Module IV: A Simulation-based Deep Reinforcement Learning Framework

Zeyu Liu and Xueping Li

THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

Reinforcement Learning (RL)

- *"Reinforcement learning is learning what to do — how to map situations to actions — so as to maximize a numerical reward signal."*

  — Sutton and Barto, 2018

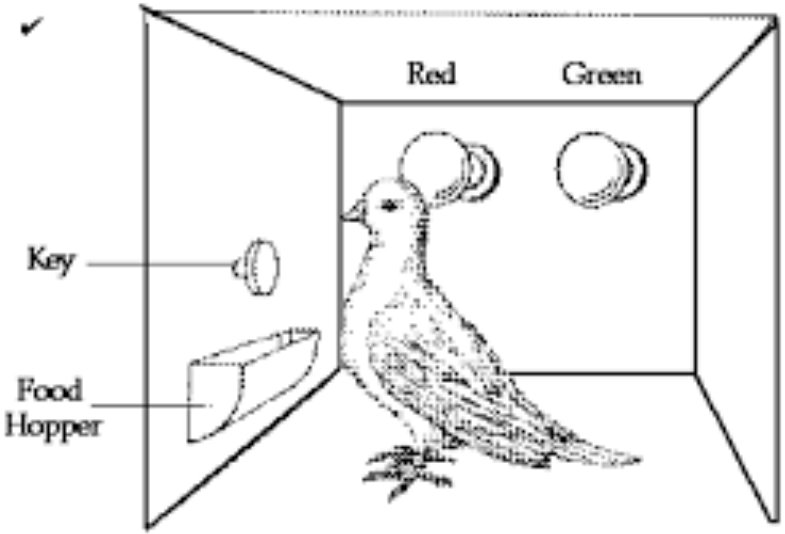
- Scenario 1 → Action 1;
- Scenario 2 → Action 2;
- Scenario 3 → Action 3;
- …

“Reinforcement”

- A term in psychology;
- Pigeon experiment (by B. F. Skinner);
- Red light:

  press key → no food
- Green light:

  press key → food
- Learn to press key on green light.



Pigeon Operant Chamber

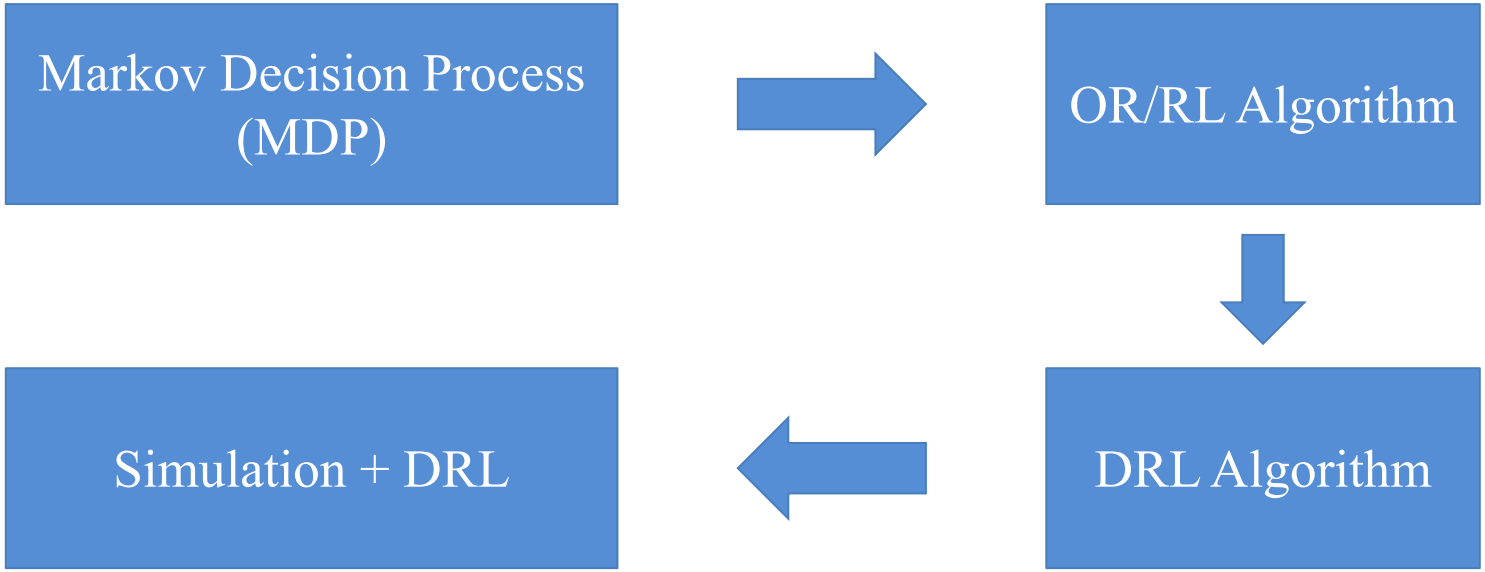
“Reinforcement” + Learning

- Pigeon (animal) → Agent (machine);
- Choice of action → Function;
- Food → Reward;
- Learn the maximized reward by optimizing the function.

“Deep” Reinforcement Learning (DRL)

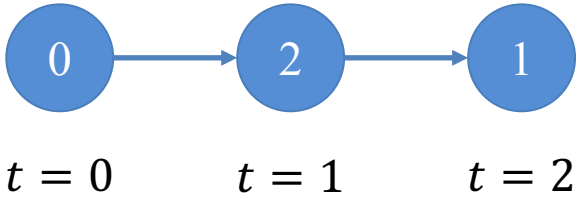
- Deep: deep neural networks;
- Use neural networks to approximate the function.

Roadmap


Markov Decision Process (MDP)
OR/RL Algorithm
DRL Algorithm
Simulation + DRL

Markov Chain

- Epoch $t = 0, 1, 2, \ \ldots, T$ (discrete);
- States $s \in S$;
- E.g.,
  - $t = 0, 1, 2$;
  - $s \in S = \{0, 1, 2\}$;
  - A possible chain:



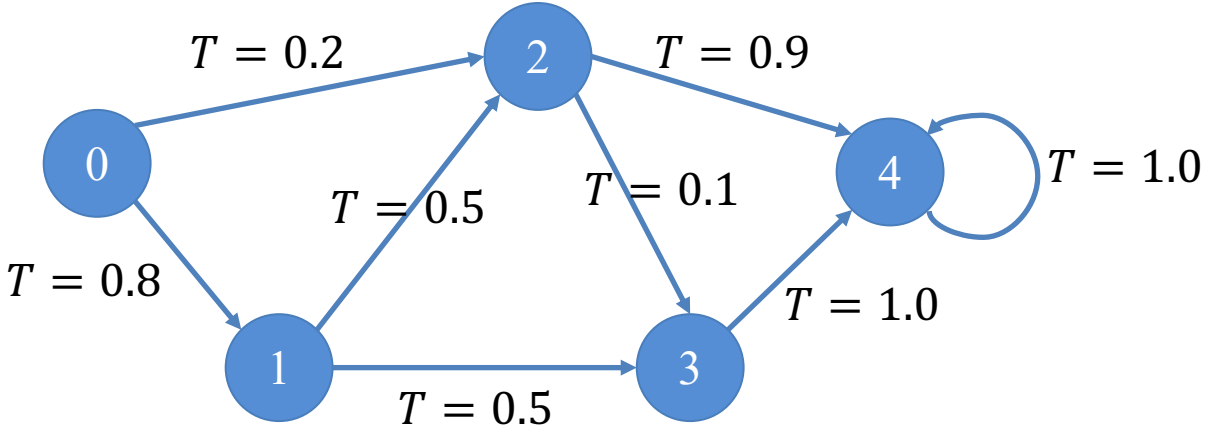
(Transition)

Markov Chain

- Transition probability: $T(s_t|s_{t-1}, s_{t-2}, \dots, s_0)$, $\forall\, s, s' \in S$;
  - $0 \leq T(s_t|s_{t-1}, s_{t-2}, \dots, s_0) \leq 1$;
  - $\sum_{s_t \in S} T(s_t|s_{t-1}, s_{t-2}, \dots, s_0) = 1$;
- Markov property: $T(s_t|s_{t-1}, s_{t-2}, \dots, s_0) = T(s_t|s_{t-1})$;
- Rewrite: $T(s'|s)$, $\forall\, s', s \in S$;
- Time-invariant;

Markov Chain

- Transition diagram:



Markov Chain

- Transition matrix:

$$\begin{array}{cc} & \text{to} \\ \text{from} & \begin{pmatrix} 0.0 & 0.8 & 0.2 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.5 & 0.5 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.1 & 0.9 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} \end{array}$$

Equilibrium of Markov Chain

- Let P be a transition matrix;
- If P satisfies some conditions (finite, irreducible, aperiodic), there exists a stable distribution $\boldsymbol{\psi}$, such that

$$\boldsymbol{\psi}^T \cdot P = \boldsymbol{\psi}^T$$

MDP

- Markov chain with decisions;
- Decision epochs $t = 0, 1, 2, \ \ldots, T$ (discrete);
- **Definition**: (S, A, T, R, γ);
- Action set A;
- Reward function $R: S \times A \rightarrow \mathbb{R}$;
- Discount factor $0 \leq \gamma \leq 1$;

States

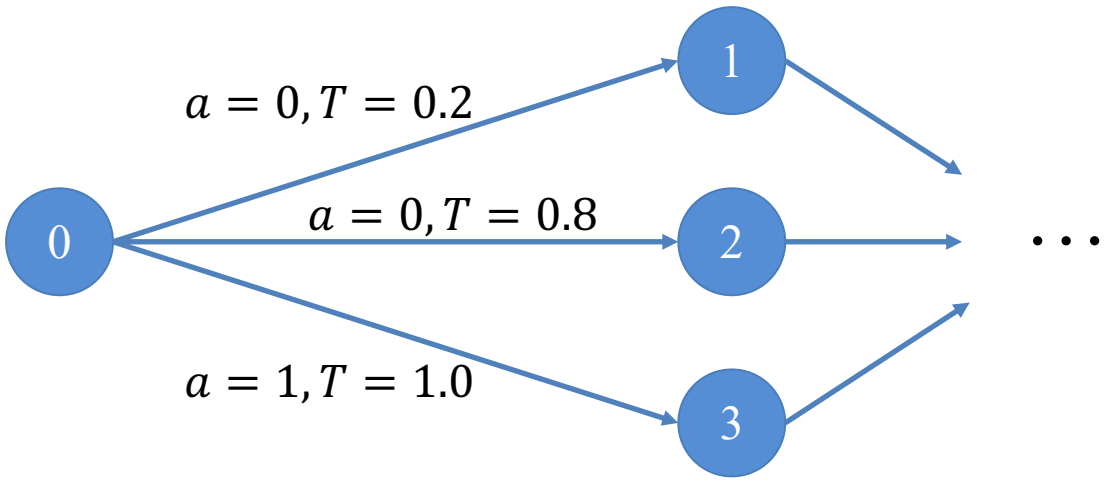
- States $s \in S$;
- Initial distribution: $\boldsymbol{\alpha} = (\dots, \alpha_s, \dots)^T$,
  - $0 \leq \alpha \leq 1$;
  - $\sum_{s \in S} \alpha_s = 1$.

