Reinforcement Learning and Control

Workshop on Learning and Control
IIT Mandi

Pramod P. Khargonekar and Deepan Muthirayan

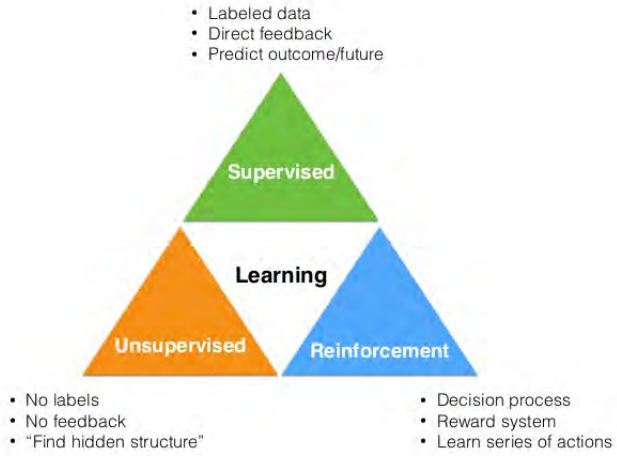
Department of Electrical Engineering and Computer Science
University of California, Irvine

July 2019

Outline

1. Introduction and History
2. RL Theoretical Foundations
   - Bellman's Principle of Optimality and Dynamic Programming
   - Monte-Carlo Methods
   - Temporal Difference Learning
   - Q Learning
3. Deep Reinforcement Learning
   - What is DRL?
   - DQN Achievements
   - Asynchronous and Parallel RL
   - Rollout Based Planning for RL and Monte-Carlo Tree Search
4. AlphaGo Zero
5. Recap and Concluding Remarks

Machine Learning



Source: Saagie

Some History

- 1950s-60s: Bellman - Dynamic Programming (DP)
- 1980s-early90s: Approximation and simulation-based methods: Barto/Sutton (TD(λ)), Watkins (Q-learning), Tesauro (backgammon, self-learning)
- 1990s: Rigorous mathematical understanding
- Late 90s-Present: Rollout, Monte-Carlo Tree Search, Deep Neural Nets, Deep RL

Source: Bertsekas

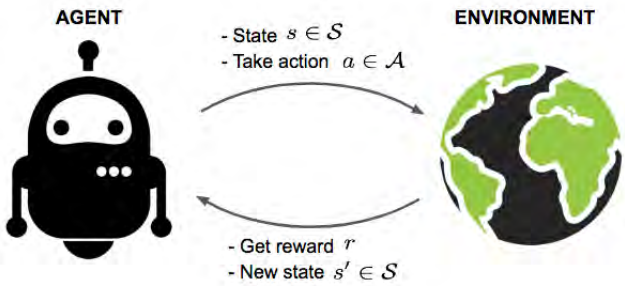
Why RL?

- A major direction in the current revival of machine learning for unsupervised learning
- Spectacular achievements of AlphaGo, AlphaGoZero, and AlphaZero
- Historical and technical connections to stochastic dynamic control and optimization
- Potential for new developments at the intersection of learning and control

Reinforcement Learning: Source Materials

- Book: R. L. Sutton and A. Barto, *Reinforcement Learning*, 1998 (2nd ed. on-line, 2018)
- Book, slides, videos: D. P. Bertsekas, *Reinforcement Learning and Optimal Control*, 2019.
- Monograph, slides: C. Szepesvari, *Algorithms for Reinforcement Learning*, 2018.
- Lecture slides: David Silver, *UCL Course on RL, 2015.*

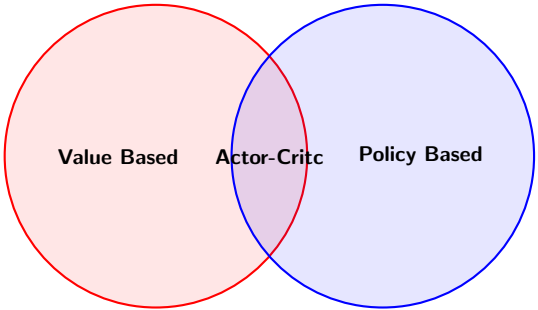
RL Framework

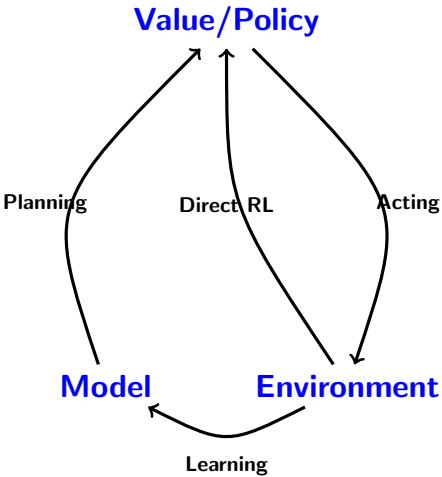


From control systems viewpoint, the "agent" is the controller and the "environment" includes the plant, uncertainty, disturbances, noise.

Source: github

Classification of Reinforcement Learning Algorithms





Key Topics

- Markov Decision Processes
- Dynamic Programming and Bellman's Optimality Principle
- Q-Learning
- Policy Gradient, Actor-Critic Framework
- Deep reinforcement learning
- Key innovations in modern RL
- AlphaGo, AlphaGoZero, AlphaZero, and latest achievements
- Future Directions

Reinforcement Learning and Dynamic Programming Terminologies

Reinforcement Learning

- Reward
- Value Function
- Agent
- Action
- Enviornment

Dynamic Programming

- Cost
- Cost Function
- Controller or Decision Maker
- Control
- System

Definitions and Framework

General Dynamical System

- $x(k+1) = f(x_k, u_k, w_k)$
  $y(k) = h(x_k, u_k, w_k)$
- k: Discrete time
- x_k: State; summarizes past information that is relevant for future evolution of the system
- u_k: Control; decision to be selected at time k from a given set
- w_k: Disturbance, noise

Markov Decision Process

- $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}_0)$
- $\mathcal{X}$: a countable, non-empty set of states
- $\mathcal{A}$: a countable, non-empty set of actions
- $\mathcal{P}_0$: *transition probability and reward kernel*
- $\mathcal{P}_0(.|x, a)$ = probability measure over $\mathcal{X} \times \mathbb{R}$ giving probability distribution of the next state and associated reward (which is in $\mathcal{X} \times \mathbb{R}$).
- Markov Decision Process is called finite if both $\mathcal{X}, \mathcal{A}$ are finite sets.

Markov Decision Process as a Sequential Decision System

- $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}_0)$ induces a natural sequential decision system
- $t =$ current time (discrete variable)
- $X_t \in \mathcal{X}$ is the current state and $A_t \in \mathcal{A}$ is the current action
- A sequence of states, actions and rewards ensues
- $(X_{t+1}, R_{t+1}) \sim \mathcal{P}_0(.|(X_t, A_t)$
- $\mathbb{P}_0(X_{t+1} = y | X_t = x, A_t = a) =: \mathcal{P}(x, a, y)$
- $\mathbb{E}\left[R_{t+1} | (X_t, A_t)\right] =: r(X_t, A_t)$

Discounted Rewards

- MDP leads to a sequence of states, actions, and rewards
- $X_0, A_0, R_1, X_1, A_1, R_2, \ldots X_{T-1}, A_{t-1}, R_t, \ldots$
- Let $\gamma < 1$ denote the discount factor.
- Total discounted reward is given by

$$\mathcal{R} = \sum_{t=0}^{\infty} \gamma^t R_{t+1} \tag{1}$$

- The discounted reward $\mathcal{R}$ is a random variable.
- The goal of the decision maker is to maximize expected value of $\mathcal{R}$.

Optimal Value Function and Stationary Policies - Definitions

- Let $x \in \mathcal{X}$ denote the initial state.
- The optimal value $V^*(x)$ is defined to be the highest achievable expected discounted reward.
- $V^* : \mathcal{X} \to \mathbb{R} : x \mapsto V^*(x)$ is called the *optimal value function*.
- A decision maker's (controller's) objective is to choose actions that leads to optimal value function starting from any initial state.
- Deterministic Stationary Policy: $\pi : \mathcal{X} \to \mathcal{A}$. Following π means choosing

$$A_t = \pi(X_t), \forall t \geq 0 \tag{2}$$

- Stochastic Stationary Policy: $\pi : \mathcal{X} \to \mathcal{P}(\mathcal{A})$. Following π means choosing

$$A_t \sim \pi(.|X_t), \forall t \geq 0 \tag{3}$$

Markov Reward Process

- A stationary control or decision policy leads to a Markov Reward Process (MRP) defined next.
- An MRP is specified by a pair $(\mathcal{X}, \mathcal{P}_0)$ where $\mathcal{P}_0$ assigns a probability distribution to each state in $\mathcal{X}$.
- We define a time-invariant Markov reward process $Z_t = (X_t, R_t), t \geq 0$ where $(X_t, R_t) \sim \mathcal{P}_0(.|X_{t-1})$.
- Now suppose we have an MDP $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}_0)$ and a stationary control policy π. Define

$$\mathcal{P}_0^\pi(.|x) = \sum_{a \in \mathcal{A}} \pi(a|x) \mathcal{P}_0(.|x, a) \tag{4}$$

- The MRP induced by the policy π is specified by $(\mathcal{X}, \mathcal{P}_0^\pi)$.

Value and Action-Value Function Definitions for Stationary Policies

- Let $\pi \in \Pi_{stat}$ be a stationary policy. Then the value function $V^\pi : \mathcal{X} \to \mathbb{R}$ is defined by

$$V^\pi(x) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | X_0 = x\right], x \in \mathcal{X} \tag{5}$$

  where R_t is the reward process resulting from application of the policy π.

