CS5242 : Neural Networks and Deep Learning

Lecture 13: Deep Reinforcement Learning

Semester 1 2021/22

Xavier Bresson

https://twitter.com/xbresson

Department of Computer Science
National University of Singapore (NUS)

Outline

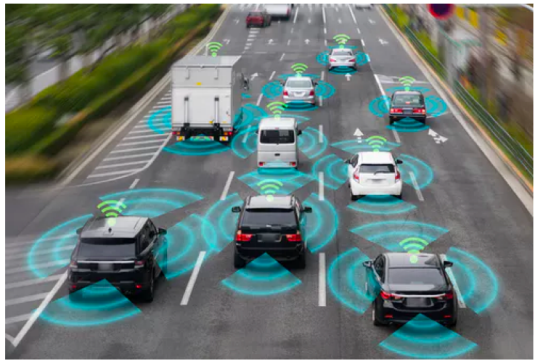
- RL and Deep Learning
- Agent, Environment and MDP
- Policy, Value Function and Model
- Optimal Value Function and Policy
- Deep Q-Learning (DQN)
- Policy Networks
- Actor-Critic Algorithm
- RL and Supervised Learning
- Learning and Planning

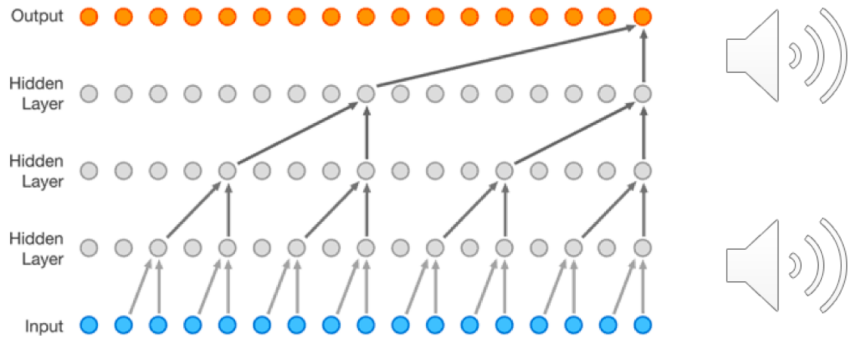
Outline

- RL and Deep Learning
- Agent, Environment and MDP
- Policy, Value Function and Model
- Optimal Value Function and Policy
- Deep Q-Learning (DQN)
- Policy Networks
- Actor-Critic Algorithm
- RL and Supervised Learning
- Learning and Planning

Deep Learning

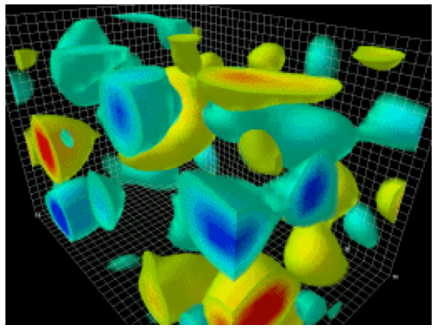
- DL defines a framework to learn the best possible representation of data that can solve tasks like classification, regression, recommender systems, etc.
- DL is a supervised learning technique :
  - Data are all labeled.
- Research and industrial breakthrough applications in CV (autonomous cars, etc), Speech Recognition (speech-to-text and inversely), NLP (machine translation, etc).



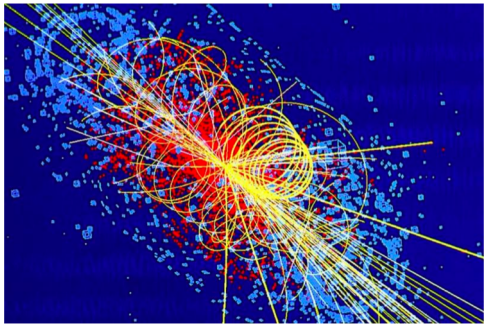


Deep Learning

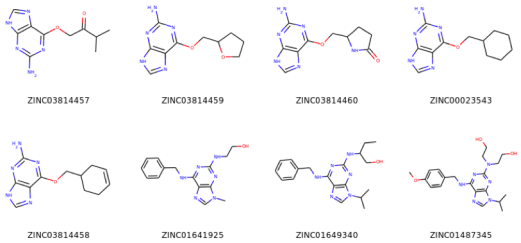
- Potential revolutionary applications in other fields like physics (simulation acceleration, particle detection with complex geometry and 10^8 sensors, generative models for machine calibration) and chemistry (drugs and materials design).
- Learn a deep compositional parametric functions by backpropagation on the loss function.
- Loss functions are continuous and differentiable.



Quark and gluon field fluctuations



Particle jet

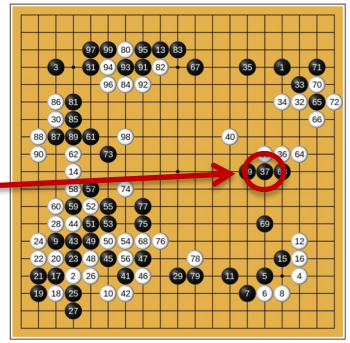


Molecule design

Reinforcement Learning

- RL defines a framework to learn the best possible sequence of decisions in a given environment that can solve tasks like playing Go/Chess, making autonomous robots, recommending ads on internet, etc.

- RL is the science of decision-making (D. Silver) :
  - Goal is to maximize future reward.
  - Decisions may have long-term consequences :
    - AlphaGo vs Lee Sedol with move 37 in game 2.
    - High positive and negative rewards are not instantaneous.
    - Reward may be delayed for higher values (giving up short-term rewards).


First 99 moves

- RL is a causal framework.
  - Time is important – RL is a sequential decision-making technique.

Reinforcement Learning

- RL is different from supervised learning, but has also similarities (later discussed).
- Unlike DL, no real-world breakthrough applications of RL (except super-human game players).
  - Applications to real-world problems are limited, but it might change quickly (DeepMind).
- Applications :
  - Playing games (super-human performances): Go, Chess, Poker, Atari, StarCraft II
  - Exploring virtual worlds : Maze
  - Robots (still preliminary) : Simulation and real autonomous robots
  - Learning to learn (meta-learning): Compute simultaneously architectures and weights.
    - End of humans designing deep learning architectures ?

Reinforcement Learning

- Applications :

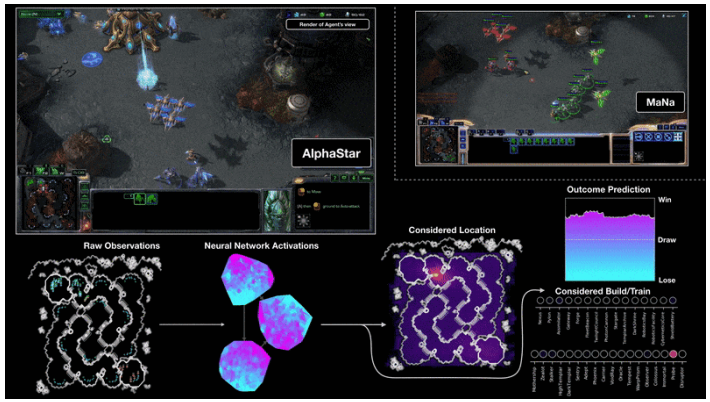
Helicopters that learn to be autonomous
(Stanford)

Robots that learn to grasp objects
(Google)

Reinforcement Learning

- Applications :

Historical AI milestone
Game of Go – March 2016
(DeepMind)

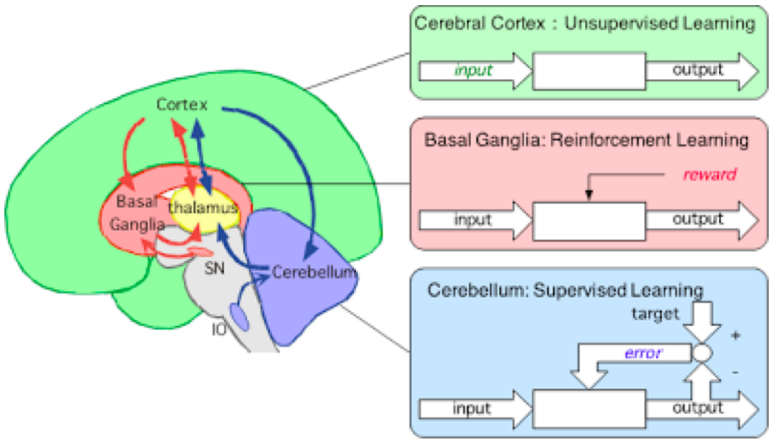


AlphaStar for StarCraft II – Jan 2019
(DeepMind)

AlphaStar beat 5-0 Team Liquid's Grzegorz "MaNa" Komincz, one of the world's strongest professional StarCraft players
https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii

Brain Inspiration

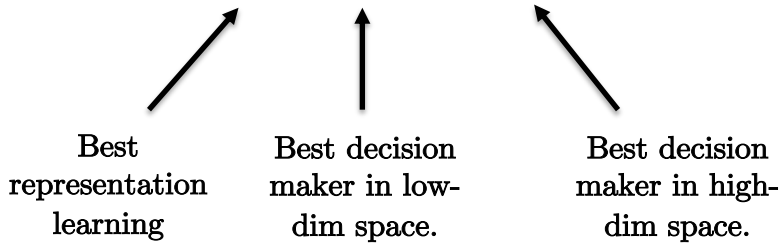
- The human brain may use the cerebellum for supervised learning representation of the world data.
- The human brain may use the cortex for unsupervised learning representation of the world data.
- The human brain may use the basal ganglia for making decisions on problems.

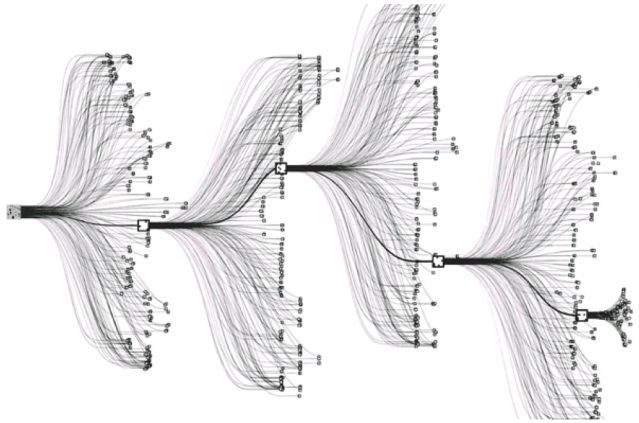


John Sowa

Deep RL

- Reinforcement learning has developed powerful algorithms to solve control problems :
  - Dynamic programming 1953 (Richard Bellman) :
    - Solve a difficult problem by breaking it into simpler sub-problems in a recursive manner.
  - DP can solve optimally low-dimensional control problems by looking at the whole search space.
  - DP becomes intractable in high-dimensional search space.
    - Solution :
      - Combine DL + RL = Deep RL





Each move in Go leads to ~250 possible moves.
The exploration space has 10^{170} paths.

Outline

- RL and Deep Learning
- **Agent, Environment and MDP**
- Policy, Value Function and Model
- Optimal Value Function and Policy
- Deep Q-Learning (DQN)
- Policy Networks
- Actor-Critic Algorithm
- RL and Supervised Learning
- Learning and Planning

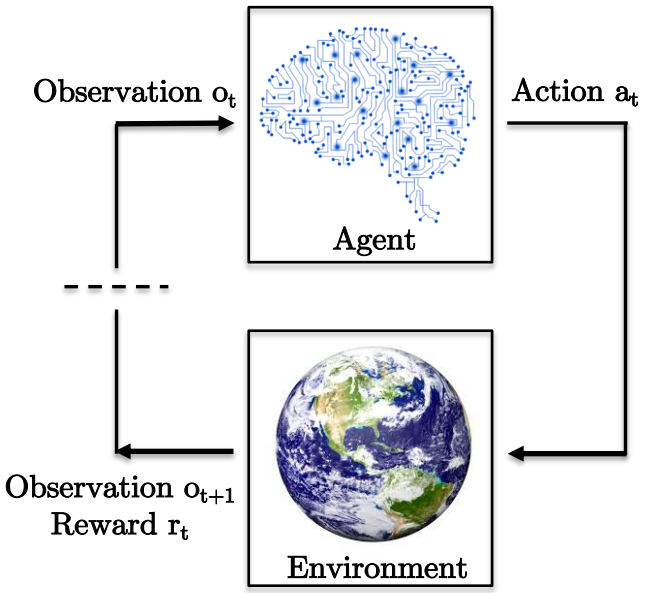
Agent and Environment

- RL framework :
  - Splitting agent and environment :
    - The environment/world can be anything, from simple world like Go board to complex world like Earth.
    - The agent can make actions/decisions.
      - Each action influences/changes the state of the environment (where the agent lives).
      - Each action produces a scalar reward from the environment.

- Goal of RL :
  - Design an agent that acts optimally to optimize reward.

Agent and Environment

- RL framework :
  - At each time step t :
    - The agent:
      - Receives an observation o_t and executes an action a_t.
      - Observes o_{t+1} the result of its action.
      - Receives a scalar reward r_t.
    - The environment :
      - Receives an action a_t.
      - Produces two outputs:
        - Observation o_{t+1}
        - Reward r_t

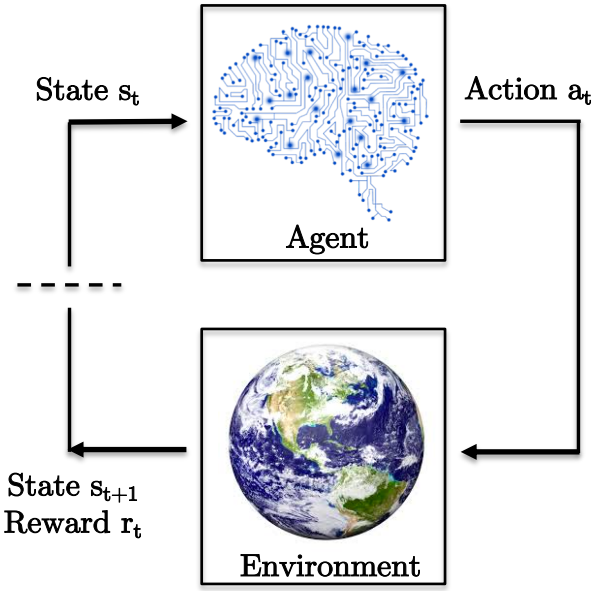


Episode, State and Environment

- An episode/trajectory/experience e_t is a sequence of observations, actions, rewards :
  - $e_t = o_1, a_1, o_2, r_1, a_2, o_3, r_2, \dots, o_t, a_t, o_{t+1}, r_t$

- A state s_t is a summary of the episode/experience :
  - $s_t = f(e_t)$
    $= f(o_1, a_1, o_2, r_1, a_2, o_3, r_2, \dots, o_t, a_t, o_{t+1}, r_t)$

- If the environment is fully observable :
  - Go, Chess, but not Poker
  - $s_t = f(e_t) = f(o_1, a_1, o_2, r_1, a_2, o_3, r_2, \dots, o_t, a_t, o_{t+1}, r_t) = o_t$
    $\Rightarrow s_t = o_t$
  - An episode is written as:
    - $e_t = s_1, a_1, s_2, r_1, a_2, s_3, r_2, \dots, s_t, a_t, s_{t+1}, r_t$



At time step t :

s_t, a_t, s_{t+1}, r_t

MDP

- A fully observable environment is defined by a Markov Decision Process (MDP) :
  - The current state s_t completely characterizes the future, independently of the past.

- A state s_t is Markov if the state transition probability P is defined as :
  - $P(s_{t+1}|s_t) = P(s_{+1}|s_t, s_{t-1},...,s_1)$
  - The current state s_t captures all relevant information from the history $s_{t-1},...,s_1$.
    - The history is not useful and can be discarded.

- A Markov process/chain is a memoryless random process.
  - A sequence of states $s_1,s_2,...,s_t$ with Markov property.

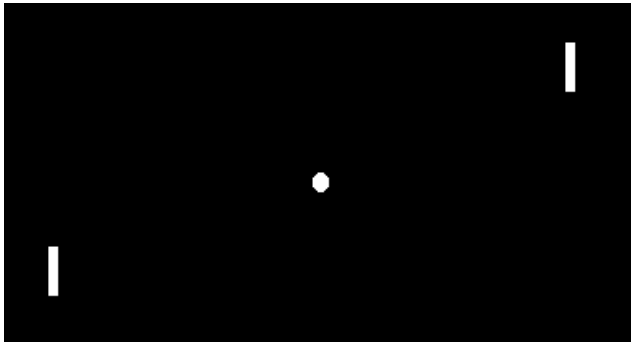
MDP

- A Markov decision process (MDP) :
  - A Markov process with decisions/actions.

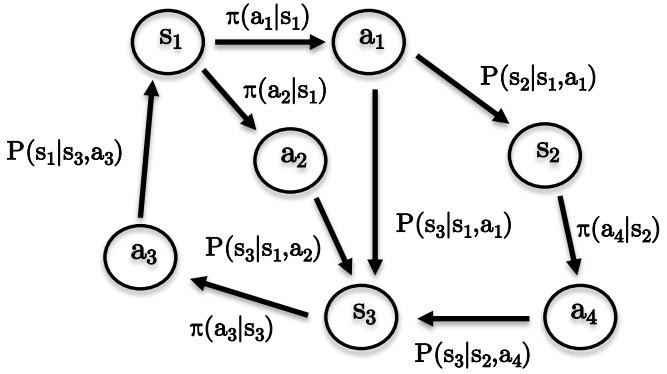
- The state transition probability matrix is defined as :
  - $P(s_{t+1}|s_t,a_t)$
  - It encodes the dynamics/behavior of the environment where the agent lives.