Actions

- Decisions that are available to take at each epoch;
- E.g.,
  - $A = \{0, 1, 2, \cdots, N\}$;
  - $A = \{north, south, east\ west\}$;
  - …
- Can depend on the state: $a \in A_s \subseteq A$.

State Transition

- Actions affect the transition:



State Transition

- Transition probability: $T(s'|s,a), \forall\, s', s \in S, a \in A$;
- Transition matrices: $P_a, \forall\, a \in A$;
- Completeness:

$$\sum_{s' \in S} T(s'|s,a) = 1 \qquad \forall\, a \in A$$

Reward Function

- Maps a state-action tuple to a real number;
- $R_t(s, a) \in \mathbb{R}, t = 0, 1, \ldots, T - 1$;
- $R_T(s)$: terminal reward.

Objective

- Maximize the total reward accumulated over the horizon $t = 0, 1, \ldots, T$;
- Finite horizon: $T < +\infty$;
- Infinite horizon: $T = +\infty$.

Policy

- The solution of the MDP

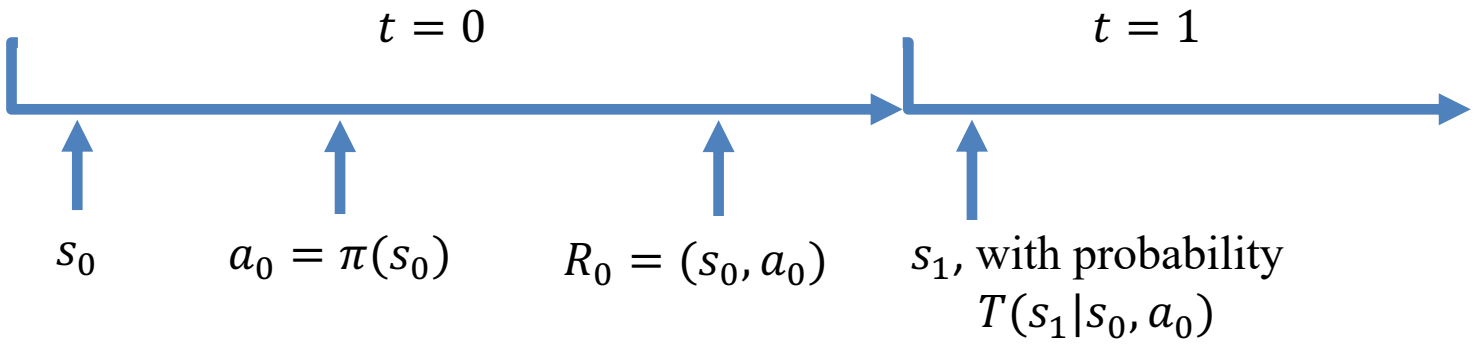
$$\pi: S \rightarrow A;$$

- The policy describes what action to take under each state;
- Finite horizon: $\pi_t(s) = a \in A$;
- Infinite horizon: $\pi(s) = a \in A$ – stationary policy.

Types of Policy

- Time-dependency:
  - Stationary: $\pi_t(s) = \pi_{t'}(s)$, equivalent to $\pi(s)$;
  - Non-stationary: $\pi_t(s) \neq \pi_{t'}(s)$;
- Randomness:
  - Deterministic: $\pi(s) = a$;
  - Randomized: $\Pr\{\pi(s) = a\}, \sum_a \Pr\{\pi(s) = a\} = 1$;
- Pure: stationary + deterministic.

Visualize



Value Function

- Finite horizon:

$$V_t^a(s) = R_t(s,a) + \gamma \sum_{s_{t+1} \in S} T(s_{t+1}|s_t, a_t) \cdot V_{t+1}^*(s), \qquad \forall\, s \in S, a \in A_s$$

- Infinite horizon (stationary):

$$V^a(s) = R(s,a) + \gamma \sum_{s' \in S} T(s'|s,a) \cdot V^*(s'), \qquad \forall\, s \in S, a \in A_s$$

Optimal Value Function

- Bellman Optimality Equation;
- How we find the optimal policy (solution);
- Finite horizon:

$$V_t^*(s) = \max_{a_t \in A_{s_t}} \left\{ R_t(s,a) + \gamma \sum_{s_{t+1} \in S} T(s_{t+1}|s_t, a_t) \cdot V_{t+1}^*(s) \right\}, \qquad \forall\, s_t \in S$$

- Infinite horizon (stationary):

$$V^*(s) = \max_{a \in A_s} \left\{ R(s,a) + \gamma \sum_{s' \in S} T(s'|s,a) \cdot V^*(s') \right\}, \qquad \forall\, s \in S$$

To Find The Policy -- Deterministic

- Finite:

$$\pi_t^*(s) = \underset{a_t \in A_{s_t}}{\operatorname{argmax}} \{V_t^a(s)\}$$

- Infinite:

$$\pi^*(s) = \underset{a \in A_s}{\operatorname{argmax}} \{V^a(s)\}$$

To Find The Policy -- Randomized

- Finite:

$$\Pr\{\pi_t^*(s) = a\} = \frac{V_t^a(s)}{\sum_{a' \in A_{s_t}} V_t^a(s)}$$

- Infinite:

$$\Pr\{\pi^*(s) = a\} = \frac{V^a(s)}{\sum_{a' \in A_s} V^a(s)}$$

Example: Machine Maintenance

- Consider a machine in $N + 1$ conditions:
  - Condition 0: good as new;
  - Condition 1 -- N: degradation levels, large means worse;
  - Under each condition c, $C_o(c)$ is the operating cost;
- The machine's condition degrades at time $t = 0, 1, \ldots$:
  - Probability $p(c, d)$ to degrade from condition c to d;

Example: Machine Maintenance

- Decision maker choose one of M maintenance plans:
  - $m = 1, \dots, M$ is the quality of the maintenance;
  - Has probability $q(m)$ to fully restore the machine;
  - Has maintenance cost $C_m(m)$;
- Fixed revenue C_r at each time epoch;
- Total revenue over time T?

Example: MDP Formulation

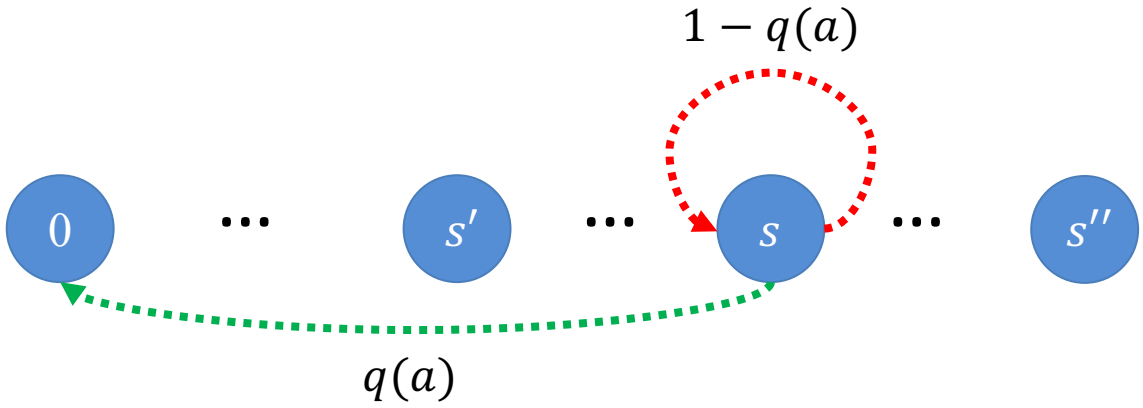
- Decision epoch: $t = 0, 1, \ldots, T$;

- States: $s_t \in S \coloneqq \{0, 1, 2, \ldots, N\}$,
  - Representing the condition of the machine;

- Actions: $a_t \in A \coloneqq \{1, 2, \ldots, M\}$,
  - Representing the type of maintenance;

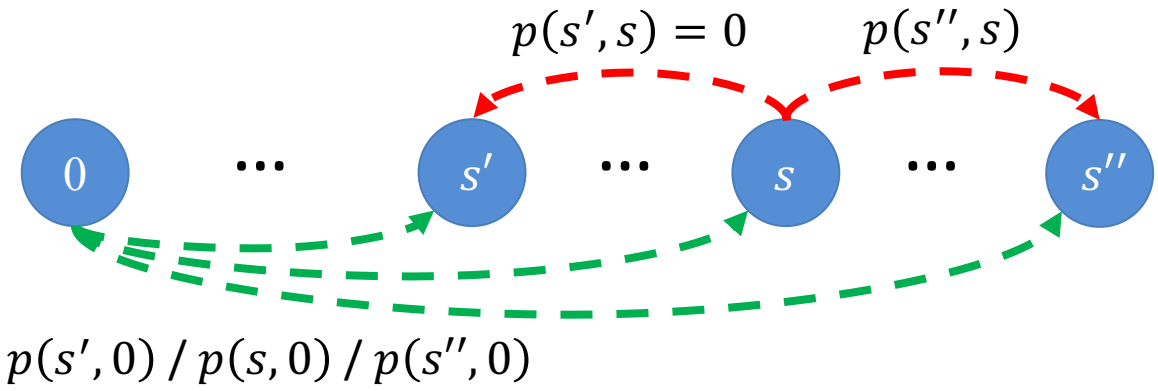
Example: MDP Formulation

- Transition diagram, step 1:



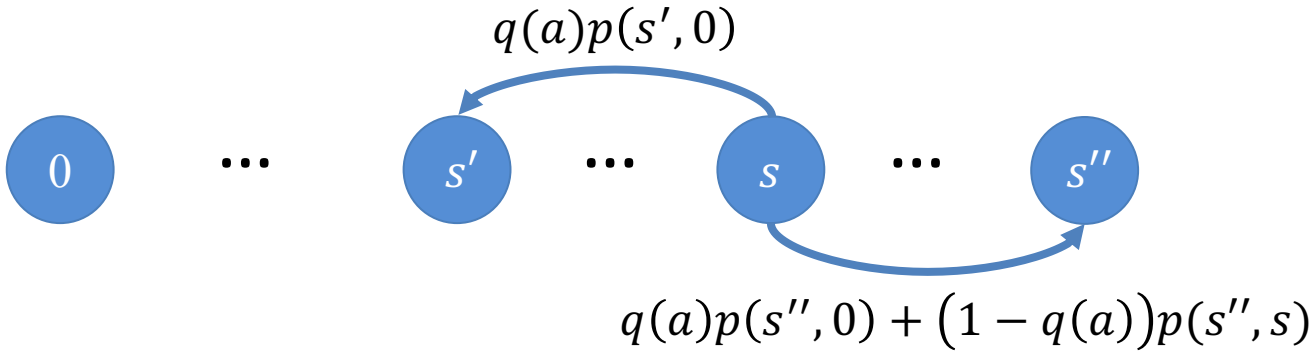
Example: MDP Formulation

- Transition diagram, step 2:



Example: MDP Formulation

- Transition diagram, combine, lose 0:



Example: MDP Formulation

- Transition probability:

$$T(s'|s,a) = \begin{cases} q(a) \cdot p(s',0) & if\ s' < s; \\ q(a) \cdot p(s',0) + \big(1 - q(a)\big) \cdot p(s',s) & otherwise. \end{cases}$$

- Completeness?

