Reinforcement Learning

Reinforcement Learning - Agenda

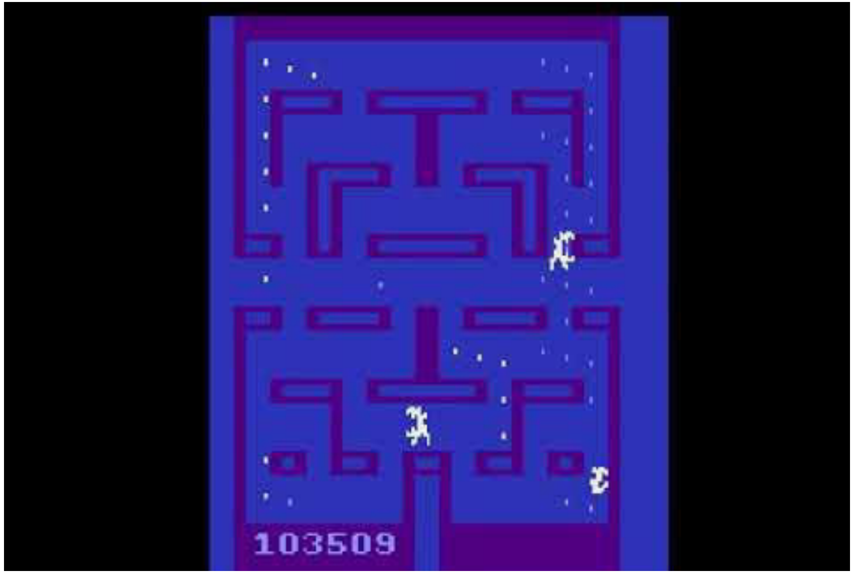
- What is Reinforcement Learning
- Learning to Optimize Rewards
- Policy Search
- Intro to OpenAI Gym
- Neural Network Policies
- The Credit Assignment Problem
- Policy Gradients
- Markov Decision Processes
- Temporal Difference Learning
- Q-Learning
- Deep Q-Learning
- Deep Q-Learning variants
- The TF-Agents Library
- Overview of Some Popular RL Algorithms

Reinforcement Learning

- Since 1950s, Reinforcement Learning produced many interesting applications like TD-Gammon

Reinforcement Learning

- In 2013, the **DeepMind** created a system which played any Atari game from scratch



Play the Video

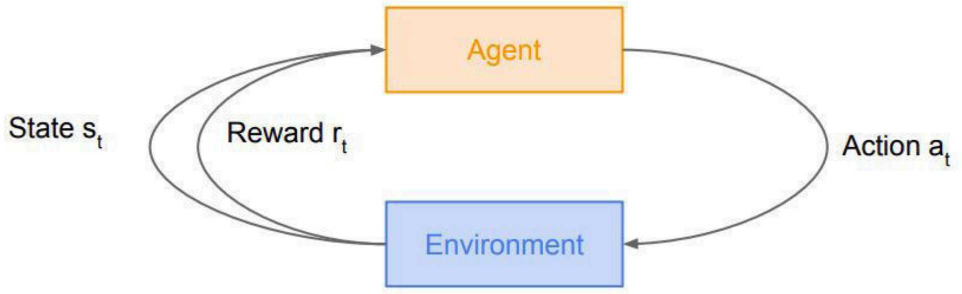
Reinforcement Learning

- In March 2016, AlphaGo defeated Lee Sedol in Go using Reinforcement Learning

Learning to Optimize Rewards

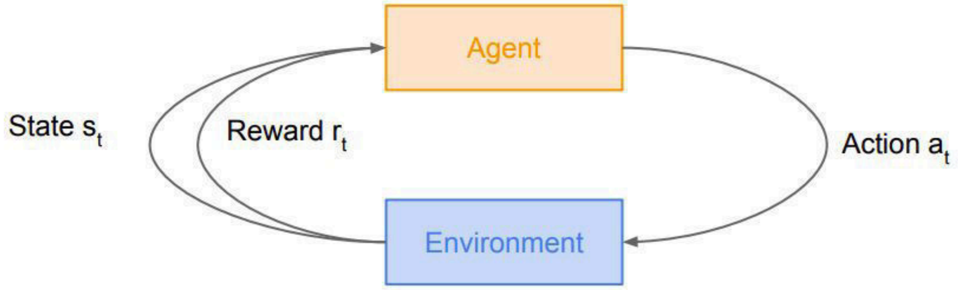
Learning to Optimize Rewards

- In Reinforcement Learning
  - A software **agent** makes **observations**
  - Takes **actions** within an **environment** and
  - Receives **rewards** in return



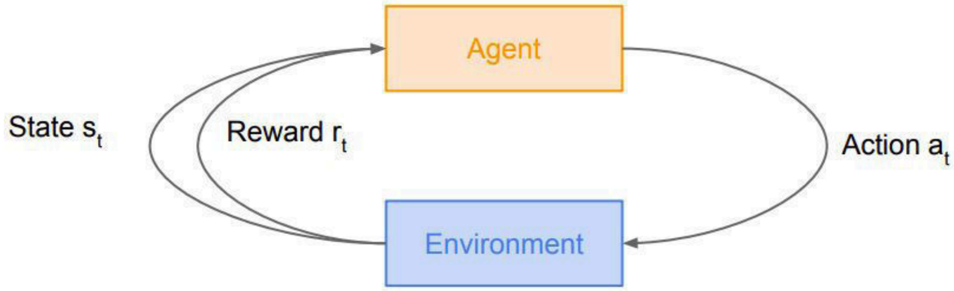
Learning to Optimize Rewards

Goal?



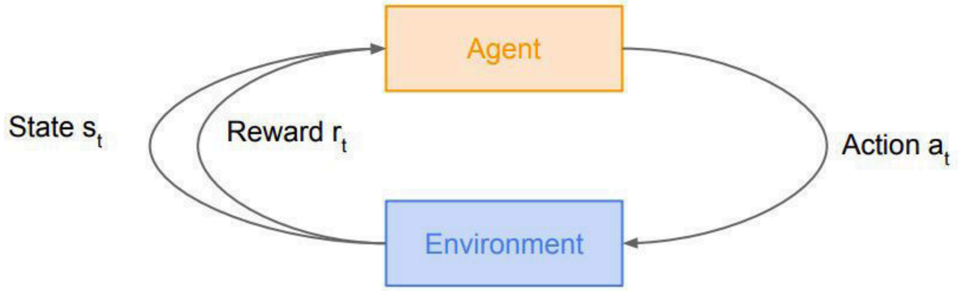
Goal

Learn how to take actions in order to **maximize reward**



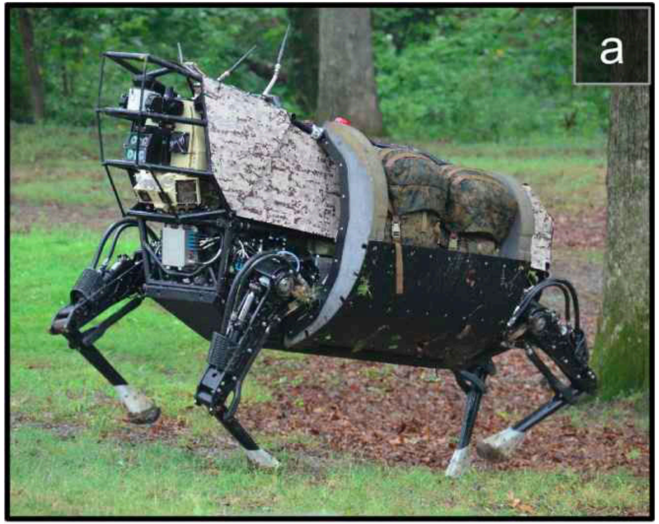
Learning to Optimize Rewards

- The agent acts in the environment and
  - Learns by trial and error to
  - Maximize its reward



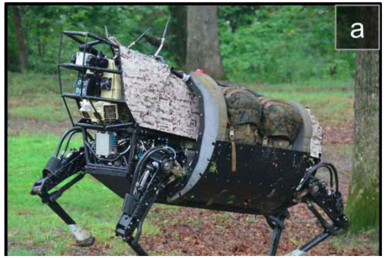
So how can we apply this in real-life applications?

Learning to Optimize Rewards – Walking Robot



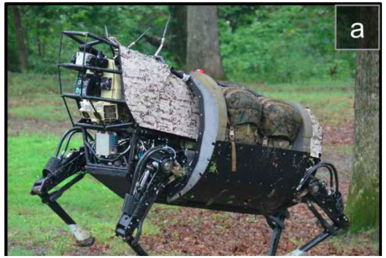
Learning to Optimize Rewards – Walking Robot

- Agent – Program controlling walking robot
- Environment – Real world
- Agent observes environment through set of sensors such as
    - Cameras and touch sensors
- Actions – Sending signals to activate motors



Learning to Optimize Rewards - Walking Robot

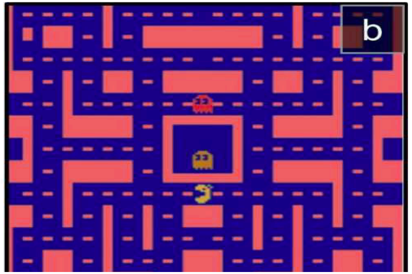
- It may be programmed to get
  - Positive rewards when approaches target destination
  - Negative rewards when
    - Wastes time
    - Goes in wrong direction
    - Falls down



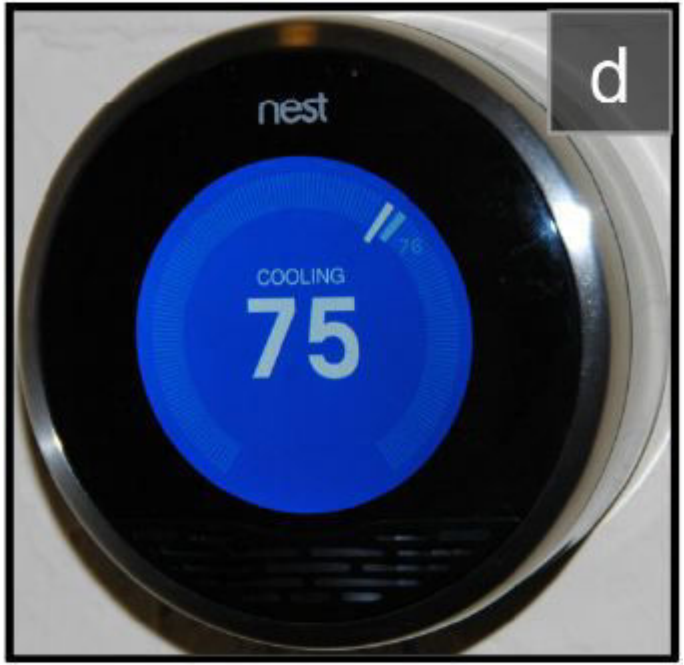
Learning to Optimize Rewards - Ms. Pac-Man

Learning to Optimize Rewards - Ms. Pac-Man

- Agent - Program controlling Ms. Pac-Man
- Environment - Simulation of Atari game
- Actions - Nine possible joystick positions
- Observations - Screenshots
- Rewards - Game points



Learning to Optimize Rewards – Thermostat



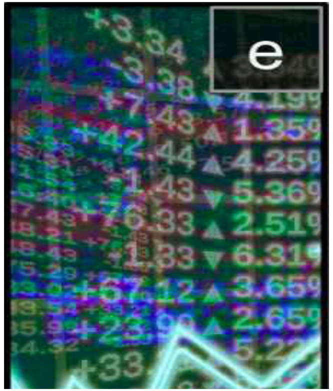
Learning to Optimize Rewards – Thermostat

- Agent – Thermostat
- Get
  - Positive rewards when close to target temperature
  - Negative rewards when temperature needs to be tweaked
- Important – Agent must learn to anticipate human needs

Learning to Optimize Rewards – Auto Trader

Learning to Optimize Rewards – Auto Trader

- Agent – Observes stock market prices and decide how much to buy or sell
- Rewards – The monetary gains and losses

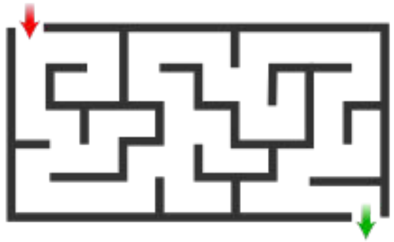


Learning to Optimize Rewards

- Other examples includes
    - Self-driving cars
    - Placing ads on web page
    - Controlling focus of image classification system

