Introduction to Reinforcement Learning

Lecture 2: Frontiers of RL

Joshua Evans

Bath Reinforcement Learning Laboratory

Department of Computer Science

UNIVERSITY OF BATH

Dynamic Programming

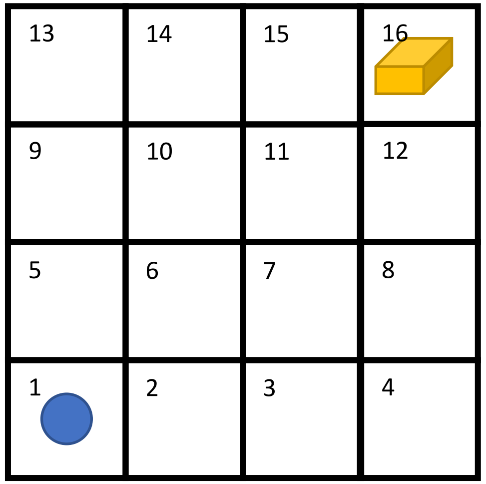
Monte Carlo Methods

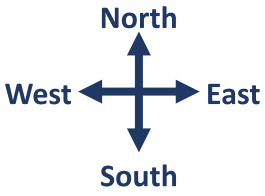
Temporal Difference Methods

Learn $V(s)$ or $Q(s, a)$, then derive a policy.

Tabular Methods

- So far, our value functions could all be represented as tables.
  - For $V(s)$, one entry per state.
  - For $Q(s,a)$, one entry per state-action pair.





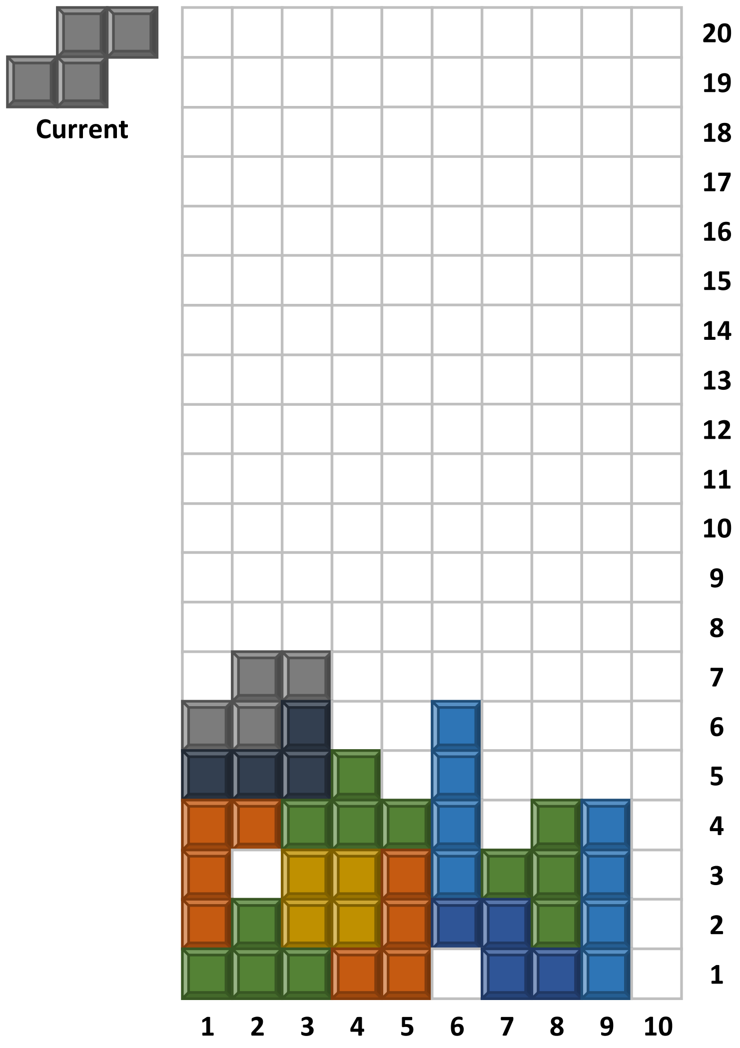
State	Action	$\mathrm{Q(s,a)}$
1	N	$Q(0,\mathrm{N})$
1	S	$Q(0,S)$
1	E	$Q(0,E)$
1	W	$Q(0,W)$
…	…	…
16	W	$Q(24,W)$

16 states $\times$ 4 actions $\rightarrow$ 64 state-action pairs

- We need to store a value in memory for each state-action pair.
- We need to experience a state-action pair to learn about it.
- Feasible here, but what about larger problems?

Larger State-Spaces

- Tetris
  - 10×20 grid, 7 current tetriminos.
  - Upper bound of 7×2^{200} states.
- Not feasible to learn to play Tetris using tabular methods.
  - Can't fit the Q-table in memory.
  - Cannot reasonably experience all state-action pairs.

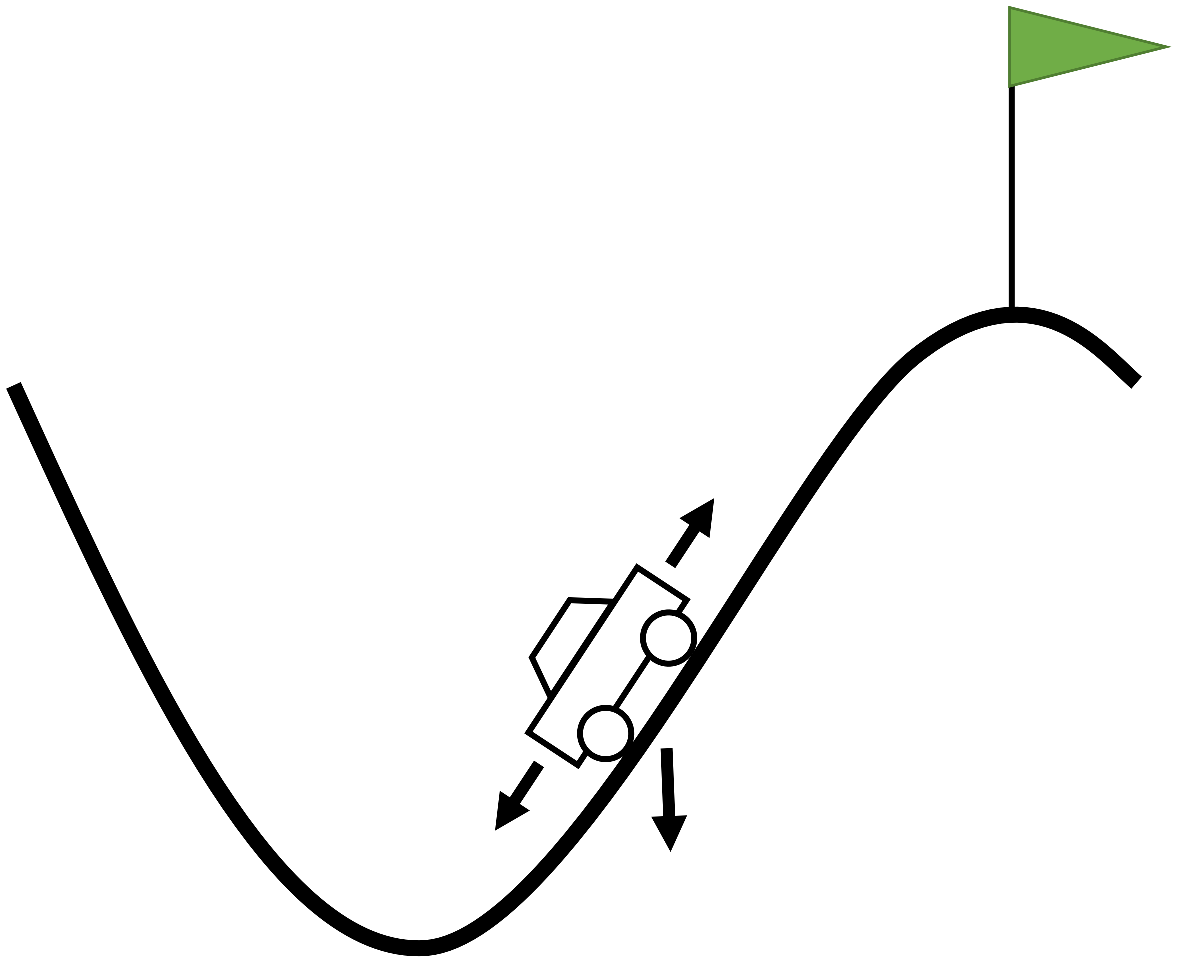


Larger State-Spaces

- Mountain Car Problem
  - Continuous State-Space
  - Position ($-1.2 < x < 0.5$)
  - Velocity ($-0.07 < v < 0.07$)

- Actions:
  - Full Throttle Forward
  - Full Throttle Backwards
  - Zero Throttle

- Infinite number of states!



Mountain Car Problem
The underpowered car must reach the top of the hill as quickly as possible.

Generalisation

- We can only reasonably expect to experience a small subset of the entire state-action-space.
- Can we **generalise** what we know about states we have visited to states we haven't visited?
- Can we do this in a way which avoids storing information about every state separately?

Yes, using **function approximation**!

Function Approximation

- Function Approximated Methods
  - Represent $v_\pi(s)$ as a real-valued function with parameter vector $\boldsymbol{\theta}$.
  - We only store and update $\boldsymbol{\theta}$.

$$\hat{v}(s, \boldsymbol{\theta}) \approx v_\pi(s)$$

- Function could be a linear function, polynomial, **neural network**…

- Learn a function with parameters $\boldsymbol{\theta}$ which maps states to values.
  - How can we go about doing this?

Approximating Value Functions

State-Values

$$s \longmapsto v$$

- Dynamic Programming

$$s \longmapsto E_\pi\{R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{\theta_t}) | S_t = s\}$$

- Monte-Carlo

$$s \longmapsto G_t$$

- Temporal-Difference

$$s \longmapsto R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{\theta_t})$$

Action-Values

$$s, a \longmapsto q$$

- Dynamic Programming

$$s, a \longmapsto E_\pi\{\ldots | S_t = s, A_t = a\}$$

- Monte-Carlo

$$s, a \longmapsto G_t$$

- Temporal-Difference

$$s, a \longmapsto R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{\theta_t})$$

We can use these as training examples in
a supervised learning context!

Approximating Value Functions

- Our goal is to train a function to correctly map inputs to outputs.
    - We do this all the time in supervised learning!

$f($  $, \boldsymbol{\theta}) = 0$

$f($  $, \boldsymbol{\theta}) = 1$

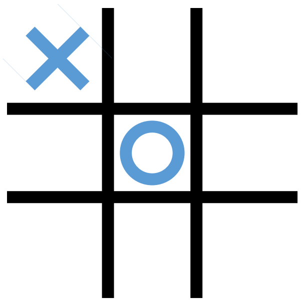
Change $\boldsymbol{\theta}$ so that f maps images to classes correctly.

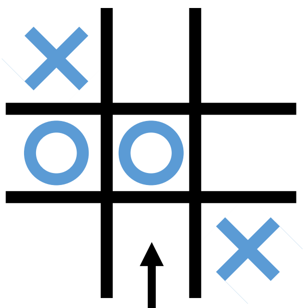
Class 0: Cat
Class 1: Dog
Image Credit: Wikimedia Commons

Approximating Value Functions

- Our goal is to train a function to correctly map inputs to outputs.
  - We do this all the time in supervised learning!
  - Here, our inputs are states, and our outputs are values.

Value Estimate (e.g. Monte Carlo sample return)

$\hat{v}($  $, \boldsymbol{\theta}) = G_1$

$\hat{v}($  $, \boldsymbol{\theta}) = G_2$

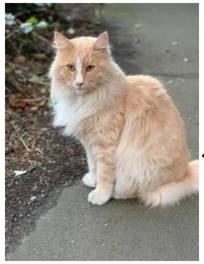
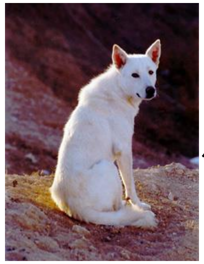
State Input

Change $\boldsymbol{\theta}$ so that $\hat{v}$ maps states to values correctly.

Approximating Value Functions

- We can measure how well our function is mapping inputs to outputs using objective functions.

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}[y_i - \hat{y}_i]^2$$

$f($  $, \boldsymbol{\theta}) = 0$ $f($  $, \boldsymbol{\theta}) = 1$ $f($  $, \boldsymbol{\theta}) = 1$

Class 0: Cat
Class 1: Dog
Image Credit: Wikimedia Commons

Approximating Value Functions

- We can measure how well our function is mapping inputs to outputs using objective functions.

