UiO Department of Informatics
University of Oslo

INF 5860 Machine learning for image classification

Lecture 14: Reinforcement learning

May 9 , 2018

Outline

- Motivation
- Introduction to reinforcement learning (RL)
- Value function based methods (Q-learning)
- Policy based methods (policy gradients)
- Value function and policy based methods (Actor-Critic)
- Miscellaneous

About today

- Introduction to main concepts and terminology of reinforcement learning
- The goal is for you to be familiar with policy gradients and Q-learning

Readings

- **Video**:
  – CS231n: Lecture 14 | Deep Reinforcement Learning: https://www.youtube.com/watch?v=lvoHnicueoE&index=14&list=PLC1qU-LWwrF64f4QKQT-Vg5Wr4qEE1Zxk&t=3s

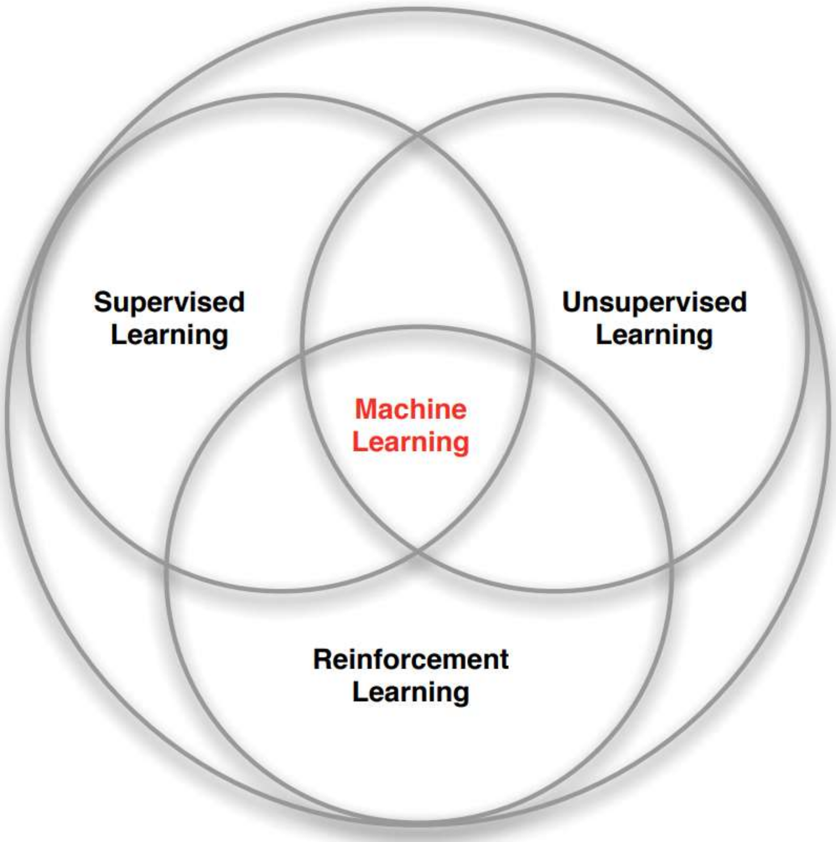
- **Text**:
  – Karpathy blog: (Reinforcement learning/Policy learning) http://karpathy.github.io/2016/05/31/rl/

- **Optional**:
  – RL Course by David Silver: https://www.youtube.com/watch?v=2pWv7GOvuf0&list=PL7-jPKtc4r78-wCZcQn5IqyuWhBZ8fOxT&index=0

Progress

- **Motivation**
- Introduction to reinforcement learning (RL)
- Value function based methods (Q-learning)
- Policy based methods (policy gradients)
- Value function and policy based methods (Actor-Critic)
- Miscellaneous

Branches of Machine Learning



Supervised learning

- Given a training set with input x and desired output y:
  - $\Omega_{train} = \{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(m)}, y^{(m)})\}$

- The goal is to create a function f that “approximates” this mapping:
  - $f(x) \approx y, \quad \forall(x,y) \in \Omega_{train}$

- Hope that this generalizes well to unseen examples:
  - $f(x) = \hat{y} \approx y, \quad \forall(x,y) \in \Omega_{test}$

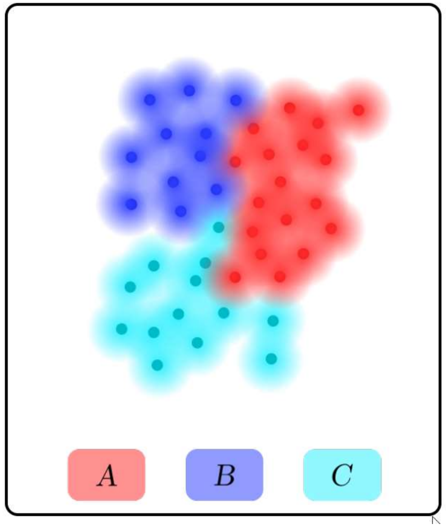
- Examples:
  - Classification, regression, object detection,
  - Segmentation, image captioning.

Unsupervised learning

- Our training set consists of input x only:
  - $\Omega_{train} = \{x^{(1)}, x^{(2)}, \dots, x^{(m)}\}$

- We do not have any labeled data. Our goal is to find an underlaying structure of the data.

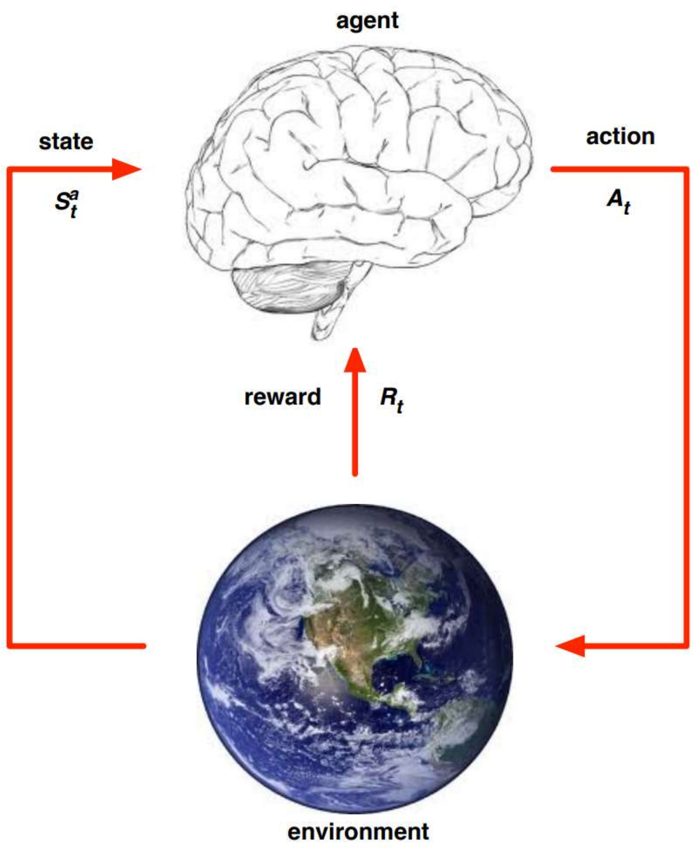
- Examples:
  - Data clustering
  - Anomality detection
  - Signal generation
  - Signal compression



Variational autoencoder (latent space z)

Reinforcement learning

- Reinforcement Learning ~ Science of decision making
- In RL an agent learns from the experiences it gains by interacting with the environment.
- The goal is to maximize an accumulated reward given by the environment.
- An agent interacts with the environment via states, actions and rewards.

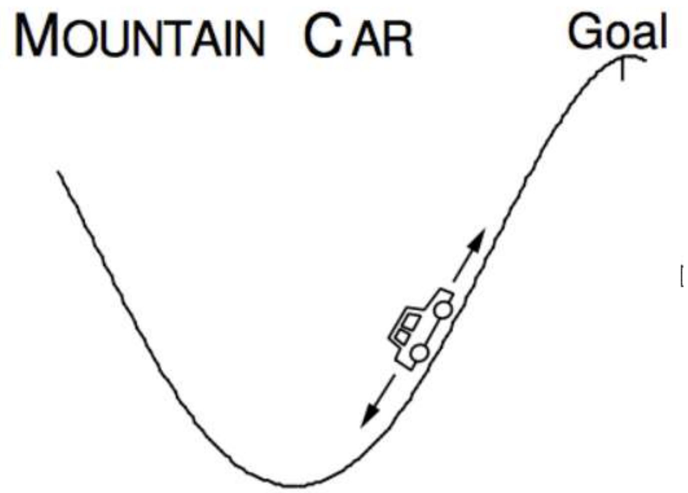


RL Course by David Silver

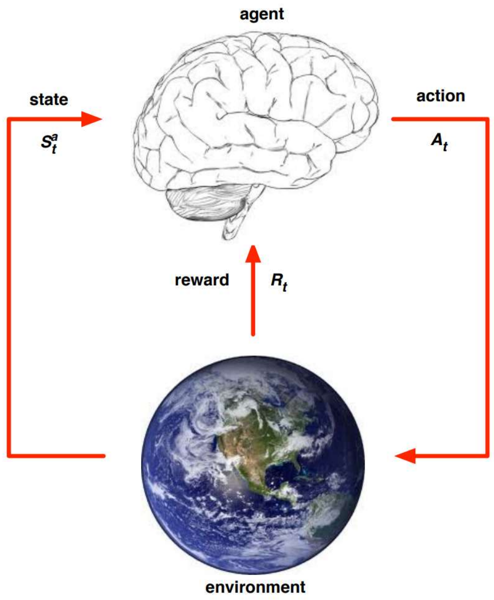
Reinforcement learning

- What makes reinforcement learning different from other machine learning paradigms?
  - There is no supervisor, only a reward signal
  - Feedback is delayed, not instantaneous
  - Time really matters (sequential, non i.i.d data)
  - Agent’s actions affect the subsequent data it receives

