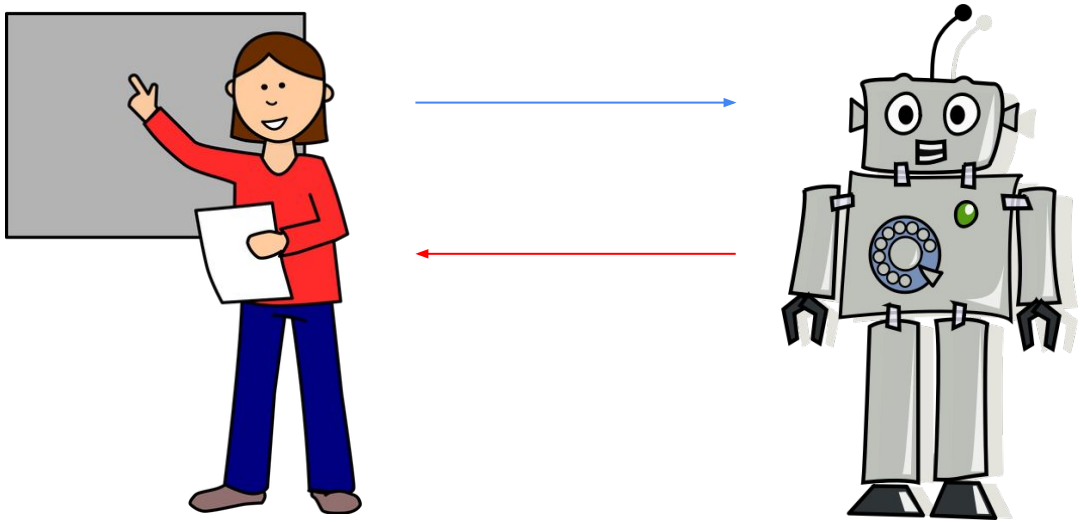
Deep Reinforcement Learning

Presented by Jonathan Zamora, Stone Tao, and Arth Shukla

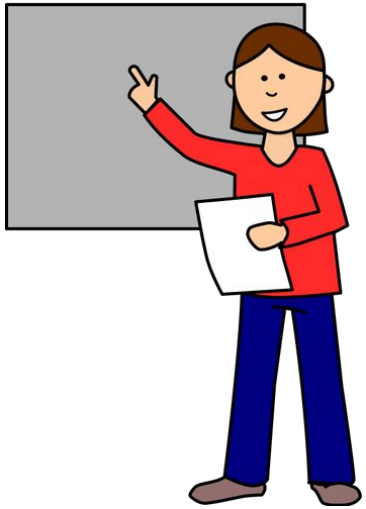
ai-sp-ws3

The Path Towards Intelligence



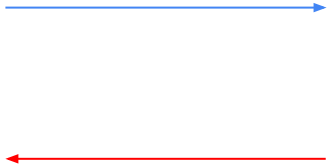
Teacher

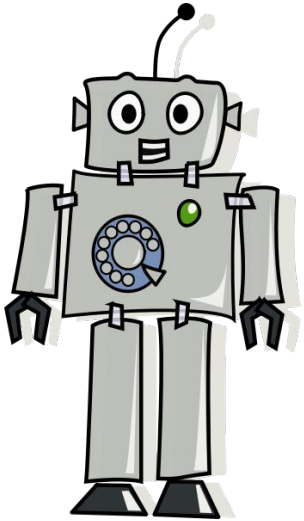
Robot



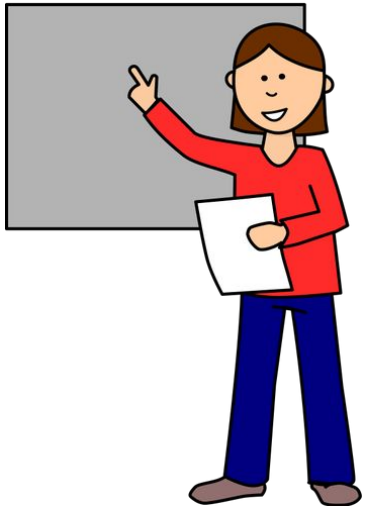
Teacher

Data



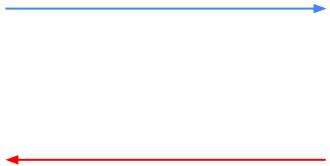


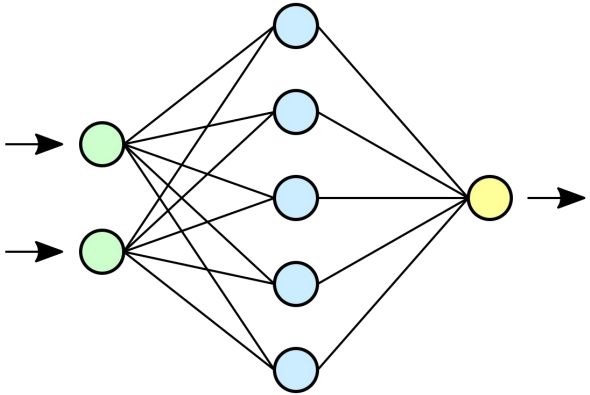
Robot



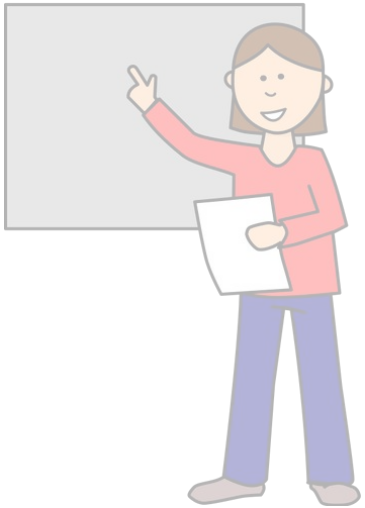
Teacher

Data



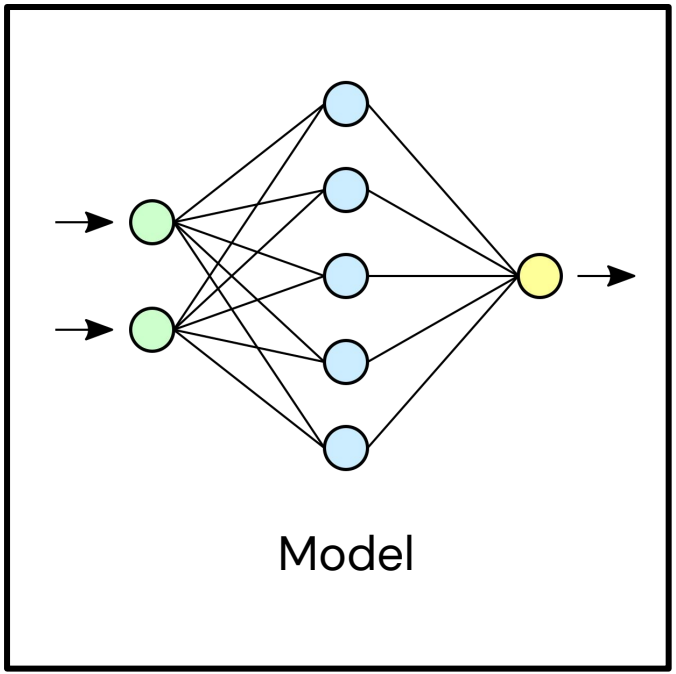


Model

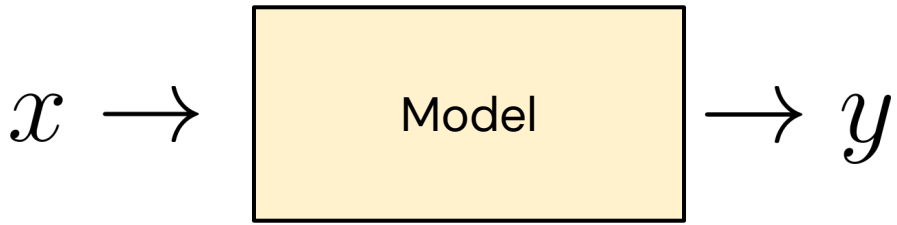


Teacher

Data



What is a Model?



What is a Model?

image

$x \rightarrow$ Model $\rightarrow y$

What is a Model?

image

$x \rightarrow$ Model $\rightarrow y$

Question: What game am I playing?

What is a Model?

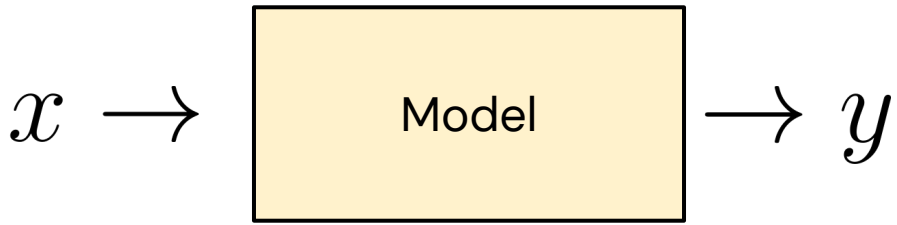
image

$x \rightarrow$ Model $\rightarrow y$

Prediction: Super Mario Bros

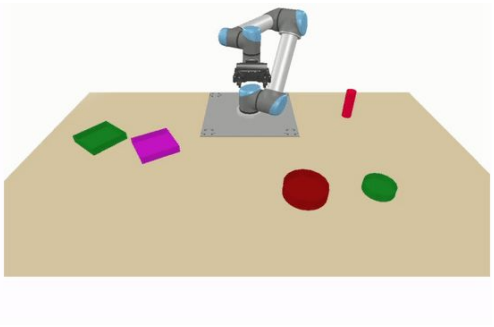
Question: What game am I playing?

Can we utilize models for making decisions?



