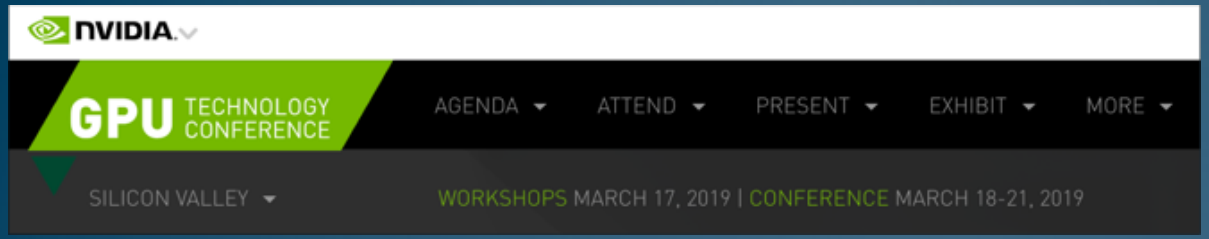
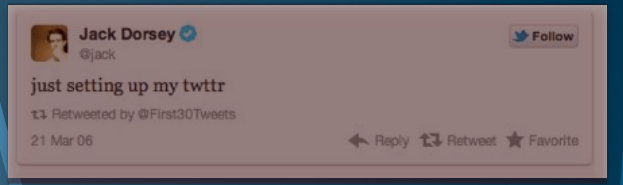
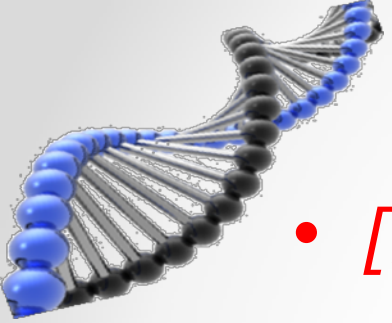




Demystifying Deep Reinforcement Learning – Part 1 : DQN

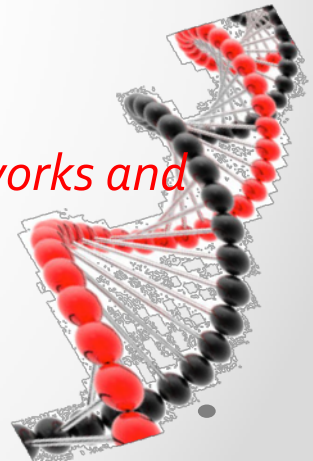
All observations are my own & do not represent my employer's or anyone else's

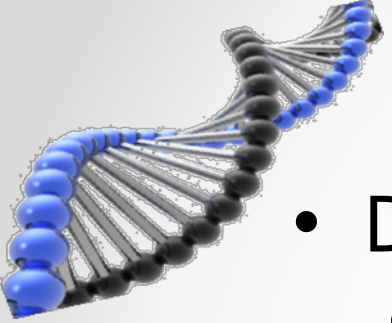
krishna sankar
@ksankar



Notes to GTC2019 Lab Reviewers

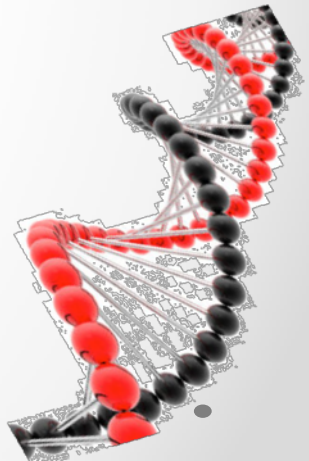
- *[1/14/19]*
  - *Both labs L9105 and L9103 are 95% done.*
    - *Thanks Andy for giving me both sessions*
  - *Materials in github <>*
  - *We can move it to wherever GTC'19 materials are*
  - *The run time requirements are not that big.*
  - *I can dockerize the notebooks, if needed*
  - *Also if you have GPU instances, I can run deeper networks and show results*





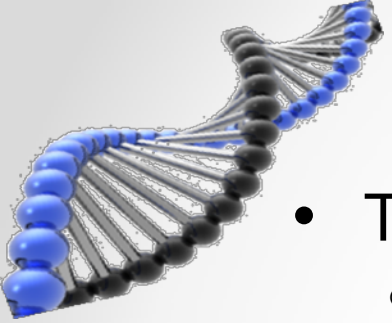
GTC 2019 SV : Lab Overview

- Deep Reinforcement Learning
  - Principles, Paradigms & Equations in 2 Parts
- Part 1 : Start from the Fundamentals ↦
  - Program Value Methods
  - Monte Carlo, TD & Q-learning
  - DQN and it's extensions ↦
- Part 2 : Policy Gradients
  - REINFORCE
  - A3C & DDPG
  - Intro to PPO, TRPO & the rest



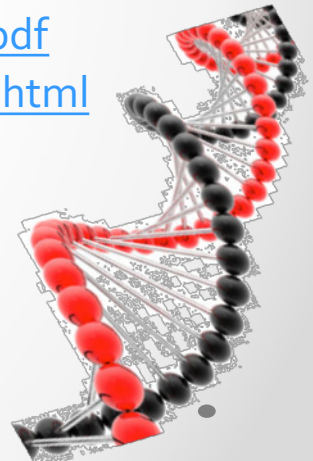
DRL Lab Goals

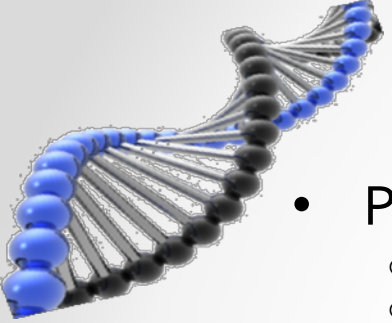
1. Get a reasonably good intro to DRL (as much as we can do in 4 hrs)
   - Sometimes the ideas take time to rattle around the brain and sink in; they need reflection; hence we stop at DQN and start Policy Gradients as Part 2
2. Alleviate the Initial Barrier
   - Discuss intuition and formal notations
   - Understand the underpinnings
   - Do hands-on programming and finally,
   - Motivate to learn, explore on your own & be proficient in Deep Reinforcement Learning
3. We will use Autonomous Driving & AlphaGo as background examples
4. Hands on programming
   - Gridworld = Frozen Lake,
   - Continuous = Cartpole
   - 2 simper examples across all algorithms



Please Do …

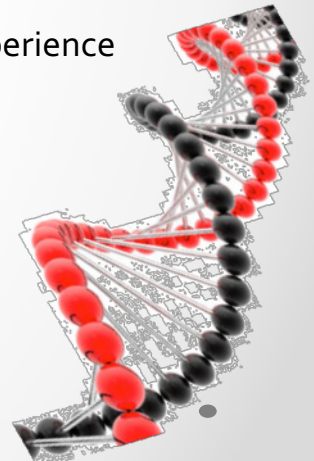
- The Programming Exercises & Quizzes
  - This is a good time to do these - you have a captivated time away from the daily trifles
- Write the formulas, at least 3 times !
  - Why write down formula ? Levine UCB classes !
- Read the essential papers
  - Good comprehensive intro https://arxiv.org/pdf/1810.06339.pdf
  - Sutton Book http://incompleteideas.net/book/the-book-2nd.html
  - Minh Paper : https://arxiv.org/abs/1602.01783
  - Rainbow Paper https://arxiv.org/pdf/1710.02298.pdf
  - AlphaGo nature papers, movies, AlphaZero et al https://deepmind.com/research/alphago/

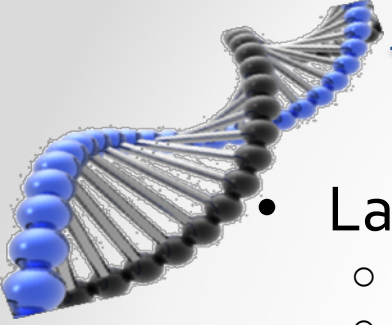




You have …

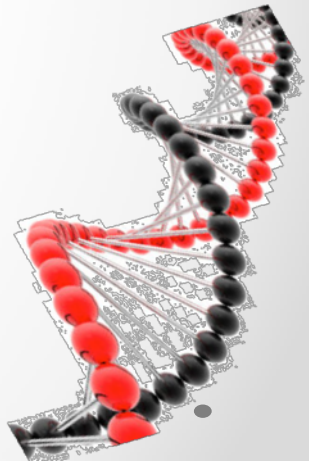
- Programming experience
  - Knows about DeepLearning,
  - Experience using Tensorflow or pytorch,
  - Python programming and familiar with Jupyter/iPython notebook
- No Reinforcement Learning experience is expected or assumed
- Will cover enough of Deep Learning & Pytorch
- Why Pytorch ?
  - Easier & compact, those of you from the world of tf, this will be a new experience
  - DL is only a small part
- Have installed the prerequisites
- Have downloaded the materials from github
- The notebooks in working order



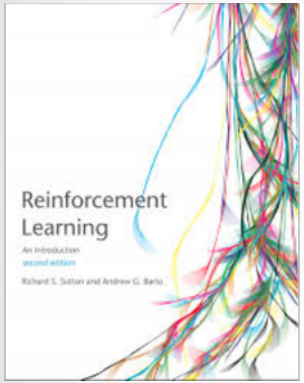


Lab notebooks & presentation

- Labs have working examples *good enough*
  - Look at them as frameworks - for you to tweak & improve
  - Use as a study tool
  - Share your comments, clarifications, functions and algorithms as pull requests
  - And when you have mastered an algorithm or an environment, add your successes to the OpenAI Gym leaderboard
- Lab pragmatics
  - In github <<url tbd>>
  - No esoteric requirements
  - You can run them without docker
  - pip install -r requirements.txt
  - Requirements
    - python 3.6, pytorch, openAI gym, numpy, matplotlib
  - anaconda is easier but not needed
  - Miniconda works fine



Thanks to the Giants, whose work helped me to prepare this lab !!

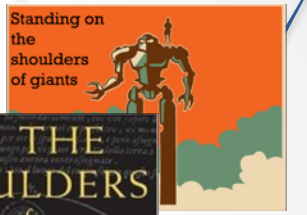


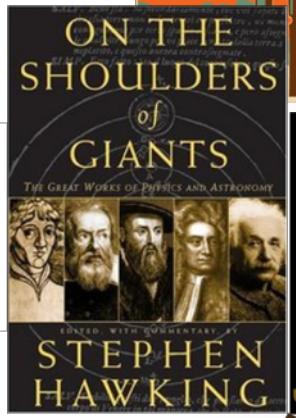
School of Artificial Intelligence

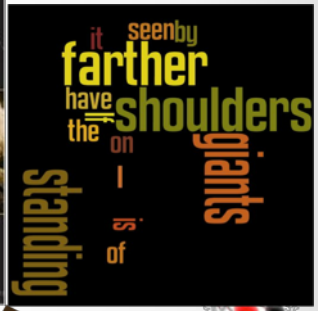
Deep Reinforcement Learning

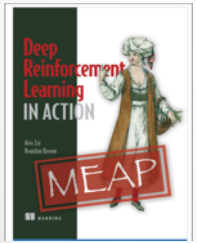
In Collaboration With

Unity, NVIDIA Deep Learning Institute









OpenAI

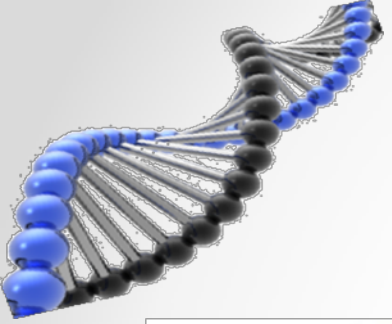
CS 294-112 at UC Berkeley

Deep Reinforcement Learning

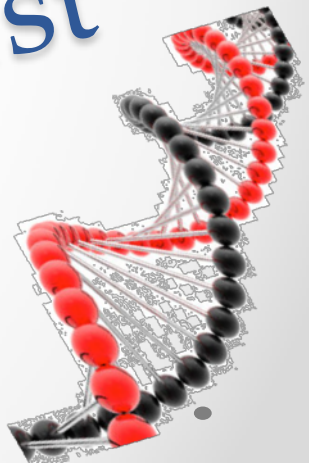
Abbeel & Levine

Agenda

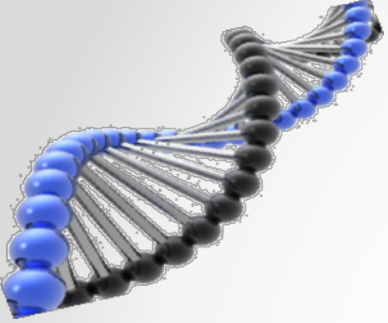
1. Framing
2. Reinforcement learning – A Formal Expedition
3. Algorithms & Programming
   - Monte Carlo, TD, SARSA, Q-Learning & DQN
4. The world of Atari, AlphaGo & AlphaZero
5. Policy Gradients – REINFORCE, DDPG
6. Conclusion & Forward

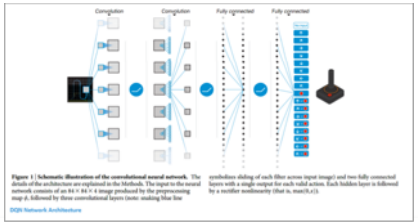


1. Framing : Open This First



What exactly is this Reinforcement Learning ?





Agent

3

State

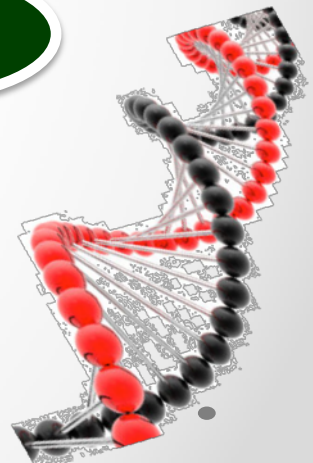
Reward

2

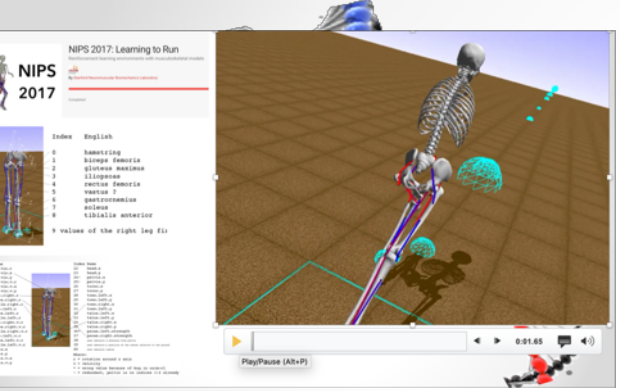
1

Action

Environment



DRL in Action (?) – Video Time !!



1. Walking Humanoid figures - UC Berkeley

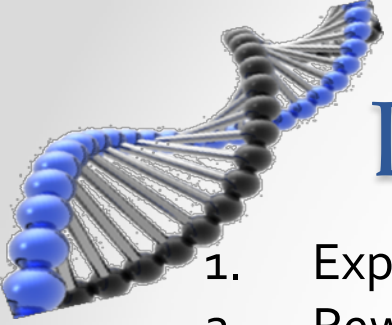
2. Humanoid avoiding obstacles - UCB

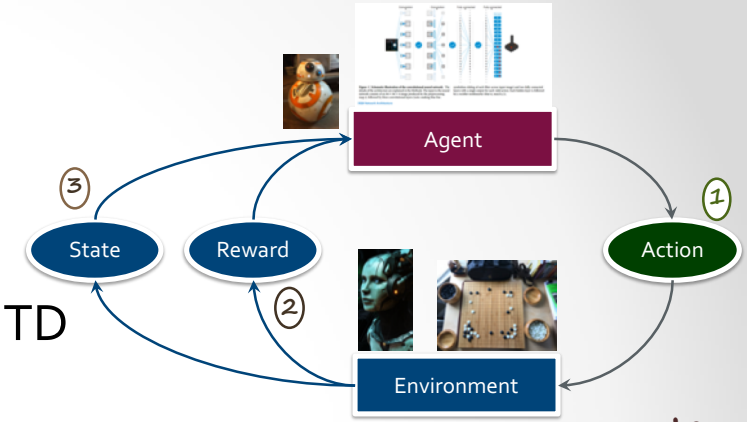
3. Autonomous Cars - Double Lane

4. Navigating Complex Roundabouts

Goals : Motivation for DRL

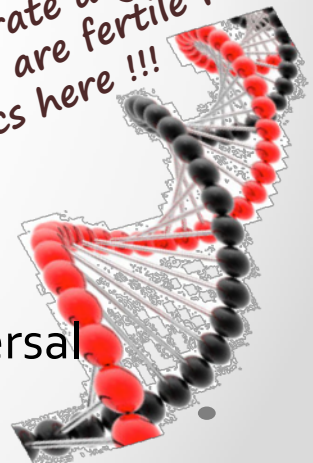
DRL Challenges

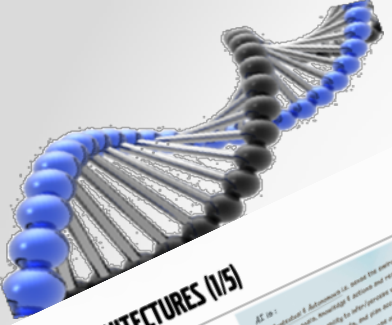


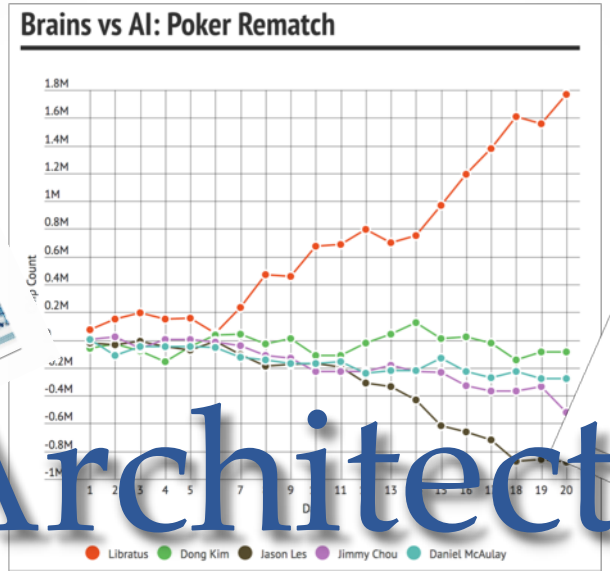


1. Exploration & Convergence – Bandit
2. Reward Shaping & Credit Assignment – TD
3. Representation
   - DQN, CNN (LSTM,RNN Not yet) Capsule Networks/Transformer Architectures (?)
   - Lower dimensional than tabular representation
   - Compact as possible, but can capture the full complexity of the underlying state space
   - Sample efficiency - replay buffer w/ DQN
   - Atari representation
4. Generalization - DQN
   - Atari generalization - same architecture, but trained for each game
5. Context & Content - Knowledge Graph
   - Encode expert knowledge
   - Atari - pixels - no knowledge
   - If it's Tuesday, this must be Belgium - you need Tuesday Belgium 1000 times !
6. Common Sense Reasoning - Logic Representation & Traversal
7. Conversation Chain - Jarvis/Jeeves

If you are planning to do PhD (I urge all to earn a doctorate degree in AI) there are fertile research topics here !!!



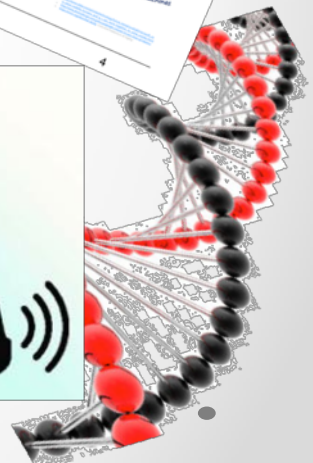




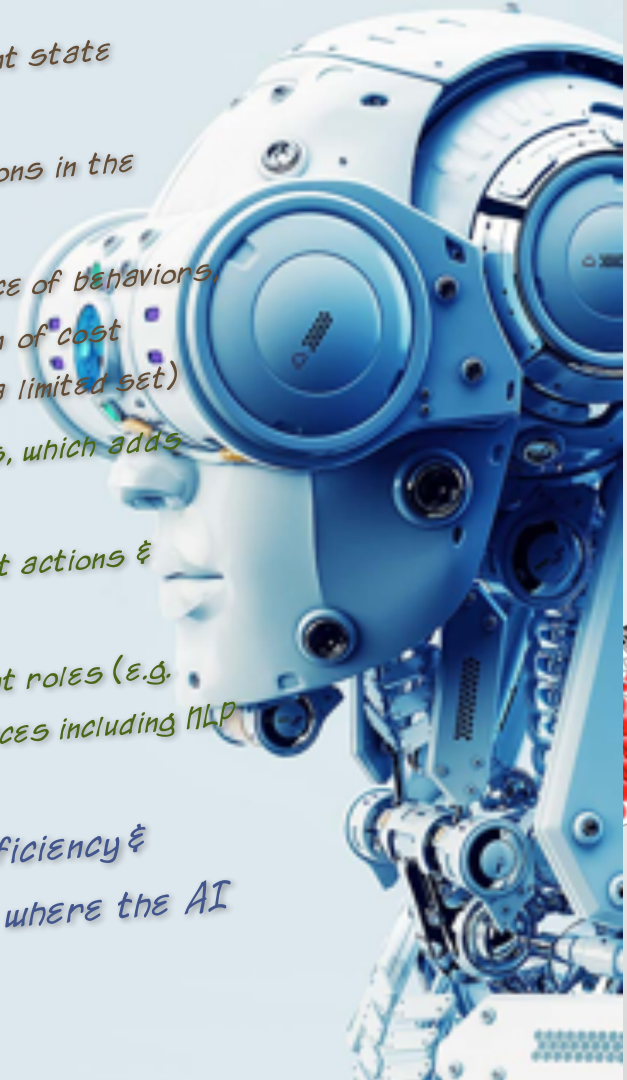
AI Architectures

Quick Context

- What is AI in our context ?
- A Robot's Rules of Order
- AI Frameworks
- ...



- AI is :
  - Contextual & Autonomous i.e. sense the environment, propagate current state across goals, knowledge & actions and respond appropriately,
  - Rational i.e. The ability to infer/perceive value/payoff of a set of actions in the current context/ beliefs, and plan accordingly
  - Optimal i.e. The capacity to choose one or more actions over the space of behaviors, based upon a set of policies including short & long term optimization of cost functions & goals (bounded rationality as the system can only see a limited set)
  - Remembering i.e. the capability to hold episodic and long term beliefs, which adds representational formalisms and retrieval mechanisms
  - Reflective, Adaptive & Self-evolving i.e. the ability to learn from past actions & incorporate the lessons in future behavior
  - Interactive i.e. the capability to interact with humans with different roles (e.g. passengers, other drivers, pedestrians et al) via multiple interfaces including NLP & NLU
- We also add the constraints of reactivity, redundancy, efficiency & scalability for the infrastructure & the framework stack where the AI system is deployed



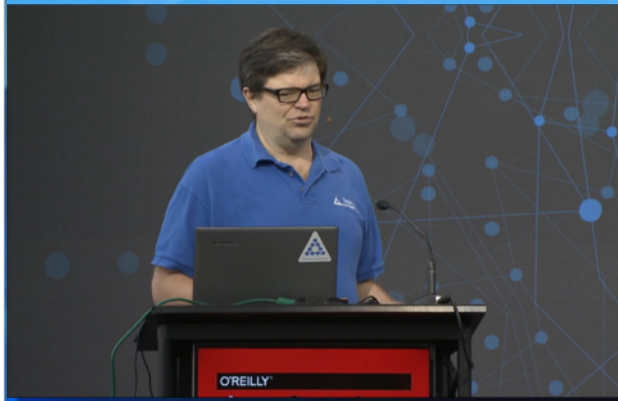
Obstacles to Progress in AI

Y LeCun

- Machines need to learn/understand how the world works
  - Physical world, digital world, people,....
  - They need to acquire some level of common sense
- They need to learn a very large amount of background knowledge
  - Through observation and action
- Machines need to perceive the state of the world
  - So as to make accurate predictions and planning
- Machines need to update and remember estimates of the state of the world
  - Paying attention to important events. Remember relevant events
- Machines neet to reason and plan
  - Predict which sequence of actions will lead to a desired state of the world

- Intelligence & Common Sense =

  Perception + Predictive Model + Memory + Reasoning & Planning



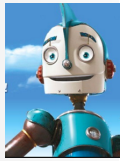
Yann LeCun Retweeted

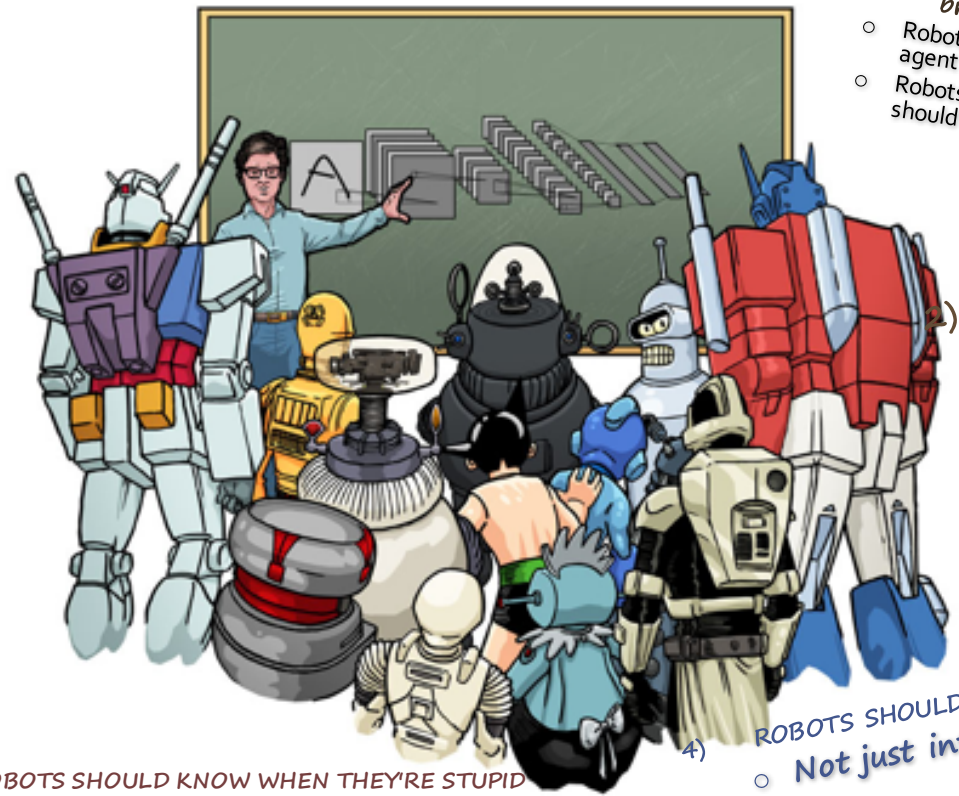
Andrew McAfee @amcafee · 17h

.@ylecun Teaching machines common sense is a "Big mountain. And we don't know how many mountains are behind it." #WHFrontiers

Robot's Rules of Order







1) ROBOTS SHOULDN'T MAKE THINGS WORSE
   - i.e. Robots should be programmed to understand broad categories of side effects & externalities
   - Robots should be programmed with impact regularizers, multi-agent reward autoencoder & reward uncertainty
   - Robots should never obsess about only one thing, their AIs should be designed with a dynamic reward system

2) ROBOTS SHOULDN'T CHEAT
   - One possible solution to this problem is to program robots to give rewards on anticipated future states

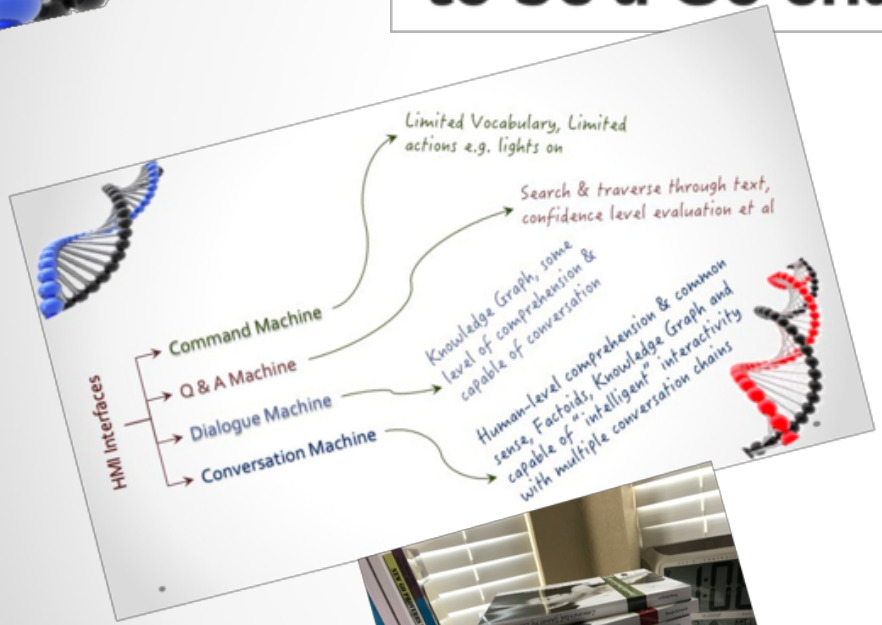
3) ROBOTS SHOULD BE DESIGNED FOR:
   - INTELLIGENT PRIVACY &
   - ALGORITHMIC ACCOUNTABILITY

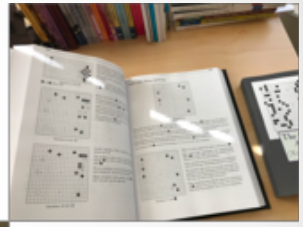
4) ROBOTS SHOULD BE TRANSPARENT
   - Not just intelligent machines but intelligible machines

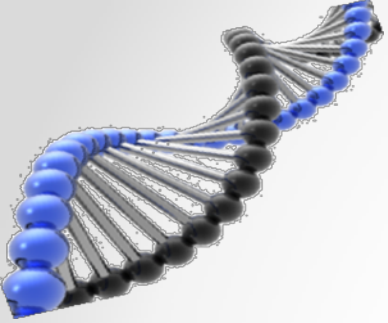
5) ROBOTS SHOULD KNOW WHEN THEY'RE STUPID
   - Programming artificial intelligence is one thing. But programming robots to be intelligent about their idiocy is another thing entirely
   - Policy-Consequence Framework.

