Tutorial: Brief intro to Reinforcement learning

Dries Sels

These notes are for a tutorial at the *Machine learning for quantum many-body physics* program at KITP. In these notes I will discuss the basics of reinforcement learning and optimal control.

What's RL?

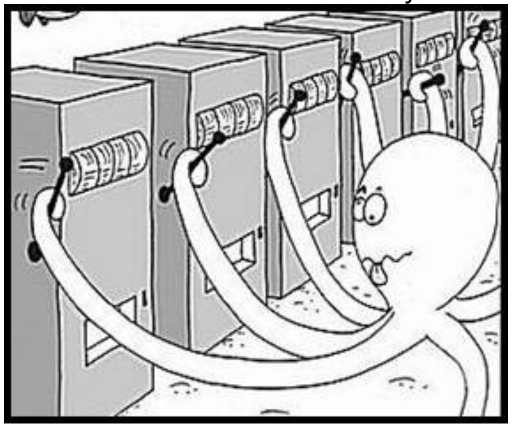
It's probably safe to say that intelligence doesn't come from backpropagation of errors. The idea that we learn from interactions with our environment is quite natural. Whether it's learning to walk or drive a car, we are aware of how our environment reacts to what we do and we constantly try to influence what happens.

Reinforcement learning can be summarized as *the problem of learning what to do, so as the maximize a certain reward.* In many cases our actions do not only affect the immediate reward but they can influence all subsequent rewards. Moreover, in almost all cases those rewards are a priori unknown. We thus immediately face a problem. If we wish to maximize our total reward we must *explore* our environment by probing it in different ways. Many actions might however result in in a low reward and at some point, we should try to *exploit* the knowledge we have. Reinforcement learning is a way of dealing with this exploration-exploitation dilemma.

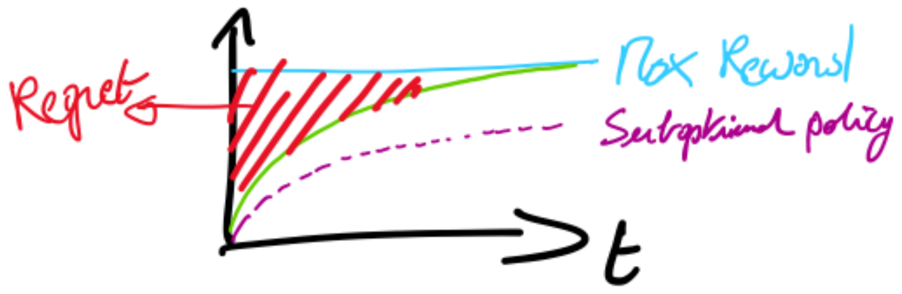
RL therefore has applications in cases in which the exploration comes at a cost; in contrast to supervised or unsupervised learning with the data is usually given. Say, I am running a clinical trial to figure out whether a certain drug works better than another. If I gather enough data there is a clear answer to my problem and I only need to do statistics at the end. However, people might die during the trial. Curing or helping the people in the clinical trial is the fundamental goal and once a certain treatment shows good results you might want to exploit it.

K-armed Bandits

The k-armed bandit is one of the simplest problems that captures the dilemma. A one-armed bandit is just a slot machine. It has this nickname because you can never win.



The k-armed bandit is a generalization of this to k-arms. The problem is the following, each of the arms has a different return. The return is unknown to me and could be stochastic, so I have to explore different arms. I'm losing money by playing suboptimal arms so I need to find an algorithm that efficiently figures out the arm with the highest expected reward as fast as possible.



Let's look at the expected reward for doing a certain action. If we'd know this the optimal action would simply maximize that:

$$Q_t(a) = \mathbb{E}[R_t \mid A_t = a]$$

$$\begin{tikzcd} & \boxed{\text{Agent}} \arrow[d, "\text{Action } a"] \\ & \boxed{\text{Env}} \arrow[u, "\text{Reward } R_t"] \end{tikzcd}$$

We have to estimate this from samples:

$$Q_{n+1} = \frac{1}{n} \sum_{i=1}^{n} R_i = \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right)$$

$$\Rightarrow \quad Q_{n+1} = \frac{1}{n}\left(R_n + (n-1)\, Q_n\right)$$

$$\Rightarrow \quad Q_{n+1} = Q_n + \underset{\text{weight of a sample}}{\frac{1}{n}} \underbrace{\left(R_n - Q_n\right)}_{\text{"surprise"}}$$

