Autonomous ‘*pong*’ player-agents

Machine Learning Term Project

Authors: Rodrigo Arias <rodarima@gmail.com>
Cedric Bhihe <cedric.bhihe@gmail.com>

Delivery: before 2018.06.22– 23:55
UPC – FIB / MIRI Program

Table of Contents

A. Introduction

This report describes the study of a closed system consisting of two autonomous and independent, temporally situated, learning agents, playing a game of *Pong*[i]. Each-agent-player must overcome its opponent by learning to play the game better and faster in order to score points. An agent is computationally autonomous in that it *learns* to interact with its environment by being rewarded, whenever its scores, and penalized whenever its opponent scores. The goal-directed machine learning (ML) methods of choice in our case are reinforced learning (RL) methods, which we set out to implement and benchmark.

Pong is a simple game and its rules are outlined in the framed inset at the end of this introductory section. We choose to focus not on the implementation of the game [1], although it is far far from being devoid of interest, but rather on that of the ML methods we propose to study. By endowing the two player-agents with different characteristics and learning method's parameters, we set an explicit objective for them: to maximize their own score. For that we make them aware of their environment in a manner detailed later. Our goal is to compare the relative performances of different ML methods.

Apart from the simplicity of the game, there are at least two ML-related main reasons to choose Pong to study the relative performance of different RL methods.

1) Pong has two players. That affords us the possibility to either pit one learning agent against another or to appraise an agent's learning curve when opposed to a human player or to a training wall. This permits the design of a parametric study of learning performance, as a function of learning methods' parameters. We can also allow two (differently configured) learning agents to compete directly in gradually more complex learning environments, i.e. environments with increasing numbers of actions and states.

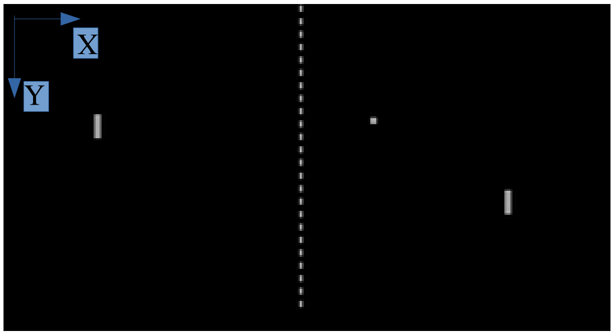
2) Given the nature of the problem, we can study several RL methods [2],[3], in particular:

- basic, off policy, model-free Q-Learning (QL), parametrized by:
  reward pattern,
  discount factor,
  learning rate or "step size",
  pseudo-greedy parameter, ε.
- basic, on-policy, model-free State-Action-Reward-State-Action (SARSA), parametrized by:
  reward pattern,
  discount factor,
  learning rate or "step size",
- Deep Q Neural-Networks (DQN), parametrized by:
  reward pattern,

  ….

<u>The simple game of 'pong'</u>

The game consists of a rectangular arena (in the XY plane), a ball, and two paddles to hit the ball back and forth across the arena. A player-agent (represented by a paddle) scores when the ball flies past the opposite player's paddle and hits the back-wall opposite the scoring player's side. When this occurs a new episode, made of a sequence of exchanges, starts.
Each player can only move vertically (i.e. along direction Y).
The ball can bounce off the paddles as well as the side walls running parallel to axis X.



At writing time, we have no guarantee, that we can include every above-mentioned RL method in our final results.

This report is organized as follows:

[i] The game of 'pong' is one of the earliest video games, released in 1972 by Atari. It is built with simple 2D graphics.

In section B-1 we briefly review the fundamentals of the 3 above-mentioned RL methods. Although RL is particularly indebted to Markov Decision Process (MDP) and Stochastic Approximation theories, we chose not touch upon those subjects, in favor of a much more practical and intuitive approach.

Apart from listing pseudo-codes, Section B-2 is devoted in some detail to the implementation of Q-Learning from a simple 2-action 3-state problem to an |A| -action, |S| -state one, where:

- |A| denotes cardinality of A, the set of all possible actions a, greater than or equal to 2 and
- |S| denotes cardinality of S, the set of all possible states s, is greater than or equal to 3.

In particular we give an account of how we experimented moving away from a greedy action-picking policy mechanism to an ε-greedy policy mechanism, based on the current reward matrix. We report intermediate results.