Example: MDP Formulation

- Calculation:

$$\sum_{s'} T(s'|s,a) = \sum_{s'<s} q(a)p(s',0) + \sum_{s'\geq s} q(a)p(s',0) + \big(1-q(a)\big)p(s',s)$$

$$= \sum_{s'<s} q(a)p(s',0) + \sum_{s'\geq s} q(a)p(s',0) + \sum_{s'\geq s} \big(1-q(a)\big)p(s',s)$$

$$= \sum_{s'} q(a)p(s',0) + \sum_{s'\geq s} \big(1-q(a)\big)p(s',s)$$

- For all $s' < s$, $p(s',s) = 0$, freely add

$$\sum_{s'<s} \big(1-q(a)\big)p(s',s)$$

Example: MDP Formulation

- Calculation:

$$\sum_{s'} T(s'|s,a) = \sum_{s'} q(a)p(s',0) + \sum_{s'\geq s}\big(1-q(a)\big)p(s',s) + \sum_{s'<s}\big(1-q(a)\big)p(s',s)$$

$$= \sum_{s'} q(a)p(s',0) + \sum_{s'}\big(1-q(a)\big)p(s',s)$$

$$= q(a)\sum_{s'} p(s',0) + \big(1-q(a)\big)\sum_{s'} p(s',s)$$

$$= q(a)\cdot 1 + \big(1-q(a)\big)\cdot 1$$

$$= 1$$

Example: MDP Formulation

- Reward function:

$$R_t(s,a) = C_r - C_o(s_t) - C_m(a_t)$$

- Discount factor: $0 < \gamma < 1$;
- Value function:

$$V_t(s) = C_r - C_o(s_t) - C_m(a_t) + \gamma \sum_{s'<s} q(a)p(s',0) \cdot V_{t+1}(s')$$

$$+\gamma \sum_{s'\geq s} \left[q(a)p(s',0) + \left(1 - q(a)\right)p(s',s)\right] \cdot V_{t+1}\,(s')$$

Monotonicity of Finite MDP

- Finite MDP, $t = 0, 1, \ldots, \infty$;
- Value Function;
- Optimal Policy.

Monotonicity Value Function

- Ordered states and actions:

$$s_0 \leq s_1 \leq s_2 \leq \cdots \leq s_N, \qquad a_0 \leq a_1 \leq a_2 \leq \cdots \leq a_M$$

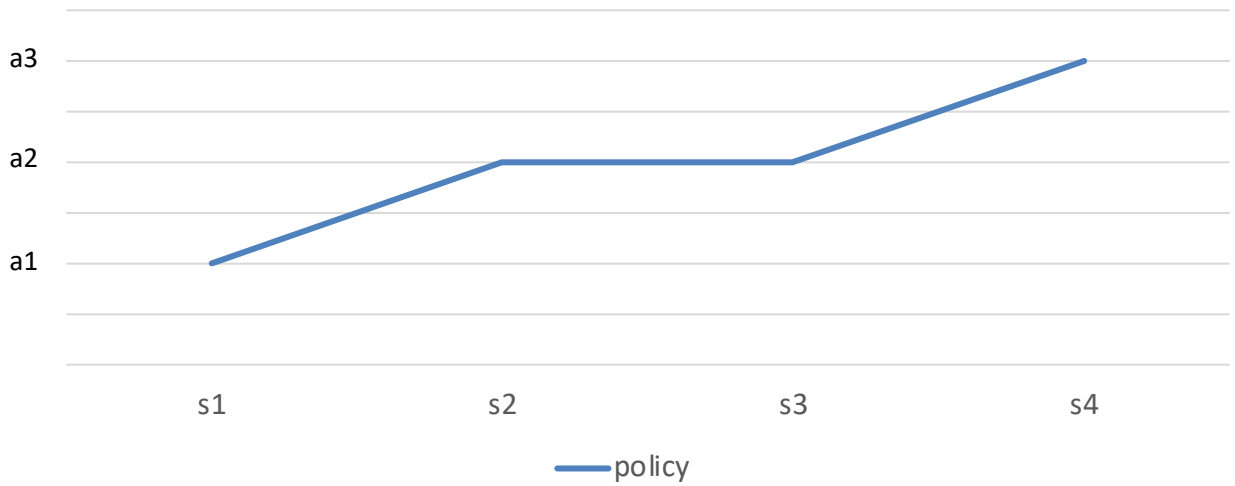
- Monotone Value Function:

$$V_t(s) \leq V_t(s'), \qquad \forall\, s \leq s'$$

Monotone Optimal Policies

- Monotone optimal policy:

$$\pi_t^*(s) = a \leq a' = \pi_t^*(s'), \qquad \forall\, s \leq s'$$



Lemma 1

- Let $\{x_j\}$ and $\{x_j'\}$ be two non-negative series, if

$$\sum_{j=k}^{N} x_j \leq \sum_{j=k}^{N} x_j', \qquad \forall\, k \leq N$$

Then for any $v_{j+1} \geq v_j, j = 0, 1, \ldots, N$, we have

$$\sum_{j=0}^{N} v_j x_j \leq \sum_{j=0}^{N} v_j x_j'$$

Theorem 1: Monotone Value Function

- When:
  - $R_t(s,a) \leq R_t(s',a), \ \ \forall\, s \leq s', \ a \in A;$
  - $R_T(s) \leq R_T(s'), \ \ \forall\, s \leq s';$
  - $\sum_{j \geq k} T(s_j|s,a) \leq \sum_{j \geq k} T\left(s_j \middle| s',a\right), \ \ \forall\, s \leq s', k \leq N, a \in A;$
- We have $V_t^*(s) \leq V_t^*(s'), \ \forall\, s \leq s'.$

Proof: Monotone Value Function

- Prove by induction;
- When $t = T, V_T^*(s) = R_T(s) \leq R_T(s') = V_T^*(s'), \forall\, s \leq s'$;
- Suppose at time $t+1$, $V_{t+1}^*(s) \leq V_{t+1}^*(s'), \forall\, s \leq s'$;
- Look at time t;
- Let a_t^* be the optimal actions:

$$V_t^*(s) = R_t(s, a^*) + \gamma \sum_{j=0}^{N} T\left(s_j \middle| s, a\right) \cdot V_{t+1}^*(s_j)$$

Proof: Monotone Value Function

- Consider $s' \geq s$,

$$V_t^*(s) = R_t(s, a^*) + \gamma \sum_{j=0}^{N} T(s_j | s, a) \cdot V_{t+1}^*(s_j)$$

$$\leq R_t(s', a^*) + \gamma \sum_{j=0}^{N} T(s_j | s, a) \cdot V_{t+1}^*(s_j)$$

$$\leq R_t(s', a^*) + \gamma \sum_{j=0}^{N} T(s_j | s', a) \cdot V_{t+1}^*(s_j)$$

Lemma 1:

$$\sum_{j=0}^{N} v_j x_j \leq \sum_{j=0}^{N} v_j x_j'$$

Proof: Monotone Value Function

$$V_t^*(s) \leq R_t(s', a^*) + \gamma \sum_{j=0}^{N} T\left(s_j \middle| s', a^*\right) \cdot V_{t+1}^*\left(s_j\right)$$

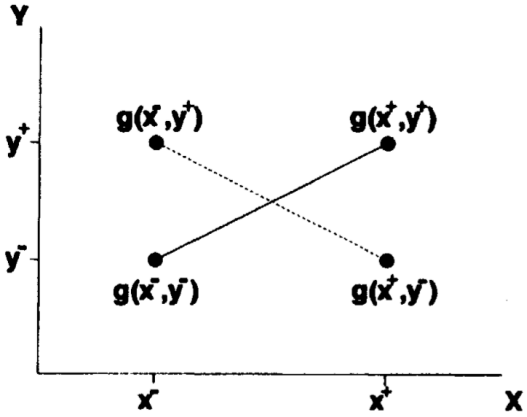
$$\leq \max_{a \in A_{s_t}} \; R_t(s', a) + \gamma \sum_{j=0}^{N} T\left(s_j \middle| s', a\right) \cdot V_{t+1}^*\left(s_j\right)$$

$$= V_t^*(s')$$

Superadditive Functions

- For $x^- \leq x^+, x^-, x^+ \in X$ and $y^- \leq y^+, y^-, y^+ \in Y$,
- A superadditive function $g: X \times Y \to \mathbb{R}$ is defined as

$$g(x^+, y^-) + g(x^-, y^+) \leq g(x^+, y^+) + g(x^-, y^-)$$



Lemma 2

- Let g be a superadditive function on $X \times Y$;
- Define $f: X \to Y$,

$$f(x) = \underset{y \in Y}{\operatorname{argmax}}\ g(x, y)$$

- Then, $f(x)$ is monotone nondecreasing in x.

Theorem: Monotone Optimal Policy (Finite)

- When:
  - $R_t(s,a) \leq R_t(s',a), \quad \forall\, s \leq s',\ a \in A;$
  - $R_T(s) \leq R_T(s'), \quad \forall\, s \leq s';$
  - $\sum_{j \geq k} T(s_j|s,a) \leq \sum_{j \geq k} T\left(s_j \middle| s',a\right), \quad \forall\, s \leq s', k \leq N, a \in A_s;$

  Monotone Value Function

  - $R_t(s,a)$ is superadditive on $S \times A$;
  - $\sum_{j \geq k} T(s_j|s,a)$ is superadditive on $S \times A$ for all $k = 0, 1, \ldots, N$,

  Super-additive
- There exists $\pi_t^*(s)$ nondecreasing in s, for all $t = 0, 1, \ldots, T$.

Proof: Monotone Optimal Policy

- $\sum_{j \geq k} T(s_j|s,a)$ is superadditive:

$$\sum_{j=k}^{N} T(s_j|s^+,a^-) + T(s_j|s^-,a^+) \leq \sum_{j=k}^{N} T(s_j|s^+,a^+) + T(s_j|s^-,a^-)$$

- $V_t^*(s)$ is nondecreasing (Lemma 1):

$$\sum_{j=0}^{N} [T(s_j|s^+,a^-) + T(s_j|s^-,a^+)] V_t^*(s_j) \leq \sum_{j=0}^{N} [T(s_j|s^+,a^+) + T(s_j|s^-,a^-)] V_t^*(s_j)$$

- $\sum_{j=0}^{N} T(s_j|s,a) V_t^*(s_j)$ is superadditive on $S \times A$.

Proof: Monotone Optimal Policy

- We have: $R_t(s,a)$ is superadditive;
- Trivial: $R_t(s,a) + \sum_{j=0}^{N} T\left(s_j \middle| s,a\right) V_t^*(s_j)$ is supperadditive;
- Lemma 2:

$$\pi_t^*(s) = \underset{a \in A}{\operatorname{argmax}}\ R_t(s,a) + \sum_{j=0}^{N} T\left(s_j \middle| s,a\right) V_t^*(s_j)$$

- $\pi_t^*(s)$ is monotone nondecreasing is s.

