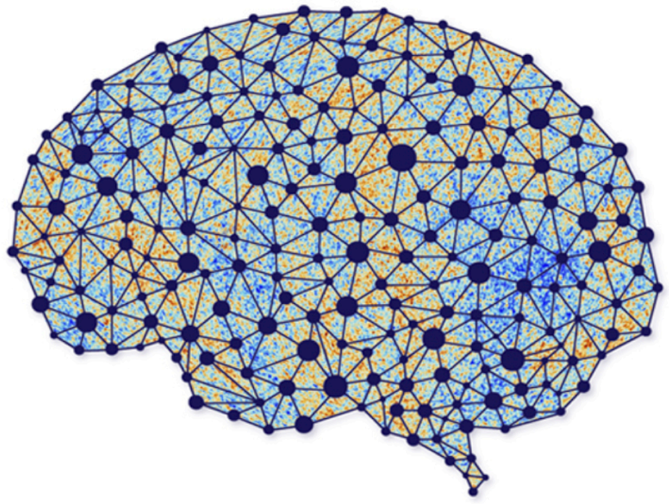
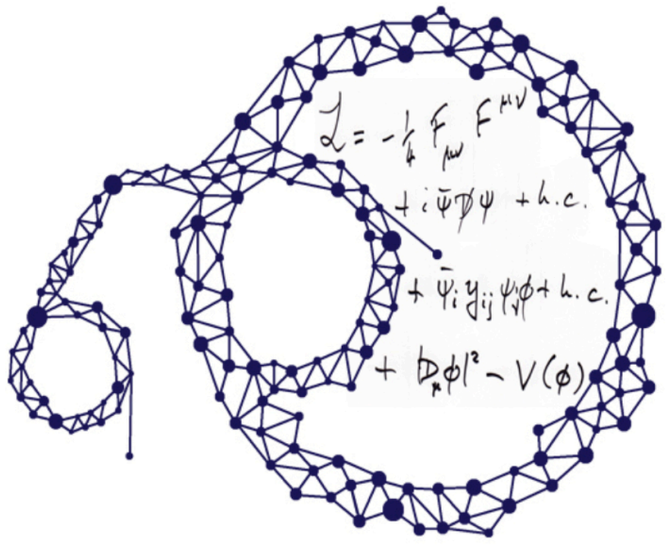
Physics 361 - Machine Learning in Physics

Lecture 19 – Reinforcement Learning 2

April 1st 2025

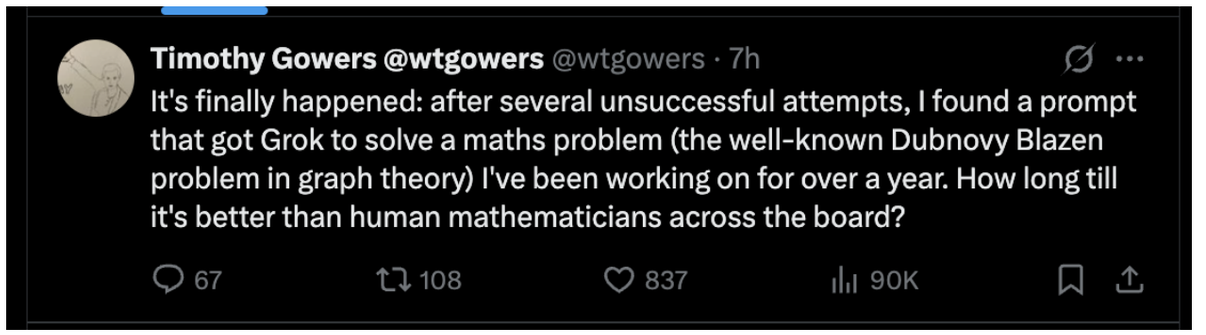


AI
∩
Universe



Moritz Münchmeyer

Huge day for AI in Science ;)



A Search for “New Physics” “Beyond the Standard Model” in Open Data with Machine Learning

April 1, 2025

Rikab Gambhir,[a,b]

[a] *Center for Theoretical Physics, Massachusetts Institute of Technology, Cambridge, MA 02139, USA*
[b] *The NSF AI Institute for Artificial Intelligence and Fundamental Interactions, USA*

E-mail: rikab@mit.edu

Abstract: In this new era of large data, it is important to make sure we do not miss any signs of new physics. Using the publicly-available open data collected by the arXiv.org experiment in the `hep-ph` channel, corresponding to a raw total integrated $\mathcal{L}$iterature of 65,276 papers, we perform a search for “New Physics” and related signals. In the worst-case, we are able to detect “New Physics” with “the LHC” at a significance level of at least 6.5σ. This “New Physics” signature is primarily “Dark” in nature, and is potentially axion(-like) dark matter. We also show the potential for further improvement in the future, and that “New Physics” can be found with “a Future Collider” at at least 8.9σ, as well as the potential to find “New Physics” without any collider at all. This search is performed using code that was 80% written by Machine Learning methods.

Reinforcement Learning

Introduction

Introduction

- Reinforcement leaning is a sequential decision making framework in which agents learned to perform actions in an environment with the goal of maximizing rewards.

- RL controls the **actions** of an **agent** in an **environment** to maximize the **reward**.

- RL applications: Go/Chess/Atari, robotics, financial trading, string theory, optimal experimental design, robotics, reasoning, ….

- RL is often used when problem involves searching a large configuration space.

- References:

  - Sutton and Barto: http://incompleteideas.net/book/the-book-2nd.html,

  - **Simon Prince, Understanding Deep Learning: https://udlbook.github.io/udlbook/ (primary reference used here)**

- https://github.com/Farama-Foundation/Gymnasium (formerly https://github.com/openai/gym)

Challenges of RL

- Illustrate the challenges with **chess game**. A reward of +1, –1, or 0 is given at the end of the game if the agent wins, loses, or draws and 0 at every other time step. The challenges:

  - The reward is sparse; we must play an entire game to receive feedback.

  - **Temporal credit assignment problem**: The reward is temporally offset from the action that caused it; a decisive advantage might be gained thirty moves before victory. We must associate the reward with this critical action. (other examples?)

  - **The environment is stochastic**; the opponent doesn't always make the same move in the same situation, so it's hard to know if an action was truly good or just lucky.

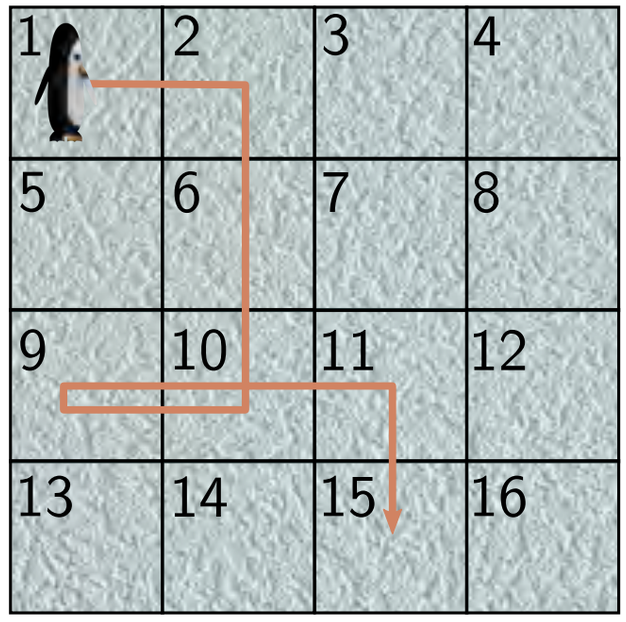
  - **Exploration-exploitation trade-off**: The agent must balance exploring the environment (e.g., trying new opening moves) with exploiting what it already knows .