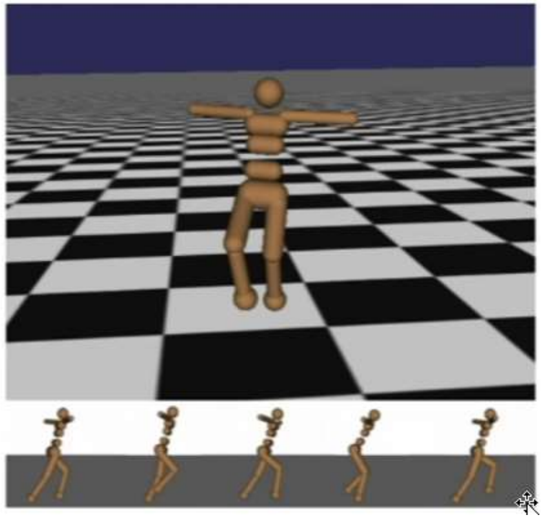
Mountain Car

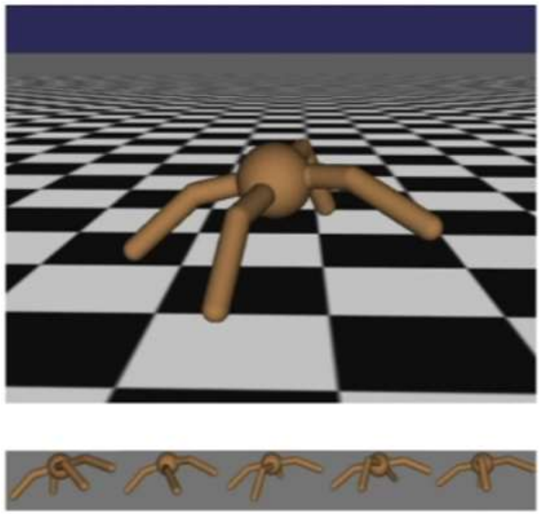


- **Objective:**
  – Get to the goal
- **State variables:**
  – Position and velocity
- **Actions:**
  – Motor: Left, Neutral, right
- **Reward:**
  – (-1) for each time step



Robots



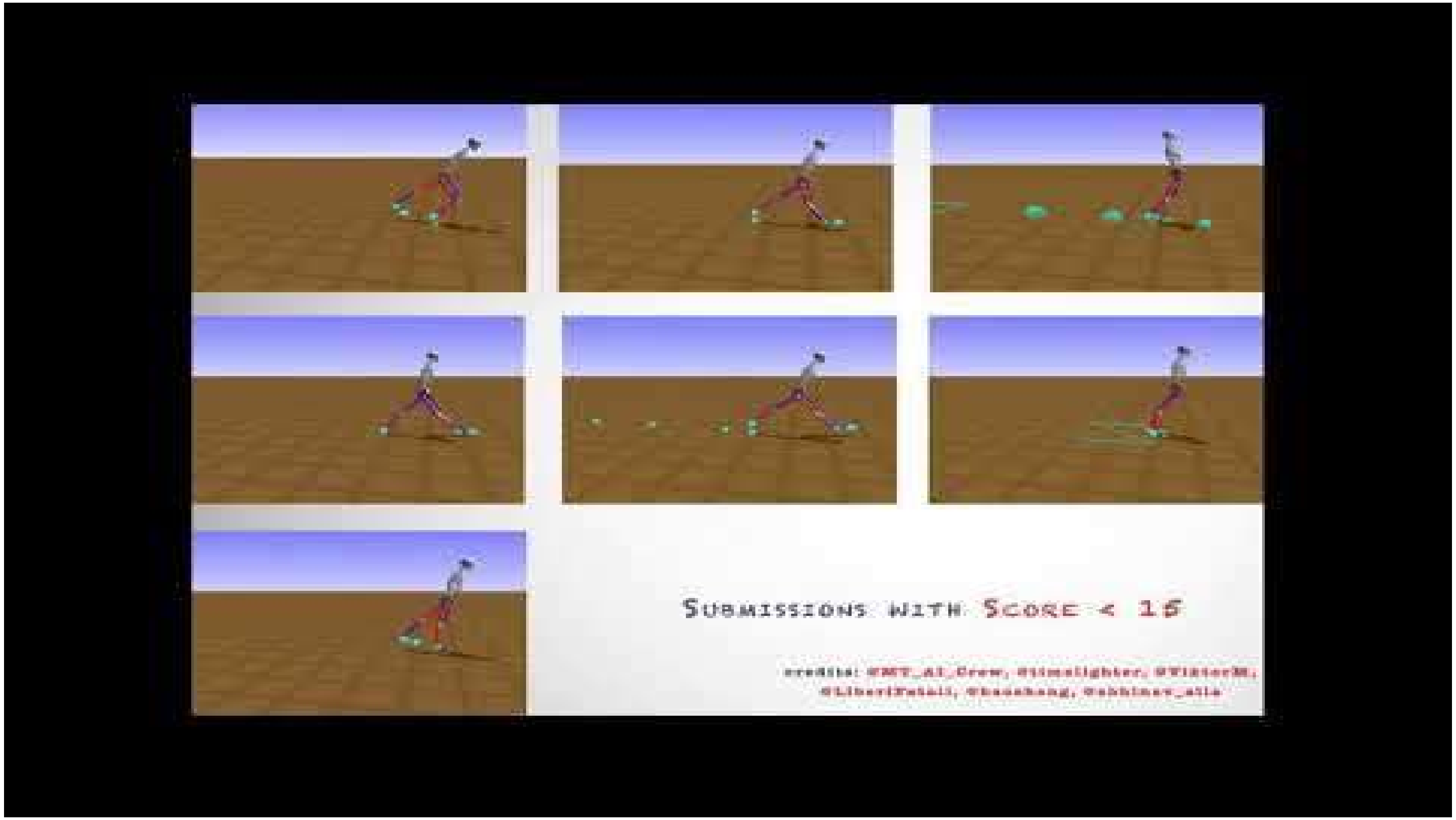


Objective:
Get the robot to move forward

State variables:
Angle and positions of the joints

Actions:
Torques applied on joints

Reward:
(+1) at each time step upright +
forward movement



- https://www.youtube.com/watch?v=rhNxt0VccsE

Atari games

Objective:

Complete the game with the highest score

State variables:

Raw pixel inputs of the game state

Actions:

Game controls, e.g. left, right, up, down, shoot.

Reward:

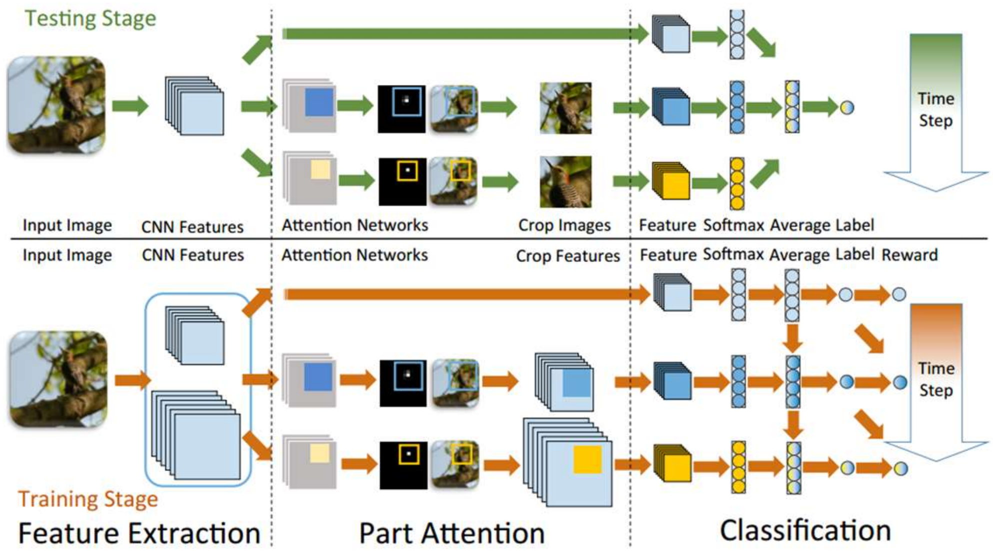
Score increases/decreases at each time step

Distinguishing images with small differences

- You have high resolution images
- You separate classes based on small, but characteristic differences
  - Birds
  - Tumors
  - Brands