Algorithms

- Infinite horizon:
  - **Linear programming;**
  - **Value iteration (VI);**
  - Policy iteration;
  - …
- Finite horizon:
  - Backward induction;
  - **Q-Learning;**
  - **Deep Q-network (DQN);**
  - **Advantage Actor Critic (A2C);**
  - …

Algorithm: Linear Programming

- Variable: $v_s \in \mathbb{R}$;
- $0 < \alpha_s \leq 1$;

$$\min_{v_s} \sum_{s \in S} \alpha_s \cdot v_s$$

$$\text{s.t.} \quad v_s - \gamma \sum_{s' \in S} T(s'|s,a) v_{s'} \geq R(s,a) \quad \forall\, s \in S, a \in A;$$

$$v_s \text{ unrestricted} \quad \forall\, s \in S.$$

Algorithm: Dual of Linear Programming

- Dual variable: $x_{s,a} \in \mathbb{R}$;

$$\max_{x_{s,a}} \sum_{s \in S} \sum_{a \in A} R(s,a) \cdot x_{s,a}$$

$$\text{s.t.} \quad \sum_{a \in A} x_{s,a} - \gamma \sum_{s' \in S} \sum_{a \in A} T(s|s',a) x_{s',a} = \alpha_s \quad \forall\, s \in S;$$

$$x_{s,a} \geq 0 \quad \forall\, a \in A.$$

- Policy:

$$\Pr\{\pi^*(s) = a\} = \frac{x_{s,a}}{\sum_{a'} x_{s,a'}}$$

Algorithm: VI

- Set $\tau = 0$;
- Initialize $v_\tau(s) = 0$, for all $s \in S$;
- For each $s \in S$, calculate

$$v_{\tau+1}(s) = \max_{a \in A} \; R(s,a) + \gamma \sum_{s'} T(s'|s,a) \cdot v_\tau(s')$$

- Termination condition:

$$\left|\left|v_{\tau+1}(s) - v_\tau(s)\right|\right| \leq \epsilon, \qquad \forall s \in S$$

- Set $\tau = \tau + 1$.

Algorithm: VI

Algorithm 1: The VI algorithm (Puterman 2014).

Initialize $\tau = 0$ and threshold $\epsilon > 0$;
Initialize $v_\tau(s) = 0, \forall\ s \in S$;
repeat
  $v_{\tau+1}(s) \leftarrow \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in S} T(s'|s,a) \cdot v_\tau(s) \right\}, \quad \forall\ s \in S;$
  $\tau \leftarrow \tau + 1;$
until $||v_\tau - v_{\tau-1}|| \leq \epsilon;$

Complexity: VI

- For each $s \in S$, calculate

$$v_{\tau+1}(s) = \max_{a \in A} \; R(s,a) + \gamma \sum_{s'} T(s'|s,a) \cdot v_\tau(s')$$

$$S \times A \times S$$

Monte-Carlo Simulation

- Recall:

$$V_t^a(s) = R_t(s,a) + \gamma \sum_{s' \in S} T(s'|s_t, a_t) \cdot V_t^*(s'), \qquad \forall\, s_t \in S, a_t \in A_s$$

- Change of notation, **state-action** value function (Q-value):

$$Q_t(s,a) = R_t(s,a) + \gamma \sum_{s' \in S} T(s'|s_t, a_t) \cdot V_{t+1}^*(s'), \qquad \forall\, s_t \in S, a_t \in A_s$$

Monte-Carlo Simulation

- Intuition:

$$Q_t(s,a) = R_t(s,a) + \gamma \sum_{s' \in S} T(s'|s,a) \cdot V^*_{t+1}(s'), \qquad \forall\, s \in S, a \in A_s$$

- Equivalent to

$$Q_t(s,a) = R_t(s,a) + \mathbb{E}[\, R_{t+1}(s,a) + R_{t+2}(s,a) + \cdots + R_T(s,a) \mid s_t, a_t], \qquad \forall\, s_t, a_t$$

- Expectation ≅ Repeated simulations.

Time Difference

- Return $G = R_0 + R_1 + \cdots + R_T$;
- A series of returns $G_1, G_2, \ldots, G_{n-1}$ from repeated experiments, starting in state s;
- Estimate the value with the average:

$$V_0^{n-1}(s) = \frac{\sum_{k=1}^{n-1} G_k}{n-1}$$

Time Difference

- Adding another return G_n;

$$V_0^n(s) = \frac{\sum_{k=1}^{n-1} G_k + G_n}{n-1+1} = \frac{1}{n}\left(G_n + \sum_{k=1}^{n-1} G_k\right)$$

$$= \frac{1}{n}\left(G_n + (n-1)\frac{1}{n-1}\sum_{k=1}^{n-1} G_k\right)$$

$$= \frac{1}{n}(G_n + (n-1)V_0^{n-1}(s))$$

$$= V_0^{n-1}(s) + \frac{1}{n}(G_n - V_0^{n-1}(s))$$

Time Difference

- Summarizing:

$$V_t(s) = V_t(s) + \delta \cdot (G_t - V_t(s))$$

- δ: controllable step size;
- $G_t = R_t + R_{t+1} + \cdots R_T$: return starting from t;
- Equivalently,

$$V_t(s) = V_t(s) + \delta \cdot [R_t + \gamma V_t(s') - V_t(s)]$$

- s': next state in the simulation.

Q-Learning

- Simulate the MDP Ψ times;
- In each simulation:
  - Follow an ε-greedy policy;
  - Update the state-action value using

$$Q\ (s, a) = Q(s, a) + \delta \cdot [R_t + \gamma V(s') - Q(s, a)]$$

Q-Learning

No Transition Probability!

Algorithm 2: Q-learning (Sutton and Barto 2018).

Initialize $Q(s,a) = 0, \forall\, s \in S, a \in A$;
Initialize step size $\delta = 0$, threshold $\varepsilon > 0$ and the maximum iteration Ψ;
for $i \leftarrow 1$ **to** Ψ **by** 1 **do**
  $t \leftarrow 0$;
  repeat
    Choose $a_t = \begin{cases} \arg\max_{a \in A}\{Q(s_t,a)\} & \text{with probability } 1-\varepsilon, \\ \text{randomly} & \text{with probability } \varepsilon; \end{cases}$
    Run simulation using (s_t, a_t), and observe R_t and s_{t+1};
    $Q(s,a) \leftarrow Q(s,a) + \delta\Big[R_t + \gamma \cdot \max_{a \in A}\{Q(s_{t+1},a)\} - Q(s,a)\Big], \forall\, s \in S, a \in A$;
    $t \leftarrow t+1$;
  until $t \geq T+1$;
end

On/Off-policy Learning

- Target policy: final solution, the policy we learn;
  - E.g. $\pi(s) = \underset{a \in A}{\operatorname{argmax}}\, Q(s, a)$;
- Behavior policy: a policy used in the simulations;
  - E.g. ε-greedy;
- On-policy: target policy = behavior policy;
- Off- policy: target policy ≠ behavior policy;
- Exploration vs. Exploitation.

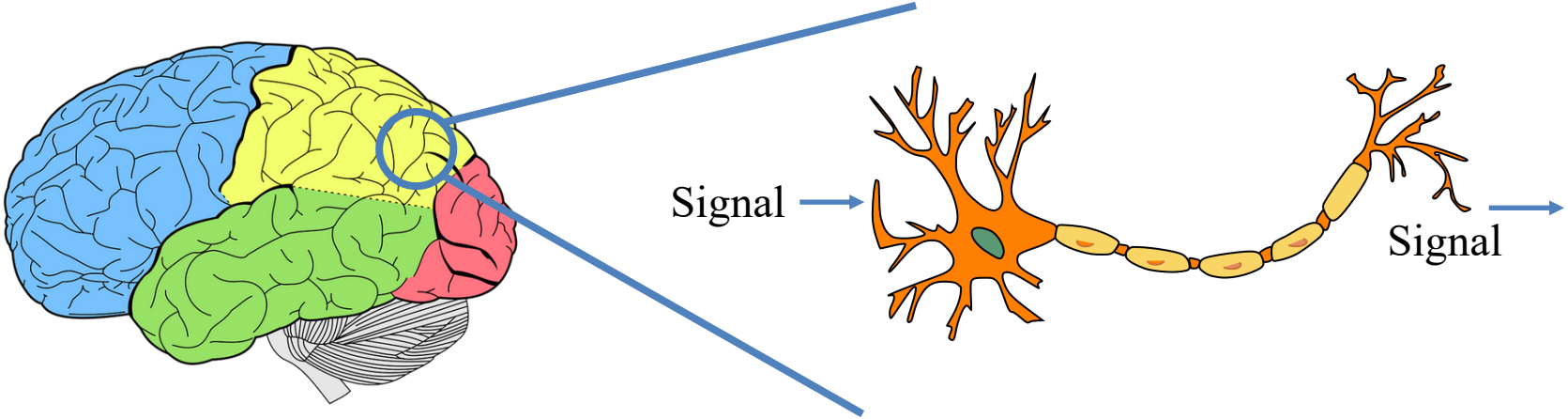
Tabular Method vs. Function Approximator

- Tabular methods: Keeps a "table" of Q-values;

$Q(s,a)$	a_0	a_1	…	a_m
s_0				
s_1				
…				
s_n				

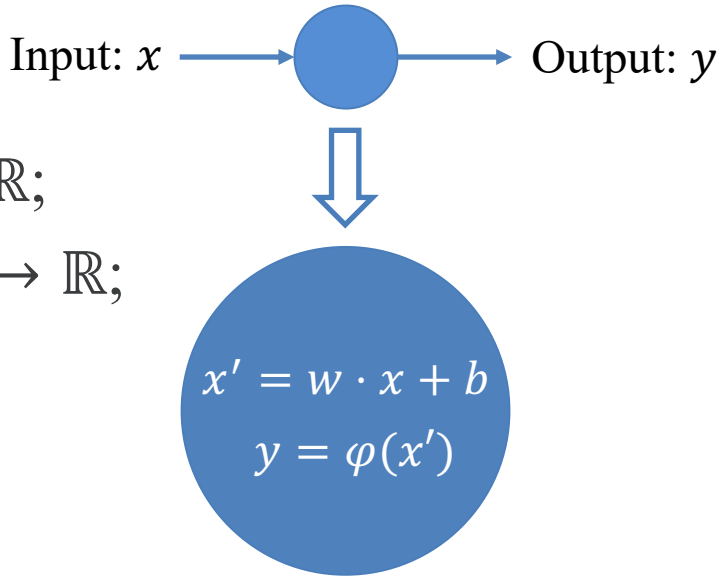
- Function approximator: $f: S \times A \rightarrow \mathbb{R}$;
  - More compact than a table;
  - Generalizable to unexplored (s, a);

Neural Network (NN)



Neuron

- Input $x \in \mathbb{R}$;
- Output $y \in \mathbb{R}$;
- Linear: weight $w \in \mathbb{R}$ and bias $b \in \mathbb{R}$;
- Nonlinear: activation function $\varphi: \mathbb{R} \to \mathbb{R}$;



Activation Function

- Linear:

$$\varphi(x) = x$$

- Step:

$$\varphi(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Activation Function

