Reinforcement Learning

ANNA REALI, FELIPE LENO, REINALDO BIANCHI

Advanced Institute for Artificial Intelligence

Content

- Introduction
- MDP and Policies
- Reinforcement Learning and Q-learning
- Deep RL and DQN
- Policy Gradient Methods
- Applications

Q-Learning

A simple algorithm: Q-learning

- **Key idea:**
  - Update the action-value function Q(s,a) **using the experience sequences**
  - Then use Q(s,a) to estimate π
- **Characteristics:**

Model-free, value-based, off-policy

Q-learning

Initialize Q(s,a) arbitrarily
Observe the current state s_t
do forever
  select an action a_t and execute it in s_t
  receive immediate reward $r(s_t,a_t)$
  observe the new state s_{t+1}
  update Q(s,a) as follows:

$$Q_{t+1}(s_t,a_t) \leftarrow (1-\alpha)Q_t(s_t,a_t) + \alpha[r(s_t,a_t) + \gamma \max_a Q_t(s_{t+1},a)]$$

  $s_t \leftarrow s_{t+1}$

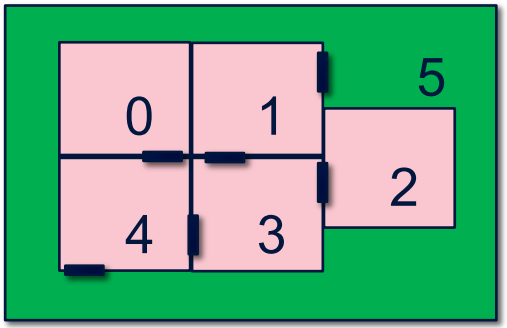
Q-learning

Initialize Q(s,a) arbitrarily
Observe the current state s_t
do forever
  select an action a_t and execute it in s_t
  receive immediate reward $r(s_t,a_t)$
  observe the new state s_{t+1}
  update Q(s,a) as follows:

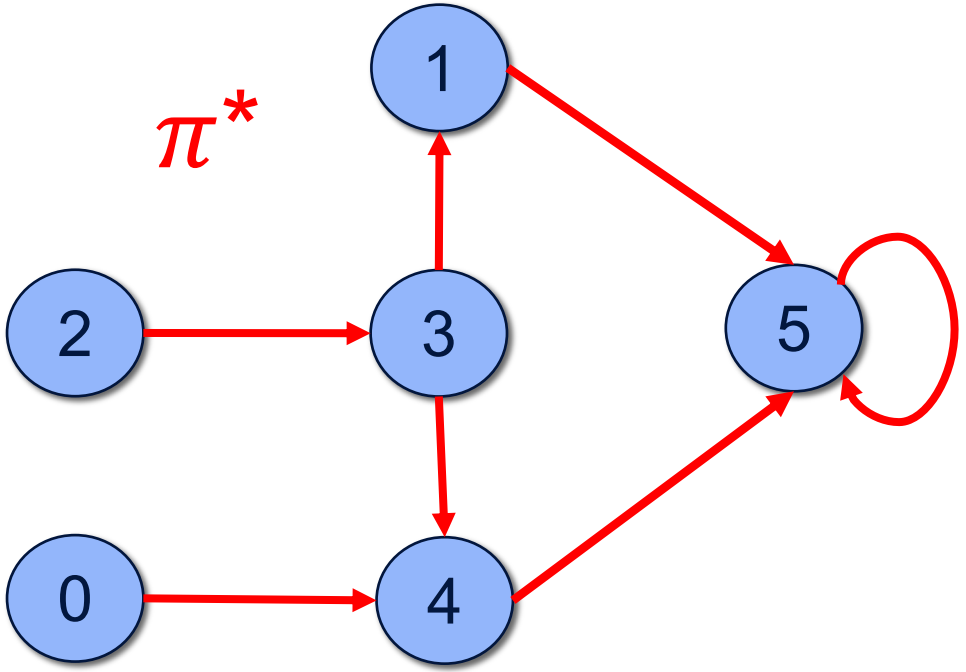
$$Q_{t+1}(s_t,a_t) \leftarrow (1-\alpha)Q_t(s_t,a_t) + \alpha[r(s_t,a_t) + \gamma \max_a Q_t(s_{t+1},a)]$$

  $s_t \leftarrow s_{t+1}$

A Simple Example of Q-Learning



≡



A Simple Example of Q-Learning

If our agent learns more through further episodes, it will finally reach convergence values in Q-table like:

s a	0	1	2	3	4	5
0	0	0	0	0	80	0
1	0	0	0	64	0	100
2	0	0	0	64	0	0
3	0	80	51	0	80	0
4	64	0	0	64	0	100
5	0	0	0	0	0	0

V*(0)=80 $\pi^*(0)$ = go to 4

V*(1)=100 $\pi^*(1)$ = go to 5

V*(2)=64 $\pi^*(2)$ = go to 3

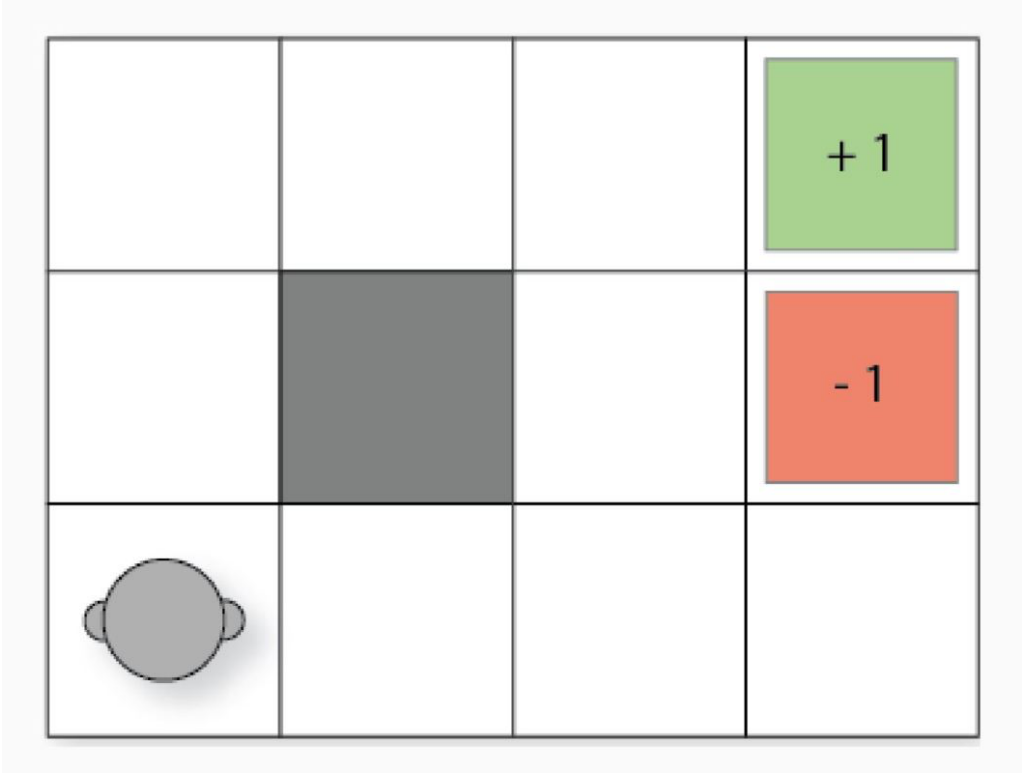
V*(3)=80 $\pi^*(3)$ = go to 4 or 1

V*(4)=100 $\pi^*(4)$ = go to 5

V*(5)=0 $\pi^*(5)$ = go to 5 (goal)

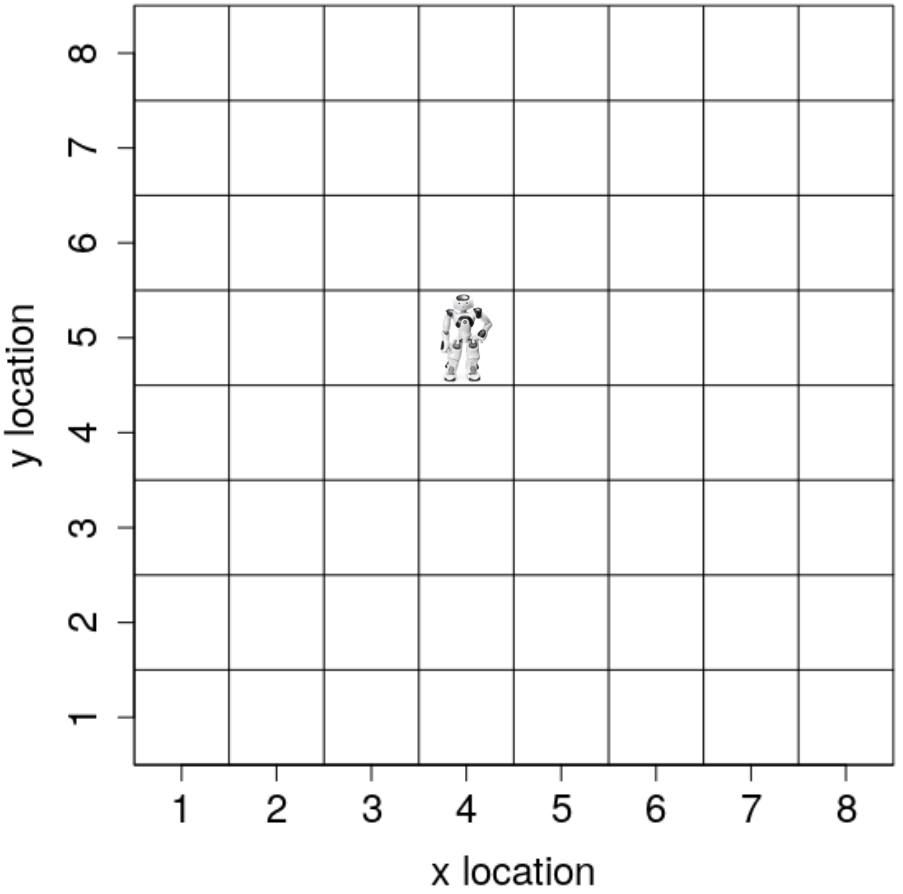
Other examples:

