Artificial Intelligence

Augustin Cosse.

Fall 2021

November 24, 2021

MDPs and RL: short recap

- Remember that a Markov Decision Process (MDP) is a sequential decision problem (where the utility depends on a sequence of decisions) in a fully observable, stochastic environment with a Markovian transition model (that is the probability of reaching state s' from s depends only on s and not the history of earlier states)
- Recall that we (reasonably) assumed that the preferences of the agent between sequences was stationnary
- This assumption reduces the set of possible definitions of the utility to definitions based either on additive or discounted rewards.

$$U^{\pi} = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^{t} R(s_t)\right]$$

MDPs and RL: short recap

- We also saw that we could define a Maximum Expected Utility (MEU) policy by selecting our actions as

$$\begin{aligned}\pi^*(s) &= \underset{\pi}{\operatorname{argmax}} U^\pi(s) \\ &= \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s'|s,a)U(s')\end{aligned}$$

- And that if we assumed our agent selected the optimal actions in the future, the utility obeyed the Bellman equation

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

MDPs and RL: short recap

- To find the utility of all states, given a transition model $P(s'|s,a)$ and the rewards associated to each states, we saw that we could solve the MDP (i.e. get the utility at all states) through value iteration, updating the utilities through the Bellman update

$$U_{i+1} \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

which was guanranteed to converge to a fixed point, solution to the Bellman equation.

MDPs and RL: short recap

- We also saw that it might be a good idea, after a few steps, to use our current knowledge to define a policy and update our estimate of the utility based on this policy. This gave the policy iteration based on alternating between two steps
    - Policy evaluation which updates the utility U^i based on a policy π_i
    - Policy improvement which calculates a new MEU policy π_{i+1} based on the utility U^i

We saw that in the framework of policy iteration, we could either solve the Bellman system directly or (for large state spaces) iteratively through the update

$$U^{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi_i(s)) U_i(s')$$

MDPs and RL: short recap

- When discussing MDPs, we assumed that the environment was fully known (the transition model $P(s'|s,a)$ as well as the rewards for each state were given).
- In practice, this is rarely the case and we saw that learning in the general framework in which the environment is unknown (i.e. where the transition model has to be learned alongside the utilities and policy) is the objective of Reinforcement Learning.
- In Reinforcement Learning, we saw that one makes the distinction between two main types of agents:
  - Passive reinforcement learning (where the agent's policy is fixed and the task is to learn the utilities of the states)
  - Active reinforcement learning (where the agent must also learn what to do)

MDPs and RL: short recap

- In passive reinforcement learning, we have discussed three approaches at learning the utility :
  - Direct Utility Estimation (where each completion of an episode provide a set of samples for the "expected reward to go" expression). We saw that this approach however misses the connections between states
  - Adaptive Dynamic Programming where the agent learns the transition model $P(s'|s,a)$ by keeping track of how each action outcome
  - Finally, Temporal Difference Learning relies on correcting the current utility estimate through the temporal difference update

$$U^\pi(s) \leftarrow U^\pi(s) + \eta\left(R(s) + \gamma U^\pi(s') - U^\pi(s)\right)$$

Reinforcement learning (continued)

- We saw that an efficient agent should always achieve a tradeoff between exploitation and exploration.
- In particular, we saw that pure exploitation risks getting stuck in a rut and that pure exploration to improve one's knowledge is of no use if one never puts that knowledge into practice. But can we be a little more precise? Is there an optimal exploration policy?
- This question of an optimal exploration policy has been studied within the framework of the Bandit problem.
- Although Bandit problems are extremely difficult to solve exactly, it is nonetheless possible to come up with a reasonable scheme that will ultimately lead to an optimal behavior by the agent.

Reinforcement learning (continued)

- Technically, any such scheme needs to be greedy in the limit of infinite exploration or **GLIE**
- A **GLIE** scheme must also try each action in each state an unbouded number of times to avoid having an infinite probability that an action is missed because of an usually bad series of outcomes.
- An ADP (Active Dynamic Programming) agent using such a scheme will eventually learn the true environment model.
- A GLIE scheme must also eventually become greedy, so that the agent's actions become optimal with respect to the learned (and hence the true) model

Reinforcement learning (continued)

- There are several **GLIE** schemes. One of them is to have the agent choose a random action a fraction $1/t$ of the time and to follow the greedy policy otherwise.
- While this does eventually converge to an optimal policy, it can be extremely slow.
- A more sensible approach would give some weight to actions that the agent has not tried very often, while tending to avoid actions that are believed to be of low utility.
- This can be done by altering the Bellman update so that it assigns higher utility estimates to relatively unexplored state-action pairs.