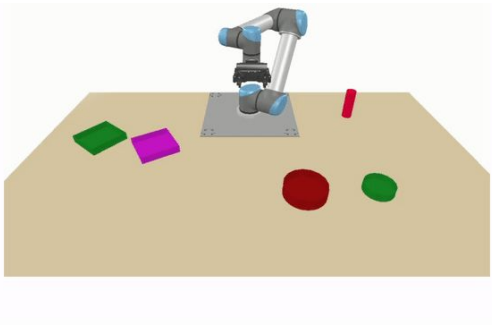
Games

Games

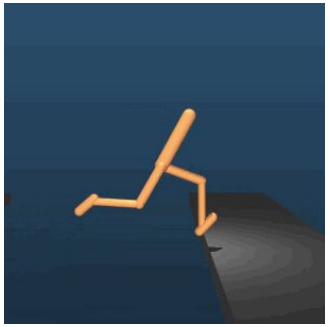


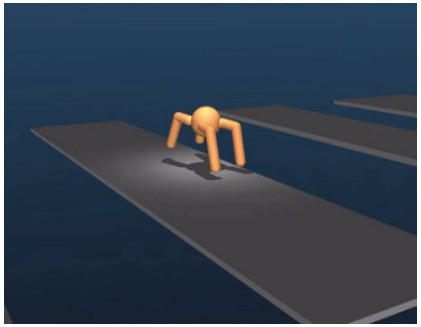
Robotics

Games



Robotics

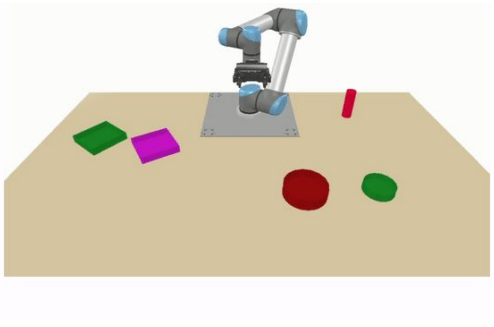




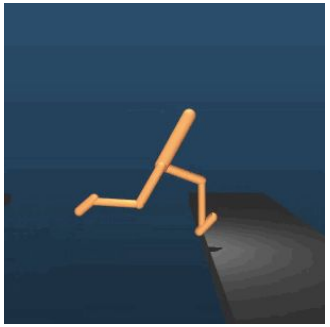
Control Problems

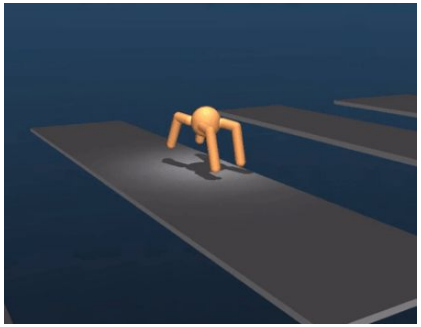
The Key: Deep Reinforcement Learning

Games



Robotics





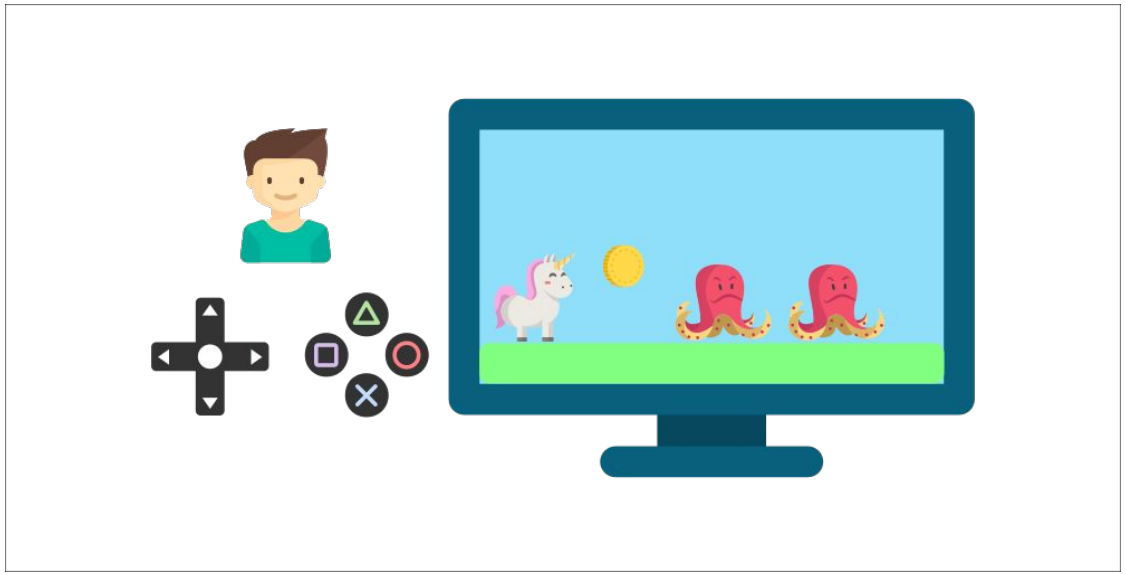
Control Problems

The Key: Deep Reinforcement Learning

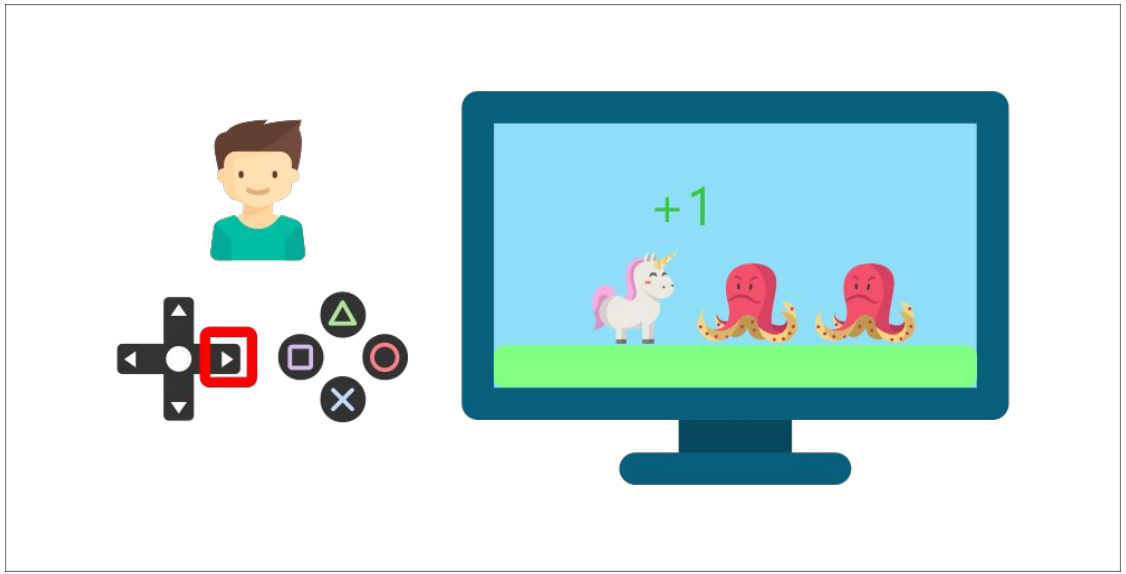
An Agent (an AI) will learn from its environment by:

- **interacting** with it (through trial and error)
- receiving **Rewards** (negative (-) or positive (+)) after performing actions

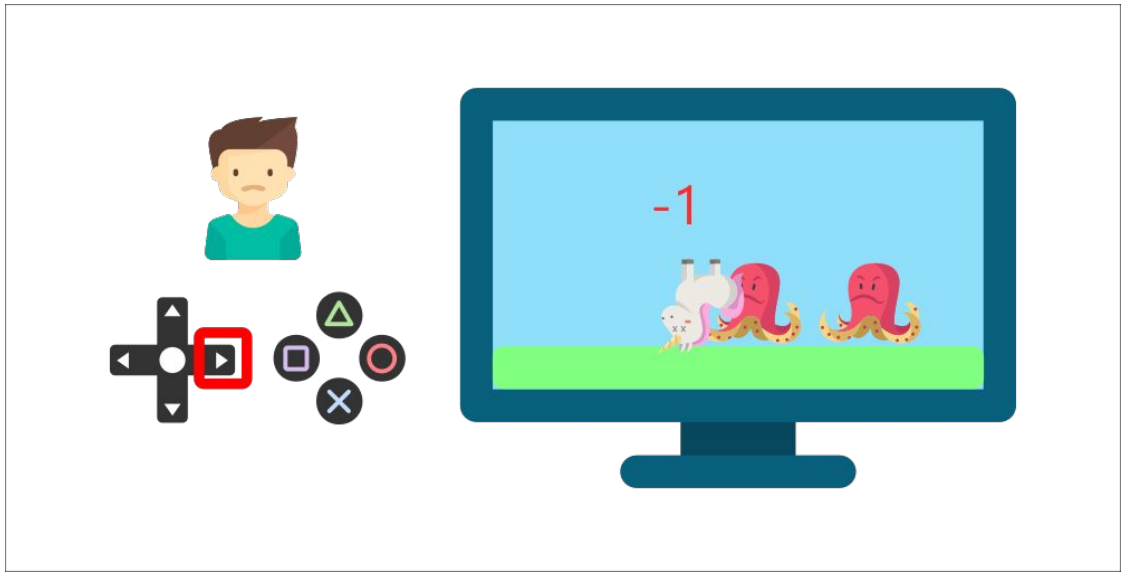
For Instance:



For Instance:

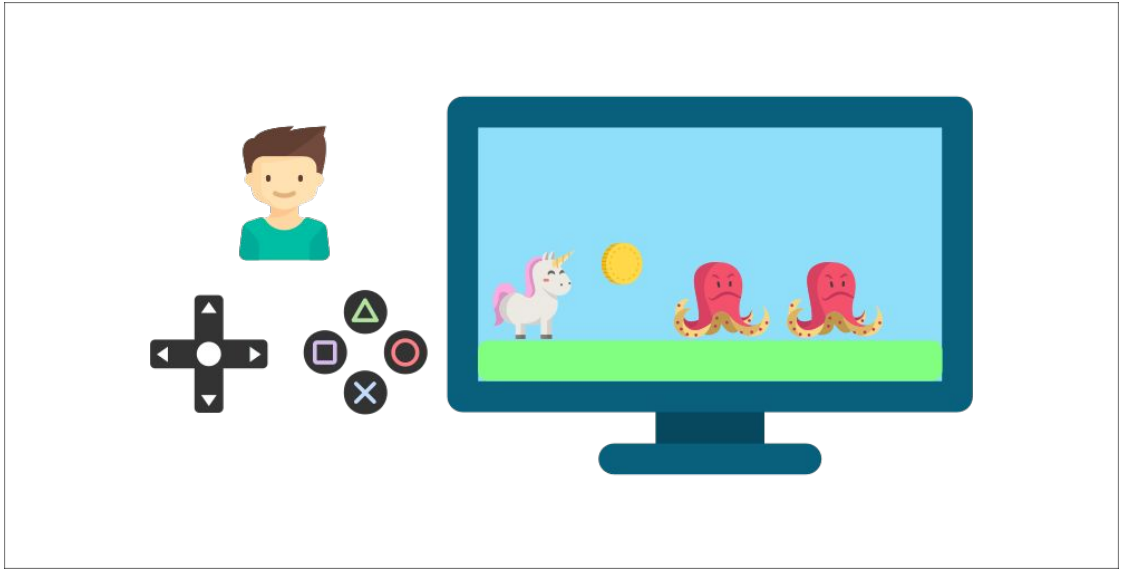


For Instance:



For Instance:

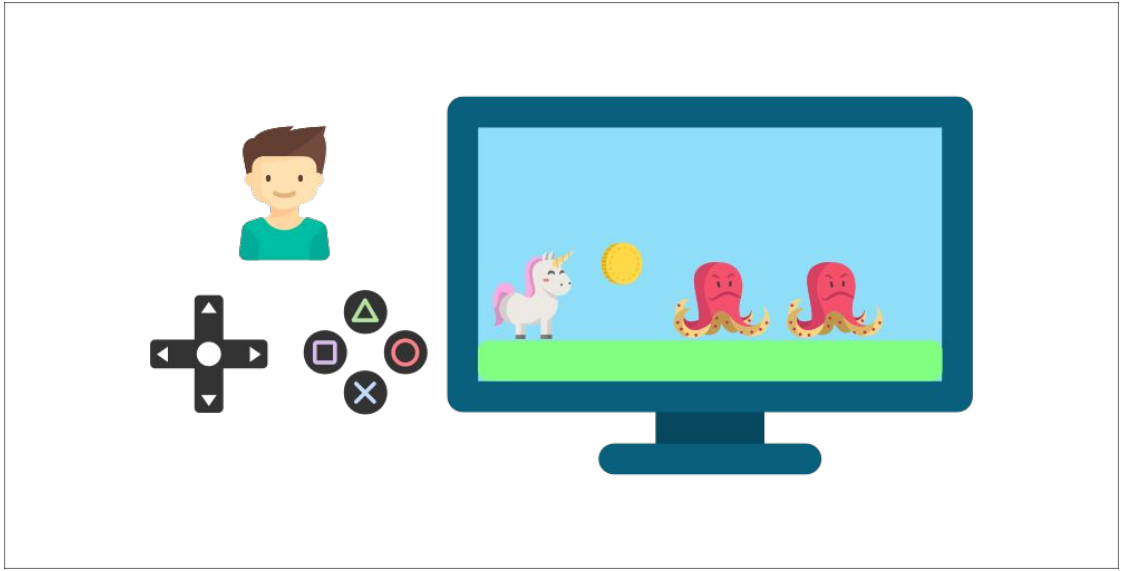
Without supervision, the child will get better at the game



For Instance:

Without supervision, the child will get better at the game

This is how humans and animals learn – **through interaction**

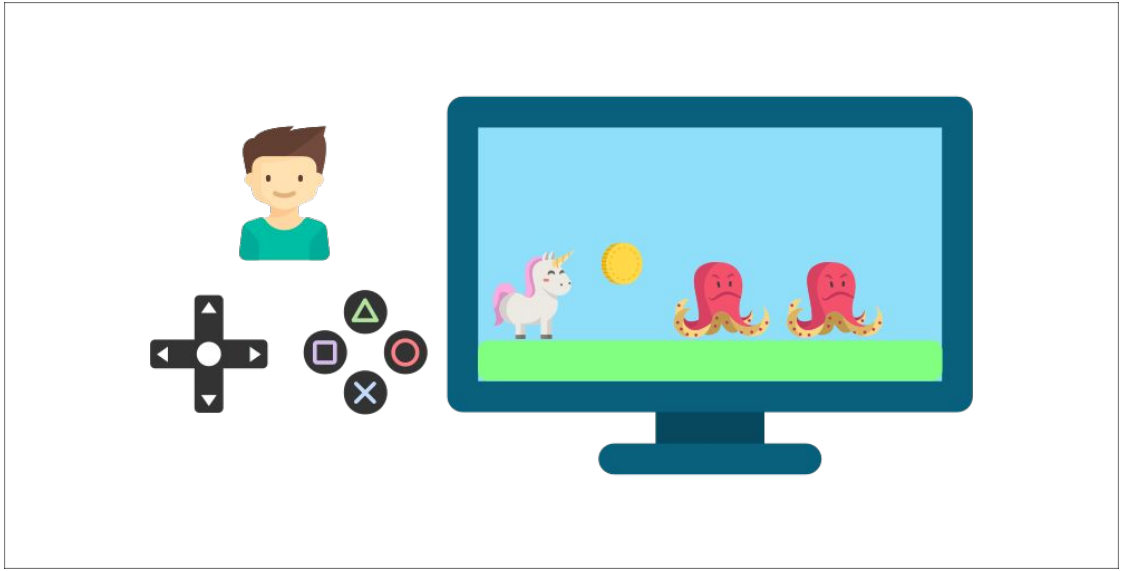


For Instance:

