Notes on learning in games[1]

Jonathan Shapiro

October 2021

[1]Version 1.1

Preface

There are many equations in this topic. You may wonder, do I need to memorize equations. The answer is, No! My philosophy is, you need to know when an equation is relevant, what it means, perhaps some properties of it, but not the details of the equation. In life, we would look these things up.

Take as an example the Q-learning equation in Chapter 3. Don't memorize the equation. But understand what it converges to, i.e. what it is trying to learn. You might want to use these in your lab project.

Chapter 1

Introduction

1.1 Introduction — why use learning

We have seen that in two-player, zero-sum games with perfect information and no chance, very effective game-playing agents can be produced using mini-max search with pruning, and perhaps a database of openings and end games. This has worked in chess, checkers, and many other games, and does not require any machine learning. Why do we need to consider learning in games?

In a game which does not have too large a branching factor, and when good heuristics can be found in a two-player, zero-sum game, minimax search with pruning is almost always the best choice. However, in some games good heuristics cannot be found. Go is a well-known example. In addition, in games with chance and incomplete information, like poker, non-zero sum games, multi-player games, and other complex games, approaches in which agents are played against each other multiple times in a process in which they explore the game tree looking for high-value nodes to exploit, can yield the best results.

In games with hidden information, such as poker, Nash strategies potentially involve probabilistic choices of child nodes at each move, so the minimax solution, which is a pure (deterministic) strategy will not help here. Currently, the best performing approach to learning in poker is counterfactual regret minimisation, which involves learning.

The learning algorithms which have been effective in game learning are from the class of learning methods called "Reinforcement Learning" or RL.

Game learning is an example of reinforcement learning. Reinforcement learning is a machine learning situation in which the feedback is a reward. A reinforcement learning agent must explore different actions or sequences of actions in order discover good ones to exploit. An example is an agent learning to navigate a maze with a GPS but no map. The agent has to make a choice at each junction which way to go. The agent must learn by trial and error which action to take at each junction to find the way to the goal. In the language of reinforcement learning, the agent must find the value $Q(s, a)$ of the possible actions a at the current state s (the junction, which the GPS reveals). The value $Q(s, a)$ could be the estimated distance to the goal, or the likelihood that this path leads to the goal at all. In early stages of learning, the paths found could be very long, so exploration needs to continue even as the acquired knowledge is being used. In general, the value will be the reward in situations with no randomness, or the *expected* reward. The expected reward is the average reward in a huge (conceptually infinite) number of trials.

Another example of a reinforcement learning problem is a game. The agent has to learn which paths through the game tree lead to a win or a large positive reward. I.e. at each decision point in a game tree, the agent needs to learn which action will lead to a good outcome sometime in the future.

Two key ideas from reinforcement learning are important in understanding learning in games:

1. The exploration-exploitation trade-off.

2. Learning to take actions based on their expected *future* payoffs.

These can be illustrated in the maze-learning example above. Exploration is needed to discover a path which leads to the goal. Exploration continues to be needed even after a path to the goal has been found because shorter paths might be discovered. When the agent arrives at a junction and needs to choose an action, it is a good move if it leads to the goal in the future if not immediately. The exploitation strategy is needed to build up the knowledge about decisions which lead to high rewards in a short amount of time.

A third key idea, specific to learning in games, is that of *"self-play"*. The learning agent will learn to play the game by playing against a similar learning agent or itself many, many times. In many cases, there is a separation between the learning phase, where self-play is used, and the phase in which the agent is trained and ready to play the game against humans or other agents.

The focus of these notes is the use of machine learning, and particularly reinforcement learning to play two-player extensive-form, turn-based games. There are other more game theoretic and less computational approaches. See "The theory of learning in games" by Fudenberg, Drew and Levine, David 1998. This will not be covered in this course.

Chapter 2

Introduction to reinforcement learning

Reinforcement learning (RL) is a type of machine learning in which the learning agent must learn from rewards, which might be negative or positive, what the best actions are in a given situation. An obvious example of learning from rewards is a pet dog who learns not climb up on the table and eat the family dinner, but learns to come when called, from the negative and positive rewards received. Any learning scenario in which the learning agent must discover by trial and error the best action to take to receive the best rewards, is a suitable candidate for the use of reinforcement learning.

If you took the Machine Learning course in the second year, you learned about "supervised learning". In this learning paradigm, there are examples which are inputs labeled with the desired output. The goal of supervised learning is to learn to give the appropriate output on inputs which have not been seen before. This is sometimes called "learning from a teacher"; the teacher being the labelled examples. In reinforcement learning, there are no training examples. The best responses must be learned by the learning agent. This is usually carried out in real time.

