Imperial College London

# A Shallow Dive in Deep Reinforcement Learning - pt. 1 : Tabular & Q-Learning

DS3 2019 - Ecole Polytechnique - Google DeepMind

Pierre Harvey Richemond

# Speakers

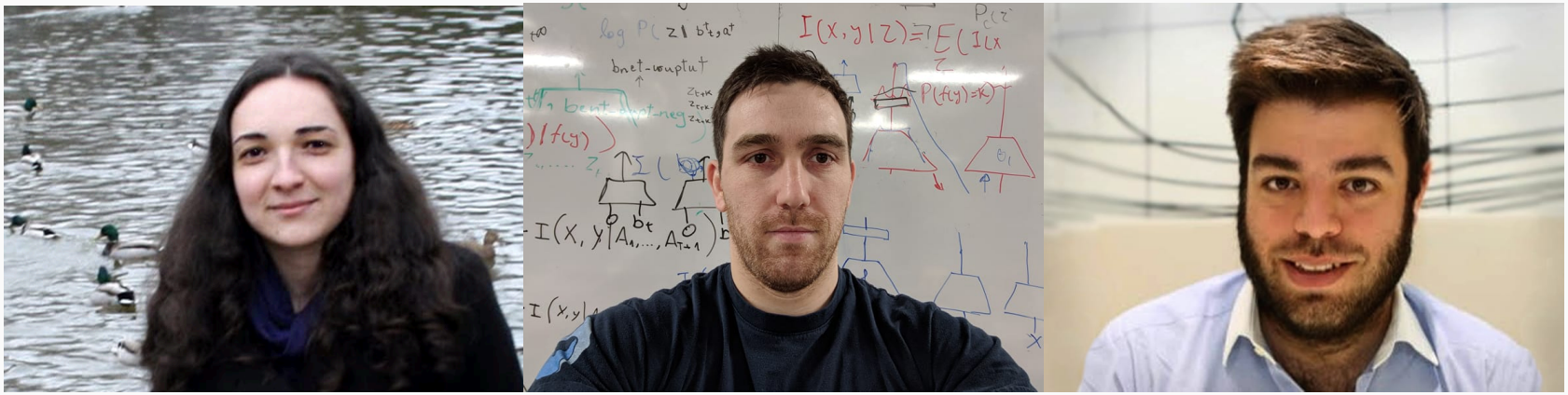

**Figure 1:** Diana Borsa, Bilal Piot, Pierre Richemond

# RL in one minute !

- Reinforcement learning = adaptive, extreme trial-and-error
- Goal = solve sequential decision problems.
- To quote Remi Munos (head of DeepMind Paris), 'a child doesn't learn to ride a bicycle by solving equations; they try, fall, learn a bit and try again'
- Deep RL = RL combined with deep neural networks
- RL takes into account new, recent experience ; in that sense it goes beyond the standard supervised vs. unsupervised learning paradigm, and enables machine learning on an evolving dataset.
- There are way too many references to quote, but a great and just updated starting point is the classic book [Sutton and Barto, 2018].

# Examples of RL - Successes & use cases

- Flying stunt with a helicopter
- More generally, robotics control
- Potentially, self-driving cars
- Superhuman performance in board games (Backgammon, Chess, Go) just from rules
- Control of a power station with a view of reducing energy consumption
- Trade the financial markets
- Learn to play video games better than humans just from pixels and score - see video.

RL can be seen as a form of supervised, optimal control.

# Chronology

- 1984 Rich Sutton's thesis
- 1989 Q-learning (Watkins)
- 1991 REINFORCE (Williams & Peng)
- 2006 Monte-Carlo Tree Search (Szepesvari)
- 2009 David Silver's Ph.D. thesis [Silver, 2009]
- 2013 Deep Q-Networks
- 2015 DQN scales - *Nature* paper
- 2016 Actor critic + deep networks
- 2017 Soft Q-learning = soft actor-critic ; AlphaGo Zero

Key algorithms are 25+ years old
*(but trained at scale only now)*

# Key basic concepts

- Environment
- Agent
- State
- Action
- Next state
- Reward

# The reward is the most important

- A reward $R_t$ is a scalar feedback signal
- Indicates how well the agent is doing at step $t$
- The agent's job is ALWAYS to maximize cumulative reward
- The *reward hypothesis* posits *all goals* can be described by the maximisation of the *expected cumulative reward*
- A philosophical debate...

# Examples of rewards

- Helicopter : + reward for going higher and following trajectory ; - reward for crashing
- Chess or Go : no reward for most moves (sparse signal), +1 for winning the game, -1 for losing
- Manage an investment portfolio : reward is the change in dollar market value of the portfolio
- Humanoid robot walk : +something for forward motion, -1 for falling over
- Atari : reward is the change in score in your emulator

Sometimes the reward is exogeneous (imposed by the world you're in), sometimes it's designed-engineered by the algorithm creator.

# Sequential Decision Making

- Goal : select actions in order to maximise total future reward
- Actions may have long term consequences
- Rewards may be delayed
- It may be better to practice delayed gratification, and sacrifice immediate reward for the longer-term gain.

# Agent and environment : the RL loop.

At each step $t$, the agent

- Executes action $A_t$
- Receives observation $O_t$
- Receives scalar reward $R_t$

The environment then:

- Receives action $A_t$
- Emits observation $O_{t+1}$
- Emits scalar reward $R_{t+1}$

Then, $t$ is incremented (at environment step).

# History, transitions, and state

- The history is the sequence of transitions.
- A transition is a triplet state, action, reward
- This represents the sensorimotor stream of a robot, or (embodied) agent
- The history can be written as $H_t = O_1, R_1, A_1, \cdots, A_{t-1}, O_t, R_t$
- State is the information used to determine what happens next
- State is a function of history : $S_t = f(H_t)$
- Examples : Chess, Atari, Humanoid robot walk

# Markov state

Definition : a state is *Markov* iff

$$\mathbb{P}[S_{t+1}|(S_1, \cdots, S_t)] = \mathbb{P}[S_{t+1}|S_t]$$

- ie 'the future is independent of the past, given the present'
- ie 'all of the history is contained in the current state'
- ie 'the state is a sufficient statistic of the future'
- ie 'we are memory-less'

We can *markovianize* (sic) the state by adding more information to it - for instance, the last few frames in a video game so as to be able to infer speed and momentum.

# To summarize till now

- Given a certain environment (game abstraction), an agent (player) selects between actions $a$ available to her, and in doing so, collects a reward $r$ and moves from state $s$ to state $s'$.
- The agent's objective is to pick actions and build a trajectory so as to maximize the sum of their rewards.
- Rewards and states are not known in advance.