Distinguishing images with small differences

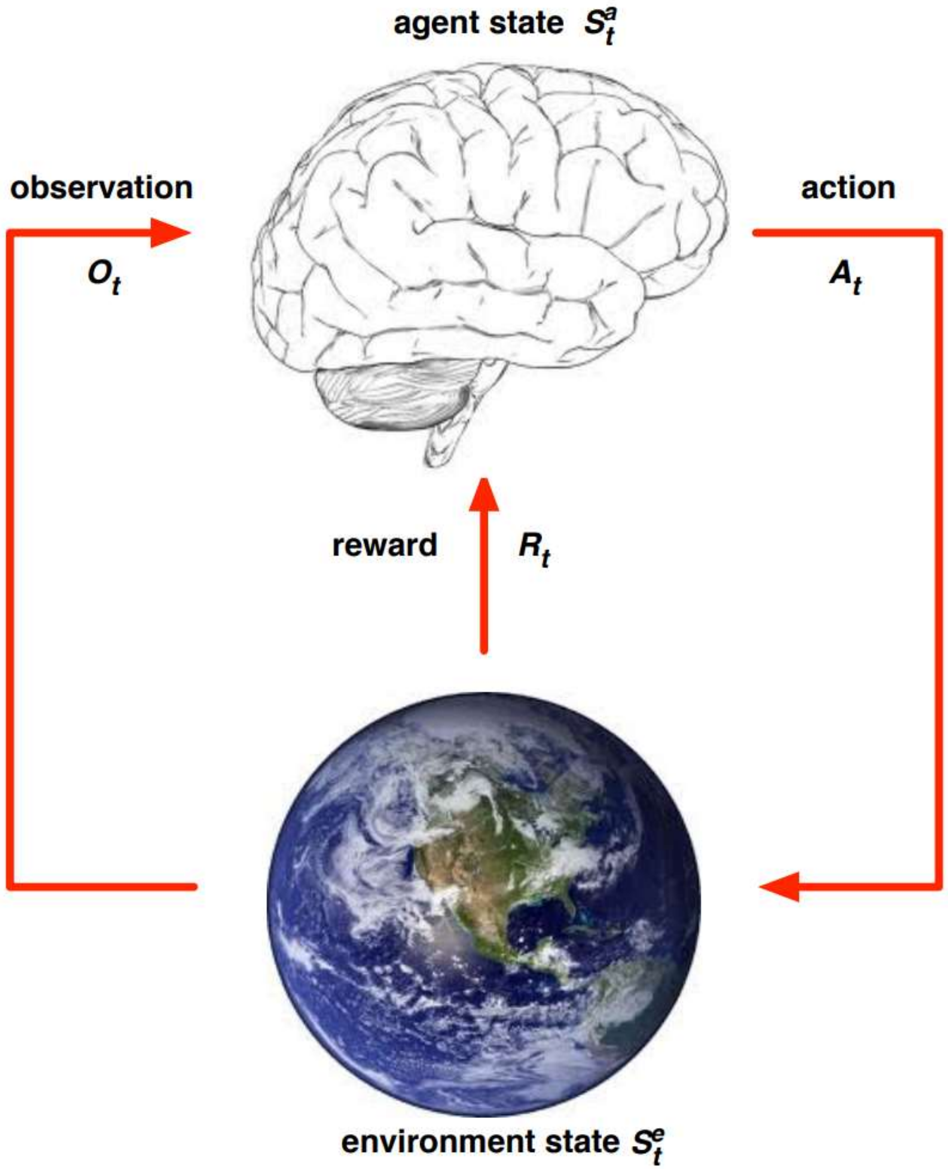
- Pre-trained CNN features
- Attention network output confidence map
- Spatial softmax for finding probabilities of locations
- Crop and resize image features



[Fully Convolutional Attention Networks for Fine-Grained Recognition]

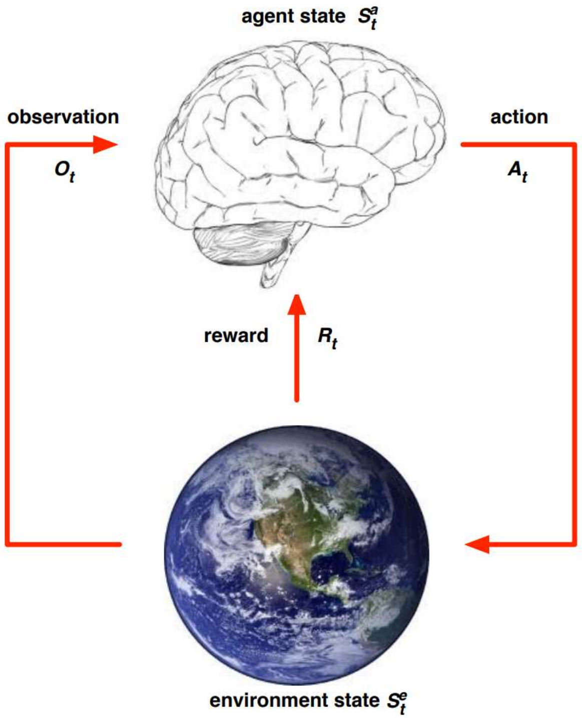
Progress

- Motivation
- **Introduction to reinforcement learning (RL)**
- Value function based methods (Q-learning)
- Policy based methods (policy gradients)
- Value function and policy based methods (Actor-Critic)
- Miscellaneous


agent state S_t^a
observation
O_t
action
A_t
reward
R_t
environment state S_t^e

History (trajectory) and State

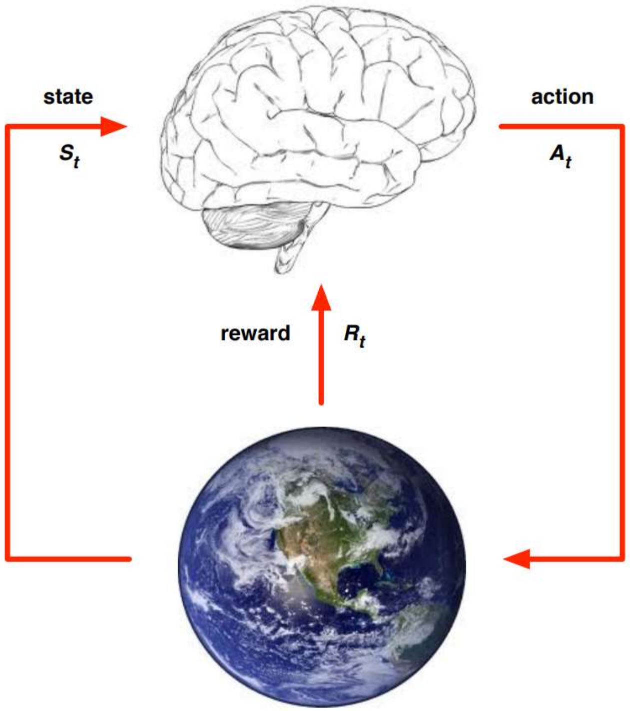
- **History / trajectory :**
  - $H_t = \tau_t = O_1, A_1, R_1, O_2, A_2, R_2, \ldots, O_t, A_t, R_t$
- **Full observatory:**
  - Agent direct observe the environment state.
  - $O_t = S_t^e = S_t^a$
- **State:**
  - The state is a summary (of the actions and observations) that determines what happens next given an action.
  - $S_t = f(H_t)$
- **Partially observability:**
  - The agent indirectly observes the environment.
  - Robot with a camera



Markov Property

- **Definition**:
  - A state S_t is Markov if and only if:
    $\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1} \mid S_1, S_2, \ldots, S_t]$

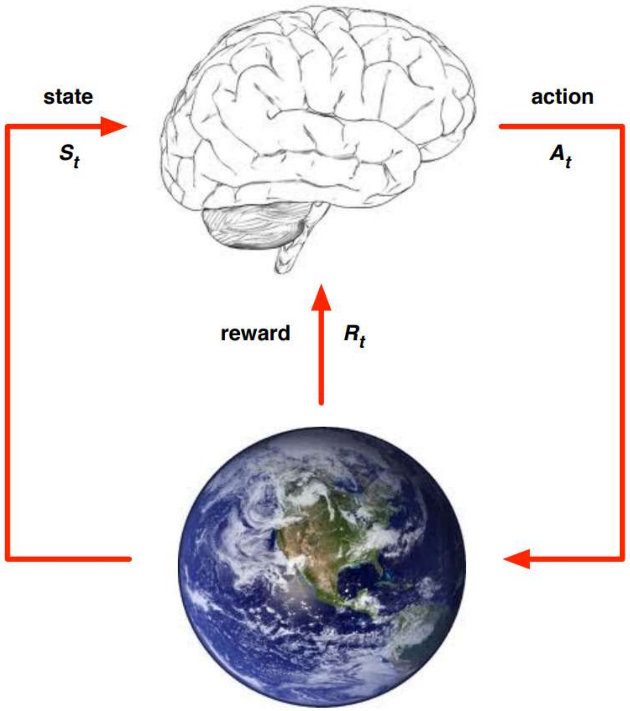
- The state capture all relevant information from the history
- The state is sufficient to describe the statistics of the future.



Policy

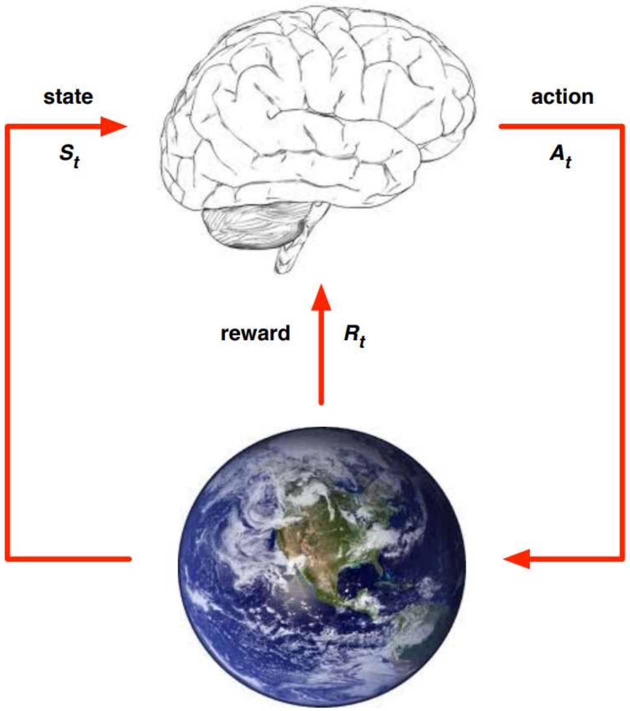
- The agent's policy defines it's behavior.

- A policy, π, is a map from state to actions
  - **Deterministic** policy: $\pi(s_t)$
  - **Stochastic** policy: $\pi(a_t|s_t) = \mathbb{P}(A_t = a_t|S_t = s_t)$



Reward and Return

- The **reward**, R_t, is a scalar value the agent receives for each step t.
- The **return,** G_t, is the total discounted accumulated reward form a given time-step t.
  - $G_t = R_t + \gamma R_{t+1} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$
- **Discount factor**:
  - We can apply a discord factor, $\gamma \in [0,1]$, to weight how we evaluate return.
- The agent's goal is to maximize the **return**



Markov Decision Process (MDP)

- The mathematical formulation of the reinforcement learning (RL) problem.