Ex 2.0.1. Consider a professor who bikes to and from work every day. She chooses from three routes: Route A is down a busy road, but is the shortest and one can cycle fast. But, if there is a lot of traffic, it can be very slow. Route B is on bike paths. But it runs through 4 school zones, where one must cycle slowly to avoid all the kiddies and parents. Route C is on back roads with little traffic, but three very busy roads must be crossed without crosswalks, and it can take some time to find a gap to cross through. Every time the professor wants to go to work, she wants to take the fastest route.

The professor could learn the best route using reinforcement learning. Which route is the best will depend on the context — time of day, workday/non-workday, and is school in session. The context we will call *the state.* The choice of route for each context we will call *the action.* The reward will be the travel time which we want to minimize. Or, to be more consistent with RL ideas, the reward could be minus the travel time which we want to maximize. Travel time is not a reward; it is more like a cost. Note that the travel time will not be the same even with the same route (action) and the same state (context), because the density of human and vehicular traffic is stochastic. It has some randomness. So, to learn the best route for each context, the professor will have try different routes in different contexts and observe average journey times for each route-context pair (state-action pair). As a best route starts to emerge for each state, the professor should use that route more often in that state, but still she should occasionally try the other routes to see if they continue to be longer than the current best. Circumstances might have changed regarding these routes, or there might have been a run of good luck which led to the belief that a given route was the best. Shortly we will see a learning algorithm which makes this formal.

2.1 Concepts and notations

In its simplest form, reinforcement learning describes the interaction between an agent and an environment. The agent is some state at time-step t. Based on the state, The agent chooses an action and takes that action. The environment updates the state based on the action, and returns a reward or no reward based on the action and the new state. The reward is a number, where positive is a good reward and negative is a bad reward.

The method the agent using to choose the action is called its *policy.* It is usually denoted $\pi(A_t|S_t)$. This is a conditional probability. The state at time t, S_t is known by the agent, and the action is chosen probabilistically conditioned on the state. It could be chosen deterministically once the best action is known, but during learning is is usually chosen based on a probability function, as we shall see. The action will have two effects. First it will change the state to a new state, and second it might result in a reward. These come from the environment. So, associated with the environment is a transition probability, which takes the current state and action, (S_t, A_t), and returns the new state, S_{t+1}. We could call this function $\mathrm{Env}\,(S_{t+1}|S_t, A_t)$. This gives the next state from the current state and the current action. It could be probabilistic or deterministic. There is also a reward function, $R(r_t|S_t, A_t)$, which is also associated with the environment. This returns the reward for a given state-action pair. This could also be stochastic or deterministic.

A goal of reinforcement learning is to find a policy, $\pi(A_t|S_t)$, for every possible state and action from the state, which optimizes the reward, where the reward function is not known. (When the reward function is known, the problem is called a Markov Decision Process MDP. MDP is **not** a topic of this course.) The optimal policy is often denoted π^*.

Because the reward function is not known it must be discovered by trial and error. In other words, there must be *exploration* of the state space and the space of actions from each state. As properties of the state-action-reward relations are discovered and more is known about good and bad actions from each state, that knowledge should be used to get better rewards. Even as the algorithm continues to explore, it should spend increasing time performing actions in states which have been found to yield high rewards. Performing the actions which based on current knowledge give the best rewards is called *exploitation.* This trade-off between exploration and exploitation is at the heart of reinforcement learning. Every reinforcement learning algorithm will need mechanisms to achieve those two goals. Exploration — to discover actions which yield high

Game theory	Reinforcement learning
Strategy	Policy
Payoff	Reward
Node (or board position)	State
Heuristic (evaluation)	Function approximatation

Table 2.1: Terms used in game theory and reinforcement learning to describe similar concepts.

rewards. Exploitation — to benefit from this knowledge. Reinforcement learning uses different terms than those used in game theory, the concepts are very similar. A concordance is shown in Table 2.1.

2.2 Epsilon-greedy search strategy

The most basic exploration-exploitation search strategy is called *epsilon-greedy.* It is actually quite effective, and it is often the default approach. The *epsilon-greedy* policy is as follows:

$$\text{selected action} = \begin{cases} \text{current best action;} & \text{with probability } 1-\epsilon \\ \text{random action;} & \text{with probability } \epsilon. \end{cases}$$

where ϵ is a small number between 0 and 1. In words, the best action according to current knowledge is used most of the time, but occasionally a random action is taken. The value of ϵ has to be chosen, and there is no systematic way to do that, unfortunately.

