- HuggingFace Deep RL course (https://huggingface.co/deep-rl-course/unit0/introduction?fw=pt)
- HuggingFace Deep RL course github (https://github.com/huggingface/deep-rl-class)
- Reinforcement Learning book: Sutton (http://incompleteideas.net/book/RLbook2020.pdf)

Introduction: What is Reinforcement Learning (RL) ?

We have previously learned a form of learning called *Supervised Learning*

- learning a function from examples/demonstrations of the input/output relationship

We will now consider another form of learning called *Reinforcement Learning*.

Reinforcement Learning is the process whereby an *Agent* (the learner)

- learns a function by trial and error

In contrast to Supervised Learning

- where the learner learns from labeled examples
    - mappings from input to output

in Reinforcement Learning, the learner gathers information by interacting with the world

- The Agent is able to partially observe information (the *State*) about the world
- Given the State, the Agent has an available set of *Actions* that can be performed.
- the Agent's function (the *Policy*) guides its behavior: mapping the current State to an Action to perform
- the Agent performs the action
- The *Environment* responds to the action
    - with a *Reward*
    - and a new State

This interaction between Agent and Environment may continue for multiple steps

- multi-step sequence of State/Action/Reward/New State is called an *episode* or *trajectory*

The Agent's goal in formulating its Policy is maximization of reward received over the trajectory

- *Return* is the cumulative Reward (received over the sequence of chosen actions)

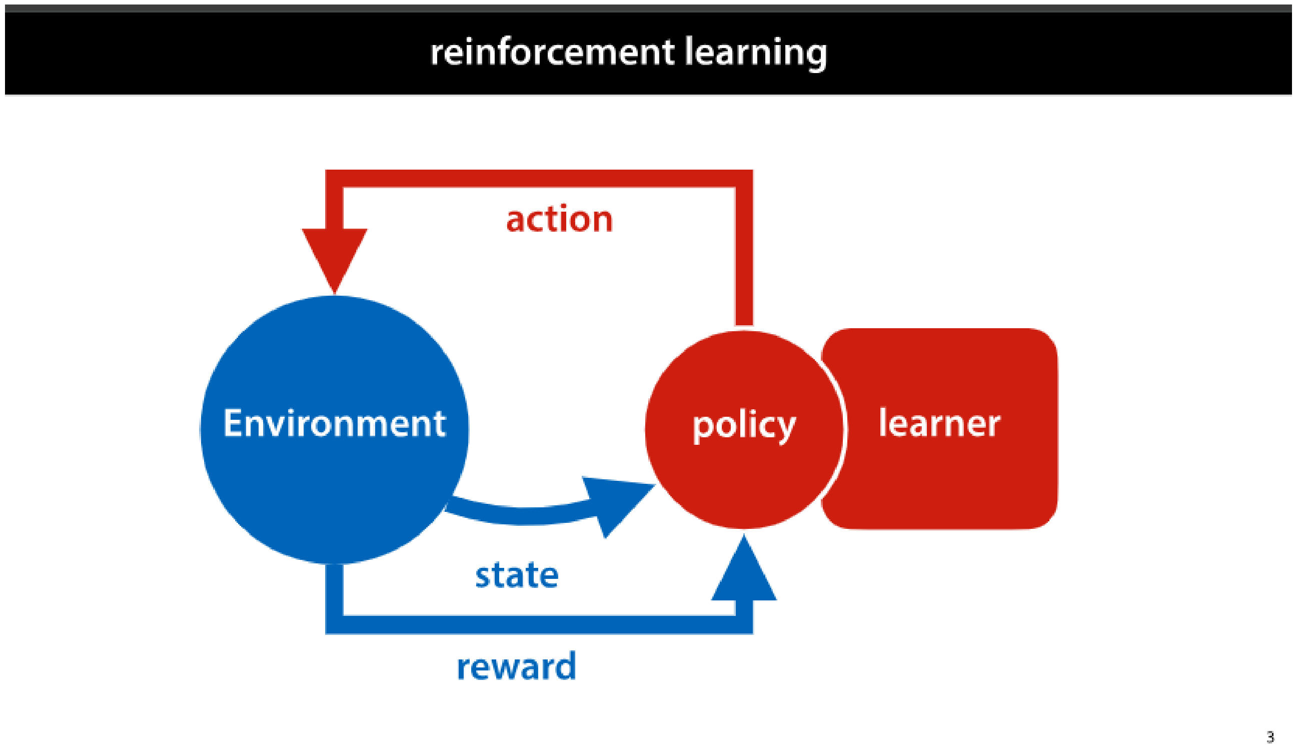
So Rewards are used by the Agent as a form of feedback

