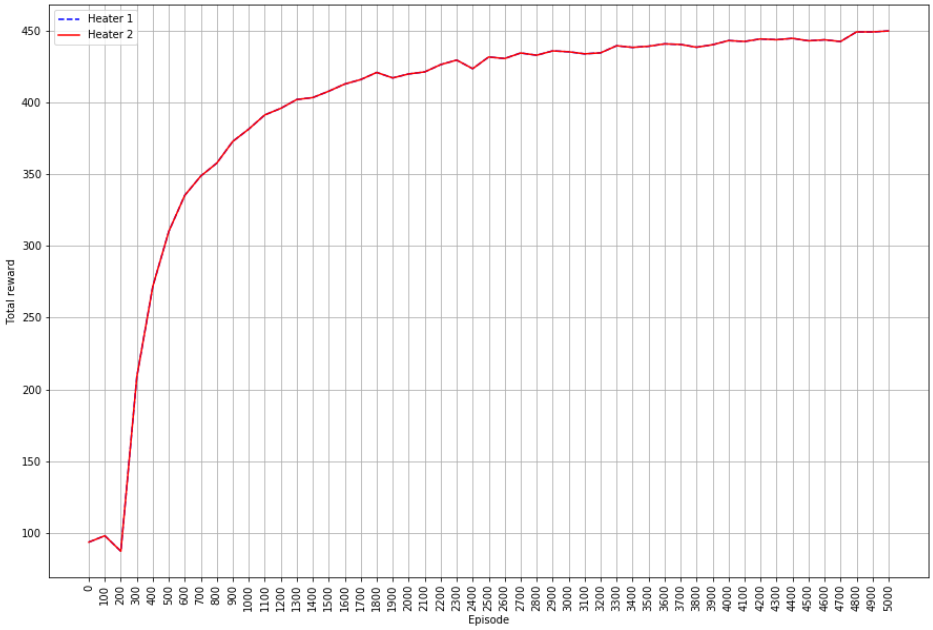

Figure 6: Average reward per each agent (heater) using WoLF-PHC in hard reward mode.

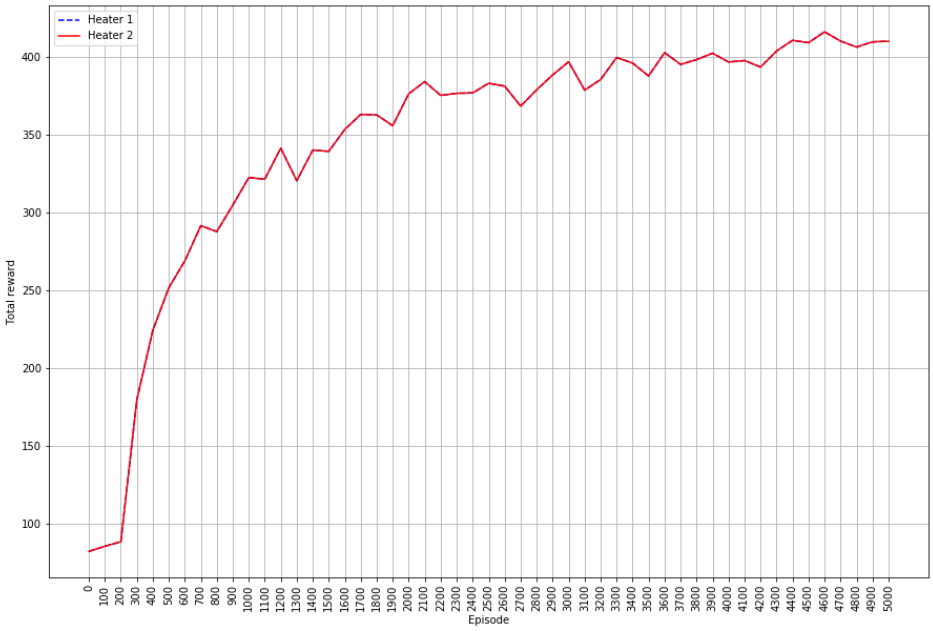

Figure 7: Average reward per each agent (heater) using Q-learning in hard reward mode.