Example

- Pong can be cast as a MDP :
  - A MDP can be visualized as a graph where :
    - Each node is either a game state s_t or an action a_t.
    - Each edge is a probabilistic transition given either by the state transition probability matrix $P(s_{t+1}|s_t,a_t)$, or the policy function $\pi(a_t|s_t)$ (later discussed).



PONG : A game state s_t is the image of the game at time state t. An action a_t is a move, either Up or Down.



MDP of PONG

Outline

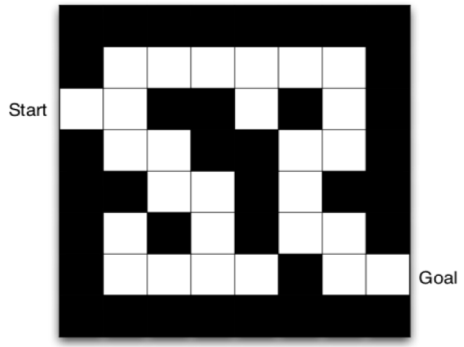
- RL and Deep Learning
- Agent, Environment and MDP
- **Policy, Value Function and Model**
- Optimal Value Function and Policy
- Deep Q-Learning (DQN)
- Policy Networks
- Actor-Critic Algorithm
- RL and Supervised Learning
- Learning and Planning

Properties of RL

- The goal of RL is to act optimally to maximize a value function that represents the total reward of a sequence of actions.

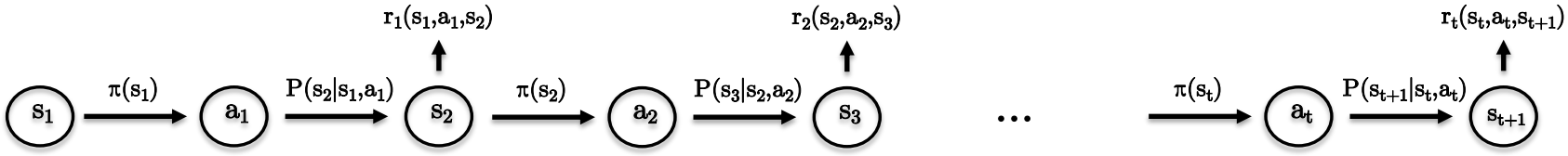
- There are three types of RL problems, based on three fundamental properties of RL :
  - Policy function :
    - Agent's behavior function (select the agent's action).
  - Value function:
    - Function that quantifies how good or bad is each state or action in state.
  - Model function :
    - Agent's representation of the environment (for planning/reasoning).

- We will use a simple example to illustrate these properties:
  - Maze



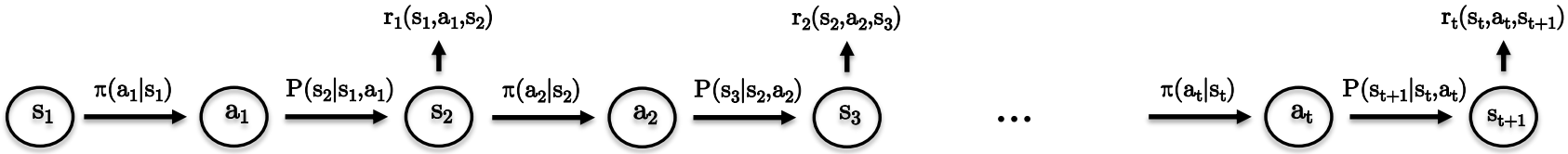
Policy

- A policy encodes the agent's behavior :
  - It is a map from states to actions.
    - Deterministic policy :
      - $a = \pi(s)$
      - Given deterministic π, a new episode can be drawn as follows :
        - $e_t = s_1,\ a_1 = \pi(s_1),\ s_2 \sim P(s|s_1,a_1),\ r_1,\ a_2 = \pi(s_2),\ s_3 \sim P(s|s_2,a_2),\ r_2,$ $\ldots\ s_t \sim P(s|s_{t-1},a_{t-1}),\ a_t = \pi(s_t),\ s_{t+1} \sim P(s|s_t,a_t),\ r_t$



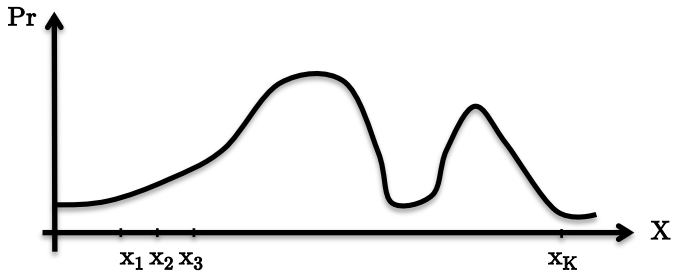
Policy

- Stochastic policy :
  - $\pi(a|s)$ = Probability of action a in state s
  - Given stochastic π, a new episode can be drawn as follows :
    - $e_t = s_1,\ a_1 \sim \pi(a|s_1),\ s_2 \sim P(s|s_1,a_1),\ r_1,\ a_2 \sim \pi(a|s_2),\ s_3 \sim P(s|s_2,a_2),\ r_2,$
      $\ldots\ s_t \sim P(s|s_{t-1},a_{t-1}),\ a_t \sim \pi(a|s_t),\ s_{t+1} \sim P(s|s_t,a_t),\ r_t$
      - From a given state s_t, there are multiple possible actions, unlike for the deterministic policy.

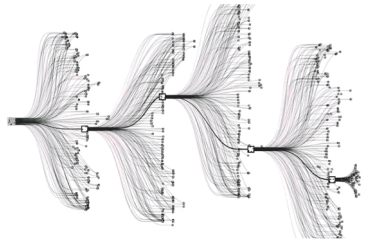


Policy

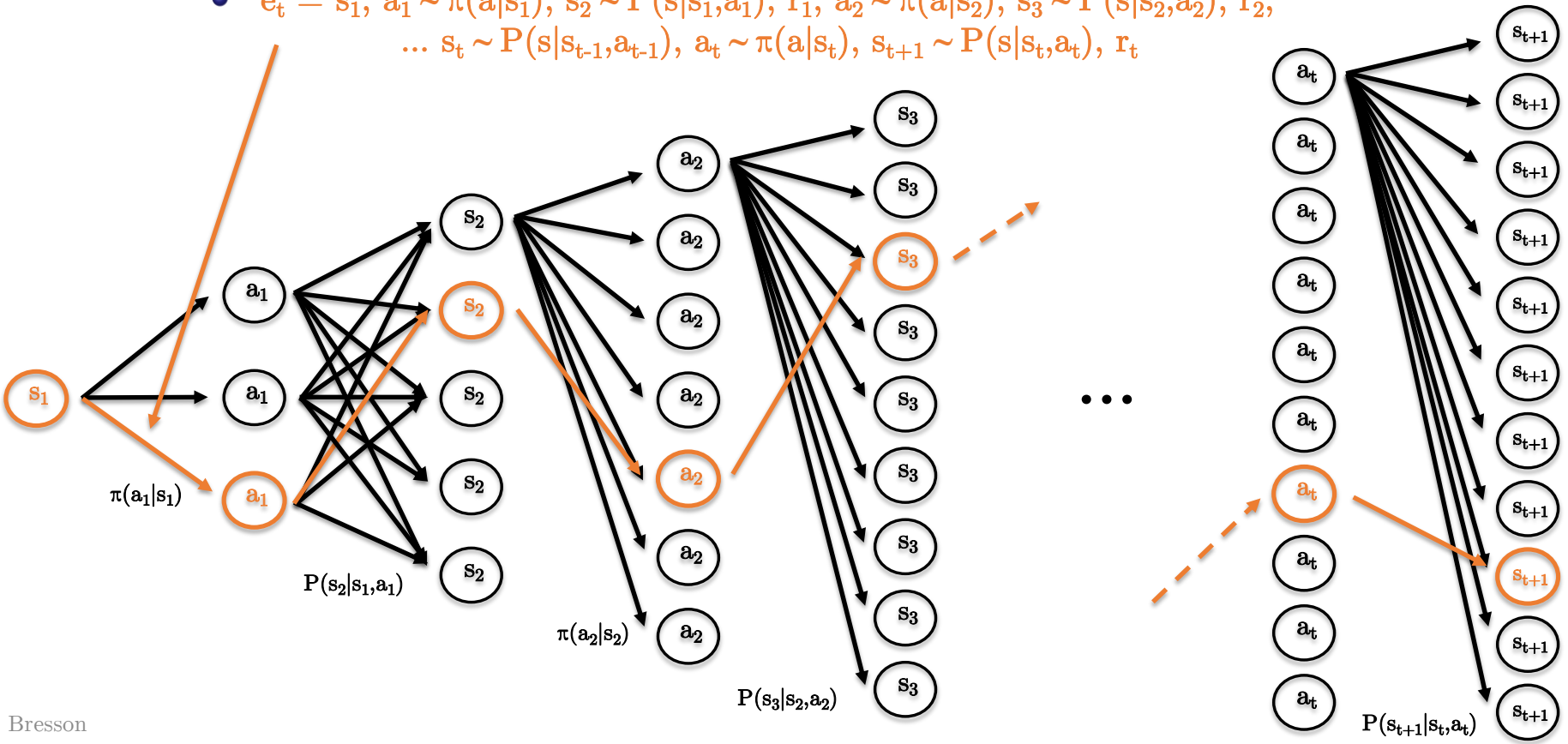
- How to sample $a_t \sim \pi(a|s_t)$ and $s_t \sim P(s|s_{t-1},a_{t-1})$?
  - Bernoulli sampling :
    - A random draw with exactly K possible outcomes, in which the probability Pr of having $X=x_k$ depends on a stationary distribution (i.e. the distribution stays unchanged at each time step).
    - NumPy/PyTorch :
      - s = np.random.choice(p=Pr(X))
      - s = torch.distributions.Categorical(p=Pr(X)).sample()



Policy

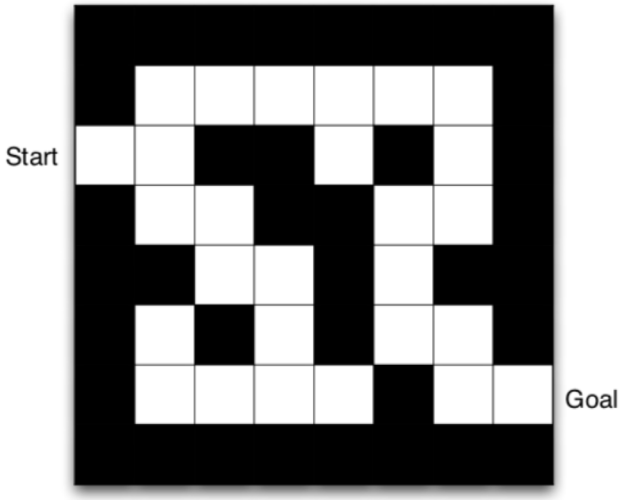


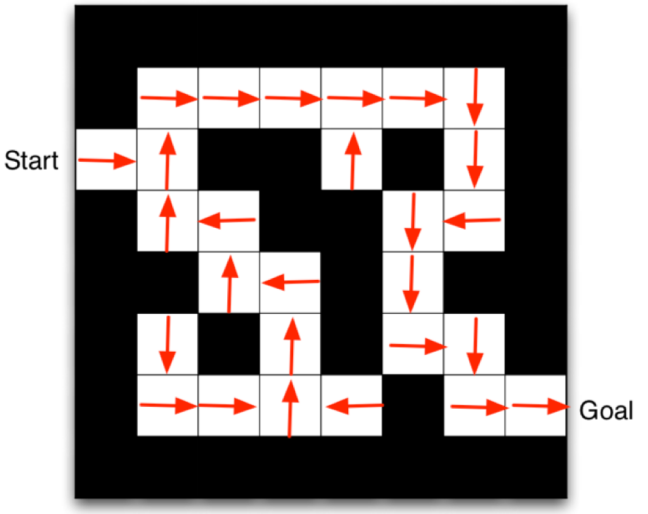
- Given probability functions $\pi(a|s)$ and $P(a|s)$,
  - A new episode e_t can be drawn by rolling out/sampling π and P :
    - $e_t = s_1,\ a_1 \sim \pi(a|s_1),\ s_2 \sim P(s|s_1,a_1),\ r_1,\ a_2 \sim \pi(a|s_2),\ s_3 \sim P(s|s_2,a_2),\ r_2,$
      $\dots\ s_t \sim P(s|s_{t-1},a_{t-1}),\ a_t \sim \pi(a|s_t),\ s_{t+1} \sim P(s|s_t,a_t),\ r_t$



Policy

- Maze example :
  - Actions are : Up, Down, Left, Right
  - States are : Agent's locations





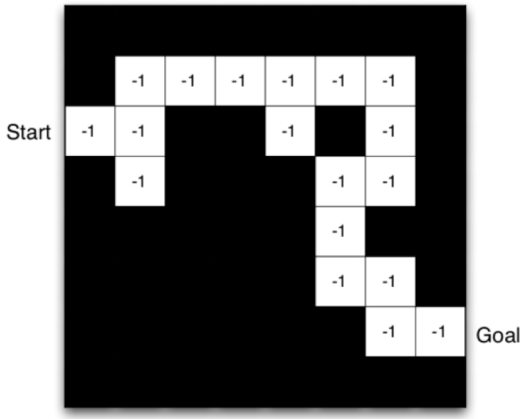
Arrows represent optimal policy $\pi(s)$
for each state s

Model

- Maze example :
  - Actions are : Up, Down, Left, Right
  - States are : Agent's locations
  - Model is :
    - Dynamics of the environment modeled by $P(s|s_t,a_t)$
    - Immediate reward $r_t(s_t,a_t,s_{t+1})$ given by the environment

Dynamics : Grid layout represents the transition model P
(same probability value for all state-action)

Reward : Number -1 represents immediate reward r from each state s
(same reward value for all actions a)



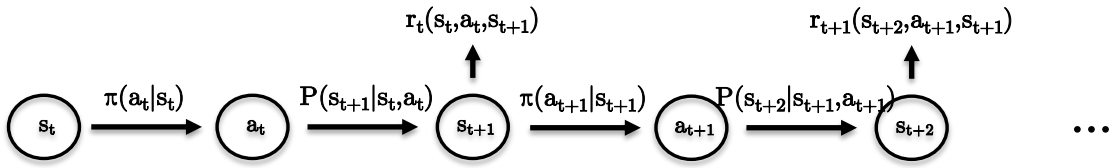
State-Value Function

- The state value function $Q(s)$ is the expected future reward for being in state s.
  - It is used to evaluate the goodness/badness of each state s.
    - It acts as the label information in supervised learning (later discussed).

- Let us compute the value function Q in state s_t considering only one episode :
  - $e_t = s_t, a_t \sim \pi(a|s_t), s_{t+1} \sim P(s|s_t,a_t), r_t, a_{t+1} \sim \pi(a|s_{t+1}), s_{t+2} \sim P(s|s_{t+1},a_{t+1}), r_{t+1}, \ldots$
  - $Q(s_t) = r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + \ldots = \Sigma_{k=0}^{\infty} \gamma^k.r_{t+k}$

Return for being in state s_t | Immediate reward | Sum of discounted future rewards

  - The value function $Q(s_t)$ is the total discounted reward (a.k.a. return) we will receive for being in state s_t.

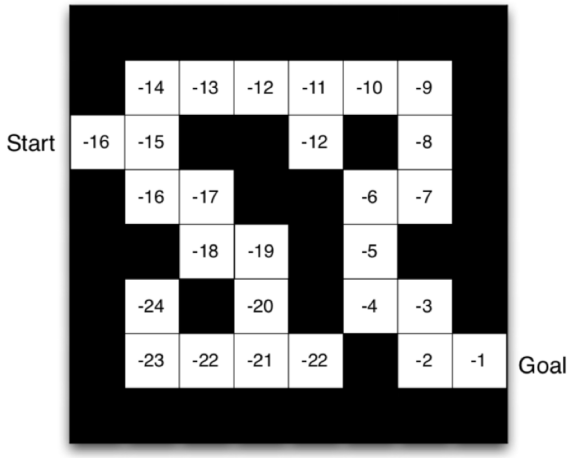


State-Value Function

- Discounted reward using hyper-parameter γ :
  - $Q(s_t) = r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + ... = \Sigma_{k=0}^{\infty} \gamma^k.r_{t+k}$
  - Exponential decay factor γ^k :
    - $0 < \gamma < 1$
    - γ=0.9 $\Rightarrow \gamma^k \approx 0$ for $k \geq 50$: short-term consequences
    - γ=0.99 $\Rightarrow \gamma^k \approx 0$ for $k \geq 500$: long-term consequences (harder to learn)
  - Justifications of γ :
    - Math :
      - Avoid infinite total reward with value γ<1.
    - Model uncertainty about the policy network :
      - The action taken is not absolutely certain to be optimal, so it is better to discount its future reward.
    - Short-term return is usually preferred than long-term (e.g. Finance).