- Logistic (Sigmoid):

$$\varphi(x) = \frac{1}{1 + e^{-x}}$$

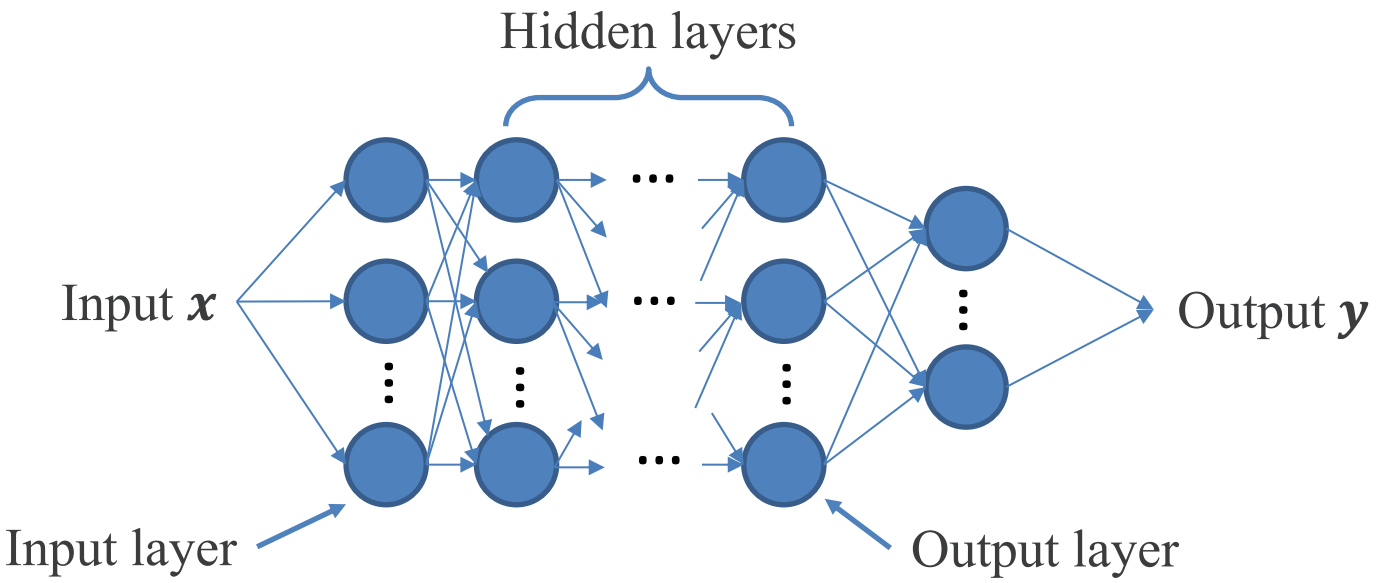
- Rectified Linear (ReLU):

$$\varphi = \max(x, 0)$$

- ……

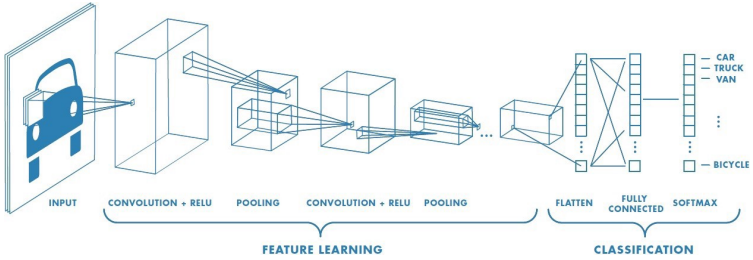
Neural "Network"

- Function, $f: \mathbb{R}^N \to \mathbb{R}^M$;

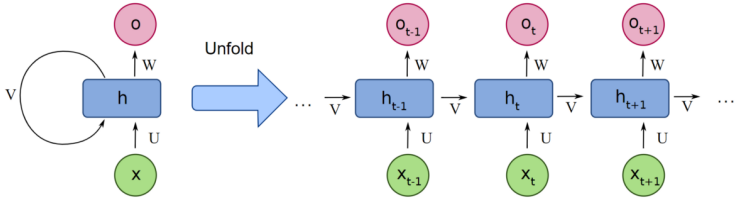


Types of NN

- Multi-layer perceptron (MLP);
- Convolutional NN;



- Recurrent NN;



Output Quality

- Dataset $X_{N\times M}$;
  - M variables and N records;
- Target values (response) $Y_{N\times L}$;
  - Output has dimension L, N records;
- Input to NN f;
  - M neurons in input and L neuron in output;
- Output $Y'_{N\times L}$;
- Measure $Y'_{N\times L}$: loss function.

Loss Function -- Regression

- Mean square error:

$$E(Y, Y') = \frac{1}{N}\sum_i (Y_i' - Y_i)^2$$

- Mean absolute error:

$$E(Y, Y') = \frac{1}{N}\sum_i |Y_i' - Y_i|$$

- ……

Loss Function -- Classification

- Binary cross entropy error:

$$E(Y, Y') = \frac{1}{N}\sum_{i} -(Y_i \cdot \log(Y_i') + (1 - Y_i) \cdot \log(1 - Y_i'))$$

- Categorical cross entropy error:

$$E(Y, Y') = \frac{1}{N}\sum_{i}^{N}\sum_{j}^{C} -Y_i \cdot \log(Y_i')$$

- ……

Optimize NN (Training)

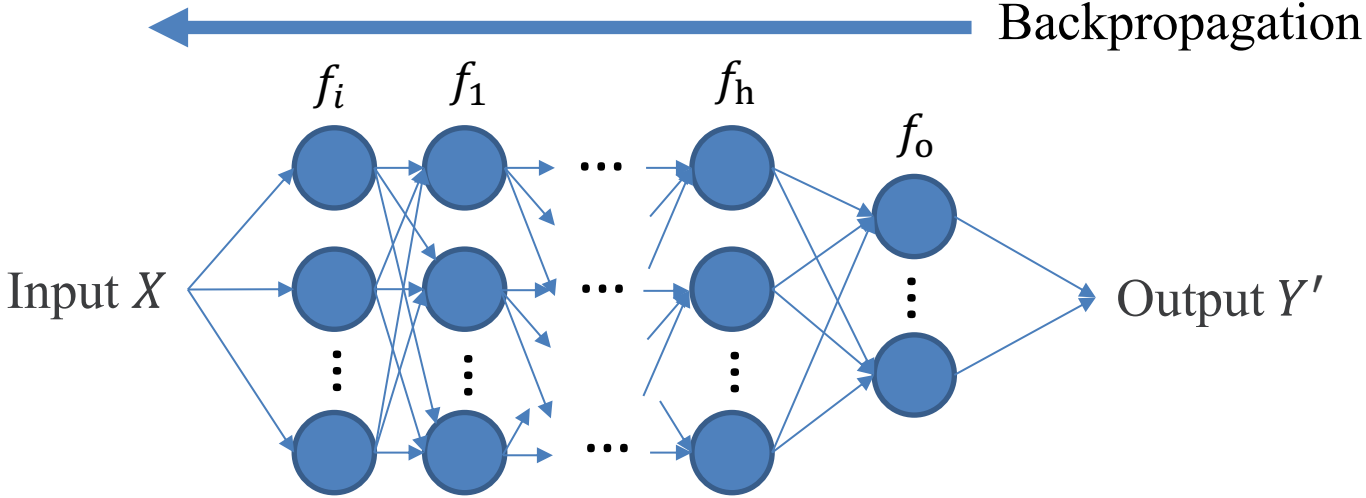
- NN with weights $\boldsymbol{w}$ and input X: $f^{\boldsymbol{w}}(X)$;
- Objective:

$$\min_{\boldsymbol{w}} \; E(Y, f^{\boldsymbol{w}}(X))$$

- Gradient descend;

Calculate Gradient

- $Y' = f_o(\, f_h(\, f_{h-1}(\, \ldots f_1(\, f_i(X)\,)\, \ldots\,)\,)\,);$
- $\dfrac{\partial Y'}{\partial \boldsymbol{w}} = \dfrac{\partial f_o(\cdot)}{\partial \boldsymbol{w}} \cdot \dfrac{\partial f_h(\cdot)}{\partial \boldsymbol{w}} \cdot \ldots \cdot \dfrac{\partial f_1(\cdot)}{\partial \boldsymbol{w}} \cdot \dfrac{\partial f_i(\cdot)}{\partial \boldsymbol{w}}$

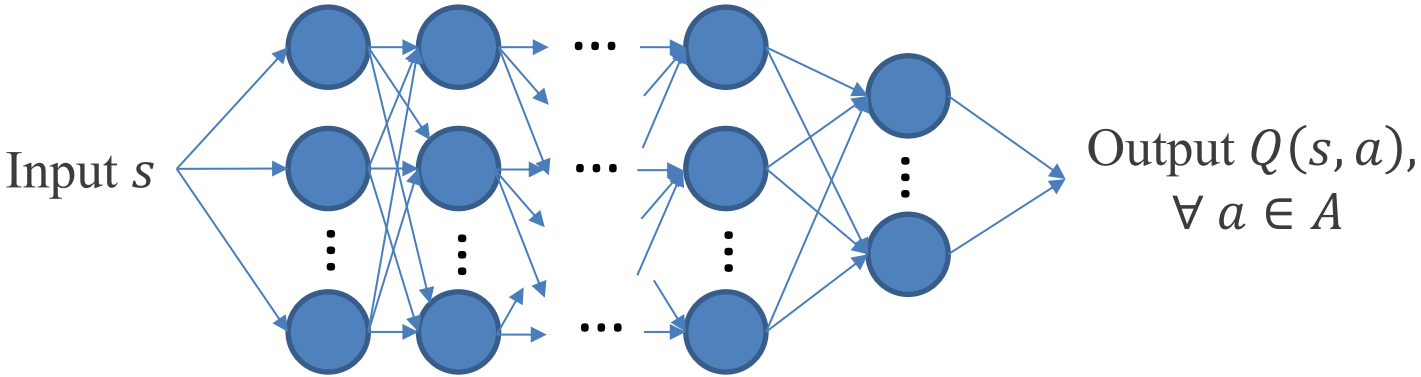


Algorithms to Train NN

- Gradient descend;
- Stochastic gradient descend (SGD);
- **Adam**;
- Adadelta;
- RMSPROP;
- ……

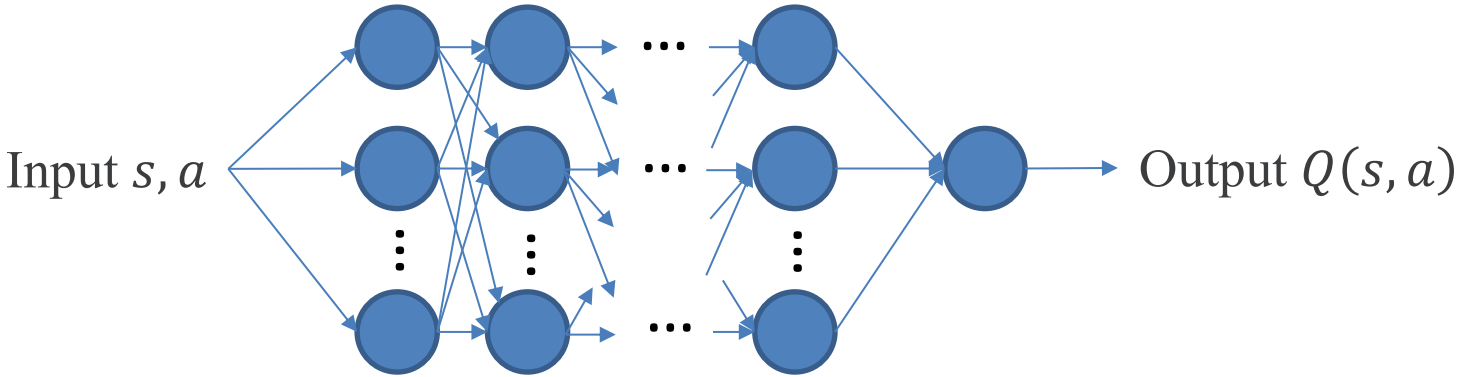
Algorithm: Deep Q-Network (DQN)

