Reinforcement learning for mathematical applications

Andreas Bøgh Poulsen
201805425

6th April 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.

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, here placing a piece in the upper left corner.

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, but 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 $Q : state \times action \rightarrow reward$. 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.

That was vague, so let's do some math. We have an agent with some parameters θ and a reward function $F(t)$ taking a trajectory t and giving us a total reward for that trajectory. Since we want to optimize $E[F(t) \mid \theta]$ we can do that by taking the gradient wrt. *theta* i.e. $\nabla_\theta E[F(t) \mid \theta]$. This expression is difficult to compute, so let's do some rearranging:

$$\begin{aligned}
\nabla_\theta E[F(t) \mid \theta] &= \nabla_\theta \sum_t P(t \mid \theta) F(t) \\
&= \sum_t F(t) \nabla_\theta P(t \mid \theta) \\
&= \sum_t F(t) P(t \mid \theta) \frac{\nabla_\theta P(t \mid \theta)}{P(t \mid \theta)} \\
&= \sum_t P(t \mid \theta) \nabla_\theta \log P(t \mid \theta) \qquad \text{since } \frac{\nabla_\theta x}{x} = \nabla_\theta \log x. \\
&= E[F(t) \nabla_\theta \log P(t \mid \theta)]
\end{aligned} \tag{4}$$

Now we're in good shape. Since log is a monotonically growing function we now get, that if $F(t) > 0$ we can increase $E[F(t) \mid \theta]$ by increasing $P(t \mid \theta)$, at least locally. Similarly, if $F(t) < 0$, decreasing $P(T \mid \theta)$ will increase $E[F(t) \mid \theta]$.

Remark. The above derivation can be done more generally than we've done in 4. By assuming an arbitrary, not necessarily discrete, probility distribution of states, we could replace the sum with an integral for a more general derivation.□

Right, so how do we increase the probability of a trajectory? Intuitively, we just increase the probability of the actions the trajectory consists of, and this turns out to be sufficient. Since each action is taken independently of previous actions, we have

$$P(t \mid \theta) = P(s_0) \prod_{i=0}^{T-1} \pi_\theta(a_i \mid s_i) P(s_{i+1}, r_i \mid s_i, a_i)$$

where pi_θ gives the probability distribution of actions that the agent is sampling from. Taking the logarithm and gradient on both sides, we get that

$$\nabla_\theta \log P(t \mid \theta) = \nabla_\theta \sum_{i=0}^{T-1} \log \pi_\theta(a_i \mid s_i)$$

and thus, by taking this together with 4 that

$$\nabla_\theta E[F(t) \mid \theta] = E\left[F(t) \nabla_\theta \sum_{i=0}^{T-1} \log \pi_\theta(a_i \mid s_i) \right]$$

and this finally gives us a good way to update our neural network: simply increase $\pi_\theta(a_i \mid s_i)$ by following the gradient of the neural network.

Now, this is great if we have direct access to a gradient descent optimizer. However, most modern machine learning frameworks is fixed to the model of minimizing an error function. The solution is very simple; to maximize $F(t)\sum_i \log \pi_\theta(s_i \mid s_i)$ we can minimize $-F(t)\sum_i \log \pi_\theta(a_i \mid s_i)$. Thus our loss could be just that.

2.3.1 Normalization

There are a number of challenges when doing policy gradient learning in practise. One is that rewards are often very high variance and not balanced. For example, if we only care about achieving a goal, every reward will be positive so every action will be encouraged. The reason things still work in this situation is that actions leading to higher rewards will lead to steeper gradients and be "encouraged more". It might give better results if we could normalize our rewards by subtracting the mean and dividing by the standard deviation.

One way of achieving this is to normalize the rewards from every episode indpedendantly of other episodes. This gives better result than the naive approach. Another method is to train another agent to be the baseline, preferably using a simpler learning technique, and use the performance of that agent to approximate the mean of the rewards.

3 Other methods

3.1 Monte Carlo Tree Search

3.2 Advantage Estimation

4 Gröbner bases

Since the problem we're considering id the construction of Gröbner bases, let's give a short introduction to Gröbner bases.

We fix a field k and consider the polynomial ring $R = k[x_1, \ldots, x_n]$. For at set of polynomials $F = \{f_1, \ldots, f_l\}$ we consider the ideal $I = \langle f_1, \ldots, f_l \rangle$ generated by these polynomials. Now, we wish to efficiently determine whether a given polynomial f lies in I or not. We know that R is a *unique factorization domain* so it is decidable, but giving an efficient algorithm is tricky. Instead, we extend the

generating set of the ideal in a way that doesn't change the ideal but gives us stronger properties, enabling an efficient desicion algorithm.

Fix a *term order* which is a well-order relation $>$ on $\mathbb{N}^n$ such that $a > b$ implies $a + c > b + c$ for any $a, b, c \in \mathbb{N}^n$. This naturally extends to a so called *monomial order* on monomials $\{x^v = x_1^{v_1} \cdots x_n^{v_n} \mid v \in \mathbb{N}^n\}$ by comparing exponent vectors of the monomials. We'll write $>$ for both orders.

There are two common term orders: lexicographic and grevlex. Lexicographic has $a > b$ if there exists k s.t. $a_i = b_i$ for $i < k$ and $a_k > b_k$. This is the usual "alphabetix order" on tuples. Grevlex ordering is often used in implementations using Gröbner bases and has $a > b$ if $\sum_i a_i > \sum_j b_j$ or $\sum_i a_i = \sum_j b_j$ and the last non-zero entry of $a - b$ is negative.

