University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 24: November 15

Lecturer: Alexander Schwing Scribe: Mingyu Yang

1 Recap

1.1 Recap the MDP(Markov Decision Process)

Like HMM, MDP also has Markov property, which means given the present state, the future and the past are independent.

Unlike HMM, MDP also considers motion, which means current state is related to not only current state, but also current motion. The new $s^{'}$ is determined by state s and motion a.

An MDP is composed of a quadruples, $(S, \mathcal{A}_s, P(s^{'}|s,a), R(s^{'}|s,a))$

We want to perform actions according to a policy π^* so as to maximize the expected future reward with 3 methods:

- Exhaustive search
- Policy iteration
- Value iteration

And compute V^* and Q^* by Bellman optimality principle:

$$Q^*(s,a) = \sum_{s^{'} \in S} P(s^{'}|s,a)[R(s,a,s^{'}) + \max_{s^{'} \in \mathcal{A}_{s^{'}}} Q^*(s^{'}, a^{'})]$$

And use policy evaluation to evaluate fixed policy π

2 Q-learning

2.1 Idea

But in MDP, what could we do if

- Transition probabilities and rewards are not known
- No model available (model free RL)

Solutions:

- Run a simulator to collect experience tuples/sample $(s, a, r, s^{'})$
- Approximate transition probability using samples

This is the idea of Q-learning.

2.2 Algorithm sketch

- Obtain a sample transition $(s, a, r, s^{'})$
- Obtained sample suggests:

$$Q(s, a) \approx y(s, a, r, s^{'} = r + \max Q(s^{'}, a^{'}))$$

- To account for missing transition probability we keep running average

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha y(s, a, r, s^{'})$$

which could be desribed as Figure 1 [2]

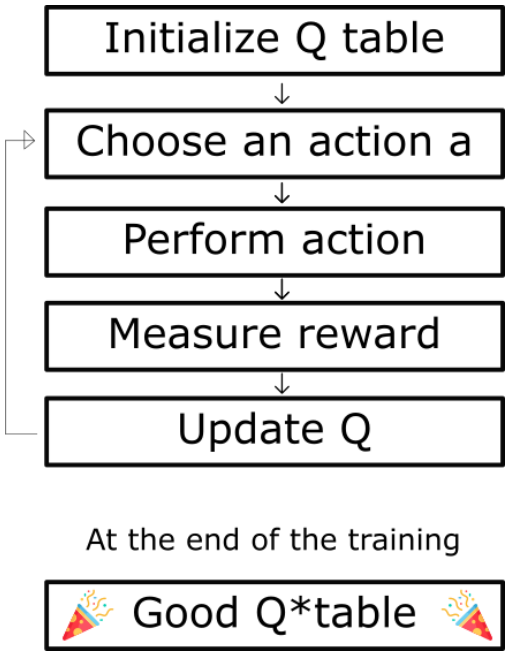

Figure 1: Q-learning process

2.3 Summary

- Known MDP
  - To compute V^*,Q^*,π^*: use value/policy iteration
  - To evaluate fixed policy π: use polic evaluation
- Unknown MDP: Model free
  - To compute V^*,Q^*,π^*: use Q-learning
  - To evaluate fixed policy π: use value learning

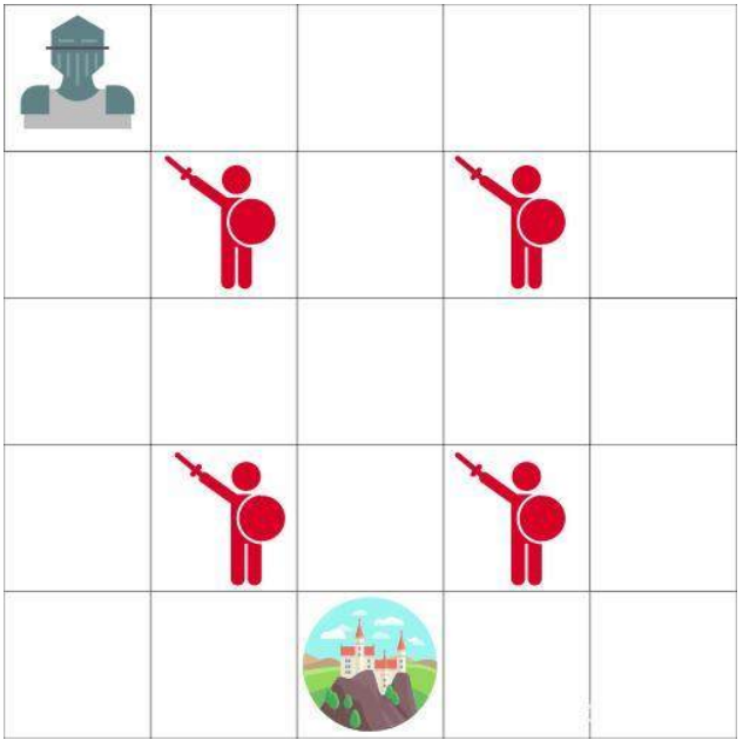

Figure 2: Knight and princess game

3 Examples

3.1 Example 1: Knight and princess [2]

Suppose you are a knight, and you need to suvive the princess in castle shown in the Figure 2.