- A **Markov Decision Process** is a tuple, $\mathcal{M} = \langle S, A, P, R, \gamma \rangle$, where every state has the Markov property.

S: A finite set of states

A: A finite set of *actions*

P: The transition probability matrix

$$P^a_{s_t s_{t+1}} = \mathbb{P}[S_{t+1} = s_{t+1} \mid S_t = s_t, A_t = a_t]$$

R: Reward function:

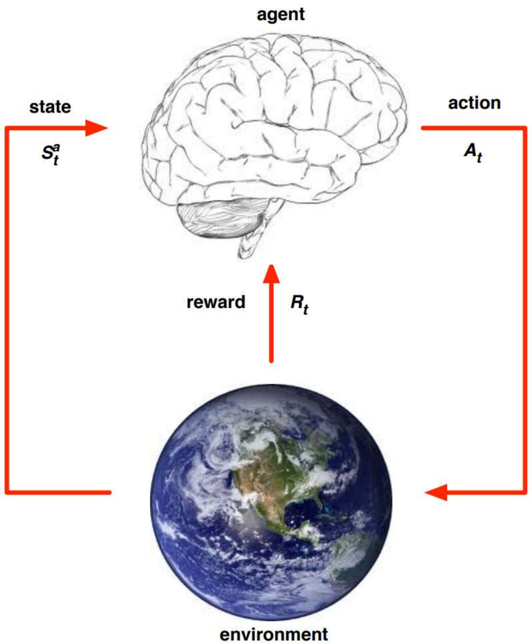
$$R^a_s = \mathbb{E}[S_t = s_t,\ A_t = a_t]$$

γ: is a discount factor $\gamma \in [0,1]$

Markov Decision Process (timeline)

- The environment samples an initial state, s_0, for time-step t=0.

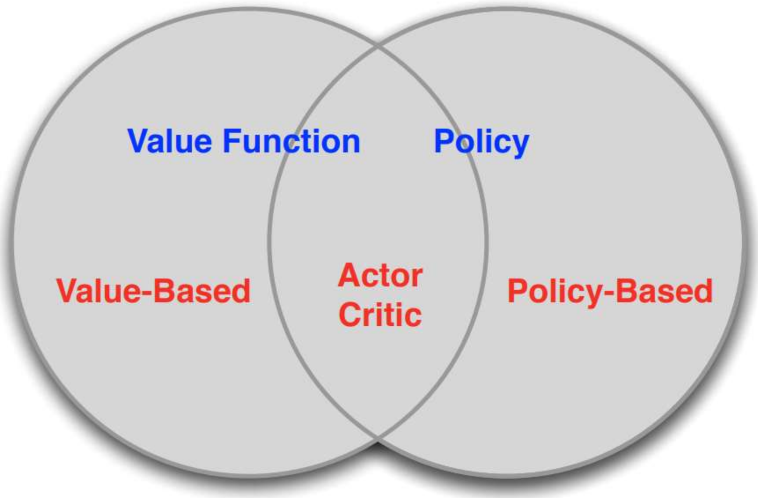
- For time-step, t, until termination:
  - Agent selects an action given a policy: $\mathrm{a}_\mathrm{t} = \pi(a_t|s_t) = \mathbb{P}(A_t = a_t|S_t = s_t)$
  - Environment samples a reward: $r_t = \mathbb{P}[R_t = r_t \mid S_t = s_t, A_t = a_t]$
  - Environment samples next state: $s_{t+1} = \mathbb{P}[S_{t+1} = s_{t+1} \mid S_t = s_t, A_t = a_t]$
  - Agent receives the next state, s_{t+1}, and the reward, r_t.



Objective

- The objective in reinforcement learning (RL) is to find the optimal policy, π_*, which maximize the expected accumulated reward.

- Agent's taxonomy to find the optimal policy in reinforcement learning



Progress

- Motivation
- Introduction to reinforcement learning (RL)
- **Value function based methods (Q-learning)**
- Policy based methods (policy gradients)
- Value function and policy based methods (Actor-Critic)
- Miscellaneous

Objective

- Our goal is it find the policy which maximize the accumulated reward:
  $G_t = R_t + \gamma R_{t+1} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$

- Due to the randomness of the transition probability and the reward function, we use the expected value in the definition of the optimal policy.

$$\pi_* = \arg\max_{\pi} \mathbb{E}\,[G_t]$$

State-value function and action-value function

- While we follow our policy, we would like to know if we are not a good or bad state/position. Imagine trajectory: $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$

- Definition: a **state-*value function***, $v_\pi(s)$, of an MDP is the expected return starting from state, s, and then following the policy π. In general, how good is it to be in this state.
  $v_\pi(s) = \mathbb{E}_\pi[\, G_t \mid \mathrm{S_t} = \mathrm{s}]$

- Definition: an ***action-value (q-value) function***, $q_\pi(s, a)$, is the expected return starting from state, s, taking action, a, and following policy, π. In general, how good it is to take this action.
  $q_\pi(s, a) = \mathbb{E}_\pi[\, G_t \mid A_t = a, S_t = s]$

State-value function and action-value function

- **Define**: $\pi \geq \pi' \quad if \quad v_\pi(s) \geq v_{\pi'}(s), \quad \forall s$

- **Definition:** The optimal state-value function $v_*(s)$, is the maximum value function over all policies:
- $v_*(s) = \max_\pi v_\pi(s)$

- **Definition:** The optimal action-value function $q_*(s,a)$, is the maximum action-value function over all policies:
- $q_*(s,a) = \max_\pi q_\pi(s,a)$

- **Note**: If we knew $q_*(s,a)$ the RL problem is solved.

Bellman (expectation) equation

- The Bellman equation is a recursive equation which can decompose the value function into two part:
  - Immediate reward, R_t
  - Discounted value of successor state, $\gamma v(S_{t+1})$

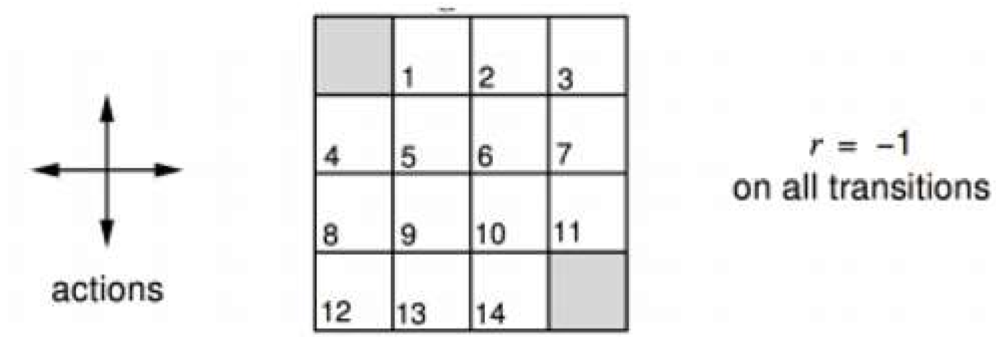
$$
\begin{aligned}
v(s) &= \mathbb{E}_\pi \left[G_t \mid \mathrm{S_t} = \mathrm{s} \right] \\
&= \mathbb{E}_\pi \left[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \cdots \mid \mathrm{S_t} = \mathrm{s} \right] \\
&= \mathbb{E}_\pi \left[R_t + \gamma (R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \cdots) \mid \mathrm{S_t} = \mathrm{s} \right] \\
&= \mathbb{E}_\pi \left[R_t + \gamma G_{t+1} \mid \mathrm{S_t} = \mathrm{s} \right] \\
&= \mathbb{E}_\pi \left[R_t + \gamma v(S_{t+1}) \mid \mathrm{S_t} = \mathrm{s} \right]
\end{aligned}
$$

How to find the best policy?

- We will go though a simple example, Gridworld, to show how the Bellman equation can be used iteratively to evaluate a policy, π. The Goal is to give an intuition of how the **Bellman equation** is used.

$$v(s) = \mathbb{E}_{\pi}\left[R_t + \gamma v(S_{t+1}) \mid S_t = s\right]$$

Evaluating a Random Policy in Gridworld using the Bellman eq.



