Deep Reinforcement Learning

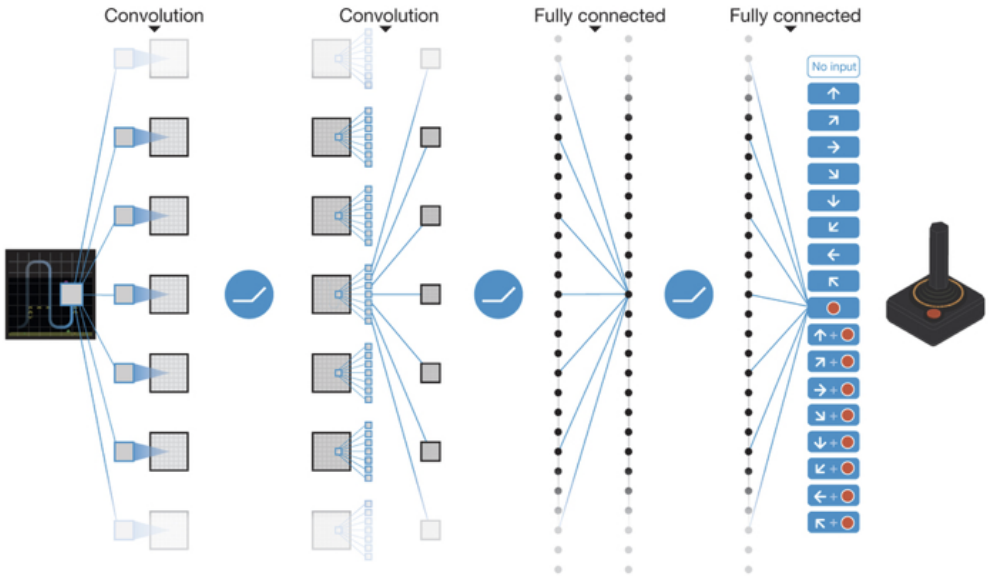
10 - Deep Q-Network

Prof. Dr. Malte Schilling

Autonomous Intelligent Systems Group

Overview Lecture

- Function Approximation in Reinforcement Learning
  - Difficulties, due to characteristics of RL
  - Approaches
- Deep Q-Networks (for playing ATARI)



(Mnih u. a. 2015)

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve large problems, e.g.

- Backgammon: 10^{20} states
- Go: 10^{170} states
- Helicopter: continuous state space
- Robots: real world tasks

How can we apply our methods for prediction and control?

(Hasselt und Borsa 2021)

Function approximation and deep reinforcement learning

- The policy, value function, model, and agent state update are all functions.
- Goal: learn these from experience.
- But, if there are too many states, we need approximation.

When using neural networks to represent these functions, this is called deep reinforcement learning.

While the term is fairly new (around 8 years) — the combination is fairly old (more then 50 years).

(Hasselt und Borsa 2021)

Value Function Approximation

So far, we mostly considered lookup tables

- Every state s has an entry $v(s)$ or every state-action pair s, a has an entry $q(s, a)$

Problem with large MDPs:

- There are too many states and/or actions to store in memory.
- It is too slow to learn the value of each state individually.
- Individual environment states are often not fully observable

(Hasselt und Borsa 2021)

Generalization over States

In order to deal with continuous or large state spaces, we want to generalize. For this, we use *Function Approximation* to estimate value functions:

$$v_{\mathbf{w}}(s) \approx v_{\pi}(s)$$

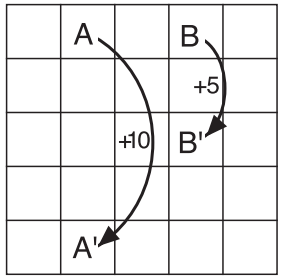
$$q_{\mathbf{w}}(s, a) \approx q_{\pi}(s, a)$$

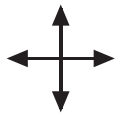
- Learn about a small number of training states from experiences and update parameter $\mathbf{w}$ that describe our function approximation.
- Generalize these experiences to new, similar situations.

As a basic idea for *Deep Reinforcement Learning*: use Neural Networks for function approximation.

Recap – Value Function in Grid World Example

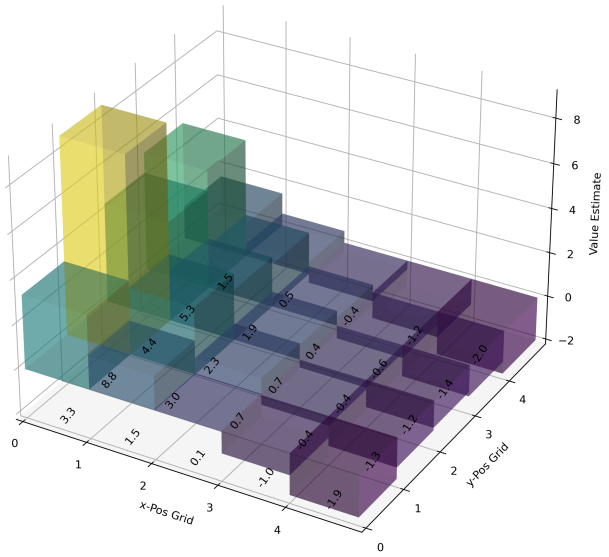
- We considered the simple discrete grid environment.
- For a real robot: We would consider a continuous state space (position).

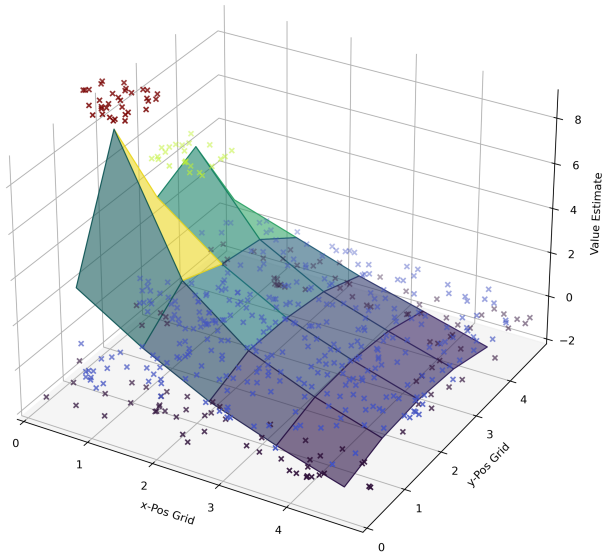




Actions

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0


Value Estimate
x-Pos Grid
y-Pos Grid
8.8
4.4
5.3
1.5
-3.3
3.0
2.3
1.9
1.5
0.5
0.7
0.4
0.7
-0.4
-0.4
-0.6
-0.4
-1.2
-1.0
-1.3
-1.4
-1.2
-1.9
-2.0


Value Estimate
x-Pos Grid
y-Pos Grid

Classes of Function Approximation

- **Tabular**: a table with an entry for each MDP state
- **State aggregation**: Partition environment states (or observations) into a discrete set
- Linear function approximation
  - Consider fixed agent state update
  - Fixed feature map $\mathbf{x} : S \to \mathbb{R}^n$
  - Values are linear function of features: $v_{\mathbf{w}}(s) = \mathbf{w}^\intercal \mathbf{x}(s)$
- **Differentiable function approximation**: $v_{\mathbf{w}}(s)$ is a differentiable function of $\mathbf{w}$ that could be non-linear
  - e.g., a convolutional neural network that takes pixels as input.
  - Another interpretation: features are not fixed, but learnt.

(Hasselt und Borsa 2021)

Classes of Function Approximation

In principle, any function approximator can be used, but RL has specific properties:

- Experience is not i.i.d. — successive time-steps are correlated
- Agent's policy affects the data it receives

Regression targets can be **non-stationary**

- …because of changing policies (which can change the target and the data!)
- …because of bootstrapping
- …because of non-stationary dynamics (e.g., other learning agents)
- …because the world is large (never quite in the same state)

(Hasselt und Borsa 2021)

Classes of Function Approximation

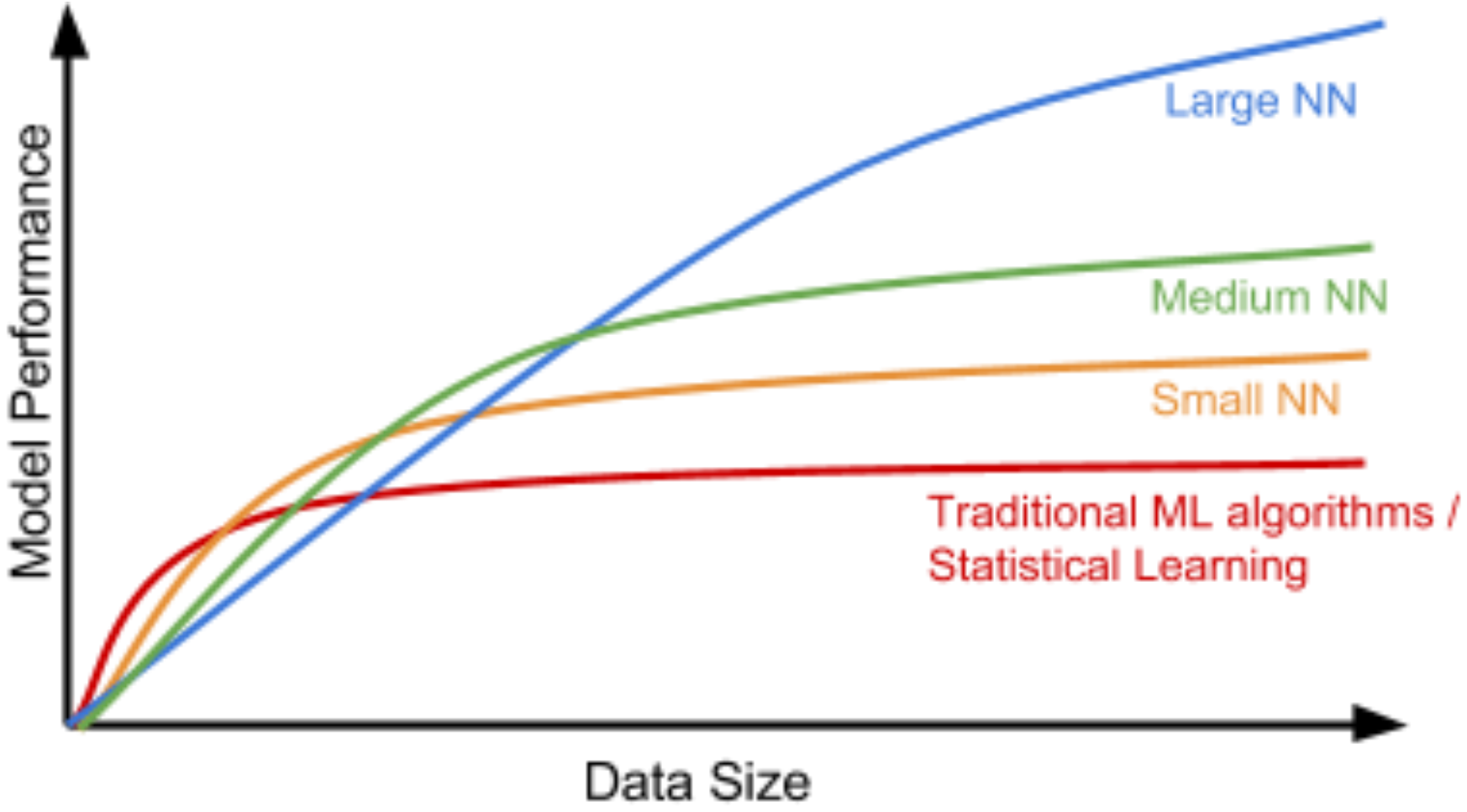
The choice of function approximation depends on the task and yourgoals:

- **Tabular**: good theory but does not scale/generalise
- **Linear**: reasonably good theory, but requires good features
- **Non-linear**: less well-understood, but scales well. Flexible, and less reliant on picking good features first (e.g., by hand)

(Deep) neural nets often perform quite well, and are a popular choice.

