Introduction to reinforcement learning

Erik Pärlstrand

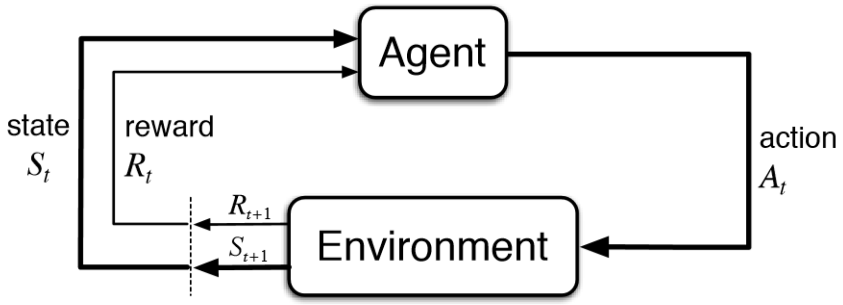
March 2018

Introduction

In supervised learning, we are try to find a good mapping from the feature-space to the target (x to y). In unsupervised learning, we are try to transform the features, do clustering etc. They are "fairly" similar. However, reinforcement learning is different from these two. Common terminology in reinforcement learning:

- **Agent** - sense the environment and operates in it.
- **Environment** - real/simulated world where the agent lives in.
- **States** - different configurations of the environment that the agent can act in. In this course we will only consider finite number of states.
- **Rewards** - rewards that the agent receives. It tries to maximize immediate and long-term rewards. The rewards are real numbers.
- **Actions** - what the agent does in it's environment.

Below is an example of the setup:



Episode: represents one run of the game. Our rl-agent learn across multiple episodes. Note that the number of episodes is a hyper-parameter.
There are two types of episodes: **episodic task** have a terminal state and we play it over and over again. **Continuous task** never ends (no terminal state). A state is called terminal if no action can be taken from that state.

Exploration vs exploitation dilemma

The exploration versus exploitation dilemma refers to the problems where we want to acquire new knowledge and maximize the reward at the same time. Think of a row of slot machines in a casino. Each machine has its own probability of winning.

How do we figure out which of the machine has the best odds, while at the same time maximizing the profit? If you constantly played one of the machines you would never learn anything about the other machines. If you always picked a machine at random you would learn a lot about each machine but would not make as much money as you could have always playing the "best" machine.

Another similar problem is the **Credit assignment problem**; what did I do in the past that led to the rewards I'm receiving now?

Epsilon greedy

Let us image the multi-armed bandit described above. We have N different slot machines which gives a reward of 0 or 1. However, the win rate is unknown. One way to solve this problem is with the epsilon greedy algorithm. One chooses a small number ϵ as the probability of exploration, typically $0.05 - 0.1$. The algorithm for epsilon greedy follows:

```
for i in range(0, num_runs):
    p = uniform_random(lower=0.0, upper=1.0)

    if p < epsilon:
        pull random arm
    else:
        pull current best arm
```

where the current best arm is defined as the slot machine with the highest estimated win-rate. In the long run it will allow us to explore each arm an infinite number of time. However, one problem is that we reach a point where we explore even when we don't need to. This can be solved by decaying ϵ, i.e.

$$\epsilon(t) = \frac{\epsilon_0}{t}$$

However, decaying epsilon is only applicable if we have stationary bandits, i.e. the win-rate doesn't change with time. If the win-rate does change with time (non-stationary bandits), we don't want to decay ϵ.

MDP

The (first) order Markov property is defined as:

$$p(x_t|x_{t-1}, ..., x_1) = p(x_t|x_{t-1})$$

In the reinforcement learning we have:

$$p(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a, ...,) = p(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a) = p(s', r|s, a)$$

where s' represent the state we end up in at time $t+1$, r is the reward we receive in time $t+1$, s is the current state and a is the action we took.

Any reinforcement learning task with a set of states, actions and rewards that follow the Markov property is a MDP (Markov Decision Process). A MDP is defined by:

- A set of states.
- A set of actions.
- A set of rewards.
- A set of state-transition probabilities.

