DSP Project TIC TAC TOE

احمد رافت عبدالوهاب فصل:1 رقم: 8 بيشوي وائل فؤاد فصل:1 رقم: 50 حسين نجيب حسين فصل:2 رقم:6
خالد محمد الانصاري فصل:2 رقم:7 شريف عصام ميهوب فصل:2 رقم: 22

Submitted to Prof. Mohsen Rashwan

***Abstract* — another side of deep learning through reinforcement learning, in a game show case TIC TAC TOE game**

I. Introduction

In 1952 — Arthur Samuel wrote the first computer learning program. The program was the game of checkers, and the IBM computer improved at the game the more it played, studying which moves made up winning strategies and incorporating those moves into its program.

1997 — IBM's Deep Blue beats the world champion at chess.

2006 — Geoffrey Hinton coins the term "deep learning" to explain new algorithms that let computers "see" and distinguish objects and text in images and videos. 2016__ AlphaGo versus Lee Sedol, or Google DeepMind Challenge Match, was a five-game Go match between 18-time world champion Lee Sedol and AlphaGo and AlphaGo won 4 games to 1 then they made AlphaGo Zero who beat AlphaGo 100 games to 0.

Arthur Samuel (1959). Machine Learning is a sub-set of artificial intelligence where computer algorithms are used to autonomously learn from data and information. In machine learning computers don't have to be explicitly programmed but can change and improve their algorithms by themselves.

Tom Mitchell (1998) Well-posed Learning Problem: A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.

II. Machine learning algorithms.

- Supervised learning.
- Unsupervised learning.
- Reinforcement learning.

A. *Reinforcement learning:*

This part is an overview of the field of reinforcement learning and concepts that are relevant to the proposed work. The field of reinforcement learning is not very well-known and although the learning paradigm is easily understandable, some of the more detailed concepts can be difficult to grasp. Accordingly, reinforcement learning is presented at beginning with a review of the fundamental concepts and methods.

This introduction to reinforcement learning is followed by a review of the three major components of the reinforcement learning method: the environment, the learning algorithm, and the representation of the learned knowledge.

III. Reinforcement Learning

Reinforcement learning is learning by interacting with an environment. An agent learns from its actions, and it selects its actions based on its past experiences (exploitation) and also by new choices (exploration), which is simply a trial and error learning. The agent receives a numerical reward, and the agent seeks to learn selecting actions that maximize the accumulated reward over time.

A. Reinforcement learning block diagram:

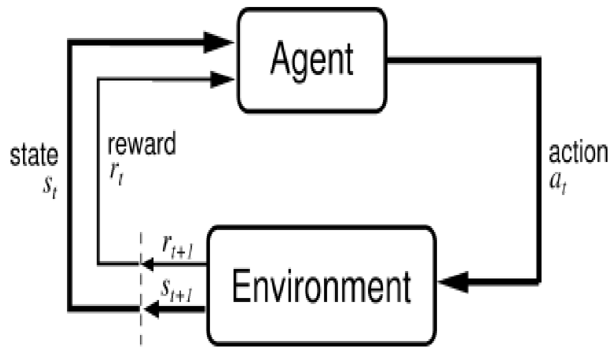


Figure 1 Reinforcement learning block

Then, the next step is how the agent takes action or which action should be taken, and this will lead us to Markov Decision Processes.

Consider an example of a child learning to walk.

Here are the steps of a child taking steps to learn walking:

1. The first thing the child will observe, is to notice how you are walking. You use two legs, taking a step at a time in order to walk. Grasping this concept, the child tries to replicate you.
2. But soon he/she will understand that before walking, the child has to stand up! This is a challenge that comes along while trying to walk. So now the child attempts to get up, staggering and slipping but still determinant to get up.
3. Then there's another challenge to cope up with. Standing up was easy, but to remain still is another task altogether! Clutching thin air to find support, the child manages to stay standing.
4. Now the real task for the child is to start walking. But it's easy to say than actually do it. There are so many things to keep in mind, like balancing the body weight, deciding which foot to put next and where to put it.

Let's formalize the above example, the "problem statement" of the example is **to walk**, where **the child is the agent** trying to manipulate the **environment (which is the surface on which it walks)** by **taking actions (walking)** and he/she tries to go from one **state (each step he/she takes)** to another.