State-Value Function

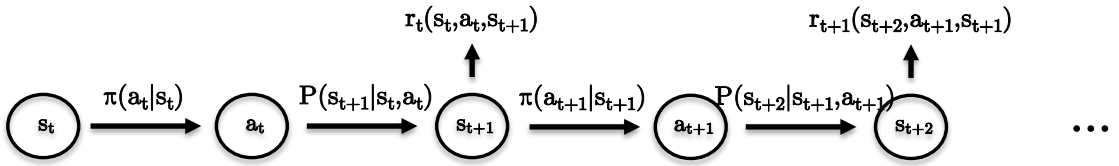
- Maze example :
  - Actions are : Up, Down, Left, Right
  - States are : Agent's locations
  - Model is :
    - Dynamics of the environment modeled by $P(s|s_t,a_t)$
    - Immediate reward $r_t(s_t,a_t,s_{t+1}) = -1$ given by the environment



Numbers represent optimal value function $Q(s)$ (w/ γ=1) for each state s.

State-Value Function

- Let us compute the value function Q in state s_t considering all possible episodes drawn from π and P :
  - $e_t = s_t, a_t \sim \pi(a|s_t), s_{t+1} \sim P(s|s_t,a_t), r_t, a_{t+1} \sim \pi(a|s_{t+1}), s_{t+2} \sim P(s|s_{t+1},a_{t+1}), r_{t+1}, \ldots$
  - $Q(s_t) = \mathbb{E}_{\pi,P} (r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + \ldots)$
    - $Q(s_t)$ is the expected total discounted reward for being in state s_t.



State-Value Function

- Let us write the Bellman equation of the value function Q :
  - $Q(s_t) = \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + \ldots\ )$
    $= \mathbb{E}_{\pi,P}\ (\ r_t + \mathbb{E}_{\pi,P}\ (\ \gamma.r_{t+1} + \gamma^2.r_{t+2} + \ldots\ )\ )$
    $= \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.\ \mathbb{E}_{\pi,P}\ (\ r_{t+1} + \gamma.r_{t+2} + \ldots\ )\ )$
    $= \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.\ Q(s_{t+1})\ )$
  - Recursive formulation (dynamic programming)
  - The sum of total reward Q for being in state s_t can be decomposed into 2 parts :
    - The immediate reward r_t we get after one time step.
    - The discounted sum of future rewards $\gamma.Q(s_{t+1})$ we will receive.
      - It is the expectation over all possible states s_{t+1}.

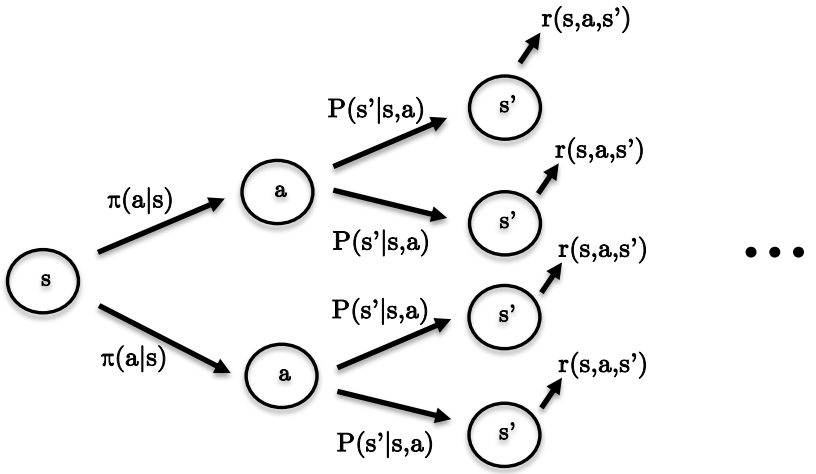
State-Value Function

- The Bellman equation of the value function is stationary, i.e. independent of the time step t :
  - Let s_t, a_t, s_{t+1}, r_t be re-written as
    - s, a, s', r
  - Let $Q(s_t) = \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.\ Q(s_{t+1})\ )$ be re-written as
    - $Q(s) = \mathbb{E}_{\pi,P}\ (\ r(s,a,s') + \gamma.\ Q(s')\ )$

      $= \Sigma_a\ \underbrace{\pi(a|s)}\ \Sigma_{s'}\ \underbrace{P(s'|a,s)}\ (\ r(s,a,s') + \gamma.\ Q(s')\ )$

Probability of action a in state s

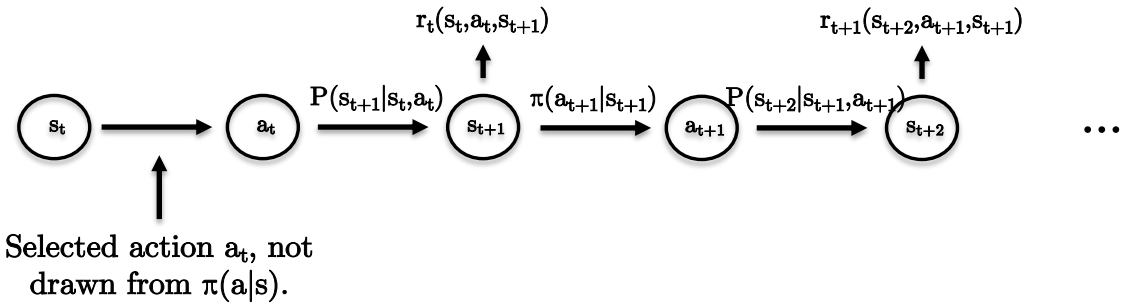
Transition probability from action a in state s to state s'



Action-Value Function

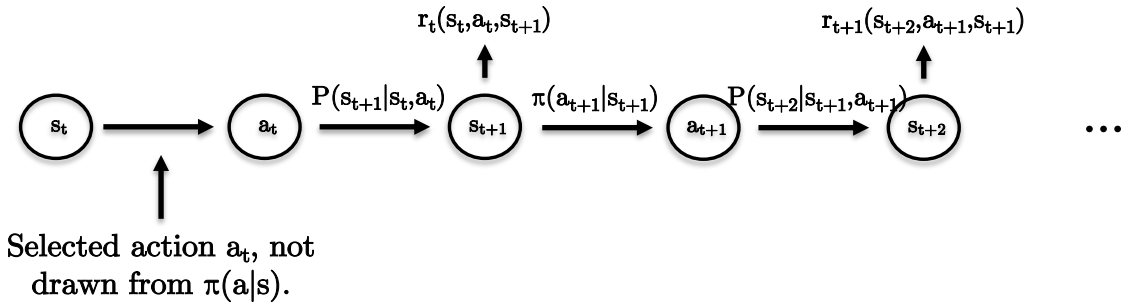
- The state-value function $Q(s_t)$ can be extended to an action-value function $Q(a_t|s_t)$:
  - $Q(a_t|s_t)$ = Future reward (good or bad) of selecting action a_t in state s_t.

- Let us compute the value function Q of taking action a_t in state s_t considering only one episode :
  - $e_t = s_t$, a_t selected, $s_{t+1} \sim P(s|s_t,a_t)$, r_t, $a_{t+1} \sim \pi(a|s_{t+1})$, $s_{t+2} \sim P(s|s_{t+1},a_{t+1})$, r_{t+1}, ...
  - $Q(a_t|s_t) = r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + ... = \Sigma_{k=0}^{\infty} \gamma^k.r_{t+k}$
    - Same equation as state-value function $Q(s_t)$ except the action a_t is not a random variable but an arbitrary selection.



Action-Value Function

- Let us compute the value function Q of taking action a_t in state s_t considering all possible episodes drawn from π and P :
  - $e_t = s_t$, a_t selected, $s_{t+1} \sim P(s|s_t,a_t)$, r_t, $a_{t+1} \sim \pi(a|s_{t+1})$, $s_{t+2} \sim P(s|s_{t+1},a_{t+1})$, r_{t+1}, ...
  - $Q(a_t|s_t) = \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + \dots\ )$

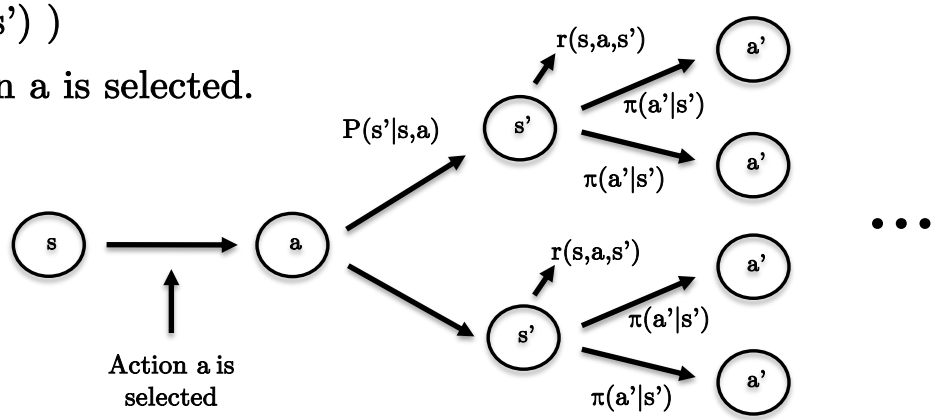


Action-Value Function

- Let us write the Bellman equation of the action-value function Q :
  - $Q(a_t|s_t) = \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + \ldots\ )$
    $= \mathbb{E}_{\pi,P}\ (\ r_t + \mathbb{E}_{\pi,P}\ (\ \gamma.r_{t+1} + \gamma^2.r_{t+2} + \ldots\ )\ )$
    $= \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.\ \mathbb{E}_{\pi,P}\ (\ r_{t+1} + \gamma.r_{t+2} + \ldots\ )\ )$
    $= \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.\ Q(a_{t+1}|s_{t+1})\ )$
  - The sum of total reward Q for taking action a_t in state s_t can be decomposed into 2 parts :
    - The immediate reward r_t we get from action a_t in state s_t.
    - The discounted expected sum of future rewards $\gamma.Q(a_{t+1}|s_{t+1})$ we will receive.
      - It is the expectation over all possible actions a_{t+1} in all possible states s_{t+1}.

Action-Value Function

- The Bellman equation of the action-value function is stationary, i.e. independent of the time step t :
  - Let s_t, a_t, s_{t+1}, r_t , a_{t+1} be re-written as
    - s, a, s', r, a'
  - Let $Q(a_t|s_t) = \mathbb{E}_{\pi,P}\ (\ r_t + \gamma.\ Q(a_{t+1}|s_{t+1})\ )$ be re-written as
    - $Q(a|s) = \mathbb{E}_{\pi,P}\ (\ r(s,a,s') + \gamma.\ Q(a'|s')\ )$
      $= \Sigma_{a',s'}\ P(s'|a,s)\ (\ r(s,a,s') + \gamma.\ Q(a'|s')\ )$
      $= \Sigma_{s'}\ P(s'|a,s)\ (\ r(s,a,s') + \gamma.\ \Sigma_{a'}\ Q(a'|s')\ )$
      $= \mathbb{E}_{P}\ (\ r(s,a,s') + \gamma.\ \Sigma_{a'}\ Q(a'|s')\ )$
    - There is no more $\pi(a|s)$ as action a is selected.



Outline

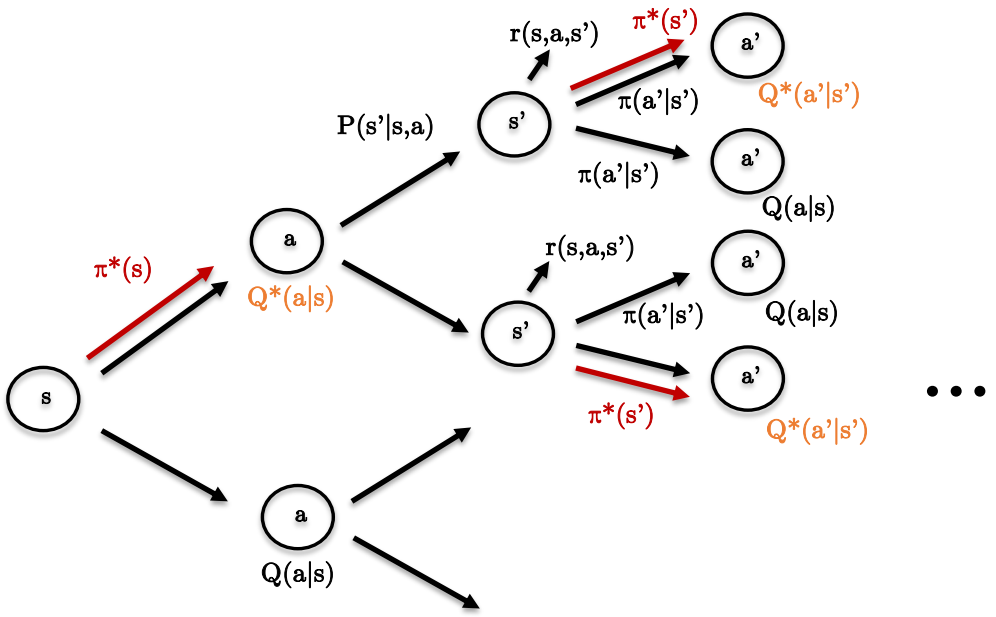
- RL and Deep Learning
- Agent, Environment and MDP
- Policy, Value Function and Model
- **Optimal Value Function and Policy**
- Deep Q-Learning (DQN)
- Policy Networks
- Actor-Critic Algorithm
- RL and Supervised Learning
- Learning and Planning

Optimal Value Function and Policy

- Goals of RL :
  - Find the optimal value function $Q^*(a|s)$:
    - We search for the maximum reward given by action a in state s.
  - Find the optimal policy $\pi^*(a|s)$:
    - We search for the policy that provides the best action a in state s.
    - The optimal policy is deterministic, $\pi^*(a|s) := \pi^*(s)$.
  - Both objectives are equivalent :
    - $Q^*(a|s) = \max_\pi Q(a|s) = \max_a Q(a|s) = Q^{\pi^*}(a|s)$
    - Once the optimal state-value function Q^* is found, then the problem is solved as it is possible to act optimally :
      - $\pi^*(s) = \text{argmax}_a Q^*(a|s)$

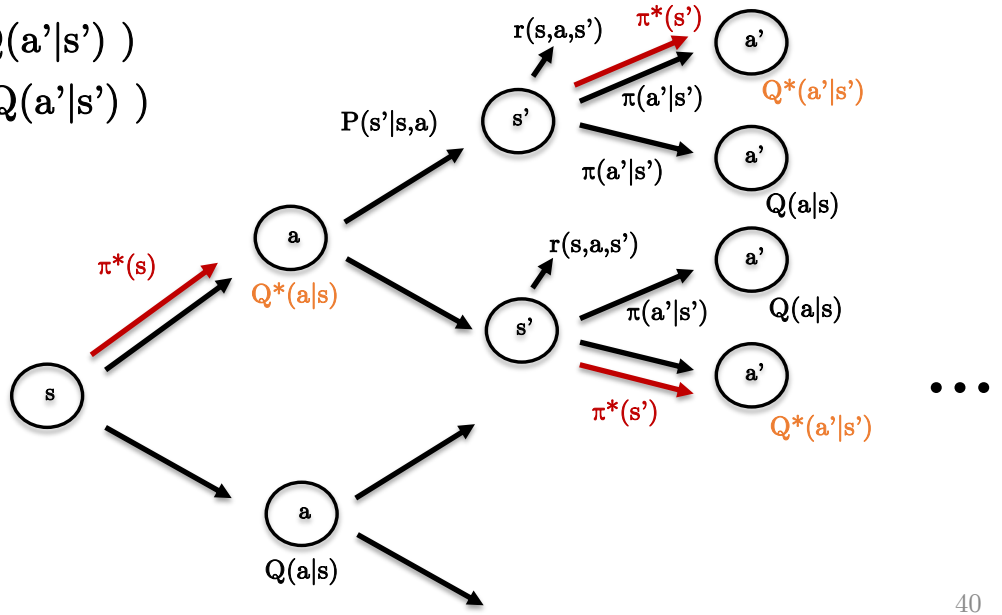
Optimal Value Function and Policy

- Optimal value function $Q^*(a|s)$ and optimal policy $\pi^*(s)$:
  - $Q^*(a|s) = \max_\pi Q(a|s) = \max_a Q(a|s) = Q^{\pi^*}(a|s)$
  - $\pi^*(s) = \pi^*(a|s) = \text{argmax}_a Q^*(a|s)$



Optimal Value Function and Policy

- The Bellman equation of the optimal action-value function (the maximum Q value for taking action a in state s) :
  - $Q^*(a|s) = \max_\pi Q(a|s)$
    $= \max_\pi \mathbb{E}_{\pi,P}\ (\ r(s,a,s') + \gamma.\ Q(a'|s')\ )$
    $= \max_\pi \mathbb{E}_{P}\ (\ r(s,a,s') + \gamma.\ Q(a'|s')\ )$
    $= \max_\pi \Sigma_{s'}\ P(s'|a,s)\ (\ r(s,a,s') + \gamma.\ Q(a'|s')\ )$
    $= \Sigma_{s'}\ P(s'|a,s)\ (\ r(s,a,s') + \gamma.\ \max_\pi Q(a'|s')\ )$
    $= \Sigma_{s'}\ P(s'|a,s)\ (\ r(s,a,s') + \gamma.\ \max_{a'} Q(a'|s')\ )$
    $= \mathbb{E}_{s'}\ (\ r + \gamma.\ Q^*(a'|s')\ )$