In section C, we outline results and experimental observations.

In section D, we analyze our results and draw conclusions based on them. We suggest sensible practical extensions to our work, in order to conduct further exploration in potentially interesting directions.

At the end a short reference section gathers the most salient papers and texts we used throughout this report.

B. Road-map to implementing Reinforced Learning methods

Algorithmic learning methods, where an player-agent (cum decision-maker, cum action-taker) combines environment sensing, explicit goal-directed action and interaction with its environment through action, qualify as Reinforcement Learning (RL).

We wish to compare computational approaches to learning from interaction between an agent and its environment. The circumstances of such goal-directed learning experiments are schematically represented by Fig. 1.

QL, SARSA and DQN are differently adapted to incomplete knowledge situations and never-seen-before states, where - as is our case - player-agents are memory-less and rely on a limited set of actions to maximize their utility. Each player's action is rewarded by a numerical score, as it eventually learns what the optimal action is in any given state.

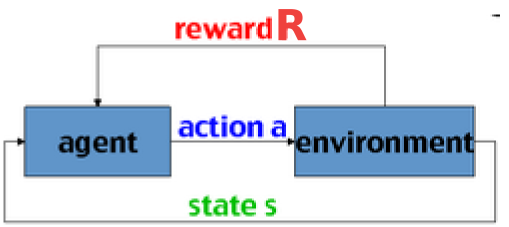


***Fig. 1**: In RL, an autonomous agent, with an explicit objective function optimization goal, perceives its environment (state "s"). Interacting with "s", it explores the rewards "r" its actions "a" may glean.*

B.1 – Background

RL is about how an agent learns to perform best with either incomplete (model-free) or complete knowledge (model-based) of the environment's dynamics. Even in cases where the agent has a complete knowledge of its environment, computational considerations such as the amount of available memory vs. the large number of states make the agent unable to fully capitalize on its knowledge at each computational time step. Model-based algorithms are impractical when the state variable space become realistically large, as complexity grows as $|S|^2 |A|$. In such cases, judiciously chosen approximations (when an agent picks an action and the ensuing reward is appraised) become central in developing efficient algorithms which converge toward an optimal policy.

For practical algorithmic reasons (i.e. in order not to store all combinations of states and actions at all times), we consider only model-free systems, where learning agents cannot envision how their action will change their current state, even as a transition-probability between two states may be successfully learned. We briefly highlight three such model-free methods hereafter.

B-1.1 Q-Learning

Off-policy:

The goal of the Q-Learning agent is to learn how to guide its own actions in an initially unknown environment. In Q-learning, Q stand for "quality" in that a system learns to maximize the quality of its actions by learning an optimal behavior (or action-selection policy) as a function of its state. It is an ***off-policy*** method in that it does not rely on a known policy, but rather creates and shapes its own (eventually optimal) policy at it trains and learns.

State value – quality of state-action – reward (Bellman equation):

The quality of a state-action, also called the current reward, is a relation: $Q: \boldsymbol{S}^{(t)}\ x\ \boldsymbol{A} \rightarrow \mathbb{R}$. The Bellman[i] equation (Eq. 1) defines an instantaneous reward R_t, plus an incentive in the form of a discounted future reward calculated as the maximum of all potential future rewards attainable from state s_{t+1}. It translates the fact that as the agent's environment transitions from *st* to *st+1* both in *S*, via an action *at in A*, the learning agent evaluates its current best action, a_t, as a function of a current reward, R_t, plus a discounted future reward:

$$Q_t \;=\; Q(s_t, a_t) \;=\; R_t + \gamma(\max_a(Q(s_{t+1}, a))) \qquad \text{(Eq.1)}$$

Exploration versus exploitation:

During an episodic game such as pong, a player-agent learns by relying on both:

- exploration of uncharted regions of its environment variable space (never seen before states), and
- exploitation of already seen states for which some measure of learning experience has occured,

such that:

$$Q_t^{new} \leftarrow (1-\alpha) Q_t^{old} + \alpha\left(R_t + \gamma \max_a Q^{old}(s_{t+1}, a)\right) \qquad \text{(Eq. 2)}$$

where, as before, $Q_t = Q(s_t, a_t)$ is the quality of the state-action, R_t is the reward expected along the way, indexed with time step *t* for the current state[ii], $0 \leq \alpha \leq 1$ is the learning rate, and $0 \leq \gamma \leq 1$ is the discount factor, trading off the reative importance of immediate (current) and future rewards . Whereas our learning agent only one goal in mind: to learn the best it can, we are computationally interested in that happening as fast as possible. The quality matrix converges when:

$$\lim_{t \rightarrow +\infty} Q_t \;=\; Q^{stationary} \qquad \text{(Eq. 3)}$$

Eq. 2 essentially translates the Bellman equation into an iterated value update for the state-action quality value. It is at the core of the QL algorithm. At every time step, *t*, the update of the state-action reward mapping, or quality matrix Q_t, results from the weighted average between the old policy state-value, multiplied by $1\text{-}\alpha$, and the learnt policy state-value, multiplied by α.

▪ For $\alpha = 0$, the agent will not update the *Q* function (commonly referred to as the quality matrix) mapping state-action to reward, and therefore will not learn a new policy. It remains stuck at: $Q_t^{new} \;=\; Q_t^{old}$

▪ For $\alpha = 1$ the agent pays no attention to its previously learned quality matrix Q_t^{old} and only favors its instantaneous reward R_t, along with another term, mediated by the discount factor, γ. The latter term is the maximum discounted potential future reward achievable at future state s_{t+1}, reached from s_t by action a_t. The discount factor, γ, effectively parametrizes the importance of future rewards.

▪ $\gamma = 0$ makes the agent "short-sighted", in that it becomes solely interested in current rewards immediately following its actions.

▪ $\gamma = 1$ or slightly smaller than 1 on the other hand was shown to produce instability [4],[5] as the agent place the highest possible weight on discounted future rewards.

Greedy versus pseudo-greedy:

Calling $Q_t = Q(s_t, a_t)$ the reward to the agent in state "s" for adopting action "a" at time "t": $A_t := \underset{a}{argmax}(Q(s_t, a))$ will select the greedy action which maximizes its immediate reward based on the current (initially arbitrarily chosen) action-

[i] *See: Richard Bellman, "A Problem in the Sequential Design of Experiments," Sankhya, 16 (1956), 221-229.*

[ii] *Current reward, R_t, is sometimes referenced with time step subscript t+1, without any change in the algorithmic quantities involved.*

to-reward mapping function Q. In case of a tie in state, s_t, between two distinct actions, a and b, such that $Q(s_t,a)=Q(s_t,b)$, we may break the tie in some pre-ordained way, for instance randomly. To favor convergence our algorithmic system may also evolve from a greedy action model to an ε-greedy policy mechanism by introducing uniformly random action selection in ε % of cases.

Simplified computational strategies:

Caveat 1: Assume for a moment that the distribution of actions' reward values becomes constant *in time* (after a sound policy has been learned) and, so, that our RL system has reached a stationary regime. This means that the true reward values or long-term reward distribution (in the limit of infinite time-steps) do not change. This in turn might lead us to expect that a greedy approach ($\varepsilon = 0$) always yields the best possible choice of action. However non-stationary regimes are one of the principal problems encountered in RL, often (but not exclusively) in the form of periodic, pseudo-periodic or apparently chaotic policy changes as learning proceeds and the agent's policy is updated step-wise. Thus even for apparently deterministic systems (problems with reward probability distribution which do not change over time), it is often preferable to introduce a degree of randomness when choosing an action, to balance exploration and exploitation.

Caveat 2: Choosing a constant value of ε can also be far from optimal. Intuitively comparing larger values of ε to smaller ones, we see that the former will lead to faster exploration, but sub-optimal exploitation of decisions with highest rewards. Thus at any time t, it is may be advantageous to use greater values of ε for larger rewards' distribution variance (examining the set of rewards associated to all possible actions at a given time step).

Arranging for ε to be a function of possible reward distribution's variance in addition to generally decreasing as a function of computational time-step may bring better results.

B-1.2 State-Action-Reward-State-Action (SARSA)

The question remains the same and is: how do we estimate $Q_t = Q(s_t,a_t)$?