- Grid World 3 x 4:
  - States = 12
  - Actions = 4
  - Size = 48



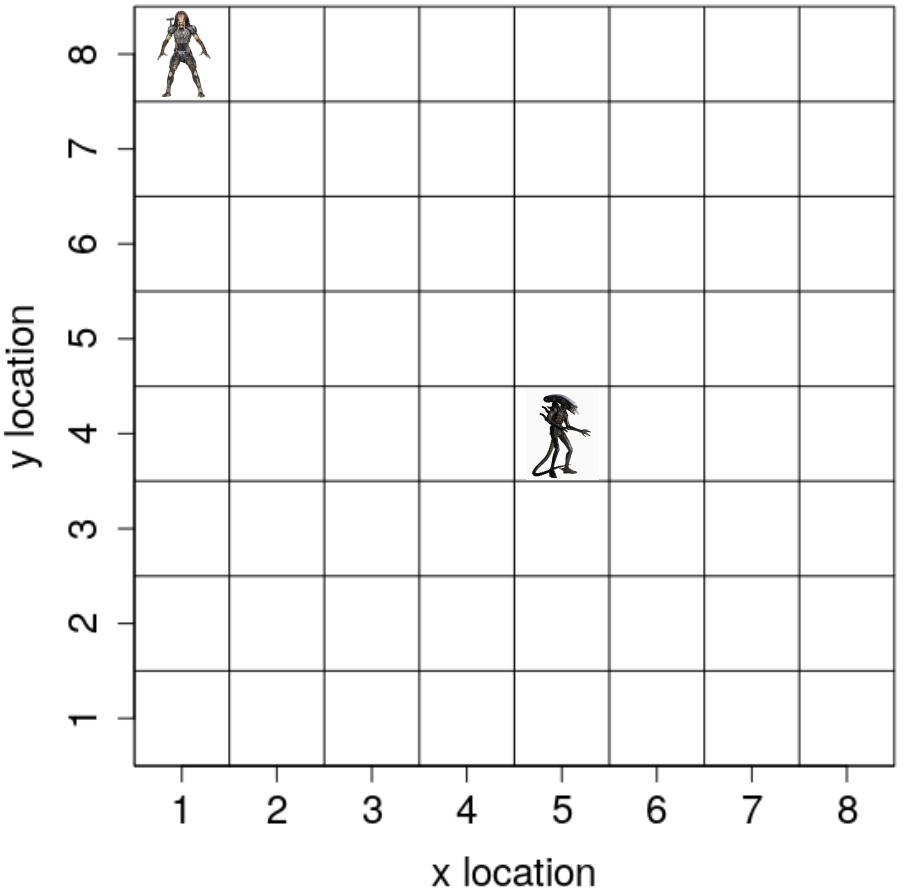
Other examples:

- Grid World 10 x 10:
  - States = 100
  - Actions = 4
  - Size = 4,000



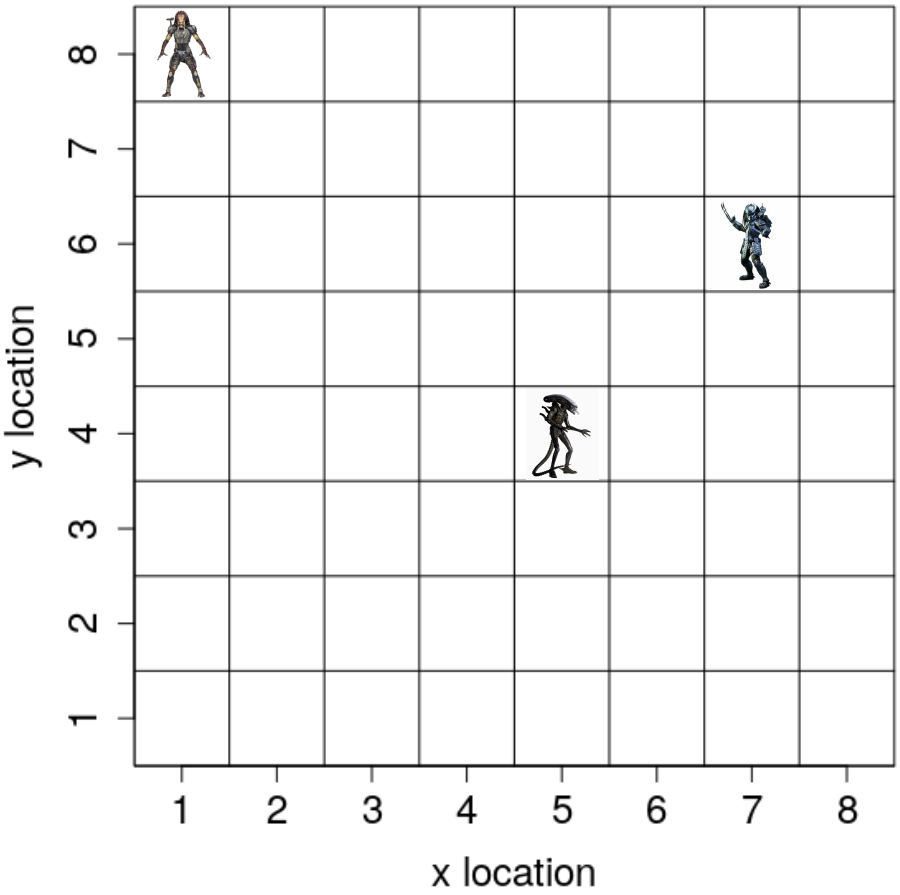
Other examples:

- Predator - Prey Problem:
  - 1 vs 1:
  - States = 100 x 99
  - Actions = 4 (N, S, E, W)
  - Size = 39,600



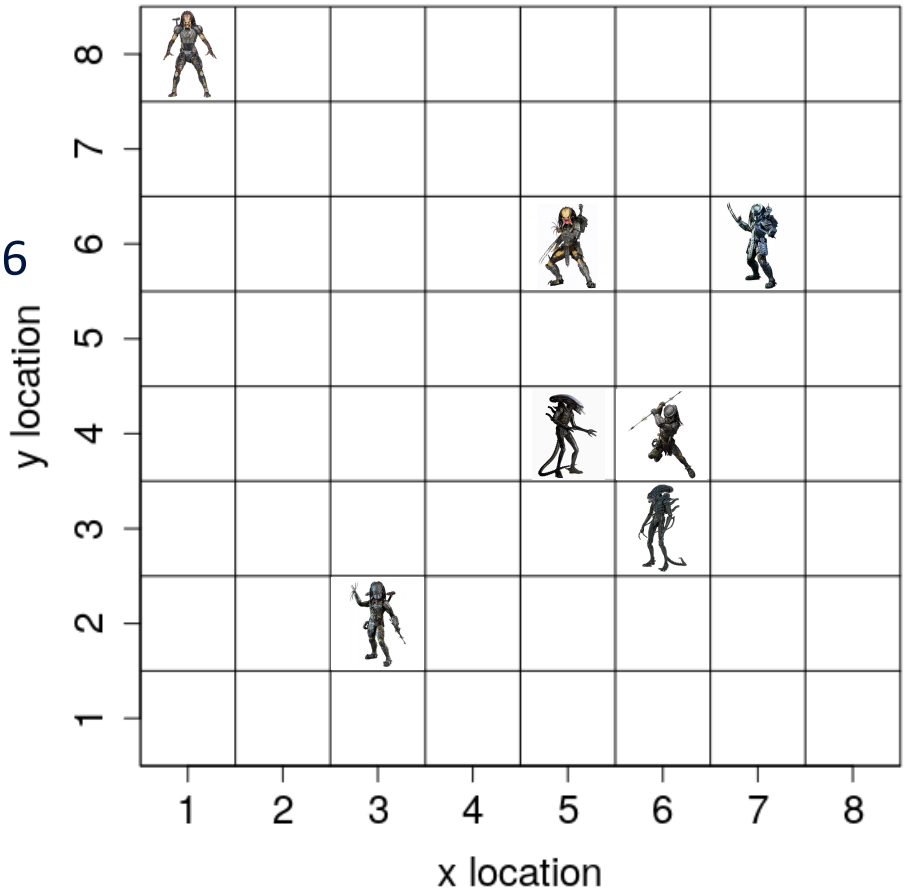
Other examples:

- Predator - Prey Problem:
  - 2 vs 1:
  - States = 100 x 99 x 98
  - Actions = 4 (N, S, E, W)
  - Size = 3,880,800



Other examples:

- Predator - Prey Problem:
  - 5 vs 2:
  - States = 100 x 99 x 98 x 97 x 96 x 95 x 94
  - Actions = 4 (N, S, E, W)
  - Size = 322,712,425,728,000
  - Size = 332×10^{12} table cells
  - If using float
  - Size = 1.29 PETABYTES



Problems with tabular Q-Learning

- In realistic situations we cannot learn about each state!
  - Too many states to visit them all in training (**slow convergence)**
  - Too many space to hold the Q-tables in memory (demand for **memory resources)**
- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar states

Q-learning Algorithm: Possible solutions

- **If we know a model:**
  1. We use the known model to build a **simulation**.
  2. Using Q-learning **plus a function approximation technique**, we learn to behave in the simulated environment, which yields a good control policy for the original problem

Hot topics: Deep RL, batch-RL, transfer learning!