(Hasselt und Borsa 2021)

The data scale versus the model performance.



(Weng 2018)

Recap – Gradient Descent

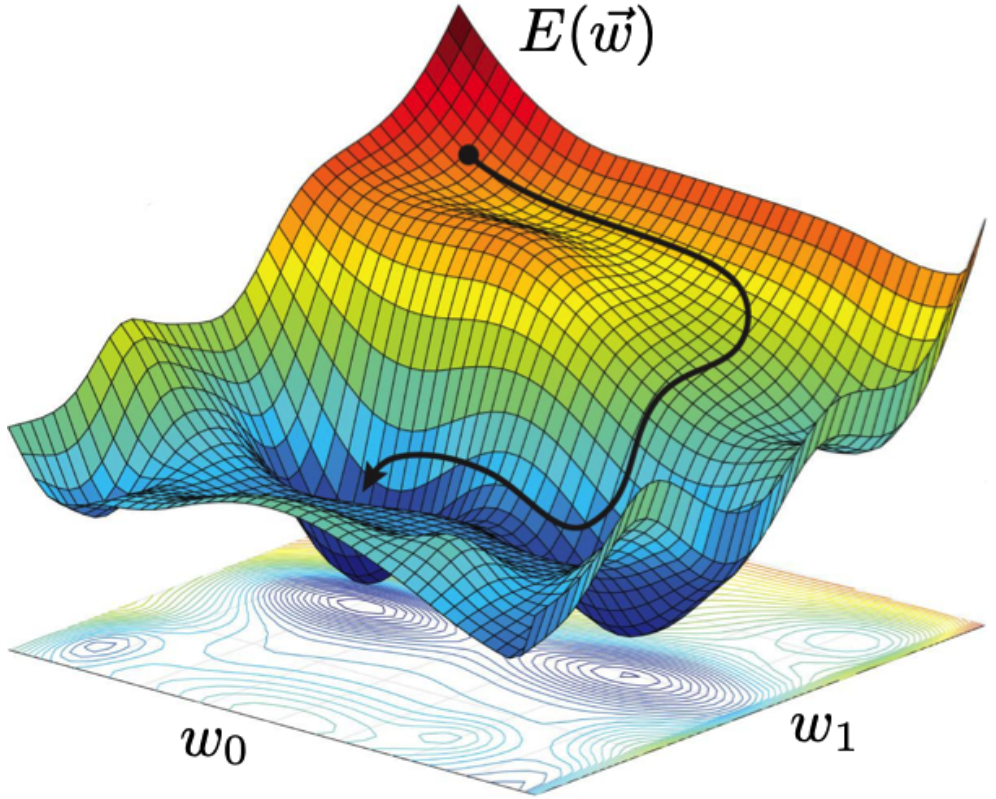
For a differentiable function $J(\mathbf{w}$ the gradient is given as

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{pmatrix}$$

Goal: Minimize $J(\mathbf{w})$

Approach: Move $\mathbf{w}$ in the direction of negative gradient (step size parameter α)

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$



Approximate Values By Stochastic Gradient Descent

Goal: find $\mathbf{w}$ that minimizes the difference between $v_{\mathbf{w}}(s)$ and $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_{S\sim d}\left[(v_\pi(S) - v_{\mathbf{w}}(S))^2\right]$$

where d is a distribution over states (induced by the policy and dynamics)

Gradient descent:

$$\Delta\mathbf{w} = -\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) = \alpha\mathbb{E}_d(v_\pi(S) - v_{\mathbf{w}}(S))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S)$$

Stochastic Gradient Descent:

Sample the gradient: $\Delta\mathbf{w} = \alpha(G_t - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$

Note: MC return G_t is a sample for $v_\pi(S_t)$; $\nabla_v(S_t)$ is short hand for $\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)\big|_{\mathbf{w}=\mathbf{w}_t}$

(Hasselt und Borsa 2021)

Linear model-free prediction

Feature Vectors as Observations

Represent the observed state as a feature vector:

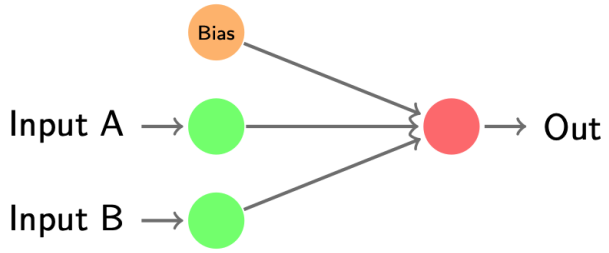
$$\mathbf{x}(S) = \begin{pmatrix} x_1(S) \\ \vdots \\ x_n(S) \end{pmatrix}$$

For example:

- Distance of robot from landmarks
- Trends in the stock market
- Piece and pawn configurations in chess

(Silver 2015)

Linear Model Formulation (as a simple Neural network)



$$a(\vec{x}) = \vec{w}^T \vec{x} = \begin{pmatrix} w_0 & w_1 & w_2 \end{pmatrix} \begin{pmatrix} 1 \\ x_A \\ x_B \end{pmatrix} \Leftrightarrow$$

Error $E(\vec{w})$

Sample Input → $\sum_i (\vec{w}^1)^T \vec{x}^{(i)}$ → → *Predicted Output* − *Sample Label*

weighted sum | activation fct | **Score**

Forward = Computation

Linear Value Function Approximation

Approximate value function by a linear combination of features

$$v_{\mathbf{w}}(s) = \mathbf{w}^\intercal \mathbf{x}(s) = \sum_{j=1}^{n} x_j(s) w_j$$

Objective function ('loss') is quadratic in $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}_{S \sim d}\Big[(v_\pi(S) - \mathbf{w}^\intercal \mathbf{x}(S))^2\Big]$$

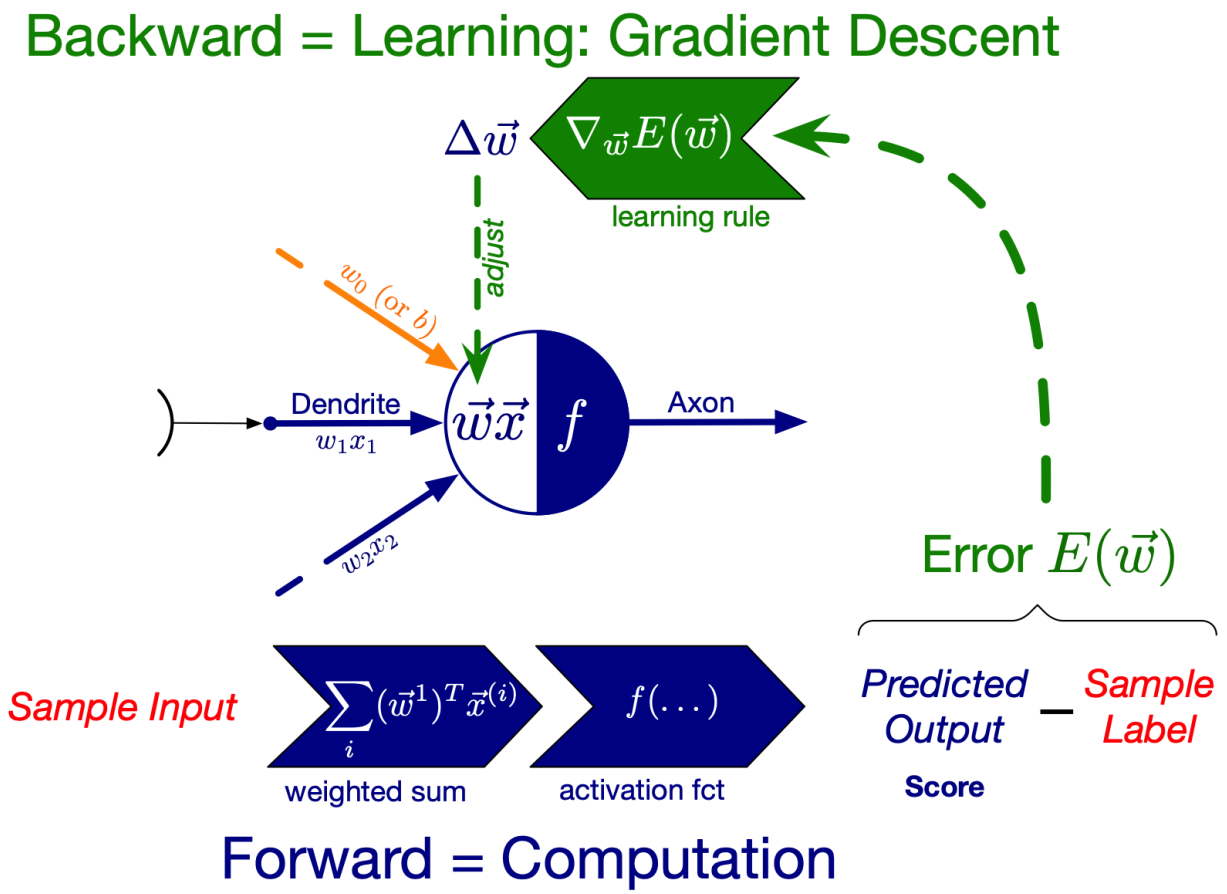
Stochastic gradient descent converges on global optimum

Update rule is simply (Update = step-size $\times$ prediction error $\times$ feature vector)

$$\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \mathbf{x}(S_t) = \mathbf{x}_t \Rightarrow \Delta \mathbf{w} = \alpha(v_\pi(S_t) - v_{\mathbf{w}}(S_t))\mathbf{x}_t$$

(Hasselt und Borsa 2021)

Computation in Linear Model: Overview Learning Cycle



Incremental prediction algorithms

We can't update towards the true value function $v_\pi(s)$. Therefore, we (as done before) substitute a target for $v_\pi(s)$:

- For MC, the target is the return G_t

$$\Delta \mathbf{w}_t = \alpha(G_t - v_{\mathbf{w}}(s))\nabla_{\mathbf{w}} v_{\mathbf{w}}(s)$$

- For TD, the target is the TD target $R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1})$:

$$\Delta \mathbf{w}_t = \alpha(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

(Hasselt und Borsa 2021)