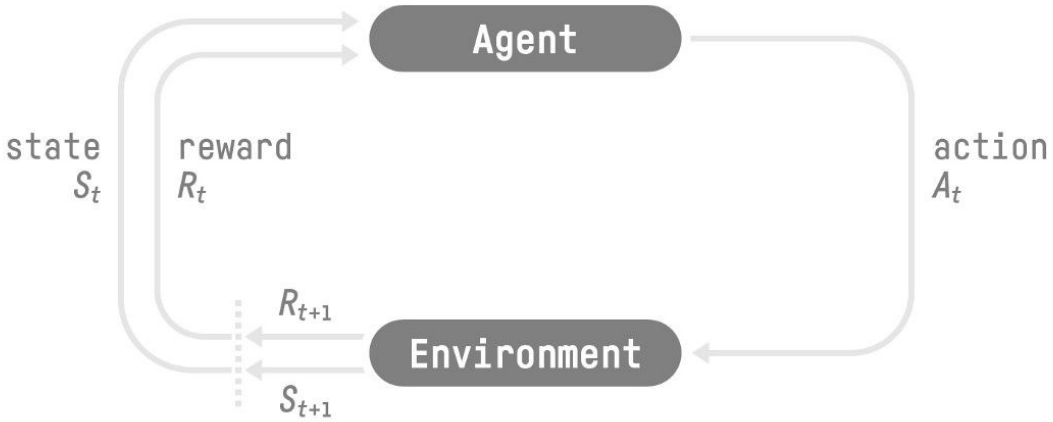
Without supervision, the child will get better at the game

This is how humans and animals learn – **through interaction**

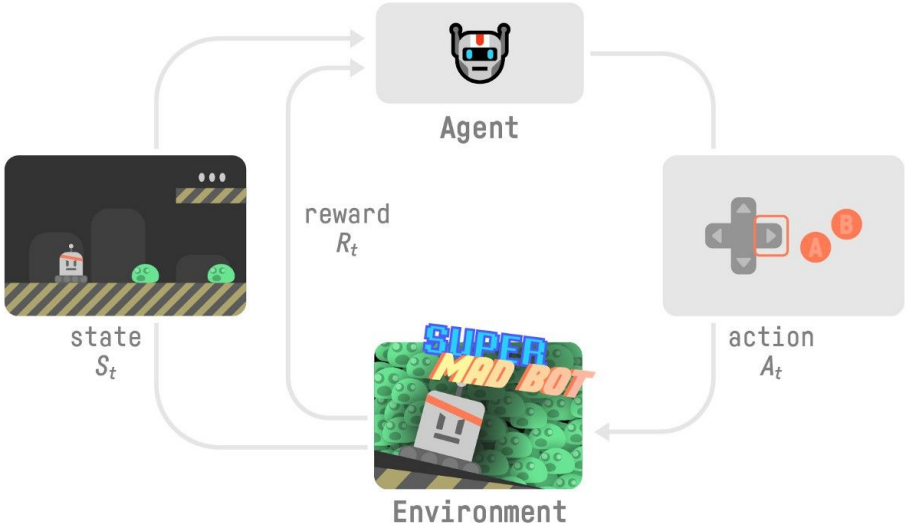
Reinforcement Learning is just a **computational approach for learning from actions**



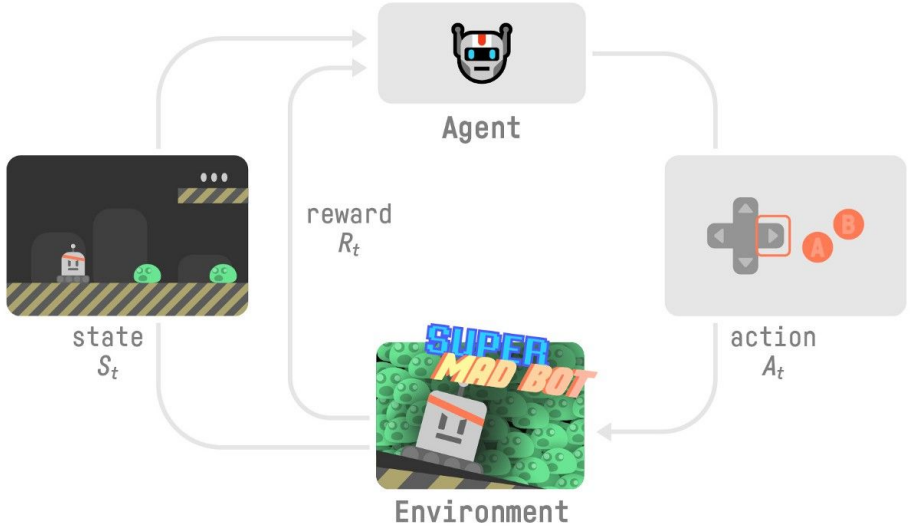
The RL Process



The RL Process: Platformer Game Example



The RL Process: Platformer Game Example



S_0, A_0, R_1, S_1

State Action Reward Next State

Observations/State Space

State S – A complete description of the state of the world. In a fully observed environment.

Observations/State Space

State S – A complete description of the state of the world. In a fully observed environment.

Observation O – A partial description of the state of the world. In a partially observed environment.

Action Space

Discrete Space: # of possible actions is finite

Action Space

Discrete Space: # of possible actions is finite

Continuous Space: # of possible actions is infinite

Rewards

$$R(\tau) = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + \dots$$

Return: cumulative reward

Trajectory (read Tau)
Sequence of states and actions

Rewards

$$R(\tau) = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + \dots$$

Return: cumulative reward

Trajectory (read Tau)
Sequence of states and actions

Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

Rewards and Discounting

Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

1 Problem: We can't just add the rewards like this.

Rewards and Discounting

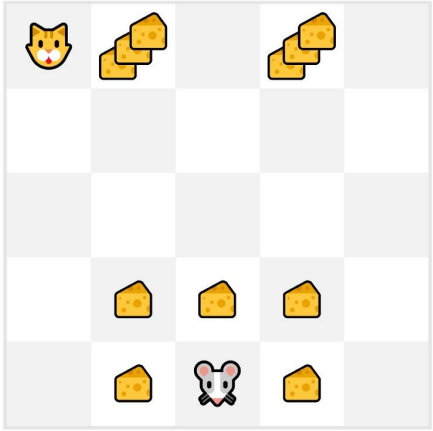
Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

1 Problem: We can't just add the rewards like this.

Rewards at start of game are more likely to happen since they are more predictable than future reward

Rewards and Discounting



Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

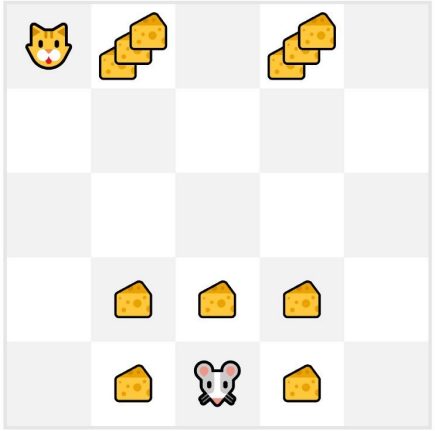
1 Problem: We can't just add the rewards like this.

Rewards at start of game are more likely to happen since they are more predictable than future reward

Rewards and Discounting

We define a discount rate **gamma**

- Must be between 0 and 1



Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

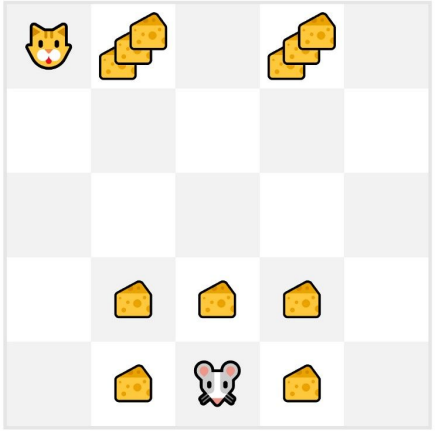
1 Problem: We can't just add the rewards like this.

Rewards at start of game are more likely to happen since they are more predictable than future reward

Rewards and Discounting

We define a discount rate **gamma**

- Must be between 0 and 1
- Large Gamma, Smaller Discount - We care about long-term reward



Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

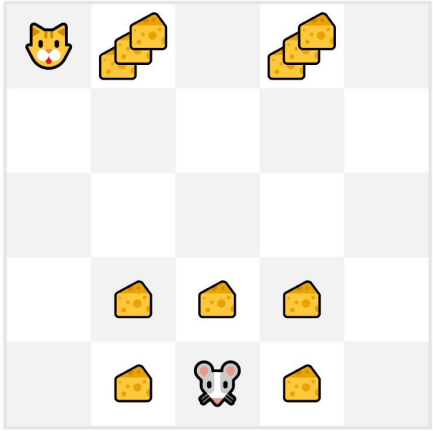
1 Problem: We can't just add the rewards like this.

Rewards at start of game are more likely to happen since they are more predictable than future reward

Rewards and Discounting

We define a discount rate **gamma**

- Must be between 0 and 1
- Large Gamma, Smaller Discount - We care about long-term reward
- Small Gamma, Larger Discount - We care about short-term reward



Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

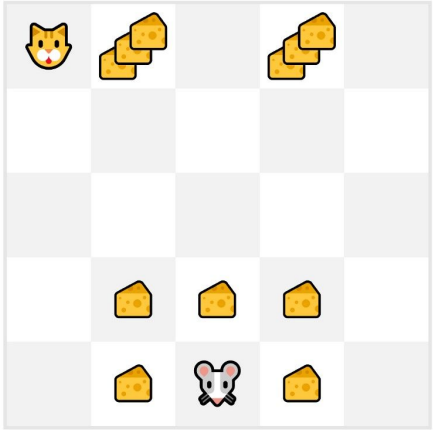
1 Problem: We can't just add the rewards like this.

Rewards at start of game are more likely to happen since they are more predictable than future reward

Rewards and Discounting

We define a discount rate **gamma**

- Must be between 0 and 1
- Large Gamma, Smaller Discount - We care about long-term reward
- Small Gamma, Larger Discount - We care about short-term reward
- Each reward is discounted by gamma to the exponent of the timestep



Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

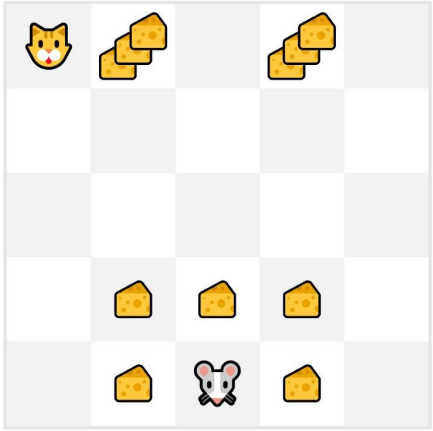
1 Problem: We can't just add the rewards like this.

Rewards at start of game are more likely to happen since they are more predictable than future reward

Rewards and Discounting

We define a discount rate **gamma**

- Must be between 0 and 1
- Large Gamma, Smaller Discount - We care about long-term reward
- Small Gamma, Larger Discount - We care about short-term reward
- Each reward is discounted by gamma to the exponent of the timestep
- As timestep increases, the cat gets closer to us, so future reward is less likely to happen



Cumulative Reward

$$R(\tau) = \sum_{k=0}^{\infty} r_{t+k+1}$$

1 Problem: We can't just add the rewards like this.

Rewards at start of game are more likely to happen since they are more predictable than future reward

Rewards and Discounting

We define a discount rate **gamma**

- Must be between 0 and 1
- Large Gamma, Smaller Discount - We care about long-term reward
- Small Gamma, Larger Discount - We care about short-term reward
- Each reward is discounted by gamma to the exponent of the timestep
- As timestep increases, the cat gets closer to us, so future reward is less likely to happen

Discounted Cumulative Reward

$$R(\tau) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots$$

Return: cumulative reward

Gamma: discount rate

Trajectory (read Tau)
Sequence of states and actions

$$R(\tau) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Types of Tasks

Episodic Task: We have a starting point and an ending point (terminal state)

This creates an Episode - a list of States, Actions, Rewards, and Next States

Types of Tasks

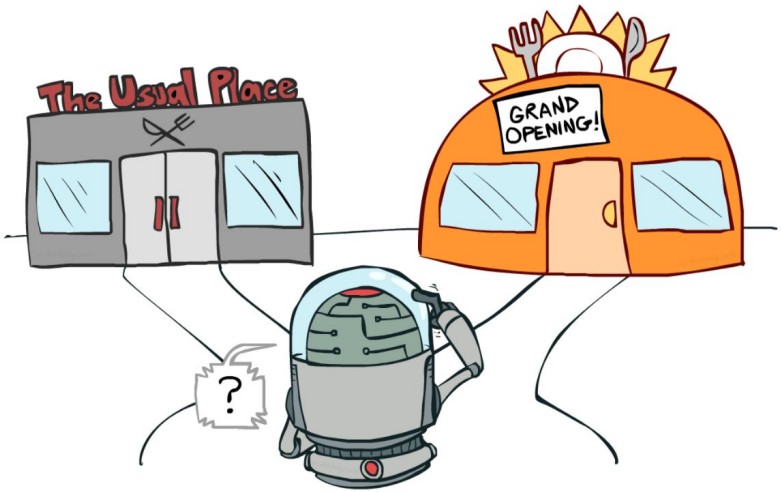
Episodic Task: We have a starting point and an ending point (terminal state)

