# Implementing the Actor-Critic Model of Reinforcement Learning

## 1 Introduction

Reinforcement Learning (RL) consists of a diverse collection of methods, several of which have driven major breakthroughs in Artificial Intelligence (AI).[1] Most recently, Google Deepmind's AlphaGo system (Silver et. al., 2016) and its extensions defeated the world's top Go players using a combination of RL, Monte Carlo Tree Search (MCTS) and Deep Neural Networks (DNNs). That system's motivation came from earlier work by the same group (Mnih et. al., 2015) that merged RL and DNNs to play a wide variety of Atari games at human levels (and beyond). The methodological precursor to that work came much earlier, in 1995, when Gerald Tesauro's Backgammon player achieved human-level competence via the interaction of Temporal Differencing (TD, a form of RL) with (simple, shallow) neural networks.

This document explains temporal differencing and several of its incarnations, including the Actor-Critic Model (ACM), with the aim of providing enough details and insights for implenting a general-purpose ACM: one capable of solving different types of search problems.

Although the actor-critic method can be summarized by a few simple equations and lines of pseudocode, a proper, general, implementation of ACM requires significant programming. Don't be fooled by the elegance of the basic model.

## 2 Temporal Difference (TD) Learning

In all forms of Reinforcement Learning (as in many other AI algorithms), the numeric value (a.k.a. *evaluation*) associated with each search state plays a pivotal role in problem solving. In algorithms such as A*, these evaluations are typically performed by heuristic functions, which estimate the distance from the current state to a goal state. In RL, these values represent an estimate of the total amount of reinforcement (i.e. reward and punishment) that will be accumulated on the path from the current state to a goal state; but their computation in RL involves (past and future) experience, not heuristics.

The neat trick of TD learning is a *bootstrapping* process in which the value of state s (denoted V(s)) derives from V(s'), where s' is a successor state to s. So if the agent is in state s and chooses action a, which converts the system to state s' and incurs reinforcement r, then the agent *backs up* this information to immediately update V(s) via the following formula:

[1] For more details on RL theory, the *bible* of the field is well worth consulting: *Reinforcement Learning: An Introduction (2nd Edition)*, Sutton and Barto, 2018.

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)] \tag{1}$$

where $\alpha$ is the learning rate, $\gamma$ is the discount factor (with a typical value of 0.9 or even 0.99), and the rightmost bracketed term is known as the *TD error* and denoted $\delta$:

$$\delta = r + \gamma V(s') - V(s) \tag{2}$$

TD error is the difference between a) the reward plus the discounted future state value, and b) the current state value. Intuitively, this represents the difference between a) the actual reward at time t+1 (r) plus the prediction of future rewards from time t+1 until the end of the problem-solving episode (V(s')), and b) the prediction of future rewards from time t to episode end, V(s). Thus, TD learning refines the prediction of the future given by V(s) to incorporate one new fact (r) and a prediction (V(s')) that is further into the future, but presumably closer to the goal (and thus a more accurate prediction than V(s)). Figure 1 illustrates these relationships between V(s), V(s'), r, $\delta$, and $\gamma$.

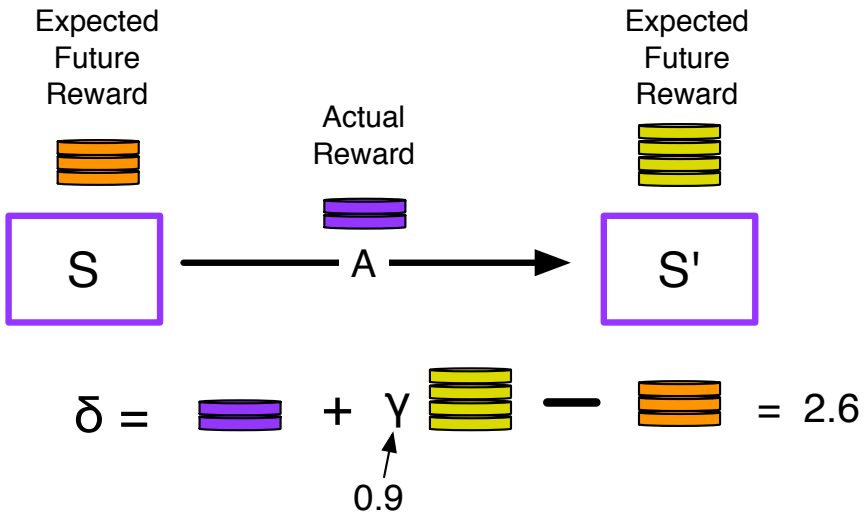


Figure 1: Basic components of Temporal-Difference Learning. Stacks of chips above each state represent that state's evaluation, i.e. estimate of future reward. $\gamma$ signifies the discount factor that TD Learning always applies to the evaluations of downstream states, while $\delta$ denotes the TD error, whose positive value (2.6) indicates that V(s) had been underestimated and will now be increased to take account of both V(s') and the reward of 2 garned by the s-to-s' transition.

In many TD-Learning tasks, r = 0 for most state transitions, with r being positive or negative only on the final step that leads to a goal state, which may constitute a win (positive) or loss (negative). In domains in which a good problem solution involves the fewest number of moves, a small negative reinforcement is often applied to each step, with a large positive reinforcement given for the transition from the penultimate state to the goal.

