Reinforecement Learning - Assignemnt #1

Ivan Horokhovskyi

Abstract

This assignment was dedicated to some standard methods for value-based tabular reinforcement learning namely Q-learning, SARSA, n-step Q-learning and Monte-Carlo methods. In addition, these algorithms were compared to classic dynamic-programming approach. After that hyperparameter tuning, exploration/exploitation tradeoff was discussed, on/off-policy algorithms comparison experiments and their analysis were done.

1. Introduction

1.1. Theoretical notes

Markov Decision Process (MDP) - a decision making modeling mathematical framework. The outcomes of MDP are partly random and partly under the control of a decision maker. An MDP is defined by (S, A, T_a, R_a, γ). The next state only depends on the current state and the actions which was made.

- S - a finite set of *states* in the environment, the starting state is usually denoted as s_0
- A - a finite set of *actions* in the environment, if the set of actions depends on a state then it is denoted as A_s
- $T_a(s, s')$ - *transition function* which is responsible for the probability of getting from a state s to state s' taking action a in next moment of time.
- $R_a(s, s')$ - reward function which is responsible for providing a reward for an agent for moving from state s to state s' after taking action a.
- $\gamma \in [0; 1], \gamma \in \mathbb{R}$ - *discount factor* which is responsible for discount (reduction) of the reward in future. In extreme cases when $\gamma = 0$ agent only cares about current rewards and when $\gamma = 1$ it considers long-term rewards in same manner is considers current one.

$\pi(a|s)$ - *policy function* which determines which action *a* should make agent in state *s*.

$Q(s_t, a_t)$ - *state-action value function* (also called Q-function) for a state *s* taking action *a* at the moment *t* and shows potential reward for an agent which will make the action *a* with a currently used policy π.

Environments can either deterministic or stochastic. Space of actions are divided into discrete and continuous. The aim of reinforcement learning is to find an optimal policy which will allow to gain maximal cumulative future reward. When agent is capable to use transition model are called model-based, in other case - model-free. In model free approach exploration/exploitation trade-off is very important because it balances existing "best-practices" and search for new approaches of dealing with an environment. The mentioned above symbols and definitions will be used though-out methods.

1.2. Environment

The base environment for the algorithms testing was picked a simple plane (10 x 7) with a starting and finishing point with no obstacles (Figure 1). The starting and finishing points are at the same spots each time: (0,3) and (7,3) correspondingly. Nevertheless, in columns 3,4,5 & 8 and columns 6 & 7 there is wind presence which with 80% probability pushes the agent on 1 or 2 steps to the top accordingly.

This wind is very important in the environment because it adds stochasticity to the process making the agent training more challenging.

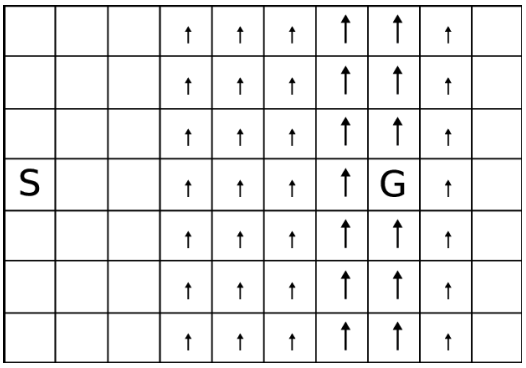

Figure 1. Environment illustration, starting point "S", finishing point "G", arrows represent wind with it's power and direction

1.3. System settings

For the reason of experiments reproducibility setup (environment) are provided:

- Intel® Core™ i7-1065G7 CPU @ 1.30GHz × 8
- Pop!_OS 21.10 (Linux x64)
- Python 3.9.7 (Numpy 1.22.2, Matplotlib 3.5.1)
- gcc 11.2.0 (Ubuntu 11.2.0-7ubuntu2)

Each experiment and plot was averaged and smoothed on 50 runs to reduce variance in experiments. Error bar were not provided to make plots more perceivable. Random states were not fixed, but the results are still highly reproducible and do not differ significantly from run to run.

2. Dynamic Programming

The first observed method is dynamic programming which is often used for various optimization problems (like Knapsack NP-complete problem). The most distinguishing difference for this method is the fact that this algorithm has full access to the environment and transition model. In addition, unlike other method agent starts acting only after all the computations were done, which can be a huge disadvantage due to latency (computational complexity) in less trivial environments.

