Q-learning for mathematical applications

Andreas Bøgh Poulsen
201805425

10th March 2021

1 Preliminaries

1.1 Abstract framework of deep learning

In abstract, modern machine learning is about fitting a universal approximator to some given data. In recent instances of it, this universal approximator would be a deep neural network, possibly with some variations like convolution, recurrence or attention. Using differentiation of a neural network and gradient descent we can minimize the error between the predictions of the network and some training data.

I will not spend much time on this part and take the training of a neural network for granted. The important point is that we can minimize any error function, not just the difference between a prediction and some known data.

This is an important insight. If we can check if the network is producing right result after the fact, we don't have to know the facit beforehand. This is good because we can use the network to discover solutions without having to actually solve the problem ourselves. For example, if we wanted to learn to find a divisor of a number, it's a lot easier to check a solution than finding a factor ourselves, so this is a good problem for machine learning. We can just set the error to zero if we found a factor and 1 if we didn't. On the other hand, it's difficult to check wether a number is a prime number or not, so this probably wouldn't be a good problem to solve with machine learning.

1.2 Reinforcement-learning

Let's build a bit more on the insight from before: we can minimize any error function. We can use this to extend the domain of our learning. One interesting case is learning to interact with an environment. This could be a robot interacting with a physical environment or a mathematician trying to prove a theorem. Both cases can be thought of as an agent doing an action, seeing how it worked out and taking a new action. Let's model such an interaction formally.

1.1 · Definition. Fix two non-empty sets S, A and call S the *state-space* and A the *action-space*. Then, an *environment* is a tuple $(S, A, P, R, \epsilon, t)$ where

- $P : S \to 2^A$ gives the permissible actions of a given state.
- $R : s \in S \times P(s) \to \mathbb{R}$ is called the *reward-function*.
- $\epsilon : s \in S \times P(s) \to S$ is called the *update-function*. This might a probabilistic function i.e. non-deterministic.
- $t : S \to \{True, False\}$ determines if a state is a *terminal* state.

Given an environment, an *agent* is a function $M : s \in S \times P(s) \to A$.

In most cases all actions are always permissible, i.e. $P(s) = A$ for all $s \in S$. The intuition is the following setup:

(1) An agent observes an environment in state s and decides to take some action $a \in P(s)$.

(2) A reward of $R(s, a)$ is given to the agent to tell it wether the action was good or not.

(3) The environment is updated to $(S, s' = \epsilon(s), A, P, R, \epsilon, t)$.

(4) If $t(s') = True$ the interaction is done. If not, repeat from step 1.

Of course the idea is that the agent would learn to take actions in such a way that the reward is maximized.

In order to facilitate learning, we need to repeat the process described above a number of times, so let's fix some naming and notations for that:

1.2 · Definition. Given an *environment* $(S, A, P, Q, \epsilon, t)$, an *agent* $M : s \in S \times P(s) \to A$ and an initial state $s_0 \in S$ a *Markov decision process* is an iterative

process of states and actions, defined recursively as

$$s_0 = s_0 \tag{1}$$
$$s_{t+1} = \epsilon(s_t, a_t) \tag{2}$$
$$\text{where } a_t = M(s_t) \tag{3}$$

A *trajectory* is a record of a Markov Decision Process, formally a tuple

$$(s_t, a_t, r_t, t_t)$$

where s_t and a_t are described above and $r_t = R(s_t, a_t)$, $t_t = t(s_t)$.

2 Learning strategies

The abstract framework of Markov Decision Processes is a rather simple idea. And indeed, that is not where the magic of reinforcement learning is. None of the above even hinted at how any learning would occur, we have only simulated an environment and doled our rewards. It shouldn't be surprising that there are many strategies when it comes to learning. We'll outline few of them here, as well as few "tricks" that can be used to improve performance.

We'll focus on temporal-difference strategies as these strategies are most widely used today. The idea here is to do an update at every step along the trajectory, trying to figure out if the step our agent just took was a good or a bad step.