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}[\underbrace{y_i}_{\text{True Output}} - \underbrace{\hat{y}_i}_{\text{Predicted Output}}]^2$$

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}[(\underbrace{v_\pi(S_t)}_{\text{New Value Estimate}} - \underbrace{\hat{v}(S_t, \boldsymbol{\theta}_t)}_{\text{Existing Value Estimate}})]^2$$

Approximating Value Functions

- We can measure how well our function is mapping inputs to outputs using objective functions.

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}[\underbrace{y_i}_{\text{True Output}} - \underbrace{\hat{y}_i}_{\text{Predicted Output}}]^2$$

$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}[\underbrace{\boxed{R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{\theta}_t)}}_{\text{TD Target}} - \underbrace{\hat{v}(S_t, \boldsymbol{\theta}_t)}_{\text{Existing Value Estimate}}]^2$$

Approximating Value Functions

- How do we know how to change $\boldsymbol{\theta}$ in order to lower our loss?

- Calculate the gradient of our loss function with respect to each of the parameters in $\boldsymbol{\theta}$.

- Take a step in the direction which lowers the loss.

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \mathrm{MSE}$$

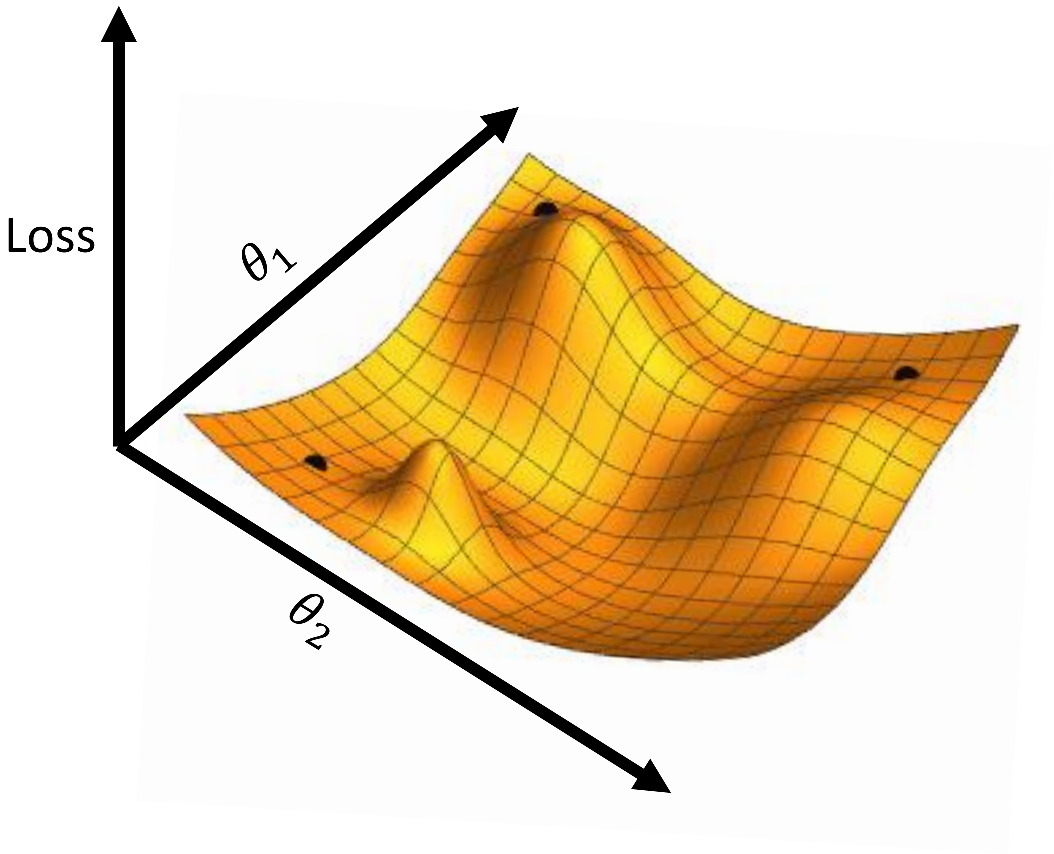


Image Credit: [Wikimedia Commons]

Function Approximation

- Function Approximated Methods
  - Represent $v_\pi(s)$ as a real-valued function with parameter vector $\boldsymbol{\theta}$.
  - We only store and update $\boldsymbol{\theta}$.

$$\hat{v}(s, \boldsymbol{\theta}) \approx v_\pi(s)$$

Deep RL

- Function could be a linear function, polynomial, **neural network…**

- Learn a function with parameters $\boldsymbol{\theta}$ which maps states to values.
  - How can we go about doing this?

Tesauro, G., 1995. Temporal difference learning and TD-Gammon. Communications of the ACM, 38(3), pp.58-68. [URL]

A Brief History of Deep RL

- 1992 – Gerald Tesauro's TD-Gammon

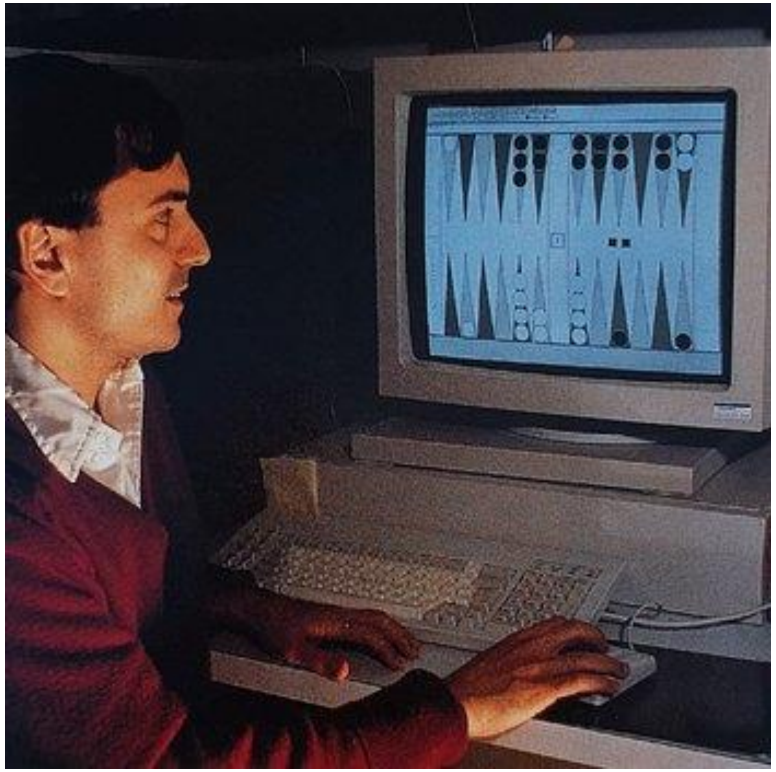


Image Credit: IBM

Hafner, R. and Riedmiller, M., 2007, April. Neural reinforcement learning controllers for a real robot application. In Proceedings 2007 IEEE International Conference on Robotics and Automation (pp. 2098-2103). IEEE. [URL]

A Brief History of Deep RL

- 2009 – Riedmiller et al. Robocup

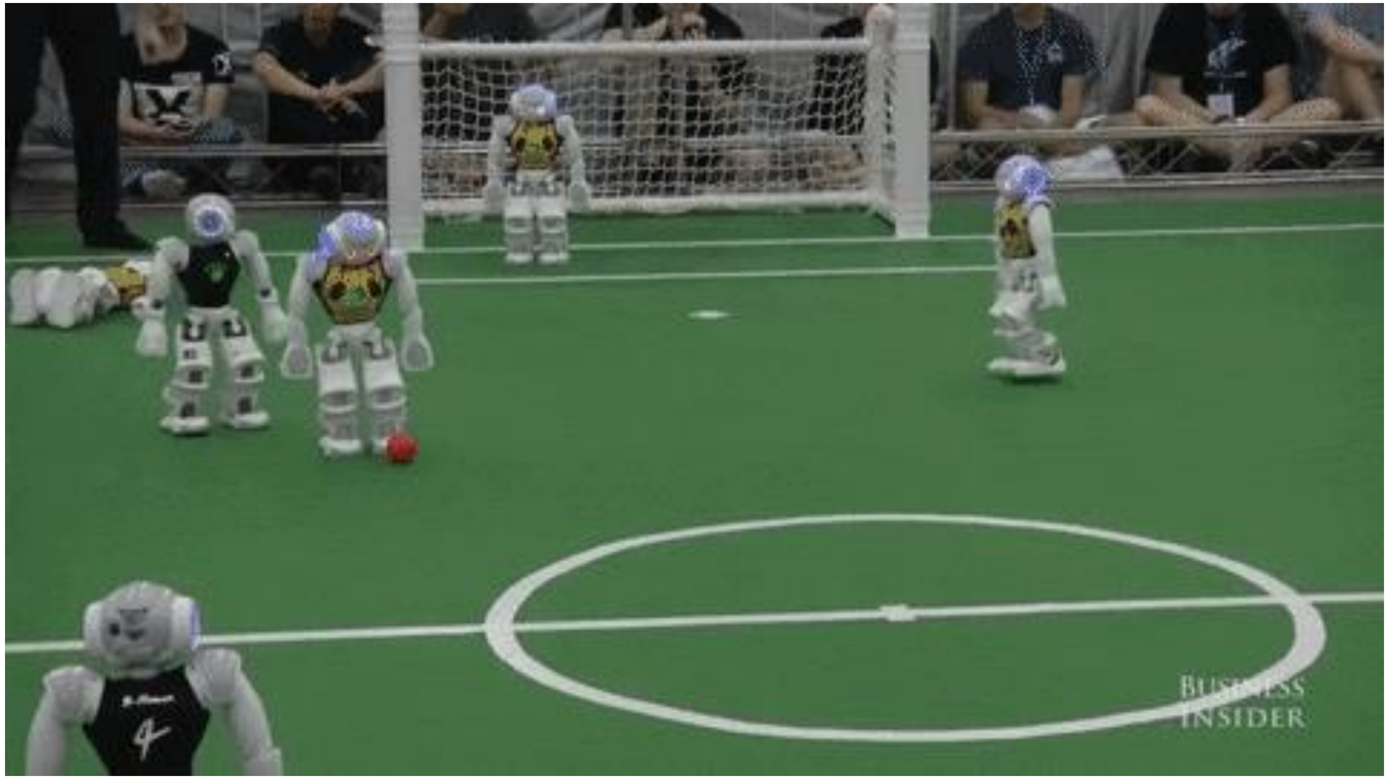


Image Credit: https://www.robocup.org/

Hafner, R. and Riedmiller, M., 2007, April. Neural reinforcement learning controllers for a real robot application. In Proceedings 2007 IEEE International Conference on Robotics and Automation (pp. 2098-2103). IEEE. [URL]

A Brief History of Deep RL

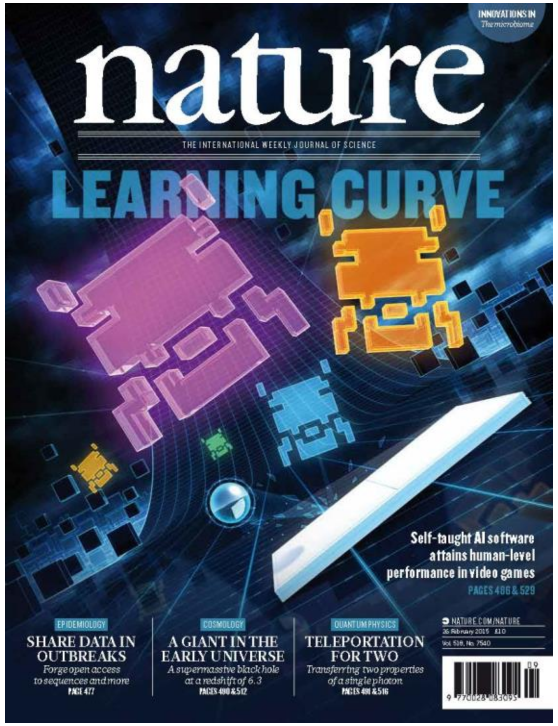
- 2009 – Riedmiller et al. Robocup

Image Credit: https://www.robocup.org/

Mnih et al., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602. [URL]
Mnih et al., 2015. Human-level control through deep reinforcement learning. *nature*, *518*(7540), pp.529-533. [URL]

A Brief History of Deep RL

- 2013-2015 – DeepMind train agents to play Atari games.