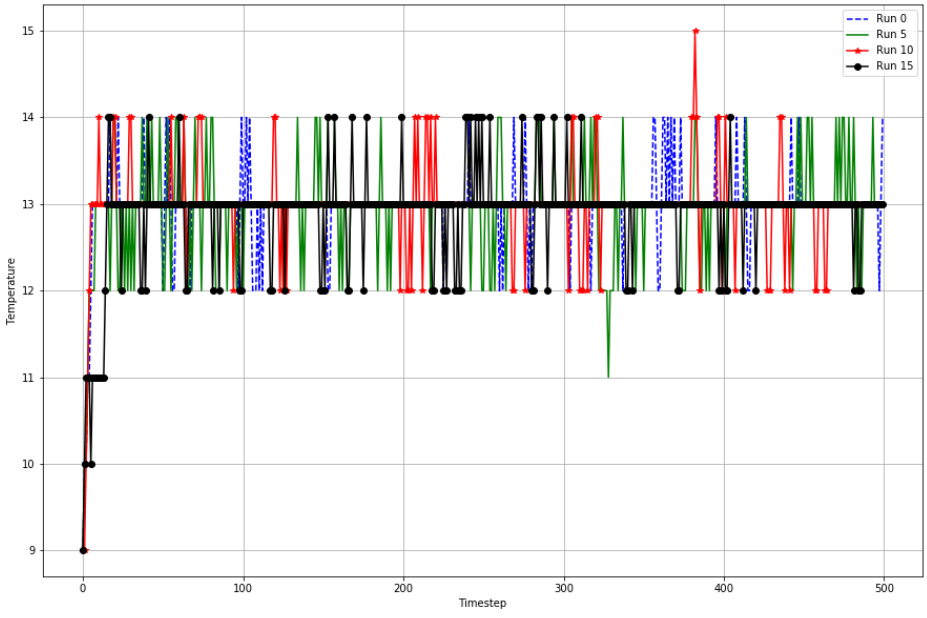

Figure 8: Trajectory of temperature in 4000th episode for Q-learning and hard reward mode.

Figures 2 - 5 show the evolution of how much reward both agents accumulate together by the end of each episode. As one can see, there is a start-up period of about 300 episodes where the algorithm is exploring heavily (this can be attributed to the ϵ decay). After the initial periods, total reward of both agents starts steeply increasing. After some episodes, it settles at around +900 reward per episode.

It is interesting to note that it almost does not matter whether we use the hard or soft reward mode. Both converge to very similar total reward value.

Comparing the Q-learning vs WoLF-PHC, we can see that Q-learning box-plots show higher variance. This indicates that WoLF-PHC approach is more stable. The average episode return of both agents (Figures 6 and 7) are higher for the WoLF-PHC algorithm. The reward lines on both pictures are overlapping since both get reward always at the same time.

Figure 8 shows a evolution of room's temperature. One can see that it steadily fluctuates around the common ideal state of 13.

3.4.2 Mixed Reward Mode

Here, we switch to the mixed reward mode. Recall that the ideal temperature for agent 1 is now $s_i^1 = 11$, whereas for agent 2 it is $s_i^2 = 14$. One should expect clashes between the two.

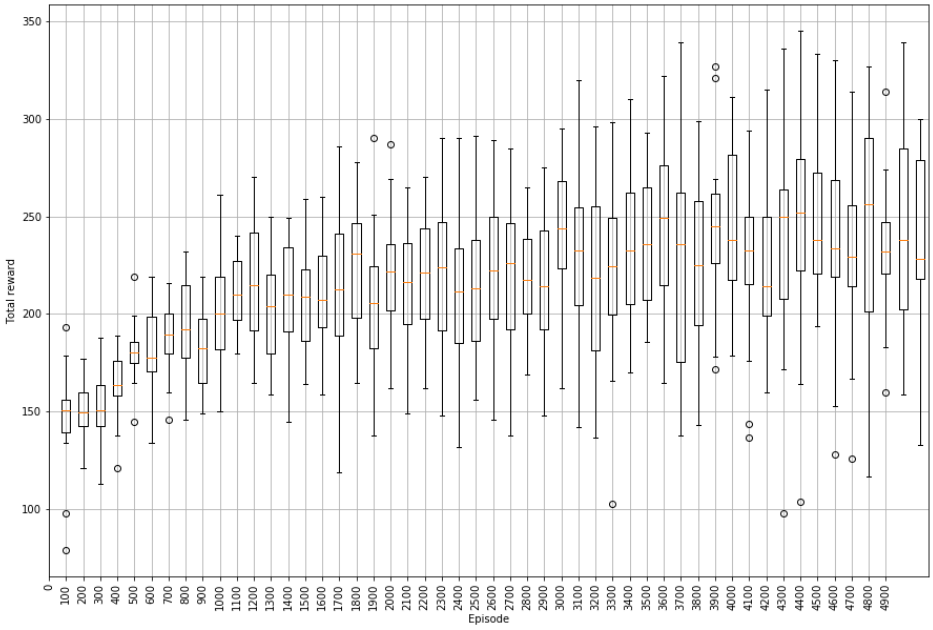

Figure 9: Box-plot of sum of both agent's total reward after each episode using Q-learning with hard reward function.