The policy (π) is not part of MDP, but along with the value function form the solution. Think of π as a shorthand for the algorithm the agent is using to navigate the environment.

Value function

The return is defined as:

$$G(t) = \sum_{\tau=1}^{\infty} R(t+\tau)$$

We usually want to weight the rewards long into the future less then the close ones. Therefore we can introduce a discount factor γ:

$$G(t) = \sum_{\tau=0}^{\infty} \gamma^{\tau} R(t+\tau+1)$$

If $\gamma = 1$ then we weight all rewards equally. If $\gamma = 0$ then we have a truly greedy method. Usually one chose $\gamma \approx 0.9$.

The value function is defined as the expected future reward from a state s. Estimating the value function is a central task in reinforcement learning. It is defined as:

$$V_\pi(s) = E_\pi[G(t)|S_t = s]$$

We can also define the value-action function as:

$$Q_\pi(s,a) = E_\pi[G(t)|S_t = s, A_t = a]$$

Optimal policy

In reinforcement learning we are typically interested in two aspects:

- Finding $V(s)$ given a policy. This is called the prediction problem.
- Finding the optimal policy π. This is called the control problem.

Optimal policy and optimal value function are interdependent. We can measure relative goodness between two policies:

$$\pi_1 \geq \pi_2 \text{ if } V_{\pi_1}(s) \geq V_{\pi_2}(s) \ \forall s \in S$$

Therefore the optimal policy is the best policy:

$$V_*(s) = \max_\pi V_\pi(s) \ \forall s \in S$$

Policy evaluation

In the previous section we got the definition of the value function for a given policy. It can be shown (using the definition of $G(t)$ and conditional probability) that the value function can be expressed as:

$$V_\pi(s) = E_\pi[G(t)|S_t = s] = ... = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V_\pi(s')]$$

This is called the Bellman equation. Note that the expected future reward is only dependent on the next state's value function $V_\pi(s')$. In a similar fashion we can define the value-action function as:

$$Q_\pi(s,a) = E_\pi[G(t)|S_t = s, A_t = a] = ... = \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma V_\pi(s')]$$

Policy improvement/iteration

Given a current policy π, find π' s.t. $V_\pi(s) \leq V_{\pi'}(s)$. If we have Q:

$$\pi'(s) = \text{argmax}_a Q_\pi(s, a)$$

If we have V:

$$\pi'(s) = \text{argmax}_a \sum_{s'} \sum_{r} p(s', r|s, a)[r + \gamma V_\pi(s')]$$

Notice that it's greedy. Continue looping this until the policy doesn't change.

For policy iteration, alternate between policy evaluation and policy improvement. Keep doing this until the policy doesn't change.

Value iteration

Value iteration is an alternative technique for solving the control problem. One disadvantage of policy iteration is that we use nested "loops".

Value iteration combines policy evaluation and policy improvements into one step:

$$V_{k+1}(s) = \max_a \sum_{s'} \sum_{r} p(s', r|s, a)[r + \gamma V_k(s')]$$

Monte Carlo methods

Dynamic programming required us to known the probability distribution $p(s', r|s, a)$. However, this is usually not the case in real applications.

The Monte Carlo method is model free, i.e. we don't need to know/specify $p(s', r|s, a)$. Instead of calculating the true expected value of G as we did in dynamic programming, we calculate the sample mean instead.

Monte Carlo policy evaluation

Recall that the value function is given by:

$$V_\pi(s) = E_\pi[G(t)|S_t = s]$$

This expectation can be approximated with Monte Carlo, i.e:

$$\bar{V}_\pi(s) = \frac{1}{N} \sum_{i=1}^{N} G_{i,s}$$

where i is the episode index. For a given episode we can generate tuples of states and rewards, i.e. $(s_1, r_1), ..., (s_T, r_T)$. Using these, we can calculate G (backwards) using the definition of G, i.e.:

$$G(t) = r_{t+1} + \gamma G(t+1)$$

Note that we need to wait until the episode is completed before we can update the sample mean.

Exploring starts

If the policy (π) is deterministic, some/many (s, a)-pairs will never be visited. One approach to solve this is the exploring starts method. For an episodic task, choose a random initial state and action. Thereafter follow the policy π.

