Deep Reinforcement Learning

A quick review
by
Ali Sharifi Boroujerdi

- Reinforcement learning is an important model of how we (and all animals in general) learn.
- Reinforcement learning lies somewhere in between supervised and unsupervised learning.
- 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.
- **credit assignment problem**: which of the preceding actions was responsible for getting the reward and to what extent.
- **explore-exploit dilemma**: should you exploit the known working strategy or explore other, possibly better strategies.
- **Markov Decision Process:** is the most common method to formalize a reinforcement learning problem.
- An **agent**, situated in an **environment**. The environment is in a certain **state**. The agent can perform certain **actions** in the environment. These actions sometimes result in a **reward**. 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.
- The set of states and actions, together with rules for transitioning from one state to another, make up a Markov decision process.
- **Markov assumption:** the probability of the next state s_{i+1} depends only on current state s_i and action a_i, but not on preceding states or actions.

- One **episode** of this process forms a finite sequence of states, actions and rewards.
- To perform well in the long-term, we need to take into account not only the immediate rewards, but also the future rewards.
- 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 by inserting **discount factor (γ)** into total future reward so that the more into the future the reward is, the less we take it into consideration.
- A good strategy for an agent would be to always choose an action that maximizes the **(discounted) future reward**.
- In **Q-learning** we define a [magical] 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. It is called Q-function, because it represents the "**quality**" of a certain action in a given state.
- The **Bellman equation** expresses the Q-value of state s and action a in terms of the Q-value of the next state s'. We can iteratively approximate the Q-function using the Bellman equation.
- Q-learning algorithm:

```
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
```

- α in the algorithm is a learning rate that controls how much of the difference between previous Q-value and newly proposed Q-value

is taken into account. In particular, when α=1, then two Q[s,a] cancel and the update is exactly the same as the Bellman equation.

- Ideally, we would like to have a good guess for Q-values for states we have never seen before. This is the point where deep learning steps in. 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.
- The network architecture that DeepMind used is as follows:

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

- For some games the location of the ball is crucial in determining the potential reward and we wouldn't want to discard this information, therefore there are no pooling layers in this network. (see Quantifying Translation-Invariance in Convolutional Neural Networks paper)
- Input to the network are four 84×84 grayscale game screens. Outputs of the network are Q-values for each possible action (18 in Atari).
- Q-values can be any real values, which makes it a regression task, that can be optimized with simple squared error loss:

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

- Workflow:
    - Do a **feedforward pass for the current state** s to get predicted Q-values for all actions.
    - Do a **feedforward pass for the next state** s' and calculate maximum overall network outputs max a' Q(s', a').
    - Set Q-value target for action to r + γmax a' Q(s', a') (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
    - Update the weights using **backpropagation**.
- Approximation of Q-values using non-linear functions is not very stable. There is a whole bag of tricks that you have to use to actually make it converge. And it takes a long time, almost a week on a single GPU.
- The most important trick is **experience replay**:
    - During gameplay all the experiences < s, a, r, s' > are stored in a replay memory. When training the network, random mini batches from the replay memory are used instead of the most recent transition. This breaks the similarity of subsequent training samples, which otherwise might drive the network into a local minimum. Also experience replay makes the training task more similar to usual supervised learning, which simplifies debugging and testing the algorithm. One could actually collect all those experiences from human gameplay and then train network on these.
- Q-learning incorporates the exploration as part of the algorithm. But this exploration is "greedy", it settles with the first effective strategy it finds. Effective fix for the above problem is **ε-greedy exploration**: with probability ε choose a random action, otherwise go with the "greedy" action with the highest Q-value.
- In their system ***DeepMind*** actually decreases ε over time from 1 to 0.1 in such a way that 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.

- Final deep **Q-learning algorithm with experience replay**:

```
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
```

- There are many more tricks that DeepMind used to actually make it work such as **target network**, **error clipping** and **reward clipping**.

Reference:

[Demystifying Deep Reinforcement Learning]