This creates an Episode - a list of States, Actions, Rewards, and Next States

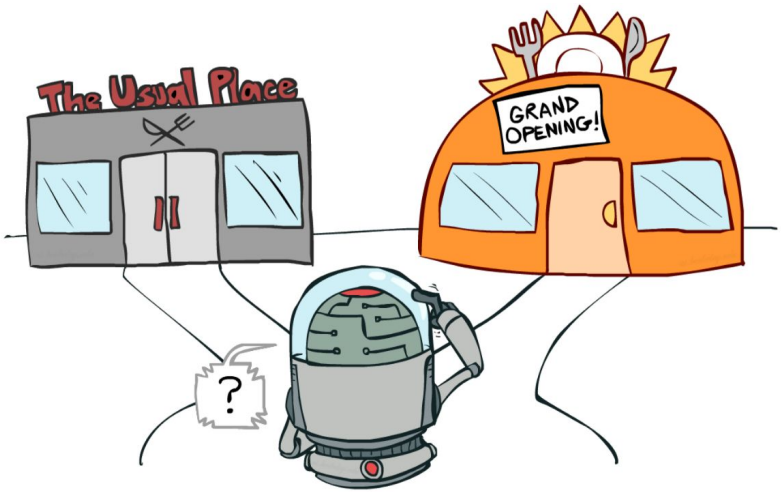
Continuing Task: We have no terminal state

The agent must learn (1) how to choose best actions and (2) how to best interact with the environment

Exploration and Exploitation Tradeoff

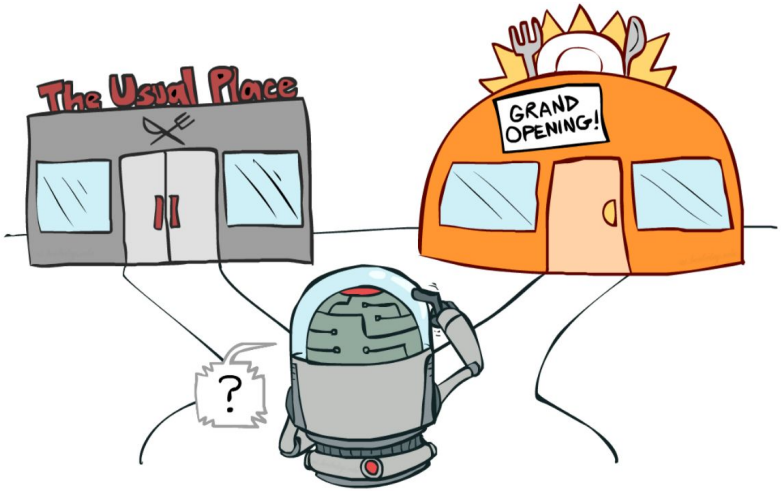


Exploration and Exploitation Tradeoff



Exploitation – Exploit known information to maximize reward

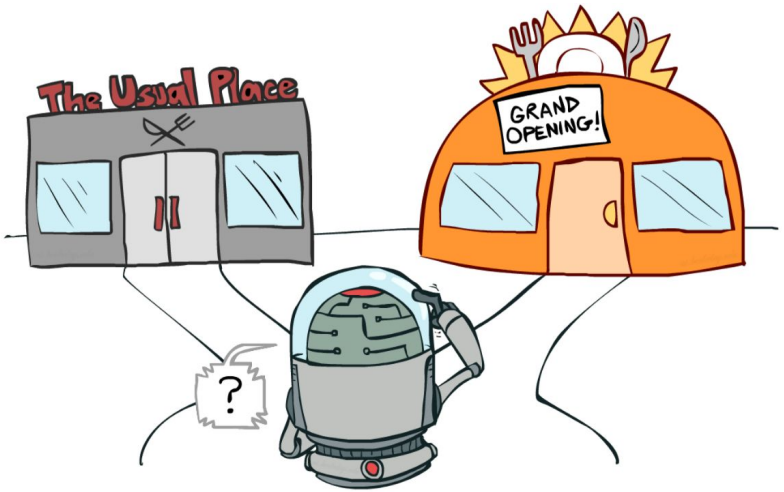
Exploration and Exploitation Tradeoff

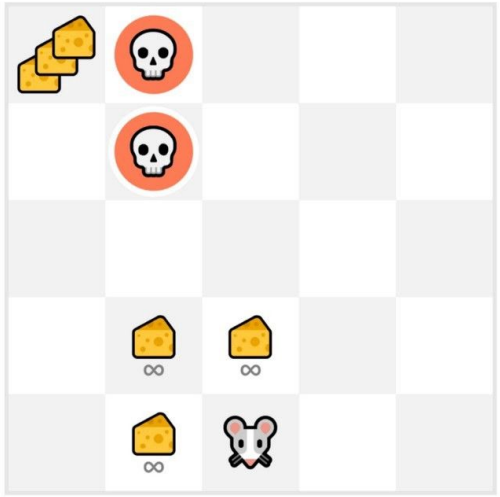


Exploitation – Exploit known information to maximize reward

Exploration – Explore environment by trying random actions to learn more about the env

Exploration and Exploitation Tradeoff



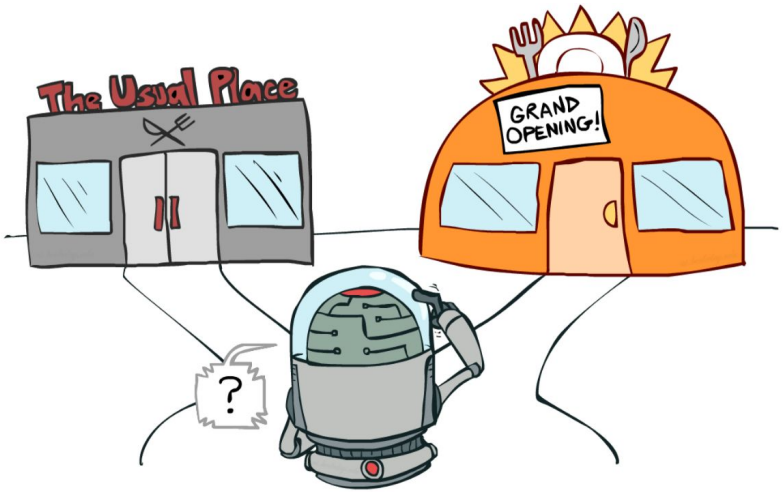


Exploitation – Exploit known information to maximize reward

Exploration – Explore environment by trying random actions to learn more about the env

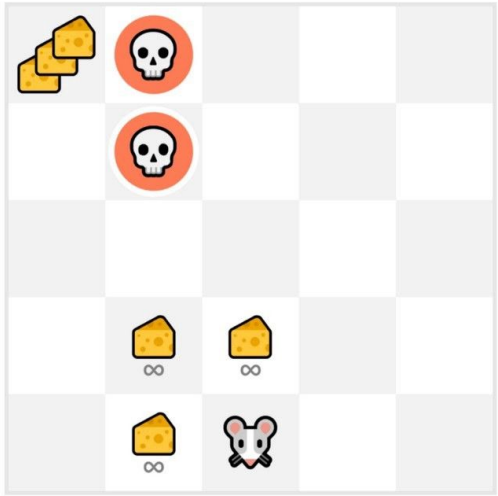
Our mouse can have an infinite amount of small cheese (+1)

Exploration and Exploitation Tradeoff



Exploitation – Exploit known information to maximize reward

Exploration – Explore environment by trying random actions to learn more about the env

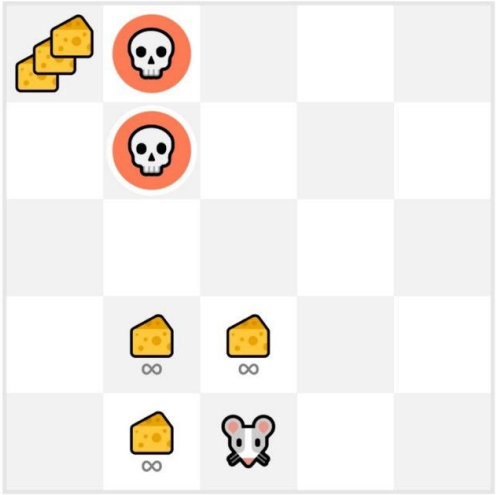


Our mouse can have an infinite amount of small cheese (+1)

However, there is a gigantic sum of cheese (+1000) at the top of the maze

Exploration and Exploitation Tradeoff

This is the Exploration and Exploitation Tradeoff



Exploitation – Exploit known information to maximize reward

Exploration – Explore environment by trying random actions to learn more about the env

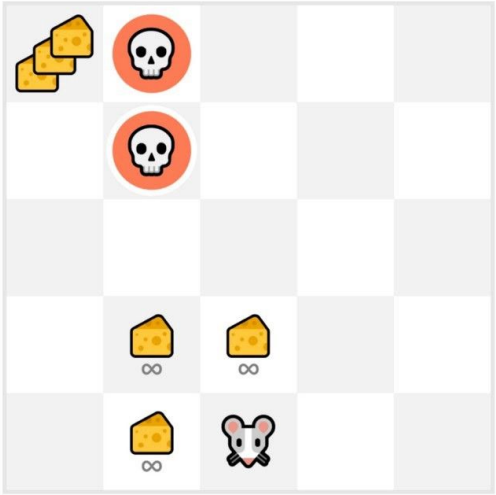
Our mouse can have an infinite amount of small cheese (+1)

However, there is a gigantic sum of cheese (+1000) at the top of the maze

Exploration and Exploitation Tradeoff

We must define a rule to help handle this tradeoff

We will go over this soon



Exploitation – Exploit known information to maximize reward

Exploration – Explore environment by trying random actions to learn more about the env

Our mouse can have an infinite amount of small cheese (+1)

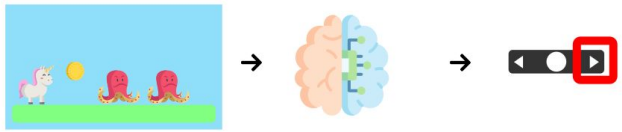
However, there is a gigantic sum of cheese (+1000) at the top of the maze

2 Main Approaches for Solving Problems w/ RL

How do we build an RL agent that can select actions which maximize its expected cumulative reward?

2 Main Approaches for Solving Problems w/ RL

How do we build an RL agent that can select actions which maximize its expected cumulative reward?



State → π(State) → Action

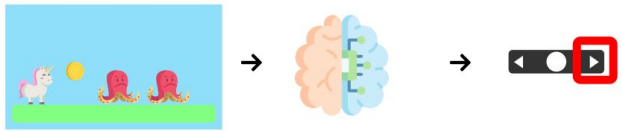
The Policy π: The Agent's Brain

This is a function which tells us what action to take, given the state we are in

This is a function we want to **learn**, and our goal is find the optimal policy **π***, the policy which **maximizes expected return**

2 Main Approaches for Solving Problems w/ RL

How do we build an RL agent that can select actions which maximize its expected cumulative reward?



State → π(State) → Action

The Policy π: The Agent's Brain

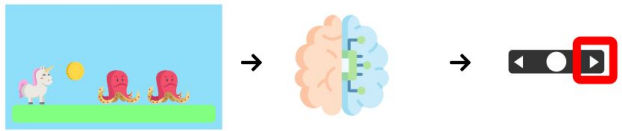
This is a function which tells us what action to take, given the state we are in

This is a function we want to **learn**, and our goal is find the optimal policy **π***, the policy which **maximizes expected return**

There are 2 approaches for training our agent to find the optimal policy π*

2 Main Approaches for Solving Problems w/ RL

How do we build an RL agent that can select actions which maximize its expected cumulative reward?



State → π(State) → Action

The Policy π: The Agent's Brain

This is a function which tells us what action to take, given the state we are in

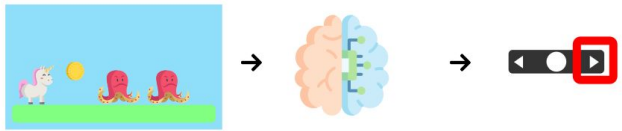
This is a function we want to **learn**, and our goal is find the optimal policy **π***, the policy which **maximizes expected return**

There are 2 approaches for training our agent to find the optimal policy π*

1. **Directly**, by teaching the agent to **learn which action to take, given the current state** – this is a Policy-Based Method

2 Main Approaches for Solving Problems w/ RL

How do we build an RL agent that can select actions which maximize its expected cumulative reward?



