

Human level control through deep reinforcement learning

Zhuoyan Xu

University of Wisconsin-Madison

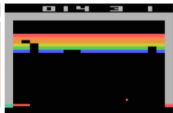
April 23, 2019

- 1 Introduction to reinforcement learning
- 2 Q-learning
 - Action value function
 - Value iteration
 - Simple example
- 3 Deep Q network
 - Action value function approximation
 - Architecture
 - Optimization
 - Learning Stability
 - Result

- Playing Atari with Deep Reinforcement Learning. ArXiv (2013)
- Human-level control through deep reinforcement learning. Nature (2015)



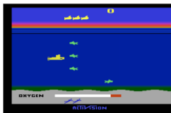
Pong



Breakout



Space Invaders



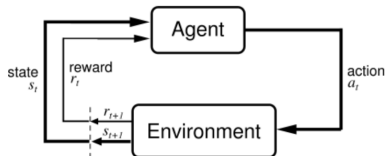
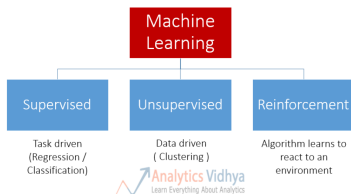
Seaquest



Beam Rider

Introduction

Types of Machine Learning



Markov decision process

The mathematical framework for defining a solution in reinforcement learning scenario is called Markov Decision Process. This (simple version) can be defined as:

- Set of states, S .
- Set of actions, A .
- Reward function, R .
- Policy, π .
- Value, V .

We have to take an action (A) to transition from our start state to our end state (S). In return getting rewards (R) for each action we take. Our actions can lead to a positive reward or negative reward.

Markov decision process

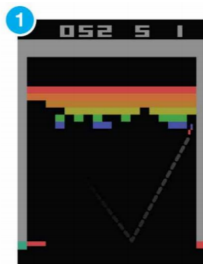
The set of actions we took define our policy (π) and the rewards we get in return defines our value (V). Our task here is to maximize our rewards by choosing the correct policy. So we have to maximize:

$$E(r_t | \pi, s_t, a)$$

For all states and time t .

State representation

Consider **Breakout** game:



State:

{
 Location of the paddle
 Location/direction of the ball
 Presence/absence of each individual brick
} \Rightarrow screen pixels

- 1 Introduction to reinforcement learning
- 2 Q-learning
 - Action value function
 - Value iteration
 - Simple example
- 3 Deep Q network
 - Action value function approximation
 - Architecture
 - Optimization
 - Learning Stability
 - Result

Future reward

Consider a Markov decision process

$s_0, a_0, r_1, s_1, a_1, r_2, \dots, s_{n-1}, a_{n-1}, r_n, s_n$.

The future reward at time/iteration(t) can be defined as:

$$R_t = r_t + r_{t+1} + r_{t+2} + \dots$$

Due to the stochastic environment, we change it to discount future reward:

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots + \gamma^{n-t} r_n \\ &= r_t + \gamma (r_{t+1} + \gamma (r_{t+2} + \dots)) \\ &= r_t + \gamma R_{t+1} \end{aligned}$$

Action value function

Given a state s , an action a and a policy π , the action-value of the pair (s,a) under π is defined by:

$$Q^{\pi}(s, a) = E[R_t | s_t, a_t, \pi]$$

which represents the "Quality" of a certain action in a given state. We want to get policy to maximize the this action value function. The optimal action-value function is:

$$Q^*(s, a) = \max_{\pi} E[R_t | s_t, a_t, \pi]$$

Bellman equation

The optimal action-value function obeys an important identity known as the Bellman equation. This is based on the following intuition: if the optimal value $Q(s', a')$ of the sequence s' at the next time-step was known for all possible actions a' , then the optimal strategy is to select the action a' maximizing the expected value of $r + \gamma Q(s', a')$.