Learning to Optimize Rewards

- Rewards are not always positive
- For example
    - Agent moves around in a maze
    - Gets negative reward at every time step
    - So it needs to find the exit quickly



Policy Search

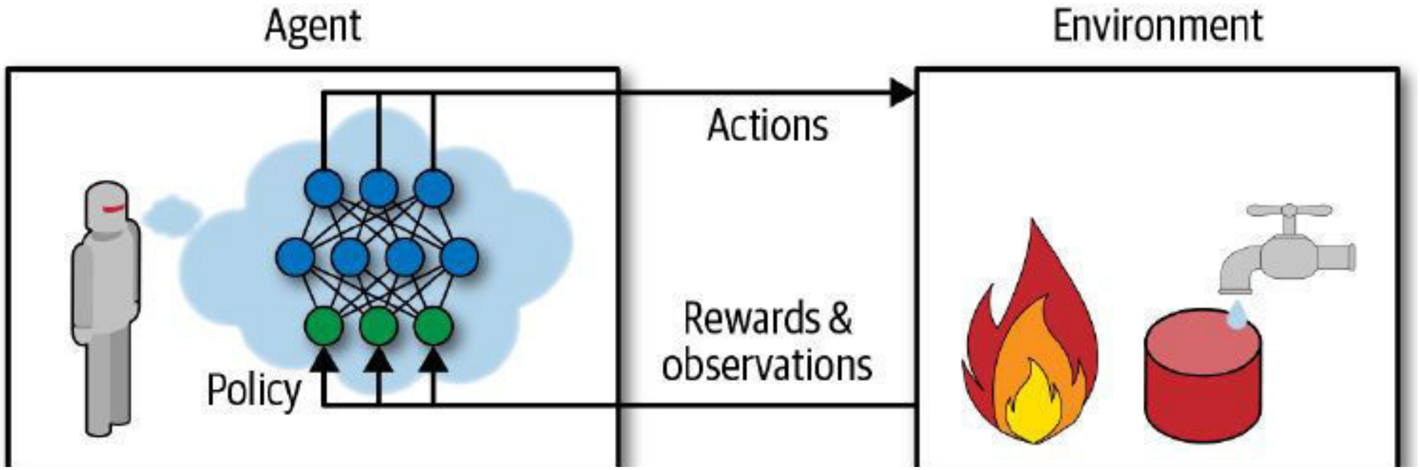
Policy Search

What is a policy?

The algorithm used by software agent to determine its actions.

Policy Search – Example

- A neural network
  - Taking observations as inputs and
  - Outputting the action to take



Policy Search

- It can be any algorithm, not necessarily deterministic
- For example, robotic vacuum cleaner with
  - Amount of dust it picks in 30 minutes as reward

Policy Search

- Its policy could be to
  - Move forward with probability p every second or
  - Randomly rotate left/right with probability 1 – p
  - Rotation angle = Random angle between –r and +r

- Since this policy involves some randomness
  - It is called a **stochastic policy**

Policy Search

- The robot will have erratic trajectory, which guarantees
  - It can get to any place it can reach and
  - Pick up all the dust

Policy Search

The question is:

How much dust will it pick up in 30 minutes?

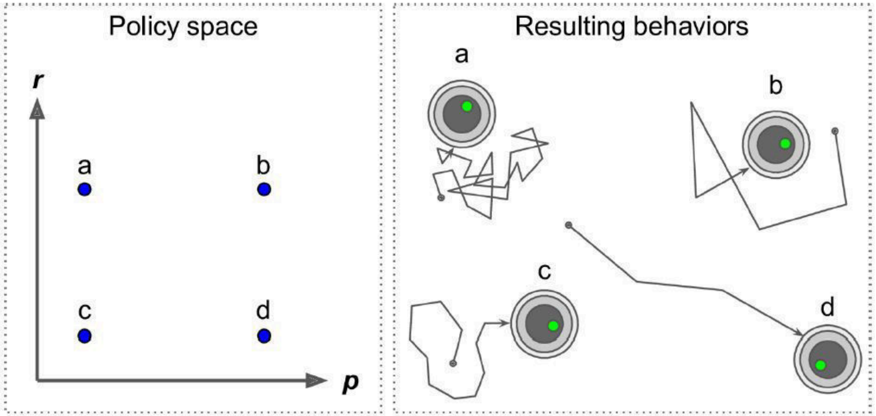
How would we train such a robot?

Policy Search

- Using just two policy parameters
  - Probability **p**
  - Angle range **r**

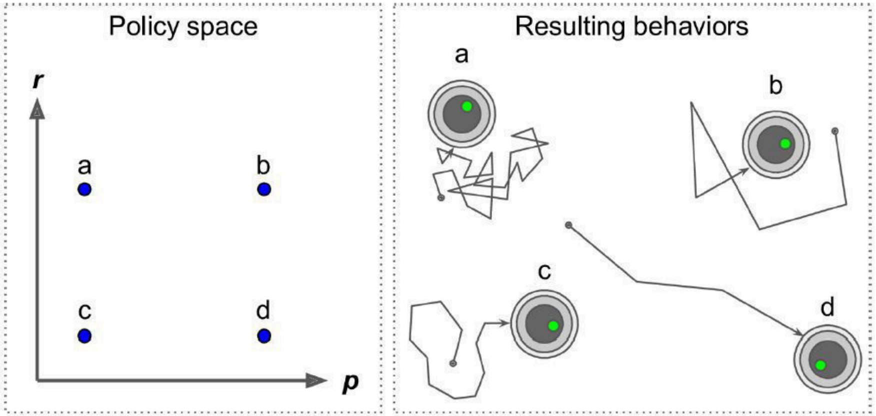
Policy Search – Brute Force Approach

- With brute force approach, the algorithm
  - Tries different values for these parameters, and
  - Picks best performing combination



Policy Search – Brute Force Approach

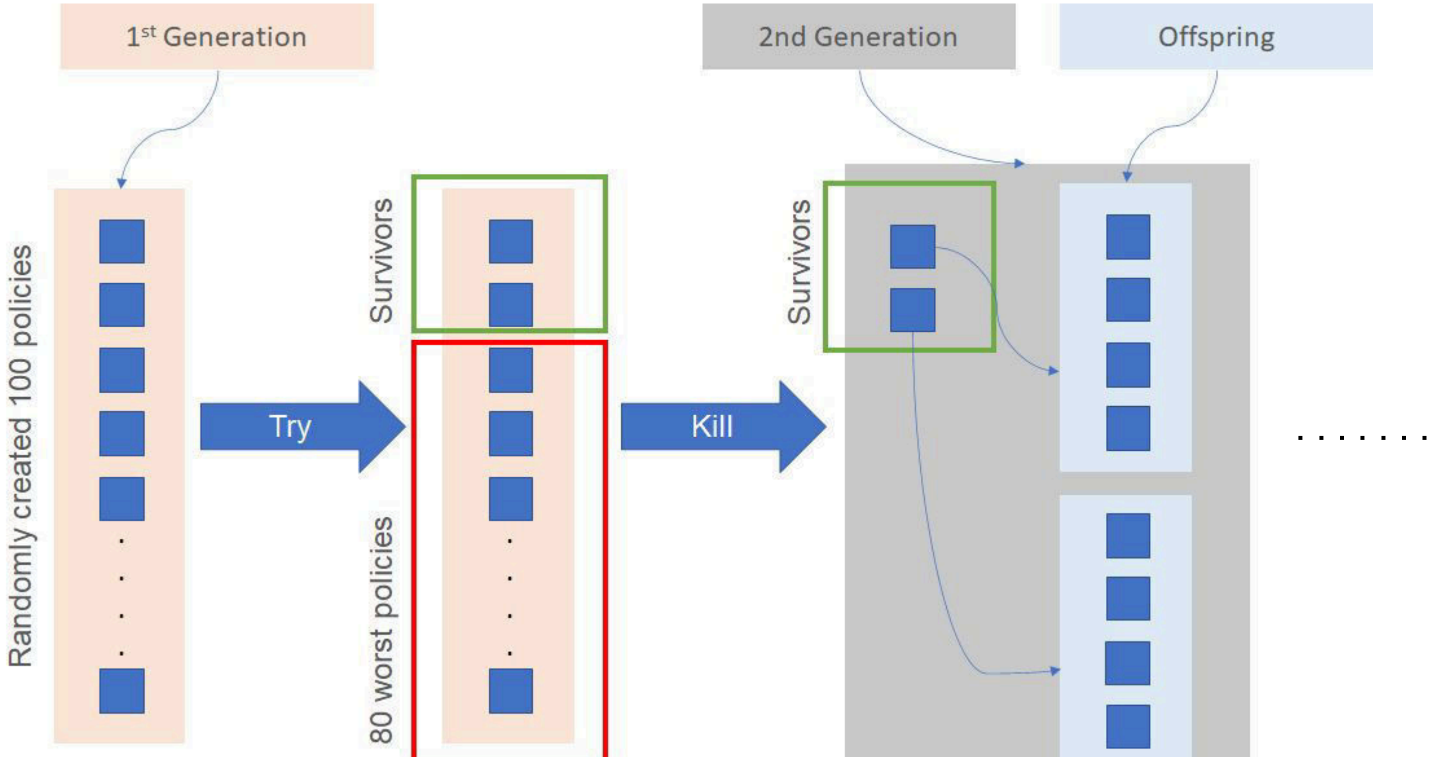
- When policy space is too large then this approach of
  - Finding good set of parameters
  - Is like searching a needle in haystack



Policy Search – Genetic Algorithms

Other method is using **genetic algorithms**.

Policy Search – Genetic Algorithms



Policy Search – Optimization Techniques

Another approach is using **Optimization Techniques**

Policy Search - Optimization Techniques

- Evaluate gradients rewards w.r.t. policy parameters
- Tweak parameters by following gradient towards
  - Higher rewards (gradient ascent)

This approach is called **policy gradients**

Policy Search - Optimization Techniques - Example

- With vacuum cleaner robot, we can
  - Slightly increase p and evaluate
    - Amount of dust collected
  - If it increases, then increase p some more, or else
  - Reduce p

Introduction to OpenAI Gym

Introduction to OpenAI Gym

- To train agent in Reinforcement Learning we need working environment
- For example
    - Agent - play Atari games
    - Environment - Atari game simulator

Introduction to OpenAI Gym

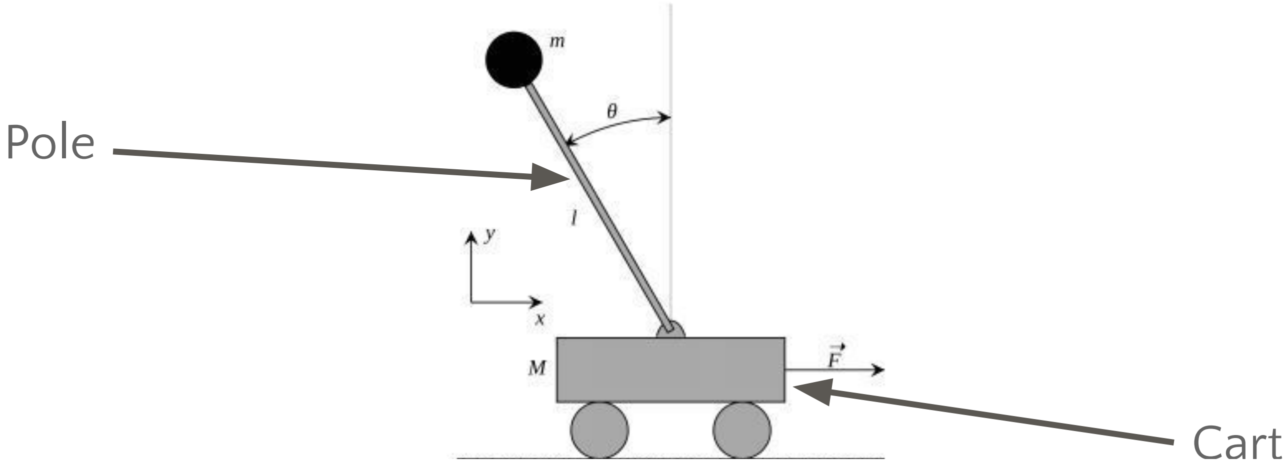
OpenAI gym - Toolkit that provides wide variety of simulations like

