DESIGN DOCUMENT

Decision making using Reinforced Deep Learning

November 13, 2016

Prepared for:
Btech Project Interim Evaluation

Prepared by:
Kevin Joseph, Jayadeep K M, Mohammed Nisham K

Guide:
Vipin Vasu
vipin@cet.ac.in

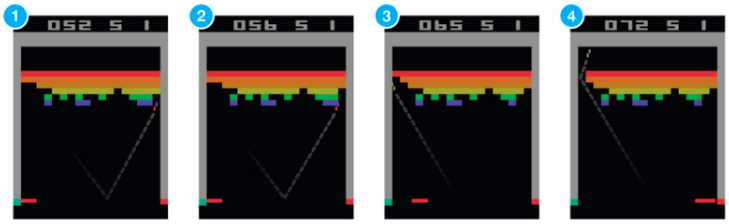
College Of Engineering, Trivandrum

TABLE OF CONTENTS

1 Background and Literature

1.1 Reinforcement Learning

Consider the game Breakout. In this game you control a paddle at the bottom of the screen and have to bounce the ball back to clear all the bricks in the upper half of the screen. Each time you hit a brick, it disappears and your score increases you get a reward.



Suppose you want to teach a neural network to play this game. Input to your network would be screen images, and output would be one of four actions left, right, do nothing or fire to launch the ball. It would make sense to treat it as a classification problem for each game screen you have to decide, Which action to take. Sounds straightforward, Sure, but then you need training examples, and a lots of them. Of course you could go and record game sessions using expert players, but thats not really how we learn. We dont need somebody to tell us a million times which move to choose at each screen. We just need occasional feedback that we did the right thing and can then figure out everything else ourselves.

This is the task reinforcement learning tries to solve. Reinforcement learning lies somewhere in between supervised and unsupervised learning. Whereas in supervised learning one has a target label for each training example and in unsupervised learning one has no labels at all, in reinforcement learning one has sparse and time-delayed labels, the rewards. Based only on those rewards the agent has to learn to behave in the environment.

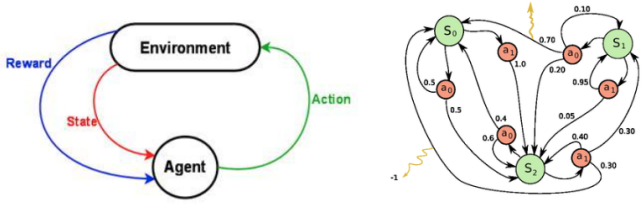
While the idea is quite intuitive, in practice there are numerous challenges. For example when you hit a brick and score a reward in the Breakout game, it often has nothing to do with the actions (paddle movements) you did just before getting the reward. All the hard work was already done, when you positioned the paddle correctly and bounced the ball back. This is called the credit assignment problem i.e., which of the preceding actions was responsible for getting the reward and to what extent.

Once you have figured out a strategy to collect a certain number of rewards, should you stick with it or experiment with something that could result in even bigger rewards? In the above Breakout game a simple strategy is to move to the left edge and wait there. When launched, the ball tends to fly left more often than right and you will easily score about 10 points before you die. Will you be satisfied with this or do you want more? This is called the explore-exploit dilemma should you exploit the known working strategy or explore other, possibly better strategies.

Reinforcement learning is an important model of how we (and all animals in general) learn. Praise from our parents, grades in school, salary at work these are all examples of rewards. Credit assignment problems and exploration exploitation dilemmas come up every day both in business and in relationships. Thats why it is important to study this problem, and games form a wonderful sandbox for trying out new approaches.

1.2 Markov Decision Process

Suppose you are an agent, situated in an environment (e.g. Breakout game). The environment is in a certain state (e.g. location of the paddle, location and direction of the ball, existence of every brick and so on). The agent can perform certain actions in the environment (e.g. move the paddle to the left or to the right). These actions sometimes result in a reward (e.g. increase in score). Actions transform the environment and lead to a new state, where the agent can perform another action, and so on. The rules for how you choose those actions are called policy. The environment in general is stochastic, which means the next state may be somewhat random (e.g. when you lose a ball and launch a new one, it goes towards a random direction).



The set of states and actions, together with rules for transitioning from one state to another, make up a Markov decision process. One episode of this process (e.g. one game) forms a finite sequence of states, actions and rewards:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, s_{n-1}, a_{n-1}, r_n, s_n$$

Here s_i represents the state, ai is the action and ri+1 is the reward after performing the action. The episode ends with terminal state sn (e.g. game over screen). A Markov decision process relies on the Markov assumption, that the probability of the next state si+1 depends only on current state si and action ai, but not on preceding states or actions.