Of course, early during learning the knowledge about what is the best action could be pretty poor, Whereas, after much learning and exploration, the information could be sufficient to give a good idea of what the best actions are. Thus, it is often beneficial to start with a lot of exploration, but decrease it over time. There are theoretical reasons to like the inverse iteration, which is a very slow. If ϵ is decrease too fast, e.g. exponentially, it will essentially stop exploring after a finite time. Whereas, having ϵ decrease inversely in time, exploration continues, just less and less and less. Two approaches you could use are:

$$\epsilon_t = \epsilon_0/t; \tag{2.1}$$
$$t = t+1, \tag{2.2}$$

where ϵ_0 could be 1 for example. So, first you explore all the time, then half the time and so forth. Alternatively, you could use the following,

$$\epsilon_t = \min(T/t, 1); \tag{2.3}$$
$$t = t+1, \tag{2.4}$$

where T is the number of iterations when the algorithm will explore only. Once t gets bigger than T, exploration will start to reduce.

Note that decreasing ϵ is only a good idea when the environment is static and not changing. If the environment is changing, just keep ϵ fixed.

2.3 Immediate reward reinforcement learning

The simplest case of RL is when the reward is immediate after the action. You are in a state, you take an action, you receive a reward, and then you update your knowledge about the relationship between the state, action and the reward. The knowledge update is related to the difference between what you predict the reward to be, and what it actually is.

2.3.1 Tabular learning

If the number of states and actions is not too large, a look-up table can be used as a learning model. The cells of the table will be indexed by the state-action pair, and will contain the current estimate of expected reward. We will call this table Q-table (but this is not Q-learning yet). If S is a state, and A is an action, then $Q(S, A)$ is where the current estimate of the expected reward for taking action A from state S. The procedure is

1. Observe state S.
2. Choose action A, using an exploring-exploiting strategy such as epsilon-greedy.' Observer reward R.
3. Update that particular cell in the Q-table using the following equation,

$$Q(S, A) \leftarrow Q(S, A) + \alpha \times [R - Q(S, A)] \,. \tag{2.5}$$

   where α is a small 'learning rate' (e.g.0.1), and the left arrow means 'is replaced by'.

In the above, the left arrow ($\leftarrow$) means 'is replaced by'.

A few points to note:

1. The learning couples to the difference between the predicted reward, $Q(S, A)$ and observed reward R.
2. If the prediction is too low it will increase, and if it is too high it will decrease.
3. The prediction of R is the maximum is the value of Q where the maximising is over the chosen action. I.e. the prediction is $\arg\max_{a'} Q(S, a')$.

2.3.2 A trivial example

Suppose we have only one state and three actions. So, the table will only one row and three columns, $Q(a_0)$, $Q(a_1)$, $Q(a_2)$. The three actions provide the following rewards: $a_0 = 1.0$, $a_1 = 1.5$, and $a_2 = 0.0$. I use ϵ=α = 0.1.

Figure 2.1 shows the Q-values during the first 100 iterations of learning. The value for action 0 is close to the true value, namely 1. However the value for action 1 is too low. Remember, this only learns about actions it takes. As the value of action 1 is currently less than that of action 0, epsilon-greedy will explore action 0 more often.

Figure 2.2 shows the Q-values during the first 1000 iterations of learning. All actions are now close to their correct values.