Monte-Carlo with Value Function Approximation

- In MC: The return G_t is an unbiased sample of $v_\pi(s)$.
- Can therefore apply “supervised learning” to (online) “training data”: $(S_0, G_0), \ldots, (S_t, G_t)$
- For example, using **linear Monte-Carlo policy evaluation**

$$\begin{aligned}\Delta \mathbf{w}_t &= \alpha(G_t - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) \\ &= \alpha(G_t - v_{\mathbf{w}}(S_t))\mathbf{x}_t\end{aligned}$$

- Linear Monte-Carlo evaluation converges to the global optimum
- Even when using non-linear value function approximation it converges (but perhaps to a local optimum)

(Hasselt und Borsa 2021)

TD Learning with Value Function Approximation

- The TD-target $R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1})$ is a biased sample of true value $v_\pi(S_t)$.
- We still can apply supervised learning to "training data"
  $(S_0, R_1 + \gamma v_{\mathbf{w}}(S_1)), \dots, (S_t, R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}))$
- For example, using **linear TD**

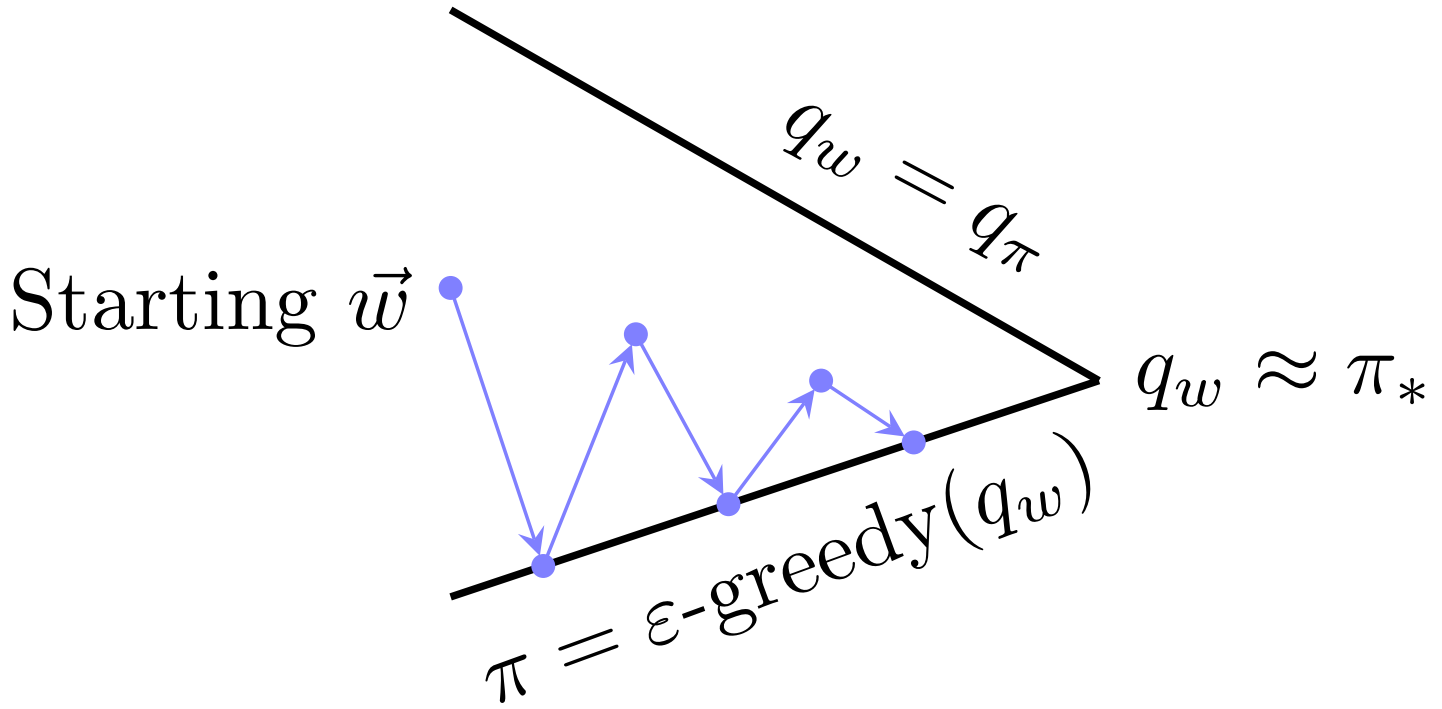
$$\begin{aligned}\Delta \mathbf{w}_t &= \alpha \underbrace{(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t))}_{=\delta_t, \text{TD error}} \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) \\ &= \alpha \delta_t \mathbf{x}_t\end{aligned}$$

- This is akin to a non-stationary regression problem
- But: target depends on our parameters – therefore called *semi-gradient*!

(Hasselt und Borsa 2021)

Control with Value Function Approximation

Control with Value Function Approximation



Policy evaluation: Approximate policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement: ε-greedy policy improvement

Action-Value Function Approximation

- Approximate the action-value function $\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$
- Minimize mean-squared error between approximate action-value function $\hat{q}(S, A, \mathbf{w})$ and true action-value function $q_\pi(S, A)$:

$$J(\mathbf{w}) = \mathbb{E}_\pi[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2]$$

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2}\nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

$$\Delta\mathbf{w} = \alpha\Big(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

(Silver 2015)

Linear Action-Value Function Approximation

- Represent state and action by a feature vector

$$\mathbf{x}(S, A) = \begin{pmatrix} x_1(S, A) \\ \vdots \\ x_n(S, A) \end{pmatrix}$$

- Represent action-value function by linear combination of features

$$\hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)^\intercal \mathbf{w} = \sum_{j=1}^{n} x_j(S, A) w_j$$

- Stochastic gradient descent update

$$\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A), \Delta \mathbf{w} = \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))\mathbf{x}(S, A)$$

(Silver 2015)

Incremental Control Algorithms

Like prediction, we must substitute a target for $q_\pi(S, A)$:

- For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For TD, the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$:

$$\Delta \mathbf{w}_t = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

(Silver 2015)

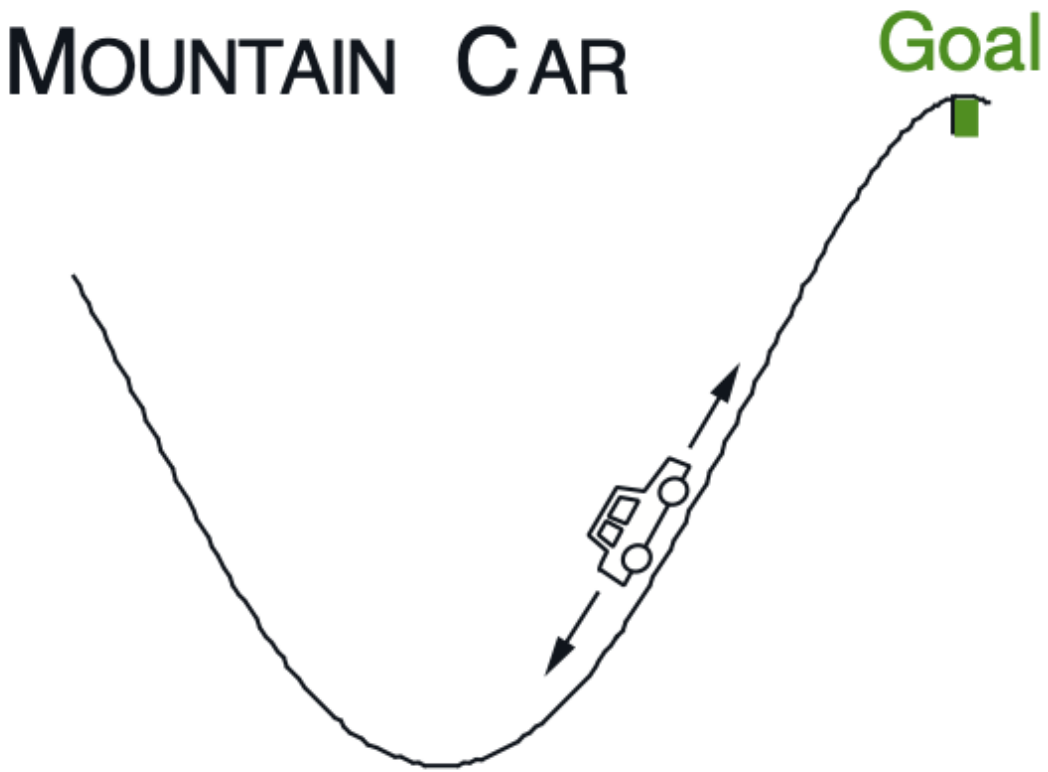
Semi-Gradient SARSA Algorithm

Input: a differentiable function $\hat{q}$ and a policy π
Initializ parameter vector $\vec{w}$
for *episode* $= 1, \ldots, M$ **do**
  $S, A \leftarrow$ initial state and action of episode episode
  for $t = 1, \ldots, T$ **do**
    Take action A, observe R, S'
    if S' *is terminal* **then**
      $\vec{w} \leftarrow \vec{w} + \alpha[R - \hat{q}(S, A, \vec{w})]\nabla\hat{q}(S, A, \vec{w})$
      GoTo next episode
    end
    Choose A' as a function of $\hat{q}(S', , \vec{w})$, e.g. ε-greedy
    $\vec{w} \leftarrow \vec{w} + \alpha[R + \gamma\hat{q}(S', A', \vec{w}) - \hat{q}(S, A, \vec{w})]\nabla\hat{q}(S, A, \vec{w})$
    $S \leftarrow S', A \leftarrow A'$
  end
end

Environment interaction

Example: SARSA for Mountain Car

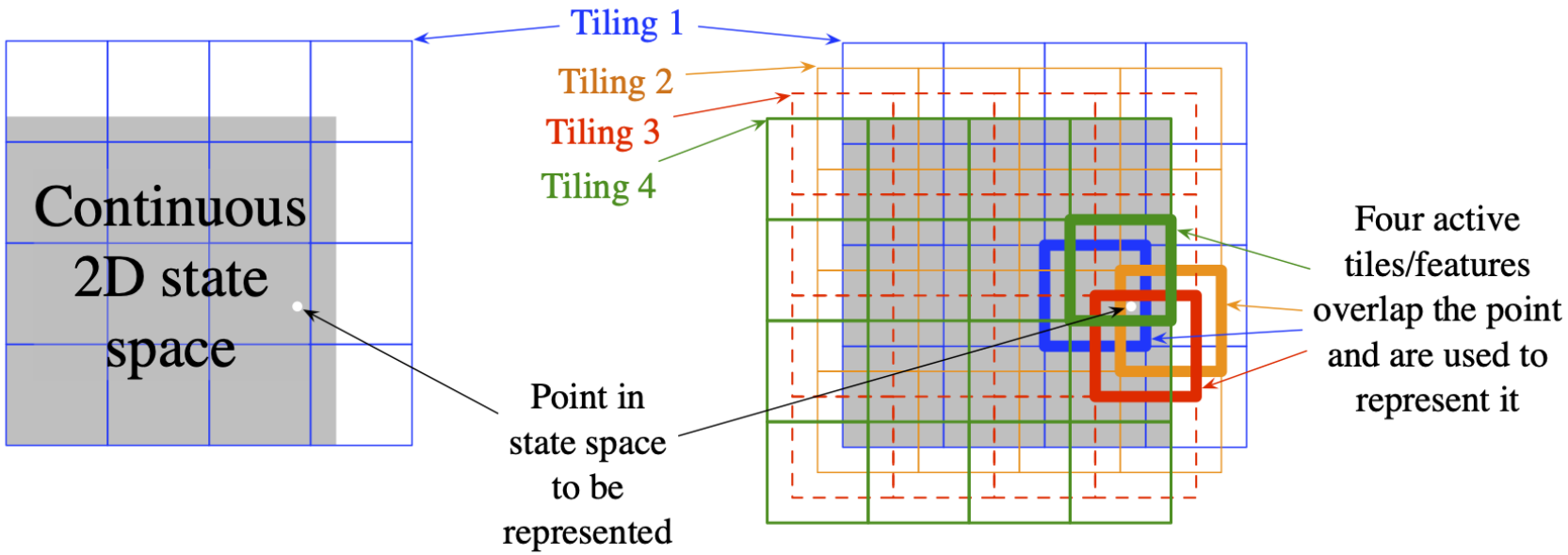
- Observation: position, velocity
- Action: discrete = left, none, right
- Reward: $r = -1$, goal is to terminate as quickly as possible
- Episode terminates when car reaches the flag (or max steps or too far left)
- Simplified longitudinal car physics
- Init: Random position, zero velocity
- car is underpowered, requires swing-up



