Deep Reinforcement Learning

Lecture 1: Deep-Q Networks

Dr. Matthia Sabatelli
m.sabatelli@rug.nl

university of groningen

Today's Agenda

- Why Neural Networks?
- Approximate Solution Methods
- Deep-Q Networks

Why Neural Networks?

Last week we have seen that the core mathematical framework of Reinforcement Learning (RL) is that of Markov Decision Processes (MDPs), which we defined as the following tuple:

$$\mathcal{M} := (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathfrak{R}, \gamma).$$

We have also seen that the goal of value-based RL algorithms is that of learning a value function that comes in one of these three forms:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, \pi\right];$$

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, a_t = a, \pi\right];$$

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s).$$

While we have seen several RL algorithms that are able to learn such functions, they do come with some important **limitations**:

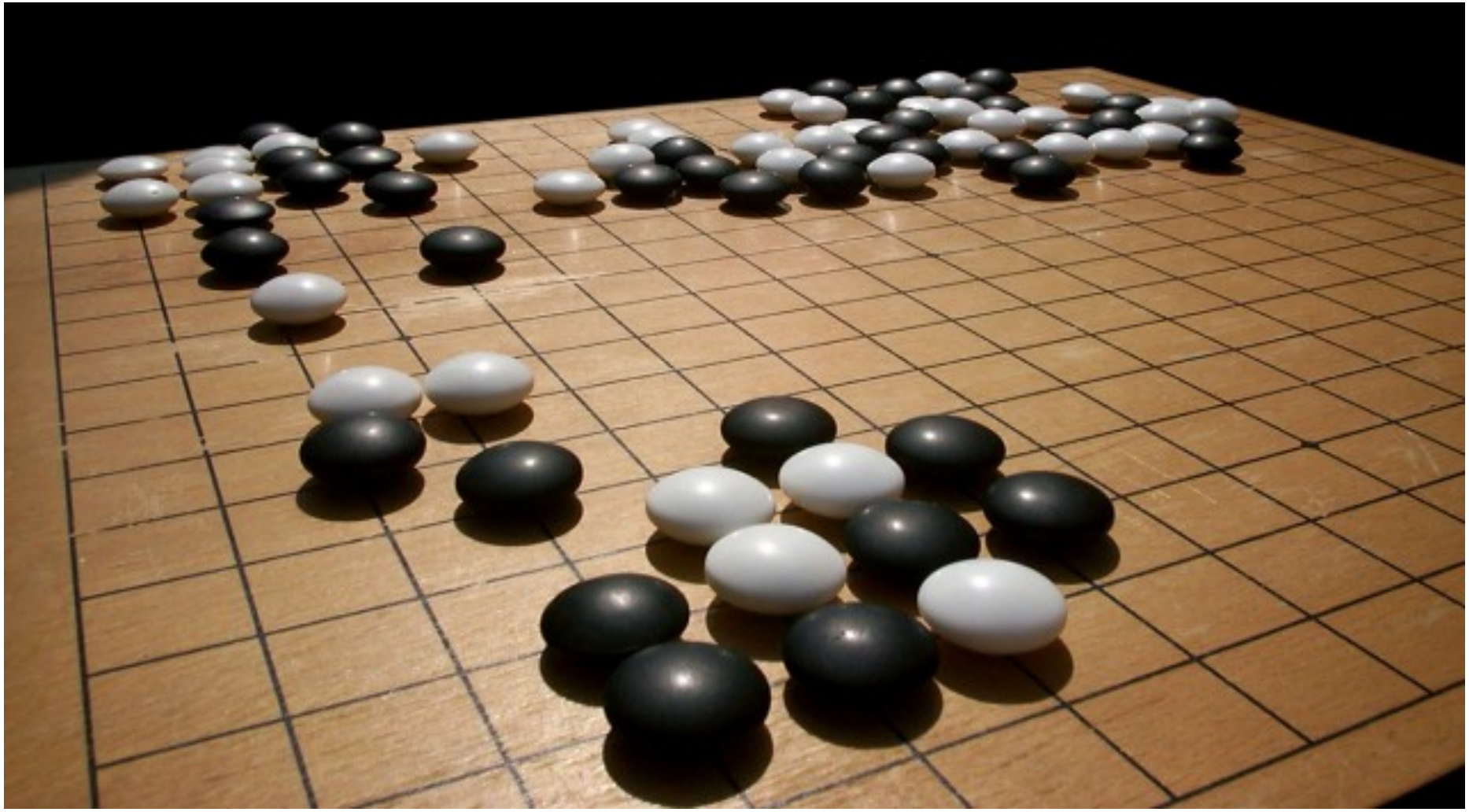
First we need to consider that

- If our goal is to learn $V^{\pi}(s)$ we need to use a table of size $|\mathcal{S}|$ to store all possible state-values.
- If our goal is to learn $Q^{\pi}(s, a)$ we need to use a table of size $|\mathcal{S} \times \mathcal{A}|$ to store all possible state-action values.

If the dimensionality of the state and action spaces of the MDP becomes very large it is impossible to use this approach, which is typically called **Tabular Reinforcement Learning**.

Most real world problems have state and action spaces that make it computationally impossible to rely on tabular RL algorithms.

An example of how quickly the state-space can grow is given by the popular **GO** boardgame:



The game of GO has two simple rules:

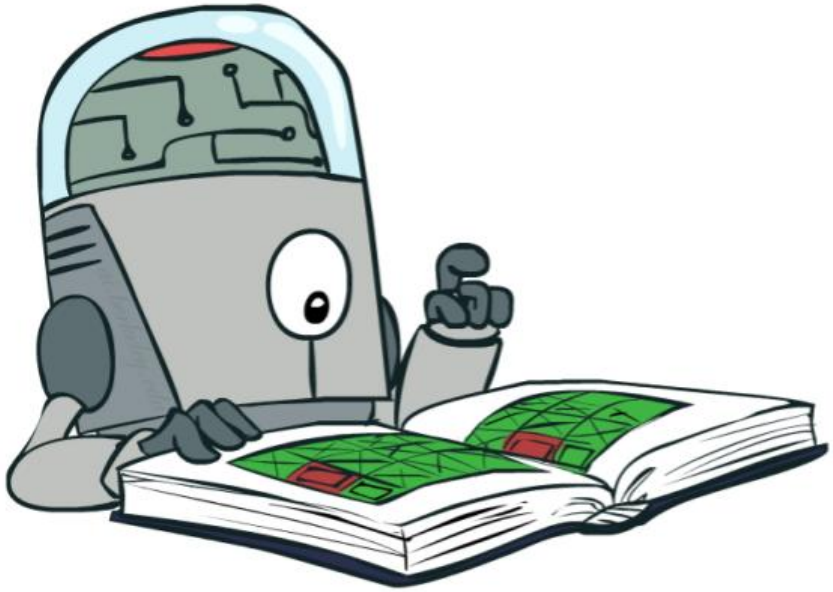
- It is played on a board of size 19×19
- On each location there can, or can't, be a stone (white or black)

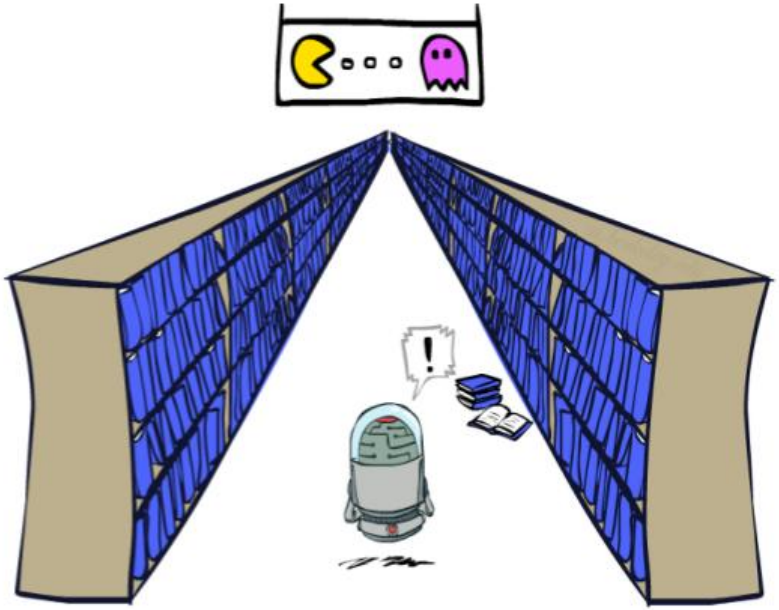
This results in the following state-space $|\mathcal{S}| = 3^{19 \times 19} = 3^{361}$

It is clear that under these conditions it is computationally impossible to learn the exact state-value function $V^{\pi}(s) \text{ for all } s \in \mathcal{S}$.