You can move one tile at a time. The enemy can't, but land on the same tile as the enemy, and you will die. Your goal is to go the castle by the fastest route possible. This can be evaluated using a "points scoring" system.

- You lose -1 at each step (losing points at each step helps our agent to be fast).
- If you touch an enemy, you lose -100 points, and the episode ends.
- f you are in the castle you win, you get +100 points.

The question is: how do you create an agent that will be able to do that?

Here's a first strategy. Let say our agent tries to go to each tile, and then colors each tile. Green for "safe," and red if not.(shown in Figure 3)

Then, we can tell our agent to take only green tiles.

But the problem is that it's not really helpful. We don't know the best tile to take when green tiles are adjacent each other. So our agent can fall into an infinite loop by trying to find the castle. Thus, we introduce the Q-table.

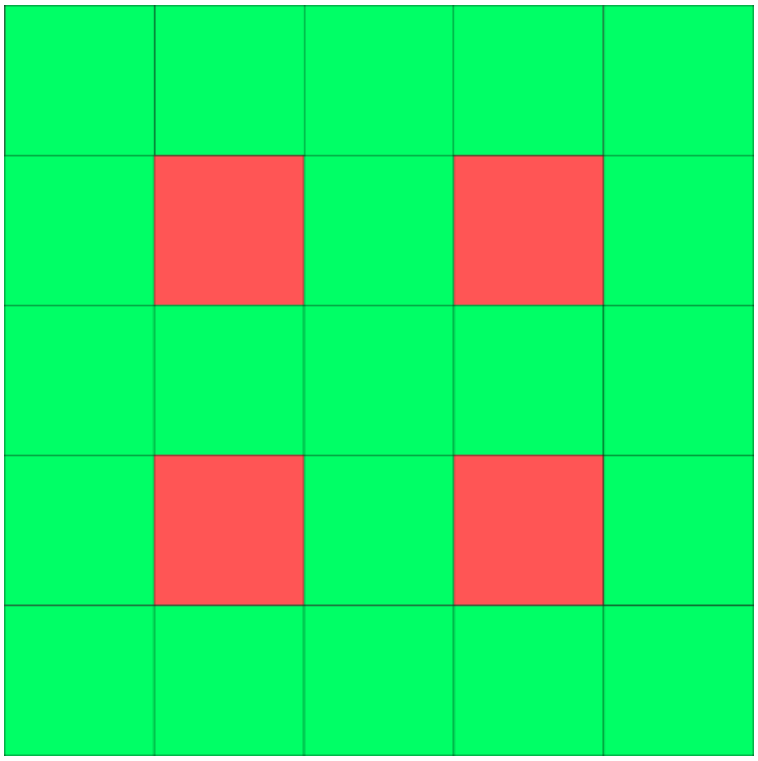

Figure 3: The same map, but colored in to show which tiles are safe to visit

Here's a second strategy: create a table where we'll calculate the maximum expected future reward, for each action at each state.

Thanks to that, we'll know what's the best action to take for each state.

Each state (tile) allows four possible actions. These are moving left, right, up, or down (shown in Figure 4)

In terms of computation, we can transform this grid into Q table. The columns will be the four actions (left, right, up, down). The rows will be the states. The value of each cell will be the maximum expected future reward for that given state and action (shown in Figure 5).

Each Q-table score will be the maximum expected future reward that I'll get if I take that action at that state with the best policy given.

Why do we say "with the policy given?" It's because we don't implement a policy. Instead, we just improve our Q-table to always choose the best action.

Think of this Q-table as a game "cheat sheet." Thanks to that, we know for each state (each line in the Q-table) what's the best action to take, by finding the highest score in that line.

Yeah! We solved the castle problem! But wait... How do we calculate the values for each element of the Q table?

To learn each value of this Q-table, we'll use the Q learning algorithm.

