Reinforcement learning for mathematical applications

Andreas Bøgh Poulsen

201805425

28th 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 *policy* is a propability distribution $\pi(s \mid a)$ of taking an action a given a state s.

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

$$\tau = (s_t, a_t, r_t, t_t)$$

where s_t and a_t are described above and $r_t = R(s_t, a_t)$, $t_t = t(s_t)$. Given a trajectory τ, τ_p is τ starting from index p.

A trajectory might be finite or infinite. In order to give a meaningful reward for an infinite trajectory, we'll introduce a *discount* factor γ, usually given the value 0.99.

Given a trajectory $\tau = (s_t, a_t, r_t, t_t)$ we will define the *value* of this trajectory

$$G(\tau_p) = \sum_{k=p}^{T} \gamma^k r_k.$$

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 and Q-learning

Almost all reinforcement learning is about estimating value functions – how good a state or action is. For example, we can define the *value* of a state s given an agent π to be

$$v_\pi(s) = \mathbb{E}\Big[G(\tau_p) \mid s_p = s, \pi\Big] = \mathbb{E}\left[\sum_{k=p}^{T} \gamma^k r_k \mid s_p = s, \pi\right].$$

The value-function can be expressed recursively:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[G(\tau_p) \mid s_p = s, \pi] && (4)\\ &= \mathbb{E}\left[\sum_{k=p}^{T} \gamma^k r_k \mid s_p = s, \pi\right] && (5)\\ &= \mathbb{E}\Big[r_p + \gamma G(\tau_{p+1}), \mid s_p = s, \pi\Big] && (6) \end{aligned}$$

If we define $p(s' \mid s, a) = P(\epsilon(s, a) = s')$ i.e. the probability of reaching state s' from state s by taking action a, we can further use the law of total probaility to get

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a \mid s) \sum_{s'} p(s' \mid s, a)\Big(R(s, a) + \gamma \mathbb{E}[G_{p+1} \mid s_p = s, \pi\Big) && (7)\\ &= \sum_a \pi(a \mid s) \sum_{s'} p(s' \mid s, a)(R(s, a) + \gamma v_\pi(s')) && (8) \end{aligned}$$

By shifting our focus slightly to consider the value of *actions* instead of states, we can define an *action-value* function, denoted $q_\pi(s, a) = \mathbb{E}[G(\tau_p) \mid s_p = s, a_p = a, \pi]$.

Using these we can define the *optimal* value functions:

2.1 · Definition.

$$v_*(s) = \max_\pi v_\pi(s)$$

is called the *optimal value function* and

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

is called the *optimal action-valur function*

These two functions can be expressed in terms of each other:

$$\begin{aligned} v_*(s) &= \max_\pi v_\pi(s) \\ &= \max_\pi \mathbb{E}[G(\tau_p \mid s_p = s, \pi)] \\ &= \max_\pi \sum_a \pi(a \mid s)\,\mathbb{E}[G(\tau_p) \mid s_p = s, a_p = a, \pi] \\ &= \max_\pi \sum_a \pi(a \mid s)\, q_\pi(s,a) \\ &= \max_a q_*(s,a) \end{aligned}$$

and similarly we can get

$$q_*(s,a) = R(s,a) + v_*(s')$$

. These of course also satisfies the recursive relations from before:

$$\begin{aligned} v_*(s) &= \max_a q_*(s,a) \\ &= \max_a \mathbb{E}[G(\tau_p) \mid s_p = s, a_p = a, \pi_*] \\ &= \max_a \mathbb{E}[r_p + \gamma v_*(s_{p+1}) \mid s_p = a, a_p = a, \pi_*] \\ &= \max_a \sum_{s'} p(s' \mid s,a)(R(s,a) + \gamma v_*(s')) \end{aligned}$$

which is called the Bellman equation for v_*. The Bellman equation for q_* is

$$q_*(s,a) = \sum_a p(s' \mid s,a)(R(s,a) + \gamma \max_{a'} q_*(s',a')).$$

These equations inspires the learning equation

$$V(s_p) = (1-\alpha)V(s_p) + \alpha \max_a \{R(s_p,a) + \gamma V(s')\} \text{ where } s' = \epsilon(s_p,a).$$

Here, α is called the learning rate. Convergence of this learning equation is guaranteed by a general theorem in control theory, see [WD92].