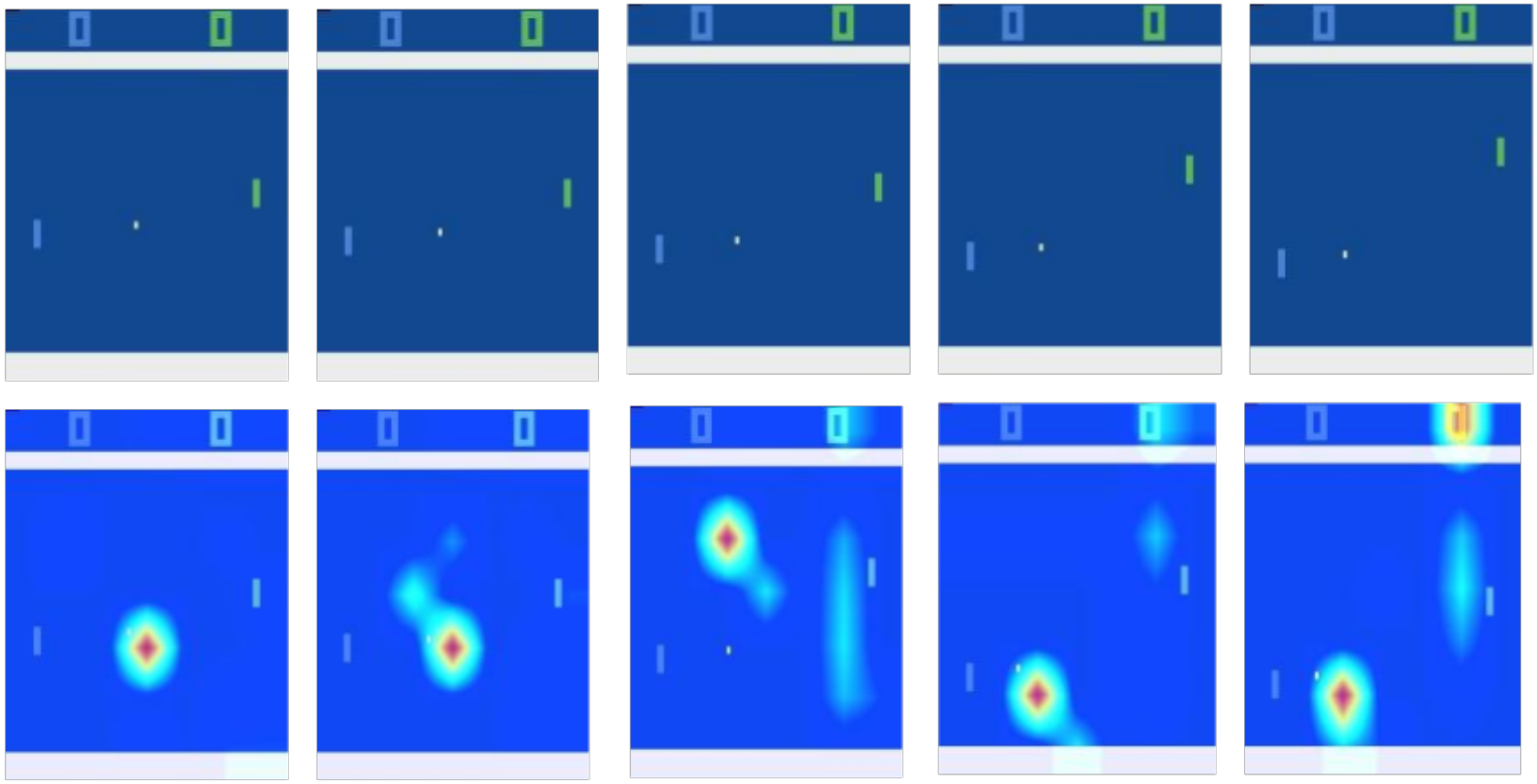
Being unable to deal with extremely large state-action spaces is not the only **limitation** of tabular RL, in fact last week's algorithms are also:

- Impossible to use in a continuous setting
- Require a thorough discretization of the environment
- Lack Generalization





Another concrete **example**: the game of Pong



- All states look very similar among each other
- Small portions of the state-space are actually informative
- Despite some pre-processing operations the state-space stays highly-dimensional

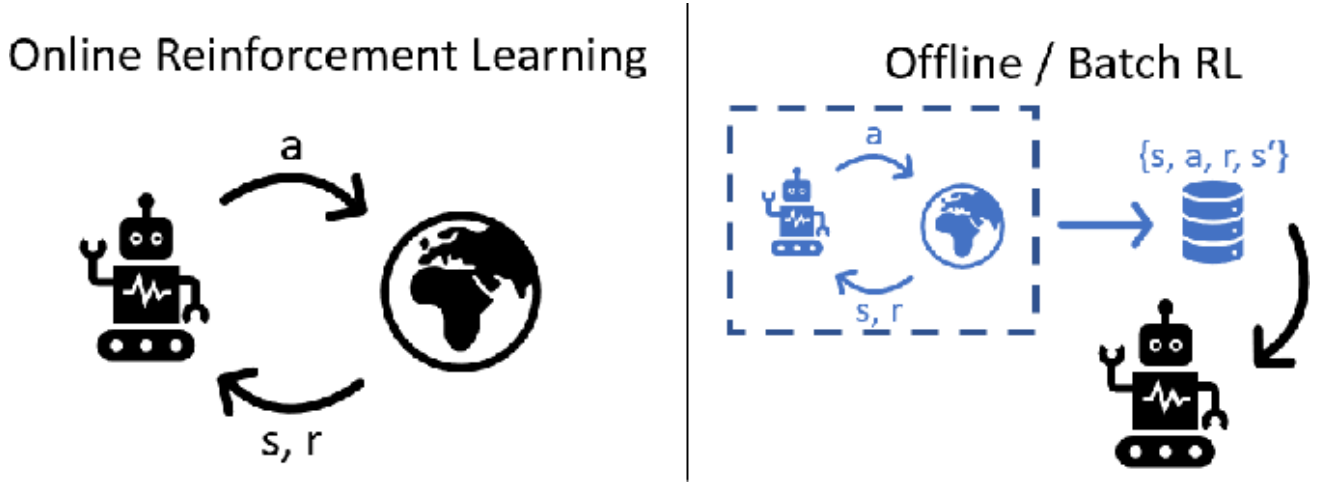

INNOVATIONS IN
nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
LEARNING CURVE
Self-taught AI software attains human-level performance in video games
PAGES 469 & 529
EPIDEMIOLOGY
SHARE DATA IN OUTBREAKS
Forge open access to sequences and more
PAGE 477
COSMOLOGY
A GIANT IN THE EARLY UNIVERSE
A supermassive black hole at a redshift of 6.3
PAGES 490 & 512
QUANTUM PHYSICS
TELEPORTATION FOR TWO
Transferring two properties of a single photon
PAGES 491 & 516
NATURE.COM/NATURE


nature
THE INTERNATIONAL WEEKLY JOURNAL OF SCIENCE
At last — a computer program that can beat a champion Go player PAGE 484
ALL SYSTEMS GO
CONSERVATION
SONGBIRDS À LA CARTE
Illegal harvest of millions of Mediterranean birds
PAGE 452
RESEARCH ETHICS
SAFEGUARD TRANSPARENCY
Don't let openness backfire on individuals
PAGE 459
POPULAR SCIENCE
WHEN GENES GOT 'SELFISH'
Dawkins's calling card 40 years on
PAGE 462
NATUREASIA.COM

Offline Approximate Methods

Perhaps the most popular algorithm in the family of Approximate Reinforcement Learning is Fitted Q-Iteration (FQI):

- Developed for the Batch/Offline Reinforcement Learning setting
- Aims to learn an approximation of the optimal state-action value function $Q^\star$ given a dataset of n sample transitions $\mathcal{D} = (s_i, a_i, r_i, s'_i)$
- No interaction with the environment is allowed



The Fitted Q-Iteration algorithm works as follows:

- It starts from an initial value function $Q_0 \in \mathcal{F}$ where $\mathcal{F}$ is the set of all possible value functions.
- At each iteration $k \geq 0$ FQI approximates the application of the Bellman optimality operator to Q_k by solving a supervised-learning problem from D and Q_k
- The i-th regression targets come in the following form:
  $y_i = r_i + \gamma \max_{a \in \mathcal{A}} Q_k(s', a)$
- The next approximate value function Q_{k+1} is then:

$$Q_{k+1} = \operatorname*{argmin}_{f \in \mathcal{F}} \frac{1}{n} \sum_{i=1}^{n} (f(s_i, a_i) - y_i)^2$$

Some properties of FQI:

- The problem is framed as the minimization of the Mean-Square error, but as mentioned last week other losses can also be used.
- As we are dealing with the Batch/Offline Learning setting, the problem becomes stationary and the algorithm is guaranteed to converge.
- You might wonder what machine learning algorithm is used for representing the approximated value function ...

Online Approximate Methods

If we go beyond the Batch Reinforcement Learning setting the most straightforward function approximator that we can use is a **Linear Function**.

- We define a state-action tuple as a feature vector $\mathbf{x}(s) = [x_1(s), x_2(s), ..., x_q(s)] \in \mathbb{R}^q$,
- We also define a function that is parametrized by a vector of parameters $\theta^a \in \mathbb{R}^q$ for each action $a \in \mathcal{A}$,

Given a trajectory $\langle s_t, a_t, r_t, s_{t+1} \rangle$, we can use last week's Q-Learning update rule for updating the parameters θ_i^a for all i as follows:

$$\theta_i^{a_t} := \theta_i^{a_t} + \alpha(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t))x_i(s_t).$$

We can observe that this update rule modifies the parameter vectors θ^a by minimizing the **Mean Squared Error** loss between a given state-action tuple and Q-Learning's TD-target since:

$$\mathcal{L}(\theta) = \frac{1}{2}\big(y_t - Q(s_t, a_t)\big)^2 \text{ with } y_t = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_{i,a_t}} = -\big(y_t - Q(s_t, a_t)\big)x_i(s_t)$$

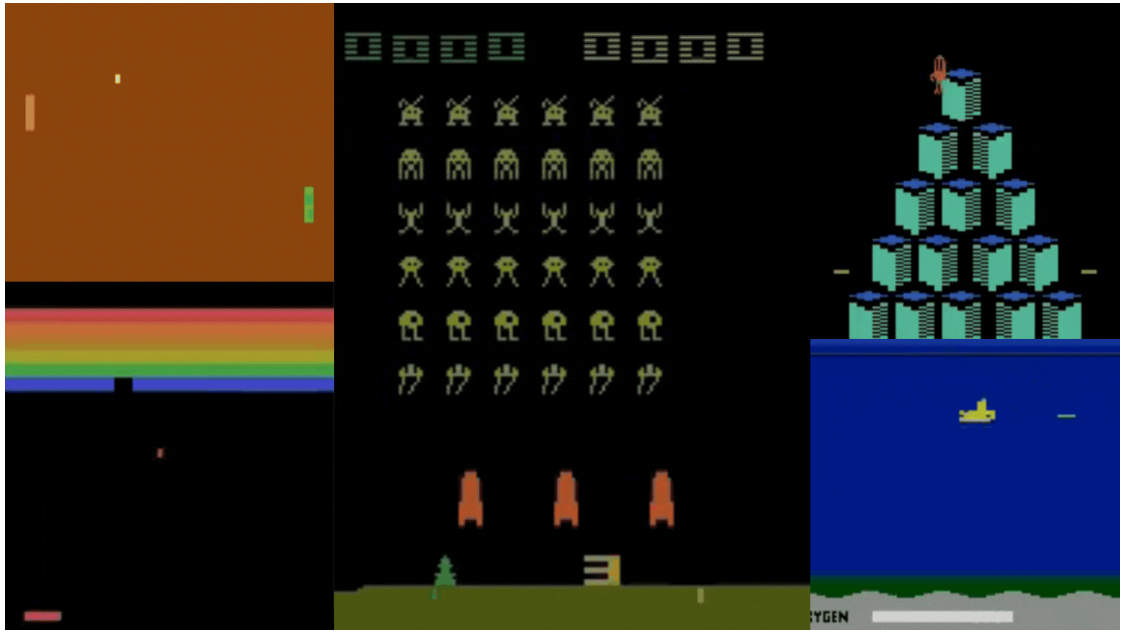
$$\theta_i^{a_t} := \theta_i^{a_t} + \alpha(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a) - Q(s_t, a_t))x_i(s_t).$$

A common strategy for finding the parameters θ which minimize $\mathcal{L}$ is gradient based optimization, although this is not strictly necessary in the linear case where a closed form solution exists.