Epsilon-greedy

The simplest possible algorithm just maximizes the current expected reward. That's greedy and will in general lead to suboptimal solution but if we simply add some small exploration to this, it is actually not that bad. In an epsilon-greedy algorithm we do what we believe to optimal almost all of the time, except in epsilon cases we take a random action.

```
Initialize, for $a = 1$ to $k$:
    $Q(a) \leftarrow 0$
    $N(a) \leftarrow 0$

Repeat forever:
    $A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
    $R \leftarrow bandit(A)$
    $N(A) \leftarrow N(A) + 1$
    $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$
```

UCB

Another algorithm that works very well is based on the upper confidence bound. While we can keep track of an estimate of the expected reward, our sample only consists of a finite number of data point. We thus have a certain confidence about our estimate. UCB is an algorithm that expresses extreme optimism regarding this confidence. It doesn't take the action with the highest expect reward but with the highest upper confidence bound.

$$A_t = \arg\max_a \left(Q_t + c\sqrt{\frac{\log t}{N(a_t)}} \right)$$

This is completely deterministic but the algorithm is forced to take actions that it hasn't used because those are highly uncertain.

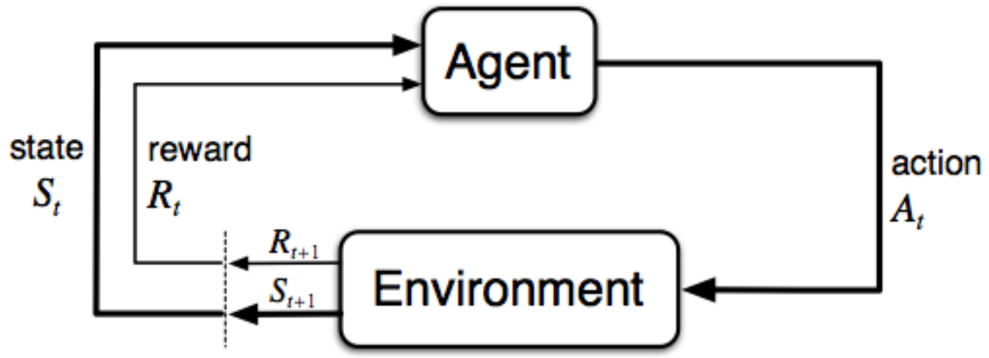
Soft max or what physicists would do

Epsilon-greedy does exploration but it isn't really smart about it and does it in a random way. UCB is smart about what it tries but it is completely deterministic. One way to introduce randomness but still bias the exploration towards more likely actions is to sample a policy from a distribution:

$$\pi(a) = \frac{e^{Q(a)}}{Z}$$

Bellman-equations

Many systems are of course more complicated than our k-armed bandit example. The most notable difference is that the state of the system might change upon applying a certain action. Consequently, the reward you get doesn't just depend on the action you take, it also depends on the state in which the system is. Pictorially we can represent this as:



Mathematically we can now formulate our RL problem as a Markov decision process. Our goal is still to maximize the total expected reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_k \gamma^k R_{t+1+k}$$

Since the reward is now different for different states we will have to specify a *policy,* that is we have to specify which action to take given we the state we are in. This policy can be deterministic or stochastic:

$$\pi(a|s)$$

Using the same notation as before, we can now define the expected total reward given a current action-value pair:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t | A_t = a, S_t = s]$$

If we take the expectation value of this over a given policy we obtain the value of a given state under a given policy:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

The fact that states and actions have a value is one of the major differences with genetic programs. The latter only care of the end result and no value is assigned to intermediate states in the process. The best possible possibility is simply the one that maximizes the value function:

$$v_*(s) = \max_\pi v_\pi(s) \quad \Rightarrow \quad v_*(s) = \max_a Q_{\pi_*}(s,a)$$