- Action-value function is denoted by $Q^\pi : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ and is defined by

$$Q^\pi(x, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | X_0 = x, A_0 = a\right], x \in \mathcal{X}, a \in \mathcal{A} \tag{6}$$

Optimal Value and Value-Action Functions

Now we can define optimal value and action-value functions for the class of stationary policies:

$$V^*(x) = \sup_{\pi \in \Pi_{stat}} V^\pi(x) \tag{7}$$

$$Q^*(x,a) = \sup_{\pi \in \Pi_{stat}} Q^\pi(x,a) \tag{8}$$

It is not hard to see that V^* and Q^* are related as follows:

$$V^*(x) = \sup_{a \in \mathcal{A}} Q^*(x,a) \tag{9}$$

$$Q^*(x,a) = r(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,a,y) V^*(y) \tag{10}$$

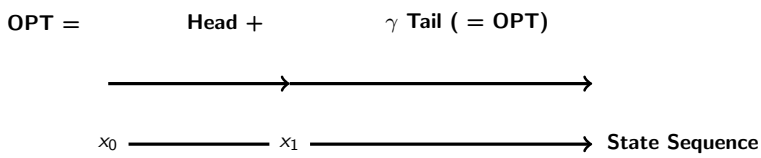
Bellman's Principle of Optimality and Dynamic Programming

Bellman's Principle of Optimality - Central to DP in Control

Principle of Optimality:

The *tail* of an optimal policy must be optimal.

- Richard E. Bellman



Bellman's Optimality Condition

- Bellman Equation

$$V^*(x) = \min_{a \in \mathcal{A}} \left[r(x,a) + \gamma \sum P(x,a,y) V^*(y) \right]$$

- Optimality condition - If $a = \pi(x)$ solves the Bellman equation then

$$V^{\pi}(x) = V^*(x)$$

Fundamental Bellman Equation and Bellman Operator

- Let us start with an MDP $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P}_0)$, a discount factor γ and a deterministic stationary policy $\pi \in \Pi_{stat}$. Let $r(x, a)$ be the reward function of $\mathcal{M}$. Using previous definitions, it follows that:

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi(x), y) V^\pi(y) \tag{11}$$

- Now we define the Bellman operator induced by a stationary policy π. It acts on a value function and produces another value function. Formally,

$$T^\pi \quad : \quad \mathbb{R}^{\mathcal{X}} \to \mathbb{R}^{\mathcal{X}} : V \to T^\pi V \tag{12}$$

$$(T^\pi V)(x) \quad = \quad r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi(x), y) V^(y) \tag{13}$$

- With this definition of the Bellman operator, we get a very compact equation for stationary policies:

$$T^\pi V^\pi = V^\pi \tag{14}$$

Properties of Bellman Operator

- Monotonicity property: For any functions V and V' on the state space $\mathcal{X}$ such that $V(x) \leq V'(x)$ for all $x \in \mathcal{X}$, and any policy π,

$$TV(x) \leq TV'(x), \ \forall \ x \in \mathcal{X} \tag{15}$$

$$T^{\pi}V(x) \leq T^{\pi}V'(x), \ \forall \ x \in \mathcal{X}$$

- Contraction property: With $\gamma < 1$, for any bounded functions J and J', and any π,

$$\max_x \left| TV(x) - TV'(x) \right| \leq \gamma \max_x \left| V(x) - V'(x) \right|$$

$$\max_x \left| T^{\pi}V(x) - T^{\pi}V'(x) \right| \leq \gamma \max_x \left| V(x) - V'(x) \right|$$

Bellman Operator for a Stationary Policy is a Contraction

We can now state a very simple yet fundamental theorem.

Theorem: With the notation and definitions above, (and assuming some technical conditions), for each stationary policy π, the Bellman operator T^π is an affine linear operator. If $0 < \gamma < 1$, and with the ℓ^∞ norm on $\mathbb{R}^{\mathcal{X}}$, T^π is a contraction, and the Bellman equation

$$T^\pi V = V \tag{16}$$

has a unique solution, namely V^π.

Thus, the Bellman operator equation for this fixed point becomes:

$$V^\pi(x) = r(x, \pi(x)) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x, \pi(x), y) V^\pi(y) \tag{17}$$

Bellman Optimality Equation and Bellman Optimality Operator

Theorem: With notation and definitions above and $0 < \gamma < 1$, the optimal value function $V^*(x)$ satisfies

$$V^*(x) = \sup_{a \in \mathcal{A}} \left[r(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,a,y) V^*(y) \right] \tag{18}$$

Define the Bellman optimality operator

$$(T^*V)(x) = \sup_{a \in \mathcal{A}} \left[r(x,a) + \gamma \sum_{y \in \mathcal{X}} \mathcal{P}(x,a,y) V(y) \right] \tag{19}$$

Then T^* is a *(nonlinear) contraction operator and* V^* *is the unique solution to the fixed point equation*

$$T^*V = V \tag{20}$$

Bellman Operators and Equations for Action-Value Functions

Define $T^\pi : \mathbb{R}^{\mathcal{X}\times\mathcal{A}} \to \mathbb{R}^{\mathcal{X}\times\mathcal{A}}$ and $T^* : \mathbb{R}^{\mathcal{X}\times\mathcal{A}} \to \mathbb{R}^{\mathcal{X}\times\mathcal{A}}$ as follows

$$(T^\pi Q)(x,a) = r(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathcal{P}(x,a,y) Q(y,\pi(y)) \tag{21}$$

$$(T^* Q)(x,a) = r(x,a) + \gamma \sum_{y\in\mathcal{X}} \mathcal{P}(x,a,y) \sup_{a'\in\mathcal{A}} Q(y,a') \tag{22}$$

T^π is an affine linear operator and T^* is a nonlinear operator. They are contractions under the ℓ_∞ norm. Moreover Q^π and Q^* are unique solutions respectively to the fixed point equations

$$T^\pi Q = Q \tag{23}$$

$$T^* Q = Q \tag{24}$$

Greedy Policy

- Suppose we know V^*. Then one easy way to find the optimal control policy is to be greedy in a one-step search using V^*:

$$\pi^*(x) = \arg\max_a \left[r(x,a) + \gamma \sum P(x,a,y) V^*(y) \right] \tag{25}$$

- Suppose we know Q^*. Then we can use the zero-step greedy solution to find the optimal policy:

$$\pi^*(x) = \max_a Q^*(x,a) \tag{26}$$

- To implement the above approach, we need to know the model for the MDP.

Value Iteration Method

The contractive property of the various Bellman operators leads naturally to some simple iterative schemes.

- Value iteration: Starting with an initial value function V_0, calculate the sequence:

$$V_{k+1} = T^* V_k \tag{27}$$

  V_k converges to V^* at a geometric rate.

- Action-Value function iteration: Stating with an initial function Q_0, calculate:

$$Q_{k+1} = T^* Q_k \tag{28}$$

  Q_k converges to Q^* at a geometric rate.

- Theorem (Singh and Yee, 1994): Let π be the greedy optimal policy with respect to Q. Then

$$V^\pi \geq V^* - \frac{2}{1-\gamma} \|Q - Q^*\|_\infty \tag{29}$$

Policy Evaluation Step

We want to find the value function V^π for a given policy π. The idea is to use the fixed point property to calculate a sequence of approximations as follows:

$$\forall x : V^\pi_{k+1}(x) = \Big[r(x, \pi(x)) + \gamma \sum P(x, \pi(x), y) V^\pi_k(y) \Big] \tag{30}$$

Policy Iteration Method

- Start with an initial stationary policy π_0.
- Policy evaluation step: For $k > 0$, first estimate the value function corresponding to π_k, i. e., solution V^{π_k} to

$$T^{\pi_k} V^{\pi_k} = V^{\pi_k} \tag{31}$$

- Policy update step: Define the greedy policy with respect to V^{π_k}:

$$\pi_{k+1}(x) = \arg\max_{a \in \mathcal{A}} \left[r(x,a) + \gamma \sum_{y \in \mathcal{X}} P(x,a,y) V^{\pi_k}(y) \right] \tag{32}$$

Thus, we get a sequence of policy, its value function, next policy, its value function, and so on . . . But this can be very slow as each policy evaluation step and policy update step can be computationally demanding.

Policy Improvement Theorem

Theorem: If π and π' are two deterministic stationary policies such that

$$\forall x : Q^{\pi}(x, \pi'(x)) \geq Q^{\pi}(x, \pi(x)) = V^{\pi}(x) \implies V^{\pi'} \geq V^{\pi} \tag{33}$$

In other words, under the above condition, π' improves π.

Proof of this theorem is quite straightforward.

For stochastic policies, the following formula should be used:

$$Q^{\pi}(x, \pi'(x)) = \mathbb{E}_a \left[Q^{\pi}(x, a) \right], a \sim \pi' \tag{34}$$

Policy Improvement Approach

Let us use the result to develop an approach to policy improvement. One can try to find the new policy π' using the following optimization problem:

$$\forall x : \pi'(x) = \arg\max_a \left[r(x,a) + \gamma \sum P(x,a,y) V^{\pi}(y) \right] \quad (35)$$

$$= \arg\max_a Q^{\pi}(x,a) \quad (36)$$

If a model for the MDP is known, we can use the first equation. *Even if the model is not known, we can use the second equation as long as Q is known.*

Value Iteration Approach

In this approach, the policy evaluation step is not carried out and only value function is iterated upon as follows:

$$V^{k+1}(x) = \max_{a} \left[r(x,a) + \gamma \sum P(x,a,y) V^{k}(y) \right] \tag{37}$$

One can view this as applying the Bellman Optimality operator to update the value function and iterate on it.