Deep Q-Networks

We will now shift our attention to function approximators that come in the form of a Convolutional Neural Network.

- We assume that the state representation is highly dimensional and spatially organized such as an image.
- All the coming design choices consider the popular Atari Learning Environment as benchmark, but they do scale well to other types of problems.



How to design a Deep Q-Network?

Our goal will be that of learning an approximation of the optimal state-action value function

$$Q(s, a; \theta) \approx Q^{\star}(s, a)$$

The structure of a typical Deep Reinforcement Learning agent follows that of most Convolutional Neural Networks that are also used for supervised learning:

- $\texttt{INPUT} \rightarrow \texttt{CONV} \rightarrow \texttt{RELU} \rightarrow \texttt{FC}$.
- $\texttt{INPUT} \rightarrow [\texttt{CONV} \rightarrow \texttt{RELU} \rightarrow \texttt{POOL}]\texttt{*2} \rightarrow \texttt{FC} \rightarrow \texttt{RELU} \rightarrow \texttt{FC}$.
- $\texttt{INPUT} \rightarrow [\texttt{CONV} \rightarrow \texttt{RELU}] \rightarrow [[\texttt{FC} \rightarrow \texttt{RELU}]]\texttt{*2} \rightarrow \texttt{FC}$.

Although, especially when it comes to model-free RL, the overall **depth** of the models is smaller.

A typical design choice that one needs to face is how to define the size of the **output** layer.

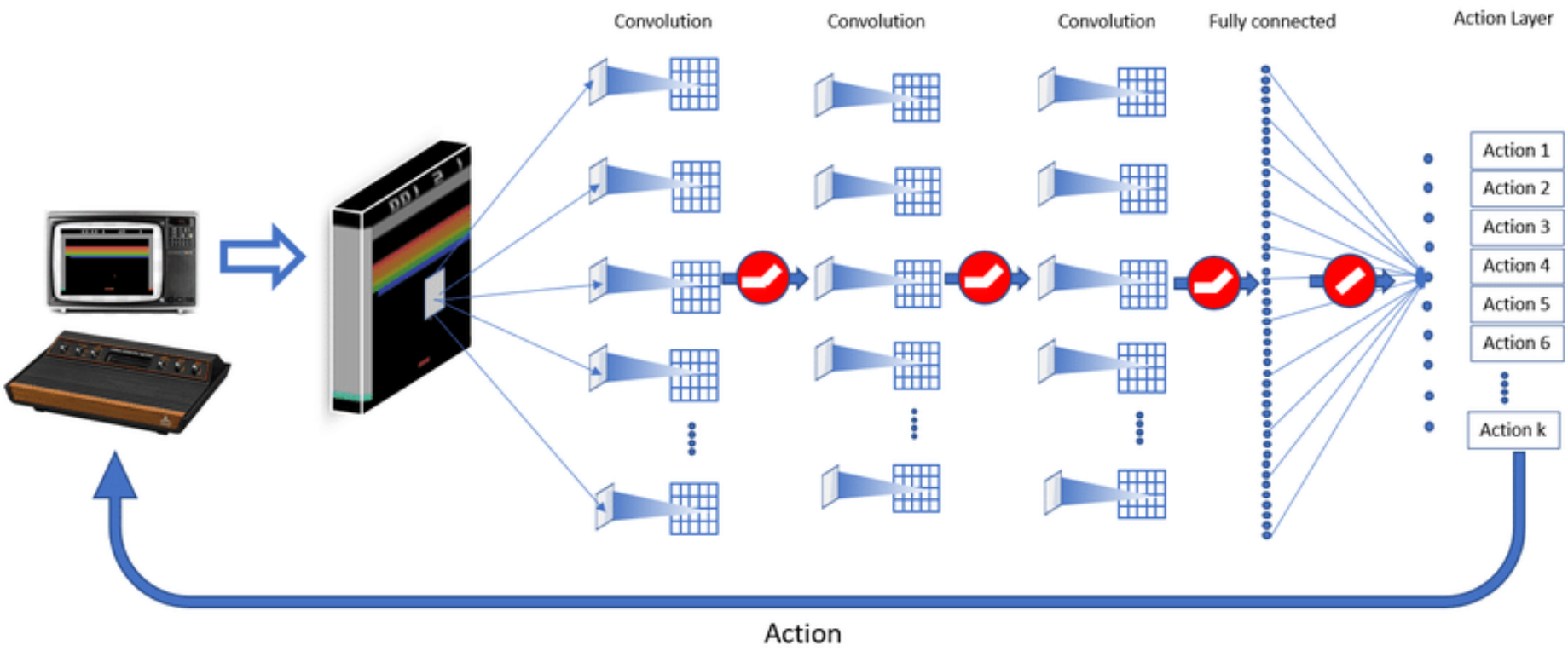
Recall, that our goal is to have a function parametrized by θ that is able to estimate:

$$Q^{\pi}(s, a; \theta) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k} \middle| s_t = s, a_t = a, \pi\right].$$

As a result the model will have as many output nodes as there are actions available to the agent.

In terms of **activation functions**, as our goal is that of estimating a real value, a linear activation function is used for the final layer.

Another look at one of the Atari games ...



Just like in Supervised Learning defining all hyperparameters of a Deep Q-Network is a complicated process that requires **trial and error**.

However, there are some design choices that are known to work better in practice than others:

- Number of layers: these types of networks are not very deep and usually 3/4 hidden layers are enough.
- Non-Linearities: the popular ReLU activation function $f(x) = \max(0, x)$ is used to introduce non-linearity between the hidden layers.
- Optimizers: as the problem is even less-convex than usual, the most used optimizer is RMSProp:

$$r_t = \rho r_{t-1} + (1 - \rho) g_t \odot g_t$$
$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + \sqrt{r_t}} \odot g_t.$$

How to pre-process the input for a Deep Q-Network?

Similar to supervised learning tasks, also when it comes to Deep Reinforcement Learning the input representations need to be carefully pre-processed.

- A human playing one of the aforementioned Atari games sees 210×160 pixel image frames with 128 colors at 60Hz
- In principle these images can be used as input for a Convolutional Neural Network however:
    - Memory Expensive
    - Large processing power

Typical **pre-processing** operations that are performed on high dimensional and spatially organized inputs in a Deep Reinforcement Learning context are:

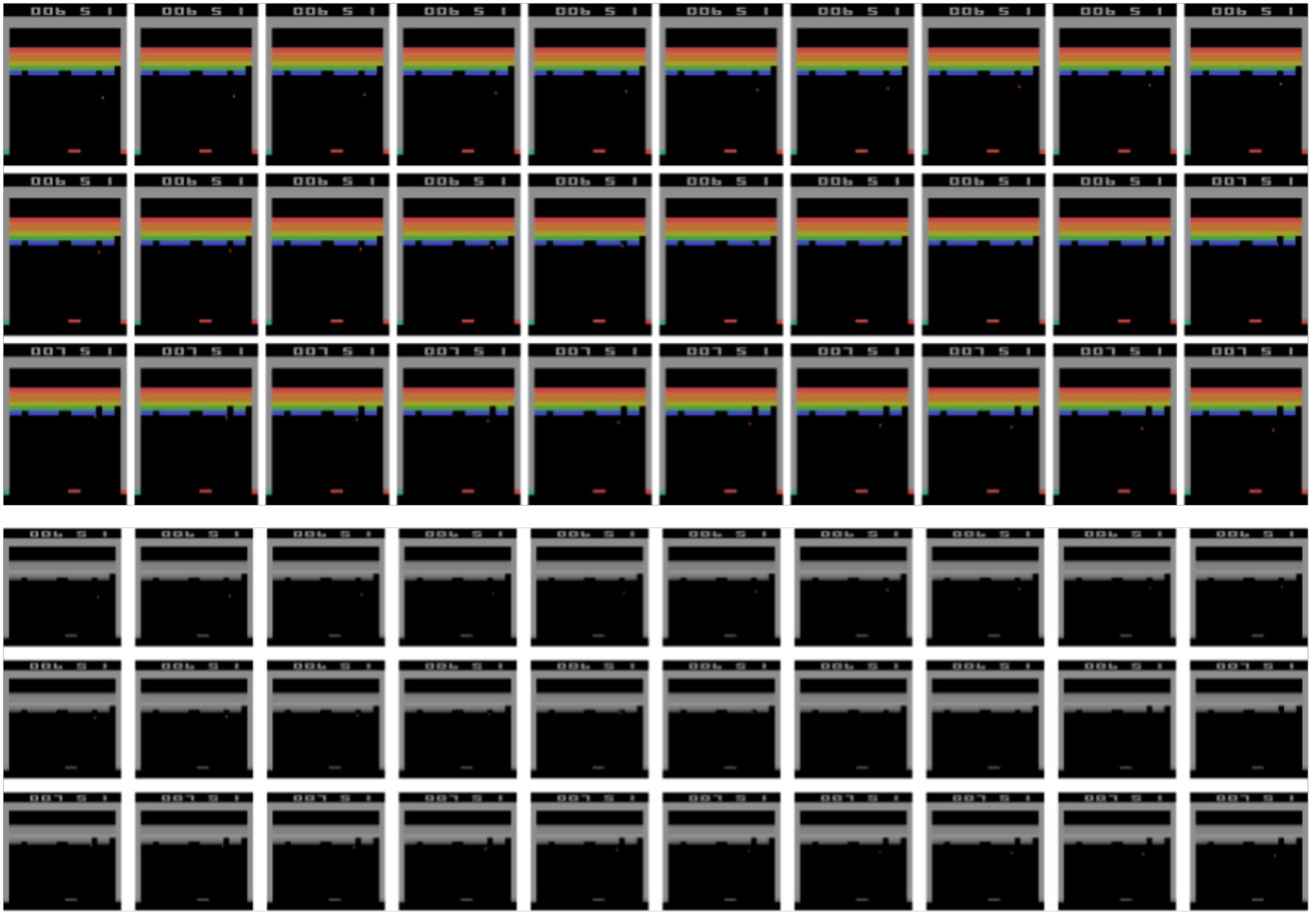
- Grayscaling: as colors do not provide any informative information the input image gets converted into a black and white image.
- Rescaling: the size of the input image gets reduced until the point no meaningful information is lost.
- Frame Stacking: the input stacks the four most recent frames one after the other with the goal of making the state representation more Markovian and capture temporal dependencies.