Reinforcement learning (continued)

- Essentially, this altering of the Bellman update amounts to an optimistic prior over the possible environments and causes the agent to behave initially as if there were wonderful rewards scattered all around the place
- Let us use $U^+(s)$ to denote the optimistic estimate (expected reward to go) of the utility of the state s and let $N(s,a)$ be the number of times action a has been tried in state s
- Suppose we are using value iteration in an ADP learning agent; then the update equation can be rewritten as

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s'|s,a)U^+(s'), N(s,a)\right)$$

where $f(u,n)$ is called the **exploration function**.

Reinforcement learning (continued)

- The **exploration function** determines how greed (i.e. preference for higher values of U) is traded off against curiosity (preference for actions that have not been tried often and have low n).

- The function $f(u, n)$ should be increasing in u and decreasing in n. Among the many possible functions that fit these conditions, one particularly simple definition is

$$f(u, n) = \begin{cases} R^{+} & \text{if } n < N_e \\ u & u \text{ otherwise} \end{cases}$$

  where R^{+} is an optimistic estimate of the best possible reward obtained in any state and N_e is a fixed parameter.

- Such a function will have the effect of making the agent try each action-state pair at least N_e times

Reinforcement learning (continued)

- Consider the modified Bellman update

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s'|s,a)U^+(s'), N(s,a)\right)$$

- The fact that U^+ rather than U appears on the RHS of this update is very important

- As exploration proceeds, the states and actions near the start state might well be tried a large number of times

- If we used U, the more pessimistic utility estimate, then the agent would soon become desinclined to explore further afield.

- The use of U^+ means that actions that lead toward unexplored regions are weighted more highly than just actions that are themselves unfamiliar

Reinforcement learning (continued)

- Now that we have seen how we could design an active **ADP** agent, let us consider how to design an active temporal difference (TD) learning agent
- The first difference with the passive case is that the agent is not equipped with a fixed policy anymore, so, if it learns a utility function U, it will need to learn a model in order to be able to choose an action based on U.
- The model acquisition problem for the TD agent is identical to that for the ADP agent: What of the TD update rule itself?
- In active TD learning, there is a alternative TD method, called **Q-learning** which learns an action-utility representation instead of learning utilities

Reinforcement learning (continued)

- In **Q-learning**, we will use the notation $Q[s, a]$ to denote the value of doing action a in state s. Q-values are directly related to utility values as follows

$$U(s) = \max_a Q[s, a] \tag{1}$$

- Q functions may seem like just another way of storing utility information, but they have a very important property: a TD agent that learns a Q-function does not need a model of the form $P(s'|s, a)$, either for learning or for action selection
- For this reason, Q-learning is called a **model-free** method.
- As with utilities, we can write a constraint equation that must hold at equilibrium when the Q values are correct

$$Q[s, a] = R[s] + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q[s', a']$$

Reinforcement learning (continued)

- As in the ADP learning agent, we can use the update equation on $Q[s', a']$ directly as an update equation for an iterative process that calculates exact Q-values given an estimated model.

- This does however, require that a model also be learned as the equation uses $P(s'|s, a)$

- If you remember, the temporal difference approach however required no model for state transitions (all it needed were the Q values)

- The update equation for TD Q-learning is

$$Q[s, a] \leftarrow Q[s, a] + \alpha \left(R[s] + \gamma \max_{a'} Q[s', a'] - Q[s, a] \right)$$

  which is calculated whenever action a is executed in state s leading to state s'

```
Function Q-learning-Agent(percept):
    input      : percept, a percept indicating the current state $s'$
                 and reward signal $r'$
    persistent : $Q$, a table of action values indexed by state and action,
                 initially zero
                 $N_{sa}$, a table of frequencies for state action pairs,
                 initially zero
                 $s, a, r$, the previous state, action, and reward,
                 initially null
    if s is not null then
        increment $N_{sa}[s, a]$
        $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])\,(R + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
    end
    $s, a, r \leftarrow s', \underset{a'}{\operatorname{argmax}} f\,(Q[S', A'], N_{sa}[s', a'])\,, r'$
    return $a$
```

Reinforcement learning (continued)

- Q-learning has a close relative called **SARSA** (for State-Action-Reward-State-Action). The update rule for SARSA is very similar to traditional Q-learning

  $$Q[s,a] \leftarrow Q[s,a] + \alpha \left(R[s] + \gamma Q[s',a'] - Q[s,a] \right)$$

  where a' is the action actually taken in state s'.