State → π(State) → Action

The Policy π: The Agent's Brain

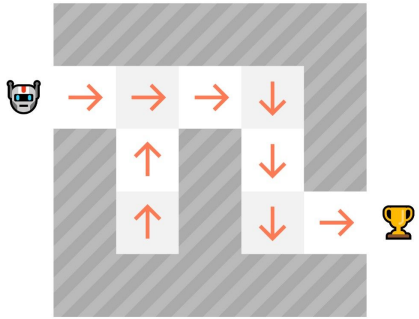
This is a function which tells us what action to take, given the state we are in

This is a function we want to **learn**, and our goal is find the optimal policy **π***, the policy which **maximizes expected return**

There are 2 approaches for training our agent to find the optimal policy π*

1. **Directly**, by teaching the agent to **learn which action to take, given the current state** – this is a Policy-Based Method
2. **Indirectly**, by teaching the agent to **learn which state is more valuable, and then take the action that leads to this more valuable state** – this is a Value-Based Method

Policy-Based Methods



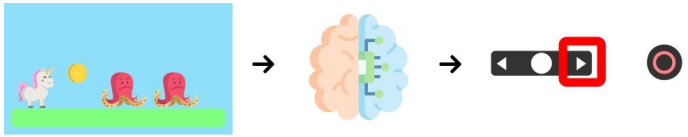
In Policy-Based Methods, **we learn a policy function directly**

This function will define a mapping from each state to the best corresponding action

Alternatively, it can also define a **probability distribution over the set of possible actions at that state**

$$a = \pi(s)$$

action = policy(state)



State s_0 → $\pi(s_0)$ → a_0 = Right

We have **2 Types of Policies**

1. Deterministic: A Policy at a Given State **will always return the same action**

e.g. pi(s_0) -> a_0 = Right

$$\pi(a|s) = P[A|s]$$

Probability Distribution over the set of actions given the state

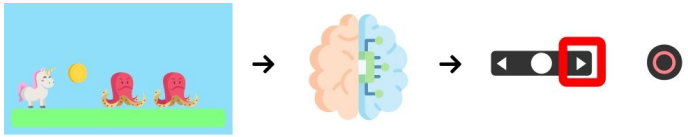
2. Stochastic: A Policy at a Given State **outputs a probability distribution over actions**

- Given an initial state, our stochastic policy will output the probability distributions over the possible actions at that state

e.g. pi(A|s_0) -> [Left: 0.1, Right: 0.7, Jump: 0.2]

$$a = \pi(s)$$

action = policy(state)



State s_0 → $\pi(s_0)$ → a_0 = Right

We have **2 Types of Policies**

1. Deterministic: A Policy at a Given State **will always return the same action**

e.g. pi(s_0) -> a_0 = Right

$$\pi(a|s) = P[A|s]$$

Probability Distribution over the set of actions given the state

2. Stochastic: A Policy at a Given State **outputs a probability distribution over actions**

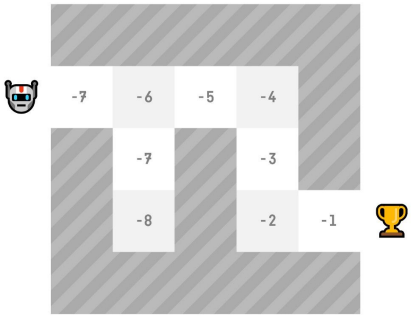
- Given an initial state, our stochastic policy will output the probability distributions over the possible actions at that state

e.g. pi(A|s_0) -> [Left: 0.1, Right: 0.7, Jump: 0.2]

Value-Based Methods

$$v_\pi(s) = \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s\right]$$

Value function — Expected discounted return — Starting at state s



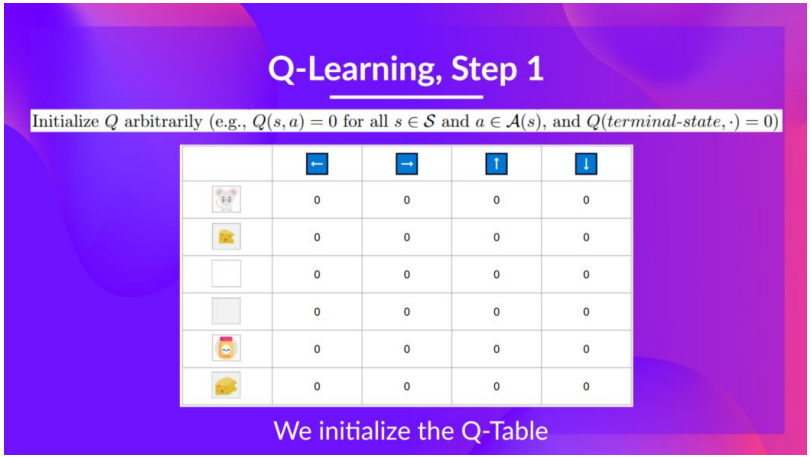
In Value-Based Methods, instead of learning a policy function, we learn a **value function indirectly**

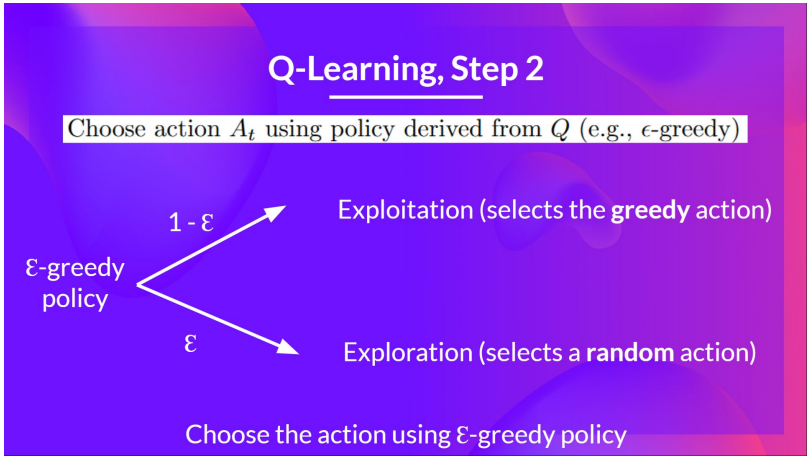
The value function maps a state to the expected value of being at that state

The value of a state is the **expected discounted return** the agent can get if it **starts in that state, and then acts according to the policy**

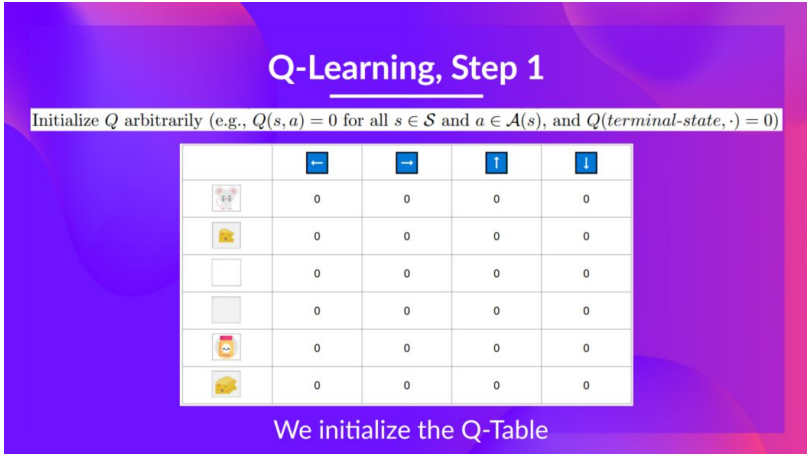
Our value function **defines values for each possible state**

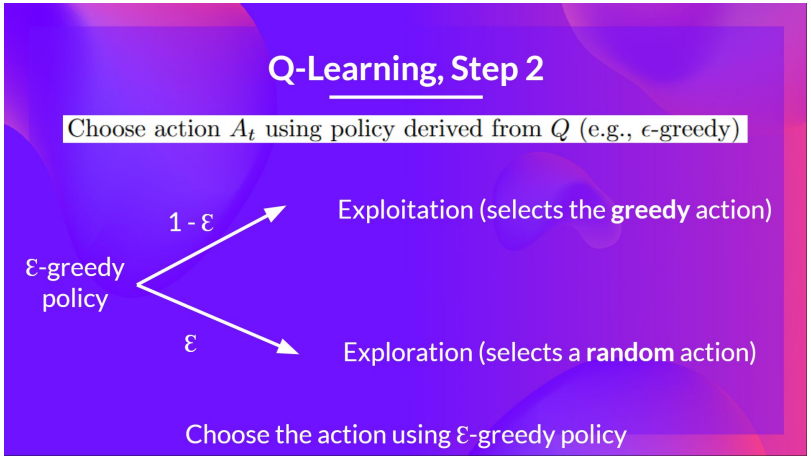
Our **policy will select the state w/ the largest value** defined by the value function: [-7, -6, -5, -4, -3, -2, -1] to attain the goal





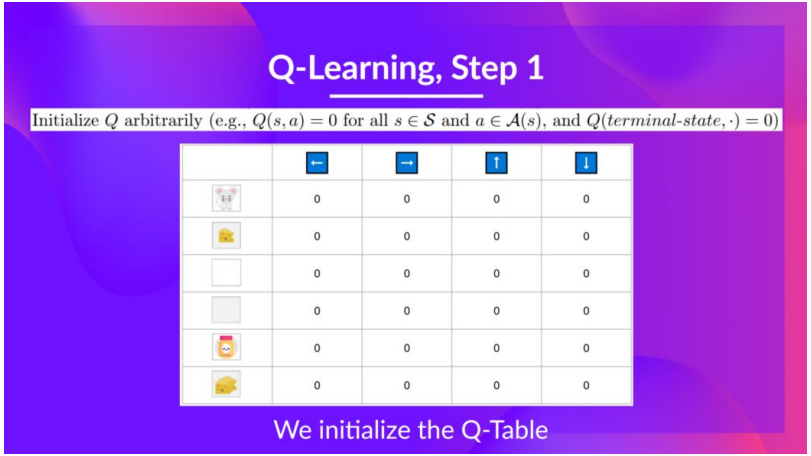
Q-Learning – We learn a Q-function, which is an **action-value function** that determines the value of being at a particular state and taking an action at that state. **Q comes from "the Quality" (the value) of the action at that state**

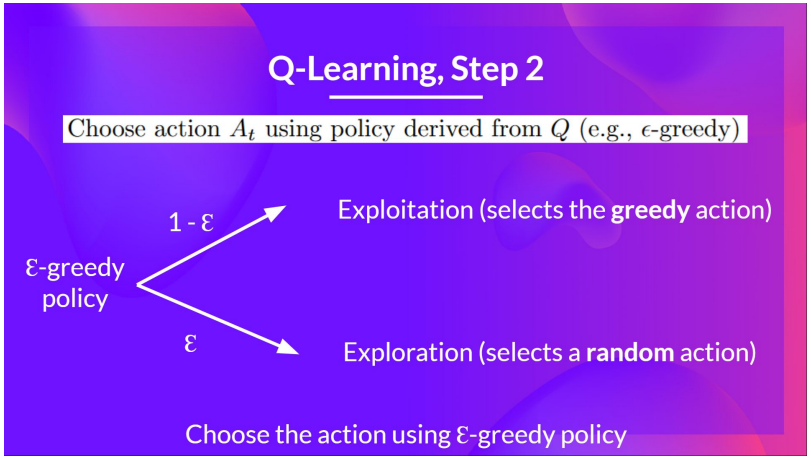




Q-Learning – We learn a Q-function, which is an **action-value function** that determines the value of being at a particular state and taking an action at that state. **Q comes from "the Quality" (the value) of the action at that state**

This is an **Off-Policy method** – it uses a **different policy for acting (inference) and updating (training)**





Q-Learning – We learn a Q-function, which is an **action-value function** that determines the value of being at a particular state and taking an action at that state. **Q comes from "the Quality" (the value) of the action at that state**

This is an **Off-Policy method** – it uses a **different policy for acting (inference) and updating (training)**

Updating uses **Epsilon-Greedy Policy**

1. With probability 1 - eps: we do exploitation (select action w/ highest state-action pair value)
2. With probability eps: we do exploration (random actions)