(Sutton und Barto 2018)

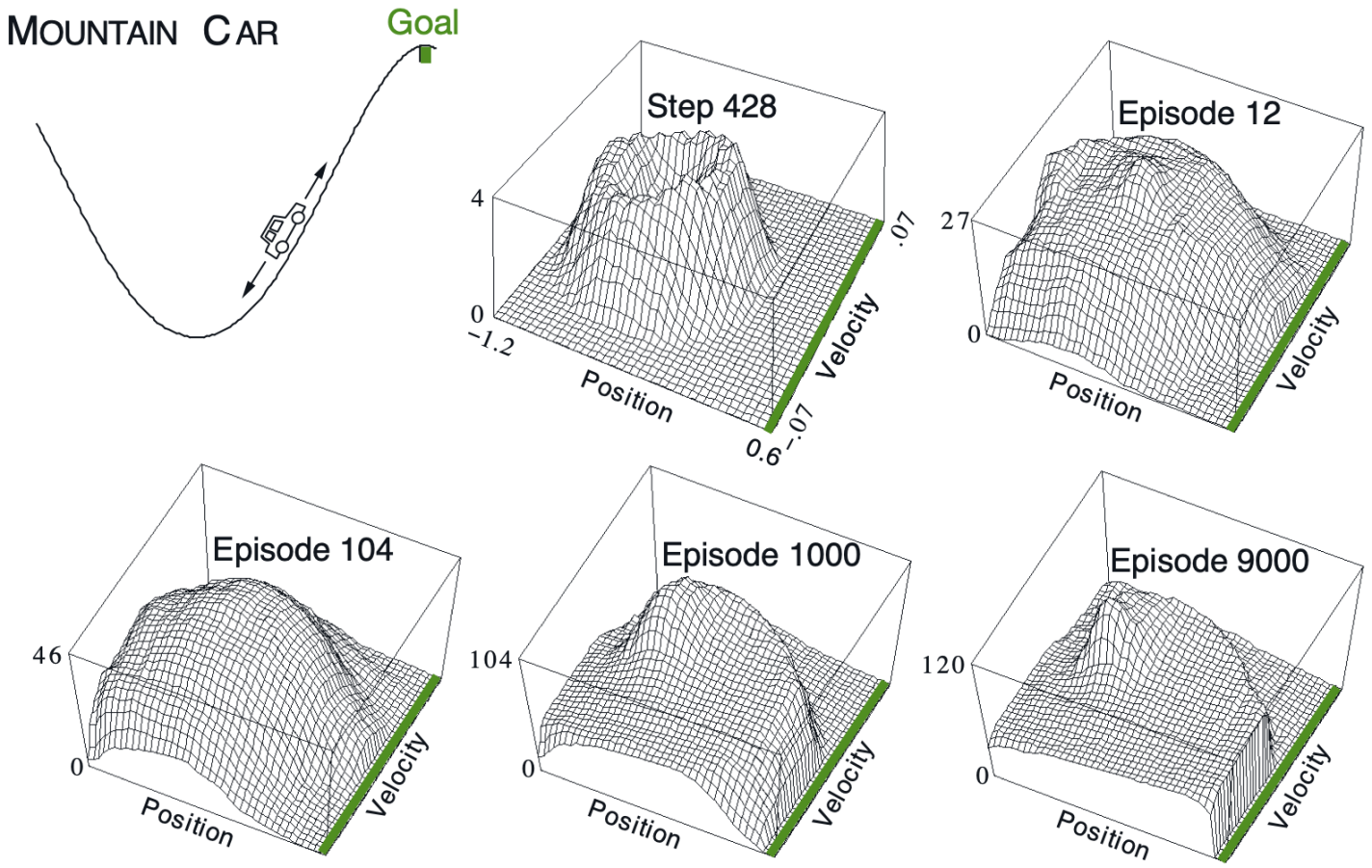
Coarse Coding for Mountain Car

- Problem space is grouped into (overlapping) partitions / tiles.
- Performs a discretization of the problem space.
- Function approximation used for interpolation between tiles.



(Sutton und Barto 2018)

Linear SARSA with Coarse Coding in Mountain Car



(Sutton und Barto 2018)

Deep Reinforcement Learning

Batch Reinforcement Learning

- Gradient descent is simple and appealing
- But it is not sample efficient
- Batch methods seek to find the best fitting value function
- Given the agent's experience ("training data")

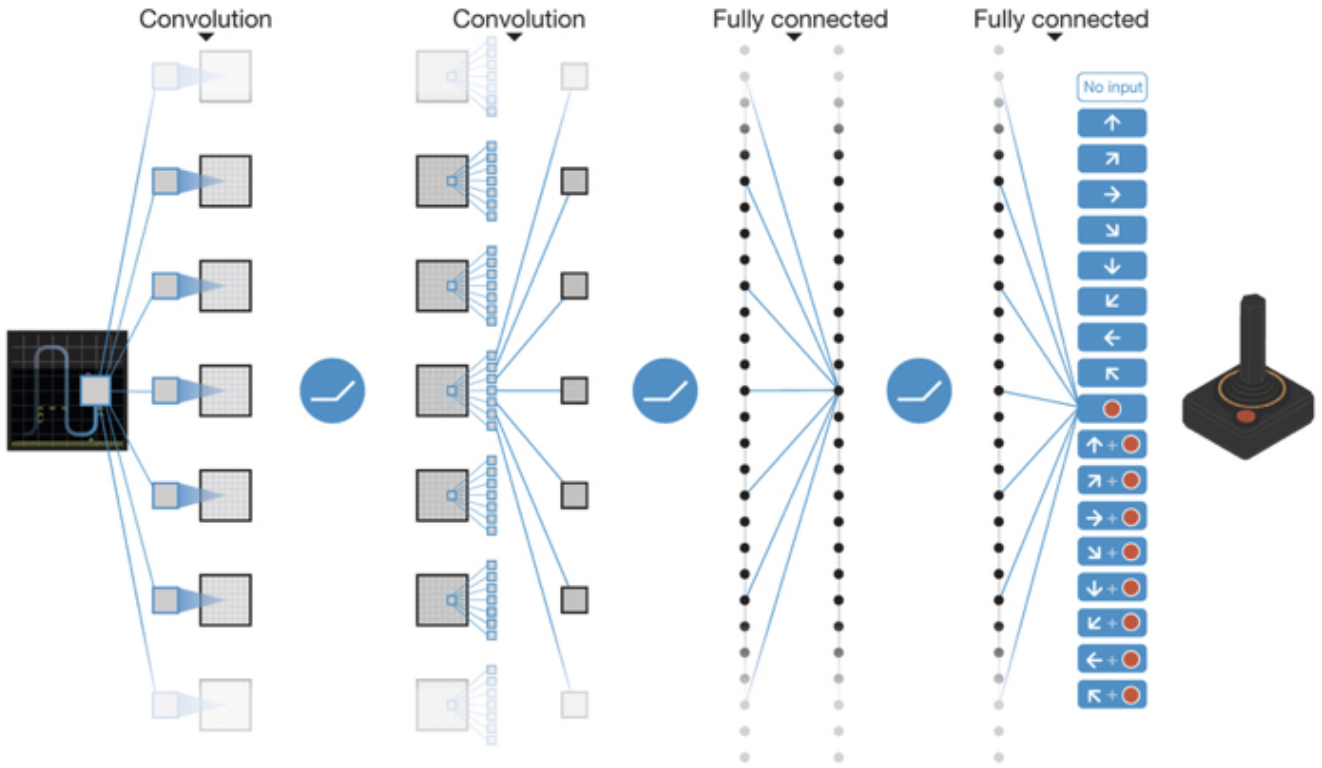
(Silver 2015)

DQN

Bringing together

- Deep Neural Networks for function approximation
- and Q-Learning for action-value function learning