## 2.1 Episodes

In RL, the term *episode* has special meaning, refering to one attempt at solving the given problem; and RL is famous (or infamous) for requiring many episodes before reaching a good solution. For example, if trying to solve the control

problem of steering a (simulated) truck into a tight loading zone, RL would go through a series of episodes, each beginning with the truck at some start location, and each ending with success (truck in zone), failure (truck submerged in the harbor, crashing into a wall, etc.) or a simple time out (truck moves very little). In theory (and sometimes even in practice)[2], the RL controller's performance will improve from episode to episode as it gradually learns the true value of its states and actions.

## 2.2 State-Action Pairs (SAPs)

Although many forms of RL base their search process upon state evaluations, V(s), the cornerstone of many other RL algorithms is the state-action pair (SAP) and its evaluation. Hence, it is not the state (s), alone, to which value is attached, but s **plus the action(a) applied to that state**. The evaluation of this pair is often denoted by Q(s,a), due to its roots in a form of RL known as Q-Learning.

## 2.3 On-Policy versus Off-Policy RL

The overall goal of any RL algorithm is to find the best strategy (a.k.a. *policy*) for a given problem, where, formally, a policy is a mapping from SAPs to an evaluation (or probability). The goal strategy is known as the *target policy*. RL constitutes *learning by doing*, by, in fact, searching and learning from the experience, i.e. trial and error. During the many episodes of search, the target policy is gradually refined.

The search process itself is also driven by a policy, known as the *behavior policy*. It may or may not be the same as the target policy. In On-Policy RL, the behavior and target policy are equivalent, while in Off-Policy RL, they are different. This distinction comes up quite often in the RL literature.

## 2.4 Q-Learning

Q-learning is an *off-policy* form of TD Learning popularized by Watkins and Dayan in 1992. Temporal differencing in Q-Learning arises in updating Q(s,a) for the current state, $s_t$, and most recent action, $a_t$, by using the difference between the current value of $Q(s_t, a_t)$ and the sum of a) the reward, $r_{t+1}$, received after executing $a_t$ in $s_t$, and b) the discounted value of the resulting new state, $s_{t+1}$, where this value is greedily based on the **best possible action** that can be taken from $s_{t+1}$, or $max_a Q(s_{t+1}, a)$. Hence, the complete update equation is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{3}$$

As before, $\gamma$ is the discount rate and $\alpha$ is the step size or learning rate. Once again, the bracketed expression is $\delta$, the TD error. The logic behind this update rule is straightforward: if performing a in s leads to positive (negative) rewards and good (bad) next states, then $Q(s,a)$ will increase (decrease), with the degree of change governed by $\alpha$ and $\gamma$.

The use of $max_a$ in Equation 3 makes this an off-policy rather than on-policy learner, since it uses the **best** action (i.e. that leading to the best neighboring SAP) instead of the action recommended by the current policy, which, for many reasons, may not be the current optimal move. One common reason is that the behavior policy often mixes exploration with exploitation such that it tries many different actions, not only those known to produce (currently) good next states.

[2]RL methods, despite their general elegance and recent popularity, can be very difficult to coerce to success on real-world and toy problems alike.

## 2.5 Eligibility Traces

One important difference between TD Learning and many (simpler) RL variants is the backup process. Some reinforcement learners use a restricted backup of one (or just a few) steps. For example, if the algorithm uses two-step backup and search transitions from state $s_0$ to $s_1$ to $s_2$, then upon reaching $s_2$, the system would update V($s_0$) based on V($s_2$). If search then moved on to state $s_3$, this would only incur a backed-up update to V($s_1$), i.e., two steps into the past. Hence, it would take many successful trips to a goal state from any early state, such as $s_0$, before V($s_0$) would reflect the advantage of being in state $s_0$.

Eligibility traces alleviate this problem by essentially providing backup to all states after every move, although, in practice, algorithms normally backup only to those states involved in the current RL episode. These traces are implemented as simple continuous-valued flags, attached to each state, s, (or SAP) that indirectly indicate the elapsed time since s was last encountered during problem-solving search. As this time increases, the eligibility decreases, indicating that s (or (s,a)) is *less deserving* of an update to V(s) (or Q(s,a)), based on any recent rewards or penalties. Conversely, states with high eligibility should reap the full benefits (or suffer the full punishment) of a recent reinforcement. In short, eligibility traces handle the *temporal credit assignment problem*: apportioning credit and blame to states and actions that happened in the past but eventually led to a reward or punishment.

Formally, the eligibility trace for state s at time t is denoted $e_t(s)$ and updated as follows:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \tag{4}$$

where $s_t$ is the state encountered at time t, $\gamma$ is the now-familiar discount factor, and $\lambda$ is the *trace-decay* factor. Together, these two factors (both in the range [0, 1]) determine the rate at which an eligibility trace decreases with each passing time step after a state has been visited, i.e., after $s = s_t$. Notice that when $s = s_t$, $e_t(s)$ gets incremented by 1, but on all other steps, it will decay (assuming $\gamma\lambda < 1$).