1.3 Discounted Future Reward

To perform well in the long-term, we need to take into account not only the immediate rewards, but also the future rewards we are going to get. How should we go about that?
Given one run of the Markov decision process, we can easily calculate the total reward for one episode:

$$R = r_1 + r_2 + r_3 + \ldots + r_n$$

Given that, the total future reward from time point t onward can be expressed as:

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

But because our environment is stochastic, we can never be sure, if we will get the same rewards the next time we perform the same actions. The more into the future we go, the more it may diverge. For that reason it is common to use discounted future reward instead:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^{n-t} r_n$$

Here Y is the discount factor between 0 and 1 the more into the future the reward is, the less we take it into consideration. It is easy to see, that discounted future reward at time step t can be expressed in terms of the same thing at time step t+1:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

If we set the discount factor Y=0, then our strategy will be short sighted and we rely only on the immediate rewards. If we want to balance between immediate and future rewards, we should set discount factor to something like Y=0.9. If our environment is deterministic and the same actions always result in same rewards, then we can set discount factor Y=1.
A good strategy for an agent would be to always choose an action that maximizes the (discounted) future reward.

1.4 Q-Learning

In Q-learning we define a function Q(s, a) representing the maximum discounted future reward when we perform action a in state s, and continue optimally from that point on.

$$Q(s_t, a_t) = max\ R_{t+1}$$

The way to think about Q(s, a) is that it is the best possible score at the end of the game after performing action a in state s. It is called Qfunction, because it represents the quality of a certain action in a given state.
This may sound like quite a puzzling definition. How can we estimate the score at the end of game, if we know just the current state and action, and not the actions and rewards coming after that? We really cant. But as a theoretical construct we can assume existence of such a function. Just close your eyes and repeat to yourself five times: Q(s, a) exists, Q(s, a) exists, . Feel it?
If youre still not convinced, then consider what the implications of having such a function would be. Suppose you are in state and pondering whether you should take action a or b. You want to select the action that results in the highest score at the end of game. Once you have the magical Q-function, the answer becomes really simple pick the action with the highest Q value!

$$\pi(s) = argmax_a\ Q(s, a)$$

The above policy, the rule how we choose an action in each state. OK, how do we get that Q function then? Lets focus on just one transition <s, a, r, s>. Just like with discounted future rewards in the previous section, we can express the Q value of state s and action a in terms of the Q value of the next state s.

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

This is called the Bellman equation. If you think about it, it is quite logical maximum future reward for this state and action is the immediate reward plus maximum future reward for the next state. The main idea in Q-learning is that we can iteratively approximate the Q-function using the Bellman equation. In the simplest case the Q-function is implemented as a table, with states as rows and actions as columns. The gist of the Q learning algorithm is as simple as the following:

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

2 Atari 2600

The Atari 2600 (or Atari VCS before 1982) is a home video game console released on September 11, 1977, by Atari, Inc. It is credited with popularizing the use of microprocessor-based hardware and ROM

cartridges containing game code, a format first used with the Fairchild Channel F video game console in 1976. This format contrasts with the older model of having non-microprocessor dedicated hardware, which could only play the games that were physically built into the unit.
The console was originally sold as the Atari VCS, an abbreviation for Video Computer System. Following the release of the Atari 5200 in 1982, the VCS was renamed to the "Atari 2600", after the unit's Atari part number, CX2600. The 2600 was typically bundled with two joystick controllers, a conjoined pair of paddle controllers, and a game cartridge: initially Combat, and later Pac-Man.

2.1 Controls

The Atari 2600 comes with a joystick controller consisting of a d-pad joystick and a fire button. The joystick can move in 8 different direction and the fire button can be pressed with or without joystick movement, those along with no-move present the 18 valid controls available in Atari.

2.2 Screen

Atari emulator consists of a 160x192 pixel screen, (though mostly it is converted into a square for use), The small input size can be further reduced by converting the screen to greyscale and decreasing image resolution to 84x84 pixels, Transient data is included by including multiple screens (4).
Thus the atari games are perfect for deep learning due to their small input size (84x84x4) and small output size (18) which works well with a deep neural network.