As a result the **input** has the following dimensions:

$$84 \times 84 \times 4$$

Note that these operations are **not restricted** to the Atari domain only!

The result typically looks like this:



How to train a Deep Q-Network?

Break

How to train a Deep Q-Network?

W keep formulating our learning problem as a regression task.

The most popular Deep Reinforcement Learning algorithm is arguably the **DQN** algorithm.

Its goal is to train a Convolutional Neural Network with the following objective function:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right],$$

which resembles the tabular Q-Learning algorithm.

Differently from the linear case we have some novel key components:

$$\mathcal{L}(\theta) = \frac{1}{2}\big(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a - Q(s_t, a_t)\big)^2)$$

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \Bigg[\big(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta)\big)^2 \Bigg]$$

- Experience Replay Buffer: D
- Mini-Batch Training: $\mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)}$
- A Target Network: θ^-
- Reward Clipping

Experience Replay

Reinforcement Learning is a form of online-learning, meaning that each time the agent observes one trajectory $\tau = \langle s_t, a_t, r_t, s_{t+1} \rangle$ learning takes place.

However this comes with some disadvantages:

- RL trajectories can be used for learning only once.
- There is a high correlation across the trajectories that are used for training, given by the Markovian property of the environment.

It is hard to train a neural network when:

- The training samples are unique, and therefore accessible to the model only once.
- The training data is not i.i.d.

Experience Replay is a technique that was already introduced in 1992 with the aim of making Reinforcement Learning agents more sample efficient.

The idea is very simple.

Throughout learning we store all the trajectories τ that the agent has encountered while interacting with the environment and construct a dataset out of these:

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ & & & \end{pmatrix}$$

This dataset comes in the form of a queue:

- It has a maximum capacity, which for the Atari games was set to 1,000,000
- Once the full capacity of the buffer is reached the older trajectories get removed and make space for the newest ones

Mini-Batch Training

At training time the agent does not have to only rely on the latest encountered trajectory anymore, but on a sample of previously encountered trajectories as well.

As trajectories get uniformly sampled from the buffer

$$\mathbb{E}_{\sim U(D)}\langle s_t, a_t, r_t, s_{t+1}\rangle$$

- We construct a batch of samples that is very similar to the ones that are typically used for tackling supervised learning problems. Our goal becomes minimizing the **expectation** $\mathbb{E}$ over these trajectories.
- We remove the temporal correlation across training samples as different trajectories will come from very different episodes and satisfy the i.i.d. property.

Target Network θ^{-}

Each of the trajctories that we sample from the buffer needs to be associated to its respective Temporal-Difference (TD) target, which in the case of DQN is:

$$r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a)$$

This target, which corresponds to the numerical quantity we want to regress towards, needs to be computed for every $\tau = \langle s_t, a_t, r_t, s_{t+1} \rangle$

In the linear approximation case, the TD-target is estimated by the same value function approximator that we are trying to train, however when a Convolutional Neural Network is used this leads to **divergence**.

To avoid this, one additional function approximator is included, parametrized by θ^-.

Its only job is that of estimating the TD-targets that are necessary for training

$$r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-)$$

However this network never gets **explicitly trained**, as gradient descent is performed on the θ parameters of the Deep-Q Network only:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

- As the θ^- parameters do not change each time a mini-batch of trajectories is sampled, the learning problem becomes more stationary.
- Every n iterations we update $\theta^- \leftarrow \theta$.

Reward Clipping

As different Atari games have different rules, the reward signals r_t coming from the environment can vary greatly.

This means that every game will result in a Temporal-Difference error of different magnitude:

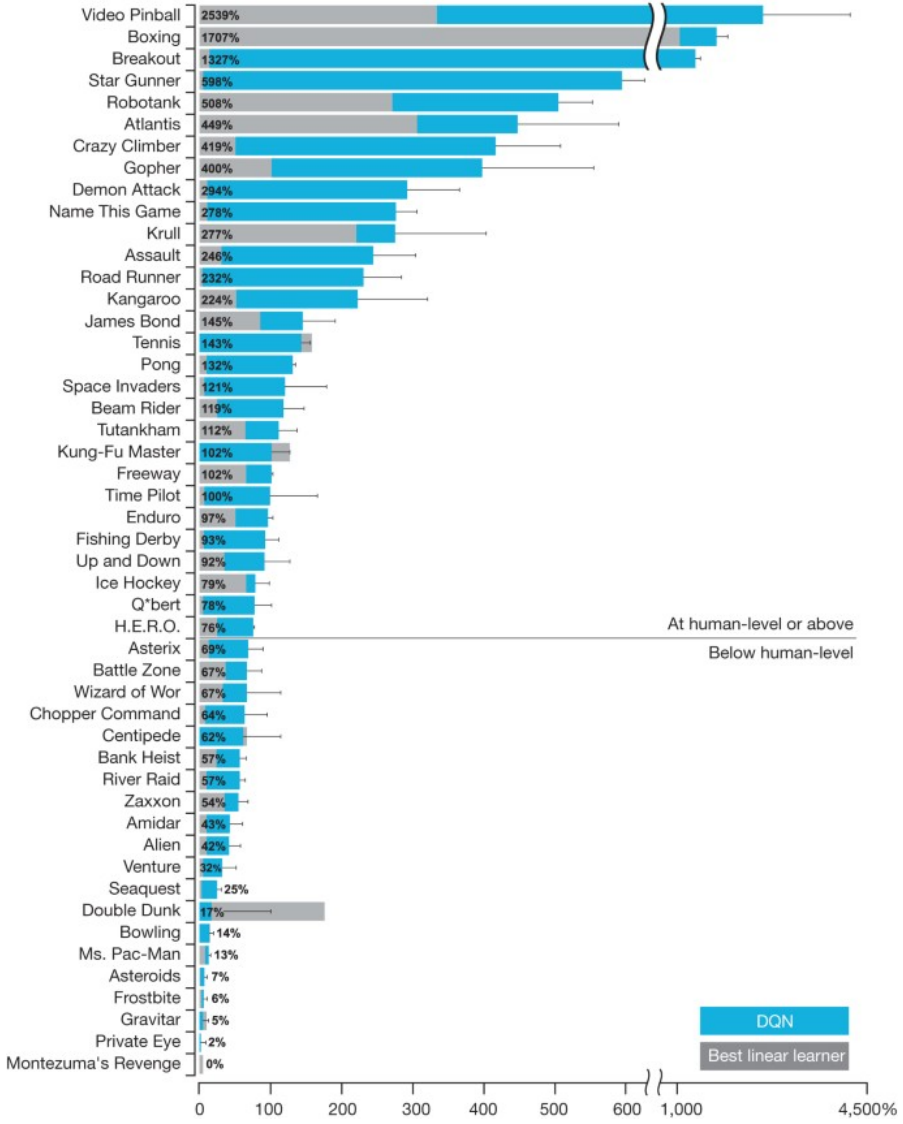
$$r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-)$$

The main goal of the DQN algorithm was to have a **unique** neural architecture that was able to play across all the different Atari games.

To achieve this all positive rewards r_t were set to 1, and all negative rewards were set to -1, leaving 0 rewards unchanged.

This results in bounded TD errors which helped the overall learning greatly.

While the **DQN** algorithm resulted in a significant breakthrough for the Deep Reinforcement Learning community, it also showed some limitations on some games.



The "Below Human Level" performance obtained on some games can be attributed to the following limitations of DQN:

- Biased value estimates.
- Only focuses on learning an approximation of the state-action value function $Q^{\pi}(s, a)$.
- Inefficient sampling of trajectories τ from the buffer D.
- Inefficient exploration.
- The loss function $\mathcal{L}$ is:
    - Non-Convex
    - Non-Smooth
    - Has unbounded gradients

Biased Value Estimates

The main goal of model-free RL algorithms is that of estimating the state-action value function as accurately as possible:

$$Q^{\pi}(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k} \middle| s_t = s, a_t = a, \pi\right].$$

A popular problem that influences the quality of the learned policy π is that of overestimation bias:

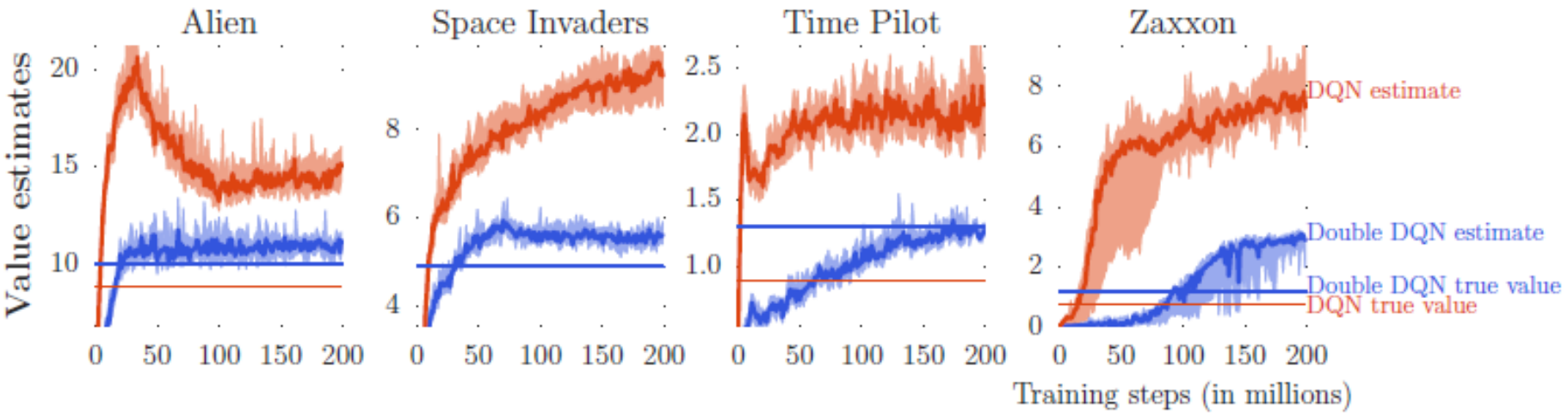
The agent overestimates the quality of certain state-action pairs and thinks it is going to get rewards r_t that are much higher than the ones that the environment returns.