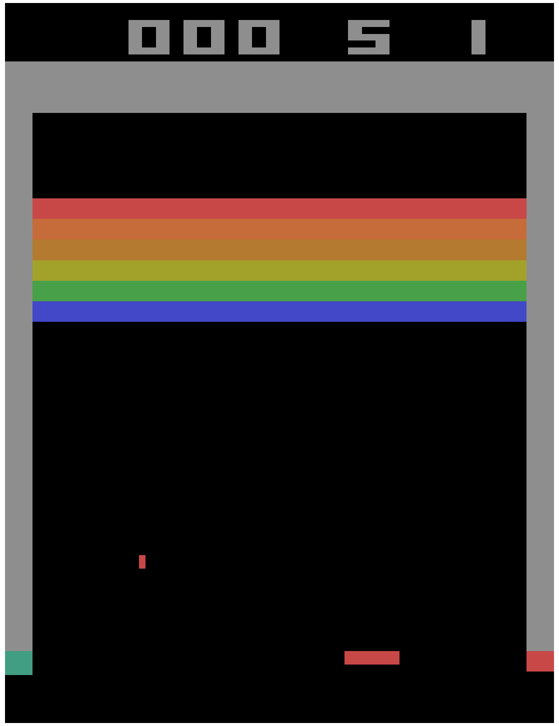


Image Credit: Nature & DeepMind

Silver et al., 2016. Mastering the game of Go with deep neural networks and tree search. *nature*, *529*(7587), pp.484-489. [URL]

A Brief History of Deep RL

- 2016 – DeepMind train agents to beat world champions at Go.

Image Credit: Nature & DeepMind

More Recent Results

- 2017-2019 – OpenAI Five (DotA 2 Agents)

Image Credit: OpenAI

OpenAI Five Benchmark Match Replay

More Recent Results

- 2019 – DeepMind's AlphaStar (StarCraft II Agent)

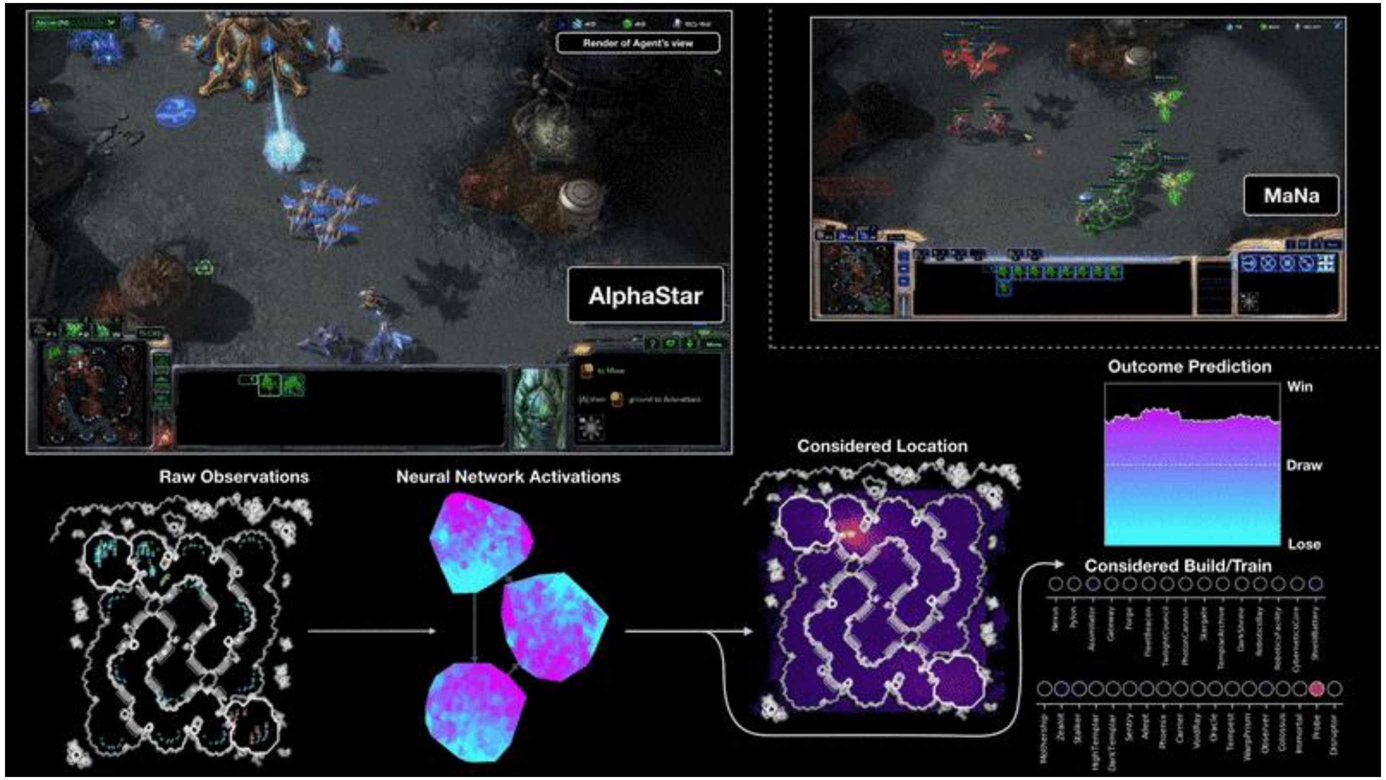
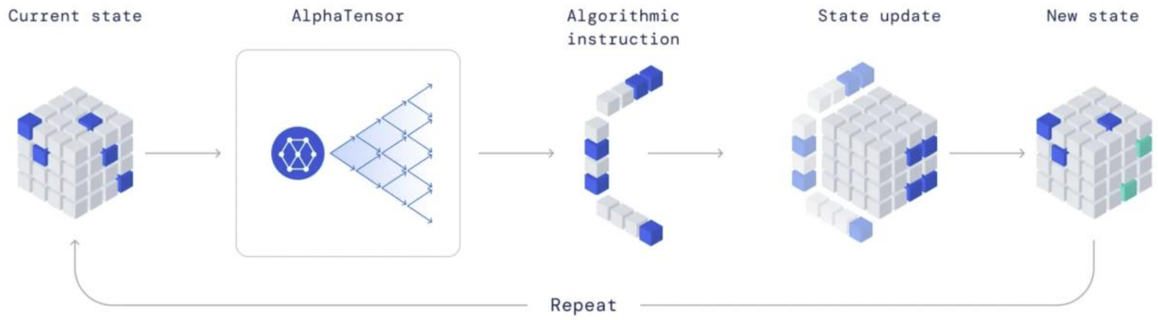


Image Credit: DeepMind

Cutting-Edge Results

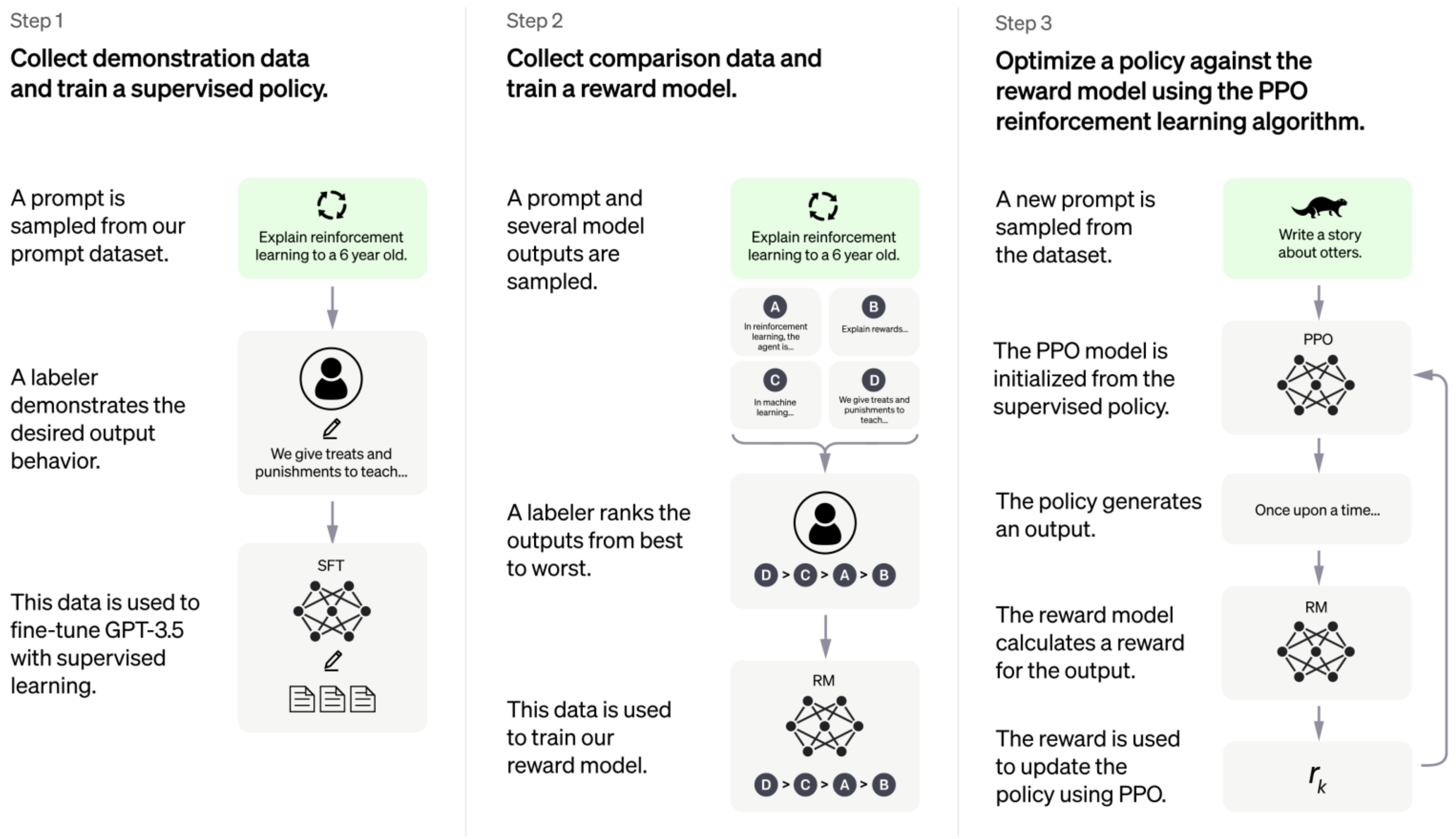
AlphaTensor (2022)
Discovery of novel, efficient, and provably-correct algorithms.



Autonomous Driving (Present)
Using deep reinforcement learning to help guide autonomous vehicles.