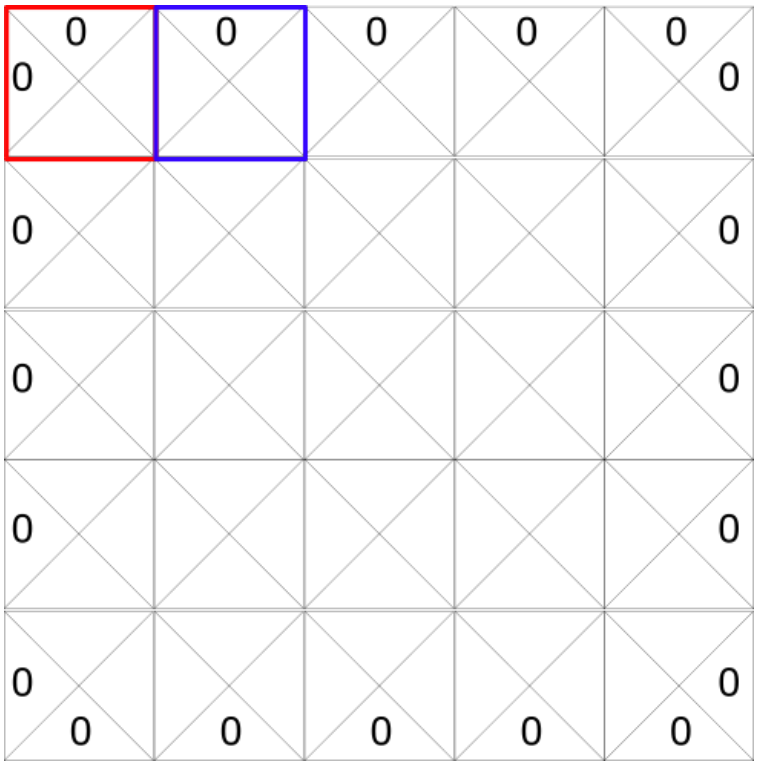

Figure 4: 0 are impossible moves (if you're in top left hand corner you can't go left or up)

The Action Value Function (or "Q-function") takes two inputs: "state" and "action." It returns the expected future reward of that action at that state. (Shown in Figure 6)
We can see this Q function as a reader that scrolls through the Q-table to find the line associated with our state, and the column associated with our action. It returns the Q value from the matching cell. This is the "expected future reward."

But before we explore the environment, the Q-table gives the same arbitrary fixed value (most of the time 0). As we explore the environment, the Q-table will give us a better and better approximation by iteratively updating Q(s,a) using the Bellman Equation.

Obey the process in 2.2, we can solve this game.

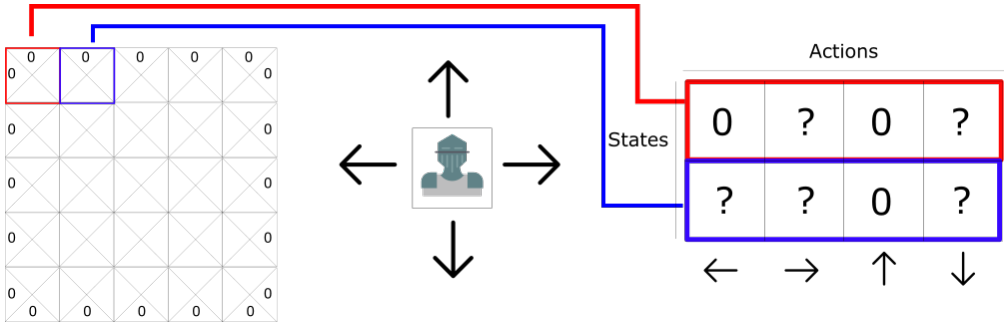

Figure 5: transfer into Q table

$$Q^{\pi}(s_t, a_t) = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... | s_t, a_t]$$

Q value for that state given that action

Expected discounted cumulative reward ...

given that state and that action

Figure 6: Q-function

Step 1: Initialize Q-values

We build a Q-table, with m cols (m= number of actions), and n rows (n = number of states). We initialize the values at 0. (shown in Figure 7)

Step 2: For life (or until learning is stopped)

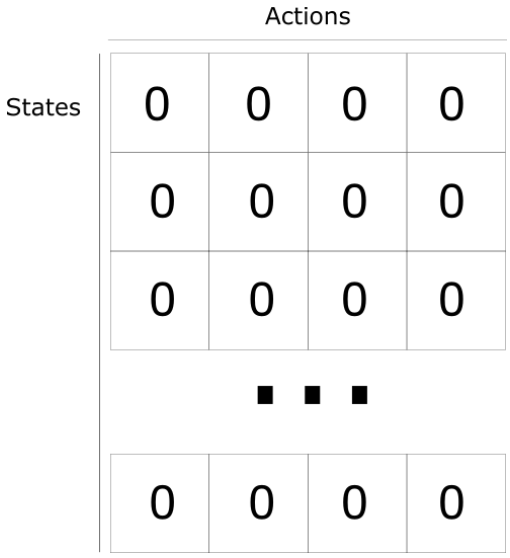

Figure 7: Initial Q-values

Steps 3 to 5 will be repeated until we reached a maximum number of episodes (specified by the user) or until we manually stop the training.

Step 3: Choose an action
Choose an action a in the current state s based on the current Q-value estimates.

In the beginning, we'll use the epsilon greedy strategy:

- We specify an exploration rate "epsilon," which we set to 1 in the beginning. This is the rate of steps that we'll do randomly. In the beginning, this rate must be at its highest value, because we don't know anything about the values in Q-table. This means we need to do a lot of exploration, by randomly choosing our actions.
- We generate a random number. If this number ¿ epsilon, then we will do "exploitation" (this means we use what we already know to select the best action at each step). Else, we'll do exploration.

- The idea is that we must have a big epsilon at the beginning of the training of the Q-function. Then, reduce it progressively as the agent becomes more confident at estimating Q-values.

Steps 4–5: Evaluate
Take the action a and observe the outcome state s' and reward r. Now update the function Q(s,a).