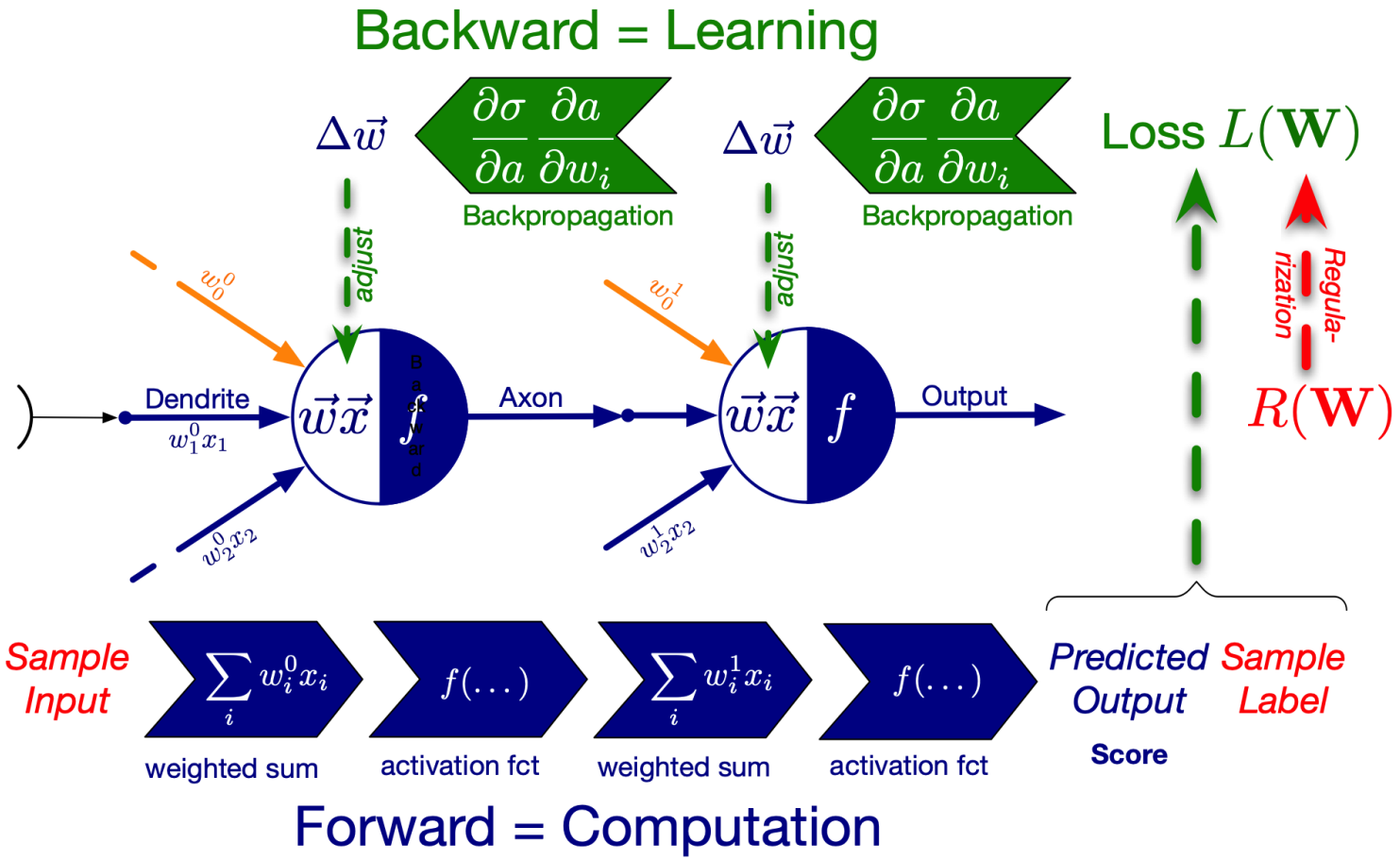
DQN Architecture Overview



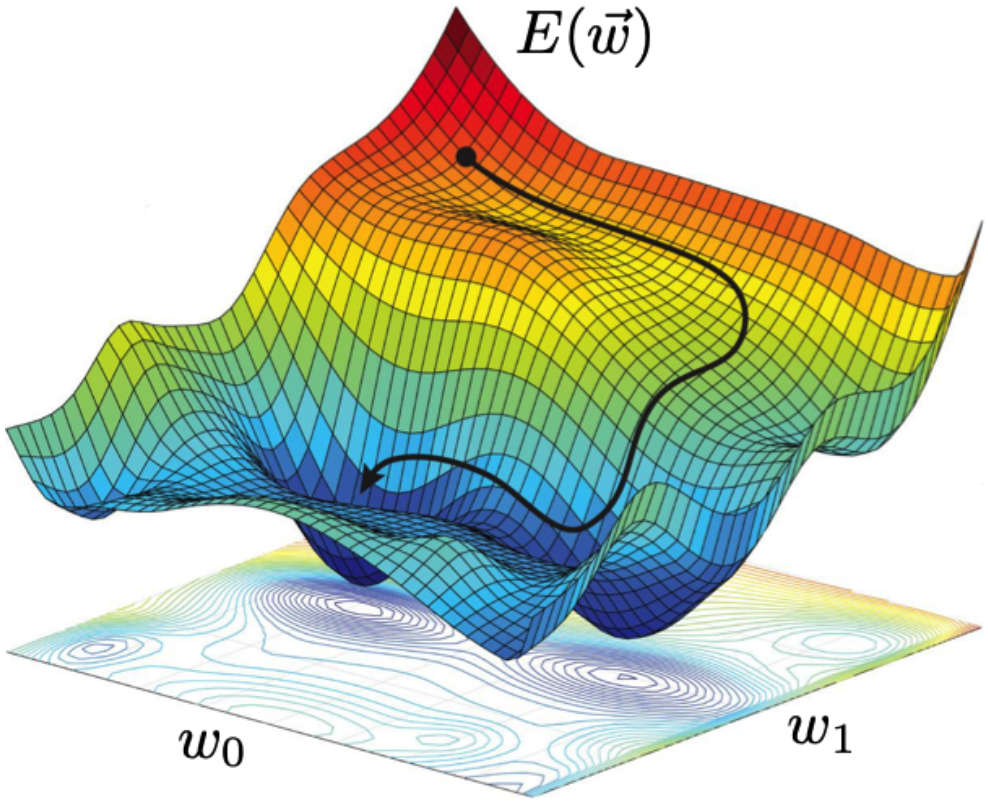
“we developed a novel agent, a deep Q-network (DQN), which is able to combine reinforcement learning with a class of artificial neural network known as deep neural networks.”

(Mnih u. a. 2015)

Recap – Overview Learning Cycle



Recap – Gradient Descent: Iterative Search for Minimum



Iterative optimization algorithm

- start from an initial point $\mathbf{u}$ (initial guess) on the error function
- Iterate (k = iteration, η = learning rate):

Determine the gradient at that point and make a step: $\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla_{\mathbf{w}} E^{(i)}(\mathbf{w})$

Until: $\nabla_{\mathbf{w}} E^{(i)}(\mathbf{w}) \approx 0$

Then we found a minimum.

Summary Q-Learning

Q-Learning is an off-policy approach for learning of action-values $q(s, a)$:

- Next action is chosen using behaviour policy $A_{t+1} \sim b(\cdot|S_t)$
- But we consider alternative successor action $a' \sim \pi(\cdot|S_t)$
- And update $q(S_t, A_t)$ towards value of alternative action

$$q'(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha_t \Big(R_{t+1} + \gamma \max_{a'} q(S_{t+1}, a') - q(S_t, A_t) \Big)$$

(Silver 2015)

Goal of DQN: Approximation of Q-Function

- Q-learning can be used to find an optimal action-selection policy for any given (finite) Markov decision process (MDP).
- It works by learning an action-value function that ultimately gives the expected utility of taking a given action in a given state and following the optimal policy thereafter.
- One of the strengths of Q-learning is that it is able to compare the expected utility of the available actions without requiring a model of the environment.
- Q-learning learns estimates of the optimal Q-values of an MDP, which means that behavior can be dictated by taking actions greedily with respect to the learned Q-values.

(Mnih u. a. 2015)

Problems for RL and Deep Neural Networks

Reinforcement learning is known to be unstable when a nonlinear function approximator such as a neural network is used to represent the action-value function.

This instability has several causes:

- the correlations present in the sequence of observations,
- the fact that small updates to Q may significantly change the policy and therefore change the data distribution,
- and the correlations between the action-values and the target values

(Mnih u. a. 2015)

Possible Problems for Function Approximation

Goal: apply efficiency and flexibility of TD methods to realistic problems

Problem: Deadly Triad

Approach is …

- off-policy,
- employs non-linear function approximation,
- and uses bootstrapping.

Combined: can become unstable or does not converge!

(Weng 2018)

Deep Q-Networks

… improved and stabilized training of Q-learning when using a Deep Neural Network for function approximation.

Two innovative mechanisms:

- *Experience Replay*: use a replay buffer for storing experiences.
- Periodically Update *Target network* that are employed for bootstrapping.

(Weng 2018)

Key Ideas for DQN: 1. Experience Replay

"First, we used a biologically inspired mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution."

- All episode steps $e_t = (S_t, A_t, R_t, S_{t+1})$ are collected in one replay memory.
- During Q-learning updates: sample steps are drawn randomly from the replay memory.

Experience replay

- improves data efficiency,
- removes correlations in the observation sequences,
- and smooths over changes in the data distribution

(Mnih u. a. 2015; Weng 2018)

Key Ideas for DQN: 2. Stabilize Bootstrapping

"Second, we used an iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target."

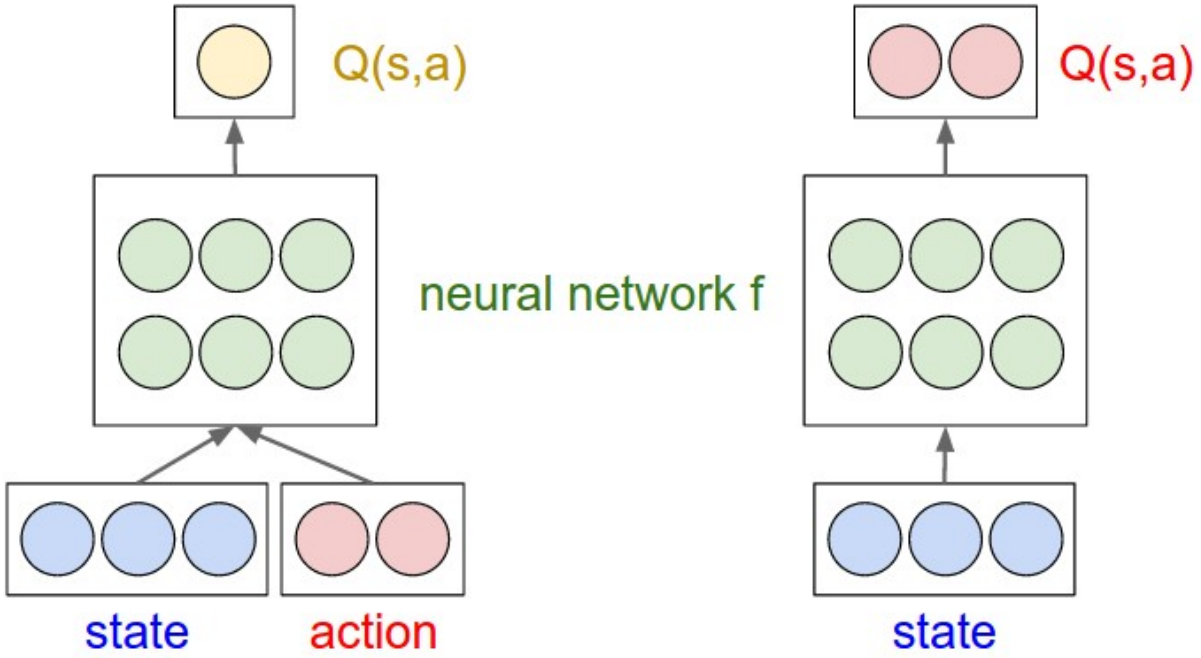
Periodically Updated Target:

- Q is optimized towards target values that are only periodically updated.
- The Q network is cloned and kept frozen as the optimization target every C steps (C is a hyperparameter).

This modification makes the training more stable as it overcomes the short-term oscillations.

(Weng 2018)

Deep Q Network Overview



3-dimensional state space (blue) and 2 actions (red); green nodes represent a NN.

Left: naive approach that takes multiple forward passes to find the argmax action. Right: more efficient approach, $Q(s, a)$ computation is effectively shared among the neurons in the network.

(Karpathy 2015)

DQN Algorithm: Preprocessing

Frame: 210×160 pixel images with a 128-colour palette

Preprocessing: Mapping $\phi(s)$

- remove flickering (max. value from two frames)
- extract luminance from the RGB frame and rescale to 84×84
- stack $m = 4$ frames

Input to Convolutional Neural Network

$84 \times 84 \times 4$ image

DQN: Q-Network Architecture

Input $84 \times 84 \times 4$ image

Convolutional Network

- Convolutional Layer, 32 filters, 8×8, stride 4, ReLU
- Convolutional Layer, 64 filters, 4×4, stride 2, ReLU
- Convolutional Layer, 64 filters, 3×3, stride 1, ReLU

Output Layer

Fully-connected linear layer with a single output for each valid action (4 to 18 depending on game).

