



Universiteit
Leiden

Master Computer Science

Historical Information for Enhancing Q-learning in Maze Navigation Tasks

Name: Zhao Yang
Student ID: s2191989
Date: 30/07/2020
Specialisation: Data Science: Computer Science
1st supervisor: Mike Preuss
2nd supervisor: Aske Plaat

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Historical Information for Enhancing Q-learning in Maze Navigation Tasks

July 30, 2020

Abstract

When people are facing problems, they first search in their memories and try to find out useful information, or sometimes they ask help from others who might know the solutions. Here, people are using the knowledge they or others have learned before for solving problems. This historical knowledge could largely accelerate people's solving process and enhance their performance when they are in any task. In this project, we are inspired by this idea and examined two different historical information under reinforcement learning(RL) settings and more precisely in maze navigation tasks using two RL methods Q-learning and its variant Deep Q-learning. The goal of the task is to control the agent in the maze to learn to find the optimal path from the start to the target. We believe historical information could also have positive impacts in our context as it does in real life. Historical information I is the record of visited cells, when we have this record, whenever the agent reaches cells which are in the record, a penalty will be assigned to the agent. Historical information II is the guide from the expert, the expert here knows the optimal path from the start to the target. These two different types of historical information are added to games and we experimentally proved that they help enhance these two RL methods to some extent and further make the agent reach the target way faster and stronger than the agent in normal games. Besides, in using historical information I, we studied how to set the optimal penalty which could make the agent the best performance. We trained a classifier that could predict the optimal penalty for unseen mazes and the accuracy could be up to 80%. Also, everything we developed is open-sourced for further study.

Acknowledgements

I first would like to thank Mike Preuss for his patient guides and supervision. He taught me a lot about doing research, adjusting mindset, and staying curious about knowledge. He also encouraged me a lot when I was in trouble with my experiments. I really appreciate his help and encouragement. Also, I want to thank Aske Plaat for bringing me to RL field and his instructive supervision, I was impressed with his strong passion for RL. He gave me many crucial instructions and hints before I started, and these instructions played important roles in my whole thesis. And I really appreciate Thomas Moerland for giving me a lot of instructions for reading papers and gathering information. Finally, I want to thank my family and friends for supporting me. I could not have done this work without their encouragements and supports.

Contents

1	Introduction	5
1.1	Motivation	6
1.2	Research Questions	6
1.3	Contributions	6
1.4	Structure of the Paper	7
2	Background	7
2.1	Elements in RL Settings	7
2.2	Markov Decision Process	8
2.3	Q-learning	10
2.4	Deep Q-Learning	12
2.5	Mazes	14
2.6	State-of-the-art Techniques	14
3	Methods	15
3.1	Mazes Generating	16
3.2	Environment Implementation	17
3.3	Q-Learning	18
3.4	Deep Q-Learning	19
3.5	Historical Information Implementation	20
4	Experiments	21
4.1	Experimental Settings	23
4.2	Experimental Design	23
4.3	Experimental Results	24
4.3.1	Results on Historical Information I	24
4.3.2	Results on Historical Information II	26
4.3.3	Trail and Error	29
5	Conclusion and Discussion	31
5.1	Conclusions	31
5.2	Limitations and Future Works	33
	Appendices	36
A	Used Mazes	36
B	All Experimental Results for Historical Information I	40

1 Introduction

Reinforcement learning (RL) has been successfully applied to different fields such as board games [1], video games [2], robotics [3], recommendation systems [4], and logistic planning[5]. Different from supervised learning and unsupervised learning, training RL methods does not need off-the-shelf data but collecting data on the fly from the environment. More accurately, RL methods learn how to behave by interactions with the world, which is similar to the way a baby learns to walk. He falls then knows that falling will hurt, and he will try to adjust the walking style to avoid falling again. He keeps falling while keeps adjusting until he could walk steadily.

After AlphaGo [1] shocked people from all the fields. Recently, DeepMind¹ proposed AlphaStar [6] and Agent57 [7] destroyed human-beings' performance in the real-time strategy game StarCraft² and the retro 90's video games platform Atari 2600³ respectively. StarCraft is even a complex game for human-beings and professional players could perform 200-300 actions per minute, more than 100 units could be controlled for various tasks. The great complexity indicates the action space and the state space are huge, and games not only require short-term fast reactions but also long-term planning which all cause difficulties. And Atari 2600 has been used as the benchmark for artificial intelligence(AI) methods since several years ago, it contains a lot of simple well-known games like Breakout, Pac-man, and Pong, also some games like Pitfall or Montezumas Revenge are quite challenging and make previous AI methods keep failing. However, the aforementioned works from DeepMind show how powerful and how strong that AI could be. The core idea of these amazing techniques is reinforcement learning. Although there is no need to feed data to RL methods, they need millions of interactions with the environment which is ridiculously expensive and sometimes even impossible. Some researchers estimate it costs around \$35 million⁴ in computing power to reproduce the experiments reported in the AlphaGo Zero [8] paper and even a simple parking task needs to be trained more than 300k attempts⁵. Thus, researchers sometimes utilize historical information such as the knowledge learned from previous tasks or other related tasks in the current reinforcement learning problem to achieve acceleration and generalization, this technique is also called transfer learning. Here, historical information could be previous related experiences, helpful hints or external guides, etc.

Maze navigation tasks are entry-level tasks for AI algorithms. It is simple enough for beginners but also could be added complexities and difficulties by modifying the environment or redesigning rules. The basic goal of maze navigation is to control an agent to walk from the start point to the target point under some restrictions like the number of steps the agent could be taken or a given time limit.

In this project, we use mazes navigation tasks as the demonstration to show the achievement of acceleration and enhancement for Q-learning including both original Q-learning and Deep Q-learning by utilizing different historical information. Q-learning is one of the most popular and straightforward reinforcement learning methods. It aims for learning an optimal policy that

¹<https://deepmind.com/>

²<https://en.wikipedia.org/wiki/StarCraft>

³<https://en.wikipedia.org/wiki/Atari>

⁴<https://www.yuzeh.com/data/agz-cost.html>

⁵https://www.youtube.com/watch?v=VMp6pq6_QjI

could guide the agent's behavior under certain situations.

1.1 Motivation

Imagine we, as real people are in a maze now, we will start exploring again and again until find the target. Meanwhile, since revisiting visited paths is a waste of time and energy, we will also try to remember paths we passed by for avoiding to revisit them. Of course, the optimal strategy for reaching the target is every time we are exploring, the path we walk along is an entirely new path, which means we make the best use of the exploration. Sometimes, it is even possible that we ask for help from people who might know the path towards the target. And finally, we will reach the target faster compared with just exploring unreasonably. From the examples above, we could know that humans can utilize historical information like the memory of visited paths or guides from others for better performance. We believe such historical information will be helpful for navigation or planning. Thus, we want to add such historical information to the agent in maze navigation tasks and to see if they work or not.

1.2 Research Questions

In this project, we will introduce different historical information in mazes navigation tasks by the use of Q-learning and try to answer the research questions below:

- **If we record visited paths of the agent and give penalties to the agent when it tries to reach such paths again, will the agent learn to avoid visited paths and then find the target faster? If yes, how can we set optimal visited penalties that could optimize the performance of the agent in unseen mazes?**
- **If we use an expert to guide the agent, will the agent perform better?**

So here, we say the record of the visited paths of the agent is historical information I and guides from the expert are historical information II. Questions above might seem easy to answer for humans, but for a machine or a specific algorithm could be hard. The key point is that we have to utilize historical information in the fashion that could be understood and learned by the agent itself, instead of thinking it from an entirely human's perspective or intuition.

1.3 Contributions

We list our contributions in this project below:

- Implemented the maze environment and the maze generator which could both be used for further study;
- Proved giving penalties to the agent when it reaches cells which are in the record(historical information I) is helpful in our context experimentally and further trained a classifier for predicting the optimal penalty for unseen naive mazes;
- Proved guides from the expert(historical information II) are helpful for the agent's performance experimentally and showed that giving a different amount of guides to the agent will cause different results;

- Did some trails and results are not what we expected, then we realized it is important to design experiments in a fashion that could be understood by the algorithm we are using;
- Gave intuitive understandings and interpretations of experimental results and pointed out potential future working directions;
- Guaranteed that all the experiments we did are repeatable and all the codes we wrote are open-sourced.

1.4 Structure of the Paper

In Section 2, we will introduce some preliminary knowledge about reinforcement learning and mazes, also some state-of-the-art works have already done in the related direction that we are interested in; details of the ideas we come up with and methods we are using will be explained explicitly in Section 3; the design of experiments and experimental results will be given with interpretations in Section 4; we will conclude our project and discuss drawbacks and future works in the last section. More experimental details of results are listed in Appendices.

2 Background

Reinforcement learning problems can be formed into MDP expressions, and the way to solve MDP problems is to maximize the accumulated rewards. Q-learning is one of the popular RL methods, it is implemented by a look-up table called Q-table and recursively updates policy towards optimal. When the state space is large, maintaining the Q-table will be expensive then a deep neural network is used instead as a Q-value approximator which could deal with huge state space and be able to generalize to more tasks. In this section, we will give brief introductions and explanations of basic reinforcement learning conceptions such as MDP, Q-learning, Deep Q-learning. Then we will describe the mazes we used in our project and following with some state-of-the-art works have already done in navigation tasks or maze-like games under RL settings.

2.1 Elements in RL Settings

In RL settings, we train an agent to learn a policy to act via interactions with the environment. Each step the agent takes is an action, and each interaction contains feedback the environment gives to the agent for this certain action. We will introduce the needed terms below.

Agent The Agent is the decision-maker or the main actor in games or tasks. It is the object who takes action and gets rewards. It could be the character or the player in games or the machine in industrial tasks such as a robotic arm or a vehicle.

Environment The Environment is the place or space where the agent could interact with and live in. It provides the rules of interactions and feedback to the agent. It might be a game simulator or the area where the task performs.

State The state, denoted as s , is a specific condition of the environment after the agent takes a certain action, it is determined by the previous state and the action the agent takes and contains information about what is going on with the environment and the agent currently. It could be the state of the game simulator at a specific moment.

Action The action, denoted as a , is the step or behavior the agent takes every time. It could be going up in a navigation task or turning the arm 10 degrees to the right in a robotic arm task, etc.

Reward The reward is the value the agent will get from the environment after taking a certain action. It could be both positive or negative, which refers to reward and penalty respectively. It could act as the pain when you fall and the candy when you walk steadily. In RL settings, the agent aims to maximize the overall reward during the task.

Policy The policy, denoted as $\pi(a|s)$. It maps states s to actions a , for the agent in a certain state, the policy can tell the agent how to behave actions. Furthermore, the optimal policy will tell the agent optimal actions which could lead the agent to reach the target faster or to have the largest accumulated reward. It could be strategies or guides in games.

State Transition Probability The state transition probability is the mathematical representation of the transition between different states. It tells the probability of the transiting from a current state to another successor state after performing a certain action. It is usually represented by a matrix, p_{nm} refers to the probability of the transiting from state n to another state m .

$$\begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \\ p_{41} & p_{42} & p_{43} & p_{44} \end{bmatrix}$$

Exploration and Exploitation For RL methods, we don't have off-the-shelf data for training but collecting information on the fly. The agent has to explore the environment to collect more data, but if it explores too much and never exploits the knowledge that has already been learned, the performance will not be improved. So the agent should do both explorations and exploitation and the key point is when to do what. The trade-off between explorations and exploitation will further affect the training process and the agent's performance. It is still an open question in research fields. ϵ -greedy, Upper Confidence Bound [9] and Thompson Sampling [10] are mostly used strategies for dealing with explorations and exploitation trade-off.

2.2 Markov Decision Process

Markov Decision Process is short for MDP, it can formally represent reinforcement learning problems and could be used as a framework to describe the process that the agent learns how to act to reach the final goal via interacting with the environment. Our goal is to maximize accumulated rewards or reach the final state. As the Fig. 1 shows, the whole procedure starts from the action(A_t) that the agent performs in the environment based on the current state(S_t),

and then the environment will output the new state(S_{t+1}) and give the reward(R_{t+1}) to the agent for the certain action it takes, now the agent observes the new state(S_{t+1}) and could take next action based on the new state. Then the environment will output the next new state based on the action the agent just takes, and the procedure goes back to the beginning. So the agent will repeat this procedure until the final state is reached or the restriction is met.

Markov decision process is the extension of Markov process/Markov chain, and they both satisfy Markov property. Bellman equation could be used to solve MDPs problems. We will start from Markov property, then explain Markov process and Bellman equation, and the way we form our problem into MDP expressions will be given in the end.

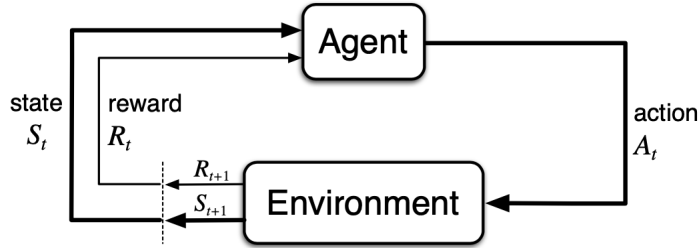


Figure 1: The agent-environment interaction in a Markov decision process.[11]

Markov Property A process has the Markov property if future states in the process only depend on the current state instead of historical states. The example is shown in Fig. 2, if the weather tomorrow only be decided by today's weather regardless of the weather of the day before yesterday, then we could say this weather report has Markov property.

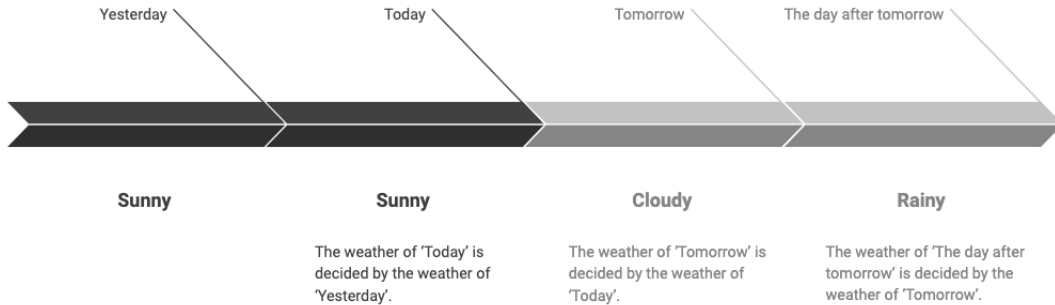


Figure 2: An example of Markov property, the weather of the next day only depends on the weather of the day before.

Markov Process Markov process is a sequence of states which all have Markov properties. It is also known as the Markov chain. The weather report in Fig. 2 could be treated as a Markov process. The weather of the day only determined by the day before and this property suits every day. In reinforcement learning settings, a Markov process can be written as a 2-tuple, $\langle S, P \rangle$. S is a sequence of states (s_1, s_2, \dots, s_n) and P is the transition probabilities $(p_{11}, p_{12}, \dots, p_{nm})$

which indicates the probability of transiting from one state to another state. The dynamic transition is shown as follows:

$$p_{n,n+1} = Pr(S_{n+1} = s_{n+1} | S_n = s_n)$$

Bellman Equation Bellman equation is a method for optimization problems and is widely used in reinforcement learning, economics, and control theories, it breaks a whole optimization problem into several simpler sub-problems recursively. For example, in MDPs, value function $V(s)$ indicates how good the agent to be in a certain state, which is the expected overall future reward. It could be formalized as:

$$V(s) = \mathbb{E}[\gamma^0 r_{t+1} + \gamma^1 r_{t+2} + \gamma^2 r_{t+3} + \dots | S_t = s] = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s\right]$$

Now we use Bellman equation to rewrite the value function:

$$V(s) = \mathbb{E}[r_{t+1} + \lambda V(S_{t+1} | S_t = s)]$$

r is the reward that the agent gets when the agent move from one state to another state; γ is the discount factor, the smaller γ will emphasize closer rewards and larger γ will take more further rewards into account. The Bellman equation breaks the original value function into a function that is expressed by the value function of the next state and the reward. Recursively, the value function could always be expressed by the next state, thus each state perfectly satisfies the Markov property. Bellman equation simplifies the whole problem and still keeps the problem satisfying the Markov property.

MDP Markov decision process(MDP) is represented as a 5-tuple, $\langle S, A, P, R, \gamma \rangle$. S is a sequence of states; A is a set of actions; P is transition probabilities; R is a sequence of corresponding rewards and λ is the discount factor. In our project, the agent is the player that tries to start from the entry and navigate to the target in the maze. S is the maze with the agent located at different positions; A represents actions that could be taken in the maze, they are going up, going down, going left and going right; P is the probability of the state transition after the agent act, in our project, it's always 1 because after the agent takes an action it will always go to another certain state; R are rewards we set, they could be negative penalties or positive final rewards; λ we are using is 0.1 which is the most commonly used value.

2.3 Q-learning

Q-learning is a reinforcement learning algorithm that can be used to find out the optimal action under a certain state or namely the optimal policy. Q refers to 'quality' and generally, this algorithm is implemented through a look-up table termed Q-table. The rows of the table are 'states' while the columns are 'actions'. Each element of the Q-table is a Q-value, it indicates the reliability of taking a specific action under the specific state, the higher value the more convinced this action is and mathematically represented as:

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

, which is also called state-action value. It measures the expectation of overall future rewards from the current state to the terminal state. The update rule for Q-values in the table is:

$$Q'(s, a) = Q(s, a) + \alpha[R(s, a) + \gamma \max_{a'} Q'(s', a') - Q(s, a)] \quad (1)$$

α is the learning rate, and Q' and Q are Q-values of next state and current state respectively. Q-table looks like Fig. 3. The pseudo code of Q-learning is shown in Alg. 1.



Actions States	↑	↓	←	→
	0.01	0.01	0.05	0.01
	0.2	0.02	0.01	0
...

Figure 3: How the Q-table looks like, rows are states while columns are possible actions, and every entry is the Q-value for each action under the certain state.

Algorithm 1: Q-Learning

Result: Output convinced actions

Initialize Q-table arbitrarily;

for *each training episode* **do**

 state s = starting point;

while *game is not over* **do**

 choose an action a under ϵ -greedy policy;

 take the action a and observe reward r and next state s' ;

 update Q-table: $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$;

$s = s'$;

end

end

look up the Q-table and get the optimal action;

After the Q-table finishes updating, we can utilize the information in the table. Whenever the agent takes action, the Q-table will be looked up and the action which has the maximal Q value will be taken.

ϵ -greedy policy It is a method to trade-off between explorations and exploitation for the agent. By tuning ϵ ranged from 0 to 1, the agent will either choose the optimal action according to the Q-table with a probability $1-\epsilon$ or choose a random action with a probability ϵ . It works like Fig. 4. First, we generate a random number r ranged from 0 to 1, and compare r with ϵ we set, if r is larger than ϵ , we choose the optimal action for the agent according to the latest Q-table; while r is smaller than ϵ , a random action will be taken by the agent.

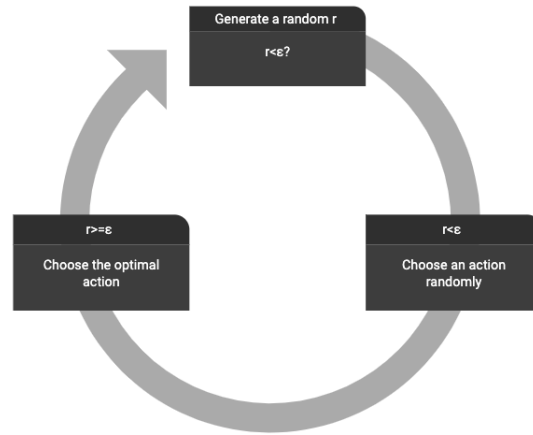


Figure 4: ϵ -greedy policy in choosing actions, if the random number is larger than ϵ the optimal action will be chosen, and if it's smaller than ϵ the random action will be performed.

2.4 Deep Q-Learning

If we use Q-learning described above, when the state space is large, the Q-table may not be compatible. The size of the table will be out of memory soon, in order to overcome this problem, DeepMind⁶ proposed a new paradigm combined with a deep neural network called Deep Q-learning [12] which are much powerful and much easier to be generalized. There is no table for Deep Q-learning, neural networks are used instead as a Q-value approximator to produce Q-value for each state. The difference between Q-learning and Deep Q-learning is shown in Fig. 5. It could be simply seen as that a neural network is used to replace the Q-table to output the optimal action. Deep Q-learning uses images as input, adjacent frames will be highly correlated, which will cause instability during the training. Experience replay technique [13] is introduced to guarantee that data for training is independent and identically distributed and it also could improve the sample efficiency. Hundreds of older transitions are put into a buffer and a mini-batch will be sampled every time randomly for further training. See details of Deep Q-learning in Alg. 2.

First, the weights of the Q-network will be initialized arbitrarily. For each training episode, the game simulation is performed. During the simulation, the agent acts under ϵ -greedy policy based on the current state. After every action, the environment will produce the reward and the next state. Then the current state, the action the agent takes, the reward the agent gets, and the next state will be packed together to an item and be stored into the experience replay buffer. Then a mini-batch of items will be sampled from the replay buffer to train the neural network. Weights of the neural network are optimized by minimizing the loss between two Q-values which are both produced by the Q-network, but one is calculated using the current states from the mini-batch as the input and another one is calculated using next states from the mini-batch as the input and then plus rewards that the agent will get from this transition. Weights will converge after several iterations. After finishing updating, the Q-network will be utilized for producing Q-values of actions by using different states as the input, and the action

⁶See more on <https://deepmind.com/about>

which has the maximal Q-value will be taken by the agent.

When it comes to neural networks, they generally consist of several different components, such as layers, activation functions, loss functions, optimization methods, etc. For each component, there are a lot of choices shown in Tab. 1. Layers are most important components of a neural network, it functions as feature extractor, features can be used for further classifying or other artificial intelligence tasks; activation functions aim to introduce some non-linear factors to the network to be able to express more complex relations between inputs and outputs; loss functions are measurements of how close the network's performance compared to the ideal performance is; optimization methods are used for updating neural network weights towards optimal and aim for minimizing the loss which is produced by the loss function. Different combinations over the aforementioned components might suit for different tasks and situations. Convolutional layers could capture visual features, and it suits for images or other visual tasks like subjects recognition; the linear layer is also called a fully connected layer, it mostly be used for regression tasks, also could be used for simple classification tasks, or embedded on the top of a neural network for classifying; recurrent layers use historical information to catch relationships between different sequences, it's good at processing context-sensitive text data.

Layers	activation functions	loss functions	optimization methods
Convolution	ReLU	MSE	Adam
Linear	sigmoid	cross-entropy	SGD
Recurrent	softmax	L1	Momentum
...

Table 1: Choices of different neural networks components, there are layers, activation functions, loss functions and optimization methods.

