Reinforcement Learning

Aditya Vasantharao

April 2020

1 Introduction

Reinforcement Learning (RL) is a branch of machine learning which uses a reward system to determine the action that the algorithm should take. These algorithms are very popular in the game development industry, with many games using these algorithms to play as "bots" in single-player modes, but they are also very useful in the fields of robotics, advertising, content recommendation systems, and, in general, fields which have little-to-no training data.

2 RL Overview

RL algorithms contain two main components: the agent (the algorithm that makes decisions to try to solve the main problem) and the environment. In each iteration of this algorithm, the environment sends the agent a state (the position of the agent in the environment), the agent performs an action based on that state, and the environment sends the agent a reward based on the success of that action.

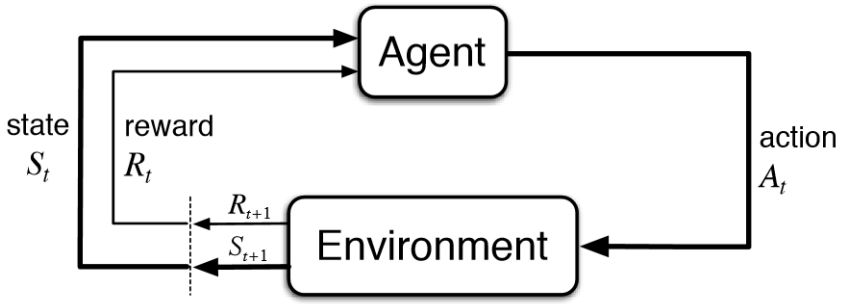

Figure 1: RL Algorithm Illustration

2.1 Markov Decision Process

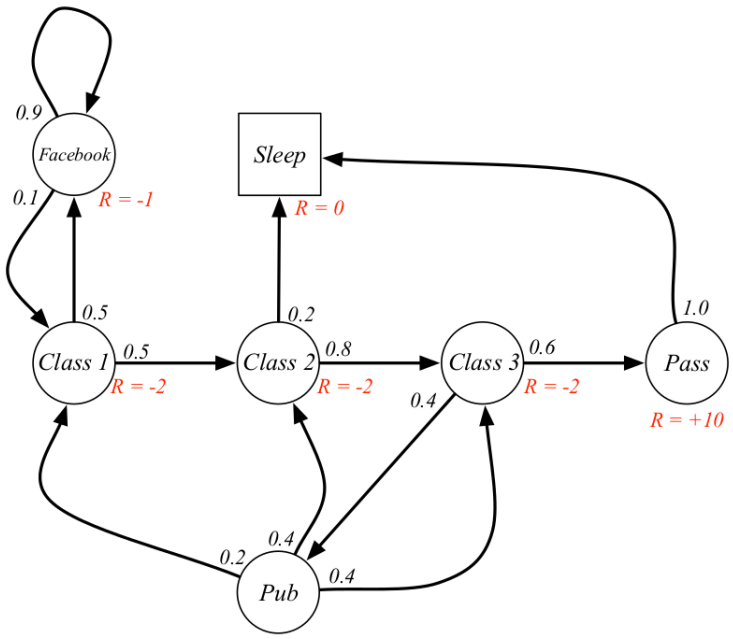

Figure 2: MDP Illustration

In Figure 2, $R =$ indicates rewards, each node indicates a state, and the decimal values placed along the edges that connect any two nodes indicate state transition probabilities.

The process in which the agent and the environment interact with each other is called the Markov Decision Process (MDP). The first part of the MDP, which describes the environment, contains the following variables: S, A, P_a, R_a, where

- S is a set of all the states the agent can go through
- A is a set of all the possible actions an agent can take
- $P_a(s, s') = Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$, called the state transition function, is a function which is used to determine which state s' the agent will go to after completing action a_t in state s_t at a given time t
- $R_a(s, s')$, called the reward function, is a function which determines the reward that an agent gets after moving from state s to state s'

For context, P_a is typically implemented either using a lookup table (deterministic) or using an algorithm which determines the probability that the agent will enter a certain state given the parameters, choosing the state with the

highest associated probability (stochastic), and R_a is typically implemented either using a lookup table (deterministic) or using a continuous function, such as a polynomial (non-deterministic). The deterministic methods are only used in problems with a discrete and relatively small action and state space, and the stochastic/non-deterministic methods are used in problems with a large, continuous action and state space. Both P_a and R_a are implemented by the programmer, so it is up to them to determine how often the agent should receive a reward and what exactly constitutes as a state.