DQN: Training

Tasks

- 49 Atari 2600 games
- results available for comparison (human performance)
- different network trained for each game
- clip rewards to $-1, 0, +1$
- frame skipping $k = 4$

Features DRL

- ε-greedy: during first million frames scale from 1.0 to 0.1. Afterwards $\varepsilon = 0.1$.
- Replay buffer size 1 million recent frames
- training time: 50 million frames (around 38 days of gaming)

Hyperparameter NN

- Optimizier: RMSProp
- minibatch size 32

Evaluation of Trained Agents

- play each game 30 times
- for up to 5 min each time
- with different initial random conditions (‘no- op’)
- ε-greedy policy with $\varepsilon = 0.05$

DQN Algorithm

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for *episode* $= 1, \ldots, M$ **do**
  Initialize sequence $s_1 = x_1$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, \ldots, T$ **do**
    with probability ε select random action a_t,
    otherwise $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$
    Execute action a_t 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 D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if terminates at } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a', \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient step on $(y_j - Q(\phi_j, a_j; \theta))^2$ wrt. θ
    Every C steps reset $\hat{Q} = Q$
  end
end

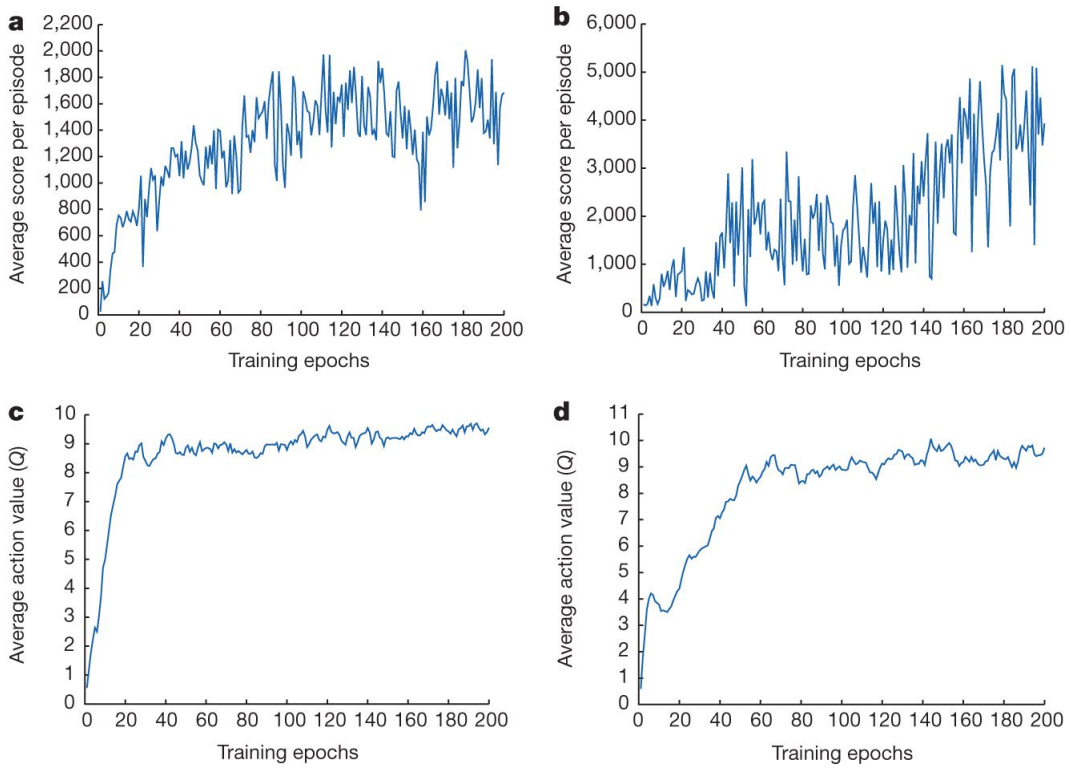
Agent part of the algorithm ($\hat{Q}$); Environment interaction (history of inputs); Experience Replay

DQN Example: Puckworld

(Karpathy 2015)

Example Game: Space Invaders

Learning over Time

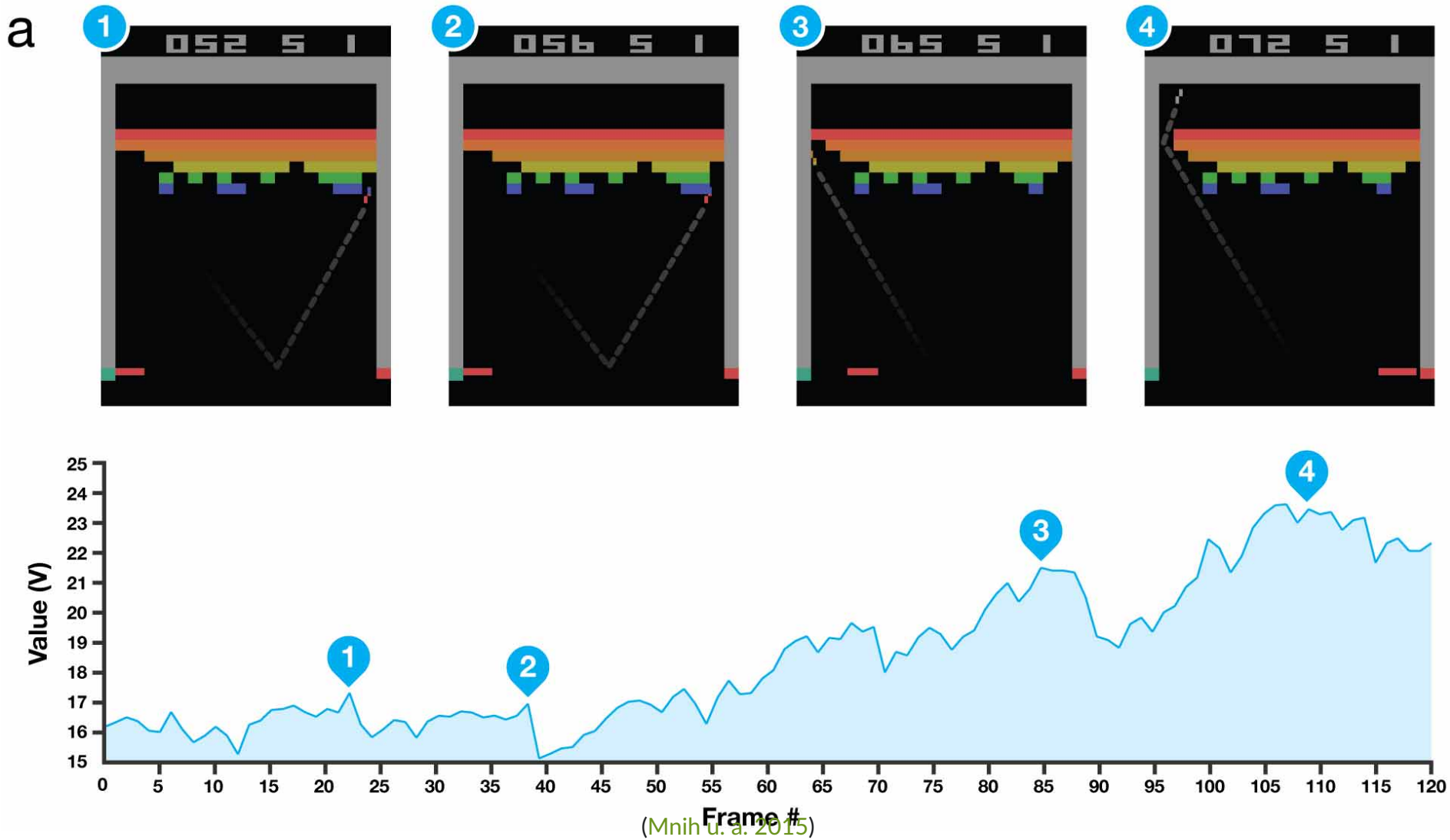


Average score achieved per episode. a) Space Invaders. b) Seaquest. c) Average predicted action-value on a held-out set of states on Space Invaders. d) Average predicted action-value on Seaquest.

(Mnih u. a. 2015)

Example: Learning in Breakout

Example: Learning in Breakout 2

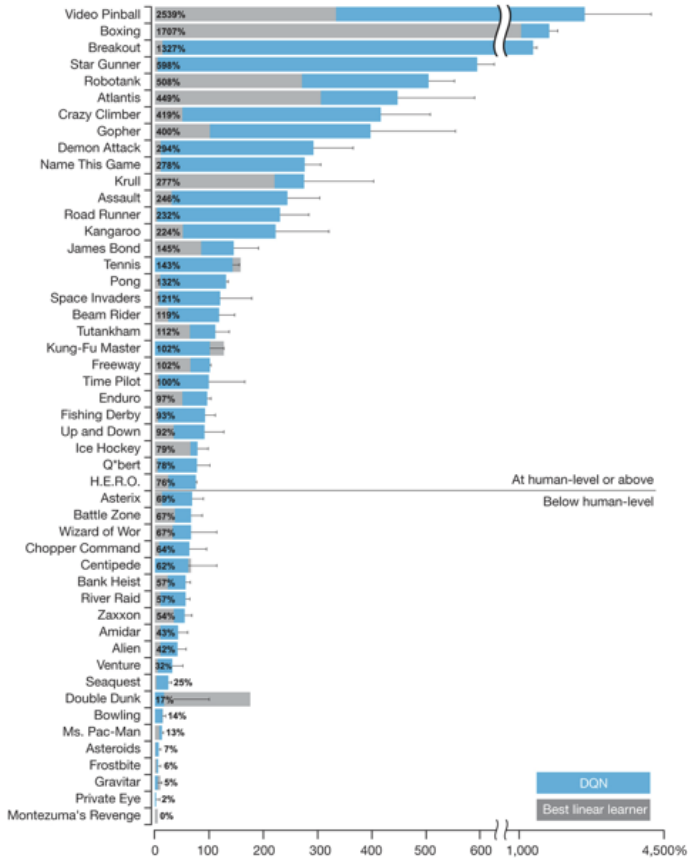


(Mnih u. a. 2015)

Results - “Superhuman” Performance

Summary

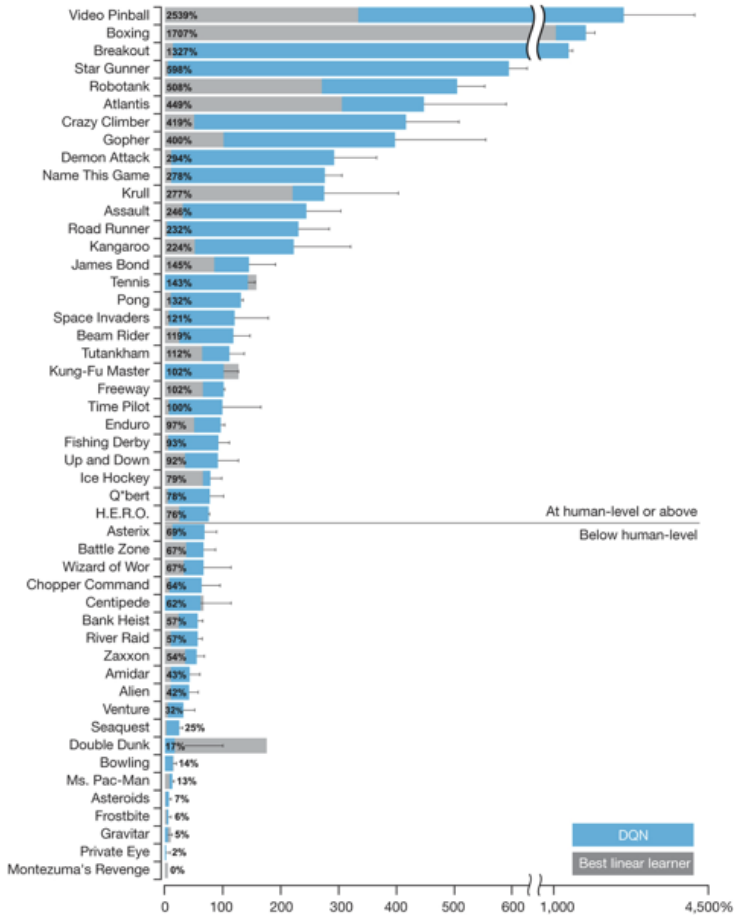
“Our DQN method outperforms the best existing reinforcement learning methods on 43 [out of 49] of the games without incorporating any of the additional prior knowledge about Atari 2600 games used by other approaches.”