We get a similar update equation for Q-learning:

$$Q(s_p,a_p) = (1-\alpha)Q(s_p,a_p) + \alpha(R(s_p,a_p) + \gamma \max_a Q(s',a))$$

Example. Consider the graph in figure 1. If the agent initially takes each possible action with equal probability, we might observe the trajectory $\tau = (A,B,C,A,B,A,B,C,B,C)$. With learning rate $\alpha = 0.5$ and decay $\gamma = 1$ we get the following updates:

$$\begin{tikzcd}
\boxed{A} \arrow[rr, bend left, "15"] & & \boxed{B} \arrow[ll, bend left, "3"] \arrow[dl, bend left, "1"] \\
& \boxed{C} \arrow[ul, "20"] \arrow[ur, bend left, "8"] &
\end{tikzcd}$$

$\square$

Figure 1: A simple graph-based environment

s	s'	r	Update		
A	B	15	$Q(A,B) = \frac{1}{2}0 + \frac{1}{2}(15+0)$	=	7.50
B	C	1	$Q(B,C) = \frac{1}{2}0 + \frac{1}{2}(1+0)$	=	0.50
C	A	5	$Q(C,A) = \frac{1}{2}0 + \frac{1}{2}(20+7.5)$	=	13.70
A	B	15	$Q(A,B) = \frac{1}{2}7.5 + \frac{1}{2}(15+0.5)$	=	11.40
B	A	3	$Q(B,A) = \frac{1}{2}0 + \frac{1}{2}(3+11.4)$	=	7.20
A	B	15	$Q(A,B) = \frac{1}{2}11.4 + \frac{1}{2}(15+7.2)$	=	16.80
B	C	1	$Q(B,C) = \frac{1}{2}0.5 + \frac{1}{2}(1+13.7)$	=	5.70
C	B	8	$Q(C,B) = \frac{1}{2}0 + \frac{1}{2}(8+7.2)$	=	11.20
B	C	1	$Q(B,C) = \frac{1}{2}5.7 + \frac{1}{2}(1+13.7)$	=	10.10

Yielding the final values

$$\begin{aligned}
Q(A,B) &= 16.80 \\
Q(B,A) &= 7.20 \\
Q(B,C) &= 10.10 \\
Q(C,A) &= 13.70 \\
Q(C,B) &= 11.20
\end{aligned}$$

As we see, the agent have already learned the non-obvious but optimal tour $A \rightarrow B \rightarrow C \rightarrow A \rightarrow \ldots$ since the large reward from going $C \rightarrow A$ is propagated backwards in line 7.

2.2 Deep Q-learning

The method described above is called tabular learning, as it stores every state-action mapping in a table. This becomes infeasible as the state- and action-space grows. Deep Q-learning attempts to approximate the Q-function with a single neural network. The change is simple, instead of directly updating $Q(s,a)$ as above we compute the loos $loss = (Q(s_p, a_p) - R(s_p, a_p) + \max_a Q(s_{p+1}, a))$ and use backpropagation to copmute the gradient of that loss function. Then we can use gradient descent to minimize the loss.

2.2.1 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 $G(\tau)$ taking a trajectory τ and giving us a total reward for that trajectory. Since we want to optimize $E[G(\tau) \mid \theta]$ we can do that by taking the gradient wrt. *theta* i.e. $\nabla_\theta E[G(\tau) \mid \theta]$. This expression is difficult to compute, so let's do some rearranging:

$$\begin{aligned}
\nabla_\theta E[G(\tau) \mid \theta] &= \nabla_\theta \sum_t P(t \mid \theta) G(\tau) \\
&= \sum_t G(\tau) \nabla_\theta P(t \mid \theta) \\
&= \sum_t G(\tau) P(t \mid \theta) \frac{\nabla_\theta P(t \mid \theta)}{P(t \mid \theta)} \qquad (9) \\
&= \sum_t G(\tau) P(t \mid \theta) \nabla_\theta \log P(t \mid \theta) \quad \text{since } \frac{\nabla_\theta x}{x} = \nabla_\theta \log x. \\
&= E[G(\tau) \nabla_\theta \log P(t \mid \theta)]
\end{aligned}$$

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

Remark. The above derivation can be done more generally than we've done in 9. 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 π_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 9 that

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

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.