What to do if the Model is not Available?

Monte-Carlo Methods

Monte-Carlo Methods

The basic idea is to estimate the value function or action-value function from trajectories.

We begin with the value function for a policy:

$$V^{\pi}(x) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^{k} r_{k+1} | \pi, x(0) = x\right] \tag{38}$$

Thus, we could try and estimate V^{π} online from the sampled trajectories of the MDP without needing the model.

Sampling Setup

- Fix the control policy π and the time horizon of length T
- A set of initial states x_0 and generate a trajectory τ
- Observe rewards sequence $r_1, r_2, \ldots r_T$
- Value estimate computed at the end of the horizon

First Visit Monte Carlo Algorithm

- Generate a large number of trajectories τ_i according to the previous setup
- For each trajectory τ_i, calculate $R_i(x)$ as the discounted reward for the sequence that comes after x is observed for the first time. Thus, for each x, there is a set of $R_i(x)$ when x occurs in a trajectory τ_i.
- For each $x \in \mathcal{X}$, calculate the average $R(x)$ of all $R_i(x)$ over those i where x is encountered in trajectory τ_i.
- $$V^{\pi}(x) \sim R(x) \tag{39}$$

First Visit Monte Carlo Algorithm for Q Functions

- Generate a large number of trajectories τ_i according to the previous setup
- For each trajectory τ_i, calculate $R_i(x, a)$ as the discounted reward for the sequence that comes after (x, a) is observed for the first time. Thus, for each (x, a), there is a set of $R_i(x, a)$ when (x, a) occurs in a trajectory τ_i.
- For each $x \in \mathcal{X}$, calculate the average $R(x, a)$ of all $R_i(x)$ over those i where x is encountered in trajectory τ_i.
- $$Q^{\pi}(x, a) \sim R(x, a) \tag{40}$$

Limitations

- Limited range of (x, a) unless each such pair is used at $k = 0$
- Need for large number of simulations
- Limited exploration

What if we use stochastic policies?

Exploration using Stochastic Policies

Idea: use a stochastic policy to visit all (x, a) combinations and thereby explore the decision space

- Choose π such that

$$\forall x \in \mathcal{X}, a \in \mathcal{A}; p_\pi(a|x) > 0$$

- Example: ϵ-greedy policy: π be the greedy policy and modify it as follows

$$\pi(x) = \begin{cases} \pi(x) \text{ with probability } 1 - \epsilon \\ \mathcal{U}(\mathcal{A}) \text{ with probability } \epsilon \end{cases} \tag{41}$$

On-Policy Monte Carlo Policy Improvement

The idea is to combine the Monte-Carlo method for policy evaluation and greedy optimization for policy improvement. More specifically:

- Use a stochastic policy (e.g., ϵ-greedy stochastic policy) to estimate Q^π
- Update the control policy by maximizing $Q^\pi(x, a)$ with respect to a.
- Repeat.

Off-Policy Monte Carlo Policy Improvement

The idea is to decouple the control policy from learning of Q function. More specifically:

- Use a policy π' for most of the actions. Goal is to learn Q^π while using π'.
- Choose the exploration policy to satisfy:

$$p_\pi(x, a) > 0 \implies p_{\pi'}(x, a) > 0. \tag{42}$$

- Use Monte-Carlo method to estimate $Q^\pi(x, a)$.

Temporal Difference Learning

Temporal Difference Learning

- In Monte-Carlo methods, we *wait till the end* of the run to update Q, V functions.
- Also, there is an assumption that system can be "reset" and "rerun" from various initial states. What happens if this cannot be done as in online control?
- Can we learn during the runs as well without having to wait?
- Temporal difference (TD) learning is a key idea in RL to do just this.
- TD learning blends the bootstrapping concept and Monte-Carlo.
- TD updates a guess towards a guess.
- TD has low variance and some bias as compared with MC which has high variance but is unbiased.

TD(0) Algorithm

Consider a Markov Reward Process over a finite state-space. Let (X_t, R_{t+1}) be the sequence of states and rewards associated with this MRP. Let V_t be a vector of the same length as the number of states and let us use $V_t(x)$ to denote the estimate of the value function at time t for the state x. Now use the following equation to update $V_t(x)$:

$$\delta_{t+1} = R_{t+1} + \gamma V_t(X_{t+1}) - V_t(X_t) \tag{43}$$

$$V_{t+1}(x) = V_t(x) + \alpha_t \delta_{t+1} \mathbb{I}_{X_t = x} \tag{44}$$

- $R_{t+1} + \gamma V_t(X_{t+1})$ is called *TD target.*
- Note that δ_{t+1} is connected to the Bellman operator and is called *TD error.*
- The value function is updated at each time instant for only the previous value of the state.
- Assuming $\alpha_{t+1} \leq 1$, the update moves the value function towards the target, i.e., the predicted value.

Stochastic Approximation and Robbins-Monro Condition

There are classical and current research efforts to use stochastic approximation theory to study convergence of such update procedures. The Robbins-Monro condition on α_t is often invoked to study such convergence properties.

$$\textit{RM Condition} \quad : \quad \sum_{t=0}^{\infty} \alpha_t = \infty, \quad \sum_{t=0}^{\infty} \alpha_t^2 < \infty. \tag{45}$$

$$\textit{Example} \quad : \quad \alpha_t = \frac{c}{t^{\eta}}, \quad 1/2 < \eta \leq 1 \tag{46}$$

Under such conditions, it has been shown (assuming X_T is a stationary, ergodic Markov chain) that V_t converges almost surely to the unique solution V to the Bellman equation for evaluating the value function for the MRP.

TD(N) Algorithms

- In TD(0) algorithm, we only took one time step to calculate $\delta(t)$.
- We could consider longer horizons, e. g. N time steps.
- Thus, consider

$$\begin{aligned} \delta_{t+1}^{N} &= R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{N-1} R_{t+N} + \gamma^{N} V_t(X_{t+N}) - V_t(X_t) \quad (47) \\ V_{t+1}(x) &= V_t(x) + \alpha_t \delta_{t+1}^{N} \mathbb{I}_{X_t = x} \quad (48) \end{aligned}$$

- $G_t^N := R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{N-1} R_{t+N} + \gamma^{N} V_t(X_{t+N})$ is called N-step return.
- There are results on convergence of $TD(N)$ algorithms.

TD(λ) Algorithm: Forward View

- The idea behind $TD(\lambda)$ algorithms is to take a weighted average of all N-step backups
- Define

$$G_t^{(\lambda)} := (1-\lambda)\sum_{N+1}^{\infty} \lambda^{N-1} G_t^N$$

- Forward view of TD(λ) update:

$$V_{t+1}(x) = V_t(x) + \alpha \left[{G_t}^{\lambda} - V_t(x) \right]$$

- This is good for theoretical understanding but for implementation we need to look backawards.

TD(λ) Backward View: Eligibility Traces

- While the forward view provides a theoretical frame for TD(λ) algorithm, for implementation we need to learn based on what has been observed till the current time.
- Backward view leads to a "credit assignment" issue: how to assign rewards to past actions?
- There are two distinct heuristics: frequency heuristic and recent heuristic.
- Frequency heuristic: assign credit to most frequent states.
- Recency heuristic: assign credit to more recent states.
- *Eligibility traces* concept combines these two heuristics.

True Online TD(λ), van Seijen and Sutton, ICML, 2014

TD(λ) Algorithm

- Consider a Markov Reward Process $(\mathcal{X}, \mathcal{P}_0)$ where the state-space $\mathcal{X}$ is finite of cardinality n. Recall that the value function is given by

$$V(x) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^t r_{t+1}, x(0) = x\right]$$

- We can think of V as an n-dimensional real vector. Now consider approximations of V of the functional form

$$\hat{V} = (\Phi w) \tag{49}$$

- Here Φ is an $n \times k$ matrix and w is a k-dimensional column vector, and Φw is an n-dimensional column vector. Denote by $\phi(i)$ a k-dimensional column vector corresponding to the i-th row of Φ.
- Now suppose (x_t, r_{t+1}) is a trajectory of the MRP. Suppose that the parameter vector w is set to be w_t at time t.
- The TD(λ) method will evolve the vector w_t as a function of time t.