The core idea is to enter an infinite loop until the $Q(s, a)$ matrix will converge (why it converges will be discussed later in this section). By convergence we mean that maximum difference within each matrix cell differs less than on a predefined threshold between current and previous iteration. Within external loop we iterate over all states and actions from these states and update it as shown in Formula 2. Where $p(s'|s, a)$ is probability to get to state s' from state s after making action a and $r(s, a, s')$ is a reward for reaching state s' from state s after action a was made.

$$Q(s,a) \leftarrow \sum_{s'} [p(s'|s,a) \cdot (r(s,a,s') + \gamma \cdot max_{a'} Q(s',a'))]$$

Figure 2. Dynamic Programming Q-value update

Even though a stochastic environment is used dynamic programming model is deterministic and calculates expected value for each each state this is why it converges and gets to optimum the same way independently of a run state.

On figures (3, 4, 5) Q-value matrix on different stages can be observed. Even after first iteration some pattern can be already observed and an reward propagation from the terminal state is visible. First, the optimal route looked counter intuitive, but after a while it became meaningful. On 10th iteration algorithm almost converges, but still requires a bit of tuning near the starting point which is done on a final 17th iteration.

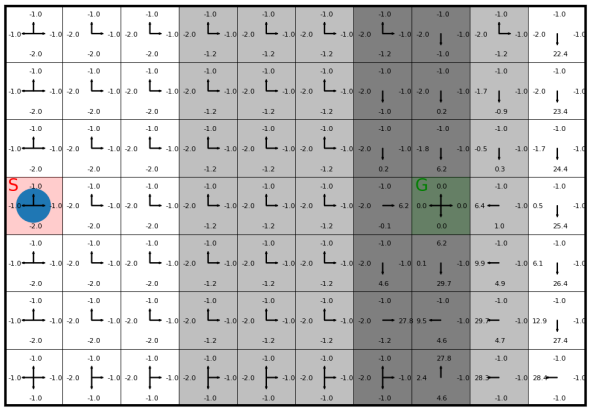

Figure 3. Q-value matrix for dynamic programming, after 1st iteration

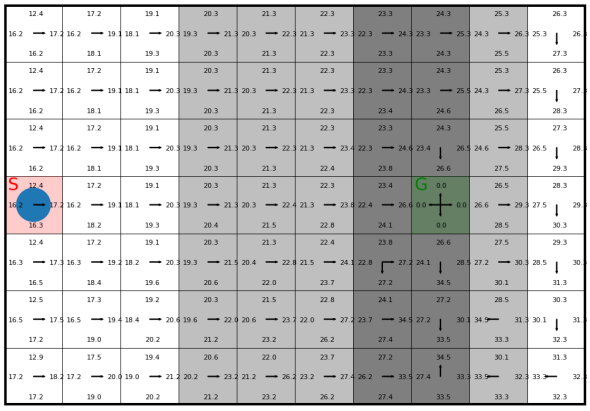

Figure 4. Q-value matrix for dynamic programming, after 10th iteration

Optimal $V^*(s = 3) \approx 18.3$ which is an expected future reward which agent will achieve from the starting state and is equal to the best Q-value in the position.

Average number of steps (X) can be calculated from the equation:

$$-1 * (X - 1) + 35 = V^*(s == 3)$$

So $X \approx 17.7$. From here, average reward per timestep = $\frac{V^*}{X} = \frac{18.3}{17.7} \approx 1.0339$

Regarding convergence, after checking the *Environment.py* file on lines 129-133 terminal state transition/reward logic is present. If terminal state was reached, environment ensures