- http://www.slate.com/articles/technology/future_tense/2016/06/microsoft_ceo_satya_nadella_humans_and_a_i_can_work_together_to_solve_society.html
- http://www.fastcodesign.com/3061230/google-wants-us-new-laws-of-robotics
- https://arxiv.org/abs/1606.06565 – Concrete Problems in AI Safety

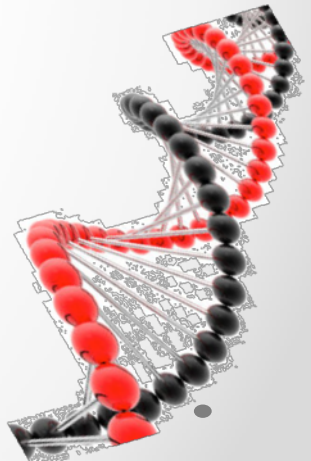
Ask not if AlphaZero can beat humans in Go—Ask if AlphaZero can teach humans to be a Go champion !!!



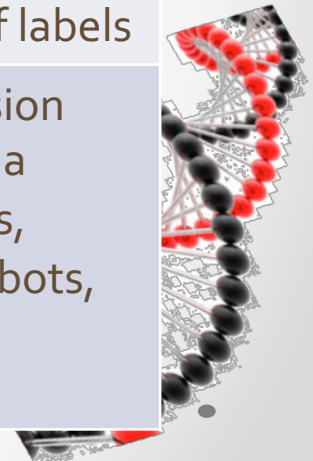




2. Reinforcement Learning
A Formal Expedition



	Supervised	Unsupervised	Reinforcement Learning
Data	Curated	Curated	Generates data while exploring
Labels	Curated	Machine Generated	Learn via interactions
Signals & Feedback			Curated Reward Signals and Goals
Actions			Actions instead of labels
Defining Characteristics	Classification, Linear Regression	Human Meaning e.g. clusters, survival analysis	End result = Decision making - winning a game, driving cars, manufacturing robots, cleaning robots, conversations, …

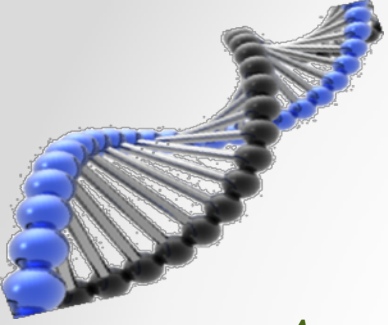


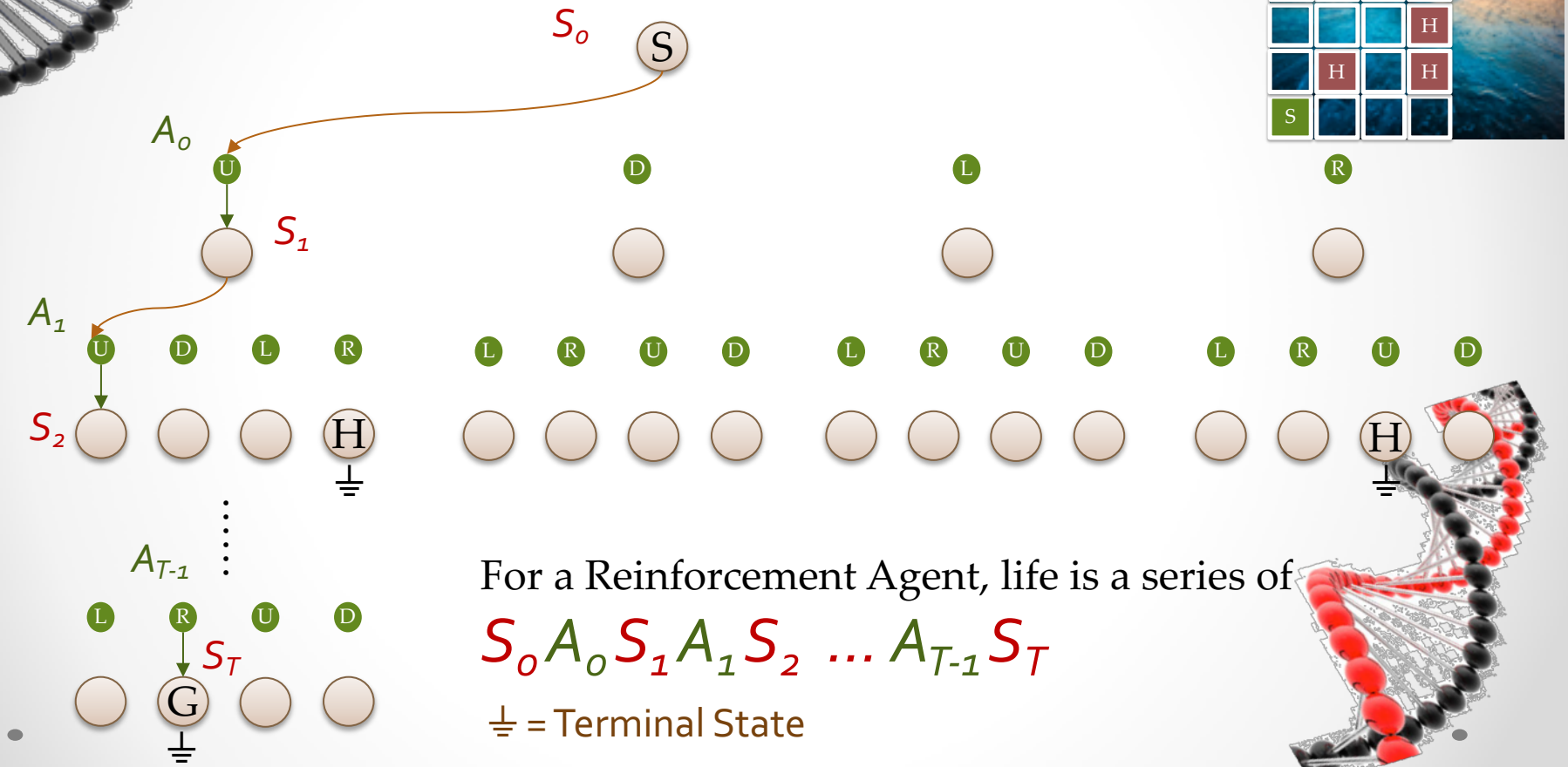
Frozen Lake !

- You are in Siberia and want to visit the foothill under the rock, which is the Goal (G), starting at S
- But the lake is a little treacherous with holes(H)

https://psmag.com/magazine/a-frozen-lake-in-siberia

Notation : State, Action



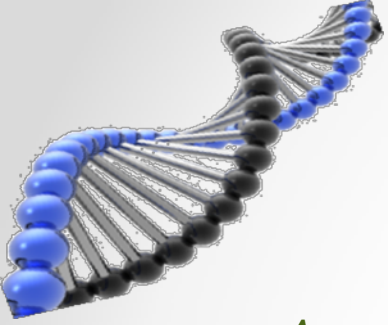


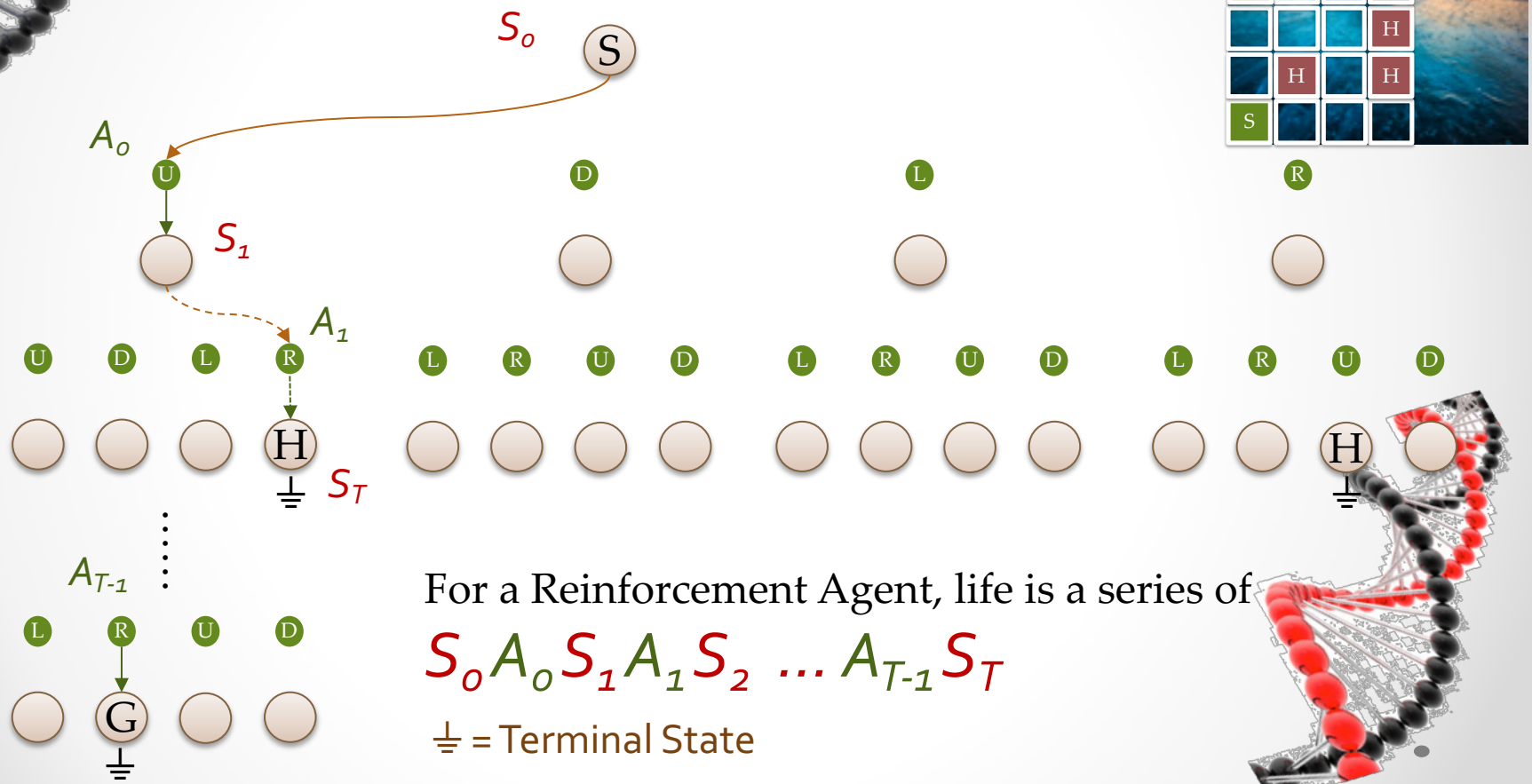
For a Reinforcement Agent, life is a series of

$$S_0 A_0 S_1 A_1 S_2 \ \ldots \ A_{T-1} S_T$$

⏚ = Terminal State

Notation : State, Action



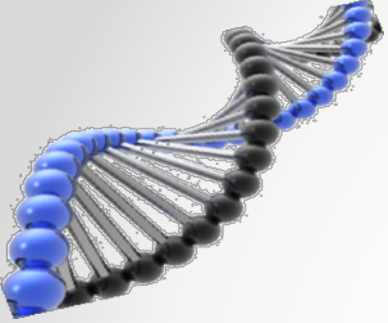


For a Reinforcement Agent, life is a series of

$S_0 A_0 S_1 A_1 S_2 \ldots A_{T-1} S_T$

⏚ = Terminal State

Notation : State, Action



S_0

S

A_0

S_1

A_1

S_T

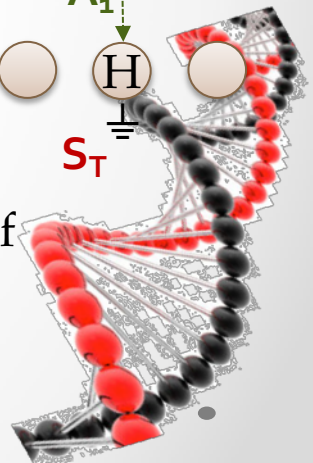
U D L R H

U D L R G

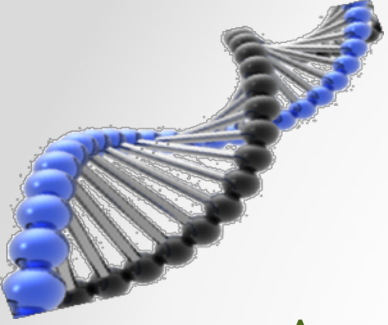
For a Reinforcement Agent, life is a series of

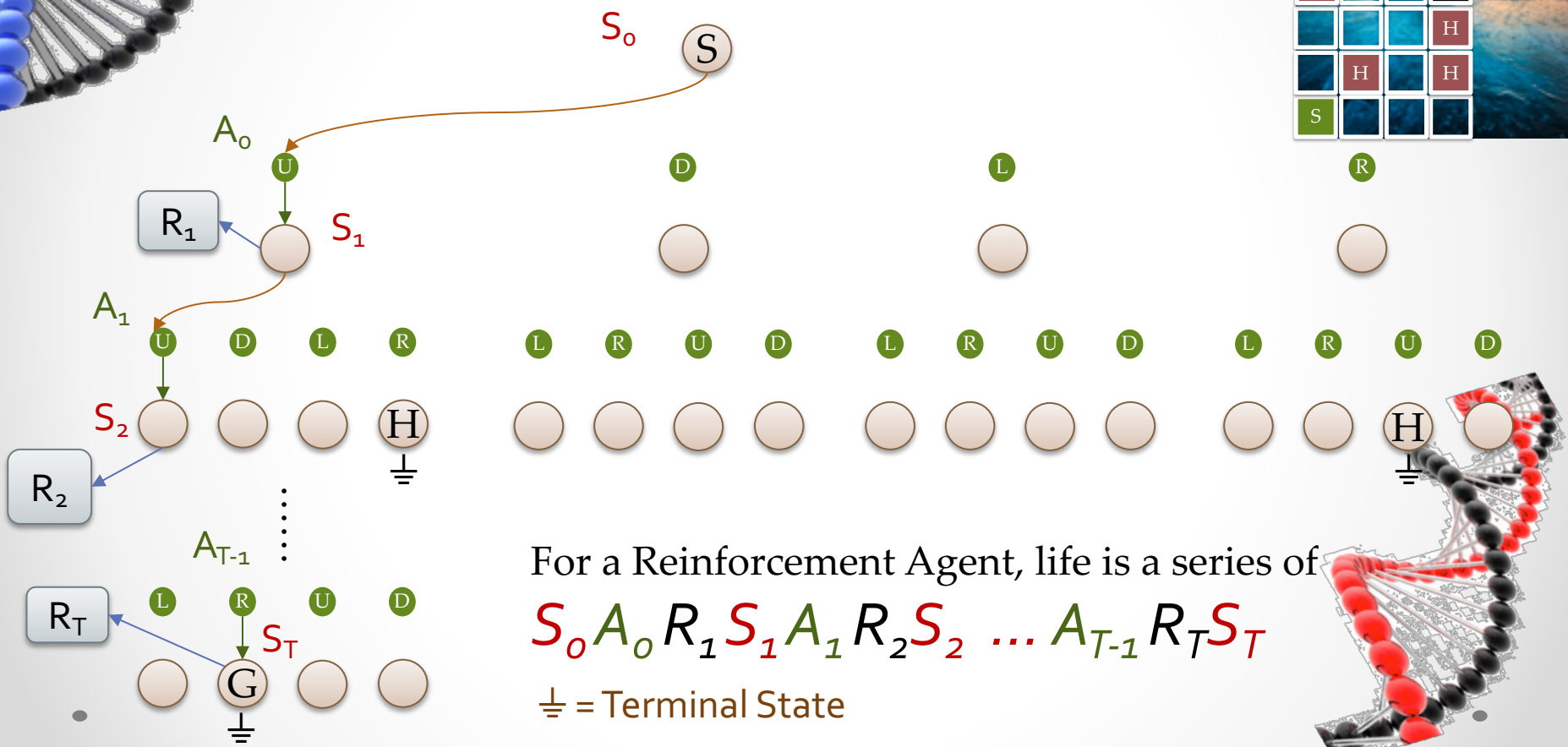
$S_0 A_0 S_1 A_1 S_2 \ \ldots \ A_{T-1} S_T$

⏚ = Terminal State



Notation : State, Action, Reward



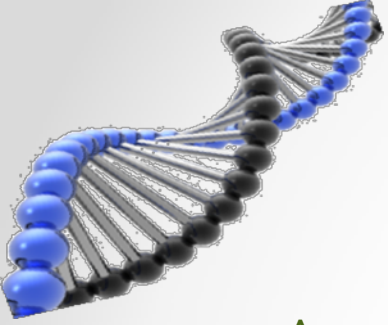


For a Reinforcement Agent, life is a series of

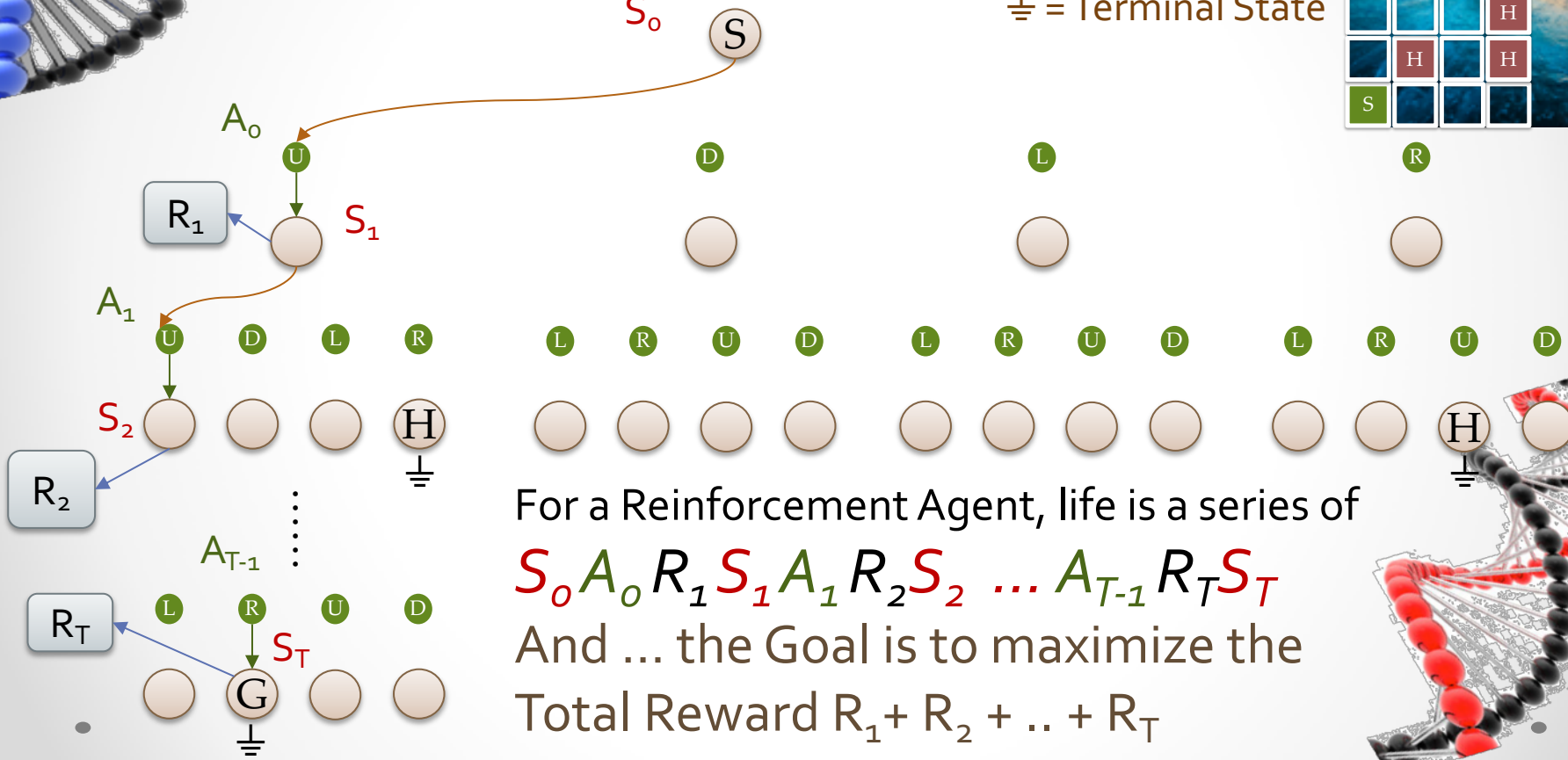
$$S_0 A_0 R_1 S_1 A_1 R_2 S_2 \ \ldots\ A_{T-1} R_T S_T$$

⏚ = Terminal State

Notation : State, Action, Reward



⏚ = Terminal State

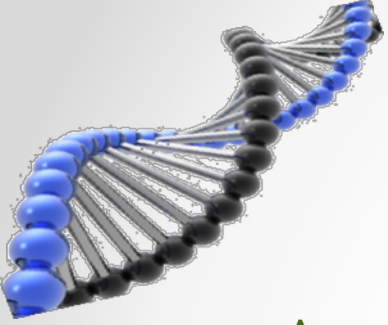


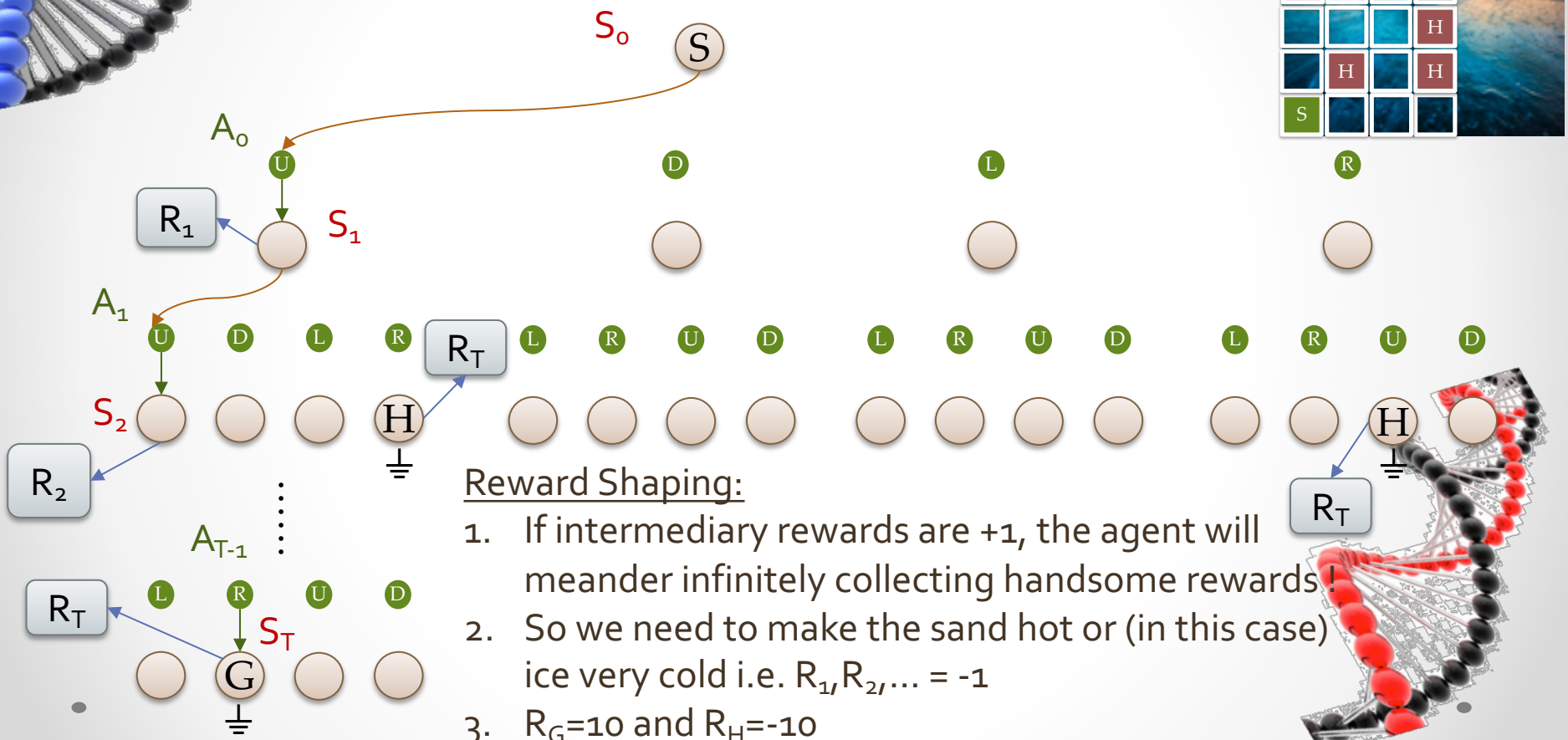
For a Reinforcement Agent, life is a series of

$S_0 A_0 R_1 S_1 A_1 R_2 S_2 \ \ldots\ A_{T-1} R_T S_T$

And … the Goal is to maximize the Total Reward $R_1 + R_2 + .. + R_T$

Notation : State, Action, Reward

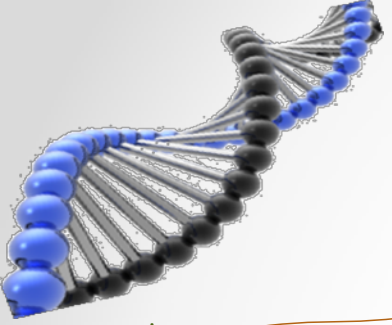


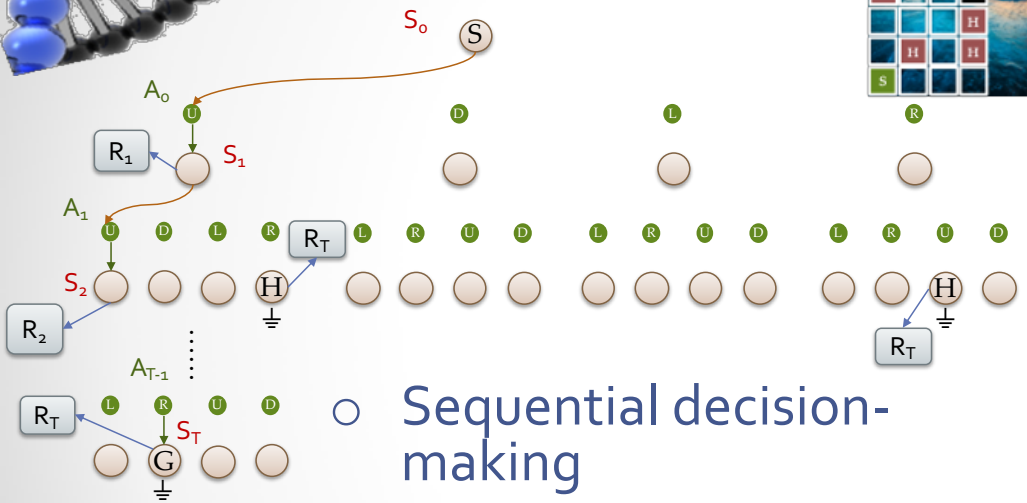


Reward Shaping:

1. If intermediary rewards are +1, the agent will meander infinitely collecting handsome rewards !
2. So we need to make the sand hot or (in this case) ice very cold i.e. $R_1, R_2, \ldots = -1$
3. $R_G=10$ and $R_H=-10$

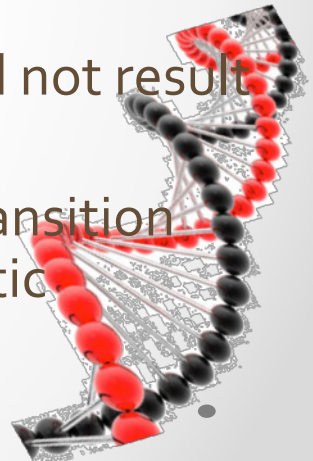
In short DRL is …



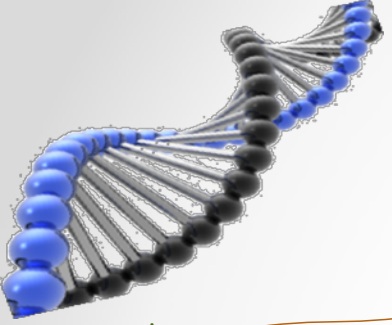


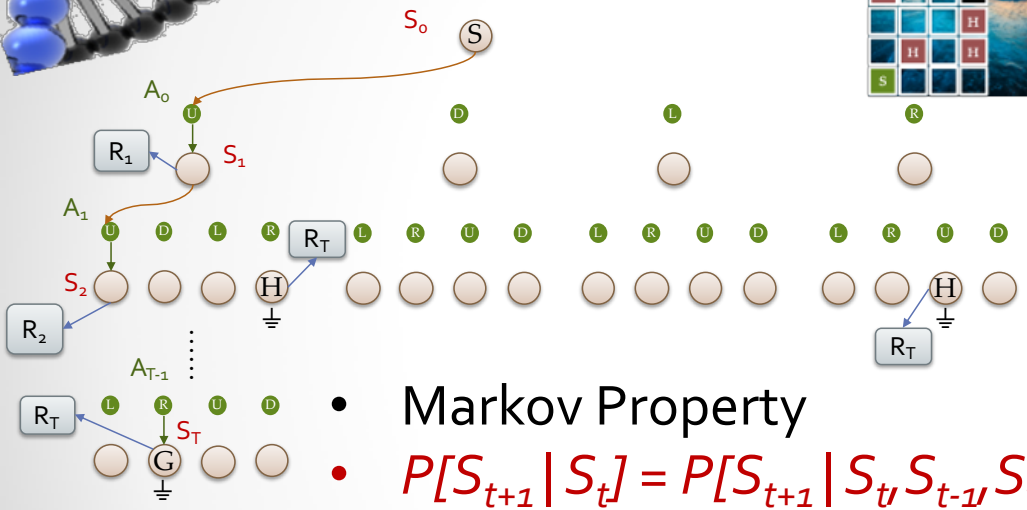
- Sequential decision-making
  - Under incomplete/probabilistic information even if we have full observability
- Evaluative feedback
- Exhaustive sampling

- There is a State Space
  - State is actually an observation, need not represent the environment
- There is an action space
- There are rewards
  - Goal is to maximize total rewards (= Return)
  - R_t = One step Return
- Same action need not result in same return
- Action to state transition might be stochastic



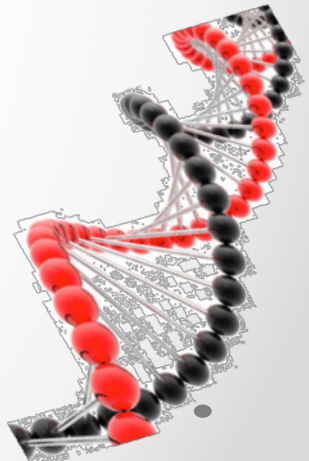
Markov Process



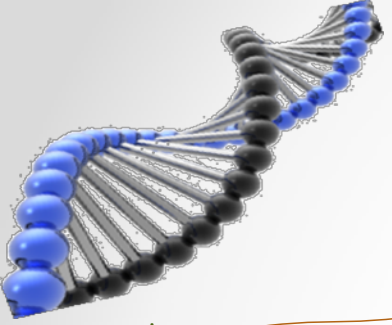


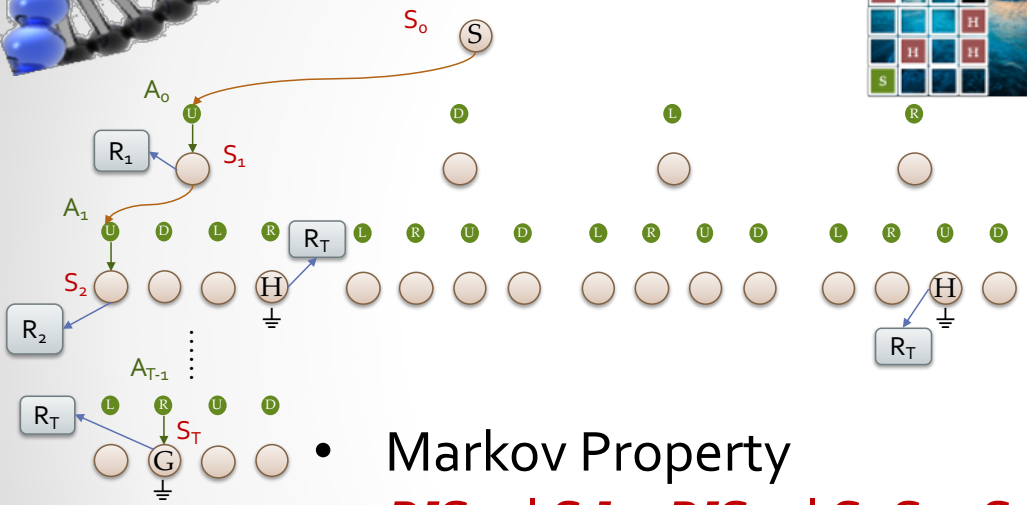
- Markov decision process formally describes an environment for Reinforcement Learning (Silver)

- Markov Property
- $P[S_{t+1} | S_t] = P[S_{t+1} | S_t, S_{t-1}, S_{t-1}, \ldots]$
- Current state is sufficient to predict the future
- A Markov process is a *memoryless sequence* of states $S_1, S_2, S_3, \ldots$ with the Markov Property



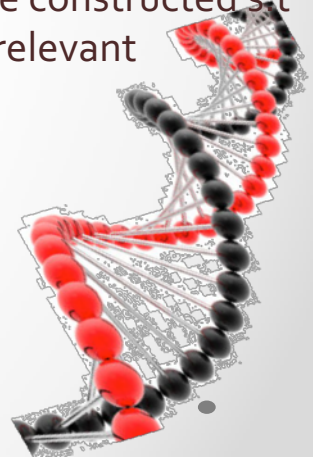
Markov Process



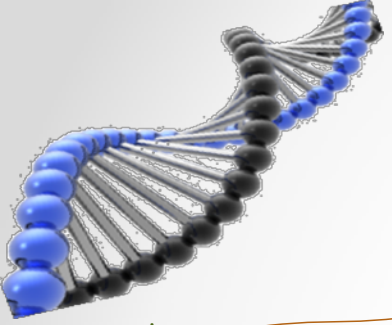


- Good
  - Don't have to remember all trajectories & paths
  - Don't have to consider them for decisions
- Bad
  - The state has to be constructed s.t it encodes all the relevant information

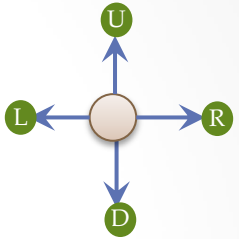
- Markov Property
- $P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_t, S_{t-1}, S_{t-1}, \ldots]$
- Current state is sufficient to predict the future
- A Markov process is a *memoryless sequence* of states $S_1, S_2, S_3, \ldots$ with the Markov Property

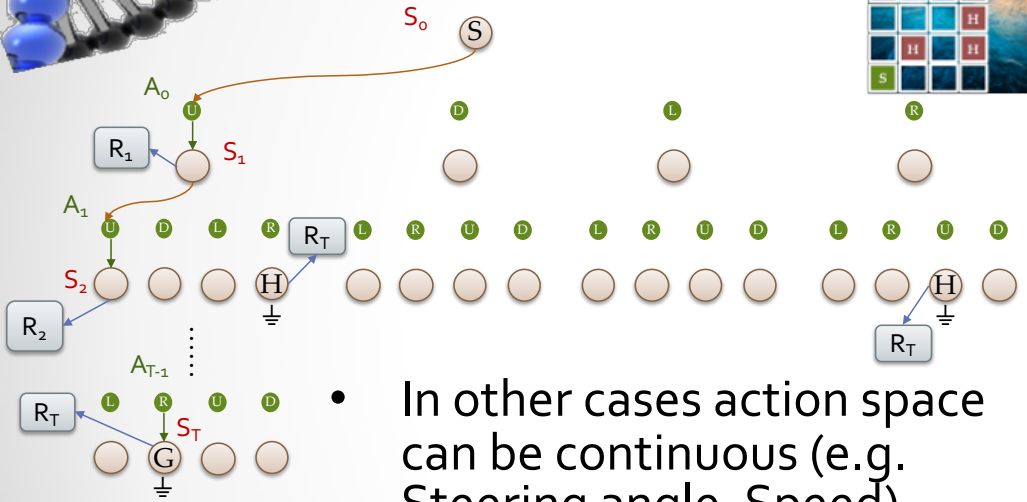


Action Space



- 4 actions





- In other cases action space can be continuous (e.g. Steering angle, Speed)
- Different actions at different states (games, parking vs. merging)
- Large Action space

- How does an agent choose the actions ?
  - Equi-probable
  - ε-greedy
  - Greedy
  - Softmax distribution
- Best action need not be deterministic
  - E.g. Rock-Paper-Scissor
- We will examine policy later

Reward Space & Discounting

R_t = Reward at a time step

G_t = Total Return

Goal is to maximize this

$$G_{t(t=0)} = R_1 + R_2 + .. + R_T$$

$$G_t = R_{t+1} + R_{t+2} + .. + R_T$$

Add Discount factor γ to account for uncertainties,

$$G_t = \Sigma_{k=0 \text{ to } \infty} \gamma^k R_{t+k+1} \quad ..(3.8)$$

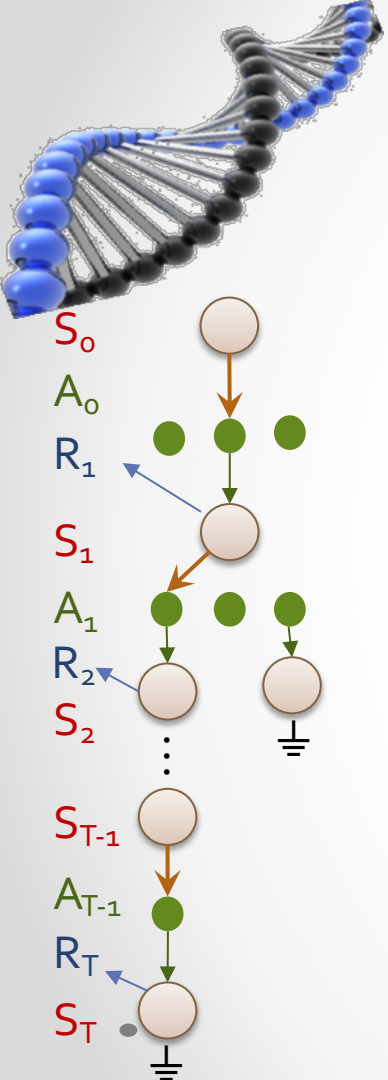
$\gamma \in (0,1)$

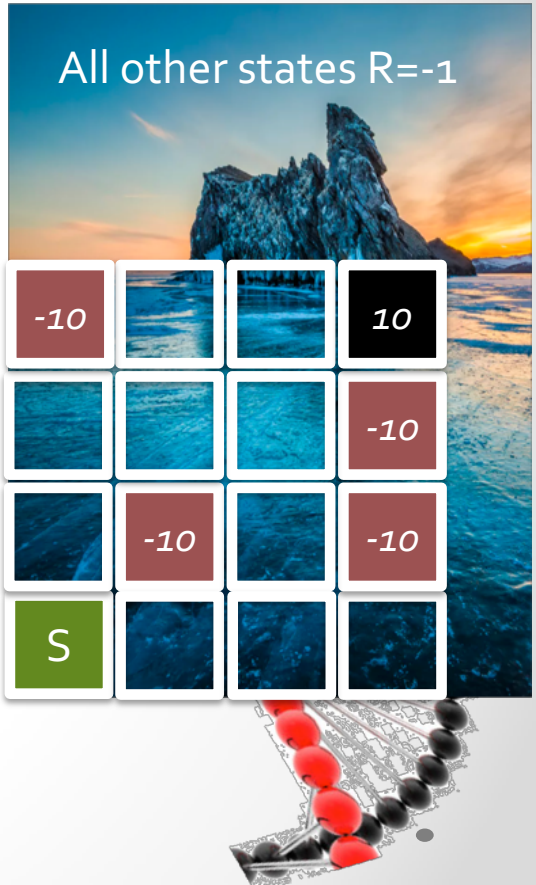
close to 0 gives "myopic"

close to 1 gives "far-sighted"

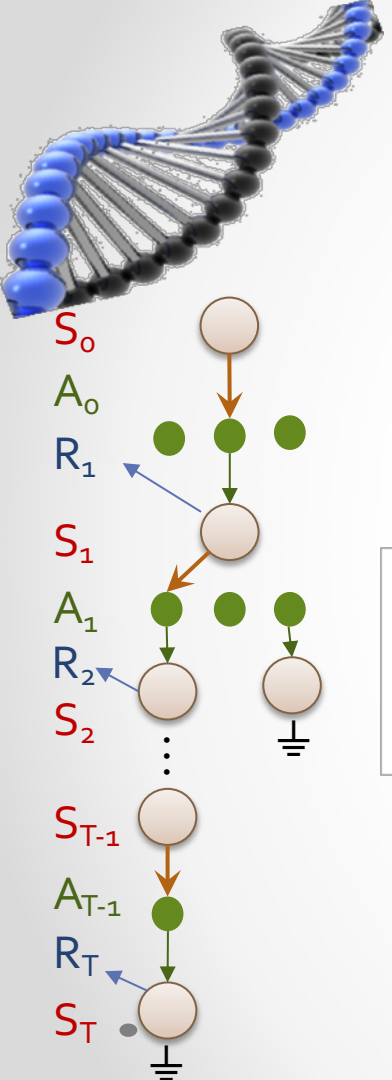
γ = 0.9 for continuous,

= 1.0 for sequences that terminate





Richard Bellman



Richard E. Bellman invented Dynamic Programming

His formulations form the foundation of Reinforcement Learning

THE THEORY OF DYNAMIC PROGRAMMING

Richard Bellman

P-550

30 July 1954

The RAND Corporation

1700 MAIN ST. • SANTA MONICA • CALIFORNIA

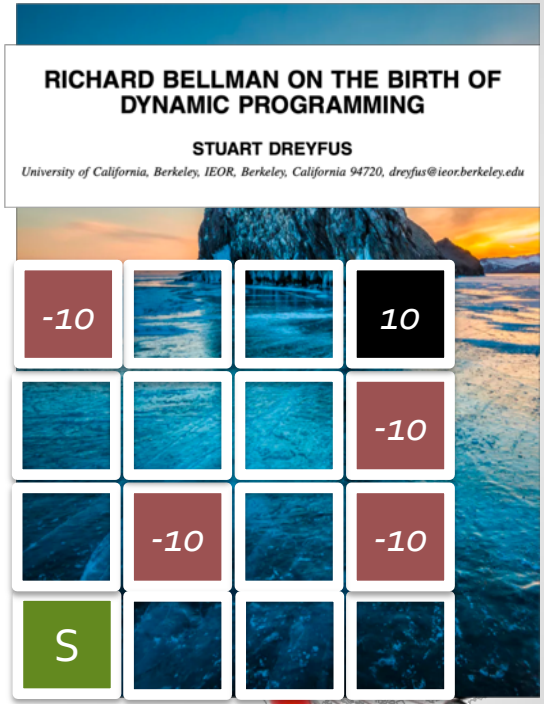
Turning to a more precise discussion, let us introduce a small amount of terminology. A sequence of decisions will be called a policy, and a policy which is most advantageous according to some preassigned criterion will be called an optimal policy.

Dynamic Programming – refactor into sub problems

$$G_t = \Sigma_{k=0 \text{ to } \infty} \gamma^k R_{t+k+1}$$

$$= R_{t+1} + \Sigma_{k=1 \text{ to } \infty} \gamma^k R_{t+k+1}$$

$$= R_{t+1} + \gamma G_{t+1} \ldots (3.9)$$



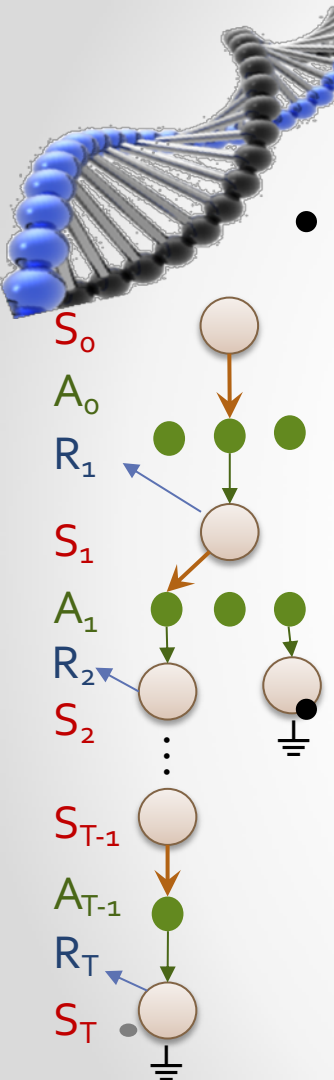
RICHARD BELLMAN ON THE BIRTH OF DYNAMIC PROGRAMMING

STUART DREYFUS

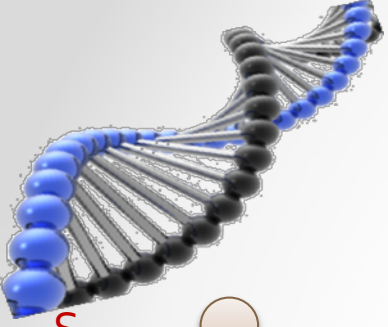
University of California, Berkeley, IEOR, Berkeley, California 94720, dreyfus@ieor.berkeley.edu

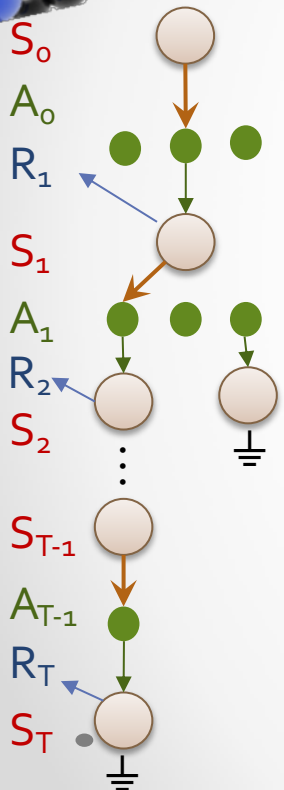
Policy

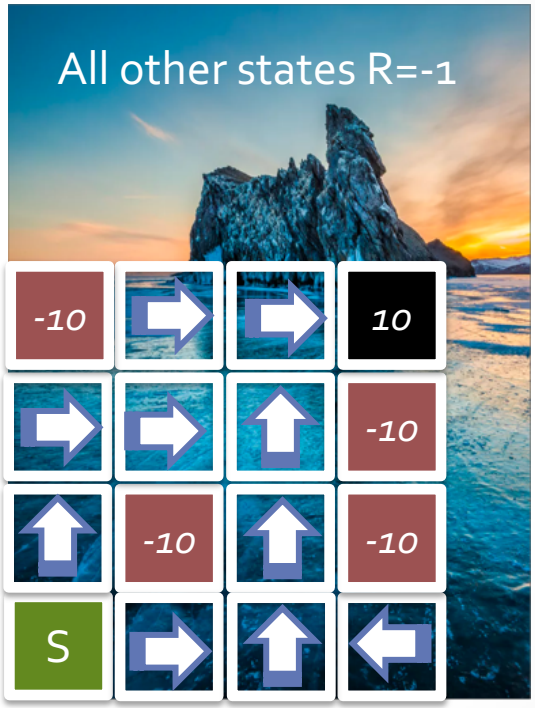
- How does an agent choose an action ?
  - Equi-probable Random
  - ε-Greedy
  - Greedy
  - Softmax distribution
- Or A deterministic policy as given here
  - Which is possible if we have the full model of the environment



State Space & Policy

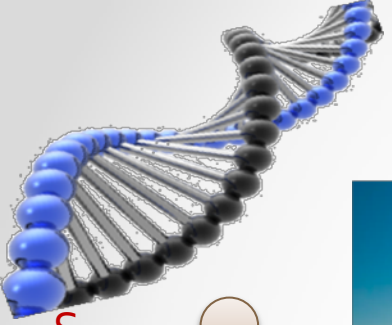


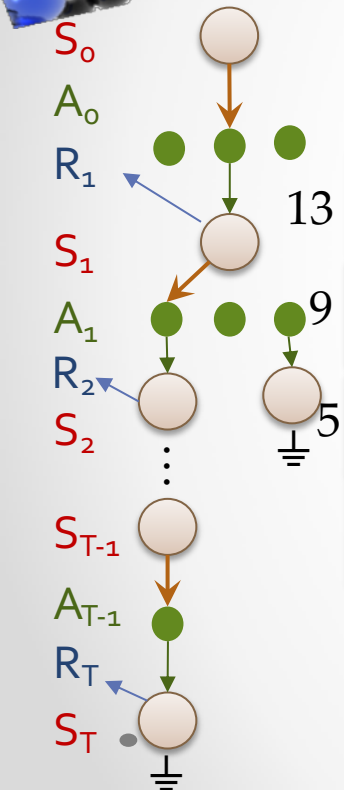




State	Action	Probability
1	U	1.0
2	R	1.0
3	U	1.0
4	L	
5	U	
6		
7	U	
8		
9	R	
10	R	
11	U	
12		
13		
14	R	
15	R	
16		

State-Value Function





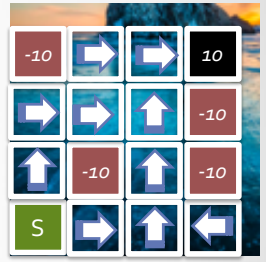
Given a policy, we can compute the value at each state using the Bellman Equation

$G_t = R_{t+1} + G_{t+1}$

On

$V_\pi(s) = R_{t+1} + V_\pi(s_{t+1} | s_t=s)$

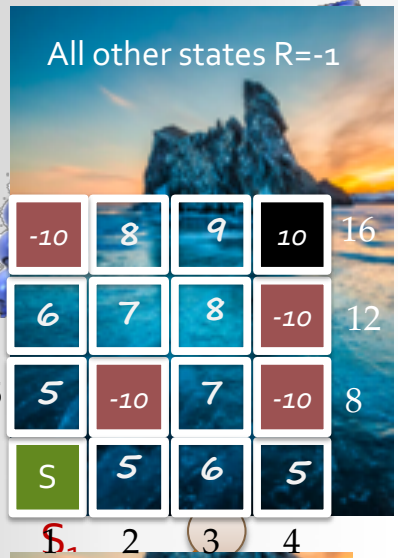
As shown in the diagram

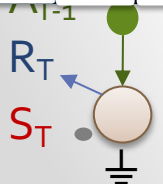


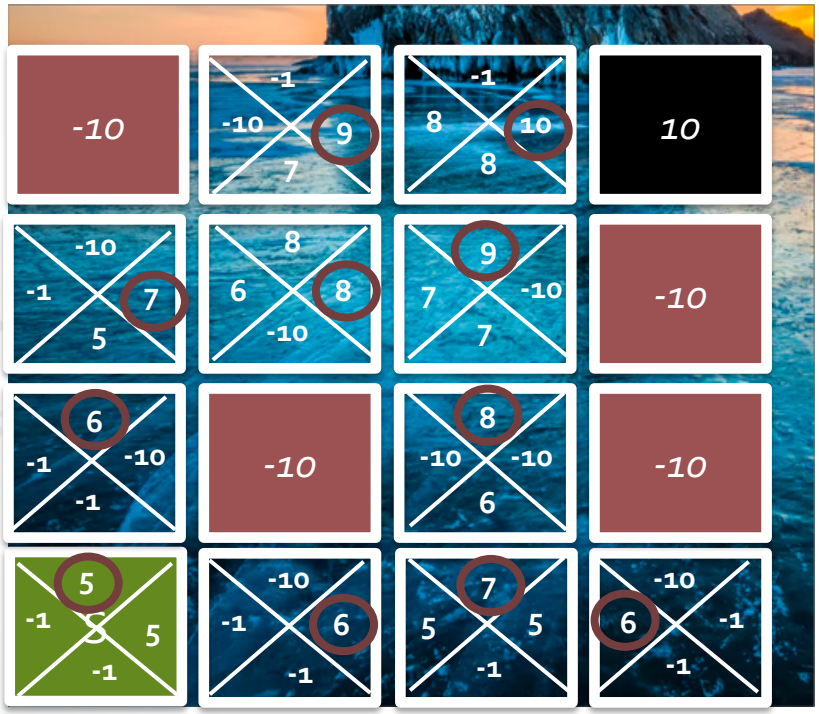
Of course, we are glossing over a lot of details, which we will explore during the rest of the lab

State-Action-Value Function

Iterative Policy Evaluation & Improvement







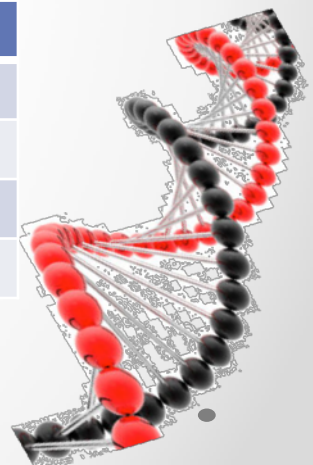
$q_\pi(s,a) = v_\pi(s_{t+1}, a_t | s_t=s, a_t=a)$