- evaluating the chosen Action
- guidance used to learn an optimal Policy.

Each episode/trajectory in analogous to an example

- deriving an optimal policy may require many episodes

Reinforcement Learning: information flow



Attribution: https://mlvu.github.io/lecture13/71.ReinforcementLearning.key-stage-0003.svg

Illustration: An Infant Learning to Walk

Imagine a baby who has never walked before.

Goal: Learn a sequence of actions that leads to walking across the room without falling.

State (s): The baby's current situation

- element of the set of States: { sitting, standing, wobbling, or stepping }.

Action (a):

- element of the set of Actions: { Move a leg forward, shift weight, stand still, or fall over }

Reward (r):

- Positive:
    - received when action a results in

Trial and Error Process:

At first: The baby tries random actions

- the Environment responds with a reward and a new State — maybe stands and immediately falls (low reward).

Over time:

- The baby notices that keeping balance while moving one leg gives a slightly better outcome
    - a bit longer before falling ⇒ slightly more reward).

Exploration:

- The baby keeps experimenting — sometimes falling, sometimes taking a step forward — to discover what works.

Exploitation:

- Once a short successful pattern of steps is found, the baby repeats it more often
    - because it consistently yields higher rewards (reaching a toy or a smiling parent).

Comparison with Supervised Learning

The baby learns to walk

- by doing
- receiving the consequences
- modifying its Policy in response to the consequences

Contrast this with Supervised Learning

- the examples are labeled:
    - a correct sequence is provided

In some sense:

- Supervised Learning is learning to *imitate* demonstrations
- Reinforcement Learning is learning by trial/error/feedback

We can continue the comparison

- Supervised Learning's optimization is: Loss Minimization
    - Loss is continuous, differentiable
- Reinforcement Learning's optimization is: Return Maximization
    - Rewards may be discrete, rather than continuous

Comment

Imitation feels "shallow", quantitative

- emphasizing syntactic equivalence with the target
- the "what" rather than the "why"

Learning from experience feels "deeper", qualitative

- emphasizing the result (semantics) rather than the exact path to the result

Consider how a model might learn to answer the question

```
How are you today ?
```

In Supervised Learning, there is a single Target

```
I feel fine, thank you
```

But there may be an equally acceptable response

```
Great, thanks for asking
```

An answers that is syntactically different than the Target results in a Loss.

In Reinforcement Learning, we can acknowledge multiple answers and rank them (via reward)

```
Fine.

Fine thank you.

Fine, thank you, and how are you feeling ?
```

Comparison: Reinforcement Learning vs Supervised Learning

Aspect	**Reinforcement Learning (RL)**	**Supervised Learning (SL)**
Learning Signal	Reward signal potentially delayed and sparse	Direct feedback with labeled input-output pairs
Data Acquisition	Data collected through interaction with environment	Data typically fixed and pre-collected
Goal	Learn a policy to maximize cumulative future reward	Learn a function mapping inputs to outputs
Feedback Type	Scalar reward signal, often sparse and delayed	Exact target labels for each input
Training Setup	Trial-and-error interaction, sequential data	Independent and identically distributed (i.i.d.) samples
Exploration	Critical to discover effective actions	Usually not required, data is given
Optimization Target	Maximize expected cumulative reward (possibly stochastic)	Minimize empirical loss over dataset
Environment Model	May be unknown or partially known, learning from experience	Typically no environment dynamics involved

A more formal example

A game where a user (Agent) attempts to keep a pole on a moving cart balanced for as long as possible.