Eligibility Vectors and TD(λ) Update Algorithm

- Define the temporal difference sequence by

$$d_t = r(x_t) + \gamma \hat{V}(x_{t+1}) - \hat{V}(x_t) \tag{50}$$

- Define the sequence of eligibility vectors by

$$z_t = \sum_{k=0}^{t} (\gamma\lambda)^{t-k} \phi(x_k) \tag{51}$$

- Then the TD(λ) update equation is given by

$$w_{t+1} = w_t + \alpha_t d_t z_t \tag{52}$$

- Note that eligibility vectors follow a simple update rule that combines recency and frqeuncy:

$$z_{t+1} = \gamma\lambda z_t + \phi(x_{t+1}) \tag{53}$$

Convergence of TD(λ)

- Suppose the Markov chain has a unique stationary distribution p such that $p(x) > 0$ for all x. Then we can define a norm on $\mathbb{R}^n$ by

$$\|v\|_p = \sqrt{\sum_{i \in \mathcal{X}} p(i) v(i)^2} \tag{54}$$

- Define a projection operator Π as

$$\Pi x = \arg\min_{u = \Phi r} \|x - u\|_p \tag{55}$$

- Finally define an operator T^λ by

$$(T^\lambda V)(x) = (1-\lambda) \sum_{m=0}^{\infty} \lambda^m \mathbb{E}\left[\sum_{t=0}^{m} \gamma^t r(x_t) + \gamma^{m+1} V(x_{m+1}) | x(0) = x \right] \tag{56}$$

Convergence of TD(λ)

Theorem (Tsitsiklis and Van Roy) Suppose that the matrix Φ is full column rank. Suppose the step sizes satisfy the RM condition. Then for all $\lambda \in [0, 1]$, the TD(λ) algorithm converges with probability 1. Suppose $w*$ denotes the final value of w_t. Then $w*$ satisfies:

$$\Pi T^{\lambda}(\Phi w^*) = \Phi w^* \tag{57}$$

Q Learning

Q-Learning

- A key observation: Optimal action values can be written as expectations unlike optimal state values.
- Let us start with finite state-space MDP $(\mathcal{X}, \mathcal{A}, \mathcal{P}_0)$ and a discount factor γ.
- Let $Q_t(x, a)$ denote an estimate at time t of $Q^*(x, a)$ for each state-action pair (x, a).
- Now suppose at time t we observe $(X_t, A_t, R_{t+1}, Y_{t+1})$. Here $A_t \in \mathcal{A}$ and $(Y_{t+1}, R_{t+1}) \sim \mathcal{P}_0(.|X_t, A_t)$.
- Update the estimate using

$$\delta_{t+1}(Q_t) = R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q_t(Y_{t+1}, a') - Q_t(X_t, A_t), \tag{58}$$

$$Q_{t+1}(x, a) = Q_t(x, a) + \alpha_t \delta_{t+1}(Q_t) \mathbb{I}_{(x=X_t, a=A_t)} \tag{59}$$

- Q-learning is related to TD-learning due to the use of the TD error $\delta_t(Q)$

Intuitive Reasoning regarding Q-Learning

- If a pair (x, a) is visited infinitely often, then it is plausible to argue that the $\mathbb{E}\left[\delta_{t+1}(Q)|X_t = a, A_t = a\right] = 0$.
- Using the definition of the Bellman optimality operator, one can see that

$$\mathbb{E}\left[\delta_{t+1}(Q)|X_t = a, A_t = a\right] = T^*Q(x, a) - Q(x, a), x \in \mathcal{X}, a \in \mathcal{A} \tag{60}$$

- So if every state-action pair is visited infinitely often, then the observations above indicate

$$T^*Q \sim Q \tag{61}$$

- Lots of papers on convergence of Q-learning algorithm based around these arguments.

Policy During the Learning Phase

- Once we are close enough to the optimal Q, we can use a greedy policy to select the action.
- What exploration policy should be used during the Q-Learning phase?
- One choice is ϵ-greedy policy.
- Another idea is to use Boltzmann scheme for exploration. For this define a probability distribution on the actions for a given $Q_t(x, a)$ by

$$p(a) = \frac{e^{\beta Q_t(a)}}{\sum_{a' \in \mathcal{A}} e^{\beta Q_t(a')}} \tag{62}$$

- β controls the greediness of the action selection
- A key difference between ϵ-greedy algorithm and the Boltzmann exploration is that the latter takes into account values of the possible actions unlike in the ϵ-greedy case.

Q-Learning with Function Approximation

- For large state and action spaces (e. g., continuous state and action spaces,) the approach of updating tables of $Q(x, a)$ values runs into computational difficulties.
- A popular alternative is to use parametrized functions to represent $Q(x, a)$ and update parameters as learning proceeds.
- Consider a representation of the form

$$Q(x, a) = Q_\theta(x, a), \theta \in \mathbb{R}^d \tag{63}$$

- A natural extension of the Q-learning algorithm then becomes

$$\theta_{t+1} = \theta_t + \alpha_t \delta_{t+1}(Q_{\theta_t}) \nabla_\theta Q_{\theta_t}(X_t, A_t) \tag{64}$$

- A simple special case if Q_θ is a linear function of θ in which case the formula above becomes simpler. In this case, there is a convergence result in Melo et al (2008). But not much is known otherwise about convergence properties of this algorithm.

DL Meets RL: Deep Reinforcement Learning

What is Deep RL?

DRL - Deep Neural Networks as Function Approximators in conjunction with Reinforcement Learning Algorithms.

Context for DRL

- Approximate Dynamic Programming based methods are not necessarily contractive when function approximations are used, for example, Gordon (1995).
- Aside: k-nearest neighbor approximation is non-expansive, Gordon (1995).
- Online approximate algorithms like Approximate Q-learning, Online Approximate SARSA (State-Action-Reward-State-Action) have been shown to diverge even for simple approximations and simple MDPs Sutton et al. (2000).
- Note that Deep Q-Networks (DQN) claim to overcome this issue by using *experience replay and fixed targets*. There is no rigorous proof for convergence, to the best of our knowledge, but they seem to work well.

Deep Q-Networks

- Online Q-function Iteration (gradient based instead of fitted Q-iteration)
- Deep Neural Networks to approximate Q factor/function.

Key Reference: V. Mnih, et al., "Human-level control through deep reinforcement learning," Nature, 2015.

Notation

- $L(.)$: Loss function for the learner
- D: pool of stored samples from learner's experience
- $\mathbb{E}_{\{x,a,r,x'\}\sim U(D)}[.]$: denotes expectation over samples of {state, action, reward, state}($\{x, u, r, x'\}$) drawn from data, D.
- θ: parameters of Deep Neural Network that approximates the Q factor
- Neural Network: Deep Convolutional Neural Network (CNN)

DQN Loss Function

- Loss function for learning:

$$L(\theta) = \underbrace{\mathbb{E}_{\{x,a,r,x'\}\sim U(D)}}_{\textit{experience replay}} \left[\left(\underbrace{r + \gamma \max_{u'} Q(x', u'; \theta^-)}_{\textit{fixed target}} - Q(x, u; \theta) \right) \right]^2$$

- Loss function is essentially the squared Bellman error (recall Bellman's Optimality Condition).

DQN Ideas: Experience Replay and Fixed Target

- Loss function for learning:

$$L(\theta) = \underbrace{\mathbb{E}_{\{x,a,r,x'\}\sim U(D)}}_{\textit{experience replay}} \left[\left(\underbrace{r + \gamma \max_{u'} Q(x', u'; \theta^-)}_{\textit{fixed target}} - Q(x, u; \theta) \right) \right]^2$$

- Experience replay: samples of experience are stored in D and replayed.
- Fixed target: the target value does not use the current iterate of the NN parameter θ. Rather it usesθ^- which is updated at certain intervals. Aimed at overcoming instability problems due to function approximations.

Benefits of Replay and Fixed Target

- Each sample is used many times through replay. This improves data efficiency.
- Not learning from sequences avoids correlations and improves learning by reducing variances of the updates. Correlations are broken through experience replay.
- Current samples depend on the current control policy decisions that in turn depend on the current parameters which can bias the learning. Experience replay avoids this.
- Fixed taret improves stability of learning (stability from the point of view convergence. Not the control stability.)

Approximate Q-Learning Algorithm

- Online version of fitted Q-iteration
- Loss function,

$$L(\theta) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{u'} Q(x', u'; \theta)}_{target} - Q(x, a; \theta)\right)\right]^2$$

- Stochastic Gradient Descent (update by gradient at a sample, holding the target fixed),

$$\theta_{k+1} = \theta_k + \alpha_k \left(r + \gamma \max_{u'} Q(x', u'; \theta) - Q(x, u; \theta)\right) \nabla_\theta Q(x, u; \theta)$$

- Note that exploration is critical. Exploration similar to Q-learning is used.

Iteration in DQN

- Loss function,

$$L(\theta) = \mathbb{E}_{\{x,a,r,x'\}\sim U(D)} \left[\left(r + \gamma \max_{u'} Q(x', u'; \theta^-) - Q(x, a; \theta) \right) \right]^2$$

- Stochastic Gradient Descent (update by gradient at a sample, target is anyways fixed),

$$\theta_{k+1} = \theta_k + \alpha_k \left(r + \gamma \max_{u'} Q(x', u'; \theta^-) - Q(x, a; \theta) \right) \nabla_\theta Q(x, a; \theta)$$

- Note that exploration is critical. Exploration similar to Q-learning is used.

Putting it Together: DQN Algorithm

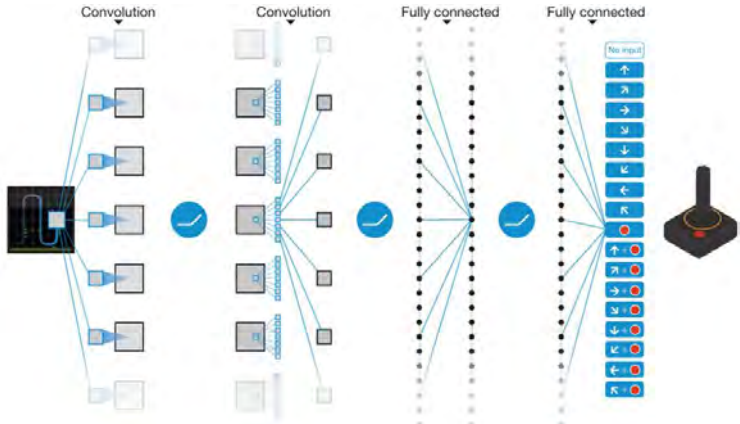
- Choose action a with an exploration policy
- Add sample $\{x, a, r, x'\}$ to D
- Sample mini-batch from D
- Perform the gradient descent step on the mini-batch loss
- Reset the target network θ^- after every n steps. Loop until convergence.

DQN Achievements

DQN Case Studies

- Testing on Atari 2600 platform. There are 49 games.
- High-dimensional data: 210×160 colour video at $60Hz$
- Human level performance in Atari games.