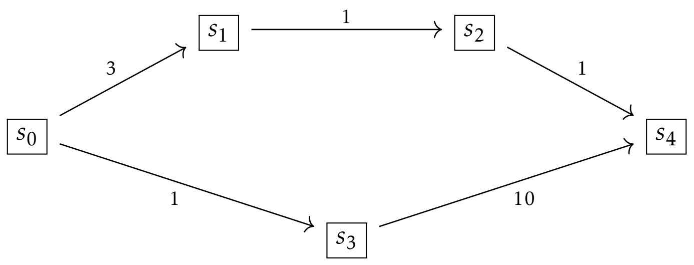

Figure 2: Example of a situation in which normalization of rewards is a bad idea.

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 $G(\tau)\sum_i \log \pi_\theta(s_i \mid s_i)$ we can minimize $-G(\tau)\sum_i \log \pi_\theta(a_i \mid s_i)$. Thus our loss could be just that. A small tweak would be to use the inner product between the rewards vector and the log-probabilities, i.e. $\overrightarrow{r} \cdot \log \pi(\overrightarrow{a} \mid \overrightarrow{s})$ instead of just the product of the sums. This ensures that if we took an unlikely action and it paid off, we encourage that more.

2.3.1 Normalization & Advantage estimation

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 without the possibility of failure, 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". In this case we can better performance by normalizing rewards by subtracting the mean and dividing by the standard deviation. This of course only works if we use the inner product loss described above, as the loss would otherwise just be 0.

This method, however, is not always a good idea. Consider for example the graph in figure 2 with two possible reward trajectories: $r_1 = (1, 10)$ and $r_2 = (3, 1, 1)$. Normalized, these become $r_1' = (-0.99, 0.99)$ while $r_2' = (1.41, -0.7, -0.7)$, hence we encourage the second trajectory. This method works if we can take actions independently of previous actions but in other cases it might not. It gets better if we maintain a running average, but it can still fail.

Another way of normalizing is by comparing to a baseline. We could train a value estimator alongside our policy gradient network and use the predictions from this network to normalize the observed rewards. That is, if we observe a trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$ and we have network approximating $r'_p = \mathbb{E}[r_p \mid s_p]$ we can normalize to get $\tau' = (s_0, a_0, r_0 - r'_0, \dots)$. If the approximation is good, this would reduce the variance.

2.4 Advantage estimation

An alternative to normalization is *advantage estimation*. Here, we compute a *value function* of each state and normalize using the value difference. Specifically, given a trajectory $\tau = (\overrightarrow{s}_t, \overrightarrow{a}_t, \overrightarrow{r}_t)$ we compute values $v_t = V(s_t)$ and new rewards $r'_t = r_t + (v_{t+1} - v_t)$. The idea is that if we get a good state, where many actions yields a high reward, we want to rescale those rewards to ensure we still pick the best and not just any of the good ones.

A variation on advantage estimation is *generalized* advantage estimation using the formula $r'_t = r_t + (v_{t+1} - v_t) + \lambda r_{t+1}$, introduced in [Sch+18].

3 Gröbner bases

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

We fix a field k and consider the polynomial ring $R = k[x_1, \dots, x_n]$. For at set of polynomials $F = \{f_1, \dots, f_l\}$ we consider the ideal $I = \langle f_1, \dots, f_l \rangle$ 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.