Reinforcement Learning

General Definitions

Markov Processes

- In RL, we learn a **policy** that maximizes the expected return in a Markov decision process.
- The word Markov implies that the probability of being in a state depends only on the previous state and not on the states before.
- The changes between states are captured by the transition probabilities $Pr(s_{t+1} \mid s_t)$ of moving to the next state s_{t+1} given the current state s_t, where t indexes the time step.
- A Markov process is an evolving system that produces a sequence $s_1, s_2, s_3, \ldots$ of states.
- τ = [s1, s2, s3, . . .] is called the **trajectory**



$s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8$

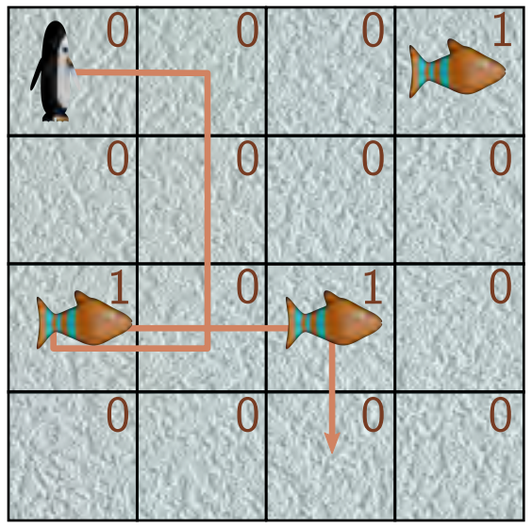
$\tau = [1, 2, 6, 10, 9, 10, 11, 15]$

Markov Reward Processes

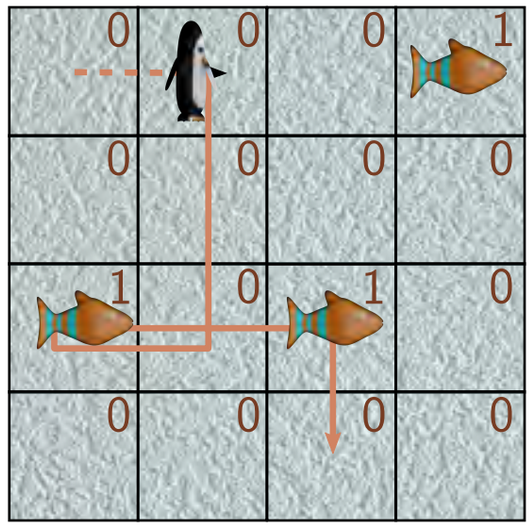
- A Markov reward process also includes a distribution $Pr(r_{t+1} \mid s_t)$over the possible rewards r_{t+1} received at the next step, given s_t.
- Introduce a discount factor $\gamma \in (0,1]$ to compute the (cumulative) **return** G_t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}.$$

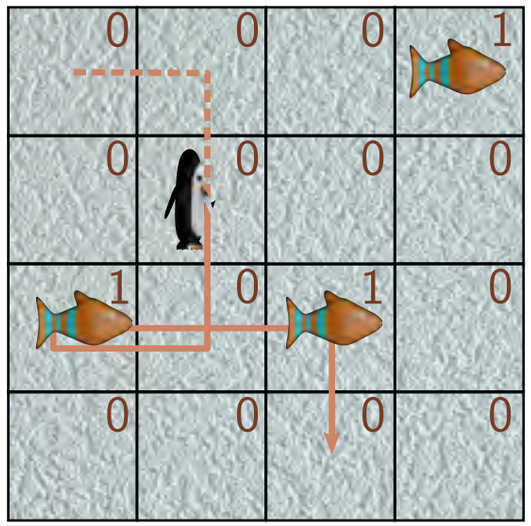
a) $G_1 = 0 + \gamma \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 0 + \gamma^4 \cdot 1 + \gamma^5 \cdot 0 + \gamma^6 \cdot 1 + \gamma^7 \cdot 0 = 1.19$



b) $G_2 = 0 + \gamma \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 1 + \gamma^4 \cdot 0 + \gamma^5 \cdot 1 + \gamma^6 \cdot 0 = 1.31$



c) $G_3 = 0 + \gamma \cdot 0 + \gamma^2 \cdot 1 + \gamma^3 \cdot 0 + \gamma^4 \cdot 1 + \gamma^5 \cdot 0 = 1.47$