SARSA stands very close to QL. However being an on-policy method, SARSA learns the new updated Q-values based on the action performed by the current policy, π, instead of a greedy or pseudo-greedy policy. Hence Eq. 2 becomes:

$$Q_t^{new} \leftarrow (1-\alpha)Q_t^{old} + \alpha\left(R_t + \gamma Q^{old}(s_{t+1}, a_{t+1})\right) \qquad \text{(Eq. 4)}$$

The name SARSA is derived from the formulation of Eq. 4 rewritten as:

$$Q_t^{new} \leftarrow Q^{old}(\boldsymbol{s_t}, \boldsymbol{a_t}) + \alpha\left(\boldsymbol{R_t} + \gamma Q^{old}(\boldsymbol{s_{t+1}}, \boldsymbol{a_{t+1}}) - Q^{old}(S_t, A_t)\right)$$

where the update of the Q-matrix depends on the knowledge of the quintuple (s_t,a_t,r_t,s_{t+1},a_{t+1}).

B-1.3 Deep Q-Neural Network (DQN)

QL's tabular approach to state values and optimal policy determination suffers from serious practical limitations. When the variable-space size increases to reach realistically large values, the large number of possible states prohibits a complete (even model-based) recalculation of state transition probabilities at every computational step.

QL is also limited in that it exhibits no predictive power and thus cannot be generalized to variable space domains containing never-seen-before-states or features difficult to discern or to parameterize. To extend the QL framework and endow it with predictive capability, one may think of state-values as a large set of real numbers, to which we would like to fit a state-value function by regression:

$$Q(s_t,a_t,w^*) \approx Q^*_{t \rightarrow +\infty} \qquad \text{(Eq. 5)}$$

where w^* is the optimal set of fit-parameters (weights) and $Q^*_{t \rightarrow +\infty} = Q^{stationary}$ is the optimal state/action-value function.

Computational deep-learning advances in the last decade and more recently[6][7][8] at Google DeepMind[i] forged new methods to approximate state-values in real-time applications. In the following we briefly describe how Deep Neural Networks supplements QL and gave rise to DQN.

In DQN the neural network may consists of 3 hidden layers with a varying number of intermediate nodes. It receives the current state, s_t, (or more prosaically screen pixel information and score in a episodic game of pong) as input and returns the set of state values for each available actions $Q(s_t,a)$ as output.

At an elementary level, as pictured in Figure 2, input can include actions in addition to states (left) which gives rise to an output in the form of one Q-matrix. This computational schema requires carrying out as many forward passes as there are possible actions, at a cost proportional to $|A_t|$, the cardinal of the set of possible actions at step *t*. Observing the right hand side schema, the need for multiple forward passes in the Neural Network is obviated by only considering states as input. The output consists of various Q-matrices, one for every possible actions taken at the given time step.

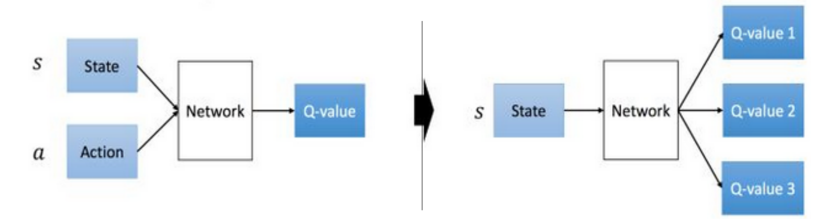


Fig. 2: *DQN can either accept all states and all possible actions as input (left), or all states as input and differentiate output as a function of each possible action (right).*

Following accepted naming conventions, $Q(s_t,a_t,\boldsymbol{w_t})$ is a "*neural network state-value function approximator*" with weights $\boldsymbol{w_t}$, commonly referred to as a "*Q-network*". As in Multilayer Perceptrons, a Q-network is trained by minimizing $L_{t+1}(\boldsymbol{w_t})$, a loss function which changes at each iteration:

$$L_{t+1}(\theta) = E_{s,a\sim\rho(\cdot)}[\ (y_{t+1} - Q(s_t,a_t,\boldsymbol{w_t}))^2\ ] \qquad \text{(Eq. 6)}$$