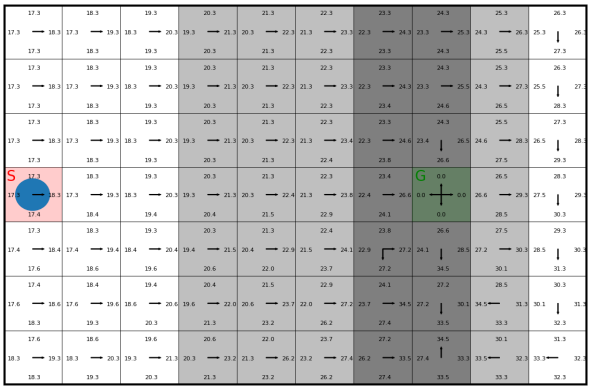

Figure 5. Q-value matrix for dynamic programming, after 17th (last) iteration

that with 1.0 probability agent will remain there, this allows to prevent "system hacks" when agent will go to neighbour cell and come back to the terminal state to gain reward again, in this case the loop will never end. Another way would be having a flag (boolean variable) if reward was already collected, this will not prevent agent from further exploration, but will teach agent not to move anywhere from there (due to the negative extra movement reward).

3. Exploration

3.1. Methods

Starting from this point the actual model-free reinforcement learning is involved, since transition function is no longer accessible and we can't use the transition function. This is why multi-armed bandit problem of exploration vs exploitation appears which means that it is not desirable to get stuck in local optimum using greedy methods, in the same time we want to explore to get some more optimal policies. This trade off is not trivial and various methods exist to tackle the problem. After choosing an algorithm it is crucial to pick hyperparameters for the exact problem.

The Q-function update policy is presented on Formula 6 (abbreviations are the same as were declared above).

$$G_t = r_t + \gamma \cdot max_{a'} \hat{Q}(s_{t+1}, a'))]$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t))]$$

Figure 6. Q-learning update rule

3.2. ε-greedy

ϵ-greedy policy is kind of a greedy policy with an exploration ability. $\epsilon \in [0, 1]$ in extreme cases when $\epsilon = 0$ it is just a greedy algorithm and when $\epsilon = 1$ it is just universally distributed random choice from all possible actions.

$$\pi(a|s) = \begin{cases} 1.0 - \epsilon, if a = argmax_{b \in \mathbb{A}} \hat{Q}(s, b) \\ \frac{\epsilon}{|\mathbb{A}|}, otherwise \end{cases}$$

Figure 7. the ϵ-greedy policy

3.3. Boltzmann (softmax) policy

Boltzmann policy uses softmax function which is pretty intuitive since it scales function values to probabilities which are easier to be interpreted by humans. $\tau \in (0, \infty)$, if $\tau \to \infty$ then we will be dividing numbers on infinity which will give 0, this is why choosing between 0 will give us uniformal distribution which is actually a random walking. Similar situation would be if τ approaches to 0, the only difference is that in this case the values would explode. In addition, it gives flexibility to choose similar options with similar probabilities which ϵ-greedy policy does not provide. However, there are some disadvantages of softmax, for instance if all values have large scale and do not differ a lot from each other - the probabilities would be almost same which will lead to random walking.

$$\pi(a|s) = \frac{e^{\hat{Q}(s,b)/\tau}}{\sum_{b \in \mathbb{A}} e^{\hat{Q}(s,b)/\tau}}$$

Figure 8. Boltzmann (softmax) policy

3.4. Hyperparameters experiment

Since both ϵ-greedy and Boltzmann policy algorithm depend on their hyperparameters it was decided to make an experiment to find their optimal values for the given problem. The used grid is following: $\epsilon \in [0.01, 0.05, 0.2]$ and $\tau \in [0.01, 0.1, 1.0]$. The results can be observed on Figure 9. Initially 50k steps were made, but it was observed that approximately after 12k steps the values converge and do not provide additional information, this is why only first 15k steps period was investigated.

Since the problem environment is pretty simple: not huge state space (which could be even brute-forced) and does not have a lot of local optimums, greedy policies work better which can be observed on the plot. The interesting observation is that ϵ-greedy algorithm with $\epsilon = 0.2$ converges on much lower score probably because of high random walking

which decreases the performance.