2.1.1 The Discount Factor

Along with these 4 parameters, there is another parameter that is used as well: the discount factor, represented as γ. The discount factor, which is a value such that $0 \leq \gamma \leq 1$, is multiplied by the output of the reward function to give the final reward that the agent receives. The discount factor mainly serves a purpose in the return and the greediness of the agent.

In all RL problems, the agent's ultimate goal is to **maximize its return** (the total discount reward that the agent will receive from now, time t, onwards). The return is given by this equation,

$$G(t) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

where R is the reward function. In this equation, smaller discount factors make the agent care more about short-term rewards and larger discount factors make the agent care more about long term-rewards. To clarify, a discount factor of 0 makes the return R_{t+1}, meaning that the agent would only care about the immediate reward gained by taking its next action, and a discount factor of 1 means that the agent thinks that future actions are equally as important as the next action that the agent will take, so the agent is far more willing to experiment to reach the large long-term reward. In practice, however, discount factors of both 0 and 1 are typically not used, because both are too extreme in how they prioritize short-term rewards and neglect long-term rewards or vice-versa.

Discount factors are not applicable to every problem, however, because if the RL problem is an episodic task (a task where there is a terminating condition, such as a game of tic-tac-toe, where the task ends when someone wins) then rewards are typically assigned once—at the end of the task—and the return only consists of R_{t+1}, thus the discount factor is always 0 in these tasks. Instead, discount factors are only applicable to continuous tasks (tasks which don't end, such as YouTube's video recommendation system), because rewards are assigned more often and there's no terminating condition which resets the return.

2.1.2 The Agent

The second part of the MDP is the decision-making process, which describes the agent. The agent uses one or more of the following components to determine

what action to do next:

- π, called the policy function. This function decides which action to take in a given state and involves two variables: a, the action, and s, the state. π can be written deterministically (with a lookup table implementation),

  $$a = \pi(s)$$

  or stochastically,

  $$\pi(a|s) = Pr(A_t = a|S_t = s)$$

- $v_\pi(s)$, called the state-value function. This function calculates the value of a state by predicting the return that the agent would get if it entered that state given that it executes policy π. It's used in the optimal state-value function,

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

  which finds the policy which returns the maximum possible reward. $v_*(s)$ is used in **policy learning**.

- $q_\pi(s, a)$, called the action-value function, determines how good it is to take a particular action in a certain state by calculating the expected return after taking action a in state s and following policy π from then onwards. This function is used in the optimal action-value function,

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

  which finds the action which returns the maximum possible reward. $q_*(s)$ is used in **q-learning**, a method which determines the best action to take at the current state.

- The agent can also have a model, which is an algorithm which allows the agent to predict what the environment will do next (e.g the next state, the next immediate reward, etc.).

RL agents can be categorized as the following:

- **Value based**: the agent uses $q_*(s)$ to pick the action that gives the highest reward in the current state. These agents use q-learning.
- **Policy based**: the agent tries to optimize its policy as much as it can and picks actions based on $v_*(s)$.
- **Actor Critic**: the agent uses both $q_*(s)$ and the policy to pick the action.
- **Model free**: regardless of its use of the action-value function and the state-value function, the agent does not use a model.
- **Model based**: regardless of its use of the action-value function and the state-value function, the agent uses a model.

3 RL Algorithms

So far, we have discussed the basics of RL and how to set up an RL program, but we haven't dove into the specific algorithms that RL programs use to get the job done. These algorithms are typically the main focus of an RL program (you'll often see the specific type of algorithm used in the title of a research paper that used RL to solve a problem), and they're also the main focus of RL research. RL algorithms are implemented in the agent and fall into one of the categories discussed at the bottom of section 2.1.2. The following algorithms are, by no means, and exhaustive list, but these are popular algorithms that can be applied to a wide variety of topics. Other than these, some other algorithms that you may want to research before writing an RL program to solve a task are SARSA, NAF, A2C/A3C, and DDQN.

3.1 Q-Learning

The main focus of Q-Learning is to maximize the Q-value (the value returned by $q_*(s, a)$). This value determines how good the best action that the agent can take really is. To do this, we have to make the agent learn $q_*(s, a)$. The algorithm to do this is below.

Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
Sates $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
procedure QLEARNING($\mathcal{X}$, A, R, T, α, γ)
    Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
    while Q is not converged **do**
        Start in state $s \in \mathcal{X}$
        while s is not terminal **do**
            Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow \arg\max_a Q(x, a)$)
            $a \leftarrow \pi(s)$
            $r \leftarrow R(s, a)$ ▷ Receive the reward
            $s' \leftarrow T(s, a)$ ▷ Receive the new state
            $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
            $s \leftarrow s'$
    return Q

Figure 3: Q-Learning Algorithm

Note that converging, in this case, means that we are unable to improve

$q_*(s, a)$ any further, terminal means the terminating state, and the black-box transition function is the state transition function. Also, the policy is implemented deterministically (via, for example, a lookup table which matches states to actions) and is updated every iteration.
This algorithm works off of a q-table, which is a table of $(state, action)$ pairs and each pair is mapped to its respective expected total reward, which is calculated by $q_*(s, a)$, given below. Note: R is the reward function and p is the state transition probability.

$$Q^*(s,a) = R(s,a) + \gamma\, \mathbb{E}_{s'}[V^*(s')]$$
$$Q^*(s,a) = R(s,a) + \gamma \sum_{s' \in \mathbb{S}} p(s'|s,a) V^*(s')$$

Since,

$$V^*(S) = \max_a Q^*(s,a)$$
$$V^*(S) = \max_a \left[R(s,a) + \gamma \sum_{s' \in \mathbb{S}} p(s'|s,a) V^*(s') \right]$$

Figure 4: $q_*(s, a)$ and $v_*(s, a)$ Equations

After Q is learned, the RL program will use Q to pick actions until the terminal state is reached.

3.2 Deep Q Network (DQN)

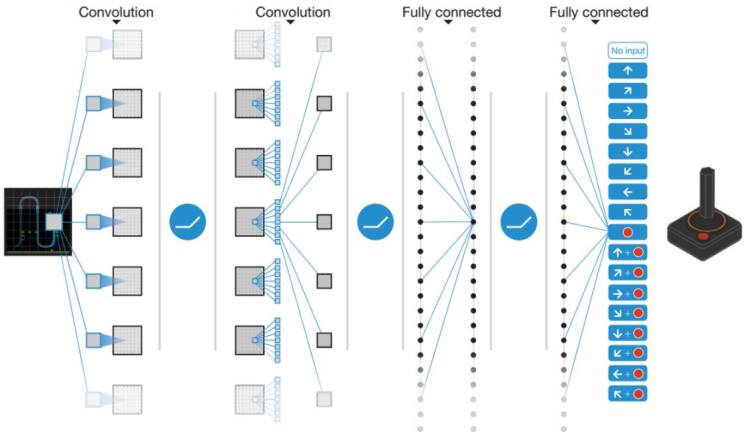

Figure 5: DQN Illustration

The problem with Q-learning algorithms, as you may have noticed, is that they only work for discrete state spaces and are inefficient for large state spaces. In scenarios where the state space is continuous and/or large, we want to use a DQN algorithm. DQN learns $q_*(s, a)$ via the use of a fully-connected neural network, as shown in Figure 5. The neural network takes in the current state as input and outputs the Q-value for each action.

The question, now, becomes the following: how do we train the neural network? We want the neural network to learn Q until it is fully optimized, so we have to maximize the highest Q-value across all actions. The equation to get the maximum Q-value across all of the current actions in the given state (where ϕ represents the state and θ represents the weights in the neural network) is the following:

$$r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta^-)$$

Using this equation, we can use the algorithm in Figure 6 to train the neural network. This algorithm uses experience replay, which is just a dataset of $(state, action, reward, next\ state)$ tuples that the algorithm uses for sampling to train the network. Experience replay is necessary for training the neural network because the data points in the experience replay dataset are separated by time, which reduces the correlation between these otherwise highly-correlated data points. Also, note that, in this algorithm, s is the current state before preprocessing, $\phi(s)$ is the function which returns the preprocessed state, and ϕ represents the preprocessed state.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 6: DQN Algorithm

3.3 Deep Deterministic Policy Gradient (DDPG)

Although DQN performs very well in situations where the action space is discrete and relatively small, in problems where the action space is continuous or very large, we must use a different algorithm. One such algorithm that you can use which works regardless of how the state and action space are configured is DDPG. DDPG is an Actor-Critic algorithm and it looks very similar to the DQN algorithm, but it has an additional section for the policy. The critic evaulates the policy generated by the actor via the Temporal Difference Error (TD Error) function,

$$E = r_{t+1} + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)$$

Figure 7 shows an illustration of how the DDPG algorithm looks.