$E_{s,a}[\,]$ denotes the expected value, and following Mnih et al[6] the NN's state-value target variable is:

$$y_{t+1}=E_{s_{t+1}\in\{s\}_{t+1}}[R_t+\gamma \max_a Q(s_{t+1},a,\boldsymbol{w_t}\mid s_t,a_t)] \qquad \text{(Eq. 7)}$$

where $\rho(.)=\rho(s,a)$ is the probability distribution over states, *s*, and actions, *a*, called the behavior distribution. Weights $\boldsymbol{w_t}$ are fixed when optimizing $L_t(\boldsymbol{w_t})$. In contrast with supervised learning, where targets are fixed before learning begins, here targets depend on NN weights. The loss function is optimized with respect to weights, $\boldsymbol{w_t}$, and $Q(s_t,a_t,\boldsymbol{w_t})$ is updated toward the state-value target y_{t+1} by stochastic gradient descent.

$$\boldsymbol{w}_{t+1} \ = \ \boldsymbol{w}_t+\alpha(y_{t+1}-Q(s_t,a_t,\boldsymbol{w}_t))\nabla_{\boldsymbol{w}_t}(s_t,a_t,\boldsymbol{w}_t) \qquad \text{(Eq. 8)}$$

By updating weights at every step, and by replacing expectations by single samples from the behavior distribution ρ and the set of possible $\{s,a\}_t$, we obtain the familiar model-free, off-policy QL framework at the end of Section B-1.1.

B-2. – Implementation

Pong player-agents, with incomplete knowledge about their environment, and potentially very large number of states, are well served by a model-free, off-policy RL settings such as QL and evolutions from QL.

We placed no particular emphasis on striking the best possible balance between exploration and exploitation. Instead we experimented with α values, allowing learning agents to improve over successive steps, so the iterated computation of the

[i] *DeepMind, based in London, UK, was bought by Google, Inc. (today's Alphabet, Inc.) in 2014.*

matrix Q would exhibit convergence from start to terminal states for every game episode. Convergence of Eq. 2, sometimes called *exponential recency-weighted average* with parameter α, is governed by the well known recursive update rule's double condition for convergence:

$$\lim_{\tau \to +\infty} \sum_{t=0}^{\tau} \alpha_t = +\infty \quad \text{and} \quad \lim_{\tau \to +\infty} \sum_{t=0}^{\tau} \alpha_t^2 \ll \tau \qquad \text{(Eq. 9)}$$

where α is made to depend on the time step *t*. Opting for a constant step parameter $\alpha_t = \alpha_o$ does not satisfy the second condition in Eq. 9. Quoting from Sutton's and Barto's [2] book on RL (page 26), *"the first condition is required to guarantee that steps are large enough to eventually overcome any initial conditions or random fluctuations. The second condition guarantees that eventually the steps become small enough to assure convergence."*

In practice, in order to speed up convergence in the sense of Eq. 3, α was not kept constant. A typical choice is:

$$\alpha_t = (1 + \mathit{nbr\ of\ prior} \{s_t, a_t\} \ \mathit{visits})^{-1} \qquad \text{(Eq. 10)}$$

We also experimented with moving away from a greedy action-picking policy mechanism to an ε-greedy policy mechanism, based on the current reward matrix, Q_t, where ε is the probability of taking a random (exploratory) action a_t from a given state at step *t*, s_t. Good results are reported for non zero values of ε, while ε is made to decrease over time. Optimal ε values are a function of reward distribution's variance, calculated for all possible actions at any step *t*. We suggest *ex nihilo* a simple formulation for ε-greedy, of the form:

$$\epsilon_t = min\left(c_0, \frac{c_1}{t+1}\left(\frac{\sigma_t^{(reward)}}{\left|\mu_t^{(reward)}\right|+1}\right)^{\frac{1}{c_2}}\right) \qquad \text{(Eq. 11)}$$

where $t \geq 0$ denotes the algorithmic time step, and our 3 adjustment parameters are:

$0 < c_0 \leq 1$

$0 < c_1$, *with* $c_0 = c_1$ in a typical case.

$1 \leq c_2$

For small values of *t*, and up to a threshold largely controlled by c_1, ε_t will likely be equal to c_0. Past that time step's threshold, ε_t shows a decreasing trend toward 0 as a function of *t*, with occasional "bumps" and "troughs" governed by the step dependent ratio $\sigma/(\mu+1)$, and capped by c_0.

B-2.1 Pseudo-code for QL

General code structure

First the game simulation was implemented, so we could visually experiment with `pygame`, and experience first-hand simple collision mechanisms between objects considered as boxes; e.g. a ball bouncing off a paddle or a side wall.