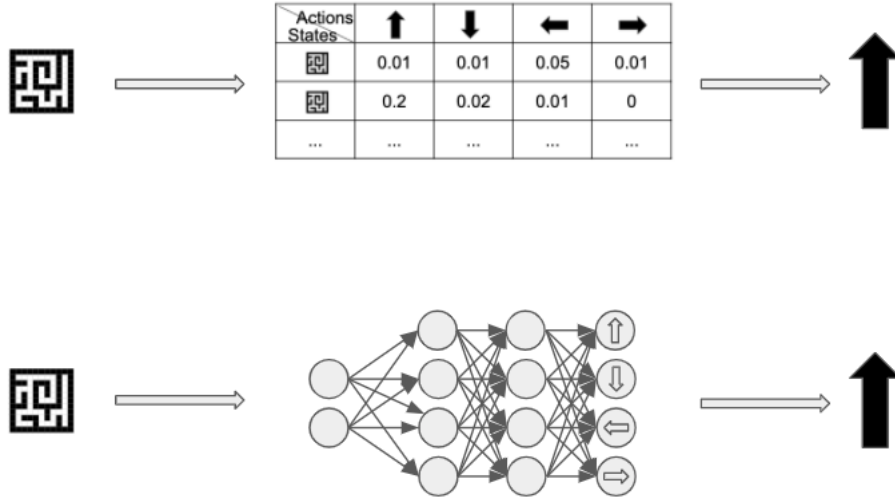


Figure 5: Q-learning and Deep Q-learning, Q-learning uses a Q-table for storing information and producing optimal actions while Deep Q-learning uses a neural network for catch information and producing optimal actions.

Algorithm 2: Deep Q learning with Experience Replay

Result: Output the optimal action

Initialize neural networks weights arbitrarily;

for *each training episode* **do**

 Start simulating one game;

for *each step in the simulation process* **do**

 play the game under ϵ -policy for one step;

 item = [current_state, action, reward, next_state];

 replay_buffer.insert(item);

 data = get_mini-batch(replay_buffer);

 Q_target = DQN(current_states in data);

 Q_next = DQN(next_states in data);

if *the game is not terminated* **then**

 Q = reward + γ * max(Q_next);

else

 Q = reward;

end

 loss = loss(Q, Q_target);

 Minimize loss and update weights of DQN ;

end

end

Plug the state into the neural network to get the optimal action;

2.5 Mazes

A good maze doesn't contain loops and has as many dead-ends as possible, which means all the areas within the maze have been used effectively. Mazes we generated in this project look like Fig. 6. Black cells are blocked cells that are not supposed to be visited by the agent, and white cells are free cells that represent paths. In our case, the most top-left cell of the maze always be set as the start point and the most bottom-right cell is set as the target. Each time the agent can only take one action which is one of the actions in the action space (going up, going down, going right, going left), and the win only is counted when the agent finds the pathway from the start to the target in the end.

Size of mazes, number of dead-ends, length of the shortest path, and distribution of valency (the number of ways in and out of a cell) are commonly used to measure the complexity and difficulty of a maze. In practice, it will depend on situations we are facing.

2.6 State-of-the-art Techniques

Mazes can be easily solved by classic algorithms such as well-known depth-first and width-first search algorithms, Pledge algorithm, and so on. By comparison, finding the solution using machine learning algorithms is more time-consuming and computation-consuming. However, researchers enjoy exploring more intelligent ways in the machine learning field for solving navigation tasks such as mazes, etc. Not only because it can be applied into several fields such as robotics and unmanned systems, but also helps human to better understand machines and

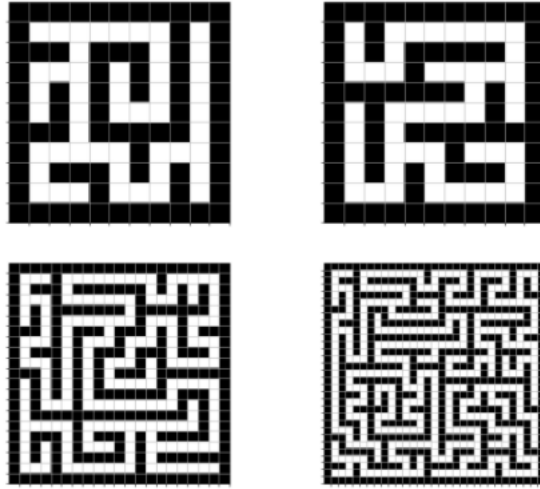


Figure 6: Different sizes of mazes we are using, they all are generated by our maze generator.

the way that machines work. Next, we will mention some state-of-the-art techniques which have been proposed in the gird-world navigation research field which looks simple but fairly complex in practice.

Value iteration network(VIN) [14] embedded a 'planning module' within a neural network, it could perform long-term planning in navigation tasks and also generalizes better to new navigation tasks; a successor-feature-based deep reinforcement learning algorithm [15] can learn from previous mastered navigation tasks and be adapted to new tasks quickly, it also substantially decreases training time after the training on the first task; Go-Explore [16] could tackle sparse reward in the navigation task, also during the exploration part it will first explore more convincing areas, which improves efficiency of exploring; curiosity is used to augment the normal rewards for training deep reinforcement learning methods [17] to improve performance in navigation tasks, it also has better generalization capabilities; based on the experience replay, interactive replay [18] is built by a single traversal of the environment which could be used for generating larger number of diverse trajectories for the training and get better performance. These techniques are improving the performance of the agent in maze-like navigation tasks from a different perspective, our ideas are from an entirely different side but have a similar positive impact on the performance.

3 Methods

In this section, we will dive into all the methods we used practically including how we generate our mazes, how we implement the experimental environment and algorithms we are using, etc. Since we have two research questions corresponding to two different historical information, we will use original table-based Q-learning to examine historical information I and answer the first research question, and use Deep Q-learning to examine historical information II and answer the second research question.

3.1 Mazes Generating

A lot of algorithms are using to generate mazes such as Aldous-Broder Algorithm [19] which uses random walks until all vertices are visited; Prim's Algorithm [20] which is a kind of breath-first search algorithm; Kruskals Algorithm [21] is the most complex one for implementing among algorithms we mentioned above, etc. The depth-first search also called Recursive Backtracking Algorithm [22] is used in our case. It is much understandable and the most common-used one also could generate mazes really fast [23]. It starts from a random cell of the grid where each cell has walls around and takes an action to a random direction which has not been visited before then it is in a new cell now, the next step is removing the wall between the current cell and the previous cell. Repeating this process until reaching a cell where all the directions have already been visited, then falls back to the predecessor cell and chooses another random action continuously. The maze will be finished when there is no predecessor cell can be fell back to. The pseudo-code is described in Alg. 3. A simple example is given in Fig 3.1. We start from **1**, there are two possible actions could be chosen, and we select **right** randomly. Now we move right to **2**, then remove the wall between **1** and **2**. Three possible directions could be taken now, they are **3,6,1** respectively, but **1** has already been visited before, then we chose from **3** and **6** randomly. Repeating this process until we are in **5** now, there is no unvisited direction that could be taken, so we return to the predecessor cell which is **9**, and we keep repeating and the maze will be finished when there is no cell could be returned to.

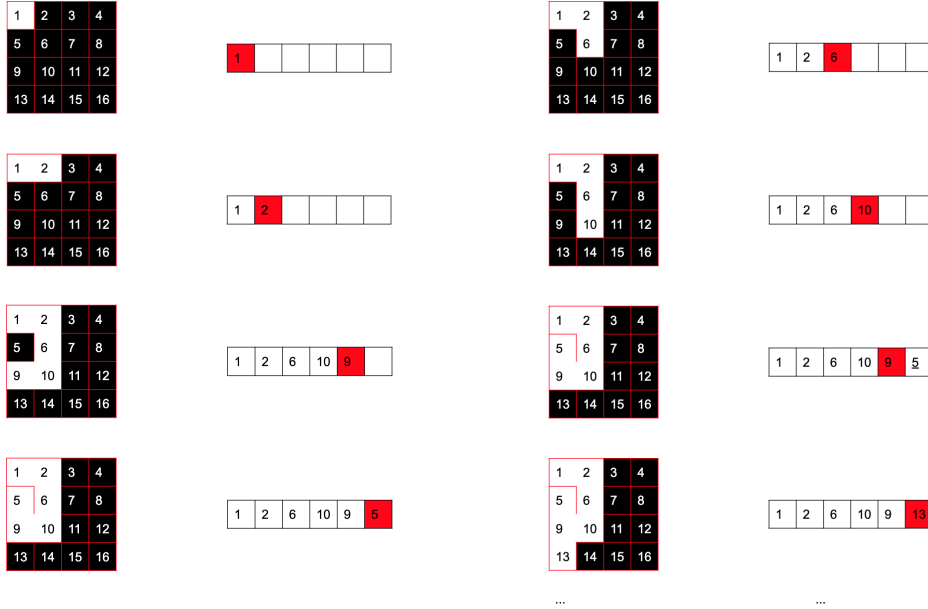


Figure 7: An example of recursive backtracking. Starts from left to right, top to bottom. Black cells are unvisited and white cells are visited, the red cell in the list is the indicator of the current cell in the maze.

Algorithm 3: Recursive backtracking algorithms

Result: Output the maze

current cell = choose a random cell and add it to the stack;

while *stack is not empty* **do**

 valid direction = choose an unvisited direction randomly based on the current cell;

if *valid directions is none* **then**

 current cell = stack.pop;

else

 next cell = take the valid direction;

 break the wall between current cell and next cell;

 current cell = next cell;

end

end

Maze Complexity To predict the optimal penalty for unseen mazes, we need to somehow build up the relation between different optimal penalties and different mazes. Thus, we need to measure the maze's complexity or difficulty properly. It depends on the way we solve it. If we solve it by eye looking, then parallel lines will make the maze more tricky; if we are using a pen, we will probably be trapped by intersections, etc. There are some common metrics for measuring the maze's complexity. The combination of the number of dead-ends and the number of intersections are used to measure complexities of mazes[24]; the length of the solution path, the number, and length of the branches and 'twistiness' of passages all could be used as measurements for the complexity [25]. Although there are plenty of ways to define the complexity of mazes, most of them are more designed from a human being's perspective and may not work for machines or algorithms. Namely, if a maze is difficult for humans, it's not always difficult for a certain algorithm. We need to think about it in an algorithm-specific fashion to decide measurements of complexities. The Q-learning will be using in our experiments, and the time complexity of Q-learning is $O(|S|^2|A|)$ [26], where $|S|$ is the number of states while $|A|$ is the number of actions. As such, when other contexts are fixed, the larger state space and larger action space will make the Q-learning 'feel' more difficult. Since our action space is fixed which is always four, the size of the state space will be the main determinant for the complexity of the maze. When it comes to the state space, it has a high correlation with the size of the maze which means a larger maze will have larger state space as well as longer shortest path, also the number of intersections will be larger. But they don't have perfect linear relationships since we generate mazes randomly, we will use all of these four metrics as measurements for the complexity of mazes.

3.2 Environment Implementation

The maze environment we are using here is implemented based on the work that Samy Zafrany did before⁷ and we make several adjustments and modifications. Mazes are square, there is only one agent in the game. Game rules and other detailed information are listed below:

- Each maze contains 3 different types of cells, free cells, a target cell and blocked cells;
- 4 actions in the action space, going up, going down, going left, going right, respectively;

⁷<https://samyzaf.com/ML/rl/qmaze.html>

- Every time the agent starts from the most top-left cell and the target is always the most bottom-right cell;
- 4 different types of penalties which the agent could get:
 - Step Penalty: The agent will get a step penalty for each step it takes for pushing itself towards to the target;
 - Invalid Penalty: When the agent is trying to reach invalid cells like blocked cells, it will get a strict invalid penalty for preventing itself from intentions of reaching such cells;
 - Target Reward: The positive reward will be given to the agent only when the agent reaches the target cell which is always the bottom-right cell in our case;
 - Visited Penalty(when historical information added): If the agent reaches cells which have already been visited before, it will get visited penalties, the value depends on which values we intend to examine.
- A training episode will be counted from the start of one game to the game is terminated. There are two possibilities that game is terminated:
 - The total reward of the agent gets reaches the minimum threshold we set, this can efficiently prevent the agent from falling into infinite hovering.
 - The agent finds the route to the target.

The details of the parameters of the environment shown in Tab. 2. The reason we set invalid penalties much larger than the step penalty is that invalid cells should be avoided strictly. The Reward could be any other positive value which only needs to be distinguishable with penalty values. For the threshold of the total reward, the value we set is $-0.5 * \text{maze.size}$. For example, if the maze we are using is 10×10 , so the threshold will be -50 , which allows the agent to take $50/0.04 = 1250$ steps when the step penalty is -0.04 . Meanwhile, the length of the shortest paths from the start to the target of mazes with the size of 10 we generated is around 25, and compare to 1250, the agent will have plenty of opportunities to find the right path.

Parameters	Values
step penalty	-0.04
invalid penalty	-0.75
target reward	1.0
reward threshold	$-0.5 * \text{maze.size}$

Table 2: Parameter settings for the environment.

In practice, a maze is represented as a array, where different digits represent different types of cells. The 'real' maze and the visualized maze shown in Fig. 8. The color map and corresponding representations shown in Fig. 9.

3.3 Q-Learning

As we mentioned before, we need to maintain a look-up table for storing Q-values and updating the Q-learning method. In the Q-table, possible actions the agent could take which are

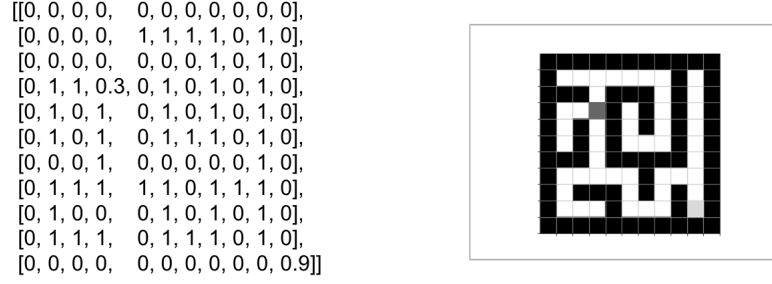


Figure 8: 'Real' maze and its visualization, 'real' maze is represented by an array.

columns of the Q-table are encoded into 4 integers, they are 0 to 3 respectively, and states will be hashed using `hash` function into a bunch of digits which will be rows of the Q-table. An example of hashing state into digits is given in Fig. 10, thus, in the Q-table, the states are represented by digits instead of 'real' mazes.

The parameters setting shown in Tab. 3. ϵ -greedy policy is used to select actions during our training process, we set ϵ to 0.1 which is the most commonly used value. It means during the training process, there is 90% of probability for the agent to choose the optimal action according to Q-values from the Q-table and 10% of probability to choose a random action. The learning rate α will be set to 0.01. Evaluation of the algorithm will be performed after every 10 training episodes, and in test games, actions are chosen to follow the current optimal policy instead of ϵ -greedy policy. The number of training episodes depends on the size of the maze, and larger mazes will be trained more times than smaller ones.

Parameters	Values
epsilon	0.1
discount factor	0.95
learning rate	0.001
training episodes	1000

Table 3: Parameters setting for the Q-learning.

3.4 Deep Q-Learning

The main reason we turn to use deep Q-learning is that it is more powerful and it has the capability of generalization. Since our tasks are relatively simple, the architecture of our neural

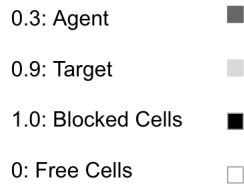


Figure 9: Color map and corresponding representations.

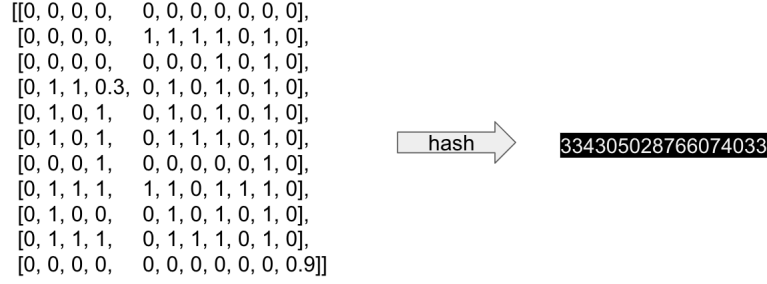


Figure 10: Example of hashing a state, a huge array becomes a bunch of numbers.

network is quite straightforward and shown in Fig. 11. The input of the neural network is the maze itself which is an array and the output is the Q-values of four different actions. Three fully connected layers are used, and a parametric rectified linear unit(PReLU) is used as the activation function. The loss function we are using is mean squared error and Adam optimizer is applied. The details of the parameters we are using shown in Tab. 4.

Parameters	values
learning rate	0.001
discount factor	0.95
initial weights	RandomNormal(mean=0, stddev=0.01)
initial biases	0
input size	maze.size
neurons of hidden layers	2 * maze.size
size of the replay buffer	1000
size of data get from replay buffer	50
batch size	16
epochs	8

Table 4: Parameter settings for the Deep Q-learning.

Since the sizes of our inputs to the neural network are quite small, we won't need to reduce the spatial size of the representation, thus any pooling layers and dropout won't be used. As we know, the design of neural network architecture is tricky and demanding and many new architectures are proposed in top conferences in the machine learning field every year. The architecture of our model is not well-designed and quite straightforward, we just use it for the demonstration. We believe a well-designed neural network will perform better.

3.5 Historical Information Implementation

The goal of this project is to use historical information to accelerate and improve Q-learning performance in maze navigation tasks. Historical information we will use correspond with the record of historical visited cells and guides from the expert which is a pre-trained model trained on the same task. Games with these two different types of historical information added will

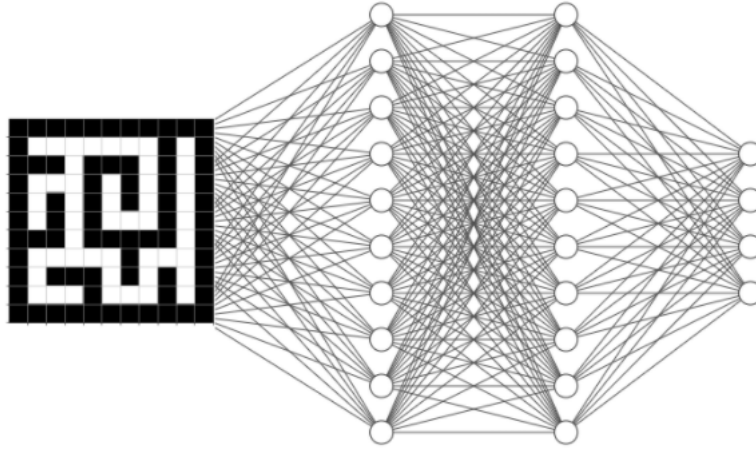


Figure 11: The architecture of our using neural network, there are three fully connected layers in total.

be examined, and normal games with nothing added will be set as the control group to see if historical information is helpful or not.

Control Group The control group is normal games, there is no historical information added, the agent will get three different rewards which are step penalty, invalid penalty, and target reward while there is no expert used.

Historical Information I The record of visited cells, we use extra memory to store corresponding coordinates of cells that have already been visited by the agent in each episode, then every time the agent reaches a successor cell after taking a certain action, the successor cell will be checked if it is already in the historical visited record and then we decide which type of penalties should be assigned to the agent. If the cell has already been visited before, the visited penalty will be given to the agent, the step penalty will be given if not. Different visited penalties also affect the performance a lot, so we will examine several visited penalty values to find the optimal one. Then according to the relation between mazes and their optimal visited penalties, we could predict the optimal visited values for unseen mazes, which could make this historical information more powerful.

Historical Information II The guide from the expert, the expert is a pre-trained model that is trained in the same maze and could solve the maze perfectly. During the training, the expert will produce Q-values for the agent every n steps in the first m training episodes. Then the agent uses Q-values produced by the expert as the target Q-values to update its model. n and m could be tuned for controlling the total amount of guides that will be given to the agent. The larger n and m means the more guides the expert offers to the agent. When the n is equal to 1, it means that the agent is entirely copying the ability of the expert.

4 Experiments

In this section, we will elaborate on experimental details. First, the tools and machines we used for implementing and running our experiments will be described. Then we will explain

why and how we design our experiments and metrics we proposed for measuring our experimental performance. In the end, experimental results will be given with intuitive interpretations.

The whole experiment can be mainly divided into two different parts, one is aimed at answering the first research question, and games with historical information I added will be performed, while another part is trying to answer the second research question so games with historical information II added will be mainly examined. The rough workflow of our experiments shown in Fig 12. We can see there is a Trail and Error part, some experiments we tried but did not work as we expected will be described.

In the first part, the original table-based Q-learning is used. We performed games with four different historical information I added on 20 different mazes and compared with the control group. Based on the metrics we proposed for measuring performance, we got the optimal penalty value for historical information I in each maze. In the end, we trained a classifier based on data we collected from experiments, and it could be used for predicting the optimal penalty for unseen naive mazes.

In the second part, Deep Q-learning is applied. Comparisons between games with historical information II added and the control group will be performed. We also further tuned key parameters of historical information II to control the number of guides the agent could get in total.

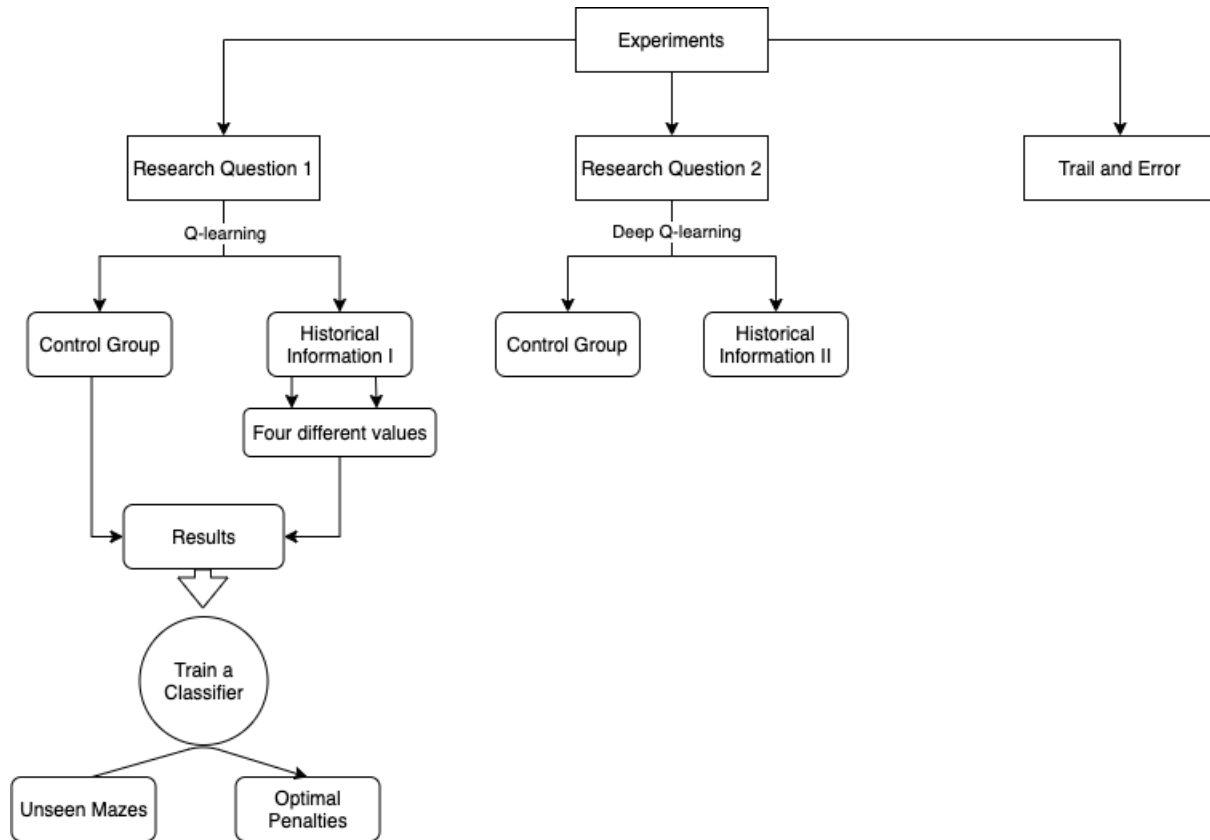


Figure 12: Experimental workflows, there are two main parts, in each part we will try to answer one research question.