- Q-Learning with NN to approximate $Q(s, a)$;
- Input dimension $|s|$, $s \in S$;
- Output dimension $|A|$;



Algorithm: Deep Q-Network (DQN)

- Another way:
- Input dimension $|s| + 1$, $s \in S$, $+1$ for $a \in A$;
- Output dimension 1, $\mathrm{Q(s, a)}$;



Algorithm: Deep Q-Network (DQN)

- Implementation, two NNs;
- Target network: $\hat{Q}(s, a)$;
  - Used to update Q-values;
- Policy network: $Q(s, a)$;
  - Used to control policy behaviors;

Update with policy NN

Only train policy NN

Algorithm: Deep Q-Network (DQN)

- Data?
- Memory: $(s_t, a_t, R_t, s_{t+1}) \in \mathcal{M}$;
- Sample training dataset from the memory:
  - Variable: $s_t = (0, 1, 2, 3, \cdots)$;
  - Response: $Q_t(s, a) = R_t + \gamma \cdot \max_{a'} \hat{Q}(s_{t+1}, a')$
- "Experience replay".

s			$(Q_{a_0}, Q_{a_1}, \dots)$
0	…	10	(100, 80, …)
3	…	9	(105, 78, …)
……			…
2	…	7	(120, 65, …)

Algorithm: Deep Q-Network (DQN)

Algorithm 3: The DQN algorithm (Mnih et al. 2015).

```
Initialize the policy network Q with weight ζ;
Initialize the target network Q̂ with weight ζ⁻ = ζ;
Initialize ε > 0, replay memory M = ∅, learning frequency C, target network
  update frequency F and maximum training iteration Ψ;
for i ← 1 to Ψ by 1 do
    Initialize t = 0 and the initial state s_0;
    repeat
        Sample action a_t using ε-greedy policy: with probability ε, randomly
          select a_t ∈ A; with probability 1 − ε, select a_t = arg max_{a'∈A} Q(s_t, a');
        Simulate reward r_t(s, a) and new state s_{t+1} from the environment;
        Store (s_t, a_t, r_t, s_{t+1}) in the replay memory M;
        t ← t + 1;
        if i % C = 0 then
            Sample a mini-batch of records (s_j, a_j, r_j, s_{j+1}) from M;
            Set γ_j = { r_j,                          if episode terminates at s'_j,
                      { r_j + λ · max_a Q̂(s'_j, a),   otherwise;
            Perform gradient descent on (γ_j − Q(s_j, a_j))^2 with respect to the
              network parameter ζ;
        else
            continue;
        end
        if i % F = 0 then
            Set ζ⁻ = ζ;
        else
            continue;
        end
    until s_t = Terminate;
end
```

Python Prerequisite

- “Pypeline”: https://github.com/t-wolfeadam/AnyLogic-Pypeline
- Installation;

Py Communicator

Never forget the " " !!!

```
// run a code
pyCom.run("import numpy as np");

pyCom.run("x = 1");
```

Command Prompt - python

```
Microsoft Windows [Version 10.0.19043.1288]
(c) Microsoft Corporation. All rights reserved.

C:\Users\louis>python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> x = 1
>>> _
```

Py Communicator

```
// run a code and get results back
int a = pyCom.runResults (
    int.class,
     "x"
);
```

Command Prompt - python

```
Microsoft Windows [Version 10.0.19043.1288]
(c) Microsoft Corporation. All rights reserved.

C:\Users\louis>python
Python 3.9.1 (tags/v3.9.1:1e5d33e, Dec  7 2020, 17:08:21) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> x = 1
>>> x
1
>>> _
```

Py Communicator

```
// Advance: use variables defined in JAVA
int a = 1;
int b = 2;

// add a and b with numpy
// import
pyCom.run("import numpy as np");
```

```
// use the function
pyCom.run(
    "np.add(a, b)"
);
```

?

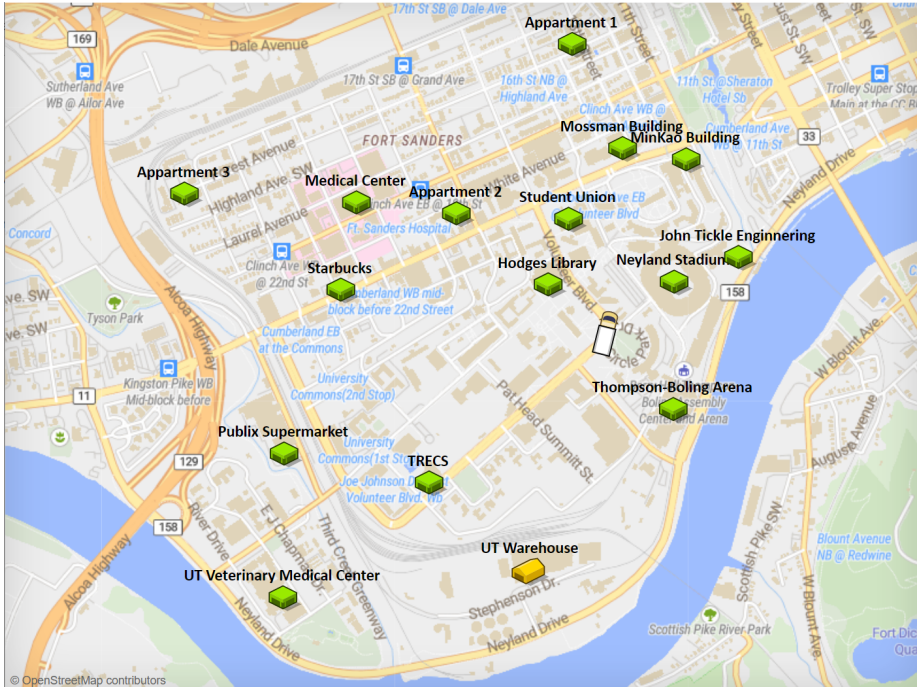
```
// use the function
pyCom.run(String.format(
    "c = np.add(%s, %s)", a, b
));
int c = pyCom.runResults(int.class, "c");
```

Py Communicator

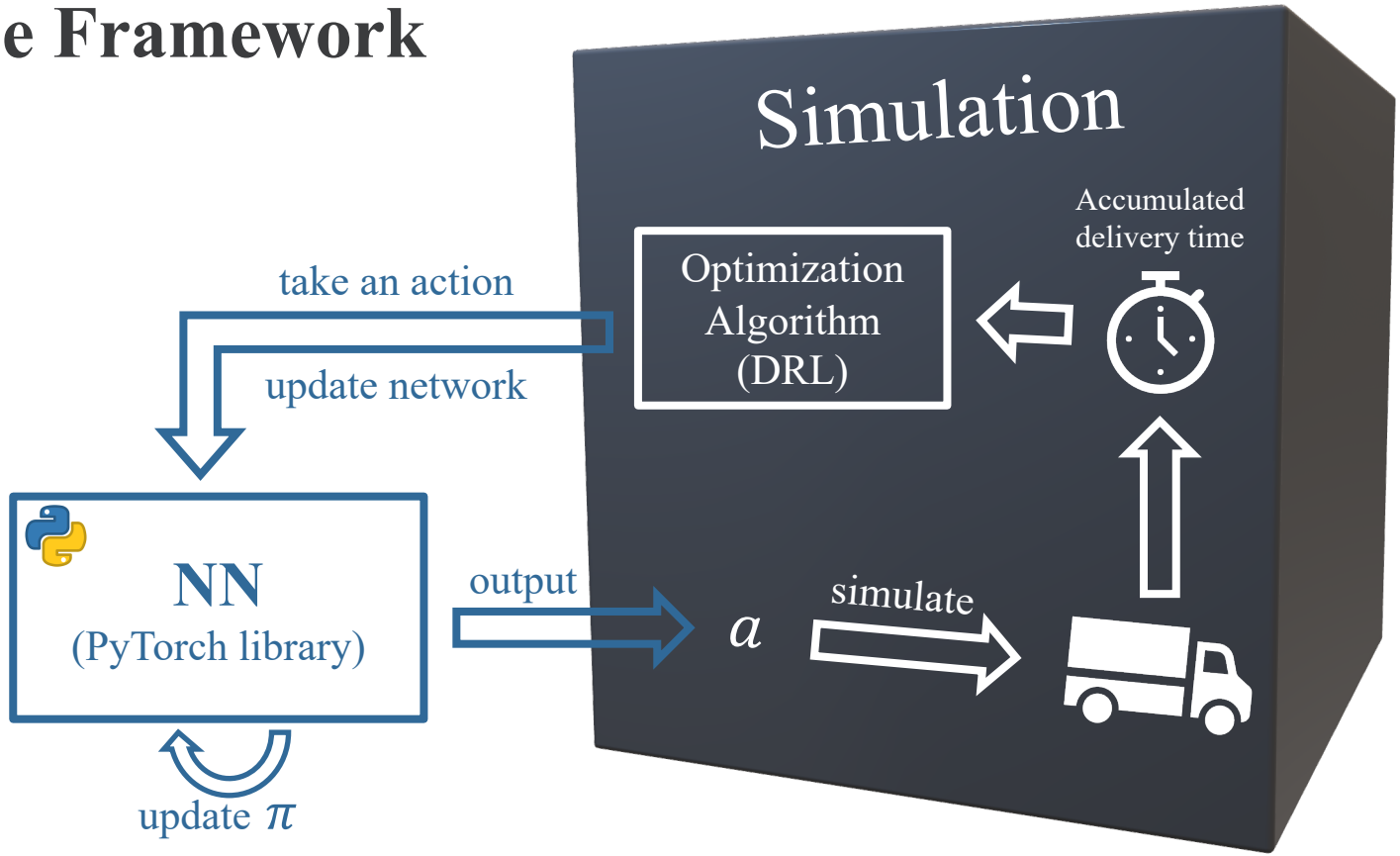
```
// Advance: use an array
int[] a = {0, 1, 2, 3};

// calculate sum
// import
pyCom.run("import numpy as np");

// use the function
pyCom.run(String.format(
     "b = np.sum(%s)", Arrays.toString(a)
));
int b = pyCom.runResults(int.class, "b");
```

The Problem

- Delivering packages to customers around University of Tennessee campus;
- A vehicle routing problem (VRP);



The Framework



VRP: States

- $s \in S,\ s = [2, 0, 1, 0, 0, \ldots, 0]$

Current location

Whether a **customer** is served (1) or not (0)

VRP: Actions

- $a \in A, \ a = 3$

Which location to go to in the next epoch

VRP: Rewards

- Delivery time, $R_t(s, a) \in \mathbb{R}_+$;

VRP: Transition?