The child gets a **reward (let's say chocolate)** when he/she accomplishes a **sub-module of the task (taking couple of steps)** and will not receive any chocolate **(negative reward)** when he/she is not able to walk. This is a simplified description of a reinforcement learning problem.

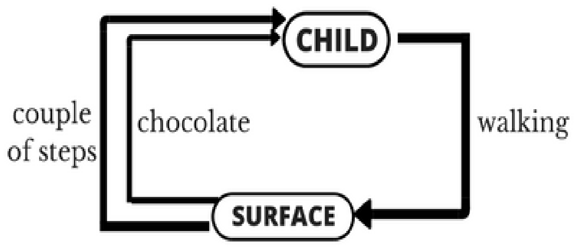


Figure 2 child example

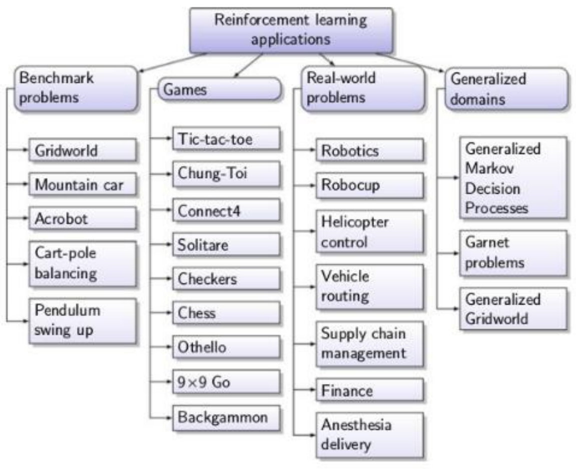


Figure 3

IV. Applications

Reinforcement is a promising deep learning technique and it can be used in many applications as shown in fig. 3

V. Markov decision processes (MDP)

A. Definition

Markov decision processes (MDPs) provide a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.

B. MDP

Formally, RL can be described as a Markov decision process (MDP), which consists of:
• A set of states S, plus a distribution of starting states p(s0).
• A set of actions A
• Transition dynamics T (st+1|st, at) that map a state- action pair at time t onto a distribution of states at time t + 1.
• An immediate/instantaneous reward function R(st, at, st+1).
• A discount factor $\gamma \in [0, 1]$, where lower values place more emphasis on immediate rewards.

Policy is a solution for MDP which specifies an action for every state of the agent. And optimal Policy achieves the highest accumulated expected reward.
In general, the policy π is a mapping from states to a probability distribution over actions: $\pi : S \rightarrow p(A = a|S)$.

let's consider in this example that we have a robot and its target is reach to the first star , here we have 11 available blocks which represent 11 states, we have 4 actions at each state which are moving actions (right,left,forward,backward) and this is the difference between random policy and optimal policy .

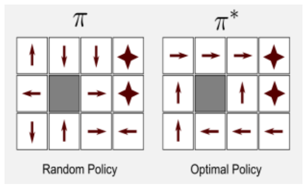


Figure 4

VI. REINFORCEMENT LEARNING SOLUTION

The task of agent is to learn policy π: S→A for selecting its next action a_t based on the current observed state s_t that is, π(s) =a_t How shall we specify precisely which policy π we would like the agent to learn?

An obvious approach is to require the policy that produces the greatest possible cumulative reward for the robot over time to achieve this requirement more precisely, we define the cumulative value V^π(s) achieved by following an arbitrary policy π form an arbitrary initial state S_o as follows

$$V^{\pi}(s)= E\,[\textstyle\sum_{s'} \gamma^{t} * R_t | S0 = S, \pi]$$

$$V^{\pi}(s)= R(s, \pi(s)) + \gamma \textstyle\sum_{t=0}^{\infty} T(s'|s,\pi(s)) V^{\pi}(s')$$

Then, value function at any state S is the expected accumulated following the policy from state S. but we want to deal with the optimal value so,

$$V^{*}(s)= \max R(s, a)+ \gamma \textstyle\sum_{s'} T(s'|s,a\,) V^{*}(s')$$

To ensure we have chosen the optimal action at every state we need to define a function that calculates the value of taking an action in a certain state, this function is called action value function (Q-function), the Q-value function at any state and action is the expected accumulator reward from taking an action in state S and then following the policy.