Image Credit: [DeepMind]() and [ML4AD@NeurIPS]()

The Obligatory "RLHF" Slide

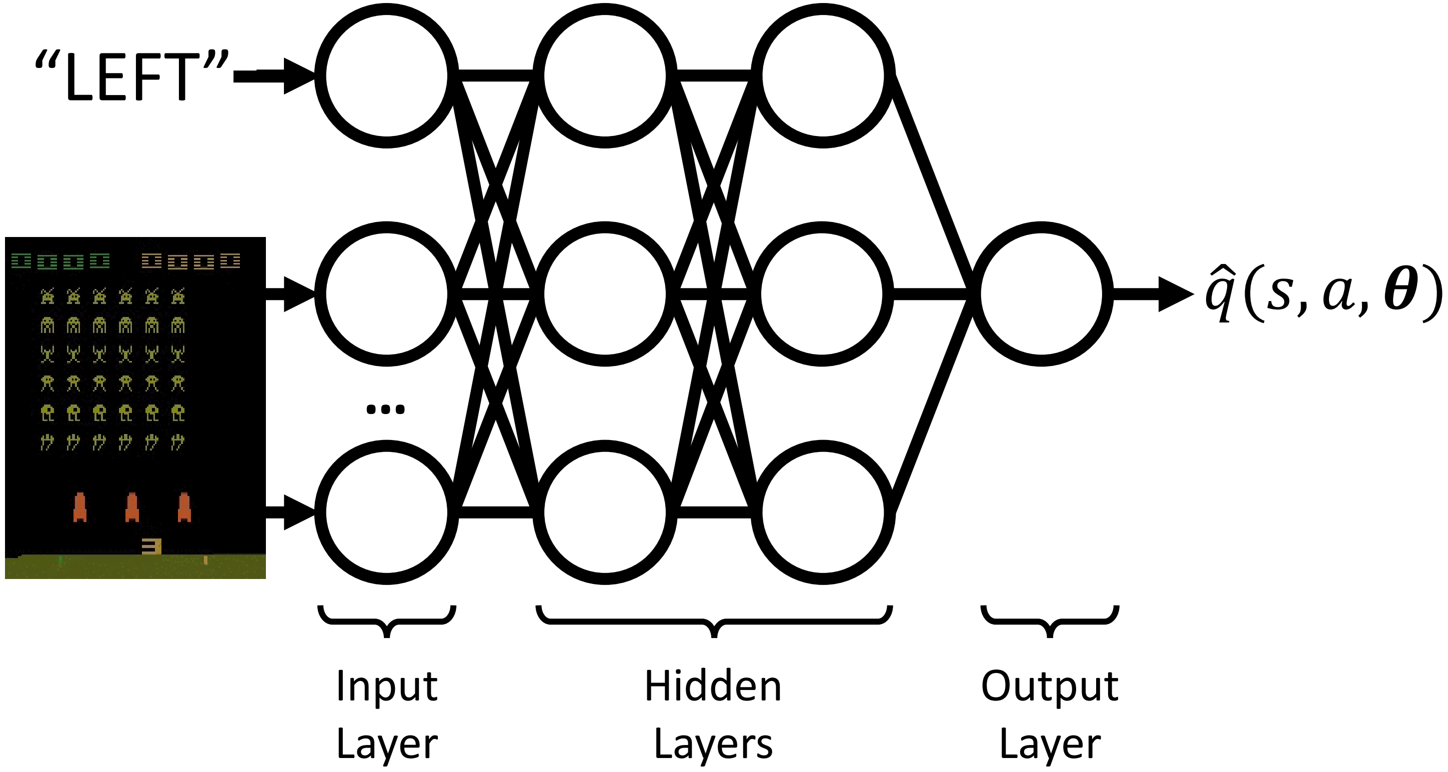


Reinforcement Learning from Human Feedback (Present)

Use human preferences to train a reward model. Then, use that reward model to fine-tune a GPT model with reinforcement learning methods.

Image Credit: [OpenAI]

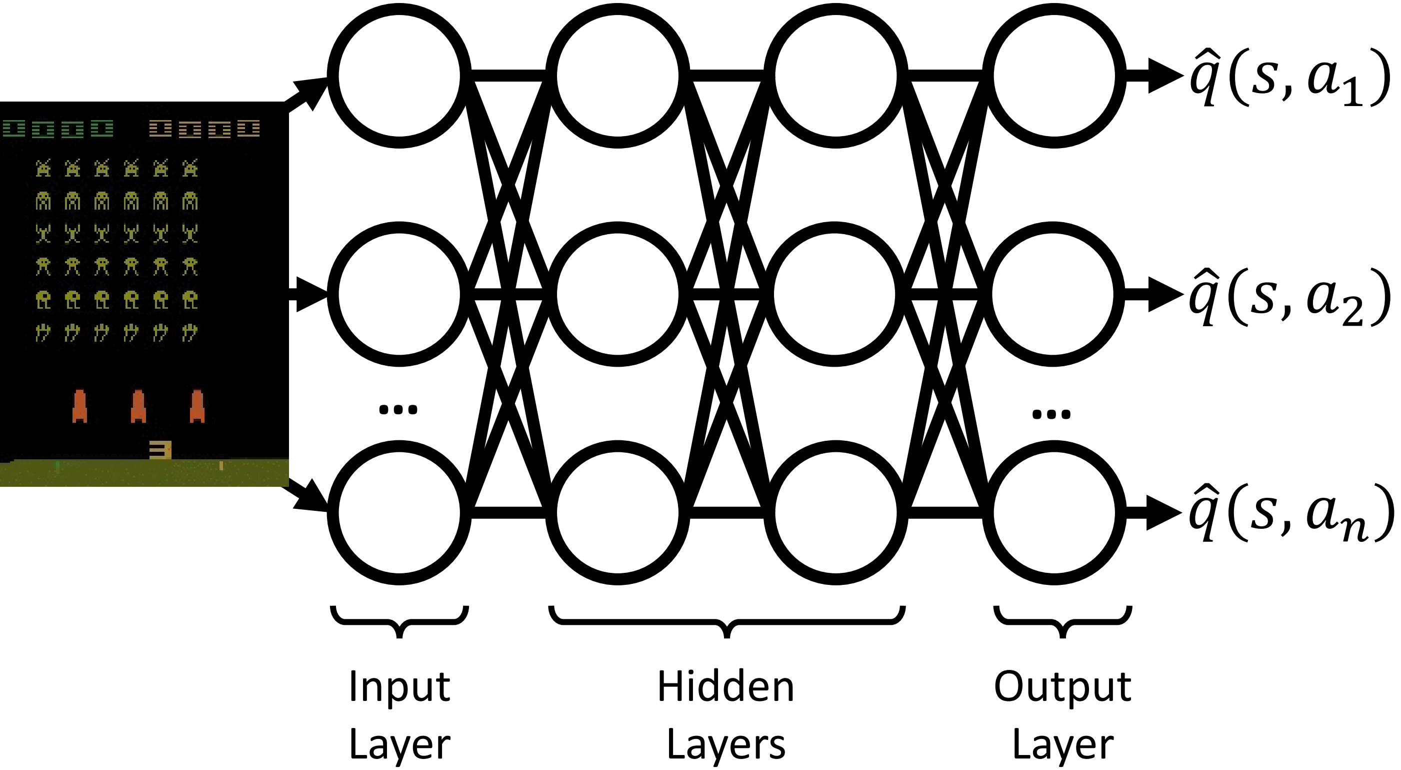
Deep Q-Networks (DQN)



What should our neural network look like?

- We could have state-action pairs as input, and a single actin-value as output.

Deep Q-Networks (DQN)



What should our neural network look like?

- We could have state-action pairs as input, and a single action-value as output.
- We could have only the state as input, then one output per available action as outputs.

Deep Q-Networks (DQN)

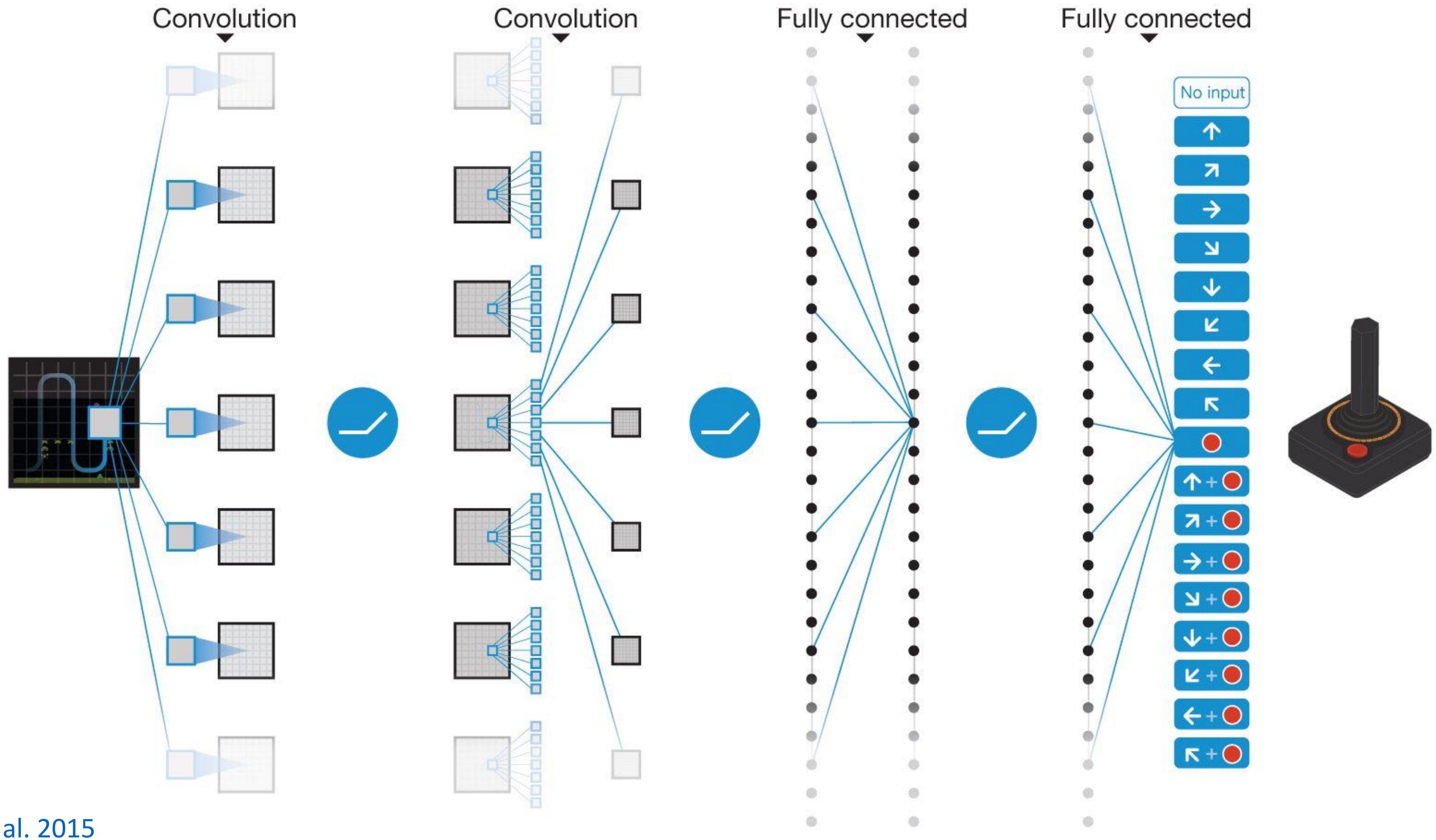


Image Credit: [Minh et al. 2015]()

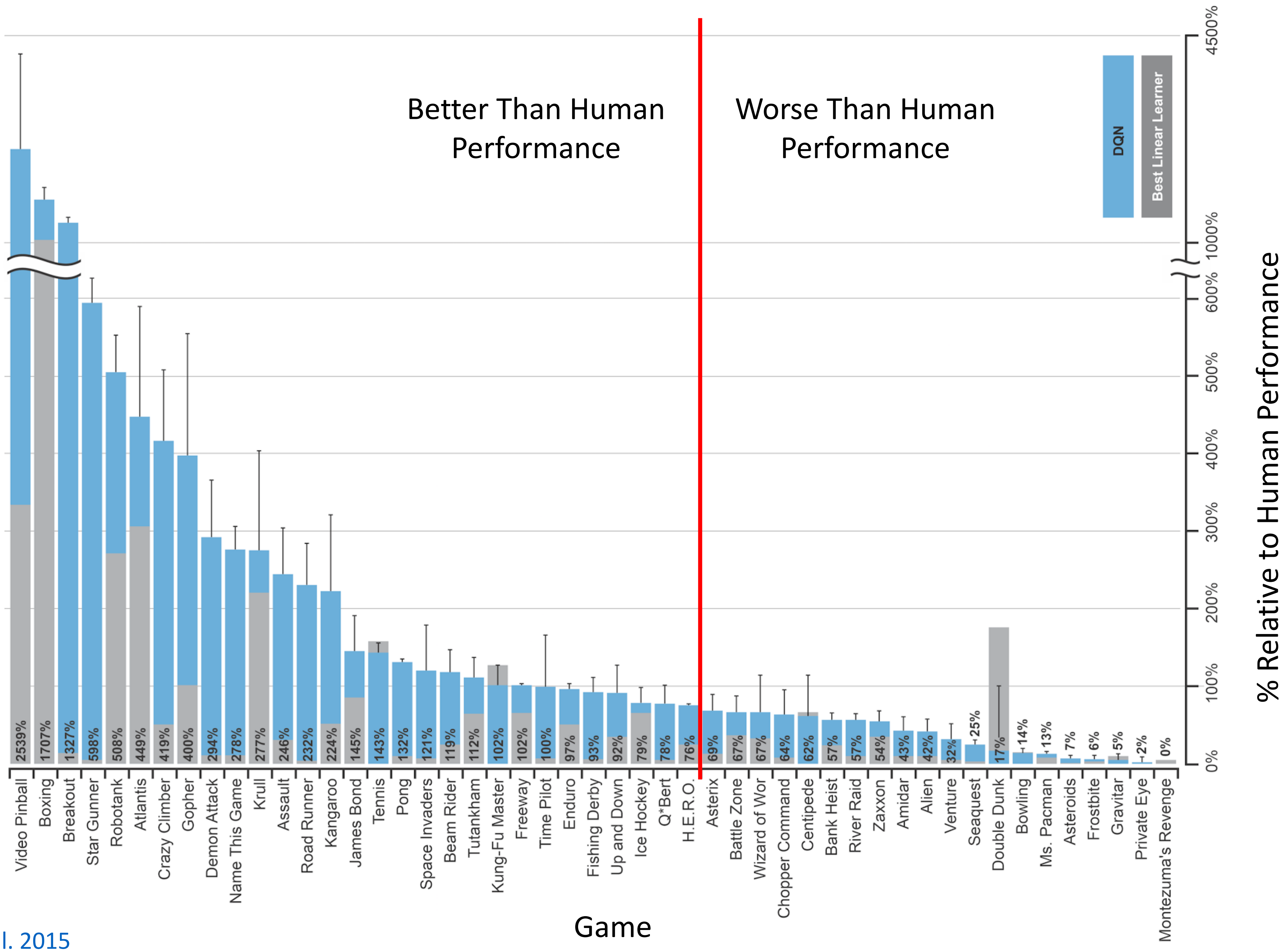


Image Credit: Minh et al. 2015

Algorithm: Ridiculously Naïve Deep Q-Learning

Initialise action-value network $\hat{q}$ with arbitrary weights $\boldsymbol{\theta}$

