Reinforcement Learning

1: Introduction

- **Reinforcement Learning –** Goal-directed learning from interaction with an environment formulated as how to map situations to actions so as to maximize a numerical reward signal using a complete, interactive goal-seeking agent. The formulation must include sensation, action, and goal.
    - **Exploration vs. Exploitation –** A reinforcement learning agent must prefer actions that it has tried in the past and found to be most effective (exploitation), but to discover such actions, it must try new actions that have never been tried before (exploration).
    - **Policy π(s) –** defines the way in which a learning agent acts in a specific situation.
    - **Reward Function R(s,a) –** defines the goal of the problem by mapping each state-action pair of the environment to a specific number – the **reward** – that is an indication of the desirability of that pair.
    - **Value Function V(s) –** specifies a long-term desirability. An approximation of the amount of expected reward an agent can gain in starting from a specific state.
    - **Model –** mimics the behavior of the environment (ie given a state and action, it tries to predict the next state and action).
        - Models are used for **planning** – deciding a course of action by considering possible future situations before actually experiencing them.
    - We seek to maximize value not rewards. Rewards are given directly from the environment whereas values must be continuously reestimated.
    - How well a reinforcement learning algorithm works in problems of large state sets is tied to how approximately it can generalize from past experience.
    - **evolutionary methods –** methods that search in the space of policies directly without ever appealing to the value function (genetic algorithms, simulated annealing, etc).

2: Evaluative Feedback

- Increment Update Rule: A new estimate of a quantity is obtained from the old estimate, the target value, and a step size on the interval [0,1]:

$$NewEstimate = OldEstimate + StepSize\underbrace{\left[\text{Target} - OldEstimate\right]}_{\text{error of estimate}}$$

- **Instructive Feedback** – Feedback independent on the action taken. For instance, the feedback might say what the correct action was → supervised learning.
  - The problem facing a supervised learning system is to construct a mapping from situations to actions that mimic the correct actions specified by the environment and generalize correctly to new situations ➔ behave as instructed by its environment.
  - Works well for deterministic rewards. Not so well with stochastic rewards.
  - Any method that takes success as an indication of correctness can easily becomes stuck choosing the wrong action in the stochastic case.
  - **Linear, reward-penalty (L_{R-P})** – if the action inferred to be correct on play t was d_t, then the probability of selecting d_t, $\pi_t(d_t)$ is updated as an incremental update:

    $$\pi_{t+1}(d_t) = \pi_t(d_t) + \alpha\left[1 - \pi_t(d_t)\right]$$

    whereas probabilities of other states are updated inversely.
  - **Linear, reward-inaction (L_{R-I})** – Identical to L_{R-P} except it updates its probabilities only upon successful plays. Failures are entirely ignored.
- **Evaluative Feedback** – Feedback that depends on the action taken.
- **n-Armed Bandit –** The agent is repeatedly faced with a choice of between n different options (actions). After each choice you receive a numerical reward chosen from a stationary distribution that depends on the action. The agent wants to maximize expected total reward over t plays. NOTE: this is a stationary distribution. To address the non-stationary task, we have associative reinforcement learning.
  - **greedy action –** action whose estimated value is the greatest.
  - **exploiting –** choosing the greedy action in order to receive the expected largest reward.
  - **exploring –** choosing a non-greedy action in order to improve the estimate of the non-greedy reward.
  - supervised methods perform poorly as they do not balance between exploration and exploitation at all.
- *Action-Value Estimation Methods*
  - **sample average** – averaging over all rewards received upon applying action a that was used k_a times. In the limit as k_a goes to infinity, by law of large numbers, the sample average $Q_t(a)$ converges to the true value $Q^*(a)$:

$$Q_t(a) = \frac{1}{k_a}\sum_{i=1}^{k_a} r_i$$

where r_i is the reward received from the *i*-th application of action *a*.

- **incremental (decaying) averages** – averaging by a decaying update based on the incremental update rule:

$$Q_{k+1}(a) = Q_k(a) + \alpha_k(a)\left[r_{k+1} - Q_k(a)\right]$$

$$\alpha_k(a) \in [0,1]$$

  - For constant α, this is a *exponential, recency-weighted average* or decaying average, that gives more emphasis to history for $\alpha \rightarrow 1$ and more emphasis on recent reward as $\alpha \rightarrow 0$.
  - For $\alpha_k(a) = 1/k$, this is equivalent to the sample-average, which is guaranteed to converge to $Q^*(a)$.
  - **Stochastic Approximation Conditions** – For an arbitrary sequence $\{\alpha_k(a)\}$, $Q_t(a)$ converges to $Q^*(a)$ with a probability 1, with the following conditions:
    1. *Large enough to overcome initial values and random fluctuation:*

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty$$

    2. *Small enough to assure convergence:*

$$\sum_{k=1}^{\infty} \alpha_k^2(a) < \infty$$

- *Action Selection Methods*

  greedy action: $a_t^* = \arg\max_a Q_t(a)$

  - **Greedy Policy** – always chooses the greedy action a_{t+1}^*.
  - **ε-Greedy Policy** – chooses the maximal action a_{t+1}^* with probability *1-ε*, but with probability ε chooses another non-greedy action.
    - Since k_a goes to infinity for all *a* as *t* goes to infinity, we still get convergence of $Q_t(a)$ to $Q^*(a)$.
    - More variance in rewards favors an ε-Greedy policy over a pure greedy policy since more exploration is needed.
    - Would be advantageous to decrease ε as time gets large since there is less uncertainty in the value of actions.
    - ε parameter chosen as a confidence.
  - **Softmax Policy** – Chooses action *a* on the *t*-th play according the Gibbs (Boltzman) distribution.

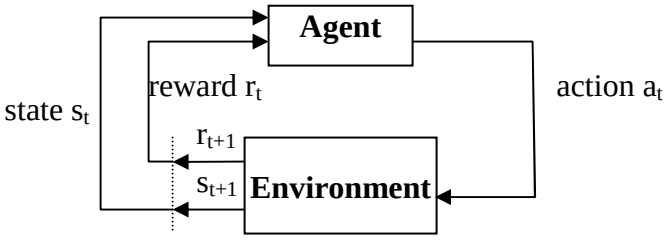
$$\pi_t(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$$