3.1 · 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 $F = \{f_1, \dots, f_l\}$
    Output: Remainder $r$ s.t. $f - r \in \langle F \rangle$ and $in(f_i) \nmid in(r)$ for all $f_i \in F$
1   $r \leftarrow f$
2   while $\exists i. in(f_i) \mid in(r)$ do
3       $r \leftarrow r - \frac{in(r)}{in(f_i)} f_i$
```

This algorithm terminates since the inistal term of r is strictly descreasing with respect to $>$. However, this algorithm has some problems. In particular, we do not always get that $reduce(f, F) = 0$ when f lies in the ideal generated by F, as we would expect. This is due to the choice of which polynomial to reduce with in line 3.

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

Now, we're ready to present Gröbner bases and the theorem that makes them so important:

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

Note that this implies $\langle F \rangle = I$.

3.3 · Theorem. *Let $G = \{f_1, \dots, f_l\}$ 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. ◻

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.

3.4 · 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.

3.5 · Theorem. *Let $F = \{f_1, \dots, f_l\}$ 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 $F = \{f_1, \dots, f_l\}$
Output: A Gröbner basis $G$ of the ideal $I = \langle F \rangle$
1 $G \leftarrow F$
2 $P \leftarrow \{(f_i, f_j) \mid 1 \leq i < j \leq l\}$
3 while $|P| > 0$ do
4     $(g,h) \leftarrow$ select($P$)
5     $P \leftarrow P \setminus \{(g,h)\}$
6     $r \leftarrow$ reduce($S(g,h), G$)
7     if $r \neq 0$ then
8         $P \leftarrow$ update($P, G, r$)
9         $G \leftarrow G \cup \{r\}$
```

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 3.5 in mind, it is clear that G will be a Gröbner basis when the algorithm terminates.

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 3.5. We'll discuss a better version in section 3.1.

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 polynomial additions by almost 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.
Queue	Treat P as a queue and select the pair that was added first.
Stack	Treat P as a stack and select the pair that was most recently added.
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.
Sugar	Pick the paie (f,g) with the smallest *sugar degree* which is the degree $S(f,g)$ woudl have had, if the polynomials had been homogenized.
TrueDegree	Introduced in [PSH20], pick the pair (f,g) whose S-polynomial has the lowest degree.

3.1 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 S-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 $i(r) \mid \gamma$ and $\gamma \neq lcm(i(f),i(r))$ and $gamma \neq lcm(i(g),i(r))$. Indeed, in that case we would have $lcm(i(f),i(r)) \mid \gamma$ and $lcm(i(g),i(r)) \mid \gamma$. Thus writing $\gamma = q_f lcm(i(f),i(r)) = q_g lcm(i(g),i(r))$ we get that $S(f,g) = q_f S(f,r) - q_g S(g,r)$. This means $S(f,g)$ is redundant in a basis containing $S(f,r)$ and $S(g,r)$.

Next, we should add all pairs (f,r) where $f \in G$. However, also here we can skip some immediately. First of all, for any two pairs (f,r) and (g,r) where $lcm(i(f),i(r)) \mid lcm(i(g),i(r))$ we can obviously remove (g,r).

After this, consider the equivalence realation $(f,r) \sim (g,r)$ if $lcm(i(f),i(r)) = lcm(i(g),i(r))$. We only need one representative from each equivalence class generated by this relation.

Finally, if $i(f)i(r) = lcm(i(f), i(r))$ then this pair can also be removed, as $S(f, r)$ would reduce to 0.

4 Experimental setup

Following the work of Dylan Peifer [PSH20] we use a neural network to learn pair selection in Buchbergers algorithm. After the agent selects a pair, we reduce the corresponding S-polynomial and give the agent a reward, which is -1 times the number of polynomial additions used in the reduction step. This is following the setup of [PSH20].

Also in line with [PSH20] we focus on binomial ideals (ideals generated by polynomials with exactly 2 non-constant terms) in order to simplify the task of representing a polynomial to a network. It is guaranteed that a Groebner basis for an ideal generated by binomials will consist of binomials. This enables us to encode a pair of binomials in known space.

Binomial ideals are a special case of *lattice ideals* wich are generated by binomials in which each variable appears in only one of the terms. These have applications in optimization and integer programming, for example https://arxiv.org/pdf/math/0508287.pdf. Thus, binomial ideals are still useful, although it would interesting to explore how to represent polynomials with an abitrary number of terms to a neural network.

4.1 Markov Decision Process

The problem of pair selection is modeled as a Markov Decision Process. During a run of Buchbergers Algorithm the *select* subroutine is implemented by the agent. At timestep t the agent sees the state P_t which is the current set of polynomial pairs. The agent must then choose a pair $a_t \in P_t$ which is fed back into Buchbergers algorithm as the result of *select*. The environment then updates by removing p from P, reducing the correpsonding and S-polynomial and updating G and P.