- Terminal states are shown as shaded
- Actions leading out of the grid leave state unchanged
- Reward is (-1) until a terminal state is reached
- Agent follows uniform random policy

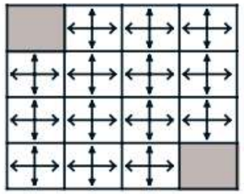
$$\pi(n|\cdot) = \pi(s|\cdot) = \pi(e|\cdot) = \pi(w|\cdot) = 0.25$$

v_k for the Random Policy

Greedy Policy w.r.t. v_k

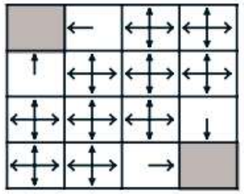
$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0



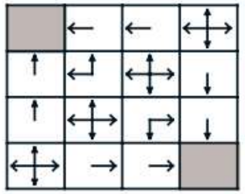
$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0



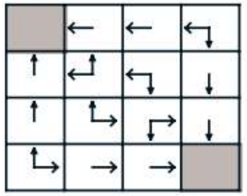
$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0



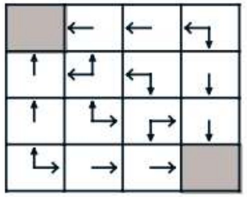
$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0



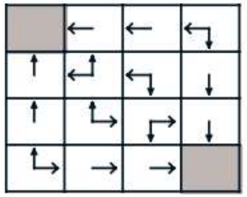
$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0



$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



$$v_{k+1}(s) = \mathbb{E}_\pi \left[R_t + \gamma v_k(S_{t+1}) \mid S_t = s\right]$$

Policy evaluation

- We can **evaluate** the policy π, by iteratively update the state-value function.

$$v_\pi(s) = \mathbb{E}_\pi\left[R_t + \gamma v(S_{t+1}) \mid S_t = s\right]$$

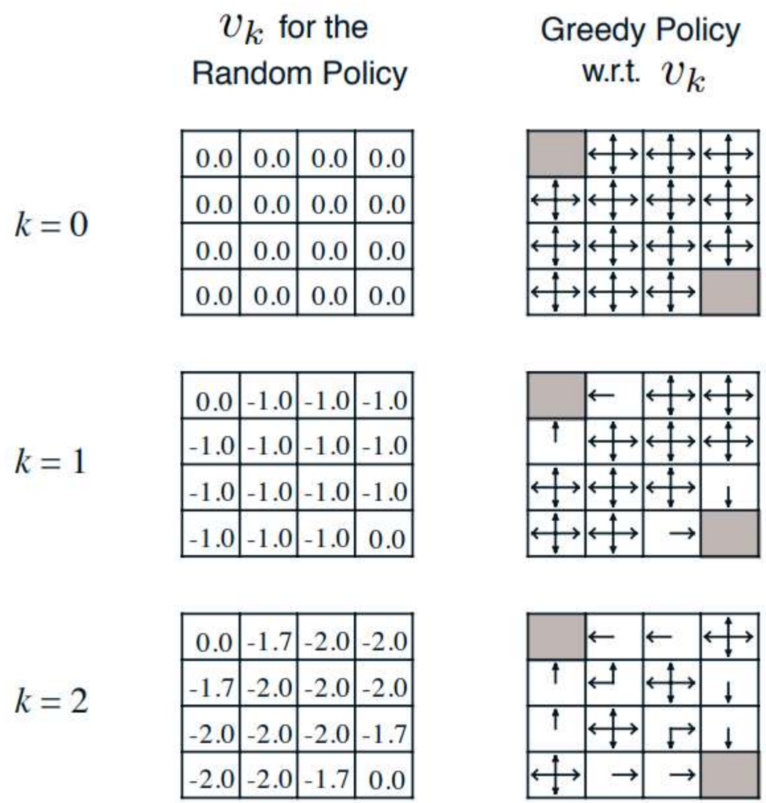
- We can **improve** the policy by acting greedily with respect to v_π.

$$\pi' = greedy(v_\pi)$$

- In our example, we found the optimal policy, $\pi' = \pi^*$, after one iteration only.

- In general, iterating between policy evaluation and policy improvement is required before finding the optimal policy

- This was an example with a known MDP, we knew the rewards and the transitions probabilities.



$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

Bellman (optimality) equation

- Lets define the optimal Q-value (*action-value*) function, Q_*, to be the maximum expected reward given an state, action pair.

$$Q_*(s_t, a_t) = \max_{\pi} \mathbb{E}_{\pi}[\, G_t \mid A_t = a_t, S_t = s_t]$$

- The optimal Q-value function, Q_*, satisfy the following form of the bellman equation:

$$Q_*(s_t, a_t) = \mathbb{E}\left[\, R_t + \gamma \max_{a_{t+1}} Q_*(s_{t+1}, a_{t+1}) \mid A_t = a_t, S_t = s_t \right]$$

- **Note**: The optimal policy, π_*, is achieved by taking the action with the highest Q-value.
- **Note**: We still need the expectation, as the randomness of the environment is unknown.

Solving for the optimal policy

The goal is to find a Q-value function which satisfy the Bellman (optimality) equation. An algorithm, **value iteration**, can be used to iteratively update our Q-value function.

$$Q_i(s_t, a_t) = \mathbb{E}\left[R_t + \gamma \max_{a_{t+1}} Q_{i-1}(s_{t+1}, a_{t+1}) \mid A_t = a_t, S_t = s_t \right]$$

- **Notation**: i, is the iteration update step, t, is the sequential time-step in an episode.

- The Q-value, Q_i, will converge to Q_* under some mathematical conditions.

- While solving for the optimal Q-value function, we normally encounter two challenges:
  - The "max" property while sampling new episodes can lead to suboptimal policy.
  - The state-action space is too large to store.

Exploration vs Exploitation

- "The "max" property while sampling new episodes can lead to suboptimal policy"

- **Exploitation:**
  - By selecting the action with the highest q-value while sampling new episodes, we can refine our policy efficiently from an already promising region in the state action space.

- **Exploration:**
  - To find a new and maybe more promising region within the state action space, we do not want to limit our search in the state action space.
  - We introduce a randomness while sampling new episodes.
  - With a probability of ϵ lets choose a random action:

$$\pi(a|s) = \begin{cases} a_* = \underset{a \in A}{\operatorname{argmax}}\, Q(s,a), & with\ probabillity\ \ 1-\epsilon \\ random\ action, & with\ \ probabillity\ \ \epsilon \end{cases}$$

Function approximation

- In the Gridworld example, we stored the state-values for each state. What if the state-action space is too large to be stored e.g. continuous?

- We approximate the Q-value using a parameterized function e.g. neural network.

$$\hat{Q}(s, a, \theta) \approx Q(s, a)$$

- We want the function to generalize:
  - Similar states should get similar action-values, $\hat{Q}(s, a, \theta)$ can also generalize to unseen states. A table version would just require to much data.

- In supervised learning:
  - Building a function approximation vs memorizing all images (table).

Solving for the optimal policy: Q-learning

- **Goal**: Find a Q-function satisfying the Bellman (optimality) equation.
- **Idea**: The Q-value at the last time step is bounded by the true Q-value, the correctness of the Q-value estimates increase with time-steps.
- **Init:** Initialize the weights in the neural network e.g. randomly.

$$Q_*(s_t, a_t, \theta_i) = \mathbb{E}\left[R_t + \gamma \max_{a_{t+1}} Q_*(s_{t+1}, a_{t+1}, \theta_{i-1}) \mid A_t = a_t, S_t = s_t \right]$$

- Reference:

$$y_i = \mathbb{E}\left[R_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}, \theta_{i-1}) \mid A_t = a_t, S_t = s_t \right]$$

- Loss:

$$L_i(\theta_i) = \mathbb{E}_{s_t, s_{t+1,} a_t, r_t \sim D_i}\left[\left(y_i - Q(s_t, a_t, \theta_i) \right)^2 \right]$$

D_i is your dataset with state action pairs $s_t, s_{t+1,} a_t, r_t$

Solving for the optimal policy: Q-learning

- Loss:

$$L_i(\theta_i) = \mathbb{E}_{s_t, s_{t+1}, a_t, r_t \sim D_i}\left[\left(y_i - Q(s_t, a_t, \theta_i)\right)^2\right]$$

- Compute gradients:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s_t, s_{t+1}, a_t, r_t \sim D_i}\left[2\left(y_i - Q(s_t, a_t, \theta_i)\right) \cdot \nabla_{\theta_i} Q(s_t, a_t, \theta_i)\right]$$

- Update weights θ:

$$\theta_i = \theta_{i-1} - \alpha \nabla_{\theta_i} L_i(\theta_i)$$

Example: Deep Q-learning (Atari Games)

Objective:
Complete the game with the highest score

State variables:
Raw pixel inputs of the game state

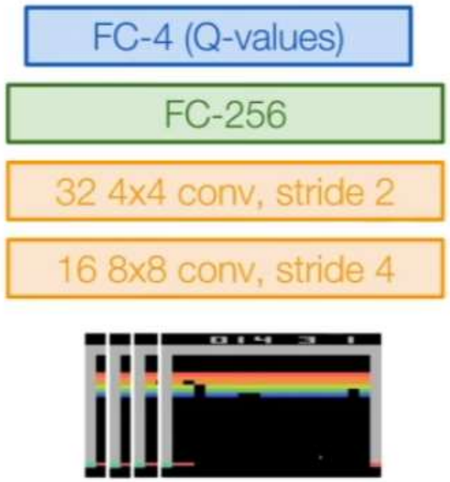
Actions:
Game controls, e.g. left, right, up, down, shoot.

Reward:
Score increases/decreases at each time step

Deep Q-learning (Atari Games)

- Example taken from: [Mnih er al. NIPS Workshop 2013; Nature 2015]

- Q-network architecture:
  - FC-4 outputs Q values for all actions
  - A state, s_t, is a set pixels from stacked frames



Current state s_t: 84x84x4 stack of last 4 frames
(after RGB->grayscale conversion, downsampling, and cropping)

Experience replay

- Loss:

$$L_i(\theta_i) = \mathbb{E}_{s_t, s_{t+1}, a_t, r_t \sim D_i}\left[\left(y_i - Q(s_t, a_t, \theta_i)\right)^2\right]$$

- The loss function is defined by two state action pairs, $\langle s_t, r_t, a_t, s_{t+1}\rangle$. We can store a **replay memory** table form the episodes played. The table is updated when new episodes are available.

- Normally, state action pairs from the same episode are used to update the network. However, we can select random mini batches for the **replay memory.** This breaks the correlation between the data used to update the network.

- More data efficient as we can reuse the data.