$\pi(a|s) = q_*(s,a) = \text{argmax}_{a'}(q(s,a'))$

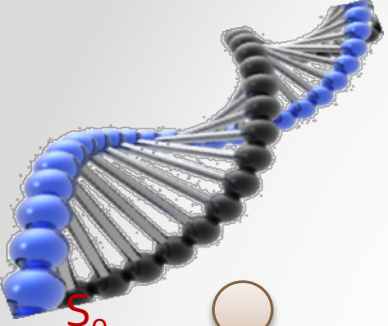
$v_\pi(s) = R_s + q_\pi(s, \pi(s))$

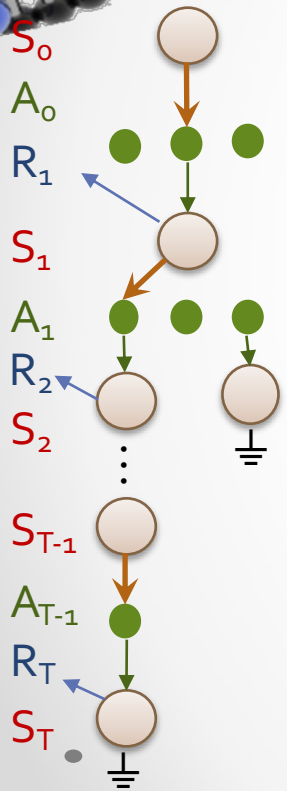
Q-Table

s,a	Q-Value
2,L	-1
2,D	-1
2,U	-10
2,R	6

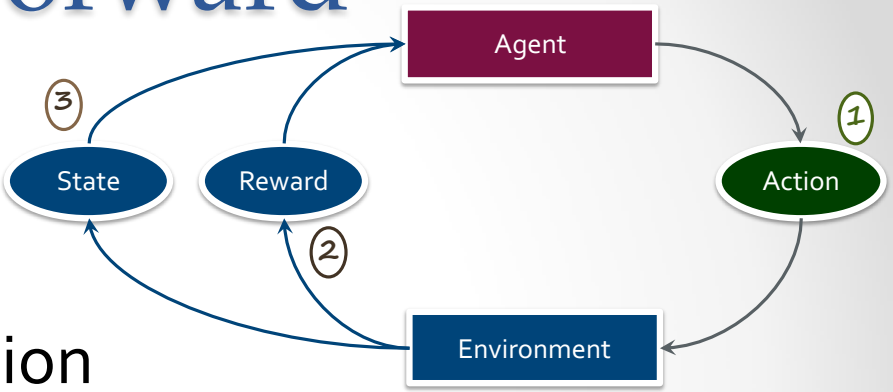


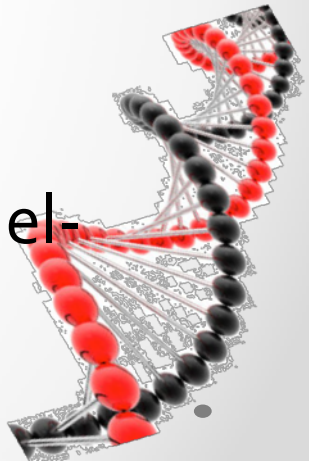
Recap & Forward



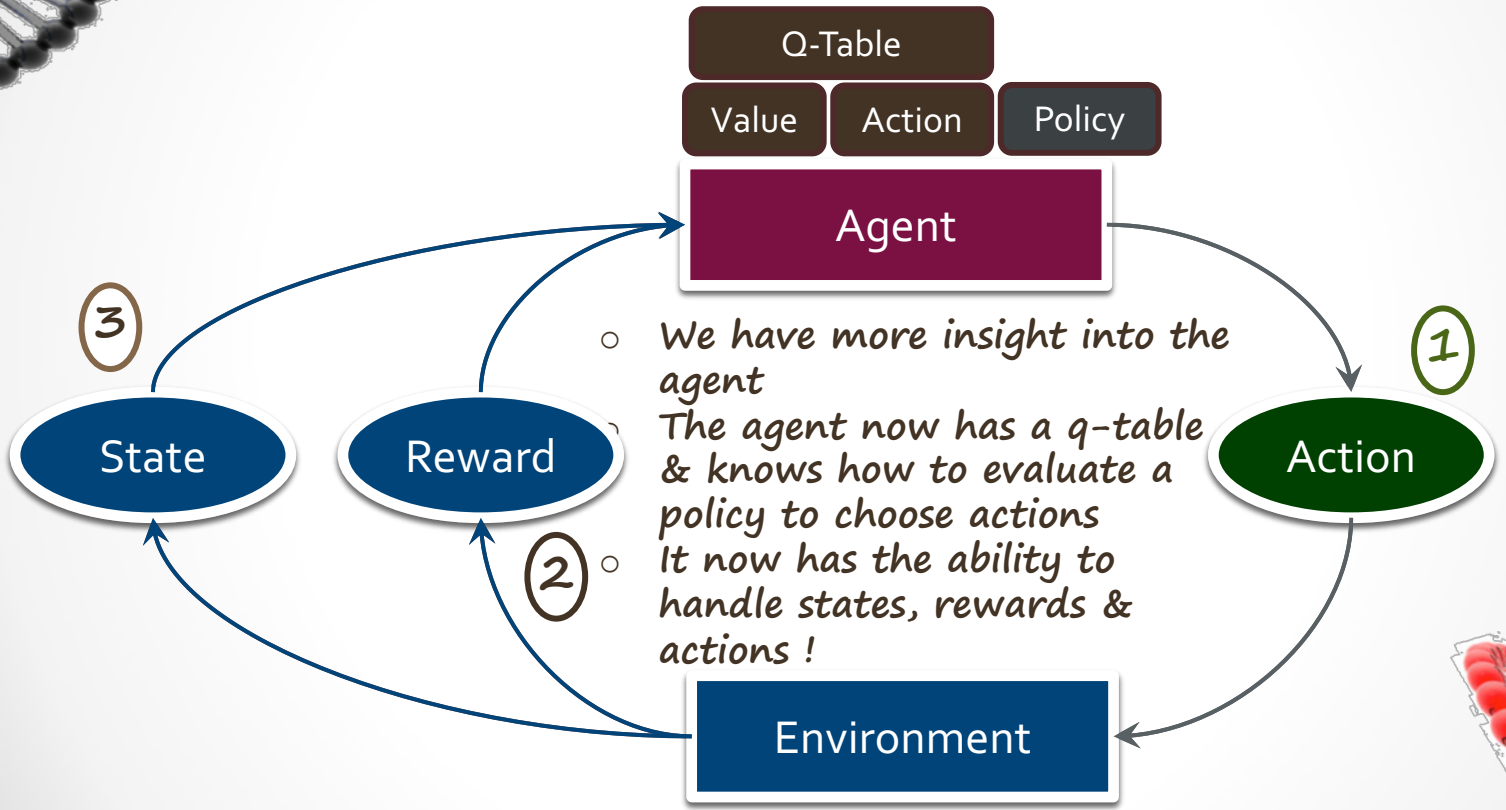


- RL Cycle
- Backup Diagram
- Value Function
- Action-Value Function
  - Value functions define a partial ordering of policies
- Policy – greedy policy ($argmax_{a'}$)
  - state sweep
- Grid World as an example
- Used Dynamic Programming in a model-based fashion
  - Decomposition & Aggregation





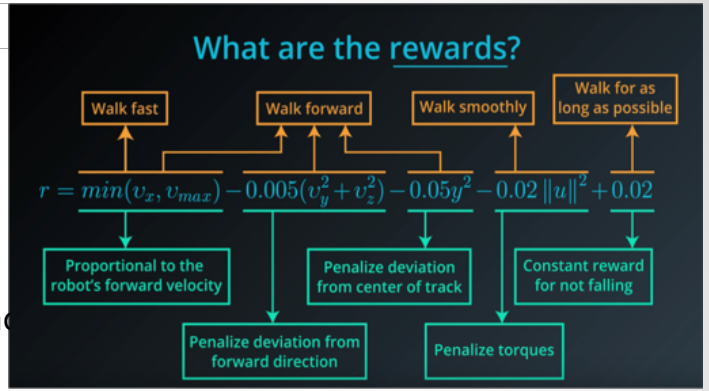
RL Flow



- We have more insight into the agent
- The agent now has a q-table & knows how to evaluate a policy to choose actions
- It now has the ability to handle states, rewards & actions !

Examples of representation

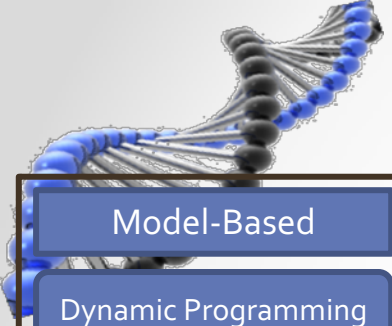
- Atari
  - Grey-scale, 84 X 84,
  - Stacking 4 frames for context
  - CNN as feature extractor



- Walking Humanoids
  - https://www.theverge.com/tldr/2017/7/10/15946542/deepmin learning
  - Emergence of Locomotion Behaviours in Rich Environments
  - https://arxiv.org/abs/1707.02286
  - Algorithms : PPO, DPPO,A3C
  - States – Position & Velocities of the joints, the height of the walker body above the ground, the difference between the agents position and the next sampling grid center
  - Actions : Forces and directions on the 21 actuated joints with 28 DoF
  - Rewards :
    - The reward consists of a main component proportional to the velocity along the x-axis, encouraging the agent to make forward progress along the track, plus a small term penalizing torques. For the walker the reward also includes the same box constraints on the pose as in se of the tr

$$r = \min(v_x, v_{\max}) - 0.005(v_x^2 + v_y^2) - 0.05y^2 - 0.02\|u\|^2 + 0.02$$

RL Algorithm Taxonomy

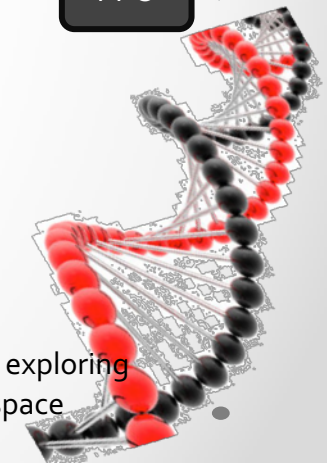


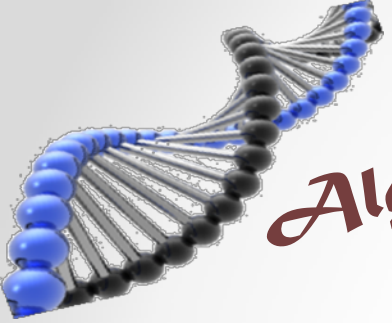
- Model-Based
  - Dynamic Programming
    - Policy Iteration
    - Value Iteration
- Model-Free
  - Monte Carlo
  - TD Learning
  - Value Methods
    - SARSA
    - Q-Learning
    - Deep Q-Learning
      - Fixed Q Targets
      - Experience replay
      - Double DQN
      - Prioritized replay
      - Dueling DQN
      - *Increase robustness*
      - *Reduce bias/variance*
      - *Sample efficiency*
  - Policy Search
    - Hill Climbing
      - Steepest Hill Climbing
        - Cross-entropy
        - Evolution Strategies
      - Random Restart
      - Simulated Annealing
      - Adaptive noise scaling
  - Policy Gradients
    - REINFORCE
    - TRPO
    - PPO
    - Actor-Critic
    - DDPG

RL ⟷ DRL

Non-linear Function Approximation

- Why Policy Based ?
  - Simplicity
  - Stochastic Action Space
    - Rock-Paper-scissor
  - Continuous
    - Can output a # without exploring values in a continuous space
  - Aliased States



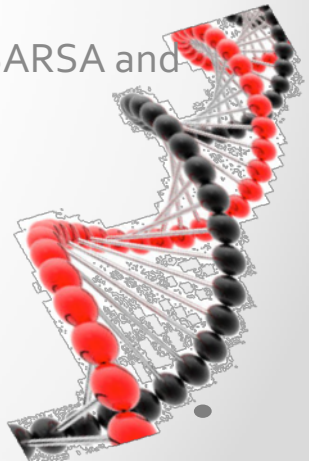


Algorithm Time

PyTorch Time

3. Algorithms & Programming : From Monte Carlo to DQN

Let us implement four algorithms and in the process discuss Monte Carlo, TD(), SARSA and Q-Learning



Notations & Definitions - Review

- Episode

  Sequence of State-Action-Reward until an episode ends in T steps

- Return

  $G_t = \Sigma_{k=0 \text{ to } \infty} \gamma^k R_{t+k+1} = R_{t+1} + \Sigma_{k=1 \text{ to } \infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1}$

- Policy

  Mapping $\pi : S \times A \mapsto [0,1]$ $\quad \pi(a|s) = \mathbb{P}(A_t=a, S_t=s)$

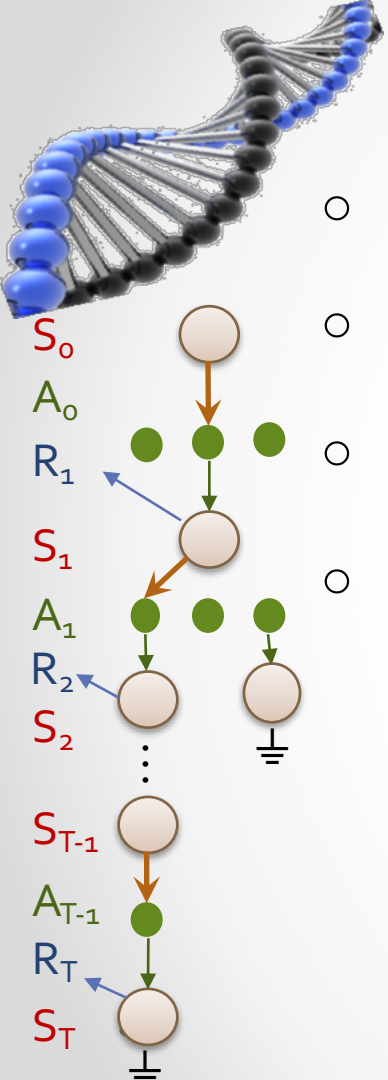
- Value

  $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t=s]$ … state-value Function

  $= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}(s_{t+1}| s_t=s)]$

  $= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(s_{t+1}| s_t=s)]$

  $q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t=s, A_t=a]$ … action-value function



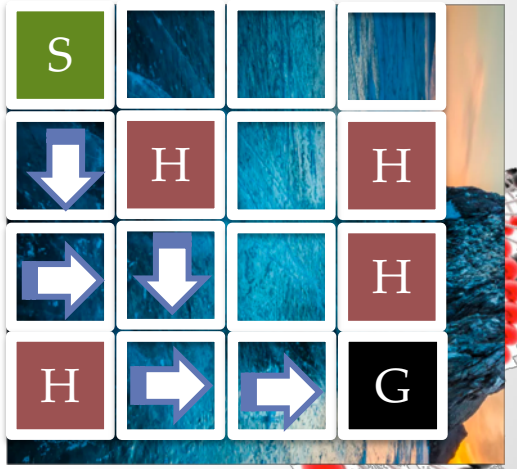
Hands On #1 - Skating the Frozen Lake

Goal:

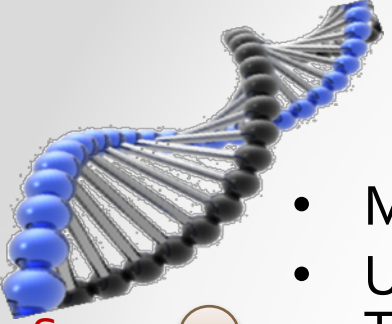
- Get familiar with programming OpenAI Gym & it's interfaces
- Solve OpenAI Gym : Frozen Lake, given an optimum deterministic policy
- Frozen Lake deterministic : episodes before solve

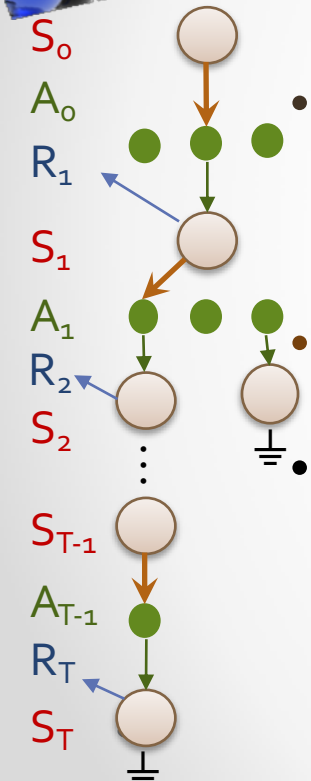
Steps:

1. Notebook : frozen_lake_01.ipynb
2. OpenAI Setup & interface
3. Random Policy Run
4. Use deterministic policy to skate to the goal
5. Metrics
   - Episodes before solve
   - Solved 0.78 over 100 consecutive runs
   - Max score over 1000 episodes
   - Avg over 1000 episodes

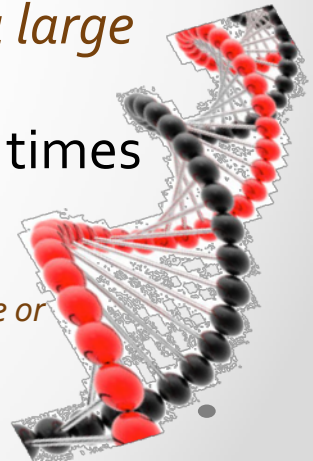


Monte Carlo





- Model-free method
- Usually we do not have the model or the transition matrix. Then how do we arrive at G_t, $q_\pi(s,a)$ *et al ?*
- Monte Carlo brings deterministic expectation by repeated random sampling. i.e. *even though the underlying problem involves a great degree of randomness, we can infer useful information we can trust by collecting lots of samples*
- *Monte Carlo estimates a value by averaging across a large number of episodes*
- Remember, a state-action might be visited several times and the q-value can and will be different.
  - *Averaging gives the best estimate across a large number of episodes*
  - *Either average (across episodes) considering only during 1st visit in an episode or*
  - *Consider all visits and average the value*



Monte Carlo Algorithm

1. Initialize variables, set policy π
2. Generate episodes using policy π - $S_0 A_0 R_1 S_1 A_1 R_2 S_2 \ \ldots \ A_{T-1} R_T S_T$
3. For each episode
   4. For each time step
      5. For each state, action pair
         6. If 1st visit to (S_t, A_t)
            7. Calculate the return G_t - $G_t = \Sigma_{k=0 \text{ to } \infty} \gamma^k R_{t+k+1}$
            8. Add G_t to the return column - $\Sigma q(s,a)$
            9. Increment $N(S_t, A_t)$
10. Average $G_t(S_t, A_t)$/ $\Sigma q(s,a)$ & return the Q table q(s,a)

$q(s,a) \mapsto q_\pi(s,a)$ *for a larger number of episodes*

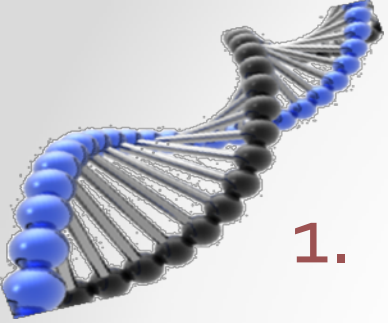
(S,A)	Σq(s,a)	N
1,0	-4	4
1,1	5	1
1,2	10	2
1,3	-2	2
2,0	-20	20
2,1	-10	1
2,2	4	24
..		

Algorithm 9: First-Visit MC Prediction (*for action values*)

Input: policy π, positive integer *num_episodes*
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize $N(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Initialize $returns\_sum(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for $i \leftarrow 1$ **to** *num_episodes* **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
  for $t \leftarrow 0$ **to** $T-1$ **do**
    if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
      $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
      $returns\_sum(S_t, A_t) \leftarrow returns\_sum(S_t, A_t) + G_t$
  end
end
$Q(s,a) \leftarrow returns\_sum(s,a)/N(s,a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
return Q

Algorithm 9: First-Visit MC Prediction (*for action values*)

Input: policy π, positive integer *num_episodes*
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize $N(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Initialize $returns\_sum(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
  for $t \leftarrow 0$ ***to*** $T-1$ **do**
    if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
      $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
      $returns\_sum(S_t, A_t) \leftarrow returns\_sum(S_t, A_t) + G_t$
  end
end
$Q(s,a) \leftarrow returns\_sum(s,a)/N(s,a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
return Q



Couple of optimizations

1. Instead of

Increment counter $N(s) \leftarrow N(s) + 1$

Increment total return $S(s) \leftarrow S(s) + G_t$

Value is estimated by mean return $V(s) = S(s)/N(s)$

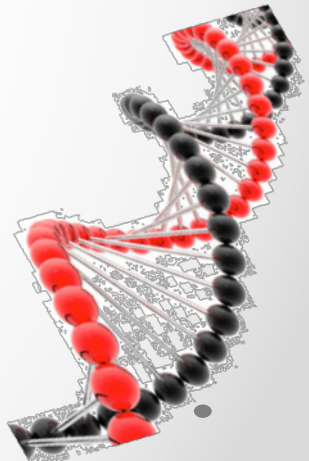
   - Do this

$$N(S_t) \leftarrow N(S_t) + 1$$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

2. Track a running mean

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$



The mean $\mu_1, \mu_2, \ldots$ of a sequence $x_1, x_2, \ldots$ can be computed incrementally,

$$\begin{aligned}
\mu_k &= \frac{1}{k} \sum_{j=1}^{k} x_j \\
&= \frac{1}{k} \left(x_k + \sum_{j=1}^{k-1} x_j \right) \\
&= \frac{1}{k} \left(x_k + (k-1)\mu_{k-1} \right) \\
&= \mu_{k-1} + \frac{1}{k} \left(x_k - \mu_{k-1} \right)
\end{aligned}$$

Policy !

- We glossed over 2 points
- Where does the π come from ?
- How do we get to

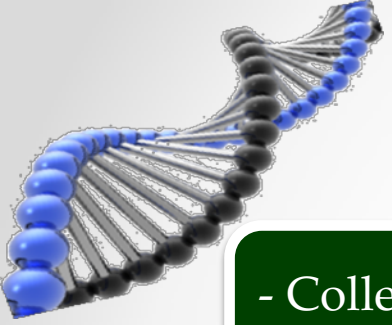
- From a Q table ?

(S,A)	Σq(s,a)	N
1,0	-4	4
1,1	5	1
1,2	10	2
1,3	-2	2
2,0	-20	20
2,1	-10	1
2,2	4	24
..		

Algorithm 9: First-Visit MC Prediction (*for action values*)

Input: policy π, positive integer *num_episodes*
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize $N(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Initialize $returns\_sum(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
  for $t \leftarrow 0$ ***to*** $T-1$ **do**
    if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
      $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
      $returns\_sum(S_t, A_t) \leftarrow returns\_sum(S_t, A_t) + G_t$
  end
end
$Q(s,a) \leftarrow returns\_sum(s,a)/N(s,a)$ for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
return Q

- *i.e. how does this all help us to make better actions ?*



Policy Evaluation-Improvement Cycle

- Collect episodes using π
- Create Q Table

Policy Evaluation

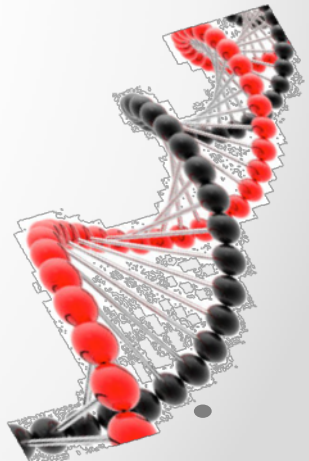
The q(s,a) tells us the utility of a policy

- Construct better policy π' based on the Q values
- (optionally) Set $\pi = \pi'$
- Take action using π'

Policy Improvement

$$\pi_0 \mapsto v_{\pi 0} \mapsto \pi_1 \mapsto v_{\pi 1} \mapsto .. .. .. \mapsto \pi_* \mapsto v_*$$

- π – Behavior Policy (sample experience)
- π' – Target Policy (to optimize behavior)
- Most Common Policies
  - Equiprobable
  - Greedy
  - ε-Greedy (Greedy + Equiprobable)
  - Softmax distribution



Policy Algorithms

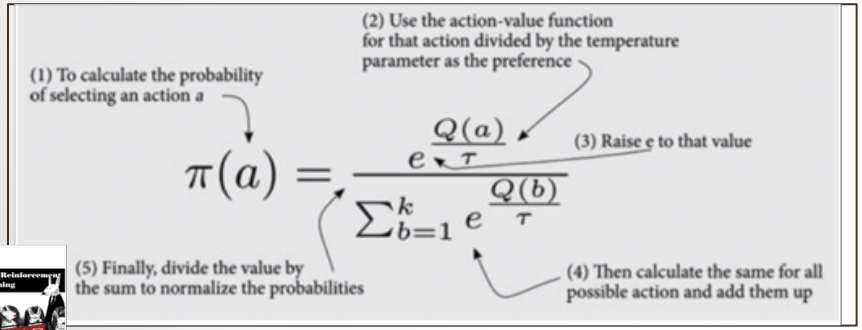
Equiprobable

- Random Uniform
- All actions equally possible
- *Good starting point, in an unknown environment*

Greedy

- Follow the action with the max value
- $\pi(a \mid s) = q_*(s,a) = \text{argmax}_{a'}(q(s,a'))$
- *Pure exploit*

Softmax



ε-Greedy

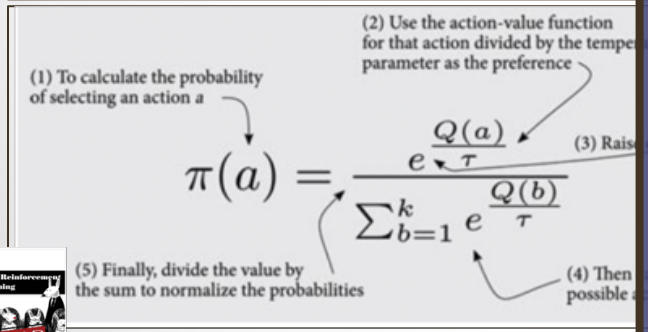
- With ε follow equiprobable
- With 1-ε follow greedy
- *Gives exploration*
- Decay ε as the gent learns more

Policy Algorithms

Equiprobable

- Random Uniform
- All actions equally possible
- *Good starting point, in an unknown environment*

Softmax



Softmax

- *A weighted probability across actions, establishing a partial ordering*
  - *A weighted choice will establish the order during action selection e.g. random.choice(arr, p=Pr_array)*
- *τ = temperature, that scales the probability distribution of actions.*
  - *A high temperature will tend the probabilities to be very similar, whereas*
  - *A low temperature will exaggerate differences in probabilities between actions*
  - *Initially, when the values are inaccurate, we are Ok with small difference between the values i.e. uniform random*
  - *Decay τ such that as our values get better we want meaningful difference, yet have a small exploration possibility*

Policy Algorithms

Equiprobable

- Random Uniform

$$\pi(a|s) \leftarrow \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|} & \text{if } a = \arg\max_{a' \in \mathcal{A}(s)} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}(s)|} & \text{otherwise} \end{cases}$$

Quiz:

1. *Which ε will always be greedy ?*
2. *Always not greedy ?*
3. *Is equiprobable a special case of ε-Greedy ? If so, what is ε for ε-Greedy Equiprobable ?*
4. *Which guarantees non greedy ?*
   ε=0 ? ε=0.25 ? ε=0.75 ? ε=0.99 ? ε=1 ?
5. *What is a good ε_{start} ? ε_{min} for decay ?*
6. *What is a good policy for Rock-Paper-Scissor ? How will it work ?*

value

- $\pi(a \mid s) = q_*(s,a) = \text{argmax}_{a'}(q(s,a'))$
- *Pure exploit*

ε-Greedy

- With ε follow equiprobable
- With 1-ε follow greedy
- *Gives exploration*
- Decay ε as the agent learns more

Algorithm 10: First-Visit GLIE MC Control

Input: positive integer *num_episodes*, GLIE $\{\epsilon_i\}$
Output: policy π ($\approx \pi_*$ if *num_episodes* is large enough)
Initialize $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Initialize $N(s,a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  $\epsilon \leftarrow \epsilon_i$
  $\pi \leftarrow \epsilon$-greedy(Q)
  Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
  for $t \leftarrow 0$ ***to*** $T-1$ **do**
    if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
      $N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$
      $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$
  end
end
return π

1. Greedy with limited Infinite Exploration
2. Construct a policy that is greedy w.r.t the Q table
3. Decay ε as the agent learns more
4. But sill maintain a small exploration

Algorithm 11: First-Visit Constant-α (GLIE) MC Control

Input: positive integer *num_episodes*, small positive fraction α, GLIE $\{\epsilon_i\}$
Output: policy π ($\approx \pi_*$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
  $\epsilon \leftarrow \epsilon_i$
  $\pi \leftarrow \epsilon$-greedy(Q)
  Generate an episode $S_0, A_0, R_1, \ldots, S_T$ using π
  for $t \leftarrow 0$ ***to*** $T-1$ **do**
    if (S_t, A_t) *is a first visit (with return* G_t*)* **then**
      $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$
  end
end
return π

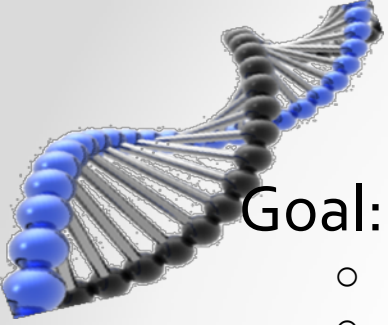
α Mean

$\alpha = 0$, never learns

$\alpha = 1$, never remembers !

