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 two ML-related main reasons to choose Pong to study the relative performance of different RL methods.

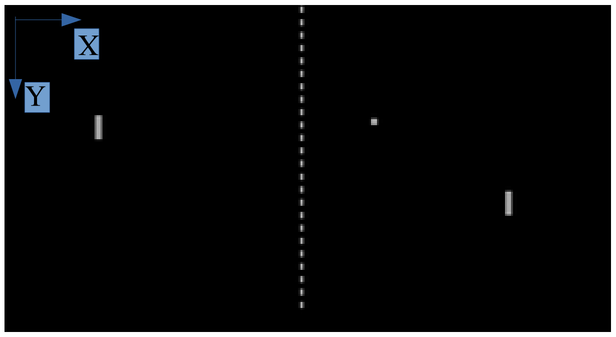
1) Pong has two players. It affords us the possibility to either pit one learning agent against another or to appraise an agent-player's learning curve when opposed to a human player or to a training wall. This permits the design of a parametric study of learning performance, as a function of learning methods' parameters. We can also allow two (differently configured) learning agents to compete 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,
    - discount factor,
    - learning rate or "step size",
    - pseudo-greedy parameter ε and
- basic, on-policy, model-free State-Action-Reward-State-Action (SARSA), parametrized by:
    - reward
    - ….
    - ….
- Deep Q Neural-Networks (DQN), parametrized by:
    - reward
    - ….
    - ….

<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.

The report is organized as follows:

In section B-1 we briefly review the fundamentals of the 3 above-mentioned RL methods. Although RL is particularly

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

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 further suggest sensible practical extensions to our work, in order to explore new methods in potentially interesting directions.

At the end a "References" section is followed by 4 Appendices. The first one, Appendix A, presents a pseudo code for our general implementation. Appendices B through D list our 3 code sections in their entirety along with numbered code lines for easier referencing throughout this report.

B. Road-map to implementing Reinforced Learning methods

We wish to explore a computational approach 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 adapt particularly well and in different ways 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 learns what the optimal action is in any given state. 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).

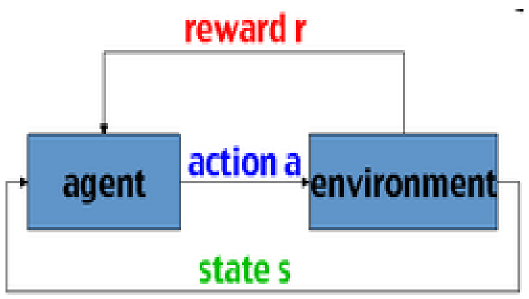


Fig. 1*: In RL, an autonomous agent, with an explicit objective function optimization goal, perceives its environment (state "s"). Interacting with "s", it explores the rewards "r" its actions "a" may glean.*

B.1 – Background highlights

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 at it trains and learns.

Reward – quality of state-action (Bellman equation):
The quality of a state-action, also called the current reward, is a relation: $Q{:}S^{(t)} \times \boldsymbol{A} \rightarrow \mathbb{R}$. The Bellman[i] equation (Eq. 1) defines a 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:
Moreover a player-agent learns during an episodic game, 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 been accumulated,

such that:

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

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, $0 \leq \gamma \leq 1$ is the discount factor. Our learning agent has only one goal in mind to learn the best it can. Our algorithmic interest is to make that happen as fast as possible. It translates in our quality matrix converging, so that:

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

The formulation of 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 value times α, and the learned policy value times 1-α.

- $\alpha = 0$, 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: $Q_t^{new} = Q_t^{old}$
- $\alpha = 1$ 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 from some current (initially unknown) action-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

i *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 t+1, without any change in the algorithmic quantities involved.*

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 is in fact stationary, meaning 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). Indexing the instantaneous value of ε on the possible reward distribution variance in addition to a general decreasing trend in ε values as a function of time may therefore bring the best results.

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

B-1.3 Deep Q-Neural Network (DQN)

B-2. – Implementation

During implementation we placed no particular emphasis on striking the best possible balance between exploration and exploitation for any particular method. Considering a pong player-agent to have incomplete knowledge about its environment was a way for us to approximate an model-free RL setting. Because In model-free settings realistically large number of possible states prohibits a complete model-based recalculation of state transition probabilities at every computational step. In that context, we were content with α values, which allowed learning agents to improve over successive time steps, so the iterated computation of the matrix Q would exhibit convergence from start to terminal states for every game play (or episode). In practice, in order to speed up convergence in the sense of Eq. 3, α was not kept constant. 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:

$$\lim_{\tau \to +\infty} \sum_{t=0}^{\tau} \alpha[a_t] = +\infty \quad \text{and} \quad \lim_{\tau \to +\infty} \sum_{t=0}^{\tau} \alpha[a_t]^2 < \frac{1}{\tau} \qquad \text{(Eq. 4)}$$

where α is made to depend on the action, a_t, taken at time step t. Opting for a constant step parameter $\alpha[a_t] = \alpha_o$ does not satisfy the second condition in Eq. 4. Quoting Sutton et al [2] (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."*

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 obtained for a non zero values of ε, while ε is made to decrease over time. Optimal ε values are an increasing function of the variance of rewards, calculated from their distribution for all possible actions at any step t. We suggest *ex nihilo* a simple formulation for ε-greedy, of the form:

$$\epsilon_t = c_1 \frac{1}{t+1} \left(\frac{\sigma_t^{(reward)}}{\left|\mu_t^{(reward)}\right| + 1} \right)^{\frac{1}{c_2}} \qquad \text{(Eq. 5)}$$

where our 2 adjustment parameters are $c_1 \geq 0$, $c_2 \geq 1$ and the algorithmic time step, $t \geq 0$.