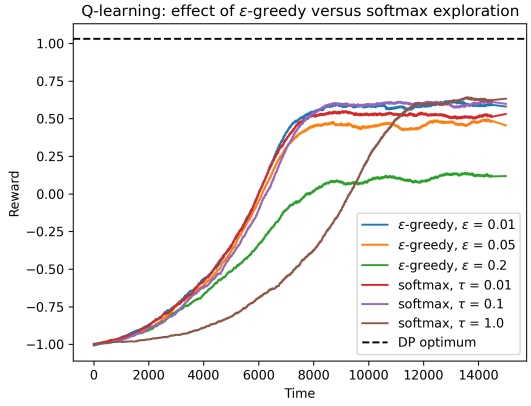

Figure 9. Reward values for Q-learning with different ϵ and τ within 15k steps averaged over 50 runs with $\gamma = 1$

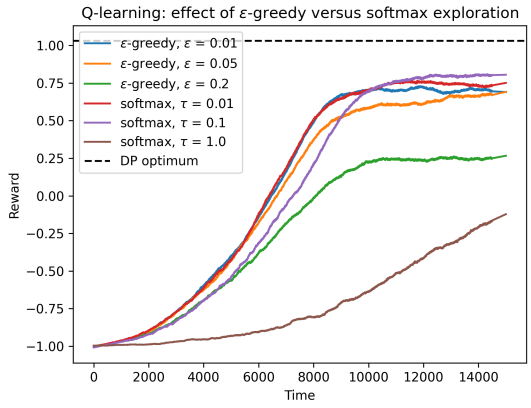

Figure 10. Reward values for Q-learning with different ϵ and τ within 15k steps averaged over 50 runs with $\gamma = 0.85$

In addition, it was decided to experiment a bit with γ to get use of late reward decay. Results of which are shown on Figure 10. The change of γ parameter dramatically influences the performance increasing average reward per step on 15%. However the convergence was achieved a bit later but it is definitely worth it.

It is seen that dynamic programming overperforms reinforcement learning, while first one achieves average score per step = 1.03, 1-step Q-learning achieves around 0.8 (can be tuned more, but trend is clear). It can be explained because Q-learning is model-free, in theory some annealing/smart learning rate changing policy should achieve similar results as dynamic programming, but our solution probably finds an local optimum and converges there.

4. Back-up: On-policy versus off-policy target

4.1. Methods

In this section 1-step on-policy (Q-learning) and off-policy (SARSA) methods were compared. The update function for Q-learnign was already presented (Formula 6). For the SARSA formula (Formula 11) is similar but instead of taking next state's maximum state-action value we are selecting an action from the current policy which is not always optimal, but gives more exploration.

$$G_t = r_t + \gamma \cdot \hat{Q}(s_{t+1}, a_{t+1}))]$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t))]$$

Figure 11. SARSA update function

4.2. SARSA vs Q-learning experiments

In this experiment SARSA was compared to 1-step Q-learning with couple of various hyperparameters like ϵ ($\epsilon \in [0.05, 0.2, 0.4]$) and γ ($\gamma \in [0.8, 0.9, 1.0]$). As in Q-learning experiment smaller γ gives a boost in performance it is observed on Figures 12 and 13. In addition, due to the convergence we again look only at the first 15k steps. Both algorithms act pretty similar the difference is that SARSA achieves slightly better score while Q-learning converges a bit faster. This is the case because we have a simple environment without local optimums, if environment had more "traps" on policy method would perform better because of higher exploration rate. However in convex optimization plane (environment) Q-learning would converge faster achieving similar (or even same performance) which in some case scenarios can be more valuable.

5. Back-up: Depth of target

5.1. Methods

In this part n-step and Monte Carlo methods are examined. There are 2 modifications of n-step methods namely n-step Q-learning and n-step SARSA, the difference between them is the method of action bootstrap. In further experiments n-step Q-learning was used, however n-step Q-learning can not be considered as an off-policy method since it samples first n rewards from the policy. The difference between 1 and *n* step Q-learning is the "depth" of the reward for Q-value calculation (Formula 14).

The second observed method is Monte Carlo update Formula 15. Which is a random walking with a reward decay. This approach is naive even for simple environment as Stochastic Windy Grid world.