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action a_t in emulator and observe reward r_t and image x_{t+1}
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N

Initialize action-value function Q with random weights

for episode $= 1, M$ **do**

  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

  for $t = 1, T$ **do**

    With probability ϵ select a random action a_t

    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$

    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

  end for

end for

Initialize replay memory, Q-network

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do** ← Play M episodes (full games)
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Initialize state (starting game screen pixels) at the beginning of each episode

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

For each timestep t of the game

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

With small probability, select a random action (explore), otherwise select greedy action from current policy

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Take the action (a_t), and observe the reward r_t and next state s_{t+1}

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$ ← Store transition in replay memory
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

[Mnih et al. NIPS Workshop 2013; Nature 2015]

Putting it together: Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Experience Replay:
Sample a random minibatch of transitions from replay memory and perform a gradient descent step

- https://www.youtube.com/watch?v=V1eYniJ0Rnk

Progress

- Motivation
- Introduction to reinforcement learning (RL)
- Value function based methods (Q-learning)
- **Policy based methods (policy gradients)**
- Value function and policy based methods (Actor-Critic)
- Miscellaneous

Policy based methods

- **Value function based methods**:
  - Learning the expected future reward for a given action.
  - The policy was to act greedily or epsilon-greedily on the estimated values.

- **Policy based methods**:
  - Learning the probability that an action is good directly.

- **Advantage of Policy based methods:**
  - We might need a less complex function for approximating the best action compared to estimate the final reward.
  - Example: Think of Pong

Policy based methods

- **Goal:**
  – The goal is to use experience/samples to try to make a policy better.

- **Idea:**
  – If a trajectory achieves a high reward, the actions were good
  – If a trajectory achieves a low reward, the actions were bad
  – We will use gradients to enforce more of the good actions and less of the bad actions. Hence the method is called Policy Gradients.

Policy Gradients

- Our policy, $\pi_\theta,$ is a parametric function of parameter θ.

- We can define an objective function for a given policy as:

$$\mathcal{J}(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right]$$

- Note:
  - γ is the discord factor
  - r_t is the reward at time-step t.

- Assuming our policy is differentiable we can use gradient ascent to maximum $\mathcal{J}$ w.r.t to θ

REINFORCE algorithm (not curriculum)

- Our environment and sampling of our action is stochastic. Lets define the return as the expected accumulated reward.

$$\begin{aligned}\mathcal{J}(\theta) &= \mathbb{E}_{\tau\sim p(\tau,\theta)}[r(\tau)] \\ &= \int_{\tau} r(\tau)p(\tau,\theta)d\tau\end{aligned}$$

- Note:
  - Trajectory: $\tau_t = s_0, a_0, r_0,\ s_1, a_1, r_1, \dots,\ s_t, a_t, r_t$

- We need the gradient of the objective function to update the parameters, θ.

$$\nabla_\theta\, \mathcal{J}(\theta) = \int_{\tau} r(\tau)\, \nabla_\theta p(\tau,\theta)d\tau$$

REINFORCE algorithm (not curriculum)

$$\nabla_\theta \mathcal{J}(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim p(\tau,\theta)}[r(\tau)]$$
$$= \int_\tau r(\tau) \nabla_\theta p(\tau,\theta) d\tau$$

Intractable! Gradient of an expectation is problematic when p depends on θ

- We can rewrite the equation to become an expectation of an gradient using the following trick:

$$\nabla_\theta p(\tau,\theta) = p(\tau,\theta) \frac{\nabla_\theta p(\tau,\theta)}{p(\tau,\theta)} = p(\tau,\theta) \nabla_\theta \log p(\tau,\theta)$$

$$\nabla_\theta \mathcal{J}(\theta) = \int_\tau p(\tau,\theta)(r(\tau) \nabla_\theta \log p(\tau,\theta)) \, d\tau$$
$$= \mathbb{E}_{\tau \sim p(\tau,\theta)} [r(\tau) \nabla_\theta \log p(\tau,\theta)]$$

REINFORCE algorithm (not curriculum)

$$\nabla_\theta \mathcal{J}(\theta) = \mathbb{E}_{\tau\sim p(\tau,\theta)}\left[r(\tau)\ \nabla_\theta \log p(\tau,\theta)\right]$$

- Expanding the probability of a trajectory based on the term, $\nabla_\theta \log p(\tau,\theta)$:

$$p(\tau,\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t,a_t)\,\pi_\theta(a_t|s_t)$$

$$\log p(\tau,\theta) = \sum_{t\geq 0}(\log p(s_{t+1}|s_t,a_t) + \log \pi_\theta(a_t|s_t))$$

$$\nabla_\theta \log p(\tau,\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

- We can sample trajectories to get estimates of the gradient.

$$\nabla_\theta \mathcal{J}(\theta) \approx \sum_{t\geq 0} r(\tau)\,\nabla_\theta \log \pi_\theta(a_t|s_t)$$

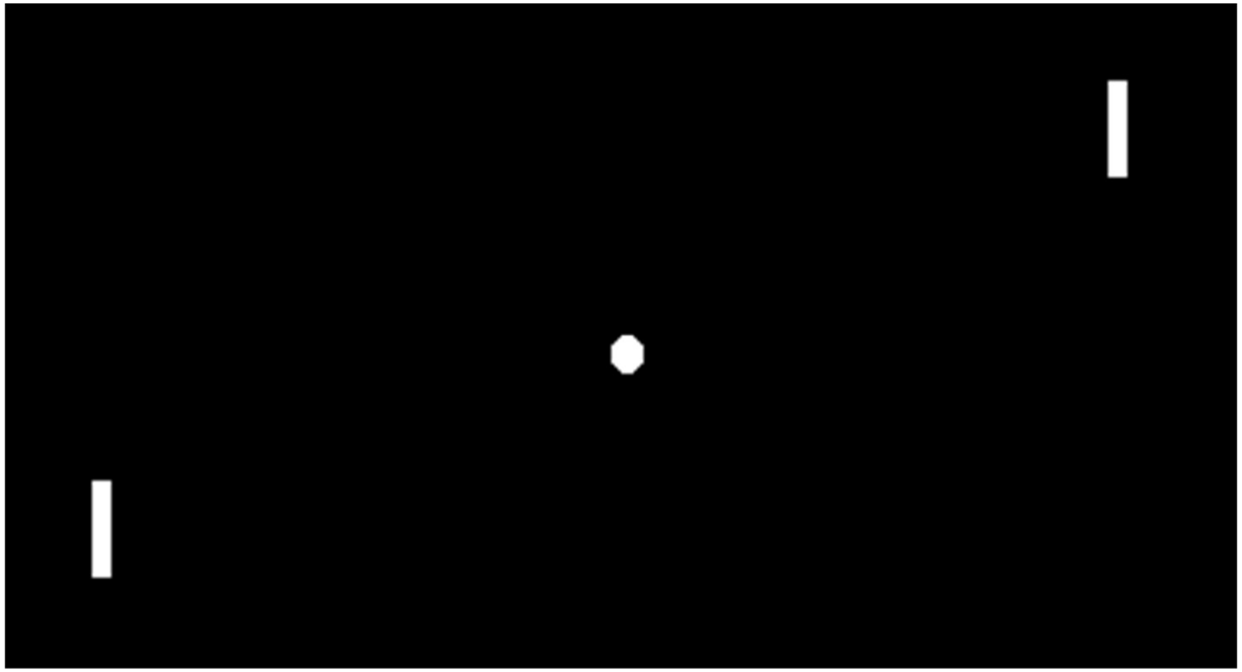
REINFORCE algorithm (Pseudocode) (not curriculum)

- Update parameters by stochastic gradient **ascent**
- Using r_t as the return at time-step t.

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)\, r_t$$

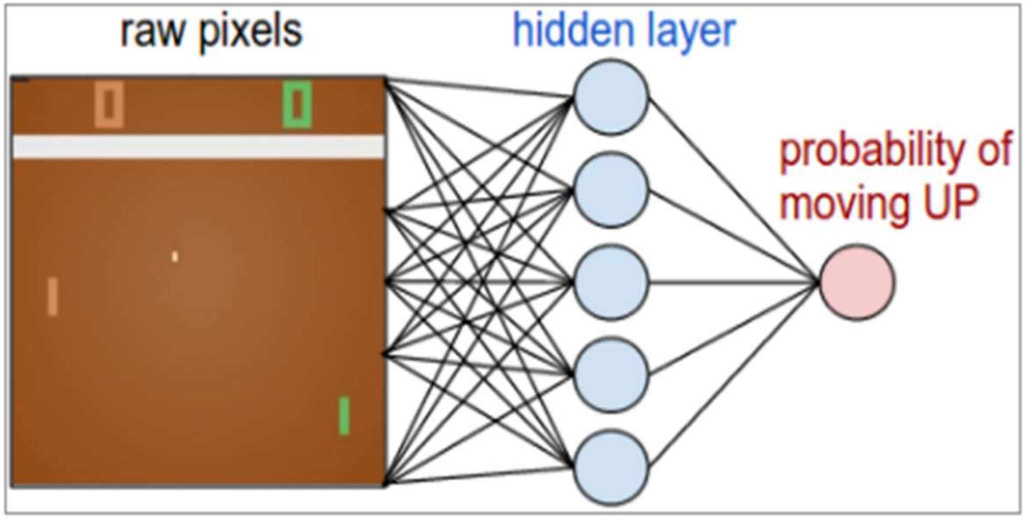
function REINFORCE
    Initialize θ arbitrarily
    for each episode $\{\tau_t = s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_t, a_t, r_t\} \sim \pi_\theta$ **do**
        for $t = 1$ to $T - 1$ **do**
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t)\, r_t$
        end
    end
    return θ
end function

Game of Pong



Policy learning: Pong

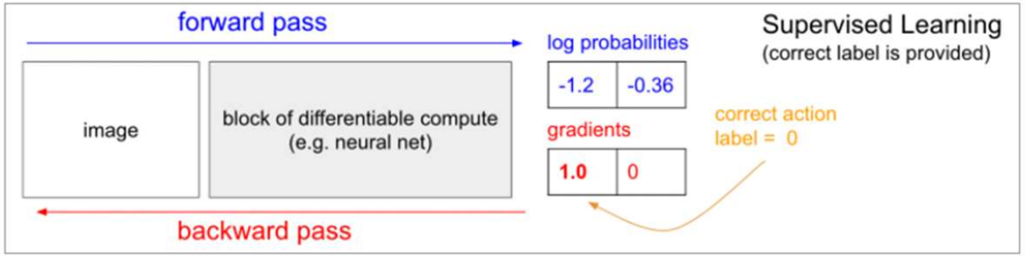
- Policy learning
  - We take input images as states
  - Output probability of being good action
  - Choose an action
  - Observe: reward (/punishment)
  - Improve