where τ is a temperature parameter: higher = more random

        - Gives more or less favoritism to actions based on their relative value estimations.
        - Hard to estimate τ parameter.
- **Optimistic Initial Values –** By initializing the initial values of an action-value to a non-zero value (*optimistic initial values*), we are essentially able to incorporate prior knowledge into the agent causing even a pure greedy approach to perform more exploration, albeit temporarily (hence, not helpful in a non-stationary case).
- **Reinforcement Comparison Methods –** methods that judge whether a given reward is small or large compared to other rewards, thus making the search for large rewards relative to those previously seen.
    - **reference reward** $\bar{r}_t$ **–** an incremental average of all recently received rewards, independent of which action was taken:
    $$\bar{r}_{t+1} = \bar{r}_t + \alpha\left[r_t - \bar{r}_t\right]$$
    - **action preference $p_t(a)$ –** the preference for action a at play t; an incremental average:
    $$p_{t+1}\left(a_t\right) = p_t\left(a_t\right)_t + \beta\left[r_t - \bar{r}_t\right]$$
    - **action selection probability –** a softmax function giving the probability of selecting an action:
    $$\pi_t\left(a\right) = \frac{e^{p_t(a)}}{\sum_{b=1}^{n} e^{p_t(b)}}$$
- **Pursuit Methods –** maintains both action-value estimates and action preferences, with the preferences continually "pursuing" the action that is greedy according to current action-value estimates.
    - After the *t*-th play the greedy action for the *t+1*-th play is a_{t+1}^*. The probability of selecting $a_{t+1} = a_{t+1}^*$ is (as in the $L_{R\text{-}P}$),
    $$\pi_{t+1}\left(a_{t+1}^*\right) = \pi_t\left(a_{t+1}^*\right) + \beta\left[1 - \pi_t\left(a_{t+1}^*\right)\right]$$
    and for all other actions,
    $$\pi_{t+1}\left(a_{t+1}\right) = \pi_t\left(a_{t+1}\right) + \beta\left[0 - \pi_t\left(a_{t+1}\right)\right] \quad a_{t+1} \neq a_{t+1}^*$$
    - $Q_{t+1}(a)$ action-value estimates are updated from an above method.
- **Associative Search –** the task of both *searching* for the best action and *associating* actions with the situations in which they are best.
    - in a general reinforcement problem, there are multiple situations and the goal is to learn a **policy**: a mapping from situations to actions that are best in those situations.
    - this addresses the problem when the bandit task changes randomly from play-to-play but we have indications about what task we are tasked with.

3: The Reinforcement Learning Problem

- **Agent –** the learner and decision maker.
- **Environment –** everything external to the agent.
- **Task –** a complete specification of an environment
- **State** $s_t \in S$ - the representation of the environment the agent receives at time *t*.
- **Action** $a_t \in A(s_t)$ **–** a choice made by the agent based on its state at time *t*.
- **Reward** $r_{t+1} \in R$ **–** a representation of the goal achieved by the agent due to its action at time *t*.
- **Policy** $\pi_t(s,a)$ **–** the probability that $a_t = a$ given that $s_t = s$, thus defining the agent's method of choosing actions for a state.
- **Goal –** maximize the total amount of reward received.
- **Agent/Environment Boundary –** the boundary between the agent and environment is often not the physical boundary that separates them.
    - The general rule is, anything that cannot be arbitrarily changed by the agent is in the environment.
    - The agent-environment boundary is the limit of the agent's realm of absolute control; not the limit of its knowledge.



- *Goals and Rewards*
    - At each time step *t*, the reward received by the agent is r_t. Our agent seeks to maximize the total amount of reward received.
    - We want to define rewards in such a way that maximizing them will cause the agent to achieve our goals.
        - Goals should indicate *WHAT* you want to achieve, *NOT HOW* you want to achieve it. Hence, rewards should not be used to impart prior knowledge of how the agent should act.
    - Rewards are computed in the environment rather than in the agent.
        - Agent's goal should be to maximize a quantity over which it has imperfect control.
- *Returns*
    - **Return –** a specific function of the reward sequence $\{r_t\}$. The agent seeks to maximizes the expected return. The following is the general formulation of the (discounted) reward.

$$R_t = \sum_{k=0}^{T} \gamma^k r_{t+k+1}$$

$$0 \le \gamma \le 1$$

- T can be infinite (continuing task) or the *discount rate* γ can be 1 (episodic task) but not both.
- If $\gamma = 0$, the agent is **myopic (greedy)** – only maximizes immediate reward.
- If $\gamma \rightarrow 1$, the agent becomes farsighted and approaches the case of a simple sum of the rewards (episodic).
- If r_t is bounded by r_{max} and $\gamma < 1$, R_t is bounded:

$$R_t \le \frac{r_{\max}}{1-\gamma}$$

- **Continuing Tasks –** an agent-environment interaction that continues without limit.
- **Episodic Tasks –** an agent-environment interaction that can be decomposed into subsequences of repeated elements, *episodes*. Each episode has a special *terminal state* followed by a restart in one of the environment's start staes. Given episode *i* that has elapsed for time *t*, we index it's states, actions, etc. with a index pair *(t,i)*. e.g. $a_{t,i}$.
  - Can be represented as a continuing task by making rewards after the terminal state 0 and making the terminal state an *absorbing state* that transitions only back to itself.

- *The Markov Property*
  - State s_t is a representation of information available to the agent at time *t*. There is no reason to limit state only to the immediate sensations but rather, in general, state should incorporate all relevant knowledge available.
  - Not all relevant or useful information is available to the agent – hidden state information.
  - Ideally, the agent should never forget relevant info, so the state must be able to represent the past compactly yet retain all the relevant parts.
  - **Markov State** – a state that retains all relevant information from the past. This induces an independence of path assumption since only current state signal is relevant:

$$p\left(s_{t+1} = s', r_{t+1} = r \mid \{s_i, a_i, r_i\}_{i=0}^{t}\right) = p\left(s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\right)$$

    - Allows us to simply compute the next state and expected reward given the current state and action.
    - Given a Markovian system, the Markov states are the best possible basis for choosing actions.
    - Even for non-Markovian systems, assuming it is Markovian often leads to good predictions and action choices.
  - **(finite) Markov Decision Process (MDP) –** a reinforcement learning task that satisfies the Markov Property. If state and action spaces are finite, the MDP is finite. The MDP simply specifies **transition probabilities**:
    - Given any state s and action a, the probability of the next state being s' is,

$$P_{ss'}^{a} = p\left(s_{t+1} = s' \mid s_t = s, a_t = a\right)$$

- Given current state s and action a and the next state s', the expected reward is,

$$R_{ss'}^{a} = \mathrm{E}\left[r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right]$$

- A *transition graph* can summarize the dynamics of a finite MDP
  - state nodes – open circles representing a state
  - action nodes – closed circles corresponding to a state-action pair.
  - transitions – each transition goes from a state *s* to a state *s'* through an action for state *s*. It is labeled by $P_{ss'}^{a}$ and $R_{ss'}^{a}$.