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:

- **pong.py**: game related elements, without logic <<<< cryptic, add something to that
- **control.py**: 2 controllers *cr()* and *cl()* include the update and control methods; handles communication 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*

Both controllers must be updated before paddles are updated. <<< Probably needs rewriting If not one 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 our first code version, 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 (lhs) trained controller (a "lefty") could 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 (and in our special case, left) side.

QL-agent controller (PCv1):
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:

a0: move up
a1: 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 (PCv2):
State information such as ball's speed and position, paddles' positions were further discretized. *With them were generated an integer number by computing the cartesian product.* <<<< cryptic: correct, explain or suppress

As the number of states grows, so does learning time. The upshot is that trained agents now become more "discerning". They become capable of directing the ball they hit toward specific areas of the board, which their opponent cannot reach or can reach only with difficulty.

Pseudocode

```
1) Set learning rate function, $\alpha(t)$ and discount factor, $\gamma$.
2) Initialize table $Q_{t=0}$, corresponding: $Q{:}S^{(t)} \times A \rightarrow \mathbb{R}$
3) While $Q_t$ has not converged, do:
   a) Start game's "episode" or "play" in state $s_{t=0} \in S$
   b) While $s_t$ is not a terminal state (no scoring point recorded)
      ▪ evaluate $a_t$ and subsequently $R_t$ from current $Q$-matrix
      ▪ new state is $s_{t+1}$
      ▪ maximize $Q_t(s_{t+1},a)$ over $a$, based on current policy, $\pi$
      ▪ Evaluate $Q_t^{new}$ according to Eq. 2 (off-policy 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

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

$$Q_t^{new} \leftarrow (1-\alpha) Q_t^{old} + \alpha \left(R_t + \gamma Q^{old}(s_{t+1}, a_{t+1}) \right) \qquad \text{(Eq. 6)}$$

and the QL procedural form above does not change, except for the iterated value update for the state action quality value of Eq. 2 is replaced by Eq. 6.

B-2.3 Pseudo-code for DQN

Discretization

- Highlight the realistic setting where time is discretized, each time step corresponding to a frame or snapshot of the game's state, where Q is recalculated completely or partially, depending o the implementation's particulars.

- Highlight the advantage of symmetrizing states to accelerate learning.

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.

Unfortunately we could not stick to the tabular approach of QL (and SARSA), as this strategy rapidly proved far too time consuming due to increasing training times for each method, and (within each method) for each parameter-setting. So the idea of round-robin bench-marking was abandoned in favor of exploring methods in terms of complexity until a given limit was reached in terms of training wall-time: typically 90min. Beyond that, for approximately 6000+ states, policy learning (to convergence) took so long (several hours), that we decided to switch from Q-Learning to DQN. The new approach was to approximate value functions for states in the the learnt policy value term of Eq. 2.

…………………

D – Concluding remarks

D-1. – Conclusions

Reinforcement Learning (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.

From a rapid literature survey, RL of all ML methods most closely approximates the way humans learn, complete with its on- vs off-policy, exploration vs. exploitation, model-based vs. model-free learning representations.

We also schematically saw how to approximate the 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, so much training time became necessary, that the tabular approach of QL/SARSA to compute optimal policy became impractical. To overcome that limitation, an alternative method to encode board information was DQN. DQN could reduce complexity greatly by using interpolation in order to approximate value functions for (state transition probabilities, and) reward.

D-2. – Limitation and extension for this work

To approximate optimal policies in high complexity RL settings, we resorted to DQN as a way to approximate value functions and thus optimal policies.

The same general approximated value-function concept holds for the Generalized Policy Iteration (GPI) technique ~~if a model is available~~. The GPI's general idea is to let policy evaluation and policy improvement processes interact 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 making the value function consistent with the current policy (policy evaluation) on one hand, and with making the policy greedy with respect to the current value function (policy improvement) on the other hand. Each process completes before the other begins. Each also competes against and cooperates with the other at the same time. The reason for this apparent *non-sequitur* is each process creates a moving target for the other, but together they make both value function and policy approach their optimal expression.

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. 5 for ε-greedy and so remained out of the scope of this work.

References

[1] T. Appleton, “Trevor Appleton: Writing Pong using Python and Pygame,” blog post by *Trevor Appleton*, https://trevorappleton.blogspot.com/2014/04/writing-pong-using-python-and-pygame.html, 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.