The main cause of the overestimation bias is the $\max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-)$ operator used within DQN's TD-target

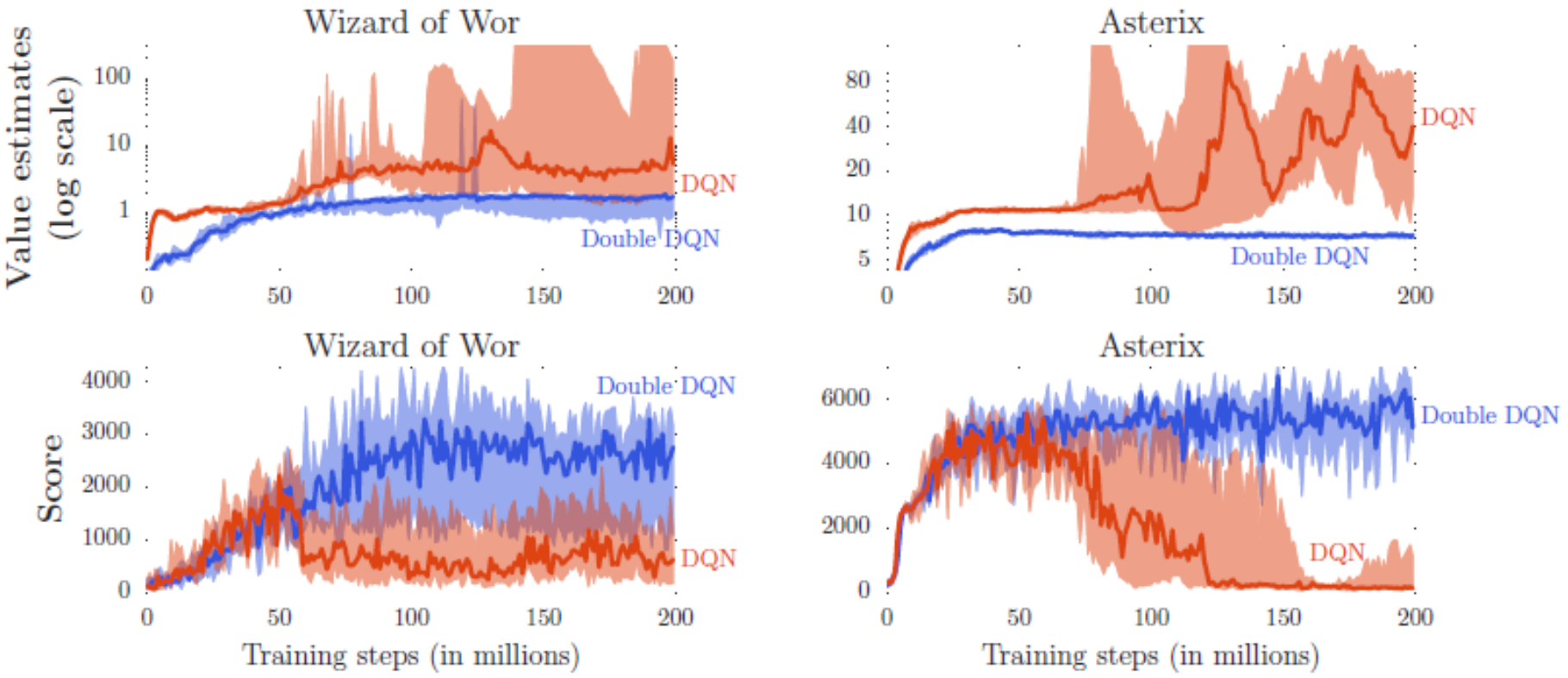
$$r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-)$$

which chooses the largest $Q(s, a)$ available at s_{t+1}.

The overestimation bias in practice:



The resulting detrimental performance on some Atari games.



As you might have guessed from the plots one solution to the overestimation bias is given by the **DDQN** algorithm:

- The "Deep" extension of the tabular Double Q-Learning algorithm encountered last week
- Exploits the target network parametrized by θ^-

DQN is prone to learn overestimated Q-values because the same values are used both for selecting an action as well as for evaluating it:

$$y_t^{DQN} = r_t + \gamma \, Q(s_{t+1}, \underset{a \in \mathcal{A}}{\operatorname{argmax}} \; Q(s_{t+1}, a; \theta); \theta^-).$$

As a result, DQN tends to approximate the expected maximum value of a state, instead of its maximum expected value.

Double Q-Learning solves this issue by keeping track of two separate Q functions, and by randomly preferring one Q function over the other when it comes to selecting which action to execute.

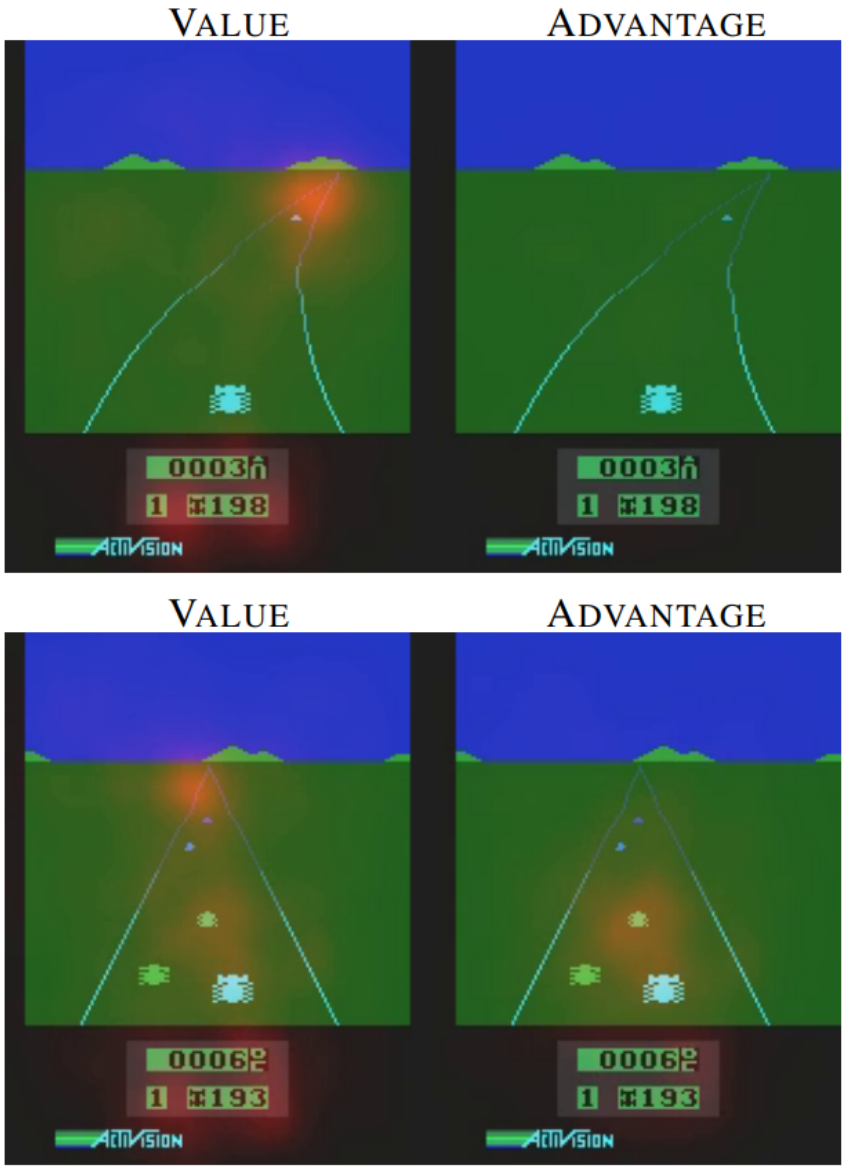
DDQN generalizes this idea and untangles the action selection process from its evaluation by taking advantage of the previously introduced target network θ^-.

- DDQN's target stays the same as in DQN with the main difference being that the selection of an action, given by the online Q-network θ, and the evaluation of the resulting policy, given by θ^-, can now result into smaller overestimations simply by symmetrically updating the two sets of weights (θ and θ^-) which can easily be achieved by regularly switching their roles during training.

Going Beyond the $Q^{\pi}(s, a)$ Function

We know from model-free RL that there are other value functions such as $V^{\pi}(s)$ and $A^{\pi}(s, a)$ that can be of interest, and that might be worth either estimating or learning.