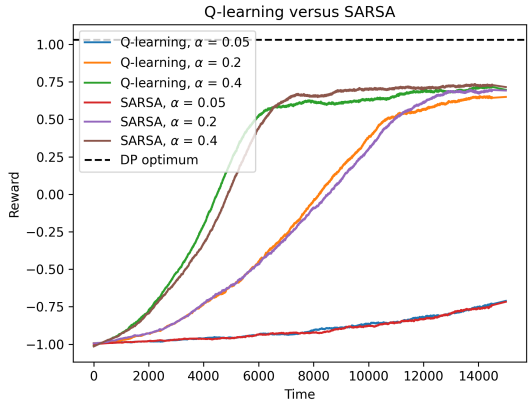

Figure 12. Reward values for SARSA with different ϵ and τ within 15k steps averaged over 50 runs with $\gamma = 0.85$

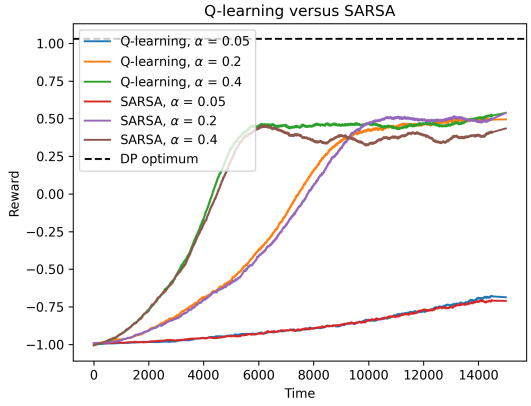

Figure 13. Reward values for SARSA with different ϵ and τ within 15k steps averaged over 50 runs with $\gamma = 1$

$$G_t = \sum_{i=1}^{n-1} (\gamma)^i \cdot r_{t+i} + (\gamma)^n \cdot max_{a'} \hat{Q}(s_{t+n}, a'))]$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t))]$$

Figure 14. n-step Q-learning update rule

$$G_t = \sum_{i=1}^{n-1} (\gamma)^i \cdot r_{t+i} + (\gamma)^n \cdot max_{a'} \hat{Q}(s_{t+n}, a'))]$$

$$\hat{Q}(s_t, a_t) \leftarrow \hat{Q}(s_t, a_t) + \alpha \cdot [G_t - \hat{Q}(s_t, a_t))]$$

Figure 15. Monte Carlo update rule

5.2. Monte Carlo vs n-step experiments

In this experiment the goal was to observe how different n ($n \in [1, 3, 5, 10, 20, 100]$) influence n-step Q-learning and how it can be compared to Monte Carlo method. The results for γ equal to 1 and 0.95 can be observed on Figures 16 and 17 correspondingly. The distinguishing feature of these plots is the fact that n-step Q-learning converges with higher variance. Even though 1-step converges longer it achieves better performance than other n. The intuition behind that is since when we dive deeper we get a bigger amount of future rewards and positive rewards propagates faster which leads to quicker learning. Nevertheless, when depth is high - variance starts to soar which leads to convergence on lower reward function values.

In addition, Monte Carlo method was not able to learn the pattern and is equal to random walking. This can be explained by the fact by the time agent gets to reward the path is already huge and it is impossible to gain any pattern from it. Taking into the account that Monte Carlo method did not work out in this simple environment it can be assumed that it is not going to work well in others as well.

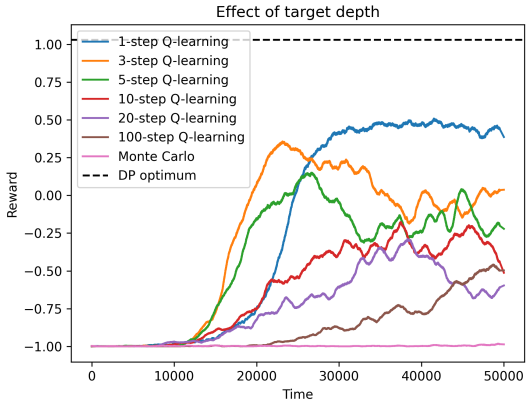

Figure 16. Reward values for n-step Q-learning and Monte Carlo method within 50k steps averaged over 50 runs with $\gamma = 1$

6. Reflection

6.1. Dynamic Programming vs Reinforcement Learning

Dynamic Programming is a well known method for finding optimal solution in model-based environments. However, in some tasks the search space is huge and it is computationally