4.1 Experimental Settings

We use Python3.7 to implement the whole project, and Keras framework(Tensorflow is used as the backend) is mainly used to build our neural networks, pandas and numpy are used as supplementary computation libraries. Tensorboard, Visdom and matplotlib are applied for monitoring our training process and visualization. For running Q-learning experiments part, we use only one-core CPU on **mithril** which is specialized in CPU computations in DS Lab. Also, we use Nvidia Tesla K80 GPU to accelerate our Deep Q-learning codes, codes are run on **tritanium** in DS Lab.

4.2 Experimental Design

Based on the research questions we proposed in the beginning, we come up with several experiments to verify our ideas and prove our guesses. We first design experiments aimed for answering the first question **Do penalties help or not, if yes, how do we set optimal visited penalties for unseen mazes?** The question contains two parts, one is **Do penalties help?** and another is **If they help, how do we set optimal visited penalties for unseen mazes?** We will first compare the performance of normal games and games with historical information I added, then compare the performance within games with different values of historical information I added to see which one could make the agent have the optimal performance. The historical information I refers the record of visited paths, and different historical information I means different values of visited penalties which will be given to the agent when the agent reaches cells in the record. The key point of these experiments is the setting of different visited penalties. Since the step penalty has already been set, to make visited penalties comparable quantitatively with the step penalty and get measured easily, visited penalties will be set as multiples of the step penalty. More precisely, four different visited penalties which are 5 times(-0.2), 10 times(-0.4), 50 times(-2.0) and 250 times(-10.0) as much as the step penalty will be examined respectively. We define the '**optimal**' visited penalty as: if the game with a certain visited penalty added outperforms the normal game and games with any other visited penalties added on most of the metrics we come up with, then we say this certain visited penalty is the optimal visited penalty. The metrics we used to evaluate performances are shown below.

Win rate During the training period, we run an evaluation after every 10 training episodes, if the agent reaches the target in the evaluation game the 'win' will be counted. The win rate is over the last 100 evaluation games. This is the intuitive measurement of the performances. It can be calculated as:

$$\text{win rate} = \begin{cases} \frac{\text{the number of win in last } n \text{ evaluation games}}{100} & \text{if } n < 100 \\ \frac{\text{the number of win in last 100 evaluation games}}{100} & \text{if } n > 100 \end{cases}$$

, where n is the number of total evaluation games. The higher win rate refers the better performance and it is based on episode-wise however the running time of each episode might be different.

Running time Apart from the win rate, we also take the running time into account. Since we run experiments on the same machine which means the capacity of the machine is the same, the running time could tell us which methods or which games are more expensive. We

measure the running time of a specific number of training episodes. Less training time refers to the faster training process and better performance.

Steps The number of steps the agent takes will indicate how the agent behaves within the game. Does it wander a lot or it is decisive? Does it find the optimal solution in the end or it wanders for a while then reach the target? Such questions could be answered by the number of steps it takes. It is defined as: the number of total steps the agent takes until a game episode is terminated.

Rewards What if agents are not able to find the way to the target, are they bad or there are still some good ones among them. According to rewards the agent holds when the episode is terminated, we can still make a distinction between performances of agents who lost games. For example, let's assume both two agents did not reach the target in the end and in the game with historical information I added, the total reward of the agent who almost does not explore new cells but revisited the same cell again and again will be much lower than the total reward of the agent who explores a lot but still wanders sometimes. The higher rewards the agent holds, the better performance it has.

After we answer the first research question using the aforementioned metrics, we turn to the second one, **Are guides from the expert helpful for improving the performance and accelerating the training process?** Deep Q-learning will be used instead to answer this question. In this part, games with both historical information I and II added will be performed, and normal games will be used as the baseline. For historical information I, only -0.25 will be examined as the visited penalty. For historical information II, the agent will get guides from the expert every n steps in the first m training episodes. After m training episodes, the agent will use the network which is trained in the m training episodes to perform the rest of the episodes. In our experiments, we train the network 500 episodes in total and we set m to 300, while we will examine different n , $n = 10, 40, 50, 100, 200$. Only the win rate will be used for measuring performances in comparisons we mentioned above.

4.3 Experimental Results

4.3.1 Results on Historical Information I

To answer the first research question, we generated 20 mazes in total using our maze generator. We will illustrate an example and remaining detailed results will be given in Appendices. Maze No.4 is shown in Fig. 13. The state space of maze No.4 is 49 which is also the number of all free cells in the maze, and the size is 10 by 10, the number of intersections is 3 and the length of the shortest path is 28.

In Fig. 4.3.1, it shows the win rate of normal games and games with different historical information I added and combined with training time(seconds). The x-axis is the number of training episodes, each episode is counted only when the game is terminated. The upper part of the y-axis is the win rate and the lower part is the training time. And in the legend, the number behind 'Historical Information I' is the value of the visited penalty. **Historical Information I: -0.4** performs the best according to the win rate in episode-wise, its win rate first reaches 1 and keeps unchanged. To better analyze the speed and the performance, we use the control

varieties method and fix the number of training episodes to 5k in this case. Therefore, the steps and rewards are calculated based on 5k training episodes. Under this context, win rates are the same and all converge to 1 in the end. **Historical Information I: -2.0** is the fastest one, it's one more time faster than Historical Information I: -.2 in fishing 5k episodes and two more times faster than normal games. Furthermore, the slope of each training time line looks dynamics in the beginning and converge to the constant after a while. That's because first the agent is trying to find the solution and when it starts wining the costed time of each episode will be almost the same so the slope is being stable and unchanged. In Fig. 4.3.1, the highest average reward is produced by **Historical Information I: -0.4** within 5k training episodes, and followed by Historical Information I: -2.0, which means the average performance of Historical Information I: -0.4 is the best under 5k training episodes regardless of win or lose. Next, we evaluate the average number of steps the agent takes within 5k training episodes shown in Fig. 17. They all find out the optimal path in which 28 steps need to be taken. The steps number of Historical Information I: -10.0 distribute the most dispersively and followed by Historical Information I: -2.0 and Historical Information I: -0.4. It seems like that the larger the visited penalty the more dispersively the steps distribute. **Historical Information I: -2.0** gives us the least average steps, and it indicates the episode is terminated using the least steps under this setting. In conclusion, for maze No.4, games with **Historical Information I added** outperforms normal games. Furthermore, **Historical Information I: -2.0** outperforms under 2 out of 4 evaluation metrics and ties for the first place with other settings, it also gives the second-best in Rewards while Historical Information I: -.4 only outperforms in 1 out of 4 metrics. Thus, we could say, in the environment and algorithm we are using, also keeping other settings fixed, **Historical Information I: -2.0** is the optimal visited penalty setting for maze No.4. Using the same judgment idea, we picked optimal penalties for the other 19 mazes. Results are shown in Tab. 5 and we plot the optimal penalties for different mazes and mazes' properties in Fig. 14. Detailed results of other experiments will be given in Appendices.

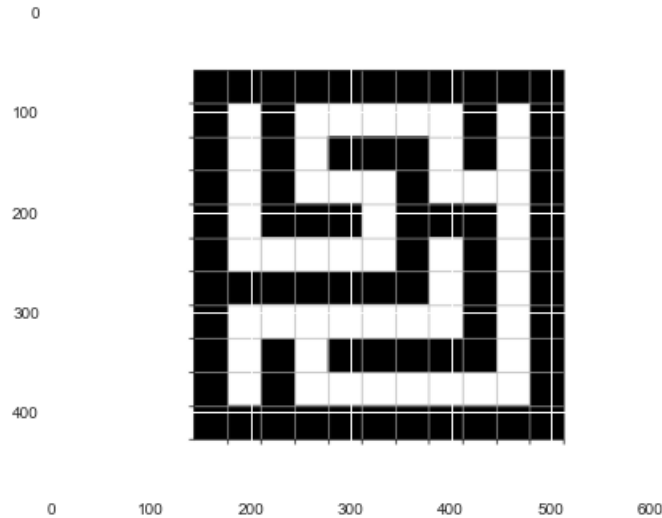


Figure 13: How Maze No.4 looks like, the size is 10, the state space is 49, the length of the shortest path is 28 and the number of intersections is 3.

It is quite explicit that the optimal visited penalty is different from the complexity of the maze changes. Metrics we proposed for measuring complexities are highly correlated, the size of

the maze is the main determinant and it further decides the size of the state space and the shortest path. Since the maze is generated randomly, there is some randomness which makes metrics are not perfectly positive linear correlated. According to Fig. 14, we group the mazes by their optimal visited penalties, and within the group, we continue to sort the mazes by their properties. It does show optimal penalties are getting smaller with four properties of the maze increase. The values of visited penalties we set are quite coarse, and we didn't set any other values for testing between -2.0 and -.4. So if the precise optimal visited penalty of a maze is in this range, it will be rounded to either -2.0 if the real optimal value is closer to -2.0 and vice versa. However, the explicit boundary or perfect linear relationship could not be found within a limited number of experiments.

To give the optimal penalty setting for unseen mazes, we then trained a classifier to predict the optimal penalty for unseen mazes based on their properties. K-Nearest Neighbors(KNN) algorithm [27] is used in our case, while logistic regression or other classification methods also could be used here. Since we have already had results on 20 mazes, a leave-one-out cross-validation is performed then. Every time we train a KNN classifier on 19 mazes and then we use the trained classifier to predict the optimal penalty for the last unseen maze. We use 4-dimensional inputs contain the shortest path, the size, the state space, and the number of intersections and the output is the optimal penalty among the four penalties we examined which are -0.2, -0.4, -2.0, and -10.0 respectively. This process will be repeated 20 times then we calculate the accuracy. We set k to 5, and the accuracy of the KNN classifier is 80%. It is a pretty good classifier, at least in our context.

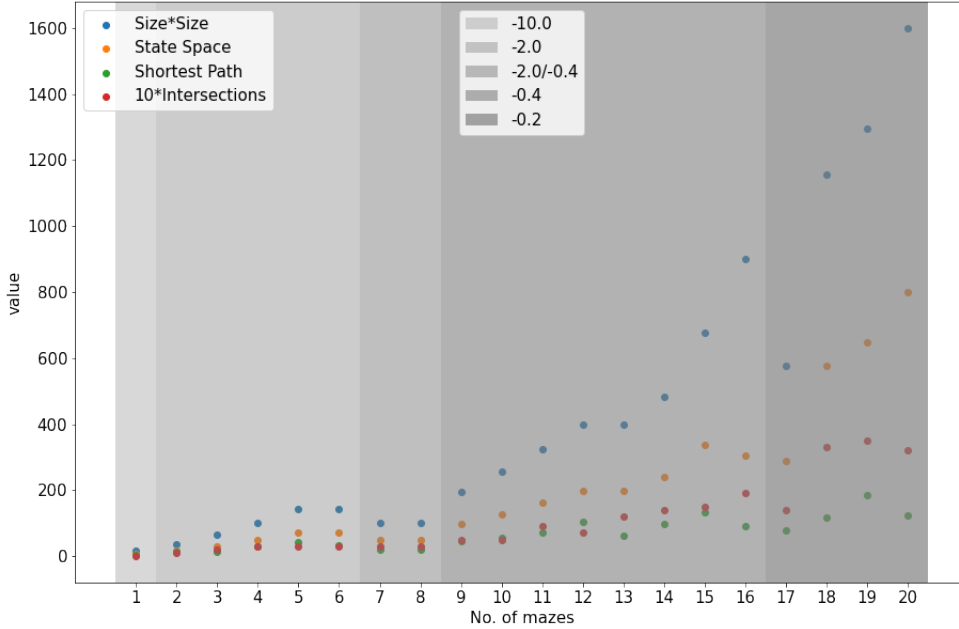


Figure 14: Properties of mazes and their optimal penalty settings.

4.3.2 Results on Historical Information II

Now let's have a look at the results we got for answering the second research question. There are two main hyper-parameters that could be tuned, which are n and m . Guides will be offered by the expert every n steps in the first m training episodes, after m training episodes

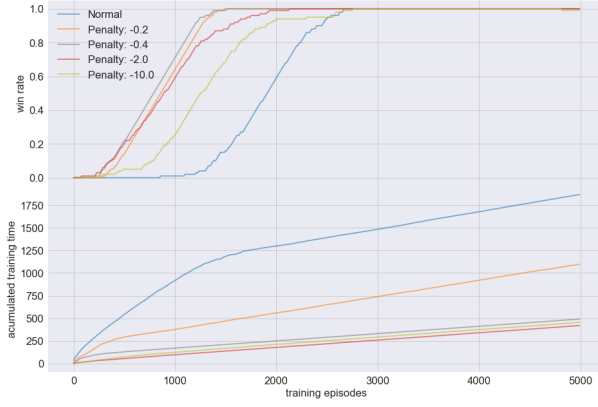


Figure 15: Win rate and training time of experiments on Maze No.4.

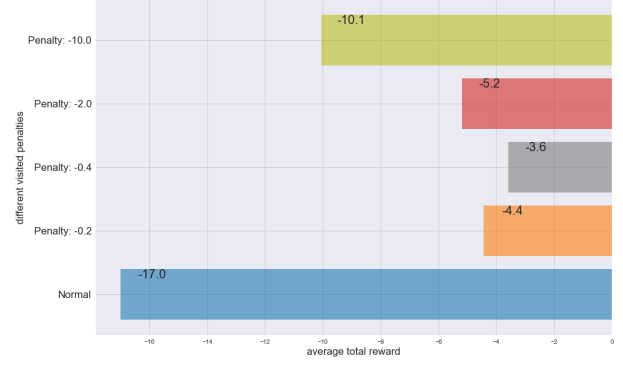


Figure 16: Average rewards of experiments on Maze No.4.

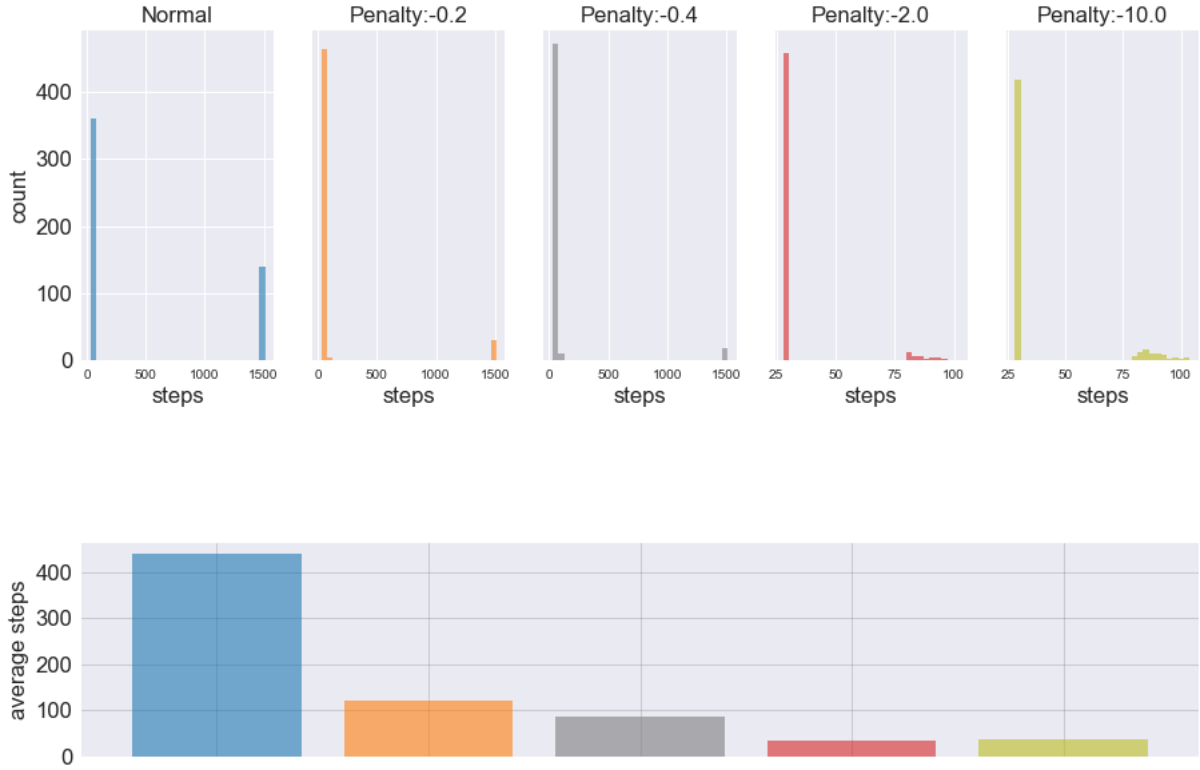


Figure 17: The distribution of steps and the average step.

we will stop training and use the neural network which is trained within this m episodes to produce actions. We fix m to 300 and try several different n on three different mazes, but we only plot results of three different n which could give us explicit different performances for each maze. The properties of these three different mazes shown in Tab. 6. Results shown in Fig. 18, Fig. 19, Fig. 20. The grey areas are 90% exploitation and 10% exploration without any further training. And in the legend, the number behind 'Historical Information II' is the value of n . In these three results, we can easily see that normal games perform the worst while

mazes	size	state space	shortest path	intersections	optimal penalty
maze 1	4	7	5	0	-10.0
maze 2	6	17	13	1	-2.0
maze 3	8	31	12	2	-2.0
maze 4	10	49	28	3	-2.0
maze 5	12	71	44	3	-2.0
maze 6	12	71	33	3	-2.0
maze 7	10	49	19	3	-2.0/-0.4
maze 8	10	49	19	3	-2.0/-0.4
maze 9	14	97	45	5	-0.4
maze 10	16	127	56	5	-0.4
maze 11	18	161	72	9	-0.4
maze 12	20	199	104	7	-0.4
maze 13	20	199	61	12	-0.4
maze 14	22	241	97	14	-0.4
maze 15	26	337	132	15	-0.4
maze 16	30	305	92	19	-0.4
maze 17	24	287	77	14	-0.2
maze 18	34	577	117	33	-0.2
maze 19	36	647	184	35	-0.2
maze 20	40	799	124	32	-0.2

Table 5: Summary results of mazes’ properties and optimal penalty could be set.

games with historical information added are much better based on the episode-wise win rate. Meanwhile, the smaller n we set the better performance the agent will have, which makes a lot of sense. The smaller the n is, the more frequent and more information the agent will get from the expert. Let’s first look at Fig. 18 which is the result based on Maze No.21, both the normal game and the game $n = 100$ not be able to reach a decent win rate with 300 episodes training; while the game with $n = 40$ is slightly slower than the game with $n = 10$ whose win rate rushes to 1 directly in 100 training episodes; the game with historical information I added also performs pretty good. In Fig. 19 which is the result of Maze No.22, there are clear twists after 300 training episodes in games with $n = 50$ and $n = 100$, that’s because the agent does not learn how to reach the target successfully in 300 training episodes, getting guides from the expert helps the agent solve the maze in the beginning but they do not learn themselves, this is why the agent keeps failing when it is asked to use its network to solve the maze after 300 training episodes; the game with $n = 10$ reaches the 1 very fast; the game with historical information I added also outperforms most of the games. In Fig. 20 which is the result on the Maze No.23, every game reaches a decent win rate in the end; the performance of the normal game increases steadily; it is interesting that the blue line soars up to 1 and then decreases to around 0.8, that is because the network learned to deal with most of the situations and there are only a few situations where the network could make mistakes, but remember we still have 10% of exploration which helps the agent keep around 80% win rate; the win rate of the normal game even reaches 90% win rate in the end.

Mazes	Size	Shortest Path	Intersections	State Space
No.21	10	24	1	51
No.22	10	24	1	51
No.23	10	21	2	51

Table 6: The properties of Mazes we are using for examining historical information II.

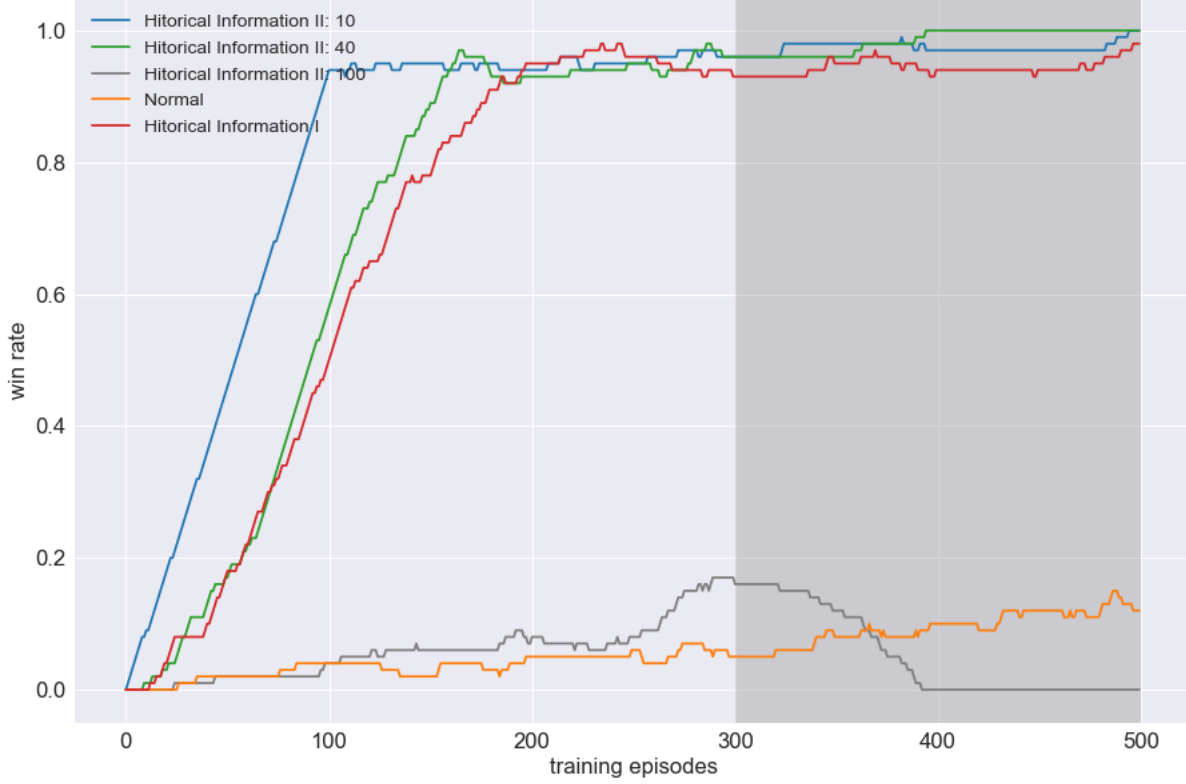


Figure 18: Win rate on Maze No.21, the agent gets guides from the expert in the first 300 episodes.