- *Value Functions – estimates of the desirability of a state (or state-action)*
  - value functions define a partial ordering over policies; that is $\pi \le \pi'$ iff $\forall s \in S \quad V^{\pi}(s) \le V^{\pi'}(s)$.
    - **optimal policy π^*** - a (set of) policies such that $\pi \le \pi^*$ for all other policies π.
  - **state-value function** $V^{\pi}(s)$ – the expected return an agent following policy π has in state *s*:

$$V^{\pi}(s) = \mathrm{E}_{\pi}\left[R_t \mid s_t = s\right] = \mathrm{E}_{\pi}\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s\right]$$

    - *Bellman equation:*

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P_{ss'}^{a}\left[R_{ss'}^{a} + \gamma V^{\pi}(s')\right]$$

    - *optimal state-value function:*

$$V^{*}(s) = \max_{\pi} V^{\pi}(s)$$

    - *Bellman optimality equation:*

$$V^{*}(s) = \max_{a} \sum_{s'} P_{ss'}^{a}\left[R_{ss'}^{a} + \gamma V^{*}(s')\right]$$

  - **action-value function** $Q^{\pi}(s,a)$ – the expected return an agent following policy π has for taking action *a* in state *s*:

$$Q^{\pi}(s,a) = \mathrm{E}_{\pi}\left[R_t \mid s_t = s, a_t = a\right] = \mathrm{E}_{\pi}\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right]$$

    - *Bellman equation:*

$$Q^{\pi}(s,a) = \sum_{s'} P_{ss'}^{a}\left[R_{ss'}^{a} + \gamma \sum_{a'}\left[\pi(s',a') Q^{\pi}(s',a')\right]\right]$$

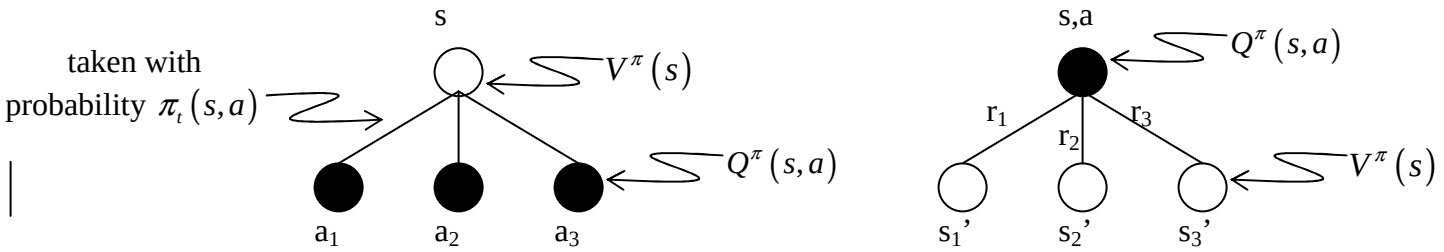
    - *optimal state-value function:*

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a)$$

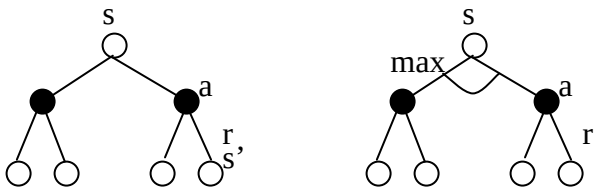
    - *Bellman optimality equation:*

$$Q^{*}(s,a) = \sum_{s'} P_{ss'}^{a}\left[R_{ss'}^{a} + \gamma \max_{a'} Q^{*}(s',a')\right]$$

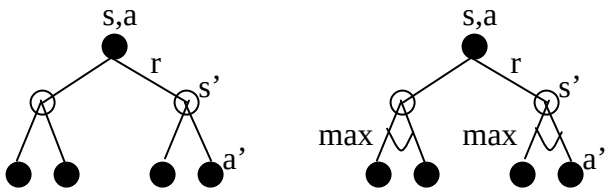
- **Backup Diagrams –** show how an update (backup) operation transfers value information back to a state (action-state) from future plausible actions/states.
  - **Full Backup –** each iteration of the iterative policy evaluation backs up the value of every state once to produce a new approximation V_{k+1} for each. e.g. DP methods use full backups.
  - **Sample Backup** – considers only a single sample successor rather than the complete distribution of successors in doing value estimates. e.g. MC and TD methods use sample backups.



  - backup diagram for V^π and V^*:



  - backup diagram for Q^π and Q^*:



- Solving Bellman optimality equations
  - For finite MDPs, Bellman optimality of $V^\pi(s)$ has a unique solution in that for N states, we have N equations in N unknowns; hence, we can solve with a nonlinear equations solver.
  - *Given we can solve for $V^*(s)$, any policy that is greedy w.r.t. $V^*(s)$ is an optimal policy since V accounts for future rewards.*
  - In practice such solutions require 3 assumptions:
    - We accurately know the dynamics of the environment
    - We have the computational resources for the system
    - The Markov Property is applicable.
  - In most applications, solutions can only be approximated as one or more of these assumptions is violated.

        - Often there are so many states/actions that a tabular representation of state and state-action functions are infeasible and parameterized functions must be used to approximate.
- **Prediction** – the problem of predicting the value of states and actions, which are used to produce an optimal policy. Three different general approaches are considered:
    - Dynamic Programming – uses Bellman optimality taking an expectation over all possible actions possible from a state… a bootstrapping approach.
    - Monte Carlo – uses the law of averages to approximate the value-function using sampling.
    - Temporal Difference – uses a combination of bootstrapping and sampling to perform prediction.

4: Dynamic Programming

- **Dynamic Programming –** a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a *Markov Decision Process* (MDP) by utilizing the process of *generalized policy iteration* (GPI).
    - *Requires a model of the environment*:
        - next-state probability distribution: $P^a_{ss'}$
        - reward probability distribution: $R^a_{ss'}$
- **Policy Evaluation (Prediction) –** the process of computing $V^\pi(s)$ for all states *s*, for a particular policy π.

  $$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right]$$

    - **Iterative Policy Evaluation –** calculates successive approximations V_{k+1} from the previous value of V_k such that $V_k \to V^\pi$ as $k \to \infty$ by the following update:

      $$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_k(s') \right]$$

        - **In place algorithm –** only a single array is used for V_k and V_{k+1}, thus using new values instead of old ones for the backups of some states; still converges, sometimes even faster than full backup.
    - **Termination –** test the quantity $\max_{s \in S} \left| V_{k+1}(s) - V_k(s) \right|$ after each sweep and stop when it is sufficiently small.
    - ***bootstrapping* –** the process of using previous value estimates for *other* states to update the value-state estimates of a state. This is the core of policy evaluation.
- **Policy Improvement –** The process of finding better policies based on the value function $V^\pi(s)$ by making the policy greedy with respect to $V^\pi(s)$.

  $$Q^\pi(s,a) = \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right]$$

    - **<u>Policy Improvement Theorem</u>** – *Let π and π' be any pair of deterministic policies such that, for all $s \in S$,*

      $$Q^\pi(s, \pi'(s)) \geq V^\pi(s).$$

      Then the policy π' must be as good as, or better than π. That is, it must obtain greater than or equal expected return from all states $s \in S$:

      $$V^{\pi'}(s) \geq V^\pi(s)$$

      Moreover, if there is strict inequality in the first equation for any state, there must be strict inequality in the second for one or more states. Thus if $Q^\pi(s,a) > V^\pi(s)$, the changed policy is indeed better than π.