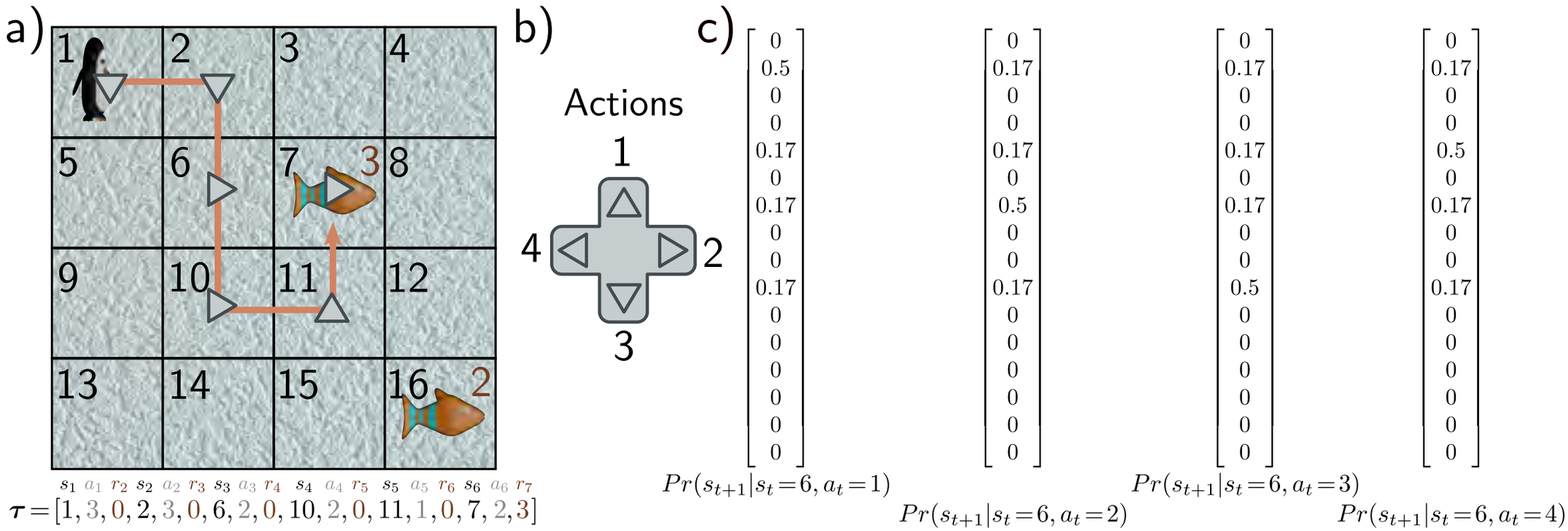


$s_1\ r_2\ s_2\ r_3\ s_3\ r_4\ s_4\ r_5\ s_5\ r_6\ s_6\ r_7\ s_7\ r_8\ s_8\ r_9$

$$\boldsymbol{\tau} = [1, 0, 2, 0, 6, 0, 10, 0, 9, 1, 10, 0, 11, 1, 15, 0]$$

Markov Decision Processes

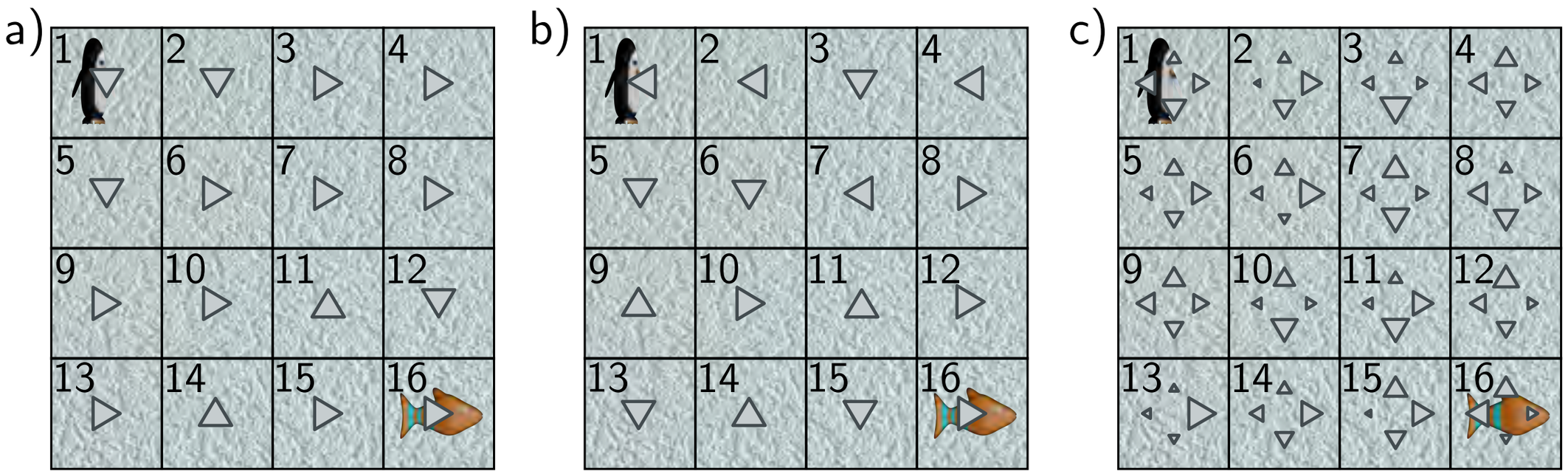
- A Markov decision process (MDP) adds a set of possible action a_t at each step which changes the transition probabilities $Pr(s_{t+1} \mid s_t, a_t)$.
- The rewards can also depend on the action: $Pr(r_{t+1} \mid s_t, a_t)$.
- MDP produces a sequence $s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, \ldots$ of states, actions & rewards. The entity that performs the actions is the **agent**.



Here: The penguin moves in the intended direction with 50% probability, but the ice is slippery, so it may slide to one of the other adjacent positions with equal probability.

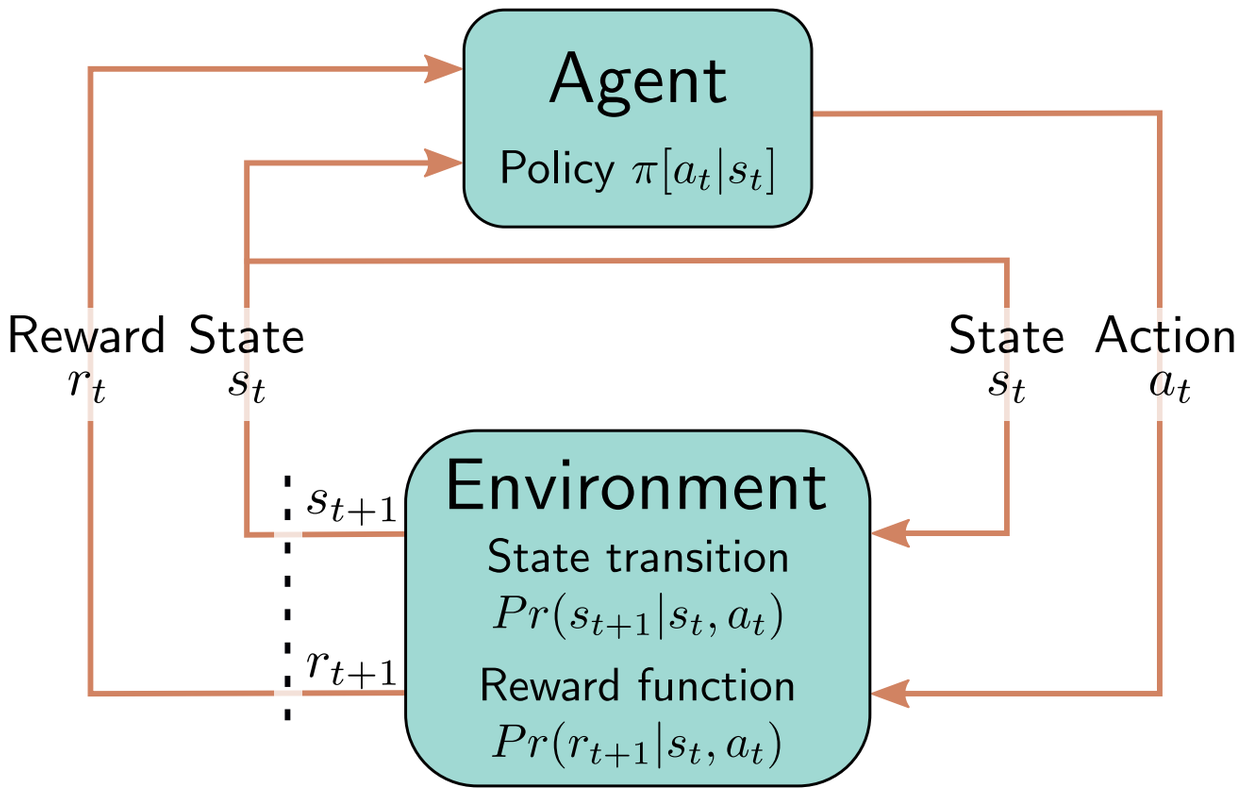
Policy

- The rules that determine the agent's action are known as the **policy**.
- The policy can be **deterministic** (one action for a given state) or **stochastic** (a probability distribution over each possible action):



Reinforcement Learning Loop

- The environment and the agent form a loop:



- The agent receives the state and reward from the last time step. Based on the policy, the agent chooses the next action.
- The environment then assigns the next state according to $Pr(s_{t+1} \mid s_t, a_t)$ and the reward according to $Pr(r_{t+1} \mid s_t, a_t)$.

