Reinforcement Learning.

Erdem Koyuncu.

- Studies the question of how agents should take actions in an environment to maximize a cumulative reward.

- Many applications, including:
  - * Robot control · Individual robots, swarm intelligence
  - * Games / Strategy.
    Chess, backgammon, Go, checkers, ...
    Atari games (Space invaders, etc ...
    Very complex games like Starcraft.

→ In all examples RL agents much better than best humans or usually best classical algorithms (non-RL).

* Telecommunication systems:
  How should the users of the spectrum use the available resources in an efficient manner?

* ....

ANY scenario where a model of the environment is known & one can interact with it, but the objective is too complex to be solved analytically or with other approaches.

Mathematical formulation of RL relies on Markov Decision Processes. (MDP)

An MDP can be defined via

$\mathcal{X}$: a finite state space

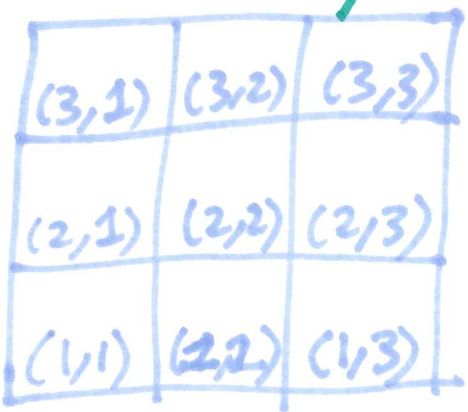
$\mathcal{A}$: a finite action space.

r: a reward function.

Specifically $r(x, a, y)$ is the reward obtained for a transition from state $x \in \mathcal{X}$ to state $y \in \mathcal{X}$ due to action $a \in \mathcal{A}$.

$P_a(x, y)$: Probability of transitionning from state x to y given that one takes action a.

Ex: 3×3 grid environment.



Person at initial location (1,1), wishes to travel to target (3,3).

State space $\mathcal{X} = \{(1,1), \ldots, (3,3)\}$ (3,3): terminal state.

Actions $\mathcal{A} = \{U, D, L, R\}$. up, down, left, right

- A step fails with prob ε, successful with probability $1-\varepsilon$
  - in which case the person cannot move from its current location.
- Goal: Reach (3,3) with the min. number of actions taken.
- Rewards: A reward of −1 for every action. A large negative reward for every action that attempts to step outside. 0 reward after (3,3).
- This reward design tries to ensure:

sequence of actions that minimize E[number of actions to reach (3,3)]

$\equiv$

sequence of actions that maximize E[cumulative reward].

Value of an initial state $X_0 = x$ given controls $\{A_t\}$:

$$J(x, \{A_t\}) = E\left[\sum_{t=0}^{\infty} \gamma^t R(X_t, A_t) \,\middle|\, X_0 = x\right]$$

γ: Reward Discount (ensures value is finite).