- Atari games
- Board games
- 2D and 3D physical simulations and so on

Let's do a hands-on

Introduction to OpenAI Gym

Goal – Balance a pole on top of a movable cart. Pole should be upright



Switch to Notebook

reinforcement_learning/reinforcement_learning.ipynb

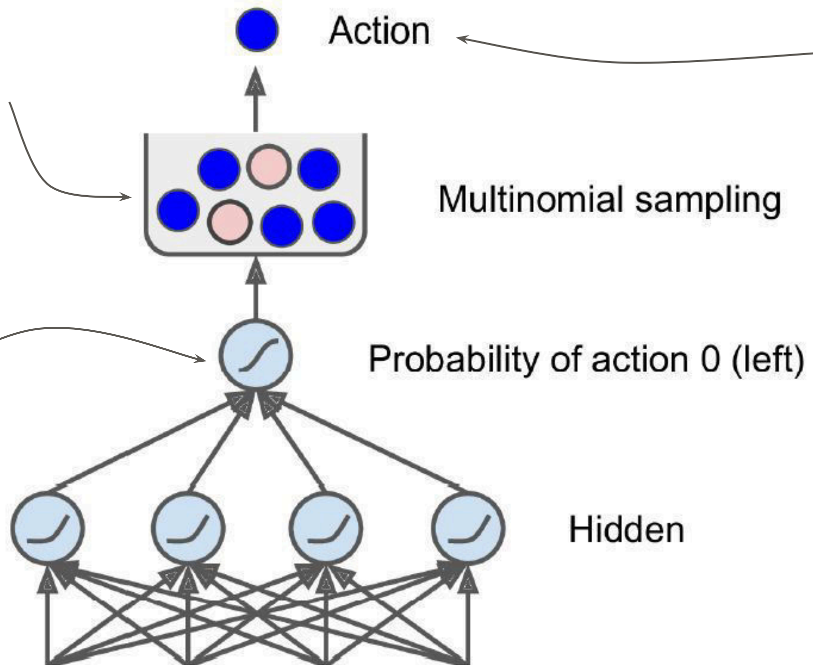
Neural Network Policies

Neural Network Policies

Let's create a neural network policy.

Select an action randomly according to the estimated probabilities

Estimate probability for each action



Output

Input

Neural Network Policies

Example, in cart pole:

- If it outputs 0.7, then
- we will pick action 0 with 70% probability, and
- action 1 with 30% probability.

Neural Network Policies

Q: Why we are picking a random action based on the probability given by the neural network, rather than just picking the action with the highest score.

Neural Network Policies

Q: Why we are picking a random action based on the probability given by the neural network, rather than just picking the action with the highest score.

A: This approach lets the agent find the **right balance between exploring new actions** and **exploiting the actions that are known to work** well.

Neural Network Policies

We will never discover a new dish at restaurant if

we don't try anything new.

Give serendipity a chance.

Switch to Notebook

`reinforcement_learning/reinforcement_learning.ipynb`

The Credit Assignment Problem

Evaluating Actions: The Credit Assignment Problem

If we knew the best action at each step,

- We could train neural network as usual,
- by minimizing cross entropy
- between estimated probability and target probability.

It would just be regular supervised learning.

Evaluating Actions: The Credit Assignment Problem

However, in Reinforcement Learning

- the only guidance the agent gets is through rewards,
- and rewards are typically **sparse and delayed.**

Evaluating Actions: The Credit Assignment Problem

For example,

If agent manages balancing pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad?

All it knows is the pole fell after last action, but surely this last action is not entirely responsible.

Evaluating Actions: The Credit Assignment Problem

This is called the **credit assignment problem**

- When agent gets reward, it’s hard for it to know which actions gets credited (or blamed) for it.

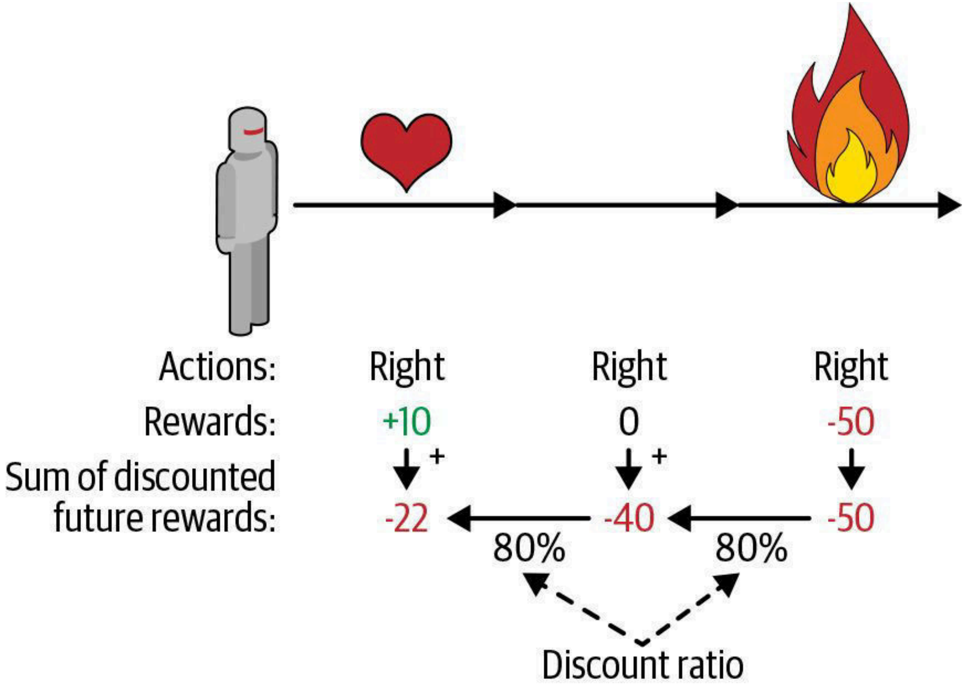
- Think of a dog that gets rewarded hours after it behaved well; will it understand what it is rewarded for?

Evaluating Actions: The Credit Assignment Problem

To tackle this problem,

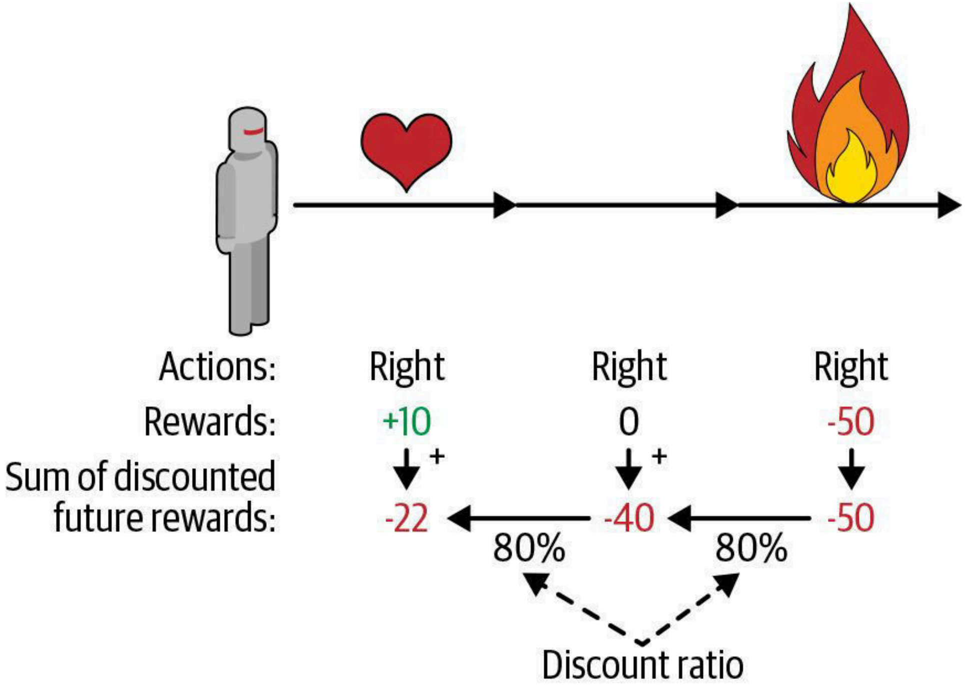
- Common strategy is evaluate action based on sum of all rewards
- Applying a discount discount factor γ (gamma) at each step
- This sum of discounted rewards is called the **action's return**

Evaluating Actions: The Credit Assignment Problem



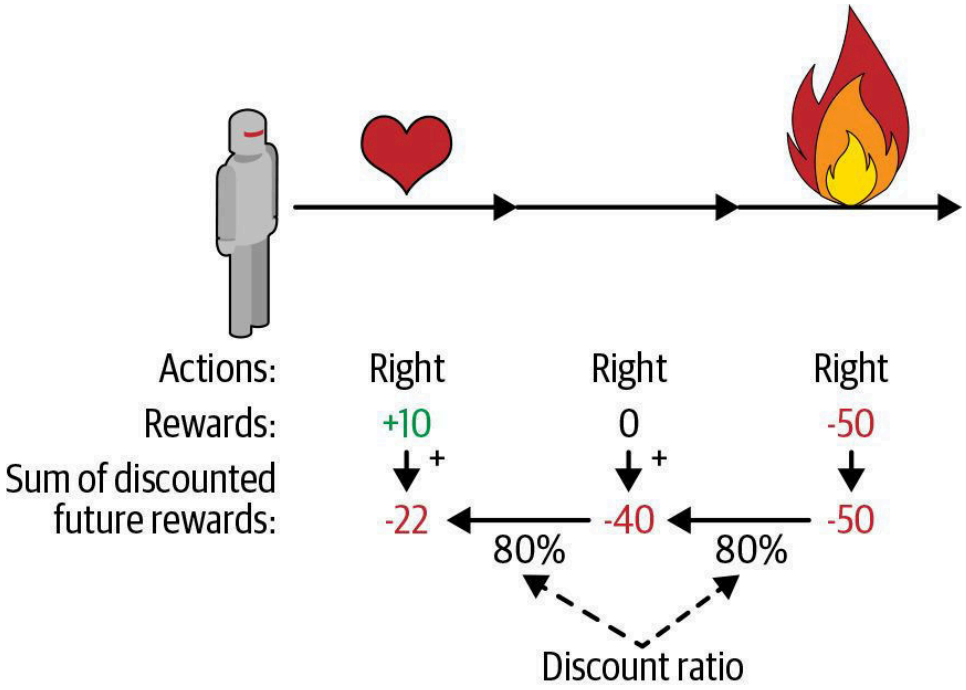
- Agent goes right 3 times
- If discount factor γ = 0.8
- Return of 10 + γ × 0 + γ2 × (−50) = −22

Evaluating Actions: The Credit Assignment Problem



- If discount factor close to 0
- Then future rewards won't count for much compared to immediate rewards

Evaluating Actions: The Credit Assignment Problem



- If discount factor close to 1
- Then rewards far into the future will count almost as much as immediate rewards
- Typical discount rates are 0.9 or 0.99

Evaluating Actions: The Credit Assignment Problem

- Good action may be followed by several bad actions
- This cause pole to fall quickly
- Resulting in good action getting low return
- But if we play enough times
- On average good actions will get higher return than bad ones

Evaluating Actions: The Credit Assignment Problem

- We want to estimate
  - How much better or worse an action is
  - Compared to other possible actions, on average
  - This is called the **action advantage**

Policy Gradients

Policy Gradients

PG algorithms

- Optimize parameters of a policy
    - following gradients toward higher rewards.

Policy Gradients

- One popular class of PG algorithms is REINFORCE algorithms:
  - Introduced in 1992 by Ronald Williams.
  - Here is one common variant:

Policy Gradients

Step 1:

Neural network policy play game several times

a. Compute gradients that makes chosen action more likely,
b. but don’t apply these gradients yet.

Policy Gradients

Step 2:

Once you have run several episodes, compute each action's score (using the method described earlier).

Policy Gradients

Step 3:

- **Action's score is positive**, action was good, **apply gradients** computed earlier
- **Score is negative**, action was bad, **apply the opposite gradients**
- Multiply each gradient vector by the corresponding action's score.

Policy Gradients

Step 4:

- Finally, compute mean of all resulting gradient vectors, and
- Use it to perform Gradient Descent step.