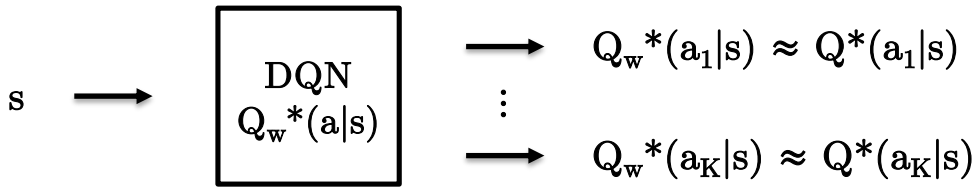
Outline

- RL and Deep Learning
- Agent, Environment and MDP
- Policy, Value Function and Model
- Optimal Value Function and Policy
- **Deep Q-Learning (DQN)**
- Policy Networks
- Actor-Critic Algorithm
- RL and Supervised Learning
- Learning and Planning

Deep Q-Learning

- The goal is to estimate the optimal action value function Q^* defined as :
  - $Q^*(a|s) = \mathbb{E}_{s'}[\, r + \gamma . \max_{a'} Q(a'|s') \,]$
- It is intractable to compute Q^* for all actions and all states.
- Q^* will be estimated with a neural network (DQN) :
  - $Q^*_w(a|s) \approx Q^*(a|s) = \underbrace{\mathbb{E}_{s'}[\, r + \gamma . \max_{a'} Q(a'|s') \,]}_{\text{Target to regress}}$

(supervised learning)



Deep Q-Learning

- We need to estimate :

$$Q^*(a|s) = \mathbb{E}_{s'}\left(r + \gamma . \max_{a'} Q(a'|s') \right)$$

$$= \Sigma_{s'} P(s'|a,s) \left(r(s,a,s') + \gamma . \max_{a'} Q(a'|s') \right)$$

- We approximate the analytical optimal value Q^* :

$$Q^*(a|s) = \mathbb{E}_{s'}\left(r + \gamma . \max_{a'} Q(a'|s') \right)$$

$$\approx 1/|s'| \; \Sigma_{s'} \left(r(s,a,s') + \gamma . \max_{a'} Q(a'|s') \right)$$

$$\approx r(s,a,s') + \gamma . \max_{a'} Q(a'|s') \quad \text{(if only one state s' is collected)}$$

$$\approx r(s,a,s') + \gamma . \max_{a'} Q^*(a'|s') \quad \text{(we have } \max_{a'} Q = Q^* = \max_{a'} Q^*\text{)}$$

Deep Q-Learning

- We approximate with DQN the expression :
  - $Q^*(a|s) \approx r(s,a,s') + \gamma . \max_{a'} Q^*(a'|s') \Rightarrow Q^*_w(a|s) \approx r(s,a,s') + \gamma . \max_{a'} Q^*_w(a'|s')$
- Training data (s,a,s',r) will be collected by rolling out episodes.
- DQN MSE loss obtained after rolling out a few of episodes:
  - $L(w) = 1/N \ \Sigma_{(s,a,s',r)} \ || \ r(s,a,s') + \gamma . \max_{a'} Q^*_w(a'|s') - Q^*_w(a|s) \ ||_2^2$
    - where $N=|(s,a,s',r)|$ is the total number of collected tuples (s,a,s',r).

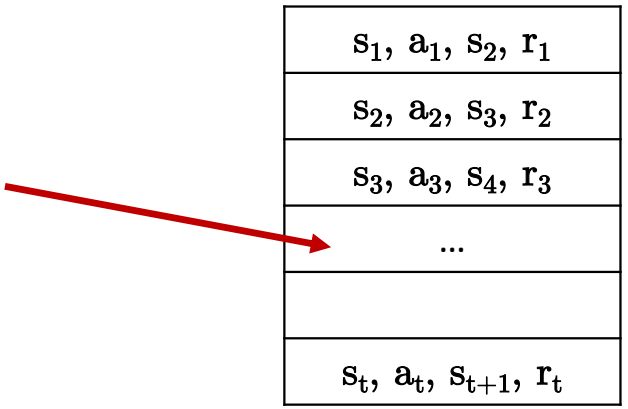
Vanilla DQN Algorithm

- Vanilla DQN algorithm :
  - Initialize w randomly
  - Repeat :
    - Roll-out an episode : s_t, $a_t \sim \pi_w$, r_t, s_{t+1}
    - Update state-value function Q^*_w by gradient descent :
      $w^{k+1} = w^k - lr \, . \, \nabla_w \, (\, 1/N \, \Sigma_t \, || \, r(s_t,a_t,s_{t+1}) + \gamma . \, \max_{a\_t+1} \, Q^*_w(a_{t+1}|s_{t+1}) \, - \, Q^*_w(a_t|s_t) \, ||_2^2 \,)$

- DQN does not converge !
  - Strong correlations between samples.
    - Predictive function over-fits (memorization) rather than generalizes.
  - Target values are non-stationary.
    - The target values change too quickly.

Vanilla DQN Algorithm

- To remove correlations between training data :
  - Memory/Experience Replay :
    - Data are randomly sampled from a lookup table.



- To stabilize the learning process w.r.t. the signal non-stationarity :
  - A baseline network w_{BL} is introduced.
    - This network keeps a copy of the network w for a fixed number of iterations.

DQN Algorithm

- DQN algorithm :
  - Initialize w randomly and $w_{BL} = w$
  - Repeat :
    - Roll-out an episode : s_t, $a_t \sim \pi_w$, r_t, s_{t+1} and store data in replay memory.
    - Update state-value function Q^*_w by mini-batch gradient descent (batch of randomly sampled data from the lookup table) :

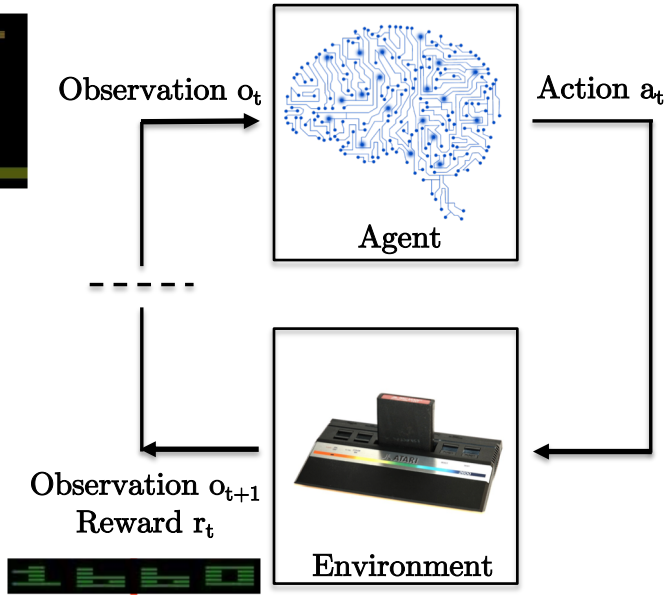
$$w^{k+1} = w^k - lr \cdot \nabla_w \left(1/N \sum_{(s,a,s',r)} || r(s,a,s') + \gamma . \max_{a'} Q^*_{w\_BL}(a'|s') - Q^*_w(a|s) ||_2^2 \right)$$

    - Update baseline network $w_{BL} = w$ if greedy evaluation of network w is better than w_{BL}.

DQN Applications

- DQN application to Atari games :

Observation is a stack of 4 consecutive image frames of the game.

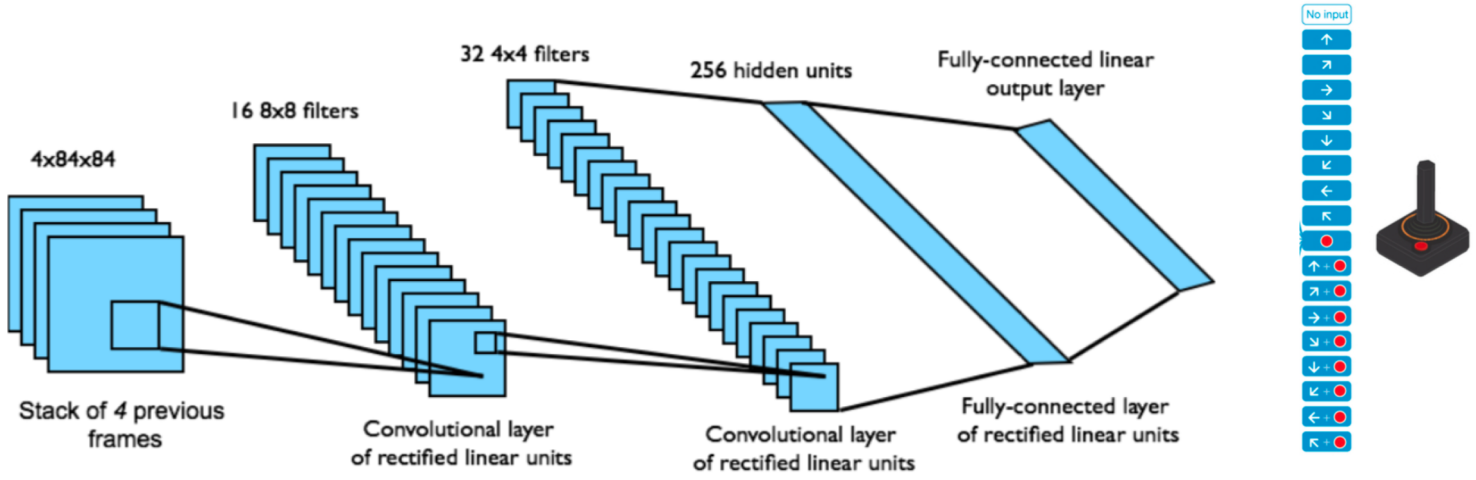


Action is the joystick position and a button.

Reward is a score for this action and state.

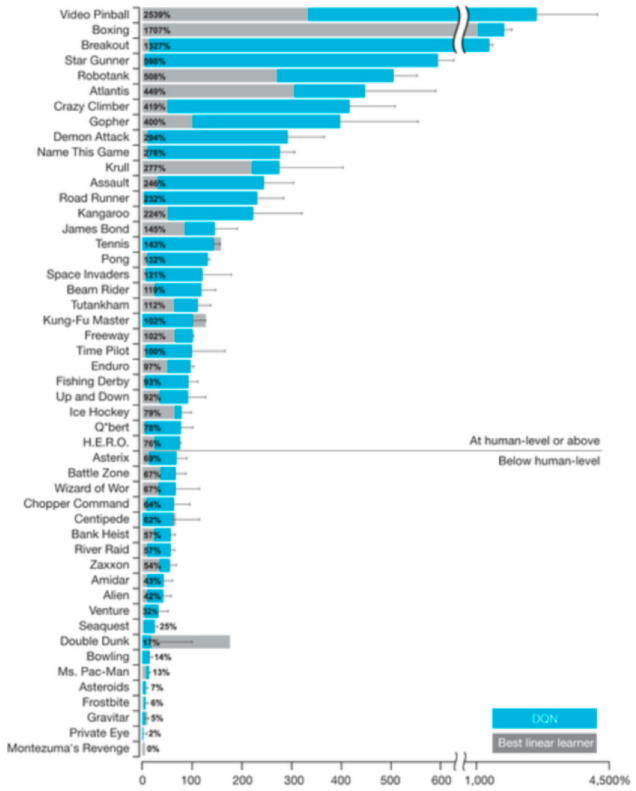
DQN Applications

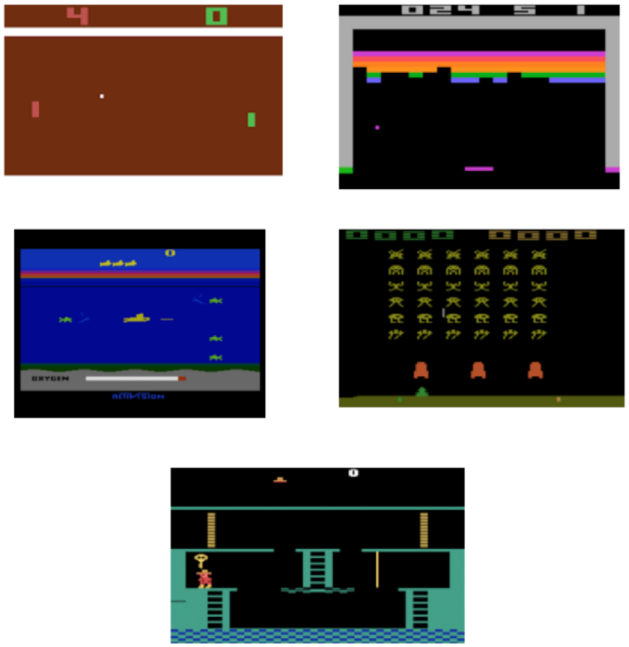
- DQN application to Atari games :
  - Learning of values $Q_w(a|s)$ from input s.
  - Input state s is a stack of raw pixels from last 4 frames.
  - Output is $Q_w(a|s)$ for the 18 joystick/button positions.



DQN Applications

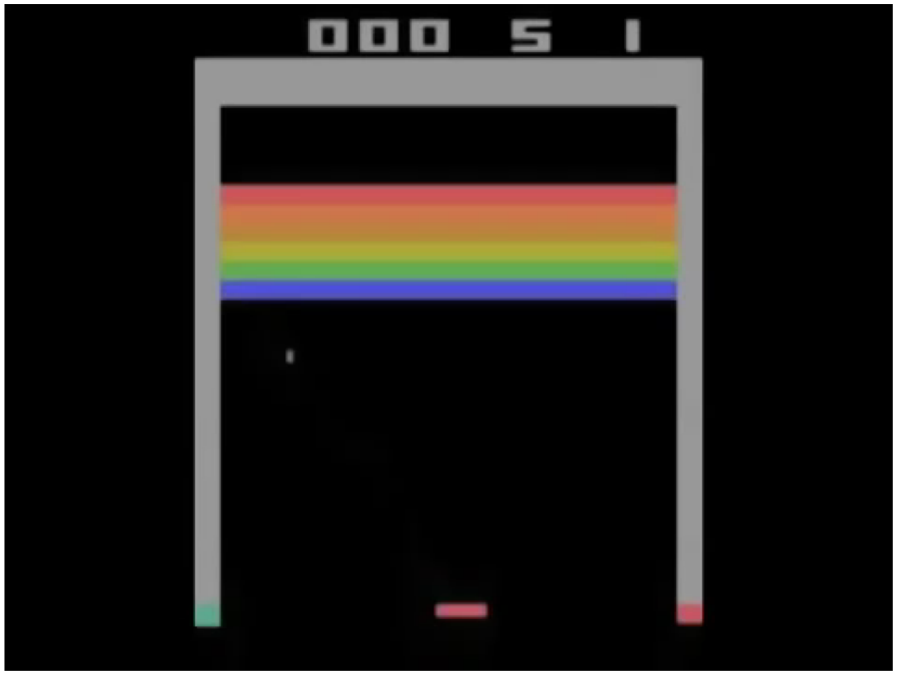
- Application to Atari games :





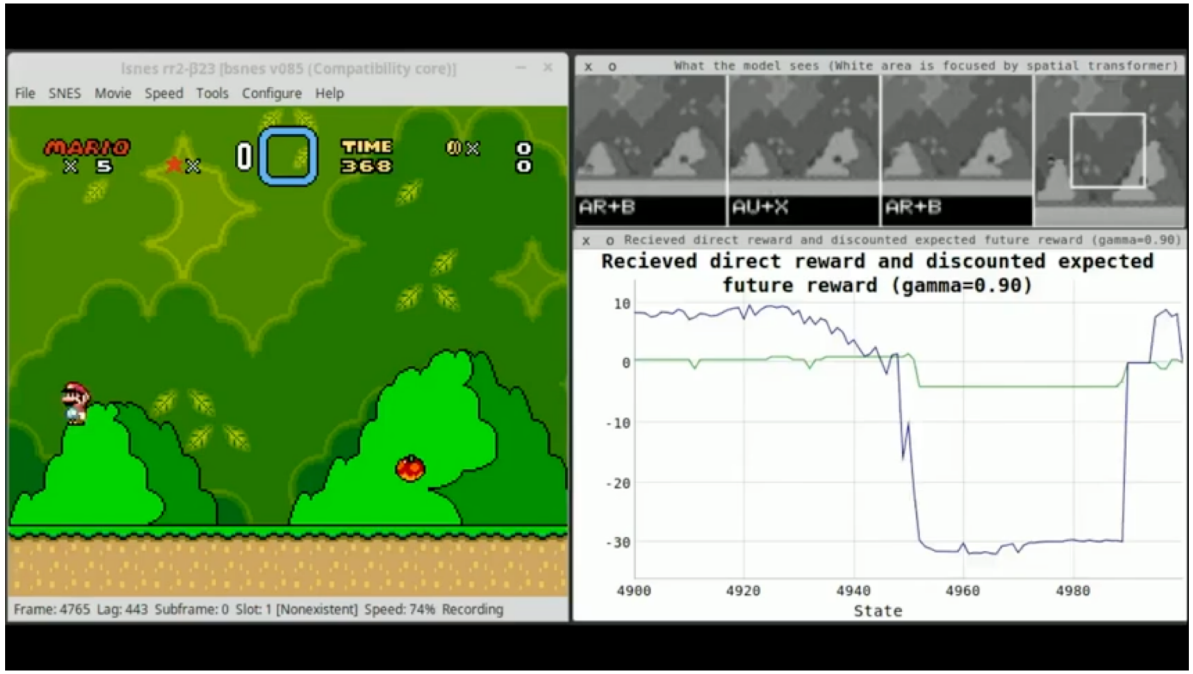
DQN Applications

- Learning optimal strategy :



DQN Applications

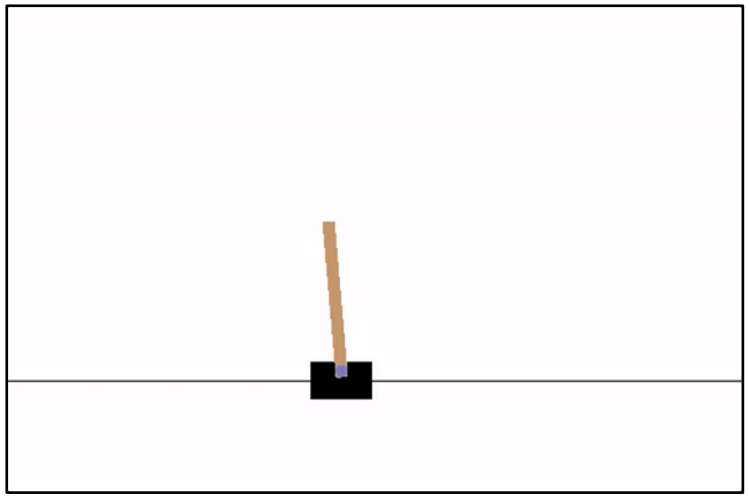
- Application to Mario Bros :



https://github.com/aleju/mario-ai

Demo 01

- Finding optimal strategy to stabilize a pole attached by an un-actuated joint to a cart, which moves along a frictionless track.
  - The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart's velocity.
  - The cart-pole environment is the "MNIST" experiment for RL.
    - Use it to develop new RL architectures !