Deep Reinforcement Learning

What are Function Approximation?

- Before – methods use lookup tables:
  - The value functions were stored in lookup tables V^{π}, Q^{π}, V^{*}, Q^{*}.
  - Each state-action pair have a defined value.
  - The table must be completely visited to be trained.
- Now – the approximate value function at time t, V_t, is represented not as a table but as a parameterized functional form with a parameter vector $\overrightarrow{\theta_t}$:
  - The value function V_t depends totally on $\overrightarrow{\theta_t}$
  - The value function V_t varies from time step to time step only when $\overrightarrow{\theta_t}$ varies.

Any FA Method?

- In principle, yes:
  - Artificial Neural Networks:
    - $\overrightarrow{\theta_t}$ is the vector of connection weights.
    - By adjusting the weights, any of a wide range of different functions can be implemented by the network.
  - Decision Trees:
    - $\overrightarrow{\theta_t}$ is the vector of all the parameters defining the split points and leaf values of the tree.
  - Multivariate Regression Methods:
    - $\overrightarrow{\theta_t}$ is the vector of features that are combined in a linear manner.

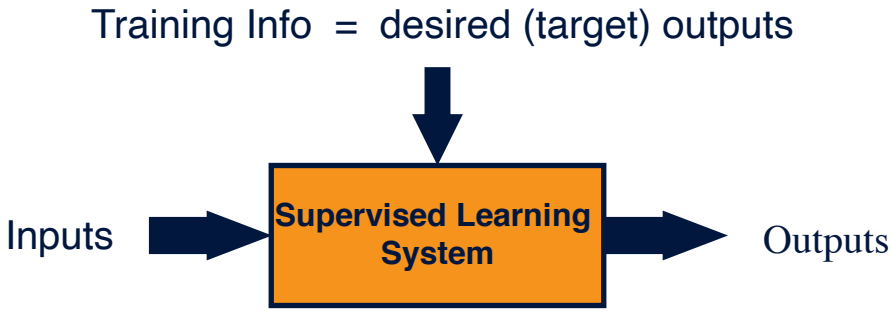
Function Approximators

- Typically, the number of parameters (the number of components of $\overrightarrow{\theta_t}$) is much less than the number of states:
  - Changing one parameter changes the estimated value of many states.
  - Consequently, when a single state is updated, the change generalizes from that state to affect the values of many other states.
- But RL has some special requirements:
  - Usually want to learn while interacting:
    - Methods must handle incrementally acquired data.
  - Ability to handle non-stationarity:
    - Target functions change over time.

How to learn $\overrightarrow{\theta_t}$?

- How to approximate $\overrightarrow{\theta_t}$ and therefore learn the value function?
  - It is natural to interpret each update of the value function as specifying an example of the desired input-output behavior of the estimated value function.
  - Viewing each update as a conventional training example in this way enables us to use any of a wide range of existing function approximation methods for value prediction.
- Several Supervised learning methods can be used:
  - Gradient Descent Methods seems promising ☺

Adapt Supervised Learning Algorithms



Training example = {input, target output}

Error = (target output – actual output)

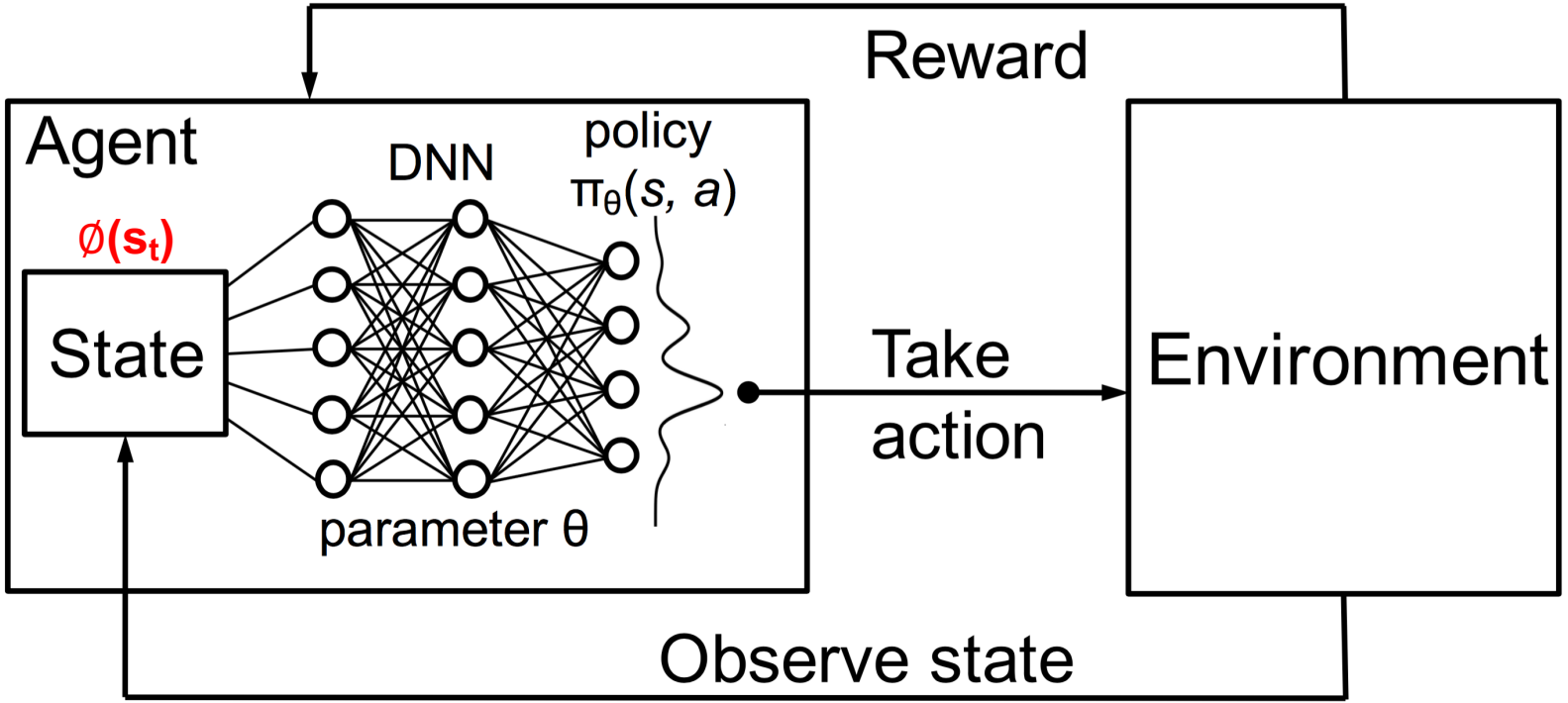
What is Deep Reinforcement Learning?

- Deep RL is standard RL where a deep neural network is used to approximate the value function $Q(s, a; \theta)$.
- Deep neural networks require lots of real/simulated interaction with the environment to learn:
  - Lots of trials/interactions is possible in simulated environments.
  - We can easily parallelize the trials/interaction in simulated environments.
  - It is difficult do this in real world because action execution takes time, accidents/failures are expensive and there may be safety concerns

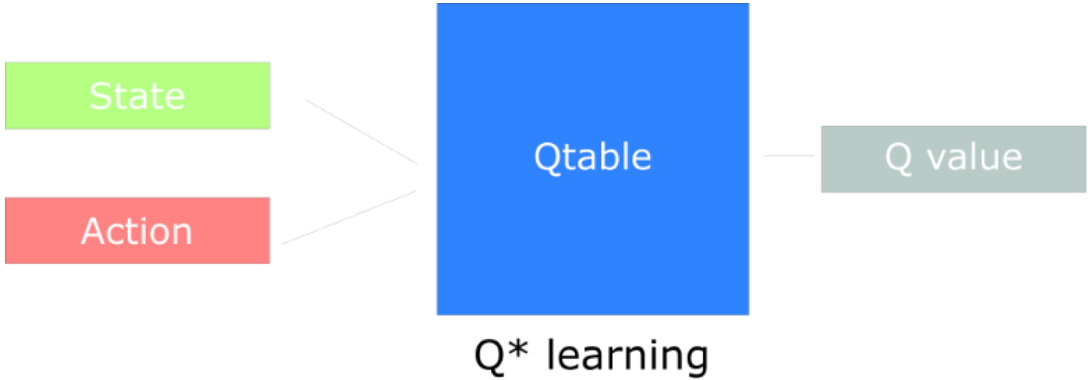
Deep Q-Networks [Mnir, 2013]

- Introduced Deep Reinforcement Learning.
- Deep Q-Networks is Q-learning with a deep neural network function approximator called the Q-network.
  - Core idea: We want the neural network to learn a non-linear hierarchy of features or feature representation that gives accurate Q-value estimates
- The neural network has a separate output unit for each possible action, which gives the Q-value estimate for that action given the input state.