For episode $= 1, M$ **do**

  Initialise initial state S_1

  For $t = 1, T$ **do**

    With probability ϵ select random action A_t

    With probability $1 - \epsilon$ select action $A_t = \text{argmax}_a \hat{q}(S_t, a, \boldsymbol{\theta})$ ← Forward Pass

    Execute action A_t, observe reward R_t and next state S_{t+1}

Compute TD Target → Set $y_t = \begin{cases} R_t, & \text{if } S_{t+1} \text{ is terminal} \\ R_t + \gamma \max_{a'} \hat{q}(S_{t+1}, A', \boldsymbol{\theta}), & \text{otherwise} \end{cases}$

    Perform gradient descent step $\nabla_{\boldsymbol{\theta}} \big(\underbrace{y_t}_{y} - \underbrace{\hat{q}(S_t, A_t, \boldsymbol{\theta})}_{\hat{y}}\big)^2$ ← Calculate Loss, Calculate Gradients, Backward Pass

  End For

End For

This pseudocode is for illustration purposes only.

<u>Never implement this – it will perform terribly.</u>

Ridiculously Naïve DQN Issues

- Data Highly Temporally Correlated
  - Samples are taken sequentially – distributional shift over time.
- The agent’s policy could change rapidly.
  - Large errors will lead to large gradients and large updates.
  - Even small changes in Q-values may drastically change the policy.
  - This may cause sudden changes to the sampling distribution.
- We’re using the same function to compute both our prediction ($\hat{y}$) and our target (y).
  - Whenever we perform an update, our target will change.
  - We’re chasing a moving target – can lead to unstable training.

DQN Fixes: Experience Replay

- Use **experience replay**.
  - Store a buffer D of experiences of the form $(s_t, a_t, r_{t+1}, s_{t+1})$.
  - Perform updates using a minibatch of experiences from the buffer.
- Decorrelates experiences temporally.
  - Recent experiences are not emphasised during training.
  - Old experiences are not forgotten.
- Requires a large amount of memory.
  - In excess of 1,000,000 recent experiences may need to be stored.
- Extensions exist to prioritise experiences and remove old ones.

DQN Fixes: Reward Clipping/Normalising

- Reward magnitude not known ahead of time.
- Rewards with large magnitudes may cause dramatic weight updates, impacting stability.
- To deal with this, we could clip rewards to lie in the range $[-1,1]$.
- We can also clip the gradients of the loss function each before performing an update.
- We could also track the rewards we observe and normalise them as we go along.

DQN Fixes: Fixed Target Network

- We currently use the same network $\hat{q}(s, a, \boldsymbol{\theta})$ to select actions and compute our target value.
  - As we update our network, the target value will also change.
  - The network is updating towards a moving target!
  - This can cause instability when training, and decrease overall performance.
- To fix this, we can use a **fixed target network**.
  - Select actions using $\hat{q}_1(s, a, \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}_1$.
  - Calculate target values using $\hat{q}_2(s, a, \boldsymbol{\theta}_2)$with parameters $\boldsymbol{\theta}_2$.
  - Every C steps, update $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$.
  - Allows us to perform update towards a (mostly) stationary target.

Algorithm: Deep Q-Learning with Experience Replay and Fixed Target Network

Initialise replay memory D to capacity N ← Initialise Replay Buffer

Initialise action-value network $\hat{q}_1$ with arbitrary weights $\boldsymbol{\theta}_1$

Initialise target action-value network $\hat{q}_2$ with weights $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$ ← Initialise Target Network

For episode $= 1, M$ **do**

  Initialise initial state S_1

  For $t = 1, T$ **do**

    With probability ϵ select random action A_t

    With probability $1 - \epsilon$ select action $A_t = \text{argmax}_a \hat{q}_1(S_t, a, \boldsymbol{\theta}_1)$ ← Select action using $\hat{q}_1$.

    Execute action A_t, observe reward R_{t+1} and next state S_{t+1}

    Store transition $(S_t, A_t, R_{t+1}, S_{t+1})$ in D ← Store Experience in Replay Buffer

    Sample random minibatch of transitions $(S_j, A_j, R_{j+1}, S_{j+1})$ from D

    Compute TD target using $\hat{q}_2$ (target network). → Set $y_j = \begin{cases} R_{j+1} + 0, & \text{if } S_{j+1} \text{ is terminal} \\ R_{j+1} + \gamma \max\limits_{a'} \hat{q}_2(S_{j+1}, a', \boldsymbol{\theta}_2), & \text{otherwise} \end{cases}$

    Perform gradient descent step $\nabla_{\boldsymbol{\theta}} \left(y_j - \hat{q}(S_j, A_j, \boldsymbol{\theta}) \right)^2$

    Update based on experience sampled from replay buffer.

    Every C steps, update $\boldsymbol{\theta}_2 = \boldsymbol{\theta}_1$ ← Periodically update target network.

  End For

End For

N.B. For simplicity, this pseudocode does not use Reward Clipping/Gradient Clipping/Reward Normalisation.

Types of Reinforcement Learning Methods

- **Value-Based** RL Methods
  - Approximate the **optimal action-value function** $Q^*(s, a)$.
- **Policy-Based** RL Methods
  - Directly search the policy-space for the **optimal policy** $\pi^*(a|s)$.

Deep RL methods use deep neural networks to represent the value function or policy.

N.B. these are not mutually exclusive!

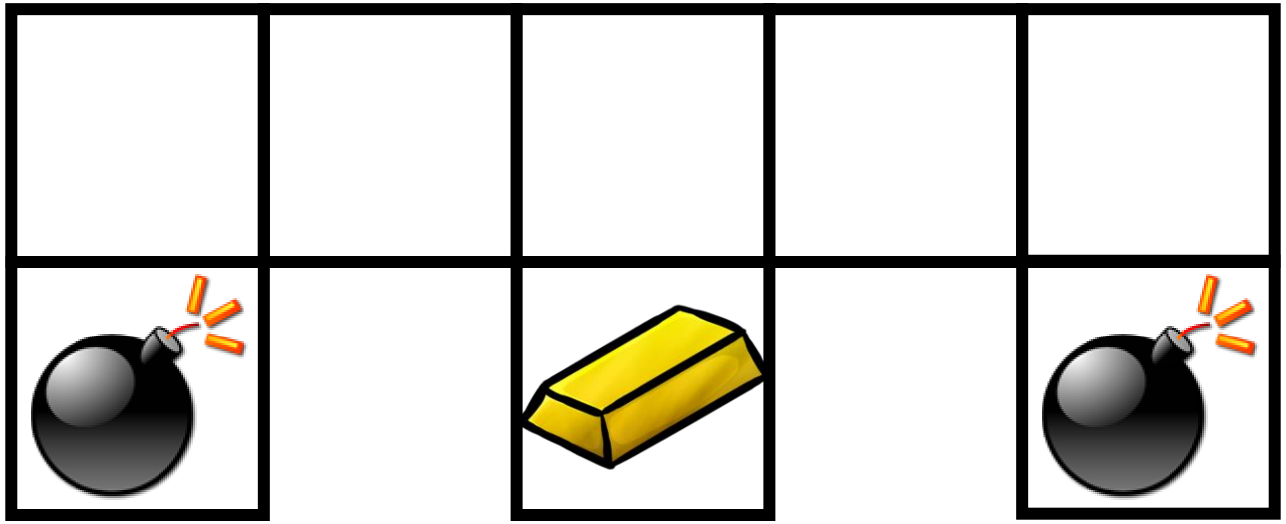
Why Policy-Based Methods?

- So far, we have worked with **value-based** methods.
  - We'd learn the action-value function, then derive a policy (e.g. ϵ-greedy).
- What if the optimal policy is **stochastic**?
  - Value-based methods have no natural way of dealing with this.

- Instead, could we learn the optimal policy directly?

Aliased Gridworld

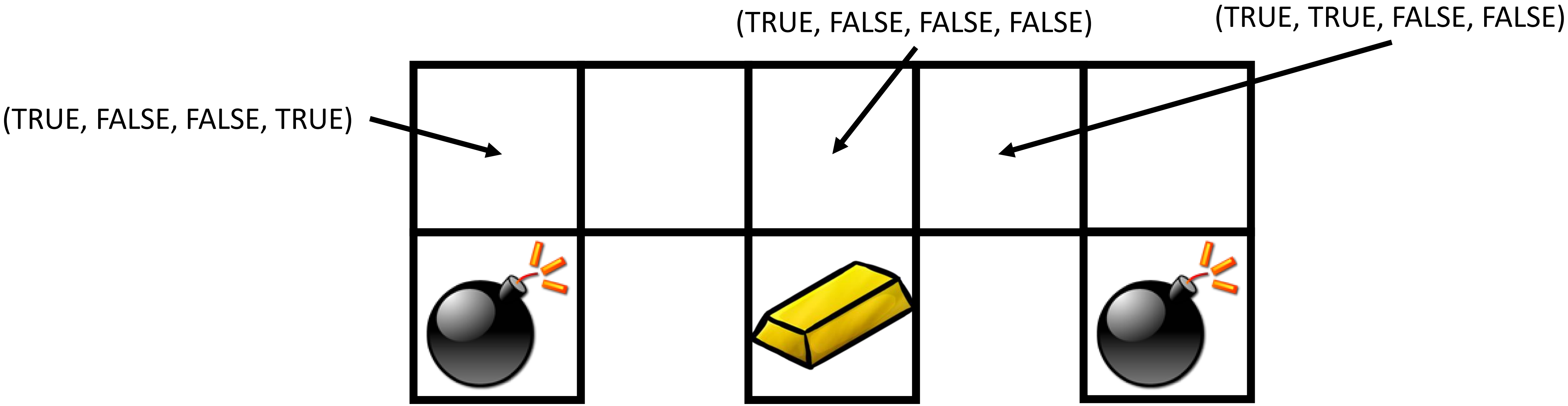
State = (Wall to North, Wall to South, Wall to East, Wall to West)



Example Credit: [David Silver]

Aliased Gridworld

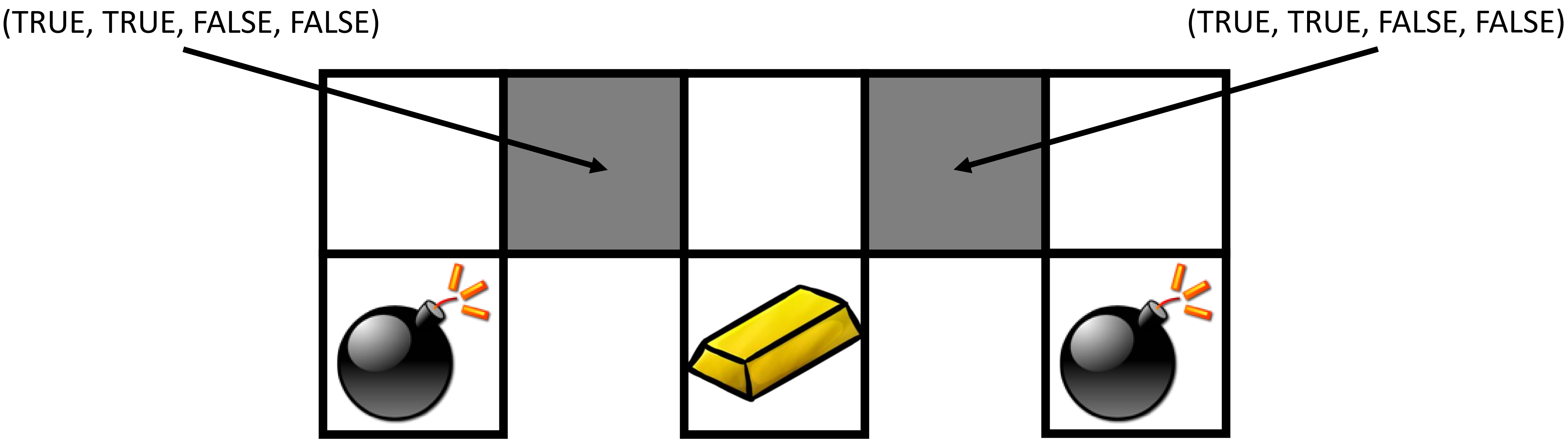
State = (Wall to North, Wall to South, Wall to East, Wall to West)