One disadvantage of exploring starts is that we need to know all states and actions for the environment beforehand. However, in many applications this is not possible.

One alternative approach to exploring starts is to use epsilon greedy, i.e. change the policy such that the actions are random with a probability ϵ.

Monte Carlo policy improvement/iteration

In Monte Carlo with a given policy we only have the value function $V(s)$ and don't know which actions leads to better $V(s)$, since we can't do a look-ahead search (as in dynamic programming). We can only play episodes and get the rewards/states. However, this can be solved by using the value-action function Q. Recall that:

$$Q_\pi(s, a) = E_\pi[G(t)|S_t = s, A_t = a]$$

As for policy evaluation, this expectation can be approximated with Monte Carlo, i.e:

$$\bar{Q}_\pi(s, a) = \sum_{i=1}^{N} G_{i,s,a}$$

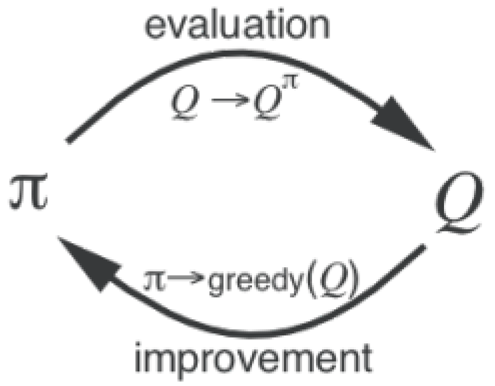
where i is the episode index. For a given episode we generate tuples of states, actions and rewards, i.e. $(s_1, a_1, r_1), ..., (s_T, a_T, r_T)$. Then we get policy by:

$$\pi(s) = \text{argmax}_a \bar{Q}_{\pi'}(s, a)$$

and the value function by:

$$V_\pi(s) = max_a \bar{Q}_\pi(s, a)$$

Now, policy iteration can be performed by alternate between policy evaluation and policy improvement, as described in the figure below:



However, this approach would imply that we need to estimate the value-action function from scratch for each policy. However, this would require a lot of iterations.

One solution to this problem is to not start with a fresh Monte Carlo evaluation each round but instead keep updating the same Q.

Temporal difference learning

Recall that one disadvantage of dynamic programming is that it requires a full model of the environment and never learned from experience. Moreover, one disadvantage of Monte Carlo is that updates can only be done after completing the episode.

Temporal difference (TD) learning combines ideas from both dynamic programming (perform updates based on current values) and Monte Carlo (sample from the environment) and is fully online, i.e. it can update the values during the episode.

In this course we will only cover TD(0), however there is a more general method called TD(λ), but it is outside of the scope for this course.

TD(0) for predict

Recall the definition of the value function V_π:

$$V_\pi(s) = E_\pi[G(t)|S_t = s]$$

We can now define it recursively:

$$V_\pi(s) = E_\pi[R(t+1) + \gamma V_\pi(S_{t+1})|S_t = s]$$

$R(t+1) + \gamma V_\pi(S_{t+1})$ is an unbiased estimate of $V_\pi(s)$. This observation motivates the following algorithm for estimating V_π.

The algorithm starts by initializing $V_\pi(s)$ arbitrarily. We then repeatedly evaluate the policy π, obtain a reward r and updating the value function for the old state using:

$$V_\pi(s) \leftarrow V_\pi(s) + \alpha(r + \gamma V_\pi(s') - V_\pi(s))$$

where α is a positive learning rate. s and s' are the old and new states, respectively.

SARSA

SARSA is similar to TD(0) predict described above, but applied to the action-value function instead, i.e.:

$$Q_\pi(s,a) \leftarrow Q_\pi(s,a) + \alpha[r + \gamma Q_\pi(s',a') - Q_\pi(s,a)]$$

As before, SARSA is usually combined with epsilon greedy in order to make sure that we visit all state-action pairs. However, it has been stated that SARSA will converge if the policy converges to a greedy policy, i.e.

$$\epsilon = \frac{\epsilon_0}{t^c}$$

where ϵ_0 is the initial value and c a hyper-parameter.