- The rule is applied at the end of each s, a, r, s'

- The difference with Q-learning is quite subtle : Whereas Q-learning backs up the best Q-value from the state reached in the observed transition, **SARSA** waits until an action is actually taken and backs up the Q-value for that action.

Reinforcement learning (continued)

- For a greedy agent that always takes the action with the best Q value, the two algorithms are identical.
- When exploration is happening, the two algorithms differ significantly however
- Because Q-learning uses the best Q-value, it pays not attention to the actual policy being followed : it is an **off policy** learning algorithm whereas SARSA is an **on policy** algorithm.
- Q-learning is more flexible than SARSA in the sense that a Q learning agent can learn how to behave well even when guided by a random or adversarial exploration policy

Reinforcement learning (continued)

- On the other hand, SARSA is more realistic : if the overall policy is even partly controled by other agents, it is better to learn a Q function for what it will actually happen than for what the agent would like to happen

Generalization in RL

- So far, we have assumed that the utility functions and Q functions learned by the agents are represented in tabular forms with one output value for each input tuple
- Such an approach works reasonably well for small state spaces, but the time to convergence and (for ADP) the time to per iteration increase rapidly as the the space gets larger
- Approximate ADP methods might be able to handle maze-like environments but more realistic worlds are out of the question. as examples, Backgammon and chess are tiny subsets of the real world, yet their state space contains on the order of 10^{20} and 10^{40} states.
- In this case, it would be absurd to suppose that one must visit all the states many times in order to learn how to play the game

Generalization in RL

- One way to handle high dimensional state spaces is to use function approximation (and in particular parametric models as we saw in learning) which simply means using any sort of representation for the Q-function other than a lookup table
- The representation is viewed as approximate because it might not be the case that the true utility function or Q function can be represented in the chosen form.
- As an example, we could choose to use an evaluation function for chess and represent this function as a weighted linear function of a set of features (or basis functions) $f_1, f_2, \ldots$

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

Generalization in RL

- A reinforcement learning algorithm can then learn values for the parameters $\theta = \theta_1, \ldots, \theta_n$ such that the evaluation function $\hat{U}_\theta$ approximates the true utility function

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

- Instead of say 10^{40} entries in a table, this function approximator is characterized by, say, $n = 20$ parameters which represents an enormous compression

- Function approximation makes it practical to represent utility functions for very large state spaces, but that is not its principal benefit

- The compression achieved by a function approximator allows the learningagent to generalize from states it has visited to states it has not visited

Generalization in RL

- To give an example of the generalization property of parametric representations, by examining only one in every 10^{12} of the possible backgammon states, it is possible to learn a utility function that allows a program to play as well as any human

- On the flip side, there is of course the problem that there could fail to be any function in the chosen hypothesis space that approximates the true utility function sufficiently well

- As in all inductive learning, there is a tradeoff between the size of the hypothesis space and the time it takes to learn the function

- A large hypothesis space increases the likelihood that a good approximation can be found, but also means that convergence is likely to be delayed.

Generalization in RL

- Let us get back to direct utility estimation where each completed simulation provides a sample for the utility.
- If we use function approximation to represent the utility, this becomes a supervised learning task
- As an example, suppose we represent the utilities for a simple 4×3 world using a simple linear function
- The features of the squares are just their x and y coordinates so we can write

$$\hat{U}(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

  Given a collection of trials, we can obtain a set of sample values for $\hat{U}_\theta(x, y)$ and we can then find the best fit

Generalization in RL

- For reinforcement learning, it makes more sense to use an online learning algorithm that updates the parameters after each trial.
- As with neural network learning, we can write an error function and compute its gradient with respect to the parameters.
- If $u_j(s)$ is the observed total reward from state s onward in the j^{th} trial, the the error is defined as (half) the squared difference of the predicted total and the actual total:

$$E_j(s) = \frac{1}{2}\left(\hat{U}_\theta(s) - u_j(s)\right)^2$$

Generalization in RL

- The rate of change of the error with respect to each parameter θ_i is $\partial E_j / \partial \theta_i$ so to move the parameters in the direction of a decreasing error, we want