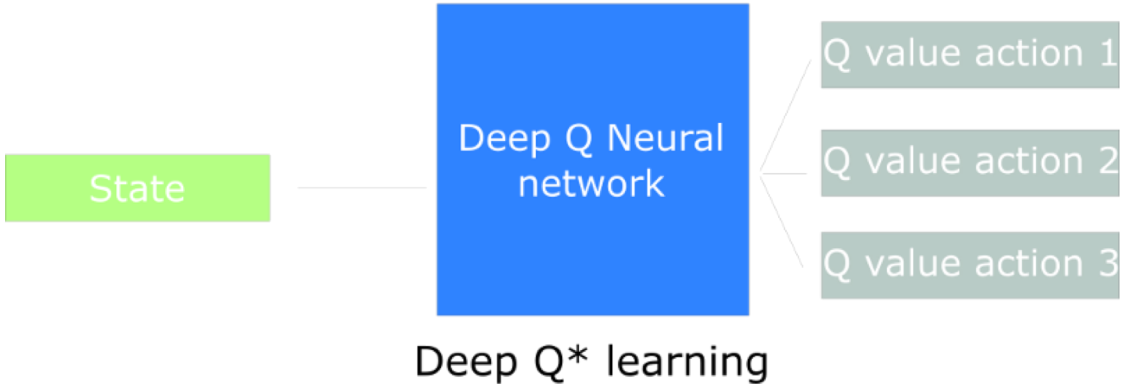
DQN Architecture



Q-Learning versus Deep Q-Learning

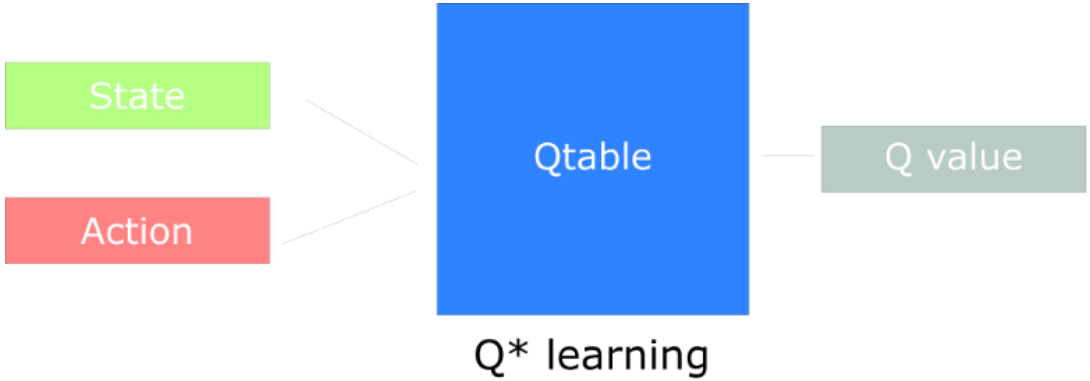


We use Q-value to find our best action.

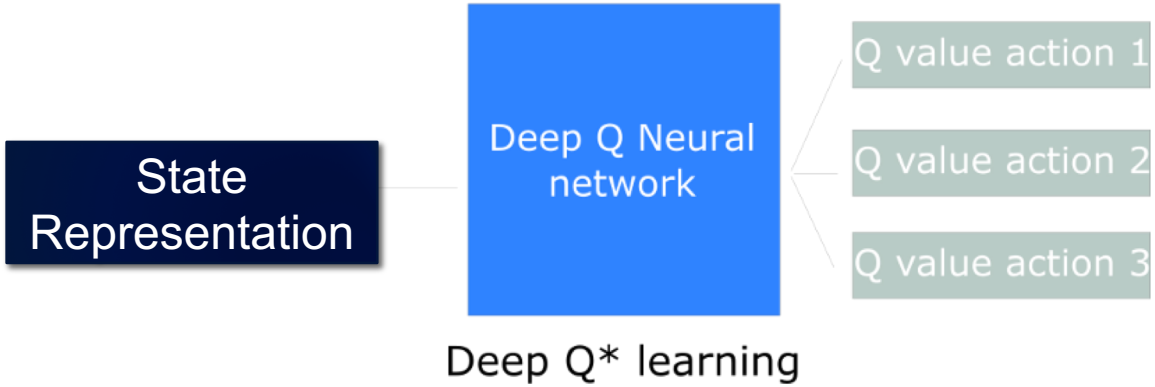


We take the biggest Q-value of this vector to find our best action.

Q-Learning versus Deep Q-Learning



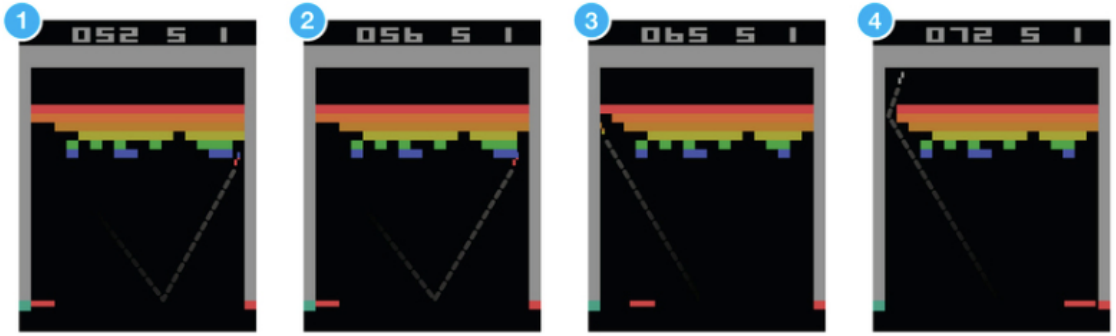
We use Q-value to find our best action.



We take the biggest Q-value of this vector to find our best action.

DQN – State representation

- It is difficult to give the Neural Network a sequence of arbitrary length as input.
- Use fixed length representation of sequence/history produced by a function $\emptyset(s_t)$:
  - Example: The last 4 image frames in the sequence of Breakout gameplay



Q-Learning and Neural Networks

- Supervised learning: can we take the value from the right-hand side of the Q-learning rule as the target for a neural network with weights **w** ?
- Minimize:

$$E = [r_{t+1} + \gamma \max_{a'} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w})]^2$$

- **Diverges** using neural networks due to:
  - Correlations between samples
  - Non-stationary learning target

DQN Learning

- First, we must stop learning while interacting with the environment.
- We should try different things and play a little randomly to explore the state space.
- We can save these experiences in the replay buffer D.
- Then, we can recall these experiences and learn from them.
- After that, we go back to play with updated value function.

Deep Q-Networks (DQN): Experience Replay

- To remove correlations, build a history data-set from the agent's own experiences: $\mathrm{D} = (s_i, a_i, r_{i+1}, s_{i+1})$
- Sample experiences from this data-set and apply the stochastic gradient descent update for the squared loss:

$$E = [r_{t+1} + \gamma \max_a Q(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w})]^2$$

- To prevent problems with non-stationary learning targets, fix the weights $\mathbf{w}^-$ while performing SGD.

Q-Network Training

- Sample random mini-batch of experience tuples uniformly at random from D.
- Similar to Q-learning update rule but:
  - Use mini-batch stochastic gradient updates.
  - Use the gradient of the loss function to update the Q function approximator.
  - The gradient of the loss function for a given iteration with respect to the parameter θ_i is the difference between the target value and the actual value, multiplied by the gradient of the Q function approximator with respect to that specific parameter.

Q-Network Training

- We want to update our NN weights to reduce the error.
- Error = Q_target – Q_value
- Loss function: $L_\theta = Error^2$.
- Differentiating L_θ with respect to θ, we arrive at the following gradient:

$$\Delta\theta = \alpha\left[\left(r + \gamma \max_a Q^*(s', a, \theta^-)\right) - Q(s, a, \theta)\right]\nabla_\theta Q(s, a, \theta)$$

Change in weights

Q_target

Q_value

Error

Gradient of our current predicted Q_value

DQN Experience Replay

- Experience replay will help us to handle two things:
- **Avoid forgetting previous experiences**
  - **Solution**: “replay buffer.” This stores experience tuples while interacting with the environment, and then we sample a small batch of tuple to feed our NN.
- **Reduce correlations between experiences** (every action affects the next state)
  - **Solution**: sample from the replay buffer at random.

DQN Hyper Parameters

- M – number of episodes (games) we want the agent to play.
- T – duration of a game (episode)
- γ – discount rate, to calculate the future discounted reward.
- ϵ – exploration rate.
- ϵ_decay – we want to decrease the number of explorations as it gets good at playing games.
- α – learning rate.
- N – capacity of the replay memory
- C – number of episodes to reset target

Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

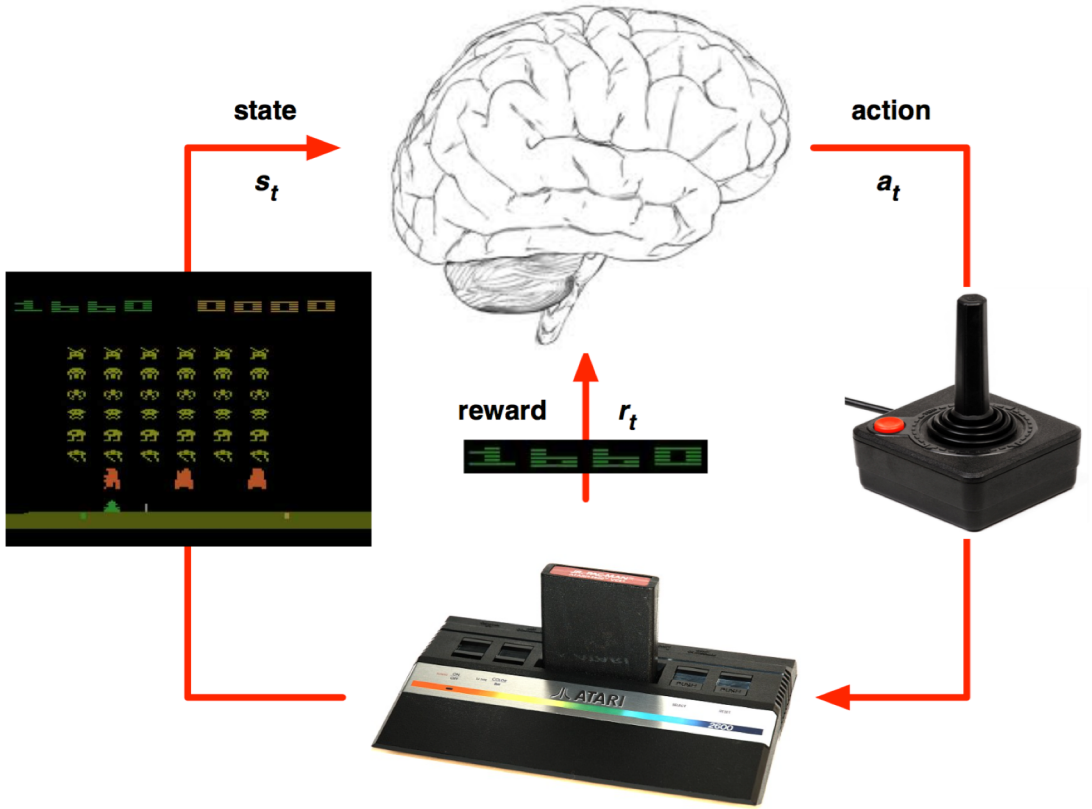
Comments

- DQN is **stable** because it stores the agent's data in **experience replay memory** so that it can be **randomly sampled** from different time-steps
- Aggregating over memory **reduces non-stationarity** and **decorrelates updates** but limits methods to **off-policy** RL algorithms
- Experience replay updates use **more memory and computation** per real interaction than online updates, and require off-policy learning algorithms that can update from data generated by an older policy