Convolutional Neural Network for DQN in Atari Games



Agent's Performance

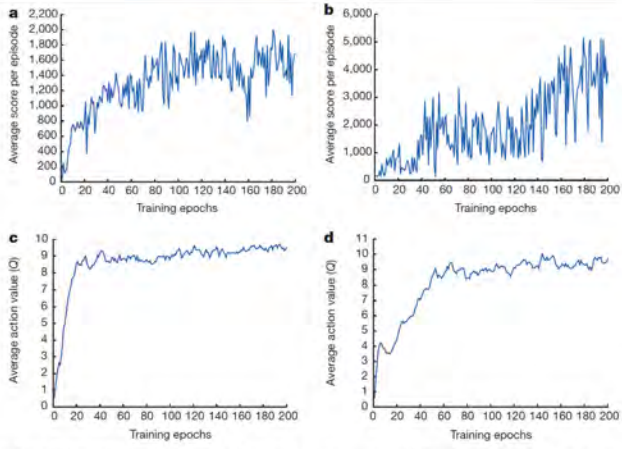


Figure 2 | Training curves tracking the agent's average score and average predicted action-value. **a**, Each point is the average score achieved per episode after the agent is run with ε-greedy policy ($\varepsilon = 0.05$) for 520 k frames on Space Invaders. **b**, Average score achieved per episode for Seaquest. **c**, Average predicted action-value on a held-out set of states on Space Invaders. Each point on the curve is the average of the action-value Q computed over the held-out set of states. Note that Q-values are scaled due to clipping of rewards (see Methods). **d**, Average predicted action-value on Seaquest. See Supplementary Discussion for details.

Performance

- Beats the best performing algorithms in 43 games
- Learns all by itself without any pretraining.
- Achieves more than 75% human score on more than half the games.

Performance Chart

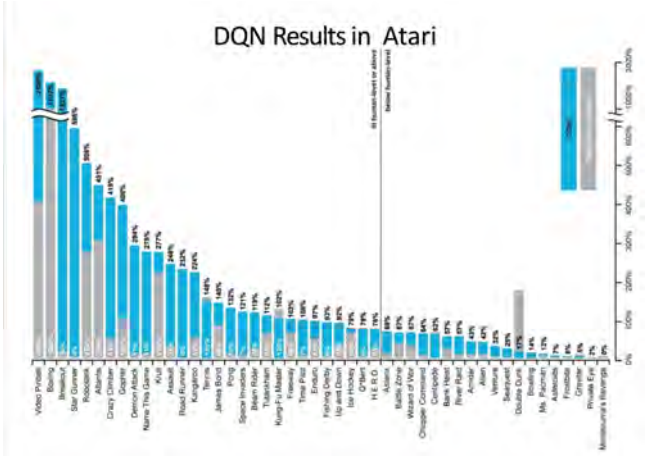


Figure: Comparison with best performing algorithm

DQN Vs. Best Linear Approximator

Game	DQN	Linear
Breakout	316.8	3.00
Enduro	1006.3	62.0
River Raid	7446.6	2346.9
Seaquest	2894.4	656.9
Space Invaders	1088.9	301.3

Effect of Replay and Fixed Target

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

Double DQN Idea

- DQN Loss function:

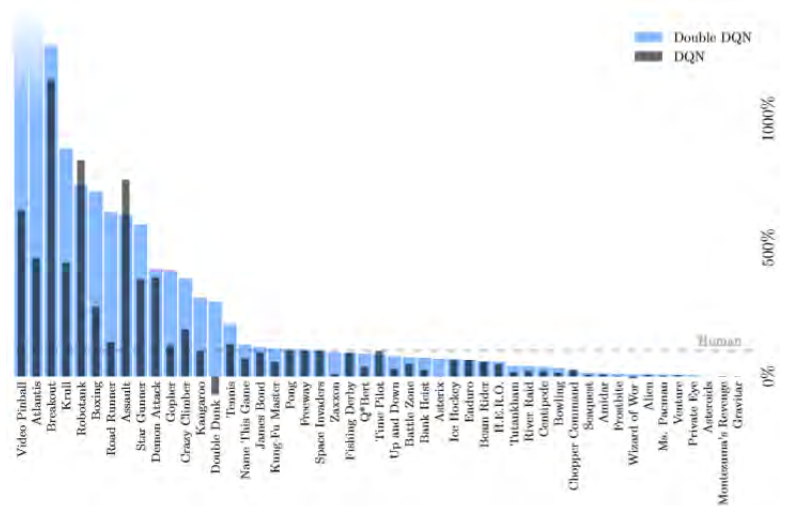
$$L(\theta) = \mathbb{E}_{\{xau,r,x'\}\sim U(D)}\left[\left(r + \gamma \max_{u'} Q(x', a'; \theta^-) - Q(x', a'; \theta)\right)\right]^2$$

- Double DQN,

$$L(w) = \mathbb{E}_{x,a,r,x'\sim D}\left[\left(r + \gamma Q(x', \text{argmax}_{u'} Q(x', u'; \theta); \theta^-) - \mathcal{Q}(x', a'; \theta)\right)^2\right]$$

- Claim: Avoids over-estimation problem, because of max. over actions, in DQN
- DDQN does not always improve performance.

Double DQN


Double DQN
DQN
1000%
500%
0%
Human
Video Pinball
Atlantis
Breakout
Krull
Robotank
Boxing
Road Runner
Assault
Star Gunner
Demon Attack
Gopher
Crazy Climber
Kangaroo
Double Dunk
Tennis
Name This Game
James Bond
Kung Fu Master
Pong
Freeway
Space Invaders
Zaxxon
Fishing Derby
Q*Bert
Time Pilot
Up and Down
Battle Zone
Bank Heist
Asterix
Ice Hockey
Enduro
Beam Rider
H.E.R.O.
Tutankham
River Raid
Centipede
Bowling
Chopper Command
Bezerk
Amidar
Frostbite
Wizard of Wor
Alien
Ms. Pacman
Venture
Private Eye
Asteroids
Montezuma's Revenge
Gravitar

Prioritized Replay DQN

- DQN samples uniformly from the replay buffer.
- Ideally, we want to sample more frequently those transitions from which there is much to learn.
- Prioritized experience replay [Schaul et al. (2015)] samples transitions with probability p_t that is in proportion to the last encountered absolute Bellman error,

$$p_t \propto \left| r_t + \gamma \max_{u'} Q(x', u'; \theta^-) - Q(x', a'; \theta) \right|$$

Multistep DQN Idea

- Q-learning accumulates a single reward and then uses the greedy action at the next step to bootstrap
- Here we use multistep target,

$$\sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^{t+n} \max_{a'} Q(x', a'; \theta^-)$$

- Recall DQN target:

$$r_t + \gamma \max_{a'} Q(x', a'; \theta^-)$$

Asynchronous and Parallel RL

Alternative: Parallel Learning and Asynchronous Update

- Multiple agents learn in parallel each in a different copy of the environment. Ex: the Gorila — General Reinforcement Learning Architecture — framework Nair et al. (2015).
- Updates from agents are made asynchronously.
- Asynchronous updates decorrelate the sequence because the updates are not from the same learner and each learner's experience is different.

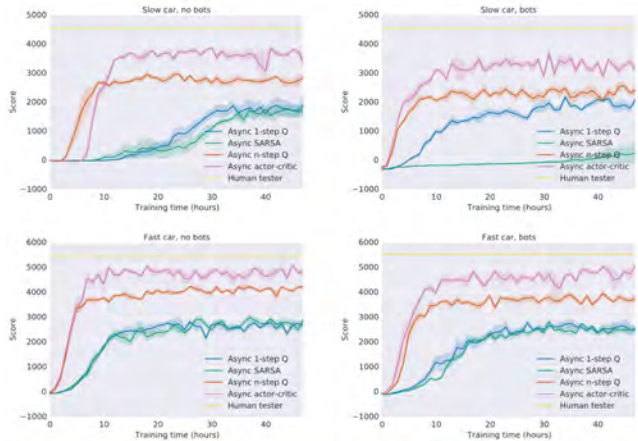
Gorila Framework

- In Gorila DQN (Nair et al. (2015)), each process contains an actor that acts in its own copy of the environment, a separate replay memory, and a learner that samples data from the replay memory and computes gradients of the DQN loss.
- By using 100 separate actor-learner processes and 30 parameter server instances, a total of 130 machines, Gorila was able to significantly outperform DQN over 49 Atari games. On many games Gorila reached the score achieved by DQN over 20 times faster than DQN.
- We introduce an *asynchronous Deep RL* method that exploits parallel learning and asynchronous update to break correlations in updates.

Different Asynchronous RL Methods

- Asynchronous 1-step Approximate Q-learning
- Asynchronous n-step Approximate Q-learning. N-step version of Q-learning reuses the per-step reward more effectively by using it in multiple iterations and improves convergence.
- Asynchronous Actor-Critic.

Numerical Performance of Various Asynchronous Methods



Observation: Asynchronous Actor-Critic $>$ Asynchronous N-step Q $>$ Asynchronous 1-step Q in terms of final score across many games.

Asynchronous Actor-Critic

- Based on the above observation, we will use asynchronous version of actor-critic to compare against DQN with experience replay and fixed target.
- First we introduce policy gradient based learning and then introduce actor-critic.
- We then introduce a classic policy gradient algorithm called REINFORCE and discuss its limitations
- And finally introduce actor-critic methods.

Policy Gradient based Learning

- These methods are not based on Dynamic Programming.
- The policy function is approximated by a parametrized representation. In some cases the output of the policy function is a probability distribution over the action space.
- The algorithm updates the parameters by a gradient descent update at every step, where the gradient is the derivative of value of the policy w.r.t the parameters.
- The gradients are not directly available. So the approach is to construct unbiased estimates of the policy gradient like the REINFORCE algorithm.