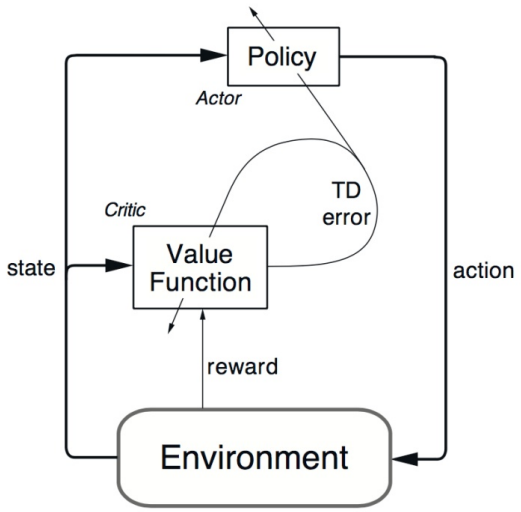

Figure 7: DDPG Illustration

An issue that DDPG produces is that it rarely explores or experiments when finding new actions. To fix this, we add noise to either the nodes in the parameter space or to the action space, as shown in Figure 8. A popular method of adding noise is the Ornstein-Uhlenbeck Random Process.

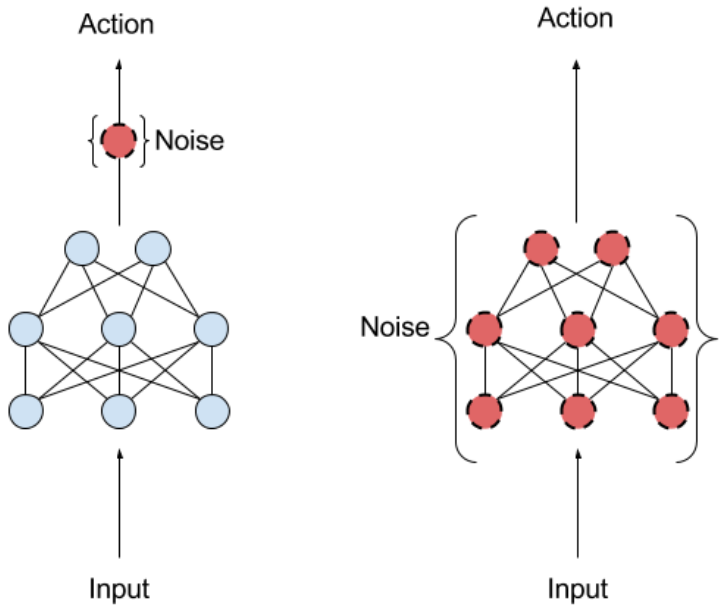

Figure 8: Adding noise to the action space (L) or the parameter space (R)

Note that in the DDPG algorithm shown in Figure 9, the random process N is the method of adding noise to the parameter or action space, J is the expected return, the replay buffer R is the dataset of experience replay data points used to update the critic, the actor (via the use of a policy gradient, similar to the gradient used in backpropogation for neural networks), and the target networks, θ represents the weights in the neural network, and the mixing factor τ is a hyperparameter which is typically very small—many people use 0.001 for τ.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 9: DDPG Algorithm

4 Applications

RL Learning is used primarily in cases where there is little-to-no training data. This is why RL is used in fields such as game development (in the making of "bots"), robotics, advertising, and content recommendation systems. For example, RL is used in "bot" development because, in this field, the algorithm initially has no knowledge or experience of how to interact with its environment and achieve the final goal. Other algorithms, such as Neural Networks or SVMs, wouldn't be able to function properly in environments like these because of the lack of training data. This becomes a problem especially in situations where the goal is to create an AI which is capable of outperforming humans (such as the creation of AlphaGo), as in those cases you can't even feed the other algorithms data from humans playing the game. In content recommendation systems, for instance, there's no way to get any data on what a user's preference for content is when that user enters the platform for the first time, so an RL algorithm must be used.

5 Conclusion

Reinforcement Learning is a rapidly expanding field which is especially useful in fields where there is little prior data or testing, such as niche research fields or

research fields where the concept behind the research is very new. This lecture should give a good background on RL and a couple of popular algorithms that are used in RL.

6 References

- https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287
- https://towardsdatascience.com/reinforcement-learning-an-introduction-to-the-concepts-applications-and-code-ced6fbfd882d
- https://towardsdatascience.com/getting-started-with-markov-decision-processes-reinforcement-learning-ada7b4572ffb
- https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/
- https://medium.com/@m.alzantot/deep-reinforcement-learning-demysitifed-episode-2-policy-iteration-value-iteration-and-q-978f9e89ddaa
- https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b