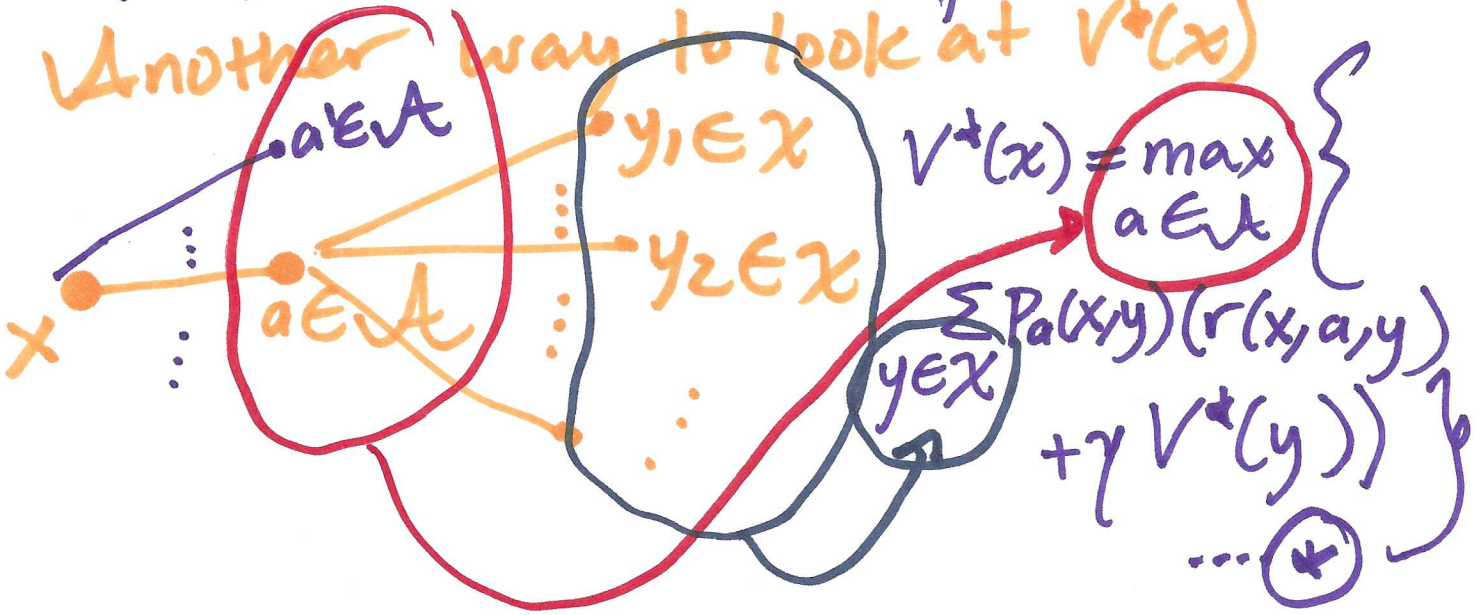
$R(X_t, A_t)$: Reward obtained at state X_t with actions A_t

Optimal Value Function:

$$V^*(x) = \max_{\{A_t\}} J(x, \{A_t\})$$

ie which sequence of (possibly random) actions should be taken so as to maximize the value function?

Another way to look at $V^*(x)$



To understand how the equation is derived, note that the optimal value function is the best (maximum) expected cumulative reward for a given initial state x.

Given initial state x, the expected reward of a policy that chooses action a all the time is

$$\sum_{y \in X} P_a(x,y)\left(r(x,a,y) + \gamma V^*(y)\right).$$

maximizing over a should give the optimal value function given by (*).

We can now define:

$$Q^*(x,a) = \sum_{y \in X} P_a(x,y)\left(r(x,a,y) + \gamma \max_{a'} Q^*(y,a')\right)$$

where

$$V^*(x) = \max_a Q^*(x,a)$$

If we can find this optimal Θ^*, we have found the optimal policy to maximize the expected rewards.

The Θ learning algorithm can be used for this purpose. To intuitively understand how the algorithm is derived, note:

$$\Theta^*(x,a) = E_y\left[r(x,a,Y) + \gamma \max_a \Theta^*(Y,a)\right]$$

(expectation calculated over the next state).

Given x, a, and one $y \in X$ we imagine updating $\Theta(x,a)$ to be close to $r(x,a,y) + \gamma \max_a \Theta(y,a)$ for the particular samples (x,a,y) we consider. This leads to:

$$\Theta(x,a) \leftarrow (1-\alpha)\,\Theta(x,a) + \alpha\left(r(x,a,y) + \gamma \max_a \Theta(y,a)\right)$$

This is the Θ learning algorithm.

In practice, the algorithm is run through many episodes of the agent starting from the initial state of the environment and exploring it.

1. Initialize Q table (e.g. randomly)
2. for episodes = 1 to N:
   - 2.1 $x \leftarrow$ Initial state.
   - 2.2 for $t = 1$ to T:
     - 2.2.3 . With prob. ε select a random action a (this is called exploration) Otherwise set $a = \arg\max_b Q(x,b)$ to be the optimal action for state x (this is called exploitation).
     - 2.2.4 . Observe reward r and next state y.
     - 2.2.5 Update $Q(x,a) \leftarrow (1-\alpha) Q(x,a) + \alpha(r + \gamma \max_c Q(y,c))$.
     - 2.2.6 . Set $x \leftarrow y$.