The second expression of Equation 4 is known as an *accumulating trace*, since a state that occurs several times in a short period of time (or in the same episode) can accumulate an eligibility that exceeds 1.0. For some problems, this can hinder proper learning. A simple solution involves *replacing traces*, which employ a slightly different update formula:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases} \tag{5}$$

The second expression of Equation 5 guarantees that no eligibility exceeds 1.0 and tends to work much better in most cases.

In TD learning with eligibility traces, often denoted TD($\lambda$), the basic sequence of events are repeated for each step of an episode:

1. $a \leftarrow$ the action dictated by the current policy when the state is s, $\Pi(s)$.
2. Performing action a from state s moves the system to state s' and achieves the immediate reinforcement r.
3. $\delta \leftarrow r + \gamma V(s') - V(s)$
4. e(s) $\leftarrow$ 1 (note the replacing, not accumulating, trace)

5. $\forall s \in S$ (alternatively, $\forall s \in E$, where E is the sequence of states in the current episode):
   (a) V(s) $\leftarrow$ V(s) + $\alpha\delta e(s)$
   (b) e(s) $\leftarrow \gamma\lambda e(s)$

On line 3 of this algorithm, the TD error, $\delta$, is computed based on the values of the current and best-successor states, along with the reward, just as before. On line 5(a), that error, whether positive or negative, is then applied to every state of the search space, as the V(s) value of each state gets updated, and by an amount directly proportional to the eligibility trace of s, e(s). On line 5(a), $\alpha$ is a learning rate. The eligibilities are updated on line 4 (for the current state) and then decayed for all states on line 5(b).

In practice, TD($\lambda$) may use various tricks, such as a cache containing only those states in the current episode, to avoid updating the value of every state after each action choice, but in theory, each state gets an updated value on every step of the algorithm, with the size of each update diminishing quickly for $\lambda < 1$.

Be aware that the algorithm above pertains to **tabular** versions of TD($\lambda$). These use large tables to store V(s) for all s. However, for problems with large search spaces, these tables are infeasible to construct, so the reinforcement learner must rely on a function approximator, such as a neural network (as discussed below). In those situations, the basic TD($\lambda$) algorithm still holds, but line 5a is replaced with a run of backpropagation on the neural network, helping it to produce an ouput (when given an encoding of state s as input) that is closer to $r + \gamma V(s')$.

For Q-Learning, the accommodation of eligibility traces works similarly, with one key difference involving the computation of $\delta$, the TD error:

$$\delta = r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{6}$$

The Q values for each state-action pair (SAP) are then updated, using eligibility traces and the TD error, as follows:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a) \tag{7}$$

Note that now, the eligibility trace is attached to the SAP, not the individual states, but the basic update is similar to that of line 5(a) in the above algorithm.

Furthermore, several Q-Learning variants that employ eligibility traces will reset all eligibilities to 0 whenever the current policy selects a random (exploratory) action instead of the current optimal (i.e., exploitative or *greedy*) option.

## 2.6 SARSA($\lambda$) Learning

The acronym SARSA stands for "state action reward state action", indicating that it relies on transitions between SAPs, with an intervening reward. Since it is based on SAP evaluations, not state evaluations, SARSA ressembles Q-Learning. However, SARSA is **on-policy** in that the actions associated with successor states are those dictated by the current policy, and not necessarily the current optimal choices.

The basic algorithm is as follows:

- Initialize Q(s,a) with small random values.

- Repeat for each episode:
  - Reset eligibilities: e(s,a) $\leftarrow 0\ \forall s,a$
  - Initialize: s $\leftarrow s_{init}$; a $\leftarrow \Pi(s_{init})$
  - Repeat for each step of the episode:
    1. Do action a from state s, moving the system to state s' and receiving reinforcement r.
    2. $a' \leftarrow \Pi(s')$: the action dicated by the current policy for state s'.
    3. $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
    4. e(s,a) $\leftarrow 1$ (note the replacing, not accumulating, trace)
    5. $\forall(s,a) \in$ All SAPs (alternatively, $\forall(s,a) \in E$, where E is all SAPs in the current episode):
       (a) Q(s,a) $\leftarrow$ Q(s,a) + $\alpha\delta e(s,a)$
       (b) e(s,a) $\leftarrow \gamma\lambda e(s,a)$
    6. s $\leftarrow$ s' ; a $\leftarrow$ a'
  - Until s is an end state.

In the above algorithm, note the calculation (line 3) of TD-error, as repeated in equation 8. The contrast between this and equation 3 signifies the key difference beween SARSA($\lambda$) and Q-Learning, both of which employ Q values.

$$\delta \leftarrow r + \gamma Q(s',a') - Q(s,a) \tag{8}$$

It is important to note that the algorithm above pertains to **tabular** versions of SARSA($\lambda$), wherein the Q(s,a) values are stored in (large) tables. For RL architectures that employ function approximators (e.g. neural networks) instead of tables, the essential difference lies in line 5a, where (as detailed below), updates occur to many of the weights in the neural network as part of the backpropagation procedure, which is also influenced by eligibility traces. The cumulative effect of all these weight changes produces changes in the net's output (representing Q(s,a)) when given an encoded version of (s,a) as input.

If you choose to base your RL system on Q(s,a) values instead of V(s) values, then SARSA($\lambda$) is a bit easier to implement than Q-Learning.