Switch to Notebook

reinforcement_learning/reinforcement_learning.ipynb

Markov Decision Processes

Markov Decision Processes

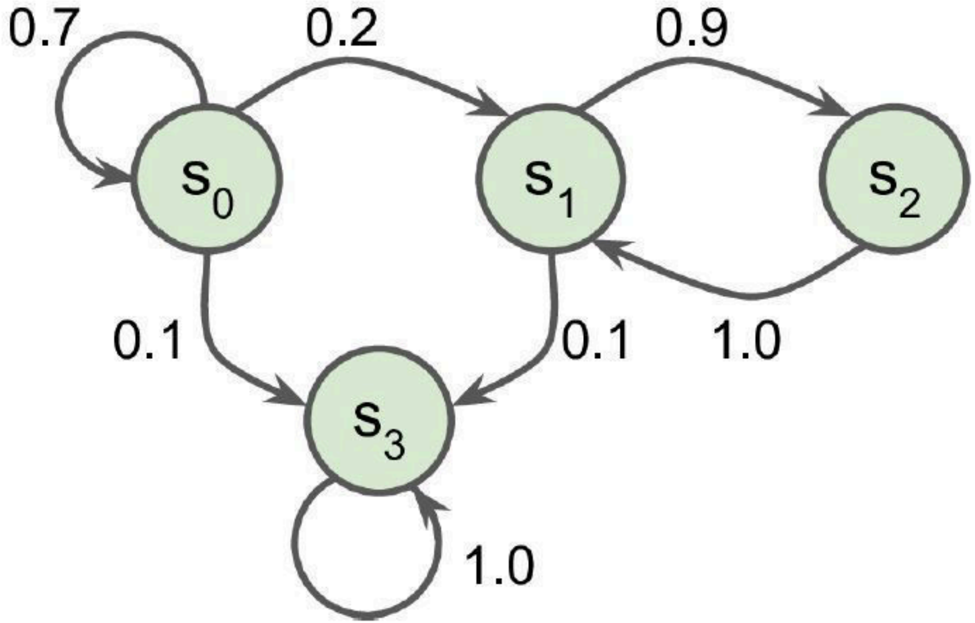
- Markov Decision Processes were inspired by Markov Chains
- These are stochastic processes with no memory

Markov Decision Processes

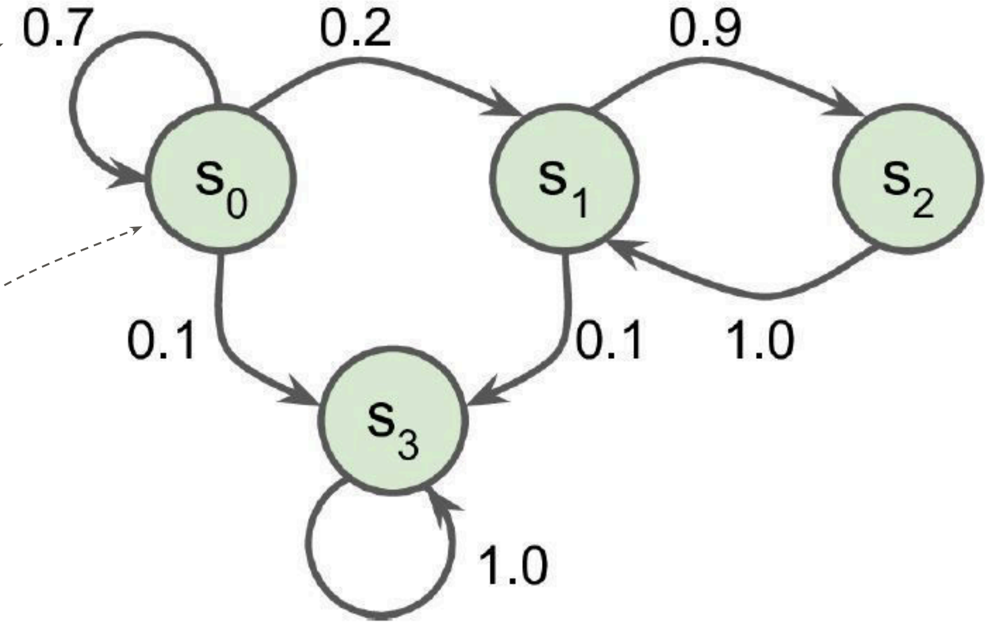
- This process has fixed number of states
- It randomly evolves from one state s to s′
- The probability this is fixed,
  - It depends only on (s, s′)
  - And not on past states