Advantages of Policy Gradient Methods Over Q-Learning

- Q-function based methods target policies are deterministic, whereas many problems have stochastic optimal policies.
- In Q-function based methods finding the greedy action with respect to the Q-function becomes problematic for larger action spaces.
- A small change in the Q-function can cause large changes in the policy, which creates difficulties for convergence proofs.

Reference: Degris et al. (2013).

The REINFORCE Algorithm

- One of the earliest policy gradient based method is called REINFORCE, Williams (1992).
- REINFORCE is slow to converge because of large variance of estimate of the gradient (it is unbiased), which requires a large time horizon of samples of experience, used at every step for update. We shall discuss REINFORCE later.

Actor-Critic based Policy Gradient

- There was renewed interest in policy gradient based methods in the late 90s because of convergence issues in value function based methods with function approximation as discussed earlier.
- Sutton et al. (2000) devised a policy gradient method using value function approximators (critic) and proved convergence to a locally optimal policy.
- Use of the critic function to approximate the gradient reduced the variance in the gradient estimate when compared to REINFORCE (Sutton et al. (2000)).
- If the critic function is not appropriately designed then it can bias the gradient estimate and affect convergence.
- This was the first paper, to the best of our knowledge, to prove convergence to a locally optimal policy for an approximate iterative algorithm for RL.

Natural Actor-Critic

- The actor-critic method by Sutton et al. (2000) requires knowledge of transition dynamics of the environment.
- Natural actor-critic is the first fully incremental (update by stochastic gradient descent) and provably locally convergent algorithm: Bhatnagar et al. (2009).
- It uses a natural gradient of the value of the policy w.r.t the parameters.
- Natural gradient reduces variance further in some cases.

Asynchronous Advantage Actor-Critic (A3C)

- A3C uses a deep neural network to approximate the policy (actor), $\mu(x; \theta_a)$, where θ_a are the neural network parameters. The output of this function is a *stochastic policy* i.e. a probability distribution over the action space.
- A3C uses a second deep neural network to approximate the policy's value function (critic of the policy), $V_\mu(x; \theta_v)$.
- Gradient descent update by an estimate of policy gradient. The critic network is used in the estimation the policy gradient. Hence this is an actor-critic approach.
- Asynchronous refers to many parallel learners who update the actor and critic networks asynchronously. The idea is that this parallelization speeds up learning.

Recap of Advantages over DQN

Recall DQN gradient descent,

$$\theta_{k+1} = \theta_k + \alpha_k \left(r + \gamma \max_{u'} Q(x', a'; \theta^-) - Q(x, a; \theta) \right) \nabla_\theta Q(x, u; \theta)$$

- DQN has limitations in large action spaces or continuous action spaces
- On the other hand, an actor network can be used to approximate a policy over continuous action spaces and stochastic policies.
- Experience replay, the key ingredient behind DQN's success, which helps in breaking correlation in updates, requires memory and computation per time step.
- Parallelism in A3C also helps in breaking the correlation in updates since the updates are made by parallel learners asynchronously. This does not require any external memory.

Notation

- π: stochastic policy i.e. a probability distribution over the action space.
- $Q_\pi(x, a)$: Q-function of policy π.
- $V_\pi(x)$: Value function of policy.
- $\mathbb{E}_{a\sim\pi}[.]$: expectation w.r.t samples of action drawn from the stochastic policy.
- θ_a: parameters of the function approximator of the stochastic policy π
- θ_v: parameters of the function approximator of the policy's value function V_π.
- $V_\pi(.; \theta_v)$: function approximation of value function of policy.
- $\pi(.; \theta_a)$: function approximation of the stochastic policy.

Policy Gradient

- Define

$$Q_\pi(x_t, a_t) = \mathbb{E}_{x_{t+1:\infty}, a_{t+1:\infty}} \left[\sum_{k=t}^{\infty} \gamma^{k-t} r_k \right], \; a_k \sim \pi \; \forall \; k \geq t+1$$

$$V_\pi(x_t) = \mathbb{E}_{a_t \sim \pi} \left[Q_\pi(x_t, a_t) \right]$$

- Policy gradient with respect to the cumulative discounted reward of a policy (Schulman et. al. 2015b),

$$\mathbb{E}_{a \sim \pi} \left[\sum_{t \geq 0} Q_\pi(x_t, a_t) \nabla_{\theta_a} \log \left(\pi(a_t | x_t; \theta_a) \right) \right]$$

Stochastic Policy Gradient

- The stochastic policy gradient is the single trajectory approximation of the policy gradient,

$$\sum_{t \geq 0} Q_\pi(x_t, a_t) \nabla_{\theta_a} \log \left(\pi(a_t | x_t; \theta_a) \right)$$

- The above expression is an unbiased estimate of policy gradient.
- The Q-functions are usually unknown. So an unbiased estimate of the gradient can be obtained by replacing Q_μ by rewards accumulated from the trajectory, $R_t = \sum_{t' \geq t} \gamma^{t'-t} r_{t'}$. An unbiased estimate of the gradient is then given by,

$$\sum_{t \geq 0} R_t \nabla_{\theta_a} \log \left(\pi(a_t | x_t; \theta_a) \right)$$

(REINFORCE)

Reducing Variance in REINFORCE

- The REINFORCE policy gradient has high variance
- It is possible to reduce variance by introducing a baseline $b(x_t)$ as below,

$$\sum_{t \geq 0} (R_t - b(x_t)) \nabla_{\theta_a} \log \left(\pi(a_t | x_t; \theta_a) \right)$$

- Note that in spite of the inclusion of a baseline the gradient is still an unbiased estimate.

Policy Gradient with Advantage Estimation

- A choice for baseline $b(x_t)$ can be $V_\pi(x_t)$ (defined earlier). With such a choice of baseline, $R_t - b(x_t) = R_t - V_\pi(x_t)$ becomes an unbiased estimate of the advantage function,
$$A_\pi(x, u) = Q_\pi(x, u) - V_\pi(x) \tag{65}$$
- Denote by $\hat{A}$ any unbiased estimate of the advantage function then the stochastic policy gradient can be written as,
$$\sum_{t \geq 0} \hat{A} \nabla_{\theta_a} \log\left(\pi(a_t | x_t; \theta_a)\right)$$
- Note that the value function is also an unknown in which case the baseline $b(x_t) = V_\pi$ is replaced by an approximation of the value function.

Policy Gradient in A3C

- A3C uses the following unbiased estimate of advantange function,

$$\hat{A}_{A3C} = \sum_{i \geq t}^{k-1} \gamma^{i-t} r_t + \gamma^{t+k} V_\pi(x_{t+k}) - V_\pi(x_t) \tag{66}$$

- In addition, it approximates the value function by a second network, the critic network, $V_\pi(x; \theta_v)$, as it is usually an unknown.
- Truncating the reward sum at k steps and replacing the rest of the sum by a value function, as in $\hat{A}_{A3C}$, reduces the variance in the estimate of $\hat{A}$. But the approximation of the value function induces bias. So bias and variance can be balanced by varying the depth of the sum i.e. k in the above expression.

Policy Gradient in A3C

- Stochastic Policy Gradient in A3C,

$$\sum_{t\geq 0}\left(\sum_{i\geq t}^{k-1}\gamma^{i-t}r_t+\gamma^{t+k}V_\pi(x_{t+k};\theta_v)-V_\pi(x_t;\theta_v)\right)\nabla_{\theta_a}\log\left(\pi(a_t|x_t;\theta_a)\right)$$

where $V_\pi(x_t;\theta_v)$ is the DNN approximation of the value function

A3C Algorithm (Single Learner) with Regular Gradient Descent

- Each thread (paraller learner) acquires θ_a and θ_v (global policy and value parameters)
- Till t reaches t_{max}: $u_t \sim \pi(a_t|s_t;\theta_a)$, receive reward r_t and state x_{t+1}
- Compute backwards, $R_t = r_t + \gamma R_{t+1}$ where $R_{t_{max}} = V_\pi(x_{t_{max}};\theta_v)$
- Policy gradient update,

$$\theta_a \leftarrow \theta_a + \sum_{t=0}^{t_{max}} \left(R_t - V_\pi(x_t;\theta_v)\right) \nabla_{\theta_a} \log\left(\pi(a_t|x_t;\theta_a)\right)$$

- Value function update,

$$\theta_v \leftarrow \theta_v + \sum_{t=0}^{t_{max}} \partial \left(R_t - V_\pi(x_t;\theta_v)\right)^2 / \partial\theta_v$$

- Asynchronous update of the global parameters. Loop

Implementation

- A single Deep Convolutional Neural Network to approximate both the policy and value function
- Two separate top layers for the policy and the value function
- A soft-max layer (recall the soft-max output from Lecture 6) to output the policy
- A linear output for the value function.
- All other parameters are shared.
- Soft-max layer outputs the probability of choosing an action. This is not possible with value function based methods, which is an issue that we highlighted before.
- A3C uses RMSprop gradient descent algorithm instead of the regular gradient descent.
- Hardware: multiple CPU threads as parallel learners instead of seperate machines (Gorilla Framework).

Accelerating Gradient Descent with Momentum

- The A3C algorithm discussed before uses a standard stochastic gradient descent update.
- There are other versions of A3C that are based on improved versions of gradient descent update.
- Stochastic Gradient Descent with Momentum

$$\theta \leftarrow \theta - \alpha m$$

where $m = \beta m + (1 - \beta)\Delta\theta$ and $\Delta\theta$ is the regular gradient update. The final update m is a moving average of the regular stochastic gradient updates. Moving averages better approximate the true gradient of the loss function leading to better convergence.