We take the action a that we chose in step 3, and then performing this action returns us a new state s' and a reward r

Then, to update Q(s,a) we use the Bellman equation (shown in Figure 8).

$$\underbrace{NewQ(s,a)}_{\text{New Q value for that state and that action}} = \underbrace{Q(s,a)}_{\text{Current Q value}} + \alpha[\underbrace{R(s,a)}_{\text{Reward for taking that action at that state}} + \gamma \underbrace{max\, Q'(s',a')}_{\text{Maximum expected future reward given the new s' and all possible actions at that new state}} - Q(s,a)]$$

Learning Rate

Discount rate

Figure 8: Bellman equation

3.2 Example 2: Mouse and cheese [2]

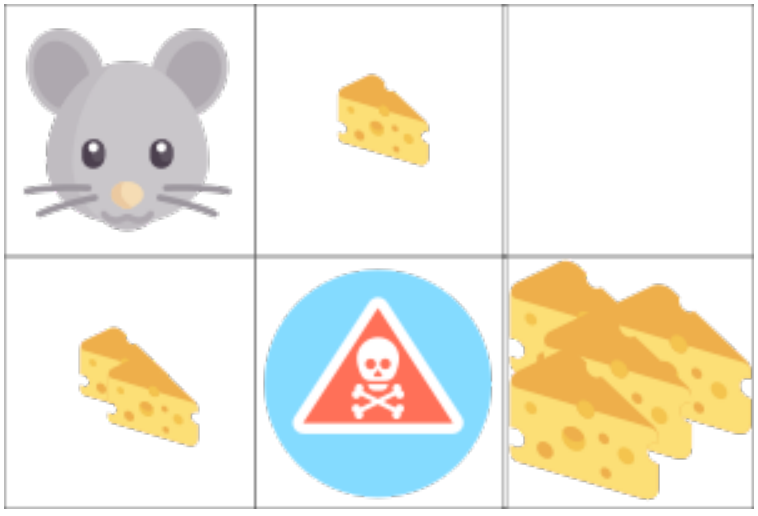

Figure 9: Mouse and cheese game

Suppose we play a game named 'mouse and cheese', which shown in Figure 9

- One cheese = +1
- Two cheese = +2
- Big pile of cheese = +10 (end of the episode)
- If you eat rat poison =-10 (end of the episode)

	←	→	↑	↓
Start	0	0	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Figure 10: The initialized Q-table

Step 1:Init Q-table(shown in Figure 10)

Step 2: Choose an action From the starting position, you can choose between going right or down. Because we have a big epsilon rate (since we don't know anything about the environment yet), we choose randomly. For example... move right (shown in Figure 11).

Figure 11: Move right

We found a piece of cheese (+1), and we can now update the Q-value of being at start and going right. We do this by using the Bellman equation.

Steps 4–5: Update the Q-function

First, we calculate the change in Q value ΔQ(start, right) Then we add the initial Q value to the ΔQ(start, right) multiplied by a learning rate.

Think of the learning rate as a way of how quickly a network abandons the former value for

the new. If the learning rate is 1, the new estimate will be the new Q-value.

	←	→	↑	↓
Start	0	0.1	0	0
Small cheese	0	0	0	0
Nothing	0	0	0	0
2 small cheese	0	0	0	0
Death	0	0	0	0
Big cheese	0	0	0	0

Figure 12: The updated Q-table

We've just updated our first Q value. Now we need to do that again and again until the learning is stopped.

3.3 Example 3: Get out of room [1]

Suppose we have 5 rooms in a building connected by doors as shown in the figure below. We'll number each room 0 through 4. The outside of the building can be thought of as one big room (5). Notice that doors 1 and 4 lead into the building from room 5 (outside).

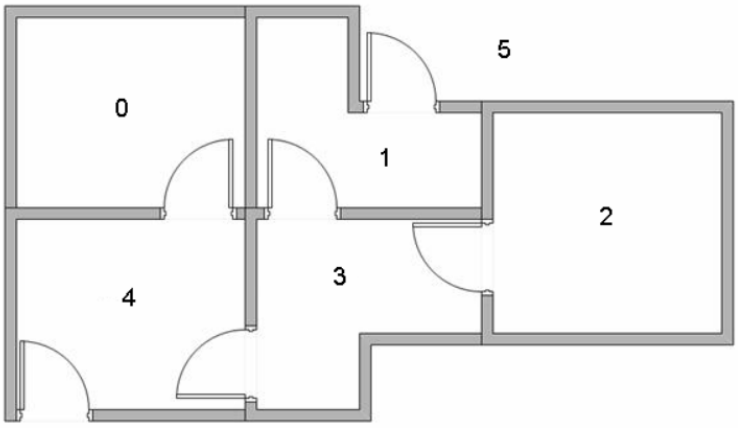

Figure 13: The room map

We can represent the rooms on a graph, each room as a node, and each door as a link, as shown in Figure 14.