Bellman equation:

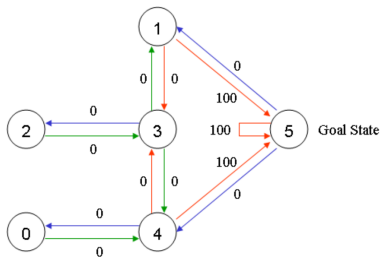
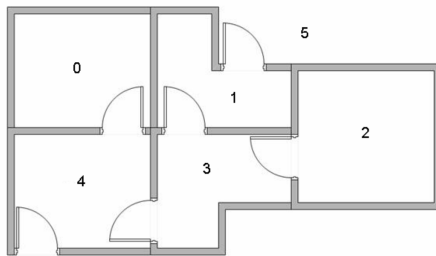
$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

The core algorithm of Q learning is a value iteration update until convergence:

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

where α is the learning rate, set between 0 and 1. The learning rate determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities). In fully deterministic environments, a learning rate to be 1 is optimal. When the problem is stochastic, the algorithm converges under some technical conditions on the learning rate that require it to decrease to zero.

Simple example



Simple example

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

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

updating

We set $\alpha = 1, \gamma = 0.8$, then:

$$Q[s, a] = r[s, a] + 0.8 \max_{a'} Q[s', a']$$

Until convergence:

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

Final result

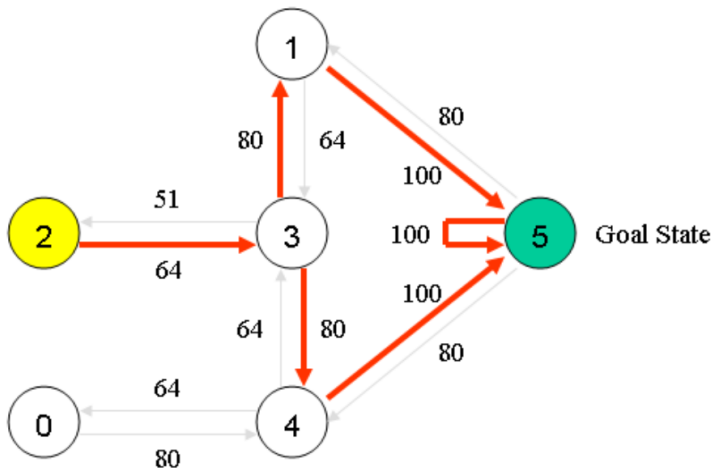


Figure: apply Q table

- 1 Introduction to reinforcement learning
- 2 Q-learning
 - Action value function
 - Value iteration
 - Simple example
- 3 Deep Q network
 - Action value function approximation
 - Architecture
 - Optimization
 - Learning Stability
 - Result

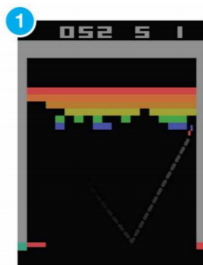
drawback of value iteration

In practice, value iteration is impractical.

- Action-value function is estimated separately for each sequence, without any generalization.
- Limited of states and actions.

drawback of value iteration

Consider **Breakout** game:



State: screen pixels

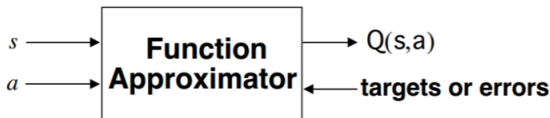
- Image size: 84×84 (resized)
- Consecutive 4 images, each with with 256 gray levels.

It requires $256^{84 \times 84 \times 4}$ rows in Q-table.

Function approximation

It is common to use a function approximator to estimate the action-value function, i.e. to obtain features and then represent states with a simple function of those features.

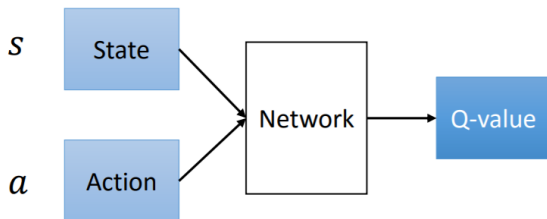
$$Q(s, a; \theta) \approx Q(s, a)$$



Linear and non-linear (such as neural network).