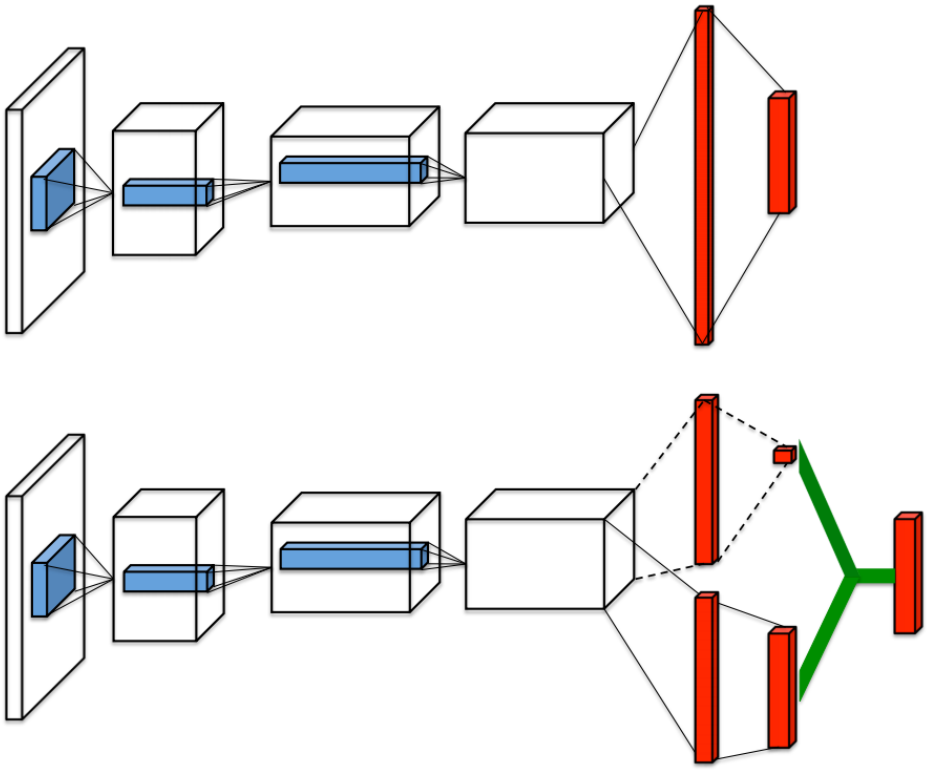
The Dueling Architecture aims at estimating state values, advantages and state-action values simultaneously, as the more values get estimated, the more informative a state becomes.


VALUE
ADVANTAGE
0003
1 198
ACTIVISION
0003
1 198
ACTIVISION
VALUE
ADVANTAGE
0006
1 193
ACTIVISION
0006
1 193
ACTIVISION

The Dueling Architecture

To successfully estimate state values, advantages, and state-action values, the network requires a specific architecture consisting of three separate streams.

Each stream is responsible for estimating one of the three value functions and is initialized with its own parameters $\theta^{(\cdot)}$.



The final Q function of the model is then obtained by combining what is learned by each stream as follows:

$$Q(s,a;\theta^{(1)},\theta^{(2)},\theta^{(3)}) = V\left(s;\theta^{(1)},\theta^{(3)}\right) + \left(A(s,a;\theta^{(1)},\theta^{(2)}) - \max_{a_{t+1}\in\mathcal{A}} A(s,a_{t+1};\theta^{(1)},\theta^{(2)})\right).$$

- The way the parameters of the network get optimized is the same as with the DDQN algorithm
- Note that the state-value function $V^{\pi}(s)$ is only estimated and not explicitly learned.

The Deep Quality-Value Family of Algorithms are a family of DRL algorithms which jointly learn the state-value function $V^{\pi}(s)$ alongside the state-action value function $Q^{\pi}(s, a)$

As most Deep Reinforcement Learning algorithms they find their roots in two tabular algorithms

- $\mathrm{QV}(\lambda) - \mathrm{Learning}$
- $\mathrm{QV} - \mathrm{Max\ Learning}$

which result into the DRL algorithms **DQV-Learning** and **DQV-Max Learning**

The core idea is to have two separate networks, each with its own parameters, that learn from each other's value estimates. Therefore we have:

- $V(s; \Phi) \approx V^{\pi}(s)$
- $Q(s, a; \theta) \approx Q^{\pi}(s, a)$

DQV-Learning

The state-value function $V^{\pi}(s)$ is learned via the simplest form of TD-Learning, therefore resulting in the following objective function:

$$\mathcal{L}(\Phi) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma V(s_{t+1}; \Phi^{-}) - V(s_t; \Phi) \right)^2 \right],$$

whereas the state action-value function $Q^{\pi}(s, a)$ is learned through the following losss:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma V(s_{t+1}; \Phi^{-}) - Q(s_t, a_t; \theta) \right)^2 \right].$$

As usual D is the Experience-Replay memory buffer, used for uniformly sampling batches of RL trajectories $\langle s_t, a_t, r_t, s_{t+1} \rangle$, and Φ^{-} is the target-network used for the construction of the TD-errors.

Note that the role of this target network is different from its role within the DQN algorithm.

In DQV, this network corresponds to a copy of the network which approximates the state-value function and not the state-action value function.

It is also worth noting that both networks learn from the same TD-Target which comes in the following form:

$$y_t^{DQV} = r_t + \gamma V(s_{t+1}; \Phi^-).$$

Because of its TD-Target DQV can be seen as an **on-policy** DRL algorithm.

One can also construct and **off-policy** version of the algorithm.

DQV-Max Learning learns an approximation of the state-value and state-action value functions as follows:

First we learn $V^{\pi}(s)$ by minimizing the following loss:

$$\mathcal{L}(\Phi) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^{-}) - V(s_t; \Phi) \right)^2 \right].$$

While, we then learn the state-action value function $Q^{\pi}(s, a)$ as follows:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma V(s_{t+1}; \Phi) - Q(s_t, a_t; \theta) \right)^2 \right].$$

In **DQV-Max**:

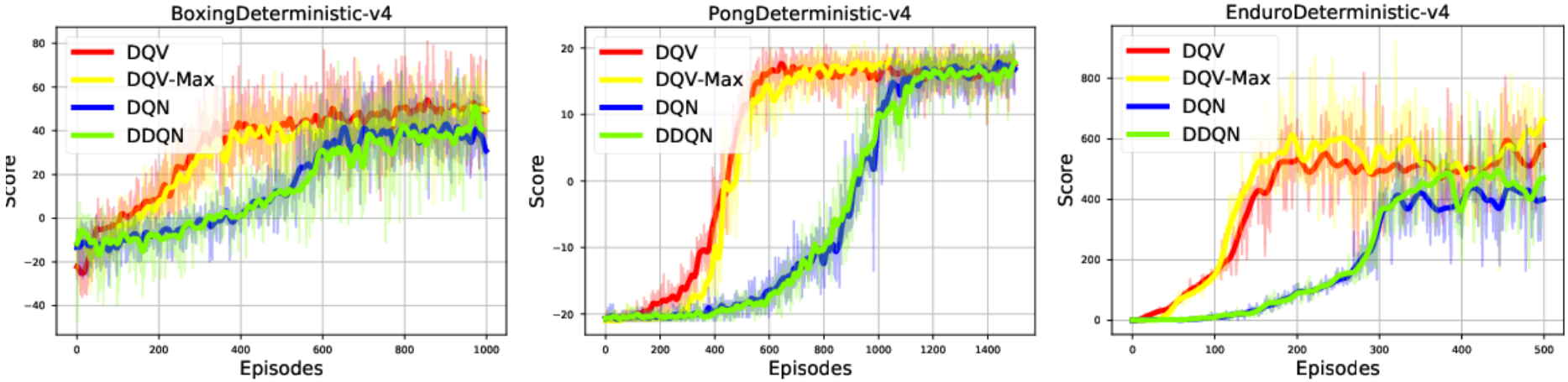
- Differently from DQV-Learning, the computation of two separate targets is required:

$$y_t^{\theta} = r_t + \gamma V(s_{t+1}; \Phi^-).$$
$$y_t^{\Phi} = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-).$$

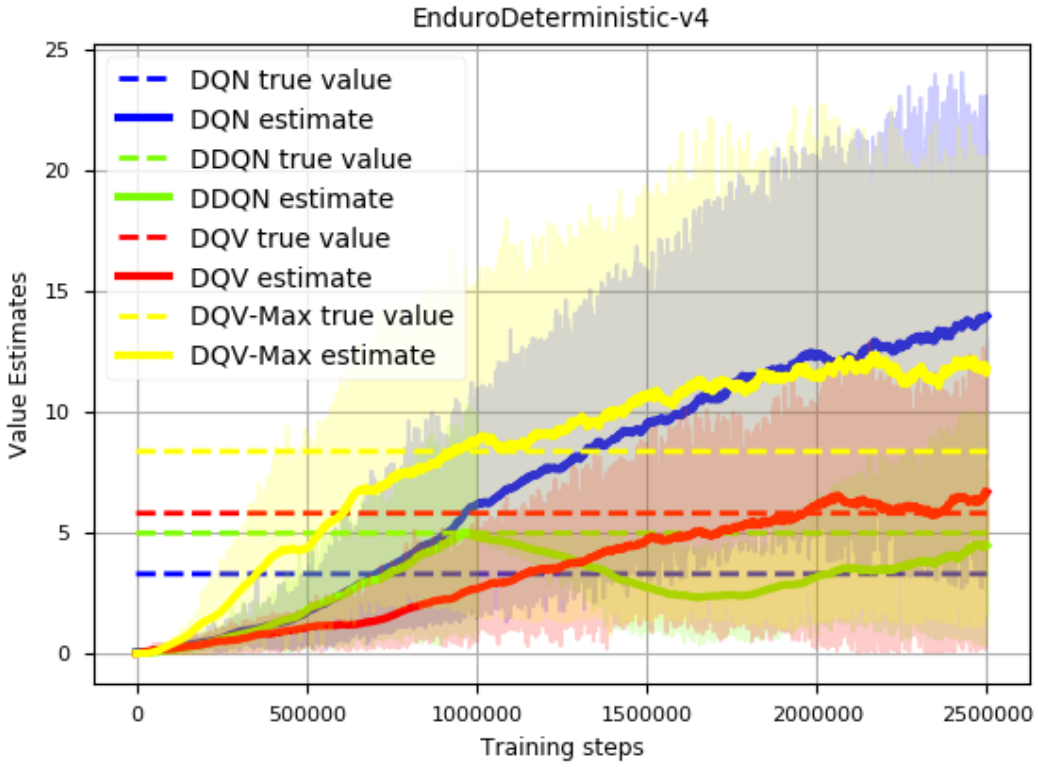
- The target network θ^- corresponds to the same target network that is also used by the DQN algorithm.