α = small, larger history, learns sowly

α = large, focuses more on recent experience, but potentially difficult to converge



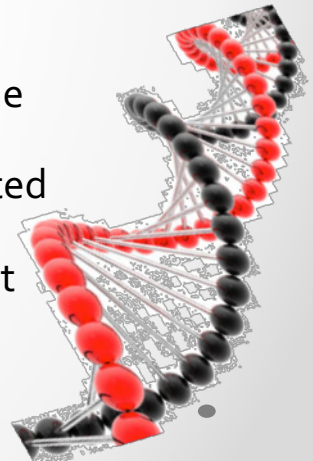
Hands On #2 – Monte Carlo

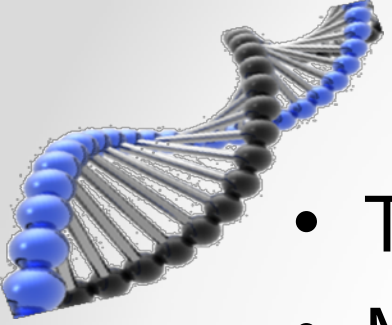
Goal:

- Implement Monte Carlo
- Explore different policies solving OpenAIGym : FrozenLake

Steps:

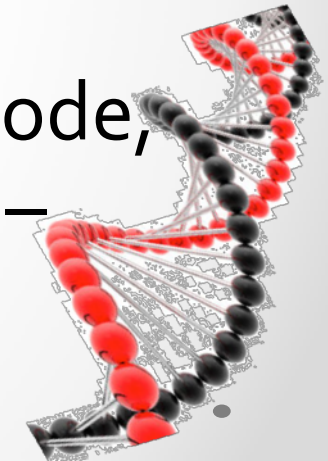
1. Notebook : Monte_Carlo_01.ipynb
2. Program Monte Carlo – 1st Visit or every visit
3. Try different ε-greedy Policies
   - GLIE – decay ε as decreasing function of episodes
   - Exponentially decay ε
     - The ϵ decay is to incorporate the learning and make the action more deterministic
   - We could use softmax and keep the policy constant ie weighted random.
     - As it learns, it's belief of the grid world changes and that shows up in the preferences



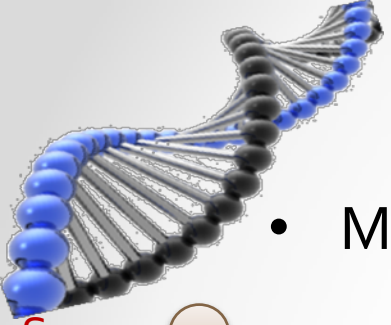


TD

- Temporal Difference Algorithm
- Motivation
  - Monte Carlo requires full episodes
  - MC has high Variance (but low Bias)
  - MC : Slow to converge
  - MC : Easy to understand
- So instead of waiting for a full episode, update the values after every step – TD(0) or every n steps – TD(ƛ)

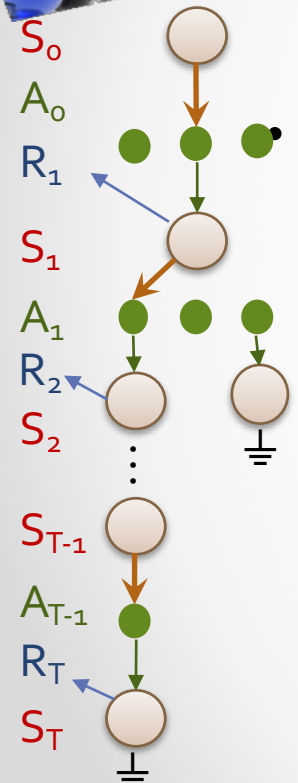


TD vs. MC



- MC : $Q(S_t, A_t) = Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t))$

We observe from a full episode & update all

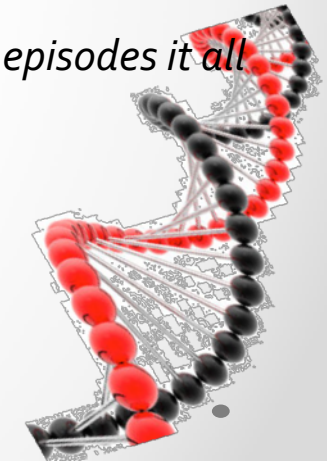


- TD : $Q(S_t, A_t) = Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$

TD Target = Alternate Estimate | Current Estimate

TD Error

  - *TD Target*
  - *TD Error*
  - *At every timestep, so we have R_{t+1} not G_t*
  - *We bootstrap $Q(S_{t+1}, A_{t+1})$, but as we take more steps and more episodes it all converges*
  - *TD learns from incomplete episodes by bootstrapping*
  - *TD lends itself to function approximation*
    - *And scale to large state space*
  - *Remember, we have to select S_{t+1} & A_{t+1}*
    - *That is policy & more later*



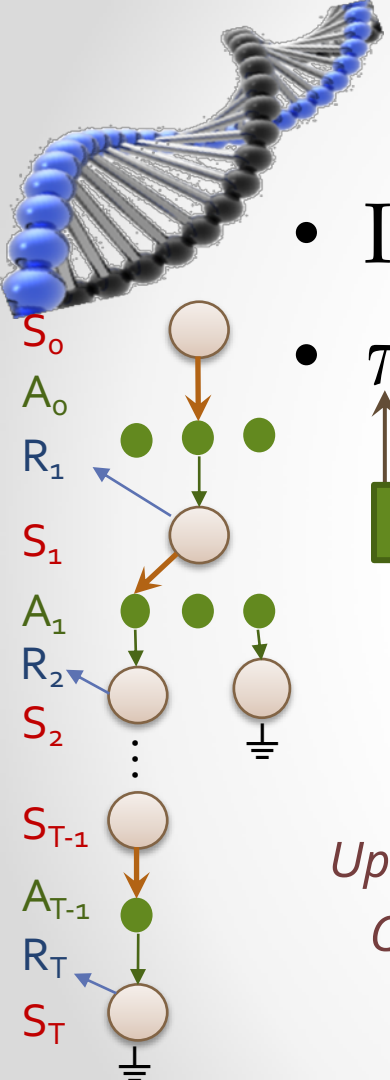
TD Intuition from Sutton Book/D.SLiver

Driving Home Example

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office	0	30	30
reach car, raining	5	35	40
exit highway	20	15	35
behind truck	30	10	40
home street	40	3	43
arrive home	43	0	43

SARSA(0)

- Initialize $Q(s,a)\ \forall\ s \in \mathbb{S},\ a \in \mathbb{A}$
- π = ε-Greedy(Q) { $argmax_{a'}(q(s,a'))$ }



$S_0\ A_0\ R_1\ S_1\ A_1\ R_2\ S_2\ A_2\ \ldots\ A_{T-1}\ R_T\ S_T$

Update $Q(S_0,A_0)$
π = ε-Greedy(Q)

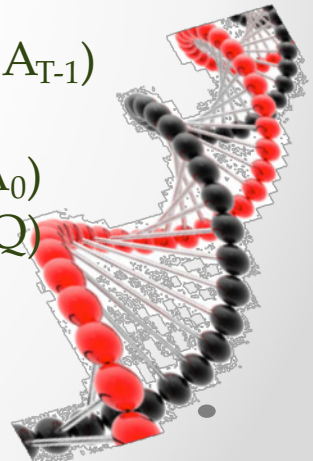
Update $Q(S_1,A_1)$
Update $Q(S_0,A_0)$
π = ε-Greedy(Q)

Update $Q(S_{T-1},A_{T-1})$
..
Update $Q(S_0,A_0)$
π = ε-Greedy(Q)

Update Rule:

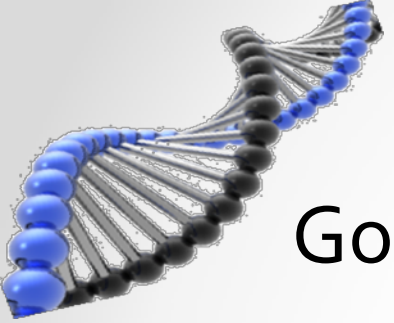
$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha\,(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

A_{t+1} *selected using* ε*-Greedy(Q)*



Algorithm 13: Sarsa

Input: policy π, positive integer *num_episodes*, small positive fraction α, GLIE $\{\epsilon_i\}$
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\textit{terminal-state}, \cdot) = 0$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
    $\epsilon \leftarrow \epsilon_i$
    Observe S_0
    Choose action A_0 using policy derived from Q (e.g., ϵ-greedy)
    $t \leftarrow 0$
    repeat
        Take action A_t and observe R_{t+1}, S_{t+1}
        Choose action A_{t+1} using policy derived from Q (e.g., ϵ-greedy)
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$
        $t \leftarrow t + 1$
    until S_t *is terminal*;
end
return Q



Hands On #3 – SARSA

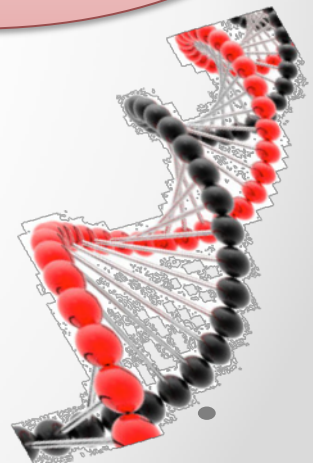
Goal:

- Implement SARSA for Frozen lake

Steps:

1. Program SARSA
2. Plot Value

Skip because we will work on $SARSA_{max}$ next



Q-Learning

SARSA Update Rule:

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

A_{t+1} : *Selected using* ε*-Greedy(Q) for update*

A_{t+1} : *Selected using* ε*-Greedy(Q) for evaluation*

Q Learning Update Rule:

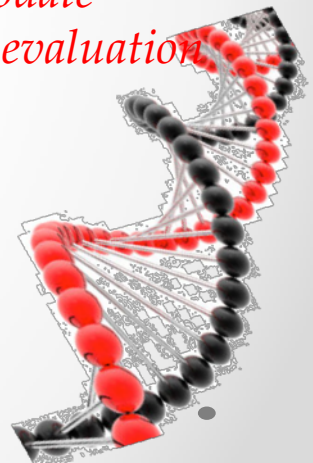
$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha (R_{t+1} + \gamma \max_{a \in \mathbb{A}} Q(S_{t+1}, a) - Q(S_t, A_t))$$

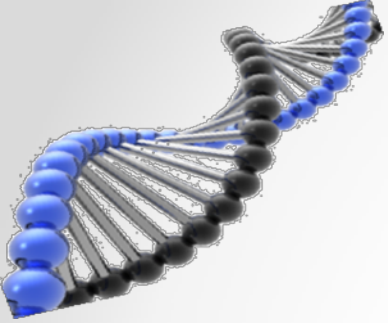
$\max_{a \in \mathbb{A}} Q(S_{t+1}, a)$: *Selected using Greedy(Q) for update*

A_{t+1} : *Selected using* ε*-Greedy(Q) for evaluation*

- SARSA : Behavior = Target ↦ On-Policy
- Q-Learning : Behavior ≠ Target ↦ Off-Policy

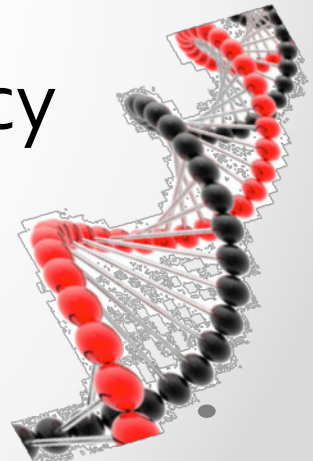
- *Learn Optimal policy while following exploratory policy*
- *Reuse data collected/Experience Replay/Prioritized Replay*





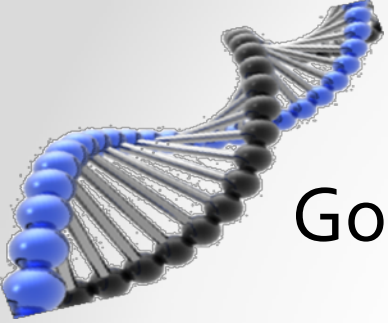
Quiz

- This algorithm learns the best exploratory policy
  - What is SARSA ?
- This algorithm learns the optimum policy
  - What is Q-Learning ?



Algorithm 14: Sarsamax (Q-Learning)

Input: policy π, positive integer *num_episodes*, small positive fraction α, GLIE $\{\epsilon_i\}$
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\textit{terminal-state}, \cdot) = 0$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do**
    $\epsilon \leftarrow \epsilon_i$
    Observe S_0
    $t \leftarrow 0$
    repeat
        Choose action A_t using policy derived from Q (e.g., ϵ-greedy)
        Take action A_t and observe R_{t+1}, S_{t+1}
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$
        $t \leftarrow t + 1$
    until S_t *is terminal*;
end
return Q



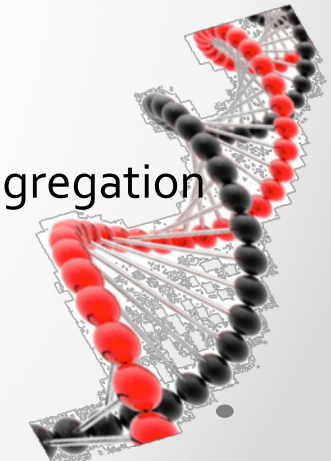
Hands On #4 – Q Learning

Goal:

- Introduce the CartPole Environment
  - *More complex, Continuous*
- Implement Q-Learning for digitized CartPole
  - *Later we will use function approximation & remove the requirement for digitization*

Steps:

1. Notebook : Q_Learning_01.ipynb
2. Get familiar with Cartpole environment
   - np.linspace() and np.digitize() for state space aggregation
3. Program Q Learning
4. Track & Plot Metrics to solve Cart Pole

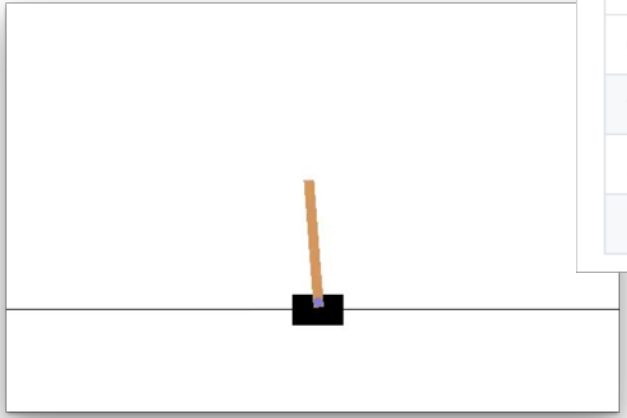


Introducing the CartPole

Balance an Inverted Pendulum by applying force on the cart

1

2



Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

Num	Action
0	Push cart to the left
1	Push cart to the right

4

Solved after 211 episodes. Best 100-episode average reward was 195.27 ± 1.53. (CartPole-v0 is considered "solved" when the agent obtains an average reward of at least 195.0 over 100 consecutive episodes.)

211	500	✓	83s	↓	
Episodes to solve	Total episodes	Solved	Time to solve	Download	Tweet

Episode Termination

3

1. Pole Angle is more than ±12°
2. Cart Position is more than ±2.4 (center of the cart reaches the edge of the display)
3. Episode length is greater than 200

State: $\{x, \dot{x}, \theta, \dot{\theta}\}$

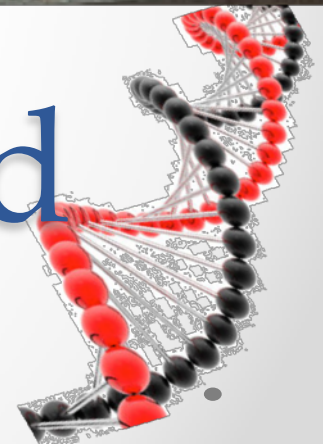
2 Discrete Actions: Apply force to cart left/right

Rewards: +1 for each step

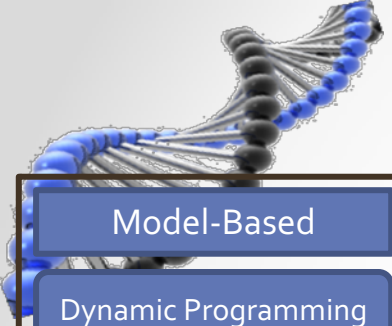
https://gym.openai.com/envs/CartPole-v0/

https://github.com/openai/gym/wiki/CartPole-v0

Down the DRL-Land



RL Algorithm Taxonomy

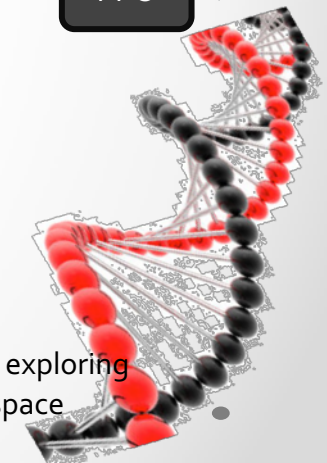


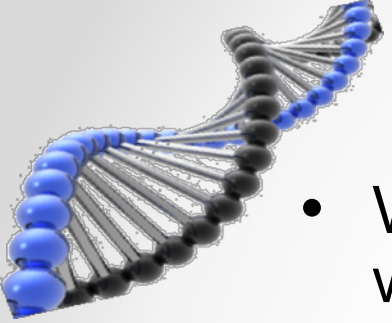
- Model-Based
  - Dynamic Programming
    - Policy Iteration
    - Value Iteration
- Model-Free
  - Monte Carlo
  - TD Learning
  - Value Methods
    - SARSA
    - Q-Learning
    - Deep Q-Learning
      - Fixed Q Targets
      - Experience replay
      - Double DQN
      - Prioritized replay
      - Dueling DQN
      - *Increase robustness*
      - *Reduce bias/variance*
      - *Sample efficiency*
  - Policy Search
    - Hill Climbing
      - Steepest Hill Climbing
        - Cross-entropy
        - Evolution Strategies
      - Random Restart
      - Simulated Annealing
      - Adaptive noise scaling
  - Policy Gradients
    - REINFORCE
    - TRPO
    - PPO
    - Actor-Critic
    - DDPG

RL ⟷ DRL

Non-linear Function Approximation

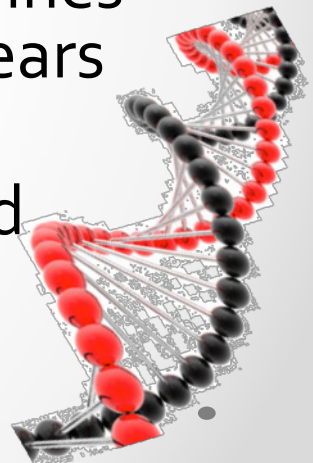
- Why Policy Based ?
  - Simplicity
  - Stochastic Action Space
    - Rock-Paper-scissor
  - Continuous
    - Can output a # without exploring values in a continuous space
  - Aliased States



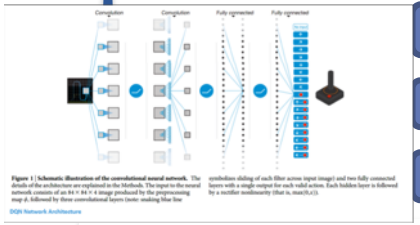


Deep reinforcement Learning

- We have interesting algorithms, but they won't scale easily for large state spaces or for continuous state spaces
  - For example, GO has possibly 10^{170} states, more than the number of atoms in the observable universe (10^{80})
- Fortunately, Reinforcement Learning research has progressed so far that machines are able to win over Go champions - 10 years ahead of the predicted timeframe !
- The latest DOTA 5 uses 120,000 CPUs and 256 GPUs to win over human players in a complex strategic game !



DeepRL Schematic



Value

Action

Policy

Agent

State

Reward

Actions

Environment

- We will first look at the simple mechanism
  - Replace our Q-Table with a function approximation ie a Neural Network, to make it work on the top of DL structures
  - Then we can get values, actions or even policies
- Then we will look at extensions
- And finally, we will see how these ideas scale to games with video frames, games like Go with huge state spaces and complex semantics

Deep Q Learning

Algorithm: Deep Q-Learning

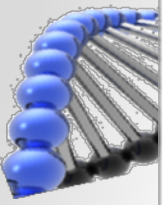
- Initialize replay memory D with capacity N
- Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
- Initialize target action-value weights $\mathbf{w}^- \leftarrow \mathbf{w}$
- **for** the episode $e \leftarrow 1$ to M:
  - Initial input frame x_1
  - Prepare initial state: $S \leftarrow \phi(\langle x_1 \rangle)$
  - **for** time step $t \leftarrow 1$ to T:

SAMPLE

Choose action A from state S using policy $\pi \leftarrow \epsilon\text{-Greedy}(\hat{q}(S, A, \mathbf{w}))$
Take action A, observe reward R, and next input frame x_{t+1}
Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
Store experience tuple (S, A, R, S') in replay memory D
$S \leftarrow S'$

LEARN

Obtain random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from D
Set target $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, \mathbf{w}^-)$
Update: $\Delta \mathbf{w} = \alpha(y_j - \hat{q}(s_j, a_j, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s_j, a_j, \mathbf{w})$
Every C steps, reset: $\mathbf{w}^- \leftarrow \mathbf{w}$

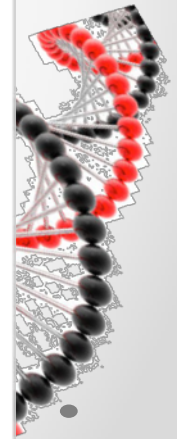


Input: the pixels and the game score
Output: Q action value function (from which we obtain a policy and select actions)

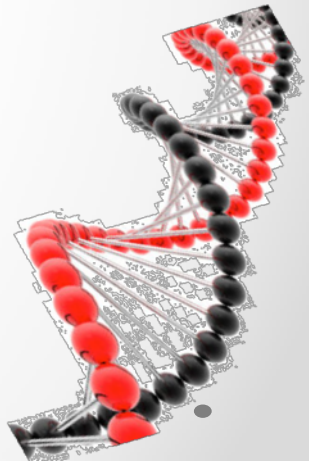
initialize replay memory D
initialize action-value function Q with random weight $\boldsymbol{\theta}$
initialize target action-value function $\hat{Q}$ with weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
for *episode* = *1 to* M **do**
  initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for t = *1 to* T **do**
    following ϵ-greedy policy, select $a_t = \begin{cases} \text{a random action} & \text{with probability } \epsilon \\ \arg\max_a Q(\phi(s_t), a; \boldsymbol{\theta}) & \text{otherwise} \end{cases}$
    execute action a_i 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 D
    `// experience replay`
    sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
    perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ w.r.t. the network parameter $\boldsymbol{\theta}$
    `// periodic update of target network`
    in every C steps, reset $\hat{Q} = Q$, i.e., set $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
  end
end

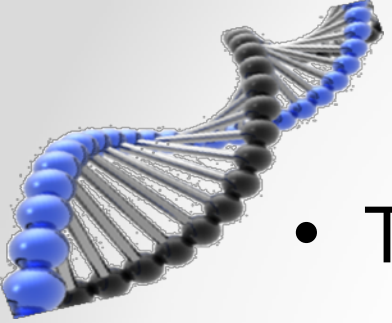
https://arxiv.org/pdf/1810.06339.pdf
Minh Paper : https://arxiv.org/abs/1602.01783

Algorithm 8: Deep Q-Nework (DQN), adapted from Mnih et al. (2015)

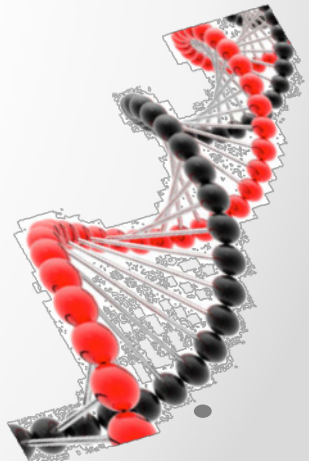


A detour – to pytorch





- TBD : Quick intro to pytorch



Hands On #5 – Deep Q Learning

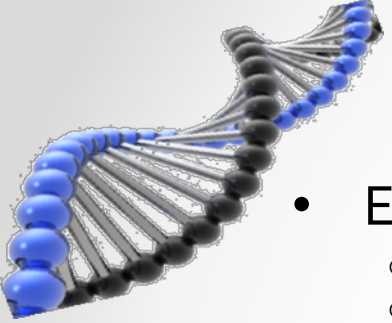
Goal:

- Implement Deep Q-Learning for CartPole
  - No digitization
  - Handle continuous state

Steps:

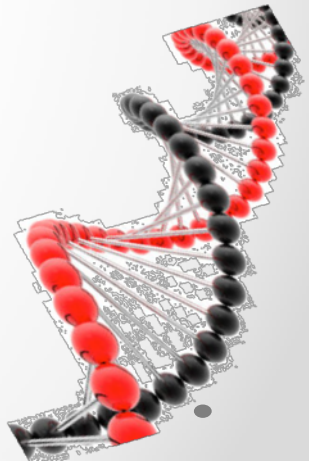
1. Notebook : DQN_01.ipynb
2. Program Deep Q Learning
3. Intro to pytorch
4. Metrics to solve Cart Pole
5. Plot Values

"Confidence is recursive ! You have to come thru a few times to be certain and confident that you can beat it !"



DeepQ Learning - Extensions

- Explain (Read papers and Explain):
  - Target Networks
  - Fixed Q Targets
  - Experience replay
    - Decorrelate
    - Data efficiency
  - Prioritized replay
    - Pay more attention to interesting experience
  - Double DQN
  - Dueling DQN
- Rainbow
  - To address challenges robustness & stability
  - Increase robustness, reduce bias/variance, sample efficiency
  - The Integrated Agent – Rainbow !!
    - https://arxiv.org/abs/1710.02298
  - Can Rainbow solve Pong ? Breakout ?

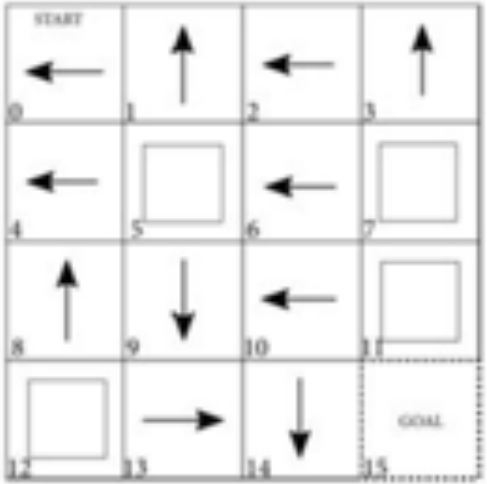


Hands On #6 – Skating slippery Frozen Lake with DQN

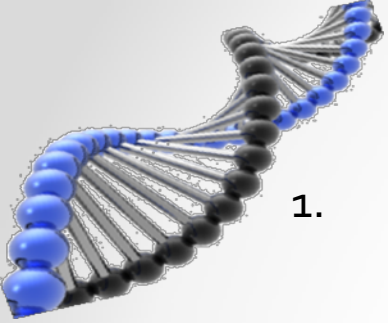
Yes, this is Frozen Lake's optimal policy!

And I wish you at least question the validity of it. How is it possible that in state 14 the optimal action is to go DOWN? Shouldn't it be RIGHT?

Since the Frozen Lake environment stochasticity is so large, it makes it a great example to illustrate the difficulty of sequential decision making under uncertainty of action effects. The fact is, the RIGHT action would send us to state 10, 14 or 15 with equal probability. While DOWN would send us to state 13, 14 or 15. If you think about it, state 13 is preferred to 10 because we can only get to state 15 safely going LEFT in 10, DOWN in 9, RIGHT in 13 and DOWN in 14. On the other hand, going DOWN in state 14 keeps us only a few steps away from 15 regardless of where the environment sends us.