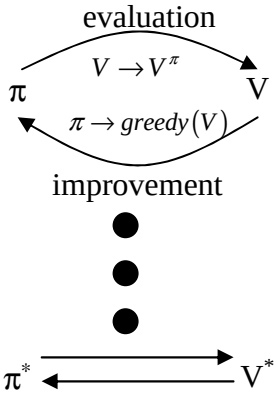
- Policy improvement considers changes at all possible states to all possible actions and selects a new greedy policy by the following:

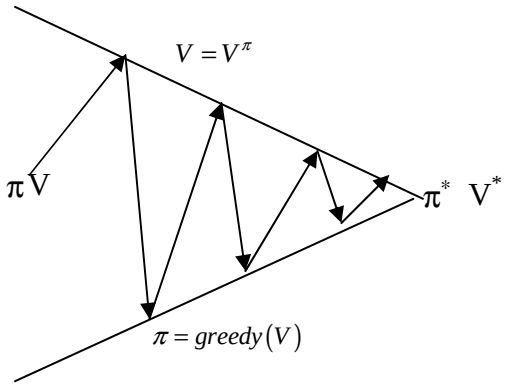
$$\pi' = \arg\max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V^\pi(s')\right]$$

  - If the new policy π' is as good but not better than the old policy π than $V^{\pi'} = V^\pi$ ➔ the policy is optimal.
- Policy improvement can also be extended to stochastic policies:

$$Q^\pi(s, \pi'(s)) = \sum_a \pi'(s,a) Q^\pi(s,a)$$

- **Generalized Policy Iteration –** *the process of using interacting policy evaluation and policy improvement processes to achieve Bellman optimality.*
  - The value function stabilizes only when it is consistent with the current policy while the policy stabilizes only when it is greedy with respect to the current value function. Evaluation and Improvement thereby act orthogonally; making the policy greedy w.r.t. the value function makes the value function incorrect while making the value function consistent with the policy makes the policy non-greedy. *Together the two processes achieve overall optimality though neither attempts optimality directly!!!*





  - **Policy Iteration** - Uses alternating full policy evaluation and policy improvement until convergence occurs.
  - **Value Iteration** - Uses alternating policy evaluation and policy improvement, but only a single sweep of each during each pass.
    - $V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V_k(s')\right]$
    - still guaranteed to converge to V^* under the same conditions that guarantee existence of V^*.
    - Faster convergence often achieved by using multiple sweeps of policy evaluation, although not necessarily complete evaluation.
  - **Asynchronous Dynamic Programming –** in-place DP algorithms that back up the value of states in any sequence of policy evaluations and improvements.
    - For $0 \le \gamma < 1$, asymptotic convergence to V^* is guaranteed only if all states occur in the sequence $\{s_k\}$ an infinite number of times.

        - Allows us to run an iterative DP algorithm at the same time that an agent is actually experiencing the MDP. Thus, agent's experience can actually be used to determine which states are backed up.
- *Efficiency of Dynamic Programming*
    - For n states and m actions, a DP method is guaranteed to converge to an optimal policy in polynomial time even though the total number of (deterministic) policies is m^n.
    - Linear Programming can be used for MDP's but is intractable for a large number of states.
    - Dynamic Programming still limited by the *curse of dimensionality* – the number of states often grows exponentially with the number of state variables.

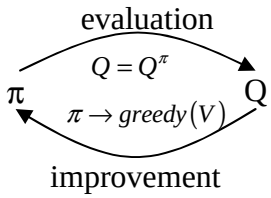
5: Monte Carlo Methods

Monte Carlo – algorithms that estimate value function and optimal policies based solely on experience; that is, sample sequences of states actions and rewards from an on-line or simulated interaction with the environment. Method based on the concept that as more returns are accumulated, the average should converge to the expected value.

- Backup Diagram – only a single path over an entire episode occurs.

- Advantages over DP:
  - Only requires sample episodes rather than a model of the environment.
  - No Bootstrapping – estimates for each state are independent.
  - The computational expense of the estimate for each state is independent of the number of states.
  - Can concentrate on a small subset of states to estimate.
  - Less harmed by violations of the Markov assumption *because they don't bootstrap*.
- MC averaging strategies
  - **every visit MC** – estimates $V^{\pi}(s)$ as the average of returns following all visits to state *s* in a set of episodes.
  - **first-visit MC** – estimates $V^{\pi}(s)$ just as the average of returns following first visits to state *s*.
  - Both strategies converge to $V^{\pi}(s)$ as the number of visits to *s* goes to infinity. Each average is an unbiased estimate whose standard error ~ $1/\sqrt{n}$ where *n* is the number of samples.
- MC Estimation of Action values
  - Without a model, state values are not sufficient to determine a policy.
  - Values $Q^{\pi}(s,a)$ are estimated from paths starting at state *s*, taking action *a*, and following policy π thereafter.
    - Both every-visit MC and first-vist MC converge quadratically.
  - **Maintaining Exploration**
    - If π is deterministic, many state-action pairs won't be visited but in order to compare, we need estimates for all actions from a state.
    - **exploring starts** - ensuring that each state-action pair is started from with some non-zero probability

        - alternatively, we maintain a stochastic policy with non-zero probabilities of selecting each state.
- MC Control – GPI



$\pi_0 \longrightarrow Q^{\pi 0} \longrightarrow \pi_1 \longrightarrow Q^{\pi 1} \quad \ldots \quad \pi^* \longrightarrow Q^*$

    - Classical Policy Iteration
        - Assumes (1) exploring starts (2) an infinite number of episodes per evaluation step
        - Under these assumptions the policy improvement theorem applies:
          $$Q^{\pi_k}\left(s, \pi_{k+1}(s)\right) \geq Q^{\pi_k}\left(s, \pi_k(s)\right) = V^{\pi_k}(s)$$
          and equality is only achieved when both π_{k+1} and π_k are optimal.
        - Hence, the iterations of evaluation and improvement lead to an optimal policy.
        - *However, the assumptions are unachievable.* One can approximate Q^{π_k} by bounding the error of the estimate in policy evaluation, but this requires too many episodes in practice.
    - Value Iteration
        - Only a single iteration of policy evaluation is done while alternating between evaluations and improvements.
        - **Monte Carlo ES** – value iteration using the exploring starts.
            - Cannot converge to a suboptimal policy
            - Stability is only achieved for optimal value function and policy.
            - However, convergence to optimal fixed point is unproved.
- Exploration vs. Exploitation:
    - *on-policy approach* – agent commits to always exploring, but tries to find the best policy that still explores
    - *off-policy approach* – agent explores with one policy but learns a deterministic optimal policy possibly unrelated to the exploring one.
- **On policy** – methods that attempt to evaluate or improve the policy that is used to make the decisions for sampling in a Monte Carlo technique.
    - **soft policy** - $\forall s \in S, a \in A \quad \pi(s,a) > 0$.
        - e.g. ε-greedy policy
    - GPI does not require the policy be strictly greed, only that it is moved toward a greedy policy ➔ move toward an ε-greedy policy.
    - Again, we have by the policy improvement theorem that
      $$Q^{\pi_k}\left(s, \pi'(s)\right) \geq V^{\pi}(s)$$

while again implies that $\pi' \geq \pi$. Moreover, equality only occurs when both are optimal.