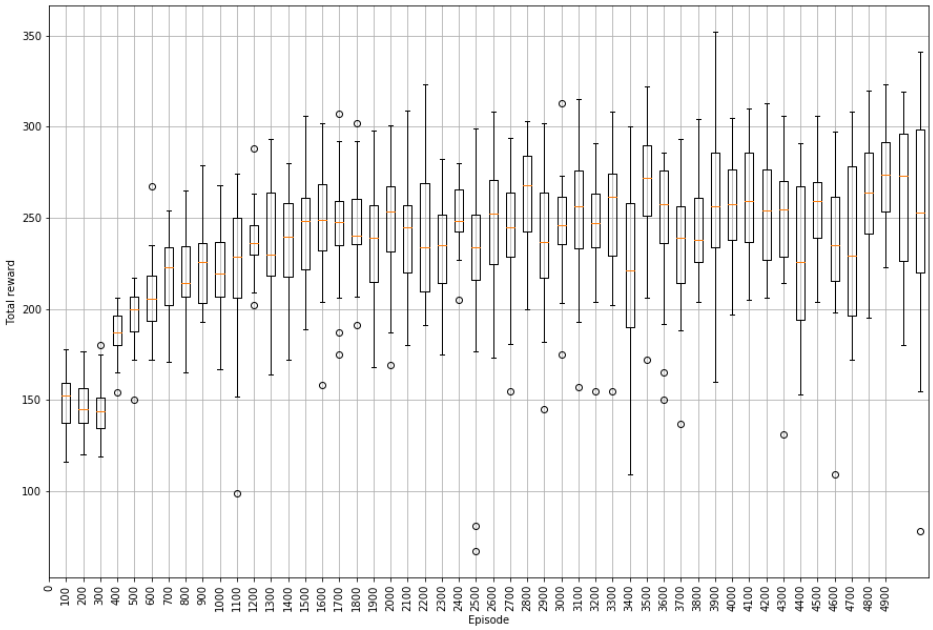

Figure 10: Box-plot of sum of both agent's total reward after each episode using WoLF-PHC with hard reward function.

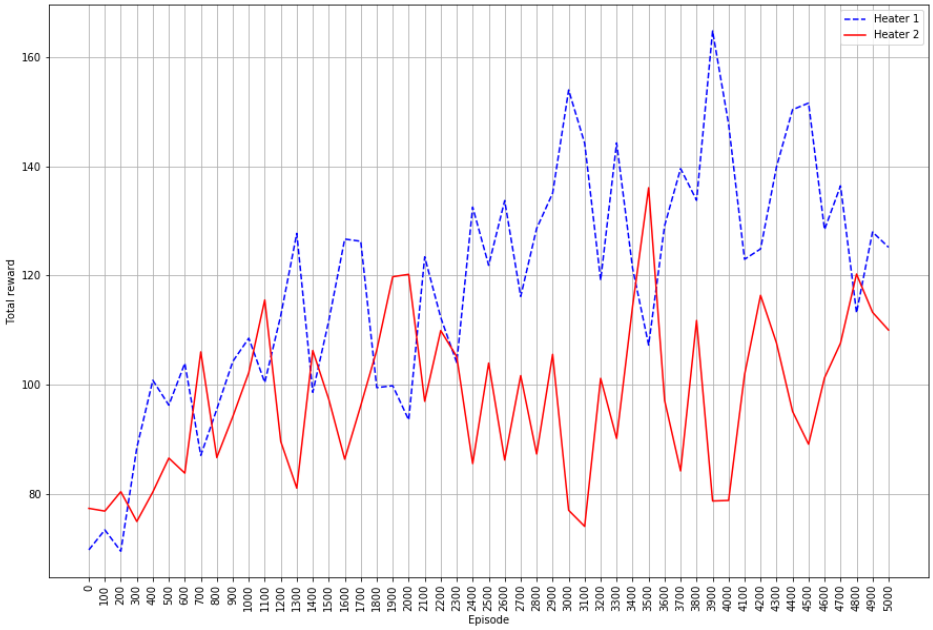

Figure 11: Average reward per each agent (heater) using Q-learning in hard reward mode.