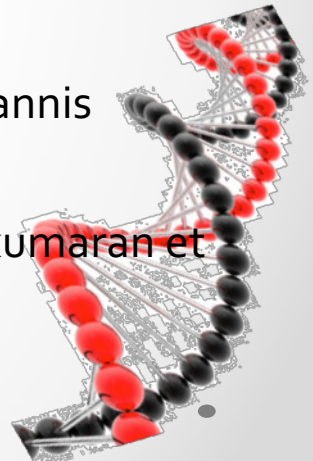


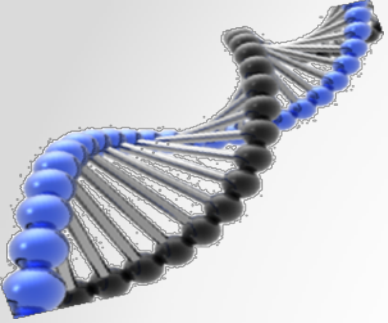
See if DQN-Rainbow finds this policy
Compare with DDPG



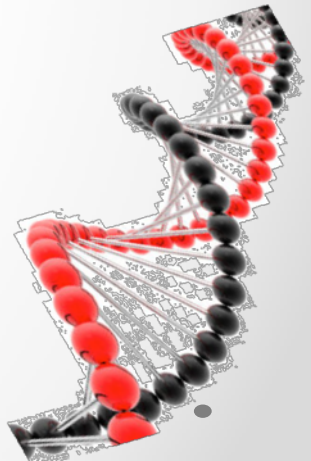
References

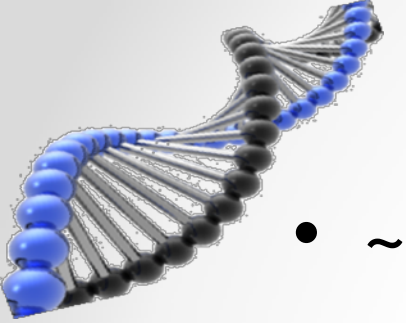
1. Human Level Control Through Deep Reinforcement Learning, Volodymyr Mnih et al
   https://deepmind.com/research/publications/human-level-control-through-deep-reinforcement-learning/
   - *We demonstrate that the deep Q-network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters*
2. Neural Fitted Q Iteration by Martin Riedmiller et al
   - https://qiita.com/Rowing0914/items/3d1b9a6ad4fe0a53cf5b
3. Deep Reinforcement Learning with Double Q-learning by Hado van Hasselt, Arthur Guez, David Silver
   - https://arxiv.org/abs/1509.06461
4. Prioritized Experience Replay by Tom Schaul, John Quan, Ioannis Antonoglou, David Silver
   - https://arxiv.org/abs/1511.05952
5. A Brief Survey of Deep Reinforcement Learning by Kai Arulkumaran et al
   - https://arxiv.org/abs/1708.05866



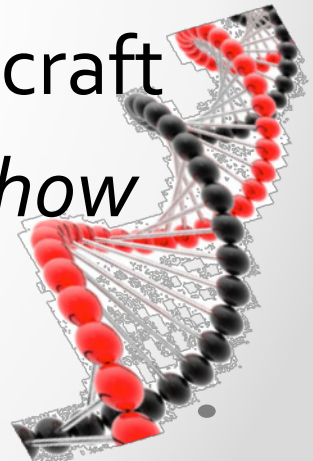


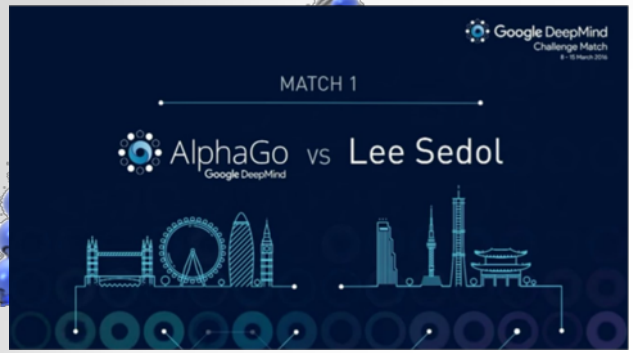
4. The world of Atari, α_{Go} & α_0





- ~5 slides on inplementations
- How states are represented, the agent architecture, hyper parameters for training et al
- For Atari, AlphaGo, DOTA and Starcraft
- *Gives the attendees an intuition on how RL is applied in real world*

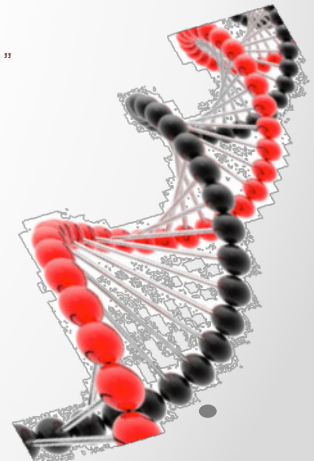




- AlphaGo AI program from Google's DeepMind defeated Go Champion
  - Lee Sedol (9-dan professional with 18 world titles)
- AlphaGo has the ability to look "globally" across a board—and find solutions that humans either have been trained not to play or would not consider.
  - This has huge potential for using AlphaGo-like technology to find solutions that humans don't necessarily see in other areas
- Search & Optimization
  - Policy Network with Values & search
  - "This ability of neural networks to bottle intuition / pattern recognition from one environment & apply to other contexts "

GENERATIONAL GAP !

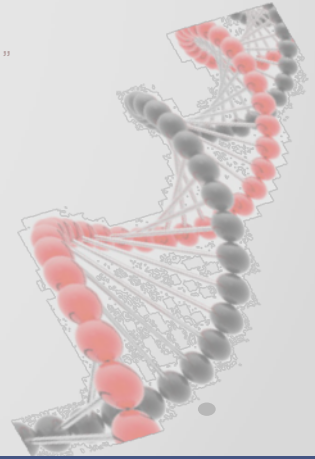
- *Unlike IBM's Deep Blue, which defeated chess champion Garry Kasparov in 1997, AlphaGo was not programmed with decision trees, or equations on how to evaluate board positions, or with if-then rules. "*
- *AlphaGo learned how to play go essentially from self-play and from observing big professional games*
  - *During training, AlphaGo played a million go games against itself.*

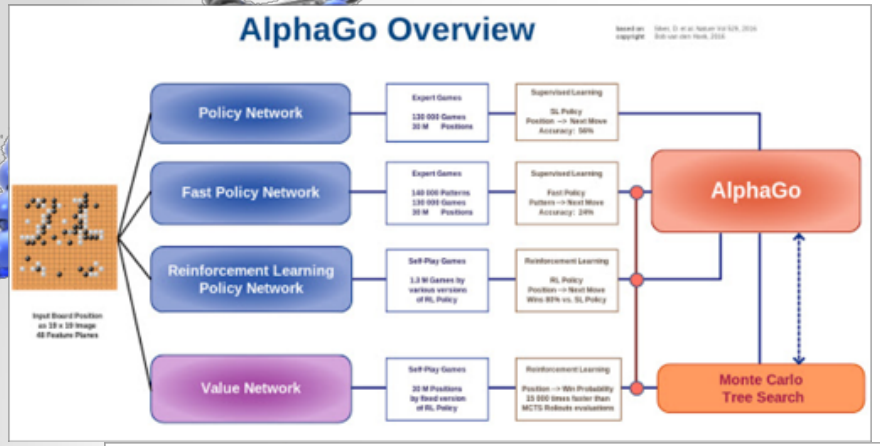


- AlphaGo AI program from Google's DeepMind defeated Go Champion
  - Lee Sedol (9-dan professional with 18 world titles)
- AlphaGo has the ability to look "globally" across a board—and find solutions that humans either have been trained not to play or would not consider.
  - This has huge potential for using AlphaGo-like technology to find solutions that humans don't necessarily see in other areas
- Search & Optimization
  - Policy Network with Values & search
  - "This ability of neural networks to bottle intuition / pattern recognition from one environment & apply to other contexts "

GENERATIONAL GAP !

- *Unlike IBM's Deep Blue, which defeated chess champion Garry Kasparov in 1997, AlphaGo was not programmed with decision trees, or equations on how to evaluate board positions, or with if-then rules. "*
- *AlphaGo learned how to play go essentially from self-play and from observing big professional games*
  - *During training, AlphaGo played a million go games against itself.*

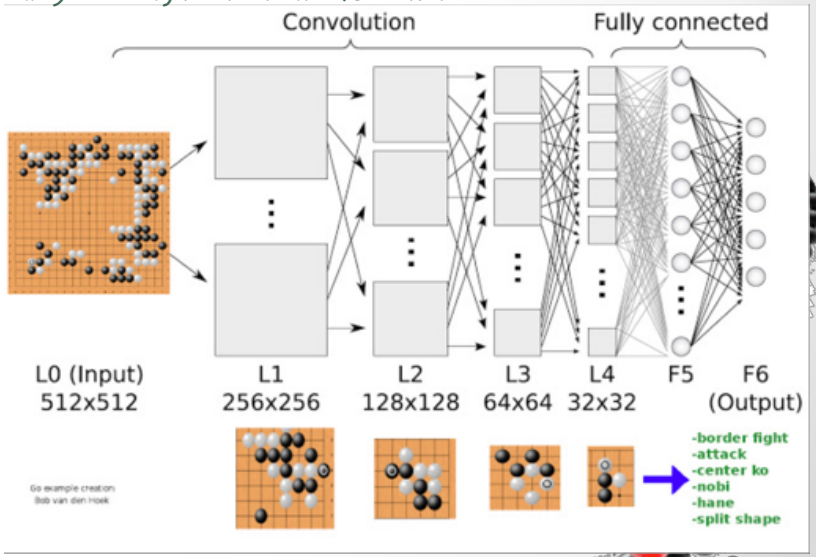


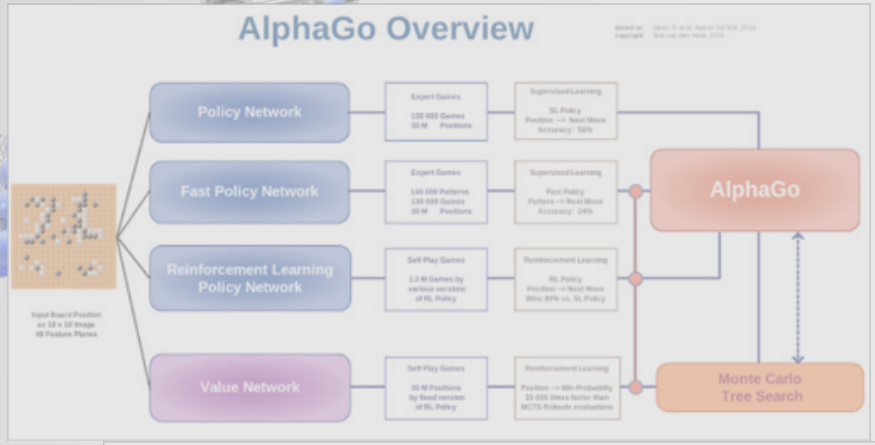


- AlphaGo is not a revolutionary breakthrough in itself, but rather a leading edge of an extremely important development
- #AlphaGo appeared to have a weakness in responding to unorthodox moves; Toward the end of the game, #AlphaGo played desperate moves similar to those of humans facing defeat …
- Ultimately, the scary thing about the rise of intelligent machines is not that they could someday have a mind of their own, but they could someday have a mind that we humans – with all our flaws and complexity – design and build for them

- It uses 4 networks :
- **Fast Rollout Policy Network** (P-Network) :It rolls out a quick plan for the game.
- **Supervised Learning Policy Network** (SL-Network) :The P-Network and the SL-Network are trained to predict human expert moves in a data set of positions.
- **Reinforcement Learning Policy Network** (RL-Network) :The RL Network is initialized as the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network.
- **Value network** : a Value Network V is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set.

https://medium.com/@arnabghosh93/deep-reinforcement-learning-driving-alphago-9881f1a90be4#.f462gnebl

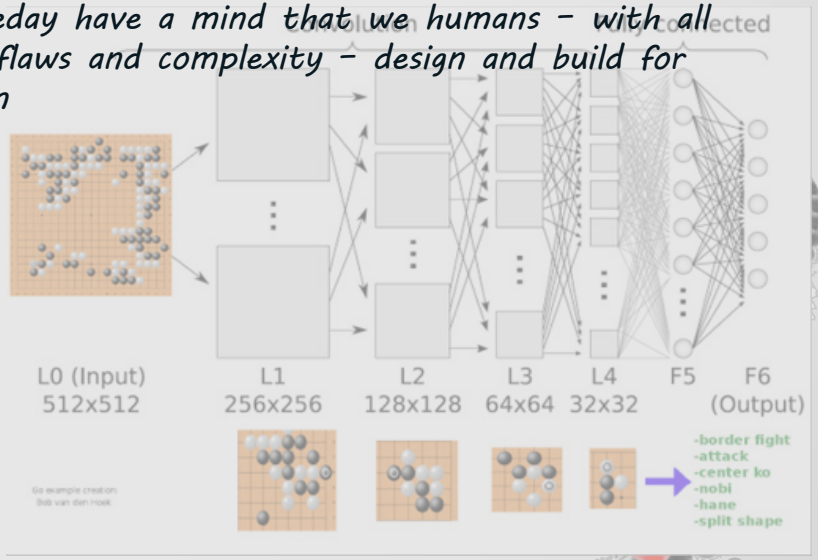




- It uses 4 networks :
- **Fast Rollout Policy Network** (P-Network) :It rolls out a quick plan for the game.
- **Supervised Learning Policy Network** (SL-Network) :The P-Network and the SL-Network are trained to predict human expert moves in a data set of positions.
- **Reinforcement Learning Policy Network** (RL-Network) :The RL Network is initialized as the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network.
- **Value network** : a Value Network V is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set.

https://medium.com/@arnabghosh93/deep-reinforcement-learning-driving-alphago-9881f1a90be4#.f462gnebl

- *AlphaGo is not a revolutionary breakthrough in itself, but rather a leading edge of an extremely important development*
- *#AlphaGo appeared to have a weakness in responding to unorthodox moves; Toward the end of the game, #AlphaGo played desperate moves similar to those of humans facing defeat …*
- *Ultimately, the scary thing about the rise of intelligent machines is not that they could someday have a mind of their own, but they could someday have a mind that we humans – with all our flaws and complexity – design and build for them*



AlphaGo in a Nutshell

based on: Silver, D. et al. Nature Vol. 529, 2016

SL Policy Network

Supervised Learning

Learning: 30 M positions from 130.000 human expert games Kiseido Go Server

input: raw board position as a 19x19 image plus convolutional layers to construct (increasingly localized) representations of the position

output: next move, fast efficient learning with immediate feedback

uses 13-layer convolutional neural network for simple board representation

Network predicted ~**50%** of expert moves using raw board position. Small improvements in this accuracy leads to large improvements in playing strength.

Fast Policy Network

Supervised Learning

Learning: Using ~140.000 patterns (e.g. nakade at captured stone, 12-point diamond pattern near previous move, 3x3 pattern around move, Mahattan distance to previous two moves, move matches 12-point diamond pattern centered around move)

input: position patterns

output: next move

a fast policy network is trained

Network predicted ~**24%** of expert moves correctly using small pattern features, this network is less accurate but about a factor 1000 faster than the SL Policy Network

RL Policy Network

Reinforcement Learning

Structure identical to SL policy network, aims at improving by reinforcement learning (final outcome of game used as feedback to the moves played)

input: plays against one of the earlier (improved) versions of itself (1..3 M games in 20 batches)

output: next move

Optimizes and adjusts RL policy to the correct goal of winning games, rather than maximizing predictive accuracy

The more successful a move, the more frequently it is played in future

With each game played, RL Policy network improves a tiny bit. When RL policy plays head-to-head to SL policy (selecting next move with highest probability), the RL policy network wins over **80%**.

Value Network

Reinforcement Learning

Focuses on position evaluation and predicts outcome (from given position to end, using the same policy for both players)

input: 30 M positions, each generated from a separate game of self-play of the RL Policy Network against (the same version) of itself

output: a single prediction of the winning probability (instead of a probability distribution)

Value network is trained by reinforcement learning to predict winner of game played by RL policy against itself

Compared to MCTS rollouts using fast policy, the positon evaluation accuracy of value network was consistently more accurate.

Single evaluation **15,000** faster than MCTS rollouts evaluation value

AlphaGo

AlphaGo efficiently combines policy and value networks with Monte Carlo Tree Search (MCTS)

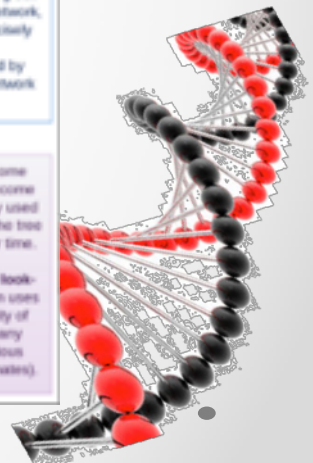
Policy and value networks to reduce:

1) the effective **depth** to which a move has to be computed (from there the outcome of the game can be estimated), and
2) the **breadth** of the search by sampling actions from a probability distribution over possible moves in position

Value network approximates outcome of games played by the strong but impractically slow RL policy network, while the MCTS rollouts precisely score outcome and evaluate games played by the weaker but fast policy network

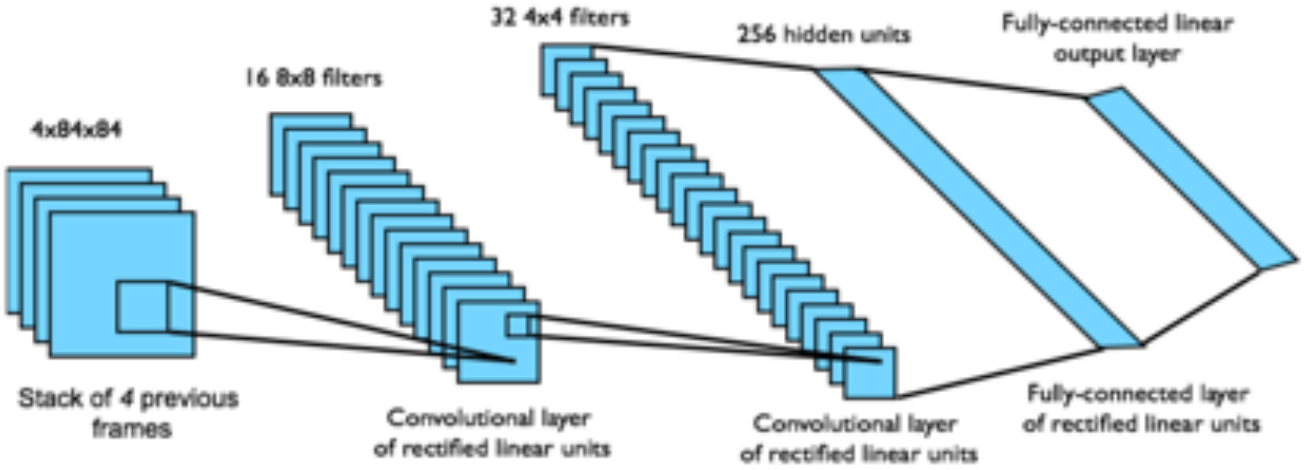
Using MCTS rollouts, outcome values of relevant moves become more accurate and the policy used to select next moves during the tree search is also improved over time.

Next moves are selected by **look-ahead search** after algorithm uses move with highest probability of winning (averaged over many different rollouts using various policies and evaluation estimates).

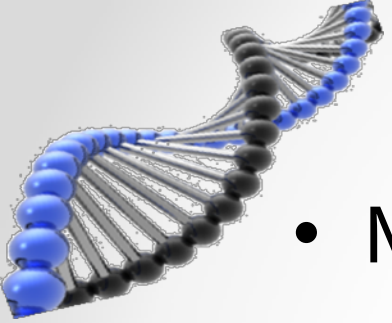


DQN in Atari

- End-to-end learning of values $Q(s, a)$ from pixels s
- Input state s is stack of raw pixels from last 4 frames
- Output is $Q(s, a)$ for 18 joystick/button positions
- Reward is change in score for that step

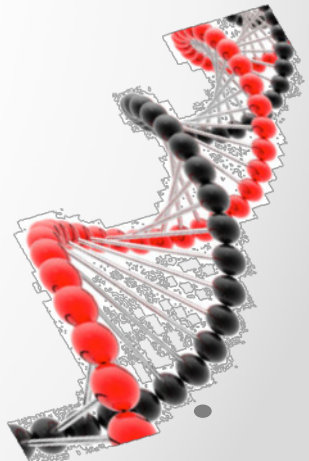


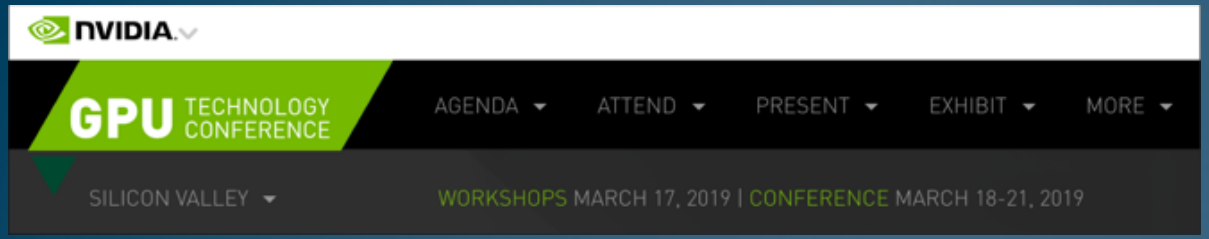
Network architecture and hyperparameters fixed across all games

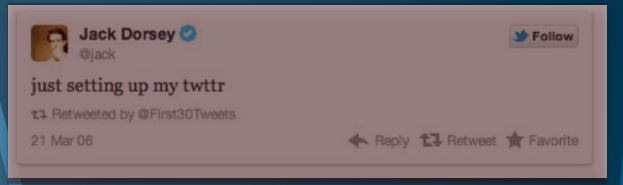


α_0 hands on

- Mooc/DRL-ND/ AlphaZero_TicTacToe



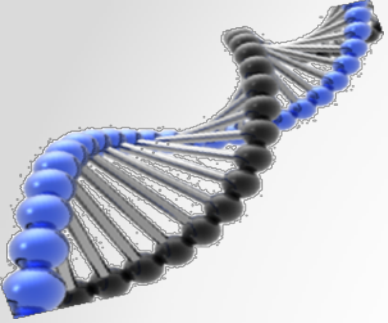




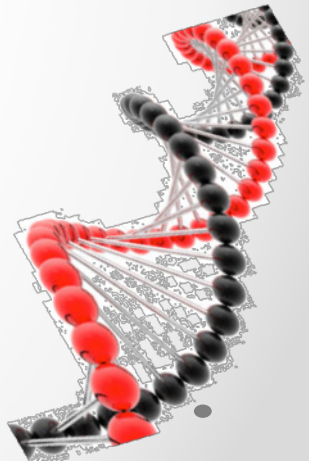
Demystifying Deep Reinforcement Learning – Part 2 : Policy Gradients

All observations are my own & do not represent my employer's or anyone else's

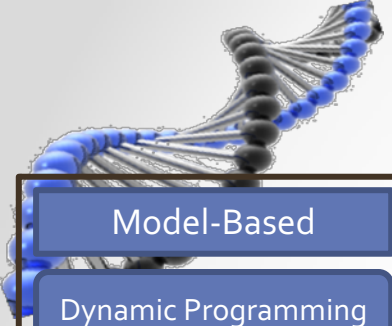
krishna sankar
@ksankar



5. Policy Gradients



RL Algorithm Taxonomy

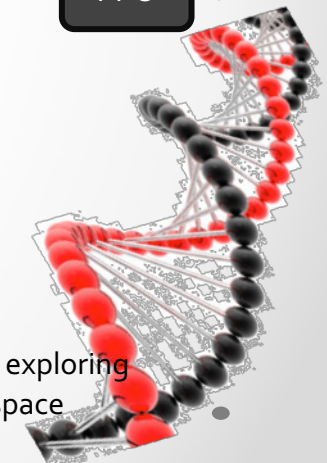


- Model-Based
  - Dynamic Programming
    - Policy Iteration
    - Value Iteration
- Model-Free
  - Monte Carlo
  - TD Learning
  - Value Methods
    - SARSA
    - Q-Learning
    - Deep Q-Learning
      - Fixed Q Targets
      - Experience replay
      - Double DQN
      - Prioritized replay
      - Dueling DQN
      - *Increase robustness*
      - *Reduce bias/variance*
      - *Sample efficiency*
  - Policy Search
    - Hill Climbing
      - Steepest Hill Climbing
        - Cross-entropy
        - Evolution Strategies
      - Random Restart
      - Simulated Annealing
      - Adaptive noise scaling
  - Policy Gradients
    - REINFORCE
    - TRPO
    - PPO
    - Actor-Critic
    - DDPG

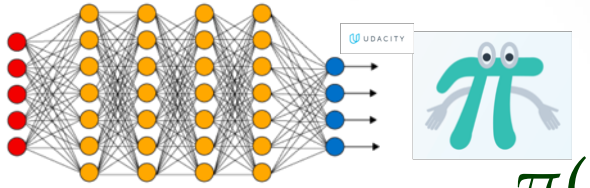
RL ⟷ DRL

Non-linear Function Approximation

- Why Policy Based ?
  - Simplicity
  - Stochastic Action Space
    - Rock-Paper-scissor
  - Continuous
    - Can output a # without exploring values in a continuous space
  - Aliased States



Policy in Policy Gradient



$\pi(a|s, \theta)$

Agent

$\pi(a|s, \theta) = \mathbb{P}\{A_t = a \mid S_t = s, \theta_t = \theta\}$

State

Reward

Parameterized Policy

Action

Discovered via Gradient Ascent

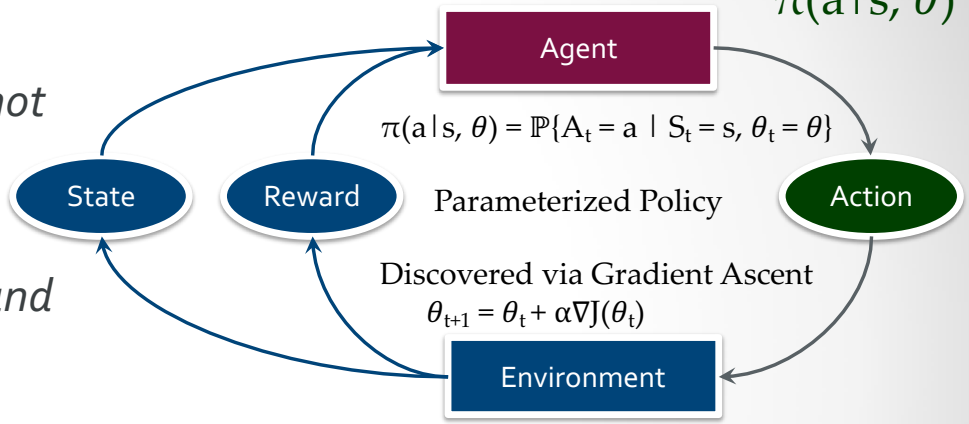
$\theta_{t+1} = \theta_t + \alpha \nabla J(\theta_t)$

Environment

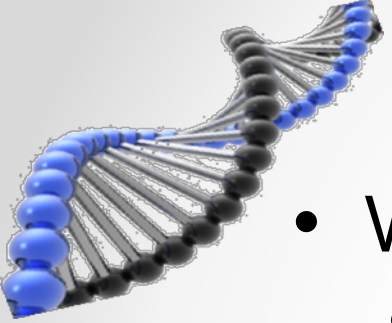
Policy in Policy Gradient

$\pi(a|s, \theta)$

- *All methods that follow this general schema we call policy gradient methods, whether or not they also learn an approximate value function.*
- *Methods that learn approximations to both policy and value functions are often called actor–critic methods*

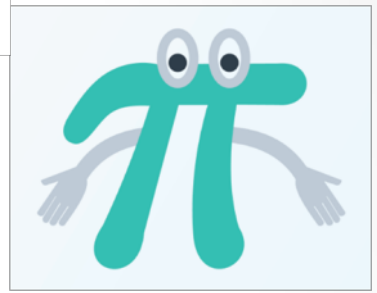


- Stochastic Policy $\pi(a|s, \theta) = \mathbb{P}\{a|s, \theta\}$
  - e.g. Softmax
- Deterministic Policy $\pi : s \mapsto a$
  - Our 1st exercise was a symbolic deterministic policy injecting domain knowledge