This can be defined as follow

$$Q^{\pi}(s\ , a)= E\,[\textstyle\sum_{s'} \gamma^{t} * R_t | S0 = S, ao = a, \pi]$$

To achieve the optimal policy Q function will be:

$$Q^{*}(s)= R(s, a)+ \gamma \textstyle\sum_{s'} T(s'|s,a) V^{*}(s')$$

Then we can define the optimal policy as follows

$$\pi^{*}(s)= \mathrm{argmax}(Q^{*}(s,a))$$

VII. DYNAMIC PROGRAMMING

Our target is to compute the optimal policy then we use dynamic programming algorithms to achieve that, the most well-known dynamic programming algorithms are:

1-value iteration

2-policy iteration

A. VALUE ITERATION

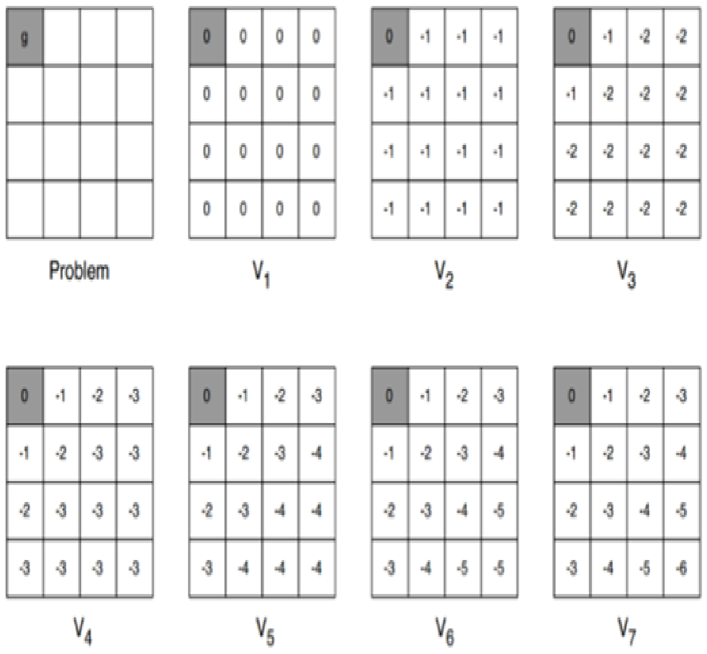
The first algorithm we will look at is value iteration. The basic idea of value iteration is if we knew the true value of each state, our decision would be always choose the action that maximizes expected utility. But we don't initially know the state's true value; we only know its immediate reward. But, for example, a state might have low initial reward but be on the path to a high-reward state.

The true value of a state is the immediate reward for that state, plus the expected discounted reward if the agent acted optimally from that point on

$$V_{i+1}(s) \leftarrow \max R(s, a)+ \gamma \textstyle\sum_{s'} T(s'|s,a\,) V_i(s')$$

Following this steps:

1. Assign each state a random value
2. For each state, calculate its new V.
3. Update each state's V based on the calculation above
4. If no change in V after more iteration, halt. This algorithm is guaranteed to converge to the optimal solutions.
5. Repeat step 1 again.



Problem:

g			

V_1:

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_2:

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_3:

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_4:

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_5:

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_6:

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_7:

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

Figure 5

B. *Policy Iteration.*

Another way to get Optimal Policy is to use Policy Iteration, Policy Iteration algorithm manipulates the policy directly. In Policy Iteration algorithms, you start with a random policy, then find the value function for each state of that policy (policy evaluation step), then find a new (improved) policy based on the previous value function, and so on till get the optimal policy.
In this process, each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Given a policy, its value function can be obtained using the Bellman equation.

$$\pi'(s) \ \leftarrow \text{argmax}\ [R(s, a) + \gamma \sum_{s'} T(s'|s,a)\, V_\pi(s')]$$

Example:

So in policy iteration algorithm we make two steps:
Step 1: Policy evaluation: calculate utilities for some fixed policy (not optimal utilities!) until convergence.
Step 2: Policy improvement: update policy using one-step look-ahead with resulting converged (but not optimal!) utilities as future values and
Repeat steps until policy converges.