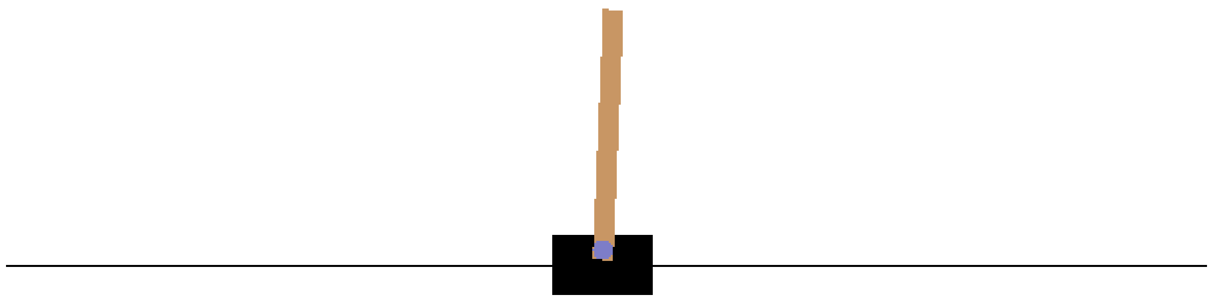
Reference

from Geron's book

- with some code (external/handson-ml2/18_reinforcement_learning.ipynb#A-simple-hard-coded-policy)

In [2]:
```
from IPython.display import Image
Image(open('images/cart_pole.gif','rb').read())
```

Out[2]:



States (aka *observations*):

- direction base is moving (right == 1)
- the angle of the pole (positive: leaning right)
- angular velocity of the pole (positive: tilting right)

Actions:

- action $\in 0, 1$: move the base left or right

Policy:

- move left if angle is negative; move right otherwise
- very naive policy

Reward:

- Positive for each action that keeps the pole upright
- maximize return by keeping pole upright for as long as possible