# The discount factor

- $\gamma$ is also known as a *discount factor* that controls how far in the future we want to look
- $\gamma$ is here to make the math work; better to build first intuitions without
- In practice it's an important parameter, slightly less than 1, that can be (carefully) tuned for performance.

# Major components of an RL agent

An RL agent may include one or more of these components :

- Policy : agent's behaviour or function
- Value function : how good is each state of action
- Model : agent's representation of the environment

# Policy

- The policy is the agent's behaviour.
- It is a map (function) from state to action, or a distribution over actions given states.
- This means:
- Deterministic policy: $a = \pi(s)$
- Stochastic policy: $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$.

A policy fully defines the behaviour of an agent, and it only depends on the current state (not the history). Hence policies are *stationary*, that is, time-independent.

# Value function

- A value function is a prediction of future reward.
- It is used to evaluate the goodness/badness of states
- And therefore to select between actions !
- Example:

$$V_\pi(s) = \mathbb{E}[r_{t+1} + r_{t+2} + \cdots | S_t = s]$$

# Model

- A model predicts what the environment will do next. (Can you say 'imagination' ?)
- $\mathcal{P}$ predicts the next state
- $\mathcal{R}$ predicts the next (immediate) reward.

# RL Agent Taxonomy : categorizing RL agents

There are many possible ways (hence algorithms) to do RL !

- Value-based : value function ; no policy
- Policy-based : policy ; no value function
- Actor-critic : policy and value function, together.

In this course (pt. 1), we will focus on value-based methods.

Another possible distinction is model-based versus model-free :

- Model-free : policy and/or value function; but no model
- Model based: policy and/or value function; model

In this course we will focus on model-free methods.

# Exploration vs Exploitation

- RL is like trial-and-error; recall that rewards and states are not known in advance.
- The agent should discover a good policy from experiencing the environment. Doing so should be accomplished without sacrificing too much reward along the way !
- Exploration finds more information about the environment
- Exploitation exploits known information to maximise reward
- It is important to do both, in a balanced way
- If always following the same policy, some states may never be visited = high possibility of being stuck with a suboptimal policy in stochastic environments.
- Examples : restaurant selection, natural resources mining, game playing

# Markov Reward Processes

- Markov chains are underpinning the transition dynamics in the states of our environment.
- Markov Rewards Processes are Markov chains with rewards attached !
- Think of it as collecting a string of rewards $r_i$ associated with the states $s_i$ taken via the (random) walk on the chain.
- Note that in RL the whole point is for the walk not to be random anymore.
- The n-step return $R_n = \sum_{i=1}^{n} r_i$ is just the sum of $n$ rewards collected.
- It is a random quantity whose expectation (the *expected return*) we'd like to maximise ; mathematically, RL is cast as an optimisation problem.

# Markov Decision Processes

- MDPs are Markov Decision Processes
- They are Markov Rewards Processes where now actions are informing randomly the next state you're in, i.e., MRPs with decisions.
- So every state $s_i$ becomes a couple of a state and action $(s_i, a_i)$.
- Because of the introduction of those actions, the maximisation on previous slide becomes an active maximisation process
- That is, RL is also an control problem - finding the best policy, in order to optimise the future[1].

[1] The flipside of this is *prediction*, that is, evaluate the future given a specific policy.

# MDPs, illustrated

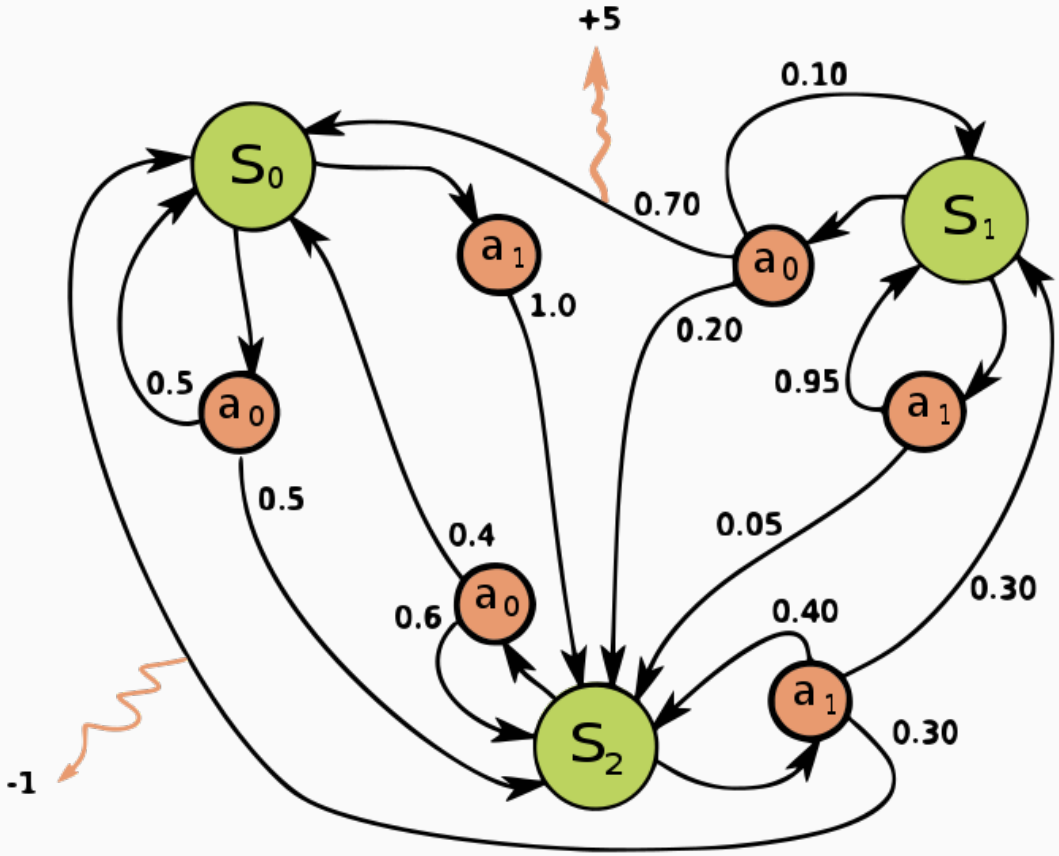


**Figure 2:** Toy MDP example with three states and two actions per states. Note transition probabilities and attached scalar rewards.