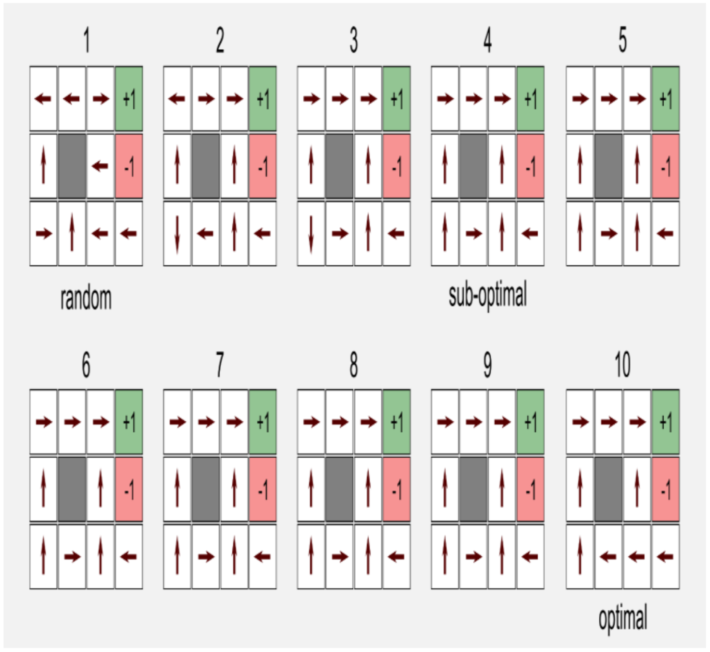


Figure 6

VIII. Q-Learning

Q-learning is a model-free reinforcement learning technique. Specifically, Q-learning can be used in finding an optimal action-selection policy for any given (finite) Markov decision process (MDP).

It comes from Q function that map states to action by computing the values of action in certain states instead of computing values of this states.

Here is the equation which used to derive Q-function independent on the state values:
$Q^*(s) = R(s, a) + \gamma \sum_{s'} T(s'|s,a)\ max_a\ Q(s', a)$

This leads to have the updated rule as follows:

$Q(s,a) \ \leftarrow Q(s,a) + \alpha[R(s, a) + \gamma\ max_a\ Q(s', a) \ - Q(s, a)]$
α is the learning rate .

The learning rate determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information. In fully deterministic environments, a learning rate of α=1 α t = 1 {\displaystyle \alpha _{t}=1} is optimal.
In the above equation the algorithm depend on choosing an arbitrary value for Q at the beginning then take actions according to Q-table and then Update the Q-table by using the assumed value and do this process again (calculate the new Q Value and use it to update the Q-Table) till reach to optimal solution by iterative method and get the optimal policy.

IX. Tic Tac Toe

Figure 7 Tic Tac Toe game

The implementation of Q-learning on the TIC TAC TOE game follows the pseudo-code given by Meeden [CS63 Lab 6]. [1]

A general introduction to Q-learning can be obtained from Chapter 13 of Mitchell (1997), Sutton & Barto (2012), or Watkins & Dayan (1992) [2]

In the game of Tic Tac Toe, at each discrete time step t, the state $s \in S$ of the system is defined by the marks on the board and which player's turn it is, and the available actions $a \in A$ by the empty squares on the board. We are looking for a policy $\pi : S \rightarrow A$ which tells us which action to take in which state to maximize our chance of winning.