Code blocks:

- **param.py**:

* control all RL methods' parameter settings

- **pong.py**:

* defines game related classes, objects, and display logic.
* defines basic update(), restart() and paddle-ball collision event related methods.

- **control.py**:

* defines controller class and object classes, in particular QL, SARSA, QLd, QDN
* defines Q matrix (class PC, method __init__)
* defines draw, update, control, reward and other methods.
* handles keyboard interactions.
* handles communications with the board

- **main.py**:

attribution of a side to player-agents' controllers.

Note: must add cli parameters method instead of modifying code by hand), e.g.: $ python main.py -l keyboard -r ql

main.py:
By definition a terminal state is when one agents scores. The game performs the following actions in sequence, in a infinite loop, until a terminal state is reached:

- *read events from keyboard and store in board*
- *update the ball*
- *update the left controller*
- *update the right controller*
- *update the left paddle*
- *update the right paddle*
- *re-start a play (episode) if needed*

Player-agents' controllers must be updated before paddles are updated If not an agent's controller could gain an unfair advantage from knowing the movements of its opponent.

The ball implements some logic to determine when a player scores, and handles the ball trajectory (collision with paddles and board boundaries).

control.py:
Each controller implements the method `update()`, and has access to object `board`. `board` provides information on the board environment, as well as which actions that can be performed.

In the first version of the code, the logic of each controller was programmed as a function of the placement of its corresponding paddle, on either the left or the right side of the board. The consequence was a chirality problem, as a left-hand-side-trained controller (a "lefty") could become confused and perform erroneously when placed on the right hand side of the pong board (and vice versa). For that reason we modified the program's design and let the board take control of symmetry issues, in such a way that all controllers see the game as if they were placed on an undifferentiated side.

Basic QL-agent controller:
Translation of play status into state followed a simple approach. The state's position information consisted in determining whether the traveling ball was placed above or below the paddle's center:

sa: state above
sb: state below

To this we added two actions:

a=0: move up
a=1: move down

Finally the agents' rewards were:

agent scores: +1
agent's opponent scores: -1
agent's paddle catches ball: +0.1 (optional)

Important note: When designing our algorithmic QL methods, the main loop calls `update()` only once per frame. So in order to access states at any step, states need to be stored in the controller, until the next update takes place and the next states are ascertained. Updates for the positions of the ball and of the two paddles are always performed outside the controller. Once those updates are complete, objects have the opportunity to interact with the new state stored in the controller.

Advanced QL-agent controller:
State information such as ball's speed and position, paddles' positions were further discretized.

As the number of states grew, so did learning time. The upshot was that trained agents now became more “discerning”. For instance, after training, agents were capable of directing the ball toward specific areas of the board, which their opponent could not reach or could reach only with difficulty (a notion that player-agents were incapable of interpreting for lack of a related quantitative feature).

Symmetrization and discretization

- Rodrigo, please add cursory highlight of the advantage of symmetrizing states to ease learning from a computationally standpoint.

- Rodrigo, highlight the realistic setting where time is discretized, each time step corresponding to a frame or snapshot of the game’s state. Show how Q is recalculated completely or partially, depending on the implementation’s particulars.

Pseudocode

In the pseudocode to the right, π denotes the policy being learnt.

QL pseudocode

```
1) Set learning rate function, $a(t)$ and discounting factor, $\gamma$.
2) Initialize arbitrary table $Q_{t=0}$, corresponding: $Q{:}S^{(t)}\ x\ \mathbf{A} \rightarrow \mathbb{R}$
3) While $Q_t$ has not converged, do:
   a) Start game’s “episode” or “play”in arbitrary state $s_{t=0} \in S$
   b) While $s_t$ is not a terminal state (no scoring point recorded)
      ■ evaluate $a_t$ and subsequently receive current (immediate)
        reward, $R_t$, from current $Q$-matrix
      ■ enter new state is $s_{t+1}$
      ■ maximize $Q_t(s_{t+1},a)$ over possible actions, $a$, based on $\pi$
      ■ Evaluate $Q_t^{new}$ according to Eq. 2 with off-policy
        greedy or ε-greedy action selection method
      ■ $s_t \leftarrow s_{t+1}$
   c) Return $Q_t \leftarrow Q_{t+1}$
4) Return $Q_t$ if $Q$-matrix based $\pi$ convergence criterion is met.
```

B-2.2 Pseudo-code for SARSA

The QL procedural form above does not change, except for the iterated value update for the state action quality value of Eq. 2 being replaced by Eq. 4.