Example Credit: [David Silver]

Aliased Gridworld

State = (Wall to North, Wall to South, Wall to East, Wall to West)

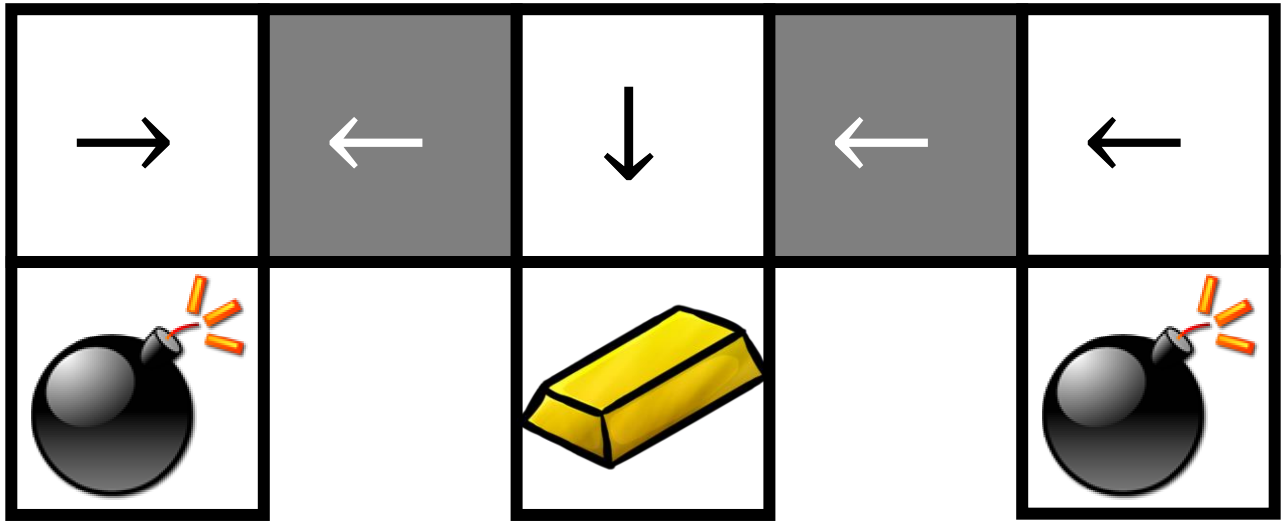


These two states have identical representations!

Example Credit: [David Silver]

Aliased Gridworld

State = (Wall to North, Wall to South, Wall to East, Wall to West)

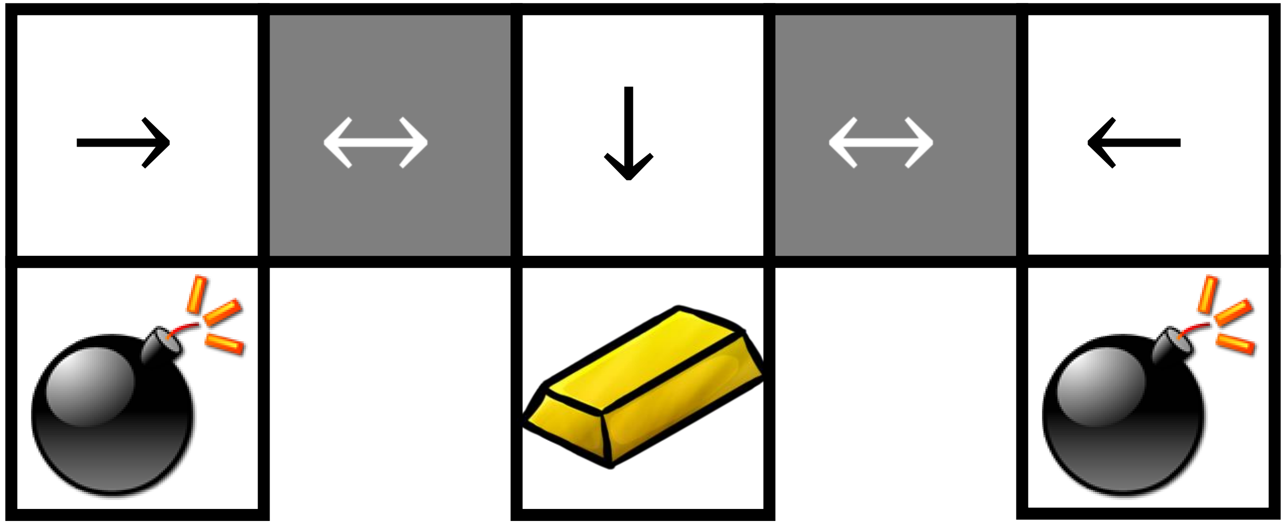


These two states have identical representations!
A deterministic policy would get stuck.

Example Credit: [David Silver]

Aliased Gridworld

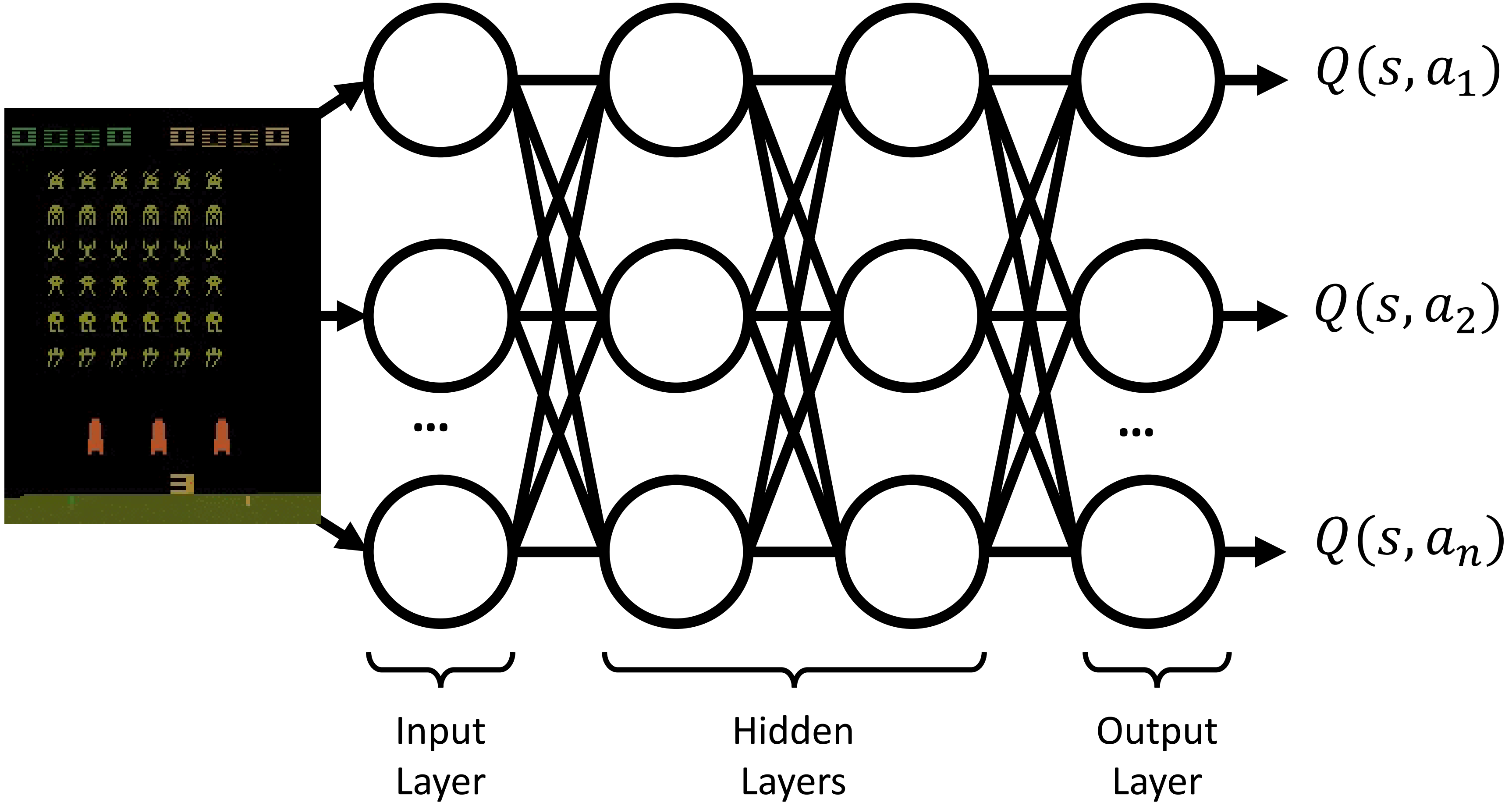
State = (Wall to North, Wall to South, Wall to East, Wall to West)



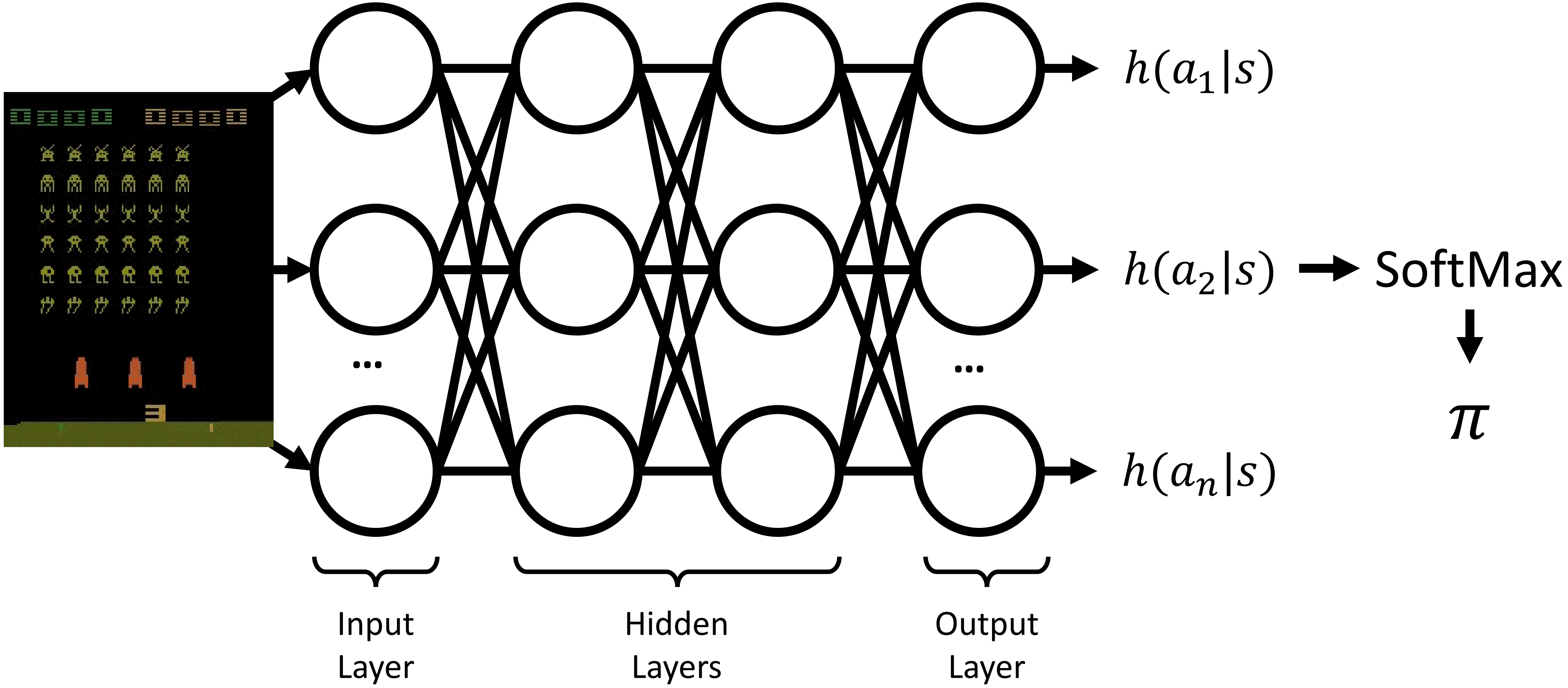
These two states have identical representations!
A stochastic policy would work much better!