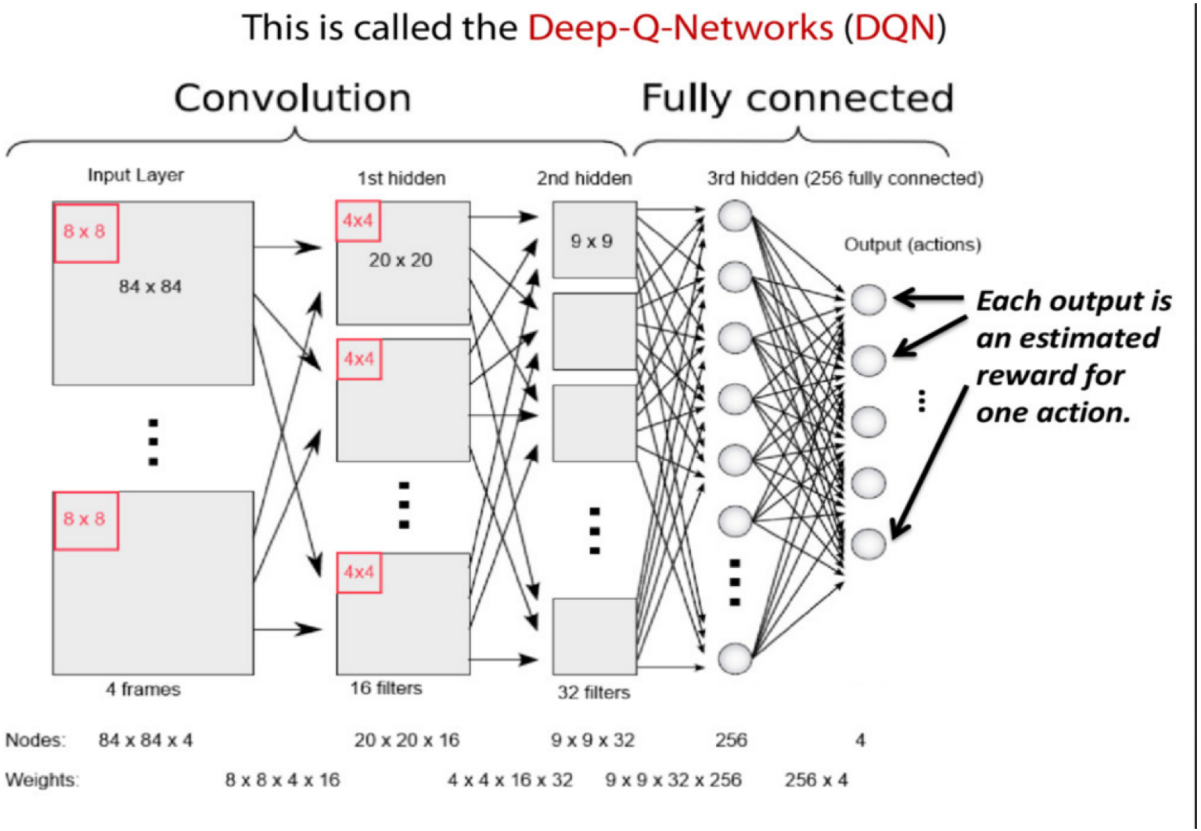
Playing Atari with Deep RL [Mnir, 2013]

- The first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning.
- The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards.
  - Input:
    - Raw Atari Frames 210 x160 pixel with 128-color pallete.
  - Output:
    - Possible Action to be taken.
- Goal: Optimal Policy with Maximal Reward!

Playing Atari with Deep RL [Mnir, 2013]



Architecture



Evaluations

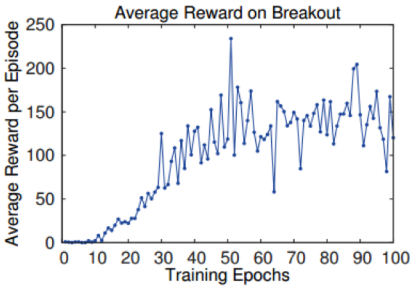
Atari Games Tried:

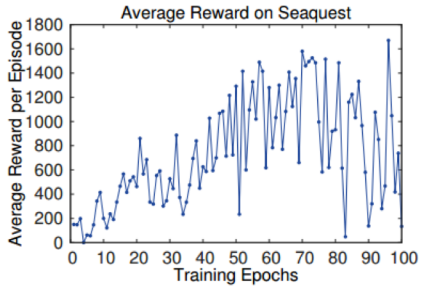
- Beam Rider
- Breakout
- Pong
- Q*bert
- Seaquest
- Space Invaders

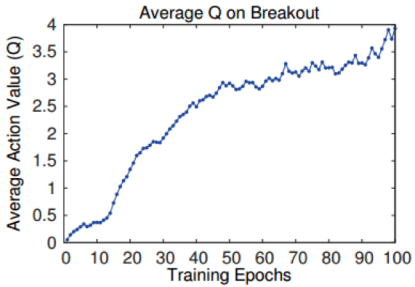
Metrics:

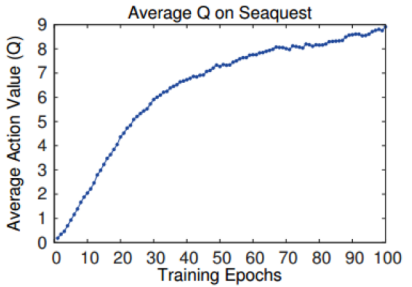
- Average Total Reward Metric
- Average action-value
- Collect a fixed set of states by running a random policy before training starts
- Average of the maximum predicted Q for these states

Evaluations









Evaluations

- Average Total Reward
- Best Performing Episode

	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Analysis

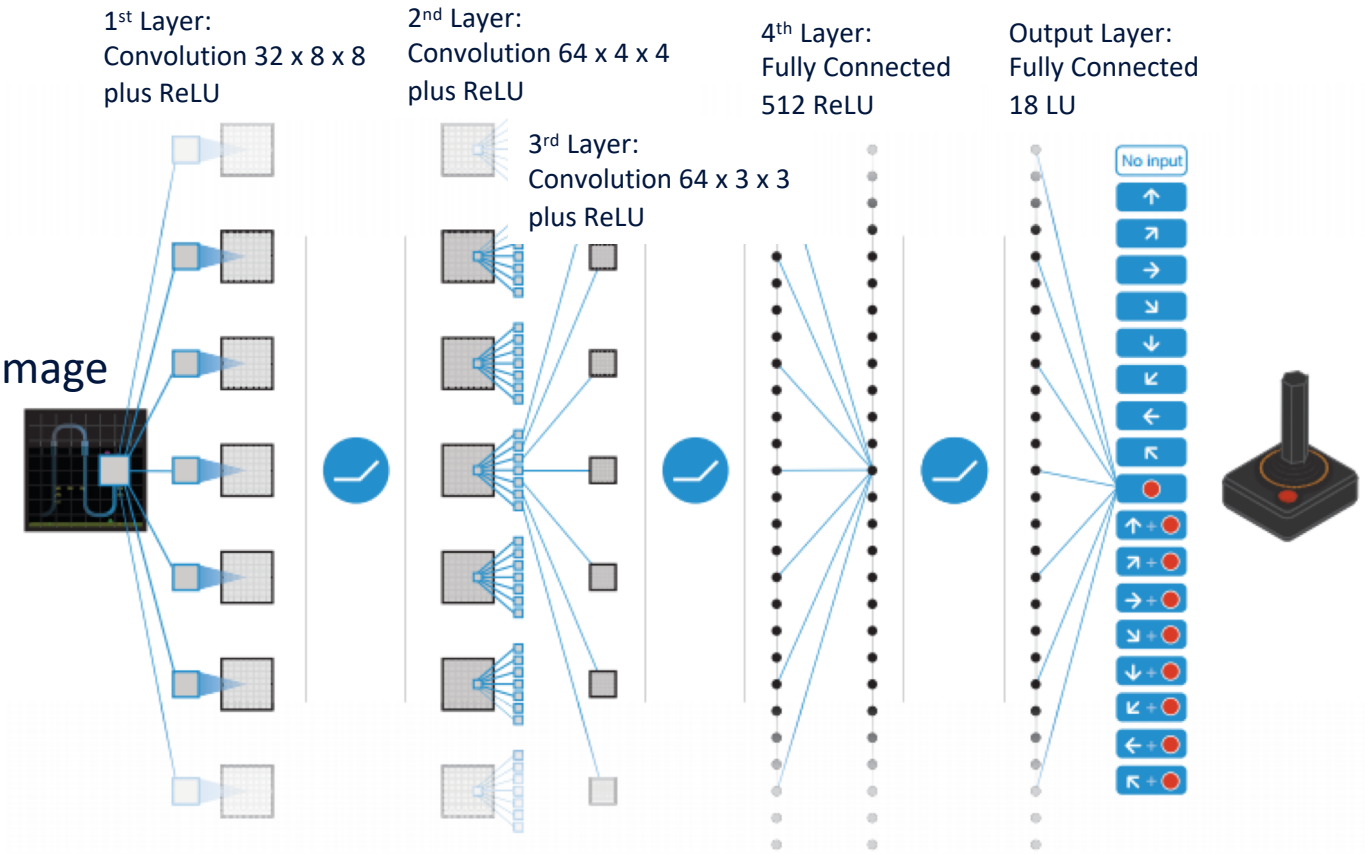
- Strengths:
  - Q-learning over non-linear function approximators
  - End-to-End Learning over multiple games
  - SGD Training
  - Seminal paper
- Weakness:
  - DQN limited to finite discrete actions
  - Long Training
  - Reward Clipping