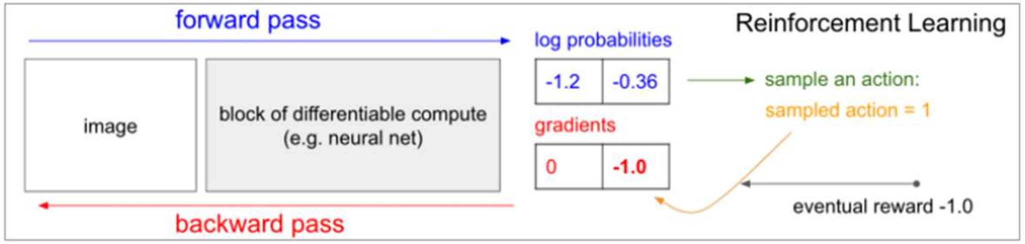
- The game:
  - Actions:
    - Up
    - Down
  - Reward:
    - Winning $= +1$
    - Losing $= -1$

Supervised learning vs Reinforcement learning

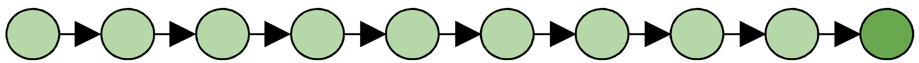
- Imagine you play pong and the agent predicts:
  - Up → $\log p = -1.2$ (30%)
  - Down → $\log p = -0.36$ (70%)
  - correct action is "Up"



- **Supervised learning**:
  - You choose the output with the highest probability
  - You get an immediate reward

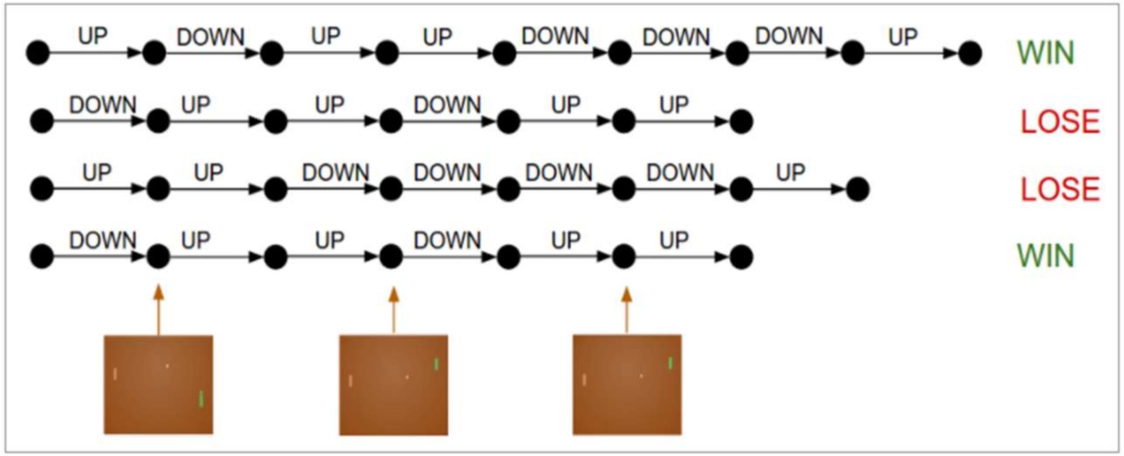


- **Policy learning:**
  - You sample an action with given the probability distribution
  - Wait until you get an reward to backprop (May be many steps)



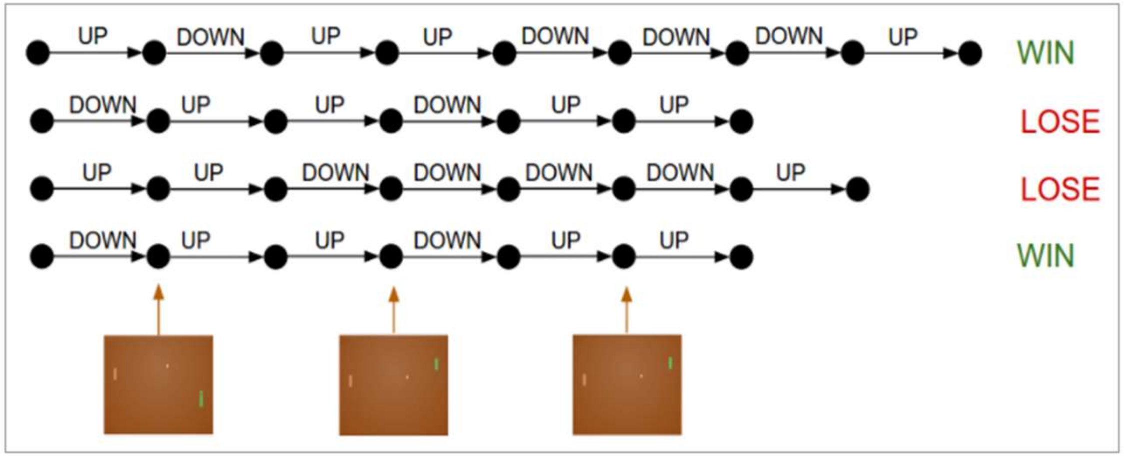
Playing games of Pong

- Examples of games/episodes
- You play a lot of actions and receive an reward at the end
- You get a result, WIN! Great, but how do you know which action, caused the victory?
  - Well… you don't



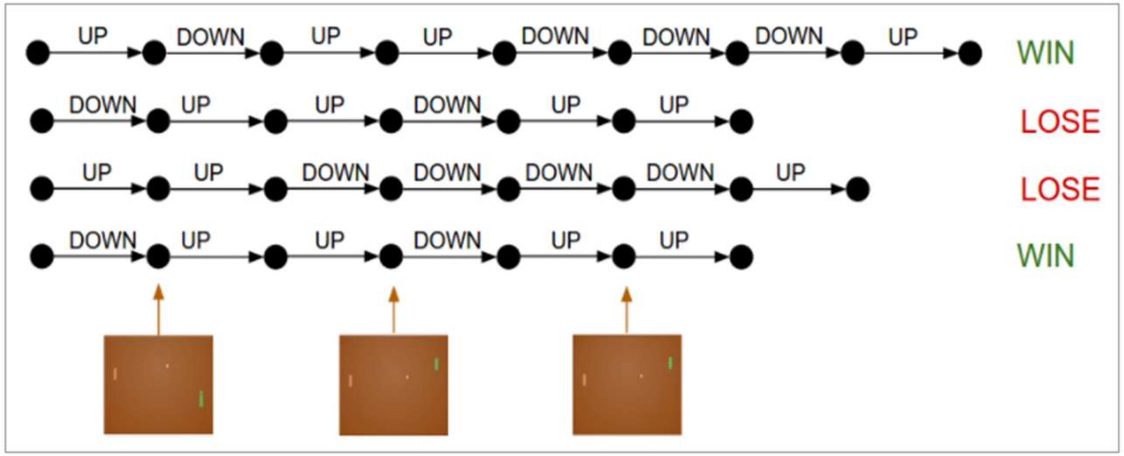
Which action caused the final results?

- In a winning series there may be many non-optimal actions
- In a losing series there may be good actions
- The **true** effect is found by averaging out the noise, as winnings series tend to have more good action and visa versa

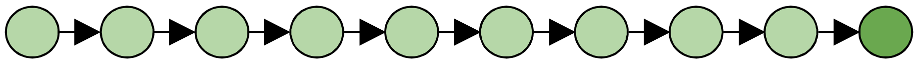


A chain of actions can cause large variations in performance

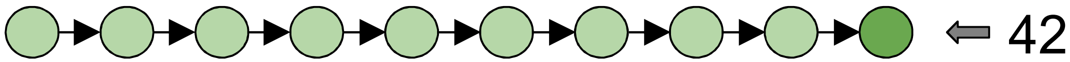
- If we change one action early in the network, we can easily move into unchartered territory.
- Imagine a self-driving car model that is used to driving on roads. If it happens to miss the road, it may have no idea what to do.
- If one action in the chain changes, other earlier actions may go from WIN, WIN, WIN to LOSE, LOSE, LOSE
- This high variance gradients make learning slow