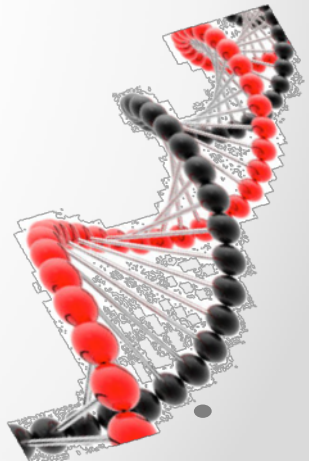


Policy Gradients

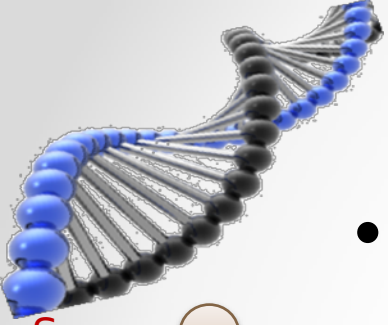
UDACITY

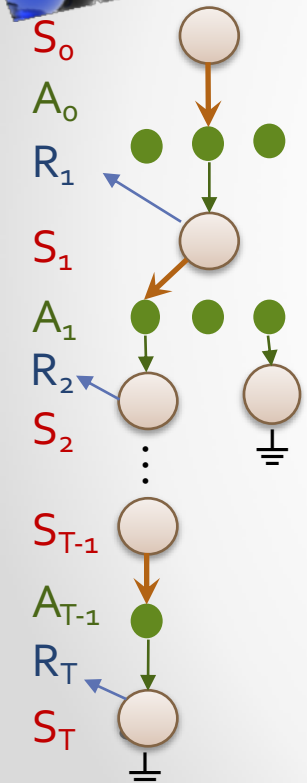


- Why Policy Gradients
  - Compact Representation
  - Better Convergence properties
  - Can handle Continuous space
  - Better for High Dimensional space
  - Stochastic Policies
  - Aliased States (Using uniform Stochastic Policy)
- Challenges
  - Susceptible to converging to local minima
  - High Variance

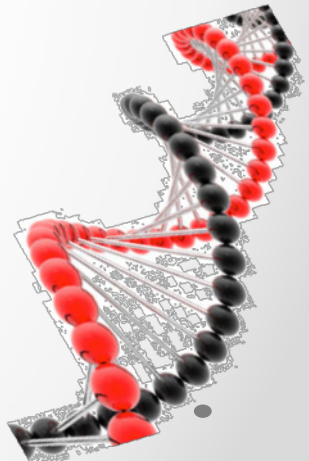


Policy Gradients - Nomenclature





- Trajectory $\boldsymbol{\tau}$ over a horizon H
  - $S_0 A_0 R_1 S_1 A_1 R_2 S_2 \ \ldots\ A_{H-1} R_H S_H$
- Reward $R(\boldsymbol{\tau})$
  - $R(\boldsymbol{\tau}) = \Sigma_{t=0 \text{ to } H} R(s_t, a_t)$
- Trajectory can be an episode (H=T), but doesn't dictate it



Unrolling the Math

- Expected Return

  $U_\theta = \mathbb{E}[\Sigma_{t=0 \text{ to } \tau} R(s_t, a_t) ; \pi(\theta)] = \Sigma_\tau \mathbb{P}(\tau, \theta) R(\tau)$ *... some literature calls it* J_θ

- Goal : Maximize J_θ

  $\max_\theta(U_\theta) = \max_\theta \{ \mathbb{E}[\Sigma_{t=0 \text{ to } \tau} R(s_t, a_t); \pi(\theta)] \} = \max_\theta \{ \Sigma_\tau \mathbb{P}(\tau, \theta) R(\tau) \}$

- Likelyhood Ratio Method
  - Forms the foundation for Policy Gradients, e.g. REINFORCE
  - In essence, it means our policy makes some trajectories more likely to happen than others
    - With weighting those returns from those trajectories by the probability that they actually take place

Likelihood Policy Gradient

$\max_\theta(U_\theta) = \max_\theta \{ \mathbb{E}[\Sigma_{t=o \text{ to } \tau} R(s_t, a_t); \pi(\theta)] \} = \max_\theta \{ \Sigma_\tau \mathbb{P}(\tau, \theta) R(\tau) \}$

Discovered via Gradient Ascent : $\theta_{t+1} = \theta_t + \alpha \nabla U(\theta_t)$

$$\nabla_\theta U(\theta) = \nabla_\theta \Sigma_\tau \mathbb{P}(\tau; \theta) R(\tau)$$

$$= \Sigma_\tau \nabla_\theta \mathbb{P}(\tau; \theta) R(\tau)$$

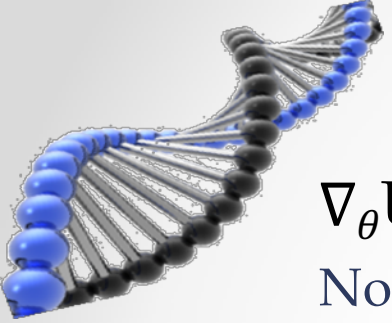
$$= \Sigma_\tau \frac{\mathbb{P}(\tau; \theta)}{\mathbb{P}(\tau; \theta)} \nabla_\theta \mathbb{P}(\tau; \theta) R(\tau)$$

$$= \Sigma_\tau \mathbb{P}(\tau; \theta) \frac{\nabla_\theta \mathbb{P}(\tau; \theta)}{\mathbb{P}(\tau; \theta)} R(\tau)$$

$$= \Sigma_\tau \mathbb{P}(\tau; \theta) \nabla_\theta \log \mathbb{P}(\tau; \theta) R(\tau)$$

Def:

$$\nabla_x \log f(x) = \frac{\nabla_\theta f(x)}{f(x)}$$



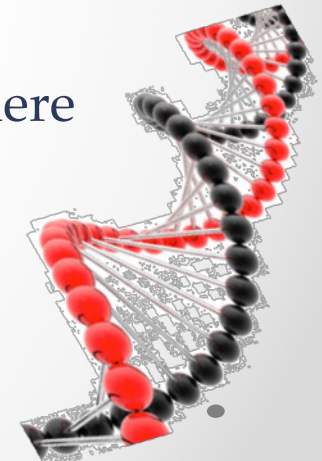
Likelihood Policy Gradient

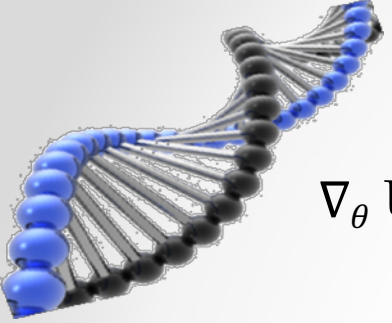
$\nabla_\theta U(\theta) = \Sigma_{\boldsymbol{\tau}} \mathbb{P}(\boldsymbol{\tau}; \theta) \nabla_\theta \log \mathbb{P}(\boldsymbol{\tau}; \theta) R(\boldsymbol{\tau})$

Now we can perturb our parameter space θ and it will ascend or descend depending on $R(\boldsymbol{\tau})$

But we have 2 problems:

1. We don't have probabilities in terms of trajectories; we have states and actions !
2. We assume infinite trajectories; not feasible for mere mortals
    - *Even though OpenAI 5 is using 120,000 CPUs and 256 GPUs, learning the equivalent of 180 years of play per day !*



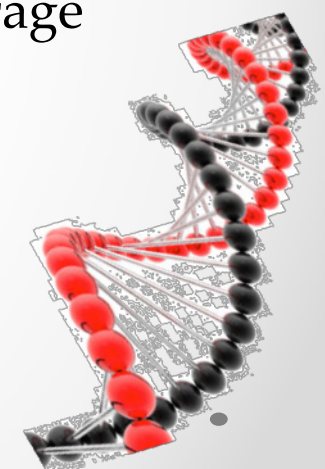


Unrolling the Math

$$\nabla_\theta U(\theta) = \Sigma_{\boldsymbol{\tau}} \mathbb{P}(\boldsymbol{\tau}; \theta) \nabla_\theta \log \mathbb{P}(\boldsymbol{\tau}; \theta) R(\boldsymbol{\tau})$$

- Considering all possible trajectories is an impossible task. While in the math world, we can image as elegant as possible.
- Down in the real world, we estimate empirically by sampling m trajectories(paths) under π_θ and average

$$\nabla_\theta U(\theta) \approx \frac{1}{m} \Sigma_{i=1 \text{ to } m} \nabla_\theta \log \mathbb{P}(\boldsymbol{\tau}^i; \theta) R(\boldsymbol{\tau}^i)$$



Decomposing trajectories to states & actions

$$\nabla_\theta U(\theta) = \frac{1}{m} \Sigma_{i=1 \text{ to } m} \nabla_\theta \log \mathbb{P}(\boldsymbol{\tau}^i ; \theta) R(\boldsymbol{\tau}^i)$$

- Let us reason about states and actions than probabilities of trajectories

$$\nabla_\theta \log \mathbb{P}(\boldsymbol{\tau} ; \theta) = \nabla_\theta \log [\Pi_{t=0 \text{ to } T} \mathbb{P}(s_{t+1} | s_t, a_t) \pi(a | s, \theta)]$$

$$= \nabla_\theta [\Sigma_{t=0 \text{ to } T} \log \mathbb{P}(s_{t+1} | s_t, a_t) + \Sigma_{t=0 \text{ to } T} \log \pi(a | s, \theta)]$$

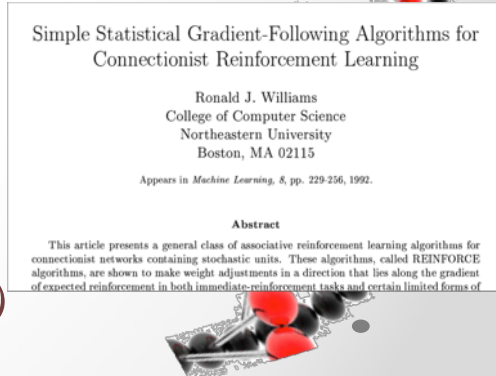
$$= \nabla_\theta \Sigma_{t=0 \text{ to } T} \log \mathbb{P}(s_{t+1} | s_t, a_t) + \nabla_\theta \Sigma_{t=0 \text{ to } T} \log \pi(a | s, \theta)$$

$$= \nabla_\theta \Sigma_{t=0 \text{ to } T} \log \pi(a | s, \theta) \ldots \text{ since } \nabla_\theta \Sigma_{t=0 \text{ to } T} \log \mathbb{P}(s_{t+1} | s_t, a_t) = 0$$

$$\nabla_\theta U(\theta) = \frac{1}{m} \Sigma_{i=1 \text{ to } m} \Sigma_{t=0 \text{ to } T} \nabla_\theta \log \pi(a | s, \theta) R(\boldsymbol{\tau}^i)$$

… and this is the REINFORCE algorithm !

(http://www-anw.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf)



Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning

Ronald J. Williams
College of Computer Science
Northeastern University
Boston, MA 02115

Appears in *Machine Learning*, 8, pp. 229-256, 1992.

Abstract

This article presents a general class of associative reinforcement learning algorithms for connectionist networks containing stochastic units. These algorithms, called REINFORCE algorithms, are shown to make weight adjustments in a direction that lies along the gradient of expected reinforcement in both immediate-reinforcement tasks and certain limited forms of

REINFORCE = Mon

REINFORCE: Monte-Carlo Policy-Gradient

Input: a differentiable policy parameterization $\pi(a|s$

Algorithm parameter: step size $\alpha > 0$

Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$

Loop for each step of the episode $t = 0, 1, \ldots, T-1$:

$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ $\quad (G_t)$

$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Discounted return for variance reduction

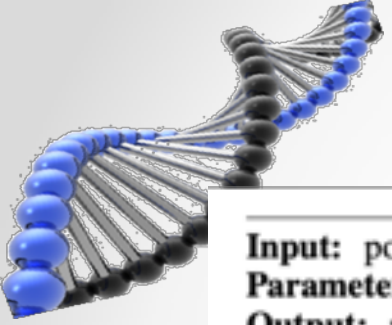
α = 0.1 or 0.2

REINFORCE

1. Create initial policy π_θ
2. Act in the environment $a_t \sim \pi(s_t)$ for an episode storing states, actions, rewards: $s_0, a_0, r_0, \ldots, s_T, a_T, r_T$
3. Compute return $R = \sum_{t=0}^{T} r_t$
4. Compute policy gradient $\nabla_\theta J(\theta) = \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R$
5. Take a step in the direction of the gradient: $\theta = \theta + \alpha \nabla_\theta J(\theta)$
6. Repeat until convergence

Microsoft

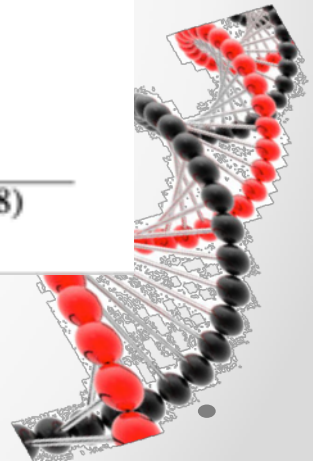
- Reinforce = MonteCarlo PG
  - i.e. full episode as Trajectory
- Low bias, but high variance
- REINFORCE Paper : http://www-anw.cs.umass.edu/~barto/courses/cs687/williams92simple.pdf



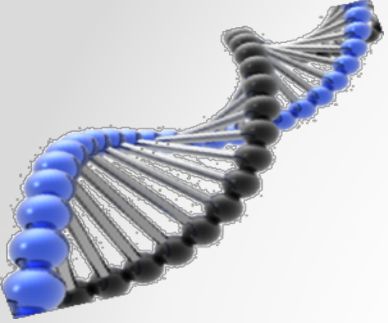
Input: policy $\pi(a|s, \boldsymbol{\theta})$, $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes, $\alpha > 0, \beta > 0$
Output: policy $\pi(a|s, \boldsymbol{\theta})$

initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\boldsymbol{w}$
for *true* **do**
  generate an episode $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  for *each step* t *of episode* $0, \cdots, T-1$ **do**
    $G_t \leftarrow$ return from step t
    $\delta \leftarrow G_t - \hat{v}(s_t, \boldsymbol{w})$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \beta\delta\nabla_{\boldsymbol{w}}\hat{v}(s_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t\boldsymbol{\delta}\nabla_{\boldsymbol{\theta}} log\pi(a_t|s_t, \boldsymbol{\theta})$
  end
end

Algorithm 6: REINFORCE with baseline (episodic), adapted from Sutton and Barto (2018)

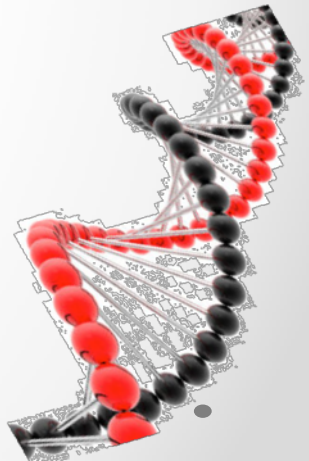


https://arxiv.org/pdf/1810.06339.pdf

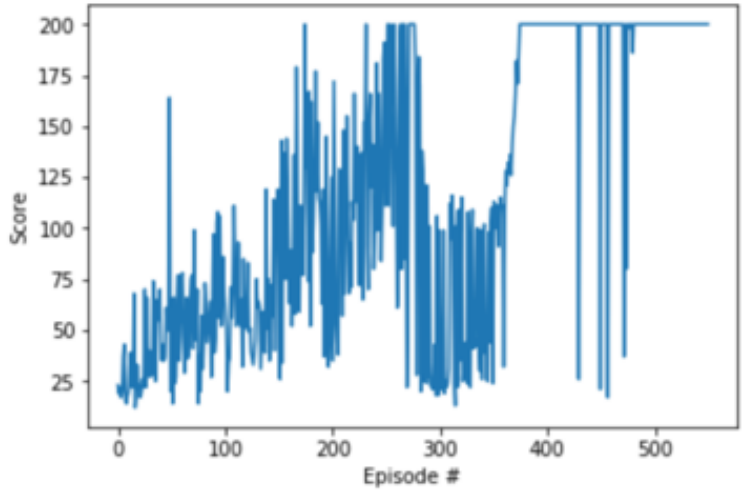


Hands-On #10 : Balancing the Cart Pole w/ Policy Gradient (REINFORCE)

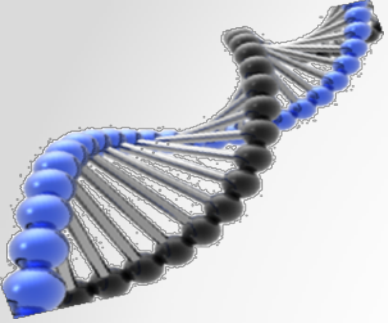
- Goal:
  - Implement Policy Gradient Network for CartPole
    - It is a lot simpler !!
- Steps:
  - Notebook : REINFORCE_01.ipynb
  - Program REINFORCE
  - Plot Values



```
Episode 100     Average Score: 47.22
Episode 200     Average Score: 80.94
Episode 300     Average Score: 116.19
Episode 400     Average Score: 112.00
Episode 500     Average Score: 191.65
Environment solved in 450 episodes!     Average Score: 195.18
Elapsed : 0:00:23.645272
2019-01-12 11:47:30.619834
```




```
Max Score 200.000000 at 174
Percentile [25,50,75] : [ 51. 111. 200.]
Variance : 4779.687
```

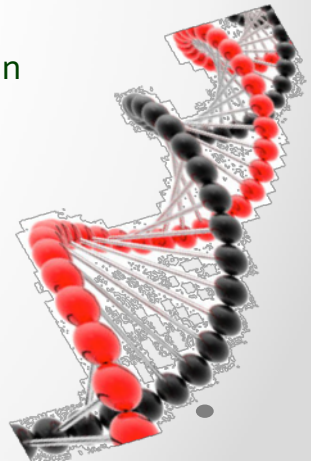


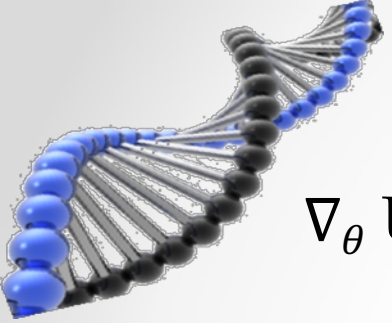
The Deadly Triad--Sutton

- Function Approximation
  - Generalization from examples

- Bootstrapping
- Off-Policy learning

- REINFORCE
  - Unbiased, very noisy
    - We are sampling one out of millions of trajectories, for each iteration
  - Sample in-efficient
  - Credit Assignment
- Fixes
  - Baseline
  - Temporal Structure
  - KL-Divergence trust Region





Reinforce with Baseline

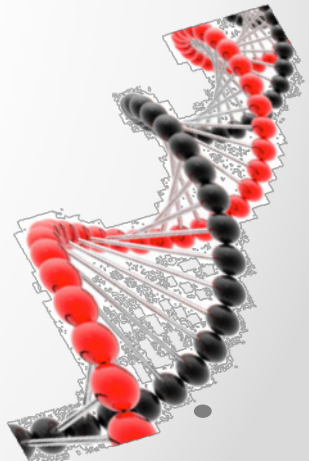
$$\nabla_\theta U(\theta) = \frac{1}{m} \sum_{i=1 \text{ to } m} \sum_{t=0 \text{ to } T} \nabla_\theta \log \pi(a \mid s, \theta)\, R(\boldsymbol{\tau}^i)$$

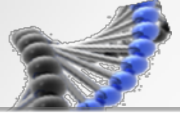
$$= \frac{1}{m} \sum_{i=1 \text{ to } m} \sum_{t=0 \text{ to } T} \nabla_\theta \log \pi(a \mid s, \theta)\, (R(\boldsymbol{\tau}^i) - b)$$

Many interesting choice for a baseline:

1. Average of all rewards so far i.e. $\mu\{R(\boldsymbol{\tau})\}$
2. Optimal Baseline
3. Time dependent baseline
4. State dependent expected baseline i.e. $V_\pi(s)$

$$\nabla_\theta U(\theta) = \frac{1}{m} \sum_{i=1 \text{ to } m} \sum_{t=0 \text{ to } T} \nabla_\theta \log \pi(a \mid s, \theta)\, (R(\boldsymbol{\tau}^i) - V_\phi(s))$$



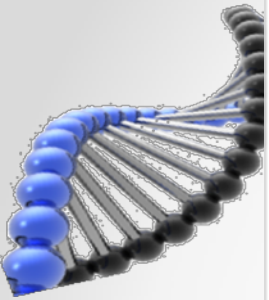


REINFORCE with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  Loop for each step of the episode $t = 0, 1, \ldots, T-1$:

$$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \qquad (G_t)$$
$$\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$$
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$$



The Baselined REINFORCE Algorithm

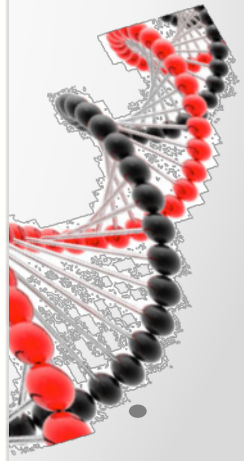
Bookmark this page

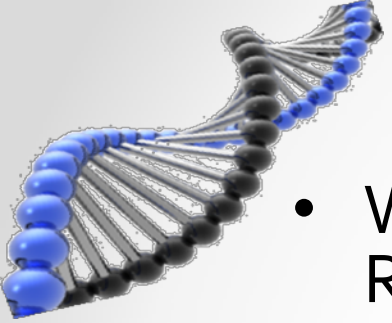
Video

Policy Gradient Recap

1. Initialize policy π_θ and value function V_ϕ
2. For episode 1,2,…,n do:
    1. Collect trajectory τ by running current policy
    2. For each step in τ compute the discounted return $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$
    3. Re-fit the baseline minimizing $\|V_\phi(s_t) - R_t\|^2$ over all steps
    4. Update π_θ using gradient $\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(R_t - V_\phi(s_t))$

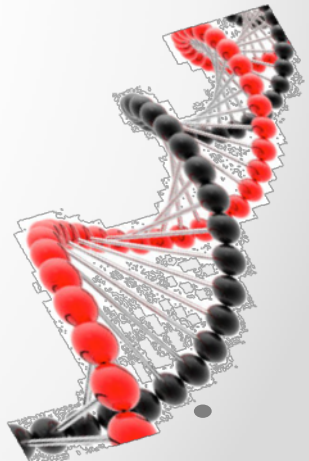
Microsoft

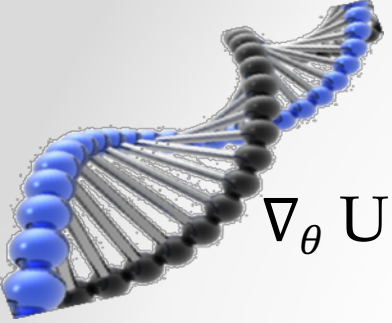




Morphing to A3C

- Would better baseline reduce the variance for REINFORCE ?
- Can we use a Neural Network to establish this baseline ?
- Of course Yes to both
  - A3C and it's synchronous twin A2C
- And, … we use a TD-Critic instead of a MonteCarlo baseline !
  - Faster Learning than MC alone
  - Consistent Convergence than value based methods
  - And we gain generalization





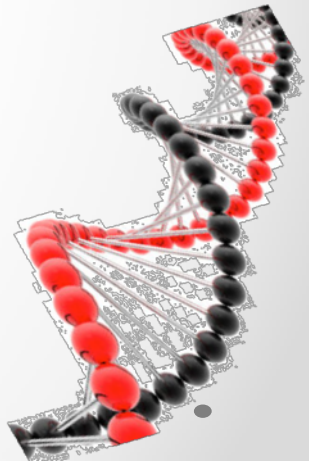
Advantage

$$\nabla_\theta U(\theta) = \frac{1}{m} \sum_{i=1 \text{ to } m} \sum_{t=0 \text{ to } T} \nabla_\theta \log \pi(a \mid s, \theta) \, (R_t - V_\phi(s_t))$$

Actor

Critic

- Interestingly, R_t is nothing but $Q_\pi(s_t, a_t)$
- So $R_t - V_\phi(s_t)$ becomes $Q_\pi(s_t, a_t) - V_\phi(s_t)$
  - Which says how much better the new state-action value is over the current value i.e. Advantage
- $A(s_t, a_t) = Q_\pi(s_t, a_t) - V_\phi(s_t)$



Basic Actor-Critic Agent

$$\nabla_\theta U(\theta) = \sum_{i=1 \text{ to } m} \sum_{t=0 \text{ to } T} \nabla_\theta \log \pi(a \mid s, \theta) A_t$$

Actor Critic

- Function Approximation ie neural network for Actor and Critic ie the Policy and the Predictor of Advantage of a policy using TD estimate
- Once you know the advantage of a policy we can tweak the policy up or down
- https://arxiv.org/abs/1602.01783

Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih[1]	VMNIH@GOOGLE.COM
Adrià Puigdomènech Badia[1]	ADRIAP@GOOGLE.COM
Mehdi Mirza[1,2]	MIRZAMOM@IRO.UMONTREAL.CA
Alex Graves[1]	GRAVESA@GOOGLE.COM
Tim Harley[1]	THARLEY@GOOGLE.COM
Timothy P. Lillicrap[1]	COUNTZERO@GOOGLE.COM
David Silver[1]	DAVIDSILVER@GOOGLE.COM
Koray Kavukcuoglu [1]	KORAYK@GOOGLE.COM

[1] Google DeepMind
[2] Montreal Institute for Learning Algorithms (MILA), University of Montreal

One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize S (first state of episode)
    $I \leftarrow 1$
    Loop while S is not terminal (for each time step):
        $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
        Take action A, observe S', R
        $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S,\boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$

Input: policy $\pi(a|s, \boldsymbol{\theta}), \hat{v}(s, \boldsymbol{w})$
Parameters: step sizes, $\alpha > 0, \beta > 0$
Output: policy $\pi(a|s, \boldsymbol{\theta})$

initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\boldsymbol{w}$
for *true* **do**
  initialize s, the first state of the episode
  $I \leftarrow 1$
  for s *is not terminal* **do**
    $a \sim \pi(\cdot|s, \boldsymbol{\theta})$
    take action a, observe s', r
    $\delta \leftarrow r + \gamma \hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})$ (if s' is terminal, $\hat{v}(s', \boldsymbol{w}) \doteq 0$)
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \beta \delta \nabla_{\boldsymbol{w}} \hat{v}(s_t, \boldsymbol{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha I \delta \nabla_{\boldsymbol{\theta}} log \pi(a_t|s_t, \boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $s \leftarrow s'$
  end
end

Algorithm 7: Actor-Critic (episodic), adapted from Sutton and Barto (2018)

https://arxiv.org/pdf/1810.06339.pdf

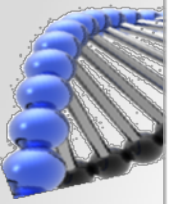
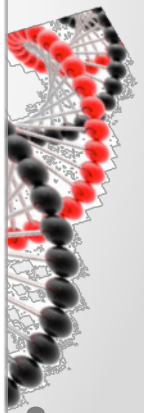
Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

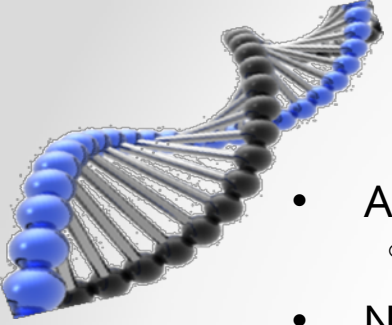
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
  Initialize S (first state of episode)
  $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
  $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
  $I \leftarrow 1$
  Loop while S is not terminal (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S,\mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \gamma \lambda^{\boldsymbol{\theta}} \mathbf{z}^{\boldsymbol{\theta}} + I \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \delta \mathbf{z}^{\boldsymbol{\theta}}$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Actor–Critic with Eligibility Traces (continuing), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: $\lambda^{\mathbf{w}} \in [0,1]$, $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\alpha^{\mathbf{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\bar{R}} > 0$
Initialize $\bar{R} \in \mathbb{R}$ (e.g., to 0)
Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)
Initialize $S \in \mathcal{S}$ (e.g., to s_0)