Human-level control through DRL
[Mnir, 2015, Nature]

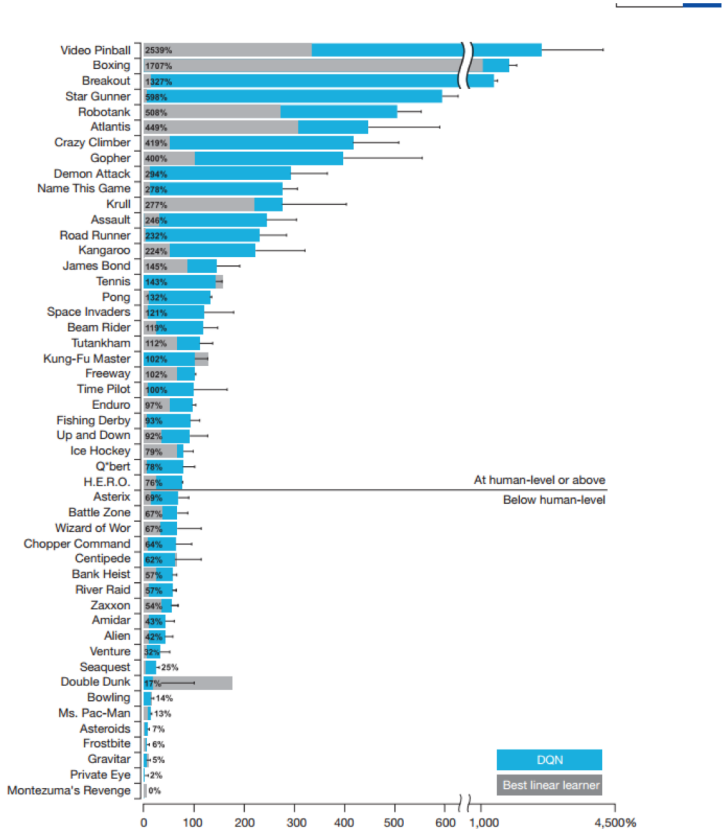
- Extending the DQN architecture to play 49 Atari 2600 arcade games.
  - No pretraining.
  - No game-specific training.
- State: Transitions from 4 frames : Experience Replay
- 18 actions:
  - 9 directions of joystick
  - 9 directions + button
- Reward - Game Score

Architecture

Input:
84 x 84 x 4 image

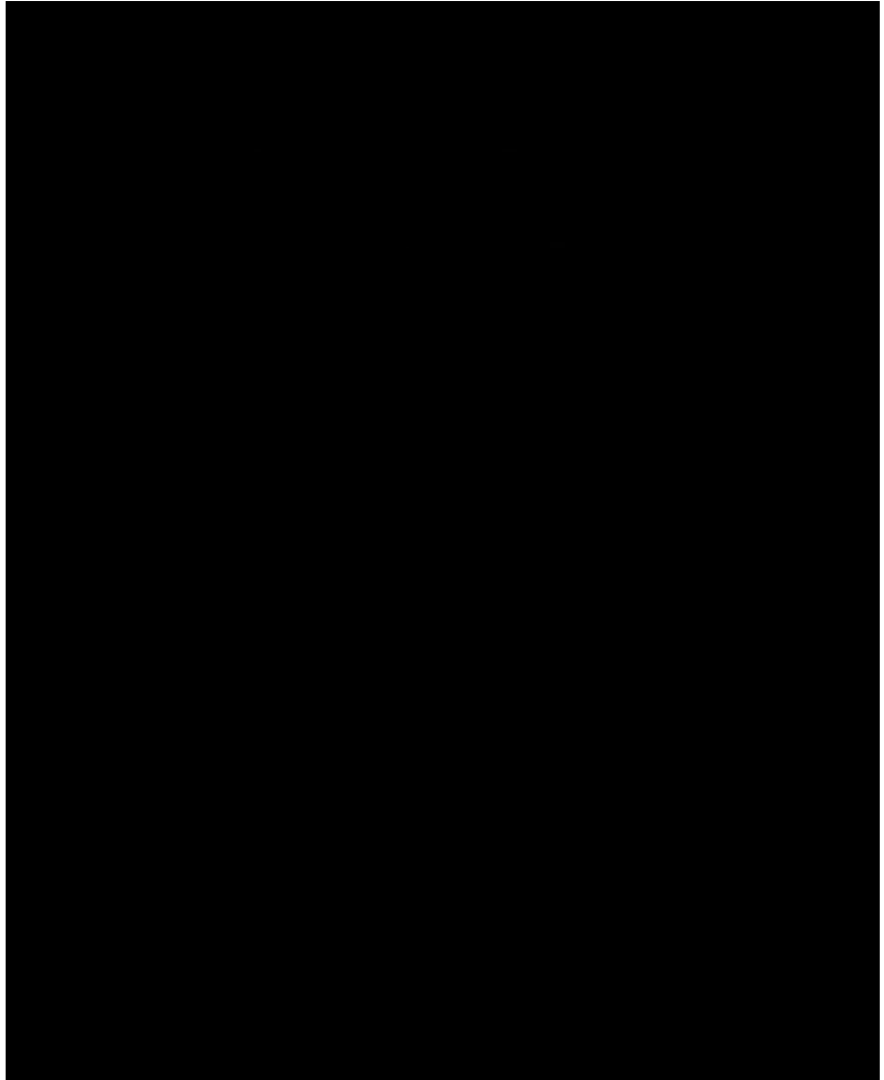


Results over 49 games



Analysis

- Strengths:
  - End-to-End Learning over multiple games
  - Beats human performance on most of the games
  - Richer Rewards
- Weakness:
  - Long Training
  - DQN limited to finite discrete actions



Deep Reinforcement Learning Summary

- Represent value function by deep Q-network with weights w

$$Q(s, a, w) \approx Q^{\pi}(s, a)$$

- Define objective function by mean-squared error in Q-values

$$\mathcal{L}(w) = \mathbb{E}\left[\left(\underbrace{r + \gamma \max_{a'} Q(s', a', w)}_{\text{target}} - Q(s, a, w)\right)^2\right]$$

- Leading to the following Q-learning gradient descent

$$\frac{\partial \mathcal{L}(w)}{\partial w} = \mathbb{E}\left[\left(r + \gamma \max_{a'} Q(s', a', w) - Q(s, a, w)\right) \frac{\partial Q(s, a, w)}{\partial w}\right]$$

Optimizations on DQN Since Then

- Double DQN: Remove Upward bias
  - Current Q-network w: Used to select actions
  - Older Q-network w-: Used to evaluate actions
- Prioritized replay : Weight experience according to TD-error
  - Store experience in priority queue according to DQN error
- Policy Gradient Methods
- Asynchronous RL
  - Joint Training using Parameter Sharing on Distributed Scale
  - GORILA

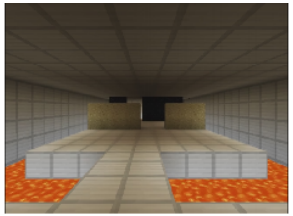
Several game domains

ALE
(Breakout)

GVG-AI
(Zelda)

VizDoom

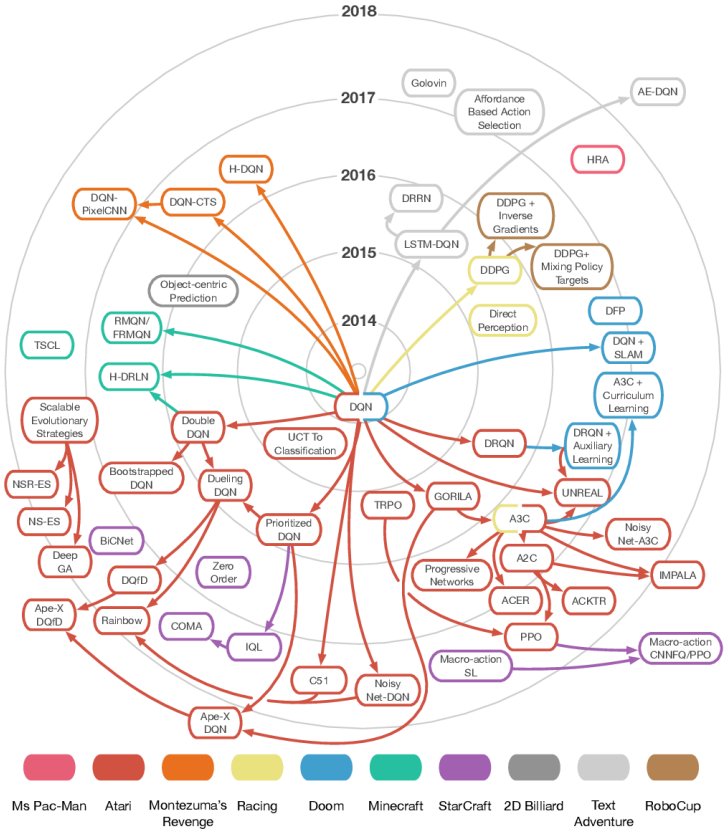
TORCS



Project Malmo
(Minecraft)

StarCraft: Brood War

Several new algorithms



Policy Gradient Methods

What are Policy Gradient Methods?