Acting uses a **Greedy Policy**

Atari State Space

160

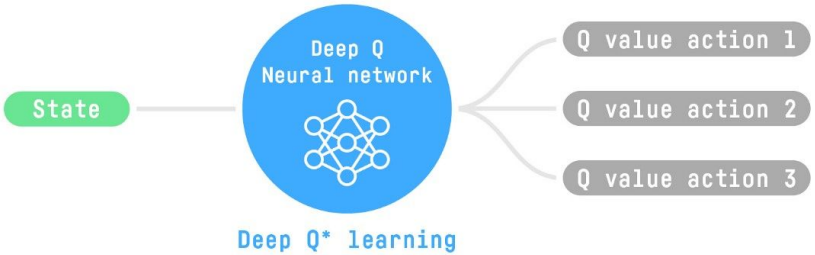
210

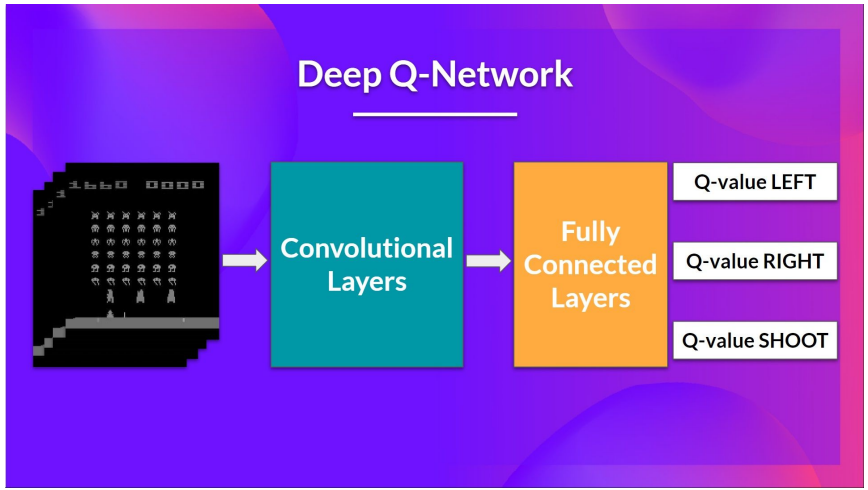
3

256 ^ (210*160*3)
256^100800

10^80 atoms in the Universe

Deep Q-Learning





We will approximate the Q-values using a parameterized Q-function

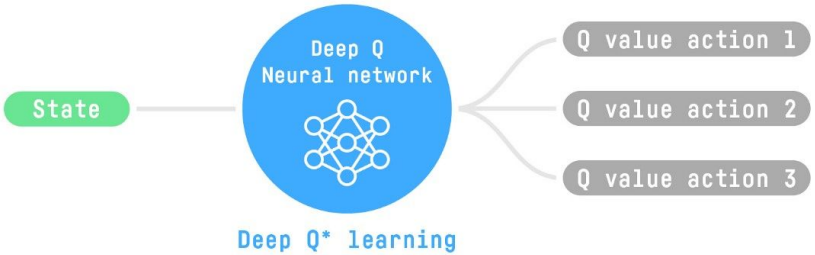
The neural network will approximate, given a state, the different Q-values for each possible action at that state.

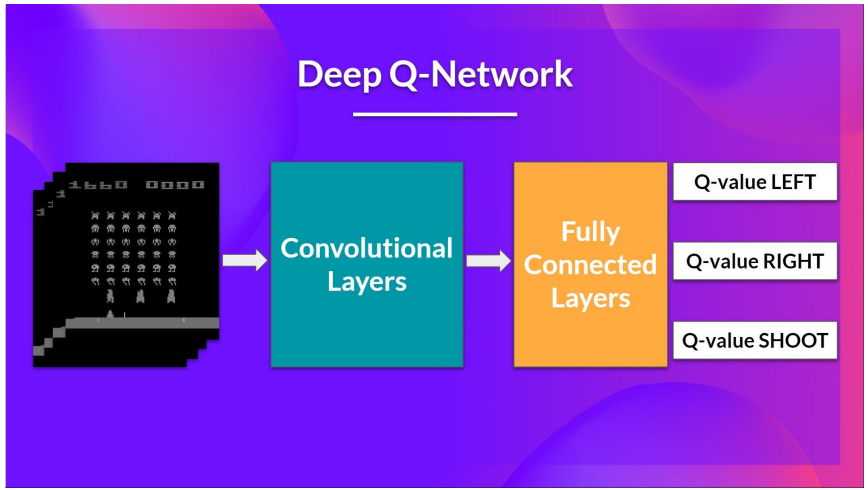
This approach is **Deep Q-Learning**

Input: Stack of 4 Frames
Output: Vector of Q-values for each possible action at that state

Like Q-learning, we use epsilon-greedy policy to select which action to take during training

Deep Q-Learning

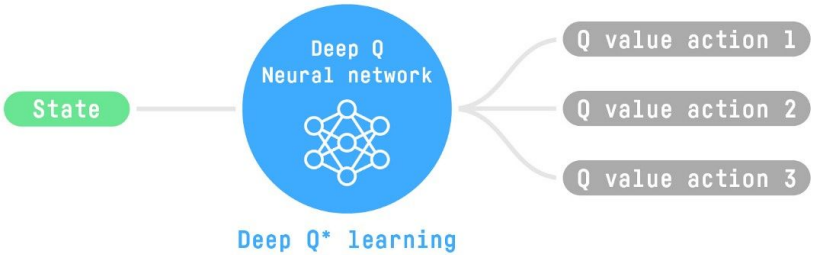




At first, the Q-value estimation is **terrible**

However, during training, our DQN agent will associate a situation (from the frames) with the appropriate action and learn to play the game well

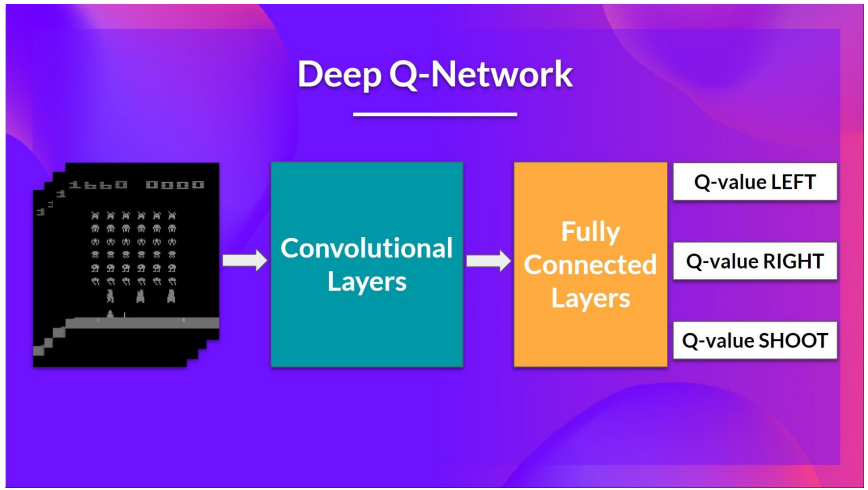
Deep Q-Learning

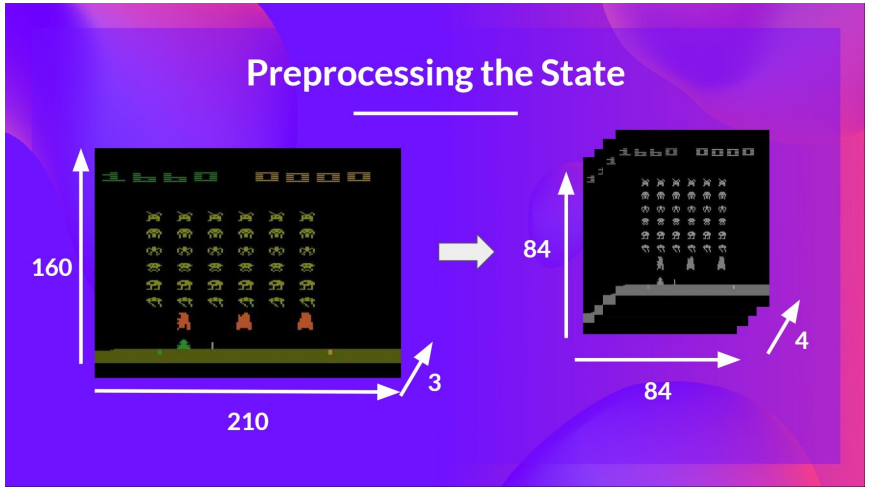


Preprocessing the input is essential since we want to reduce the complexity of our state to reduce computation time needed for training

Here, we reduce our state space to 84x84 and grayscale it.

Frame-stacking helps us handle the issue of temporal limitations





Deep Q-Learning

Double Q-Learning

With a high-level view of DQNs out of the way, we're going to discuss an altered algorithm, Double-DQN (DDQN)

But for that, we need to go over some math...

Double Q-Learning

Terminology: $Q(s_t , a_t)$:= value of action a_t at state s_t

Taking action a_t at state s_t yields state s_{t+1} with reward r_{t+1}

i.e. our action is “good” if it

1. Yields a high reward
2. Yields a state with high quality (high Q value)

Therefore, we calc $Q(s_t , a_t)$ as the reward of taking this step + the (discounted) Q value of the best possible action at the resultant state.

Finally, we have $Q(s_t , a_t) = \gamma \max_a Q(s_{t+1} , a) + r_{t+1}$

Double Q-Learning

Ideally, $Q(s_t, a_t) = \gamma \max_a Q(s_{t+1}, a) + r_{t+1}$

=> Optimal when $0 = \gamma \max_a Q(s_{t+1}, a) + r_{t+1} - Q(s_t, a_t)$

=> Temporal difference error, $TD = \gamma \max_a Q(s_{t+1}, a) + r_{t+1} - Q(s_t, a_t)$

Once TD → 0, we have finished optimizing

Introducing the Q-Learning update equation...

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

TD

Double Q-Learning **Issues**

Issues with Q-Learning update equation

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

- Works if precise values are stored for all state-action pairs
- Thrun and Schwartz (1993): if we *approximate* Q with a *function approximator* (like our nnet…)

$$Q^{\text{approx}}(s', \hat{a}) \quad = \quad Q^{\text{target}}(s', \hat{a}) \quad + \quad Y_{s'}^{\hat{a}}$$

  over time Q-Learning *overestimates* target values
- Our $\max_a Q(s_{t+1}, a)$ term converges to $\mathbb{E}(\max_a Q(s_{t+1}, a)) > \max_a Q_{\text{true}}(s_{t+1}, a)$

Double Q-Learning Issues

The big problem here is the max is doing too much heavy lifting:

$$\max_a Q(s_{t+1}, a)$$

There are two things happening here

1. Action evaluation (find Q value)
2. Action selection (pick a which yields max over Q)

What if we *separate* action selection from action selection?

Double Q-Learning

Idea: *separate* action selection from action selection by creating a local approx and target approx

$$Q^L(s_t, a_t) \leftarrow Q^L(s_t, a_t) + \alpha(r_{t+1} + \gamma \max_{a^*} Q^T(s_{t+1}, a^*) - Q^L(s_t, a_t))$$

The algorithm is as follows:

1. Pick a* which maximizes Q^L in the next state
2. Calc Q^T over next state using a*
3. Plug into update equation (which uses Q^L for everything else)

Deep Double Q Networks

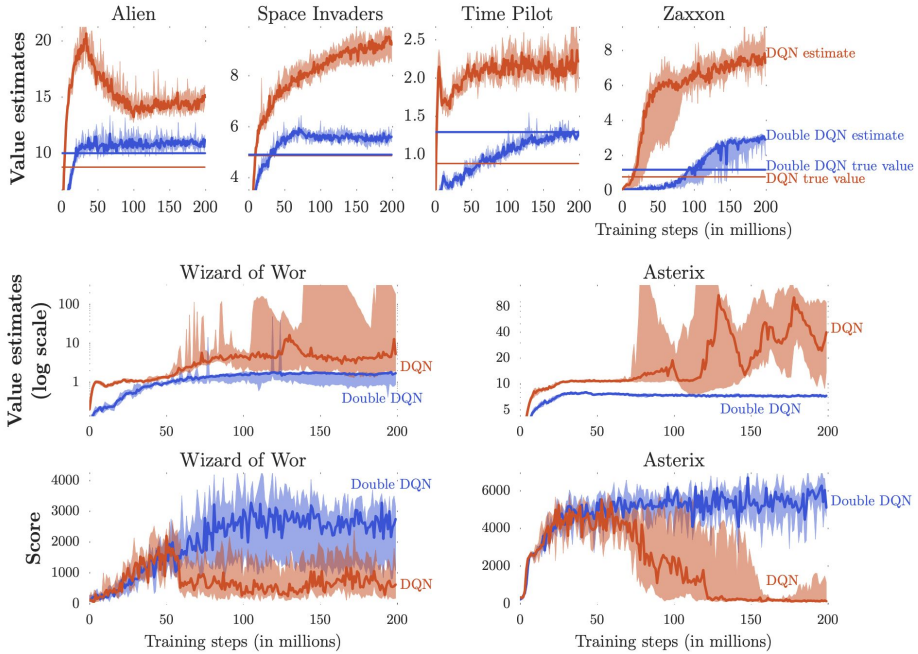
Loss Function

1. Use neural nets as our Q function approximators (local net and target net)
2. Construct loss function to minimize our temporal difference error:

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

Experience Replay: we perform experience replay as normal, but we update our target network after C steps (a hyperparam) to contain the same params as local network

Deep Double Q Networks



Hasselt, Guez, and Silver, 2015

Mario Time!

https://acmurl.com/mario-ddqn

Wahoo!

(D)DQNs vs The World - Action Sampling

1. **(D)DQNs are deterministic policies**

   Exploitation and exploration **separated**

   i. Exploration is random action, i.e. sample from *discrete continuous distribution*
   ii. Exploitation Our local network spits out q values and we select the max; **no variability, hence deterministic**

   Say you know how to drive a Prius. You just bought an F150 Truck. When you learn to drive the truck, do you do randomly try the AC, windows, e-brake, etc, or do you turn the ignition **assuming** the truck will turn on?