- Modeled in AnyLogic;
- “Model free”.

On Startup

```
// import
pyCom.run(
     "from DRL import Agent, Memory"
);
// initialize all actions
initialize_actions();
// initialize agent Env
pyCom.run(String.format(
     "agent = Agent(\"%s\", %s, %s, %s, %s, %s, action_filter=%s)",
     name, Arrays.toString(actions), input_size,
     Arrays.toString(hidden_layers), output_size,
     learning_rate, action_filter
));
// initialize memory
pyCom.run(String.format(
     "memory = Memory(%s, %s)", memory_size, sample_episodes
));
```

On Startup

```
// take an action
pyCom.run(String.format(
     "action = agent.take_action(%s,)", Arrays.toString(state)
));
action = pyCom.runResults(
     int.class, "action"
);
// new state, change current location and delivery status
state_new[0] = action;
state_new[action] = 1;
// go to destination
reward = time();
this.moveTo(
findFirst(
     main.customers,
          p -> p.number == action
     )
);
```

On Arrival to Target Location

```
//reward
reward = reward - time();
G += reward;
// update memory
pyCom.run(String.format(
     "memory.update(%s, %s, %s, %s, %s)",
     Arrays.toString(state), delta, action,
     reward, Arrays.toString(state_new)
));
// learn or not
if (step % learn_step == 0) {
// learn
pyCom.run(String.format(
     "agent.learn(memory.sample(), %s)",
     discount_factor
));
```

On Arrival to Target Location

```
// learn or not
if (step % learn_step == 0) {
    // learn
    pyCom.run(String.format(
        "agent.learn(memory.sample(), %s)",
        discount_factor
    ));
    // obtain loss
    loss = pyCom.runResults(
        double.class, "agent.loss_memory[-1]"
    );
    main.loss_plot.updateData();
}
```

On Arrival to Target Location

```
// next step
step ++;
state = state_new;
// take an action only when not done
if (!done) {
    pyCom.run(String.format(
        "action = agent.take_action(%s)",
        Arrays.toString(state)
    ));
    action = pyCom.runResults(
        int.class, "action"
    );
}
// new state, change current location and delivery status
state_new[0] = action;
if (action != 0) {
    state_new[action] = 1;
}
```

On Arrival to Target Location

```
// continue delivery
if (state_new[0] != 0) {
     ......
// all packages delivered, at warehouse
} else if (done) {
     ......
// all packages delivered, going to warehouse
} else {
     ......
}
```

On Arrival to Target Location – Continue Delivery

```
// go to destination
reward = time();
this.moveTo(
findFirst(
     main.customers,
          p -> p.number == action
     )
);
```

On Arrival to Target Location – Go to Warehouse

```
// go to warehouse
reward = time() + terminal_reward;
this.moveTo(
    findFirst(
        main.warehouses,
        p -> p.number == action
    )
);
done = true;
delta = "True";
```

On Arrival to Target Location – At Warehouse

```
// update figure
G = G - terminal_reward;
main.G_plot.updateData();
// save G
pyCom.run(String.format(
     "agent.G_memory.append(%s)", G
));
// update epsilon
pyCom.run(String.format(
     "agent.epsilon = %s", update_epsilon()
));
// check termination condition
if (main.iteration > main.max_iteration) {
     ......
} else {
     ......
}
```

On Arrival to Target Location – Termination

```
// plot using python
pyCom.run(
     "agent.plot_G()",
     "agent.plot_loss()"
);
// print best delivery time
traceln("learned!");
// stop
getEngine().finish();
```

On Arrival to Target Location – Continue

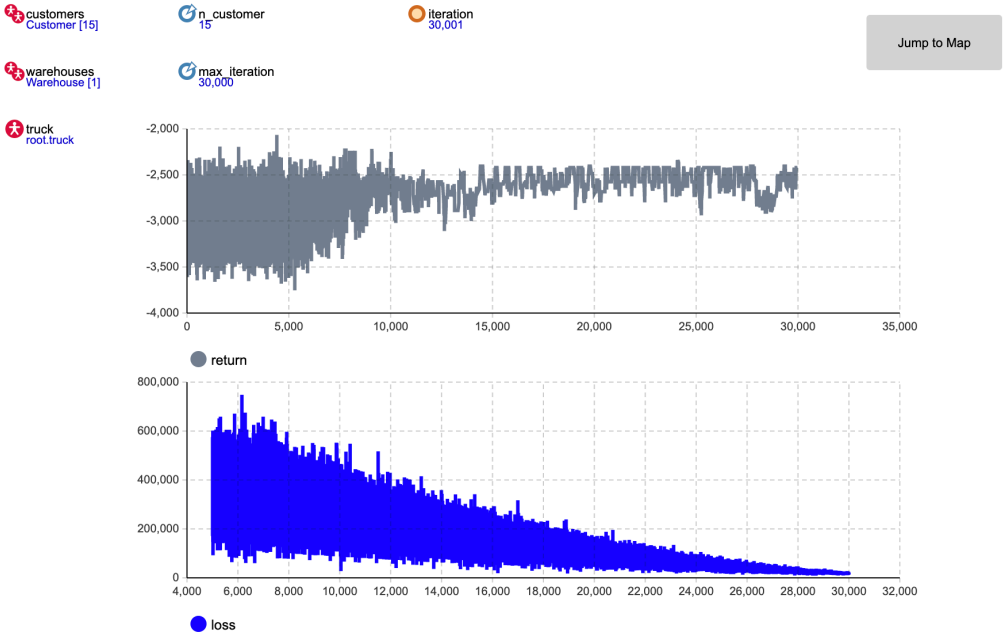
```
// continue, reinitialize
main.iteration ++;
G = 0;
done = false;
delta = "False";
state = new int[main.n_customer + 1];
state_new = state;
// take an action
pyCom.run(String.format(
     "action = agent.take_action(%s)",
     Arrays.toString(state)
));
action = pyCom.runResults(
     int.class, "action"
);
```

On Arrival to Target Location – Continue

```
// new state, change current location and delivery status
state_new[0] = action;
state_new[action] = 1;
// go to destination
reward = time();
this.moveTo(
    findFirst(
        main.customers,
        p -> p.number == action
    )
);
```

The Framework with DQN

- Final delivery time: 2420.12 seconds.



Algorithm: Advantage Actor Critic (A2C)

- Value- and Policy-based method;
- Value function of a policy:

$$J(\theta) \coloneqq \mathbb{E}_{\pi}[\sum_{t} R_t \,|\pi_\theta]$$

- Let $P(s_t, a_t|\pi_\theta)$ be the probability having state s_t and action a_t at time t under policy π_θ;

$$J(\theta) = \sum_{\pi} \sum_{t} P(s_t, a_t|\pi_\theta) \cdot R_t$$

Algorithm: Advantage Actor Critic (A2C)

- Calculating the gradient of $J(\theta)$:

$$\nabla_\theta J(\theta) = \sum_\pi \sum_t \nabla_\theta P(s_t, a_t|\pi_\theta) \cdot R_t$$

$$\frac{\partial}{\partial x} \log f(x) = \frac{f'(x)}{f(x)}$$

$$= \sum_\pi \sum_t P(s_t, a_t|\pi_\theta) \frac{\nabla_\theta P(s_t, a_t|\pi_\theta)}{P(s_t, a_t|\pi_\theta)} \cdot R_t$$

$$= \sum_\pi \sum_t P(s_t, a_t|\pi_\theta) \nabla_\theta \log P(s_t, a_t|\pi_\theta) \cdot R_t$$

$$= \mathbb{E}_\pi[\sum_t \nabla_\theta \log P(s_t, a_t|\pi_\theta) \cdot R_t]$$

Algorithm: Advantage Actor Critic (A2C)

- With further calculations (actor-critic):

$$\nabla_\theta J(\theta) \cong \log \Pr\{\pi_\theta(s_t) = a_t\} \left(Q_t(s,a) - V_t(s)\right)$$

- Advantage:

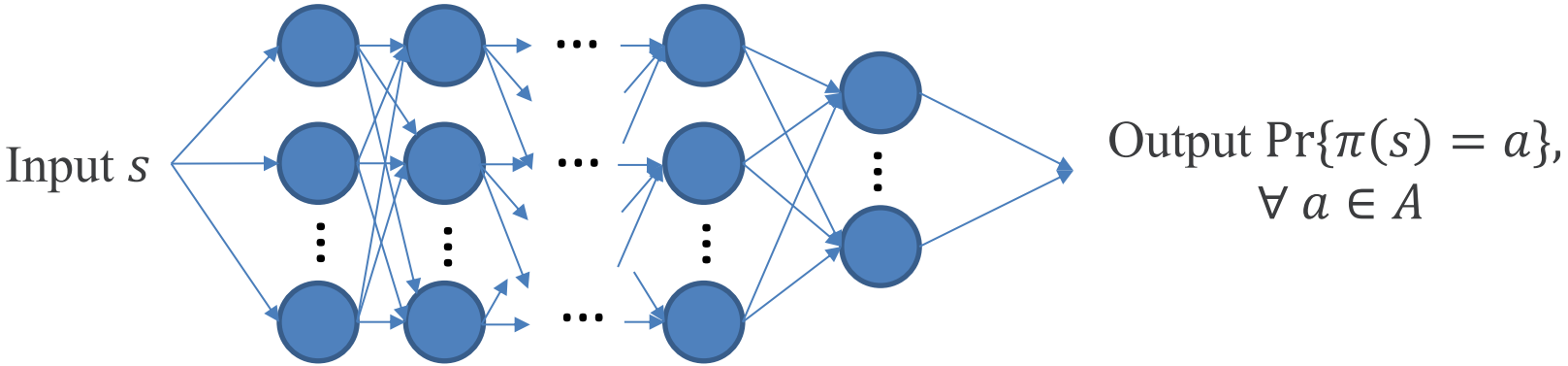
$$\mathcal{A}_t(s,a) = Q_t(s,a) - V_t(s)$$

- Gradient of policy function:

$$\nabla_\theta J(\theta) \cong \log \Pr\{\pi_\theta(s_t) = a_t\} \mathcal{A}_t(s,a)$$

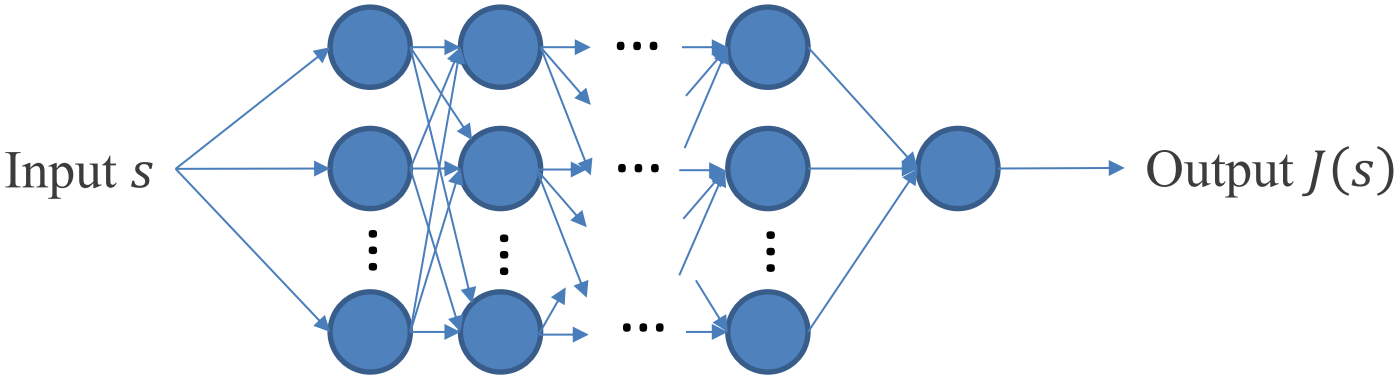
Algorithm: Advantage Actor Critic (A2C)

- Implementation, two NNs:
- An actor network $\tilde{\pi}(a|s,\theta) = \Pr\{\pi_\theta(s) = a\}$;
  - Controls the simulation;
  - Improves during learning;



Algorithm: Advantage Actor Critic (A2C)

- A critic network $J(s|\kappa) \cong V^*(s)$, with parameter κ;
  - "Criticize" the policy of the actor network;
  - Used to calculate the advantage;



Algorithm: Advantage Actor Critic (A2C)

Algorithm 4: The A2C algorithm (Mnih et al. 2016).