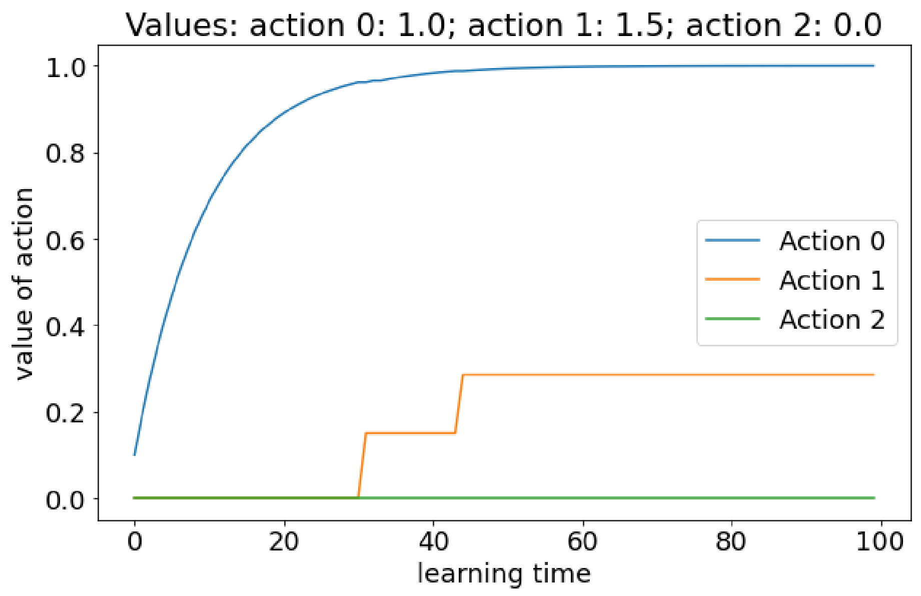

Figure 2.1: The three values of the actions during learning. We can see that the algorithm has not learned yet that action 1 is the best action

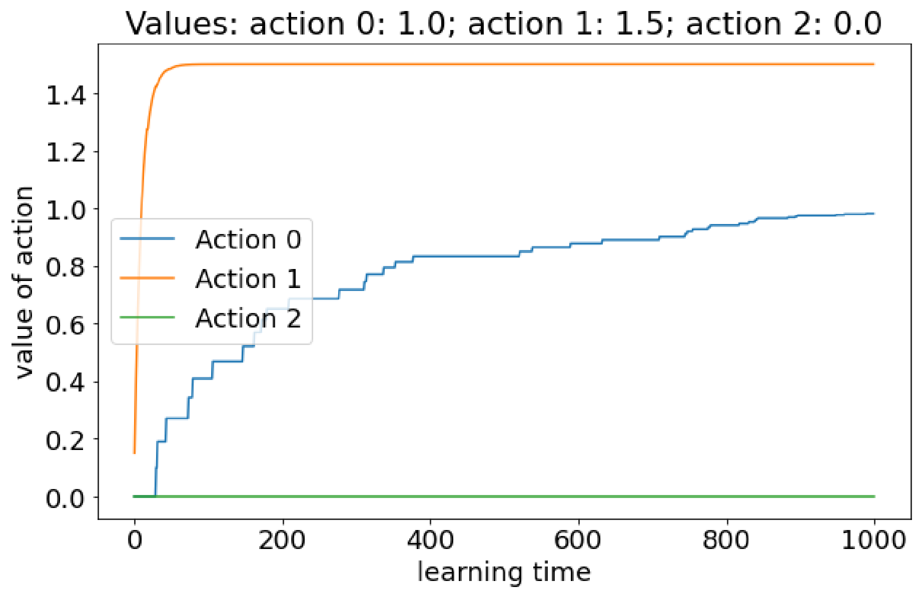

Figure 2.2: After a long learning time, the values are correctly learned. Notice that sub-optimal actions are learned slowly, because they are chosen less often.

Chapter 3

Delayed-reward reinforcement learning

In the immediate reward case, after an action is taken from a particular state the reward is immediate. So the reward for taking that action from that state is obvious. It should be the reward, or the expected (average) reward received when that action is taken from that state. It may take some time to learn the relationship between state-action and reward, but we know what the target of learning should be. However, immediate-reward reinforcement learning is not relevant to game learning, except in trivial toy cases.

Consider now the case where a sequence of actions must be taken before a reward is received. There are many examples of this situation. Most board games are in this category. From any point in the game, one has to pass through a number of board positions before you reach the result - a win or a loss. This situation is called "delayed-reward reinforcement learning".

The difficulty is, how to assign a value to the intermediate states. Perhaps you made one very bad move which caused you to lose the game. How would you identify this as the bad as compared to all the other moves you made in the losing game? This is sometimes called the *credit assignment problem*; how is credit (or blame) assigned within a sequence of actions? Or put another way, how should the reward be distributed to all the states and actions that led to that reward?

For state-actions which get no reward, you might be tempted to assign a value zero. Assign the (average) reward received for every state and action, so no reward (which is a reward of zero) should be assigned a value zero. But then almost every state-action has the same value, 0. What is needed is a method to assign higher values to intermediate states and actions which lead to final states with high rewards, and lower or negative values to states and actions which lead to low or negative rewards. One way to do this is the topic of this chapter.

There are many other examples of delayed-reward reinforcement learning besides games. Perhaps a robot has the task to move its gripper to a bottle, pick up the bottle and hand it to you. The robot has to carry out a sequence of actions to carry out this task. How does it learn to do this?