4.3.3 Trail and Error

As we know, people like using marks as reminders, they make marks on books, on roads, even on historical scenic spots. They want to use these marks to remind themselves or remind others. Here, marks are also historical information that people make before, and we are inspired by this idea and use marks as historical information III. The main idea is that: we put marks on the path which has already been visited by the agent, and theoretically, in the shortest path towards the target, visited cells are always further to the target compared with unvisited cells, also unvisited cells have information that has not been discovered yet by the agent, so the agent should avoid revisiting visited cells and visit more new cells. Thus, we put marks on visited cells, and hopefully, the agent could learn the relationship between the target and paths with marks. Since mazes that we are using are arrays, so marks we will put are different values in the array and could be interpreted as different colors of cells in the maze. The difference between mazes with marks added and no marks added shown in Fig. 22, the agent(dark grey) is in the center of the maze, and the right maze is with marks(grey) added while there is no mark on the left one. So marks are different colors we give to cells on the maze.

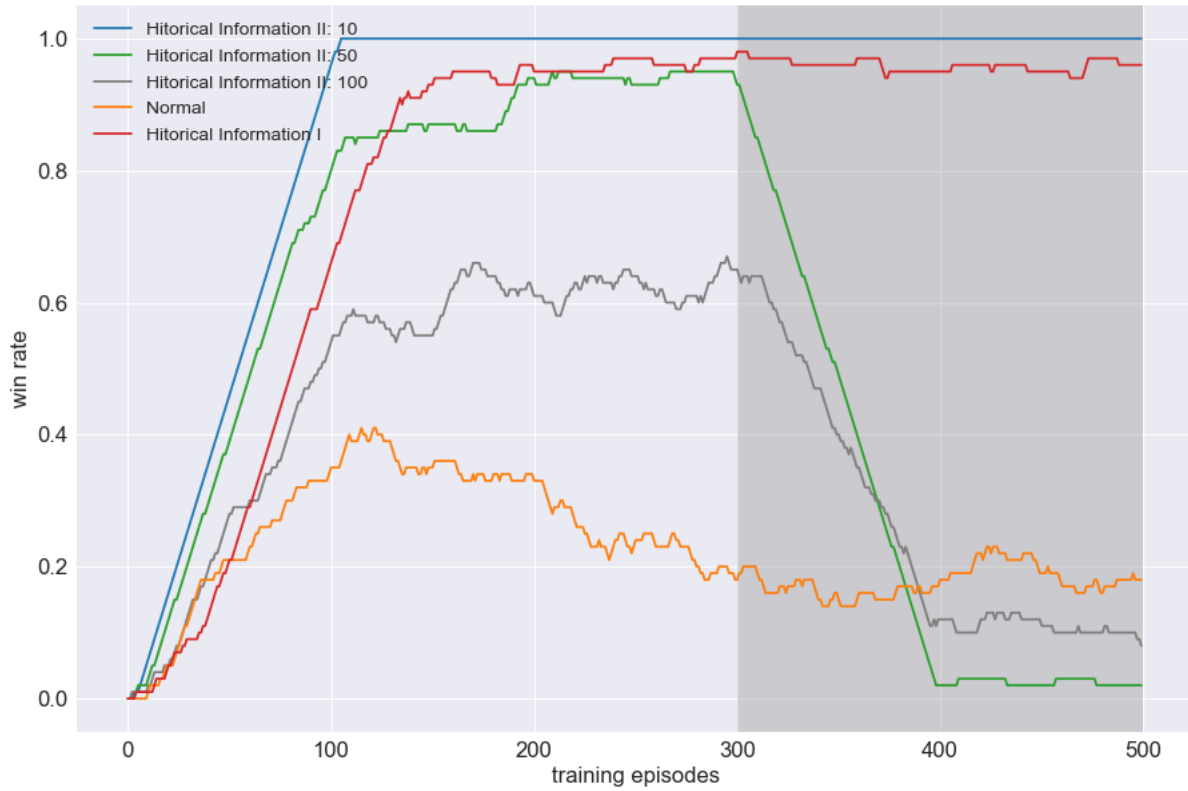


Figure 19: Win rate on Maze No.22, the agent gets guides from the expert in the first 300 episodes.

We tried 4 different strategies to put marks, shown below:

- Put marks on the fly: we put marks on visited cells during each training episode. That is: every time when the agent visits a cell, the value of this cell in the maze array will be changed into another value which indicates the mark;
- Pre-mark the right path: before the game starts, we pre-mark the shortest path to the target. That is: before the game starts, we set different values of cells in the maze array which are on the shortest path to the target;
- Pre-mark the wrong path: before the game starts, we pre-mark the non-shortest path. That is: before the game starts, we set different values of cells in the maze array which are NOT on the shortest path to the target;
- Use one more array: this is similar with the first strategy, we put marks on another empty array instead of putting them on the original maze directly, and then append it to end of the original maze.

These all different marked mazes will be used as the input of the neural network, and the same update rules will be followed. Hopefully, the network could combine marks with the target and learn the relationship between them. However, experimental results showed that the network actually did not learn anything from marks which is not what we expected, and we further roughly analyzed the reason that marks do not work. The results shown in Fig. 21. We can

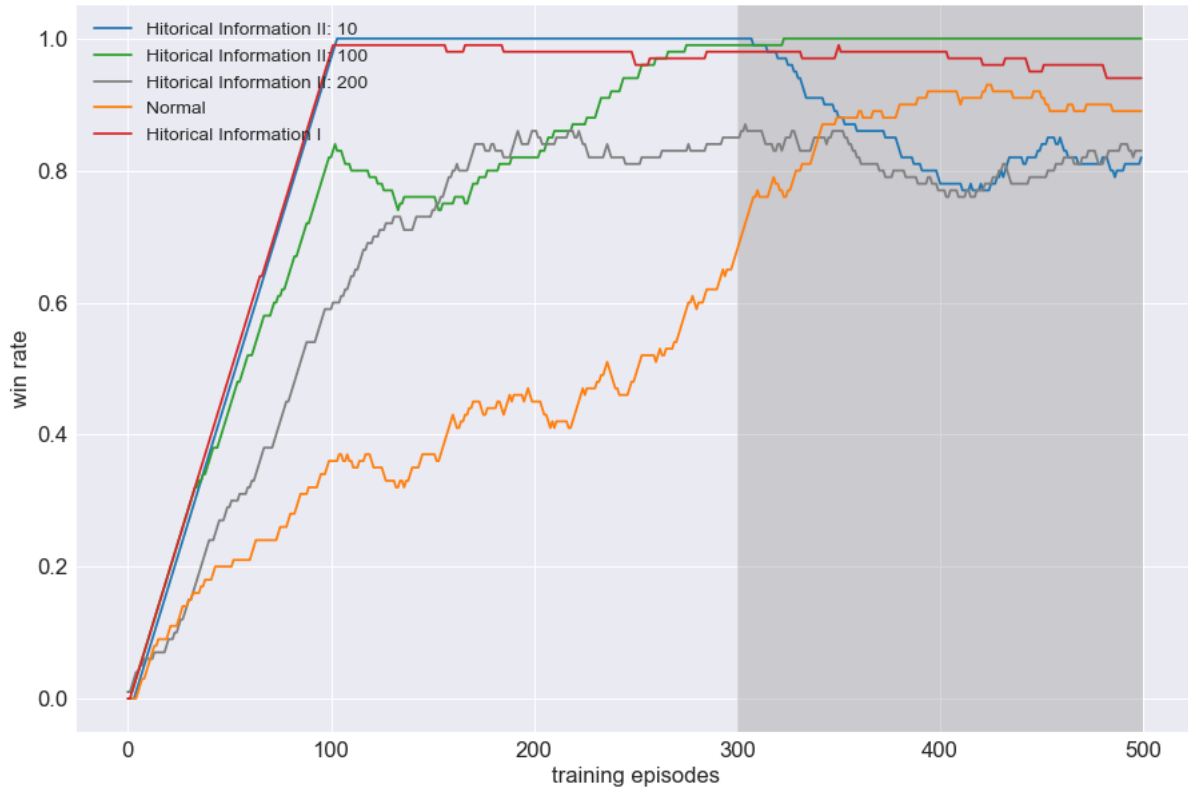


Figure 20: Win rate on Maze No.23, the agent gets guides from the expert in the first 300 episodes.

see that all strategies work similarly to the normal game, which means none of them works. Win rates might fluctuate a lot because of randomness. And compared with the game with historical information I added, they are way worse. Marks we put on the maze are just different values in the maze array which further cause differences in the input, they do affect the forward and backpropagation process during the training, but the effect is only on the surface level. Let's simplify the problem and use only one digit to represent the maze, also one weight for the neural network. The difference between marks added input and no marks added input is similar to: $5 * k$ and $3 * k$, here 5 and 3 are different values(different inputs) where one represents marks and another one indicates normal cells, k is the weight of the neural network. So the output of the neural network will be $5k$ and $3k$, and they have different gradients which further cause different updates for the weights of the network, but the difference is caused by different input values instead of the knowledge that the network learned. The way we put marks could not be understood by the machine and the algorithm we are using, then marks could not be further utilized. It also points out a potential future work, to find a way to put marks which could be understood by the machine and the algorithm.

5 Conclusion and Discussion

5.1 Conclusions

In this project, we proved that two types of historical information we came up with in maze navigation tasks are helpful for enhancing the algorithm itself and useful for helping the agent

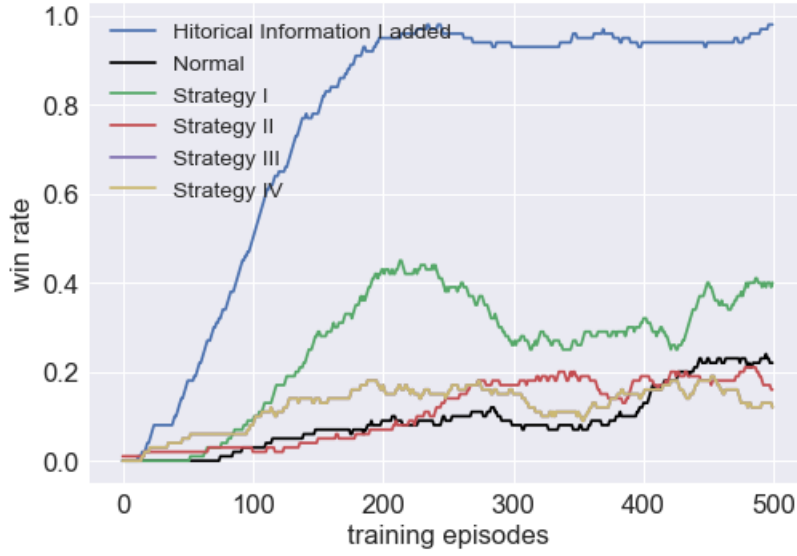


Figure 21: Performance of four different marking strategies on Maze No.4, also with the performance of the normal game and the game with historical information I added.

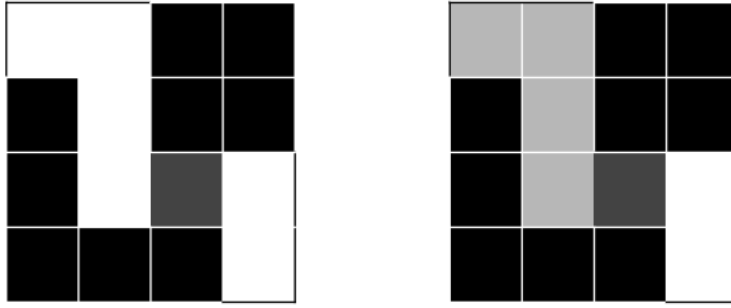


Figure 22: The difference between mazes with marks added and no marks added, the grey colored path behind the agent(the dark grey cell) is the marked path.

reach the target faster. In Q-learning context, historical information I(giving penalties when the agent reaches visited paths) is definitely helpful for accelerating and improving the algorithm. And different penalty values will cause different performances. Roughly speaking, the optimal penalty decreases with the complexity of the maze increases. Meanwhile, the classifier we trained can be used to predict the optimal penalty value for unseen mazes, the accuracy could be up to 80%. In Deep Q-learning context, both historical information I and II(the guides from the expert) are helpful, they could help the algorithm solve the problem way faster in episode-wise. We can further tune m and n for games with historical information II added to control the number of guides that the agent could get. The more guides the agent gets, the better performance it will have. Furthermore, we found that after guiding periods, sometimes the agent's performance has an obvious twist but in the end, it will converge at some point. With guides offered, the agent performs better and when guides stop, the agent needs to perform itself and the performance will change to its own ability. Thus, the 'real' ability the agent learned is actually the performance it has in the end, at least after the guiding period. These findings tell us that if historical information are used properly, they could be helpful for

enhancing performances of algorithms, and we believe not only in maze navigation tasks and not only using Q-learning, but for any other tasks and any other algorithms. The key point is that historical information have to be added in the way which could be interpreted by the algorithm.

5.2 Limitations and Future Works

First of all, we only examined four different values of historical information I and of course the classifier we trained could only produce the optimal penalty value within these four values for unseen mazes, which is pretty limited. And the environment and the task are quite simple, the findings might only make sense in our context and could not be used for other different tasks directly. Meanwhile, as we know, neural networks are kind of black-box tools and pretty sensitive to hyper-parameters, architecture as well as machines they are running on. So the results we got are not robust enough, they might not be exactly the same every round.

There are many future potential interesting works to do based on our findings. For the historical information I, the optimal penalty value for mazes could be more precise instead of just picking from four values, also more data could make the classifier more robust and accurate. In historical information II, the expert is trained in the same task, it would be interesting to see the expert trained on similar tasks to give guides to the agent. Furthermore, historical information III(marks) which is unsuccessful in our case, but we do think it could be somehow useful and it is just because the way we use it is not proper. It is worthwhile to study how marks could be made beneficially for the task and the algorithm. Also, making these historical information ideas more generalized is an interesting direction, either from algorithms-wise or tasks-wise.

References

- [1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.
- [2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.
- [3] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.
- [4] Y. Aytar, T. Pfaff, D. Budden, T. Paine, Z. Wang, and N. de Freitas, "Playing hard exploration games by watching youtube," in *Advances in Neural Information Processing Systems*, 2018, pp. 2930–2941.
- [5] D. Tamagawa, E. Taniguchi, and T. Yamada, "Evaluating city logistics measures using a multi-agent model," *Procedia-Social and Behavioral Sciences*, vol. 2, no. 3, pp. 6002–6012, 2010.

- [6] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, “Grandmaster level in starcraft ii using multi-agent reinforcement learning,” *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.
- [7] A. P. Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell, “Agent57: Outperforming the atari human benchmark,” *arXiv preprint arXiv:2003.13350*, 2020.
- [8] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, “Mastering the game of go without human knowledge,” *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.
- [9] P. Auer, “Using confidence bounds for exploitation-exploration trade-offs,” *Journal of Machine Learning Research*, vol. 3, no. Nov, pp. 397–422, 2002.
- [10] W. R. Thompson, “On the likelihood that one unknown probability exceeds another in view of the evidence of two samples,” *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933.
- [11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
- [12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing atari with deep reinforcement learning,” *arXiv preprint arXiv:1312.5602*, 2013.
- [13] L.-J. Lin, “Reinforcement learning for robots using neural networks,” Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993.
- [14] A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel, “Value iteration networks,” in *Advances in Neural Information Processing Systems*, 2016, pp. 2154–2162.
- [15] J. Zhang, J. T. Springenberg, J. Boedecker, and W. Burgard, “Deep reinforcement learning with successor features for navigation across similar environments,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 2371–2378.
- [16] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, “Go-explore: a new approach for hard-exploration problems,” *arXiv preprint arXiv:1901.10995*, 2019.
- [17] O. Zhelo, J. Zhang, L. Tai, M. Liu, and W. Burgard, “Curiosity-driven exploration for mapless navigation with deep reinforcement learning,” *arXiv preprint arXiv:1804.00456*, 2018.
- [18] J. Bruce, N. Sünderhauf, P. Mirowski, R. Hadsell, and M. Milford, “One-shot reinforcement learning for robot navigation with interactive replay,” *arXiv preprint arXiv:1711.10137*, 2017.
- [19] D. B. Wilson, “Generating random spanning trees more quickly than the cover time,” in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 296–303.

- [20] J. C. Gower and G. J. Ross, "Minimum spanning trees and single linkage cluster analysis," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 18, no. 1, pp. 54–64, 1969.
- [21] H. Yaman, O. E. KaraşAn, and M. Ç. Pınar, "The robust spanning tree problem with interval data," *Operations research letters*, vol. 29, no. 1, pp. 31–40, 2001.
- [22] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.
- [23] P. Gabrovšek, "Analysis of maze generating algorithms."
- [24] M. S. McClendon *et al.*, "The complexity and difficulty of a maze," in *Bridges: Mathematical Connections in Art, Music, and Science*. Bridges Conference, 2001, pp. 213–222.
- [25] J. Xu and C. S. Kaplan, "Vortex maze construction," *Journal of Mathematics and the Arts*, vol. 1, no. 1, pp. 7–20, 2007.
- [26] M. L. Littman, T. L. Dean, and L. P. Kaelbling, "On the complexity of solving markov decision problems," *arXiv preprint arXiv:1302.4971*, 2013.
- [27] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992.

Appendices

A Used Mazes

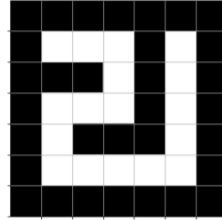
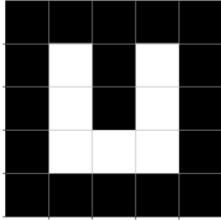


Figure 23: Maze No.1 and Maze No.2.

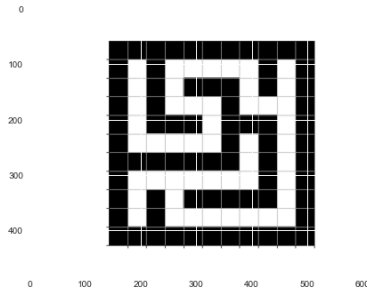
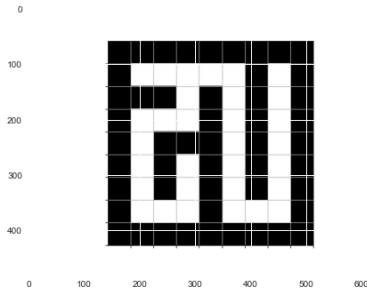


Figure 24: Maze No.3 and Maze No.4.

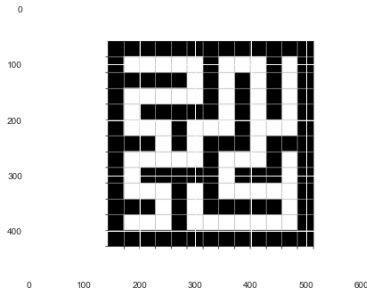


Figure 25: Maze No.5 and Maze No.6.

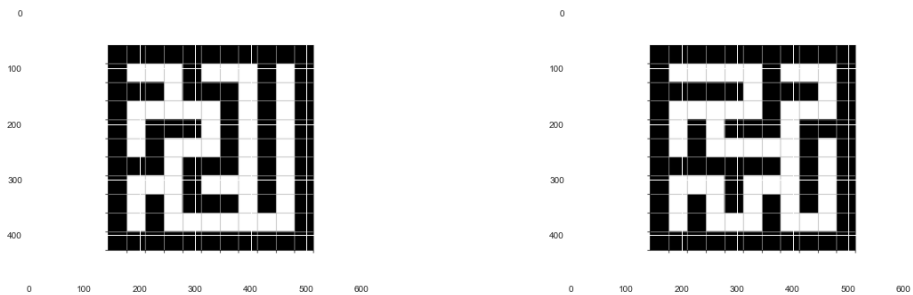


Figure 26: Maze No.7 and Maze No.8.

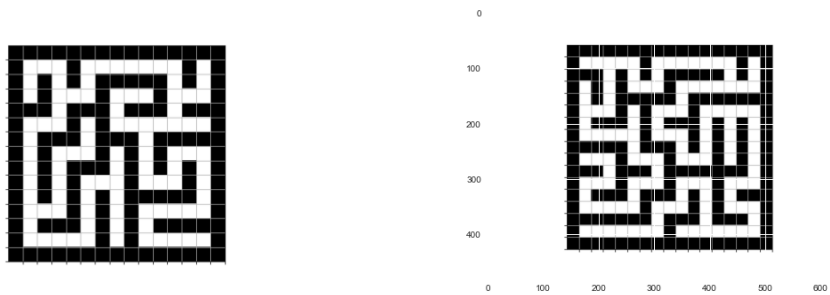


Figure 27: Maze No.9 and Maze No.10.

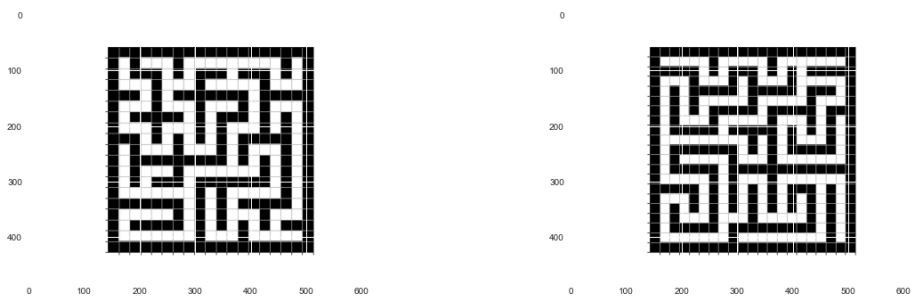


Figure 28: Maze No.11 and Maze No.12.



Figure 29: Maze No.13 and Maze No.14.

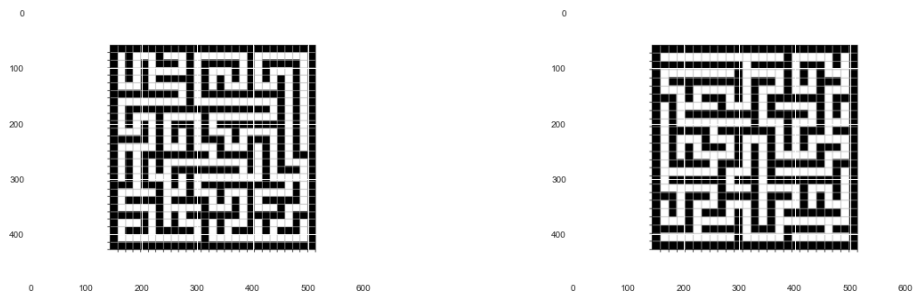


Figure 30: Maze No.15 and Maze No.16.