# 3 The Actor-Critic Model

A popular variant of TD learning, known as the Actor-Critic Model (ACM), is characterized by a complete separation between the value function and the policy. The *actor* module contains the policy, $\Pi(s)$, whereas the *critic* manages the value function, V(s) or Q(s,a). Many TD models fit into the actor-critic framework, but here we focus on TD($\lambda$) and the use of eligibility traces to update both $\Pi(s)$ and V(s).[3]

In this approach, the critic module manages V(s) and computes the TD error (which is based on V(s) for the current and successor state). The actor then utilizes the TD error for its own update. This is sketched in Figure 2.

TD error ($\delta$) is computed as before (equations 2 and 8), and eligibility traces update as earlier (equations 4 and 5). In addition, the critic modifies its value function as in line 5(a) of the TD($\lambda$) algorithm, repeated below:

[3]The notation for policies is potentially confusing. In this document, $\Pi(s)$ represents the action recommended by the actor when the system is in state s. $\Pi(s,a)$ denotes the actor's quantitative evaluation of the desirability of choosing action a when in state s. Thus $\Pi(s) = \text{argmax}_a \Pi(s,a)$.

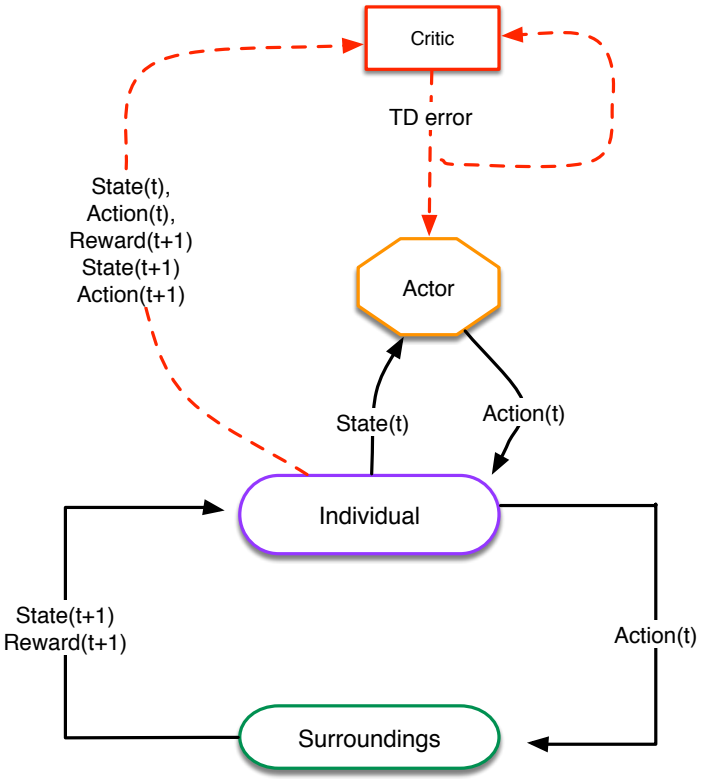


Figure 2: Overview of the actor-critic paradigm. Solid lines show a basic behavioral sequence, wherein the actor selects an action, based on the policy, and performs it in the environment, which then returns a new state and (possibly null) reward. Dotted lines show the learning process, whereby the states at two successive time points along with the reinforcement are sent to the critic, which uses that information to compute the TD error, which is then used (by the critic) to update $V(s_t)$ and sent to the actor to update the policy. If the critic is based on state-action pairs, then it also requires the actions from times t and t+1 to compute the TD error and update Q(s,a).

$$V(s) \leftarrow V(s) + \alpha\delta e(s) \tag{9}$$

Alternatively, if the critic employs SAPs, the value update mirrors that of line 5(a) in the SARSA($\lambda$) algorithm, also repeated below:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a) \tag{10}$$

With the Actor-Critic Model, the policy updates occur only within the actor and rely directly upon the TD error, and thus only indirectly upon V(s) (or Q(s,a)).

Although the critic may either work with state-action pairs or only with states, the actor must work with both states and actions, since it's main job is to maintain the policy, which determines the appropriate action for a given state. Essentially, the policy represents a mapping:

$$\Pi(s,a) \longrightarrow z \tag{11}$$

where z is a real number giving some indication of the desirability of performing action a when in state s.

In learning a good policy, i.e. in refining these z values, the actor needs to keep track of the results of performing actions in states. A fundamental aspect of these results is the TD error, $\delta$. High values of $\delta$ signify actions that led to *better than expected* outcomes, where the expectation is embodied in V(s) (or Q(s,a)), while the outcome stems from the reward plus V(s') (or Q(s',a')). Hence, SAPs that produce high TD-errors should normally experience increases to their odds of future deployment; and those yielding negative $\delta$ should see reduced selection.

In addition, the policy updates can take advantage of eligibility traces such that SAPs activated early in an episode can still garner (positive or negative) updates due to TD errors encountered much later in that same episode. Together, eligibility and TD error account for the key policy updates, as expressed in equation 12.

$$\Pi(s,a) \leftarrow \Pi(s,a) + \alpha\delta e(s,a) \tag{12}$$

$\Pi(s,a)$ (whether modelled computationally as a table or a function approximator) can then be used in several different ways to govern the actor's choice of actions. Under a strictly greedy strategy, the actor always chooses the action with the highest $\Pi(s,a)$ value: for current state, s, choose $a^* = \text{argmax}_{a^i}\,\Pi(s,a^i)$.