Let's derive one of the main results in optimal control theory, which is the Hamilton-Jacobi-Bellman equation for the value function. Recall that in the k-armed bandit we tried to write the value of a certain action in terms of the current reward, such that we could update our expectation online. Let's try something similar for the value function:

$$v_\pi = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}\left[\sum_{q=0}^{\infty} \gamma^q R_{t+1+q} \mid S_t = s\right]$$

$$\Rightarrow \quad v_\pi = \mathbb{E}\left[R_{t+1} + \gamma \sum_{q=0}^{\infty} \gamma^q R_{t+2+q} \mid S_t = s\right]$$

$$\Rightarrow \boxed{v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|a,s)\left[r + \gamma v_\pi(s')\right]}$$

This can also be done for the Q-function:

$$Q_\pi(s,a) = \sum_{s',r} p(s',r|a,s)\left[r + \gamma v_\pi(s')\right]$$

Finding this optimal policy now requires solving Bellman's equation. If the dynamics of the environment is know this is in principle possible but it will require to solve a set of N non-linear equation, where N is the dimension of the state space. The large dimensionality of the state space forms one of the major problems. We are thus looking for an efficient way of dealing with this, that's where other techniques from machine learning come in.

We can split our task in two problems, (i) we need to get an estimate of the value of a given policy (ii) given we know the value of some policy we need to improve that policy.

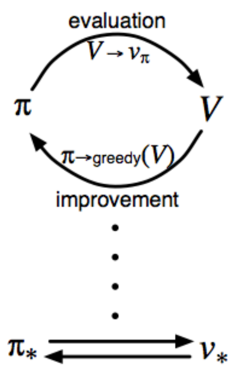
In principle (i) can be done by iterating Bellman's equation. Once the value stops changing you can stop the iteration and you have found the value of a given state under that policy. (ii) Is also quite simple, since the optimal action under a given policy is

$$\arg\max_a Q_\pi(s,a)$$

We can simply define the new optimal, greedy, policy as:

$$\pi'(s) = \arg\max_a Q_\pi(s,a)$$

By repeating this procedure, we can obtain the optimal policy. That's quite costly and one could wonder if one needs to get an exact estimate of the value before one can update the policy. As expected this is not the case and one can even get guaranteed convergence if you update the policy after a update of the value function, the latter is called *value function iteration.*



There are many ways to improve this produce by obtaining better (typically Monte Carlo) estimates of the value function at a fixed policy.

Temporal Difference learning

One serious advance in the 90's in RL was the advent of temporal difference learning algorithm. While Deepmind's alpha go is more involved it is similar in spirit. As always there are two problems, evaluating the value of a given policy and updating the policy.
Evaluating a given policy can be done analogous to the k-armed bandit by updating the value function:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right]$$

The actual total return isn't known, but we can estimate it from the current return and the value of the next state:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right]$$

This is the simplest temporal difference estimate one can get and is called TD(0). There are some advantages in estimating the value function this way over running MCMC to estimate the value function. One can do the same on the Q-function, which allows us to do on-policy control. That is we keep estimating the Q-function for a given policy while at the same time moving the policy towards a more greedy policy with respect to the Q-function. Recall that the greedy actions are optimal when performed based on the optimal Q-function. This method is called Sara:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \right]$$

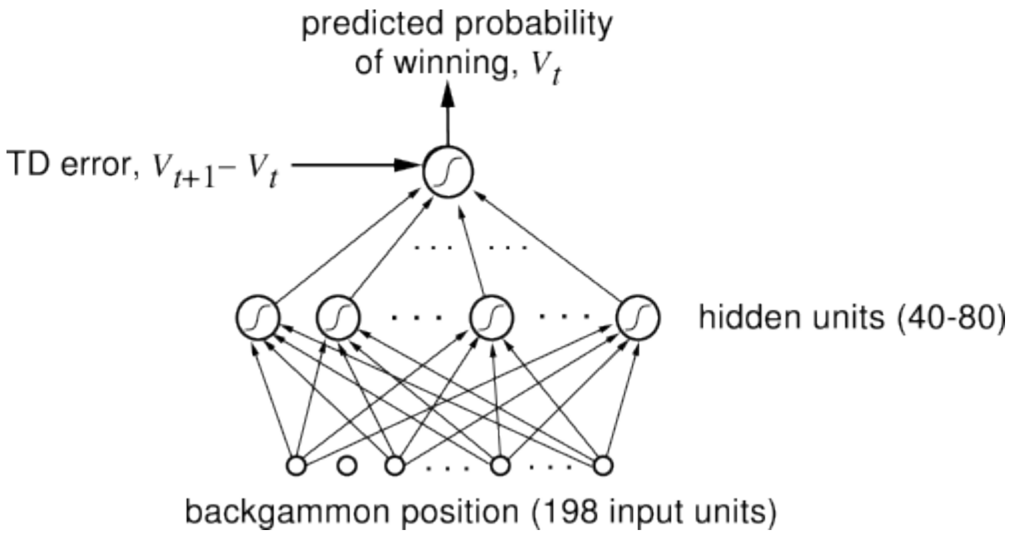
The off-policy version of this is called *Q-learning*. The difference is that we use the optimal action to estimate the value of the next step:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