- It can be shown that policy iteration works for an ε-soft policy π.

- **Off policy** – methods that use one policy for Monte-Carlo sampling to evaluate and improve a separate policy.
  - **behavior policy π'** – the policy used to generate the sampling behavior. The behavior policy must be *soft*.
  - **estimation policy π** – the policy that is evaluated and improved.
  - Under the condition that every action taken under π is taken at least occasionally under π'; $\pi(s,a) > 0 \quad \Rightarrow \quad \pi'(s,a) > 0$, we have…
    - Let $p_i(s)$ and $p_i'(s)$ be the probability of that complete sequence occurring under policies π and π', respectively, starting in state *s*:

$$V(s) = \frac{\sum_{i=1}^{n} \frac{p_i(s)}{p_i'(s)} R_i(s)}{\sum_{i=1}^{n} \frac{p_i(s)}{p_i'(s)}} \qquad \frac{p_i(s_t)}{p_i'(s_t)} = \prod_{k=t}^{T_i(s)-1} \frac{\pi(s_k, a_k)}{\pi'(s_k, a_k)}$$

  - Problem: method learns only from the *tails* of episodes; after the last nongreedy action (have 0 probability ➔ have 0 weights). If nongreedy actions are frequent, learning is slow.

- **Incremental Updates** – Monte Carlo updates can also be done incrementally.
  - Difference between MC and bandit problems
    - Monte Carlo typically involves multiple situations
    - Monte Carlo typically returns nonstationary distributions.
  - Monte Carlo weighted average:

$$V_n = \frac{\sum_{k=1}^{n} w_k R_k}{\sum_{k=1}^{n} w_k}$$

  - Incremental weighted average:

$$V_{n+1} = V_n + \frac{w_{n+1}}{W_{n+1}} \left[R_{n+1} - V_n \right]$$

$$W_{n+1} = W_n + w_{n+1}$$

$$V_0 = W_0 = 0$$

- constant-α MC:
  - once the final reward R_t is received for the episode:

$$V(s_t) = V(s_t) + \alpha \left[R_t - V(s_t) \right]$$

6: Temporal-Difference Learning

Temporal Difference (TD) Learning – learning both from raw experience (no model) as in MC, but estimate updates depend on other learned estimates as well as in DP. As with previous methods, control is done by means of GPI.

- **bootstrapping** – a method that uses previous estimates in updating an estimate.
- **temporal difference** – each error is proportional to the change in time of the prediction.
- TD(0)
  - $V(s_t) = V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$
  - Backup Diagram 
- Advantages of TD Prediction
  - Do not require an environment model
  - Naturally implemented in on-line, fully incremental fashion
    - MC not well suited to very long episodic or continuing tasks.
  - Proven to converge to V^π for any fixed policy π given that step-size is small enough or decreases according if their sum grows unbounded by the sum of their squares is finite (see Chapter 2).
    - TD methods usually converge faster than constant-α MC methods on stochastic tasks.
- Optimality of TD(0)
  - **batch updates** – updates are made after processing a "batch" of data.
  - *Batch Monte Carlo methods always find the estimates that minimize mean-squared error on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the maximum-likelihood model of the Markov process.*
  - **certainty-equivalence estimate** – the estimate of the value function that would be exactly correct if the model were exactly correct.
    - Batch TD(0) converges to certainty-equivalence estimate.
    - TD methods require N (number of states) memory on repeated computation, while finding the exact estimate may require N^2 memory and its value function N^3 computation.
- **Sarsa**: On-Policy TD Control
  - Sarsa uses every element of the tuple $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ in estimating $Q^\pi(s,a)$:

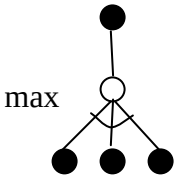
    $$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]$$

  - We continually estimate Q^π for the behavior policy π, and simultaneously change π to be greedy with respect to Q^π.

        - Step-by-step learning methods will avoid non-terminating episodes since the policy is adjusted *during the episode*.
    - Sarsa converges with probability 1 to an optimal policy/action function if:
        - all state-action pairs visited infinitely often
        - policy converges to a greedy policy (e.g. ε-greedy w/ ε=1/t).
- Q-Learning: Off-Policy TD Control
    - one-step Q-learning

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t) \right]$$

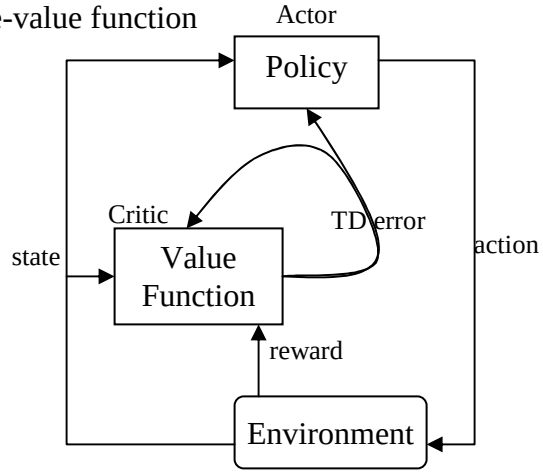
        - learned action-value function Q directly approximates Q^*.
        - Backup Diagram



    - Exercise Update Rule**
        - max of next state-action pairs replaced by their expectation

$$Q(s_t,a_t) = Q(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \sum_a \pi(s_t,a) Q(s_{t+1},a) - Q(s_t,a_t) \right]$$

- **Actor-Critic Methods** – an on-policy learning technique where policy is explicitly represented.
    - **actor** – policy structure used to select actions
    - **critic** – estimated value function used to criticize actions of the actor… typically the state-value function



    - critic's evaluation is ***TD error***:

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

        - if error is positive, tendency to select action a_t increased.
        - if error is negative, tendency to select action a_t decreased.
    - Gibbs softmax policy:

$$\pi_t(s,a) = P\{a_t = a \mid s_t = s\} = \frac{e^{p(s,a)}}{\sum_b e^{p(s,b)}}$$