Learning a joint approximation of both value functions has several benefits:

- Faster convergence, especially when the action space $|\mathcal{A}|$ of the MDP is large. This is due to the computation of - at least - one TD-target that is not conditioned on the number of actions available to the agent
- Overall better performance
- Better Value Estimates



Also DQV and DQV-Max suffer less from the overestimation bias of the Q-function, although only the first one results in fully unbiased estimates.



Inefficient Use of the Experience Replay Memory-Buffer

We have seen that next to the target network θ^{-} an equally important role within value-based DRL algorithms is played by the experience replay memory buffer D.

The way we have used the buffer so far is extremely inefficient:

- RL trajectories that get sampled from the memory buffer when constructing a mini-batch of trajectories are sampled uniformly ($\sim U(D)$)
    - Considers each τ stored in the buffer as equally important and informative
    - It is easy to think of a scenario where this is not the case

The idea is very simple: we want to use only highly informative trajectories when it comes to building the mini-batches that are used for training our favorite variant of Deep-Q Network.

However, how does one decide which trajectory τ is more informative than another?

The idea is very simple: we want to use only highly informative trajectories when it comes to building the mini-batches that are used for training our favorite variant of Deep-Q Network.

However, how does one decide which trajectory τ is more informative than another?

Based on the Temporal-Difference Error (Of Course!)

Recall the DQN objective function:

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \Bigg[\underbrace{\Big(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \Big)}_{\delta}^2 \Bigg]$$

Prioritized Experience Replay (PER) ensures that the probability of sampling trajectories is proportional to the TD-errors:

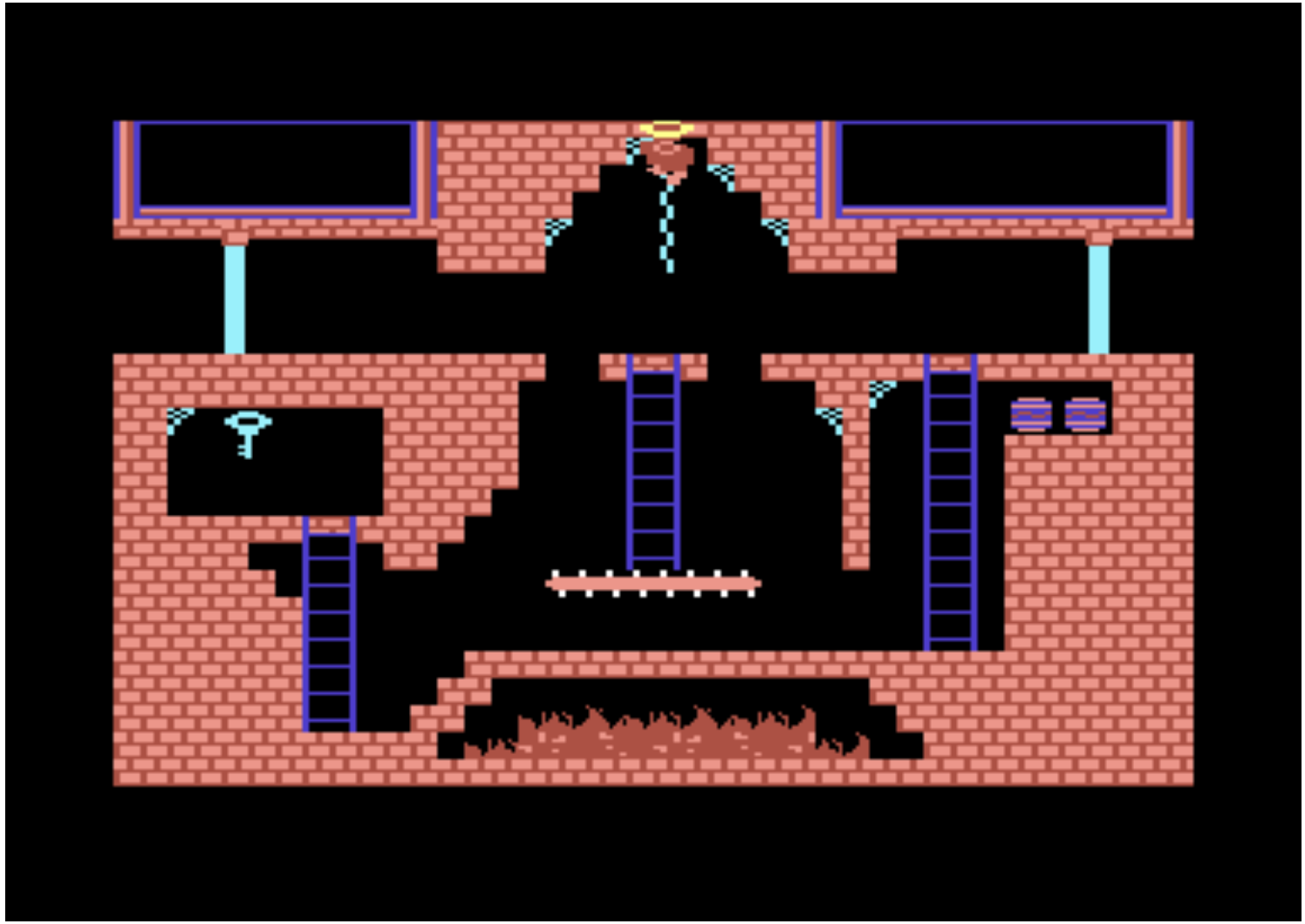
- The higher the TD-error δ, the larger the probability for a specific τ to be sampled.

In practice, given a trajectory τ, the probability of sampling it is given by the following equation:

$$P(\tau) = \frac{p_\tau^\alpha}{\sum_k p_k^\alpha}$$

where p_τ is $|\delta_\tau + \epsilon|$ with ϵ being a small positive number ensuring that the probability of sampling a trajectory remains positive even in the edge case where $\delta = 0$ and $\alpha \in [0, 1]$ is a tunable hyperparameter.

While PER helps improving the overall training process greatly it still does not help with environments with **sparse rewards**



Hindsight Experience Replay (HER): we have seen that a typical experience replay buffer stores trajectories in the following form:

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \\ s_t & a_t & \vdots & \\ s_t & \vdots & \vdots & \\ \vdots & \vdots & \vdots & \\ & & & \end{pmatrix}$$

We have seen that a typical experience replay buffer stores trajectories in the following form:

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ & & & \end{pmatrix}$$

In **sparse reward** environments most of the entries associated to $r_t = 0$.

$$D = \begin{pmatrix} s_t & a_t & 0 & s_{t+1} \\ s_t & a_t & 0 & \vdots \\ s_t & a_t & 0 & \vdots \\ s_t & a_t & 0 & \vdots \\ s_t & \vdots & 0 & \vdots \\ \vdots & \vdots & 0 & \vdots \\ & & & \end{pmatrix}$$

This can make training very hard, as almost no positive feedback is given for constructing a TD-target e.g.:

$$y_t^{DQV} = r_t + \gamma V(s_{t+1}; \Phi^-).$$

Therefore, we would like to have a way that allows us to construct informative trajectories even when the reward signal is $r_t = 0$

- The general **idea** is to divide a RL task into a set of possible goals $\mathcal{G}$, that are arguably easier to maximize than the reward signal r_t.
- Every goal $g \in \mathcal{G}$ corresponds to some reward function $r_g : \mathcal{S} \times \mathcal{A} \rightarrow \Re$

An episode starts with sampling a state-goal pair from some distribution $p(s_0, g)$, and we fix this goal for the entire length of the RL episode.

At every time-step t the agent receives as input, not only the current state s_t, but also the goal which has been sampled g, therefore defining the agent's policy as

$$\pi = \mathcal{S} \times \mathcal{G} \rightarrow \mathcal{A}.$$

The reward then becomes $r_t = r_g(s_t, a_t)$ and the state-action value function Q gets slightly modified to

$$Q^{\pi}(s_t, a_t, g) = \mathbb{E}[r_t | s_t, a_t, g]$$

as it does not only depend on a state-action pair only but also on g.

By **conditioning** every state, as well as reward signal, on a certain goal the transitions which get stored in the Experience Replay Memory Buffer will now look like:

$$D = \begin{pmatrix} s_t \parallel g & a_t & r_t(s_t, a_t, g) & s_{t+1} \parallel g' \\ s_t \parallel g & a_t & \vdots & \vdots \\ s_t \parallel g & a_t & \vdots & \vdots \\ s_t \parallel g & a_t & \vdots & \vdots \\ s_t \parallel g & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \end{pmatrix}$$

Some properties/limitations of Hindsight Experience Replay:

- Intuitively what HER does, is **modifying** failed experiences through a process which is called Goal Relabelling, or Reward Shaping.
- HER may be seen as a form of implicit curriculum learning as the goals used for replay naturally shift from ones which are simple to achieve to more difficult ones.
- It works on top of a **strong** assumption: namely that given a certain state s we can easily find a goal g which is satisfied:
  $m : \mathcal{S} \rightarrow \mathcal{G}$ s.t. $\forall_{s \in \mathcal{S}} f_m(s) = 1$
- While this assumption is not very restrictive, how does one define the set of possible goals $\mathcal{G}$ as well as the sampling strategy?

Inefficient Exploration

In traditional Deep-Q Networks, the agent selects actions based on the maximum Q-value for each state, hoping this will result in the optimal policy $\pi^\star$:

$$\pi^\star(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ Q^\star(s, a; \theta) \text{ for all } s \in \mathcal{S}.$$

- Note that in Deep-Q Networks the estimation of the different state-action values **depends** from the parameters of the network θ.
- Different $\theta \rightarrow$ Different $Q(s, a)$ values!

Noisy Networks

Noisy Networks add a small amount of noise to the weights of the neural network θ.

This noise is learned during training and helps the agent to explore more of the state-action space.

The Noisy Network architecture can be described by the following equation:

$$Q(s, a) = \mathbf{W}(s + \epsilon * z)$$

where s is the state the agent is visiting, $\mathbf{W}$ is a weight matrix, ϵ is a learned parameter that controls the amount of noise, and z is a random variable sampled from a normal distribution with mean 0 and standard deviation 1.