Expected return: state and action values

- The return G_t depends on the state s_t and the policy $\pi[a \mid s]$
- Characterize how "good" a state is under a given policy π by considering the expected return $v[s_t \mid \pi]$. **State-value function** (long-term return on average from sequences that starts from s_t):

$$v[s_t|\pi] = \mathbb{E}\Big[G_t|s_t, \pi\Big].$$

- **Action value** or state-action value function $q[s_t, a_t \mid \pi]$ is the expected return from executing action a_t in state s_t:

$$q[s_t, a_t|\pi] = \mathbb{E}\Big[G_t|s_t, a_t, \pi\Big].$$

- Through this quantity, RL algorithms connect future rewards to current actions (i.e., resolve the temporal credit assignment problem).

Optimal Policy

- We want a policy that maximizes the expected return.
- For MDPs, there ∃ a deterministic, stationary (depends only on the current state, not the time step) policy that maximizes the value of every state.
- If we know this optimal policy, then we get the optimal state-value function:

$$v^*[s_t] = \max_{\pi} \Big[\mathbb{E}\Big[G_t | s_t, \pi \Big] \Big].$$

- Similarly, the optimal state-action value function:

$$q^*[s_t, a_t] = \max_{\pi} \Big[\mathbb{E}\Big[G_t | s_t, a_t, \pi \Big] \Big].$$

- Turning this around, if we knew the optimal action-values, we can derive the optimal policy.

$$\pi[a_t | s_t] \leftarrow \operatorname*{argmax}_{a_t} \Big[q^*[s_t, a_t] \Big].$$

Consistency of state and action values

- We may not know the state values v or action values q for any policy. However, we know that they must be consistent with one another, and it's easy to write relations between these quantities.

- The state values are given by

$$v[s_t] = \sum_{a_t} \pi[a_t|s_t] \cdot q[s_t, a_t]$$

State value — Prob of action — Action value

- The action values are given by

$$q[s_t, a_t] = r[s_t, a_t] + \gamma \cdot \sum_{s_{t+1}} Pr(s_{t+1}|s_t, a_t) \cdot v[s_{t+1}]$$

Action value — Reward for action — Discount factor — Prob of next state — Value of next state

- These relations lead to the "Bellman equations", a central concept in RL.

Reinforcement Learning

Tabular RL Methods

Tabular RL

- **"Tabular RL"** refers to **reinforcement learning methods that use explicit tables to store value functions (or policies)**. This means that every state (or state-action pair) is explicitly represented in a **lookup table** rather than being approximated by a function (e.g., a neural network).

- This a**pproach is feasible when the state space is small because all possible states and actions can be stored explicitly**.

- We start with this setup.
- We will later contrast tabular RL algorithms with the use of deep learning in RL that does not require storing the large lookup table.

Model Based vs Model-Free Methods

- **Model-based methods** use the MDP structure explicitly and find the best policy from the transition matrix $Pr(s_{t+1} \mid s_t, a_t)$ and rewards $r[s, a]$.
- If the transition matrix & reward are known, this optimization problem can be solved with **dynamic programming, which is an algorithm that iteratively evaluates and improves the policy** (see Prince book for details).

Model Based vs Model-Free Methods

- **Model-free methods** do not use transition probabilities and reward structure explicitly. Instead, they **learn from interactions with the environment**. They fall into two classes:
  - Value estimation - estimate the optimal state-action value **q** and then assign the policy according to the action with the greatest value.
  - Policy estimation - estimate the optimal policy using gradient descent w/o the intermediate steps of estimating the model or values.
- Example of Model-free method applications:
  - Playing computer games: The agent interacts with the environment and learns what actions lead to higher scores, without knowing the game's internal rules.

Monte Carlo vs Temporal Difference

- **Monte Carlo** methods simulate many trajectories through the MDP for a given policy to gather information to improve this policy.
- **Temporal difference** methods update the value estimates and policy while the agent traverses the MDP, one step at a time.
- More details for both methods will be in the following slides.

Monte Carlo Value Estimation Methods

- Alternate **between computing the action values (based on repeatedly sampling trajectories)** & **updating the policy (based on action values).**
- To **estimate the action values q[s, a], a series of episodes are run**. Each starts with a given state and action and thereafter follows the current policy, producing a series of actions, states, and rewards. The **action value** for a given state-action pair under the current policy is **estimated as the average of the empirical returns that follow it**.
- The policy is updated by choosing the action with the maximum value at each state:

$$\pi[a|s] \leftarrow \underset{a}{\operatorname{argmax}}\Big[q[s,a]\Big]$$

On-policy vs off-policy methods

- In an **On-policy** method the current best policy is used to guide the agent through the environment.
- It is not possible to estimate the value of actions that have not been used, & there is nothing to encourage the algorithm to explore them.
- ϵ-greedy policy: random action is taken with ϵ probability and optimal action with $1 - \epsilon$ probability (exploitation/exploration trade-off).
- **Off-policy** method: the optimal policy π (the target policy) is learned based on episodes generated by a different behavior policy π'. Typically, the target policy is deterministic, and the behavior policy is stochastic.
- We want π' to explore the environment (stochastic) and the learned policy π to be efficient.
- The on/off policy distinction applies both to Monte Carlo and Temporal Difference methods.

Temporal difference (TD) methods

- Update the values/policy while the agent traverses the states of MDP.
- **SARSA** (State-Action-Reward-State-Action) is an on-policy value estimation algorithm with update:

$$q[s_t, a_t] \leftarrow q[s_t, a_t] + \alpha\Big(r[s_t, a_t] + \gamma \cdot q[s_{t+1}, a_{t+1}] - q[s_t, a_t]\Big),$$

where $\alpha \in \mathbb{R}^+$ is the learning rate. The bracketed term is TD error.

- **Q-learning** is an off-policy value estimation algorithm with update:

$$q[s_t, a_t] \leftarrow q[s_t, a_t] + \alpha\Big(r[s_t, a_t] + \gamma \cdot \max_a \big[q[s_{t+1}, a]\big] - q[s_t, a_t]\Big),$$

It estimates the value of the next state assuming an optimal followup action.

SARSA update rule details

Agent choses an action from the policy (which includes exploration). Then we update the q function:

$$q(s_t, a_t) \leftarrow q(s_t, a_t) + \alpha \left(r(s_t, a_t) + \gamma \cdot q(s_{t+1}, a_{t+1}) - q(s_t, a_t) \right)$$

$q(s_t, a_t)$:

- The current estimate of the **Q-value** for taking action a_t in state s_t.
- This represents the agent's **expected return** from this state-action pair.

$r(s_t, a_t)$:

- The **reward** received from the environment for taking action a_t in state s_t.

$q(s_{t+1}, a_{t+1})$:

- The **Q-value of the next state-action pair**, where a_{t+1} is chosen **according to the current policy**.
- Unlike Q-learning (which uses $\max_a q(s_{t+1}, a)$), SARSA **follows the policy's action selection**.

γ (Discount Factor):

- Determines how much the **future reward** is valued.
- A value closer to 1 makes the agent more **future-focused**, while a value closer to 0 makes it focus on **immediate rewards**.