The reward r_t given for the action a_t is -1 times the number of polynomial additions performed when reducing a_t. This metric is chosen since it is the most expensive computation and it serves as a proxy for total computation.

This loops until a complete Gröbner basis is constructed at timestep T, yielding the trajectory $\tau = (P_0, a_0, r_0, \ldots, P_T, a_T, r_T)$. The goal of the agent is to maximize the exprected return $\mathbb{E}[\sum_{t=1}^{T} r_t]$ which is minimizing the number of polynomial additions.

n-d-s	Random	First	Queue	Stack	Degree	Normal	TrueDegree
3-10-4	349.9	286.5	295.2	1001.1	216.1	239.2	191.7
3-10-10	429.1	366.7	413.1	680.1	355.6	357.9	300.0

Table 1: Performance of standard selection strategies on 500 Gröbner bases

4.2 Generating random ideals

To produce training data we generate a set of random binomials for each episode of the training. We parameterize the space of possible binomial ideals by three numbers: the number of variables in the polynomial ring n, the maximum total degree of any binomial d and the number of generating polynomials s.

We generate exponent vectors as follows: sample a random integer $1 \leq r \leq d$ uniformly and then sample a vectors from $\{\mathbb{N}^n \mid \sum_i v_i \leq r\}$ uniformly. This scheme is chosen as it distributes total degree uniformly. If we had sampled from $\{\mathbb{N}^n \mid \sum_i v_i \leq d\}$ we would get more binomials of high total degree.

For the coefficients we take the coeeficient field to be $\mathbb{Z}/32003\mathbb{Z}$ and take coefficients uniformly from $\mathbb{Z}/32002\mathbb{Z}^*$.

4.3 Evaulation of existing selection strategies

Let's take a look at how existing strategies perform, see table 1. The baseline ought to be picking a random pair. The easiest strategy to implement is an agent simply saying 1 every time since 1 is always an allowed action. This would be the queue strategy. This actually performs surprisingly well despite being very simple. It makes intuitive sense: The first added pairs are the pairs of the generating set. Especially when the generating set is small is it very likely that none of these are redundant, so it makes sense to consider dem. Compare this the "opposite" strategy: always picking the most recently added pair. This performs extremely badly with few generators, but less badly with more generators since then some generators ought to be redundant.

Notice, that queue and stack can't be learned by the neural network. Since the network scores each pair independently it can't know which was added first. It should be noted that we use the queue strategy to break ties in the other strategies.

Degree and Normal strategies both outperform the queue strategy and degree outperforms normal, but with more with few generators. However, TrueDegree, which was introduced in [PSH20], outperforms every other strategy. TrueDegree was found as an approximation of the strategy learned by their neural network.

4.4 Important implementation details

	Random			Degree		
s (ER)	Additions	Selections	(ms)	Additions	Selections	(ms)
4 (ER)	344.0	86.7	(3.37)	219.4	58.4	(5.72)
10 (ER)	431.3	109.1	(4.28)	327.8	77.6	(11.54)
4 (R)	1525.8	287.0	(11.31)	731.6	147.8	(81.53)
10 (R)	1976.8	367.9	(13.35)	1354.3	224.7	(117.99)
4 (E)	226.9	83.6	(2.33)	141.1	54.6	(5.39)
10 (E)	258.7	106.1	(3.01)	194.2	79.1	(10.68)
4 ( )	664.5	243.5	(4.91)	392.9	132.6	(59.13)
10 ( )	987.5	390.9	(7.81)	594.3	213.9	(118.8)

Table 2: Performance of two selection strategies on different settings. E=Full Gebauer&Möller pair eleminiation, R=Reduce every term instead of the leading. n=3, d=10

We can compare some common optimizations.

4.4.1 Pair elimination

Pair elemination is clearly important, the lowest numbers are achieved using pair elemination.

4.4.2 Fully reducing polynomials

The division algorithm presented in Algorithm 1 only reduces the leading term of the polynomial. However, it makes sense to reduce every term, such that no leading term in the set of polynomials divides any term of the reduced polynomial. This gives smaller polynomials (lower total degree) which should increase the chance that it divides a later polynomial. With this reasoning it has generally been accepted as a good idea, but it might not be so clear-cut.