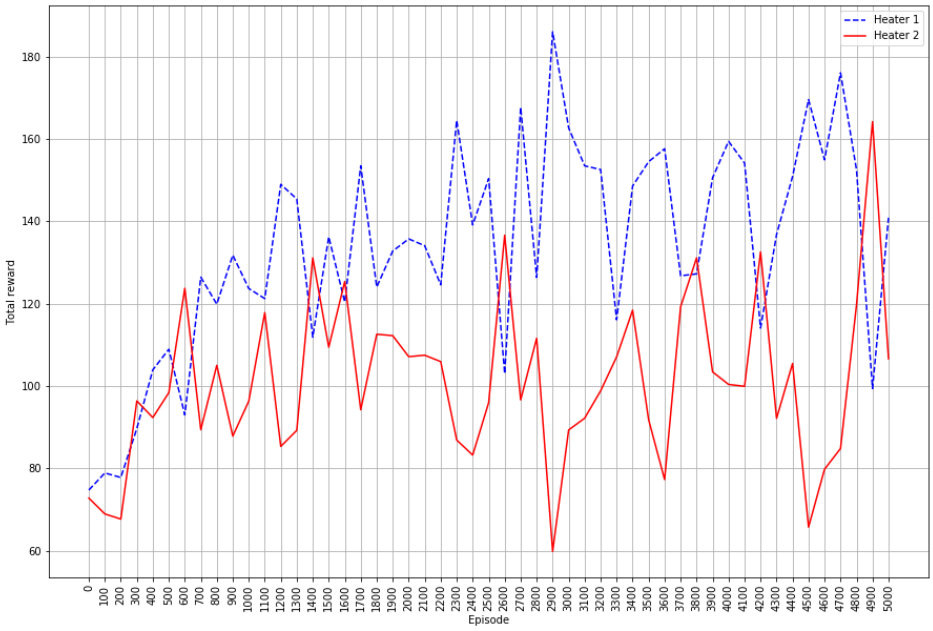

Figure 12: Average reward per each agent (heater) using WoLF-PHC in hard reward mode.

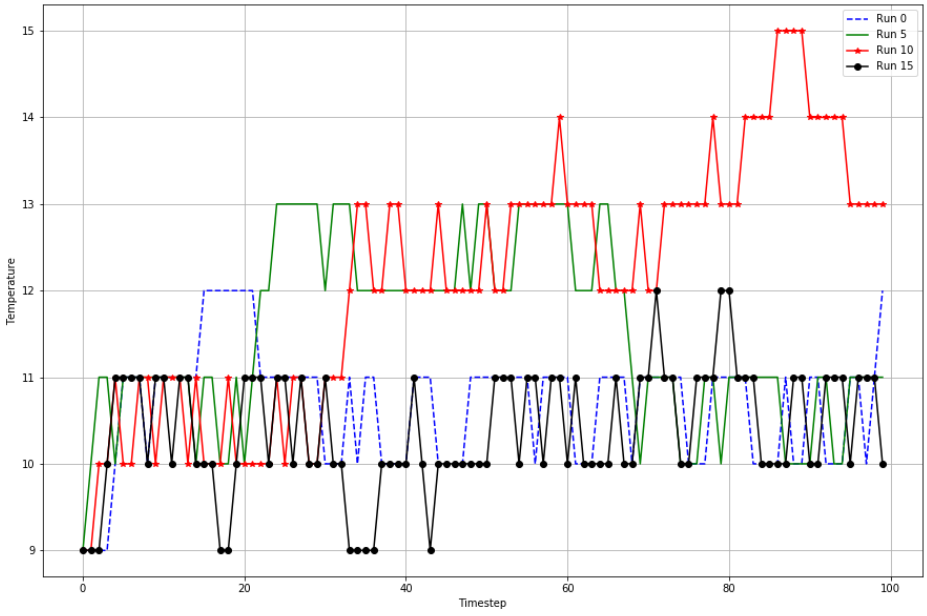

Figure 13: First 100 steps of temperature's trajectory in 4000th episode for WoLF-PHC and hard reward mode.