Optimal strategy

Demo 01

Replay Memory

```
Transition = namedtuple( 'Transition', ('state', 'action', 'next_state', 'reward', 'done') )

# class of replay memory/experience
class ReplayMemory(object):

    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def write(self, *args): # store transitions (s, a, s', r)
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def read(self, batch_size): # select a random batch of transitions
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

Demo 01

Q network

Compute Q scores given a state

Select action given a state

Select random action to explore the space of (state,action)

Loss

Read (s,a,s',r) in memory

Standard MSE loss

```python
# class of policy network
class Q_NN(nn.Module):

    def __init__(self, net_parameters):
        super(Q_NN, self).__init__()
        input_dim = net_parameters['input_dim']
        hidden_dim = net_parameters['hidden_dim']
        output_dim = net_parameters['output_dim']
        self.fc1 = nn.Linear(input_dim, hidden_dim, bias=True)
        self.fc2 = nn.Linear(hidden_dim, output_dim, bias=True)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        Q_scores = self.fc2(x) # scores over actions
        return Q_scores

    def select_action(self, state, rand_act_pr): # select action w/ Q network
        Q_scores = self.forward(state) # Q(a|s) scores of action a in state s
        coin = random.random()
        if coin < rand_act_pr: # (state,action) exploration process parametrized by rand_act_pr
            action = torch.randint(0,2,()).item()
        else:
            action = Q_scores.argmax().item()
        return action

    def loss(self, memory, baseline_Q_net, opt_parameters):
        batch_size = opt_parameters['batch_size']
        gamma = opt_parameters['gamma']
        if memory.__len__()>=batch_size: # read a batch of transitions (s,a,s',r) in replay memory
            batch_transitions = Transition(*zip(*memory.read(batch_size)))
        else:
            batch_transitions = Transition(*zip(*memory.read(memory.__len__())))
        batch_states = torch.stack([x for x in batch_transitions.state]).float() # state=s, size=B x 4
        batch_next_states = torch.stack([x for x in batch_transitions.next_state]).float() # next_state=s', size=B x 4
        batch_rewards = torch.stack([x for x in batch_transitions.reward]).float() # reward=r, size=B
        batch_actions = torch.stack([x for x in batch_transitions.action]).long() # action=a, size=B
        batch_dones = torch.stack([x for x in batch_transitions.done]).float() # done, size=B
        Q = self.forward(batch_states).gather(dim=1,index=batch_actions.unsqueeze(1)) # Q_W(a|s), size=B x 1
        max_baseline_Q_net = baseline_Q_net.forward(batch_next_states).max(dim=1)[0].detach() * batch_dones
        Q_target = batch_rewards.unsqueeze(1) + \
            gamma * max_baseline_Q_net.unsqueeze(1) # Q_target = r + gamma . max_a' Q_W^BL(a'|s'), size=B x 1
        loss = nn.MSELoss()(Q,Q_target) # MSE_Loss(error = Q_target - Q_W)
        return loss
```

Demo 01

Rollout class

Rollout one episode

Write (s,a,s',r) in memory

```python
# class of rollout episodes
class Rollout_Episodes():

    def __init__(self):
        super(Rollout_Episodes, self).__init__()

    def rollout_batch_episodes(self, env, memory, opt_parameters, Q_Net, write_memory=True):
        nb_episodes_per_batch = opt_parameters['nb_episodes_per_batch']
        env_seeds = opt_parameters['env_seed']
        rand_act_pr = opt_parameters['rand_act_pr']
        batch_episode_lengths = []
        for episode in range(nb_episodes_per_batch):
            env.seed(env_seeds[episode].item()) # start with random seed
            state = env.reset() # initial state
            for t in range(1000): # rollout one episode until it finishes
                state_pytorch = torch.from_numpy(state).float().unsqueeze(0) # state=s
                action = Q_Net.select_action(state_pytorch, rand_act_pr) # select action=a from state=s
                next_state, reward, done, _ = env.step(action) # receive next_state=s' and reward=r
                done_mask = 0.0 if done else 1.0
                if write_memory:
                    memory.write(torch.tensor(state),torch.tensor(action),torch.tensor(next_state),
                                 torch.tensor(reward),torch.tensor(done_mask))
                state = next_state
                if done:
                    batch_episode_lengths.append(t)
                    break
        return batch_episode_lengths
```

Demo 01

Rollout one episode. Compute loss, gradient, update.

```python
def train_one_epoch(env, memory, Q_net, baseline_Q_net, opt_parameters, optimizer):
    Q_net.train()
    baseline_Q_net.eval()
    rollout_Q_net = Rollout_Episodes()
    epoch_loss = 0
    nb_data = 0
    epoch_episode_length = 0
    epoch_episode_lengths = []
    nb_batches_per_epoch = opt_parameters['nb_batches_per_epoch']
    for iter in range(nb_batches_per_epoch):
        opt_parameters['env_seed'] = torch.LongTensor(opt_parameters['nb_episodes_per_batch']).random_(1,10000)
        batch_episode_lengths = rollout_Q_net.rollout_batch_episodes(env, memory, opt_parameters, Q_net)
        loss = Q_net.loss(memory, baseline_Q_net, opt_parameters)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.detach().item()
        nb_data += len(batch_episode_lengths)
        epoch_episode_length += torch.tensor(batch_episode_lengths).float().sum()
        epoch_episode_lengths.append(epoch_episode_length)
    epoch_loss /= nb_data
    epoch_episode_length /= nb_data
    return epoch_loss, epoch_episode_length, epoch_episode_lengths
```

Demo 01

Train multiple epochs

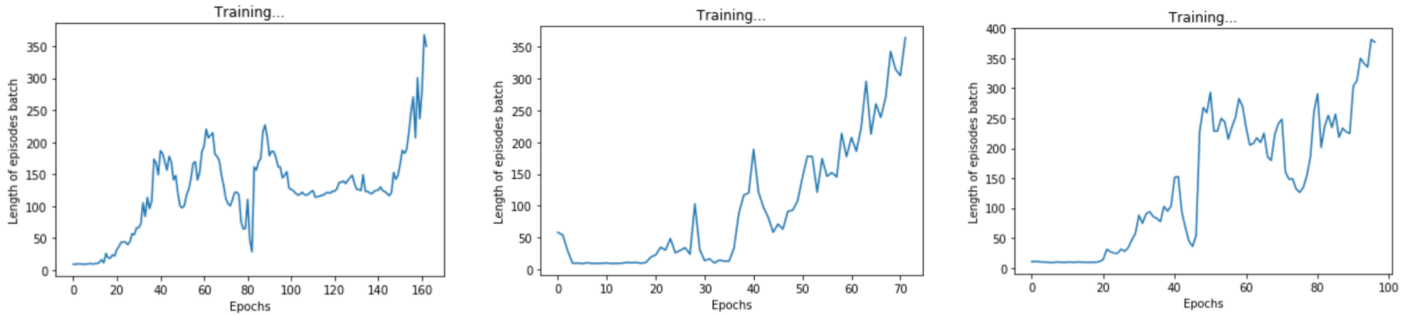
Train one epoch by rolling out a few episodes

Smoothly decreasing the random action prob (space exploration parameter)

Updating the baseline network if greedy evaluation of current Q network is better.

Stopping condition (high reward)

```
for epoch in range(200):

    # train one epoch
    epoch_train_loss, epoch_episode_length, epoch_episode_lengths = \
        train_one_epoch(env, memory, Q_net, baseline_Q_net, opt_parameters, optimizer)

    # update random_action_prob(=lr), linear annealing from xx% to 1%
    opt_parameters['rand_act_pr'] = max(0.01, init_rand_act_pr - init_rand_act_pr*(epoch/float(100)))

    # update baseline if current policy better (use greedy mode for evaluation)
    if not epoch%opt_parameters['baseline_update']:
        opt_parameters_baseline['env_seed'] = torch.LongTensor(opt_parameters_baseline['nb_episodes_per_batch']).random_(1,10000)
        opt_parameters_baseline['rand_act_pr'] = opt_parameters['rand_act_pr']
        batch_episode_lengths_update = Rollout_Episodes().rollout_batch_episodes(env, memory, opt_parameters, Q_net, False)
        batch_episode_lengths_update_baseline = Rollout_Episodes().rollout_batch_episodes(env, memory, opt_parameters, Q_net, False)
        if torch.Tensor(batch_episode_lengths_update).mean() > torch.Tensor(batch_episode_lengths_update_baseline).mean():
            print('UPDATE BASELINE - epoch:',epoch)
            baseline_Q_net.load_state_dict(Q_net.state_dict())
        else:
            print('NO UPDATE BASELINE - epoch:',epoch)

    # stop training when reward is high
    if epoch_episode_length > env.spec.reward_threshold:
        print('Training done.')
        print("Last episode length is {}, epoch is {}, random_action_prob is {}".
              format(epoch_episode_length, epoch, opt_parameters['rand_act_pr']))
        break
```



DQN Algorithm

- Advantages :
  - Memory Replay module memorizes the history of (state,action,reward).
    - Examples : All professional games of Go, Chess, etc can be memorized.
  - Memory Replay is a nice attractive feature toward AI.
- Challenges :
  - Hard to compute a good baseline.
  - Hard to control the exploration of the space (state,action).

Outline

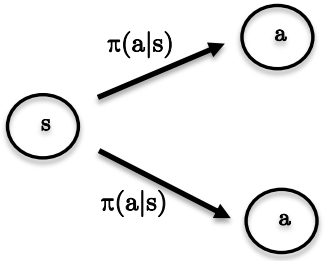
- RL and Deep Learning
- Agent, Environment and MDP
- Policy, Value Function and Model
- Optimal Value Function and Policy
- Deep Q-Learning (DQN)
- **Policy Networks**
- Actor-Critic Algorithm
- RL and Supervised Learning
- Learning and Planning

Policy Networks

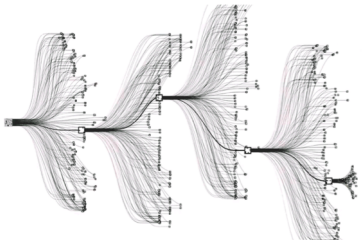
- Find optimal policy that maximizes an expected reward :
  - Local discarded rewards :
    - Optimize the expected reward of all actions and states :
      - $\max_\pi \mathbb{E}_{a,s}$ (Q(a|s))
      - Q(a|s) : Local discarded rewards for taking action a in state s in e.g. cartpole simulation.
  - Global/sparse rewards :
    - Optimize the expected reward of all episodes/trajectories :
      - $\max_\pi \mathbb{E}_{e \sim \pi, P}$ (Q(e))
      - Q(e) : Global rewards s.a. win/loss in Go or length of cartpole episode.

Policy Network for Local Discarded Rewards

- Optimize the expected reward of all actions and states :
  - $\max_\pi \mathbb{E}_{a,s}$ ($Q(a|s)$)
    $\Sigma_{a,s}$ $\pi(a|s)$. $Q(a|s)$



- It is intractable to compute the exact policy function π in high-dimensional spaces.



- We will approximate the policy function π with a neural network :
  - $\pi(a|s) \Rightarrow \pi_w(a|s)$

Policy Gradient

- Policy gradient :
  - $\nabla_w \left(\mathbb{E}_{a,s} \left(Q(a|s) \right) \right) = \nabla_w \left(\Sigma_{a,s} \pi_w(a|s) \, . \, Q(a|s) \right)$

$$
\begin{aligned}
&= \Sigma_{a,s} \left(\nabla_w \pi_w(a|s) \right) . Q(a|s) \\
&= \Sigma_{a,s} \pi_w(a|s)/\pi_w(a|s) \, \nabla_w \pi_w(a|s) \, . \, Q(a|s) \\
&= \Sigma_{a,s} \pi_w(a|s) \left(\nabla_w \pi_w(a|s)/\pi_w(a|s) \right) . Q(a|s) \\
&= \Sigma_{a,s} \pi_w(a|s) \, \nabla_w \log \pi_w(a|s) \, . \, Q(a|s) \\
&= \mathbb{E}_{a,s} \left(\nabla_w \log \pi_w(a|s) \, . \, Q(a|s) \right) \\
&= \nabla_w \underbrace{\left(\mathbb{E}_{a,s} \left(\log \pi_w(a|s) \, . \, Q(a|s) \right) \right)}
\end{aligned}
$$

$$\Downarrow$$

Policy loss for local discarded rewards :

$$L_L(w) = \mathbb{E}_{a,s} \left(- \log \pi_w(a|s) \, . \, Q(a|s) \right)$$

$$\min_w L_L(w)$$

Interpretation of Policy Gradient

- Policy gradient :

$$\mathbb{E}_{a,s}\left(- \nabla_w \log \pi_w(a|s) \cdot Q(a|s) \right)$$

Score function (log likelihood)

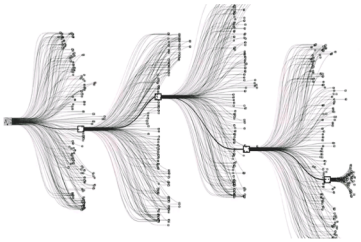
Reward for action a in state s

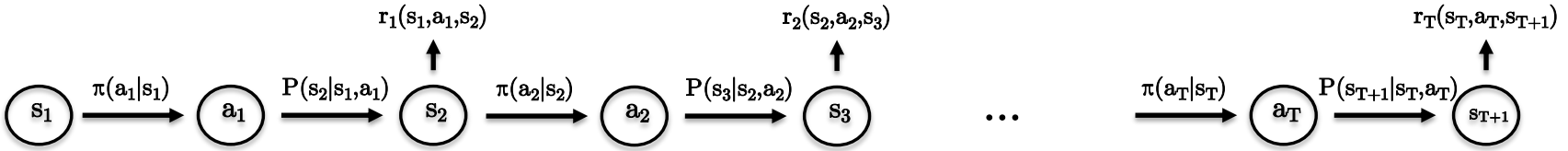
Gradient of score function favors actions of high probability, and diminishes actions with low probability.

This term increases probability of actions with positive reward Q, and decreases probability of actions with negative reward Q.

Policy Gradient Approximation

- It is intractable to compute the exact gradient in high-dimensional spaces.
- We will approximate it by running multiple episodes.
- One episode roll-out :
  - $e = s_1,\ a_1 \sim \pi(a|s_1),\ s_2 \sim P(s|s_1,a_1),\ r_1,\ a_2 \sim \pi(a|s_2),\ s_3 \sim P(s|s_2,a_2),\ r_2,$
    $\ldots\ s_T \sim P(s|s_{T-1},a_{T-1}),\ a_T \sim \pi(a|s_T),\ s_{T+1} \sim P(s|s_T,a_T),\ r_T$





Loss Approximation

- Loss for one episode :
  - $L_L(w) = \mathbb{E}_{a,s}\ (\ -\log \pi_w(a|s)\ .\ Q(a|s)\ )$

    $\approx\ 1/T\ \ \Sigma_t\ \ -\log \pi_w(a_t|s_t)\ .\ Q(a_t|s_t)$

Approximation of expected value for one episode

Number of time steps in one episode

with $Q(a_t|s_t) = r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + \ldots + \gamma^T.r_T$ Discounted reward

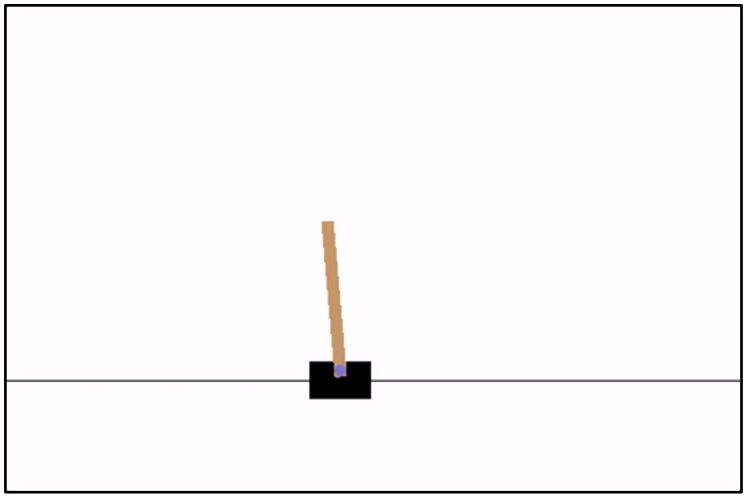
- Loss for a mini-batch of M episodes :
  - $L_L(w)\ \approx 1/M\ \ \Sigma_m\ 1/T^m\ \ \Sigma_t\ \ -\log \pi_w(a_t^m|s_t^m)\ .\ Q(a_t^m|s_t^m)$

REINFORCE Algorithm

- Deep policy network algorithm :
  - Initialize the policy network at random $w^{k=0}$ for $\pi_w(a|s)$.
  - Repeat :
    - Collect a mini-batch of M episodes e^m by rolling out the current policy network π_w.
    - Update the policy network by stochastic gradient descent :
      - $w^{k+1} = w^k - lr \cdot \underbrace{\nabla_w \left(1/M \ \Sigma_m \ 1/T^m \ \Sigma_t \ - \log \pi_w(a_t^m|s_t^m) \cdot Q(a_t^m|s_t^m) \right)}_{\approx \nabla_w L_L(w)}$
- This algorithm is known as REINFORCE or Monte-Carlo policy gradient.

Demo 02

- Finding the optimal policy network with local discarded rewards.



Optimal policy network

Demo 02

Policy network

Compute action probabilities given a state s

Selection an action a given a state s

Select action by Bernouilli random sampling to explore the space of (state,action)

Loss

Policy loss for one episode

Compute discarded rewards

Z-score discarded rewards

