

Spring 2020

An Introduction to Reinforcement Learning

Author
Kurath Samuel

May 4, 2020

Contents

1 Reinforcement Learning	2
1.1 Terminology	3
1.2 Approaches	8
1.3 Q-Learning	10
1.4 Deep Q-Learning	14
Glossary	19
Bibliography	20

1 Reinforcement Learning

Reinforcement learning is a computational approach to learn how to act under certain circumstances. The goal is to optimize the behavior of an **agent** by getting only the state and a reward from the **environment**. Hence, the main components of reinforcement learning are the environment and the agent. The environment represents the state, the possible transitions and should provide some kind of a reward. The agent is the learning part of the system and decides which action to take in relation to the state of the environment. In response to the action the agent gets a reward from the environment. This process is illustrated in figure 1.1.



Figure 1.1: *Reinforcement Learning*

Furthermore, reinforcement learning contains a policy, a reward, a value function and optionally a model of the environment.

The policy, π , defines the behavior of the agent at a certain moment depending on the current state and all the possible actions. It could be seen as a mapping from a state to an action.

The reward describes the goal of the problem. It is a numeric value often provided by the environment and it is given immediately after every action. Based on the reward, we update and improve the policy.

The value function, v , determines how good it is to be in a particular state. In contrast to the reward, the value function doesn't only depend on the current action. It does a rating based on experience and tries to include the estimated following reward.

The model is not an absolute necessary part of a reinforcement learning system but often it is very essential to taking good decisions. The aim of a model is to simulate the behavior of the environment. This simulation supports the planning mechanism by getting information about possible future situations that have not yet happened.

1.1 Terminology

The current section gives an overview of common concepts of reinforcement learning. For more information I recommend Richard Sutton's and Andrew Barto's excellent book [SB98] or the reinforcement learning course by David Silver [Sil18].

1.1.1 Markov Decision Process

A markov decision process is a method to formally describe an environment for reinforcement learning. It is a stochastic process that specifies transition probabilities from state to state. The objective of a markov decision process is to find a strategy to maximize the sum of future rewards. Further, the environment of a markov decision process is fully observable and it fulfills the markov property. The following list formalizes a markov decision process:

- S a finite set of all states
- A a finite set of all actions, the transitions between the states
- R reward associated with each transition
- $\gamma \in [0, 1]$ discount factor, quantifies the difference in importance between immediate rewards and future rewards

Markov Property The markov property means that the underlying process is memoryless [DD53]. This implies that the current state contains all the information about the history. Or in other words, all the future states and actions are independent on the past.

Example

Markov decision processes can be illustrated similarly to markov chains, also known as markov processes. But markov decision processes extend the markov chain with actions and rewards. An example is illustrated in figure 1.2. It shows a possible drive along a downhill track with different actions to take. Depending on those actions there are different probabilities to reach the goal or unfortunately go to the hospital.

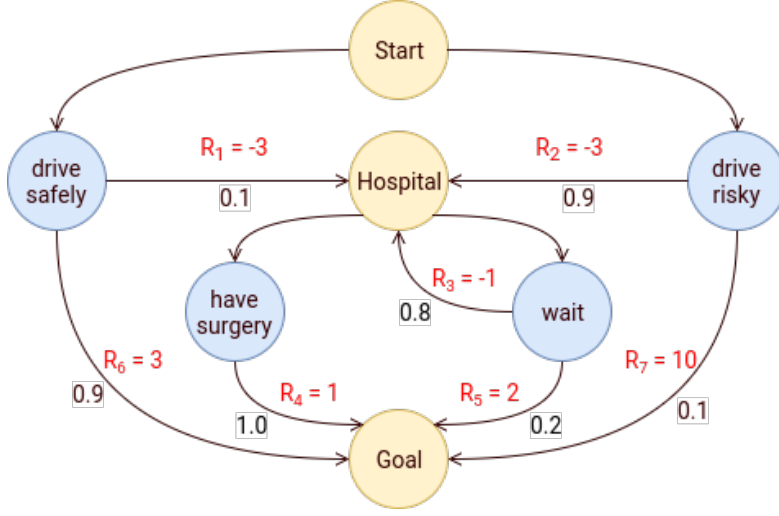


Figure 1.2: a markov decision process at the example of a downhill ride

In figure 1.2 the actions are illustrated as the blue circles, the states are represented by the yellow circles, the probabilities are written in black and the rewards are written in red.