- Before – Action-value methods:
  - Learn the values of actions and then select actions based on their estimated action-values.
  - The policy was generated directly from the value function.
  - The policies would not even exist without the action-value estimates.
- Now – Methods learn a parameterized policy that can select actions without consulting a value function:
  - Search directly for the optimal policy π^*.
  - The parameters of the policy are called policy weights.

Why?

- Learning a policy could have fewer parameters.
- Choice of parameterization gives you freedom to act randomly in sophisticated ways.
- The best policy may be stochastic.
  - But if not, policy gradient methods can still approach determinism.
- Natural extensions to continuous action spaces.
- Access to techniques from the optimization literature.

Notation: Policy Parameter $\boldsymbol{\theta}$

- The policy parameter vector is $\boldsymbol{\theta} \in \mathbb{R}^d$.
- The policy is written as:
  - $\pi(a|s, \boldsymbol{\theta})$, which represents the probability that action a is taken in state s with policy weight vector $\boldsymbol{\theta}$.
  - The policy can be parameterized in any way, as long as it is differentiable with respect to its parameters:
    - That is, as long as $\nabla\pi(a|s, \boldsymbol{\theta})$ with respect to the components of $\boldsymbol{\theta}$ exists and is finite for all $s \in S, a \in A(s)$, and $\boldsymbol{\theta} \in \mathbb{R}^d$.
  - For example, $\boldsymbol{\theta}$ might be the vector of all the connection weights of an Artificial Neural Network.

What are Policy Gradient Methods?

- Policy Gradient Methods are methods for learning the policy weights using the gradient of some performance measure $J(\boldsymbol{\theta})$ with respect to the policy weights:
  - They seek to maximize performance and so the policy weights are updated using gradient ascent in $J(\boldsymbol{\theta})$:
    - $\theta_{t+1} = \theta_t + \alpha \cdot \widehat{\nabla J(\theta)}$
    - Where $\widehat{\nabla J(\theta)}$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument θ_t.

Which performance measure $J(\boldsymbol{\theta})$?

- Recall that the optimal policy is the policy that achieves maximum future return.
- The Policy Gradient Theorem:
  - $J(\boldsymbol{\theta}) \stackrel{\text{def}}{=} v_{\pi_\theta}(s_0)$
- where:
  - v_{π_θ} is the true value function for π_θ, the policy determined by $\boldsymbol{\theta}$.
- Performance is defined as the value of the start state under the parameterized policy.
- Assume discrete and finite set of actions.
- Can use a deep neural network as the policy approximator!

Policy Approximation

- The policy can be parameterized in any way provided $\pi(a|s, \boldsymbol{\theta})$ is differentiable with respect to its weights.
- To ensure exploration, we generally require that the policy never becomes deterministic and so $\pi(a|s, \boldsymbol{\theta})$ is in interval (0, 1).
- Use SoftMax so that the most preferred actions in each state are given the highest probability of being selected:

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{\exp(h(s, a, \boldsymbol{\theta}))}{\sum_b \exp(h(s, b, \boldsymbol{\theta}))}$$

Policy Gradient and Actor-Critic Methods

- All methods that follow this general schema are called Policy Gradient Methods, whether or not they also learn an approximate value function.
- Methods that learn approximations to both policy and value functions are often called Actor–Critic Methods:
  - 'actor' is a reference to the learned policy, and 'critic' refers to the learned value function, usually a state-value function.
  - A value function may be used to learn the policy weights but this is not *required* for action selection.
  - If a method uses a learned value function as well, then the value function's weight vector is denoted $\boldsymbol{w} \in \mathbb{R}^d$ as usual.

Algorithms in Policy-Based RL

- REINFORCE:
  - Episodic updates.
  - Maximize the loss of expected reward under the objective.
- Actor-Critic:
  - Updates at each step.
  - Critic approximates the value function.
  - Actor approximates the policy.
- Asynchronous Advantage Function Actor-Critic – A3C:
  - Uses Advantage Function Estimate: $A(s,a,w) = Q(s,a,w) - V(s)$
  - Replacing the need of replay memory by using parallel agents running on CPU.
  - Relies on different exploration behavior of the parallel agents.

REINFORCE: Monte Carlo Policy Gradient

- We want a quantity that we can sample on each time step whose expectation is equal to the gradient.
- We can then perform stochastic gradient ascent with this quantity:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \gamma^t G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}.$$

REINFORCE Properties

- On-policy method.
- Uses the complete return from time t, which includes all future rewards until the end of the episode.
- REINFORCE is thus a Monte Carlo algorithm and is only well-defined for the episodic case with all updates made in retrospect after the episode is completed.

REINFORCE Algorithm

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ (G_t)
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$

Actor-Critic Methods

- Methods that learn approximations to both policy and value functions are called actor-critic methods.
- Actor refers to the learned policy.
- Critic refers to the learned value function, which is usually a state-value function.
- The critic is bootstrapped – the state-values are updated using the estimated state-values of subsequent states.
- The number of steps in the actor-critic method controls the degree of bootstrapping.

One-step Actor-Critic Update Rules

- On-policy method.
- The state-value function update rule is the TD(0) update rule.
- The policy function update rule is shown below.
- For n-step Actor-Critic, simply replace $G_t^{(1)}$ with $G_t^{(n)}$

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\left(G_t^{(1)} - \hat{v}(S_t,\mathbf{w})\right)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} \\ &= \boldsymbol{\theta}_t + \alpha\left(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\right)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}.\end{aligned}$$

One-step Actor-Critic Algorithm

One-step Actor-Critic (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta}), \forall a \in \mathcal{A}, s \in \mathcal{S}, \boldsymbol{\theta} \in \mathbb{R}^n$
Input: a differentiable state-value parameterization $\hat{v}(s,\mathbf{w}), \forall s \in \mathcal{S}, \mathbf{w} \in \mathbb{R}^m$
Parameters: step sizes $\alpha > 0$, $\beta > 0$

Initialize policy weights $\boldsymbol{\theta}$ and state-value weights $\mathbf{w}$
Repeat forever:
  Initialize S (first state of episode)
  $I \leftarrow 1$
  While S is not terminal:
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action A, observe S', R
    $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$ (if S' is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
    $\mathbf{w} \leftarrow \mathbf{w} + \beta \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha I \delta \nabla_{\boldsymbol{\theta}} \log \pi(A|S,\boldsymbol{\theta})$
    $I \leftarrow \gamma I$
    $S \leftarrow S'$

Asynchronous Advantage Actor-Critic – A3C

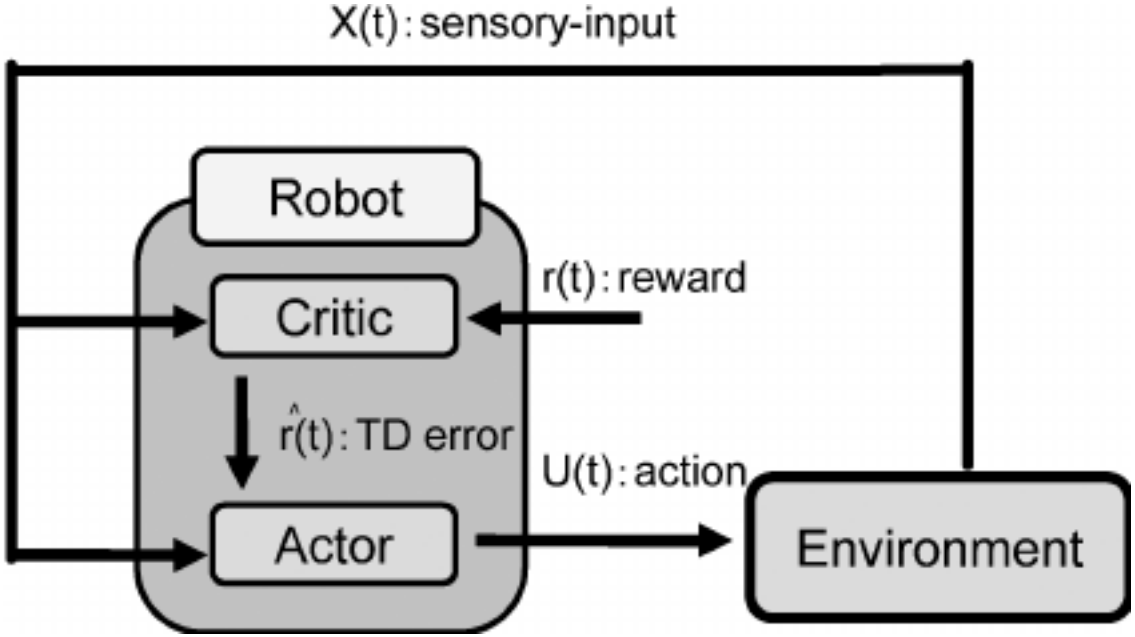
- The advantage is the value that tells us if there is an improvement in a certain action compared to the expected average value of that state based on .
- The advantage formula is: $A(s,a) = Q(s,a) - V(s)$

