University of Illinois Fall 2018

ECE 544NA: Pattern Recognition
Lecture 24: November 28

Lecturer: Alexander Schwing Scribe: Tianxi Zhao

1 Introduction

In this lecture, we investigate limitations of using Markov Decision Process (MDP) models to find optimal policies in reinforcement learning (RL) challenges, and explore Q-learning algorithm as an approach to overcome such limitations. Before we start our topics, let's first briefly review lecture 23 which introduces what reinforcement learning is and how to design MDP model to find optimal policy.

2 Review on Reinforcement Learning and MDP

In lecture so far, we have been introduced with three machine learning paradigms:

1. **Discriminative learning.** This type of learning focuses on models that learn the relationship between observed data (X) and its inferred information (Y). For example, predicting object labels from an input image. Commonly, such relationship is modeled as the posterior probability $P(Y|X)$.
2. **Generative learning.** The idea of generative learning comes from the intuition that in order to show true understanding after learning, a learner has to show the ability of constructing meaning based on learned knowledge [3]. This is achieved by learning the properties of observed data (X) itself, and generate new data that has similar meaning with X from learned properties, e.g. generating pictures of dog after learning what a dog is from looking at given dog images.
3. **Reinforcement learning.** Models and algorithms under RL aim to perform decision making in a given environment. Generally, this includes having an agent that are capable of observing its environment and carrying out certain actions, and the goal of the agent is to collect as many (good) rewards as possible. Some popular examples of reinforcement learning include self-driving AI models, AI game players and exploration algorithm on a robot.

A major difference between RL and the other two types of learning is that, models in RL have limited access to supervised ground truth most of the time. It is common that only instant reward can be directly observed. Due to the fact that the final evaluation of current action is also determined by future actions, and that agents in RL can only obtain incomplete observation of its environment at a time, the evaluation of current action/state pair has to be delayed. Therefore, MDP is introduced to formulate the procedure of making decision under such environment. MDP in RL comprises following elements:

1. A set of environments states, S.
2. A set of actions, A.
3. The probability of transition from stat s to s' under action $a \in A$, $P(s'|s, a)$.
4. A instant reward function, $R(s, a, s')$ (sometimes just $R(s)$).
5. A start and maybe a terminal state.

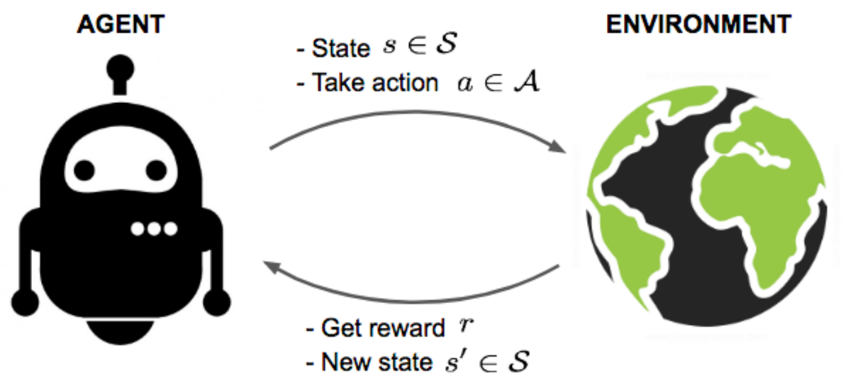
AGENT
ENVIRONMENT
- State $s \in \mathcal{S}$
- Take action $a \in \mathcal{A}$
- Get reward r
- New state $s' \in \mathcal{S}$

Figure 1: An illustration of the relationship between reinforcement learning agent and its environment [5].