(Mnih u. a. 2015)

Drawbacks of DQN (and other DRL methods)

- Delayed Rewards (makes Credit Assignment even more difficult)
- Overfitting towards a specific niche and showing no generalization
- many real world scenarios are non-Markovian or non-stationary (e.g. when other agents are co-adapting)

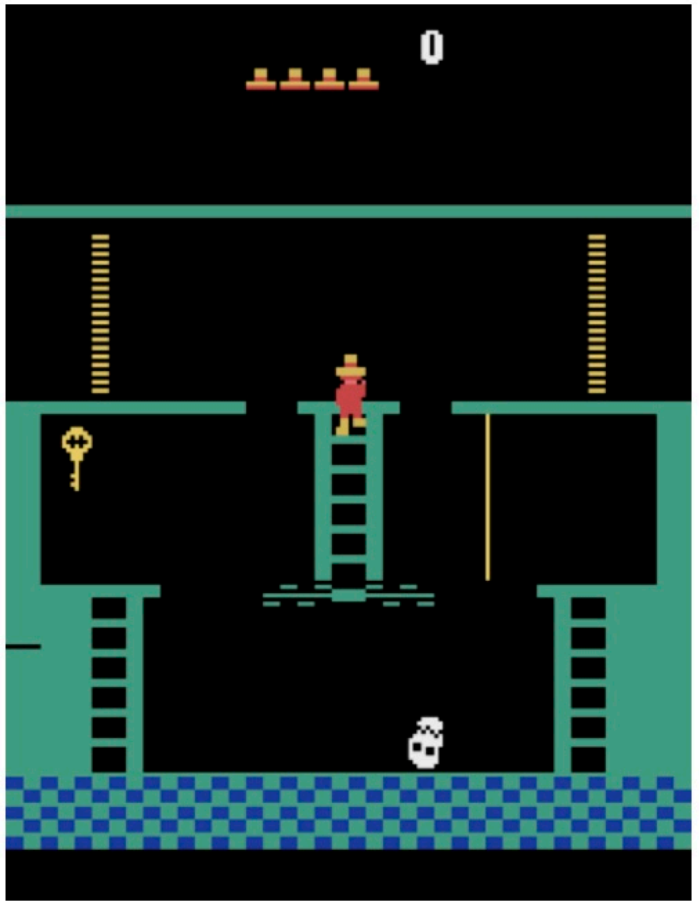


(Mnih u. a. 2015)

Delayed Rewards

"games demanding more temporally extended planning strategies still constitute a major challenge for all existing agents including DQN (e.g., Montezuma's Revenge)"

- It's difficult to explore large state spaces with sparse and delayed rewards.
- An Objective Function might not provide good guidance where to continue exploration.

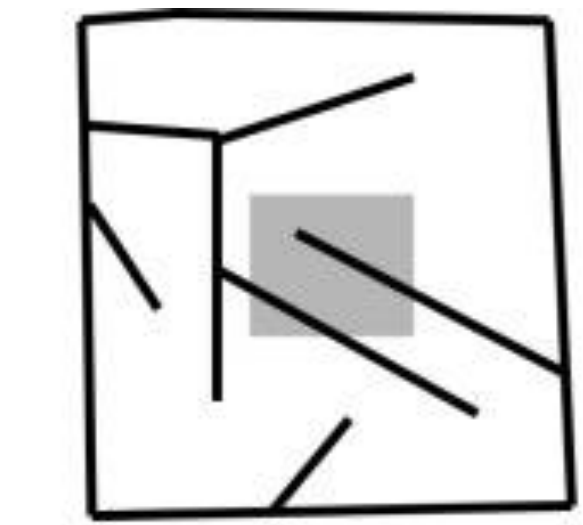


(Mnih u. a. 2015; Kulkarni u. a. 2016)

Evolutionary Robotics Perspective

Landscapes induced by objective functions are often deceptive – the objective function is misleading.

Often, stepping stones are required — initially, objective might get worse.



(a) Reducing Information

(b) Characterizing Behavior by Fitness

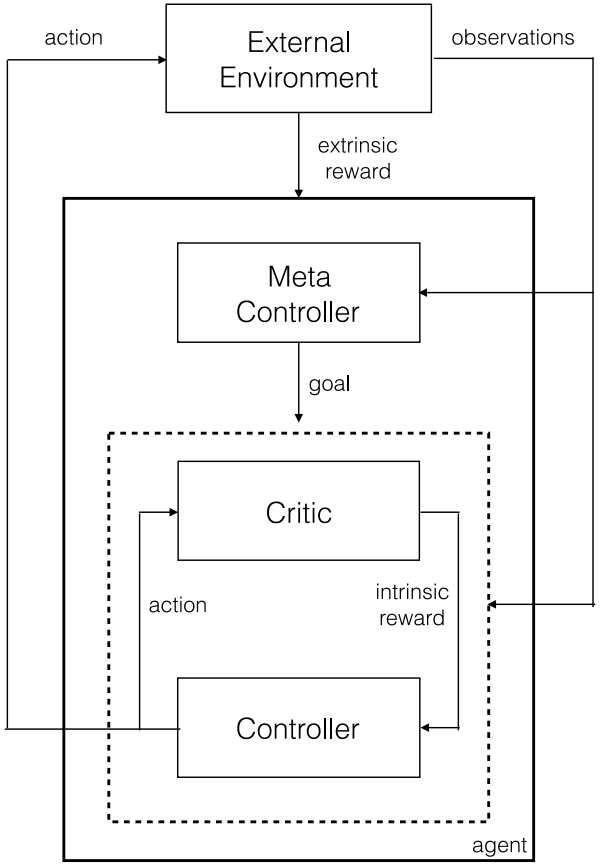
(Lehman und Stanley 2011)

Dealing with Delayed Rewards

"When the environment provides delayed rewards, we adopt a strategy to first learn ways to achieve intrinsically generated goals, and subsequently learn an optimal policy to chain them together."

Approach

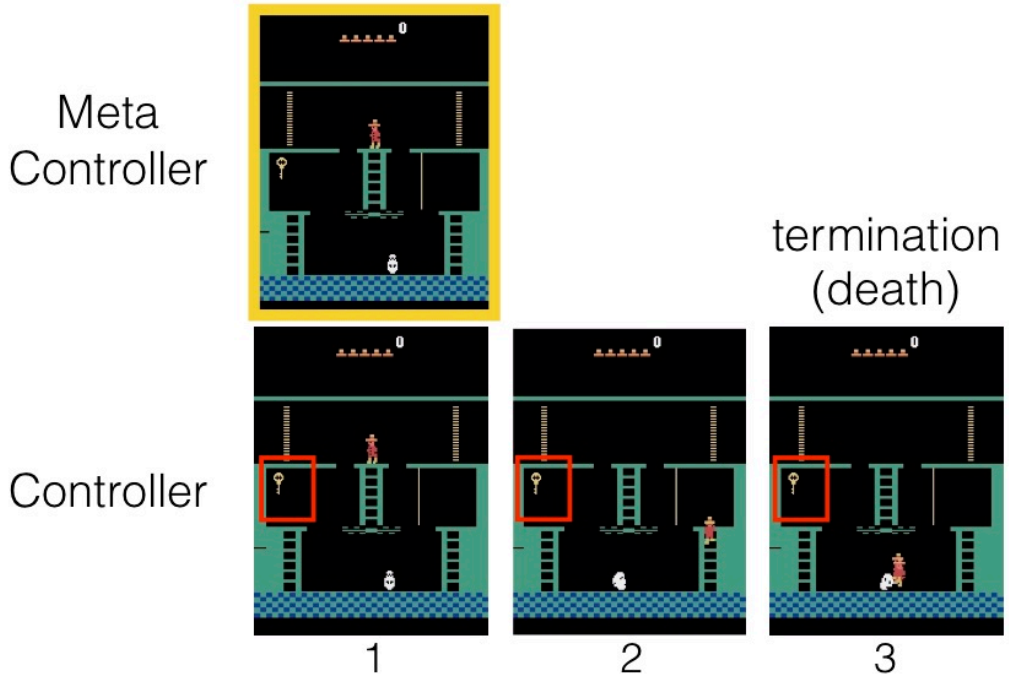
- Use a hierarchical representation.
- Exploration: Driven by a search for novelty (**Intrinsic Motivation**). This tries to cover all possible behaviors during exploration, find stepping stones.



(Kulkarni u. a. 2016)

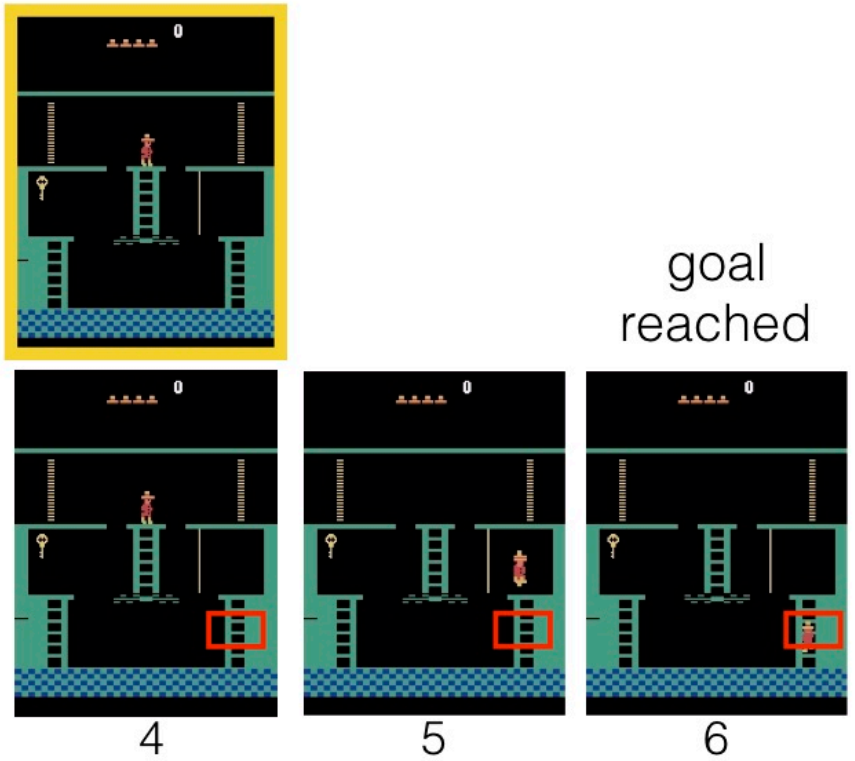
Intrinsic Motivation - Constructing a Representation

Early Learning Phase



Select key as (sub)goal – but fails.