Similarly to the previous section, Figures 9, 10 show the evolution of how much reward both agents accumulate together by the end of each episode. One can see that there is still an upward trend, but is not as steep. The variance is much higher and the total reward plateaus around +270 reward for WoLF-PHC and +230 reward for Q-learning.

Figures 11, 12 show that the Heater 1 tends to accumulate more reward than Heater 2. One could suspect that Heater 1 is exploiting the asymmetric nature of the environment (the heater may contribute to the temperature increase but the decrease depends only on the environment dynamics and on the realization of random influences) [1].

3.5 Discussion

The results show the big difference between cooperation and "competition" (mixed) tasks. When two agents work toward specific goal, it is much easier to align their actions, whereas working toward different goals inherently causes instability and exhibits higher variance.

Overall the agent-aware WoLF-PHC algorithm shows better performance compared to the agent-independent Q-learning algorithm, which was expected.

4 Conclusion

In this report a brief overview of Reinforcement Learning (both single- and multi- agent) was given and two specific algorithms (Q-learning, WoLF-PHC) were applied to a simple multi-agent problem.

Unlike in the single-agent settings, a lot of different situations can arise in the multi-agent setup – agents can act on the range from being fully cooperatively to being fully competitive; can be aware of other agents to different degree; can influence the environment differently etc. Thus, there is no "ultimate" algorithm that would fit all, but rather a palette of algorithms, where each is suited for particular settings.

Here, we employed two MARL algorithms, but as we saw there are plenty more one could use – ranging from agent-independent model-free methods to agent-tracking model-based methods.

We can conclude that multi-agent reinforcement learning is still a rapidly expanding area of research as new impulses from single-learning reinforcement learning, deep learning, game theory or control theory are continuously streaming in.

References

[1] M. Kárný and Z. Alizadeh. "Towards Fully Probabilistic Cooperative Decision Making," *EUMAS 2018, accepted, in print*, 2018.

[2] R. S. Sutton and A. G. Barto *Reinforcement Learning: An Introduction.* The MIT Press, 2012.

[3] S. Sen and G. Weiss, "Learning in multiagent systems," in Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence The MIT Press, 6:259–298, 1999

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, 4:237–285, 1996

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller "Human-level control through deep reinforcement learning," *Nature*, 518(7540):529–533, 2015

[6] D. Silver, A. Huang, C. Maddison, G. Arthur, L.Sifre, G. van den Driessche J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis "Mastering the game of Go with deep neural networks and tree search," *Nature*, 529: 484–489, 2016

[7] OpenAI "OpenAI Dota 2 1v1 bot," URL https://openai.com/the-international/, 2017.

[8] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel "Benchmarking deep reinforcement learning for continuous control," *International Conference on Machine Learning*, 1329–1338, 2016

[9] L. Buşoniu, R. Babuška, and B. De Schutter "A comprehensive survey of multi-agent reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 38(2):156–172, 2008

[10] R. Bellman "On the Theory of Dynamic Programming," *Proceedings of the National Academy of Sciences*, 1952

[11] C. J. C. H. Watkins and P. Dayan "Q-learning," *Machine Learning*, 8:279–292, 1992