3.0.1 Value learning in delayed-reward RL

If the world is deterministic - i.e. every time an action is taken from a state, the next state is the same and the reward is the same - the value of a state-action pair is either

1. The accumulated future rewards; or
2. the *discounted* accumulated future rewards.

In most cases, the world will not be deterministic, so you will use the *expected* values of these quantities (i.e the long-term average). Obviously the future rewards from taking an action from a state depends on the policy used throughout the future sequence of moves.

To keep the notation simple, I will consider the value of a state, s using policy π, denoted $V_\pi(s)$. A sensible policy would be — from the given state, try every allowed action, observe the new state that that action takes you to, and choose the state with the highest value[1]. This is called *the greedy policy.* The values we want our learning method to produce are,

accumulated future reward:

$$V_\pi(s_t) = r_t + r_{t+1} \ldots + r_T; \text{ where } T \text{ is the end of the sequence/end of the game.}$$

So, the value of the state the system is in at time t, is the immediate reward r_t plus all the rewards received in the future until the end of the sequence, episode, game, etc using the policy π.

It is more common to use the *discounted* future reward, which is

discounted future reward:

$$V_\pi(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \ldots + \gamma^T r_T; \; T \text{ is the end of the sequence/game.} \tag{3.1}$$

Here γ is a number between 0 and 1 and determines how much the future is discounted. If $\gamma = 1$ it is the same as the accumulated future reward. If $\gamma = 0$ then only the immediate reward matters. For $0 < \gamma < 1$ then rewards attained sooner are preferred over rewards achieved later.

You might wonder, why do we discount the future? There are several reasons for this. First, it values shorter sequences over longer sequences, so shorter games over longer games with the same outcomes. Second, predictions longer into the future are more likely to be in error than shorter-term predictions. Also, you can think of this financially. One hundred pounds today is worth more than one hundred pounds in six months, because you could have invested the hundred pounds for six months.

Why this form of discounted? This is somewhat off-topic but is an interesting psychological question. The future is being discounted exponentially fast using the formula above. There has been work to determine the discounting function used by humans and other animals for some time. Do we discount the future more slowly, and should we (an AI question). However, from the perspective of this course, this form of discounting is mathematically useful, as you will see shortly.

[1]This was the policy used in TD-gammon.

3.0.2 But how do we know the future rewards?

Unlike the immediate reward, the future reward is not observed. So how can an algorithm learn to produce it? The answer is to use a trick. Consider the state-value formula Equation (3.1). Assume that the value of $V_\pi(s)$ has already been learned for all states in the environment. Then, the following would be true,

$$
\begin{aligned}
V_\pi(s_t) &= r_t + r_{t+1} \ldots + r_T; && (3.2)\\
&= \sum_{t'=t}^{T} r_{t'} \gamma^{t'-t}; \text{ where } t \text{ is now; } t' \text{ sums into the future} && (3.3)\\
&= r_t + \gamma \sum_{t'=t+1}^{T} r_{t'} \gamma^{t'-(t+1)}; && (3.4)\\
&= r_t + \gamma V_\pi(s_{t+1}). && (3.5)
\end{aligned}
$$

So to learn the value of $V_\pi(s)$ we learn to make Equation (3.5) true. In other words, we use the right hand side of this equation as an estimate of the future reward and use that as the target for $V_\pi(s)$ to learn towards.

3.1 Tabular learning

For small systems, the data structure can be a lookup table. We start with an approach called "state-value learning" in strives to learn the value of every state. I am going to assume that the situation is "episodic". This means the system runs from the beginning to the end of the episode and then it stops. In game-playing, one play of the game is an episode. Another example is a search for a goal. When the goal is found and a reward is achieved, the episode ends. It takes many episodes to learn the state-value function.

For state-value learning, maintain a table large enough to hold every state in the system, initialized to zero for each state. Initialize t to 1 and the state to the initial state s_1. Then the following inner loop over an entire game or episode, over many games or episodes. At the end of each episode or game, reinitialize the state to the initial state and t to 1. *But do not reinitialize V, which continues learning over multiple episodes or games.*

1. Observe the current state s_t.

2. If the current state is a terminal state, BREAK.

3. Choose the action using a policy derived from π, e.g. epsilon-greedy.

4. Observe the new state s_{t+1}.

5. Observe any reward associated with the new state r_t.

6. Update the value of the current state in the table according to

$$V_\pi(s_t) \leftarrow V_\pi(s_t) + \alpha \left\{ r_t + \gamma V_\pi(s_{t+1}) - V_\pi(s_t) \right\}. \tag{3.6}$$

To be more explicit about the policy, V_π, it is as follows

1. Let $\mathcal{A}_t$ be the set of allowed actions from state s_t.
2. With probability ϵ choose a random action from $\mathcal{A}_t$.
3. Otherwise, choose the action which results in a state, s_{t+1} with the highest value of V.