# Why all this ?

Because we are Markov, we will only ever be interested in two states: the one we are coming from ($s$) and the one we are going to ($s'$). We can discard anything that comes before $s$. There are only two indices here, so matrices (rather than more general $d$-dimensional tensors) will be all we need. Notation-wise :

- $T(s, a, s')$ will be the *transition dynamics*, or probability to go to state $s'$ after having taken action $a$ in state $s$.
- $R(s, a, s')$ will be the reward associated with going to from state $s$ to $s'$ when taking action $a$.

Markov Decision Processes are the typical formulation used for reinforcement learning problems in the literature.

# Q and V : V

- V numbers are known as the state-value functions.
- They depend on policy $\pi$, so we often denote them by $V_\pi(s)$.
- They indicate how good a given state is (the long-term value of a state s) for a player following the policy $\pi$:

$$V_\pi(s) = \mathbb{E}_\pi[\sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i} \quad | \quad S_t = s]$$

- i.e. the *expected cumulative return* from starting in state s, and following policy $\pi$ thereafter.

# Q and V : Q

- Q (for quality !) numbers go one step beyond, and are action-value functions (shorthand for action-state-value functions).
- These numbers contain more information than the $V(s)$ above.
- They indicate how good it is, in state s, to *first take action a* before continuing to follow policy $\pi$ for all other next actions.
- They are denoted by $Q_\pi(s, a)$ :

$$\boxed{Q_\pi(s, a) = \mathbb{E}_\pi[\sum_{i=0}^{\infty} \gamma^i \cdot r_{t+i} \quad | \quad S_t = s, A_t = a]}$$

- i.e. the *expected cumulative return* from starting in state $s$, taking first action $a$, and then following policy $\pi$ thereafter.

# Q and V redux : * optimality

- In general, these numbers depend on the policy we use to choose our actions. A given chess position is not as good if played by me ($\pi_1$) or by Kasparov ($\pi_2$) !
- The optimal Q and V numbers, i.e. the ones that would be attainable by a perfect controller or player $\pi^*$, are denoted by a star :

$$V^*(s) = V_{\pi^*}(s) = \max_{\pi} \quad V_{\pi}(s)$$

- Similarly we define

$$Q^*(s,a) = Q_{\pi^*}(s,a) = \max_{\pi} \quad Q_{\pi}(s,a)$$

.

# Discounted formulae - Recap

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} \ldots + \gamma^{T-t+1} r_T + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

$$\forall s \in S \quad V^\pi(s) = E^\pi \left[ \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \, | s_0 = s \right]$$

$$\forall s \in S, a \in A \quad Q^\pi(s,a) = E^\pi \left[ \sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \, | s_0 = s, a_0 = a \right]$$

$$V^\pi(s) = E^\pi \left[ r(s,a) + \gamma \sum_{t=0}^{\infty} \gamma^t r(s_{t+1}, a_{t+1}) \, | s_0 = s, a \sim \pi(a|s) \right]$$

$$Q^\pi(s,a) = E^\pi \left[ r(s,a) + \gamma \sum_{t=0}^{\infty} \gamma^t r(s_{t+1}) \, | s_0 = s, a_0 = a \right]$$

$$\forall s \in S \quad V^\pi(s) = \sum_a \pi(s,a) Q^\pi(s|a) = Q^\pi(s, \pi(s))$$

# Policy evaluation and policy improvement

A policy is greedy wrt action-value functions $Q$ if

$$\pi(s) \in \mathbf{argmax}_a \, Q^\pi(s, a)$$

Policy improvement theorem : if $\pi'$ is greedy w.r.t $Q^\pi$, then $\pi' \geq \pi$.

Optimality theorem : if $\pi$ is greedy w.r.t its own Q-function $Q^\pi$, then $\pi$ is optimal.

Hence *we can alternate between policy evaluation and policy improvement* ($\pi \leftarrow greedy(Q)$ ) steps, and hope to converge to an optimum both in $\pi^*$ and $V^*$.

# Monte-Carlo methods

Learn $Q^\pi(s, a)$ via sampling:

- Random choice of starting state $s$
- Random choice of an action $a$ (exploration starts)
- Follow policy $\pi$ and observe gain $R_t$
- Do that 'infinitely many' times, and average $Q^\pi(s, a) = E^\pi[R_t]$
- Perform policy improvement: $\pi(s) = \textbf{argmax}_{a \in A} Q^\pi(s, a)$

Problem = this is a high-variance method.

# TD Methods - 1

In the deterministic case:

$$\begin{aligned} V(s_t) &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \dots \\ &= r_t + \gamma V(s_{t+1}) \end{aligned}$$

In practice,

$$\delta_t = [r_t + \gamma V(s_{t+1})] - V(s_t) \neq 0$$

The *TD-error* is nonzero in practice (and we usually build algorithms via attempting to minimize it).

# TD Methods - 2

Widrow-Hoff like update rule.

$$V^{t+1}(s_t) \leftarrow V^t(s_t) + \alpha\left(r_t + \gamma V^t(s_{t+1}) - V^t(s_t)\right)$$

( $\alpha$ is a learning rate, $V(s_t)$ is a target).

# SARSA (on-policy) vs Q-learning (off-policy)

SARSA update:

$$Q^{t+1}(s_t, a_t) \leftarrow Q^t(s_t, a_t) + \alpha \left( r_t + \gamma Q^t(s_{t+1}, a_{t+1}) - Q^t(s_t, a_t) \right)$$

Q-learning update:

$$Q^{t+1}(s_t, a_t) \leftarrow Q^t(s_t, a_t) + \alpha \left( r_t + \gamma \max_b Q^t(s_{t+1}, b) - Q^t(s_t, a_t) \right)$$

# Qs and Vs redux : * optimality, 2

- If we knew the optimal V numbers perfectly (through a *critic* or *value function iteration*) and we had a model of our world, we could act greedily towards choosing which state to go to.
- If we knew the $Q^*$ numbers perfectly for each action, we would be in even better shape (why ?).
- The goal of value-function based RL is to determine and act according to those $Q^*$ numbers.
- So we choose to do Q-learning : we will learn the optimal action-values $Q^*(s, a)$.

# The Bellman Equation

- The Bellman equation is possibly the most important equation in all of reinforcement learning.
- It's a time consistency property for optimal value functions. It applies to either $Q$ or $V$, and tells us how to do our neural network training.
- At optimality we get (easy to prove just by expanding the expectation sum one step forward) :

$$\boxed{Q^*(s,a) = \sum_{s'} T(s,a,s') \left[ R(s,a,s') + \gamma \max_{a'} \quad Q^*(s',a') \right]}$$

- Notice we have $Q^*(s,\cdot)$ on the left, but $Q^*(s',\cdot)$ on the right !