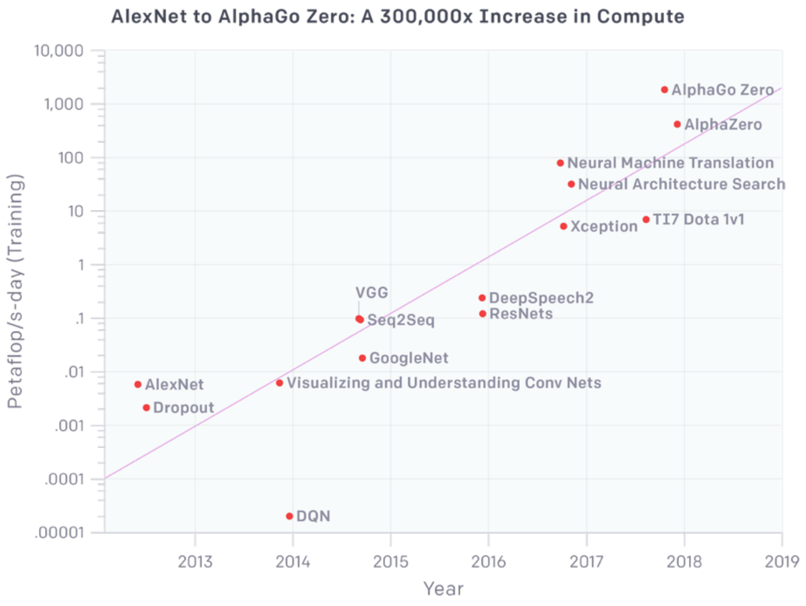
We directly learn the optimal Q-function in this way and not the Q function for the given policy. The policy still has an effect though. It will determine which states and actions we will visit. As long as we use a policy that guarantees all the state-action pairs are being updated, one can proof that the algorithm converges in the asymptotic limit. However, the efficiently clearly depends on the policy used for exploration. In practice epsilon-greedy is often used. Importance sampling from the current estimate of the Q-function is also possible.

If there is TD(0), there is a TD(n). As expected the latter simply backs up more than one step, i.e. it backs up n steps. For discounted tasks it can reduce the error on the estimate of the value function substantially.

Backgammon ('95)


predicted probability of winning, V_t
TD error, $V_{t+1} - V_t$
hidden units (40-80)
backgammon position (198 input units)

Alpha go ('18)



Gradient methods

Finally, gradients. In typical examples we cannot store the Q-function in a simple table or as a simple function. We will need to approximate the object and this can for example do by simply fitting our favorite model to the value function. Since we don't have all the data from the beginning it's natural to integrate fitting and learning.

$$C = \sum_{s} \left[v_\pi(s) - \hat{v}(s,\theta) \right]^2 \Rightarrow \theta_{t+1} = \theta_t + \eta \left[v_\pi - \hat{v}(\theta_t) \right] \nabla_\theta \hat{v}(\theta_t)$$

We could for example use TD(0) to get an estimate of the value function essentially resulting in a gradient descend algorithm for the parameters of our fitting function. If we'd do it like this we get an on-policy algorithm but we could also directly fit our Q-function. By doing this off-policy we directly learn the parameters for the optimal Q-function.

References

Reinforcement learning: an introduction, Sutton and Barto, MIT press
Atari breakout by Deepmind
Open AI about computing power