3.1.1 Convergence of tabular state-value learning

If the learning algorithm is run long enough to that every state is visited an infinite number of times, the value of each state will be reward received from that state following the greedy policy, discounted by the number of states it must pass through to achieve that reward, $r \times \gamma^p$ where p is the number of states between the current node and the reward.

Example as a simple example, consider the graph in Figure 3.1. The value of the V-table *after learning* is shown in the following table.

Node	Value
1	20γ
2	10
3	20

Nodes 2 and 3 get immediate rewards, so the value is $\gamma^0 = 1$ times the reward, but node 1 has to pass through one node to get the reward so the value is γ^1 times the reward. Following the greedy policy, the V-learner chooses the child with the highest value of V which is node 3.

However, it is not necessary to run to near convergence. The states leading to low rewards or negative rewards are visited less often than those leading to high rewards, so low-value states will converge more slowly that high-value states. But who cares? Once high-value states are explored sufficiently, learning can stop. In a game setting, once states are found that lead to wins or strong play in many situations, learning can stop.

3.2 Tabular Q-learning

Tabular Q-learning is very similar to tabular V-learning discussed in the previous section. Except here, states *and* actions are used. Required is a table large enough to hold every state and every action from that state, $Q(s, a)$. The goal is to learn the best action from every given state. The greedy policy will continue to be used. The learning procedure is as follows

1. Observe current state s_t.
2. If current state is a terminal state, BREAK.
3. Using a policy derived from π (e.g. epsilon-greedy) choose the action a_t.
4. Take that action and observe the new state s_{t+1}.
5. Update the Q-table using the following equation.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left\{ r_t + \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right\}. \tag{3.7}$$

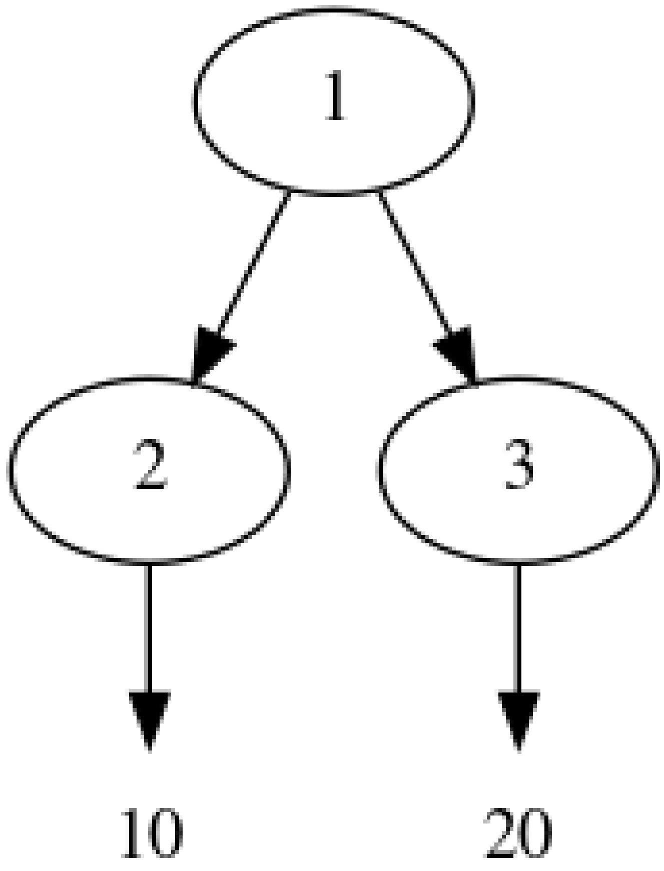

Figure 3.1: Simple example of tabular state-value learning. The system starts in state 1, and searches for the best reward. After learning, the value of node 2 is 10; the value of node 3 is 20; the value of node 1 is 20γ.

This loop is repeated over multiple episodes (many, many, many).

Notice that in both tabular V-learning and tabular Q-learning the learning couples to values at two different times, t and $t + 1$. For this reason, the methods are example of what is often called *temporal-difference* learning[2], or TD-learning.