B-2.3 Pseudo-code for DQN

In DQN, the neural network is a convolutional neural network, a variation of MLP designed to require minimal pre-processing of input and to minimize computational cost by weight sharing. Overall it is a non linear function from an |S|-dimensional state input space to an |A|-dimensional action space, built on the principle of hierarchical layers of tiles convolutional filters.

To significantly improve learning, and speed up computations, Mnih et al (2013) [6] approach the problem of correlation of the input sequence with the target values and within the sequence the sequence of observations with 2 ingredients:

- every so many steps, weights are copied from the current deep neural network configuration to yield $\boldsymbol{w}^{-}$.
Those weights are used to update action-values of the “target-network” in Eq. 7 at given iteration intervals, and are kept constant otherwise. This reduces correlation of the sequence of evaluation network with with the Q-network (target-values).

- the technique of “e*xperience replay*”, also shown to improve DQN results, consists in periodically injecting scrambled sequences of previously stored state transitions to update the target -network, yet again reducing correlation but this time within the input sequence distribution.

C – Results and discussion

At the onset of this project, our goal was to benchmark different RL methods. Performance comparison between learning autonomous QL agents was envisioned in the form of round-robin tournaments between differently programmed player-agents. The winning player-agent was to proceed to the next competition level to face a newly configured player-agent. We introduced complexity incrementally in the state-action conditions of successive plays and 3 generations of player-agents with growing state complexity. By adding more states and complexity to the agents' environments, learning by player-agents was made gradually more difficult, but conferred the trained player-agent advantages over other agents trained in a less sophisticated way.

Throughout this work, at the onset of each episode game, the ball was placed at the center of the field and its movement initiated with a random trajectory angle. The number of possible agent's actions always remained two, consistent with moving the paddle upward or downward. Table 1 exhibits an example of variable space discretization leading to more than 13 millions states, which we considered the practical upper limit for a manageable number of states using the QL methods . We decided not to pursue the tabular approach of QL, as training rapidly proved far too time-consuming for each parameter-setting. This put us on the path to DQN implementation. The new approach consisted in approximating state-values in the learnt policy value term of Eq. 2.

Object	Variable	Discretization
1st paddle	Vertical position	11
2nd paddle	Vertical position	11
Ball	Vertical position	11
Ball	Horizontal position	11
Ball	Speed	5
Paddle	Bounce surface zone	5
Ball	Angle of trajectory	36

Table 1*: Maximum number of discretized states reached with the QL method (13,176,900), before switching to DQN implementation.*

D – Concluding remarks

D-1. – Conclusions

RL is about an agent learning how to behave through interaction with its environment. Which action to preferentially take to maximize some utility function, when unchanging after enough learning steps, constitute an optimal state-action—reward probabilistic mapping, or policy, involving all encountered states, and actions. As a decision-making system, RL is reminiscent if not similar to how rewards in the animal brain are mediated by the neuro-transmitter dopamine.

To date RL of all ML methods most closely approximates the way animals learn, complete with its on- vs off-policy, exploration vs. exploitation, model-based vs. model-free learning representations. We also shed some light on what could be dubbed a learning agent's "drive" to explore, based on rudimentary mechanisms related to:

- measures of reward variance, a simplistic representation of novelty, and
- how close to the goal of "having learned" the system is.

As we further increased the complexity of our environments, training time grows dramatically, such that the tabular approach of QL to compute optimal policy became impractical. DQN was the method of choice to overcome that limitation. It constitutes an alternative method to encode state-action–reward mapping (i.e. "board information" in our algorithmic implementation) by using interpolation in order to approximate value functions for reward and (implicitly) for state transition probabilities. Working with state-value function approximations is from the point of view of computational cost equivalent to a reduction in complexity.

Deep learning algorithms generally work assuming data samples' independence a fixed underlying data distribution. RL On the other hand is often confronted with:

- sequences of correlated states,

- changing level of correlations
- changing data distributions

as the learning agent interacts with its environment and influences data input.