        - preference function $p(s,a)$ - the preference for action *a* in state *s*.
            - update rules for the preference function
                - $p(s_t,a_t) = p(s_t,a_t) + \beta\delta_t$
                - $p(s_t,a_t) = p(s_t,a_t) + \beta\delta_t\left(1-\pi_t(s_t,a_t)\right)$
    - Advantages of Actor-Critic
        - Since policy is stored explicitly, no computation for action selection
        - Capable of learning optimal probabilities for an explicit stochastic policy.
- **R-Learning** – an off-policy control method in which one neither discounts nor divides experience into distinct episodes. The learner seeks to obtain the *maximum reward per time-step*.
    - average expected reward per time step under policy π:

$$\rho^\pi = \lim_{n\to\infty}\frac{1}{n}\sum_{t=1}^{n} E_\pi[r_t]$$

        - assumes process is *ergodic*: nonzero probability of reaching any state from any other state under the policy.
        - ρ^π does not depend on the starting state.
    - Relative Values:
        - state-value function

$$\tilde{V}^\pi(s) = \sum_{k=1}^{\infty} E_\pi\left[r_{t+k} - \rho^\pi \mid s_t = s\right]$$

        - state-action value function:

$$\tilde{Q}^\pi(s,a) = \sum_{k=1}^{\infty} E_\pi\left[r_{t+k} - \rho^\pi \mid s_t = s, a_t = a\right]$$

    - R-Learning is standard TD control
        - behavior policy and estimation policy π
        - action-value function *Q*: approximation of $\tilde{Q}^\pi$
        - estimated average reward ρ: approximation of ρ^π
- **afterstate** – in many situations, it is convenient to evaluate the state after the agent has made an action. Such states are known as afterstates with their corresponding afterstate values.
    - Convenient due to the fact that many state-action sequences result in the same new state.

8: Generalization and Function Approximation

generalization – How can experience with a limited subset of the state space be used to generalize to a good approximation over a much larger subset?

- most states experienced will never be re-experienced exactly as before in realistic settings.

function approximation – taking examples from a desired function and attempting to generalize to construct an approximation of that function… a supervised learning task.

- V_t is no longer a table, but rather a function parameterized by vector $\vec{\theta}_t$ where the number of parameters is typically far fewer than the number of states.
- Prediction methods produce backups of the form $s \mapsto v$; s is the state and v is the back-up value for that state.
    - In functional approximation, the back-ups, $s \mapsto v$, are passed to the supervised learner as training examples for producing the estimated value function.
        - Must be able to occur on-line while interacting with the environment.
        - Must be able to handle nonstationary target function.
- Performance measure
    - **Mean-squared error (MSE)** of approximating V_t with parameter $\vec{\theta}_t$:

$$MSE\left[\vec{\theta}_t\right] = \sum_{s \in S} P(s)\left[V^{\pi}(s) - V_t(s)\right]^2$$

        - P is a distribution weighing the error of states; typically the distribution from which states in training examples are drawn.
        - **on-policy distribution** – the frequency with which states are encountered while agent is interacting with the environment.
        - **global optimum** $\vec{\theta}^*$ - a parameter vector such that $\forall \vec{\theta} \quad MSE\left[\vec{\theta}^*\right] \le MSE\left[\vec{\theta}\right]$
        - **local optimum** $\vec{\theta}^*$ - a parameter vector such that $MSE\left[\vec{\theta}^*\right] \le MSE\left[\vec{\theta}\right]$ for all $\vec{\theta}$ in some neighborhood of $\vec{\theta}^*$.
- **state aggregation** – states are grouped together with one table entry per group.

Gradient Descent – the parameter vector is adjusted after each example by a small amount in the direction of the negative gradient of the example's squared error – the direction in which error decreases most rapidly.

- Traditional Gradient Descent
  - parameter vector is a column vector with fixed number of components:
    $$\vec{\theta}_t = \left[\theta_{t,1}, \ldots, \theta_{t,n}\right]^T$$
  - $V_t(s)$ is a smooth differentiable function of $\vec{\theta}_t$
  - At each time step, we observe an example $s_t \mapsto V^\pi(s_t)$
  - Gradient Descent update:
    $$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \left[V^\pi(s_t) - V_t(s_t)\right] \nabla_{\vec{\theta}_t} V_t(s_t)$$
    where the gradient is,
    $$\nabla_{\vec{\theta}_t} V_t(s_t) = \left[\frac{\partial f(\vec{\theta}_t)}{\partial \vec{\theta}_{t,1}}, \ldots, \frac{\partial f(\vec{\theta}_t)}{\partial \vec{\theta}_{t,n}}\right]^T$$
  - If the step-size parameter satisfies the stochastic approximation conditions, then gradient descent will converge to a local optimum.
- Unbiased Estimate
  - If v_t is an *unbiased* estimate of $V^\pi(s_t)$, $E[v_t] = V^\pi(s_t)$, for each *t*, then $\vec{\theta}_t$ is guaranteed to converge to a local optimum if the step size satisfies the stochastic approximation conditions in the following update:
    $$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \left[v_t - V_t(s_t)\right] \nabla_{\vec{\theta}_t} V_t(s_t)$$
- *n*-step TD returns
  - Forward-view update:
    $$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \left[R_t^\lambda - V_t(s_t)\right] \nabla_{\vec{\theta}_t} V_t(s_t)$$
    - For $\lambda<1$, R_t^λ is not an unbiased estimate of $V^\pi(s_t)$.
  - Backward-view:
    $$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \delta_t \vec{e}_t$$
    - TD error:
      $$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$
    - column of eligibility traces:
      $$\vec{e}_t = \gamma\lambda \vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t)$$
- Common function approximation methods using gradient-descent:
  - Multilayer Neural Network with back-propagation.
  - Linear Form

Linear Methods – V_t is a linear function of the parameter vector $\vec{\theta}_t$ and a feature vector $\vec{\phi}_s = \left[\phi_{s,1}, \ldots, \phi_{s,n}\right]^T$ describing the the state *s*.

- value function: $V_t(s) = \left\langle \vec{\theta}_t, \vec{\phi}_s \right\rangle = \sum_{i=1}^{n} \theta_{t,i} \phi_{s,i}$
- gradient: $\nabla_{\vec{\theta}_t} V_t(s_t) = \vec{\phi}_s$
- *There is only one optimal value, $\vec{\theta}^*$, the global optimum. Hence, any method guaranteed to converge to a local optimum will converge to the global one.*
    - The TD(λ) gradient descent method will converge within a factor of the global minimal error:

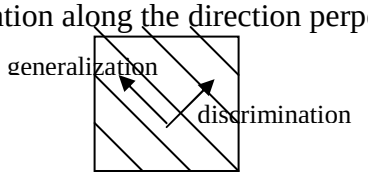
$$MSE\left[\vec{\theta}_\infty\right] \leq \frac{1-\gamma\lambda}{1-\gamma} MSE\left[\vec{\theta}^*\right]$$

- Determining the features that represent a state adds a degree of prior knowledge.
    - Linear forms prohibit interactions between features, hence one needs to explicitly introduce extra features for conjunctions of feature values when using linear function approximation methods.