3.3 Function approximation

One problem with tabular learning is that every state (V-learning) or state-action pair (Q-learning) must be visited many times. If the state-space is large, this is a problem for two reasons. First, obviously, it will require a lot of memory

[2]'Temporal' means pertaining to time

to store the table. Second, and perhaps more important, the fact that each state must be visited many times means that learning will be very slow.

The solution to this is to use a machine-learning, supervised learning model to represent the relationship between the states and actions, and the expected (discounted) future rewards. This machine learning model must be capable of producing a continuous output, so must be a regression model, not a classification model.

For state-value learning, the input to learning model would be some representation of the current state. The output of the learning model $\tilde{V}(s_t)$. (The tilde above the V is meant to represent that it is an approximation to the true V value.) The output should be the value of the current state give that a greedy policy is going to be used from here on. The learning algorithm is:

1. Observe the current state s_t.
2. Using each possible action from s_t, calculate s^*_{t+1} the next state which gives the highest value of $\tilde{V}(s*_{t+1}$.
3. Then apply learning to minimize the loss function below.

$$\left\{ r_t + \gamma \tilde{V}_{\pi_G}(s^*_{t+1}) - \tilde{V}_\pi(s_t) \right\}^2 . \tag{3.8}$$

In this equation, Eq (3.8), $\tilde{V}_\pi(s_t)$ is the output of the learning model, and the rest, is the target it is learning towards.

Function approximation in Q-learning is very similar. The learning model can either take as input a representation of the state and the action, and produce as a single output the expected discounted future reward from that state action pair. Or it can take a representation of the state, and have an output for each possible action as the expected discounted future reward when that action is taken. At each point in the sequence, the goal of learning is to minimize the difference between the output of the Q-learning network, $\tilde{Q}(s_t, a_t)$ and the desired output which is

$$\text{desired output } = r_t + \gamma \max_{a'} \tilde{Q}(s_{t+1}, a'),$$

using the squared difference as in Eq (3.8).

Important points when used in games

When using V-learning or Q-learning it games, it is important to think of the opponent(s) as part of the environment. So, if you are in state s_t at time t, the next state s_{t+1} is the state *after* the opponent(s) have moved. In particular, if an opponent has won the game, you must take that as a negative reinforcement signal associated with the move made just before the loss.

TD-lambda learning

There is another problem with the learning methods described above. The learning is slow. The reason for this is that the learning slowly moves backwards from the rewards. To see this, imagine doing V-learning from the start. Assume all Vs are initialized to 0. Until the first reward is reached, nothing happens.

The learning equations are just zero minus zero. Once a reward is reached, the state or the action just before that is updated. But still most of the values are zero, until the state or action just before the one just before the reward is reached. And so, the location of rewards is slowing moving backwards through the graph, back one step for each episode.

To speed this up, assume that every move in the sequence leading to a reward contributed to that reward, but by a decreasing amount. The amount it decreases is controlled by a quantity λ (lambda) which is a factor between 0 and 1. So the move just before the reward is updated with the learning rate times the reward, αr, as before. The move before that is updated by $\lambda \times \gamma \times \alpha r$, and the one before that by $(\lambda \times \gamma)^2 \times \alpha r$, the one before that by $(\lambda \times \gamma)^3 \times \alpha r$, and so on. This can make learning much faster. Although it does not appear to be used so much any more.

Implementing this is usually described using something called eligibility traces. However, it is much more efficient to simply remember the sequence for each episode. As you move from state s_t to s_{t+1}, provide the new state with a link to the previous state. When you get to the reward, you add αr to the current state, as before. Then work back through the sequence, multipling the update by the factor $\lambda\gamma$ and removing the link until you get to the start. Repeat.

3.4 TD-Gammon

TD(λ) learning has been used to produce a very good backgammon playing program (Tesauro, 1994). See Figure 3.2. ***It produces the board evaluation function,*** $V(s)$***.*** It uses state-value or V-learning described above, and also used TD-lambda learning. It consists of two parts:

Predictor Network: A multi-layer perceptron is used to predict the outcome of the game from the current board position (originally coded as raw board position; later hand-crafted features used). Shown in Figure 3.3.

Controller: A program generates all legal moves from the current position. Predictor network scores them; that with the highest predicted outcome is the moved used.

Tesuaro argued that the use of the controller was very important. The system should not waste learning what moves are legal. If it did try to learn what moves are legal, it might try to make moves which are almost legal. The controller allows the predictor to consider only legal moves, and focus its learning resources on trying to find good moves.

Data consists of sequences of moves from the standard start position to a result. Self-play was used - program plays another version of itself over and over. Network starts from random state (no knowledge of game). Initially, games are incredibly long (100's to 1000's of moves versus $50 - 60$ moves for typical human game).Later, basic strategies emerge: hitting opponents, building points, playing safe, ...