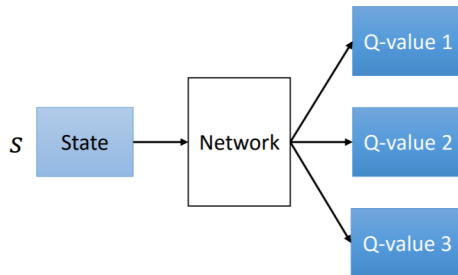
Function approximation

There are several possible ways of parameterizing Q using a neural network. Because Q maps history-action pairs to scalar estimates of their Q -value, the history and the action have been used as inputs to the neural network by some previous approaches.



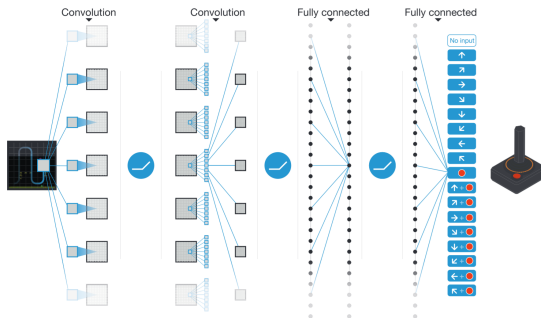
The main drawback of this type of architecture is that a separate forward pass is required to compute the Q -value of each action, resulting in a cost that scales linearly with the number of actions.

Architecture



The new architecture in which there is a separate output unit for each possible action, and only the state representation is an input to the neural network. The outputs correspond to the predicted Q-values of the individual actions for the input state. The main advantage of this type of architecture is the ability to compute Q-values for all possible actions in a given state with only a single forward pass through the network.

Architecture



Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Recall the Bellman equation:

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

We define the loss function(use MSE) at each iteration:

$$L(\theta_i) = \mathbb{E}_{s,a,r',s'} \left[\left(r + \gamma \max_{a'} Q(s', a', \theta_{i-1}) - Q(s, a, \theta_i) \right)^2 \right]$$

where the $r + \gamma \max_{a'} Q(s', a', \theta_{i-1})$ is said to be the target.
optimization:SGD etc.

Exploration-Exploitation Dilemma

At iteration t , how to choose action? ϵ greedy policy:

- Exploration: With probability ϵ select a random action.
- Exploitation: With probability $1 - \epsilon$ to select $a = \operatorname{argmax}_{a'} Q(s, a')$.

Reinforcement learning is known to be unstable or even to diverge when a nonlinear function approximator such as a neural network is used to represent the action-value (also known as Q) function. There are several causes:

- The correlations present in the sequence of observations.
- The fact that small updates to Q may significantly change the policy and therefore change the data distribution.
- And the correlations between the action-values(Q) and the target values $r + \gamma \max_{a'} Q(s', a')$.

Solution for instability

- A biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution.
- An iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target.

Experience replay

Build data-set from agents own experience:

- Store the agents experiences at each time-step, $e_t = (s_t, a_t, r_t, s_{t+1})$ in the dataset $D_t = \{e_1, \dots, e_t\}$.
- During the inner loop of the algorithm, Sample random mini-batch of transitions $\langle s, a, r, s' \rangle$ from D.
- Redefine loss function between Q-network and Q-learning targets:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right)^2 \right]$$

then optimize.

Advantage:

- Each step of experience is potentially used in many weight updates, which allows for greater data efficiency.
- Learning directly from consecutive samples is inefficient, owing to the strong correlations between the samples; randomizing the samples breaks these correlations and therefore reduces the variance of the updates.

Periodically update

Use a separate network for generating the targets y_j in the Q-learning update. i.e. Every C updates we clone the network Q to obtain a target network \hat{Q} and use \hat{Q} for generating the Q-learning targets y_j for the following C updates to Q .

Advantage: standard online Q-learning, where an update that increases $Q(s_t, a_t)$ often also increases $Q(s_{t+1}, a)$ for all a and hence also increases the target y_j , possibly leading to oscillations or divergence of the policy.

Generating the targets using an older set of parameters adds a delay between the time an update to Q is made and the time the update affects the targets y_j , making divergence or oscillations much more unlikely.

Algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N


Initialize action-value function Q with random weights θ


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

For episode = 1, M **do**

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

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

ϵ -greedy policy  With probability ϵ select a 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 reward r_t and image x_{t+1}
 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

Experience memory  Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
 Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
 Perform a gradient descent step on $(y_j - \hat{Q}(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

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

End For

End For

Result

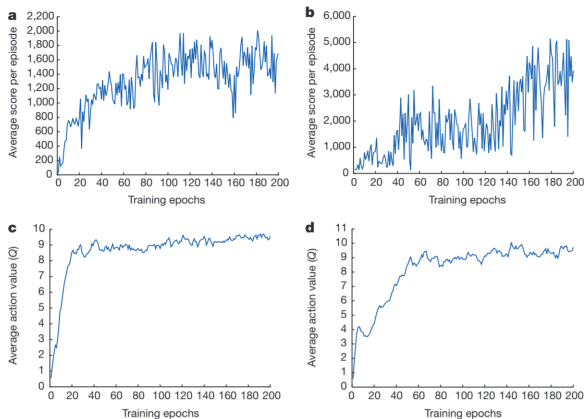
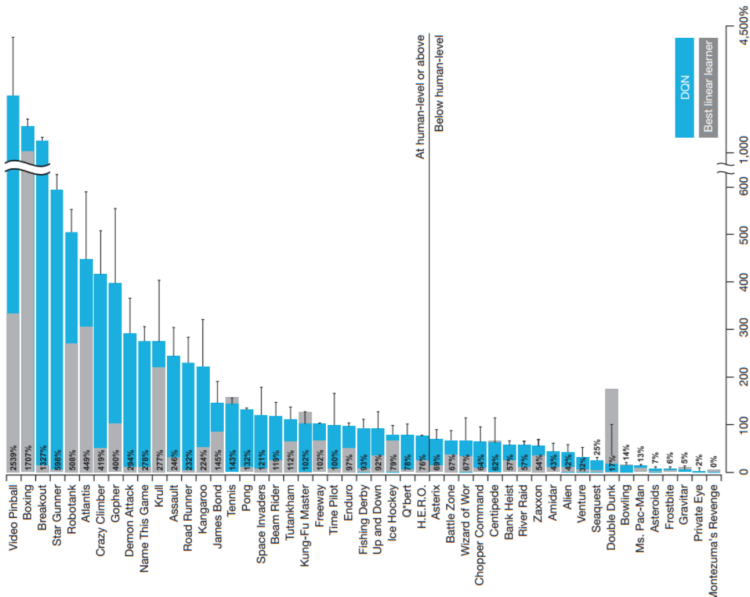


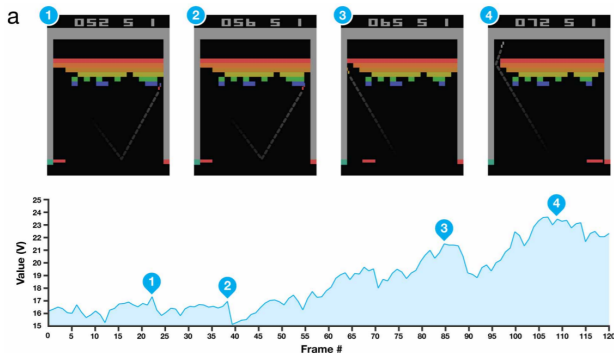
Figure 2 | Training curves tracking the agent's average score and average predicted action-value. **a**, Each point is the average score achieved per episode after the agent is run with ϵ -greedy policy ($\epsilon = 0.05$) for 520 k frames on Space Invaders. **b**, Average score achieved per episode for Seaquest. **c**, Average predicted action-value on a held-out set 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 (see Methods). **d**, Average predicted action-value on Seaquest. See Supplementary Discussion for details.

Result



Result



A visualization of the learned value function on the game Breakout. At time points 1 and 2, the state value is predicted to be around 17 and the agent is clearing the bricks at the lowest level. At time point 3, the agent is about to break through to the top level of bricks and the value increases to around 21. At point 4, the value is above 23.

Conclusion

- Came up with a new architecture: deep Q network to do function approximation in Q learning.
- Demonstrate that a single architecture can successfully learn control policies in a range of different environments with only very minimal prior knowledge (pixels and game score).
- In certain games DQN is able to discover a relatively long-term strategy (for example, Breakout: the agent learns the optimal strategy, which is to first dig a tunnel around the side of the wall allowing the ball to be sent around the back to destroy a large number of blocks).