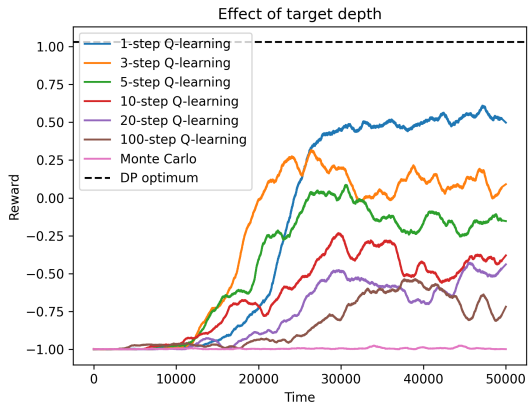

Figure 17. Reward values for n-step Q-learning and Monte Carlo method within 50k steps averaged over 50 runs with $\gamma = 0.95$

very complicated to get the optimal solution. Even though there are some methods like the branch and bound method or simply constraining the recursion depth that allow to reduce the complexity, but to refuse from guarantee of optimal solution. Moreover, in many cases we do not have an access to the transition model and can not apply DP method. In addition, computational complexity creates latency which in some cases can be crucial. This is when reinforcement learning approach becomes handy, it has variety of method for different cases and environments (like model-free/based, on/off-policy, policy/model based agents) even though it is not always giving optimal solutions and often good solutions are hyperparamter dependent and require some time for training and experimenting.

6.2. Exploration

During experiments it was shown that both ϵ-greedy and Boltzmann (softmax) policy converge and in this particular environment give similar scores out of which low exploration rate showed better results. Choosing in between the methods softmax might be potentially better because it higher variability and the exploration it provides is more conscious since action probabilities are proportional to their Q-value. In addition, softmax would provide more ”natural” agent behavior because it would behave differently in situations when the decision is not crucial and efficient when the next action is important, unlike ϵ-greedy which will behave either in the same way or randomly.

There are some ways to improve the exploration policy like normalizing softmax or even use combination of ϵ-greedy and softmax making greedy move with some probability (not to high) or otherwise choose using softmax. Also the learning rate should be changed dynamically using various schedule not only linear. For instance, cosine (or other periodic function) change of exploration rate which will allow to explore first, then to exploit to find an optimum and after increasing to high exploration rate again algorithm will be able to escape from local optimum to find a better solution.

6.3. Back-up, on-policy versus off-policy

Both on-policy and off-policy have their own advantages and disadvantages. Q-learning is an off-policy method which means that action selection via policy and policy update are different. The updates are using the maximal potential reward which may lead to faster convergence but higher risk of getting stuck in local optimum.

SARSA is an on-policy method which means that updates are using same policy as action choice. Due to higher exploration rate SARSA is able to find more optimal solutions however it takes a bit longer.

In general, if time restriction is not too strict SARSA is more recommended to classical Q-learning since it might discover better policy in an environment.

For n-step methods Q-learning is a mixed-policy since it takes n-steps via on-policy and on the last step off-policy will be applied.

6.4. Back-up, target depth

Talking about n-step and Monte Carlo approaches bias/variance tradeoff of target values occurs. Q-value based methods make their next moves taking reward into the account for each state in the episode. N-step method has bias due to the fact that the state-action values are approximation of value function. N-step approach might be more useful in more complicated environments, because in this one 1-step Q-learning converges almost as fast but gives significantly better results. This is why for this task 1-step Q-learning is the preferred algorithm. Higher n converge faster because the future rewards are perceived by an agent quicker (propagates faster) but this additional exploration can limit the learning.

Monte Carlo method has low bias and high variance because a random complete trajectory is generated and evaluated only in the end. High variance is because of the affect of all rewards of an episode. Because it is random, the reward propagation is inefficient which leads to terrible performance.

Non of the methods converged to optimal policy, but 1-step Q-learning was the closest one.

6.5. Curse of dimensionality

In the provided environment tabular reinforcement learning was used because there is not so many states and actions: 10 x 7 board and 4 moves in each. All this fits into memory and we are able to ”remember” every action which works best in each state. In addition tabular reinforcement learning is flexible because it allows to work with both discrete and continuous problems. In large problems - it is no longer possible this is why some other kind of representation is required where states are not individual like for instance embeddings in neural networks. Also since the number of states growth exponentially then the amount of observations should grow accordingly which is often impossible. However, neural networks have problems with interpretability and for simple tasks can cause lack of generalization since model will be able to remember all the provided (trained) states.