# Bellman Equation : intuition

- In the case where we are looking back at past, realized transitions, we can simply set $T(s, a, s') = 1$ for the only one transition that has actually happened, and 0 elsewhere.
- This allows us to give an ex-post, intuitive decomposition of the Bellman equation in two parts:
- A horizontal (time axis) part: for the actual action taken $a_{actual}$,

$$Q^*(s, a_{actual}) = r(s, a_{actual}) + \gamma V^*(s')$$

- A vertical (optimal action) part:

$$V^*(s') = \max_a \quad Q^*(s', a)$$

- Taken together this is the classic form

$$\boxed{Q^*(s, a) = r(s, a) + \gamma \max_{a'} \quad Q^*(s', a')}$$

.

# $\epsilon$-greedy : exploitation vs exploration, redux

Making sure we are not stuck in a suboptimal policy. In general (99% of the times) we will use $\arg\max Q(s, a)$ as an implicit policy.
However:

- Simplest idea for ensuring continual exploration
- All $m$ actions are tried with non-zero probability
- We pick a small number $\epsilon$ before running the algorithm.
- With probability $1 - \epsilon$, choose the greedy action
- With probability $\epsilon$, choose an action at random.
- $\pi(a|s) = \frac{\epsilon}{m} + (1 - \epsilon)\delta_{a=argmaxQ(s,a^*)}$

We can show theoretically that this leads to policy improvement.

# Theoretical Q-learning in 1 slide.

Q-learning is learning the numbers that enable us to act.

- The target policy $\pi$ is greedy.
- If now we are being $\epsilon$-greedy, everything simplifies and we get the Q-learning target $R_{t+1} + \max_{a'} \quad Q(s_{t+1}, a')$
- As such, we update $Q(s, a)$ in the *direction* of the target :

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha(R + \max_{a'} \quad Q_t(s_{t+1}, a') - Q_t(s, a))$$

(with $\alpha$ a given learning rate)

**Theorem**

*As we take more iterations, tabular Q-learning control converges to the optimal action-value function,* $Q(s, a) \rightarrow Q^*(s, a)$.

Proof methods almost always invoke the Banach fixed point theorem by showing the Q- (Bellman) operator is a contraction - that is, its Lipschitz constant is $< 1$.

# Advanced tabular guarantees : convergence rate

**Theorem**

*Assume a tabular, finite horizon, episodic MDP with number of states S, number of actions A, time horizon H, and number of time-steps T. Then, there exists an optimistic modification (exploration bonus) to the value iteration algorithm that achieves a provably optimal regret bound of* $O(\sqrt{HSAT})$ *with high probability.*

The proof is highly technical [Gheshlaghi Azar et al., 2017], and makes extensive use of concentration inequalities (Azuma-Hoeffding, Bernstein-Freedman).

# Description of our theoretical, tabular algorithm

We will train in *episodes* as follows.

- Initialize $Q(s, a)$ for each $s, a$ arbitrarily, and $Q(terminalstate, \cdot) = 0$.
- For each episode, repeat :
- Initialize $S$
- Repeat (for each step of episode:)
- Choose A from S using $\epsilon$-greedy policy derived from $Q$
- Take action A, observe R, S'
- $Q_{t+1}(S, A) \leftarrow Q_t(S, A) + \alpha(R + \max_a \quad Q_t(S', a) - Q_t(S, A))$
- $S \leftarrow S'$

# In practice : Tabular vs function approximation

First problem : there is a potential infinity of screenshots $s$. We will hence not be able to implement $Q(s, a)$ as a lookup table, but rather, will resort to using a neural network for *generalization*.

- Tabular : nice theoretical guarantees and regret bounds, but intractable...
- Function approximation : may or may not converge... but works, at scale !
  In general we use neural networks for function approximation and we stabilize learning using a various set of RL-specific tricks (not just NN tricks).
- We refer to the weights of our NN as $\theta$, so we get $Q_\theta(s, a)$.

# Atari Example

Our canonical example. The rules of the game are unknown, and we want to to learn directly from interactive game-play. We pick actions on joystick, see resulting pixels and scores.

- Environment = emulator
- States = stacks of 84x84x3 pixels & score (!).
- Notion of transition between states : from one screenshot to another.
- Actions = joystick button presses
- Rewards = changes in score

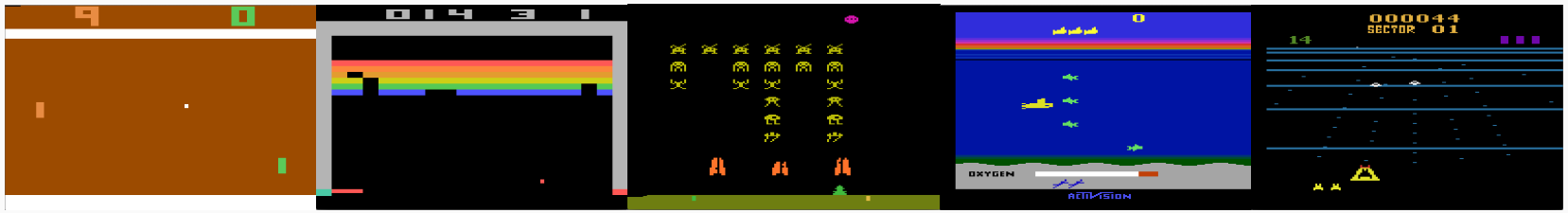


**Figure 3:** Screenshots from five Atari 2600 Games: (*Left-to-right*) Pong, Breakout, Space Invaders, Seaquest, Beam Rider. Taken from [Mnih et al., 2013].