Given a monomial order we can define the *initial term* and the S-polynomial.

4.1 · Definition. Given a monomial order $>$, the *initial term* of a polynomial $f = \sum_v \lambda_v x^v$, denoted $in_>(f)$ is the greatest term of f with respect to $>$. We will often omit the subscript when the order is either clear from context or arbitrary.

Now, a naive division algorithm would look like this:

Algorithm 1: Division algorithm $reduce(f, F)$

Input: Polynomial f and $F = \{f_1, \ldots, f_l\}$
Output: Remainder r s.t. $f - r \in \langle F \rangle$ and $in(f_i) \nmid in(r)$ for all $f_i \in F$

```
r ← f
while ∃i.in(f_i) | in(r) do
    r ← r − (in(r)/in(f_i)) f_i
```

This algorithm terminates since the inistal term of r is strictly descreasing with respect to $>$.

Now, a Gröbner basis for an ideal I is a set of polynomials $F = \{f_1, \ldots, f_l\} \subseteq$ such that $in(f_i) \mid in(f)$ for all $f \in I \setminus \{0\}$.

4.2 · Theorem. *Let $G = \{f_1, \ldots, f_l\}$ be a Gröbner basis for an ideal I. Then $reduce(f, G) = 0 \iff f \in I$.*

Proof. If $reduce(f, G) = 0$ then $f = f - 0 \in I$.

If $f \in I$ and $f - r = h \in I$ then $r = f - h \in I$. But as guaranteed by the division algorithm, there is no $f_i \in G$ where $in(f_i) \mid in(r)$ even though G is a Gröbner basis. The only element in I not subject to the Gröbner basis constraint is zero, thus r must be zero. □

An example where this does not hold without a Gröbner basis is example 5.4.3 in NL.

So, Gröbner bases are great, but how do we construct them? Do they even exist for every ideal? They do exist and we have an algorithm called Buchbergers algorithm to find them, bt first we need a construction called the S-polynomial or syzygy polynomial.

4.3 · Definition. The S-polynomial of two polynomials f and g is denoted $S(f,g) = \frac{x^w}{in(f)}f - \frac{x^w}{in(g)}g$ where $x^w = lcm(in(f), in(g))$ is a least common multiple of f and g.

The Buchberger criterion is a simple was to check if we have a Gröbner basis, and it even leads us to an algorithm for constructing Gröbner bases.

4.4 · Theorem. *Let $F = \{f_1, \ldots, f_l\}$ be a set of polynomials and let $I = \langle F \rangle$ be the ideal generated by F. If $reduce(S(f_i, f_j), F) = 0$ for all $f_i, f_j \in F$ then F is a Gröbner basis for I.*

Proof. See appendix. □

We can now present the Buchberger algorithm:

Algorithm 2: Buchbergers algorithm

Input: A set of polynomials $F = \{f_1, \ldots, f_l\}$
Output: A Gröbner basis G of the ideal $I = \langle F \rangle$

```
1 G ← F
2 P ← {(f_i, f_j) | 1 ≤ i < j ≤ l}
3 while |P| > 0 do
4     (g, h) ← select(P)
5     P ← P \ {(g, h)}
6     r ← reduce(S(g, h), G)
7     if r ≠ 0 then
8         P ← update(P, G, r)
9         G ← G ∪ {r}
```

Notice that the algorithm uses 2 routines we haven't defined: *select* and *update*. The simplest implementations are select(P) = P_1 taking the first pair in P and update(P, G, r) = $P \cup \{(f, r) \mid f \in G\}$.

Buchbergers algorithm adds remainders of syzygy polynomials until they all reduce to 0. Keeping theorem 4.4 in mind, it is clear that G will be a Gröbner basis when the algorithm terminates. TODO: dicksons lemma giver at buchberger terminerer.

To produce a faster version of Buchbergers algorithm, these two routines are good places to start. During *update* we can eliminate a large number of pairs using simpler criteria than theorem 4.4. We'll discuss a better version in section 4.1.

The problem of selecting the next pair of polynomials is not easy and it turns out to have serious consequences. Simply switching from taking the first pair to taking the last one, i.e. treating P as a stack instead of a queue, can reduce the number of poynomial additions by 50%.

There are a number of standard selection strategies:

Random Pick a random pair uniformly.

First Pick the lexicographically smallest pair, where the order of polynomials is given by their order in G.

Degree Pick the pair (f, g) with the smallest total degree of $lcm(in(f), in(g))$.

Normal Pick the pair (f, g) where $lcm(in(f), in(g))$ is smallest in the used monomial order.

4.1 A better *update*

In the Buchberger algorithm, a pair (f, g) will be removed if reduce$(S(f, g), G) = 0$. However, this is computationally expensive test. Fortunately, there are some easier tests that can eliminate a large number of pairs. I'll describe the tests described in Gebauer & Möller.

When adding a reduced polynomial r to the basis, i.e. when calling update(P, G, r), consider every pair $(f, g) \in P$ and denote $\gamma = lcm(i(f), i(g))$. Then we can remove (f, g) from P if $\gamma \mid i(r)$ and $\gamma \neq lcm(i(f), i(r))$ and $gamma \neq lcm(i(g), i(r))$.