The MDP proposed for RL is Markov because it takes the assumption that, given the present state s_t and action a_t, the future and the past are independent. A mathematical expression of this property can be written as:

$$\begin{aligned} &P(S_{t+1} = s'|S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \cdots, S_0 = s_0) \\ =&P(S_{t+1} = s'|S_t = s_t, A_t = a_t) \end{aligned} \tag{1}$$

The eventual goal of formulating the above MDP is to find a policy π^* so as to maximize the expected future reward. In this note, we denote our policy as a mapping from the set of states to the set of actions.

$$\pi(s) : S \to A_S \tag{2}$$

Based on these definitions, we evaluate a policy π by checking the expectation of future rewards using the recursive value function V and Q-function defined as:

$$V_\pi(s) = \sum_{s' \in S} \underbrace{P(s'|s, \pi(s))[R(s, \pi(s), s') + V_\pi(s')]}_{Q(s, \pi(s))} \tag{3}$$

And the optimal policy π^* is calculated by finding the policy that maximize the expected future rewards (V^*, Q^*). Through Bellman optimality principle, we have the following conclusion:

$$V^*(s) = \max_{a \in A_S} \sum_{s' \in S} \underbrace{P(s'|s, \pi(s))[R(s, a, s') + V^*(s')]}_{Q^*(s,a)} \tag{4}$$

$$Q^*(s, a) = \sum_{s' \in S} P(s'|s, a)[R(s, a, s') + \max_{a \in A_S} Q^*(s', a)] \tag{5}$$

$$\pi^*(s) = arg \max_{a \in A_S} Q^*(s, a) \tag{6}$$

With above equations, if the transition probabilities are given, then we may use techniques such as policy/-value iteration or exhaustive search to find the optimal value function V^* and Q^*, and eventually acquire an optimal policy π^*. For introduction to these techniques, please refer to previous lecture notes.

3 Q-learning

3.1 Algorithm

As mentioned in the previous section, when the transition probabilities are know, it's fairly easy to solve the bellman equations of the MDP, and find an optimal policy for decision making. However, such ideal conditions may not be available to our learning models in practice. For example, when an agent tries to make decision under an environment that cannot be fully observed, the agent won't be able to predict what state it might end up after performing certain action. Thus, in order to learn the optimal value of the V and Q functions, we can run a simulator to let an agent randomly explore the states and actions, and then we collect the observed results, and approximate our value functions (V, Q) using those observations. Such learning procedure is called **Q-learning**. It can be proven that if the state/action pairs meets certain conditions, the Q-function values learned by Q-learning will converge to the real optimal Q^* values if enough experiences are collected. Please refer to lecture notes from ECE586 and [1, 2] for detailed proofs.

Now let's take a look at the Q-learning algorithm:

```
Initialize the value of $Q(s,a)\ \forall s \in S, a \in A_S, t = 0$
repeat
    • obtain a sample transition and observed reward $(s_t, a, r, s_{t+1})$
    • obtain sample suggest $Q_t$:
          $Q_t(s_t, a) = r + \max_{a' \in A_{S_{t+1}}} Q(s_{t+1}, a')$
    • We add the sample suggest to $Q$:
          $Q(s_t, a) \leftarrow (1-\alpha)Q(s_t, a) + \alpha Q_t(s_t, a)$
      where $\alpha \in (0, 1)$.
    • $t \leftarrow t + 1$
until $Q$ values convergence;
```

Algorithm 1: Algorithm sketch for Q-learning

Note that to account for missing transition probability, we use a learning rate α when we add the sample suggests to $Q(s, a)$ in the algorithm. This learning rate ensures the final Q values are the running average of explored samples.

3.2 Tuning the learning rate and introduce exploration/exploitation factor

Note that this part of the scribe is not covered by the lecture. But I found this topic to be very interesting when I was doing research for this scribe. In general, the Q-learning algorithm described above can achieve fairly good results when applied in certain applications if we add several small modifications to it.