# Atari - DQN Architecture

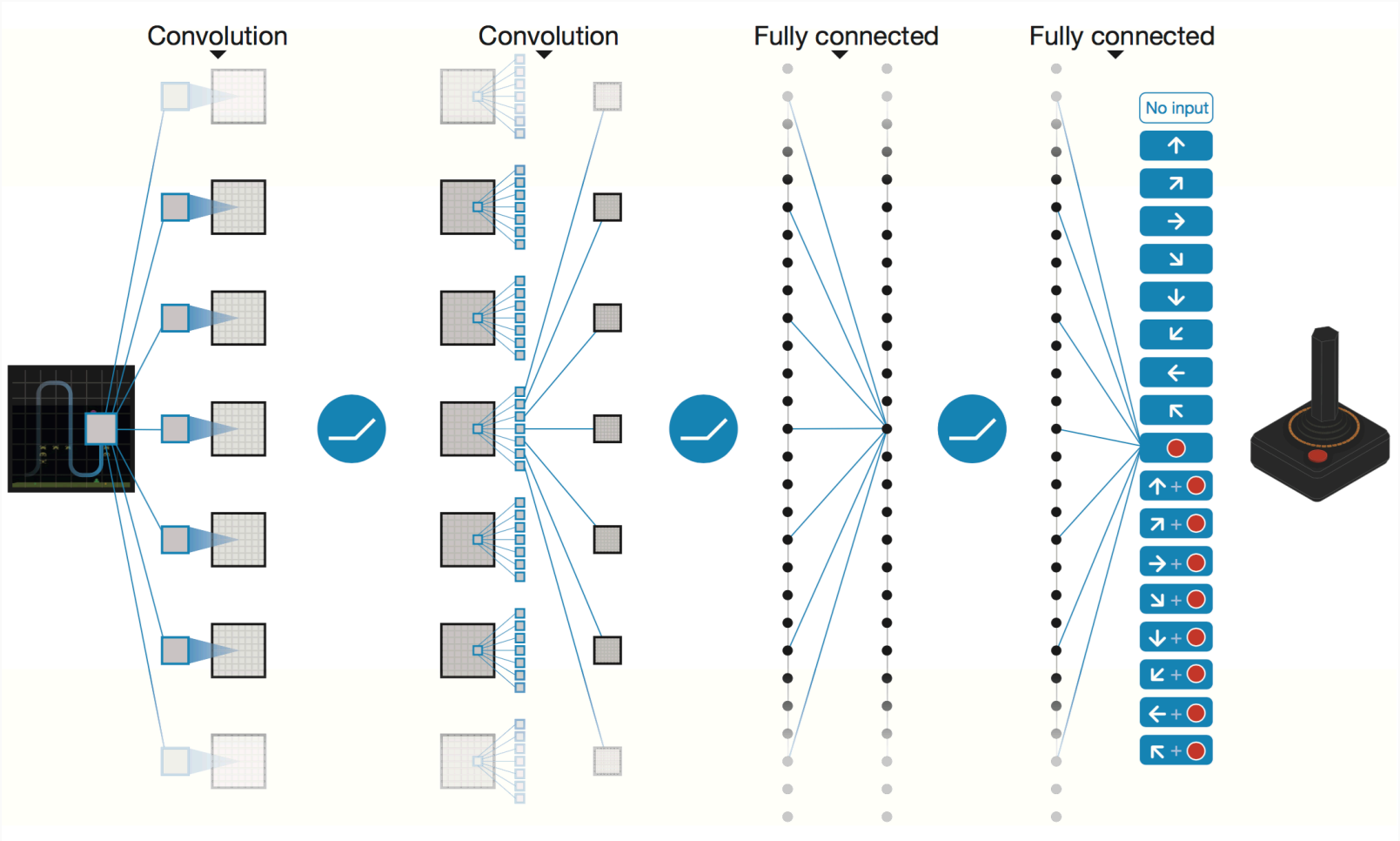


**Figure 4:** Architecture for the DQN. Input $s$ is stacked frames, output is an 18-element action-value vector $Q_\theta(s, a)$. Taken from [Mnih et al., 2015].

# In practice

- To learn, we will keep track of a table of 18 potential numbers (one per Atari action) for each state/screenshot.
- We will initialize the $Q$'s randomly and then interact with the environment by playing, collecting rewards and observations, and placing them into a memory : an *experience replay* buffer.
- Then we will use the Bellman equation to get our network to output better and better estimates for the $Q^*$.
- As we do this we will also continue playing to get better experiential data, by following the (implicit, greedy) policy given by those new $Q$ values.

# Convolutional architectures

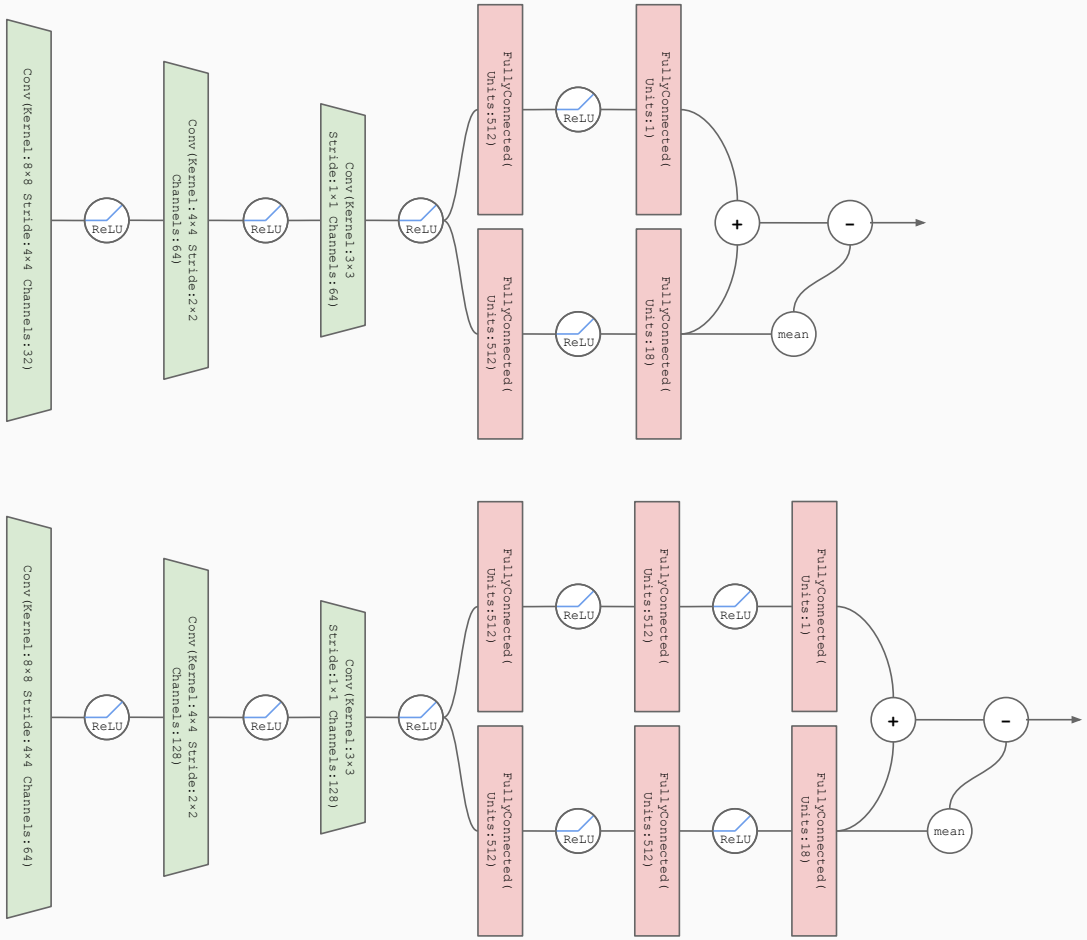


**Figure 5:** Two typical convolutional architectures for training Atari games. Note the shallow networks with at most 6 layers. Taken from [Pohlen et al., 2018].