   Hence Alternative: Stochastic Policies

(D)DQNs vs The World - Action Sampling

1. **(D)DQNs are deterministic policies**

   Stochastic policies encode action probabilities into a *distribution* (Categorical for discrete action space, Gaussian for continuous

   When we want to act, we **sample** from the distribution

   i. High probability == probably useful
   ii. Low probability == probably not useful

   Low probability actions are still sampled, so our model can explore!

   One less hyperparam B)

(D)DQNs vs The World - On or Off Policy?

2. **(D)DQNs are Off-Policy**

   (D)DQNs optimize Bellman Equation, *not* the policy – empirically can lead to better policy, but not *guaranteed to work*

   See more: Sutton and Barto Ch 11 (linked in speaker's notes)

   Bootstrap (random sampling w/ replacement) + value method + Q-approximation == failure cases

   Alternate Solution 1: Better Off-Policy algos, DDPG, TD3, SAC (SAC is research standard for off-policy)

   Alternate Solution 2: On-Policy, policy gradient, REINFORCE, TRPO, PPO (PPO is research standard for on-policy)

(D)DQNs vs The World - On or Off Policy?

2. **(D)DQNs are Off-Policy**

   (D)DQNs optimize Bellman Equation, *not* the policy – empirically can lead to better policy, but...

   When we want to act, we **sample** from the distribution

   i. High probability == probably useful
   ii. Low probability == probably not useful

   Low probability actions are still sampled, so our model can explore!

   See more: Sutton and Barto Ch 11 (linked in speaker's notes)

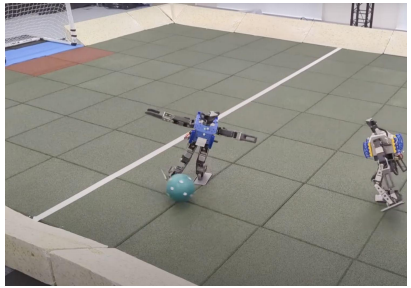
Current Problems in Deep RL

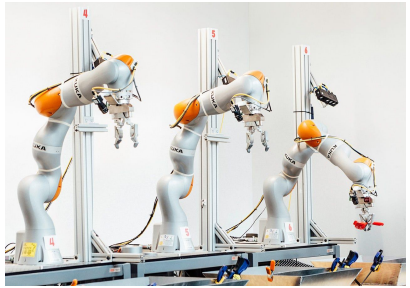
Presented by Stone Tao

Downstream Applications

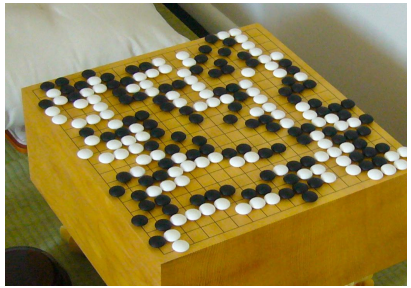
The well known ones

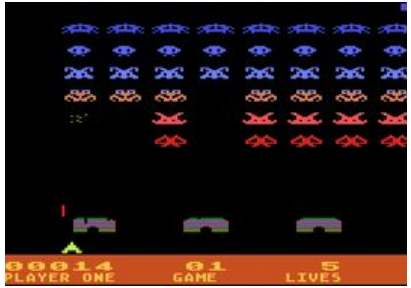
Robotics





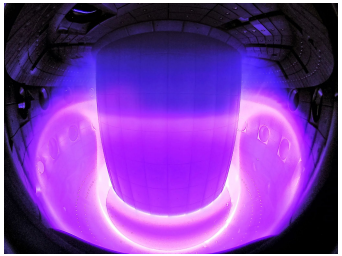
Games



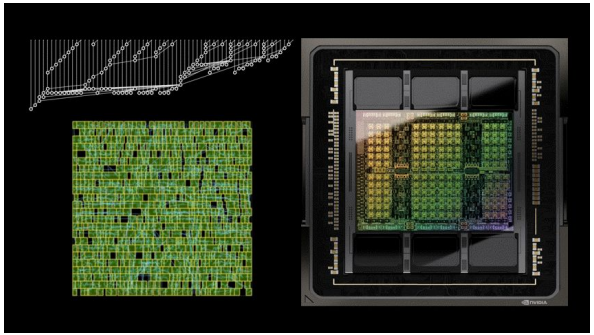


Downstream Applications

The not so well known ones

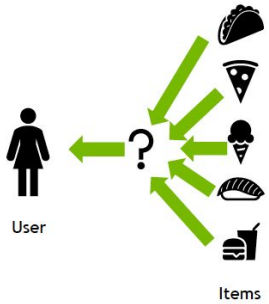


Nuclear fusion control



Chip Design

Policy making



Recommendation Systems

ChatGPT

Current Deep RL (stone)

Speed and efficiency: make training faster

Solving the impossible: do what we can't easily program solutions for

IMO these are the general 2 goals of RL research, with many points of overlap. The research that goes into make algorithms smarter and faster lie in both theory/algos and engineering.

Current Deep RL (stone)

Theory/Algorithms: Learning from rewards, learning from demos, leveraging world-models...

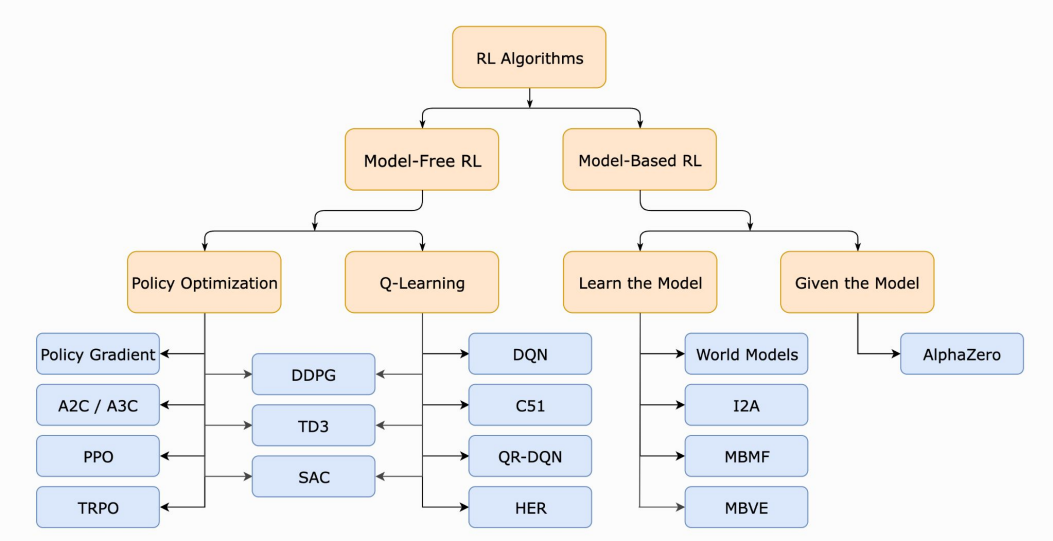
Engineering: Training speed, environment speed, scaling data collection and processing, software engineering, clever prompting...

Sample Efficiency

Goal: Make training go brrrr (fast)

However, reinforcement learning is well known to be extremely slow, how come? Humans when born can learn quite a few things in a few years but give a few years of experience to an AI it can’t do a whole lot.

Sample Efficiency

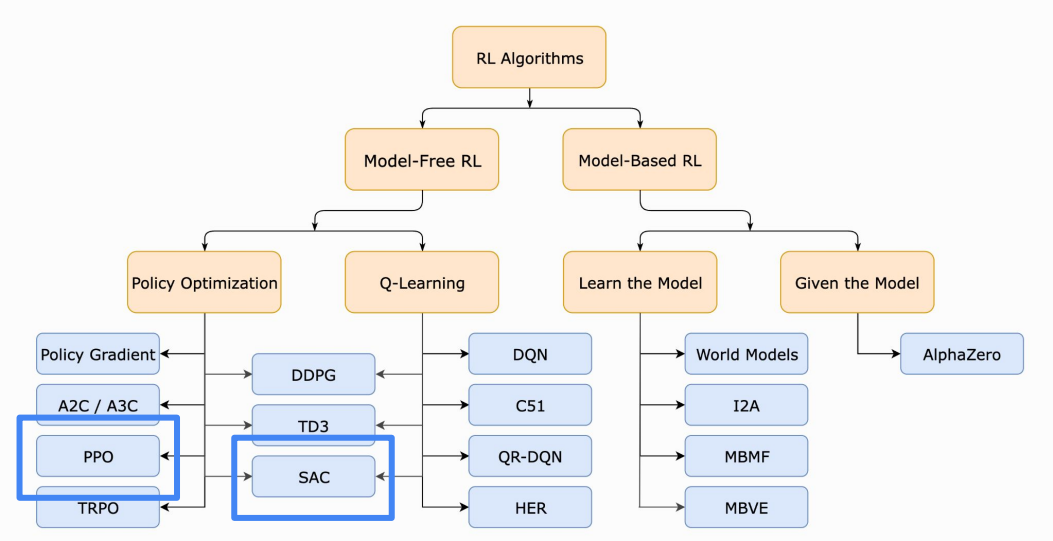


Model-Free RL: Fast inference/training but uses many samples without using world models (for e.g. planning actions ahead)

Model-Based RL: Slower inference/training due to use of a model, but needs less samples to make better decisions thanks to use of a world model

Image from: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html

Sample Efficiency



On-Policy RL:
Can only train on experiences for the current iteration of the policy - less sample inefficient

The de-facto standard: Proximal Policy Optimization (PPO)

Off-Policy RL:
Can train on any experience, doesn't have to come from the sampled policy - more sample efficient

The de-facto standard: Soft Actor Critic (SAC)

PPO (Schulman et al., 2017); SAC (Haarnoja et al., 2018)

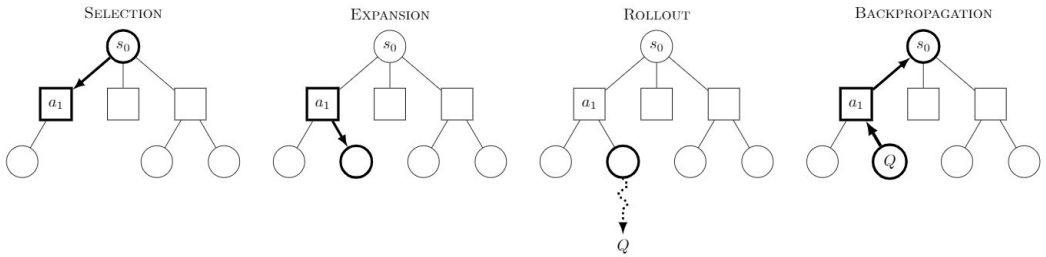
World Models

Goal: Improving sample efficiency

Generally an idea that is more reflective of how humans sometime work.

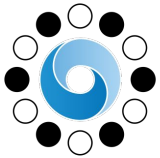
When playing the game of Go, Chess etc. you must think several moves ahead to more optimally make choices.

Modern model-based RL (RL that uses world models) are all some flavor of this, and typically learn this world model.



World Models

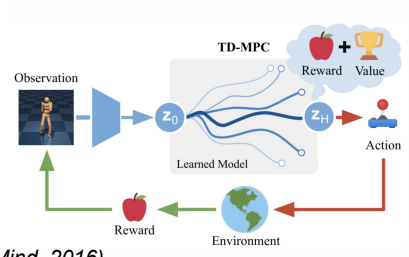
There are some various examples and flavors of using world models



AlphaGo

Planning in the original environment (MCTS)

Planning in observation space (images)



Planning in learned latent space (encoding of images)

TD-MPC (Hansen et al., 2022); DreamerV3 (Hafner et al., 2022); Alpha Go (DeepMind, 2016)

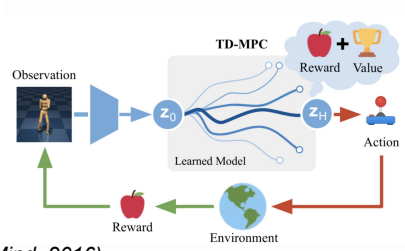
World Models

Learning a complex world model is hard and slow! Planning with them scales with number of steps ahead planned.
While DreamerV3 can solve complex minecraft tasks and TD-MPC can solve complex robotic control problems, both take a **long time to train**

AlphaGo

Planning in the original environment (MCTS)

Planning in observation space (images)



Planning in learned latent space (encoding of images)

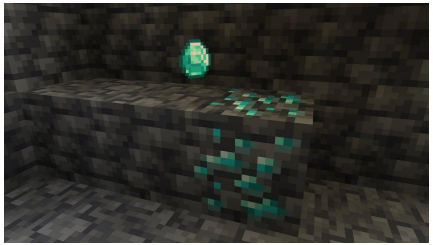
TD-MPC (Hansen et al., 2022); DreamerV3 (Hafner et al., 2022); Alpha Go (DeepMind, 2016)

Learning from Demonstrations

Goal: Tackle incredibly hard tasks where designing a reward may be difficult

Start

How to mine diamonds?
What rewards?