Furthermore, we can decay the learning rate α. However, we want to decay the learning rate according to how many times we have seen the state-action pairs since some states-actions are updated more frequently, i.e.:

$$\alpha = \frac{\alpha_0}{\text{count}(s,a)}$$

where α_0 is the initial value.

On-policy versus off-policy

Up until now we have only examine on-policy, i.e. follow the best current policy. However we can do off-policy, i.e. do random actions. There is no reason that an agent has to do the greedy action; agents can explore or they can follow options. This is what separates on-policy from off-policy learning.

Fortunately, using an off-policy still allows us to find Q_*.

Q-learning

Q-learning is an example of an off-policy method. It is defined as:

$$Q_\pi(s,a) \leftarrow Q_\pi(s,a) + \alpha[r + \gamma \text{max}_{a'} Q_\pi(s',a') - Q_\pi(s,a)]$$

The reason that Q-learning is off-policy is that it updates its Q-values using the Q-value of the next state s' and the greedy action a'. In other words, it estimates the return for state-action pairs assuming a greedy policy were followed despite the fact that it is not following a greedy policy.

The reason that SARSA is on-policy is that it updates its Q-values using the Q-value of the next state s' and the current policy's action a''. It estimates the return for state-action pairs assuming the current policy continues to be followed.

The distinction disappears if the current policy is a greedy policy. If we do Q-learning with a greedy policy then we will also do SARSA.

Tabular versus approximation methods

Up until now we have only studied tabular methods, i.e. we store the value (-action) function for each state (-action) pair. However, this becomes in-feasible when the state (and action) space becomes large. This can be solved with approximation methods, i.e. for a given state (and action) we approximate $V_\pi(s)$ ($Q_\pi(s,a)$).

In this course, we will only cover linear models with gradient descent. However, we could use other models, like neural networks. The only requirement is that the model (i.e. f) is differentiable. Furthermore, we will use feature engineering, i.e. for a given state (and action) we "come up" with the features.

Approximate Monte Carlo for prediction

Let us consider the value function. First, we do feature extraction; converting a state s to a feature vector x:

$$x = \phi(s)$$

We then want a function, parametrized by θ that approximate the value function:

$$\hat{f}(x|\theta) \approx V_\pi(s)$$

Since we only cover linear models in this course, we have that:

$$f(x|\theta) = \theta^T x$$

One thing to keep in mind is that linear models are not very expressive, so if we choose a poor set of features the linear model might not be able to approximate $V_\pi(s)$ very well. Therefore, the feature engineering part becomes a important step.

Since the value function is continuous, we will do regression. A common loss function for regression is the squared error:

$$L = [V_\pi(s) - \hat{V}_\pi(s)]^2$$

Using the definition of $V(s)$ for Monte Carlo, we get that the loss is given by:

$$L = \sum_{i=1}^{N} [G_{i,s} - V_\pi(s)]^2$$

Using the definition of gradient descent yields:

$$\theta = \theta - \alpha \frac{\partial L}{\partial \theta}$$

Which gives:

$$\theta = \theta + \alpha(G_s - \hat{V}_\pi(s))x$$

θ will be updated for each training example i.

Approximate TD(0) for predict

In Monte Carlo methods we use the return G, but for TD(0) we are using $r + \gamma V_\pi(s')$ as a proxy for the return. However, this cause a problem for approximation methods because the target is not a real target.

Fortunately, it still works even though we do not use a real target. However, the gradient is not a true gradient and therefore we will call it "semi-gradient". Using the loss function defined above yields:

$$\theta = \theta + \alpha(r + \gamma V_\pi(s') - \hat{V}_\pi(s))x$$

Approximate SARSA

For SARSA, we will approximate $Q_\pi(s, a)$ instead of $V_\pi(s)$. It is typically more difficult to approximate $Q_\pi(s, a)$ compared to $V_\pi(s)$ since we need to approximate $|S| \times |A|$ values. Similar to the discussion above, we will use a semi-gradient for SARSA. Using the loss function defined above yields:

$$\theta = \theta + \alpha(r + \gamma Q_\pi(s', a') - \hat{Q}_\pi(s, a))x$$