```python
# class of policy network
class Policy_NN(nn.Module):
    def __init__(self, net_parameters):
        super(Policy_NN, self).__init__()
        input_dim = net_parameters['input_dim']
        hidden_dim = net_parameters['hidden_dim']
        output_dim = net_parameters['output_dim']
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        actions_score = self.fc2(x)
        actions_prob = torch.softmax(actions_score, dim=1)
        return actions_prob

    def select_action(self, state): # select action w/ policy network
        probs = self.forward(state) # probability of action a in state s
        bernoulli_sampling = torch.distributions.Categorical(probs)
        action = bernoulli_sampling.sample() # sample action a with Bernoulli sampling
        log_prob = bernoulli_sampling.log_prob(action) # compute log prob of selected action
        action = action.item()
        return action, log_prob

    def loss(self, batch_rewards, batch_log_probs):
        nb_episodes_per_batch = len(batch_rewards)
        batch_policy_losses = []
        gamma = opt_parameters['gamma']
        for episode in range(nb_episodes_per_batch):
            episode_rewards = batch_rewards[episode]
            episode_log_probs = batch_log_probs[episode]
            R = 0; policy_loss = []; rewards = []
            for r in episode_rewards[::-1]: # compute the discarded award at each time step
                R = r + gamma * R
                rewards.insert(0, R)
            episode_rewards = torch.tensor(rewards)
            episode_log_probs = torch.stack(episode_log_probs)
            episode_rewards = (episode_rewards - episode_rewards.mean())
                / (episode_rewards.std() + 1e-16) # z-score the rewards
            for log_prob, reward in zip(episode_log_probs, episode_rewards):
                policy_loss.append( -log_prob * reward )
            policy_loss = torch.cat(policy_loss).sum()
            batch_policy_losses.append(policy_loss)
        loss = torch.stack(batch_policy_losses).mean()
        return loss
```

Demo 02

Rollout class

Rollout one episode

Collect all local rewards and log probability of actions

```python
# class of rollout episodes
class Rollout_Episodes():

    def __init__(self,):
        super(Rollout_Episodes, self).__init__()

    def rollout_batch_episodes(self, env, opt_parameters, policy_net):
        # storage structure of all episodes (w/ different lengths):
        #   batch_rewards =         [ [- - -] , [- - - -] , ... , [- -] ]
        #   batch_log_probs =       [ [- - -] , [- - - -] , ... , [- -] ]
        #   batch_episode_lengths = [    - ,        -      , ...,    -   ]
        nb_episodes_per_batch = opt_parameters['nb_episodes_per_batch']
        env_seeds = opt_parameters['env_seed']
        batch_rewards = []
        batch_log_probs = []
        batch_episode_lengths = []
        for episode in range(nb_episodes_per_batch):
            rewards = []
            log_probs = []
            env.seed(env_seeds[episode].item()) # start with random seed
            state = env.reset() # reset environment
            for t in range(1000): # rollout one episode
                state_pytorch = torch.from_numpy(state).float().unsqueeze(0) # state=s
                action, log_prob = policy_net.select_action(state_pytorch) # select action=a from state=s
                state, reward, done, _ = env.step(action) # receive next state=s' and reward=r
                rewards.append(reward)
                log_probs.append(log_prob)
                if done:
                    batch_episode_lengths.append(t)
                    batch_rewards.append(rewards)
                    batch_log_probs.append(log_probs)
                    break
        return batch_rewards, batch_log_probs, batch_episode_lengths
```

Demo 02

Rollout one episode. Compute loss, gradient, update.

```python
def train_one_epoch(env, policy_net, opt_parameters):
    """
    train one epoch
    """
    policy_net.train()
    rollout_policy_net = Rollout_Episodes()
    epoch_loss = 0
    nb_data = 0
    epoch_episode_length = 0
    epoch_episode_lengths = []
    nb_batches_per_epoch = opt_parameters['nb_batches_per_epoch']
    for iter in range(nb_batches_per_epoch):
        batch_rewards, batch_log_probs, batch_episode_lengths = \
            rollout_policy_net.rollout_batch_episodes(env, opt_parameters, policy_net)
        loss = policy_net.loss(batch_rewards, batch_log_probs)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.detach().item()
        nb_data += len(batch_episode_lengths)
        epoch_episode_length += torch.tensor(batch_episode_lengths).float().sum()
        epoch_episode_lengths.append(epoch_episode_length)
    epoch_loss /= nb_data
    epoch_episode_length /= nb_data
    return epoch_loss, epoch_episode_length, epoch_episode_lengths
```

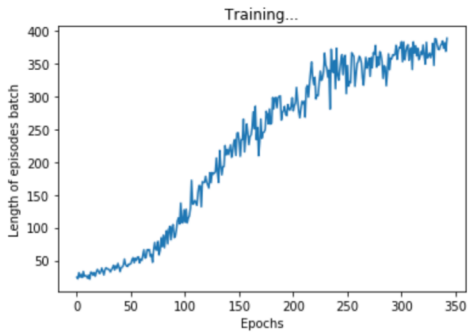
Demo 02

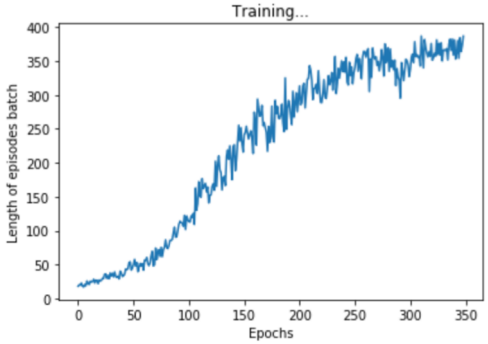
Train multiple epochs

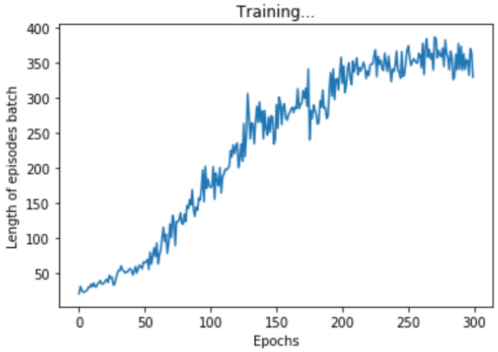
Train one epoch by rolling out a few episodes

```
for epoch in range(500):

    # train one epoch
    epoch_train_loss, epoch_episode_length, epoch_episode_lengths = train_one_epoch(env, policy_net, opt_parameters)

    # stop training when reward is high
    if epoch_episode_length > env.spec.reward_threshold:
        print('Training done.')
        print("Last episode length is {}, epoch is {}".
              format(epoch_episode_length, epoch))
        break
```

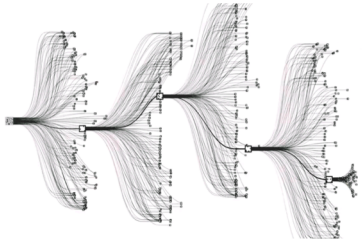
Stopping condition (high reward)







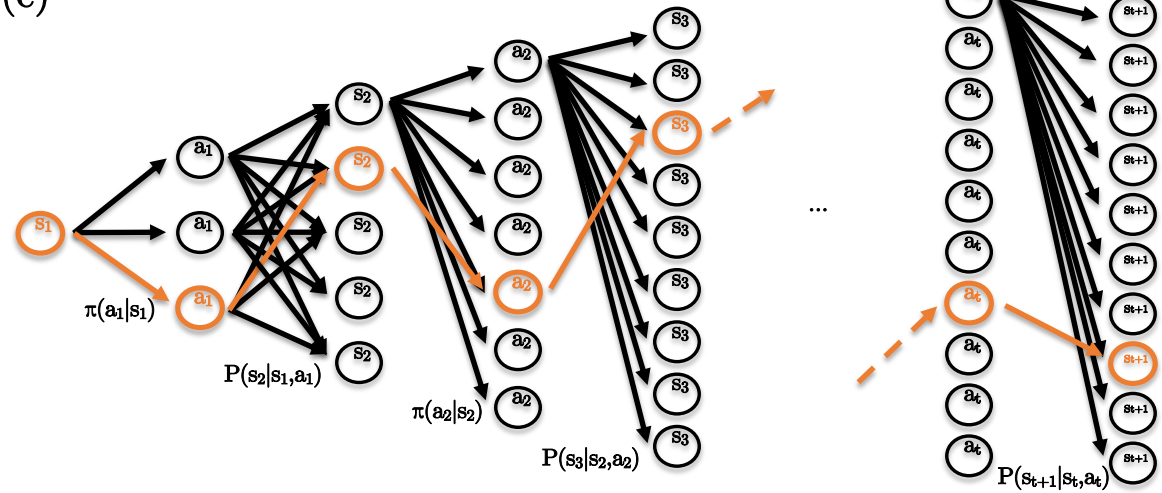
Policy Network for Global Rewards



- Optimize the expected reward of all episodes/trajectories :
  - $\max_\pi \; \mathbb{E}_{e \sim \pi,P} \; (\; Q(e) \;)$

    $\Sigma_e \; P_e(e|\pi,P) \; . \; Q(e)$

Probability of episode e given π,P



- Let us represent the policy function π with a neural network :
  - $\pi(a|s) \; \Rightarrow \; \pi_w(a|s)$

Policy Gradient

- Policy gradient :
  - $\nabla_w \left(\mathbb{E}_{e \sim \pi,P} \left(Q(e) \right) \right) = \nabla_w \left(\Sigma_e P_e(e|\pi_w,P) \cdot Q(e) \right)$

$$= \Sigma_e \nabla_w P_e(e|\pi_w,P) \cdot Q(e)$$

$$= \Sigma_e P_e(e|\pi_w,P) \left(\nabla_w P_e(e|\pi_w,P) / P_e(e|\pi_w,P) \right) \cdot Q(e)$$

$$= \Sigma_e P_e(e|\pi_w,P) \nabla_w \log P_e(e|\pi_w,P) \cdot Q(e)$$

$$= \mathbb{E}_e \left(\nabla_w \log P_e(e|\pi_w,P) \cdot Q(e) \right)$$

$$\approx 1/M \ \Sigma_m \nabla_w \log \underbrace{P_e(e^m|\pi_w,P)} \cdot Q(e^m)$$

Approximation of expected value with a mini-batch of M episodes

Probability of having the m^{th} episode

$$P_e(e^m|\pi_w,P) = \prod_t \pi_w(a_t^m|s_t^m) \cdot P(s_{t+1}^m|s_t^m,a_t^m)$$

Policy Gradient

- Policy gradient :
  - $\nabla_w \left(\mathbb{E}_{e \sim \pi,P} \left(Q(e) \right) \right) \approx 1/M \ \Sigma_m \nabla_w \log P_e(e^m|\pi_w,P) \ . \ Q(e^m)$
    $\approx 1/M \ \Sigma_m \nabla_w \log \left(\prod_t \pi_w(a_t^m|s_t^m) \ . \ P(s_{t+1}^m|s_t^m,a_t^m) \right) \ . \ Q(e^m)$
    $\approx 1/M \ \Sigma_m \nabla_w \left(\Sigma_t \log \pi_w(a_t^m|s_t^m) + \underbrace{\Sigma_t \log P(s_{t+1}^m|s_t^m,a_t^m)}_{\text{Gradient is zero as term is independent of w.}} \right) \ . \ Q(e^m)$

    The policy gradient is independent of the dynamics of the environment.

    No need to know the dynamics of the world to act optimally !

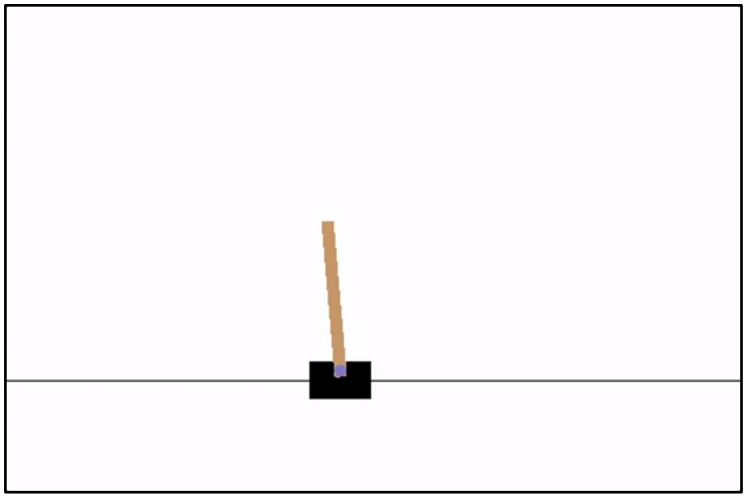
  - $\nabla_w \left(\mathbb{E}_{e \sim \pi,P} \left(Q(e) \right) \right) \approx \nabla_w \underbrace{\left(1/M \ \Sigma_m \Sigma_t \log \pi_w(a_t^m|s_t^m) \ . \ Q(e^m) \right)}_{\Downarrow}$

    Policy loss for global reward :

    $$L_G(w) = 1/M \ \Sigma_m \Sigma_t \ - \log \pi_w(a_t^m|s_t^m) \ . \ Q(e^m)$$
    $$\min_w L_G(w)$$

Demo 03

- Finding the optimal policy network with global rewards.



Optimal policy network

Demo 03

Policy network

Compute action probabilities given a state s

Selection an action a given a state s

Loss

Compare current policy with baseline

Policy loss for one episode

```python
# class of policy network
class Policy_NN(nn.Module):

    def __init__(self, net_parameters):
        super(Policy_NN, self).__init__()
        input_dim = net_parameters['input_dim']
        hidden_dim = net_parameters['hidden_dim']
        output_dim = net_parameters['output_dim']
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        actions_score = self.fc2(x)
        actions_prob = torch.softmax(actions_score, dim=1)
        return actions_prob

    def select_action(self, state): # select action w/ policy network
        probs = self.forward(state) # probability of action a in state s
        bernoulli_sampling = torch.distributions.Categorical(probs)
        action = bernoulli_sampling.sample() # sample action a with Bernoulli sampling
        log_prob = bernoulli_sampling.log_prob(action) # compute log prob of selected action
        action = action.item()
        return action, log_prob

    def loss(self, batch_rewards, batch_log_probs, batch_rewards_baseline):
        nb_episodes_per_batch = len(batch_rewards)
        batch_episode_rewards = torch.Tensor(batch_rewards)
        batch_episode_rewards_baseline = torch.Tensor(batch_rewards_baseline)
        batch_episode_rewards -= batch_episode_rewards_baseline # compare current model w.r.t. baseline model
        batch_policy_losses = []
        for episode in range(nb_episodes_per_batch):
            episode_reward = batch_episode_rewards[episode]
            episode_log_probs = torch.stack(batch_log_probs[episode])
            policy_loss = - episode_log_probs.sum() * episode_reward
            batch_policy_losses.append(policy_loss)
        loss = torch.stack(batch_policy_losses).mean()
        return loss
```

Demo 03

Rollout class

Rollout one episode

Collect all global rewards, i.e. length of each episode

```python
# class of rollout episodes
class Rollout_Episodes():

    def __init__(self):
        super(Rollout_Episodes, self).__init__()

    def rollout_batch_episodes(self, env, opt_parameters, policy_net):
        # storage structure of all episodes (w/ different lengths):
        #   batch_rewards =         [    - ,       -      , ...,    -   ]
        #   batch_log_probs =       [ [- - -] , [- - - -] , ... , [- -] ]
        #   batch_episode_lengths = [    - ,       -      , ...,    -   ]
        nb_episodes_per_batch = opt_parameters['nb_episodes_per_batch']
        env_seeds = opt_parameters['env_seed']
        batch_rewards = []
        batch_log_probs = []
        batch_episode_lengths = []
        for episode in range(nb_episodes_per_batch):
            rewards = []
            log_probs = []
            env.seed(env_seeds[episode].item()) # start with random seed
            state = env.reset() # reset environment
            for t in range(1000): # rollout one episode
                state_pytorch = torch.from_numpy(state).float().unsqueeze(0) # state=s
                action, log_prob = policy_net.select_action(state_pytorch) # select action=a from state=s
                state, reward, done, _ = env.step(action) # receive next state=s' and reward=r
                rewards.append(reward)
                log_probs.append(log_prob)
                if done:
                    batch_episode_lengths.append(t)
                    batch_rewards.append(t)
                    batch_log_probs.append(log_probs)
                    break
        return batch_rewards, batch_log_probs, batch_episode_lengths
```

Demo 03

Rollout one episode for current and baseline policies. Compute loss, gradient, update.