Example Credit: [David Silver]

Q-Network



Policy Network



Policy Gradient Methods

$\pi_{\boldsymbol{\theta}}(a|s)$ = Probability of choosing action a given state $\boldsymbol{s}$ and policy parameters $\boldsymbol{\theta}$.

- $J(\pi_{\boldsymbol{\theta}})$ is some **objective function** for our policy, which we aim to maximise (e.g. expected return, $\mathrm{E}_{\pi_\theta}[G_t]$).
- Policy gradient update rule: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}_t} J\left(\pi_{\boldsymbol{\theta}_t}\right)$
- $\nabla_{\boldsymbol{\theta}} J(\pi_{\boldsymbol{\theta}})$ is called the **policy gradient**.

Policy Gradient Methods

- Policy gradient update: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}_t} J\left(\pi_{\boldsymbol{\theta}_t}\right)$
  - Where our objective function, $J\left(\pi_{\boldsymbol{\theta}_t}\right)$, is the discounted return, $\mathrm{E}_{\pi_\theta}[G_t]$.

- **Problem**: We don’t have direct access to, the gradient of the discounted return w.r.t. our policy parameters!

- So, to actually use this update rule in an algorithm, we need an expression for the policy gradient which we can numerically compute.
  - This expression should be computable using only π, $\nabla_{\boldsymbol{\theta}}\pi$, $\boldsymbol{\theta}_{\boldsymbol{t}}$, J…
  - …and a **trajectory** τ of our agent’s experience.

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta E_{\tau\sim\pi_\theta}[G_t]$$

$$= \nabla_\theta \int_\tau P(\tau \mid \theta) G_t$$

Expand expectation.

$$= \int_\tau \nabla_\theta P(\tau \mid \theta) G_t$$

Bring gradient under integral.

$$= \int_\tau P(\tau \mid \theta) \nabla_\theta \log P(\tau \mid \theta) G_t$$

Log-derivative trick.

$$= E_{\tau\sim\pi}[\nabla_\theta \log P(\tau \mid \theta)\, G_t]$$

Return to expectation form.

$$= E_{\tau\sim\pi}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi(A_t \mid S_t) G_t\right]$$

Substitute expression for grad-log-prob.

High-level, hand-wavy intuition:

- Push up the preferences of actions that lead to high returns.
- Push down the preferences of actions that lead to low returns.

We can approximate this using the sample mean over many trajectories $\tau \in D$:

$$\nabla_\theta J(\pi_\theta) \approx \frac{1}{|D|} \sum_{\tau \in D} \sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t$$

For full derivation, see the fantastic OpenAI Spinning Up tutorial!

Algorithm: REINFORCE
Initialise parameters: step size $\alpha \in (0,1]$
Initialise policy network π with parameters $\boldsymbol{\theta}$

For episode $= 1, M$ **do**
  Generate an episode trajectory $\tau \sim \pi_{\boldsymbol{\theta}}$
  For $t = 1, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} R_k$ ← Sample Return
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \underbrace{\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(A_t | S_t)\, G}_{\text{Our Policy-Gradient!}}$
  End For
End For

In this example, we're using $\gamma = 1$ (i.e., no discounting).

Actor-Critic Methods

- With REINFORCE, we are stuck with performing Monte Carlo updates.
- If we learn **a policy function <u>and</u> a value function**, we can use the value function to do bootstrapping, letting us perform TD updates.
- This gives us all the previous benefits that we've seen from bootstrapping, and the benefits of policy-based methods.
- We call bootstrapping methods that learn the policy function directly while using estimates from value functions **Actor-Critic Methods**.
  - The policy function is the "**Actor**".
  - The value function is the "**Critic**".

Types of Reinforcement Learning Methods

- **Value-Based** RL
  - Approximates the **optimal action-value function** $Q^*(s, a)$.
- **Policy-Based** RL
  - Directly search the policy-space for the **optimal policy** $\pi^*(a|s)$.

Actor-Critic methods do both!

Deep RL uses deep neural networks to represent the value function or policy

N.B. these are not mutually exclusive! (See, I told you!)

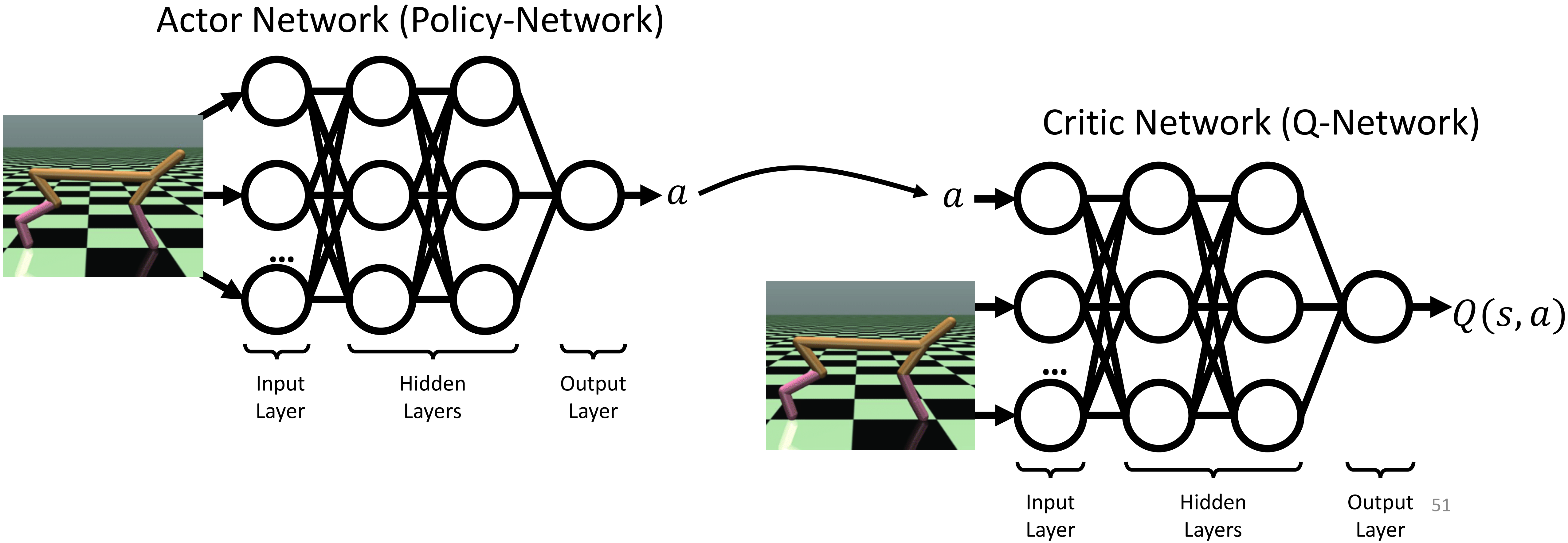
Silver, D. et al., 2014, January. Deterministic policy gradient algorithms. In International conference on machine learning (pp. 387-395). PMLR. [URL]
Lillicrap, T.P. et al. 2015. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971. [URL]

Deep Deterministic Policy Gradients

- DDPG is an actor-critic algorithm for **continuous action-spaces**.
- It makes use of many of DQN's tricks, such as replay buffers and target networks.
- It is off-policy, so can make use of old experiences.
- It makes policy-gradient updates maximising $\mathrm{E}[Q(s,a)]$.

Recall: We maximised $\mathrm{E}[G_t]$ earlier!

DDPG Network Architecture



Algorithm: Deep Deterministic Policy Gradients (DDPG)

Initialise replay memory D to capacity N ← Pre-Fill Replay Buffer

Initialise critic network Q with random weights θ^Q, actor network π with weights θ^π

Initialise target critic network $\hat{Q}$ with weights $\hat{\theta}^Q = \theta^Q$, target actor network $\hat{\pi}$ with weights $\hat{\theta}^\pi = \theta^\pi$ ← Initialise Target Networks

Initialise target network learning rate $\beta \in (0,1]$

For episode $= 1, M$ **do**

  Initialise random process $\mathcal{N}$ for action exploration ← We use random noise for exploration.

  Initialise initial state s_1

  For $t = 1, T$ **do**

    Select action $a_t = \pi(s_t) + \mathcal{N}_t$

    Execute action a_t and observe reward r_{t+1}, next state s_{t+1}

    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in D ← Store Experience in Replay Buffer

    Sample random minibatch of transitions $(s_j, a_j, r_{j+1}, s_{j+1})$ from D ← Sample From Replay Buffer

    Set $y_j = r_{j+1} + \gamma \hat{Q}\left(s_{j+1}, \hat{\pi}(s_{j+1})\right)$ ← Generate Target Using Target Networks

    Perform gradient descent step $\nabla_{\theta^Q}\left(y_j - Q(s_j, a_j)\right)^2$ on critic ← Critic update very similar to DQN's.

    Perform gradient ascent step $\nabla_{\theta^\pi}\mathrm{E}\left[Q\left(s_j, \pi(s_j)\right)\right]$ on actor ← Actor update similar to REINFORCE's.

    Update target networks $\hat{\theta}^Q \leftarrow \beta\theta^Q + (1-\beta)\hat{\theta}^Q$, $\hat{\theta}^\pi \leftarrow \beta\theta^\pi + (1-\beta)\hat{\theta}^\pi$ ← Update Target Networks

  End For

End For

Schulman, J., Levine, S., Abbeel, P., Jordan, M. and Moritz, P., 2015, June. Trust region policy optimization. In International conference on machine learning (pp. 1889-1897). PMLR. [URL]
Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347. [URL]

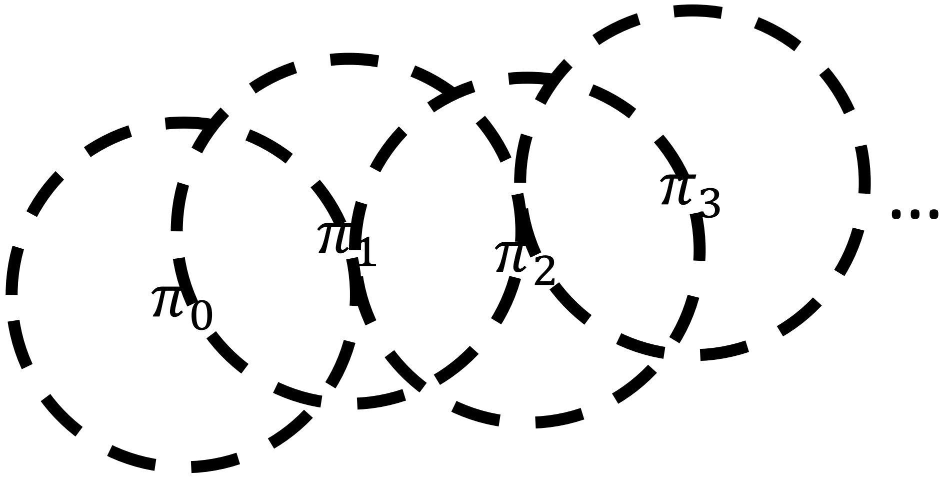
Trust Region Policy Updates

- To avoid instability in training, we don't want to change our policy too much after any given update. **How can we enforce this?**
- **Idea**: Constrain the "distance" between our current and new policies.
  - The area of the policy space we can update within is called the **trust region**.

How can we implement this? *

$$\bar{D}_{KL}(\boldsymbol{\theta}_{k+1}||\boldsymbol{\theta}_k) = E_{s\sim\pi_{\boldsymbol{\theta}_k}}\left[D_{KL}\left(\pi_{\theta_{k+1}}(\cdot\,|s)||\pi_{\theta_k}(\cdot\,|s)\right)\right]$$

During the optimisation step, constrain the KL divergence between the old and new policies across states visited by the old policy.



*Other (better) ways exist, but this is a simple case that gives you the general idea behind how these types of methods work.

