Automated optimization of a course of actions in a stochastic environment

Thomas Vicente

Abstract

I propose a principle allowing an artificial learning entity to find a right course of action possessing (like we all do) only an imperfect set of information on the current state of the world. I try to desmontrate the practicality and the generality of such a principle.

1 Introduction

A complete exercise would be to build an autonomous entity that is able to take long term oriented decisions based on information coming from the phisical reality.

This entity needs two abilities: learning from its own actions and recognising the current state's patterns. The first ability is typically made possible by a Reinforcement Learning (RL) mechanism. The second ability is done by regressors or classifiers powerful enough to adapt to the entity's sensorial features. In particular, if the entity is equipped by a visual sensor, neural networks might be a good idea.

1.1 Learning from its own actions

RL emerges easily from other Machine Learning techniques as the training is done by acting. It uses training information that *evaluates* the actions taken rather than *instructs* by giving correct actions are faced repeatedly with a choice among n different options, or actions. Such setting has its roots in the *n-armed bandit problem* where after each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the selected action. The objective is to maximize the expected total reward over all the episodes. The learner evolves in a particular state and records what actions lead to a maximal gain. In most cases the goal is to maximise a long term gain by taing a correct course of actions. These policies can be more or less explicit. I our case, the proposed algorithm would be qualified as *off-policy*.

An RL learner is thus particularly dependent on a state signal that succeeds in retaining all relevant information, a *Markov* state. For example, a checkers position (the current configuration of all the pieces on the board) would serve as a Markov state because it summarizes everything important about the complete sequence of positions that led to it. Much of the information about the sequence is lost, but all that really matters for the future of the game is retained.

A reinforcement learning task that satisfies the Markov property is called a Markov Decision Process (MDP). If the state and action spaces are finite, it is possible the MDP framework notably formalizes the probability that a state occurs given a previous state an action, and the probability of reward after the new state occured.

1.2 Pattern recognition

Equipping a reinforcement learner with an approximator function has two benefits: the ability to provide an evaluation of an infinite set of states and recognise the a finite set of states having an infinite set of possible representations. In the example below, as humans we are able to recognise the same underlying situation and take action: if I learner how to make spaghettis at home, I am able to reproduce it at my friend's place. In a similar manner, we would like our learner to overcome its imperfect sensors to learn how to behave in the long term to maximize its gain. This is made possible notably by regressor and classifier functions that take particularly well advantage of naturally generated data such as images and sounds.

A modified version of the classical tic-tac-toerobot will provide a good example for the principle we seek to study.

TWO IMAGES, caption: after enough games, the learner should be able to play the same way after vizualising either of these images

2 Litterature review

The proposed algorithm allows to optimize a course of actions in finite and infinite spaces. But it has especially been thought to act based on imperfectly sensed states as described above.

A this is where it differs for example from the State-Action-Reward-State-Action aglorithm (SARSA). It uses approximations to update the value of a whole set of actions, a policy. SARSA is well suited when the value of each action can be monitored. It notably update the parameters of the approximation function after each movement thanks to a change in value from $t-1$ to t, known as a temporal difference. The update incldues a gradient term, compute from the difference between the state's value approximation and the real state's value.

The Deep Q networks (DQN) are close to what I propose. This consists in an off-policy algorithm using neural network approximations of the states' value to decide on an action. And as the learner makes better moves, the neural network is fed with more useful information. And thanks to the so called "Experience replay", when training the network, random minibatches from the replay memory are used instead of the most recent transition. Such efficient implementation makes the DQN very adaptive to different tasks, as demonstrated for the learning of multiple arcade games while using the same network architecture.

Hopefully the proposed algorithm offers even more flexibility than the evoked algorithms. That would be thanks to removing the assumption that the state's values are permanently

monitored during the training phase. This is possible by the implementation of a *signal function*. Its role is to detect a preset objective signal thanks to a deterministic function or a well tuned sensor. The aim is notably to allow the algorithm to be practial when evaluating non-idealized states - unlike video game screenshots - involving stochastic perturbations - such as real world pictures or videos.

3 Proposed algorithm

3.1 Description

To describe the proposed principle, we are going to replicate chronologically an episode of the algorithm in action.

We call the learning algorithm the *learner*. The learner's aim is to automate a course of actions x^* observing at t a stochastic state x_t. Notice that x^* is also equivalent to a new state

At each episode, the learner is facing different choices of actions. The space of possible actions is called Ω. It can be a discrete or continuous space. It can be set deterministically ; for instance, in the context of a game, there is a well determined space of actions that a human can tune in advance. It also can be infinite ; for instance, a mobile robot can explore the real 3D world in almost infinite ways.

The choice, with probability $1 - \epsilon$, which corresponds to a non-exploratory move, is called x^*, the maximum over a set of predictions. These predictions correspond to a regressor function that we describe below. These predictions are made over X', a set of simulated actions belonging to Ω. x^* is then added to a training set, X.