α (Learning Rate):

- Controls how much new information updates the existing Q-value.
- A lower α means **slower learning**, while a higher α makes updates **more aggressive**.

Update Rule Breakdown:

- Compute the **TD Target** (what the Q-value should move toward):

$$r(s_t, a_t) + \gamma q(s_{t+1}, a_{t+1})$$

- Compute the **TD Error** (difference between target and current estimate):

$$(r(s_t, a_t) + \gamma q(s_{t+1}, a_{t+1}) - q(s_t, a_t))$$

- Scale this difference by the **learning rate** α and update $q(s_t, a_t)$.

SARSA is an on policy method:

1. From state s_t, the agent chooses action a_t using its **policy** (e.g., ε-greedy).
2. It executes a_t, observes reward r_t, and transitions to state s_{t+1}.
3. It then selects **the next action** a_{t+1} using the **same policy**.
4. The update uses $q(s_{t+1}, a_{t+1})$ — the value of the actual action the policy *did choose*, not the best possible one.

Q-learning update rule details

The **Q-learning** update rule is:

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

where:

- s_t, a_t = current state and action
- r_{t+1} = reward received after taking action a_t
- s_{t+1} = next state
- $\max_a Q(s_{t+1}, a)$ = maximum predicted reward from the next state s_{t+1} (greedy action selection)
- α = learning rate (step size)
- γ = discount factor (determines the importance of future rewards)

How It Works

1. The agent selects an action a_t, usually based on an exploration strategy like ϵ-greedy.
2. The environment provides a reward r_{t+1} and the next state s_{t+1}.
3. Instead of following the current policy, the update uses the **greedy** action $\max_a Q(s_{t+1}, a)$ assuming the best future reward.
4. The Q-table (or function) is updated with the rule above.
5. The process repeats until convergence.

That's why Q-learning is called an **off-policy** algorithm:

- It **updates the Q-values as if** we're following the greedy (optimal) policy,
- Even though the behavior might be exploring (e.g., ε-greedy or completely random).

a_t:

- This is the **action the agent actually took** in state s_t.
- It was chosen according to the agent's **behavior policy** (often ε-greedy — mostly greedy, sometimes random).
- It's the one we're updating in the Q-table.

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

- This is the **best possible action** the agent *could take* in the **next state** s_{t+1}, according to its current Q-values.
- It represents the assumption that the agent **will act optimally** from s_{t+1} onward — even if it doesn't in reality.
- It's part of the **target** in the update, not what the agent actually does.

Reinforcement Learning

Value-based RL with Neural Networks

From tabular RL to Machine Learning based RL

- The tabular Monte Carlo and TD algorithms described above repeatedly traverse the entire MDP and update the action values. However, this is **only practical if the state action space is small**. Unfortunately, this is rarely the case; even for the constrained environment of a chessboard, there are more than 10^{40} possible legal states.
- Instead of discrete state indices s_t, we now use a state vector $\mathbf{s_t}$, and we **replace discrete value function and policies with machine learning models that take in the state vector.**
- There are two main approaches to RL for this situation:
  - **Value-based methods** (e.g., Deep Q-Networks): These estimate a value function **q(s,a)** that represents the expected reward for taking action a in state s.
  - **Policy-based methods** (e.g., REINFORCE, Trust Region Policy Optimization TRPO, PPO): These directly optimize the policy π.

Fitted Q-learning

- Replace the action values $q[\mathbf{s}_t, a_t]$ by a ML model $q[\mathbf{s}_t, a_t, \boldsymbol{\phi}]$.
- Loss function which measures consistency of adjacent action values:

$$L[\boldsymbol{\phi}] = \left(r[\mathbf{s}_t, a_t] + \gamma \cdot \max_a \Big[q[\mathbf{s}_{t+1}, a, \boldsymbol{\phi}] \Big] - q[\mathbf{s}_t, a_t, \boldsymbol{\phi}] \right)^2,$$

target (what q should be)

prediction

- This is the same method as we did for Q-learning. Instead of updating a table of Q values, we minimize a loss that pushes the networks output towards the target.

Fitted Q-learning loss function details

$$L[\phi] = \left(r[s_t, a_t] + \gamma \cdot \max_a q[s_{t+1}, a, \phi] - q[s_t, a_t, \phi]\right)^2$$

$q[s_t, a_t, \phi]$:

- This is the **current Q-value estimate** for taking action a_t in state s_t, parameterized by ϕ (e.g., weights of a neural network).
- It is the model's current prediction of how good this action is.

$r[s_t, a_t]$:

- The **reward** received after taking action a_t in state s_t.
- This is provided by the environment.

$\gamma \cdot \max_a q[s_{t+1}, a, \phi]$:

- This represents the **discounted maximum future Q-value**.
- The term $\max_a q[s_{t+1}, a, \phi]$ finds the **best future Q-value** by selecting the action that maximizes q at the next state s_{t+1}.
- γ (discount factor) determines how much future rewards are valued compared to immediate rewards.

Target for Q-learning Update:

- The term inside the parentheses,

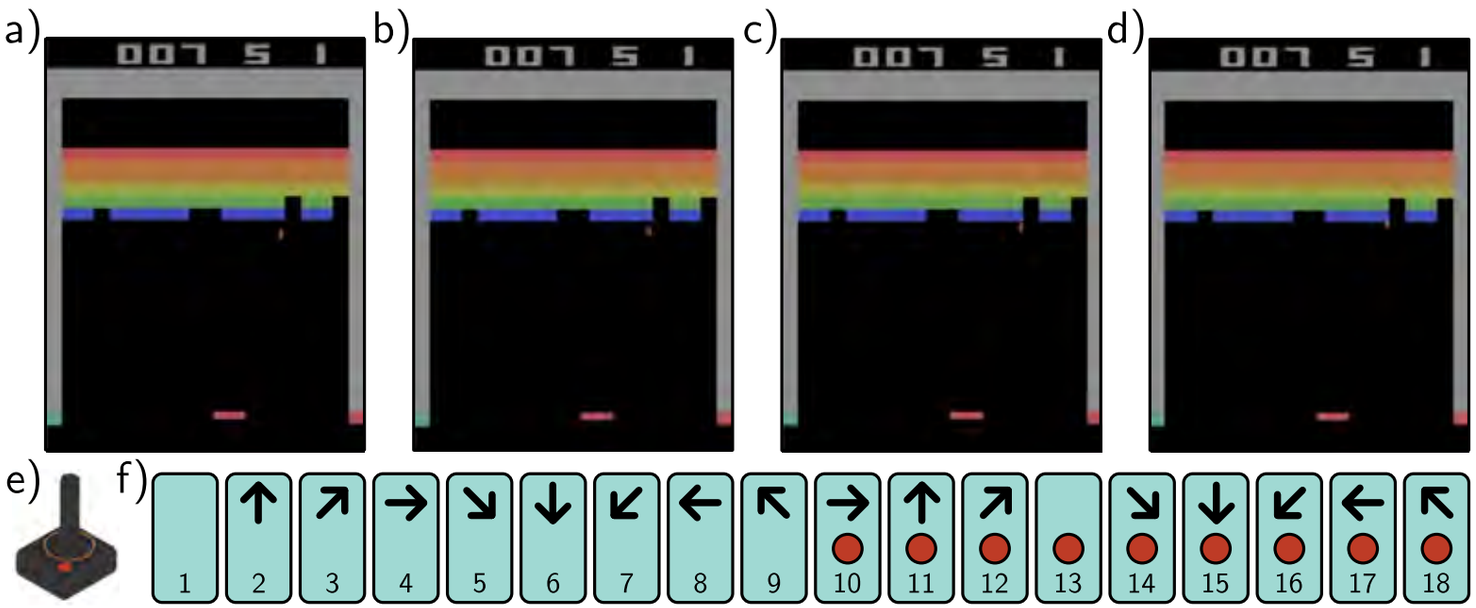
$$r[s_t, a_t] + \gamma \cdot \max_a q[s_{t+1}, a, \phi]$$

  is the **target Q-value** (what we want the model to predict).