1.1.2 Episode

If we have a look at an arbitrary problem and formalize the process as a markov decision process we get a repeated sequence of states, actions and rewards until a terminal state is reached, this is called an episode. An example is listed in (1.1).

$$S_0, A_0, R_0, S_1, A_1, R_1, S_2, A_2, R_2, \dots, S_n \quad (1.1)$$

where:

n: terminal state

Example In terms of our downhill example a possible episode is listed in (1.2).

$$S_{Start}, A_{drive_risky}, R_1, S_{Hospital}, A_{have_surgery}, R_4, S_{Goal} \quad (1.2)$$

1.1.3 Reward

As already mentioned, the goal of an agent is to maximize the cumulative received reward during an episode. This is based on the idea of the reward hypothesis [Sut18]:

That all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward).

1 Reinforcement Learning

The total reward of an episode can be obtained by applying the equation listed in (1.3).

$$R_{total} = \sum_{t=0}^n R_t \quad (1.3)$$

Example If we take the episode from our example in (1.2), we get the result that you see in (1.4).

$$R_{total} = R_1 + R_4 = -3 + 1 = -2 \quad (1.4)$$

Discounted reward

In reinforcement learning there often is a discount factor γ used to estimate the future reward. Reasons to use a discount factor are uncertainties about the future, avoiding infinity cycles in a markov process and there is a psychological effect as well. Humans tend to favor immediate reward over future one. The calculation of the discounted reward of an episode is illustrated in (1.5).

$$R_{discounted} = \sum_{t=0}^n \gamma^t R_t \quad (1.5)$$

Example Now imagine you are at the beginning of the downhill track and you think that your ride will end like the episode of example (1.2). Since you are not absolutely sure, you consider a discount factor γ in your calculation and set it to 0.9, resulting in (1.6).

$$R_{discounted} = \gamma^0 R_1 + \gamma^1 R_4 = 0.9^0 \cdot (-3) + 0.9^1 \cdot 1 = -2.1 \quad (1.6)$$

1.1.4 Value Functions

Reinforcement learning algorithms involve estimating value functions. Value functions are always dependent on a certain policy, π . There are two different functions we have to look at:

- state-value-function
- action-value-function

State-Value-Function

The state-value-function defines the expected value of the state S by following the policy π and estimates how good it is to be in a certain state.

$$V^\pi(s) = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \quad (1.7)$$

where:

E_π : is the expected value by following policy π
 S_t : determines the state at a certain timestamp

Example Let us reconsider the downhill example from figure 1.2 and define a simple policy, π , as follows.

Since we aren't professional cyclists, we always decide to drive safely and if we have an accident on the track and land in the hospital we have a surgery to get back on the track as fast as possible in order to reach the goal. This results in the policy listed in (1.8).

$$\pi : \begin{cases} S_{Start} \rightarrow A_{drive_safely} \\ S_{Hospital} \rightarrow A_{have_surgery} \end{cases} \quad (1.8)$$

Now we are able to apply the policy of our example and compute the state-value-function for the states. The calculation is illustrated at (1.9) and we set the discount factor γ to 0.9.

$$\begin{aligned} E_\pi(S_{Start}) &= 0.1 \cdot (-3) + 0.9 \cdot 3 = 2.4 \\ E_\pi(S_{Hospital}) &= 1.0 \cdot 1.0 = 1.0 \\ E_\pi(S_{Goal}) &= 0, \text{ the value of a terminal state is always zero} \\ V^\pi(S_{Start}) &= \gamma^0 \cdot E_\pi(S_{Start}) + \gamma^1 \cdot E_\pi(S_{Hospital}) + \gamma^2 \cdot E_\pi(S_{Goal}) \\ &= 0.9^0 \cdot 2.4 + 0.9^1 \cdot 1.0 + 0.9^2 \cdot 0 = 3.3 \\ V^\pi(S_{Hospital}) &= \gamma^0 \cdot E_\pi(S_{Hospital}) + \gamma^1 \cdot E_\pi(S_{Goal}) \\ &= 0.9^0 \cdot 1 + 0.9^1 \cdot 0 = 1.0 \end{aligned} \quad (1.9)$$

Action-Value-Function

The action-value-function expresses the expected value after taking an action a from state s and estimating how good it is to perform a certain action at a given state.

$$Q^\pi(s, a) = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right], \text{ for all } s \in S \quad (1.10)$$

where:

1 Reinforcement Learning

E_π : is the expected value by following policy π

S_t : determines the state at a certain timestamp

Example For the action-value-function we change our policy, π , to a more risky one.