An $\varepsilon$-greedy strategy makes a random choice of actions with probability $\varepsilon$, and the greedy choice with probability $1-\varepsilon$.

Alternatively, the actor may make a stochastic action choice, with the weights associated with each action based directly upon the $\Pi(s,a)$ values. Since these values can be positive or negative, they may require scaling prior to probabilistic action choice. For example, Boltzmann scaling produces non-negative values $\Pi^*(s,a)$ as follows:

$$\Pi^*(s,a) = \frac{e^{\Pi(s,a)}}{\sum_{a_i} e^{\Pi(s,a_i)}} \tag{13}$$

Then, for current state s, the actor chooses an action, a, with probability $\Pi^*(s,a)$.

Complex search problems normally require a good balance between exploration and exploitation, and an $\varepsilon$-greedy strategy facilitates such a mixture, particularly when the value of $\varepsilon$ changes during the course of the run. Typically, it decreases from the earlier to the later episodes, where high values entail a lot of exploration (i.e., randomness), while lower values imply more exploitation (i.e. greediness). For example, a complete run of TD Learning may begin with $\varepsilon = 0.5$ and end with $\varepsilon = 0.001$. The proper values can be problem-dependent and are best determined by experimentation, but, in general, a great many AI problem-solving strategies rely on early exploration that gradually gives way to increasing exploitation.

## 3.1 A Generic Actor-Critic Algorithm

Details of actor-critic implementations can vary greatly; the description below will not cover all cases, but it gives the gist of the approach, whose hallmark is a clean separation of state (or SAP) evaluation and policy updating.

The following basic algorithm employs a state-based critic[4] and uses labels to denote activities performed by the actor and critic. Unlabeled activities are presumably performed by an over-arching RL object that contains the actor and critic.

- CRITIC: initialize V(s) with small random values.
- ACTOR: Initialize $\Pi(s,a) \leftarrow 0\ \forall s,a$.
- Repeat for each episode:
  - Reset eligibilities in actor and critic: e(s,a) $\leftarrow$ 0: $e(s) \leftarrow 0\ \forall s,a$
  - Initialize: s $\leftarrow s_{init}$; a $\leftarrow \Pi(s_{init})$
  - Repeat for each step of the episode:
    1. Do action a from state s, moving the system to state s' and receiving reinforcement r.
    2. ACTOR: $a' \leftarrow \Pi(s')$ the action dictated by the current policy for state s'.
    3. ACTOR: e(s,a) $\leftarrow$ 1 (the actor keeps SAP-based eligibilities)
    4. CRITIC: $\delta \leftarrow r + \gamma V(s') - V(s)$
    5. CRITIC: e(s) $\leftarrow$ 1 (the critic needs state-based eligibilities)
    6. $\forall(s,a) \in$ current episode:
       (a) CRITIC: V(s) $\leftarrow$ V(s)+ $\alpha_c \delta e(s)$
       (b) CRITIC: e(s) $\leftarrow \gamma\lambda e(s)$
       (c) ACTOR: $\Pi(s,a) \leftarrow \Pi(s,a) + \alpha_a \delta e(s,a)$
       (d) ACTOR: e(s,a) $\leftarrow \gamma\lambda e(s,a)$
    7. s $\leftarrow$ s' ; a $\leftarrow$ a'
  - Until s is an end state.

In this algorithm, note that the critic computes the TD error, which both critic and actor then employ to update the evaluator and policy, respectively. Also note that the eligibilities needed by actor and critic may differ. Since the actor always deals with SAPs, it requires e(s,a); but the critic often uses states alone and thus uses e(s). Furthermore, as we will see below, eligibility traces in function approximators cannot be tied to specific states or SAPs and thus take on

[4] SAP-based critics are a straightforward extension of this algorithm.

a different form altogether, and one that cannot possibly be shared between an actor and critic. Thus, the eligibilities are best viewed (and implemented) as properties of actors and critics, not of the general RL algorithm.

Furthermore, the algorithm above employs different learning rates, $\alpha_a$ and $\alpha_c$ for the actor and critic, respectively. This separation becomes increasingly important in hybrid actor-critic models where one (e.g. the critic) employs a function approximator (such as a neural network), while the actor uses table lookup. In this case, the neural network may require a much smaller learning rate (by several orders of magnitude).

# 4 Function Approximation

Many educational examples of RL easily permit a complete enumeration of all states (or SAPs) and thus the production of evaluation tables (with one entry per state or SAP). In other cases, it may not make sense to generate all states or SAPs beforehand, but dynamically, as problem-solving episodes unfold. These incrementally-generated tables can also support effective RL. Problems arise when the state space gets very large and RL explores a wide variety of trajectories through it such that the problem solver only occasionally sees the same state twice, and thus, the statistics generated for each state (or SAP) are miserably inadequate: just because a state, s, appears on one path to a winning (or losing) final state does not imply that s is good (bad). Many paths involving s are needed to make an accurate assessment of it's true value.