Offline RL

- We’ve seen Off-Policy RL methods (e.g., Q-Learning) use data generated by one policy to learn about another.
- Offline RL methods take this idea to the extreme.
  - The agent is given a **fixed dataset of experience**. This experience could be generated by other agents, human demonstrated, etc.
  - The agent has to learn a policy using **only this fixed dataset** – it can’t interact with the environment to generate any more data itself.
- This brings with it many unique challenges!
  - Exploring is not possible!
  - Distributional shifts between dataset experience and online experience.

Inverse RL

- Regular RL:
  - What **policy** maximises the long-term reward for a given problem?
- Inverse RL:
  - What **reward function** is a given policy maximising?
- Useful when we have demonstrations from an agent completing a task, but we don’t know the reward function being maximised.
  - We can use Inverse RL methods to derive the reward function, then train a reinforcement learning agent to solve the task.

Schmidhuber, J., 1991. A possibility for implementing curiosity and boredom in model-building neural controllers. In *Proc. of the international conference on simulation of adaptive behavior: From animals to animats* (pp. 222-227). [URL]
Chentanez, N., Barto, A. and Singh, S., 2004. Intrinsically motivated reinforcement learning. Advances in neural information processing systems, 17. [URL]
Burda, Y., Edwards, H., Storkey, A. and Klimov, O., 2018, September. Exploration by random network distillation. In International Conference on Learning Representations. [URL]

Intrinsically-Motivated RL

$$\underbrace{R_t}_{\text{Total Reward}} = \underbrace{R_t^E}_{\text{Extrinsic Reward}} + \underbrace{R_t^I}_{\text{Intrinsic Reward}}$$

- Many difficult problems have very sparse rewards.
- Can we use **intrinsic** rewards to drive intelligent behaviour, even in the absence of **extrinsic** rewards from the environment?
  - Many examples of this in nature: novelty, curiosity, regularity, information-gain, empowerment, skill diversity…
- A closer look at one example: **Curiosity**
  - ~~Define intrinsic reward based on **prediction errors**~~.
  - Define intrinsic reward based on **prediction improvements**.

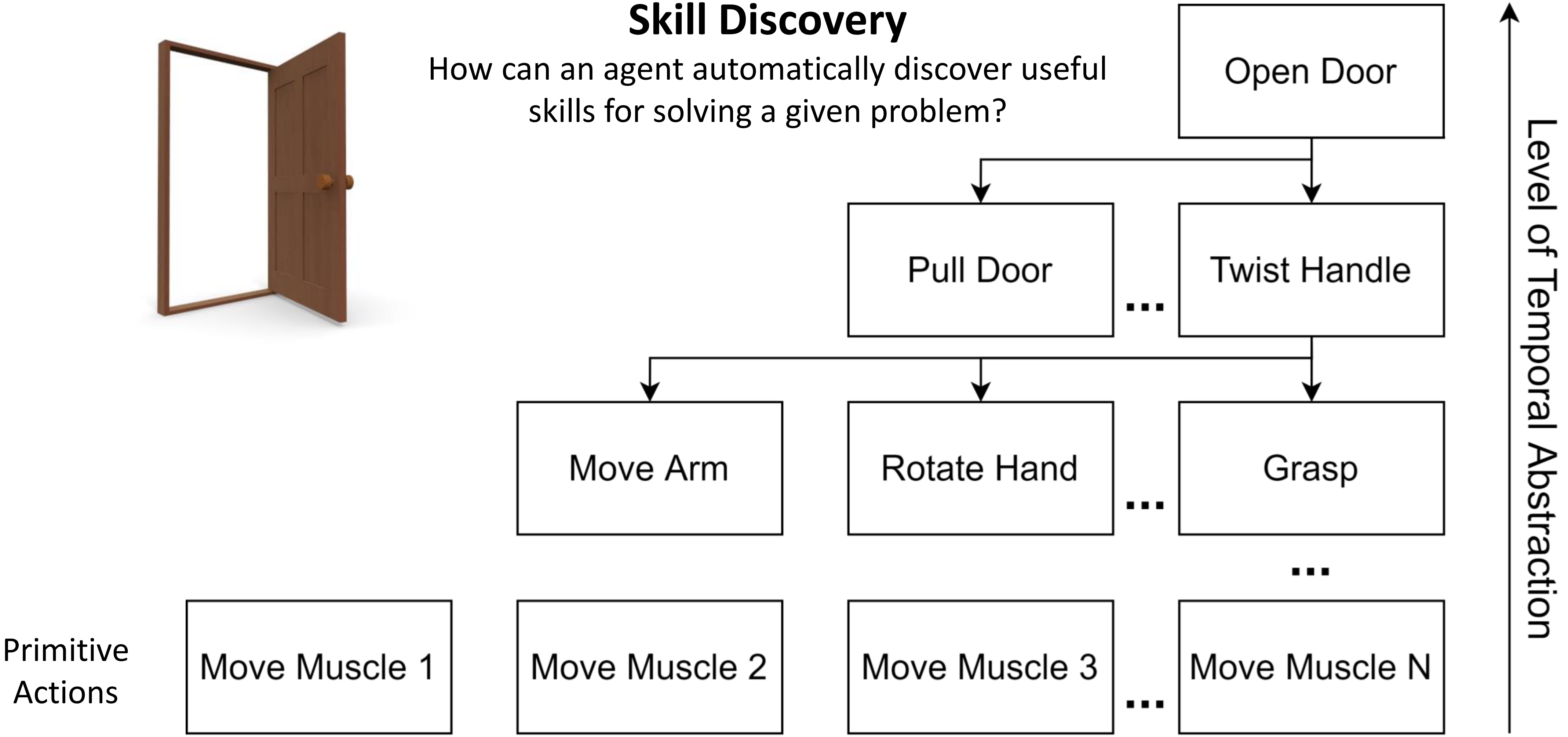
Noisy TV Problem: Our agent will never be learn to predict what's going to be shown next on this noisy TV screen.

Sutton, R.S., Precup, D. and Singh, S., 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial intelligence, 112(1-2), pp.181-211. [URL]
Şimşek, Ö. and Barto, A., 2008. Skill characterization based on betweenness. Advances in neural information processing systems, 21. [URL]
Bacon, P.L., Harb, J. and Precup, D., 2017, February. The option-critic architecture. In Proceedings of the AAAI conference on artificial intelligence (Vol. 31, No. 1). [URL]
Evans, J. B. and Şimşek, Ö. (2023) 'Creating Multi-Level Skill Hierarchies in Reinforcement Learning', in Thirty-seventh Conference on Neural Information Processing Systems. [URL]

Hierarchical RL

Skill Discovery

How can an agent automatically discover useful skills for solving a given problem?



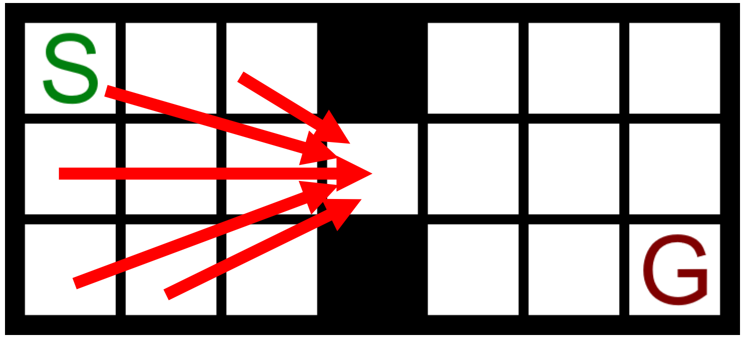
Sutton, R.S., Precup, D. and Singh, S., 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. Artificial intelligence, 112(1-2), pp.181-211. [URL]
Şimşek, Ö. and Barto, A., 2008. Skill characterization based on betweenness. Advances in neural information processing systems, 21. [URL]
Bacon, P.L., Harb, J. and Precup, D., 2017, February. The option-critic architecture. In Proceedings of the AAAI conference on artificial intelligence (Vol. 31, No. 1). [URL]
Evans, J. B. and Şimşek, Ö., 2023. Creating Multi-Level Skill Hierarchies in Reinforcement Learning. In Thirty-Seventh Conference on Neural Information Processing Systems. [URL]

Hierarchical RL

- How to represent skills? One approach: The **Options** framework.

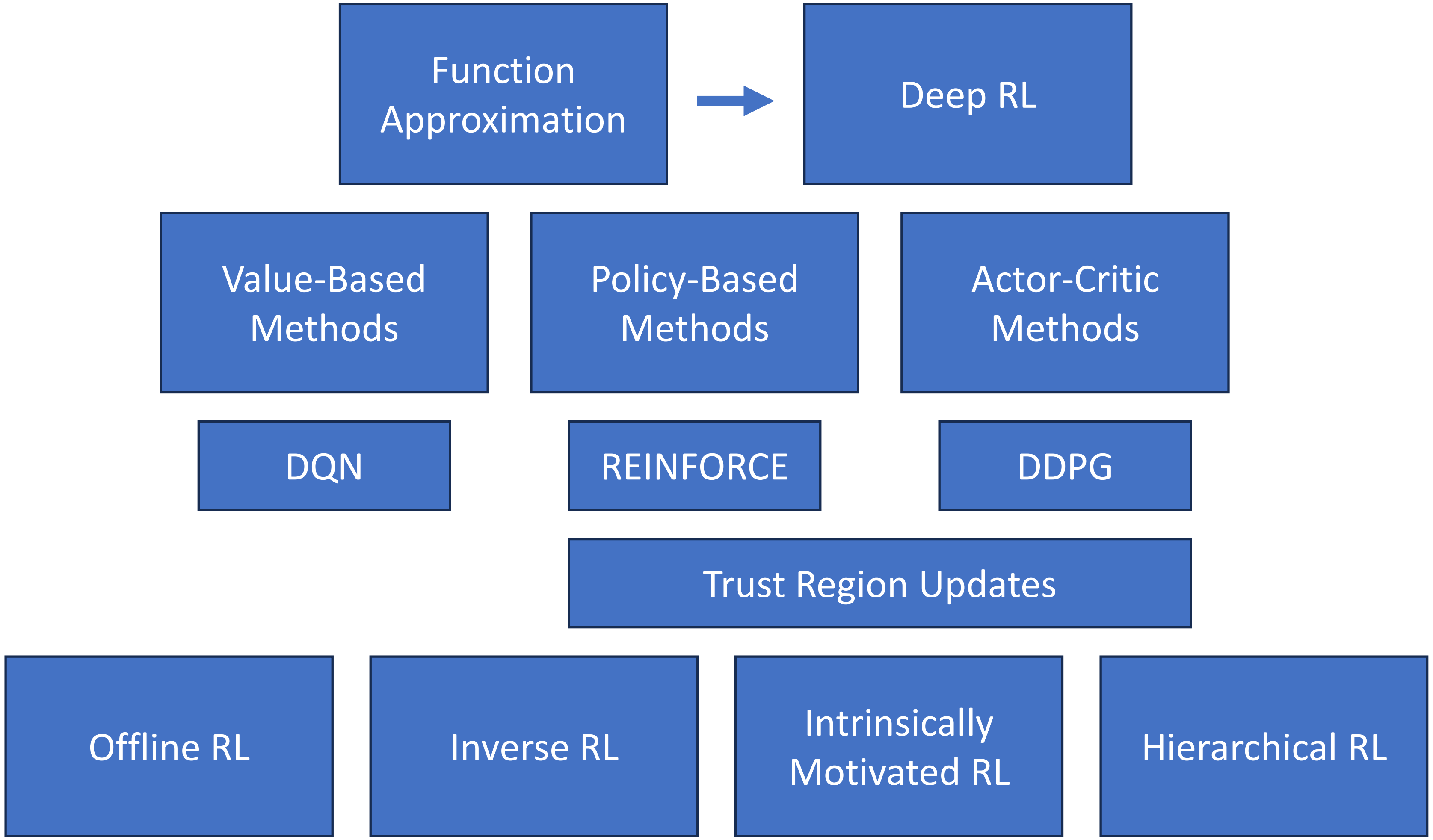
$$o = < I_o, \pi_o, \beta_o >$$

  - I_o - **Initiation Set**: Which states can I select this skill in?
  - π_o - **Option Policy**: What action does this skill select in each state?
  - β_o - **Termination Condition**: Which states does this skill terminate in?



- How to discover skills?
  - Many different approaches proposed. But still an open research question!

Summary


Function Approximation
Deep RL
Value-Based Methods
Policy-Based Methods
Actor-Critic Methods
DQN
REINFORCE
DDPG
Trust Region Updates
Offline RL
Inverse RL
Intrinsically Motivated RL
Hierarchical RL