In order to illustrate these ideas, we'll use tic-tac-toe as example. I hope that this is a familiar game to everyone. I will focus on the pen-and-paper version, where a piece is placed and never moved, but the ideas carry over to other variations of the game

2.1 Value-estimation

Think about how you would play tic-tac-toe yourself. At least when I play, I have the following heuristc: a situation where my opponent has an opportunity to win is bad. A situation where I have an opportunity to win is good. A situation where I have two opportunities to win is even better and similarly worse if my opponent have two opportunities. Also, a situation that might lead me a very good situation is pretty good, for example if I'm cross here and it's my turn:

		X
	O	
X		O

All these considerations come together, and I quickly give each situation a score and choose the action that is most likely to lead me to a good situation.

This might lead us to a good learning strategy. If we can learn how valuable each possible situation is, we're in pretty good shape. If we can then estimate how likely an action is to lead us to each situation, we can choose the action that maximizes value.

This idea can be treated using the Bellman equation. We'll consider the deterministic case first, where the Bellman equation is simply

$$V(s_0) = \max_{a_0}\{R(s_0, a_0) + V(x_1)\}, \text{ where } x_1 = \epsilon(s_0, a_0)$$

The idea is simple, the value of a given state is the value of the best next state plus the reward given by progressing to that state.

TODO: extend this to stochastic case

2.2 Q-learning

In value-estimation we determine how valuable a given state is. However, that is only a partial picture. We got into some trouble as soon as we can't predict the next state from any particular action. Another approach is Q-learning, which a somewhat different perspective.

Instead of assigning a value to each state, we estimate the expected reward coming from a particular *action*. That is

$$Q(s, a) = \mathbb{E}[\sum_{i=t}^{T} R(s_i, a_i) | s_t = s, a_t = a, \text{agent } m]$$

.

Q-learning is one of the first reinforcement learning algorithms that achieved break-through performance. One the first examples was the program TD-gammon, a program that learned to play backgammon. TD-gammon was novel because it exploited no domain knowledge of backgammon at all. It simply employed Q-learning and yet was able to match the performance of the best backgammon-bots of the time. These other bots all used extensive expert knowledge of the game so it was remarkable that Q-learning could learn to play as well as them. The first iteration of TD-gammon used simply naive Q-learning. Later iterations achieved even better performance by adding Monte Carlo Tree

search (which we'll cover later) to the algorithm. While this did improve performance it's remarkable that high performance can be achieved without anything but a simple learning algorithm.

In small cases, we can keep the Q-function in a simple lookup table. So, given a state and an action, we can do a lookup to find our estimation of the future reward given that action. In this case, the learning algorithm is simply:

$$Q(s_t, a_s) \leftarrow \begin{cases} (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \max_a Q(s_{t+1}, a)) & \text{if } s_t \text{ is non-terminal} \\ (1-\alpha)Q(s_t, a_t) + \alpha r_t & \text{if } s_t \text{ is terminal} \end{cases}$$

where α is the learning rate. The intuition here is that we now know what reward we got by taking action a_t. Therefor we can update our estimate to be a combination of our old estimate and an updated estimate based on the observed reward plus our estimate for the rest of the process.

Let's see how that would work in the our tic-tac-toe example. Whenever the agent makes a move we give it a reward of +1 if it won by that move and −1 if the opponent won on their following move. This is typical for reinforcement learning; we give a reward when a task is completed and expect our agent to learn how to perform the entire task. It sounds a bit unreasonable, doesn't it? It's obvious that the action immediately before a reward was a good action, but what about the action before that? It might have been good, or it might have been bad even though we won the action after. The cause-and-effect chain gets weaker every link. In practice, it works anyway.

Consider what happens after we won a game. The agent updates that the last action was a good one, so $Q(s_T, a_T)$ where T is the last step of the trajectory, grows. What about the rest of the actions? Well, they are not immediately updated. However, the next time the agent faces a situation just before the winning move a_T, the estimated future rewards from that situation grows since we know we can win after it. Similarly, next time the agent faces a situation just before the one we just considered, the reward is propagated backwards.

The backwards propagation of rewards allows the agent to learn complex strategies. Even though any one trajectory may contain good and bad moves, on average trajectories that leads to a win will share good moves and trajectories leading to a loss wil share bad moves. It may seem a bit wishful to think at they will cancel each other out, Watkins and Dayan proved convergence to optimal strategy.