- Nesterov's accelerated gradient is an improved version of SGD with momentum.

Suggestion to read: Various Gradient Descent Algorithms

A3C Uses RMSprop

- RMSprop is an unpublished adaptive learning rate based gradient descent by Hinton.
- A very widely used gradient descent algorithm in Deep Learning.
- RMSprop update is given by,

$$g = \beta g + (1 - \beta)\Delta\theta^2$$

$$\theta \leftarrow \theta - \alpha \frac{\Delta\theta}{\sqrt{g + \epsilon}}$$

- The denominator $\sqrt{g + \epsilon}$ is the term that adaptively tunes the learning rate unlike standard SGD where the learning rates have to be tuned manually. ϵ is a predetermined constant. Adaptive learning rate improves convergence. A3C uses RMS prop.

Reading: Various Gradient Descent Algorithms

Performance of A3C on Atari Domain

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorila	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

Table 1. Mean and median human-normalized scores on 57 Atari games using the human starts evaluation metric. Supplementary Table SS3 shows the raw scores for all games.

Note: But A3C is not sample efficient. As we shall see later.

A3C Learning Curves

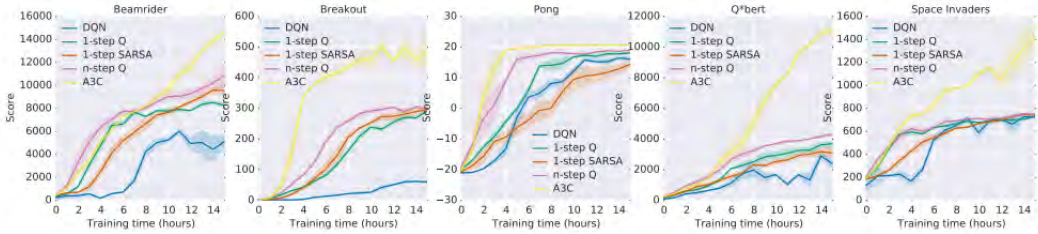
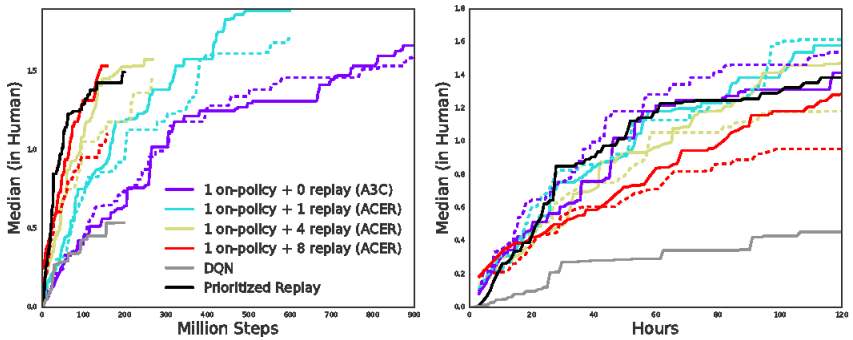


Figure 1. Learning speed comparison for DQN and the new asynchronous algorithms on five Atari 2600 games. DQN was trained on a single Nvidia K40 GPU while the asynchronous methods were trained using 16 CPU cores. The plots are averaged over 5 runs. In the case of DQN the runs were for different seeds with fixed hyperparameters. For asynchronous methods we average over the best 5 models from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

An Actor-Critic Method for Sample Efficiency

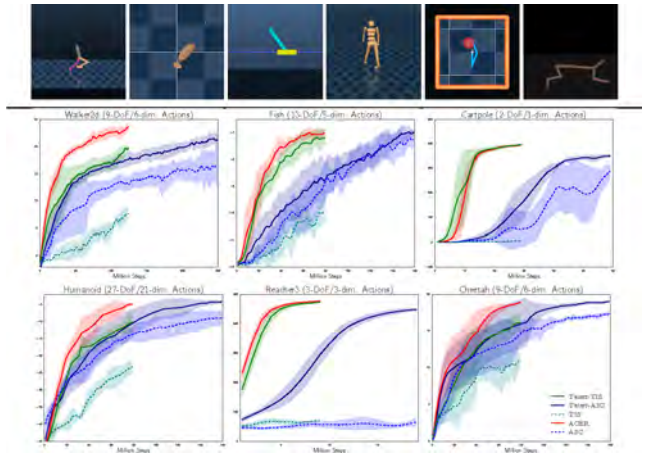
- Here we briefly discuss the results of a method that combines the best of both worlds we have discussed so far,
  - To be sample efficient as in DQN which uses experience replay
  - At the same time leverage all the advantages of A3C, which are massive parallelization, applicability to large and continuous action spaces and ability to output randomized policies.
- Using experience replay to compute the gradient induces bias and affects convergence.
- The method proposed in the paper (Wang et al. (2017) effectively combines actor-critic with experience replay and is called ACER.
- This is the state-of-the-art in actor-critic and is sample efficient. In our opinion this is a major step forward in actor-critic based methods.

ACER Matches DQN in Performance and Sample Efficiency



Left: number of steps in environment, Right: computation time.

ACER is Competitive in Continuous Action Spaces



Note: It is not feasible to implement DQN in continuous action spaces.

Rollout Based Planning for RL and Monte-Carlo Tree Search

Basic Idea behind Rollouts

- Rollouts were first introduced by Tesauro and Galperin (1996).
- Rollouts are essentially policy enhancers, Tesauro and Galperin (1996), Yan et al. (2005).
- Given a base policy by running repeated rollouts one can effectively improve the policy.
- The improvements can be significant even for few iterations as was demonstrated in Tesauro and Galperin '96.
- Can be an effective tool to improve performance of planning based on models to improve sample efficiency of model-free RL algorithms. We shall see an application of this later.

What is Rollout based Planning?

- Rollouts are performed from the current state for each possible action.
- The rollouts form a tree with the root node as the current state.
- The values of the rollouts are backed up.
- And the backed up values are used as a guide to improve the policy in the next step.
- In some cases, when a simulation environment is not available a model of the environment is learnt online and used for rollout based planning.

Monte-Carlo Tree Search (MCTS)

- Rollout based policy improvement could be computationally heavy with the number of time steps in each rollout and the number of initial branches.
- One approach to improve this is parallelization.
- Another approach is Monte Carlo Tree Search
- Monte Carlo Tree Search (MCTS) simplifes the search by selecting a certain rollout based on statistics stored from past rollouts. MCTS uses the statistic to prune away the most unlikely rollouts in the next iteration.
- This selective iteration can improve the speed of convergence and UCT (upper confidence bounds applied to trees) is one such method.

UCT Algorithm for MCTS

- UCT refers to Upper Confidence Bounds applied to Trees (UCT).
- UCT employs a selection mechanism based on a policy that balances exploration vs. exploitation.
- UCT is the Monte Carlo Tree Search version of UCB1 algorithm for bandit problems.
- Powerful search method for games. We shall explore the application of a variant of this method to the game GO. Initial success of this method in Game Go spurred interest in MCTS.

Phases in MCTS

- Tree search refers to search within a tree of nodes or states, as the case may be, visited prior to the current search. The nodes contain information pertaining to the statistics of previous visits to each node. For example: number of visits etc.
- Selection Phase: selecting a path from the search tree built so far.
- Expansion Phase: the search tree is not complete at any step. On reaching a node in the search tree which does not have any child node, child nodes are added to this node. This is tree expansion. The child nodes are chosen randomly.

Phases in MCTS Continued...

- Simulation Phase: the search is continued from this newly added node. The search from this new node is done by a base policy, like a policy that chooses the feasible branches uniformly randomly.
- Backup Phase: the outcome at the end of the simulation is used to update the statistics of all the nodes visited in this expanded tree and the search policy for the next iteration is also updated.
- The statistics of the nodes in the search tree determine the search policy.

MCTS - Selection

Starting at the root node, a child selection policy is recursively applied to descend through the tree until the most urgent expandable node is reached. A node is expandable if it represents a nonterminal state or has unvisited (i.e. unexpanded) children.

Example, UCT selection policy:

$$\frac{S_c}{n_c} + K\sqrt{\frac{2\ln n_p}{n_c}} \tag{67}$$

S_c: reward for child node, n_c: number of visits to the child node, n_p: number of visits to the parent node.

- If the child node was visited less i.e. n_c is small then its score increases. This ensures less frequented nodes are explored.
- If the child node's average reward is larger i.e. S_c/n_c is large then its score increases. This gives preference to high reward nodes from the past (exploitation).

MCTS - Expansion

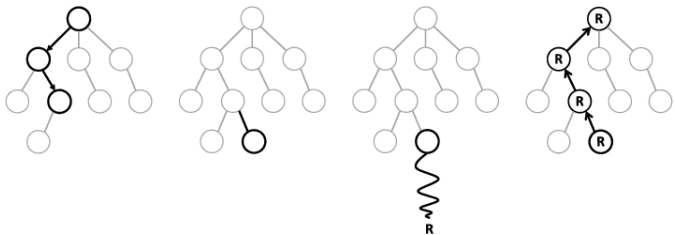
This is done at the urgent expandable node which are nodes that have children that are univisited. On descending, the MCTS continues the search through expansion of the tree by adding one or more child nodes to this expandable node.

MCTS - Simulation and Backup

A simulation is run from the newly added node(s) according to the default policy till termination. This default policy could be just a random selection of the branches or child nodes.

The simulation result is “backed-up” (i.e. backpropagated) through the selected nodes to update their statistic, for example, in games ratio of number of wins to number of times played.