Feature Methods:

- **receptive field** – the region defining a feature in state space.
- **binary feature** – a feature with a 0-1 value representing whether the state is contained within the feature's region.
- **Coarse Coding** – represents a state by overlapping features such that observations of a state affect the approximate value functions for all states in the union of the feature's that contain the observed state.
    - Small features – generalization occurs over a short distance making functional approximations bumpy.
    - Large features – generalization over large distances causing approximations to be smoother.
    - *Initial generalization from one point to another is controlled by the size and shape of receptive field, but **acuity**, the finest discrimination possible, is controlled more by the total number of features.*

- **Tile Coding** – features are grouped into a *tiling*, an exhaustive partitioning of the input space, where each tile is a receptive field for 1 binary feature.
    - since exactly 1 feature is present in each tiling, the total number of features is simply the number of tilings.
        - given the coordinates of a point in space, it is easy to determine the index of its tiling.
        - multiple tilings are offset by different amounts.
        - width and shape of tiling chosen to match generalization width.
        - number of tiles ~ resolution of final approximation.
    - *stripe tiling* – promotes generalization in the direction of the strips and discrimination along the direction perpendicular to the strips.



    - axis-aligned tilings correspond to ignoring some dimensions in some of the tilings to hyperplane slices.
    - **Hashing** – a pseudo-random collapsing of a large tiling into a much smaller set of tiles. These tiles are noncontiguous disjoint regions "randomly" spread throughout the space.
        - hashing reduces effect of curse of dimensionality.
- **Radial Basis Functions** – Features have a Gaussian response dependent only on the distance between the state *s* and the feature's center state s_i.

$$\phi_s(i) = \exp\left\{-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right\}$$

    - **RBF network** – a linear function approximator using RBF features.
        - function varies smoothly and is differentiable.
        - requires greater computation effort and more manual tuning.
- **Kanerva Coding** – each feature corresponds to a prototypical state and its receptive field is defined as the states sufficiently close in *Hamming distance* – the number of bits (dimensions) on which the two states disagree.
    - Other features choices tend to have exponentially many features in the dimensions of the space.
        - In the worse case, exponentially many features are required.
        - However, often desirable functions lie in a subspace!
    - The complexity of the learnable function for Kanerva coding depends entirely on the number of features rather than the number of dimensions!

Control with Functional Approximation

- Extending to GPI
  - *Policy Evaluation*
    - functional approximations target the action-value function, $Q_t \approx Q^\pi$, and are trained by examples $s_t, a_t \mapsto v_t$ where v_t is any approximation of $Q^\pi(s_t, a_t)$ by previously discussed methods.
    - $\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha_t \left[v_t - Q_t(s_t, a_t) \right] \nabla_{\vec{\theta}_t} Q_t(s_t, a_t)$
  - *Policy Improvement* – changes the estimation policy to the greedy policy
    - Suitable techniques for continuous actions or a large discrete set of actions is ungoing research
    - For manageable sets of discrete actions, the greedy policy for a state can be found by brute force,
      $$a_t^* = \arg\max_a Q(s_t, a)$$
  - Using replacing traces is difficult since no single trace corresponds to a state.
    - For gradient-decent linear approximations, treating each feature as a state w.r.t. replacing traces works well.
    - Often useful to clear all traces of all features for state and action not selected and set the trace of the selected state-action to 1.
  - *Exploration can be assured through either on-policy or off-policy.*
  - Algorithms for Sarsa(λ) and Q(λ) on pages 212-213 of book.
- **Bootstrapping** – updating of a value estimate on the basis of other value estimates.
  - **off-policy bootstrapping** – bootstrapping with a different distribution (*estimation policy*) for backups than the one used for exploring the state-space (*behavior policy*).
    - *In functional approximation, off-policy bootstrapping can lead to divergence and infinite MSE.*
    - Heuristic: keep the behavior policy sufficiently close to the estimation policy.
    - Stability is guaranteed if the value function is updated to the best, least-squares approximation if the feature vectors form a linearly independent set.
    - Stability is guaranteed for functional approximators called *averagers* that do not extrapolate from observed targets (e.g. Nearest-Neighbor, local weighted regression).
  - In practice bootstrapping methods are preferred since their performance is much better despite their higher asymptotic error and potential for instability.

11: Case Studies

TD-Gammon – A backgammon program by Tesauro that used the TD(λ) algorithm and nonlinear function approximations via a multilayer neural net trained by backpropagating TD errors. The player required little backgammon knowledge but learned to play extremely well.

- Rules
  - Board made of 24 *points*. White moves counterclockwise, Black moves clockwise. Objective is to get all pieces off the board.
  - Rolls of die allow selection of a piece to move by the distance rolled on each die.
  - Moving onto a *point* occupied by a single opponent piece is a "*hit*". The opponents piece is placed on the *bar* then returned to the beginning.
  - If 2 or more pieces occupy a *point*, the opponent cannot move a piece there.
  - A major strategy is to form contiguous blocks of occupied *points*.
- Characteristics of Backgammon
  - The game is highly stochastic, but a complete description of the game's state is available at all times.
  - The number of states is too large for a look-up table.
  - The opponent serves as a source of uncertainty.
- Design
  - Value function, $V_t(s)$, computed by a neural net to estimate the probability of winning starting from state *s*.
    - Also compute the probability of winning by means of a "backgammon".
    - Input was a representation of a backgammon position and the output was an estimate of the value of that position.
  - Tesauro represented state in a straightforward way. One unit was provided for each conceptually distinct possibility that seemed relevant and scaled to be between 0 and 1. Positions were afterstates.
  - TD-Gammon used the gradient descent form of TD(λ) using backpropagation.
  - TD-Gammon learned by playing against itself.
- Implementation:
  - TD-Gammon 0.0 – no previous knowledge and became competitive with other backgammon programs (with previous knowledge)
  - TD-Gammon 1.0 – specialized backgammon features incorporated.
  - TD-Gammon 2.0 & 2.1 – used a 2-ply look ahead.
  - TD-Gammon 3.0 - used a 3-ply look ahead.
- TD-Gammon learned to play some opening positions better than the previous strategies developed by humans.

Samuel's Checkers – used a form of temporal-difference learning to make an effective use of heuristic search through the state space of the game checkers – terminal board positions are scored by a value function, a "scoring polynomial".