Here is some code that "plays the game": gathers experience

- each episode in an experience

Note that

- Policy is static
- NO learning is occurring

```
env.seed(42)

def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(200):
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

We can begin to develop some notation to describe what is happening.

An *episode* (or *trajectory*) is a sequence that records the events as agent follows its policy in making decisions.

Here is a timeline of an episode

- column labeled "Agent": actions chosen by the Agent
- column labeled "Environment": the responses generated in reaction to the decision

Step	**Agent**	**Environment**	**Notes**
0		$\stateseq_0$	Environment chooses initial state
	π $(\stateseq_0)$	$\rewseq_1$, $\stateseq_1$	Agent observes $\stateseq_0$
			chooses action $\pi(\stateseq_0)$ according to policy π
			receiving reward $\rewseq_1$
			environment updates state to $\stateseq_1$
1	π $(\stateseq_1)$	$\rewseq_2$, $\stateseq_2$	Agent continues to follow policy π

$\vdots$ | $\pi(\stateseq_)$ | $\rewseq_{+1}$, $\stateseq_{+1}$ | Agent follows policy to choose action $\pi(\stateseq_)$ | | | Environment responds to action by giving reward $\rewseq_{+1}$ and changing state to $\stateseq_{+1}$ $\vdots$

Challenges of RL

State

- The "full State" (all relevant information) may not be visible
    - The State available to the agent is *partially observable*

For example, suppose you (the Agent) are playing a game against a computer (the Environment)

- Chess/Go: full state is visible to Agent
- Poker: opponent (Environment) cards are not visible to Agent

Rewards

Sparse rewards

- rewards may *not be received at every step*
- in the limit
    - single reward at end of trajectory
    - e.g., your assignment grade is received when you complete the assignment, not at every partial solution

Environment

- state transition may be stochastic rather than deterministic

Policy

- action choice may be stochastic
- Policy is based only on State, not the trajectory
    - *Partially Observable Markov Decision Process (POMDP)*

Given the possible stochastic nature of the trajectory

- goal is maximization of *Expected Reward*

Challenges in finding an optimal policy

The Agent learns to update the policy through experience.

The quality of the experience matters

- you can't become an expert by only playing against weak opponents
- the agent may never achieve true optimal policy
    - for games where the optimal opponent strategy is not known (or is intractable computationally)
    - the agent can become the best *current* player of the game of Go
        - without a truly optimal policy

Exploration versus Exploitation

The agent's policy evolves with experience.

- accumulates more information about rewards and the environment through new episodes
- in the interim, it only has partial knowledge

Thus, if the agent

- chooses the same action \act every time it visits state \state
    - when the policy has not changed since the last visit
- it will never gain knowledge about other possible continuations of the trajectory

For example

- choosing alternate action $\backslash\mathrm{act}'$ may lead to a trajectory with higher return

The dilemma is called *exploration versus exploitation*

- *exploitation*: always choose the action with highest forward return
    - based on current limited knowledge
- *exploration*: take a new action, in order to explore alternate possible forward returns

Notation

Term	Definition
\States	**Set** of possible states
\Actions	**Set** of possible actions
\Rewards	**function** \States × \Actions → \Reals
	maps state and action to a reward
\disc	discount factor for reward one step in future
\stateseq_\tt	The state at beginning of time step
\actseq_\tt	The action performed at time step
\rewseq$_{+1}$	The reward resulting form the action performed at time step
\transp	**Function** \States × \Actions → \States × \Rewards
	Transition probability: maps state and chosen action
	to new state and reward received

Definitions relative to an episode