3 Design

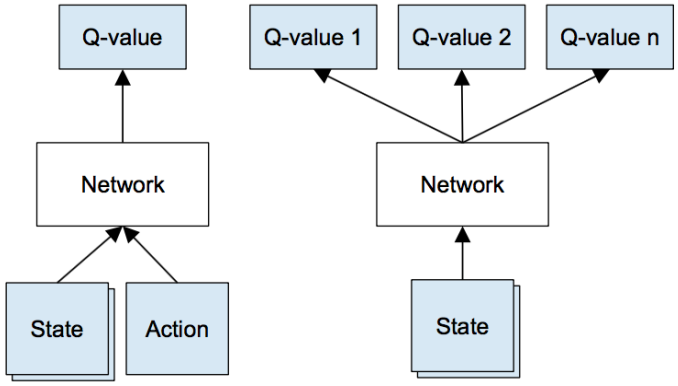
3.1 Q Learning to Deep Q Network

Although a simple game with small number of states like tic tac toe can be easily implemented using a Q table data structure, the same approach does not work for an Atari game.
The state of the environment in the Breakout game can be defined by the location of the paddle, location and direction of the ball and the presence or absence of each individual brick. But for something more universal, the obvious choice is screen pixels. They implicitly contain all of the relevant information about the game situation, except for the speed and direction of the ball. Two consecutive screens would have these covered as well.
Taking our input of 84x84x4 pixels, In greyscale each pixel can have 128 values (atari could only emulate 128 colours), This would mean that the number of states is $128^{84x84x4}$ which is more than the no of stars in the universe. Hence representing it in a Q table is out of the question.
Since there are a lot of states, repetitions of states are very rare, Q table does poorly on states that it has never encountered before, While a neural network will give us an approximate value for an unknown state based on its current learning.
The time complexity of both Q table and Neural network remain the same at O(1) after completion of training, Although neural network might be a bit slower on account of the forward propagation it has to complete.
Thus a shift from Q table approach to Deep Q network is done for the reasons

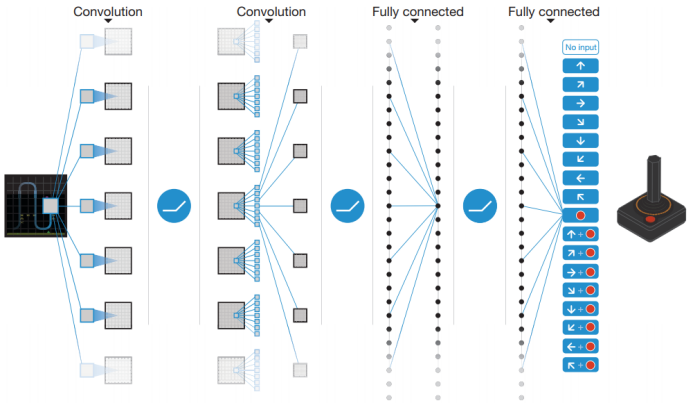
- Space complexity is containable even when number of states is very large
- Better approximation in un-encountered states
- Faster training with lesser data

3.2 Deep Q Network

Neural networks are exceptionally good at coming up with good features for highly structured data. We could represent our Q function with a neural network, that takes the state (four game screens) and action as input and outputs the corresponding Q value. Alternatively we could take only game screens as input and output the Q-value for each possible action. This approach has the advantage, that if we want to perform a Q value update or pick the action with the highest Q value, we only have to do one forward pass through the network and have all Q values for all actions available immediately.



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



3.3 Exploration Exploitation

Q-learning attempts to solve the credit assignment problem it propagates rewards back in time, until it reaches the crucial decision point which was the actual cause for the obtained reward. But we havent touched the exploration exploitation dilemma yet.
Firstly observe, that when a Q-table or Q-network is initialized randomly, then its predictions are initially random as well. If we pick an action with the highest Q-value, the action will be random and the agent

performs crude exploration. As a Q-function converges, it returns more consistent Q-values and the amount of exploration decreases. So one could say, that Q-learning incorporates the exploration as part of the algorithm. But this exploration is greedy, it settles with the first effective strategy it finds.
A simple and effective fix for the above problem is greedy exploration with probability choose a random action, otherwise go with the greedy action with the highest Q-value. In their system DeepMind actually decreases over time from 1 to 0.1 in the beginning the system makes completely random moves to explore the state space maximally, and then it settles down to a fixed exploration rate.

3.4 Deep Q-Algorithm

```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
     select an action a
          with probability ε select a random action
          otherwise select a = argmax_a' Q(s,a')
     carry out action a
     observe reward r and new state s'
     store experience <s, a, r, s'> in replay memory D

     sample random transitions <ss, aa, rr, ss'> from replay memory D
     calculate target for each minibatch transition
          if ss' is terminal state then tt = rr
          otherwise tt = rr + γmax_a' Q(ss', aa')
     train the Q network using (tt - Q(ss, aa))^2 as loss

     s = s'
until terminated
```

3.5 Class Diagram