- The function minimizes the difference between:
  - The estimated Q-value $q[s_t, a_t, \phi]$.
  - The target value $r + \gamma \max_a q[s_{t+1}, a, \phi]$.
- By **training a neural network** using gradient descent, the parameters ϕ are updated to **better predict future rewards**.

Deep Q-networks

- Use deep NN for fitted Q-learning. Q stands for action-value $q[s_t, a_t, \phi]$.
- **Deep Q-network** was a RL architecture that exploited deep NN to learn to play ATARI 2600 games.



Single frame does not specify velocity ⇒ 4 adjacent frames to represent a state

18 possible actions
(9 directions, on/off)

Deep Q-networks

- The data comprises 220 x 160 images with 128 possible colors at each pixel. Reshaped to 84 x 84 and only brightness value was kept.

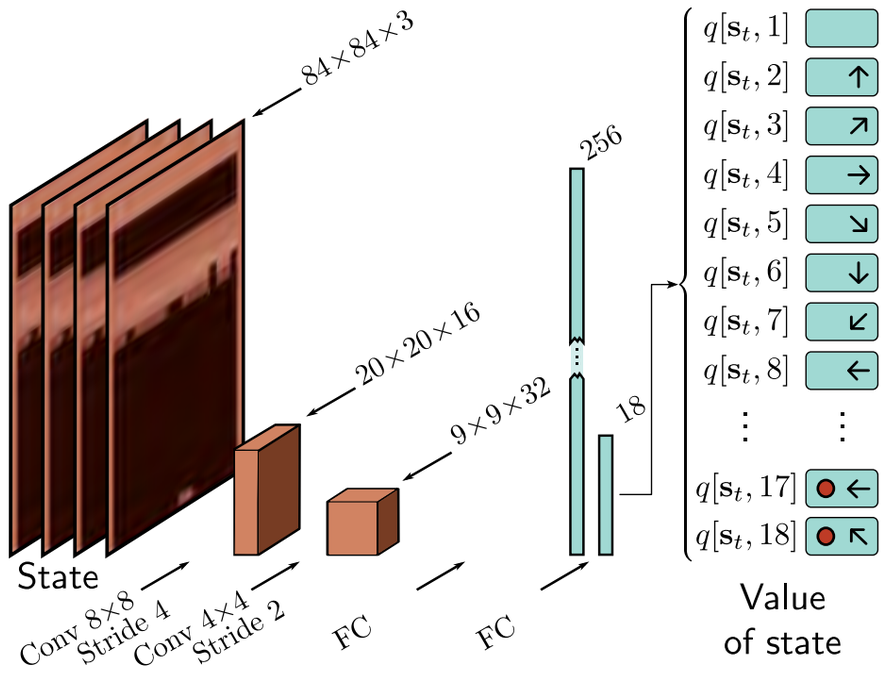


Figure 19.14 Deep Q-network architecture. The input $\mathbf{s}_t$ consists of four adjacent frames of the ATARI game. Each is resized to 84×84 and converted to grayscale. These frames are represented as four channels and processed by an 8×8 convolution with stride four, followed by a 4×4 convolution with stride 2, followed by two fully connected layers. The final output predicts the action value $q[\mathbf{s}_t, a_t]$ for each of the 18 actions in this state.

Rewards ± 1 instead of raw scores of different games, can keep the same learning rate.

Experience replay: store recent states, action, and rewards in a buffer, reuses data samples many times. Helps with training stability.

Reinforcement Learning

Policy-based RL with Neural Networks

This section is mostly based on https://rlhfbook.com/

Policy Gradient Methods

- Recall the notions of value estimation vs policy estimation. Q-learning is an example of value estimation: estimate $q[s_t, a_t, \phi]$ and update π.
- **Policy-based methods** directly learn a stochastic policy $\pi[a_t \mid s_t, \theta]$.
- For MDP, there is always an optimal deterministic policy in principle.
- However there are three reasons to use a stochastic policy:
  - **Exploration of the action-state space:** not obliged to take the best action at each step.
  - **Loss function changes smoothly:** can use gradient descent.
  - **Knowledge of the state is often incomplete:** two locations may look locally the same but nearby reward structure is different. Stochastic policy: taking different actions until ambiguity resolved.

Value based vs policy based methods

Summary: When to Choose What?

Scenario	Value-Based RL (Q-learning, DQN)	Policy-Based RL (REINFORCE, PPO)
Discrete action space	✅ Best choice	❌ Less efficient
Continuous action space	❌ Doesn't work well	✅ Best choice
Large state space	❌ Struggles	✅ Works well
Sample efficiency needed	✅ More efficient	❌ Requires more samples
Stochastic environments	❌ Not ideal	✅ Can learn stochastic policies
Real-time decision making	✅ Fast lookups	❌ Slower (sampling needed)
Stability in training	❌ Can diverge	✅ More stable

Source of this table: GPT-4o (apologies)

“Vanilla” Policy Gradient

- We want to find the optimal policy by maximizing the expected future reward:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) G_t \right] \qquad G_t = \sum_{t'=t}^{T} r_{t'}\text{: return from time } t$$

This equation tells us how to **move the policy** to improve expected reward.

- Each action's log-probability is nudged up or down depending on the **return** G_t that follows it.
  - If G_t is high → increase log-prob of action a_t
  - If G_t is low → decrease it

- We are summing over all steps t in the trajectory/episode of length T.
- The **sum over** t ensures that **every action** a_t taken in the trajectory gets updated.
- The G_t tells us **how good that action turned out to be**, based on what happened *after* time t.

“Vanilla” Policy Gradient

- We want to find the optimal policy by maximizing the expected future reward:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \,|\, s_t) G_t \right] \qquad G_t = \sum_{t'=t}^{T} r_{t'} \text{: return from time } t$$

Here the expectation value is over trajectories drawn according to the policy.

- How to estimate the expectation value? Use Monte Carlo over N trajectories. This algorithm is called REINFORCE.

$$\mathcal{L}(\theta) \approx -\frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T^{(i)}} \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \cdot G_t^{(i)}$$

Subtracting a baseline

- A common problem with vanilla policy gradient algorithms is the high variance in gradient updates, which can be mitigated in multiple ways. In order to alleviate this, various techniques are used to normalize the value estimation, called baselines.