Markov Decision Processes

Example of Markov Chain



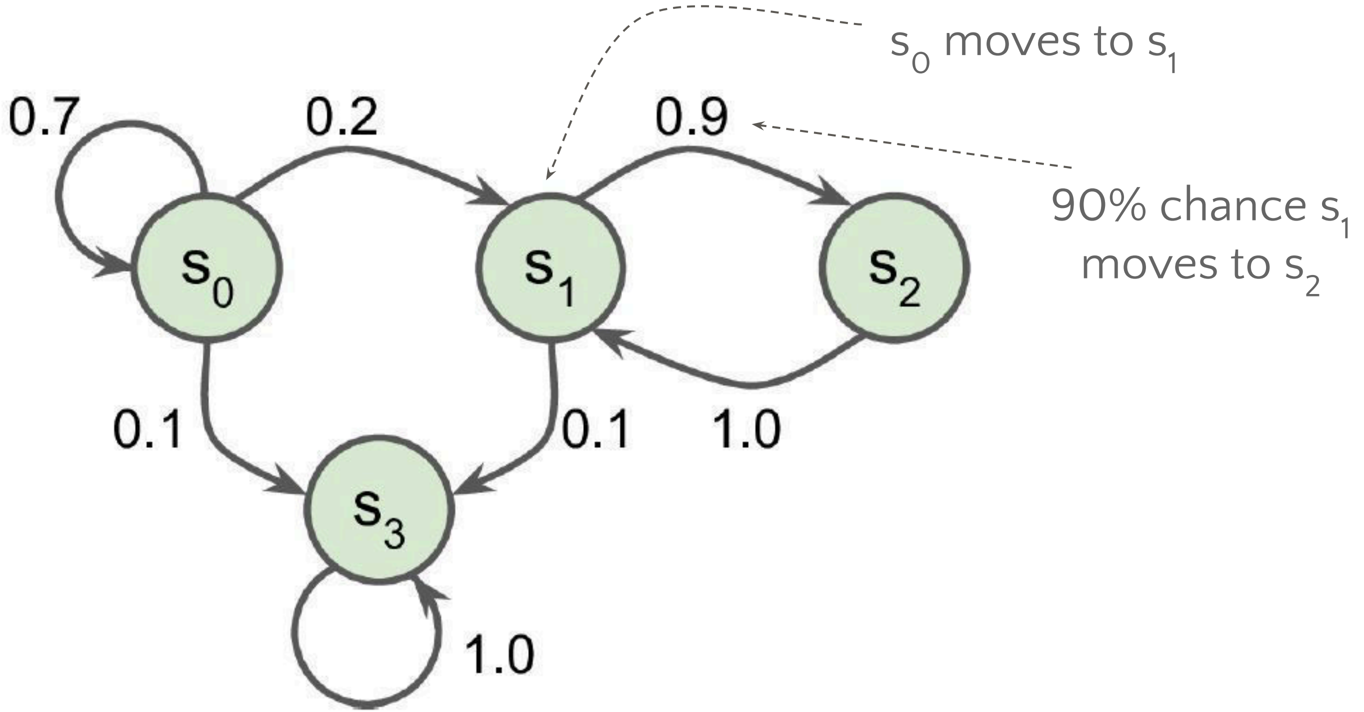
Markov Decision Processes



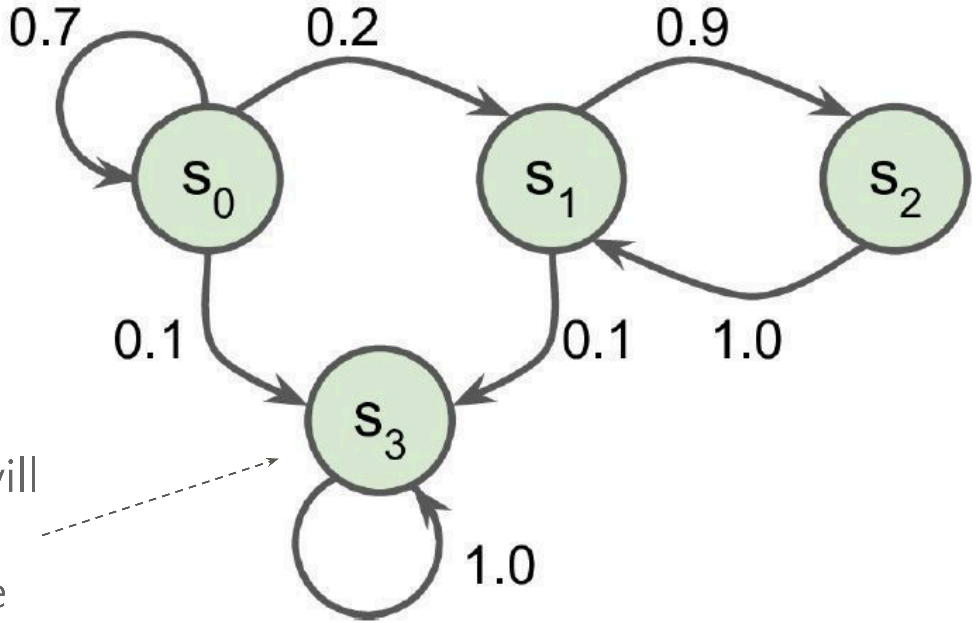
70% chance of staying there

Process Starts

Markov Decision Processes



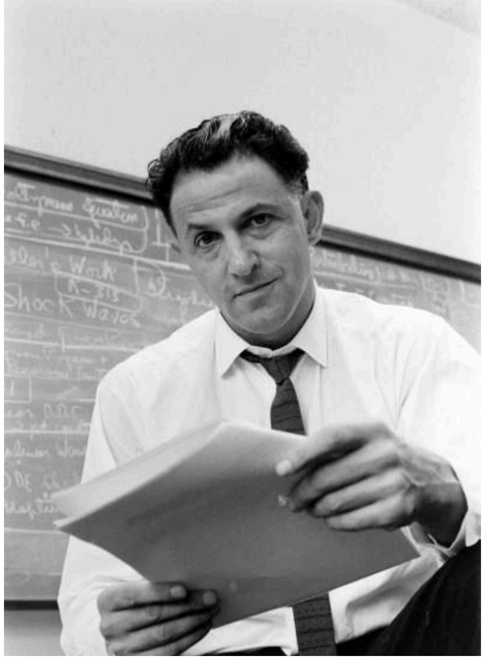
Markov Decision Processes



Eventually it will move to s_3, terminal state

Markov Decision Processes

- Markov decision processes were described in 1950s by Richard Bellman



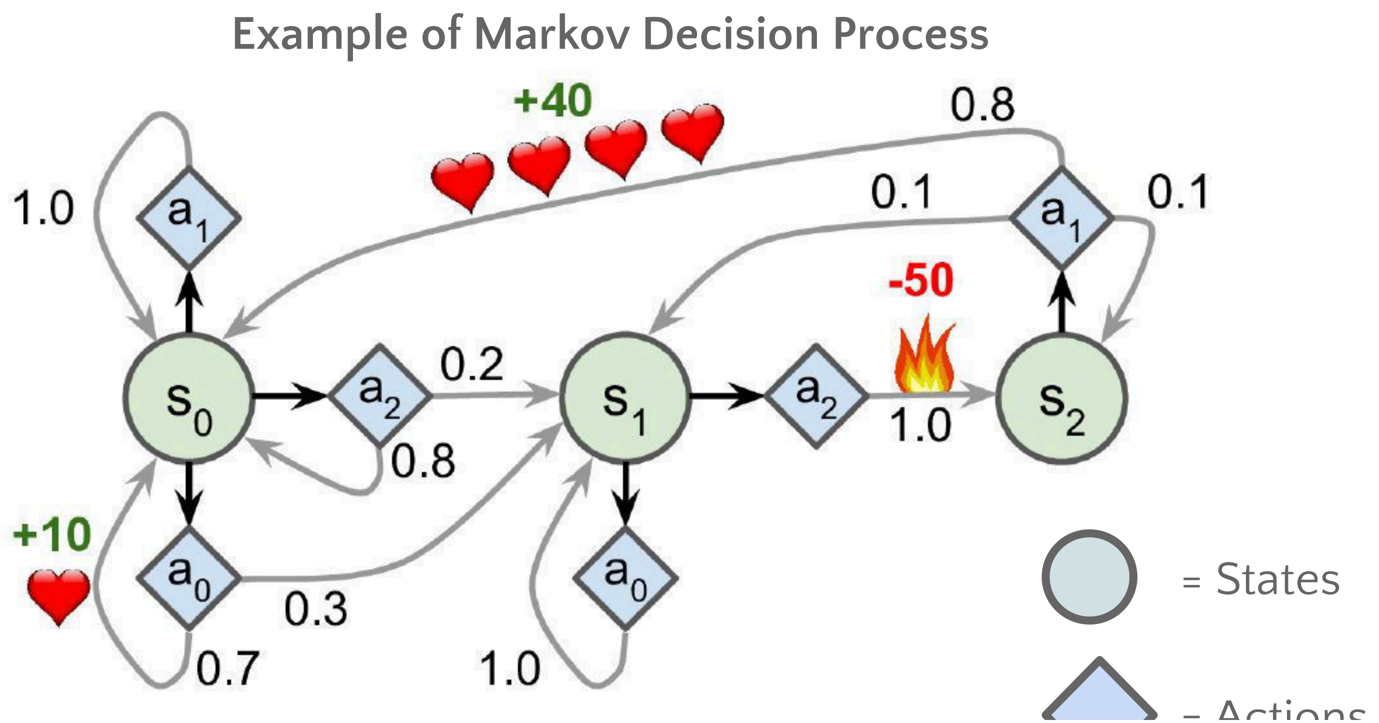
Markov Decision Processes

MDP resemble Markov chains with a twist

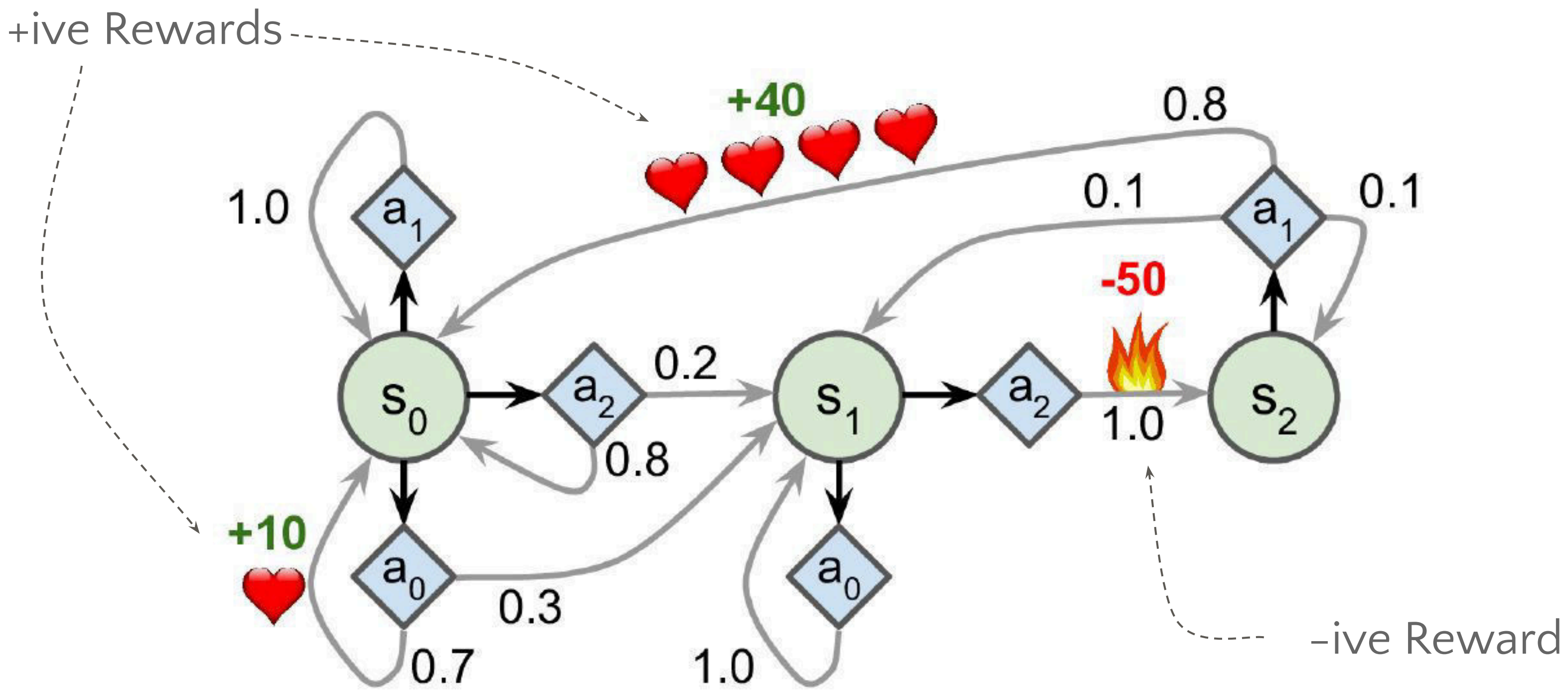
Markov Decision Processes

- At each step, agent choose one of several actions
- Transition probabilities depend on chosen action
- Some state transitions return some reward (positive or negative),
- Agent's goal is to find policy with maximum reward

Markov Decision Processes



Markov Decision Processes



Markov Decision Processes

- In state s_0 action a_0 is the best option
- In state s_2 only choice is to take action a1
- But in s_1 where should agent go?
    - a_0 or go through fire in a_2

Bellman found way to estimate optimal state value of any state s, V*(s).

Markov Decision Processes

Bellman Optimality Equation

- This recursive equation says if the agent acts optimally, then
  - Optimal value of current state = Reward on average after one optimal action

    + Expected optimal value of all possible next states for this action

Markov Decision Processes

Transition probability from state s to state s′ for action a

Discount factor

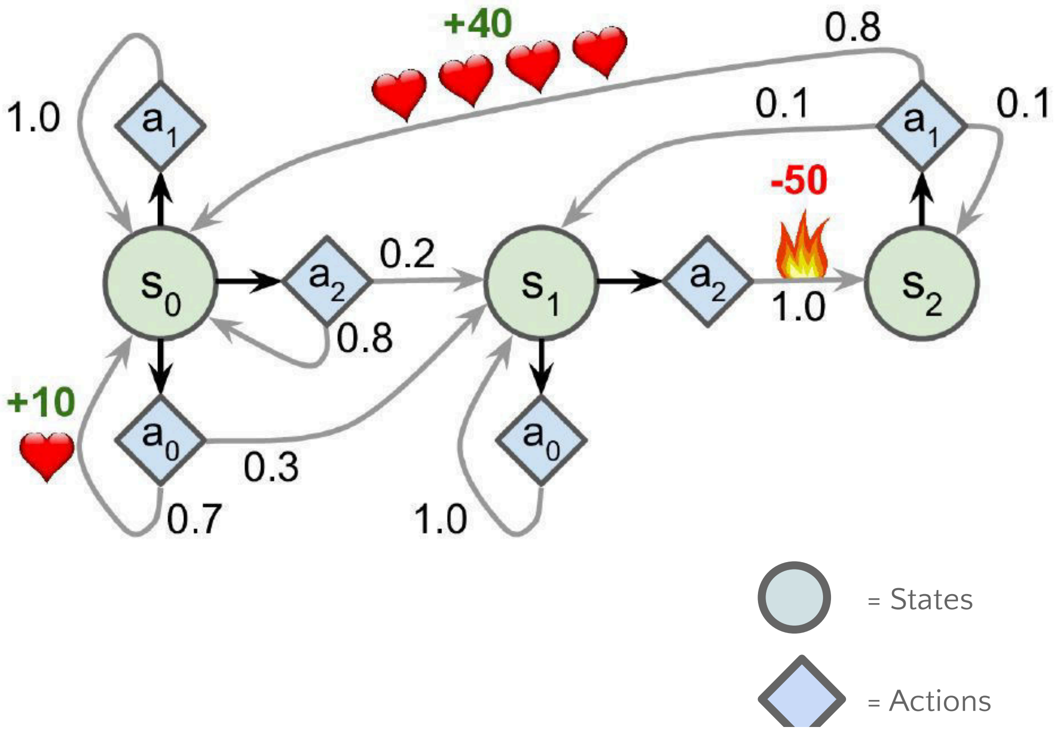
$$V^{\star}(s) = \max_a \Sigma_s T(s, a, s')[R(s, a, s') + \gamma \cdot V^{\star}(s')] \quad \text{for all } s$$

Value of state s state.

The value of s' state.

Reward for transition from state s to state s′ for action a

Markov Decision Processes

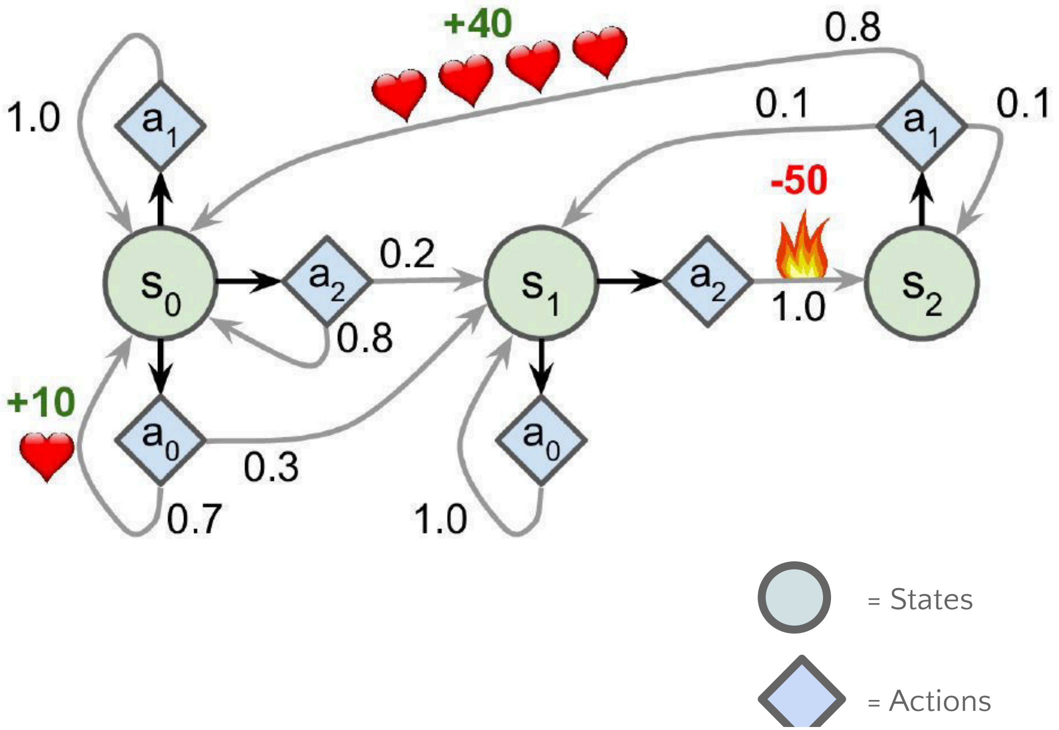


Here, transition probability from state s_2 to s_0, given that the agent chose action a_1, is 0.8

Or

$T(s_2, a_1, s_0) = 0.8$

Markov Decision Processes



Also, the reward that the agent gets when it goes from state s_2 to s_0, given that the agent chose action a_1, is +40

Or

$R(s_2, a_1, s_0) = +40$

Value Iteration Algorithm

Value Iteration Algorithm

This equation leads to **Value Iteration algorithm**

- This precisely estimates optimal state value of every possible state
  - Initialize all the state value estimates to zero
  - And then iteratively update them using the Value Iteration algorithm

Value Iteration Algorithm

Value Iteration algorithm

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \cdot V_k(s')\right] \quad \text{for all } s$$

V_k = Estimated value of state s at the k*th* iteration

Value Iteration Algorithm

- Knowing optimal state values is useful to evaluate a policy,
- But it does not give us optimal policy for the agent

Q-Value

Q-Value

- So Bellman found Q-Value Iteration algorithm
- It estimates optimal state-action values, called QValues (Quality Values)

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma \cdot \max_{a'} Q_k(s', a')\right] \quad \text{for all } (s'a)$$