Summary of Phases in Monte Carlo Tree Search (MCTS)



Reference: C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, ... and S. Colton, "A survey of monte carlo tree search methods," IEEE Transactions on Computational Intelligence and AI in games, 2012.

Pros and Cons of MCTS

- Pros: they have the potential to enhance a policy in real time using a simulation environment. In the absence of a simulation environment they can use a learnt model as the simulation environment.
- Cons:
  - A limiting factor is the computation involved in the tree search at every time step.
  - Storing the search tree requires memory.
  - The search tree is of infinite size in environments with continuous action spaces

MCTS for Continuous Action Spaces

- Kernel Regression - Upper Confidence Bound applied to Trees (UCT).
- Uses a kernel based regression to estimate the values of child nodes not visited before.
- Uses a kernel based regression to esimate the number of visits of child nodes not visited before
- These values are plugged in place of their equivalents in the regular UCT formulation for selecting the child node during the search.

Reference: T. Yee, V. Lisy, and M. H. Bowling, "Monte Carlo Tree Search in Continuous Action Spaces with Execution Uncertainty," In IJCAI, pp. 690-697, 2016.

Application of MCTS to Game Go: AlphaGo Zero

D. Silver, A. Huang, C. J. Maddison, A. Guez, L Sifre, G. Van Den Driessche, ... and S. Dieleman, "Mastering the game of Go with deep neural networks and tree search," nature, 2016.

Previous Versions of AlphaGo

- AlphaGo Fan:
  - Uses two networks, a policy network that outputs move probabilities and a value network that outputs the win probability from a particular state.
  - These networks were initialy trained by supervised learning and then fine tuned by policy gradient based reinforcement learning
  - Eventually MCTS was used to narrow down the search to moves with high probability to win.
- AlphaGo Lee:
  - A version similar to AlphaGo Fan. Defeated Lee Sidol.

Current Version: AlphaGo Zero

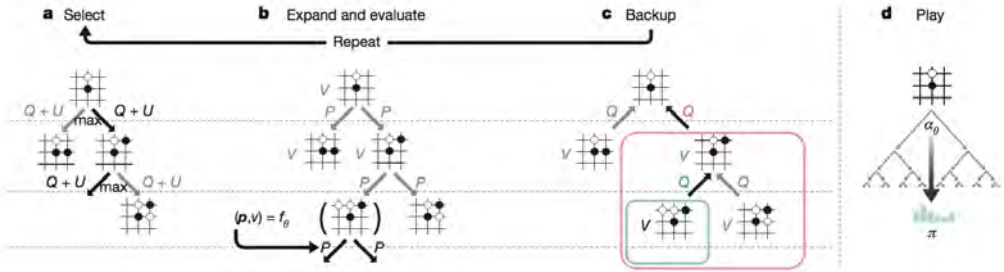
- Trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data.
- It uses only the black and white stones from the board as input features.
- It uses a single neural network, rather than separate policy and value networks.
- It uses a simpler tree search that relies upon this single neural network to evaluate positions and sample moves, without performing any Monte Carlo rollouts (i.e. any simulation step)
- Output:

$$\{P_\theta(x, .), V_\theta(x)\} = f_\theta(x) \tag{68}$$

where P_θ is the move probability function and V_θ is the value function, in this case the win probability and θ are the DNN parameters.

AlphaGo Zero

- Search by MCTS
- Policy improvement based on the updated statistics after the search



Reference: D. Silver, A. Huang, C. J. Maddison, A. Guez, L Sifre, G. Van Den Driessche, ... and S. Dieleman, "Mastering the game of Go with deep neural networks and tree search," nature, 2016.

MCTS in AlphaGo Zero - Selection

- At each iteration, an MCTS based search is carried out.
- Every node (a state x) and branch (an action a) is associated with the statistics $Q(x, a)$ (action value from last iteration), $N(x, a)$ (number of visits to (x, a)), $P_\theta(x, a)$ (move probability from last iteration).
- A branch (action a) at a node x is chosen by a criterion similar to MCTS, that which maximizes,

$$Q(x, a) + U_\theta(x, a) \ , \text{ where } U(x, a) \propto P_\theta(s, a)/(1 + N(x, a)) \tag{69}$$

- Note that the above search balances exploration vs. exploitation

MCTS in AlphaGo Zero - Expansion and Backup

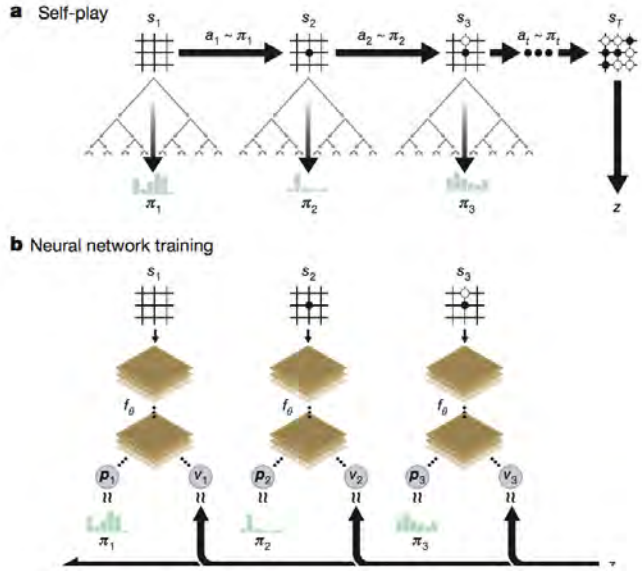
- On descending down the tree and reaching an exandable node, a single expansion is made and the search is stopped. Function V_θ is used to estimate the value of this last node. So difference is that there is no simulation here.
- Updates:
  - $N(x, a)$ is incremented by the number of visits to the pair (x, a) in this iteration
  - $Q(x, a) = 1/N(x, a) \sum_{s'|(x,a)\to s'} V_\theta(x')$ (Backup step)

Learning in AlphaGo Zero - RL by Self-Play

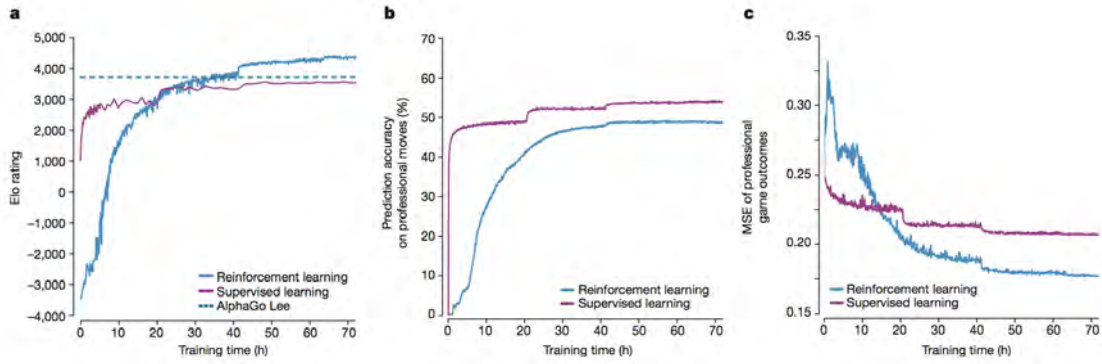
- Improved policy based on MCTS, $a \sim \pi_\theta(x, a) \propto 1/N(x, a)^{1/\tau}$
- Self-play: the policy plays itself.
- At each time step t, given the state x_t, an action a_t (branch) is chosen based on distribution π_θ
- At the end of game, say at the Tth time step, a reward r_T is generated based on the result of the game, where $r_T \in \{+1, -1\}$ and depends on which player won.
- The following tuples are recorded $\{x_t, a_t, z_t\}$ where $z_t = 1$ if the player who moved at state s_t wins and $z_t = 0$ otherwise.
- The following loss function, over the tuples $\{x_t, a_t, z_t\}$, is minimized to update the parameter θ,

$$L(\theta) = (z - V_\theta)^2 - \sum_a \pi(., a) \log(P_\theta(., a)) + c||\theta||^2 \tag{70}$$

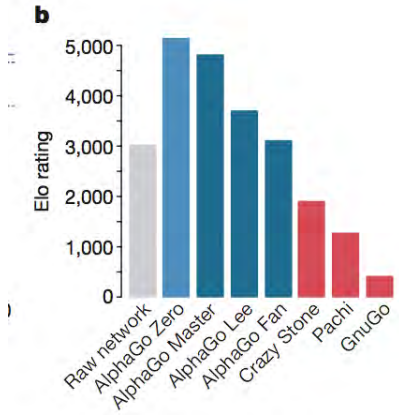
Graphical Illustration of RL by Self-Play



RL by Self-Play Outperforms Supervised Learning



AlphaGo Zero is the State-of-the-Art



Note: Inspite of learning by itself it is still the best.

Recap

- Reinforcement learning has been very successful in board games
- Some applications are emerging in engineered systems
- It is the most viable path for unsupervised learning
- There are important connections to stochastic control, dynamic programming, ...

Future Directions

- Control has historically relied on model-based approaches. RL has historical connections to stochastic control and optimization.
- RL offers an opportunity to open new frontiers where models are unavailable or are too difficult and expensive to build.
- Multi-agent, decentralized RL has great potential for Smart-X control applications. We are at very early stages in this area.
- RL is a powerful approach for autonomy.
- RL can also play a role in human-machine interactions or human augmentation.

Acknowledgement

Thanks to Dileep Kalathil, Texas A&M University, for his constructive feedback during the preparation of this presentation.

Contact Information for Further Information

- email: pramod.khargonekar@uci.edu; khargonekar@gmail.com
- Faculty website
- LinkedIn Page
- Google Scholar page