First, we take a look at how to choose the learning rate α. In order to mimic running average when accumulate sample suggests from state-action pair (s, a) to the $Q(s, a)$, α should decrease if the same (s, a) has been visited many times. Therefore, at a single iteration t in the Q-learning algorithm, we can calculate the learning rate as:

$$\alpha_t = \frac{C}{C + N(s_t, a_t)} \tag{7}$$

Where C is a constant determined before the iteration begins, and $N(s, a)$ represents the time of state-action pair (s, a) the algorithm has seen so far.

Another part of the algorithm that we can try different things with is the way we choose the next state/action pair to learn. Intuitively, there are two ways of picking an action under a given state. The first method is called exploration, which randomly picks an valid action regardless of the environment. Another method is exploitation, which greedily picks an action that maximized the reward based on the Q-values learned so far. In general, exploration ensures all state-action pair gets a fair chance being visited by the algorithm. But when the search space of state-action pair becomes too large, it might take a very long time for the algorithm to converge. On the other hand, exploitation take advantage of previously learned Q values, and would result in a faster convergence, but it may get stuck in a sub-optimal direction and miss the chance of learning the real optimal values due to lack of exploration. Hence, when designing a Q-learning algorithm for a specific problem, algorithm designer needs to let the program first use exploration to learn as many state-action pair as possible, and then use exploitation to calculate the optimal values with a faster convergence speed. Usually this is implemented by choosing an exploration time N_e to make sure that, each state-action pair are visited at least N_e number of time during the exploration phase before the algorithm switch to exploitation.

3.3 Two examples

3.3.1 Example 1: A robot looking for treasure

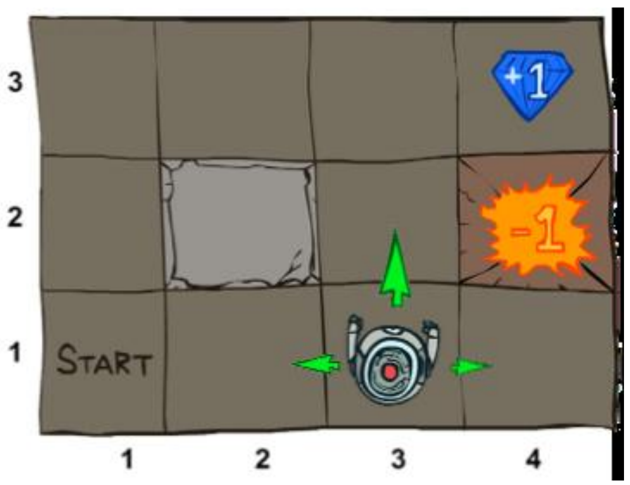

Figure 2: An illustration of example 1. It shows a robotic agent trying to find the treasure on the map.

In this first example, we show a robotic agent explores a simple map, and a positive reward is given if the agent finds the treasure, or a negative reward is given if the agent fall in to the fire. And illustration of this setup is shown in Figure 2. The MDP of this case is formulated as following:

- **State:** A tuple $(robot_y, robot_x)$.
  The state tuple indicates the location of the robotic agent. The board is a 3x4 grid. We see that the $robot_y$ takes value from 1 to 3, and $robot_x$ takes value from 1 to 4.
- **Actions:** The agent can move in four directions: up, down, left, right.
- **Rewards:** If the agent reaches the grid (3,4) which contains the treasure, it will get +1 reward, and if the agent reaches grid (2,4) which is the fire pit, it will get -1 reward. The reward of anywhere else is 0.
- **Initial State:** The robot's initial position is located at (1,1) on the board.
- **Termination:** The agent stops moving on the map once reaching either the grid with treasure or the grid with fire.