- We then have:

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{\tau\sim\pi_\theta}\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\,(G_t - b)\Big]$$

- A simple baseline is the **average return per trajectory**, across a batch of sampled episodes:

$$b = \frac{1}{N}\sum_{i=1}^{N} G_0^{(i)}$$

Value network

- A better baseline is to subtract an estimate of the value function. We now need an additional neural network, the **value network**, which estimates it. This is an example of Actor-Critic RL, where the critic is the value network. The value network also gets trained, with a different loss function.

- The policy gradient is then:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_\tau \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot (G_t - V_\phi(s_t)) \right]$$

- The quantity $A_t = G_t - V_\phi(s_t)$ is called the advantage.

- We now have two neural networks:

$\pi_\theta(a \mid s)$ is the **policy network**

$V_\phi(s)$ is the **value network**

PPO

- PPO improves over standard REINFORCE with a value network baseline by using a more stable, constrained update rule with mini-batch training, reducing variance while maintaining sample efficiency.

- For practical tasks, PPO is usually preferred over REINFORCE due to its stability and efficiency.

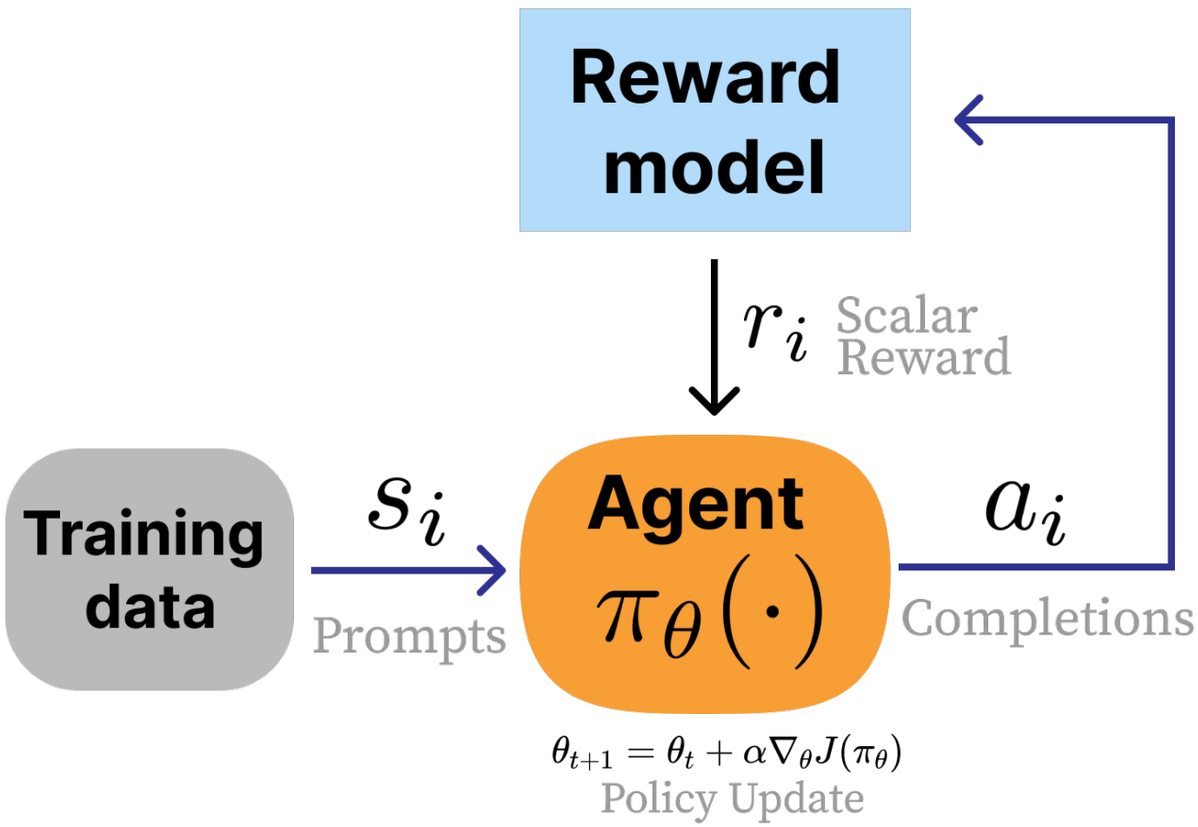
- Discussing this in detail goes too far. For completeness, the training objective is:

$$J(\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}A, \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1-\varepsilon, 1+\varepsilon\right)A\right)$$

RLHF of LLMs often uses PPO

- To fine-tune LLMs with RLHF (Reinforcement Learning with Human Feedback), people often use PPO. **The actions are now the prompt completions** (i.e. the answers the model writes to the queries).

- But what is the reward? The common approach is to train a "reward model" where humans compare different answers and compare them (**"preference learning"**). The reward model then learns to "rate like a human", and can then be used for training the LLM.



Source: rlhfbook.com

GRPO

- A new policy RL algorithm called **Group Relative Policy Optimization (GRPO)** is getting very popular, due to its success in training the Deepseek Reasoning model.
- It works similar to the PPO, but it does **NOT not require a value network**. This makes the approach **simpler**.
- The advantage A in this algorithm is calculated by **comparing multiple answers to a single question ("group relative").**
- For completeness, the training objective is

$$J(\theta) = \frac{1}{G}\sum_{i=1}^{G}\left(\min\left(\frac{\pi_\theta(a_i|s)}{\pi_{\theta_{old}}(a_i|s)}A_i, \text{clip}\left(\frac{\pi_\theta(a_i|s)}{\pi_{\theta_{old}}(a_i|s)}, 1-\varepsilon, 1+\varepsilon\right)A_i\right) - \beta D_{KL}(\pi_\theta||\pi_{ref})\right)$$

- The advantage is estimated from the responses r as

$$A_i = \frac{r_i - \text{mean}(r_1, r_2, \cdots, r_G)}{\text{std}(r_1, r_2, \cdots, r_G)}$$

Direct Preference Optimization (DPO)

- There is one more algorithm that you should have heard about when it comes to LLM fine-tuning with reinforcement learning.
- **Direct Alignment Algorithms (DAAs)** allow one to update models to solve the same RLHF objective without ever training an intermediate reward model or using reinforcement learning optimizers.
- The most prominent DAA and one that catalyzed an entire academic movement of aligning language models is **Direct Preference Optimization (DPO)**. Paper: https://arxiv.org/abs/2305.18290
- It allows to solve the standard RLHF problem with only a simple classification loss.