Appendices

A Appendix - Python codes

A.1 Env.py

```
import matplotlib.pyplot as plt
import numpy as np

def env_sim(a1: int, a2: int, x: int):
    """
    Simulate one step at the environment

    Params:
        - a1 <int>: first heater is off/on (1/2)
        - a2 <int>: 2nd header is off/on (1/2)
        - x <int>: current state

    Returns new state on discrete scale {0, ..., 19}
    """
    mean = 0.65*(a1 + a2 - 2) + 0.95*x - 0.02
    var = 0.25
    s_new_raw = np.random.normal(mean, var)
    s_new_round = np.round(s_new_raw)
    s_new = min(19, max(0, s_new_round)) # range is (0,19) instead of (1, 20) => because of python indexing
    return s_new


def reward(s_cur: int, s_ideal: float, reward_function_type: str='hard'):
    """
    Return reward comparing the current state with the ideal state (aka temperature)

    Params:
        - s_current <int>: current state
        - s_ideal <float>: ideal state (ideal temprature); can be float, will be rounded for 'hard' types
        - reward_function_type <str>: different ways to calculate reward:
            - "hard"
            - "1-step-hard"
            - "2-step-hard" (soft)
    """
    # Hard
    if reward_function_type == 'hard':
        rew = 1 if s_cur == np.round(s_ideal) else 0

    # 1-Step-Smooth
    if reward_function_type == '1-step-hard':
        if np.abs(s_cur - np.round(s_ideal)) == 1:
            rew = 0.2
        elif s_cur == np.round(s_ideal):
            rew = 1
        else:
            rew = 0

    # 2-Step-Smooth
    if reward_function_type == '2-step-hard':
        if np.abs(s_cur - np.round(s_ideal)) == 1:
            rew = 0.2
        elif np.abs(s_cur - np.round(s_ideal)) == 2:
            rew = 0.05
        elif s_cur == np.round(s_ideal):
            rew = 1
        else:
            rew = 0

    # return reward
    return rew


class TwoHeatersEnv:
    """
    Two heaters environment.

    Suffixes 1, 2 represent the wo heaters:
        - a1: action of heater one
        - a2: action of heater two
        - s_ideal1: ideal temperature (state) for heater 1
        - s_ideal2: ideal temperature (state) for heater 2

    function step(a1, a2):
        - takes the two actions (a1, a2) and returns: new state (temperature), reward1, reward2
    """
    def __init__(self, s_init:int = 9, s_ideal_1 = 11, s_ideal_2 = 14, reward_function_type = 'hard'):
        self.s_init = s_init
        self.s_cur = s_init
        self.reward_function_type = reward_function_type
        self.s_ideal_1 = s_ideal_1
        self.s_ideal_2 = s_ideal_2

    def reset(self):
```

```
        self.s_cur = self.s_init
        return self.s_cur

    def step(self, a1, a2):
        # Input actions will be 0, 1 (indexes of Q_matrix), convert them to 1, 2 for the env formula
        a1_standardized = a1 + 1
        a2_standardized = a2 + 1

        # Move
        observation = int(env_sim(a1_standardized, a2_standardized, self.s_cur))

        # Get rewards
        reward1 = reward(s_cur=observation, s_ideal=self.s_ideal_1, reward_function_type=self.reward_function_type)
        reward2 = reward(s_cur=observation, s_ideal=self.s_ideal_2, reward_function_type=self.reward_function_type)

        self.s_cur = observation

        return observation, reward1, reward2
```

A.2 SimWolfPhc.py

```
import numpy as np
from Env import TwoHeatersEnv

class WoLFPHCAgent:
    """
    Implementing a WoLFPHCAgent
    """
    def __init__(self, s_ideal=None, n_actions=2, n_states=20, delta_win=0.05, delta_loose=0.2):
        self.s_ideal = s_ideal
        self.n_actions = n_actions
        self.n_states = n_states
        self.Q = np.zeros(shape=(n_states, n_actions))
        self.H = (1/n_actions)*np.ones(shape=(n_states, n_actions))
        self.H_AVG = np.zeros(shape=(n_states, n_actions)) # (1/n_actions)*np.ones(shape=(n_states, n_actions))
        self.C = np.zeros(shape=(n_states))
        self.alpha = 0.9
        self.delta_win = delta_win # how much to update our policy if we are winning
        self.delta_loose = delta_loose # how much to update our policy if we are loosnig
        self.gamma = 0.9

#     def get_reward(self, s, s_ideal):
#         # Hard version
#         if np.round(s) == np.round(s_ideal):
#             return 1
#         else:
#             return 0

    def choose_A(self, s, epsilon):
        u = np.random.uniform()
        if u <= epsilon:
            action = np.random.choice(range(self.n_actions))
        else:
            action = np.random.choice(range(self.n_actions), p=self.H[s, :])
        return action

    def update_Q(self, s, a, r, s_next):
        self.Q[s, a] = (1-self.alpha)*self.Q[s, a] + self.alpha*(r + self.gamma*np.max(self.Q[s_next, :]))

    def update_H_AVG(self, s):
        self.C[s] += 1
        self.H_AVG[s,:] = (1/self.C[s])*(self.H[s, :] - self.H_AVG[s, :])

    def update_H(self, s, a):

        # Calculating the deltas
        if self.H[s, :].dot(self.Q[s, :]) > self.H_AVG[s, :].dot(self.Q[s, :]):
            delta = self.delta_win
        else:
            delta = self.delta_loose

        # Updating the policy with delta
        a_argmax = np.argmax(self.Q[s, :])

        for _a in range(self.n_actions):
            if _a == a_argmax:
                self.H[s, _a] += delta # argmax a => increase probability
                self.H[s, _a] = min(1, self.H[s, _a])
            else:
                self.H[s, _a] -= delta/(self.n_actions - 1) # non-argmax a => decrease probability
                self.H[s, _a] = max(0, self.H[s, _a])

    def episodic_update(self, s, a, r, s_next):
        self.update_H_AVG(s)
        self.update_Q(s, a, r, s_next)
        self.update_H(s, a)


""" ALGO RUN """
env = TwoHeatersEnv(9, 11, 14, '2-step-hard')

heater1 = WoLFPHCAgent()
heater2 = WoLFPHCAgent()

episodes_count = 5000 # episodes count
time_horizont = 100 # time-steps per one episode

# LOG:
logger_total_rewards1 = {}
logger_total_rewards2 = {}
logger_state_evolutions = {}
logger_strategies_1 = {}

for t in range(episodes_count):
    epsilon = min(1, 50/(t+1))

    observation = env.reset()

    logger_total_reward1 = []
    logger_total_reward2 = []

    logger_state_evolution = []

    if t == episodes_count:
```

```
        time_horizont = 500

    for k in range(time_horizont):
        cur_observation = observation

        # Choose actions
        action1 = heater1.choose_A(cur_observation, epsilon)
        action2 = heater2.choose_A(cur_observation, epsilon)

        # One step of the environment
        observation, reward1, reward2 = env.step(action1, action2) # 0 => second one is always off

        # Update
        heater1.episodic_update(s=cur_observation, a=action1, r=reward1, s_next=observation)
        heater2.episodic_update(s=cur_observation, a=action2, r=reward2, s_next=observation)

        # LOG: intermediate results
        logger_total_reward1 += [reward1]
        logger_total_reward2 += [reward2]
        logger_state_evolution += [cur_observation]

    # LOG: Total rewards
    logger_total_rewards1[t] = logger_total_reward1
    logger_total_rewards2[t] = logger_total_reward2
    logger_state_evolutions[t] = logger_state_evolution
```