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha \left(u_j(s) - \hat{U}_\theta(s) \right) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

- In the case of a linear function approximator, we then get the equations

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha(u_j(s) - \hat{U}_\theta(s)) \\
\theta_1 &\leftarrow \theta_1 + \alpha(u_j(s) - \hat{U}_\theta(s))x \\
\theta_2 &\leftarrow \theta_2 + \alpha(u_j(s) - \hat{U}_\theta(s))y
\end{aligned}$$

Generalization in RL

- We can apply the ideas behind function estimation to temporal difference learning. All we need to do is adjust the parameters to try to reduce the temporal difference between successive states.
- The new versions of the TD and Q-learning equations (for utilities and Q-values) are then given by

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s) \right] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a) \right] \frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i}$$

Generalization in RL

- For passive TD learning, the update rule can be shown to converge to the closest possible approximation to the true function when the function approximator is linear in the parameters
- With active learning and non linear functions such as neural networks, all bets are off: There are some very simple cases where the parameters can go off to infinity even though there are good solutions in the hypothesis space
- Function approximation can also be very helpful to learn a model of the environment
- Remember that learning a model for an observable environment is a supervised learning problem as the enxt percept gives the outcome state

Policy Search

- The idea behind policy search is to twiddle the policy as long as its performance improve, then stop
- Remember that a policy π is a function that maps states to actions. We are again interested in parametrized representations of π (in particular representations that have far fewer parameters than there are states in the state space)
- For example, we could represent π by a collection of parametrized Q-functions, one for each action, adn take the action with the highest prediction value

$$\pi(s) = \operatorname*{argmax}_a \hat{Q}_\theta(s, a)$$

- Each Q-function could be a linear function of the parameters θ, or it could be a non linear function, such as a neural network

Policy Search

- When the policy is represented by Q functions, policy search results in learning Q-tables. However, note that this is not the same as Q-learning which finds the value of θ such that $\hat{Q}_\theta$ is close to the Q table, Q^* encoding the optimal utlities.

- A policy given by the argmax gives a discontinuous function of the parameters when the actions are discrete (there will be values of θ such that an infinitesimal change in θ will cause the policy to switch from one action to another). This also implies that gradient based search will be more difficult

- For those reasons, we often use a stochastic policy representation $\pi_\theta(s, a)$ which specifies the probability of selecting action a in state s

Policy Search

- A popular representation is the softmax function

  $$\pi_\theta(s,a) = e^{\hat{Q}_\theta(s,a)} / \sum_{a'} e^{\hat{Q}_\theta(s,a')}$$

  The softmax always gives a differentiable function of the θ.

- We now let $\rho(\theta)$ denote the policy value (i.e. expected reward to go when π is executed)

- If we can get a closed form expression for $\rho(\theta)$, then we can get the policy by following the policy gradient $\nabla_\theta \rho(\theta)$ (provided that ρ is differentiable)

- If ρ is not available, we can evaluate π_θ by executing it and and observing the accumulated reward. We can then follow the empirical gradient by hill climbing (this process will converge to a local optimum in policy space)

Policy Search

- For the case of a stochastic policy $\pi_\theta(s, a)$, it is possible to obtain an unbiased estimate of the gradient at θ, $\nabla_\theta \rho(\theta)$ directly from the results of trials executed at θ

- As an example, let us assume that the reward $R(a)$ is obtained directly after doing action a in the start state s_0. In this case the policy value is just the expected value of the reward, and we have

$$\nabla_\theta \rho(\theta) = \nabla_\theta \sum_a \pi_\theta(s_0, a) R(a) = \sum_a \left(\nabla_\theta \pi_\theta(s_0, a)\right) R(a)$$

- Now we perform a simple trick so that this summation can be approximated by samples generated from the probability distribution defined by $\pi_\theta(s_0, a)$.