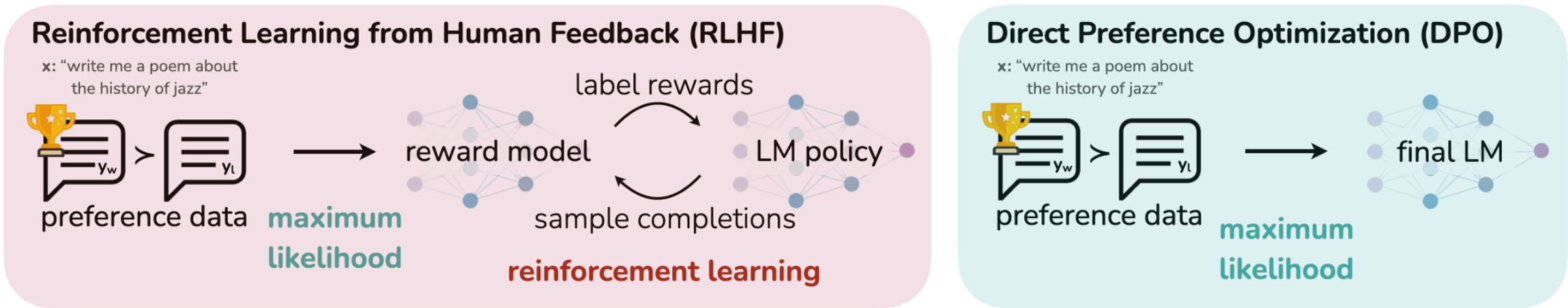


Figure 1: **DPO optimizes for human preferences while avoiding reinforcement learning.** Existing methods for fine-tuning language models with human feedback first fit a reward model to a dataset of prompts and human preferences over pairs of responses, and then use RL to find a policy that maximizes the learned reward. In contrast, DPO directly optimizes for the policy best satisfying the preferences with a simple classification objective, fitting an *implicit* reward model whose corresponding optimal policy can be extracted in closed form.

https://anukriti-ranjan.medium.com/preference-tuning-llms-ppo-dpo-grpo-a-simple-guide-135765c87090

Feature	PPO (RLHF)	DPO (Direct Preference Optimization)	GRPO (Group Relative Policy Optimization)
Reward Model	Explicitly trained neural network	Implicitly learned	Implicitly learned
Value Function (Critic)	Yes, separate neural network	No	No
Optimization	Reinforcement Learning (PPO Algorithm)	Direct Policy Optimization (Classification-like Loss)	Simplified RL (PPO-like, but with GRAE)
Training Loop Complexity	More complex (RL loop, critic training)	Simpler (no RL loop, direct optimization)	Simplified RL loop (group sampling, but no critic)
Advantage Estimation	GAE (using Value Function)	N/A (Implicit Reward)	GRAE (Group Relative Advantage Estimation)
Computational Efficiency	Lower (critic, RL sampling)	Higher (no critic, no RL sampling during training)	Higher than PPO (no critic), potentially similar to DPO
Implementation Simplicity	More complex	Simpler	Simpler than PPO, somewhat similar to DPO

Reinforcement Learning

Combining RL and Search

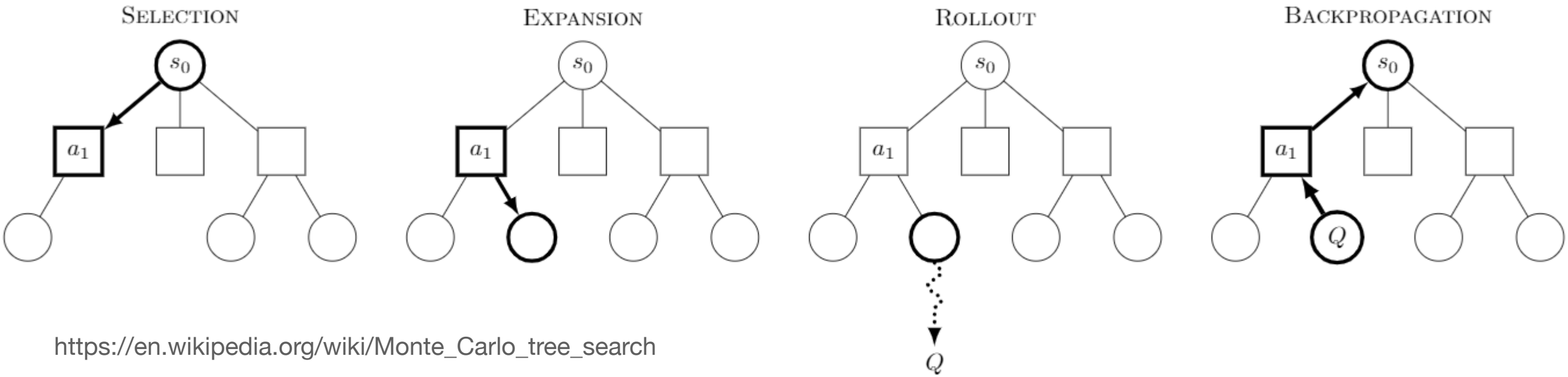
Combining RL and Search

- If you can simulate the environment and rewards are sparse, a powerful technique is to **combine RL techniques with search methods**.
- In particular, **AlphaZero and MuZero, both developed by DeepMind**, are powerful examples of reinforcement learning (RL) combined with search, specifically **Monte Carlo Tree Search (MCTS)**.
- **Alpha Zero**
  - It trains a neural network using self-play; **the network learns a policy (what move to make)** and **a value function (how good the position is)**.
  - It uses Search (MCTS) during both training and evaluation to plan moves.
  - **Think of it as an RL+search loop: the neural net helps the search, and the search helps improve the neural net.**
- **In MuZero**, in addition the environment is also learned by a neural network. E.g. it learns to play chess without having the rules for chess coded in.

Monte Carlo Tree Search (MCTS)

- Monte Carlo Tree Search is a decision-making algorithm used for planning, especially in games like Go, chess, or general AI problems.
- It's great when:
  - The **search space is huge**.
  - You d**on’t have a perfect evaluation function** (e.g. chess space is too large to be able to perfectly evaluate every position).
  - You can **simulate how things might play out** (even randomly).
- MCTS **builds a search tree incrementally and uses random simulations (Monte Carlo rollouts) to guide which parts of the tree to explore**.

Monte Carlo Tree Search



https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Q refers to the **reward** (or estimated value) obtained from a simulation that starts from a new node and proceeds until the end of the episode (or a depth cutoff).

- We build a tree of actions iteratively. Each iteration of MCTS has four phases:
  - **Selection**: Navigate from the root to a leaf using a tree policy.
  - **Expansion**: If the leaf node is not terminal, expand one or more children.
  - **Simulation**: Simulate a random playout from the new node.
  - **Backpropagation**: Propagate the result back up to update the stats of all nodes in the path.

Alpha Zero, Mu Zero

- Alpha Zero press release:
  - https://deepmind.google/discover/blog/alphazero-shedding-new-light-on-chess-shogi-and-go/
- Alpha Zero paper:
  - https://www.science.org/doi/full/10.1126/science.aar6404
- Mu Zero paper:
  - https://www.nature.com/articles/s41586-020-03051-4.epdf?sharing_token=kTk-xTZpQOF8Ym8nTQK6EdRgN0jAjWel9jnR3ZoTv0PMSWGj38iNIyNOw_ooNp2BvzZ4nIcedo7GEXD7UmLqb0M_V_fop31mMY9VBBLNmGbm0K9jETKkZnJ9SgJ8Rwhp3ySvLuTcUr888puIYbngQ0fiMf45ZGDAQ7fUI66-u7Y%3D
- Mu Zero press release:
  - https://deepmind.google/discover/blog/muzero-mastering-go-chess-shogi-and-atari-without-rules/
  - Mu Zero is a powerful general approach to learn to play games and other environments.

Course logistics

- **Reading for this lecture:**
  - This lecture was based in part on the book by Prince, linked on the website.
  - We also used rlhfbook.com