```python
def train_one_epoch(env, policy_net, baseline_policy_net, opt_parameters):
    """
    train one epoch
    """
    policy_net.train()
    baseline_policy_net.eval()
    rollout_policy = Rollout_Episodes()
    rollout_baseline_policy = Rollout_Episodes()
    epoch_loss = 0
    nb_data = 0
    epoch_episode_length = 0
    epoch_episode_lengths = []
    nb_batches_per_epoch = opt_parameters['nb_batches_per_epoch']
    for iter in range(nb_batches_per_epoch):
        opt_parameters['env_seed'] = torch.LongTensor(opt_parameters['nb_episodes_per_batch']).random_(1,10000)
        batch_rewards, batch_log_probs, batch_episode_lengths = \
            rollout_policy.rollout_batch_episodes(env, opt_parameters, policy_net)
        batch_rewards_baseline, batch_log_probs_baseline, batch_episode_lengths_baseline = \
            rollout_baseline_policy.rollout_batch_episodes(env, opt_parameters, baseline_policy_net)
        loss = policy_net.loss(batch_rewards, batch_log_probs, batch_rewards_baseline)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.detach().item()
        nb_data += len(batch_episode_lengths)
        epoch_episode_length += torch.tensor(batch_episode_lengths).float().sum()
        epoch_episode_lengths.append(epoch_episode_length)
    epoch_loss /= nb_data
    epoch_episode_length /= nb_data
    return epoch_loss, epoch_episode_length, epoch_episode_lengths
```

Demo 03

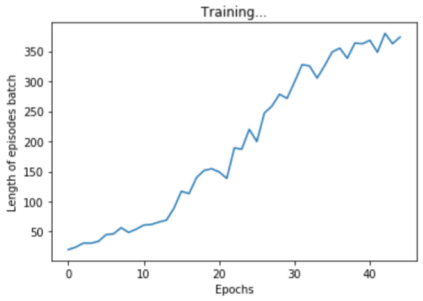
Train multiple epochs

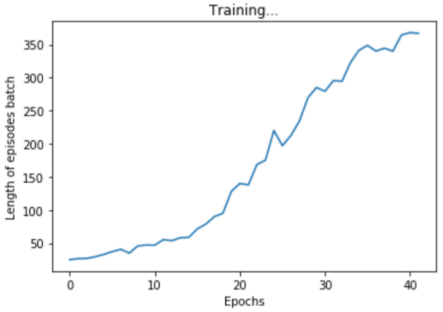
Train one epoch by rolling out a few episodes

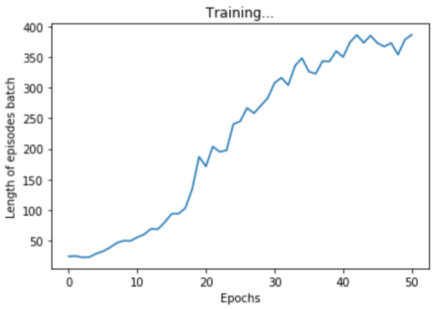
```
for epoch in range(500):

    # train one epoch
    epoch_train_loss, epoch_episode_length, epoch_episode_lengths = \
        train_one_epoch(env, policy_net, baseline_policy_net, opt_parameters)

    # update baseline if current policy better
    if epoch>0 and (not epoch%2):
        opt_parameters['env_seed'] = torch.LongTensor(10).random_(1,10000)
        _, _, batch_episode_lengths = Rollout_Episodes().rollout_batch_episodes(env, opt_parameters, policy_net)
        _, _, batch_episode_lengths_baseline = Rollout_Episodes().rollout_batch_episodes(env, opt_parameters, baseline_policy_net)
        if torch.Tensor(batch_episode_lengths).mean() > torch.Tensor(batch_episode_lengths_baseline).mean():
            print('UPDATE BASELINE - epoch:',epoch)
            baseline_policy_net.load_state_dict(policy_net.state_dict())
        else:
            print('NO UPDATE BASELINE - epoch:',epoch)

    # stop training when reward is high
    if epoch_episode_length > env.spec.reward_threshold:
        print('Training done.')
        print("Last episode length is {}, epoch is {}".
              format(epoch_episode_length, epoch))
        break
```

Updating the baseline policy if greedy evaluation of current policy is better.

Stopping condition (high reward)







Policy Algorithms

- Policy network with local rewards :
  - Good to receive lots of local rewards to learn faster and better.
  - Hard to define a good baseline (use z-score of local rewards of mini-batch of episodes).
  - Slow to learn long-term strategy

- Policy network with global rewards :
  - We receive only a global/sparse reward at the end of the episode.
  - Easy to define a good baseline (by greedy roll-out)
  - Faster to learn long-term strategy.

Outline

- RL and Deep Learning
- Agent, Environment and MDP
- Policy, Value Function and Model
- Optimal Value Function and Policy
- Deep Q-Learning (DQN)
- Policy Networks
- Actor-Critic Algorithm
- RL and Supervised Learning
- Learning and Planning

Actor-Critic Algorithm

- Monte-Carlo policy gradient (i.e. policy w/ local discarded rewards) a.k.a. REINFORCE has a high variance (because of $Q(a_t|s_t)$) :

$$\nabla_w L_L(w) = \Sigma_t \; - \nabla_w \log \pi_w(a_t|s_t) \; . \; Q(a_t|s_t)$$

$$\text{where } Q(a_t|s_t) = r_t + \gamma.r_{t+1} + \gamma^2.r_{t+2} + \ldots + \gamma^T.r_T = \Sigma_{k=t}{}^{T} \; \gamma^{k-t}.r_k$$

- The variance of the policy (actor) can be reduced by constraining $Q(a_t|s_t)$ (critic) to satisfy the Bellman equation :

$$Q(a_t|s_t) = \mathbb{E}_{\pi,P} \; (\; r_t + \gamma. \; Q(a_{t+1}|s_{t+1}) \;)$$

$$\approx \; r_t + \gamma. \; Q(a_{t+1}|s_{t+1}) \quad \text{(during one episode)}$$

- $Q(a|s)$ will be estimated with a neural network.
- One-Step Actor-Critic (QAC) loss :
  - $L_{QAC}(w,v) = \Sigma_t \; - \log \pi_w(a_t|s_t) \; . \; Q_v(a_t|s_t)$

$$+ \; \Sigma_t \; || \; Q_v(a_t|s_t) - (\; r_t + \gamma. \; Q_v(a_{t+1}|s_{t+1}) \;) \; ||_2{}^2$$

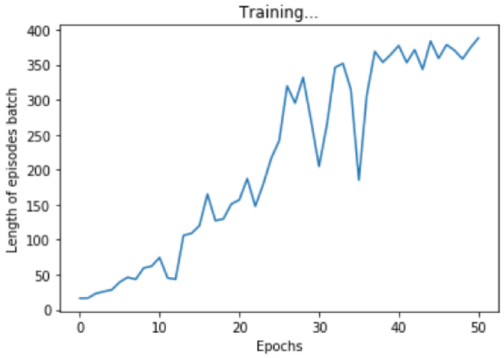
QAC Algorithm

- One-Step Actor-Critic (QAC) algorithm :
  - Initialize w and v randomly.
  - Repeat :
    - Roll-out : s_t, $a_t \sim \pi_w$, r_t, s_{t+1}, $a_{t+1} \sim \pi_w$
    - Error : $e_t = Q_v(a_t|s_t) - (r_t + \gamma . Q_v(a_{t+1}|s_{t+1}))$
    - Actor update : $w^{k+1} = w^k$ - lr . $\Sigma_t \nabla_w$ - log $\pi_w(a_t|s_t)$. $Q_v(a_t|s_t)$
    - Critic update : $v^{k+1} = v^k$ - lr . e_t . $\nabla_v Q_v(a_t|s_t)$
- We simultaneously estimate the policy π_w (actor) and the action-value function Q_v (critic) that evaluates the policy :
  - $Q_v(a_t|s_t)$ is the evaluation of long-term returned by the critic at time t.
  - e_t is the estimated error of long-term returned at time t.

Demo 04

Lab 04 : One-Step Actor-Critic (QAC) - demo

```python
# class of policy network
class ActorCritic_NN(nn.Module):

    def __init__(self, net_parameters):
        super(ActorCritic_NN, self).__init__()
        input_dim = net_parameters['input_dim']
        hidden_dim = net_parameters['hidden_dim']
        output_dim = net_parameters['output_dim']
        # policy network
        self.fc1_p = nn.Linear(input_dim, hidden_dim)
        self.fc2_p = nn.Linear(hidden_dim, output_dim)
        # state-value function network
        self.fc1_q = nn.Linear(input_dim, hidden_dim)
        self.fc2_q = nn.Linear(hidden_dim, output_dim)

    def forward_policy(self, x):
        x = torch.relu(self.fc1_p(x))
        actions_score = self.fc2_p(x)
        actions_prob = torch.softmax(actions_score, dim=1)
        return actions_prob

    def forward_Q(self, x):
        x = torch.relu(self.fc1_q(x))
        Q_scores = self.fc2_q(x) # scores over actions
        return Q_scores

    def select_action(self, state): # select action w/ policy network
        probs = actorcritic_net.forward_policy(state) # probability of action a in state s
        bernoulli_sampling = torch.distributions.Categorical(probs)
        action = bernoulli_sampling.sample() # sample action a with Bernoulli sampling
        return action

    def loss(self, batch):
```



AAC Algorithm

- One-Step Actor-Critic (QAC) loss :

$$L_{QAC}(w,v) = \Sigma_t \; - \log \pi_w(a_t|s_t) \; . \; Q_v(a_t|s_t) + \Sigma_t \; || \; Q_v(a_t|s_t) - (\; r_t + \gamma . \; Q_v(a_{t+1}|s_{t+1}) \;) \; ||_2^2$$

  and its policy gradient :

$$\nabla_w L_P(w) = \Sigma_t \; - \nabla_w \log \pi_w(a_t|s_t) \; . \; Q_v(a_t|s_t)$$

  is biased and may not find the optimal policy.

- A baseline function b(s) is introduced to remove the bias :

$$\nabla_w L_P(w) = \Sigma_t \; - \nabla_w \log \pi_w(a_t|s_t) \; . \; (\; Q_v(a_t|s_t) - b(s_t) \;)$$

- A good baseline is

  $b(s_t) = Q(s_t)$, the state-value function in state s_t, i.e. the future reward from state s_t.

- Q(s) will be estimated with a neural network.

AAC Algorithm

- Advantage Actor-Critic (AAC or A2C) loss :

$$L_{AAC}(w,v,u) = \Sigma_t \; - \log \pi_w(a_t|s_t) \; . \; (\; Q_v(a_t|s_t) - Q_u(s_t) \;)$$
$$+ \Sigma_t \; || \; Q_v(a_t|s_t) - (\; r_t + \gamma . \; Q_v(a_{t+1}|s_{t+1}) \;) \; ||_2^2$$
$$+ \Sigma_t \; || \; Q_u(s_t) \; - \; Q(a_t|s_t) \; ||_2^2 \quad \text{with} \quad Q(a_t|s_t) = \Sigma_{k=t}^{T} \; \gamma^{k-t} . r_k$$

and its policy gradient :

$$\nabla_w \, L_{AAC}(w) = \Sigma_t \; - \nabla_w \log \pi_w(a_t|s_t) \; . \; A(a_t|s_t)$$

where $A(a_t|s_t) = Q_v(a_t|s_t) - Q_u(s_t)$ is the Advantage function.

- This gradient policy is unbiased and has reduced variance.

AAC Algorithm

- Advantage Actor-Critic (AAC or A2C) algorithm :
  - Initialize w, v, u randomly.
  - Repeat :
    - Roll-out an episode and write in memory : s_t, $a_t \sim \pi_w$, r_t, s_{t+1}, $a_{t+1} \sim \pi_w$
    - Discounted reward : $Q(a_t|s_t) = \Sigma_{k=t}^{T} \gamma^{k-t}.r_k$
    - Errors : $e_t^1 = Q_v(a_t|s_t) - (r_t + \gamma. Q_v(a_{t+1}|s_{t+1}))$ and $e_t^2 = Q_u(s_t) - Q(a_t|s_t)$
    - Advantage function : $A(a_t|s_t) = Q_v(a_t|s_t) - Q_u(s_t)$
    - Update : $w^{k+1} = w^k - lr . \Sigma_t \nabla_w - \log \pi_w(a_t|s_t) . A(a_t|s_t)$
    - Update : $v^{k+1} = v^k - lr . e_t^1 . \nabla_v Q_v(a_t|s_t)$
    - Update : $u^{k+1} = u^k - lr . e_t^2 . \nabla_v Q_u(s_t)$

Demo 05

Actor-critic network

Policy (actor)

Evaluate action-value Q function (critic)

Evaluate state-value Q function (critic)

Action selected by policy

```python
# class of policy network
class ActorCritic_NN(nn.Module):

    def __init__(self, net_parameters):
        super(ActorCritic_NN, self).__init__()
        input_dim = net_parameters['input_dim']
        hidden_dim = net_parameters['hidden_dim']
        output_dim = net_parameters['output_dim']
        # policy network
        self.fc1_p = nn.Linear(input_dim, hidden_dim)
        self.fc2_p = nn.Linear(hidden_dim, output_dim)
        # action-value function network
        self.fc1_q_a = nn.Linear(input_dim, hidden_dim)
        self.fc2_q_a = nn.Linear(hidden_dim, output_dim)
        # state-value function network
        self.fc1_q_s = nn.Linear(input_dim, hidden_dim)
        self.fc2_q_s = nn.Linear(hidden_dim, 1)

    def forward_policy(self, x):
        x = torch.relu(self.fc1_p(x))
        actions_score = self.fc2_p(x)
        actions_prob = torch.softmax(actions_score, dim=1)
        return actions_prob

    def forward_Q_a(self, x):
        x = torch.relu(self.fc1_q_a(x))
        Q_scores = self.fc2_q_a(x) # scores over actions
        return Q_scores

    def forward_Q_s(self, x):
        x = torch.relu(self.fc1_q_s(x))
        Q_scores = self.fc2_q_s(x) # scores over actions
        return Q_scores

    def select_action(self, state): # select action w/ policy network
        probs = self.forward_policy(state) # probability of action a in state s
        bernoulli_sampling = torch.distributions.Categorical(probs)
        action = bernoulli_sampling.sample() # sample action a with Bernoulli sampling
        return action

    def loss(self, batch):
```

Demo 05

Loss

Reward must be negative

Policy loss for one episode

State-value Q function loss for one episode

State Q function loss for one episode

```python
def loss(self, batch):
    gamma = opt_parameters['gamma']
    nb_episodes_per_batch = len(batch.states)
    batch_losses = []
    for episode in range(nb_episodes_per_batch):
        episode_states = torch.stack( batch.states[episode] ).float() #size=B x 4
        episode_next_states = torch.stack( batch.next_states[episode] ).float() #size=B x 4
        episode_actions = torch.stack( batch.actions[episode] ).long() #size=B
        episode_rewards = torch.stack( batch.rewards[episode] ).float() #size=B
        episode_dones = torch.stack( batch.dones[episode] ).float() #size=B
        R = 0; policy_loss = []; rewards = []
        for r in batch.rewards[episode][::-1]: # compute the discarded award at each time step
            R = r + gamma * R
            rewards.insert(0, R)
        episode_discounted_rewards = torch.tensor(rewards).float() #size=B
        episode_next_actions = self.select_action(episode_next_states) #size=B
        Q = self.forward_Q_a(episode_states).gather(dim=1,index=episode_actions.unsqueeze(1)) # Qv(a|s), size=B x 1
        Q_target = episode_rewards.unsqueeze(1) + gamma *  \
            self.forward_Q_a(episode_next_states).gather(dim=1,index=episode_next_actions.unsqueeze(1)) * episode_dones.unsqueeze(1)
        Q_state = self.forward_Q_s(episode_states)
        logP = torch.log( actorcritic_net.forward_policy(episode_states).gather(dim=1,index=episode_actions.unsqueeze(1)) )
        loss1 = ( -logP * (Q-Q_state).detach() ).mean()
        loss2 = nn.MSELoss()(Q,Q_target.detach())
        loss3 = nn.MSELoss()(Q_state,episode_discounted_rewards.unsqueeze(1).detach())
        loss = loss1 + loss2 + loss3
        batch_losses.append(loss)
    loss = torch.stack(batch_losses).mean()
    return loss
```

Demo 05

Rollout class

Rollout one episode

Write in memory (s,a,s',r)

```python
# class of rollout episodes
class Rollout_Episodes():

    def __init__(self,):
        super(Rollout_Episodes, self).__init__()

    def rollout_batch_episodes(self, env, opt_parameters, actorcritic_net, write_memory=True):
        # storage structure of all episodes (w/ different lengths)
        nb_episodes_per_batch = opt_parameters['nb_episodes_per_batch']
        env_seeds = opt_parameters['env_seed']
        batch = DotDict()
        batch.states=[];  batch.actions=[]; batch.next_states=[]; batch.rewards=[]; batch.dones=[]
        batch_episode_lengths = []
        for episode in range(nb_episodes_per_batch):
            states=[]; actions=[]; next_states=[]; rewards=[]; dones = []
            env.seed(env_seeds[episode].item()) # start with random seed
            state = env.reset() # reset environment
            for t in range(1000): # rollout one episode
                state_pytorch = torch.from_numpy(state).float().unsqueeze(0) # state=s
                action = actorcritic_net.select_action(state_pytorch).item() # select action=a from state=s
                next_state, reward, done, _ = env.step(action) # receive next state=s' and reward=r
                done_mask = 0.0 if done else 1.0
                states.append(torch.tensor(state))
                actions.append(torch.tensor(action))
                next_states.append(torch.tensor(next_state))
                rewards.append(torch.tensor(reward))
                dones.append(torch.tensor(done_mask))
                state = next_state
                if done:
                    batch_episode_lengths.append(t)
                    break
            batch.states.append(states)
            batch.actions.append(actions)
            batch.next_states.append(next_states)
            batch.rewards.append(rewards)
            batch.dones.append(dones)
        return batch_episode_lengths, batch
```

Demo 05

Rollout one episode for current and baseline policy. Compute loss, gradient, update.