  q value for action a in state s

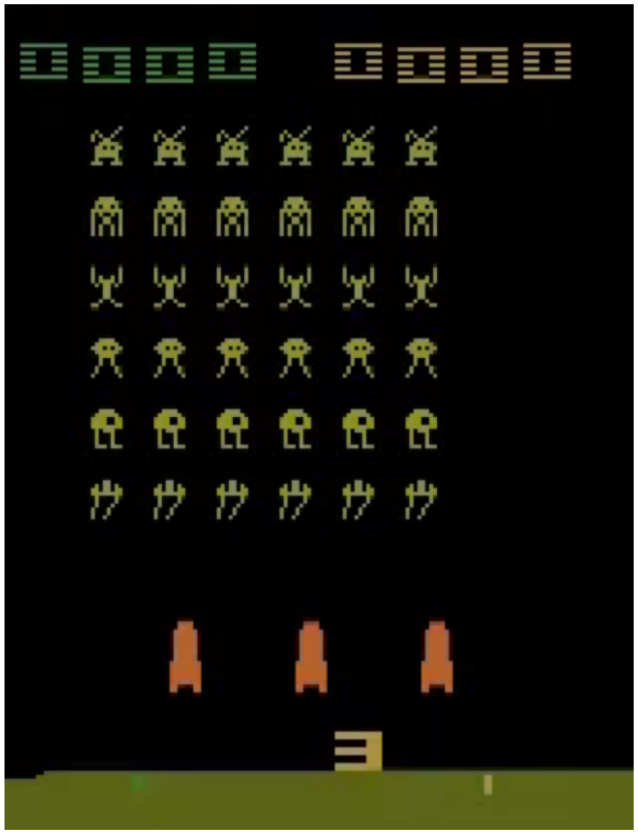
  average value of that state

  - The Q(s, a) refers to the Q value or the expected future reward of taking an action at a certain state.
  - The V(s) refers to the value of being in a certain state. The goal of the model is to maximize the advantage value.

Asynchronous Advantage Actor-Critic – A3C

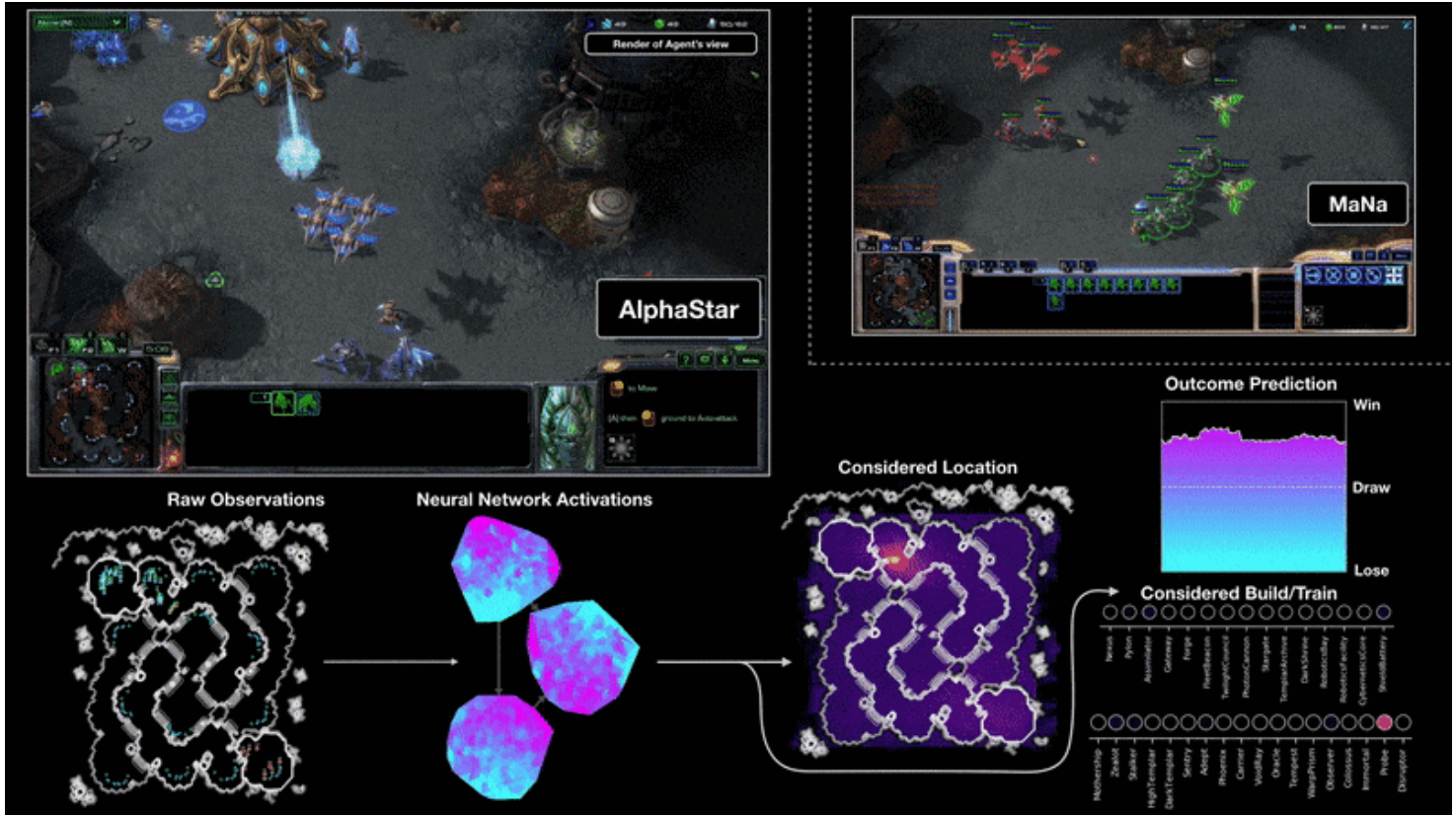


Asynchronous Advantage Actor-Critic – A3C



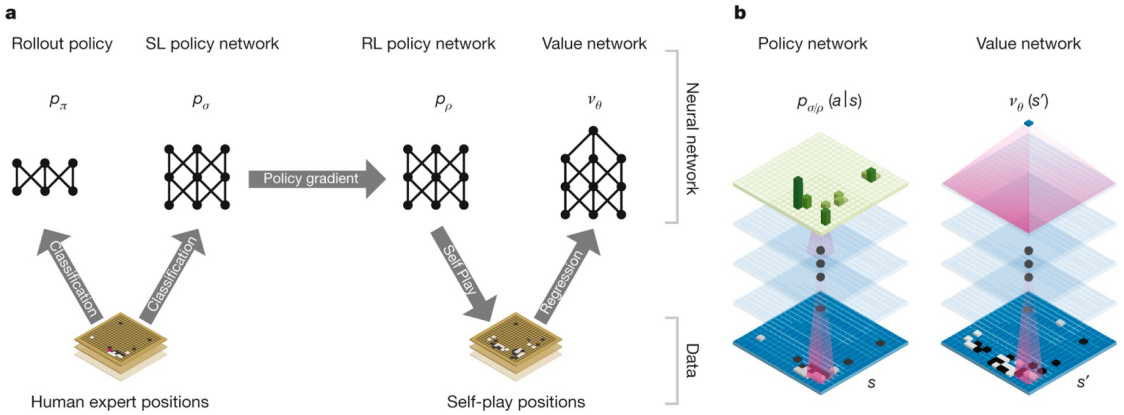
A3C – Collecting energy in Doom

AlphaStar


Render of Agent's view
AlphaStar
MaNa
Outcome Prediction
Win
Draw
Lose
Considered Location
Raw Observations
Neural Network Activations
Considered Build/Train

AlphaGo

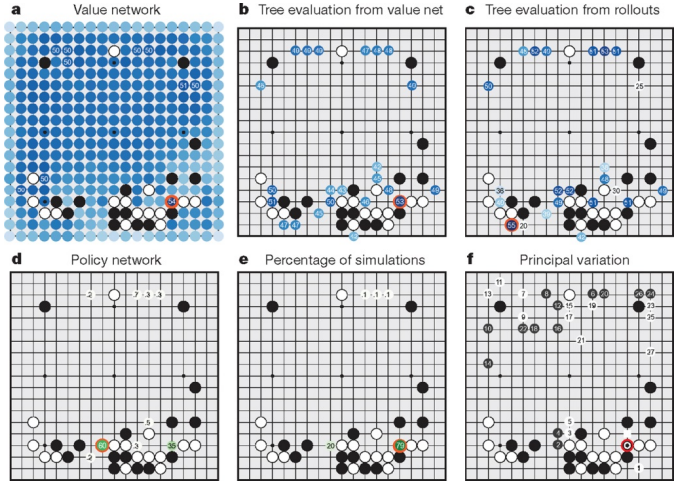
Neural network training pipeline and architecture



D Silver *et al. Nature* **529,** 484–489 (2016) doi:10.1038/nature16961

nature

How AlphaGo (black, to play) selected its move in an informal game against Fan Hui



D Silver *et al. Nature* **529,** 484–489 (2016) doi:10.1038/nature16961

nature

Conclusion

Conclusion

- Reinforcement learning is a promising technology, and lately there have been many refinements aimed at increasing the diffusion and use of this technology.

Scope / Future

- Multi-agent Deep RL
  - Share Parameters!!! (Naive approach)
- Hierarchical Deep Reinforcement Learning
  - Road to General AI!

Quotes from the Maestro

- If intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake.

  -Yann Lecun