Figure 31: Maze No.17 and Maze No.18.

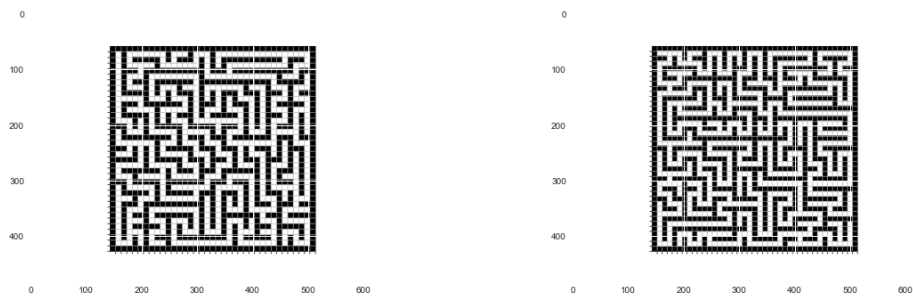


Figure 32: Maze No.19 and Maze No.20.

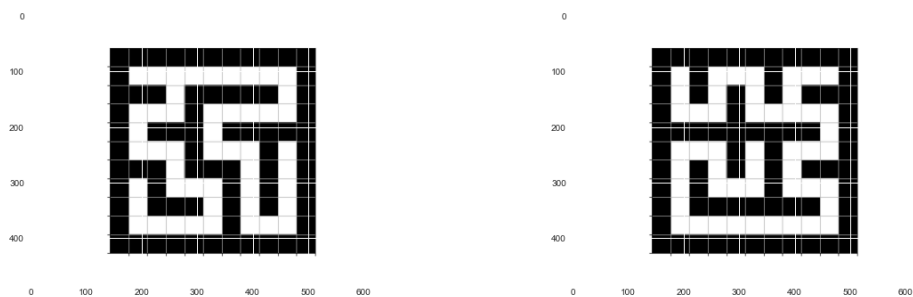


Figure 33: Maze No.21 and Maze No.22.

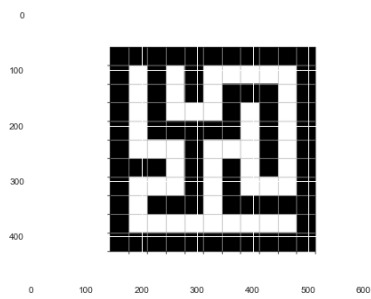


Figure 34: Maze No.23.

B All Experimental Results for Historical Information I

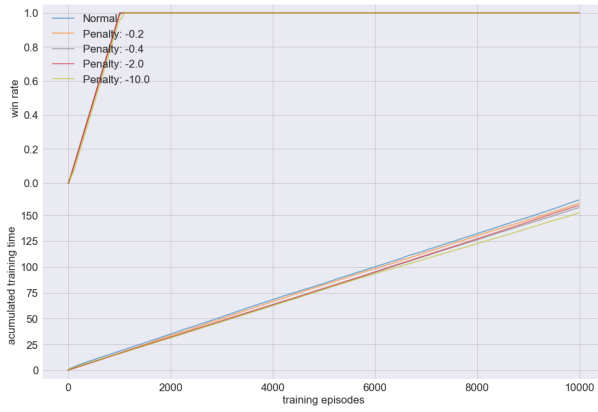


Figure 35: Win rate and training time of experiments on Maze No.1.

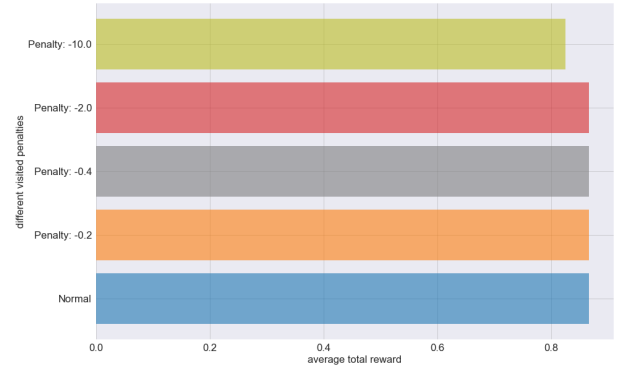


Figure 36: Average rewards of experiments on Maze No.1.

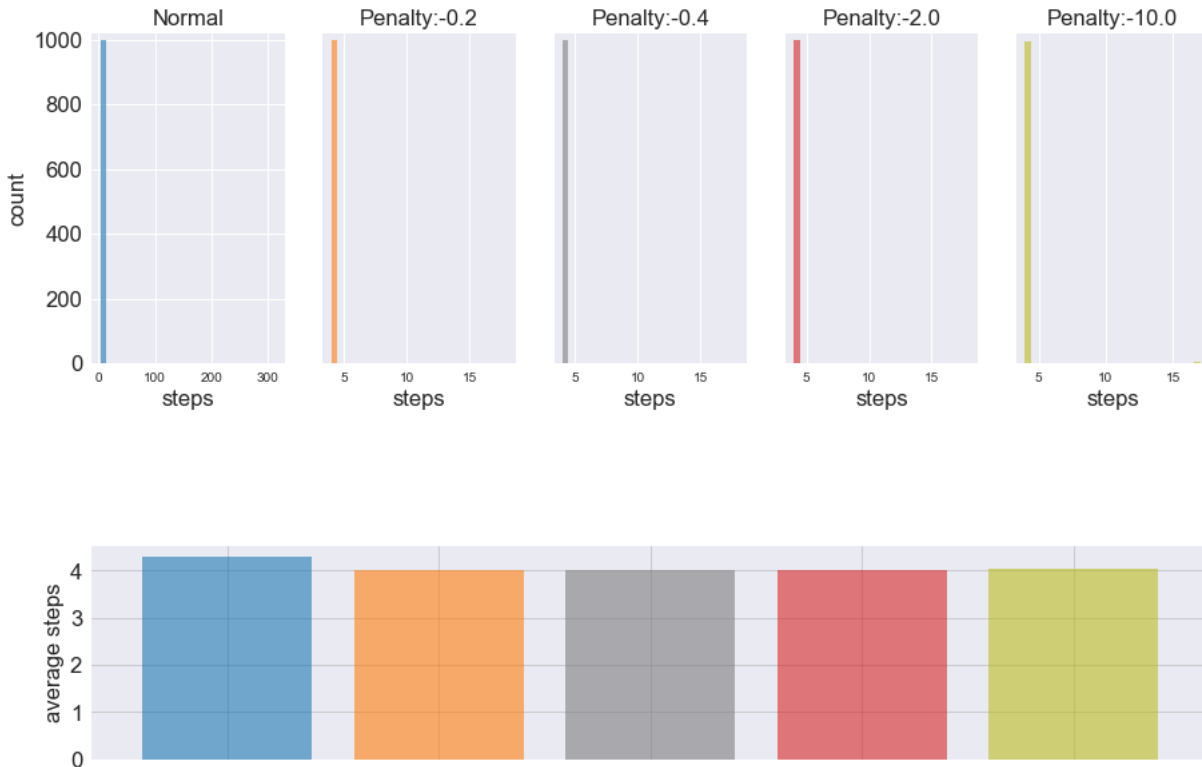


Figure 37: The distribution of steps and the average step.

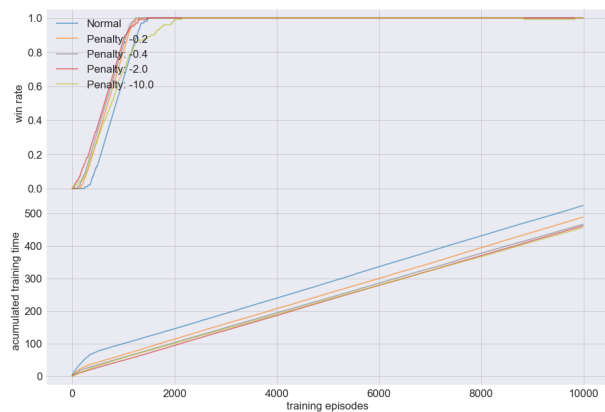


Figure 38: Win rate and training time of experiments on Maze No.2.

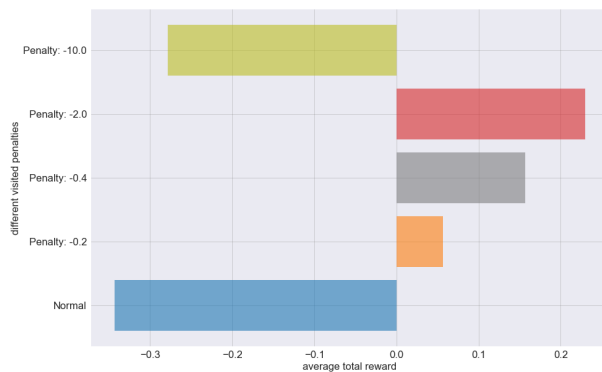


Figure 39: Average rewards of experiments on Maze No.2.

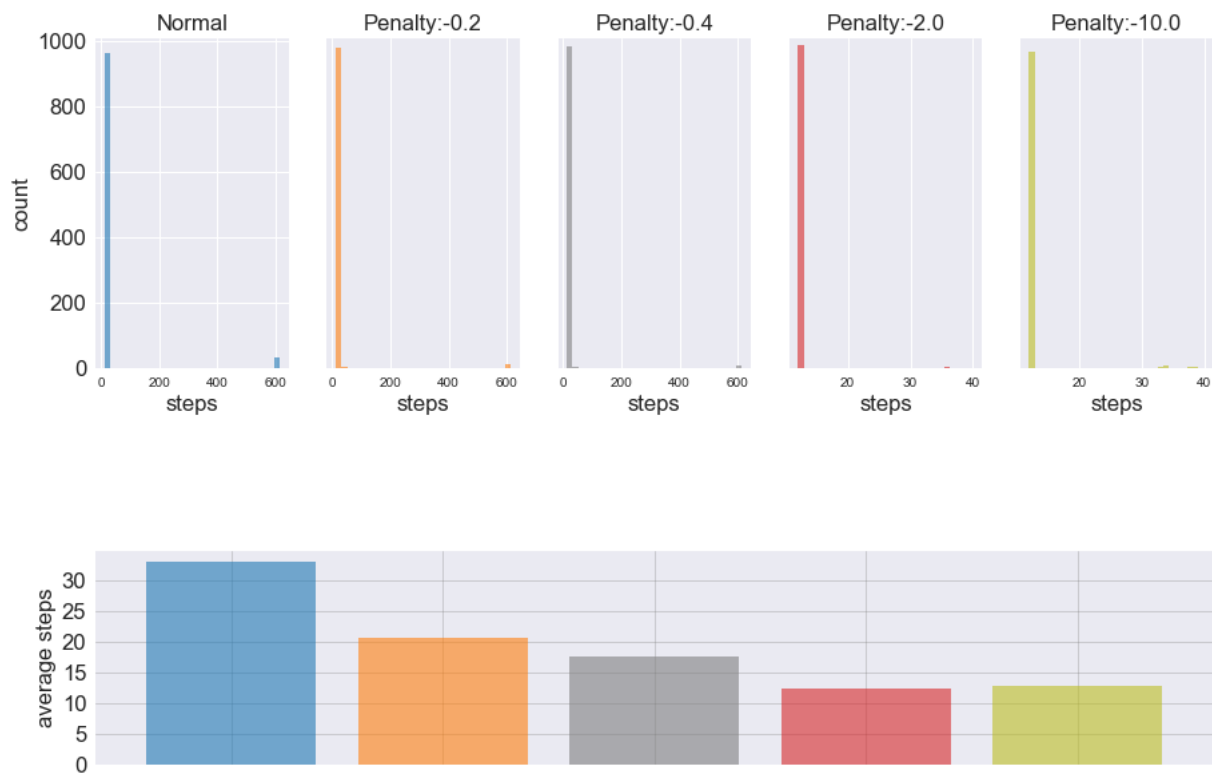


Figure 40: The distribution of steps and the average step.

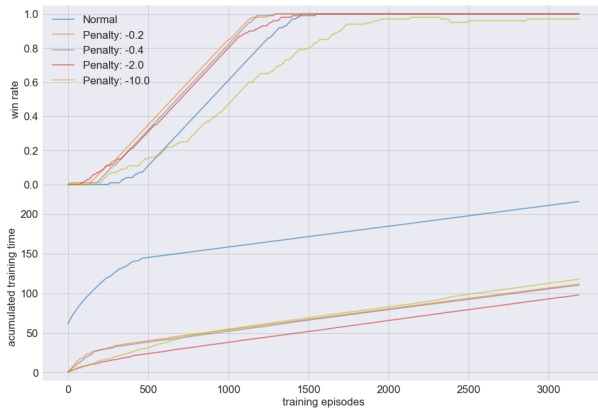


Figure 41: Win rate and training time of experiments on Maze No.3.

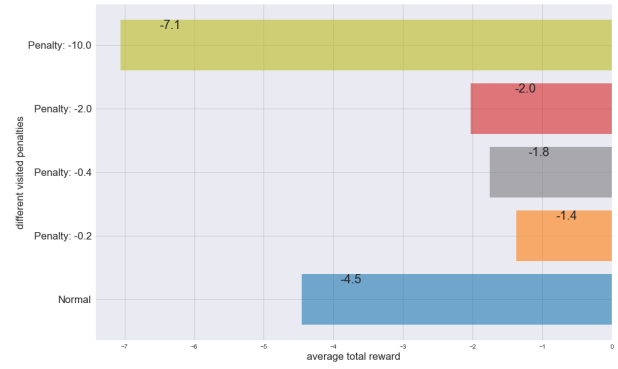


Figure 42: Average rewards of experiments on Maze No.3.

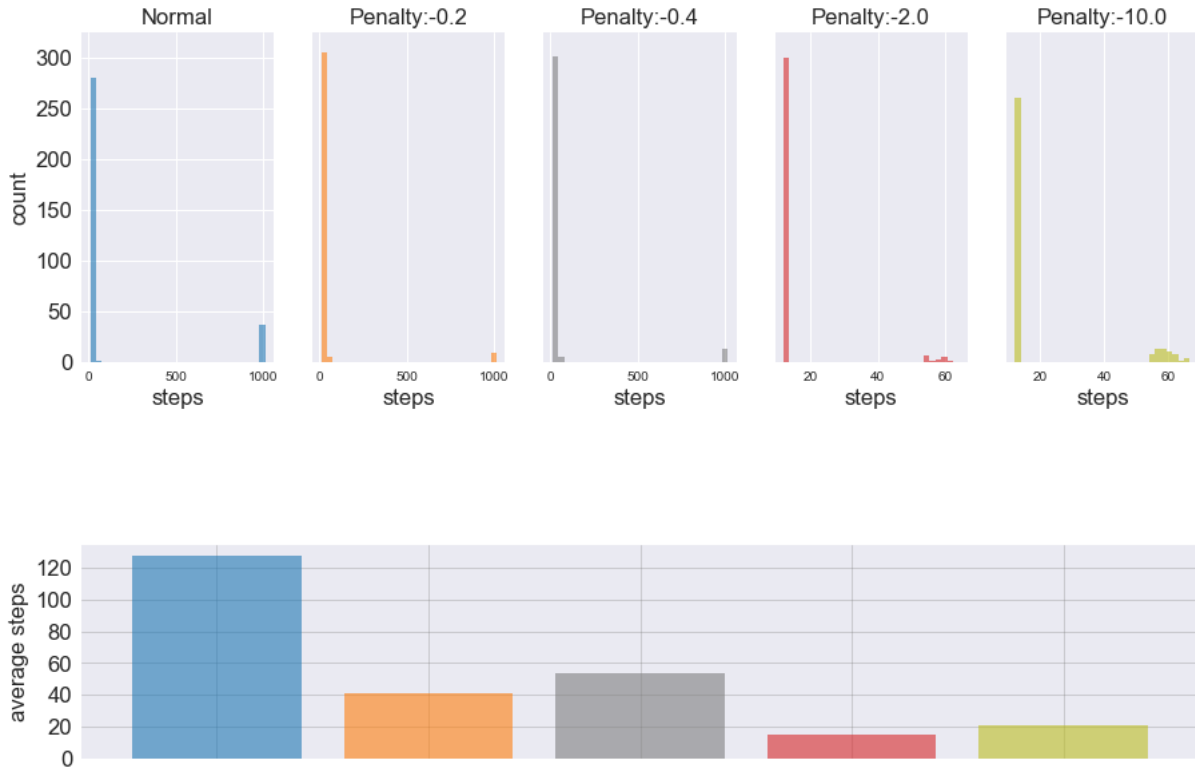


Figure 43: The distribution of steps and the average step.

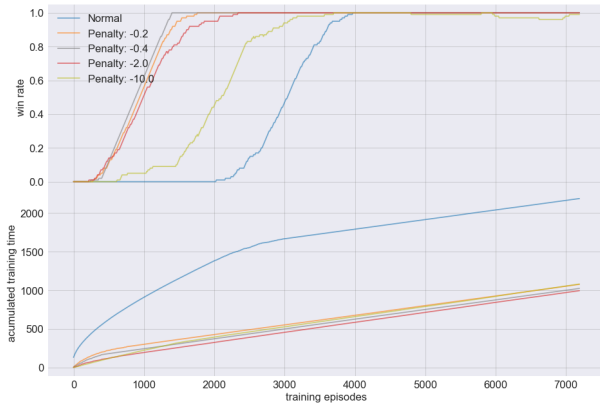


Figure 44: Win rate and training time of experiments on Maze No.5.

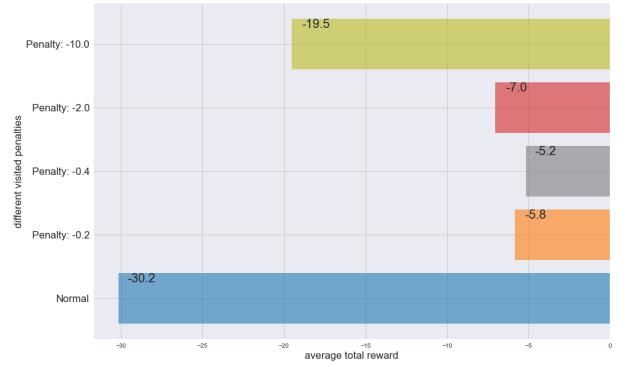


Figure 45: Average rewards of experiments on Maze No.5.

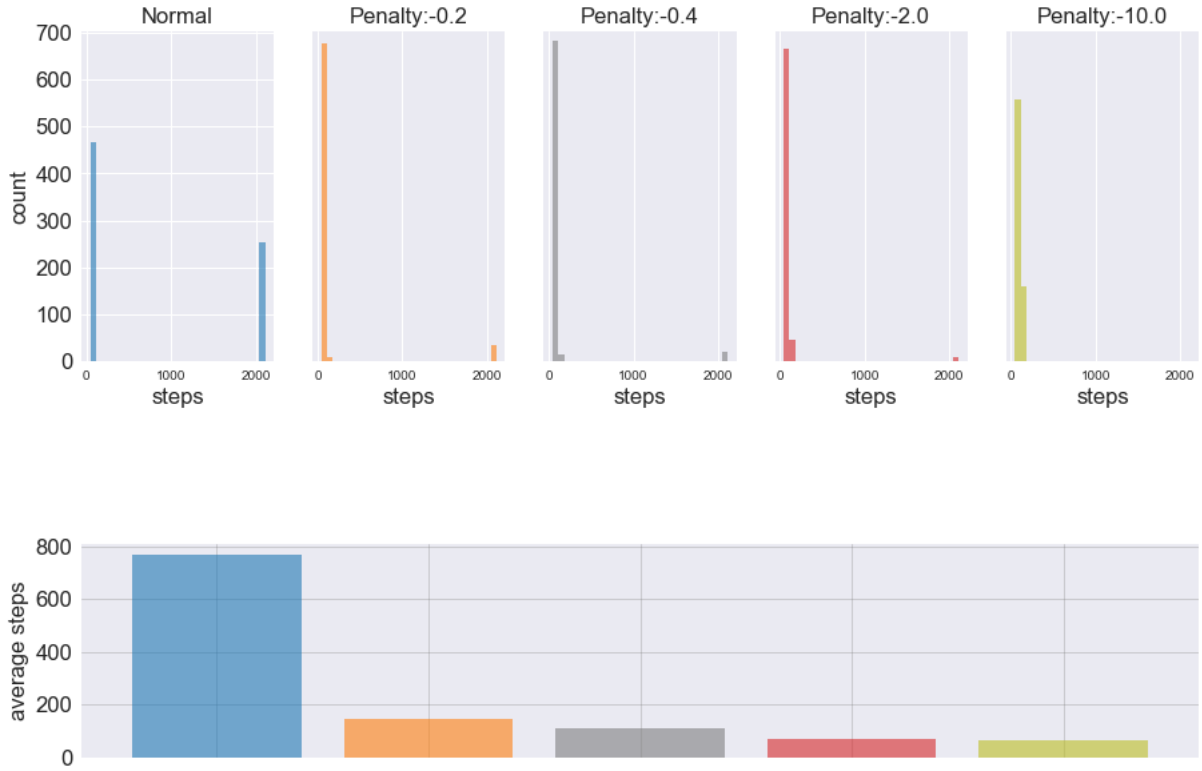


Figure 46: The distribution of steps and the average step.

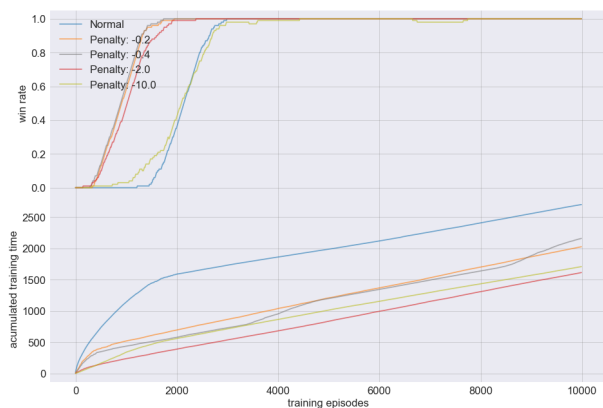


Figure 47: Win rate and training time of experiments on Maze No.6.

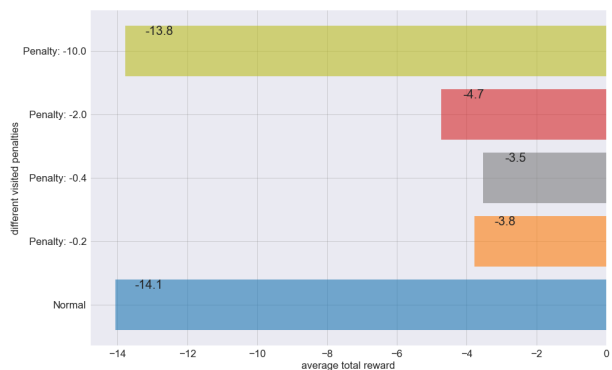


Figure 48: Average rewards of experiments on Maze No.6.