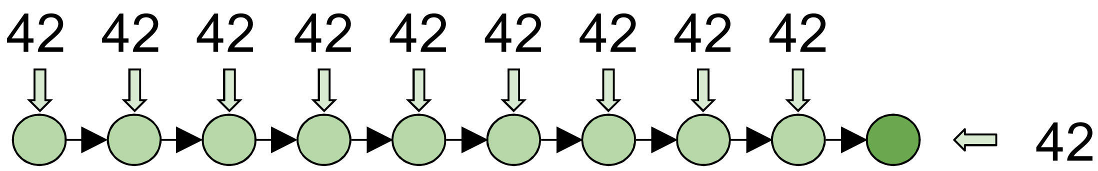
Policy gradients: High variance



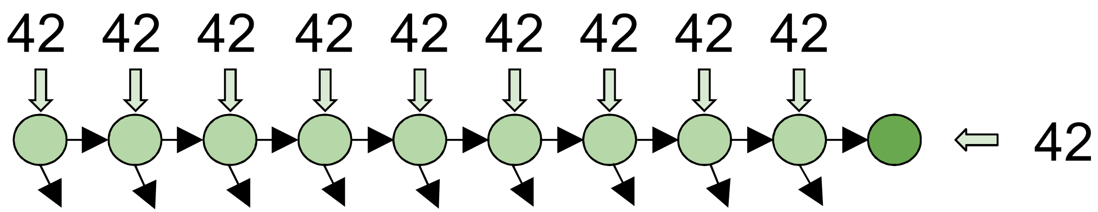
Policy gradients: High variance



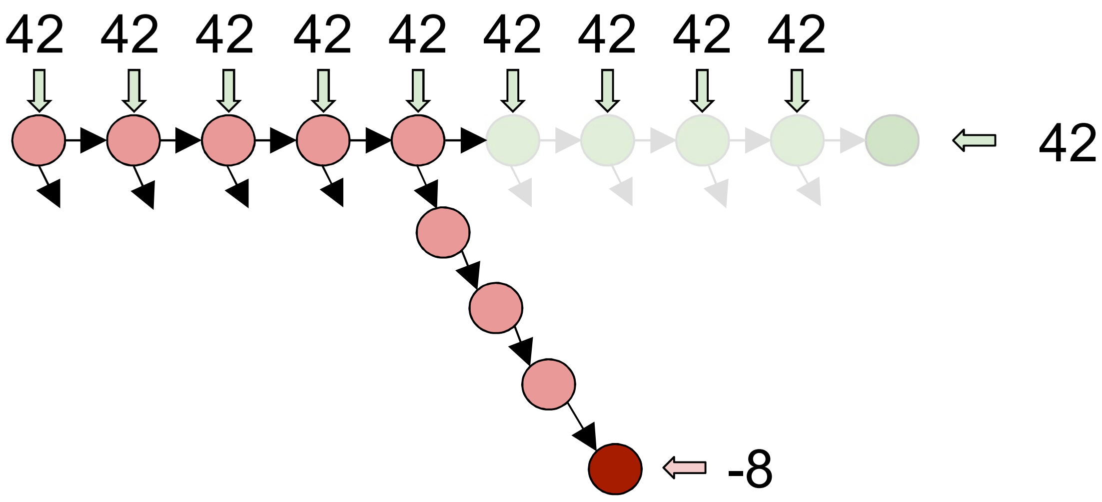
Variance - all choices get the reward



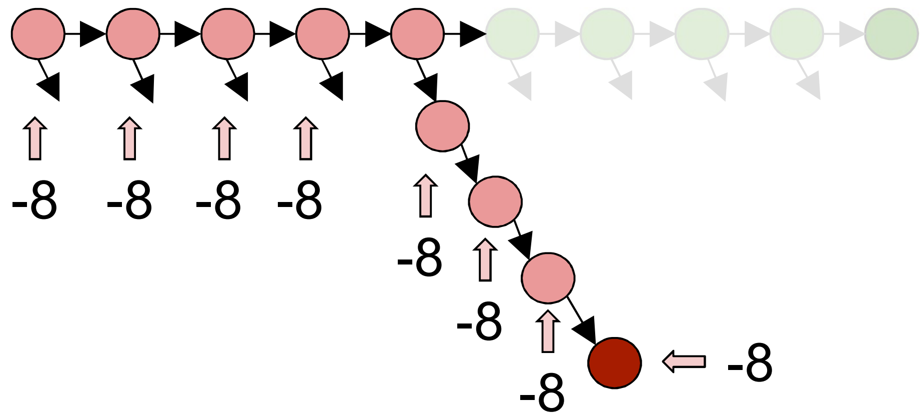
Variance - other possible paths



Variance - high probability to chose some other path

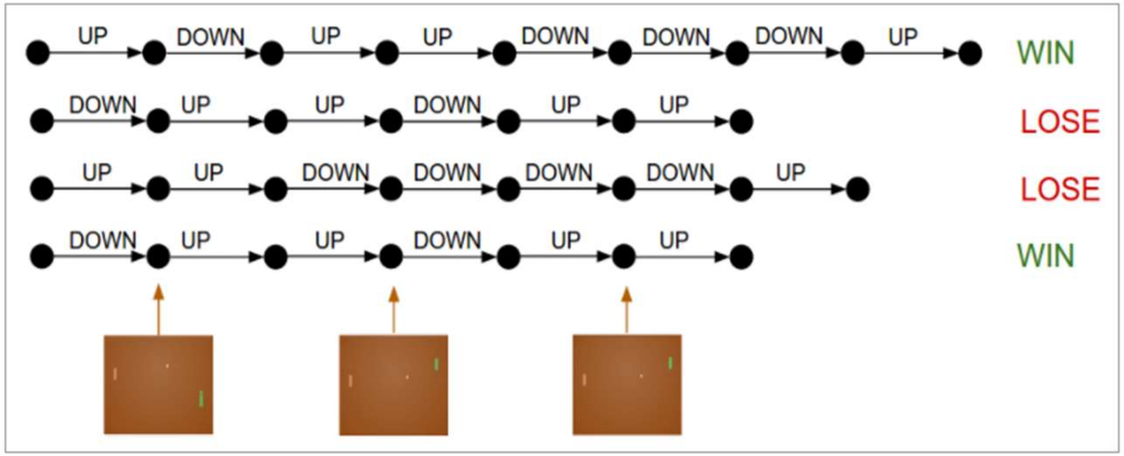


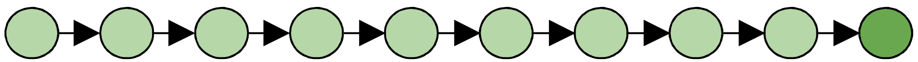
Variance - same actions for same state: now negative



Variance reduction

- In pong and most other applications, the final actions leading up to the win relate more to the final result than other actions.





Variance reduction

- Gradient estimator:

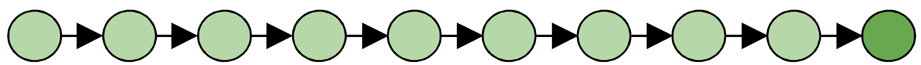
$$\nabla_\theta \mathcal{J}(\theta) \approx \textstyle\sum_{t\geq 0} r(\tau)\, \nabla_\theta \log \pi_\theta(a_t|s_t)$$

- **First idea**: The return can be the accumulative reward from the state and to the end.

$$\nabla_\theta \mathcal{J}(\theta) \approx \sum_{t\geq 0} \left(\sum_{t'\geq t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

- **Second idea**: Add the discount factor, γ, to reduce the effect of delayed rewards

$$\nabla_\theta \mathcal{J}(\theta) \approx \sum_{t\geq 0} \left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$



Variance reduction: Baseline (not curriculum)

- The accumulated discounted reward is not necessarily a reasonable value to be used for changing our probability distribution of the agent e.g. all reward are positive.

- What we care about is whether an action is better or worse then expected. We can subtract an estimate of the goodness of the state (**baseline)**.

$$\nabla_\theta \mathcal{J}(\theta) \approx \sum_{t\geq 0}\left(\sum_{t\prime\geq t}\gamma^{t'-t}r_{t'} - b(s_t)\right)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

- The most naive form of the baseline could be to use an moving average of the return experienced by all trajectories so far.

Variance reduction: Baseline (not curriculum)

- **Question**: Can we find a better alternative?

$$\nabla_\theta \mathcal{J}(\theta) \approx \sum_{t\geq 0}\left(\sum_{t'\geq t}\gamma^{t'-t}r_{t'} - b(s_t)\right)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

- In general, we want to increase the probability of choosing an action if the action is better than the expected return from the particular state.

$$\nabla_\theta \mathcal{J}(\theta) \approx \sum_{t\geq 0}(Q(a_t,s_t) - V(s_t))\,\nabla_\theta \log \pi_\theta(a_t|s_t)$$

- Q: Is the q-value (action value) function
- V: Is the state-value function

Q-learning vs Policy learning

Policy learning:

- More stabile
- The policy can be simpler to represent
- Imagine pong:
  - It can be easy to find out that you have to move in one direction
  - It can be hard to estimate the actual return for that step
- Effective:
  - You get the policy directly
- “Built-in” stochastic polices

Q-learning

- Can converge faster
- Can be more flexible as you need state pair to learn only
  - Experience replay
  - Don’t need full episodes

Progress

- Motivation
- Introduction to reinforcement learning (RL)
- Value function based methods (Q-learning)
- Policy based methods (policy gradients)
- **Value function and policy based methods (Actor-Critic)**
- Miscellaneous

Actor-Critic Algorithm (not curriculum)

- From policy gradients, we wanted to find values for Q and V:

$$\nabla_\theta \mathcal{J}(\theta) \approx \sum_{t \geq 0} (Q(a_t, s_t) - V(s_t)) \nabla_\theta \log \pi_\theta(a_t | s_t)$$

- **Solution**: We can use Q-learning to estimate Q and V. The Actor-Critic algorithm is a combination of Policy gradients and Q-learning.
  - The **actor** (policy) is defined by Policy gradients
  - The **critic** is defined by Q-learning

Actor-Critic Algorithm (not curriculum)

- The actor (policy) decides which action to take, the critic reverts back with how good the action was compared to the average action.

- We don’t get the same variance problem since we only learn transition between steps at a time.

Basic actor-critic method:

Start with state s, and sample action a

1. get reward r from **critic** for s and a
2. sample action a’ from **actor**
3. estimate new reward r’ from **critic**
4. update **critic** with difference between r and r’ (or real reward)
5. update **actor** based on estimated reward r’
6. set a <- a’, s <- s’

Progress

- Motivation
- Introduction to reinforcement learning (RL)
- Value function based methods (Q-learning)
- Policy based methods (policy gradients)
- Value function and policy based methods (Actor-Critic)
- **Miscellaneous**

Model based RL

- We can model the environment using e.g. a neural network. The network can be trained in a supervised way.
- By modeling the environment, we can "look ahead" and use search trees for evaluating our action.
- Important in e.g. games as chess and go where a single wrong action can make you loose.