This leads us to the policy listed in (1.11).

$$\pi : \begin{cases} S_{Start} \rightarrow A_{drive_risky} \\ S_{Hospital} \rightarrow A_{have_surgery} \end{cases} \quad (1.11)$$

Now the action-value-function is able to answer the question of, "How good is it if you take the risky driving strategy in policy π ?"

$$E_\pi(S_{Start}, A_{drive_risky}) = 0.9 \cdot (-3) + 0.1 \cdot 10 = -1.7$$

$$E_\pi(S_{Hospital}, A_{have_surgery}) = 1.0 \cdot 1.0 = 1.0$$

$$E_\pi(S_{Goal}) = 0, \text{ the value of a terminal state is always zero}$$

$$\begin{aligned} Q^\pi(S_{Start}, A_{drive_risky}) &= \gamma^0 \cdot E_\pi(S_{Start}, A_{drive_risky}) + \gamma^1 \cdot E_\pi(S_{Hospital}, A_{have_surgery}) + \gamma^2 \cdot E_\pi(S_{Goal}) \\ &= 0.9^0 \cdot -1.7 + 0.9^1 \cdot 1.0 + 0.9^2 \cdot 0 = -0.8 \end{aligned} \quad (1.12)$$

1.2 Approaches

The following section lists different approaches to handling reinforcement learning problems.

1.2.1 Dynamic Programming

Dynamic programming was first introduced by Richard Bellman [Bel54]. The main idea is to simplify a complicated problem, break it into smaller subproblems and repeat that process recursively. Examples of this are the Viterbi Algorithm [For73] or Dijkstra's Algorithm [Dij59] for the shortest path problem. If the state-transition probabilities and the reward function are given, these algorithms are able to compute an optimal policy under the assumption that the model is provided perfectly. In the worst case it takes polynomial time in the number of states and actions to find an optimality. In reinforcement learning, we are unfortunately often faced with problems without a perfect model and without the computational power to apply such techniques.

In terms of reinforcement learning and dynamic programming the subproblems are defined as:

- *Policy evaluation* is a description of the process used to compute the state-value-function for an arbitrary policy. It helps us to quantify how powerful a policy is.
- *Policy improvement* is the next step after evaluating a policy with the goal to make the policy better. For that purpose, we could use the action-value-function to take an other action at a certain state and determine if the new policy is an improvement.
- *Policy iteration* combines policy evaluation with policy improvement. It also iterates over these processes to find an optimal policy. A downside of policy iteration could be that you have to evaluate a policy at every step.
- *Value iteration* effectively combines, in each of its steps, one step of policy evaluation and one step of policy improvement.

1.2.2 Monte Carlo methods

In general a Monte Carlo method [MU49] describes any method that solves a problem by generating suitable random numbers and observes some properties based on them. If we apply Monte Carlo methods to reinforcement learning, the agent learns directly from experience on complete episodes. It is a model-free approach. That means that the value-function is unknown.

For instance, we could consider the downhill ride problem from figure 1.2 and remove the state-transition probabilities. Now we aren't able to determine the value-function. However, we could follow an arbitrary policy and observe the states we reach dependent on our chosen actions. If we repeat this process for enough time, we can estimate the

value-function by the gained experience. And in accordance to the law of large numbers it is even possible to find an optimal policy.

1.2.3 Temporal Difference Learning

Temporal difference learning is an enhancement of reinforcement learning that combines the Monte Carlo method and dynamic programming. Similar to Monte Carlo methods it learns by experience without a model of the problem and like dynamic programming it doesn't only adjust if the whole episode is finished. Hence, it is able to update the estimated value-function after every action it takes. Known temporal difference algorithms are Sarsa and Q-learning.