Policy Search

- Suppose that we have N trials in all and the action taken on the j^{th} trial is a_j. Then

$$\begin{aligned}\nabla_\theta \rho(\theta) &= \sum_a \pi_\theta(s_0, a) \frac{(\nabla_\theta \pi_\theta(s_0, a))\, R(a)}{\pi_\theta(s_0, a)} \\ &\approx \frac{1}{N} \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s_0, a_j))\, R(a_j)}{\pi_\theta(s_0, a_j)}\end{aligned}$$

Policy Search

- Thus, the true gradient of the policy value is approximated by a sum of terms involving the gradient of the action-selection probability in each trial. For the sequential case, this generalizes to

$$\nabla_\theta \rho(\theta) \approx \sum_{j=1}^{N} \frac{(\nabla_\theta \pi_\theta(s, a_j))\, R_j(s)}{\pi_\theta(s, a_j)}$$

  for each state s visited, where a_j is executed in s on the j^{th} trial and $R_j(s)$ is the total reward received from state s onwards in the j^{th} trial

- The resulting algorithm is called REINFORCE

Applications of RL

- The first application of reinforcement learning was also the first significant learning program of any kind: the checkers program written by Arthur Samuel (1959 - 1967)
- Samuel first used a weighted linear function for the evaluation of positions using up to 16 terms at any one time
- He applied a version of the equation

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s) \right] \frac{\partial \hat{U}_\theta}{\partial \theta_i}$$

to update the weights.

Applications of RL

- Gerald Tesauro (IBM research) 's backgammon program TD-GAMMON also illustrated the potential of Reinforcement learning techniques.

- In 1989 Tesauro and Sejnowski tried learning a neural network representation of $Q[s, a]$ directly from examples of moves labeled by a human expert. This approach proved extremely tedious for the expert but resulted in a program called NEUROGAMMON that was strong by computer standards (although not competitive when considering human standard)

- The TD-GAMMON project was an attempt to learn from self-play alone. The only reward signal was given at the end of each game.

- The evaluation function was represented by a fully connected neural network with a single hidden layer containing 40 nodes

Applications of RL

- By repeated applications of the update

$$\theta_i \leftarrow \theta_i + \alpha \left[R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s) \right] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

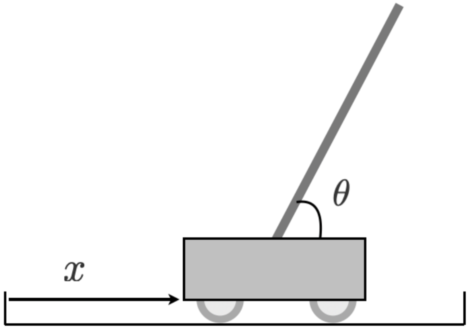
  TD-GAMMON learned to play considerably better than NEUROGAMMON even though the input representation contained just the raw board position with no computed features

- This took around 200,000 training games and two weeks of computer time

- Although that might seem like a lot of games, that represents only a vanishingly small fraction of the state space.

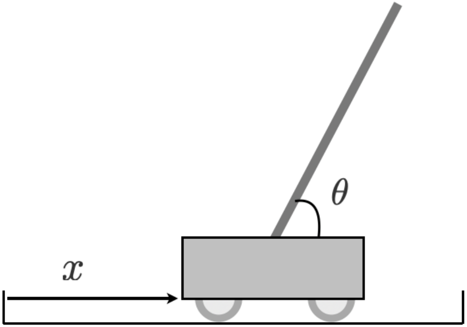
Applications of RL

- The setup for the famous cart-pole balancing problem, also known as the inverted pendulum is shown below
- The problem is to control the position x of the cart so that the pole remains roughly upright ($\theta \approx \pi/2$) while staying within the limits of the cart track.



Applications of RL

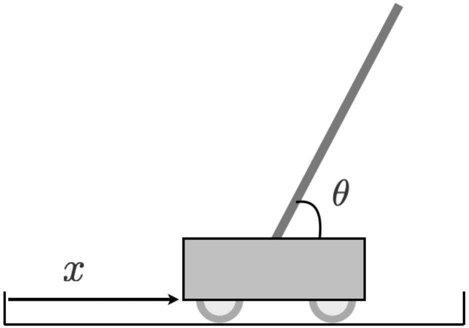
- Several thousand papers in reinforcement learning and control theory have been published on this seemingly simple problem.
- The cart pole differs from the problems described earlier in that the state variables $x, \theta, \dot{x}$ and $\dot{\theta}$ are continuous



Applications of RL

- The actions are usually discrete : move left or move right (the so-called bang bang control regime)

- The earliest workon learning for this problemwas carried out by Michie and Chambers (1968). Their algorithm was able to balance the pole for over an hour after only about 30 trials



Applications of RL

- Still more impressive is the application of Reinforcement learning to helicopter flight
- This work has generally used policy search (see Bagnell and Schneider 2001) as well as the PEGASUS algorithm with simulation based on a learned transition model (see Ng et al. 2004)