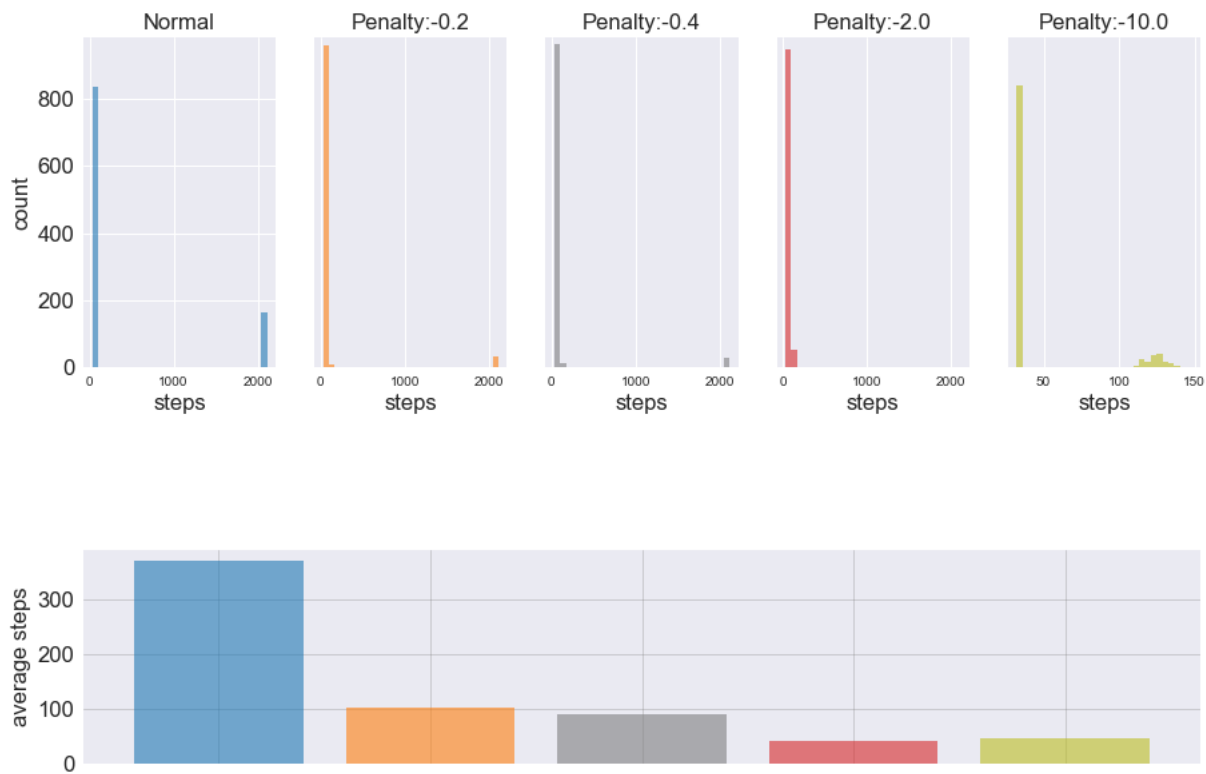


Figure 49: The distribution of steps and the average step.

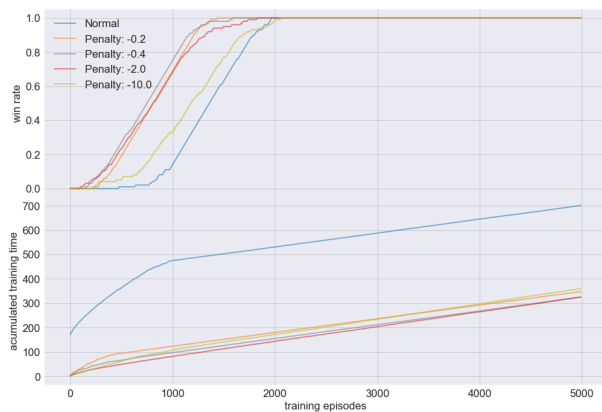


Figure 50: Win rate and training time of experiments on Maze No.7.

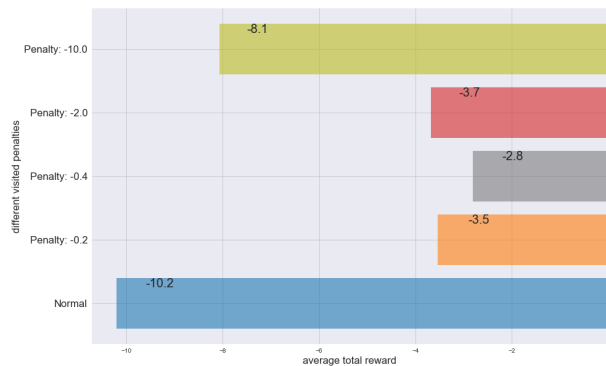


Figure 51: Average rewards of experiments on Maze No.7.

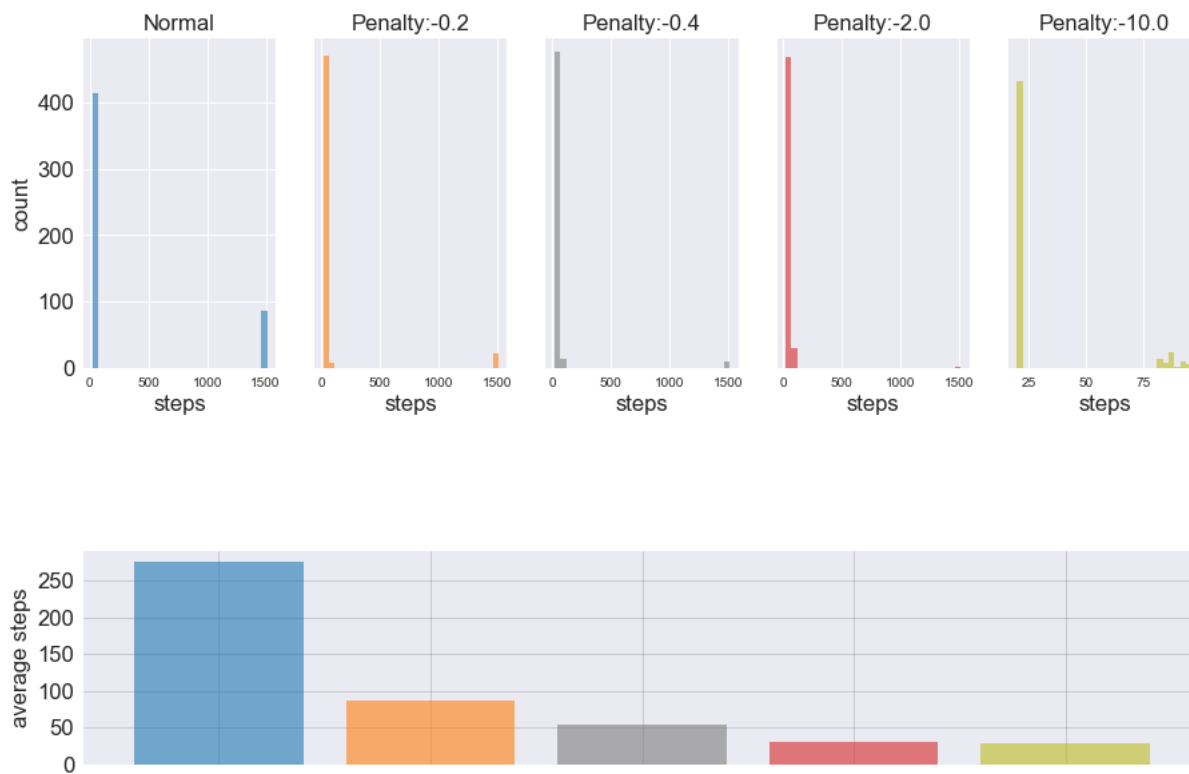


Figure 52: The distribution of steps and the average step.

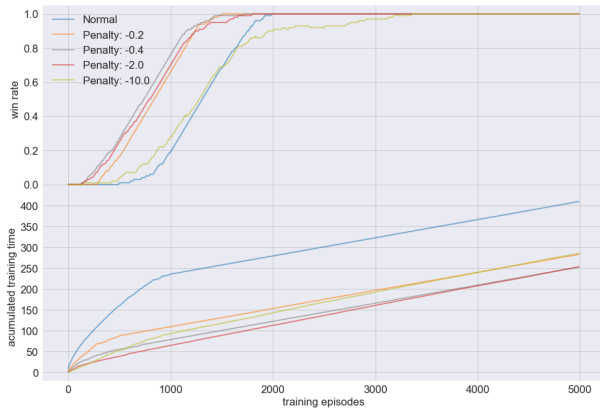


Figure 53: Win rate and training time of experiments on Maze No.8.

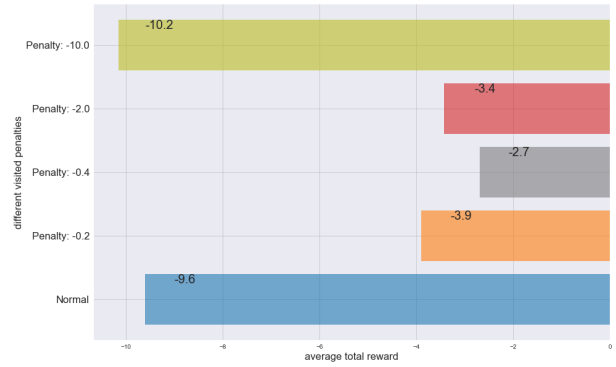


Figure 54: Average rewards of experiments on Maze No.8.

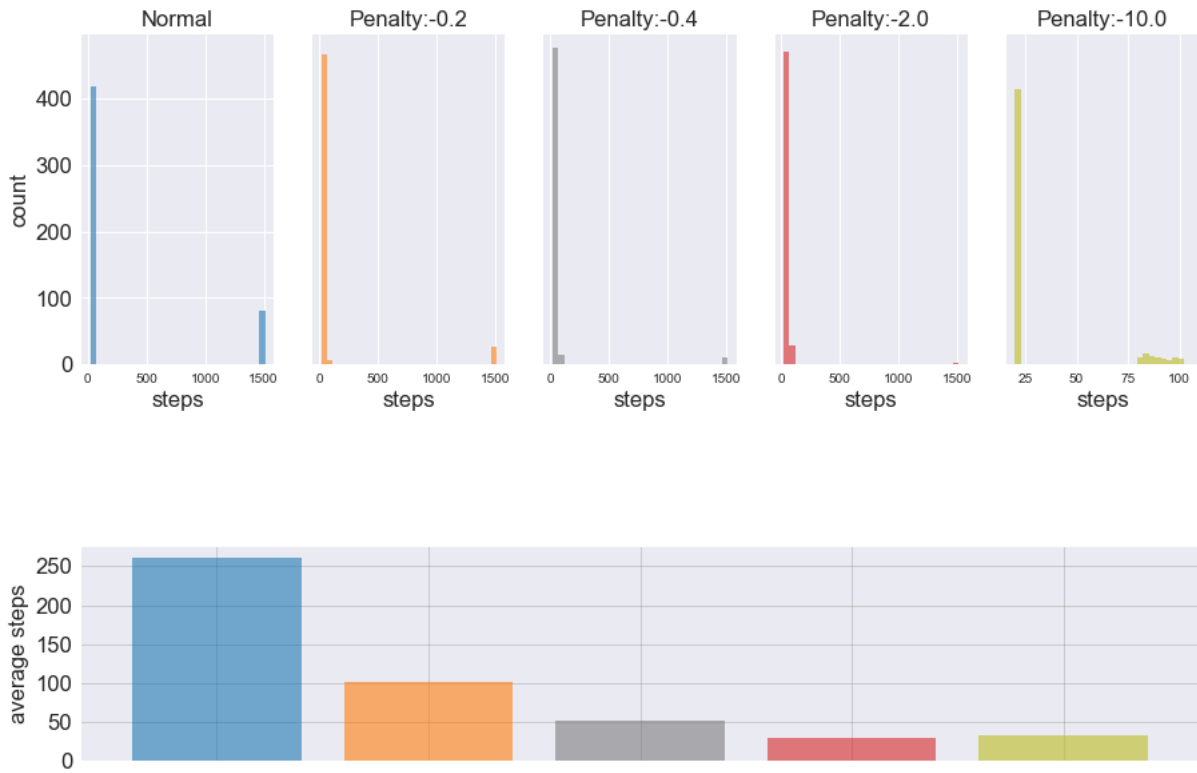


Figure 55: The distribution of steps and the average step.

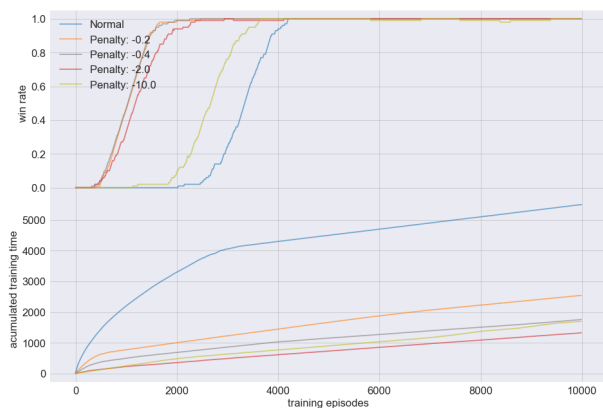


Figure 56: Win rate and training time of experiments on Maze No.9.

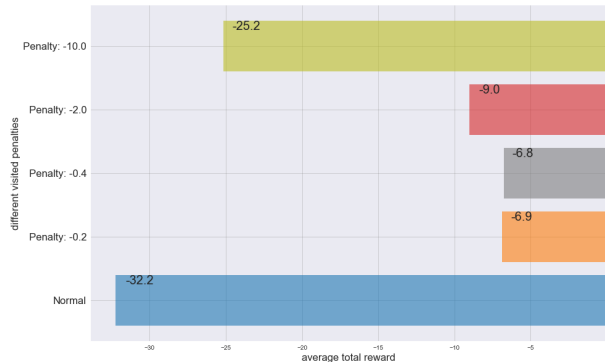


Figure 57: Average rewards of experiments on Maze No.9.

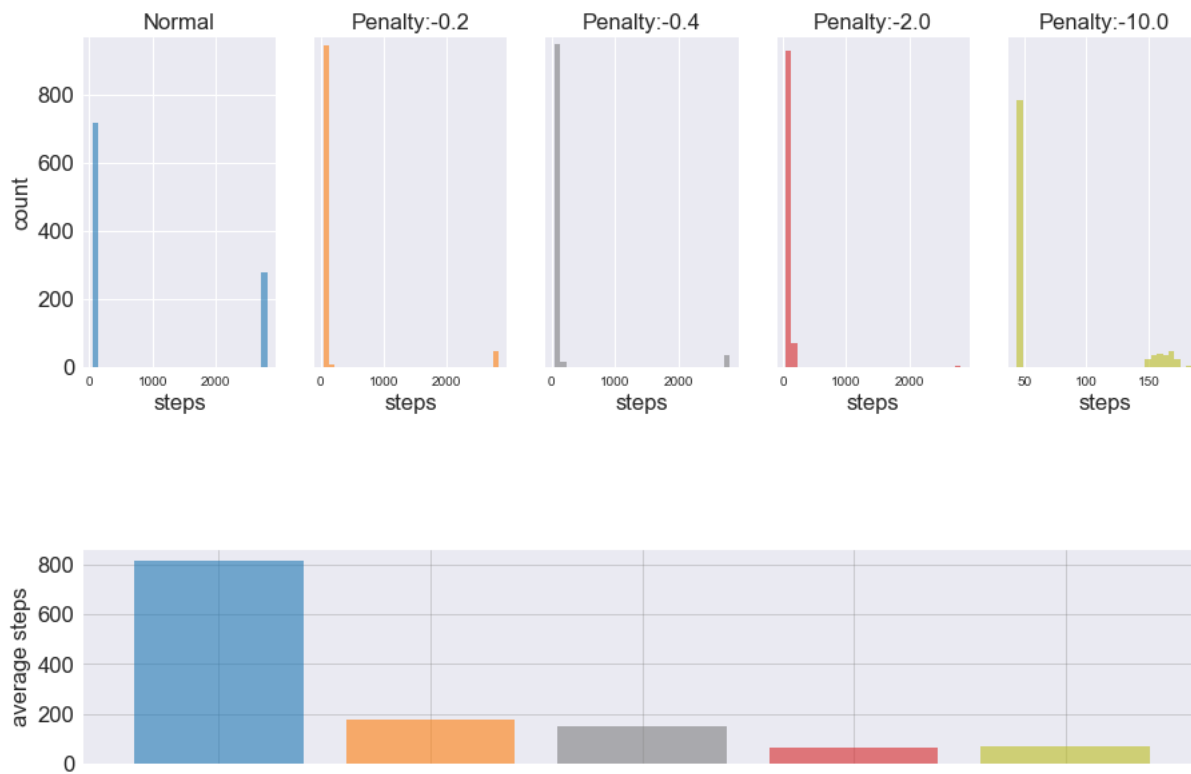


Figure 58: The distribution of steps and the average step.

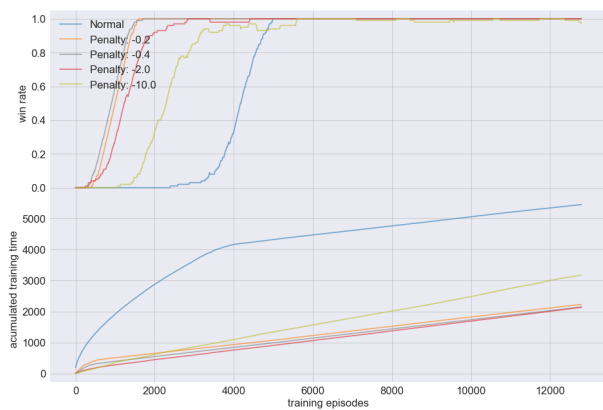


Figure 59: Win rate and training time of experiments on Maze No.10.

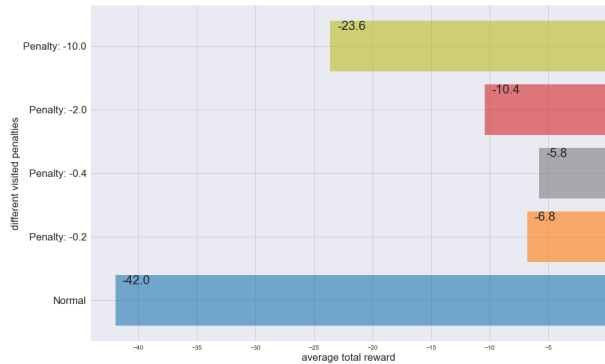


Figure 60: Average rewards of experiments on Maze No.10.

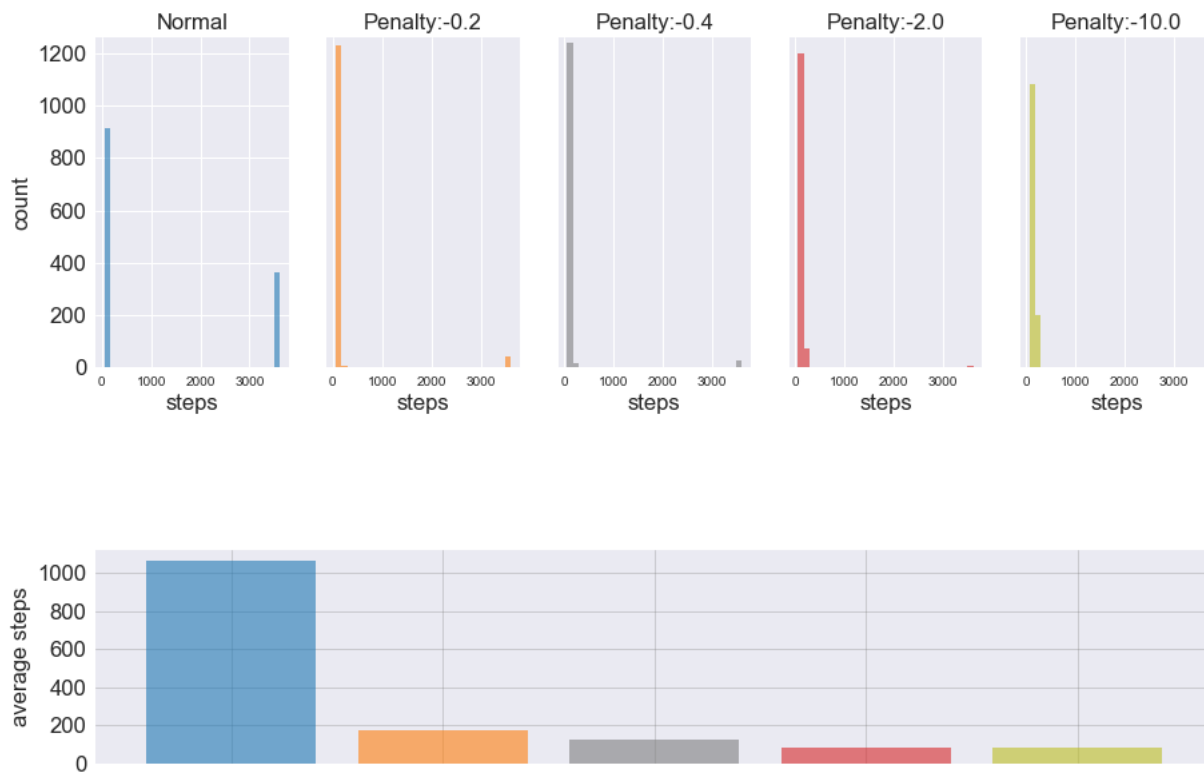


Figure 61: The distribution of steps and the average step.

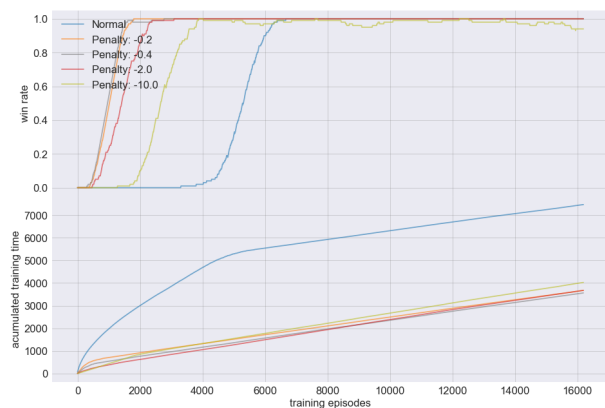


Figure 62: Win rate and training time of experiments on Maze No.11.