The classic antidote to this problem involves function approximators, which are mappings from encodings of states (or SAPs) to evaluations. These specifically do not perform table look-ups and thus do not have separate entries for each state (SAP). Instead, they perform calculations on state (SAP) encodings to produce output values. The learning aspect of RL then involves tuning of the function approximator such that it eventually generates accurate evaluations for most states, although it may only experience each of them a few times. However, by seeing **many similar** states and learning to treat them equally (or nearly so) in terms of the output evaluations given to them, the function approximator accumulates a sufficient statistical basis for intelligent appraisals.

One very common tool for function approximation is the artificial neural network (ANN). Tesauro's TD-Gammon system employed a shallow (one hidden layer) ANN as the backbone of its critic module, while its actor essentially used the critic quite directly to determine moves, choosing that move which led to the new state with the highest evaluation.

Although many recent applications of the RL-ANN combination to complex games (i.e. Atari and Go) were inspired by Tesauro's work, they have extended the use of function approximators to the actor – with Alpha Go using them in both the actor and critic. The ANNs in actors take state encodings as input and produce an (often large) vector of probabilities as output, with each denoting the weight of an action, with the actor typically using an $\varepsilon$-greedy choice strategy based on those probabilities. Proper training of such a network requires the complete enumeration of all possible actions in the beginning of an RL run, a job that is straightforward in games with a restricted action set (such as Space Invaders or Pong), but much more demanding in games with multiple pieces where any one of them can move. Unfortunately, the training of actor ANNs for these more open-ended games provides a challenge well beyond the scope of this course.

As a practical compromise, the actor can be implemented as a table (even though it will not come close to filling up with useful evaluations for all state-action pairs), while the critic can employ an ANN as a function approximator. Then, the actor's updates of specific SAPs can still take advantage of the generality embodied in the ANN (which will normally map similar input states or SAPs to similar evaluations).

## 4.1 Function Approximators in the Critic

When the core of the critic is an ANN (instead of a table), as sketched in Figure 3, it takes a state's encoding as input and produces a state evaluation as output. Alternatively, if the critic is SAP-based, the encoded input includes both the state and the action taken in that state.

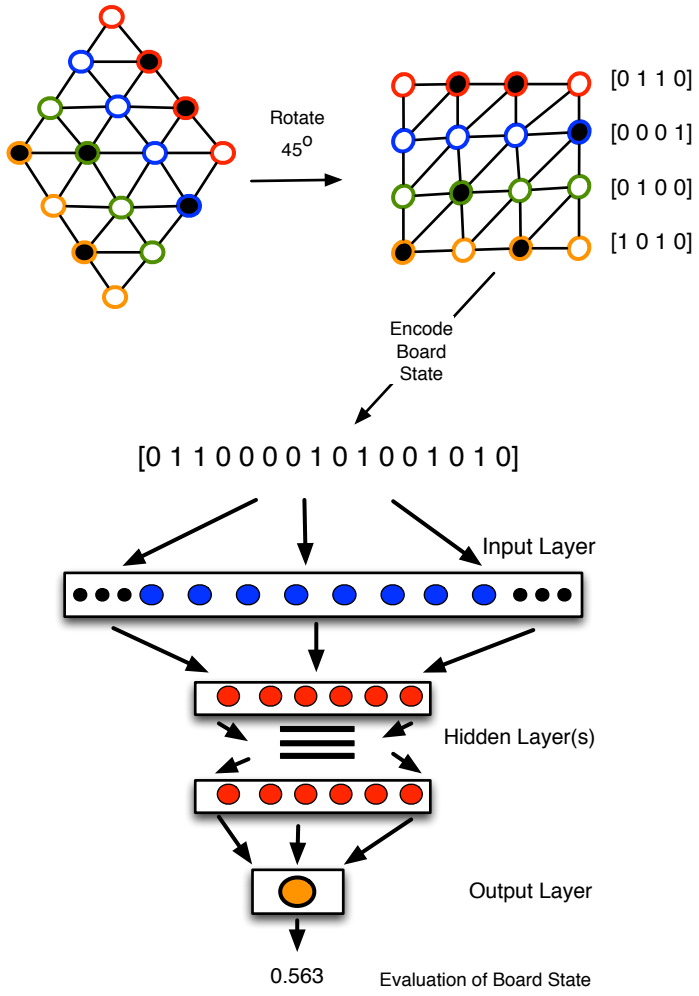


Figure 3: The basic framework for the critic's function approximator, a neural network. (Top) A diamond-shaped hexagonal board, when rotated 45 degrees, yields a square matrix. The state of this board is easily encoded as a bit vector. (Bottom) The input neurons receive this state vector, and feedforward propagation produces a single output value that represents the evaluation of the original board state.

In a table-based critic, each state has a table entry corresponding to its evaluation, which gets modified via equation 14, reprinted from the Actor-Critic algorithm, which involves the learning rate ($\alpha_c$), TD error ($\delta$), and eligibility trace (e(s)).

$$V(s) \leftarrow V(s) + \alpha_c \delta e(s) \tag{14}$$

However, when the critic uses a function approximator (F) instead of a table, no unique location within that complex

operator (e.g. neural network) corresponds to a particular problem-solving state (s) nor its value (V(s)). Still, we wish to tune F such that, when presented with s as input, it produces a realistic V(s) as output.