# Bootstrapping : an ansatz

- Bootstrapping = wishful thinking - 'what if we were already at optimality ?'
- The Bellman equation is not valid before we reached a perfect policy ; we are going to pretend it is.
- So instead of pretending the difference between the left side and the right side of the equation (the *Bellman residual*) is zero, we will take its square as a loss function and iteratively minimize it, to 0, via gradient descent :

$$L_i(\theta_i) = \mathbb{E}_{\underbrace{(s,a,r,s') \sim U(D)}_{\text{randomly sampled transitions}}}\left[\Big(\underbrace{r + \max_{a'} \quad Q(s',a',\theta_i)}_{\text{target value } y_i} - Q(s,a,\theta_i)\Big)^2\right]$$

# Bootstrapping : action-vector view

- In this case, we can repeatedly *pick a realized past transition* $(s, a, r, s')$ and do the following :
- If we pass the destination screenshot $s'$ to the network, we will get an estimate for the value of the state by just computing the action-vector $Q^*(s', a)$ and taking its maximum. We add the effective reward we got on our way there, and make a note of this total (a good proxy estimate of $V^*(s') + r(s, a)$).
- The Bellman equation says that when applying our network to the original screenshot, picking only the value in the action-vector that matches the actual chosen action, should return that very number.
- Hence, we can enforce that by now training the network (using standard supervised learning methods !) *to return that target value*.
- We will do that, and then move to another transition $(s, a, r, s')$ and repeat in batches.

# Bootstrapping : action-vector view

- Our RL setting and specifically the Bellman equation enabled us to determine target values our neural network should aim to return.
- Viewed in this way, RL is a second-level algorithm or meta-algorithm that sits on top of supervised learning.
- It is therefore not surprising it needs to be stabilized even more !
- You can see that this principle of Q-learning will work with different types of neural networks.
- There will be interplay between neural architecture (deep) and reinforcement algorithm (RL).

# Two stabilizing tricks

- First stabilizing trick is to shuffle transitions (randomize their order in time) in our replay buffer[2];
- Otherwise, consecutive updates are too correlated in time (as opposed to the SGD IID assumption), and the network diverges.
- Second trick is to use two networks : one for $Q(s')$ prediction, another one for $Q(s)$ training. This also stabilizes training by preventing a negative feedback loop where the network chases its own tail (reduce correlations between target and predicted Q-values).
- These two methods are called *experience replay* and *target network freezing* by Google DeepMind [Mnih et al., 2015].

[2] Is this what dreams are made of ? There is clear neuroscientific evidence for replay in the brain, modulated by dopamine as a vector of surprise.

# Description of the practical Q-learning algorithm

**Algorithm 1** DQN algorithm with neural function approximation.

Initialize replay memory $D$ to capacity $N$
Initialize action-value network $Q$ with random weights $\theta$
Initialize target action-value network $\hat{Q}$ with weights $\theta^- = \theta$
**for** episode 1, $M$ **do**
  Initialize sequence with $s_1$
  **for** $t = 1, T$ **do**
    With probability $\epsilon$, select a random action $a_t$
    Otherwise select $a_t = arg\,\max_a Q(\phi(s_t), a; \theta)$
    Execute action $a_t$ in the emulator
    Observe reward $r_t$ and image $s_{t+1}$
    Store experience $(s_t, a_t, r_t, s_{t+1})$ in $D$
    Sample random minibatch of $(s_j, a_j, r_j, s_{j+1})$ from $D$
    Set $y_j = \begin{cases} r_j & \text{if episode terminates step } j+1 \\ r_j + \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
    w.r.t. network weights $\theta$ (SGD, RMSProp, ADAM ?)
    Every $C$ steps, reset $\hat{Q} = Q$

# DQN : Results

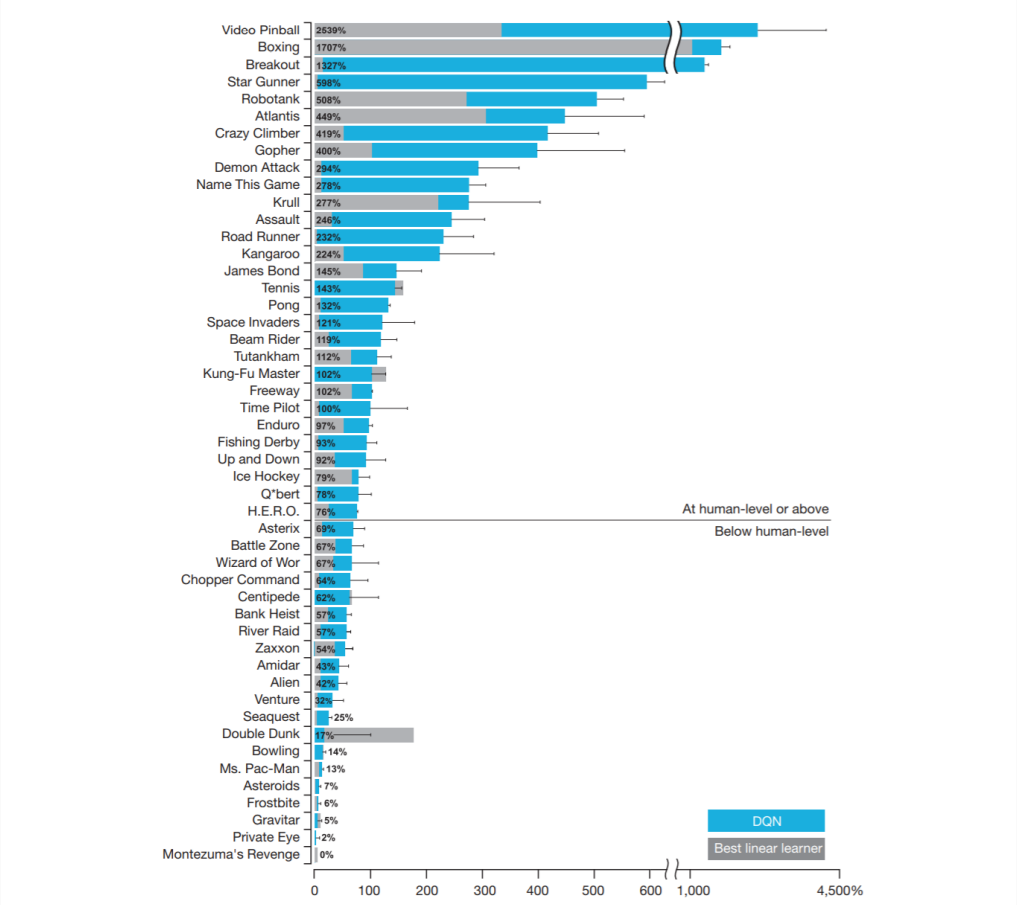


**Figure 6:** Performance on Atari games, normalized w.r.t. human professional games tester. DQN fares better on twitchy, reactive games than on planning-oriented ones. Training takes $\sim$ a day per game on a modern GPU [Stooke and Abbeel, 2018]. Taken from [Mnih et al., 2015].