For this example, we'd like to put an agent in any room, and from that room, go outside the building (this will be our target room). In other words, the goal room is number 5. To set this room as a goal, we'll associate a reward value to each door (i.e. link between nodes). The doors that lead

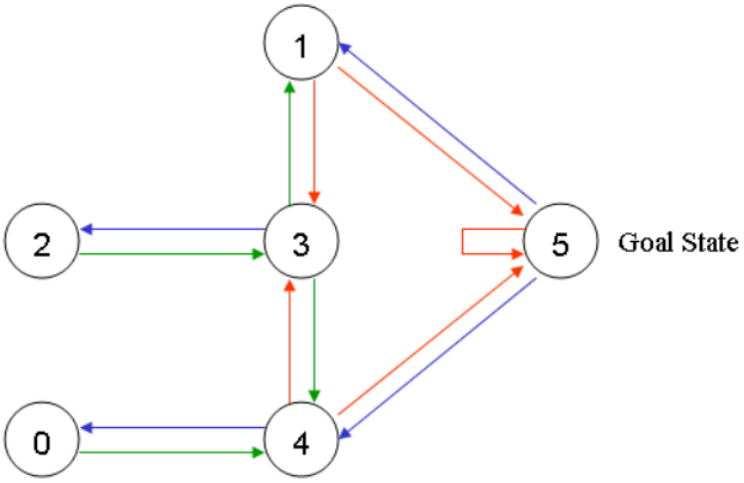

Figure 14: The room graph

immediately to the goal have an instant reward of 100. Other doors not directly connected to the target room have zero reward. Because doors are two-way (0 leads to 4, and 4 leads back to 0), two arrows are assigned to each room. Each arrow contains an instant reward value, as shown in Figure 15:

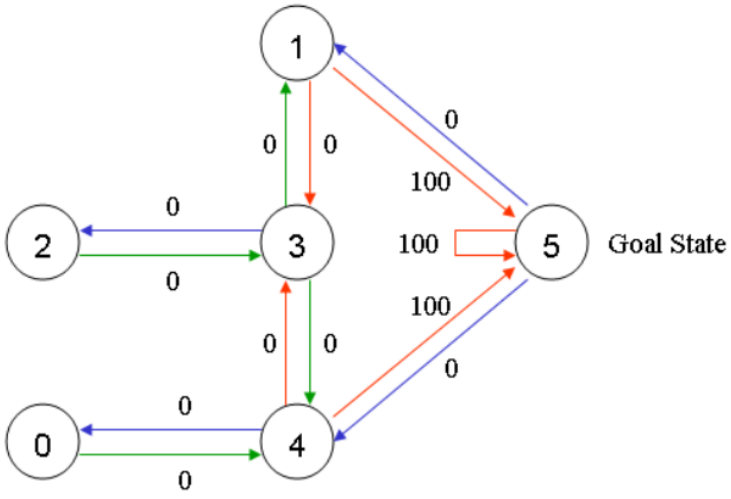

Figure 15: Rewarded room graph

Of course, Room 5 loops back to itself with a reward of 100, and all other direct connections to the goal room carry a reward of 100. In Q-learning, the goal is to reach the state with the highest reward, so that if the agent arrives at the goal, it will remain there forever. This type of goal is called an "absorbing goal".

Imagine our agent as a dumb virtual robot that can learn through experience. The agent can pass from one room to another but has no knowledge of the environment, and doesn't know which sequence of doors lead to the outside.
Suppose we want to model some kind of simple evacuation of an agent from any room in the building. Now suppose we have an agent in Room 2 and we want the agent to learn to reach outside the

house (5).
We'll call each room, including outside, a "state", and the agent's movement from one room to another will be an "action". In our diagram, a "state" is depicted as a node, while "action" is represented by the arrows.

Suppose the agent is in state 2. From state 2, it can go to state 3 because state 2 is connected to 3. From state 2, however, the agent cannot directly go to state 1 because there is no direct door connecting room 1 and 2 (thus, no arrows). From state 3, it can go either to state 1 or 4 or back to 2 (look at all the arrows about state 3). If the agent is in state 4, then the three possible actions are to go to state 0, 5 or 3. If the agent is in state 1, it can go either to state 5 or 3. From state 0, it can only go back to state 4.

We can put the state diagram and the instant reward values into the following reward table, R table (shown in Figure 16).

$$R = \begin{array}{c} \\ \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

Figure 16: R table

Step 1: Initialize Q table (shown in Figure 17):

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 17: Initialized Q table

Step 2: Choose an action

Look at the second row (state 1) of R table. There are two possible actions for the current state 1: go to state 3, or go to state 5. By random selection, we select to go to 5 as our action.

Now let's imagine what would happen if our agent were in state 5. Look at the sixth row of the reward R table (i.e. state 5). It has 3 possible actions: go to state 1, 4 or 5.

$Q(1,5) = R(1,5) + 0.8 * \max[Q(5,1), Q(5,4), Q(5,5)] = 100 + 0.8 * 0 = 100$

Thus, updated Q table is shown in Figure 18.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 18: Updated Q table

One attempt is finished.

Step 3: Choose another state and motion.