1.3 Q-Learning

Q-Learning is located in the area of model free reinforcement learning techniques [Wat89]. The goal is to learn the action-value-function Q that represents the optimal policy. The computation of Q as introduced by Watkins is listed in (1.13). If Q is known, you only have to select the action that gives the biggest reward for acting in an optimal manner.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \quad (1.13)$$

where:

Q : action-value-function
 S : state
 A : action
 t : step
 α : $(0, 1]$
 R : reward
 γ : discount factor
 a : all possible next states

1.3.1 Example

To get more familiar with Q-learning we apply the theory at an example. The example is based on the Grid World game, which aims to get as fast as possible to a certain field from a random starting position. It is illustrated in figure 1.3, and our particular version uses the following constraints:

- 3 x 3 world size
- $(0, 0)$ is the goal field
- allowed actions are left, right, up and down
- illegal actions are diagonal movements and leaving the grid


 (0, 0)	(1, 0)	(2, 0)
(0, 1)	(1, 1)	(2, 1)
(0, 2)	(1, 2)	(2, 2)

Figure 1.3: Grid World

Implementation

To illustrate how the Q-learning algorithm works, we will use an implementation in Python applied to the example of the Grid World game.

1. First, we define all the legal and illegal actions as you can see in listing 1.1.

```

1  def right(x, y):
2      return x + 1 if x < 2 else x, y
3
4
5  def left(x, y):
6      return x - 1 if x > 0 else x, y
7
8
9  def up(x, y):
10     return x, y - 1 if y > 0 else y
11
12
13 def down(x, y):
14     return x, y + 1 if y < 2 else y
15
16
17 actions = [left, right, up, down]
18
19 illegal_actions = [(0, 0, left), (0, 0, up), (1, 0, up), (2, 0, up),
20                  (2, 0, right), (0, 1, left), (0, 2, left), (0, 2, down),
21                  (1, 2, down), (2, 2, down), (2, 2, right), (2, 1, right)]

```

Listing 1.1: *action definition*

2. Second, we define the reward in relation to the state and the action. This is illustrated in 1.2. The goal state gets 100, illegal actions get -1 and the remaining are initialized with zeros.

```

1  reward = {(0, 0, left): 100, (0, 0, right): 0, (0, 0, up): 100, (0, 0, down): 0,
2           (0, 1, left): -1, (0, 1, right): 0, (0, 1, up): 100, (0, 1, down): 0,
3           (0, 2, left): -1, (0, 2, right): 0, (0, 2, up): 0, (0, 2, down): -1,
4           (1, 0, left): 100, (1, 0, right): 0, (1, 0, up): -1, (1, 0, down): 0,
5           (1, 1, left): 0, (1, 1, right): 0, (1, 1, up): 0, (1, 1, down): 0,
6           (1, 2, left): 0, (1, 2, right): 0, (1, 2, up): 0, (1, 2, down): -1,
7           (2, 0, left): 0, (2, 0, right): -1, (2, 0, up): -1, (2, 0, down): 0,
8           (2, 1, left): 0, (2, 1, right): -1, (2, 1, up): 0, (2, 1, down): 0,
9           (2, 2, left): 0, (2, 2, right): -1, (2, 2, up): 0, (2, 2, down): -1, }

```

Listing 1.2: *reward definition*

3. As you can see in listing 1.3, the agent initializes the Q values with zeros. It hosts a greedy strategy for the action choice and provides the train method.

```

1  class Agent:
2      def __init__(self, gamma=0.95, exploration_rate=0.9):
3          self.gamma = gamma
4          self.exploration_rate = exploration_rate
5          self.Q = {(0, 0, left): 0, (0, 0, right): 0, (0, 0, up): 0, (0, 0, down): 0,
6                  (0, 1, left): 0, (0, 1, right): 0, (0, 1, up): 0, (0, 1, down): 0,
7                  (0, 2, left): 0, (0, 2, right): 0, (0, 2, up): 0, (0, 2, down): 0,