Initialize the policy distribution (actor network) $\tilde{\pi}(a|s,\theta)$ with learning rate ν_θ;
Initialize the value estimator (critic network) $J(s|\kappa)$ with learning rate ν_κ;
Initialize the number of episodes Ψ;
for $i \leftarrow 1$ **to** Ψ **by** 1 **do**
  Initialize $t = 0$, the initial states_0, $a_t \sim \tilde{\pi}(a_t|s_t,\theta)$;
  repeat
    Simulate reward $r(s_t, a_t)$ and new state s_{t+1} from the environment;
    Update the policy parameter $\theta \leftarrow \theta + \nu_\theta J_t(s|\kappa)\nabla_\theta \log \tilde{\pi}(a_t|s_t,\theta)$;
    $t \leftarrow t + 1$;
    Sample new action $a_t \sim \tilde{\pi}(a_t|s_t,\theta)$;
    Calculate the advantage $\mathcal{A}_t \leftarrow r_{t-1} + J_t(s|\kappa) - J_{t-1}(s|\kappa)$;
    Update the value estimator parameter $\kappa \leftarrow \kappa + \nu_\kappa \mathcal{A}_t \nabla_\kappa J_t(s|\kappa)$;
  until $s_t = \mathit{Terminate}$;
end

On Policy!

On Startup

```
// import
pyCom.run(
     "import numpy as np",
     "from DRL import Agent",
     "np.random.seed(1)"
);
// initialize all actions
initialize_actions();
// initialize agent Env
pyCom.run(String.format(
     "agent = Agent(\"%s\", %s, %s, %s, %s, %s, %s, %s, action_filter=%s)",
      name, Arrays.toString(actions), input_size,
      Arrays.toString(actor_hidden_layers),
      Arrays.toString(critic_hidden_layers),
      output_size, actor_lr, critic_lr, action_filter
));
```

On Startup

```
// take an action
pyCom.run(String.format(
    "action = agent.take_action(%s,)", Arrays.toString(state)
));
action = pyCom.runResults(
    int.class, "action"
);
// new state, change current location and delivery status
state_new[0] = action;
state_new[action] = 1;
// go to destination
reward = time();
this.moveTo(
    findFirst(
        main.customers,
        p -> p.number == action
    )
);
```

On Arrival to Target Location

```
//reward
reward = reward - time();
G += reward;
// update reward memory
pyCom.run(String.format(
     "agent.reward_memory.append(%s)", reward
));
```

On Arrival to Target Location

```
// learn or not
if (done) {
    // learn
    pyCom.run(String.format(
        "agent.learn(%s)", discount_factor
    ));
    // obtain loss
    actor_loss = pyCom.runResults(
        double.class, "agent.actor_loss[-1]"
    );
    critic_loss = pyCom.runResults(
        double.class, "agent.critic_loss[-1]"
    );
    main.actor_loss.updateData();
    main.critic_loss.updateData();
}
```

On Arrival to Target Location

```
// next step
step ++;
state = state_new;
// take an action only when not done
if (!done) {
     pyCom.run(String.format(
          "action = agent.take_action(%s)",
          Arrays.toString(state)
     ));
     action = pyCom.runResults(
          int.class, "action"
     );
}
// new state, change current location and delivery status
state_new[0] = action;
if (action != 0) {
     state_new[action] = 1;
}
```

On Arrival to Target Location

```
// continue delivery
if (state_new[0] != 0) {
     ......
// all packages delivered, at warehouse
} else if (done) {
     ......
// all packages delivered, going to warehouse
} else {
     ......
}
```

On Arrival to Target Location – Continue Delivery

```
// go to destination
reward = time();
this.moveTo(
findFirst(
     main.customers,
          p -> p.number == action
     )
);
```

On Arrival to Target Location – Go to Warehouse

```
// go to warehouse
reward = time() + terminal_reward;
this.moveTo(
    findFirst(
        main.warehouses,
        p -> p.number == action
    )
);
done = true;
```

On Arrival to Target Location – At Warehouse

```
// update figure
G = G - terminal_reward;
main.G_plot.updateData();
// save G
pyCom.run(String.format(
     "agent.G_memory.append(%s)", G
));
// check termination condition
if (main.iteration > main.max_iteration) {
     ......
} else {
     ......
}
```

On Arrival to Target Location – Termination

```
// plot using python
pyCom.run(
     "agent.plot_G()",
     "agent.plot_loss()"
);
// print best delivery time
traceln("learned!");
// stop
getEngine().finish();
```

On Arrival to Target Location – Continue

```
// continue, reinitialize
main.iteration ++;
G = 0;
done = false;
state = new int[main.n_customer + 1];
state_new = state;
// take an action
pyCom.run(String.format(
     "action = agent.take_action(%s)",
     Arrays.toString(state)
));
action = pyCom.runResults(
     int.class, "action"
);
```

On Arrival to Target Location – Continue

```
// new state, change current location and delivery status
state_new[0] = action;
state_new[action] = 1;
// go to destination
reward = time();
this.moveTo(
    findFirst(
        main.customers,
        p -> p.number == action
    )
);
```

The Framework with A2C

- Final delivery time: 2116.78 seconds.

customers
Customer [15]

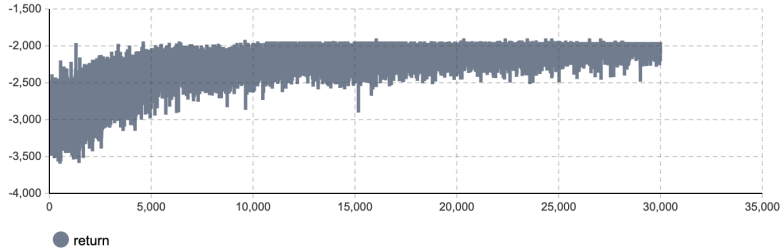
n_customer
15

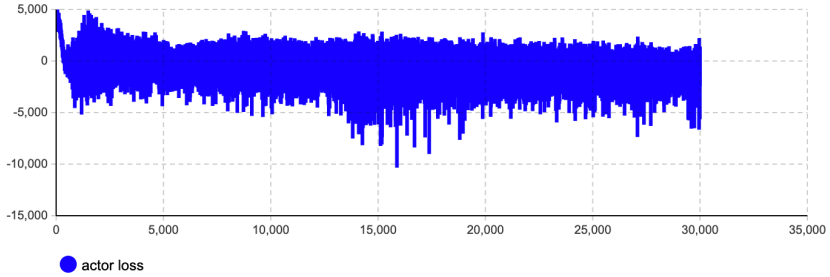
iteration
30,001

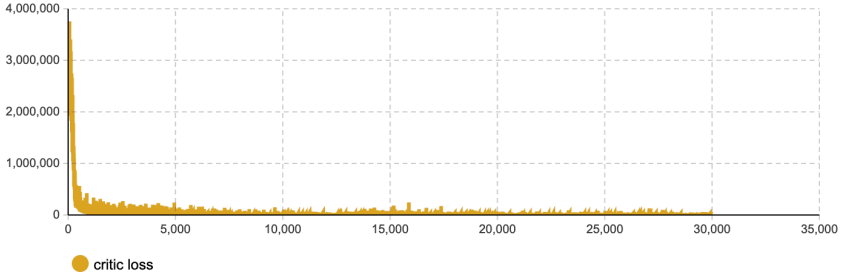
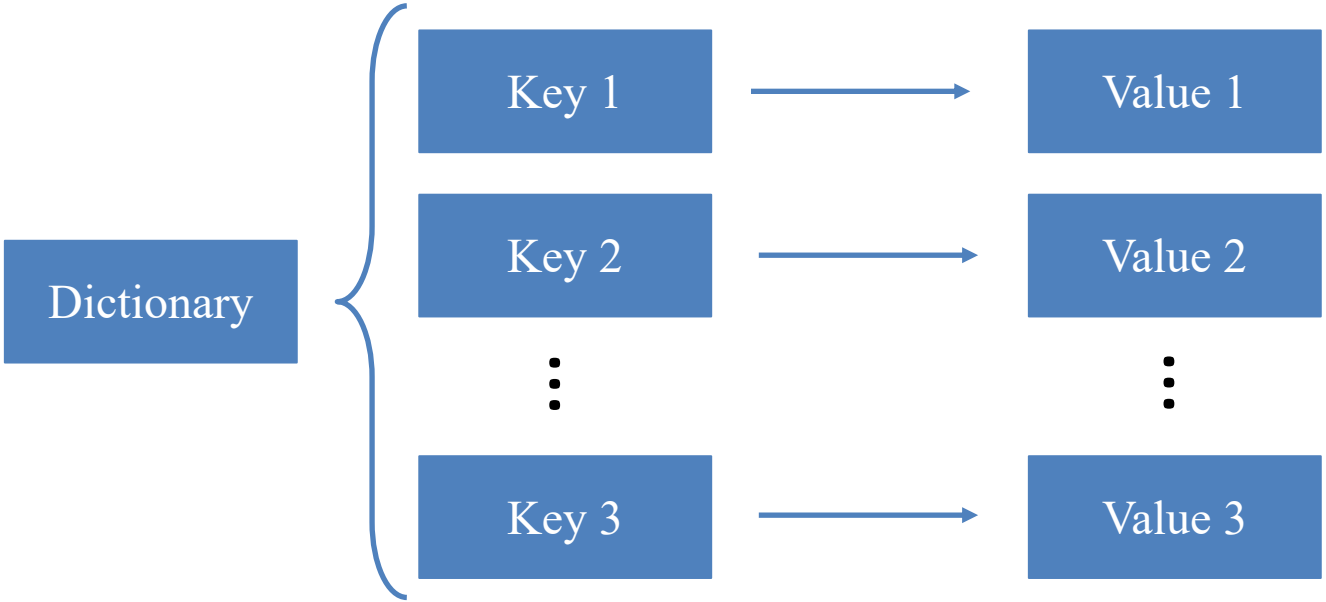
warehouses
Warehouse [1]

max_iteration
30,000

truck
root.truck







LinkedHashMap in Java