Now, we are assuming that the learner posesses at least one sensor allowing it to get an objective signal. This signal, $s(x^*)$, is a function of the state generated by the learner's last move. It is conditional on the current state and takes positive value if a *gain* is sensed, and negative value if a *loss* is sensed. At least one value should be potentially triggered. A typical example for $[gain; loss]$ would be respectively $[1; -1]$. They should be set so they are proportional to the learner's closeness to its objective.

While this signal is off, the learner counts the number of actions since the signal was last triggered. If this signal is triggered, the current episode is over and each action since the last signal was triggered is receiving a value given by the function $d(\gamma, \Delta, \delta, s(x^*))$. γ is a discount rate, Δ the number of actions in the episode and δ the index of an action in the episode (the first action in the episode has $\delta = 1$). So the closer an action was from the triggered signal, the larger the magnitude of its value will be. Each value corresponding to all the last actions are appended to the training target vector y. I realize this is only a particular way to discount these values, and different ways might be considered for different applications.

We then get a first training sample on which are going to apply a regressor function $f()$. Its aim is to minimize the loss $l(f(w, X), y)$. The parameter set corresponding to the minimal loss is w^*. It is going to be used in the next episode in order to predict the value of simulated actions

Here is a formal version of the algorithm:

Algorithm 1

Initialized variables:
X: empty m-columns feature matrix
y: empty target vector
$w^* = w_0$: random weights for the regressor
$\boldsymbol{\gamma} \in [0, 1]$ discount rate
$\Delta = 0$: number of actions
Other variables:
$f()$: regressor function
m: number of features
$p_t(x^*)$: stochastic process, function of the last existing chosen state
X': m-columns simulated states matrix
$x_t(p_t) \in \mathbb{R}^m$: current state vector
$\Omega_t(x_t)$: set of possible actions
ϵ: randomness rate
$s(x^*) \in \mathbb{R}$: signal value potentially triggered by the modified state
WHILE the learning process is on:
record $x_t(p_t)$
$X' :=$: empty m-columns matrix
FOR x' in $\Omega_t(x_t)$:
append x' to X'
With probability $1 - \epsilon$
$x^* := \arg\max_{x' \in X'} f(w^*, X')$
With probability ϵ
$x^* :=$ random sample $x' \in X'$
append x^* to X
IF $s(x^*)$!= 0 :
FOR δ in 1:Δ:
$value = d(\gamma, \Delta, \delta, s(x^*))$
append $value$ to y
$w^* := \arg\min_w(l(f(w, X), y))$
$\Delta := 0$
ELSE:
Δ += 1

3.2 Remarks

3.2.1 Regressor function

Depending of the sensor the learner has at his disposal, some regressor functions can be more or less suited.

If the possible states form a linear function of the target value, a linear regressor function can be applied. But the states can be strongly affected by stochastic processes. For structured data, non-parametric tree-based methods might be considers. Our typical case is when this happens with unstructured data, such as images, sounds and texts. Then, neural networks might be considered.

This last case is typically mimicking how human would act if faced with a particular situation. The challenge is here to recognise equivalent underlying states thanks to pattern recognition. And considering the algorithm described above, w, would be the set of a neural network's hidden layers.

One interesting aspect is the role of the regressor function. Contrarily to typical applications, the functions's task is utimately not to predict an action's value with maximum accuracy, even though it is often a dependency, but rather to give the "real value" of an action. Imagine for instance a game where two intermediary states might lead to both winning and losing. Such states will get a non-tendencial value from the regressor function, which is a good thing.

3.2.2 Signal function and discounted values

The signal function allows for minimal assumptions on the basic abilities of the learner. While most approximation-based reinforcement learning algorithms suppose the permanent presence of a value monitoring function, here we only assume the presence of a discrete, time-independent sensor triggering one or more values.

We can observe a similarity between the backpropagation mechanism of artificial neural networks. There is an activation and a backprogpagation of the values.

The use of this simple sensor is made possible by the use of discounted values. Even if not every action is monitored during an episode, each step still takes a value at the end of the episodes. In that process, tuning correctly the γ parameter is important. In the following figure, we represent a set of actions that to a gain (signal = 1 here). The discount function that we apply is $\gamma^{\Delta-\delta}s(x^*)$.

In the first case $\gamma = 0.7$, in the second case $\gamma = 0.3$. We see that with a larger discount factor, more emphasis is put on the final result of the set of actions. In some cases, γ should be set considering that two intermediary actions can lead to both a gain and a loss.

With this function, triggering a new signal can take time. But the learner can still look for best discounted value according to the current state. We then benefit from something common to RL algorithms: the use of intermediary objectives.

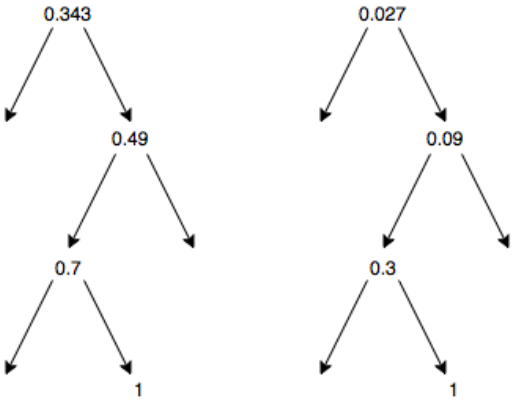


Figure 1: Different discount rates affecting intermediary states

4 Behaviour of the algorithm

include idea RL+regressor (Gabor)

4.1 Convergent estimates for equivalent underlying states

The convergence of the algorithm's evaluations is closely linked to the performance of the regressor function at extracting underlying patterns between states and values, assuming however that this relationship exists.