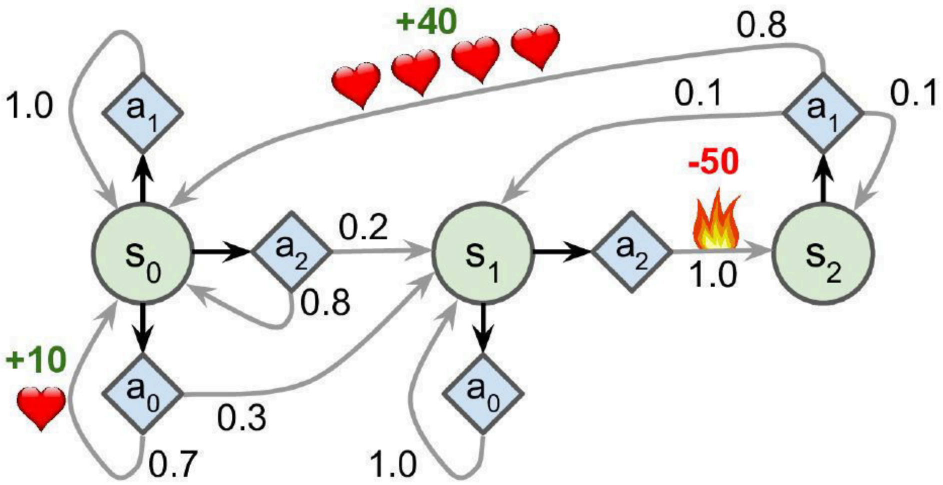
Q-Value

- Using this equation, agent finds optimal Q-Value to choose best action with highest Q-Value:

$$\pi^{\star}(s) = \underset{a}{\operatorname{argmax}}\ Q^{\star}(s, a)$$

Q-Value

Let's apply this algorithm to the MDP represented below:



Switch to Notebook

reinforcement_learning/reinforcement_learning.ipynb

Temporal Difference Learning

Temporal Difference Learning

- Problem with Markov Decision Process is that
- Agent does not know
  - What transition probabilities are
  - Or what rewards are going to be

Temporal Difference Learning

- Temporal Difference Learning (TD Learning) algorithm is similar to Value Iteration algorithm
- But it takes into account the fact that the agent has partial knowledge of the MDP – Markov Decision process

Temporal Difference Learning

- With Temporal Difference Learning (TD Learning)
- Agent uses an exploration policy to explore MDP
- And as it progresses
    - TD Learning algorithm updates estimates of state values
    - based on observed transitions and rewards

Temporal Difference Learning

Representation

$$a \underset{\alpha}{\leftarrow} b$$

Stands for

$$a_{k+1} \leftarrow (1-\alpha) \cdot a_k + \alpha \cdot b_k$$

Temporal Difference Learning

TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha\,(r + \gamma \cdot V_k(s'))$$

Or

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$
$$\text{with } \delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k(s)$$

Temporal Difference Learning

TD Learning algorithm

Learning rate (e.g., 0.01)

$$V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$$
$$\text{with } \boxed{\delta_k(s, r, s')} = \boxed{r + \gamma \cdot V_k(s')} - V_k(s)$$

R = rewards
γ = discount rate

TD Error

TD Target

Temporal Difference Learning

Similarities between TD Learning and SGD:

- Handles one sample at a time
- Only truly converge if you gradually reduce the learning rate
  - Else it keeps bouncing around optimum Q-Values

Q-Learning

Q-Learning

- Q-Learning algorithm is adaptation of Q-Value Iteration algorithm
- It works by watching an agent play (e.g., randomly)
- Gradually improving estimates of Q-Values
- Once it has accurate Q-Value estimates
- Then optimal policy is choosing action that has highest Q-Value

Q-Learning

Q-Learning algorithm

$$Q\left(s, a\right) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} \; Q\left(s', a'\right)$$

Q-Learning

$$Q(s, a) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} Q(s', a')$$

- For each state-action pair (s, a)
- This algorithm keeps track of running average of rewards r the agent gets upon leaving state s with action a
- Plus sum of discounted future rewards it expects to get

Q-Learning

$$Q(s,a) \xleftarrow[\alpha]{} r + \gamma \cdot \max_{a'} Q(s',a')$$

- To estimate this sum
- We take the maximum of the Q-Value estimates for next state s’

Switch to Notebook

reinforcement_learning/reinforcement_learning.ipynb

Q-Learning

- Q-Learning algorithm is **off-policy** algorithm because
  - Policy being trained is not necessarily the one being executed
- Policy Gradients algorithm is **on-policy** algorithm
  - It explores the world using policy being trained

Q-Learning

Can we do better?

Exploration Policies

Exploration Policies

- Q-Learning can work only if exploration policy explores MDP thoroughly
- Better option is to use ε-greedy policy (ε is epsilon)
    - At each step it acts randomly with probability ε
    - Or greedily with probability 1–ε

Exploration Policies

- Advantage of ε-greedy policy
- Spends more and more time exploring interesting parts of environment
  - The Q-Value estimates get better and better
  - While still spending some time visiting unknown regions of the MDP

Exploration Policies

Q-Learning using an exploration function

$$Q\left(s, a\right) \underset{\alpha}{\leftarrow} r + \gamma \cdot \max_{a'} \; f(Q(s', a'), N(s', a'))$$

Counts the number of times the action a′ was chosen in state s′.

Exploration function, such as
f(Q, N) = Q + κ/(1 + N)
κ is curiosity hyperparameter that measures how much gent is attracted to unknown

Approximate Q-Learning

Approximate Q-Learning

- Problem with Q-Learning is:
    - It does not scale well to large or medium MDPs
    - With many states and actions

Approximate Q-Learning

- For example Ms. Pac-Man
  - 150 pellets (present or already eaten) Ms. Pac-Man can eat
  - Number of possible states > $2^{150} \approx 10^{45}$

Approximate Q-Learning

- If we include all positions of ghosts and Ms. Pac-Man
    - Number of possible states > number of atoms in our planet !!!
    - This is impossible to keep track of

Approximate Q-Learning

- Solution is find function $Q_\theta(s, a)$ that approximates Q-Value of any state-action pair (s, a)
- Using manageable number of parameters (given by parameter vector θ)

This is **Approximate Q-Learning**

Approximate Q-Learning

- DeepMind showed that DNNs work better to estimate Q-Values
- Especially for complex problems
- Also, it does not require feature engineering

Deep Q-Learning

Deep Q-Learning

- DNN that estimates Q-Values = **Deep Q-Network** (DQN)
- DQN for Approximate Q-Learning = **Deep Q-Learning**

Deep Q-Learning

How does DQN works?

Deep Q-Learning

- Use DQN to compute Q-Value for state-action pair (s, a)

- Execute DQN on next state s′ and for all possible actions a′ to estimate sum of future discounted rewards

- Pick the highest and discount it

$$Q_{\text{target}}\left(s, a\right) = r + \gamma \cdot \max_{a'} \; Q_{\theta}\left(s', a'\right)$$

Deep Q-Learning

- Sum reward r and future discounted value estimate

- We get target Q-Value y(s, a) for the state-action pair (s, a)

$$Q_{\text{target}}\left(s, a\right) = r + \gamma \cdot \max_{a'} \; Q_{\theta}\left(s', a'\right)$$

Deep Q-Learning

- With this target Q-Value
    - We run training step using Gradient Descent
    - To minimize squared error between estimated Q-Value Q(s, a), and
    - Target Q-Value (or Huber loss to reduce algorithm's sensitivity to large errors)

Deep Q-Learning

Let's see an implementation of Deep Q-Learning to solve the Cart-Pole Environment

Switch to Notebook

`reinforcement_learning/reinforcement_learning.ipynb`

Deep Q-Learning Variants

Fixed Q-Value Targets

Fixed Q-Value Targets

- Basic Deep Q-Learning is used to make predictions and set its own targets
- This feedback loop can make the network unstable
- It can diverge, oscillate, freeze, and so on

Fixed Q-Value Targets

- To solve this problem, in their 2013 paper the DeepMind researchers used two DQNs instead of one

DQN 1 & DQN 2

Fixed Q-Value Targets

DQN 1

- Online model
- Learns at each step
- Used to move agent around

DQN 2

- Target model
- Used only to define targets
- Clone of online model

```
target = keras.models.clone_model(model)
target.set_weights(model.get_weights())
```

Fixed Q-Value Targets

- Then, in `training_step()` function
- We need to change one line to use target model
- Instead of online model
- When computing Q-Values of next states

```
next_Q_values = target.predict(next_states)
```

Fixed Q-Value Targets

- Finally, in training loop
- We copy weights of online model to target model
- At regular intervals (e.g., every 50 episodes)

```
if episode % 50 == 0:
    target.set_weights(model.get_weights())
```

Fixed Q-Value Targets

- Since target model is updated less often than online model
  - Q-Value targets are more stable
  - The feedback loop is dampened
  - Its effects are less severe

Fixed Q-Value Targets

- This approach was one of the DeepMind researchers' main contributions in their 2013 paper, allowing agents to learn to play Atari games from raw pixels

Double DQN

Double DQN

- In 2015, DeepMind researchers tweaked their DQN algorithm
- Increasing its performance and somewhat stabilizing training
- This variant is called **Double DQN**

Double DQN

- The update was based on observation that target network is prone to overestimating Q-Values

Double DQN

- If all actions are equally good
- Q-Values from target model must be identical
- But they are not as they are approximations
- Target model selects largest Q-Value > mean Q-Value
- So it's overestimating true Q-Value

Double DQN

What is the solution to this problem?

Double DQN

- Using online model instead of target model
  - for selecting best actions for next states
- Using target model only to estimate Q-Values for these best actions

Double DQN

- Here is the updated `training_step()` function:

```
def training_step(batch_size):
    experiences = sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = model.predict(next_states)
    best_next_actions = np.argmax(next_Q_values, axis=1)
    next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
    next_best_Q_values = (target.predict(next_states) * next_mask).sum(axis=1)
    target_Q_values = (rewards +
                       (1 - dones) * discount_factor * next_best_Q_values)
    mask = tf.one_hot(actions, n_outputs)
    [...] # the rest is the same as earlier
```

Switch to Notebook

`reinforcement_learning/reinforcement_learning.ipynb`

Prioritized Experience Replay

Prioritized Experience Replay

- Importance sampling (IS) or prioritized experience replay (PER), is sampling important experiences more frequently

Prioritized Experience Replay

- Here, we measure magnitude of TD error $\delta = r + \gamma \cdot V(s') - V(s)$
- Large TD error indicates a transition (s, r, s') is very surprising
- And probably worth learning from

Prioritized Experience Replay

- When an experience is recorded in replay buffer
- Its priority is set to very large value
- This ensures it gets sampled at least once

Prioritized Experience Replay

- Once it's sampled
- TD error δ is computed
- And this experience's priority is set to $p = |\delta|$

Prioritized Experience Replay

- Probability P of sampling an experience with priority p is proportional to p^{ζ}
- ζ = hyperparameter that controls how greedy we want importance sampling to be:
  - when $\zeta = 0$, we just get uniform sampling, and
  - when $\zeta = 1$, we get full-blown importance sampling

Prioritized Experience Replay

- Since we want important experiences to be sampled more often
- This also means we must give them lower weight during training, so we define each experience's training weight as:

$$w = (n\ P)^{-\beta}$$

n = # of experiences in replay buffer

β = hyperparameter to control how much we want to compensate for importance sampling bias

Prioritized Experience Replay

- The optimal value will depend on the task
- But if you increase one
- You will usually want to increase the other as well

Dueling DQN

Dueling DQN

- The **Dueling DQN algorithm** was introduced in yet another 2015 paper by DeepMind researchers

Here's how it work...

Dueling DQN

- Q-Value of a state-action pair (s, a) can be expressed as follows:

$$Q(s, a) = V(s) + A(s, a)$$

V(s) = Value of state s

A(s, a) = Advantage of taking action a in state s, compared to all other possible actions in that state

Dueling DQN

- Also, value of a state = Q-Value of best action a for that state
- So $V(s) = Q(s, a^*)$, which implies
- $A(s, a^*) = 0$

Dueling DQN

- In a Dueling DQN, the model estimates both value of the state and advantage of each possible action
- Since best action should have advantage of 0
- The model subtracts maximum predicted advantage from all predicted advantages

Dueling DQN

- Here is a simple Dueling DQN model, implemented using the Functional API:

```
K = keras.backend
input_states = keras.layers.Input(shape=[4])
hidden1 = keras.layers.Dense(32, activation="elu")(input_states)
hidden2 = keras.layers.Dense(32, activation="elu")(hidden1)
state_values = keras.layers.Dense(1)(hidden2)
raw_advantages = keras.layers.Dense(n_outputs)(hidden2)
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = keras.Model(inputs=[input_states], outputs=[Q_values])
```

The rest of the algorithm is just the same as earlier!