```

1 Reinforcement Learning

```
8         (1, 0, left): 0, (1, 0, right): 0, (1, 0, up): 0, (1, 0, down): 0,
9         (1, 1, left): 0, (1, 1, right): 0, (1, 1, up): 0, (1, 1, down): 0,
10        (1, 2, left): 0, (1, 2, right): 0, (1, 2, up): 0, (1, 2, down): 0,
11        (2, 0, left): 0, (2, 0, right): 0, (2, 0, up): 0, (2, 0, down): 0,
12        (2, 1, left): 0, (2, 1, right): 0, (2, 1, up): 0, (2, 1, down): 0,
13        (2, 2, left): 0, (2, 2, right): 0, (2, 2, up): 0, (2, 2, down): 0, }
14
15    def max_next_q_value(self, x, y):
16        next_states = [action(x, y) for action in actions]
17        all_qs = []
18        for action in actions:
19            all_qs += [self.Q[(state[0], state[1], action)] for state in next_states]
20        return max(all_qs)
21
22    def choose_greedy_action(self, x, y):
23        q_values_actions = [(self.Q[x, y, action], action) for action in actions
24                             if (x, y, action) not in illegal_actions]
25        return choice([action for q_value, action in q_values_actions
26                       if max(q_values_actions, key=lambda z: z[0])[0] == q_value])
27
28    def train(self, episodes):
29        for _ in range(episodes):
30            x, y = randint(0, 2), randint(0, 2)
31            while not (x == 0 and y == 0):
32                action = choice(actions) if uniform(0, 1) >= self.exploration_rate \
33                       else self.choose_greedy_action(x, y)
34                next_x, next_y = action(x, y)
35                self.Q[(x, y, action)] = reward[x, y, action] + \
36                    self.gamma * self.max_next_q_value(next_x, next_y)
37                x, y = next_x, next_y
38
39    def run(self, x, y):
40        action_counter = 0
41        while not (x == 0 and y == 0):
42            action = self.choose_greedy_action(x, y)
43            x, y = action(x, y)
44            action_counter += 1
45        return action_counter
```

Listing 1.3: *the agent*

The agent is the core component of the algorithm and uses the train method to optimize the action-value-function, Q . The update of the action-value-function uses the equation listed in (1.13) with $\alpha = 1$ (listing 1.3, line 36). It is important to mention that the choice of the next action is implemented in a greedy way (listing 1.3, line 22). Further, there is an exploration mechanism added (listing 1.3, line 33) to make sure that all the fields are visited and you doesn't end in a local minima.

3. Finally, we are able to train and run the agent to verify our result. See 1.4.

```
1 agent = Agent()
2 agent.train(episodes=1000)
3
4 for start_x, start_y in [(2, 2), (2, 1), (1, 1)]:
5     print('Start:({},{}) It took {} actions to the goal.'
6           .format(start_x, start_y, agent.run(start_x, start_y)))
7
8 '''output
9 Start:(2,2) It took 4 actions to the goal.
10 Start:(2,1) It took 3 actions to the goal.
11 Start:(1,1) It took 2 actions to the goal.
12 '''
```

Listing 1.4: *train and run the agent***Result**

As you can see from the output in listing 1.4, the agent took a shortest path to the goal field by using the trained action-value-function and a greedy choice of the next action. The visualization of the Q values is illustrated in 1.1. If you start at any field in the grid and choose the highest action-value, you will find a shortest path through the world.

(0,0)	(1,0)	(2,0)
←: 0.0 →: 0.0 ↑: 0.0 ↓: 0.0	←: 2000.0 →: 1900.0 ↑: 1899.0 ↓: 1900.0	←: 1900.0 →: 1899.0 ↑: 1899.0 ↓: 1805.0
(0,1)	(1,1)	(2,1)
←: 1899.0 →: 1900.0 ↑: 2000.0 ↓: 1900.0	←: 1900.0 →: 1805.0 ↑: 1900.0 ↓: 1805.0	←: 1900.0 →: 1804.0 ↑: 1900.0 ↓: 1805.0
(0,2)	(1,2)	(2,2)
←: 1899.0 →: 1805.0 ↑: 1900.0 ↓: 1899.0	←: 1900.0 →: 1805.0 ↑: 1900.0 ↓: 1804.0	←: 1805.0 →: 1804.0 ↑: 1805.0 ↓: 1804.0

Table 1.1: *values of the action-value-function, Q*

1.4 Deep Q-Learning

Most of the problems aren't as small as our Grid World game from 1.3.1 and it is often impossible to compute or traverse the whole state space in an affordable amount of time. Hence, we need techniques that are good in generalization and are able to handle problems in a variety of forms. And this is exactly where deep learning meets reinforcement learning. Deep learning does a great job in generalization and in handling unknown situations. In particular, the neural networks represent and learn to depict the action-value-function. This enabled great successes in games like Go by DeepMind [Mni+15].

1.4.1 Functionality

Deep Q-Learning roughly works as follows:

1. First it randomly initializes our neural network, which represents our action-value-function.
2. Then we can simulate some games and memorize them.
3. After that, we are able to train the neural network based on the memorized games. This process is called experience replay.
4. Finally, we repeat this process until we are pleased with the behavior of the neural network or a certain number of episodes has been completed.

The full algorithm introduced by V. Mnih et al. [Mni+15] is listed in 1.