Suppose we choose state 3 and motion of going to state 1. Then, $Q(3,1) = R(3,1) + 0.8 * \max[Q(1,3), Q(1,5)] = 0 + 0.8 * 100 = 80$

Thus, the updated Q table would be shown in Figure 19

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Figure 19: Updated Q table

Step 4: Continue this loop again and again Finally, the Q table will be like Figure 20

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

Figure 20: Updated Q table

Now we can put these values in the original room graph, and we can have Figure 21 For example, from initial State 2, the agent can use the matrix Q as a guide:

From State 2 the maximum Q values suggests the action to go to state 3.
From State 3 the maximum Q values suggest two alternatives: go to state 1 or 4. Suppose we arbitrarily choose to go to 1.
From State 1 the maximum Q values suggests the action to go to state 5.

Thus the sequence is 2 - 3 - 1 - 5.

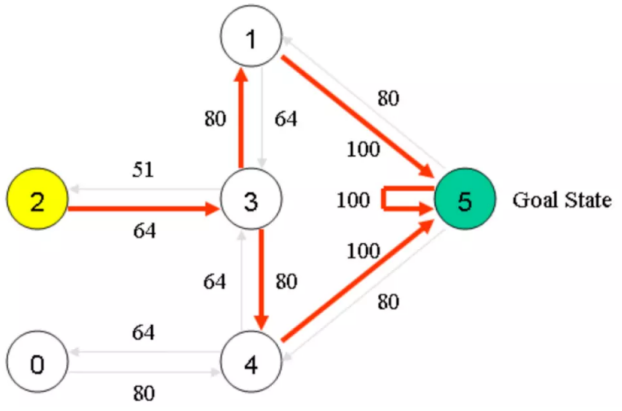

Figure 21: Updated room graph

3.4 Examples 4: playing Deep Q-learning algorithm (Deep Q-networks (DQN)) [3]

We use Q-learning idea for playing Atari 2600 platform, which offers a diverse array of tasks (n=49) designed to be difficult and engaging for human players.

3.4.1 Architecture

They used the same network architecture, hyperparameter values and learningprocedure throughout taking high-dimensional data (210*160 colour video at 60 Hz) as input—to demonstrate that our approach robustly learns successful policies over a variety of games based solely onsensory inputswithonly veryminimalprior knowledge (that is,merely the input data were visual images, and the number of actions available in each game, but not their correspondences).

Model architecture is shown in Figure 22.

The input to the neural network consists of an 84*84*4 image produced by the preprocessing map ϕ. The first hidden layer convolves 32 filters of 8*8 with stride 4 with the input image and applies a rectifier nonlinearity. The second hidden layer convolves 64 filters of 4*4 with stride 2, again followed by a rectifier nonlinearity. Thisisfollowed by a third convolutional layerthat convolves 64 filters of 3*3with stride 1 followed by a rectifier. The final hidden layer is fully-connected and consists of 512 rectifier units. The output layer is a fully-connected linear layer with a single output for each valid action. The number of valid actions varied between 4 and 18 on the games we considered.

3.4.2 Training process

Notably, this method was able to train large neural networks using a reinforcementlearning signal and stochastic gradient descentina stablemanner—illustrated by the temporal evolution of two indices of learning (the agent's average score-per-episode and average predicted Q-values; shown in Figure 23).

3.4.3 Algorithm

Given dataset $\mathcal{D} = (s_j, \mathrm{a}_j, r_j, s_{j+1})$:

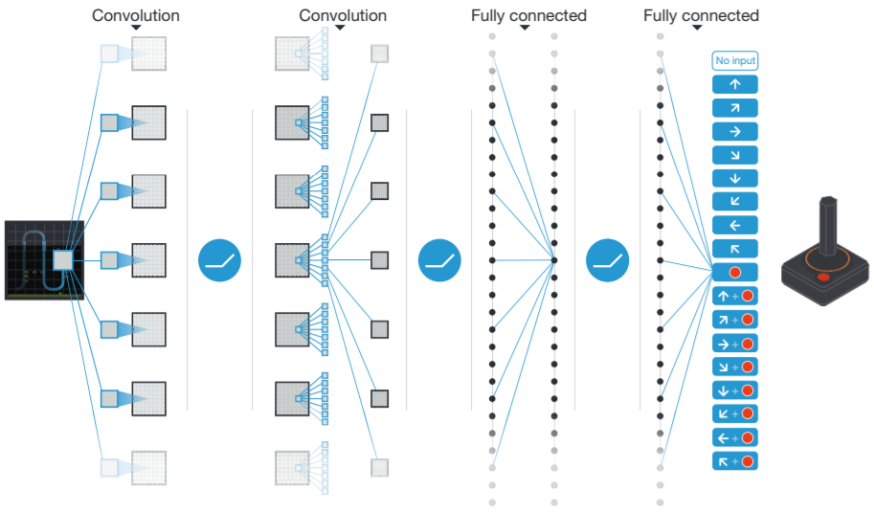

Figure 22: **Schematic illustration of the convolutional neural network.** The details of the architecture are explained in the Methods. The input to the neural network consists of an 84*84*4 image produced by the preprocessing map ϕ, followed by three convolutional layers (note: snaking blue line symbolizes sliding of each filter across input image) and two fully connected layers with a single output for each valid action. Each hidden layer is followed by a rectifier nonlinearity (that is, $\max(0, x)$.