The Huber Loss

So far, each algorithm that we have encountered minimizes the Mean Squared Error (MSE) loss function, defined as the average of the squared differences between the predicted value estimates and the "true" values (TD-target).

While working well in practice, the MSE:

- Is sensitive to outliers and tends to penalize large errors more heavily than small errors.
- This can be problematic in RL, where actions can have long-term consequences and a single large error can have a significant impact on the overall performance of the agent.

The Huber Loss

It is possible to modify the MSE loss such that it becomes less sensitive to outliers:

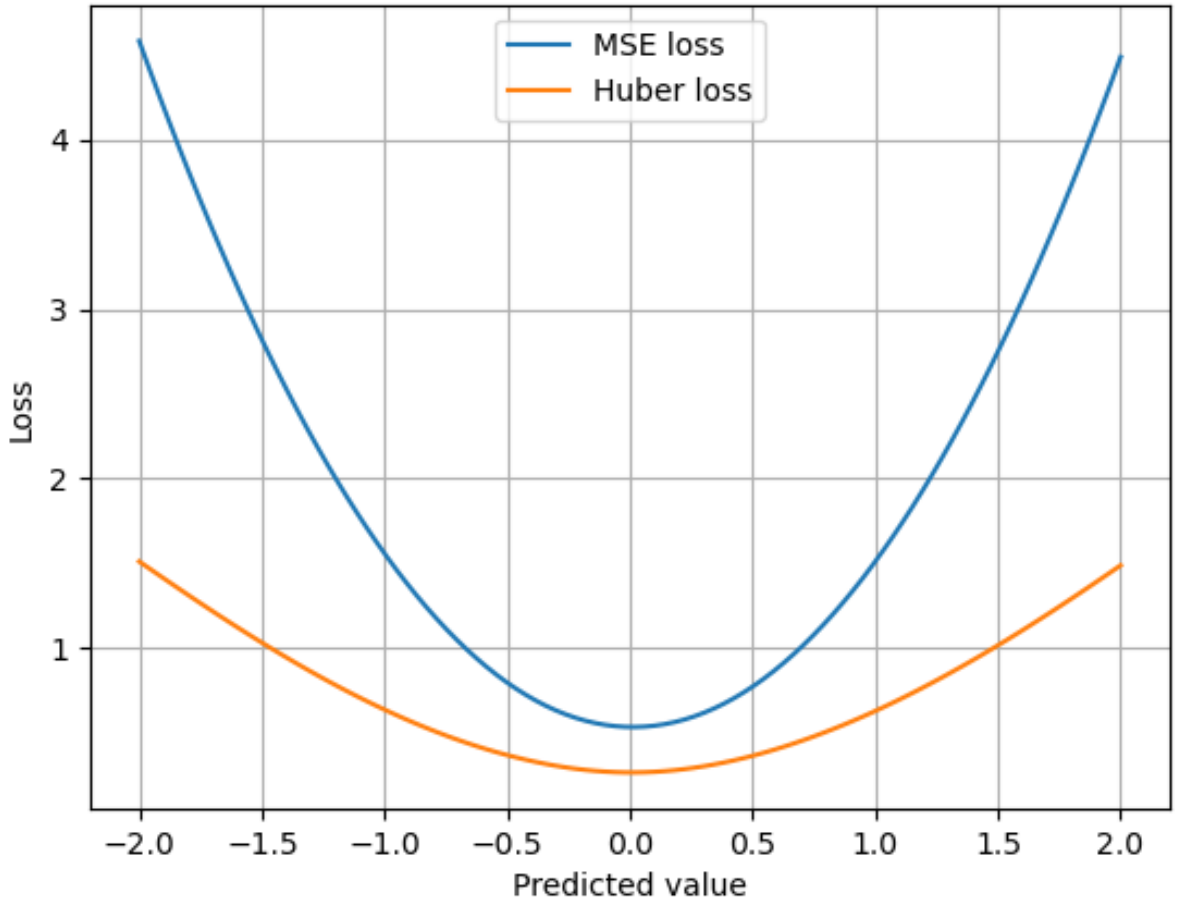
$$\mathcal{L}(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2, & \text{if } |y - f(x)| \leq \delta \\ \delta(|y - f(x)| - \frac{1}{2}\delta), & \text{otherwise} \end{cases}$$

We do this by having two components governed by the hyperparameter δ:

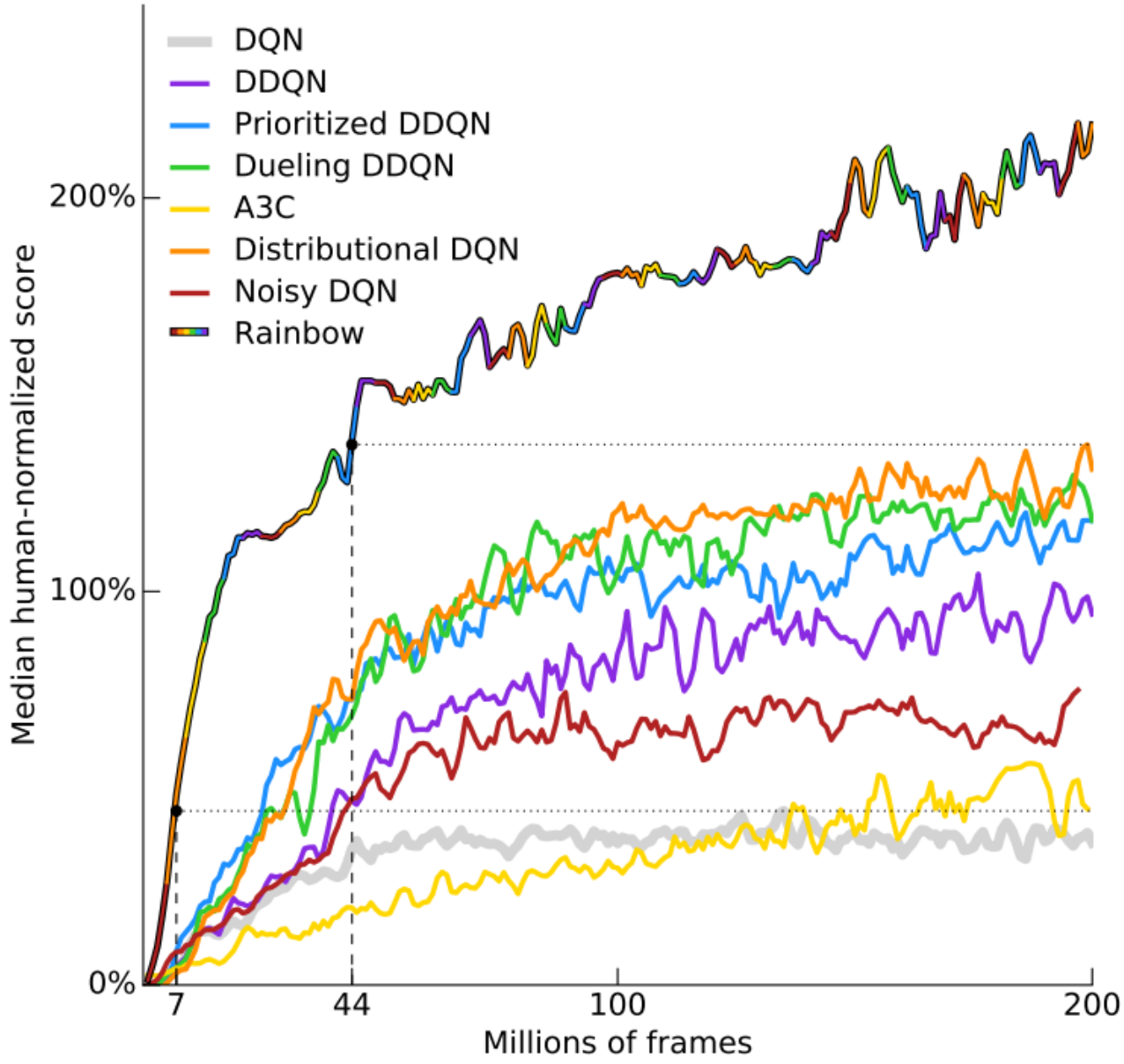
- A linear component (first case) for small errors
- A quadratic component (second case) for larger errors

The linear component makes the agent less sensitive to outliers, while the quadratic component still penalizes large errors.

The Huber Loss



Combining everything together: the allmighty **Rainbow** architecture!



A final note on Deep-Q Network's objective function!

Given a training iteration i, differentiating DQN's objective function with respect to θ gives the following gradient:

$$\nabla_{\theta_i}\mathcal{L}(\theta_i) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1}\rangle \sim U(D)}\Bigg[\Big(r_t + \gamma \max_{a\in\mathcal{A}} Q(s_{t+1}, a; \theta_{i-1}^{-}) - Q(s_t, a_t; \theta_i)\Big)\nabla_{\theta_i} Q(s_t, a_t; \theta_i)\Bigg].$$

Which makes the DQN algorithm, just like all other variants of today's Deep-Q Networks, a **semi-gradient method**.

Note in fact that the gradient is not defined for the TD-target due to the presence of the $\max$ operator.

References

- Ernst, Damien, Pierre Geurts, and Louis Wehenkel. "Tree-based batch mode reinforcement learning." Journal of Machine Learning Research 6 (2005).
- Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." nature 518.7540 (2015): 529-533.
- Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." International conference on machine learning. PMLR, 2016.
- Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 30. No. 1. 2016.
- Sabatelli, Matthia, et al. "The deep quality-value family of deep reinforcement learning algorithms." 2020 International Joint Conference on Neural Networks (IJCNN). IEEE, 2020.

References

- Fortunato, Meire, et al. "Noisy networks for exploration." arXiv preprint arXiv:1706.10295 (2017).
- Schaul, Tom, et al. "Prioritized experience replay." arXiv preprint arXiv:1511.05952 (2015).
- Andrychowicz, Marcin, et al. "Hindsight experience replay." Advances in neural information processing systems 30 (2017).
- Hessel, Matteo, et al. "Rainbow: Combining improvements in deep reinforcement learning." Proceedings of the AAAI conference on artificial intelligence. Vol. 32. No. 1. 2018.
- Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. MIT press, 2018.

See you next week