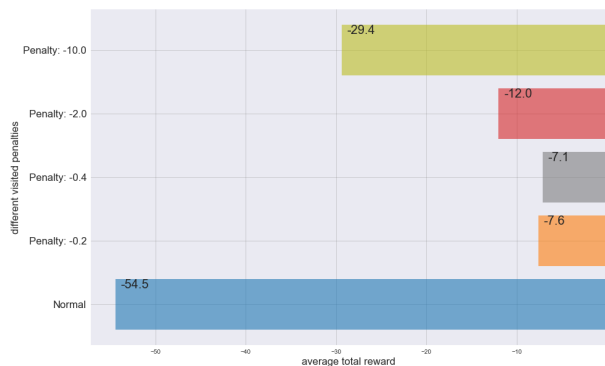


Figure 63: Average rewards of experiments on Maze No.11.

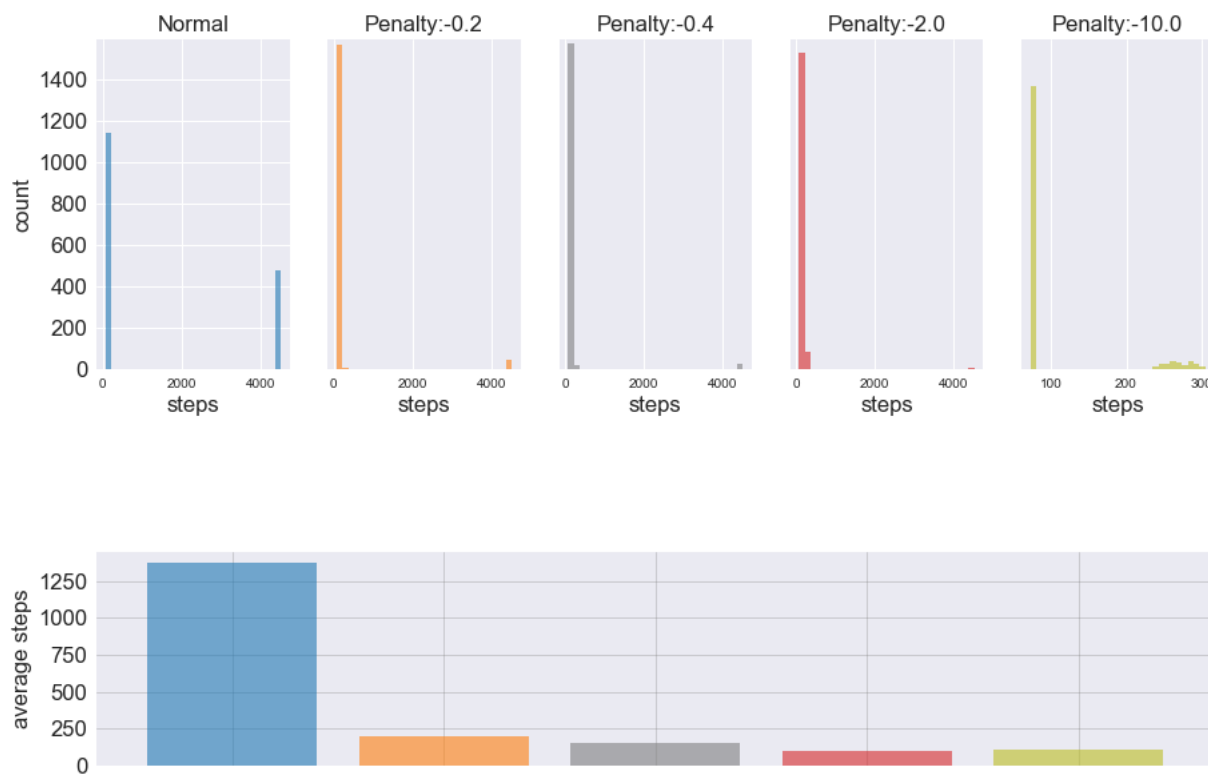


Figure 64: The distribution of steps and the average step.

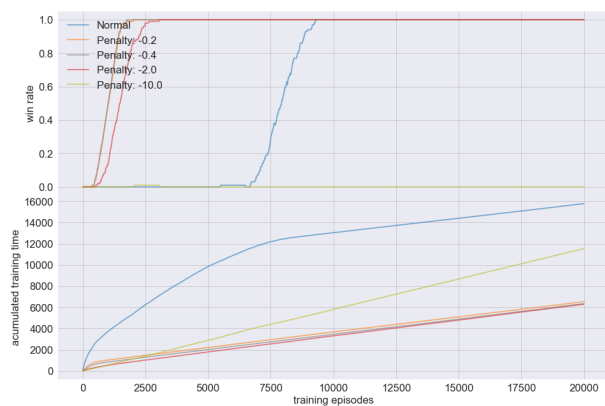


Figure 65: Win rate and training time of experiments on Maze No.12.

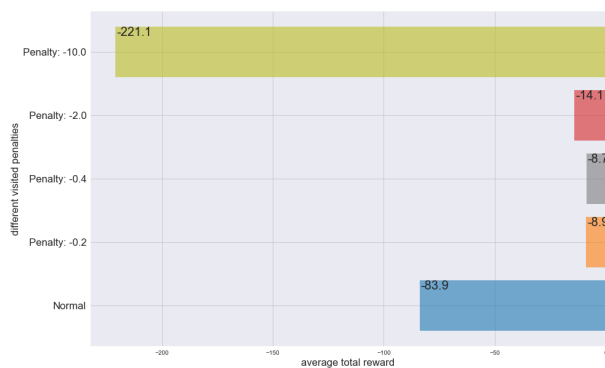


Figure 66: Average rewards of experiments on Maze No.12.

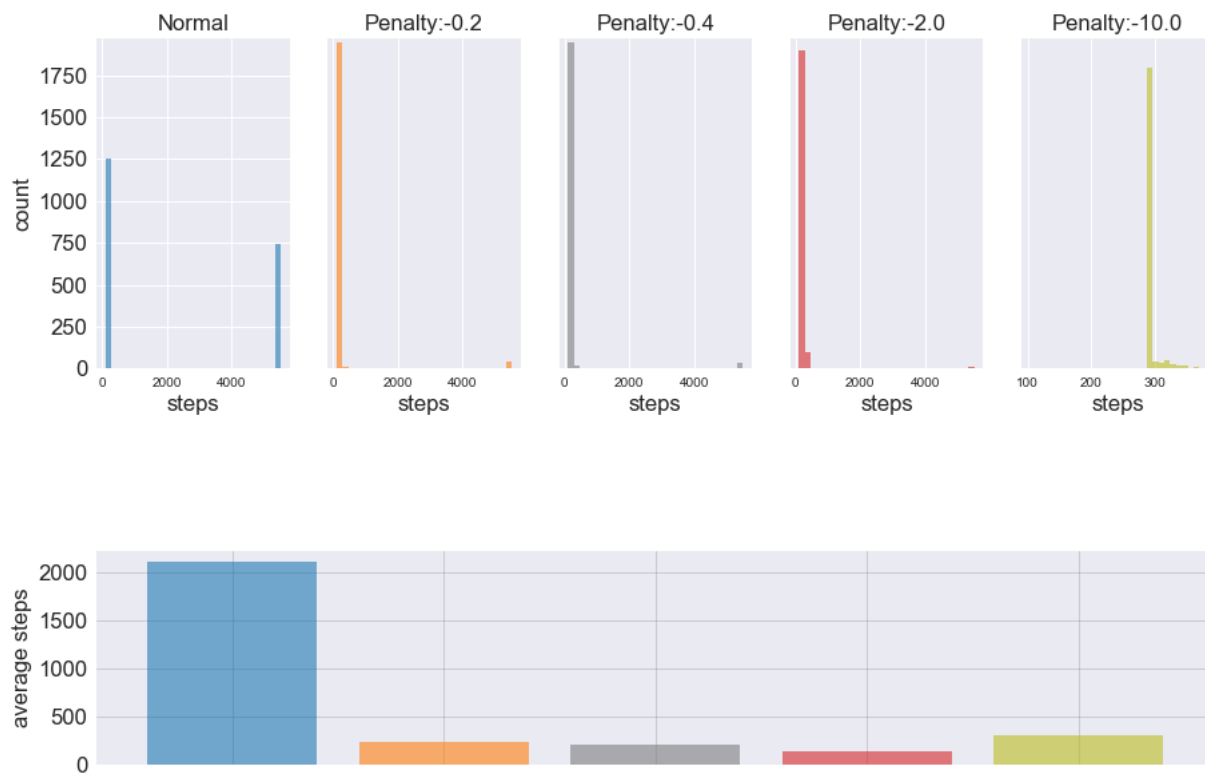


Figure 67: The distribution of steps and the average step.

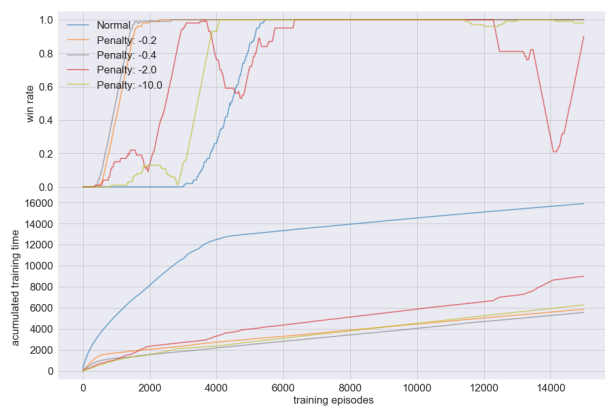


Figure 68: Win rate and training time of experiments on Maze No.13.

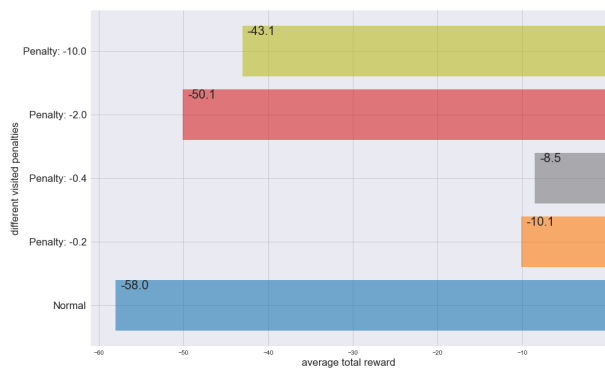


Figure 69: Average rewards of experiments on Maze No.13.

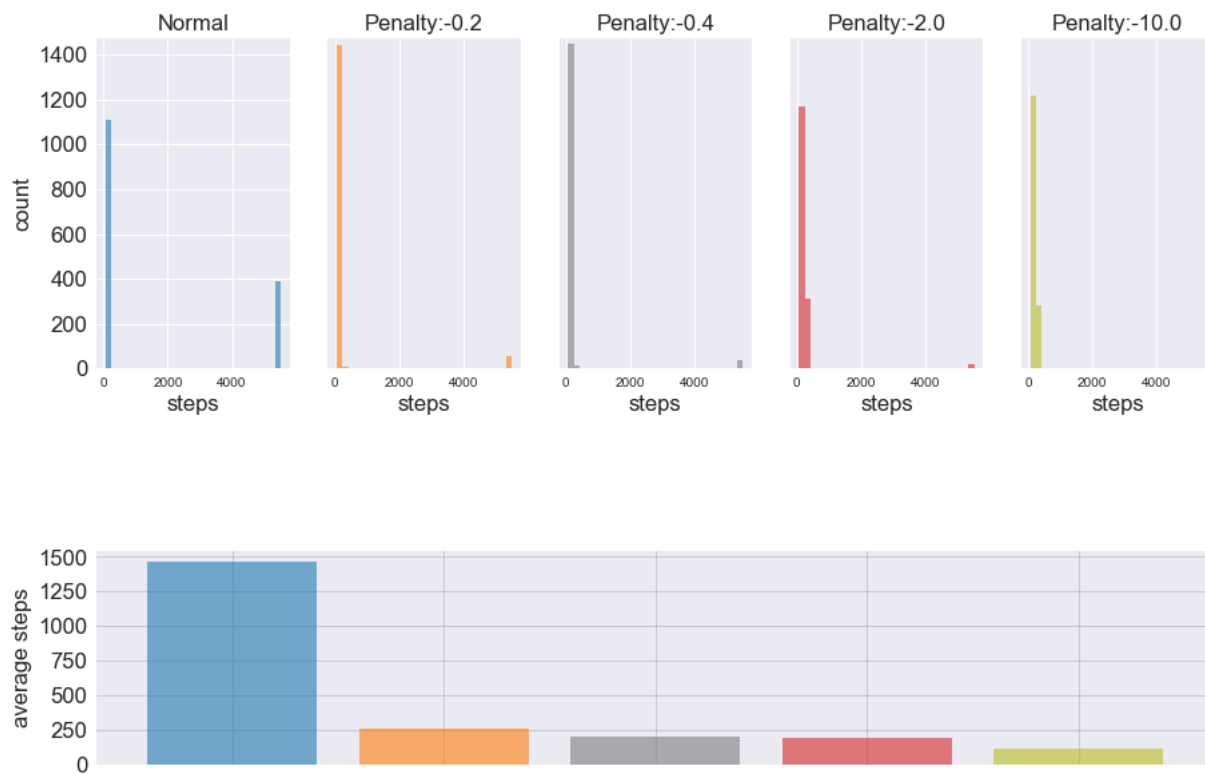


Figure 70: The distribution of steps and the average step.

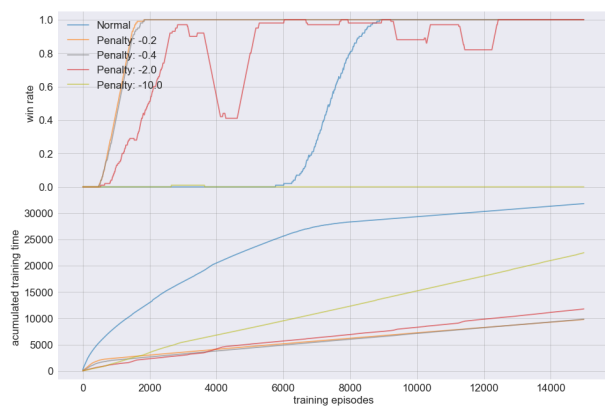


Figure 71: Win rate and training time of experiments on Maze No.14.

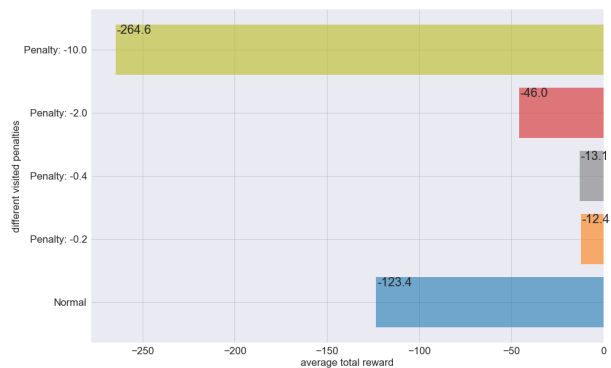


Figure 72: Average rewards of experiments on Maze No.14.

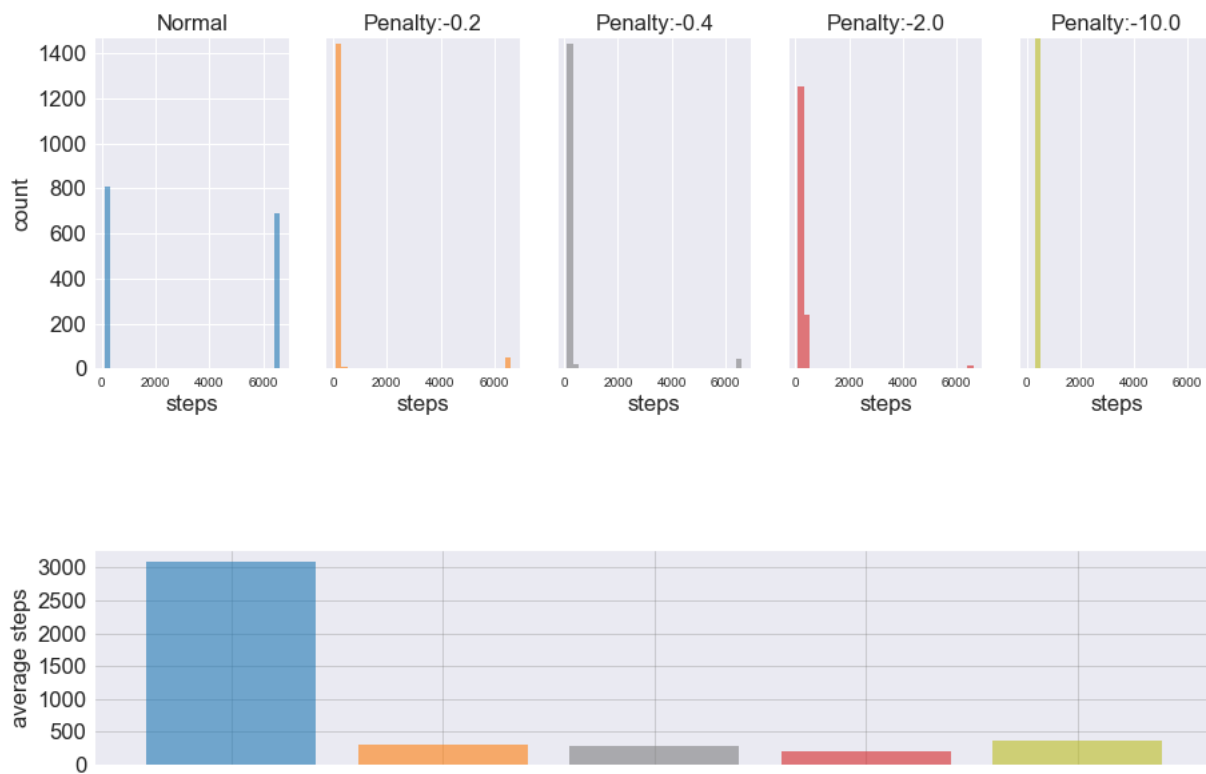


Figure 73: The distribution of steps and the average step.

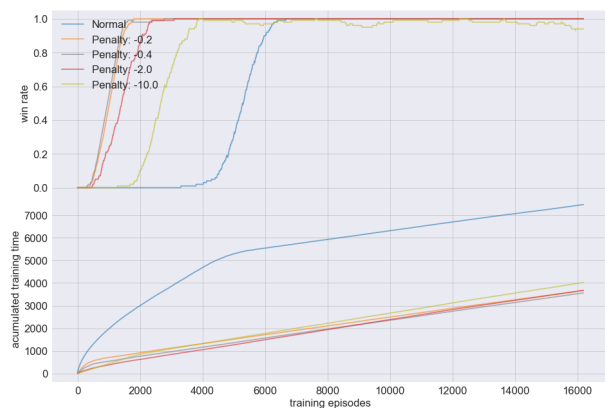


Figure 74: Win rate and training time of experiments on Maze No.15.

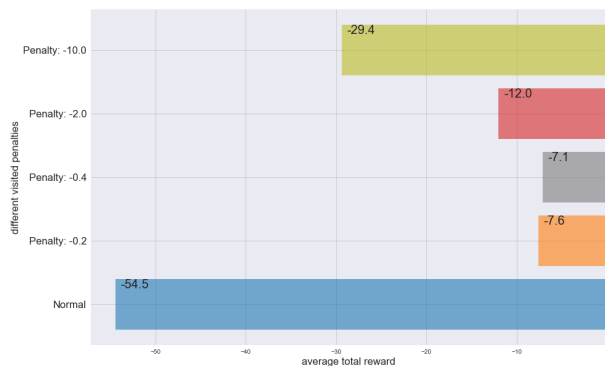


Figure 75: Average rewards of experiments on Maze No.15.

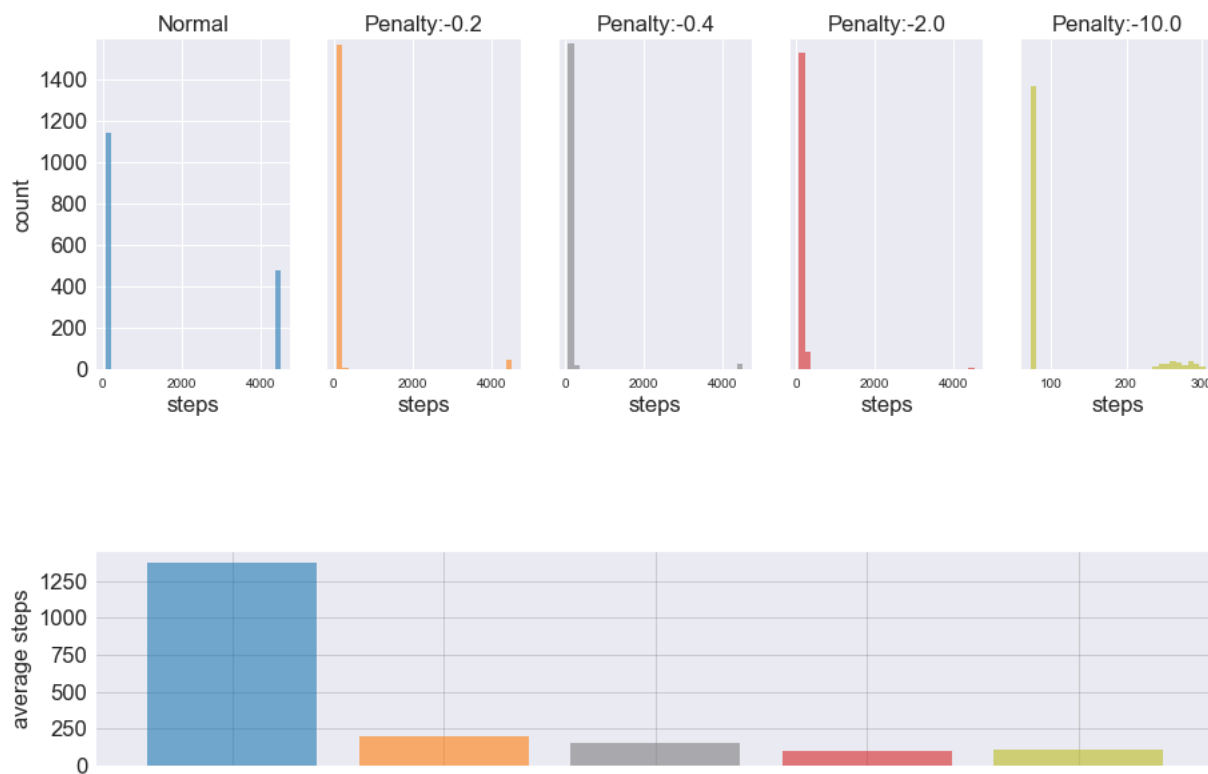


Figure 76: The distribution of steps and the average step.

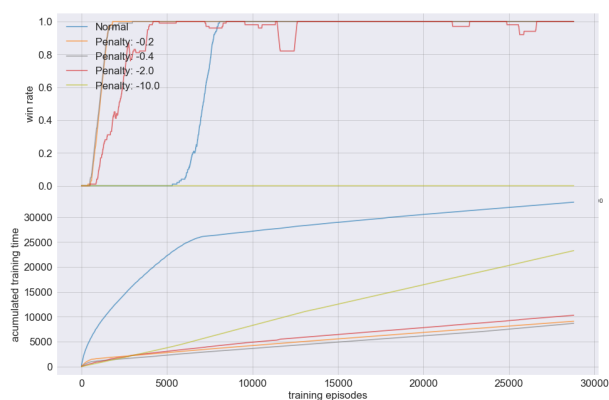


Figure 77: Win rate and training time of experiments on Maze No.16.