# DQN : Monitoring convergence

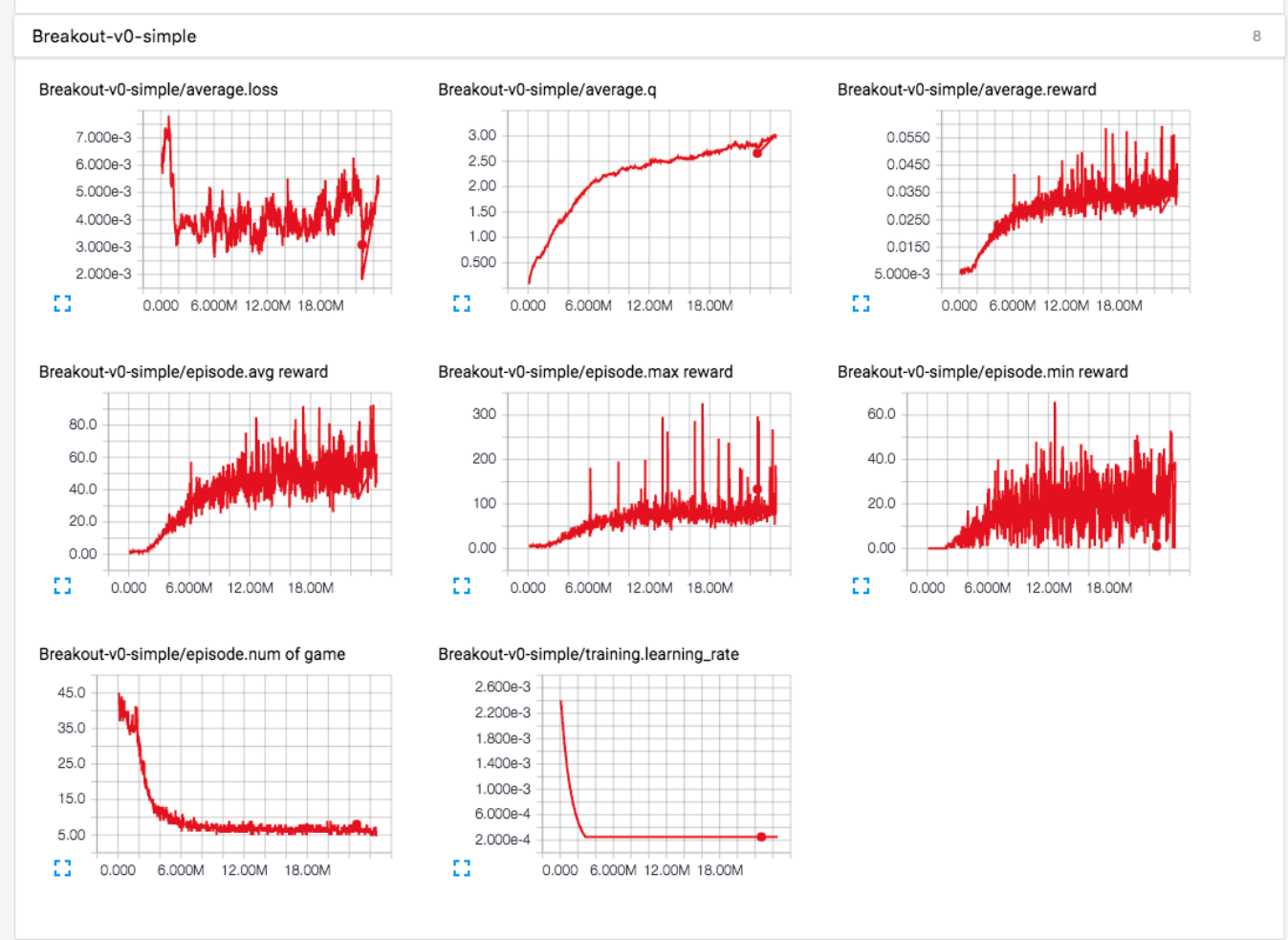


**Figure 7:** Deep RL convergence is much harder in practice than simple supervised learning - hence need to monitor indicators like reward & episode statistics with TensorBoard.

# Some DQN extensions - sample efficiency & performance

- Double DQN [van Hasselt et al., 2015] decouples action selection and evaluation, between both current and target networks, so as to avoid propagating the max of estimation noise - which makes the network overestimate action-values (upward bias).
- Duelling DQN [Wang et al., 2015] attempt to separate the learning of a baseline state value, and the relative advantage of taking each action in that state (centered on zero, ergo variance reduction).
- Prioritized Experience Replay [Schaul et al., 2015] replays 'surprising' transitions (those with a high Bellman residual error) much more often, in order to learn faster.
- NoisyNets for exploration [Fortunato et al., 2017] look to learn the variance of the exploration noise automatically, rather than relying on a preset $\epsilon$-greedy annealing schedule.