Switch to Notebook

reinforcement_learning/reinforcement_learning.ipynb

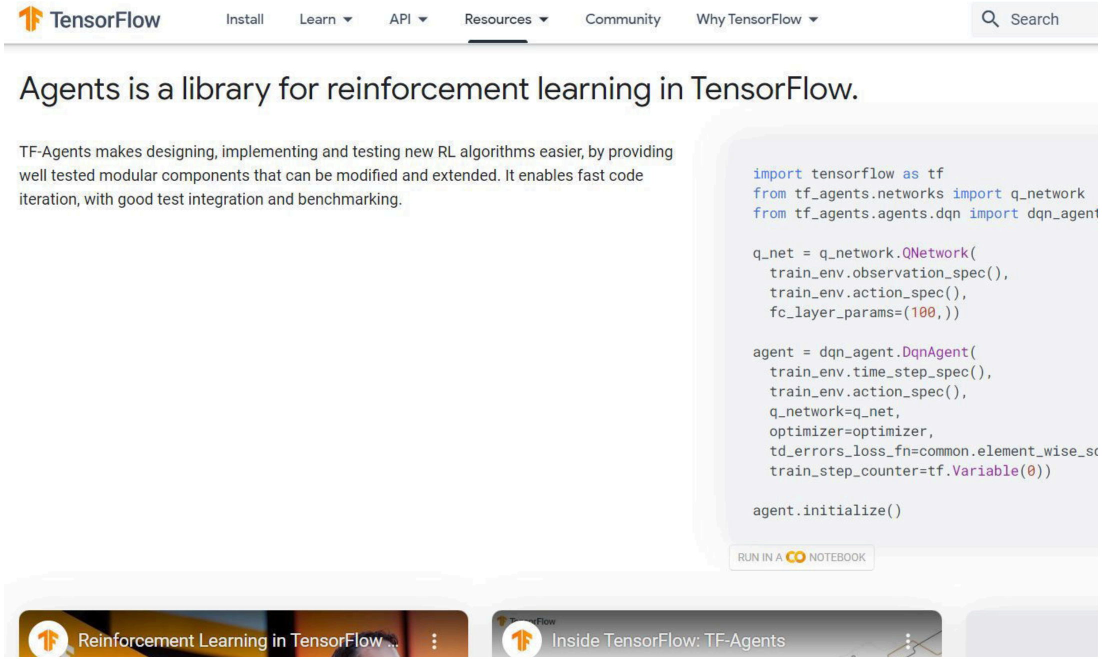
The TF-Agents Library

The TF-Agents Library

- Implementing all of the previous techniques
- Debugging them
- Fine-tuning them
- Training the models
- Require a huge amount of work

The TF-Agents Library

- So it is best to reuse scalable and well-tested libraries like TF-Agents



TensorFlow Install Learn API Resources Community Why TensorFlow Search

Agents is a library for reinforcement learning in TensorFlow.

TF-Agents makes designing, implementing and testing new RL algorithms easier, by providing well tested modular components that can be modified and extended. It enables fast code iteration, with good test integration and benchmarking.

```
import tensorflow as tf
from tf_agents.networks import q_network
from tf_agents.agents.dqn import dqn_agent

q_net = q_network.QNetwork(
  train_env.observation_spec(),
  train_env.action_spec(),
  fc_layer_params=(100,))

agent = dqn_agent.DqnAgent(
  train_env.time_step_spec(),
  train_env.action_spec(),
  q_network=q_net,
  optimizer=optimizer,
  td_errors_loss_fn=common.element_wise_sq
  train_step_counter=tf.Variable(0))

agent.initialize()
```

RUN IN A CO NOTEBOOK

Reinforcement Learning in TensorFlow ...

Inside TensorFlow: TF-Agents

The TF-Agents Library

- TF-Agents library is a Reinforcement Learning library based on TensorFlow, developed at Google and open sourced in 2018

The TF-Agents Library

Advantages of TF-Agents library:

- Provides many off-the-shelf environments
- Supports the
    - PyBullet library for 3D physics simulation
    - DeepMind's DM Control library
    - Unity's ML-Agents library

The TF-Agents Library

Advantages of TF-Agents library:

- Implements many RL algorithms
  - REINFORCE
  - DQN
  - DDQN, etc
- And various RL components e.g., efficient replay buffers and metrics

The TF-Agents Library

Advantages of TF-Agents library:

- It's fast, scalable, easy to use, and customizable
- You can create your own environments and neural nets
- You can customize pretty much any component

Isn't that something!

The TF-Agents Library

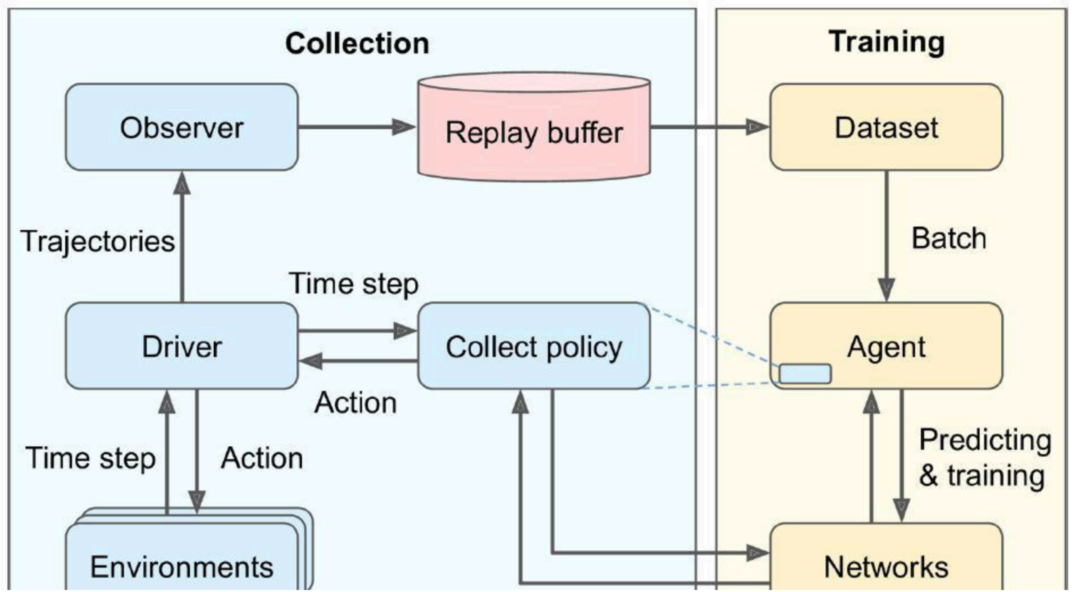
- We will use TF-Agents to train an agent to play Breakout, the famous Atari game using the DQN algorithm

Switch to Notebook

reinforcement_learning/reinforcement_learning.ipynb

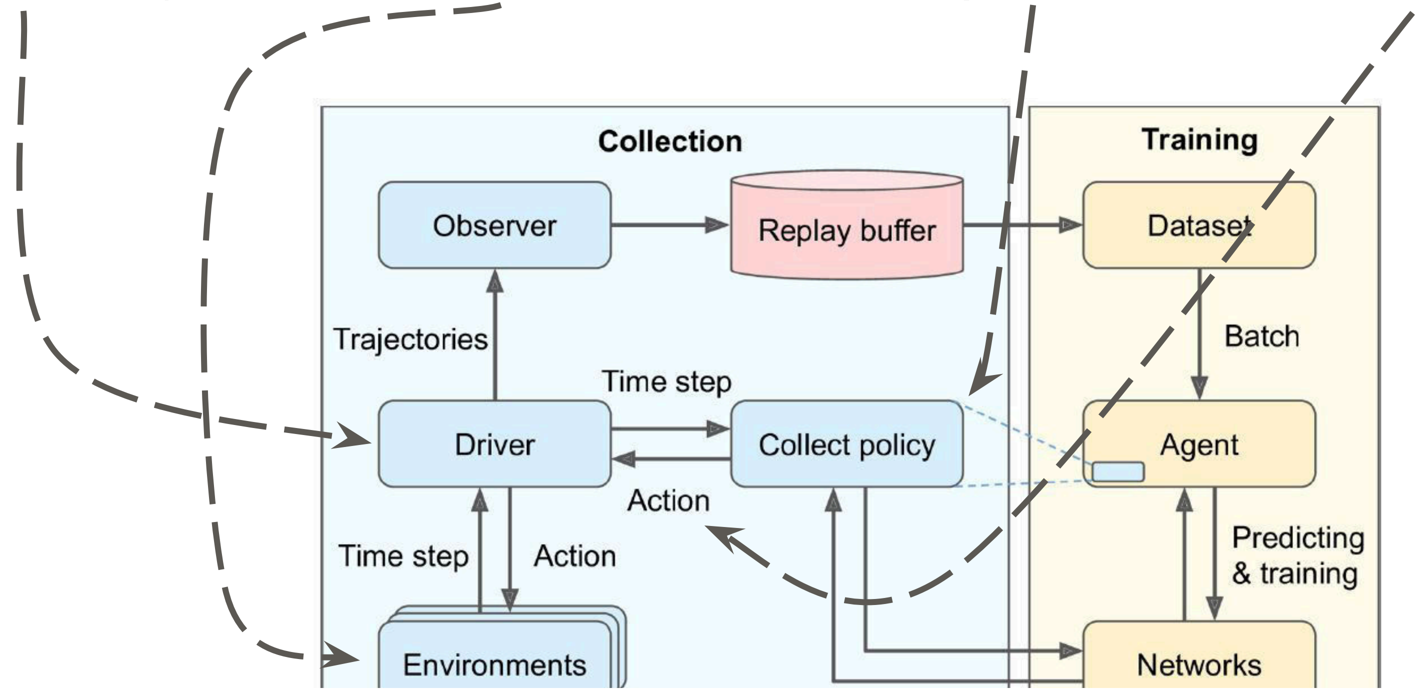
Training Architecture

TF-Agents training program is split into two parts that run in parallel



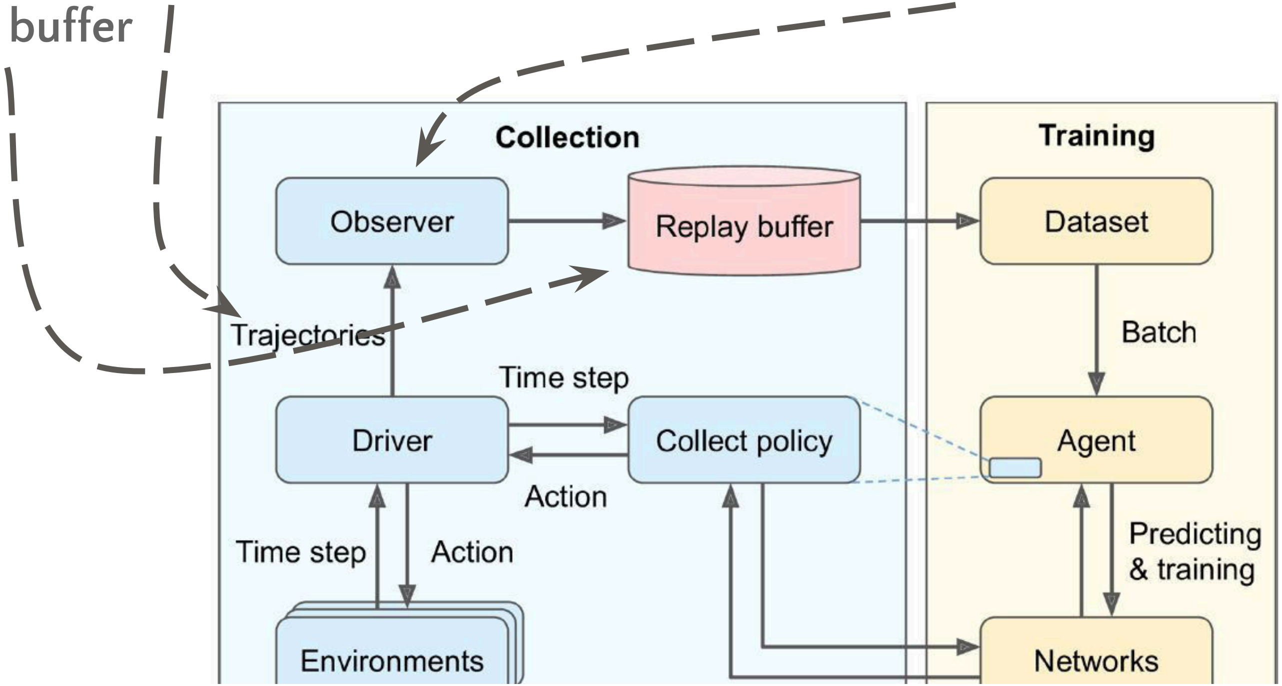
Training Architecture

Driver explores the **environment** using a **collect policy** to choose **actions**