Using the Q-learning algorithm, after certain rounds of training, the agent may obtain a policy that eventually guide itself to reach the grid that contains the treasure. A visualization of the learned Q-values are shown in

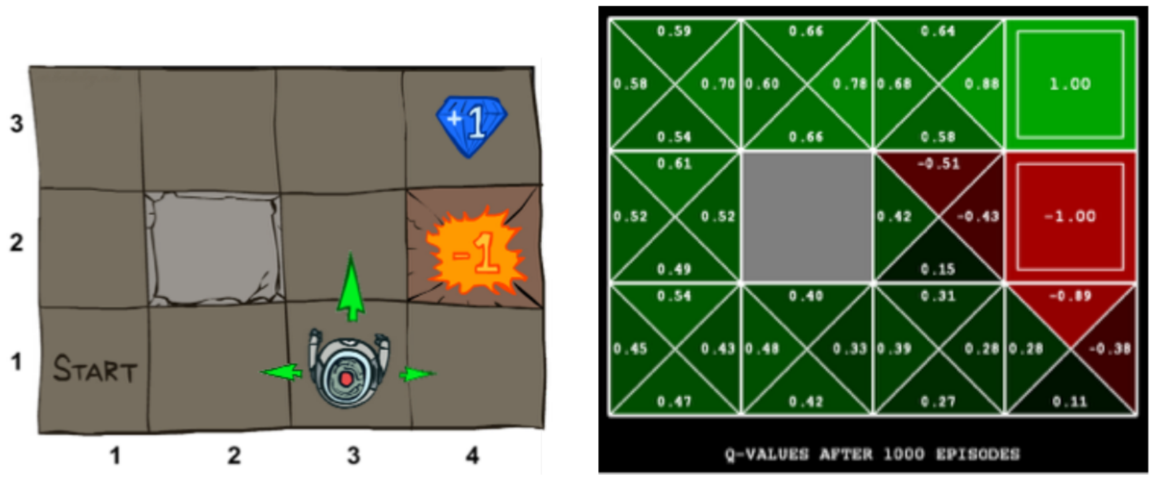

Figure 3: A visualization of the learned Q-values after 1000 episodes of training.

Figure 3, and the policy can be inferred that the robotic agent in a given grid will take the direction with the highest Q-value.
If you look at the Q-values learned at grid (2,3), you may find the results a little bit bizarre. According to the Q-values, the policy of staying on grid (2,3) is to move downward. But since this is an easy example, we can easily tell that optimal policy at this location is to move upward. This result might be caused by the fact that during the learning procedure, the algorithm doesn't bother to explore the whole map, but simply exploits paths that it already known will lead to the treasure, causing the algorithm miss the opportunity to collect enough samples at grid (2,3).

3.3.2 Example 2: Pong game agent with Q-learning

In this subsection, we look at an example of using Q-learning algorithm to teach a program to play the Pong game, a game released by Atari in 1972. An illustration of the pong game is shown in Figure 4 The MDP setup of this game is:

- **State:** A tuple $(ball_x, ball_y, velocity_x, velocity_y, paddle_y)$.
  The state contains the information of pong ball location on the screen, the speed of the ball, and paddles location (who can only move along y axis.) The board is treated as a 12x12 grid, which means both $ball_x$ and $ball_y$ take values from 0 to 11. Velocity of the ball in both direction takes values from -0.6,-0.3,0,0.3,0.6. And finally, the length of the paddle is 2, and $paddle_y$ takes value from 0 to 10. When the ball hits the paddle, the velocity of the ball is chosen randomly, making part of the transition probability unknown to the agent.
- **Actions:** the agent (paddle) only has three options: move up, move down or stay. Each time when the paddle moves, it moves by 1 division in the direction it chooses.
- **Rewards**: +1 when the ball hits the paddle. -1 when the ball has passed the paddle. Or 0 for all other states.
- **Initial State:** we use the state tuple (5,5,1,0,5) as the initial state. Notice that the ball is guaranteed to hit the paddle for the first time if paddle does not move at all in the beginning.