- Sample minibatch $\mathcal{B} \subseteq \mathcal{D}$
- Compute target $y_j = r_j + \gamma \max_a Q_{\theta^-}(S_{j+1}, (a)) \quad \forall j \in \mathcal{B}$
- Use stochastic (semi-)gradient descent to optimize w.r.t. parameters θ

$$\min_{\theta} \sum_{s_j, \mathrm{a}_j, r_j, s_{j+1} \in \mathcal{B}} Q_\theta(s_j, \mathrm{a}_j - y_j)^2$$

- Perform ϵ-greedy action and augment $\mathcal{D}$

3.4.4 Consequence

They compared DQN with the best performing methods from the reinforcement learning literature on the 49 games where results were available. In addition to the learned agents,they also report scores for a professional human gamestester playing under controlled conditions and a policy that selects actions uniformly at random (Figure 24, denoted by 100% (human) and 0% (random) on y axis). Their DQN method outperforms the best existing reinforcement learning methods on 43 of the games without incorporating any of the additional prior knowledge about Atari 2600 games used by other approaches. Furthermore, the DQN agent performed at a level that was comparable to that of a professional human gamestester acrossthe set of 49 games, achieving more than 75%ofthe human score on more than half ofthe games(29 games;see Figure 24)

References

[1] J. McCullock. Q-learning tutorial. `http://mnemstudio.org/path-finding-q-learning-tutorial.htm`.

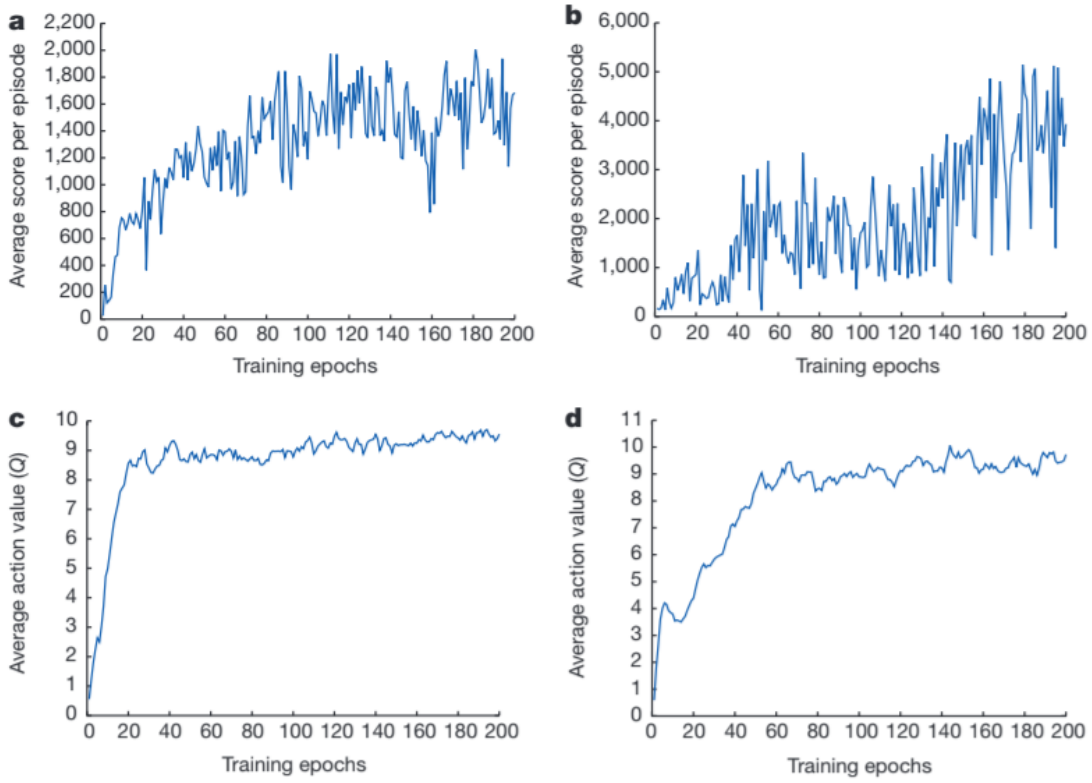

Figure 23: **Training curves tracking the agent's average score and average predicted action-value.** a, Each point is the average score achieved per episode after the agent is run with e-greedy policy (ϵ=0.05) for 520 k frames on Space Invaders. b, Average score achieved per episode for Seaquest. c, Average predicted action-value on a held-outset of states on Space Invaders. Each point on the curve is the average of the action-value Q computed over the held-out set of states. Note that Q-values are scaled due to clipping of rewards. d, Average predicted action-value on Seaquest.

