Reinforcement learning problems are typically viewed as Markov decision processes, these are a special case of decision processes that possess the Markov property. That is, the future evolution of the process can be predicted using only on the current state.

There exists a set of states , and a set of actions.

Time progresses in discrete steps. At each time-step t, the agent receives a state and reward , then takes an action at ∈ A. The process then transitions to a new , chosen stochastically with a probability dependent on and . The process continues until a terminal state is reached, then halts. In the context of reinforcement learning, the manner in which an MDP transitions between states is often called the “transition model”.

The problem of training an agent with the best possible performance on a control problem can be framed as finding a policy that maximises the accumulated reward as t → ∞

Stationary problems VS non-stationary problems (harder).

A non-stationary problem is one in which the transition model changes as the process progresses (significantly harder).

**V-Value.**

This function gives the expected sum of discounted reward after the state , under the policy . Sutton & Barto define the V-value as:

**Q-Value.**

This function gives the expected sum of discounted reward after taking action in state , then following the policy afterwards. Sutton & Barto defines the Q-value as:

**Advantage.**

The advantage of an action isolates the action’s contribution to the reward from the inherent value of the state. Sutton & Barto defines the Advantage function as:

There are a variety of ways to classify reinforcement learning methods.

**Policy-Based.** The model produces a probability spread of actions for a given state, this can be sampled from to enact a policy. During training, the model learns to produce the action probabilities that map to the optimal policy.

**State-Value-Based.** The model learns to estimate the V-Values of states. This alone is not useful in constructing a policy. However if the transition model of the process is also known, state-values can be used to construct policies i.e. by calculating the weighted average value of the successor states reached by each action.

**State-Action-Value-Based.** The model learns to approximate the Q-Value of state-action pairs. These values are used to enact a policy i.e. by selecting the action with the highest Q-Value for the current state. Unlike a state-value-based approach, this can be used to construct a policy even if the transition model is not known.

**Actor-Critic.** The model has two components; the actor, and the critic. The actor is policy-based, the critic is value based. Either V-values or Q-values can be used, depending on the algorithm. The critic’s output is used to approximate the advantage function, and the approximation is used to direct the actor during the learning process. Only the actor’s output is necessary to enact the learned policy.

The method by which training data is gathered is also an important factor in the final behaviour of agents. Note that these classifications are orthogonal to the output-structural methods, any combination of the two is possible.

**Monte-Carlo.**

The agent is allowed to interact with the environment until it reaches a terminal state (a full epoch), no learning occurs during the epoch. Once the epoch is over, the model parameters are updated according to the total reward received during the epoch. This means that Monte-Carlo methods can only be applied to problems with epochs of finite length.

**Temporal Difference.**

The agent is allowed to interact with the environment. Model parameters are updated according to the reward received during each transition between states. This allows for learning over the course of a single epoch, which can quicken convergence.

**N-Step.**

This term describes represents a middle-point between Monte-Carlo & Temporal-Difference methods. Updates are performed according to multiple transitions, but not the entire epoch. i.e. training on the first 50 steps of a 100-step epoch.

**Dynamic Programming.**

In contrast to the three prior strategies, in dynamic programming the agent does not interact with the environment. Instead data is gathered by iterating the state space (in an arbitrary order), using the transition model to explore all possible successor states, and updating the model parameters according to the reward obtained. DP requires a finite state and action space, as well as knowledge of the transition model, so it cannot be applied to many real-world problems.

There are a number of non-drl approaches; tabular methods, linear regression.

Tabular approaches achieve the desired input-output mapping by constructing a look-up table containing every possible input state, which is held in memory. Tabular methods are backed by theoretical proofs of their ability to converge to optimal policies [3, 4]. Despite this useful property, such methods are not typically practical due to the large state-spaces of real-world machine-learning problems. The associated table would be so large that storing it and updating it during training would be computationally infeasible.

Modern approaches typically use neural networks to approximate TODO. While perceptrons have been proven to converge to the optimal policy over time [5], neural networks in general are not guaranteed to converge, or even to improve. Though in practice they often do.

Methods based on gradient descent can find only the local optimum.

The use of neural networks with hidden layers to solve reinforcement learning problems is known as deep reinforcement learning, and in the last decade it has been applied to achieve human-level performance in complex control problems [6, 7].

Modern RL encompasses a diverse variety of approaches, and there are a variety of overlapping categories by which reinforcement learning algorithms can be classified. The taxonomy of those approaches is of interest if they are to be meaningfully compared.

**Entropy**

The entropy of a policy captures the variance in the actions it prescribes. The entropy of a policy w/r to a particular state is given by:

[1]

Where is the action space, and is the function that gives the probability of the network taking action in state . The policy entropy can be added to the reward at each step of the training process to produce a new target for optimization. Doing so can discourage models from reaching locally optimal policies that prescribe the same action in all, or nearly all states.