Algorithm 1 Deep Q Learning with experience replay

Initialize replay memory D to size N

Initialize action-value-function Q with random weights θ

Initialize target action-value-function \hat{Q} with random weights $\theta^- = \theta$

for episode = 1, M **do**

 Initialize state $s_1 = \{x_1\}$ and preprocess sequence $\phi_1 = \phi(s_1)$

for $t = 1, T$ **do**

 With probability ε select random action a_t

 Otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and process $\phi_{t+1} = \phi(s_{t+1})$

 Store transition $(\phi, a_t, r_t, \phi_{t+1})$ in D

 Sample a minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D

 Set $y_j := \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \cdot \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

 Perform a gradient step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to θ

 Every C steps reset $\hat{Q} = Q$

end for

end for

Now, I'd like to point out some of the details of the algorithm.

Experience replay

To train the neural network, we first store the current state, the taken action, the gained reward and the next state for all the steps of the game. After a lot of simulated games, we train the model based on random mini-batches of the memorized experiences.

Two neural networks

As you might have noticed, in the algorithm there are two neural network used. One is updated during the experience replay and the other does the predictions for the taken actions. When a certain amount of mini-batches are trained, the decision network will be updated. The reason for this process is that it helps to stabilize the learning of the non-linear function.

Loss function

As you can see in equation (1.14), deep Q-learning often uses a mean squared error loss.

$$loss = (r + \gamma \cdot \max_{a'} \hat{Q}(s', a') - Q(s, a))^2 \quad (1.14)$$

where:

- r: reward
- γ : discount factor
- a' : next action
- a: action action
- s' : next state
- s: state
- \hat{Q} : target network
- Q: prediction network

If you are using a library like Keras [Cho18], you only have to provide the state and the target value as input for the neural network and set the loss function to the mean squared error.

Exploration

To reduce the problem of getting stuck in a local minima, we take a random action with a certain possibility. In algorithm 1, this is shown with probability ϵ to select a random action a_t .

1.4.2 Example

To get a better understanding of deep Q-learning, let's reconsider our Grid World example from 1.3.1 and change the Q-learning action-value-function to a deep Q-learning action-value-function.

Implementation

1. Since neural networks work on NumPy, we need a helper function that brings the coordinates into the right shape. This is shown in listing 1.5.

```
1 def position_to_state(x, y):
2     state = np.zeros(16, dtype='float32')
3     state[4 * y + x] = 1.
4     return np.expand_dims(state, axis=0)
```

Listing 1.5: *position to state*

2. The agent stores played games to train the neural network. Further, it implements an act method that recommends actions based on the decisions of the neural network and provides the related neural network model. See in listing 1.6.

```
1 class Agent:
2     def __init__(self, gamma=0.95, exploration_rate=0.9):
3         self.gamma = gamma
4         self.exploration_rate = exploration_rate
5         self.memory = deque([], maxlen=1000)
6         self.model = self.get_model()
7         self.target_model = self.get_model()
8
9     def get_model(self):
10        model = Sequential()
11        model.add(Dense(24, input_shape=(16,), activation='relu'))
12        model.add(Dense(24, activation='relu'))
13        model.add(Dense(4, activation='linear'))
14        model.compile(loss='mse', optimizer=Adam(lr=0.01))
15        return model
16
17    def remember(self, state, action, reward, next_state, done):
18        self.memory.append((state, action, reward, next_state, done))
19
20    def replay(self):
21        batch = sample(self.memory, 100) if len(self.memory) > 100 else self.memory
22        for state, action, r, next_state, done in batch:
23            target = r
24            if not done:
25                state = position_to_state(state[0], state[1])
26                next_state = position_to_state(next_state[0], next_state[1])
27                target = r + self.gamma \
28                    * np.amax(self.target_model.predict(next_state)[0])
29            target_values = self.model.predict(state)
30            target_values[0][actions.index(action)] = target
31            self.model.fit(state, target_values, epochs=1, verbose=0)
32            self.target_model.set_weights(self.model.get_weights())
33
34    def act(self, x, y):
```