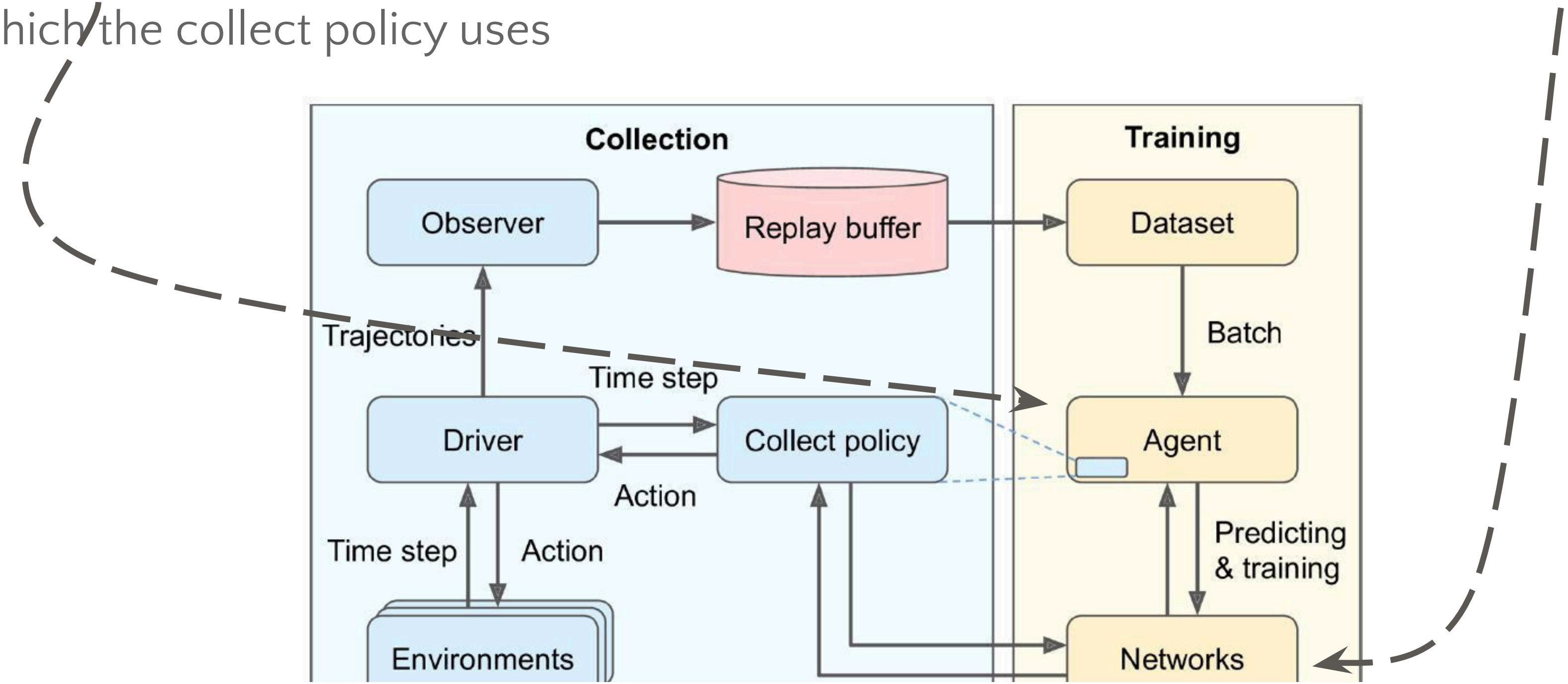
Training Architecture

It collects **trajectories**/experiences, sending them to **observer**, which saves them to a **replay buffer**



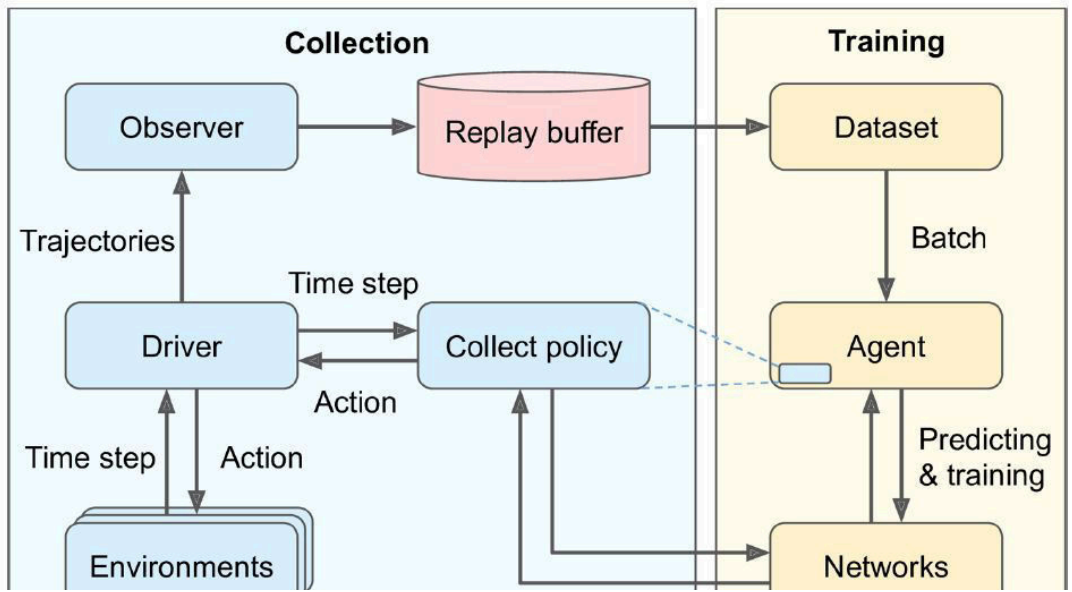
Training Architecture

An **agent** pulls batches of trajectories from replay buffer and trains some **networks**, which the collect policy uses



Training Architecture

The *left part* explores the environment and collects trajectories, the *right part* learns and updates the collect policy



The TF-Agents Library

You may have a few queries at this point.

The TF-Agents Library

Q: Why are there multiple environments?

A:

- To take advantage of the power of all CPU cores
- Keep the training GPUs busy
- Provide less-correlated trajectories to training algorithm

The TF-Agents Library

Q: What is a trajectory?

A:

- A concise representation of transition from one time step to the next
- Or a sequence of consecutive transitions from time step n to time step n + t

The TF-Agents Library

Q: Why do we need an observer?

A:

- To save trajectories to replay buffer
- To save them to TFRecord file
- To compute metrics
- You can pass multiple observers to driver, it will broadcast trajectories to all

The TF-Agents Library

Now we will create all these components

Switch to Notebook

reinforcement_learning/reinforcement_learning.ipynb

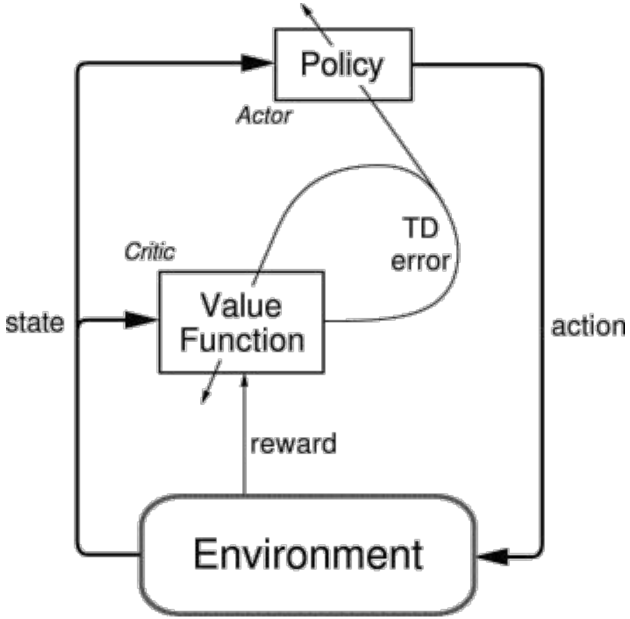
Overview of Some Popular RL Algorithms

Overview of Some Popular RL Algorithms

Let's take a quick look at a few popular RL algorithms!

Actor–Critic algorithms

- Family of RL algorithms that combine Policy Gradients with Deep Q–Networks



Actor-Critic algorithms

- Have separate memory structure
- Policy structure is known as the **actor**
  - Used to select actions
- Estimated value function is **critic**
  - Criticizes the actions made by the actor
- Learning is always on-policy

Actor-Critic algorithms

- Critic must learn about and
    - Critique whatever policy is currently being followed by the actor
    - Critique takes form of TD error
- This scalar signal is sole output of critic and drives all learning in both actor and critic

Asynchronous Advantage Actor-Critic (A3C)

- Actor-Critic variant introduced by DeepMind in 2016
- Multiple agents learn in parallel
  - Exploring different copies of environment
- Each agent pushes weight updates to master network
  - At regular intervals, but asynchronously
  - Then pulls latest weights from network
  - Thus contributing to improve master network
  - And benefit from what the other agents have learned
- Instead of estimating Q-Values, DQN estimates advantage of each action

Advantage Actor-Critic (A2C)

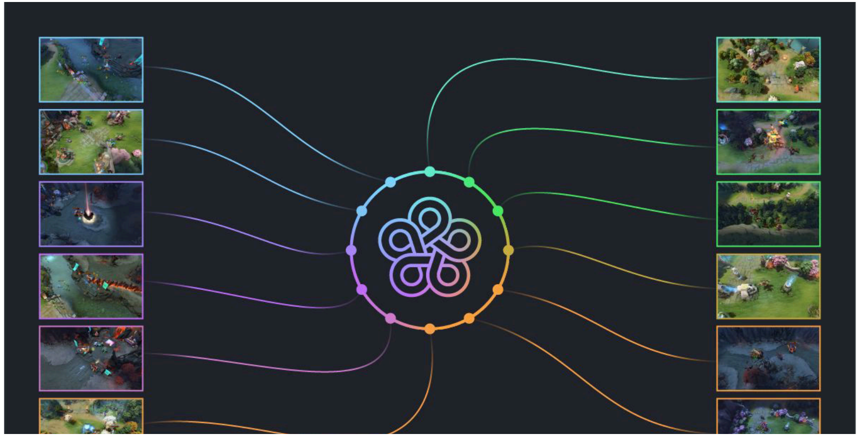
- Variant of A3C algorithm that removes the asynchronicity
- All model updates are synchronous
- Gradient updates are performed over larger batches
- Allows model to better utilize the GPU

Soft Actor-Critic (SAC)

- Actor-Critic variant proposed in 2018
- Learns not only rewards
- But also maximize the entropy of its actions
- SAC is available in TF-Agents

Proximal Policy Optimization (PPO)

- Algorithm based on A2C, available in TF-Agents
- Clips loss function to avoid excessively large weight updates
- Defeated world champions at multiplayer game Dota 2



Curiosity-based exploration

- Ignore the rewards to make agent extremely curious to explore environment
- Rewards becomes intrinsic to the agent
- Rather than coming from the environment

How does this work?

Curiosity-based exploration

- Agent continuously tries to predict outcome of its actions
- Seeks situations where outcome does not match predictions
- If outcome is predictable (boring), it goes elsewhere
- If outcome is unpredictable but the agent notices that it has no control over it, it also gets bored after a while
- Authors succeeded in training an agent at many video games
- https://pathak22.github.io/large-scale-curiosity/

References

- Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems by Aurelien Geron

- Incomplete Ideas by Richard S. Sutton
  http://incompleteideas.net/

Questions?