```
\begin{array}[lll]\\
\stateseq_0, \actseq_0, \rewseq_1, \ldots \stateseq_\tt, \actseq_\tt, \rewseq_{\tt+1}, \ldots
 & \textbf{sequence}: \text{Episode/Trajectory} \\
& \text{sequence of states, action performed, reward received} \\
\transp({ \state', \rew | \state, \act }) & \textbf{Transition probability} \text{ (rules of the game)} \\
& = \transp({\stateseq_\tt = \state', \rewseq_\tt = \rew | \stateseq_{\tt-1} = \state, \actseq_{\tt-1} = \a
 & \textbf{Markov Decision Process} \text{: depends only on }  \stateseq_{\tt-1} \\
 & \text{and not on history } \stateseq_0, \ldots, \stateseq_{\tt-1} \\
\pi(\act | \state) & \text{Policy (decision process for agent)} \\
            & \textbf{probability distribution over } \Actions \\
\pi(\state)       & \text{the action } \act \text{ chosen by the policy from state } \state \\
            & \text{abuse of notation: } \pi(\state) \text{ is a probability distribution over } \Actions \\
G_\tt & \text{return/return to go} \\
            & \text{cumulative rewards (discounted by } \gamma \text{) in the episode starting from ste
            & = \sum_{k=0}^\tt {  \gamma^k * \rewseq_{\tt+k+1} } \\
            & = \rew_{\tt+1}  + \gamma * G_{\tt+1} \\
            & & \text{ where } \gamma \text{ is a factor for discounting future returns.}\\
\end{array}
```

Reinforcement Learning: information flow (with labels)

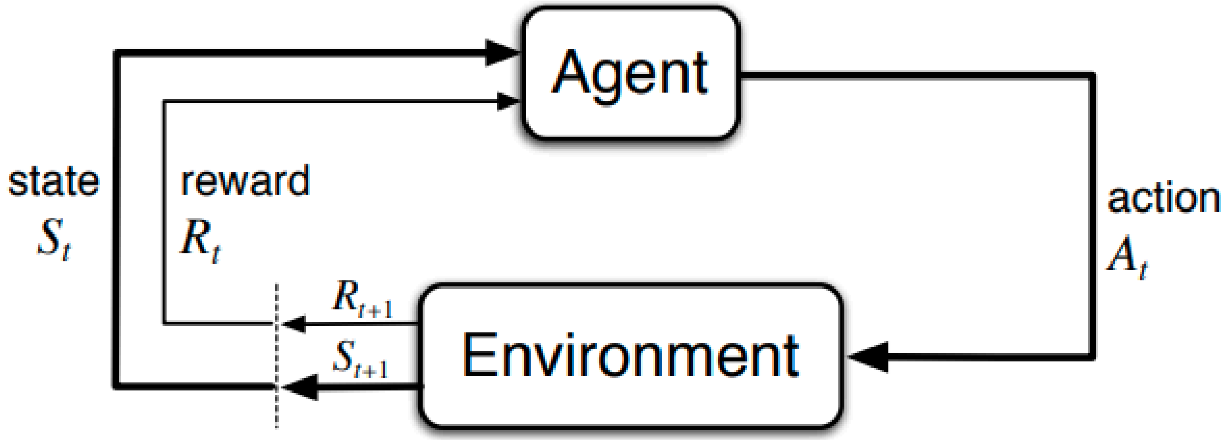


Figure 3.1: The agent–environment interaction in a Markov decision process.

Attribution: completeideas.net/book/RLbook2020.pdf#page=70

Notes on episodes

- The triple of elements corresponding to experience number
    - is $\stateseq_{}, \actseq_{}, \rewseq_{+1}$
    - **not** $\boxed{\text{\stateseq\_\tt, \actseq\_\tt, \rewseq\_\tt}}$
    - **not** $\boxed{\text{\rewseq\_\tt, \stateseq\_\tt, \actseq\_\tt}}$
        - i.e., there is a reward (without action) from being in the initial state
        - some presentations use this; we will adopt the notational standard of the [Sutton and Barto book (http://incompleteideas.net/book/RLbook2020.pdf)](http://incompleteideas.net/book/RLbook2020.pdf).
- The Algorithms evaluating experience number
    - **do not have access** to future experiences numbered $' > +1$
    - they gain access to the next experience $+1$
        - by "playing the game"
        - submitting $\boxed{\text{\actseq\_\tt}}$ to the environment and receiving $\rewseq_{+1}$ and $\stateseq_{+1}$
    - Under the assumption of MDP (Markov Decision Process)
        - the algorithm *does not need* access to experiences numbered $' <$
- Episodes are more a notation/record than a piece of data used by an algorithm

Finding an optimal Policy: Solving an RL system

A policy π is a function mapping a state to an action.

It is the "algorithm" that guides the actions behavior.

The "solution" to an RL system is the optimal policy π^* that maximizes *return* from the initial state

- return is sum of discounted rewards accumulated by following the policy from a given state $$ \begin{array} \ G*tt & = & \sum*{k=0}^\tt { \gamma^k * \rewseq_{\tt+k+1} } \