1 Reinforcement Learning

```
35     state = position_to_state(x, y)
36     if np.random.rand() >= self.exploration_rate: @\label{line:exploration}@
37         return choice(actions)
38     act_values = self.model.predict(state)
39     return actions[np.argmax(act_values[0])]
40
41     def train(self, episodes=1):
42         for _ in range(episodes):
43             x, y = randint(0, 2), randint(0, 2)
44             while not (x == 0 and y == 0):
45                 action = self.act(x, y)
46                 next_x, next_y = action(x, y)
47                 done = False if not (x == 0 and y == 0) else True
48                 self.remember(state=(x, y), action=action, reward=reward[x, y, action],
49                             next_state=(next_x, next_y), done=done)
50                 x, y = next_x, next_y
51             self.replay()
52
53     def run(self, x, y):
54         action_counter = 0
55         self.exploration_rate = 1.1
56         while not (x == 0 and y == 0):
57             action = self.act(x, y)
58             x, y = action(x, y)
59             action_counter += 1
60         return action_counter
```

Listing 1.6: *reinforcement learning agent*

Important corner points to mention about the agent are the fact that it uses two neural networks, a model to train after every decision and a sporadically updated target model. Further, there is the act method that takes with the probability of the exploration rate a random action or lets the neural network decide (listing 1.6, line 33).

2. To verify the result, we can train and run the agent. See listing 1.7.

```
1 agent = Agent()
2 agent.train(episodes=1000)
3
4 for start_x, start_y in [(2, 2), (2, 1), (1, 1)]:
5     print('Start:({},{}) It took {} actions to the goal.'
6         .format(start_x, start_y, agent.run(start_x, start_y)))
7
8     '''
9     Start:(2,2) It took 4 actions to the goal.
10    Start:(2,1) It took 3 actions to the goal.
11    Start:(1,1) It took 2 actions to the goal.
12    '''
```

Listing 1.7: *train agent*

As you can see in listing 1.7, after the training phase, the agent takes actions that led to an optimal path through the Grid World.

Result

To visualize the output, we let the neural network predict all the fields and plotted the values for all the possible actions. The result is listed in table 1.2. As you can see, if you follow a greedy strategy considering the values of the table, you end up with a shortest path through the Grid World.

(0,0)	(1,0)	(2,0)
←: 355.4 →: 329.5 ↑: 359.6 ↓: 328.5	←: 438.8 →: 402.5 ↑: 420.4 ↓: 398.5	←: 421.2 →: 388.0 ↑: 406.7 ↓: 384.6
(0,1)	(1,1)	(2,1)
←: 406.5 →: 385.5 ↑: 438.5 ↓: 389.1	←: 416.8 →: 381.9 ↑: 420.2 ↓: 378.1	←: 395.7 →: 365.4 ↑: 400.9 ↓: 363.3
(0,2)	(1,2)	(2,2)
←: 389.6 →: 369.7 ↑: 418.7 ↓: 373.5	←: 392.6 →: 363.7 ↑: 398.4 ↓: 362.3	←: 375.4 →: 352.9 ↑: 384.5 ↓: 354.4

Table 1.2: *values of the neural network*

Glossary

Go Go is strategy board game with the goal of surrounding more territory than your opponent. 14

loss function A quantity that represents a measure of success for the task at hand. 15

NumPy NumPy is the fundamental package for scientific computing with Python. 16

Sarsa State-action-reward-state-action is an algorithm belonging to the temporal difference techniques for learning a markov decision process policy. 9

Bibliography

- [Bel54] Richard Bellman. “The theory of dynamic programming”. In: *Bulletin of the American Mathematical Society* 60.6 (1954), pp. 503–515.
- [Cho18] François Chollet. *Keras: The Python Deep Learning library*. June 2018. URL: <https://keras.io/>.
- [DD53] Joseph L Doob and Joseph L Doob. *Stochastic processes*. Vol. 7. 2. Wiley New York, 1953.
- [Dij59] Edsger W Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.
- [For73] G David Forney. “The viterbi algorithm”. In: *Proceedings of the IEEE* 61.3 (1973), pp. 268–278.
- [Mni+15] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (2015), p. 529.
- [MU49] Nicholas Metropolis and Stanislaw Ulam. “The monte carlo method”. In: *Journal of the American statistical association* 44.247 (1949), pp. 335–341.
- [SB98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.
- [Sil18] David Silver. *UCL Course on RL*. June 2018. URL: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>.
- [Sut18] Rich Sutton. *The reward hypothesis*. June 2018. URL: <http://incompleteideas.net/rlai.cs.ualberta.ca/RLAI/rewardhypothesis.html>.
- [Wat89] Christopher John Cornish Hellaby Watkins. “Learning from delayed rewards”. PhD thesis. King’s College, Cambridge, 1989.