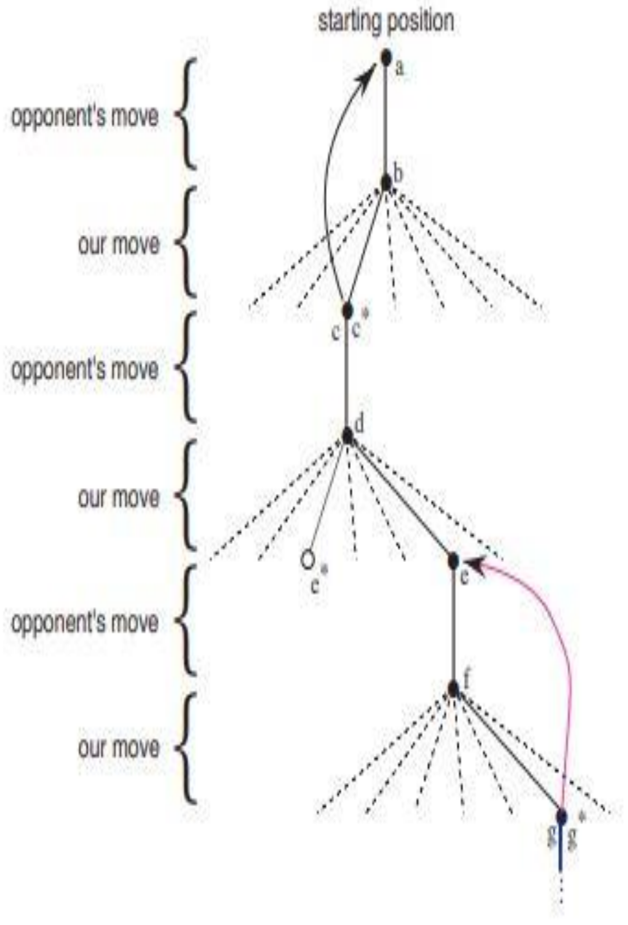


Figure 8 sequence of actions and moves

Given a policy π, at any given time each state has a certain value V^{π}, which is the expected discounted reward from following that policy for all future time:

where the reward is at time step t and γ represents the discount factor.

In the implementation of Tic Tac Toe, we adopt the 'sign convention' that reward is positive for player "X" -- specifically, a reinforcement of 1.0 is awarded when player "X" wins, -1.0 when player "O" wins, and 0.5 in the case of a tie. Hence, player "X" seeks to maximize value, whereas player "O" seeks to minimize it. The discount factor γ is given a value of 0.9. (Its value is not so important in Tic Tac Toe as the game is deterministic with a finite time horizon).

We seek to find the optimum policy π^* such that the value achieves a unique optimum value $V^{\pi^*} = V^*$. To this end, we define the evaluation function $Q(s, a)$ as the maximum discounted cumulative reward that can be obtained by starting from state s and applying a as first action:

,

where s' is the state that results from applying action a in state s.

However, to optimize the total future discounted reward, the action a must be the one which maximizes Q:

$$V^*(s) = \max_{a'} Q(s, a')$$

which, upon inserting into the above equation, leads to Bellman's equation:

$$Q(s, a) = r(s, a) + \gamma \max_{a'} Q(s, a')$$

This recursive definition of Q provides the basis for Q-learning. Suppose we start with some initial estimate $\hat{Q}(s, a)$ of $Q(s, a)$and choose an action a, thereby obtaining an immediate reward r and arriving at a new state s'. If our estimate $\hat{Q}$ is correct, we would expect the difference

$$r + \gamma \max_{a'} \hat{Q}(s', a') - \hat{Q}(s, a)$$

to be zero. If it is not zero, but slightly positive (negative), then the action and resulting reward can be viewed as 'evidence' that our estimate was too low (high).

The way this 'discrepancy' is handled is by simply adding this difference, weighted by a learning factor α, to our previous estimate $\hat{Q}_n$ to obtain a revised estimate $\hat{Q}_{n+1}$:

In this implementation of Q-learning for Tic Tac Toe, Q has the form of a dictionary, the keys of which are the states of the game (represented by the game's Board and the mark of player whose turn it is) and the values are again dictionaries containing the current estimate of the Q for each available move (i.e., empty square). Bellman's equation is implemented in the learn_Q method of the Game class, which is called on every move.
The game's Q is shared with any instances of QPlayer playing the game, which uses it to make its move decisions. Following the implementation by Heisler [q.py, slides],
the QPlayer follows an "ϵ-greedy" policy, meaning that with probability ϵ it chooses a random move, and otherwise it follows the policy dictated by Q -- that is, if the player has mark "X" ("O"), choose the move with the highest (lowest) Q-value, in accordance with our 'sign convention'. During training, epsilon is set to a high value to encourage exploration, whereas for the actual match against a human, it is set to zero for optimal performance. [3]

X. Conclusion

Reinforcement learning techniques will play a great tool in shaping the future, due to its high capabilities to learn the optimum solution to some problems with overseeing the consequences of its actions.

References

[1] https://www.cs.swarthmore.edu/~meeden/cs63/f11/lab6.php

[2] https://www.cs.swarthmore.edu/~meeden/cs63/f11/ml-ch13.pdf

[3] https://github.com/khpeek/Q-learning-Tic-Tac-Toe