2.2.1 Deep Q-learning

A major challenge for Q-learning is the table of states and actions. As our problem grows in complexity, this table becomes large very quickly. To remedy

this, modern Q-learning uses function approximation to learn a single $Q : state \times action \to reward$. The DeepMind team used in 2015 a convolutional neaural network to learn the Q-function and succeeded in learning to play a variety of 49 different Atari games.

In this setting, we need to adapt the learning formula given above. In that case we updated our estimated Q-function to be a linear combination of the previous estimate and our new knowledge. This can be viewed as a simple form of gradient descent towards the observed data.

To do gradient descent on a neural network, we need a loss function L that is the difference between our current estimate and the observed data. It looks like this:

$$L_t = \begin{cases} (r + max_a Q(s_{t+1}, a) - Q(s_t, a_t))^2 & \text{if } s_t \text{ is non-terminal} \\ (r - Q(s_t, a_t))^2 & \text{if } s_t \text{ is terminal} \end{cases}$$

By computing gradients on this function we can update our neural network (or other function approximator) towards the updated values.

2.2.2 Experience replay

Normal Q-learning is a temporal-difference learning method, which means it only learns from the timestep it just took. Experience replay keeps track of previous experiences and uses them when doing an update. To do this, we keep a trajectory of the currect episode, sample some random times from it and perform gradient descent on every remembered timestep simultaneously. Specifically, for a random sample $D = \{t_i\} \subset \mathbb{N}$ of timesteps, we compute the vector

$$G_t = \begin{cases} r_t + max_a Q(s_{t+1}, a) & \text{if } s_t \text{ is non-terminal} \\ r_t & \text{if } s_t \text{ is terminal} \end{cases}$$

for every $t \in D$ and let our loss-function be the euclidean distance from G to the vector $H_t = Q(s_t, a_t)$ for $t \in D$.

There is no proven reason why experience replay should help learning. DeepMind provides that it reduces variance in the gradients, which would help stabilize the learning. I don't want to risk saying something wrong, so I'll just say that it seems to work and that's a good enough reason to use it.

2.3 Policy gradient

Value estimation and Q-learning had the nice property that there was in some sense a concrete function we learned, the value associated with something.

Policy gradient is different and exploits the backpropagation algorithm. The idea is that instead of trying to approximate something that we know, compute an error function between our current agent and some observed data, we can just specify that some behaviour was good and have the neural network optimize towards doing that again.

The technique is to simply posit that the final gradient of the network must be 1, if we just won the game. Then we can use backpropagation to compute derivatives on every part of the network and update accordingly. This way, good actions get a higher score by the network and bad actions get a lower score. What does this score mean? Who cares! Higher means better and that's usually all we need.

This idea is almost too simple. We don't try to learn any sort of value function that is a proxy for what would be a good action, we can simply tell the network "this was a good sequence of actions. Do those more".

A policy gradient learner would explore the environment by sampling which action to take weighted by the learned scores. This ensures that the learner takes new actions, but it also keeps it somewhat on course with respect to what it learned so far.

Let's work through a game of tic-tac-toe. The game looks like this:

At each step the agent takes the action having the highest score. Only at the end of the game can we update our network, since that's the only point at which we know anything. At the end, the reward is 1, so that's the gradient of all the previous steps. Backpropagating and doing gradient descent increases the score of those actions in the future, making them more likely to be taken. Notice that action nr. 3 actually didn't go directly for the win and instead opted to prevent the opponents win. I have two comments to make here: First, this behaviour is not punished by our reward. As long as the agent makes sure to win eventually (which it does here by giving itself two winning moves) this is as good a play as winning as quickly as possible. Second, this behaviour might be discouraged over time by the fact that we do sample actions randomly and an immediate win gives a more certain reward than a delayed win. Note that we do not reward the sweet feeling of superiority given by having two winning moves.

2.3.1 Proximal policy gradient

3 Other methods

3.1 Monte Carlo Tree Search

3.2 Advantage Estimation