We see that while fully reducing the polynomials doesn't change the number of required selections much, it adds a lot of polynomial additions, which are expensive. It seems (at least in our implementation) that number of additions correlates much better with execution time than number of selections.

4.5 Network architecture

A challenge when applying neural networks to this problem is the uneven size of the input. The agent needs to select a pair from an abitrarily large list of pairs. To solve this, we try employing Q-learning and policy gradient networks. Instead of having the network select a pair, we try to score each pair and select the one with the highest score. This removes the need to encode the state.

For the Q-network we try to learn the function

$$Q_\pi(\langle f,g\rangle) = \mathbb{E}[G(\tau_p) \mid \langle f,g\rangle \in s_p, \pi].$$

i.e. the expected number of polynomial reductions after choosing this pair.

For the policy gradient network, there is no particular function we try to learn. However, the two networks have one thing in common: since they only "see" one pair of polynomials at a time, they are very limited. They cannot exploit knowledge of what the Gröbner basis curretly looks like and they cannot explicitly compare polynomials.

This means the networks don't actually learn pair selection. Instead they learn an embedding of polynomial pairs into $\mathbb{R}$.

The encoding of a binomial pair follows the encoding given by Dylan Peifer, sending a polynomial pair $\langle c_1 x^{a_1} + c_2 x^{a_2}, c_1' x^{a_1'} + c_2' c^{a_2'}\rangle$ to the vector $[a_1 \mid a_2 \mid a_1' \mid a_2']$.

Example. Consider the situation with $n = 3$ variables and an ideal generated by $F = \{4xyz + x^2z, 19z + x^2y^5z^3, x + y\}$. This gives the pairs $P = \{(1,2),(1,3),(2,3)\}$. Considering each pair as a row-vector, we can represent this as the following $|P| \times 4n$ matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 2 & 0 & 1 & | & 0 & 0 & 1 & 2 & 5 & 3 \\ 1 & 2 & 3 & 2 & 0 & 1 & | & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 2 & 5 & 3 & | & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$ □

The architecture of the neural network is given below:

$$\begin{tikzcd} \boxed{1 \times 4n} \arrow[r, "relu \circ (A_{64,4n} + b_{64})"] & \boxed{1 \times 64} \arrow[r, "(A_{1,64} + b)"] & \boxed{1 \times 1} \end{tikzcd}$$

However, we could not get that to work. It seemed to learn something but it learned very slowly and I don't know enough about deep learning to figure out why.

4.6 Technological choices

We chose to write the code for this experiment in Julia using the Flux framework for deep learning and building the ReinforcementLearning.jl project as a skeleton for reinforcement learning.

Writing the program in julia had two advantages: we could keep all the code in a single language and we improved running time significantly over the implementation by [PSH20].

Taking a look at `https://github.com/dylanpeifer/deepgroebner` we see that $\sim 25\%$ of the project is written in C++. This is an implementation of the Buchberger algorithm and code supporting that and it's nescesary to keep the code fast. However, a TensorFlow model is built in Python and they must use Python code to bridge between TensorFlow and the custom Buchberger code, even though both are written in C++. This incurs a cost, both on the programmer who needs to learn two languages, and on the runtime. Converting between Python and C++ is not free and since this exchange needs to happen at each selection, taking approximately 100 selections to produce a Gröbner basis, this cross-over happens about 8.000.000 times during a training run.

By keeping everything in Julia, which is very fast language, we only need to learn a single language, and we prevent the overhead. In practice, this means that we can train the same models as they did in similar timeframes, but on one core of an Intel i5 CPU instead of an c5n.xlarge AWS instance.

References

[PSH20] Dylan Peifer, Michael Stillman and Daniel Halpern-Leistner. *Learning selection strategies in Buchberger's algorithm*. 2020. arXiv: `2005.01917 [cs.LG]`.

[Sch+18] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018. arXiv: `1506.02438 [cs.LG]`.

[WD92] C. Watkins and P. Dyan. In: (1992). URL: `https://www.gatsby.ucl.ac.uk/~dayan/papers/cjch.pdf`.