One approach (taken by Tesauro and others to follow) involves modifying each component (i.e. weight, $w_i$) of F to a degree proportional to the influence that $w_i$ has upon the output value (i.e. V(s)) when given s as input: $\frac{\partial V(s)}{\partial w_i}$, which is a standard part of the loss gradient in backpropagation. In addition, $w_i$ should get a modification that reflects its contribution to the evaluation of recent predecessor states in the current episode. This approach incorporates eligibility traces, one per weight, and ties them directly to $\frac{\partial V(s)}{\partial w_i}$ (as detailed below).

To understand these relationships, examine Figure 4, which shows the neighborhood of weight $w_i$ in a neural network that maps state s to output V(s).

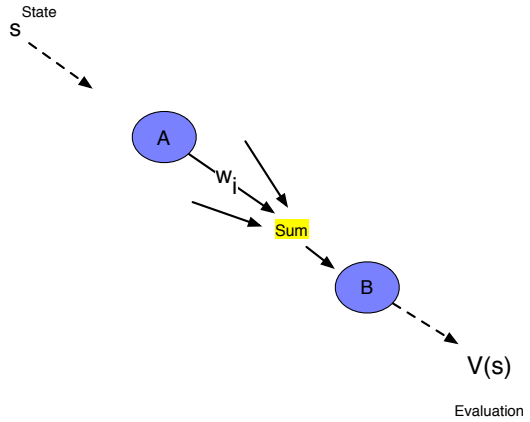


Figure 4: Dissecting the derivative of a neural net output, V(s), with respect to a weight, $w_i$.

To compute V(s), the network propagates the encoding of s forward, thus causing diverse neural activity such that some nodes and weights will contribute more or less to the production of V(s). To train this network in a supervised fashion requires a target value for V(s). The key assumption behind temporal differencing is that the correct value for V(s) is $r + \gamma V(s')$ (where s' is the next state and r is the reward received in going from s to s'). Let us label that value T, for target.

In backpropagation, the typical error term (a.k.a. loss) is the square of the difference between the target and the actual output. The derivative of that loss (L) with respect to each weight, $w_i$, is then the basis for weight updates.

Mathematically: [5]

$$\frac{\partial L}{\partial w_i} = \frac{\partial (T - V(s))^2}{\partial w_i} = 2(T - V(s)) \frac{-\partial V(s)}{\partial w_i} \tag{15}$$

Since T = $r + \gamma V(s')$, it follows that (T - V(s)) is simply the TD error, $\delta$. Hence:

$$\frac{\partial L}{\partial w_i} = -2\delta \frac{\partial V(s)}{\partial w_i} \tag{16}$$

[5] Astute readers will notice that since T contains V(s'), it is not really independent of $w_i$ and thus cannot be so easily removed from the derivative below. However, this assumption – of T being *the correct* value, or at least a better output value – provides enough information to drive learning in the right direction.

The standard update rule for weights under backpropagation is:

$$w_i \leftarrow w_i - \alpha \frac{\partial L}{\partial w_i} \tag{17}$$

where $\alpha$ is the learning rate. Thus (assuming the 2 in Equation 16 is absorbed into the learning rate):

$$w_i \leftarrow w_i + \alpha \delta \frac{\partial V(s)}{\partial w_i} \tag{18}$$

Equation 18 represents the weight update that a standard off-the-shelf deep-learning tool (such as Keras or PyTorch) will automatically perform when given the loss function (L) described above, and repeated below:

$$L = [T - V(s)]^2 = [r + \gamma V(s') - V(s)]^2 = \delta^2 \tag{19}$$

Thus, to perform the basic form of Temporal Differencing, TD(0), with a neural network, your main task is to simply compute the TD error ($\delta$), square it, and supply that to Keras or PyTorch as the loss. It will then compute the proper weight updates for your critic.

In fact, all you really have to do is declare the loss function as mean-squared error and supply the proper target value: $r + \gamma V(s')$ for the input state s. In short, for TD(0), you can basically treat the deep-learning system as a black box and supply it with input-target pairs of the form [s, $r + \gamma V(s')$]. However, if you wish to perform TD($\lambda$) with a neural network critic, you will need to dig inside the black box and supplement the standard gradients that these packages compute with eligibility traces.

## 4.2 Including Eligibility Traces in a Neural Network Critic

Recall that the calculus behind backpropagation uses the chain rule of differentiation to compute $\frac{\partial V(s)}{\partial w_i}$, such that:

$$\frac{\partial V(s)}{\partial w_i} = \frac{\partial Sum}{\partial w_i} \frac{\partial y_B}{\partial Sum} \frac{\partial V(s)}{\partial y_B} \tag{20}$$

where Sum is the weighted sum of inputs to neuron B (in Figure 4); B's output is $y_B$, and $\frac{\partial V(s)}{\partial y_B}$ is a term that back-propagation would further expand by the chain rule.[6] Furthermore, note that:

$$\frac{\partial Sum}{\partial w_i} = y_A \tag{21}$$

where $y_A$ is the output of neuron A. Thus, only if neuron A contributes significantly to propagation of the encoding for state s will $\frac{\partial Sum}{\partial w_i}$ have an influential magnitude, thus enabling (though not insuring that) $\frac{\partial V(s)}{\partial w_i}$ has a significant magnitude. And according to our original argument, $w_i$ should only be eligible for an update if it contributes to V(s) or has contributed to V(s*) for some recent predecessor state, s*, to s. This line of reasoning leads to the following core updates employed by Tesauro and others:

[6] But doing so only complicates the equations without enhancing the explanation for our current purpose.

$$e_i \leftarrow \underbrace{\gamma\lambda e_i}_{decay} + \frac{\partial V(s_t)}{\partial w_i}$$

$$w_i \leftarrow w_i + \alpha\delta e_i$$

where $e_i$, the eligibility of $w_i$, changes in response to the standard partial derivative, $\frac{\partial V(s_t)}{\partial w_i}$, and $w_i$ then updates based on $e_i$ and the TD error ($\delta$). In addition, all eligibilities decay after each action in an episode (as in all versions of TD learning presented earlier). Notice how this update to $w_i$ incorporates both the standard gradient of backpropagation (Equation 18) and the *memory* of other recent contributions embodied in the eligibility trace.