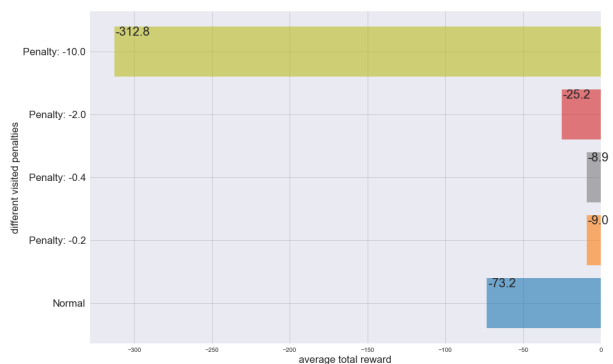


Figure 78: Average rewards of experiments on Maze No.16.

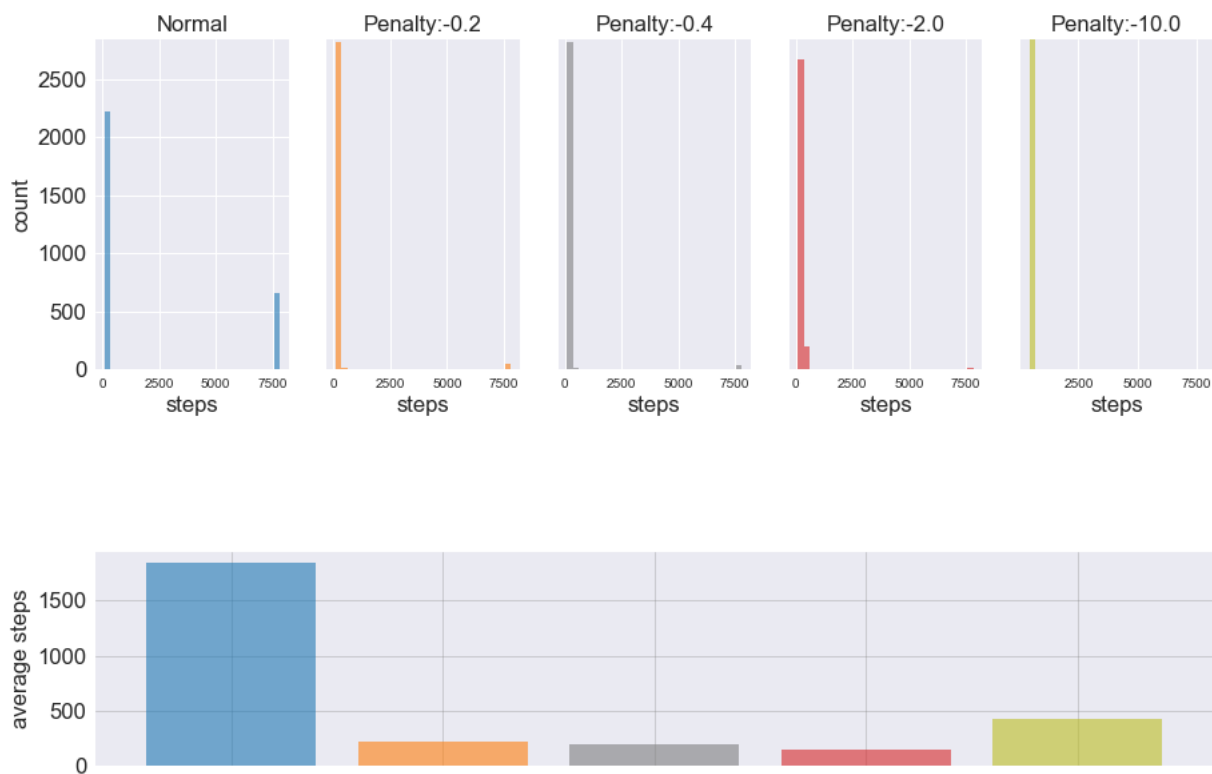


Figure 79: The distribution of steps and the average step.

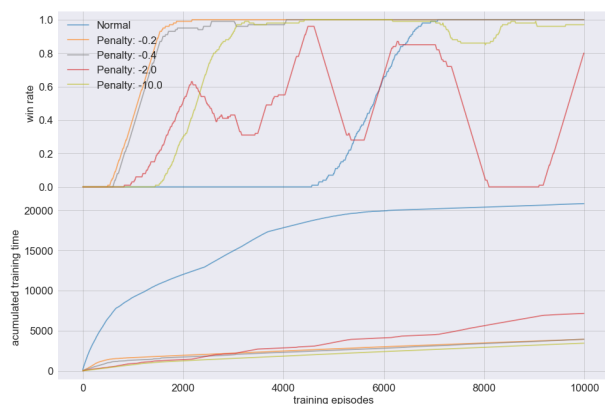


Figure 80: Win rate and training time of experiments on Maze No.17.

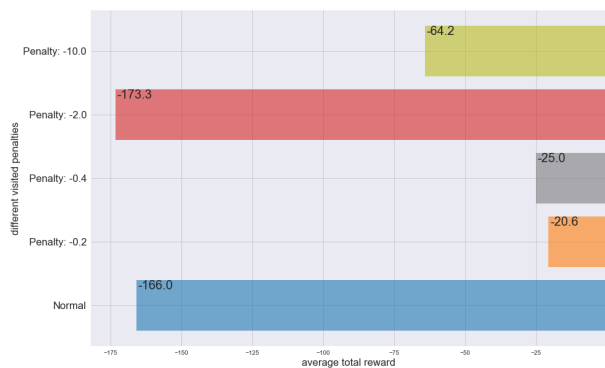


Figure 81: Average rewards of experiments on Maze No.17.

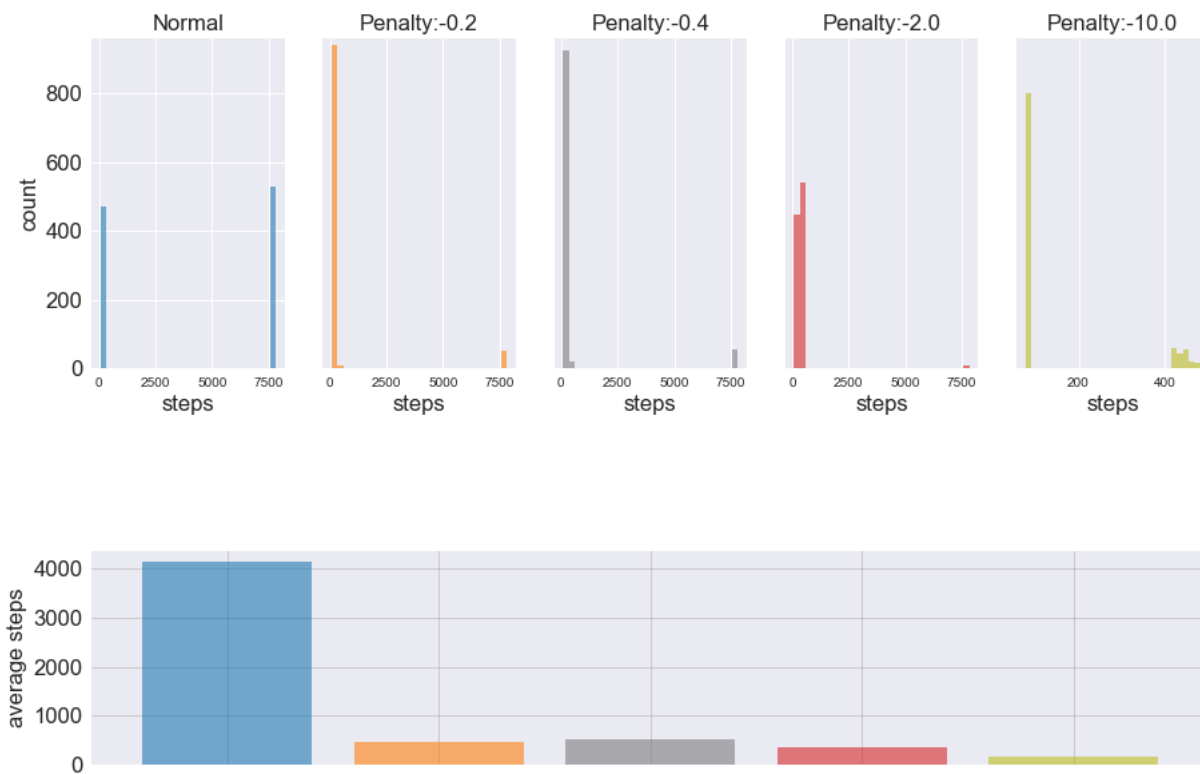


Figure 82: The distribution of steps and the average step.

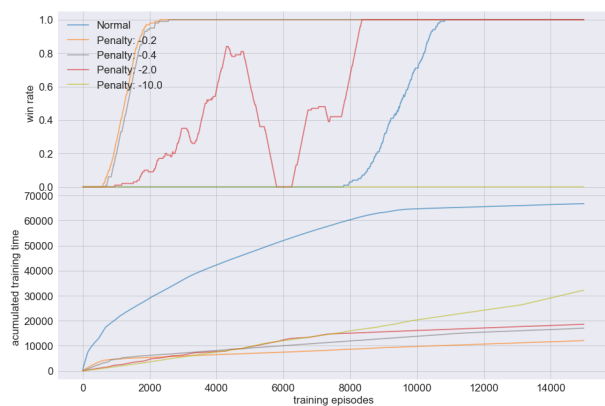


Figure 83: Win rate and training time of experiments on Maze No.18.

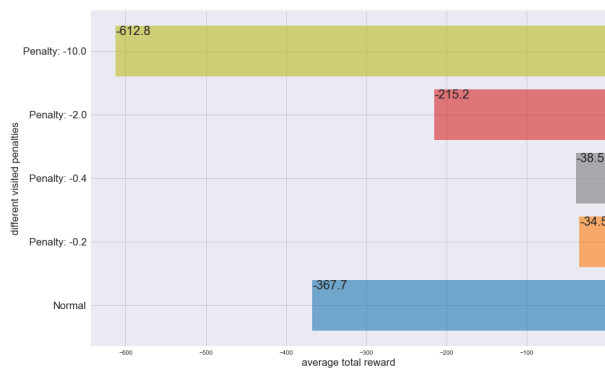


Figure 84: Average rewards of experiments on Maze No.18.

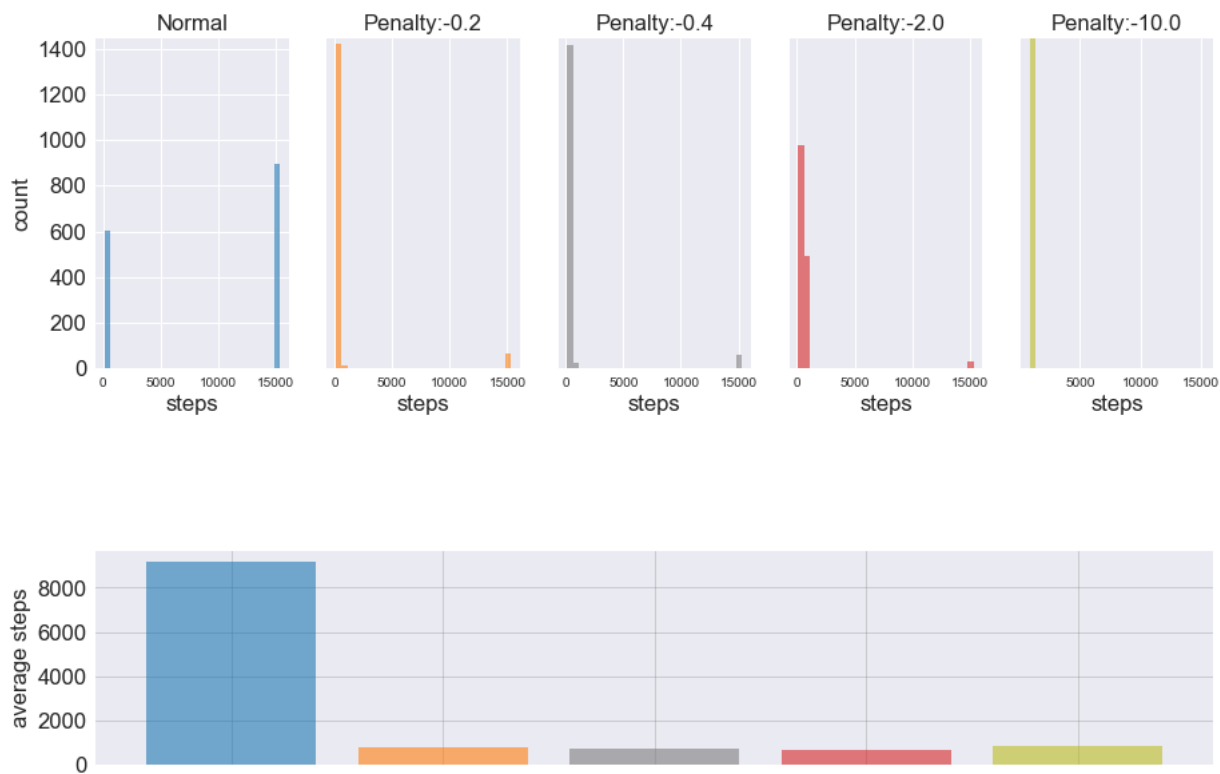


Figure 85: The distribution of steps and the average step.

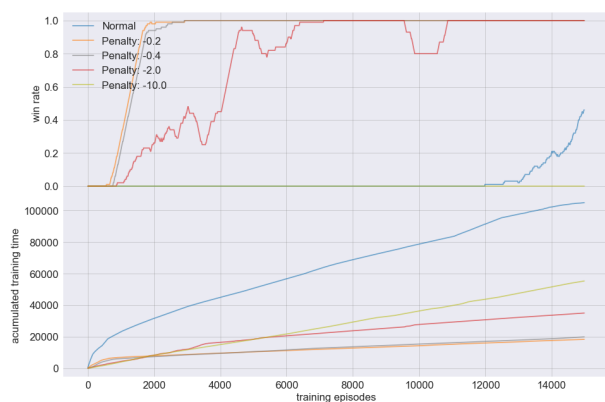


Figure 86: Win rate and training time of experiments on Maze No.19.

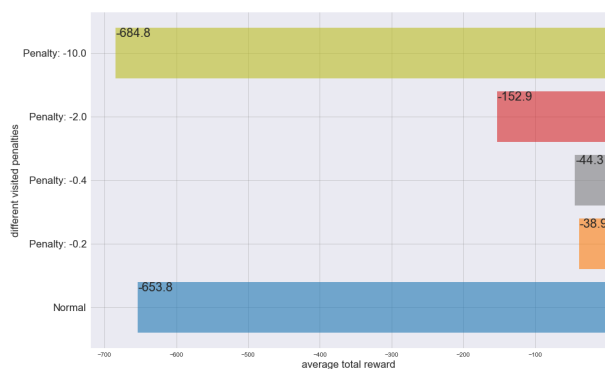


Figure 87: Average rewards of experiments on Maze No.19.

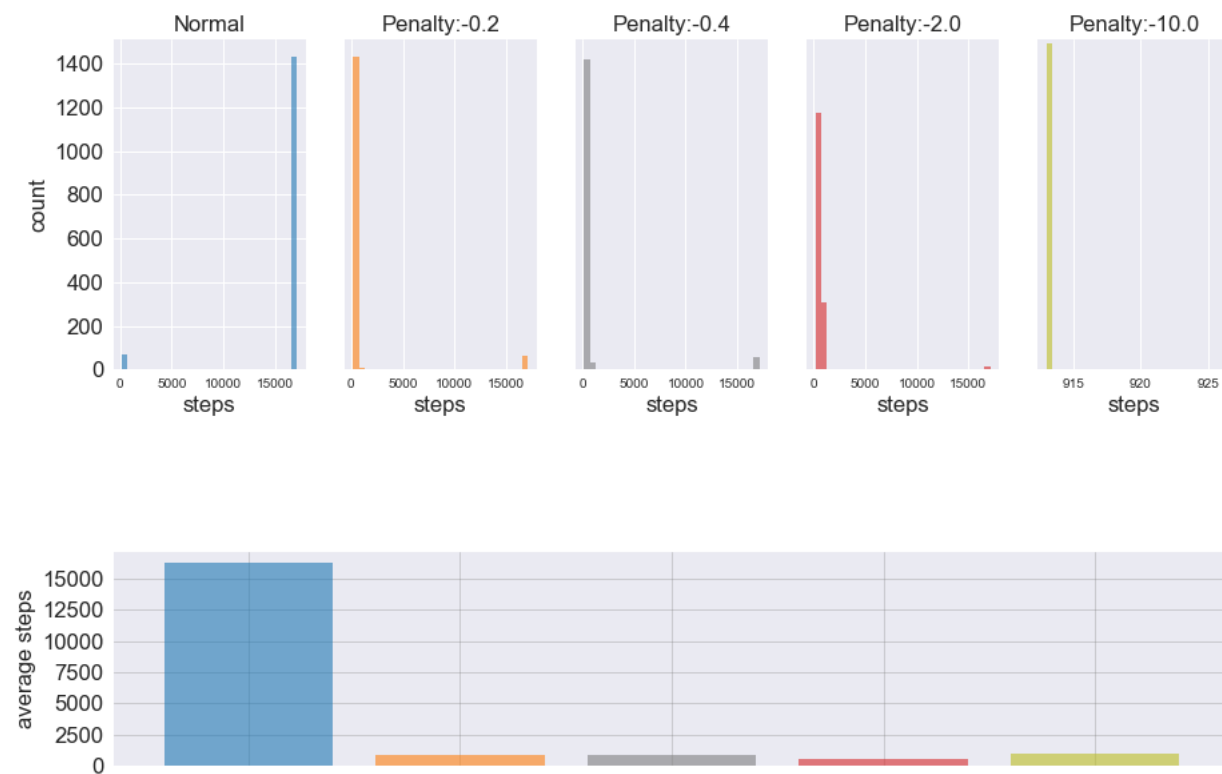


Figure 88: The distribution of steps and the average step.

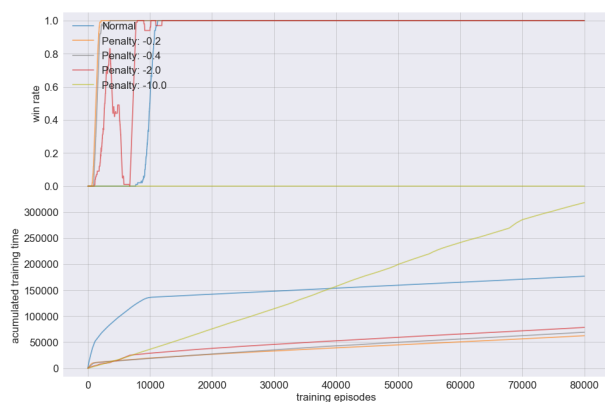


Figure 89: Win rate and training time of experiments on Maze No.20.

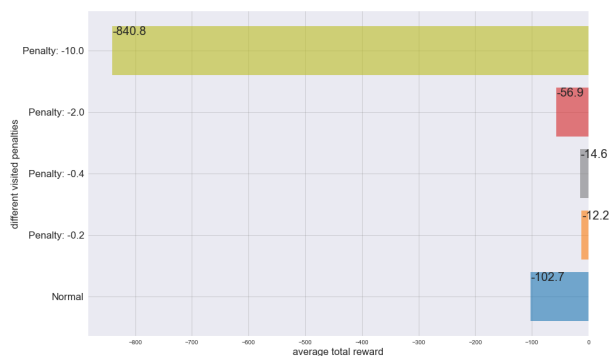


Figure 90: Average rewards of experiments on Maze No.20.

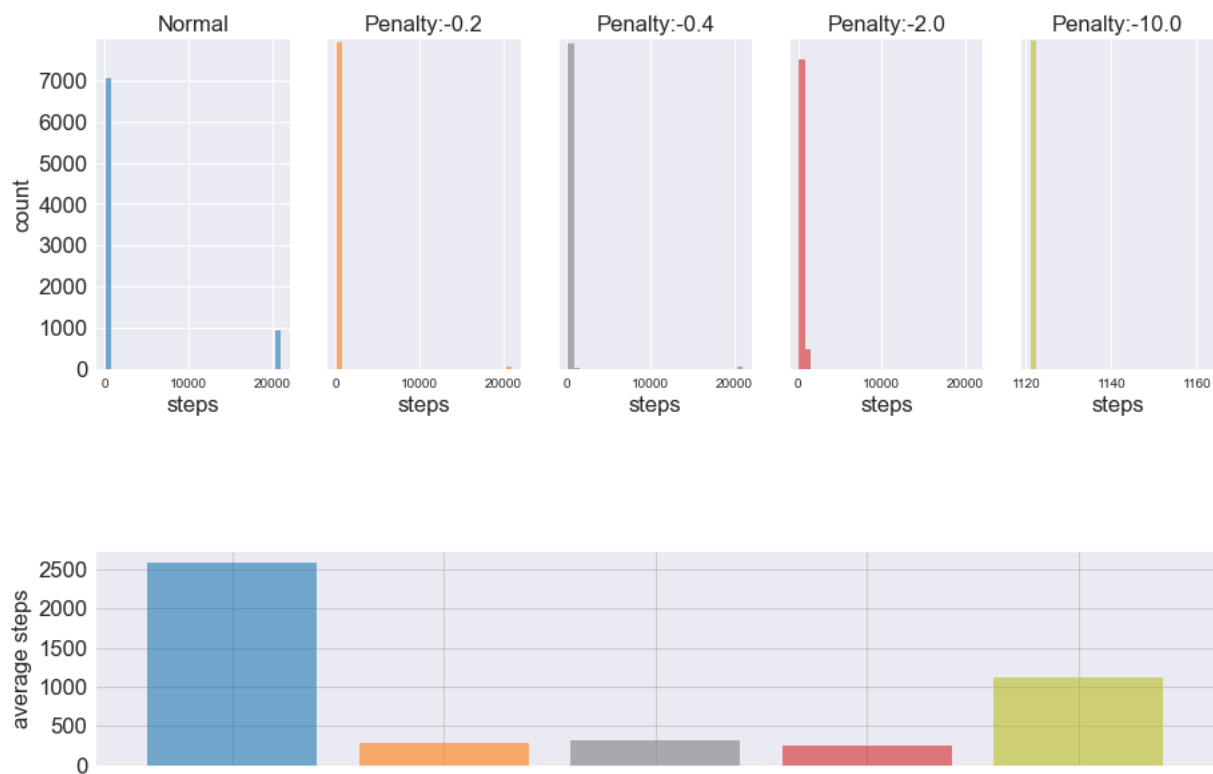


Figure 91: The distribution of steps and the average step.