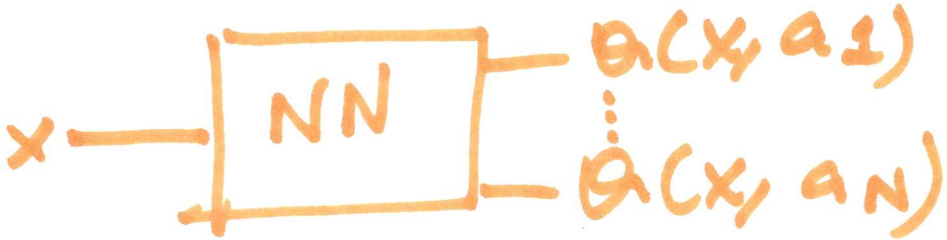
The algorithm is guaranteed to converge to the optimal policy (when $\varepsilon = 0$). Setting $\varepsilon = 0$ in the beginning may allow faster convergence.

Deep Q Networks.

In certain applications, the dimension/size of the state space may be impractically large so that ordinary Q-learning will not be applicable (e.g. atari games, chess, ...).

One can then approximate $Q(x,a)$ via a neural network!

A typical architecture accepts the state x as the input and provides the Q values $Q(x,a)$ for every possible action $a \in \mathcal{A}$.



The reason this design is favored is that one needs all the Q values $Q(x,a)$, $a \in \mathcal{A}$ to determine the optimal action anyway.

How to train a deep Q network? (10)

In an optimal policy, we need:

$$Q^*(x,a) = E_y\left(r(x,a,y) + \gamma \max_a Q^*(y,a)\right)$$

We can convert this to a loss function form that is more suitable to neural network training. Suppose we have a sequence of (state, action, reward, next state) vectors, i.e, (x_i, a_i, r_i, y_i), $i=1,\ldots,K$. Then, we essentially wish to minimize:

$$\sum_{i=1}^{K}\left(Q(x_i,a_i) - \left(r_i + \gamma \max_a Q(y_i,a)\right)\right)^2$$

If we can minimize this, we can find the optimal policy (provided y_i, r_i is of course, generated by the environment as a function of x_i, a_i).

The nice thing about this form is that it is almost in the form of a NN loss function, $Q(x_i,a_i)$ is the NN output and the rest of the terms is the desired output for the particular inputs...

One problem is that the desired output is not fixed, but will change as we change the network parameters. This results in instability during training. To fix this, the idea is to create two networks. One will be used to keep track of the target values

$$r_i + \gamma \max_a Q(y_i, a; W_t)$$

and will thus be called a target network.

The other will be used to set & update the policy

$$Q(x_i, a_i; W_p)$$

and thus called the policy network.

Note during training that the networks will have different parameters.

Only the policy network will be optimized using the backpropagation algorithm.

Every once in a while, we will copy the parameters of the policy network to the target network, updating $W_t \leftarrow W_p$.

Another issue is related to how the sequence $(x_i, a_i, r_i, y_i)_{i=1,\dots,K}$ is generated to train $Q(x_i, a_i; W_p)$. Ideally, this sequence will be generated in a model-free fashion in a way similar to the Q learning algorithm. But then, there will be a lot of dependency between elements of the sequence. In particular, $x_{i+1} = y_i$, for example. We have discussed that it is very hard to train neural networks over correlated data. The solution is to use experience replay:

Store experiences (x_i, a_i, r_i, y_i) to a buffer as the agents experience them. When training the policy network, sample a random batch of samples from this replay memory and use the batch to train W_p.

The DQN Algorithm.

1. Initialize replay memory D to capacity M.
2. " policy network with weights W_p.
3. " target " " " W_t.
4. for episodes = 1 to N
   - 4.1 Initialize $x \leftarrow$ Initial state.
   - 4.2 for $t = 1$ to T.
     - 4.2.1 With prob ε set a = random action. otherwise set $a = \arg\max_b Q(x, b; W_p)$
     - 4.2.2 Observe reward r, new state y.
     - 4.2.3 Store (x, r, a, y) in replay memory
     - 4.2.4 Sample random minibatch of transitions (x_i, r_i, a_i, y_i), $i = 1, \ldots, L$ from replay memory.

       Let $z_i = r_i + \gamma \max_c Q(y_i, c; W_t)$ be the target values.

       !! careful W_t, not W_p

       Use the Backprop algorithm to minimize

       $$\sum_{i=1}^{L} \left(Q(x_i, a_i; W_p) - z_i \right)^2$$

     - 4.2.5 Every C steps set $W_t \leftarrow W_p$.
     - 4.2.6 Set $x \leftarrow y$.