Thus, as an episode transitions through a state sequence, the critic's neural network produces different activity patterns, each of which influences eligibility traces in different ways based on the values of $\frac{\partial V(s_t)}{\partial w_i}$ for all i. A weight that has an active influence upon V(s) early in this sequence will receive modification (commensurate with the TD error) at that time but will witness a decaying eligibility trace (and thus reduced updates) if it fails to influence V(s) during later states in the sequence. Other weights may be actively involved in computing V(s) throughout the sequence and will therefore experience a steady sensitivity to the TD error: the larger its magnitude, the larger the weight change. All of this is in the same spirit as the original, table-based approach to TD Learning, but now the updates must be distributed over the weights of the function approximator.

## 4.3 Implementing Eligibility Traces on a Deep Learning Platform

The popular Deep Learing (DL) platforms, such as Tensorflow and PyTorch, provide excellent libraries that take most of the drudgery out of DL coding. However, a convenient high-level abstraction can sometimes block access to important low-level code that a programmer really needs in order to do an unconvential task. As it turns out, eligibility traces for neural networks are somewhat unconventional. Luckily, the main DL platforms provide relatively easy access to code that lay slightly beneath the abstraction level found in most DL applications.

Both Tensorflow and PyTorch allow users to easily build and run neural networks. Tensorflow 2.0 supports Keras, a powerful and convenient high-level API. In Keras, neural networks are quickly configured (in just a few lines of code) and consolidated into a *model*. Models are then compiled and run using the methods *model.compile* and *model.fit*. The former method combines the model with an optimizer, loss functions, evaluation metrics, etc. The latter, model.fit, does all of the hard work associated with backpropagation: it loops through alternating processes of gradient computation and parameter (i.e. weight and bias) updating, where the updates depend upon the gradients.

Most DL applications in Keras can employ model.fit *out of the box*, as one atomic operation. However, the implementation of eligibility traces requires access to the subprocesses of model.fit. It requires the interjection of tailor-made code after the computation of gradients and before the updating of parameters, since the eligibility traces are a) modified by the gradients, and then b) used in place of the gradients as the basis for parameter updates.

Hence, the use of eligibility traces forces the programmer to become just a little more knowledgeable of the workings of his or her favorite DL platform.

Note that the updates for eligibilities and weights in the calculations above involve $\frac{\partial V(s_t)}{\partial w_i}$, whereas these standard DL packages typically compute the gradient of the loss, $\frac{\partial L}{\partial w_i}$. Although one can easily compute one from the other using equation 16, this can introduce discontinuities when $\frac{\partial V(s_t)}{\partial w_i}$ is computed as $\frac{-1}{2\delta}\frac{\partial L}{\partial w_i}$ and the TD error ($\delta$) is zero.

A (somewhat cleaner) alternative is to simply define a loss function that is really just a pass-through of the network's

output: loss(target,output) = output. Then, $\frac{\partial V(s_t)}{\partial w_i} = \frac{\partial L}{\partial w_i}$.[7]

The file splitgd.py (see the *Materials* section of the course webpage) provides a class and a few methods for handling this basic situation in Tensorflow/Keras. Similar solutions exist for PyTorch, which, in many respects, provides even easier access to the low-level operations of backpropagation.

# 5 Summary

This document serves as a basic introduction to the important technical aspects of the Actor-Critic Model of Temporal Differencing. It places the actor-critic within the larger scheme of TD methods before delving into reasonably detailed pseudocode, which can serve as a general framework for implementing actors, critics and their interactions.

Many contemporary applications of TD learning involve state spaces of such size and complexity that tabular representations become infeasible, and mechanisms such as neural networks become vital for function approximation: for mapping states to values, and states to actions. Another core focus of this document is an explanation of the mathematical basis for training neural-network function approximators via eligibility traces, a central element of many TD methods. Hints as to how to implement eligibility traces in popular Deep Learning toolkits are also provided.

However, this document only covers the use of function approximators in the critic. Using them in the actor has many advantages but presents more (and greater) technical challenges. The curious reader should consult sections on *policy gradients* in the main textbook of the field: *Reinforcement Learning: An Introduction*, 2nd Edition, Sutton and Barto, 2018. That book also provides in-depth explanations of nearly everything covered in this document. If you're serious about doing RL, you need to buy it, read it, and keep it close to your computer.

[7] When using a neural network as an RL critic, the only important error (loss) is the TD error, and it is still involved in the modification of weights (as shown above). Using mean-squared error (or a similar loss function) is basically superfluous.