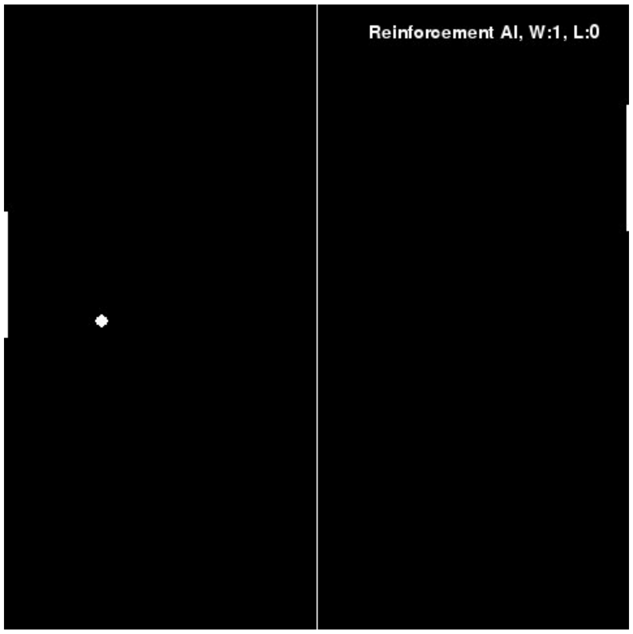

Figure 4: A screen shot of the Pong game. The agent that uses Q-learning is the one on the right.

- **Termination:** A terminal state is reached when the ball hit the board boundary without hitting the paddle.

With the above setup, we use the algorithm listed in section 3.1 to train an agent. We also use a decreasing learning rate α defined as:

$$\alpha_t = \frac{80}{80 + N(s_t, a_t)}$$

As for the exploration factor N_e. We observe that with very small N_e, the performance of learning converges very quickly. With $N_e = 3$, the performance converges at 7.5 bounces on average over 1000 games, and it took around < 10000 rounds of training before achieving this score. We then keep increasing N_e to 15, the performance of our agent finally converges at 10.3 bounces on average, and it takes around 17000 round of games as training before the Q value converges. Therefore, the final exploration factor we choose is

$$N_e = 15$$

Besides tuning these parameters, we tried modifying the MDP states by increasing the division of the grid. During experiment, we tried splitting the board into a 15x15 grid instead. We observe that now it takes around 40,000 rounds of training before the performance converges, and the final average score on average is around 12.5 bounces. A fun observation we had was that during the training with modified MDP states, the highest score achieved was 89, and it happened at 88562^{th} round of training.

3.4 Parameterized Q-learning

With previous examples, we see that our current Q-learning works fine in case we have a tabular environment, i.e. the Q-values of state-action pairs can be stored in tabular format. Nevertheless, when the environments get more and more complicated, the storage space needed for storing Q-values would be impractically large, and the time required for Q-learning will become too long to be considered feasible. Good examples comes from reinforcement learning agents that try to play Atari games. Even though the graphics presented by Atari games are considered simple and pixelated in modern standards, it's still very challenging

Figure 5: A screen shot of the Atari 2600 game Pole Station.

for software agent to understand the graphics as observed states directly. In the example of the Atari game Pole Station (see Fig 5), which uses 128-color palette and the resolution is 160x192. An agent using Q-learning would be required to store Q-values for $(160 \times 192)^{128}$ number of states. Furthermore, real-world applications often entails continuous environment space, making Q-learning totally invalid in such cases.

A strategy that has been proven effective for solving such problem is to parameterize the original Q-learning, so that during the iteration of learning from samples, the algorithm no longer stores a gigantic Q-value table for all the state-action pair, but instead it learns an approximated Q-function. Such idea makes it possible that a Q-learning agent can correctly evaluate the Q-value even when it encounters a state it has never seen before. In the next part, we will explore two methods: (1) approximate Q-learning with linear functions approximation (LFA) and (2) approximate Q-learning with deep neural networks.