  ```
   & = & \rew_{\tt+1}  + \gamma * G_{\tt+1} \\
  ```

  \end{array} $$ where $\gamma \leq 1$ is a factor for discounting future returns.

Note on the discount factor

What is the purpose of the discount factor ?

Given two trajectories with *equal* return

- we sometimes want to favor the *shorter* trajectory
    - favor direct path over indirect path
    - favor immediate rewards to deferred rewards
- the discount factor is a way of expressing our preference

For simplicity of presentation, we often assume

$$\gamma = 1$$

How do we find the optimal policy π ?

The way we find the optimal policy is typically via an iterative process

- We construct a sequence of improving policies
  $$\pi_0, \ldots, \pi_p, \ldots$$
  that hopefully converges to π^*.

For simplification, let us assume for the moment that

- the sets of states, actions and rewards be *finite*.

RL Problem Types: Taxonomy and Varieties

This section serves as an overview of the field and preview of the rest of the topics.

Model-Based vs Model-Free RL

There are two main approaches to solving an RL system.

A *model-based* approach uses a model of the environment in forming a solution.

The model defines the response of the Environment to an action of the Agent

- $\backslash\mathrm{transp}(\backslash\mathrm{state}', \backslash\mathrm{rew} | \backslash\mathrm{state}, \backslash\mathrm{act})$

The model can be either

- given
- learned

The advantage of having a model is that

- the Agent can explore the consequences of an action without having to perform an episode.

A *model-free* approach does not rely on a pre-defined model

- it learns from interacting with the environment

- Model-Based RL: Learns or uses the dynamics model $P(s'|s,a)$.
    - Example methods: Dynamic Programming, Dyna, Monte Carlo Tree Search.
- Model-Free RL: Learns policies or value functions without modeling P.
    - Example methods: Q-Learning, SARSA, Policy Gradient.

Analogy

Route-finding

Imagine you are trying to find a route from point A to point B.

You can easily explore many alternatives if given a map: this is model-based.

Without a map, you only learn the direction/relationships between streets by doing: model-free

Value-Based vs Policy-Based Methods

There are several major approaches to solving an RL problem

- Value-Based Methods
  - Learn function to either
    - map state $\state$ to an expected return $\statevalfun(\state)$