Claimed results:

- TD-Gammon 1.0 Contained 40 hidden units, trained for 200,000 games.

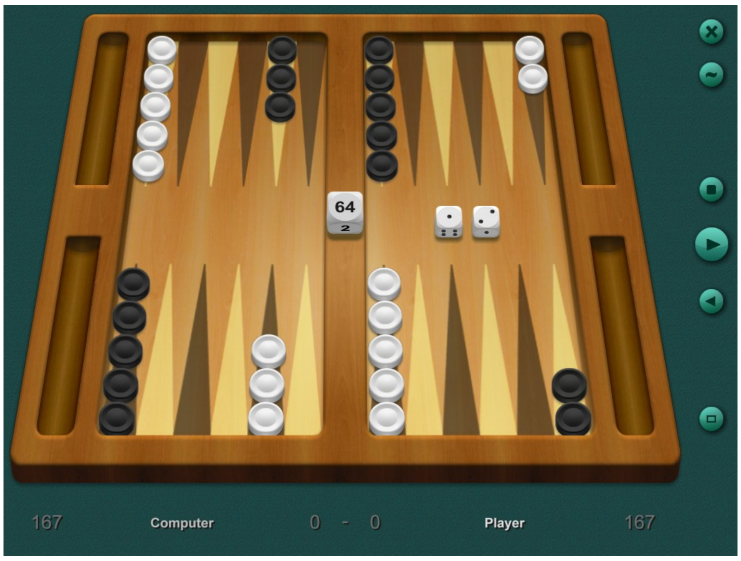

Figure 3.2: Standard backgammon board. It uses dice. It is a two-player, zero-sum game with chance.

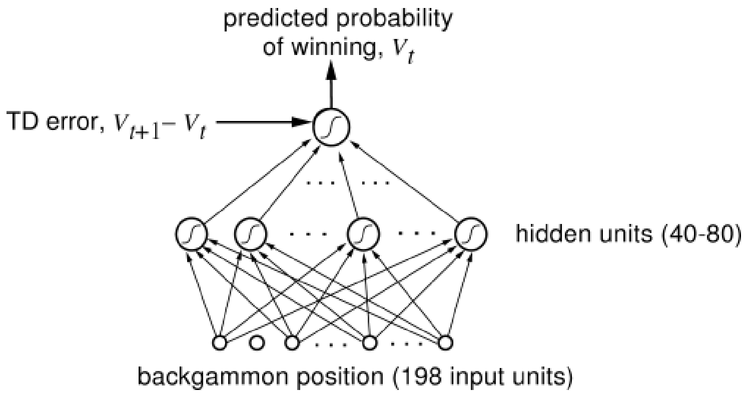
predicted probability of winning, V_t
TD error, $V_{t+1} - V_t$
hidden units (40-80)
backgammon position (198 input units)

Figure 3.3: The predictor network. This is the function approximator for state-value learning using TD-lambda learning.

Plays at strong intermediate level; good enough to win regional tournaments.

- TD-Gammon 2.1 Close to the world's best player.
- Has changed how humans play certain board positions (discovered new strategies which appear to be better).

A quote from Kit Woolsey, perennial world top 10 backgammon player (ranked #3 when this was written), quoted in Tesauro, 1995.

> "TD-Gammon has definitely come into its own. There is no question in my mind that its positional judgement is far better than mine. Only on small technical areas can I claim a definite advantage over it .... I find a comparison of TD-Gammon and the high-level chess computers fascinating. The chess computers are tremendous in tactical positions where variations can be calculated out. Their weakness is in vague positional games, where it is not obvious what is going on .... TD-Gammon is just the opposite. Its strength is in the vague positional battles where judgement, not calculation , is the key. There, it has a definite edge over humans .... In particular, its judgement on bold vs. safe play decisions, which is what backgammon really is all about, is nothing short of phenomenal .... Instead of a dumb machine which can calculate things much faster than humans such as the chess playing computers, you have built a smart machine which learns from experience pretty much the same way that humans do"

(Jon says: a bit of anthropomorphism here on the part of Kit.)

Why did TD-gammon work? One thought is that TD-gammon learned via self-play. Pollack and Blair (1997) argue that this was key, by showing a simpler model learned effectively in self-play. The randomness in the game (the dice) means that exploration happens automatically.

A second reason might be the use of a clever board representation — use of hand-crafted features specific to backgammon improved performance considerably.