3.4.1 Q-learning with linear function approximation

One way to parameterize the Q-function is to approximate it with a linear combination of basis feature functions. The approximation can be formulated as:

- Hand pick $\phi_1, \ldots, \phi_k$ be k basis functions (also called features), where $\phi_i : (S, A_S) \rightarrow \mathbb{R}$
- We define Φ to be:

$$\Phi = \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_m \end{bmatrix} \tag{8}$$

- We define the approximated Q function as:

$$\hat{Q}(s,a) = \sum_{i=1}^{k} \theta_i \phi_i(s,a) = \Theta^T \Phi(s,a) \tag{9}$$

Note that in the above equation, $k << |S \times A_S|$, and instead of learning Q-values, the algorithm learns the parameter Θ.

Now with the definition of linear function approximation, we can have a new Q-learning algorithm that uses minibatch:

```
Select feature vector $\Phi^T = [\phi_1, \ldots, \phi_k]$
Initialize the value of $\hat{\Theta}_0^T = [\theta_1, \ldots, \theta_k]$
$t = 0$
**repeat**
    • Obtain a minibach of samples $\mathcal{B} = \{(s_j, a_j, r_j, s_{j+1})\}$
    • Compute target $\forall j \in \mathcal{B}$
          $y_j = r_j + \gamma \max_{a_{j+1} \in A_{S_{j+1}}} \hat{\Theta}_t^T \Phi(s_{j+1}, a_{j+1})$
      $\gamma$ is the discount factor where $\gamma \in (0, 1]$.
    • Use stochastic gradient descent to optimize w.r.t parameter $\Theta$
          $\hat{\Theta}_{t+1} = arg \max_{\Theta} \sum_{(s_j, a_j, r_j, s_{j+1}) \in \mathcal{B}} (\Theta^T \Phi(s_j, a_j) - y_j)^2$
    • $t \leftarrow t + 1$
**until** $\hat{\Theta}$ *convergence*;
```

Algorithm 2: Algorithm sketch for Q-learning with LFA

With the learned $\hat{\Theta}$ parameter, Q-values can be calculated online during test time, and policy can be established by choosing the action that gives the optimal Q-value.

3.4.2 Deep Q-learning

The concept of Deep Q-learning networks (DQN) are very similar to Q-learning algorithm with LFA. In DQN, instead of using linear functions to approximate the Q function, we use a neural network (which is just a sophisticate non-linear function) to handle state-action pairs. In the example of playing Atari games, we can parameterize our $Q_\theta(s, a)$ as a neural network which takes images as input and produces a number for each of the possible actions. Figure **??** illustrates such procedures.

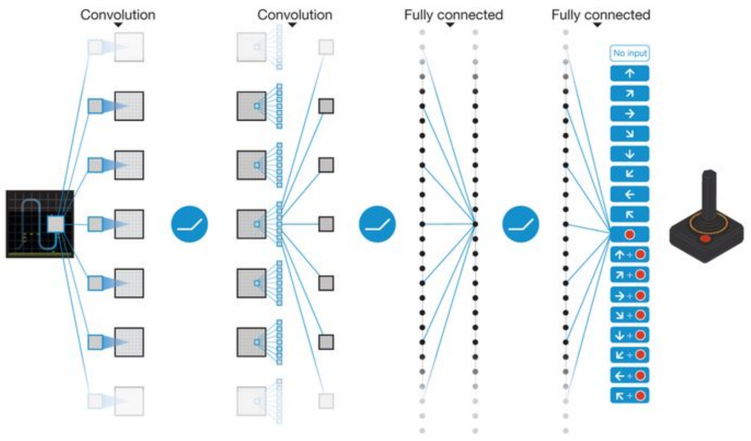

Figure 6: Schematic illustration of using deep neural network for decision making [4].