    - map state-action pair to an expected return $\actvalfun_{\pi}(\state, \act)$
  - Derive policies indirectly from these functions
  - Examples: Q-learning, DQN.
- Policy-Based Methods
  - Learn the policy $ \pi(\act \; | \; \state)$ directly.
  - Examples: REINFORCE, Actor-Critic, PPO.
- Actor-Critic
  - Hybrid methods combining policy and value learning.

On-Policy vs Off-Policy Methods

Some methods involve two potentially distinct choices for actions.

- the *behavior policy*: the one that choose an action *while learning*
- the *target policy*: the one that we are trying to learn; reflected in the update

An *On-Policy* method

- Behavior and Target policies are the same
- Learn from the policy used to generate data ($ \pi $).
    - Examples: SARSA, REINFORCE.

An *Off-policy* method

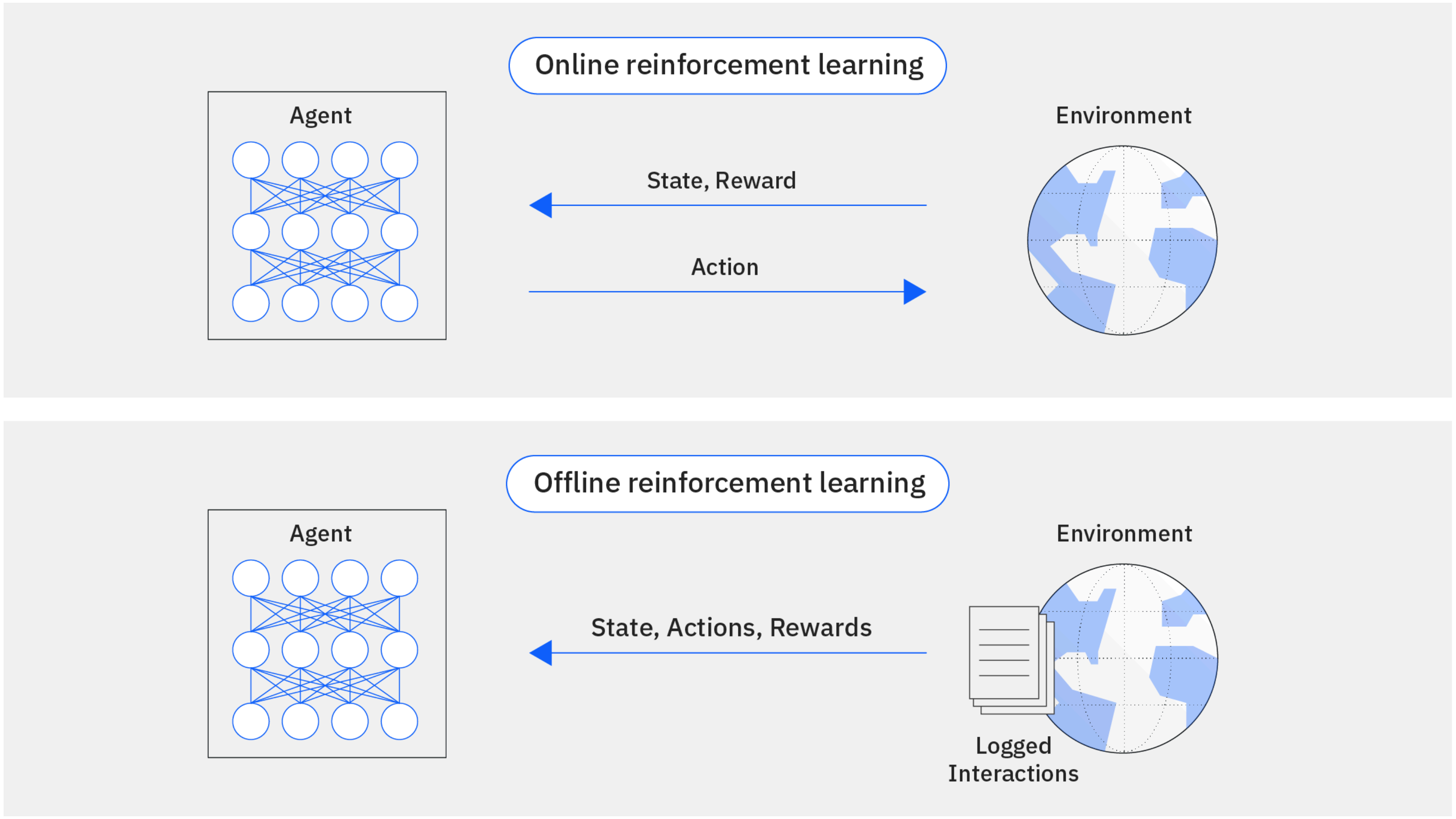
- Behavior and Target policies are different
- Learn about a target policy $ \pi $ different from behavior policy $ \mu $.
    - Examples: Q-learning, DQN.

On-Line vs Off-Line RL

The experience could be gained either

- *On-line*: playing the game
- *Off-line*: replaying the experiences of prior attempts
- On-Line RL: Updates and learns from interaction with environment in real time.
- Off-Line RL: Learns from a fixed dataset without further interaction.
    - Applications: healthcare, robotics.

Reinforcement Learning: Online vs Offline



Attribution: https://www.ibm.com/think/topics/reinforcement-learning

Core RL Algorithms and Their Categories

Algorithm	Model-Based	Model-Free	Value-Based	Policy-Based	On-Policy	Off-Policy	On-Line	Off-Line
Value Iteration	✓		✓					
Policy Iteration	✓		✓	✓				
SARSA		✓	✓		✓		✓	
Q-learning		✓	✓			✓	✓	
REINFORCE		✓		✓	✓		✓	
Actor-Critic		✓	✓	✓	✓	Some	✓	
Deep Q-Networks (DQN)		✓	✓			✓	✓	✓
PPO, A2C/A3C		✓	✓	✓	✓		✓	
Batch RL		✓	✓			✓		✓

Deep Reinforcement Learning

Deep Reinforcement Learning refers to the special case of Reinforcement Learning where

- the *policy* is a parameterized (by θ) function mapping states \stateseq to (a probability distribution) actions
$$\pi_\theta(\text{\\actseq}|\text{\\stateseq})$$
- implemented as a Neural Network

Our initial presentation will be of fixed (non-parameterized) policies.

- We will subsequently introduce parameterized Neural Networks to implement the functions we define

```
In [3]: print("Done")
```

```
Done
```