# Combining improvements : RAINBOW agent performance

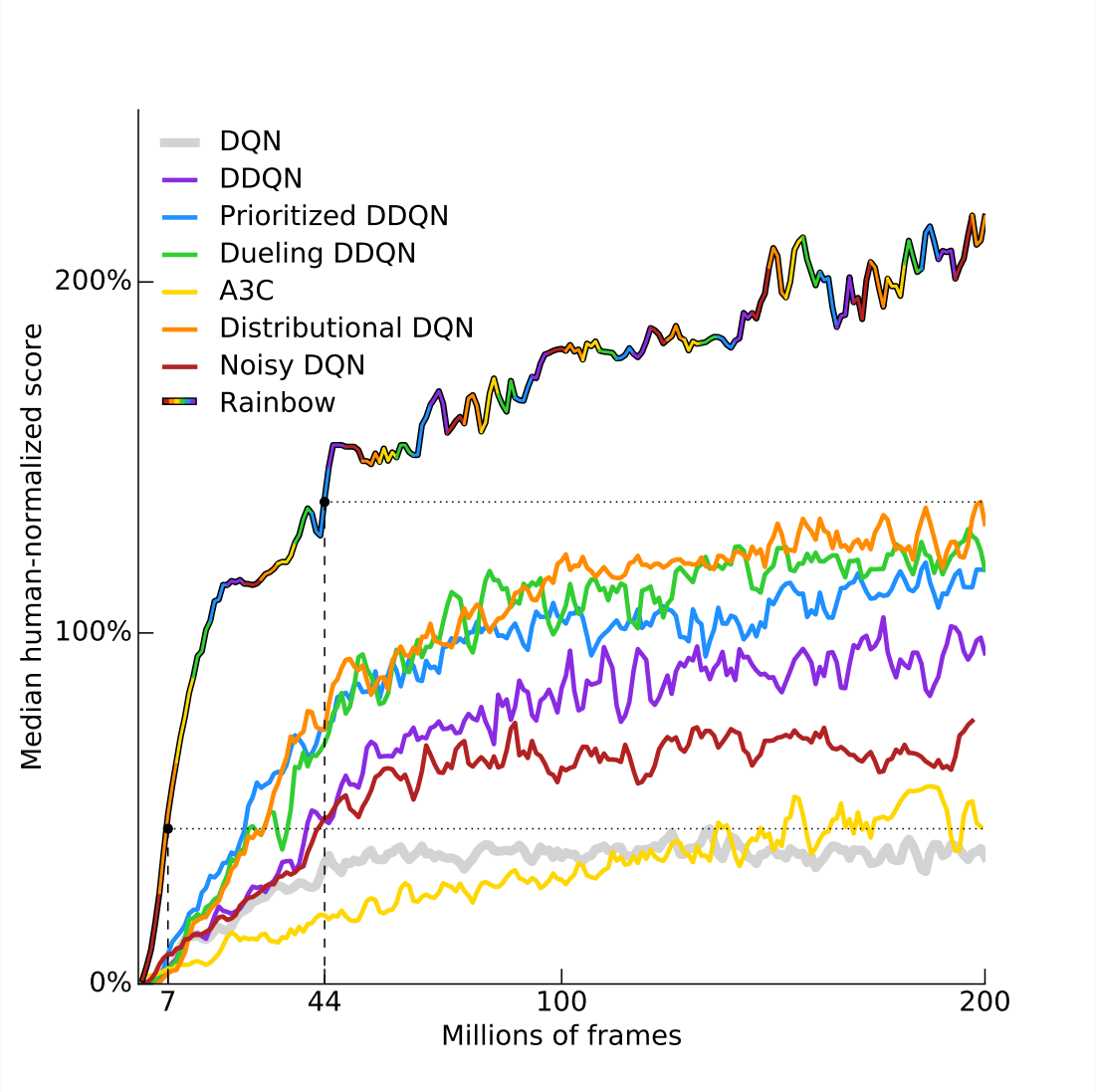


**Figure 8:** Data-efficient combination & ablation study for DQN extensions across 57 Atari games. Taken from [Hessel et al., 2017].

# A note on discounting

- Here we reintroduce the discount factor $\gamma$ that controls the weighting trade-off between immediate and future rewards. It also guarantees convergence of returns in potentially infinite MDPs. In practice you can try a value around 0.99.
- The discounted $n$-step return is

$$\sum \gamma^i r_i$$

- The discounted V-value estimate is

$$\mathbb{E}[\sum \gamma^i r_i \quad | s_0]$$

- The discounted Q-value estimate is

$$\mathbb{E}[\sum \gamma^i r_i \quad | s_0, a_0]$$

- Discounted Q-learning targets become

$$r + \gamma \cdot \max_b Q(s_{t+1}, b) - Q(s_t, a_t)$$

# Practical work with Colab

To access the Colab notebook for this session:

- Browse to `https://github.com/prichemond/ds3` and copy the *1ZL9xzASHzwSYQC6U0fDVMDuwotZxbP0M* suffix.
- Append this to *colab.research.google.com/drive/*, browsing to `https://colab.research.google.com/drive/1ZL9xzASHzwSYQC6U0fDVMDuwotZxbP0M`
- Once that page is loaded in your browser, please click on 'File - Save a copy in Drive'.
- From there on you're good to go, Colab is very similar to Jupyter Notebook.

# Github code repositories

TensorFlow

Deep-rl-tensorflow DQN Atari-RL Google-Dopamine Deepmind-TRFL

Keras

Keras-RL DQN-FlappyBird

Pytorch

RL-Adventure Rainbow VEL

OpenAI

Gym Baselines

# References i



Fortunato, M., Gheshlaghi Azar, M., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. (2017).
**Noisy Networks for Exploration.**
*ArXiv e-prints.*


Gheshlaghi Azar, M., Osband, I., and Munos, R. (2017).
**Minimax Regret Bounds for Reinforcement Learning.**
*JMLR.*


He, F. S., Liu, Y., Schwing, A. G., and Peng, J. (2016).
**Learning to Play in a Day: Faster Deep Reinforcement Learning by Optimality Tightening.**
*ArXiv e-prints.*

# References ii


Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2017).
**Rainbow: Combining Improvements in Deep Reinforcement Learning.**
*ArXiv e-prints*.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013).
**Playing Atari with Deep Reinforcement Learning.**
*ArXiv e-prints*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Ostrovski, G., Legg, S., and Hassabis, D. (2015).
**Human-level control through deep reinforcement learning.**
*Nature*.

# References iii



Pohlen, T., Piot, B., Hester, T., Gheshlaghi Azar, M., Horgan, D., Budden, D., Barth-Maron, G., van Hasselt, H., Quan, J., Večerík, M., Hessel, M., Munos, R., and Pietquin, O. (2018).
**Observe and Look Further: Achieving Consistent Performance on Atari.**
*ArXiv e-prints.*


Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015).
**Prioritized Experience Replay.**
*ArXiv e-prints.*


Silver, D. (2009).
**Reinforcement Learning and Simulation-Based Search in Computer Go.**
*Ph.D. thesis.*

# References iv



Stooke, A. and Abbeel, P. (2018).
**Accelerated Methods for Deep Reinforcement Learning.**
*ArXiv e-prints.*


Sutton, R. and Barto, A. (2018).
**Reinforcement Learning : An Introduction.**
*Online book - second edition.*


van Hasselt, H., Guez, A., and Silver, D. (2015).
**Deep Reinforcement Learning with Double Q-learning.**
*ArXiv e-prints.*


Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2015).
**Dueling Network Architectures for Deep Reinforcement Learning.**
*ArXiv e-prints.*