The algorithm of DQN is again very similar as the one shown in Algorithm 3:

```
Given dataset $\mathcal{D} = \{(s_j, a_j, r_j, s_{j+1})\}$
Initialize the DQN parameter $\hat{\theta}_0$
$t = 0$
**repeat**
    • Obtain a minibach of samples $\mathcal{B} \subseteq \mathcal{D}$
    • Compute target $\forall j \in \mathcal{B}$
          $$y_j = r_j + \gamma \max_{a_{j+1} \in A_{S_{j+1}}} Q_{\hat{\theta}_t}(s_{j+1}, a_{j+1})$$
      $\gamma$ is the discount factor where $\gamma \in (0, 1]$.
    • Use stochastic gradient descent to optimize w.r.t parameter $\theta$
          $$\hat{\theta}_{t+1} = arg \max_{\theta} \sum_{(s_j, a_j, r_j, s_{j+1}) \in \mathcal{B}} (Q_{\theta_t}(s_j, a_j) - y_j)^2$$
    • Perform $\epsilon$-greedy action and augment $\mathcal{D}$
    • $t \leftarrow t + 1$
**until** $\hat{\Theta}$ *convergence*;
```

Algorithm 3: Algorithm sketch for DQN

By using DQN, research has shown that DQN agent with sufficient training can outperform average human on many of the Atari games (see Fig 7).

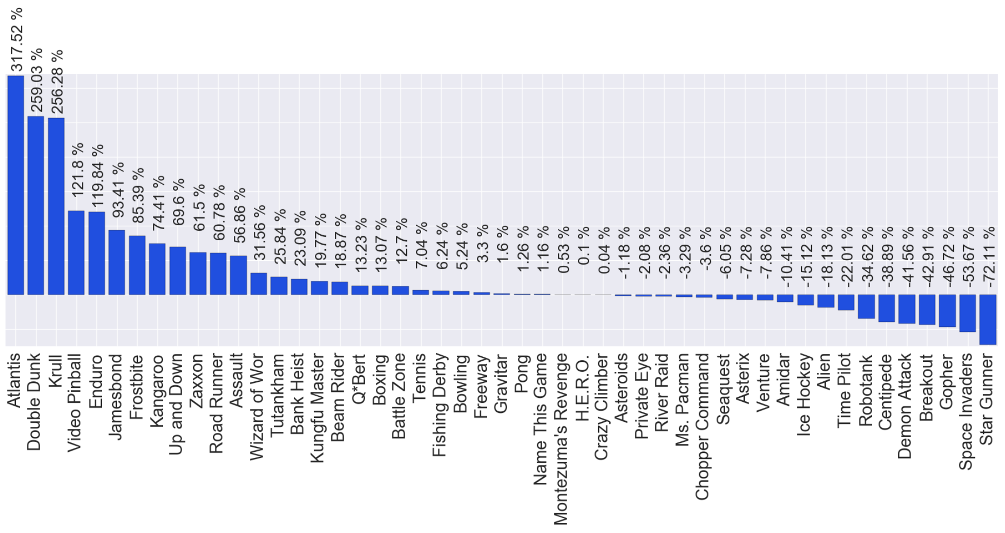

Figure 7: DQN performance vs. Human performance on Atari games (diagram quote from lecture slides).

3.5 Summary

In this lecture, we focus on reinforcement learning under conditions such that state transition probabilities are unknown, and environments are hard to translate into small number of states. In order to overcome these challenges, Q-learning and parameterized Q-learning are introduced. We see that the Q-learning algorithm can be applied when we don't know the transition probabilities. Furthermore, using either LFA or DQN, we can approximate the Q-functions so the algorithm no longer needs to store Q-values for all possible state-action pairs.

References

[1] D. P. Bertsekas. *Dynamic Programming and Optimal Control Vol. II.* Athena Scientific, 2012.

[2] D. P. Bertsekas. *Dynamic Programming and Optimal Control Vol. I.* Athena Scientific, 2017.

[3] U. Hanke. *Generative Learning*, pages 1356–1358. Springer US, Boston, MA, 2012.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

[5] L. Wang. *A (Long) Peek into Reinforcement Learning*. Lil'Log, 2018.