[2] T. Simonini. Diving deeper into reinforcement learning with q-learning. `https://medium.freecodecamp.org/diving-deeper-into-reinforcement-learning-with-q-learning-c18d0db58efe`. Lasted accessed Apr 10, 2018.

[3] D. S. A. A. R. J. V. M. G. B. A. G. M. R. A. K. F. G. O. S. P. C. B. A. S. I. A. H. K. D. K. D. W. S. L. . D. H. Volodymyr Mnih, Koray Kavukcuoglu. Human-level control through deep reinforcement learning). In *Nature*, page 529–533, 2015.

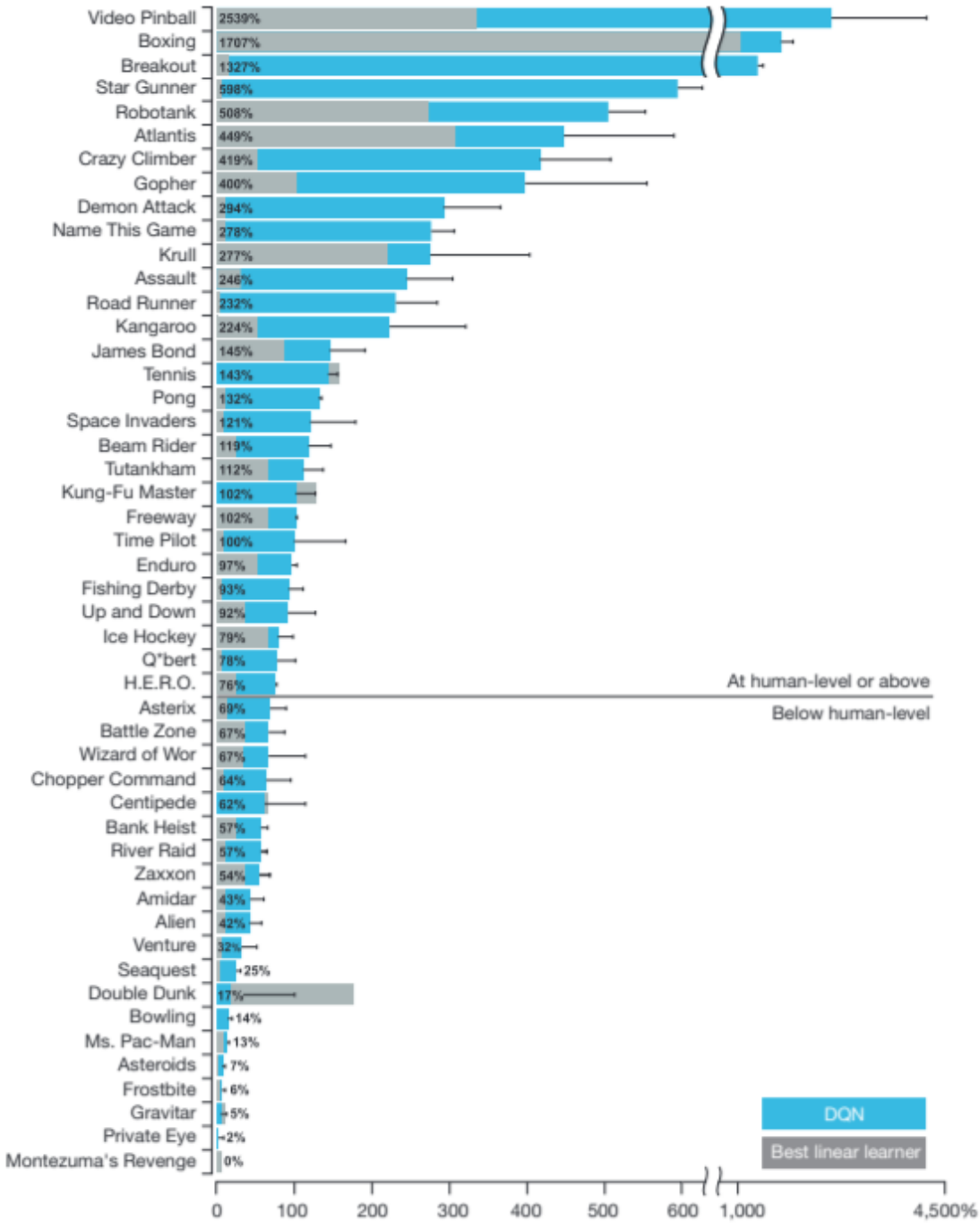

Figure 24: **Comparison of the DQN agent with the best reinforcement learning methods 15 in the literature.** The performance of DQN is normalized with respect to a professional human games tester (that is, 100% level) and random play (that is, 0% level). Note that the normalized performance of DQN, expressed as a percentage, is calculated as: 100 * (DQN score 2 random play score)/(human score 2 random play score). It can be seen that DQN outperforms competing methods(also see Extended Data Table 2) in almost all the games, and performs at a level that is broadly comparable with or superior to a professional human games tester (that is, operationalized as a level of 75% or above) in the majority of games. Audio output was disabled for both human players and agents. Error bars indicate s.d. across the 30 evaluation episodes, starting with different initial conditions.