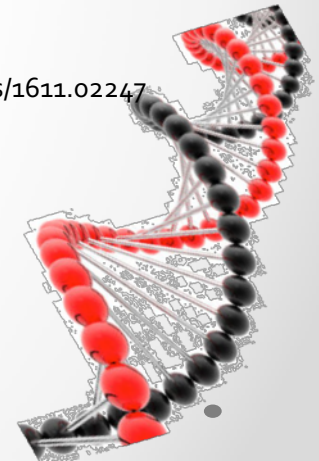
$\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ (d-component eligibility trace vector)
$\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ (d'-component eligibility trace vector)
Loop forever (for each time step):
$A \sim \pi(\cdot|S,\boldsymbol{\theta})$
Take action A, observe S', R
$\delta \leftarrow R - \bar{R} + \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
$\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}}\delta$
$\mathbf{z}^{\mathbf{w}} \leftarrow \lambda^{\mathbf{w}}\mathbf{z}^{\mathbf{w}} + \nabla\hat{v}(S,\mathbf{w})$
$\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \lambda^{\boldsymbol{\theta}}\mathbf{z}^{\boldsymbol{\theta}} + \nabla \ln \pi(A|S,\boldsymbol{\theta})$
$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\mathbf{z}^{\mathbf{w}}$
$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathbf{z}^{\boldsymbol{\theta}}$
$S \leftarrow S'$



A3C

- A3C can share networks between the Actor and Critic.
  - But harder to train – so start with 2 networks, do hyperparameter and architecture tuning and then combine for performance
- N-step bootstrapping TD(lambda) with lambda = n
  - More n less bias while keeping variance under control
  - Usually 4-5 step bootstrapping are the best
    - Update last n steps
- A3C doesn't use experience replay but gets diversity from parallel training ie running multiple agents to a common buffer
- On or off policy ? On ! Stable and consistent convergence property !
  - Off policy often unstable and can diverge with nn
  - But there is work to make off-policy critic better
    - Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic https://arxiv.org/abs/1611.02247
- (show these in the code)
- A2C – Synchronous A3C !
  - Synchronous point, update network and update gradients of all agents
  - A3C CPU is Ok. A2C – better on a GPU
- UCB Soft Actor Critic
  - https://bair.berkeley.edu/blog/2018/12/14/sac/



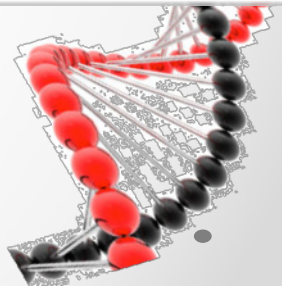
Actor-Critic with A3C or GAE

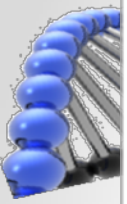
- Policy Gradient + Generalized Advantage Estimation:
  - Init π_{θ_0} $V^{\pi}_{\phi_0}$
  - Collect roll-outs {s, u, s', r} and $\hat{Q}_i(s,u)$
  - Update: $\phi_{i+1} \leftarrow \min_{\phi} \sum_{(s,u,s',r)} \|\hat{Q}_i(s,u) - V^{\pi}_{\phi}(s)\|_2^2 + \kappa\|\phi - \phi_i\|_2^2$

$$\theta_{i+1} \leftarrow \theta_i + \alpha \frac{1}{m} \sum_{k=1}^{m} \sum_{t=0}^{H-1} \nabla_\theta \log \pi_{\theta_i}(u_t^{(k)}|s_t^{(k)}) \left(\hat{Q}_i(s_t^{(k)}, u_t^{(k)}) - V^{\pi}_{\phi_i}(s_t^{(k)})\right)$$

Async Advantage Actor Cri<c (A3C) *[Mnih et al, 2016]*

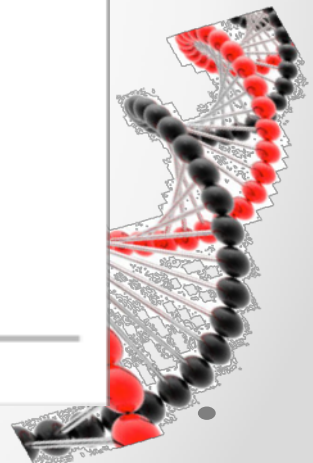
- Q^ one of the above choices (e.g. k=5 step lookahead)





global shared parameter vectors $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_v$, thread-specific parameter vectors $\boldsymbol{\theta}'$ and $\boldsymbol{\theta}'_v$
global shared counter $T = 0, T_{max}$
initialize step counter $t \leftarrow 1$
for $T \leq T_{max}$ **do**
    reset gradients, $d\boldsymbol{\theta} \leftarrow 0$ and $d\boldsymbol{\theta}_v \leftarrow 0$
    synchronize thread-specific parameters $\boldsymbol{\theta}' = \boldsymbol{\theta}$ and $\boldsymbol{\theta}'_v = \boldsymbol{\theta}_v$
    set $t_{start} = t$, get state s_t
    for s_t *not terminal and* $t - t_{start} \leq t_{max}$ **do**
        take a_t according to policy $\pi(a_t|s_t;\boldsymbol{\theta}')$
        receive reward r_t and new state s_{t+1}
        $t \leftarrow t+1, T \leftarrow T+1$
    end
    $$R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \boldsymbol{\theta}'_v) & \text{otherwise} \end{cases}$$
    for $i \in \{t-1, ..., t_{start}\}$ **do**
        $R \leftarrow r_i + \gamma R$
        accumulate gradients wrt $\boldsymbol{\theta}'$: $d\boldsymbol{\theta} \leftarrow d\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}'} \log \pi(a_i|s_i;\boldsymbol{\theta}')(R - V(s_i;\boldsymbol{\theta}'_v))$
        accumulate gradients wrt $\boldsymbol{\theta}'_v$: $d\boldsymbol{\theta}_v \leftarrow d\boldsymbol{\theta}_v + \nabla_{\boldsymbol{\theta}'_v}(R - V(s_i;\boldsymbol{\theta}'_v))^2$
    end
    update asynchronously $\boldsymbol{\theta}$ using $d\boldsymbol{\theta}$, and $\boldsymbol{\theta}_v$ using $d\boldsymbol{\theta}_v$
end

Algorithm 10: A3C, each actor-learner thread, based on Mnih et al. (2016)



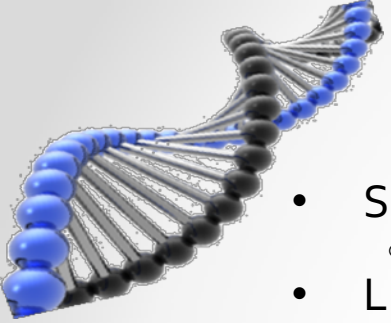
https://arxiv.org/pdf/1810.06339.pdf

Algorithm S2 Asynchronous n-step Q-learning - pseudocode for each actor-learner thread.

// Assume global shared parameter vector θ.
// Assume global shared target parameter vector θ^-.
// Assume global shared counter $T = 0$.
Initialize thread step counter $t \leftarrow 1$
Initialize target network parameters $\theta^- \leftarrow \theta$
Initialize thread-specific parameters $\theta' = \theta$
Initialize network gradients $d\theta \leftarrow 0$
repeat
  Clear gradients $d\theta \leftarrow 0$
  Synchronize thread-specific parameters $\theta' = \theta$
  $t_{start} = t$
  Get state s_t
  repeat
    Take action a_t according to the ϵ-greedy policy based on $Q(s_t, a; \theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ \max_a Q(s_t, a; \theta^-) & \text{for non-terminal } s_t \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \frac{\partial \left(R - Q(s_i, a_i; \theta')\right)^2}{\partial \theta'}$
  end for
  Perform asynchronous update of θ using $d\theta$.
  if $T \mod I_{target} == 0$ **then**
    $\theta^- \leftarrow \theta$
  end if
until $T > T_{max}$

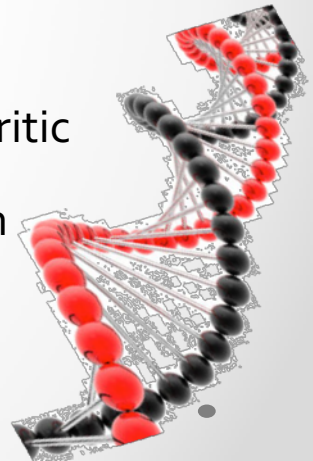
Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

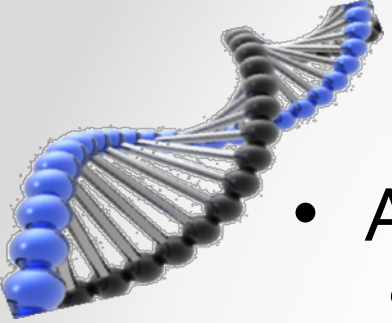
// Assume global shared parameter vectors θ *and* θ_v *and global shared counter* $T = 0$
// Assume thread-specific parameter vectors θ' *and* θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial \left(R - V(s_i;\theta'_v)\right)^2 / \partial \theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$



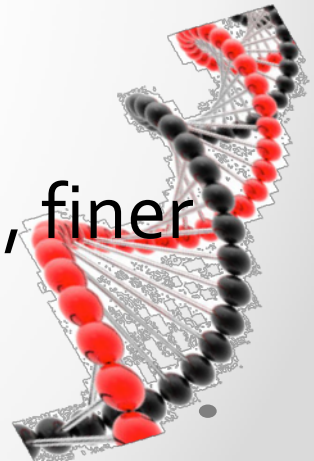
Intro to DDPG

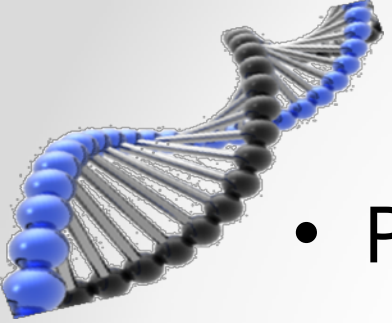
- Silver et al. released the DPG algorithm in 2014
  - http://proceedings.mlr.press/v32/silver14.pdf
- Lillicrap et al. iterated DPG with DDPG in 2015
  - https://arxiv.org/abs/1509.02971
- Ornstein–Uhlenbeck process (?)
  - https://en.wikipedia.org/wiki/Ornstein–Uhlenbeck_process
- PPO (Proximal Policy Optimization Algorithms)
  - https://arxiv.org/abs/1707.06347
- In 2016, Schulman et al. released TRPO() and GAE algorithms(GAE on TRPO)
  - TRPO : https://arxiv.org/abs/1502.05477
  - GAE : https://arxiv.org/abs/1506.02438
- Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic
  - Gu, Lillicrap, Levine : https://arxiv.org/abs/1611.02247
- Sergey Levine, Chelsea Finn, et al. released the GPS algorithm
  - Guided Policy Search http://rll.berkeley.edu/gps/
- A3c vs DDPG
  - https://yodahuang.github.io/articles/DDPG-vs-A2C/





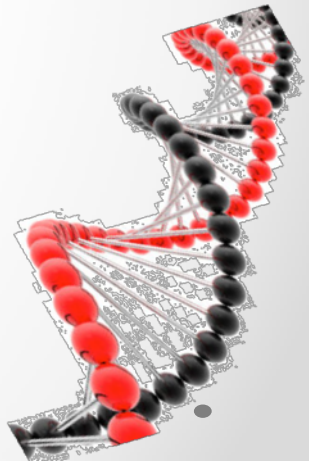
- A3C
  - Episode (~MC paradigm)
  - On-line, on-policy
  - Loss = loglikelihood
  - Continuous modelled as a Gaussian vector
- DDPG
  - TD() and TD(ƛ)
  - Experience Replay
  - Better for continuous state and action space
- They both use the Actor-Critic design, finer semantics might differ

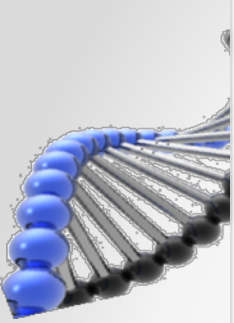




DDPG

- Paper https://arxiv.org/abs/1509.02971
- Paper says it is actor-critic
  - Morales – it is more of an approx. DQN for continuous *action* space similar to NAF(https://arxiv.org/abs/1603.00748)
  - The critic is a maximizer not a baseline
- Policy = Deterministic action





Algorithm 1 DDPG algorithm

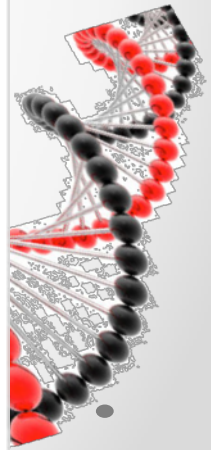
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

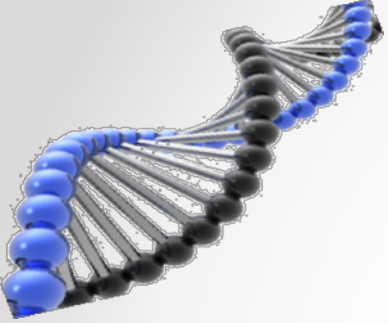
$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

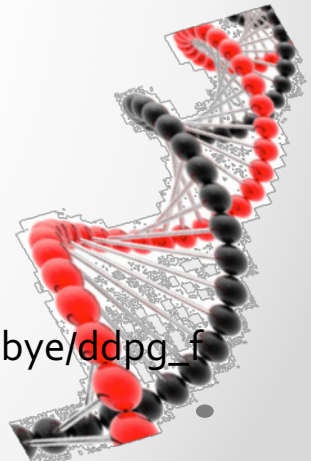
  end for
end for



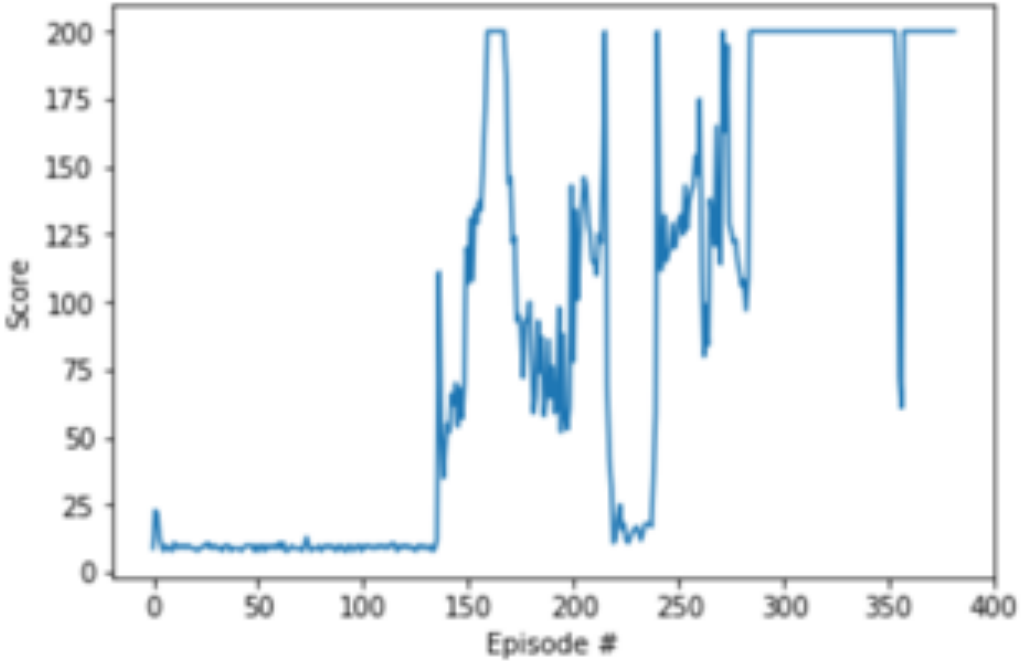


Hands-On #11 : Balancing the Cart Pole w/ DDPG

- Goal:
  - Implement DDPG on the CartPole Environment
  - It is an overkill, but we will get a good understanding and also can compare against other algorithms
- Steps:
  - Program DDPG Algorithm
  - Run and Optimize
  - Plot Values, like we did in other exercises
- DDPG for discrete actions
  - DDPG for Discrete Action Domains https://www.reddit.com/r/reinforcementlearning/comments/8k8bye/ddpg_for_discrete_action_domains/

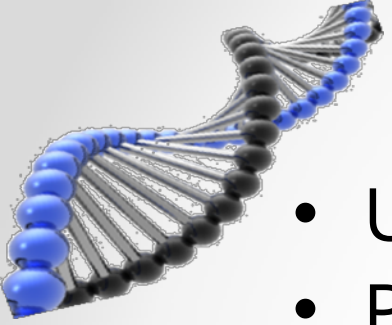


```
Episode 100     Average Score: 9.61     Score: 8.00     Max_steps :   79
Episode 200     Average Score: 71.03    Score: 143.00   Max_steps : 142
Episode 300     Average Score: 114.96   Score: 200.00   Max_steps : 199
Episode 382     Average Score: 195.17   Score: 200.00   Max_steps : 199
Environment solved in 282 episodes!     Average Score: 195.17
Elapsed : 0:02:29.415469
2019-01-13 18:14:48.885969
```



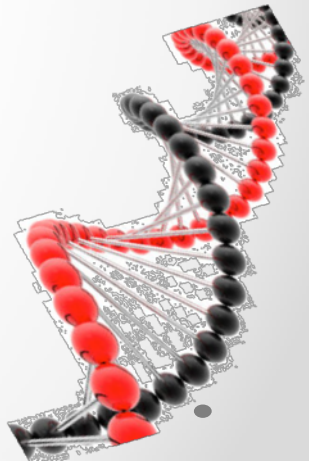
Perfect Test Score !

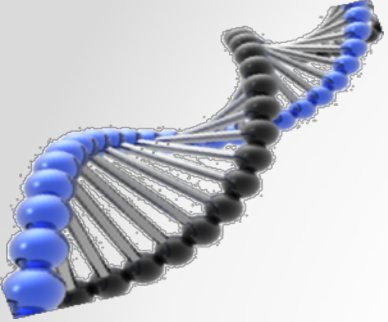
```
Episode :  1 Score : 200.00 Steps : 199
Episode :  2 Score : 200.00 Steps : 199
Episode :  3 Score : 200.00 Steps : 199
Episode :  4 Score : 200.00 Steps : 199
Episode :  5 Score : 200.00 Steps : 199
Episode :  6 Score : 200.00 Steps : 199
Episode :  7 Score : 200.00 Steps : 199
Episode :  8 Score : 200.00 Steps : 199
Episode :  9 Score : 200.00 Steps : 199
Episode : 10 Score : 200.00 Steps : 199
Mean of 10 episodes = 200.0
Elapsed : 0:00:00.517737
2019-01-13 18:18:07.700678
```



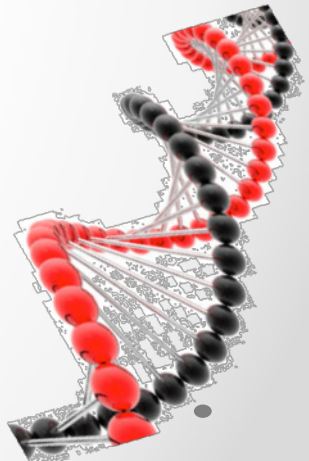
Hands-On PG-03 Pong

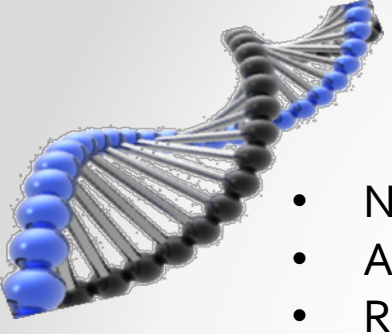
- Udacity pong reinforce
- Pongdeterministic – doesn't contain random frame skipping
- Rightfire,leftfire easier to train – Udacity
- Black & white, crop irrelevant boundaries
- Stack 2
- Rollout – parallel agents
- Progressbar
- Train & Play





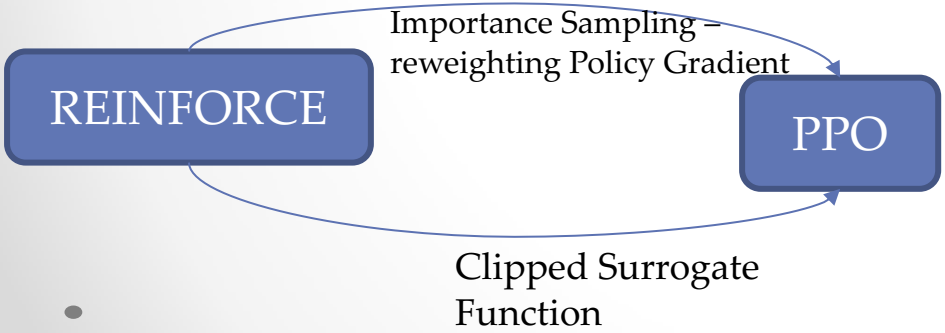
Hands-On PG-04 Breakout

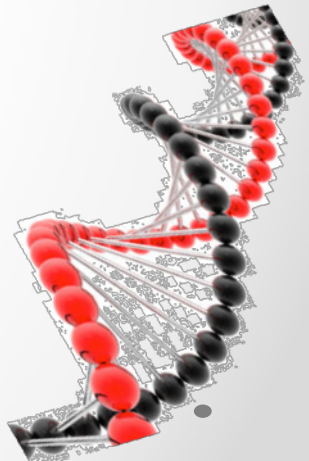


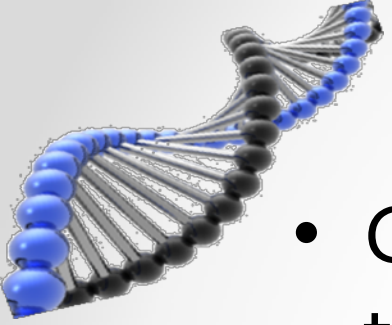


Intro to PPO

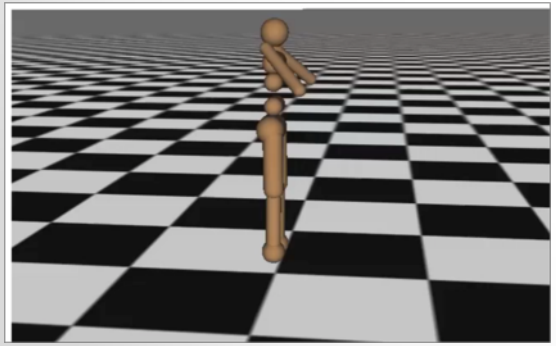
- Noise reduction et al from Udacity
- Asynchronous collection
- Reward Normalization
- Read PPO paper and update, Add TRPO and DDPG
  - PPO paper https://arxiv.org/abs/1707.06347
- Show PPO-pong and see how it defeats – show in GPU/CPU. time
- Is DDPG similar to DQN why ? Why not ?
- How DDPG related to DQN ? TRPO ? Draw diagram w/ paper & main ideas

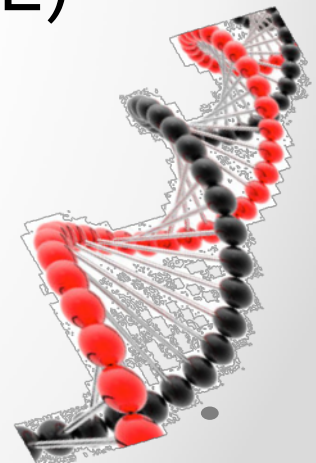


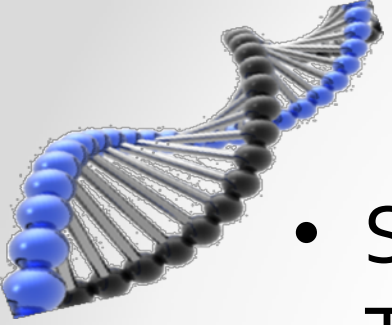




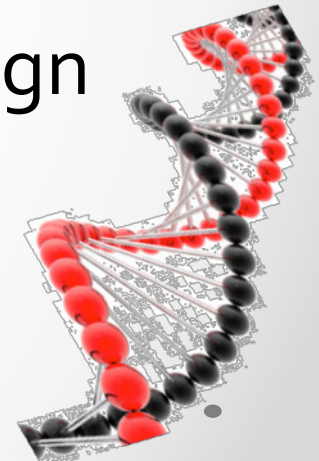
- GAE – mixture of lambda return. Difficult to san which n is better
  - Combination of all n step weighted by exponentially decaying lambda
- Learning LocomoQon (TRPO + GAE)
  - Schulman, Moritz, Levine, Jordan, Abbeel, 2016

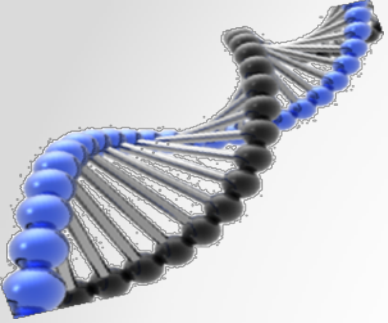




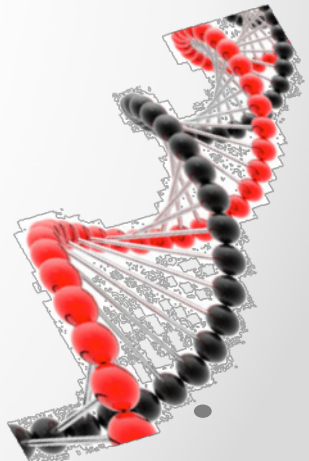


- Show schematic and ask lambda = 0 ? TD(0)
- Lambda = 1 ? MC
- Read and summarize Reinforcement Learning for Improving Agent Design
  - https://arxiv.org/pdf/1810.03779.pdf



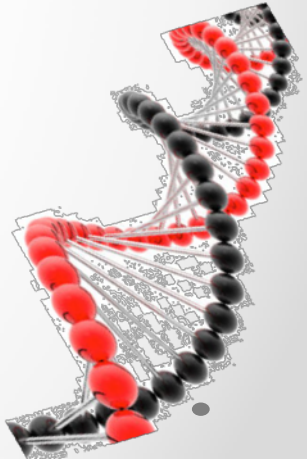


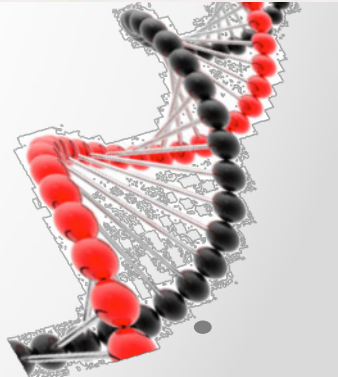
6. Conclusion



We use our visual system not only to survive & navigate, but also to socialize, entertain, understand & learn the world @drfeifei

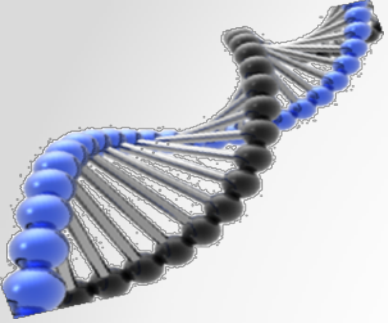
Can a machine understand the nuances and humor in this picture ?



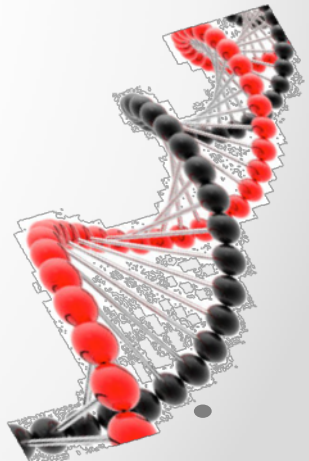


I had a good time researching & preparing for this Tutorial.

I hope you learned a few new things & inspired to follow a few vectors



WIP, Notes & backup



Proposal Links

- http://proposals.gputechconf.com/gtc-2019-training-sessions/edit.php?s=JYs1r9U3CUsh3ADzPpgs
- http://proposals.gputechconf.com/gtc-2019-training-sessions/edit.php?s=vYOO796coqXyMGcIfbK7