Reinforcement Learning for Model-Free Policy Selection

We consider the standard reinforcement learning setting in which an agent interacts with an environment over a discrete number of timesteps. At each timestep , the agent receives an observation of its current state and selects an action from some set of possible actions according to its policy , where is a mapping from states to actions. The agent moves to the next state and receives reward . The central problem across many kinds of reinforcement learning (RL) problems is how to take a set of *observations* and use them to choose the optimal *action*. At first glance, this may seem identical to the challenge of traditional machine learning, with the goal of fitting a function to the input data in order to minimize error. There are indeed many similarities to traditional "curve fitting", but there is one key feature which sets the RL task apart - time. When an RL agent takes an action, the action's value may not be immediately apparent; instead, the agent's success depends on the many future actions it will take until the agent reaches a terminal state, or the process restarts. A simple regression technique may choose a greedy strategy which maximizes reward one step at a time with no thought for the future, or even fail to converge if the reward is not obtained until the end of the task, leaving the agent with no guidance on the value of its individual actions. For some problems, this obstacle can be overcome by simply modifying the perceived reward. Consider a soccer robot, which instead of just tracking goals scored, might be programmed to understand the laws of physics and simple rules such as "kicking the ball straight to an opponent is bad" or "controlling the ball near the goal is good." However, this approach requires sophisticated domain knowledge and must be redesigned for every new problem. A general learning agent must teach itself to recognize the value of *states*, in terms of expected total future reward, and this requires increased complexity.

**Q-learning:** Q-learning, first introduced in 1989(Watkins 1989), is designed to learn the best *policy*, π, which maps every possible state to the best action. This mapping assumes that the environment is Markovian, meaning that the outcome of an action depends only on the current state and the action itself, with perhaps some stochastic noise. For example, it is not sufficient to define the state of a Q-learning self-driving car as its current position plus the position of the surrounding objects; the outcome is strongly dependent on velocity and acceleration also, and these must be included in the state explicitly to eliminate the history-dependence and approximate a Markovian process. The policy is used to choose the action *a* to move from a state *x* to a future state *y*. *A* is chosen to have the maximum action-value *Q* under a policy *π*, given as follows (Watkins and Dayan 1992):



Rx(a) is the immediate reward for action *a*, while γ <= 1 is a discounting factor for future rewards, useful because a future reward is less certain to occur than an immediate reward. Pxy and the summation over possible outcomes *y* are only necessary if the system is probabilistic instead of deterministic. Finally, *V*π(y) is the state-value discussed in the introduction, and is the key to the system of learning. *V* is defined recursively, with the value of a state equal to the discounted value of the next state plus the reward gained by changing states. This may seem pointless; how can we determine the state value if doing so requires calculating another state value, repeated infinitely into the future? This is why the agent must be trained. The agent starts with an initial (very inaccurate) value of *V* for every possible state, and acts to maximize *Q*, and every time it gains a reward it updates the value of the previous states to reflect their outcome. Knowing the value for each individual state reduces the problem to a traditional machine learning problem, where the actual controller no longer needs to consider further than one timestep into the future, and it is “fit” to locally optimize *Q* one step at a time.

Although the previous algorithm works if the agent has explored every possible state, this leads to very slow training and we would like to estimate *V* for states not yet visited. One way to estimate *V* is a neural network with weights *θ*, and a loss function which is the squared error between the predicted and actual value of a state. The Q-network can be updated by backpropagation with stochastic gradient descent. One recent modification of the *Q*-value neural network is the Deep Q-Network (DQN)(Mnih et al 2013), which stores a long sequence of past experience in memory and trains the Q-network on randomly selected passages from that memory, re-using each state many times to increase efficiency. The DQN takes the state as an input, and has an output node for every possible action giving the value *Q* for that action, which may require breaking a continuous action space into discrete intervals. Mnih et al. successfully applied a DQN to a variety of different Atari arcade games, and the number of output nodes was the only change needed to the architecture when transferring from one game to the next. After training for 10 million frames for each game (approximately 46 hours at regular game speed) the model was able to outperform human players in simple games such as Pong, and the DQN performed better than other models for complex games such as Space Invaders even though it could not match human performance.