D-2. – Limitation and extension for this work

Much can be done to complete our study.

▪ Further exploring episodes' starts with random state-action pairs and optimistic first moves, something we have not even hinted at in this report, but is well documented in the survey on RL by Kaelbling et al (1996), which predates the advent of DQN [8].

▪ We know that dealing with realistically large numbers of states makes computational loads impractical in the face of memory limitations: enters DQN. However we identified at least one interesting alternative to DQN worth mentioning here, based on *Monte-Carlo*[i] (MC) control methods.

The same general approximated value-function concept at the root of DQN holds for the Generalized Policy Iteration (GPI) technique. The GPI's general idea is to let *policy evaluation* and *policy improvement* processes interact programmatically in an alternated way. Such processes may be truncated and arranged asynchronously, from the point of view of "which states are included when" in successive iteration's sweeps. In its classical form, GPI is concerned with:

(i) policy evaluation: i.e. making the state value function consistent with the current policy,
(ii) policy improvement: i.e. making the policy greedy (or ε-greedy) with respect to the current value function.

Each process completes before the other begins. We can also say that each simultaneously competes against and cooperates with the other. This apparent *non-sequitur* is explained in that each process creates a moving target for the other, but together they make both value function and policy approach their optimal expression.

MC control methods follow that briefly outlined schema. However in this case, rather than computing each state's value (Q-matrix) based on a system's model as in classical GPI for *policy evaluation*, each state's values (i.e. each expected return) is the result of returns averaged over many MC trials, starting from that state.

- this does not require a model of the environment transitions' dynamics.
- in cases where one suspects that the Markov memoryless property may be violated, MC control methods for policy learning fare better, because they do not (as in eq. 2) rely on updating states' values based on values' estimates of subsequent states ("*bootstraping"*).
- it is simple to simulate sample episodes and to run MC trials to evaluate expected states' values at episode's end .
- in very large problems, it is possible to apply the MC control method on a subset of states, without detracting from policy evaluation results accuracy.

All of the above would make MC Control methods of policy evaluation attractive in the context of an extended benchmark study on off-policy learning[ii].

▪ Generally speaking we only considered RL settings where systems are stationary, i.e. where agents are not confronted with best actions changing over time for given states, beyond the starting point. The rationale for the study of such non-stationary settings would be to program agents, so they are capable of recognizing new best actions in the face of already seen system's states. Action selection in that case cannot rely on a simplistic greedy mechanism. The *ε*-greedy mechanism is also not optimal (too slow in particular) in that its random action selection is by definition indiscriminate. Instead it might be more interesting for the agent to have a means to favor actions seldom taken in particular states before, in order to explore rapidly the changing state-"best-action" space. However interesting, this approach would also have been more complicated than our earlier proposal of Eq. 11 for *ε*-greedy and so remained out of the scope of this work.

[i] *The term "Monte Carlo" dates back to the 1940s, when physicists at Los Alamos studied games of chance to understand complex physical phenomena relating to the Manhattan project*.

[ii] *Among the best and most recent exponents in that area are – yet again – Sutton and Barto [2], in Chap. 5 of their 2018 book on RL*

References

[1] T. Appleton, “Trevor Appleton: Writing Pong using Python and Pygame,” *Trevor Appleton*, Apr-2014. .

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.

[3] C. Bishop, *Pattern Recognition and Machine Learning*. New York: Springer-Verlag, 2006.

[4] L. Baird, “Residual Algorithms: Reinforcement Learning with Function Approximation,” in *Proc. of 12th Int’l Conf. on Machine Learning*, 1995, pp. 30–37.

[5] V. François-Lavet, R. Fonteneau, and D. Ernst, “How to Discount Deep Reinforcement Learning: Towards New Dynamic Strategies,” *arXiv:1512.02011 [cs]*, Dec. 2015.

[6] V. Mnih *et al.*, “Playing Atari with Deep Reinforcement Learning,” *arXiv*, no. arXiv:1312.5602 [cs.LG], 2013.

[7] Google DeepMind, *https://www.github.com/deepmind/dqn: Lua/Torch implementation of DQN (Nature, 2015)*. DeepMind, 2018.

[8] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement Learning: A Survey,” *1*, vol. 4, pp. 237–285, May 1996.