- *Shannon's minimax procedure* – machine always tries to maximize score while opponent attempts to minimize it. Sophisticated search techniques use this to prune the search space in *alpha-beta cutoffs*.
- *rote learning* – saving a representation of a board along with its computed backed-up value as determined by the minimax procedure → caching.
- "sense of direction" – decrease in a position's value each time it is backed up a ply (turn). Thus, the learner will choose the low-ply alternative while winning and the high-ply alternative if losing.
- *learning by generalization* – program played against itself and performed a backup operation after each move.
    - Close to the central idea of temporal difference learning – the value of the state should equal the value of likely
- No explicit reward; instead the weight of the feature *piece advantage* was fixed. Thus, the goal of the program became to improve its piece advantage.
    - Since it was not constrained to find a useful evaluation function, it was possible for it to become worse with experience → evaluation functions could be made consistent while having nothing to do with winning or losing.
- The checker's player developed a good middle game but remained weak in opening and endgame play.

The Acrobat – A 2 link robot analogous to a gymnast on a high bar. The first joint cannot exert torque while the 2nd can. One objective is to swing the tip above the first joint by the length of 1 of its links in minimal time.

- Actions – either positive, negative, or 0 torque. Only a single value of positive/negative torque was allowed.
- Trained with *Sarsa(λ)* with linear function approximation via tile coding and replacing traces.
    - state – a direct representation of the position and velocities of the 2 links. A more clever representation might have been the angular position and velocity of the 2nd link.
    - *Tiling* divided the 4 dimensional state space in various ways resulting in over 25,000 tiles.
    - A greedy policy was used since long sequences of correct actions were required for success to occur. Exploration was maintained by starting action-values *optimistically* at the low value 0.

Elevator Dispatching – How can elevators be dispatched (and distributed while idle) to minimize the wait time of the customers.

- state – each elevator has a position, direction, speed along with a set of requests.
- System Metrics
  - **average waiting time** – how long the passenger waits before getting on.
  - **average system time** – how long the passenger waits before arriving.
  - percentage of passengers whose wait exceeds 60 seconds
  - **average squared waiting time** – tends to keep waiting times low while encouraging fairness of service.
- Constraints
  1. each elevator makes its decisions independently
  2. elevators cannot pass a requested floor
  3. elevators cannot reverse direction until all passengers have departed.
  4. elevator cannot stop at a floor unless requested.
  5. When given a choice, elevator constrained to go up (prevent rush hour from pushing all elevators to the lobby)
- Design
  - **semi-Markov decision process** – system makes discrete jumps between times when decisions had to be made.
  - Return is generalized to an integral of future rewards

    $$R_t = \int_0^{\infty} e^{-\beta\tau} r_{t+\tau} d\tau$$

    $r_{t+\tau}$ is the instantaneous reward

    $\beta > 0$ serves an analogous role as $\gamma \in [0,1]$
  - Semi-Markov backup for a tabular action-value function

    $$Q(s,a) = Q(s,a) + \alpha\left[\int_{t_1}^{t_2} e^{-\beta(\tau - t_1)} r_\tau d\tau + e^{-\beta(t_2 - t_1)} \max_{a'} Q(s',a') - Q(s,a)\right]$$
  - Action-value function represented by nonlinear neural net trained by backpropagation.
  - Actions are selected using the *Gibbs softmax* procedure and gradually raising the temperature
- Implementations
  - RL1 – each elevator is given its own action-value function and neural net.
  - RL2 – only 1 shared action-value function and neural net.

Dynamic Channel Allocation – How to make efficient use of available bandwidth to provide service to as many possible customers by taking advantage of the fact that a communication channel (band of frequencies) can be used simultaneously by several callers if they are physically spaced far enough apart not to interfere.

- Terminology
  - **channel** – a division of the bandwidth in a certain range of frequencies.
  - **channel reuse constraint** – the minimum distance at which there is no interference.
  - **cells** – a division of the service region.
  - **hand off** – the change of service for a call from cell to cell when a crossing a boundary.
  - **blocked call** – a call that can not be made when no channels are available.
- The core problem is to allocate channels with the goal of minimizing blocking.
  - **fixed assignment** – permanent assignment of channels to cells to assure that the channel reuse constraint is never violated.
  - **dynamic assignment** – all channels are available to all cells and assigned dynamically as call arrive.
- RL Solution
  - State – treated as an afterstate
    - usage state for each channel for each cell.
    - event indicator (arrival, departure, or handoff)
  - Reward
    - At time t, the immediate reward r_t, is the number of calls.
    - Return is,

$$R_t = \int_0^\infty e^{-\beta\tau} r_{t+\tau} d\tau$$

  - Value Function – a weighted sum of features.
    1. *availability feature* – number of additional calls that could be accepted without conflict assuming all other cells didn't change.
    2. *packing feature* – a cell-channel pair that gave the number of times the channel was being used in that configuration within 4 cells.
  - Simulation
    - 7x7 array of cells with 70 channels.
    - channel reuse constraint – 3 cells.
    - calls arrived randomly according to a Poisson possibly with different means for each cell.
    - call duration was exponential with a mean of 3 minutes.

Job-Shop Scheduling – create a schedule specifying when each task is to begin and what resources it will use that satisfies all the constraints while taking as little overall time as possible

- *Temporal Constraints* – some tasks have to be finished before others can begin.
- *Resource Constraints* – Tasks requiring the same resource cannot be executed simultaneously.
- *plan*-space – states are complete plans and actions are plan modifications.
- In general, Job-Shop Scheduling is NP-complete.
    - Not hard to find a conflict-free schedule, but the shortest is hard.
    - Realm of Combinatorial Optimization
- Zhang and Dietterich wanted an RL system to learn a policy that could quickly find good schedules for SSCP (Space Shuttle Payload Processing).
    - **iterative repair method** – begins with a critical path schedule and modifies it to remove resource constraint violations
        - **critical path schedule** s_0 – a schedule that meets the temporal constraints but ignores the resource constraints.
        - REASSIGN-POOL – changes the pool assigned to a task's resources.
        - MOVE – moves a task to an earlier or later time to satisfy a resource constraint, then reschedules by the critical path method.
    - episodic problem with rewards designed to promote quick (few repairs) conflict-free schedules of short duration.
        - Problem: The shortest schedule for a difficult instance (many tasks and constraints) will be longer than the shortest schedule for a simpler instance.
        - **Resource Dilation Factor (RDF)** – an instance-independent measure of a schedule's duration.
    - Methodology
        - TD(λ) used to learn the value function.
        - function approximation with a multilayer perceptron
        - actions selected by an ε-greedy policy.
        - TD(λ) algorithm used *backward* after each episode, with the eligibility trace extending to future rather than past states.
        - <u>experience replay</u> – replaying a remembered "best" episode during training.
        - <u>random sample greedy search</u> – estimates greedy action by considering only a random sample of actions.
        - TD-error used in back-propagation for neural net weight updates.
        - Later version used a time-delay neural net (TDNN) and a set of "kernel" networks to create more abstract features.
    - Results
        - Showed that the learning system produced scheduling algorithms that needed fewer repairs to find conflict-free schedules of similar quality as iterative repair algorithms.
        - Learned how to quickly find good schedules for a class of related scheduling problems.