```python
def train_one_epoch(env, actorcritic_net, opt_parameters):
    """
    train one epoch
    """
    actorcritic_net.train()
    epoch_loss = 0
    nb_data = 0
    epoch_episode_length = 0
    epoch_episode_lengths = []
    nb_batches_per_epoch = opt_parameters['nb_batches_per_epoch']
    for iter in range(nb_batches_per_epoch):
        batch_episode_lengths, batch = \
            rollout_policy_net.rollout_batch_episodes(env, opt_parameters, actorcritic_net)
        loss = actorcritic_net.loss(batch)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.detach().item()
        nb_data += len(batch_episode_lengths)
        epoch_episode_length += torch.tensor(batch_episode_lengths).float().sum()
        epoch_episode_lengths.append(epoch_episode_length)
    epoch_loss /= nb_data
    epoch_episode_length /= nb_data
    return epoch_loss, epoch_episode_length, epoch_episode_lengths
```

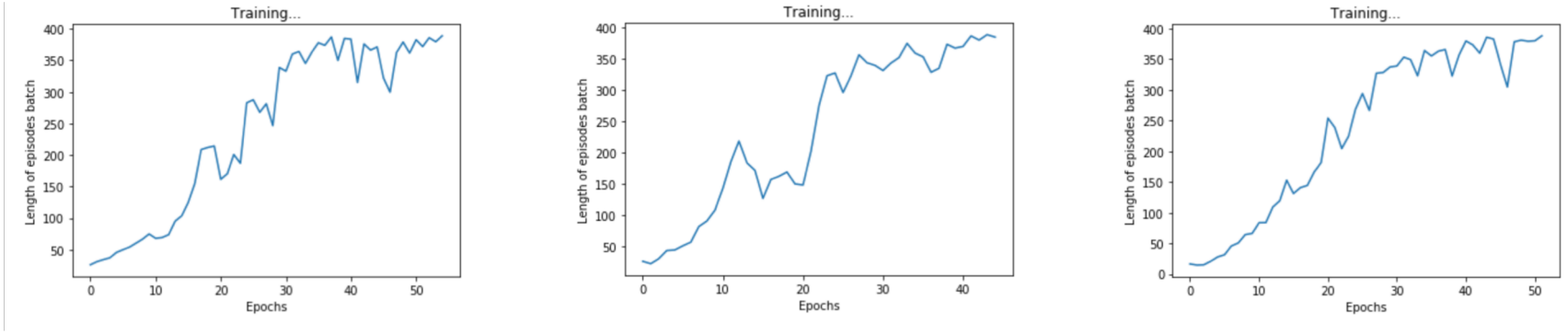
Demo 05

Train multiple epochs

Train one epoch by rolling out a few episodes

```python
for epoch in range(500):

    # train one epoch
    epoch_train_loss, epoch_episode_length, epoch_episode_lengths = train_one_epoch(env, actorcritic_net, opt_parameters)

    # stop training when reward is high
    if epoch_episode_length > env.spec.reward_threshold:
        print('Training done.')
        print("Last episode length is {}, epoch is {}".
              format(epoch_episode_length, epoch))
        break
```

Stopping condition (high reward)



Continuous Actor-Critic Algorithm

- Continuous control :
  - Deterministic policy and continuous action space :
    - $a = \pi(s)$
    - a in R (continuous variable)
  - Find the deterministic policy π that maximizes the expected total reward over all possible actions and states:

$$\max_\pi \; \mathbb{E}_{a=\pi(s),s} \; (\; Q(a|s) \;)$$

$$\Sigma_{a=\pi(s),s} \; Q(a|s)$$

$$\Sigma_s \; Q(\pi(s)|s)$$

Continuous Actor-Critic Algorithm

- Continuous control :
  - Policy gradient :

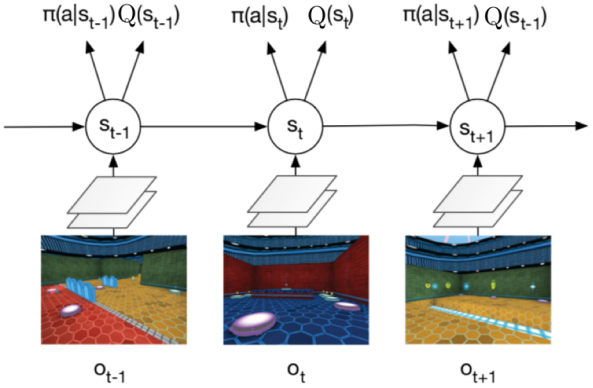
$$
\begin{aligned}
\nabla_w \left(\mathbb{E}_{a=\pi(s),s} \left(Q(a|s) \right) \right) &= \nabla_w \left(\mathbb{E}_s \left(Q(\pi_w(s)|s) \right) \right) \\
&= \mathbb{E}_s \left(\nabla_w \left(Q(\pi_w(s)|s) \right) \right) \\
&= \mathbb{E}_s \left(dQ(\pi_w)/d\pi_w \, . \, \nabla_w \pi_w \right) \text{ // chain rule on differentiable Q (as } \pi_w = a \text{ in R)} \\
&= \mathbb{E}_s \left(\underbrace{dQ(a|s)/da}_{\text{Gradient of the Q-value function. How to adjust the Q-value to take action a in state s more often (or less often).}} \, . \, \underbrace{\nabla_w a}_{\text{How to adjust the policy so that we can take these actions more often (or less often).}} \right) \text{ // as } \pi_w = a
\end{aligned}
$$

$$\underbrace{\qquad\qquad}_{\text{The product is the gradient to follow to make our policy better to get higher Q value.}}$$

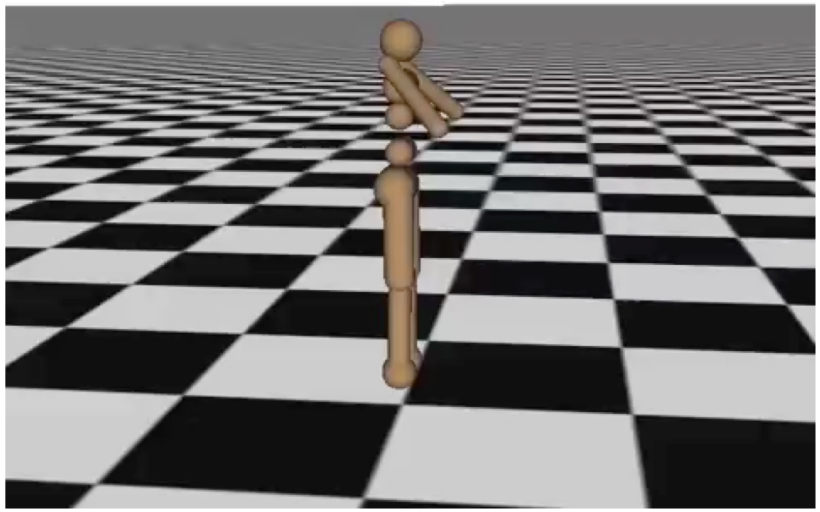
Further Improvements

- Observations o_t are raw pixels from current frame.
- Observations o_t of the maze are partial.
- State $s_t = f(o_1,...,o_t)$ is modeled by a recurrent neural network (LSTM).
- One network to output action/state functions Q(a|s), Q(s) and policy function π(a|s).
- Task is to collect apples (+1 reward) and escape (+10 reward).

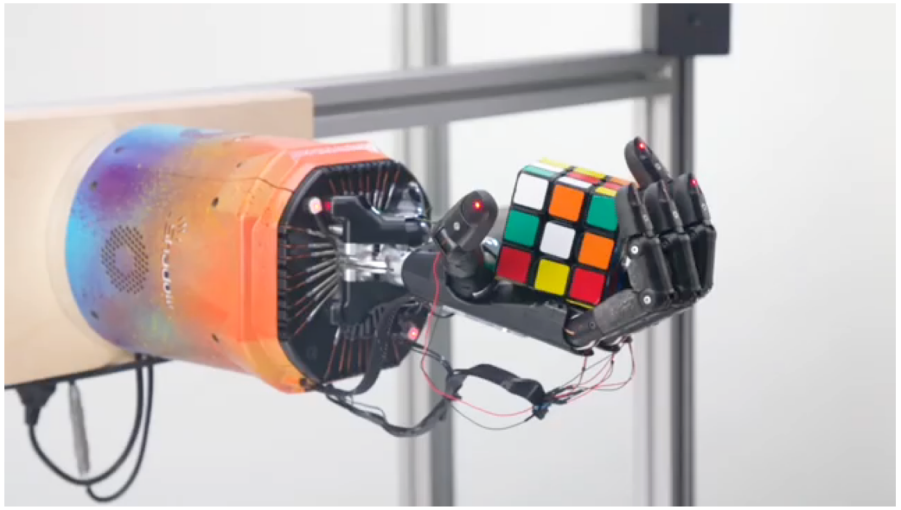
A3C in Labyrinth



Applications



Learning locomotion
Humanoid and spider



Learning human-like robot hand gesture
Application to Rubik's Cube

Outline

- RL and Deep Learning
- Agent, Environment and MDP
- Policy, Value Function and Model
- Optimal Value Function and Policy
- Deep Q-Learning (DQN)
- Policy Networks
- Actor-Critic Algorithm
- **RL and Supervised Learning**
- Learning and Planning

Supervised Learning

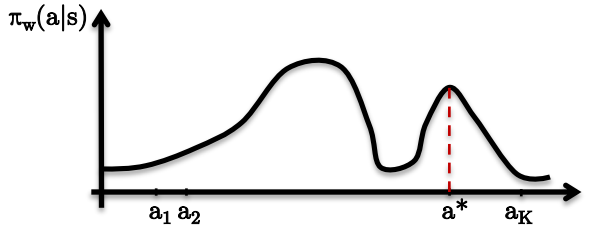
- SL is learning with a teacher who tells the best action to take given any state of the environment.
- Training set for SL :
  - $(s_t, a_t)_{t=1:T}$ where a_t is the label/target action that optimizes the return.
    - By analogy to computer vision :
      - s = image
      - a = class
- Likelihood probability of class a given image s :
  - $\pi(a|s)$
  - It corresponds to the policy function π in RL.

Supervised Learning

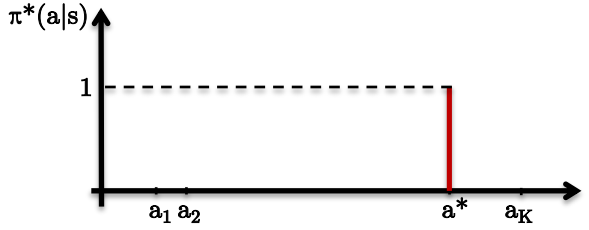
- Loss function :
  - $\min_w L_\pi(W) = \Sigma_t$ cross_entropy $(\pi_w(a_t|s_t) , \pi^*(a_t|s_t))$
    $= - \Sigma_t \log \pi_w(a_t|s_t) \cdot \pi^*(a_t|s_t)$
  - $\max_w L_\pi(W) = - \min_w L_\pi(W)$
    $= \Sigma_t \log \pi_w(a_t|s_t) \cdot \pi^*(a_t|s_t)$

- Gradient of loss function :
  - $\nabla_w L_\pi(W) = \nabla_w (\Sigma_t \log \pi_w(a_t|s_t) \cdot \pi^*(a_t|s_t))$
    $= \Sigma_t \nabla_w \log \pi_w(a_t|s_t) \cdot \pi^*(a_t|s_t)$

In SL, we know the best action to take to optimize the reward function (i.e. the cross entropy loss).



Predicted probability



Target probability

Reinforcement Learning

- RL has no teacher (no label) to tell the best action to take.
- RL is a trial-and-error learning paradigm.
- RL draws sequence of actions that provide reward.
  - Depending on the reward, the actions are encouraged and discouraged.
- Training set :
  - A batch of episodes drawn from probabilities π and P :
    - $e_T = s_1, a_1 \sim \pi(a|s_1), s_2 \sim P(s|s_1,a_1), r_1, a_2 \sim \pi(a|s_2), s_3 \sim P(s|s_2,a_2), r_2,$ $\ldots s_T \sim P(s|s_{T-1},a_{T-1}), a_T \sim \pi(a|s_T), s_{T+1} \sim P(s|s_T,a_T), r_T$
- For each state and action, a reward r_t is arbitrary defined :
  - Game Go : r_t=0,1,-1 (1 for won game, -1 for lost game, 0 otherwise).
  - Robot : 1 if it does not fall, -1 if it falls.
    - Reward can be highly sparse, unlike SL.

RL and SL Similarities

- Gradient of loss function :
  - Reinforcement learning :

$$\begin{aligned} \nabla_w R(\pi_w) &= \mathbb{E}_{a,s} \left(\nabla_w \log \pi_w(a|s) \, . \, Q(a|s) \right) \\ &= \Sigma_t \, \nabla_w \log \pi_w(a_t|s_t) \, . \, \underbrace{Q(a_t|s_t)}_{\text{In RL, we estimate the future reward of taking action } a_t \text{ in state } s_t.} \end{aligned}$$

  - Supervised learning :

$$\begin{aligned} \nabla_w L_\pi(w) &= \nabla_w \left(\Sigma_t \log \pi_w(a_t|s_t) \, . \, \pi^*(a_t|s_t) \right) \\ &= \Sigma_t \, \nabla_w \log \pi_w(a_t|s_t) \, . \, \underbrace{\pi^*(a_t|s_t)}_{\text{In SL, we know the best action } a_t \text{ to take in state } s_t.} \end{aligned}$$

Outline

- RL and Deep Learning
- Agent, Environment and MDP
- Policy, Value Function and Model
- Optimal Value Function and Policy
- Deep Q-Learning (DQN)
- Policy Networks
- Actor-Critic Algorithm
- RL and Supervised Learning
- **Learning and Planning**

Learning and Planning

- RL :
  - Agent interacts with the environment to understand it and improve its policy to get the best possible reward from this environment.
  - Agent learns implicitly the environment.
- Planning :
  - We assume the environment is known (or it was learned).
  - The agent can inquiry multiple times the simulated environment (no real action taken in the real environment), and decides what to do from these inquiries.
    - The agent is "thinking" what to do before acting.
- RL and planning are usually done together.

Learning and Planning

- If the environment is high-dimensional :
  - Curse of dimensionality :
    - The space is too large to simultaneously explore and exploit.
- How to balance the trade-off between exploration and exploitation ?
  - Exploration :
    - Give up some rewards to get more information about the environment, which can eventually lead to higher rewards.
  - Exploitation :
    - Give up potential more rewards by only using the known environment to get good rewards.

Learning and Planning

- Examples :
  - Game of Go :
    - Play the move that is known to be the best.
    - Try a new strategy.
  - Online advertisement :
    - Show the consumer the most successful ads.
    - Try a new ad.

Learning and Planning

- Model-based RL :
  - Learn a model of a proxy of the environment.
  - This model can be used to interact virtually with the environment, just as we would interact with the real environment.
  - A model is essential to do a look-ahead search by querying the environment with some actions and observe the consequences/rewards of this action.
  - This allows to think ahead and make better actions in high-dimensional spaces.

- Material :
  - David Silver :
    - Tutorial on Deep Reinforcement Learning, ICML'16
    - UCL Course on Reinforcement Learning, 2015
  - Pieter Abbeel :
    - Talk at Simons Institute for the Theory of Computing, Berkeley, 2017
  - Mario Martin :
    - Slides on Policy Search: Actor-Critic and Gradient Policy search, 2019

Questions?