Intermediate Phase

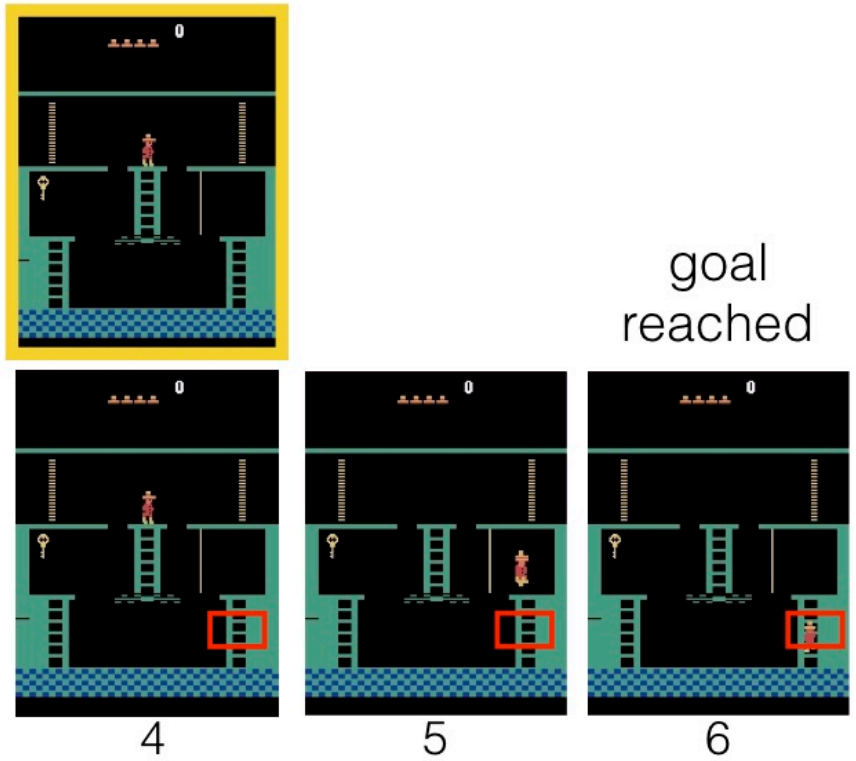


Select ladder successful as goal.

(Kulkarni u. a. 2016)

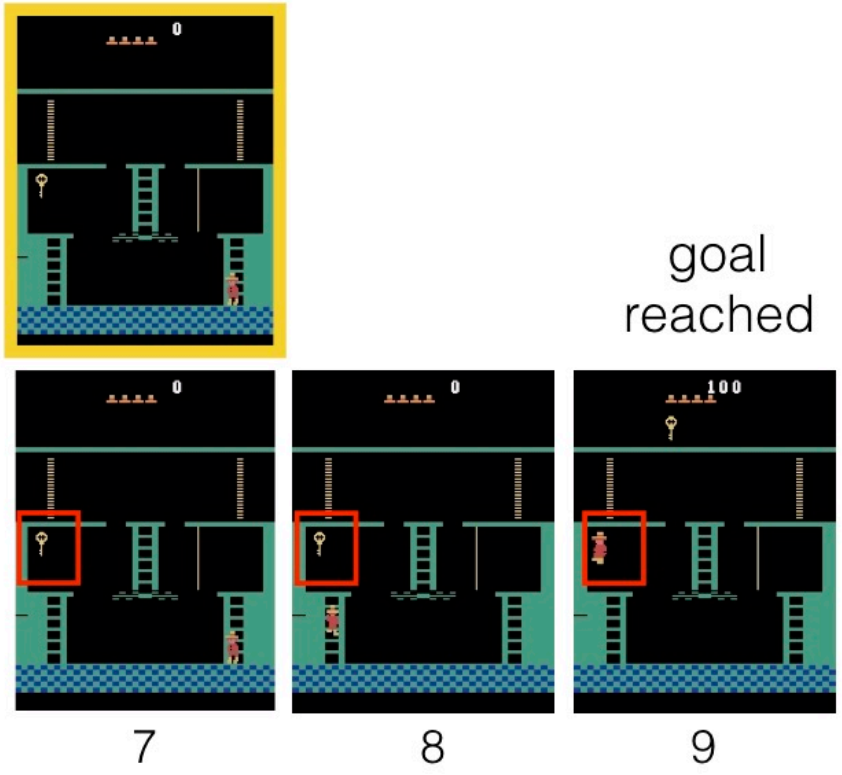
Intrinsic Motivation - Constructing an Abstraction

Intermediate Phase



Select ladder successful as goal.

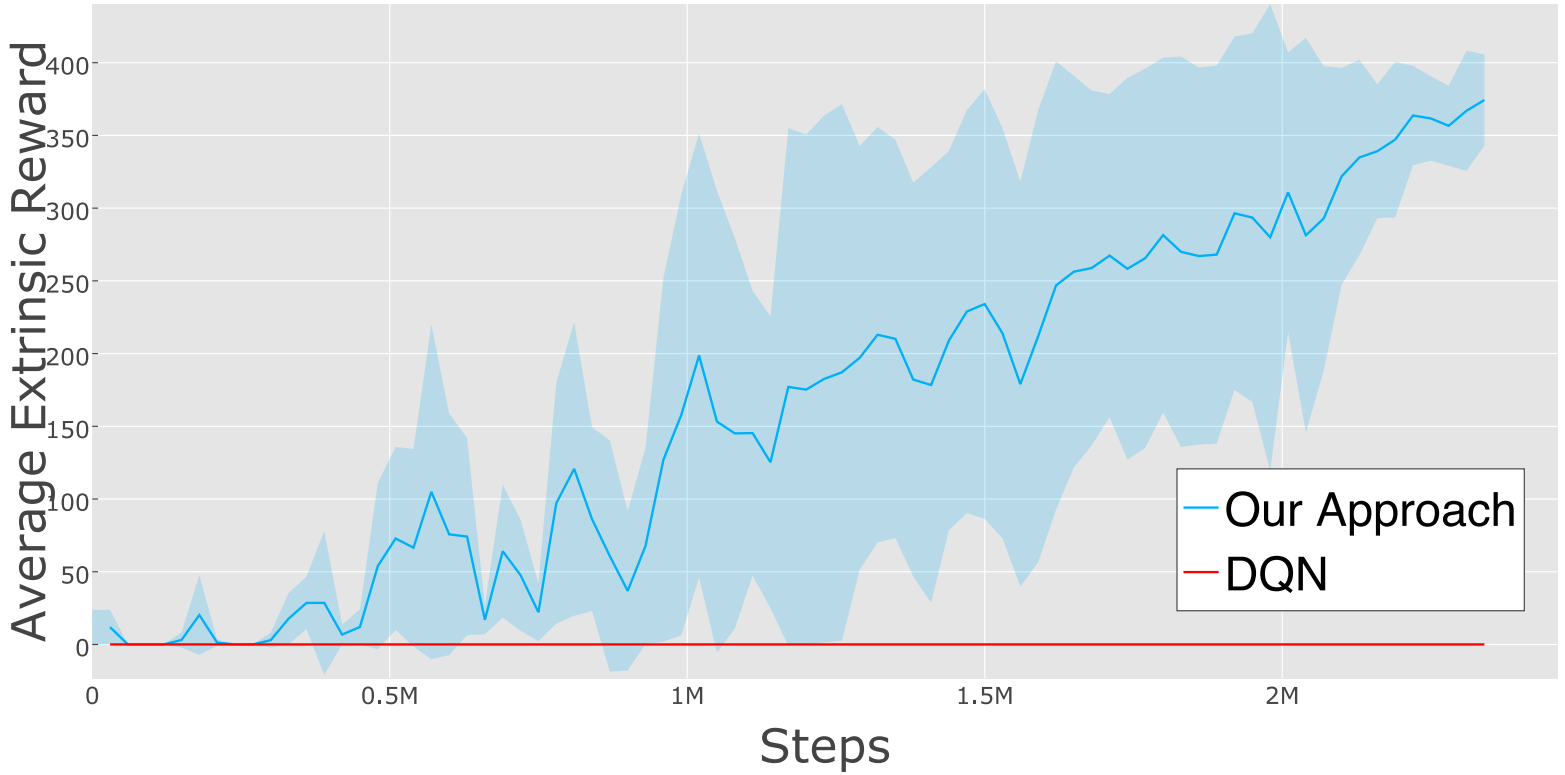
Intermediate Phase



Select key successful as goal.

(Kulkarni u. a. 2016)

Learning with Intrinsic Motivation



(a) Total extrinsic reward

(Kulkarni u. a. 2016)

Problematic: Markov Assumption

In many real world scenarios the Markov Property does not hold.

In ATARI games: many require information on direction of movement.

Simple Solution: add information from different time steps – as an input 4 frames were used.

But difficult in non-stationary environments

- in game like scenarios, opponents can use different strategies (rock-paper-scissor),
- or other agents co-adapt and learn over time as well.

References

Hasselt, Hado van, und Diana Borsa. 2021. „Reinforcement Learning Lecture Series 2021“. https://www.deepmind.com/learning-resources/reinforcement-learning-lecture-series-2021.

Karpathy, Andrej. 2015. „REINFORCEjs“. https://github.com/karpathy/reinforcejs.

Kulkarni, Tejas D., Karthik Narasimhan, Ardavan Saeedi, und Joshua B. Tenenbaum. 2016. „Hierarchical Deep Reinforcement Learning: Integrating Temporal Abstraction and Intrinsic Motivation“. *CoRR* abs/1604.06057. http://arxiv.org/abs/1604.06057.

Lehman, J., und K. O. Stanley. 2011. „Abandoning Objectives: Evolution Through the Search for Novelty Alone“. *Evolutionary Computation* 19 (2): 189–223. doi:10.1162/EVCO_a_00025.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, u. a. 2015. „Human-level control through deep reinforcement learning“. *Nature* 518 (7540): 529–33. http://dx.doi.org/10.1038/nature14236.

Silver, David. 2015. „UCL Course on RL UCL Course on RL UCL Course on Reinforcement Learning“. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.

Sutton, Richard S., und Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. Second. The MIT Press.

Weng, Lilian. 2018. „A (Long) Peek into Reinforcement Learning“. https://lilianweng.github.io/lil-log/2018/02/19/a-long-peek-into-reinforcement-learning.html.