Goal 1: Diamonds

How to build a house?
What rewards?

Goal 2: Build a house

Learning from Demonstrations

Goal: Tackle incredibly hard tasks where designing a reward may be difficult

Getting diamonds are a long horizon, multi-staged problem…

Easily take 100 million samples on even the best algorithms, even with lots of hacks to make it easier

100M samples is about a month of straight playing time at 30FPS.

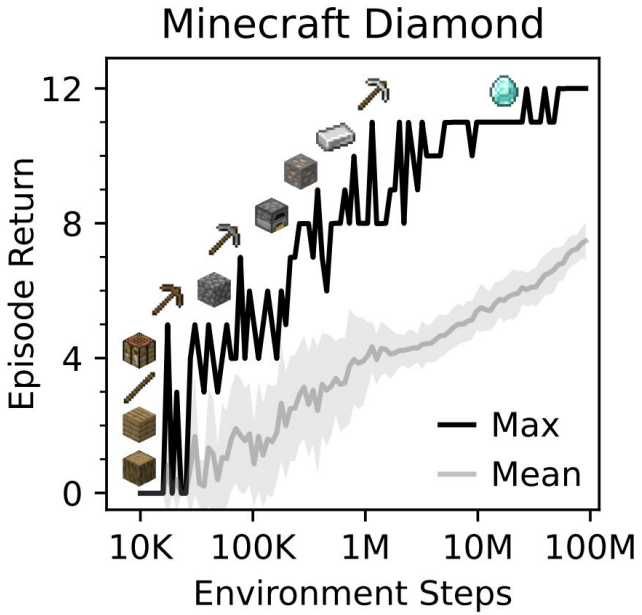


Figure from DreamerV3 (Hafner et al., 2022);

Learning from Demonstrations

Goal: Tackle incredibly hard tasks where designing a reward may be difficult

RL will work well if you provide an optimal reward function, but this is seldomly possible in incredibly complex environments like minecraft, robotics environments etc. and especially very long horizon problems (e.g. a sequence of tasks to arrange a dinner table)

A suitable replacement is to use **demonstrations**, past examples of how another person/robot/agent did a particular task (not necessarily optimally).

Learning from Demonstrations

Some typical approaches to leverage demonstrations

Behavior Cloning: Supervise learning the actions chosen for each observation in demo. Input is observation o_t, target/label is action a_t taken in the demonstration.

Offline RL: Generally encomposes methods that use no online interaction and use all kinds of information in a demonstration dataset (e.g. even modelling the reward to determine the optimality of demos and learn better)

Offline+Online RL: Learning from demonstrations while also using online interactions to improve the policy performance

Inverse RL: Learning a reward model from demonstrations to then generate rewards for an off-the-shelf RL algorithm like PPO or SAC

Learning from Demonstrations: Offline RL

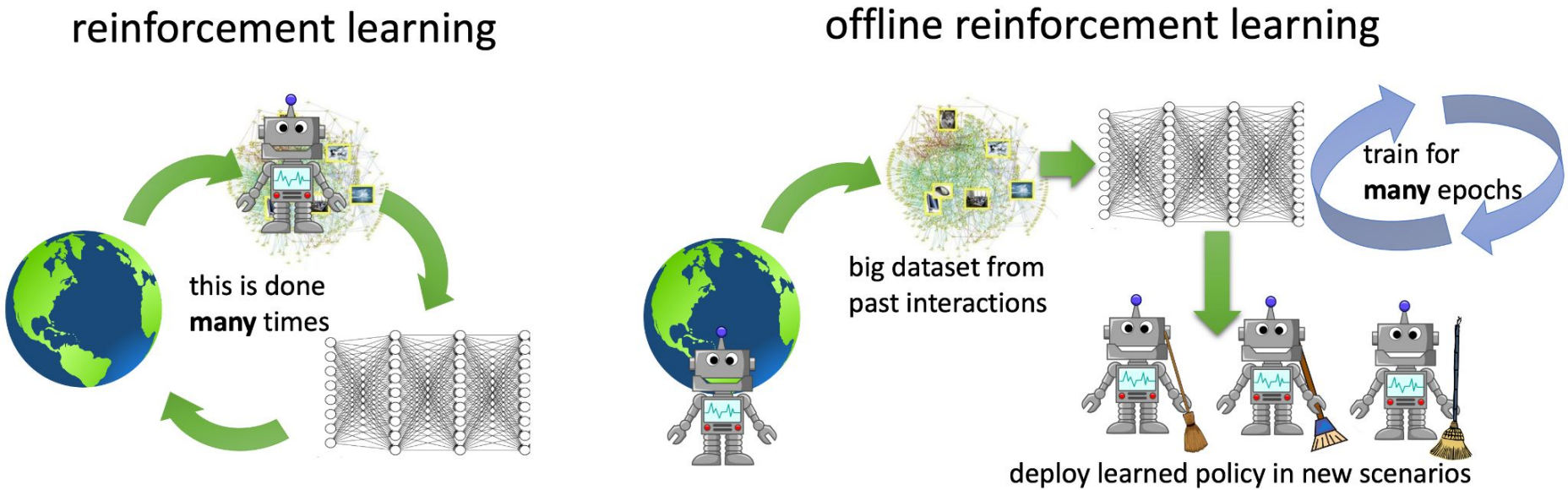


Image from: https://bair.berkeley.edu/blog/2020/12/07/offline/

Training/Inference Speed

Goal: Make the already slow RL training process go faster

There are a number of components to reinforcement learning training which differ from typical supervised learning and most machine learning methods. Each component contributes to the overall training time, but which components are actually horribly slow and need improving?

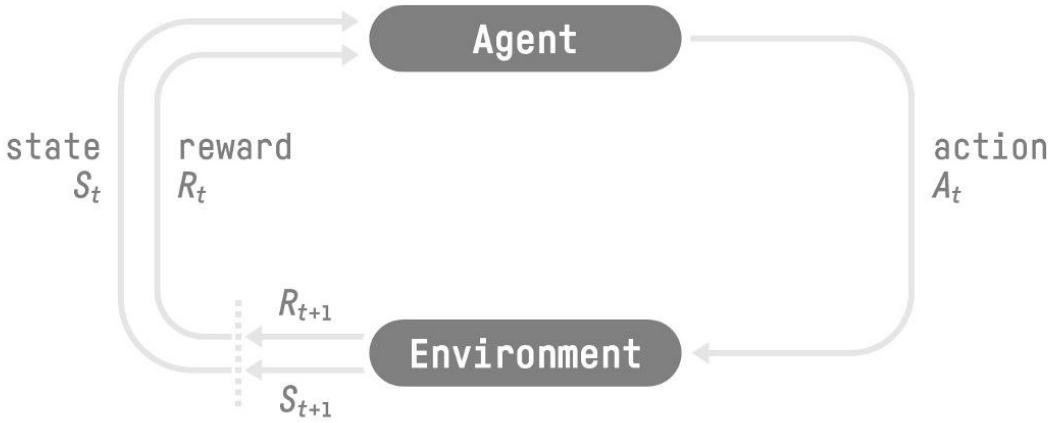
Training/Inference Speed

Where does the time go?

Training
- Learning from online experiences
- Learning from offline experiences
- Augmenting experiences
- Learning world models

Producing actions
- Forward pass of model
- Planning

Sampling the Environment
- Process action
- Generating observations



Training/Inference Speed

The big time eaters:

Training
- Learning from online experiences
- Learning from offline experiences
- Augmenting experiences
- Learning world models

Producing actions
- Forward pass of model
- Planning

Sampling the Environment
- Process action
- Generating observations

😠

Training/Inference Speed

Environment speed/FPS is usually the biggest bottleneck, especially for on-policy algorithms which are the most data hungry.

Speed is impacted by:

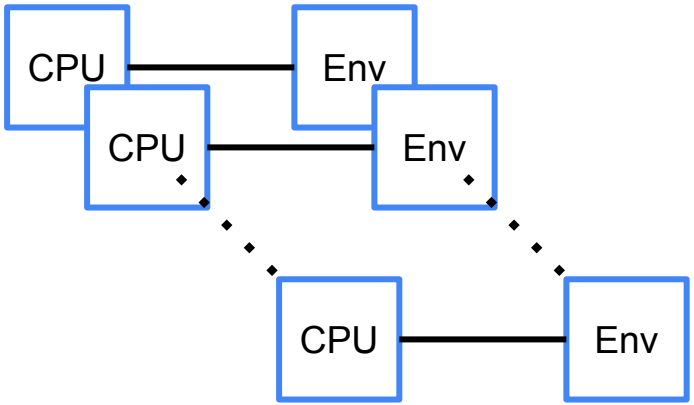
- environment dynamics
- sequential nature of environments
- observation generation, especially visual observations

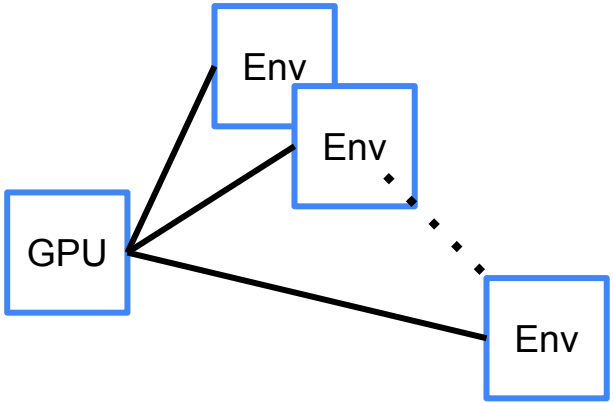
Training/Inference Speed

Solutions:

CPU based parallelization: Easiest to code for and scale. Simply parallel run an environment n times and get “n” times the FPS.

GPU based parallelization: Harder to code for usually but scales better -> cheaper.



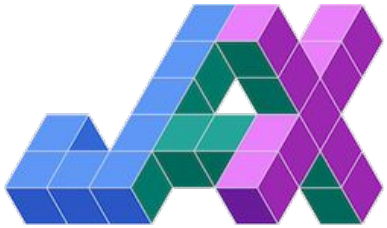


Training/Inference Speed

GPU Based Parallelization (The future?)
For CPUs number of environments scale with number of CPU threads and RAM. As a result, 1 consumer-level GPU is easily equivalent to >= 2000 CPUs! (for simpler environments)

What GPU parallelization is available?
Jax-based environments allow for python style writing of GPU/TPU accelerated code instead of figuring out the vectorization math yourself

NVIDIA Isaac-Sim provides GPU parallelizable simulations with a PyTorch based backend (returns data on torch tensors)

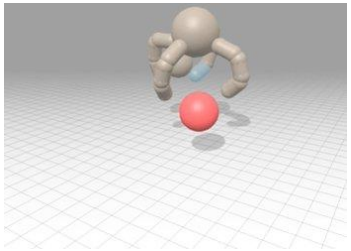


Training/Inference Speed

Some GPU parallelized environments out there now

Brax: Robotics simulation on Jax
Jumanji: Variety of games on Jax
Gymnax: Classical RL problems on Jax
Lux AI: Large-scale multi-agent games on Jax
IsaacGym / Orbit: Robotics simulation on IsaacSim

Jumanji



Brax

Lux AI

IsaacGym

Software Engineering

Goal: Make RL easy to use

A huge part of the deep learning revolution and the frequency of downstream applications being built stems from good software and libraries.

Research has massively been accelerated thanks to the prototyping encouraging behavior of tools like PyTorch and Jax

People can build all kinds of LLM powered applications easily thanks to so many tools like HuggingFace transformers, LangChain etc.

RL is no different but engineering for RL has been very difficult...

Software Engineering

RL Standardization Problem

RL work often is composed of a few libraries: An RL algorithm library, an environment library, and potentially libraries to scale the previous two parts to many machines.

Only 3 components really, but it's a **mess**

- Many researchers will use their own algorithms library (e.g. me), or one of the many popular ones out there (Tianshou, Stable Baselines, RL Games, jaxrl, Clean RL etc.).
- The environment interface is never consistent across research, there is the most popular one known as Gym/Gymnasium, but there is a different API for Brax (jax powered robotics), DeepMind environments, and more...
- The way environments are designed (different control frequencies, horizon sizes) meaning it's near impossible to reliably compare results on one env vs another.

Software Engineering

What then?

I'm not really sure to be honest. This can be in part due to the relative lack of maturity of modern deep RL, as well as how much more complex RL is compared to supervised learning etc.

Currently the Farama Foundation (who manage Gymnasium) are trying to consolidate this but this will definitely be a multi-year effort.

What is RL still missing?

What would be a breakthrough?

It can easily be said that language modelling has had a breakthrough thanks to transformers and recently large language models

Why hasn't RL kicked off?

Sample efficiency remains a huge problem, too data hungry and can't solve everything

Maturity needed for engineering to ensure end-users / laypeople can use it too