*Table 1: Overview of Targeted RL Methods*

|  |  |
| --- | --- |
| Class | Method |
| Q-Value | Deep Q-Learning. |
| SARSA |
| Pure Policy-Based | REINFORCE |
| REINFORCE-MENT |
| Actor-Critic | Basic Actor-Critic |
| Advantage Actor-Critic (A2C) |
| Proximal Policy Optimisation |

**Policy Gradient Methods.**

The network’s weights are updated according to a rule.

Where Â is an approximation of the advantage, , and is the function that gives the probability of the network taking action in state .

If the advantage is positive, it is desirable to reinforce this behaviour, so the weights are updated to maximize the probability mass function, making the action more likely to be taken in the future. If it’s negative, they are updated in the opposite direction.

**REINFORCE.**

This is a policy-based Monte-Carlo method, one of the earliest policy-based methods to be discussed. At the end of each epoch, the network parameters are updated according to the rule.

[2]

Where and are hyperparameters of the network, is the length of the epoch, and is the cumulative reward received during the epoch, which serves as an estimate of the action advantage.

**REINFORCE-MENT**

This is a variant of REINFORCE. Instead of maximising reward, it targets the sum of reward & policy entropy, discouraging the adoption of low entropy policies.

**SARSA & Q-Learning**

These are closely related temporal-difference methods.

A transition is a tuple

The network outputs an estimate of the future reward If this prediction is accurate, it should be exactly equal to the value of the successor state, plus the observed reward at t. We have access to this observed reward, as well as the network’s prediction of the value of the successor state We can then calculate the squared error and use it as the target of gradient descent.

**Q-Learning Loss Function**

**SARSA Loss Function**

The methods differ in that SARSA is on-policy, whereas Q-learning is off-policy. That is, under SARSA, is the action prescribed by the policy that produced the transition. Whereas under Q-Learning is the action with the highest Q-value (following a greedy policy with respect to ). Even if these policies are initially identical, they will diverge as weight updates are made.

**Actor-Critic Methods.**

The key feature of actor-critic methods is their use of two networks; the actor & the critic. The actor learns in the same manner as policy-gradient methods, however Â is calculated using the output of the critic.

The critic learns a value function using the same method as the value-based techniques discussed earlier. Note that as the transitions are being produced according to the policy of the actor, critic learning is necessarily off-policy.

There are multiple ways to produce Â, and these methods are described in more detail in the three implementations below. The actor & critic can be two separate networks, or a single network. In the latter, the network has outputs for both action probabilities and the critic outputs.

**Basic Actor-Critic.**

In this algorithm the critic learns to approximate . The critic’s estimate of the Q-value of the actor’s action is normalised and used as Â. This algorithm includes an entropy term.

**Advantage Actor-Critic (A2C).**

This is an n-step algorithm.

This is a synchronous version of Asynchronous Advantage Actor-Critic, from [4] (algorithm 3).

The critic learns to approximate , and the estimate is used to calculate Â according to the equation:

This algorithm includes an entropy term.

**PPO**

Similar to advantage A2C, however the gradient updates are limited within a range. Loss function:

This algorithm includes an entropy term.

REFERENCES

# References

|  |  |
| --- | --- |
| [1] | R. J. Williams and J. Peng, “Function Optimization Using Connectionist Reinforcement Learning Algorithms,” *Connection Science,* no. 3, pp. 241-268, 1991. |
| [2] | R. J. Williams, “Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning,” *Machine Learning,* vol. 8, p. 229–256, 1992. |
| [3] | V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” *arXiv preprint arXiv:1312.5602,* 2013. |
| [4] | A. Novikoff, “On Convergence Proofs For Perceptrons,” *Proceedings of the Symposium on the Mathematical Theory of Automata,* vol. 12, no. 1, pp. 615-622, 1963. |
| [5] | P. Dayan and C. J. Watkins, “Q-Learning,” *Machine Learning,* vol. 8, pp. 279-292, 1992. |
| [6] | M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar and D. Silver, “Rainbow: Combining improvements in deep reinforcement learning,” *Proceedings of the AAAI conference on artificial intelligence,* vol. 32, no. 1, 2017. |
| [7] | V. Konda and J. Tsitsiklis, “Actor-Critic Algorithms,” *Advances in neural information processing systems,* vol. 12, 1999. |
| [8] | R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, Second Edition, MIT Press, 2018. |
| [9] | M. Volodymyr, A. P. Badia, M. Mirza, A. Graves, T. Harley, L. P. Timothy, S. David and K. Kavukcuoglu, “Asynchronous Methods for Deep Reinforcement Learning,” *arXiv preprint arXiv:1602.01783,* 2016. |