A.3 SimQlearning.py

```
import numpy as np
from Env import TwoHeatersEnv

def choose_egreedy_action(Q, state, epsilon):
    """
    Function that returns q_greedy action
    """
    u = np.random.uniform()
    if u <= epsilon:
        action = np.random.choice(range(Q.shape[1]))
    else:
        action = np.argmax(Q[state, :])
    return action

def find_max_q(Q, state):
    return np.max(Q[state, :])

""" ALGO RUN """
env = TwoHeatersEnv(9, 11, 14, 'hard')

# Params
n_actions = 2 # actions: 0 = off, 1 = on <- 0 = OFF and 1 = ON is important!! See the `step()` definition
n_states = 20 # states: {1, .., 20}
episodes_count = 1000 # episodes count
time_horizont = 500 # time-steps per one episode

Q1 = np.zeros(shape=(n_states, n_actions)) # Agent 1's Q matrix
Q2 = np.zeros(shape=(n_states, n_actions)) # Agent 2's Q matrix
alpha = 0.9 # learning rate
gamma = 0.9 # discount rate

# LOG: Dict loggers
logger_total_rewards1 = {}
logger_total_rewards2 = {}
logger_state_evolutions = {}

for t in range(episodes_count):
    epsilon = min(1, 100/(t+1))

    observation = env.reset()

    logger_total_reward1 = []
    logger_total_reward2 = []

    logger_state_evolution = []
    for k in range(time_horizont):
        cur_observation = observation

        action1 = choose_egreedy_action(Q1, observation, epsilon)
        action2 = choose_egreedy_action(Q2, observation, epsilon)

        observation, reward1, reward2 = env.step(action1, action2) # 0 => second one is always off

        Q1[cur_observation, action1] += alpha*(reward1 + gamma*find_max_q(Q1, observation) - Q1[cur_observation, action1])
        Q2[cur_observation, action2] += alpha*(reward2 + gamma*find_max_q(Q2, observation) - Q2[cur_observation, action2])

        # LOG: intermediate logs
        logger_total_reward1 += [reward1]
        logger_total_reward2 += [reward2]
        logger_state_evolution += [cur_observation]

    # LOG: Total rewards
    logger_total_rewards1[t] = logger_total_reward1
    logger_total_rewards2[t] = logger_total_reward2
    logger_state_evolutions[t] = logger_state_evolution
```