Let's consider a finite set of underlying states. If the conditions are favorable, we expect overtime, as the training sample is fed after each episode, a better evaluation of the different states and, asymptotically, underlying equivalent states will get the same value. Under another perspective, we expect w^* to converge as X grows.

An interesting point is the expected non-linearity of this process. Indeed, as long as the learner did not discover all gain-yielding states, the regressor's parameters do not permanently stabilize. Once all states have been uncovered, this convergence happens.

If the set of underlying states is infinite, the regressor's parameters can stabilize only if there exist patterns between the states and the value. An example of this situation is a vision-equipped learner whose task is to catch the same ball in different environments.

4.2 Long term gain maximization

Evaluating well the value of equivalent states is only a first step to obtain a learner that is useful in practice. To behave in a long term-maximizing way, the algorithm needs to have a relatively balanced experience between gains and losses.

First, tuning γ in a proper way will help to allow the regressor to evaluate states well faster, via the learner's actions. If possible, a good way to tune this parameter is to make such that

$$d(\gamma, \Delta, \delta, s_{gain}) \simeq d(\gamma, \Delta, \delta, s_{loss})$$

for the δth intermediary action that led to a gain in one episode but also to a loss in another episode.

Second, tuning ϵ well will insure that the learner explores potentially good actions fast enough. A number of episodes-dependent ϵ might be desirable in that regard.

5 Implementaion for a "real-world" tic-tac-toe game

The tic-tac-toe game is a game with initially nine positions organised as a three by three square. After the first player played the first move, the second player is left with eight available moves. This is until one player won or once the whole board is covered, yielding a draw. To win, a player must align his moves on the same line or on the same column, or by filling one of the board's diagonal.

The tic-tac-toe game is a RL classical application, but this time the learner has to play according to his "vision". Notably, when playing against a human, the information available to the robot corresponds to an image of the type: PICTURES OF VISION caption: pictures of the same underlying state This is thus a "stochastic representation" from a finite set of states. The stochastic processes affecting this representation are notably the varying handwrittings and the environment's brightness.

After a sufficient number of games, the robot is able to effectively assign values to the potential actions it might take: ADD 150*150 PICTURE WITH PROBAS ON TOP, FOR 100, 1000, 5000 selfplays REVALUE SO LOSS = -1, DRAW=0

5.1 Particularities of the implementation

The stochastic state x_t is resulting from another player's move represented on an image as seen above. In terms of strategy optimization, here the learner would have to think ahead. The states here are the fruit of *rival* actions, either from a human player or from another robot. And in this implmentation, the learner plays after the rival, so it can win either in three or four moves.

The signal function is determincally set by a human. It triggers the values 1 if the learner wins, -0.1 if there is a draw and -1 if it loses or if it does an illegal move. Notice that an illegal move corresponds only to playing on top of the other player. The learner here knows where it last played.

In this implementation, the discount function $d(\gamma, \Delta, \delta, s(x^*))$ has the form

$$\gamma^{\Delta-\delta} s(x^*)$$

More precisely, I picked $\gamma = 0.3$ as it yields the same value for an early move. The discounted value for moves leading to a win thus are: 1, 0.3, 0.09 and 0.081. For a draw: -0.1, -0.01, -0.001 and -0.0001. For a loss: -1, -0.3, -0.09 and -0.081. For an illegal move, only the signal value is given to the last move.

I implemented an adaptative randomness factor. It has the very *ad hoc* form

$$\epsilon = 50000/log_10(S)^6,$$

where S is the size of the data in bytes. I chose this formula so it equals 99% for a data corresponding to 10000 games. The other learner is penalized and has $\epsilon = 25000/log_10(S)^6$ so our learner has time to learn how to win.

The chosen regressor is an artificial neural network. I chose this type of regressor as I believe it supports the complex relationship between the pixels representing a state and the value of this state. A sufficiently complex structure is thus able to handle the very large number of ways to represent a game combined with the many game combinations. In this implementation, the network is has the following form: PICTURE NN

5.2 Necessary adjustments

A few adjustments were to be made. First, notice that the learning phase is done against a Q-learning robot. After each of the players' move, an image representing the current state of the board is generated. This is image is then reduced and appended to the training set. SHOW LARGE IMAGE -> SMALL IMAGE transition

Second, our learner had the disadvantage of receiving imperfect information about the game compared to the other robot, knowing exactly where the moves are made. To compensate for this, our learner after each move rotates the game three times and appends the learned values accordingly.

Third, as we do not have an objective sensor at our disposal, the signal recieved by the learner is the same as the signal coming from the oponent robot. It is an objective information indicating the final state of game when it happens.

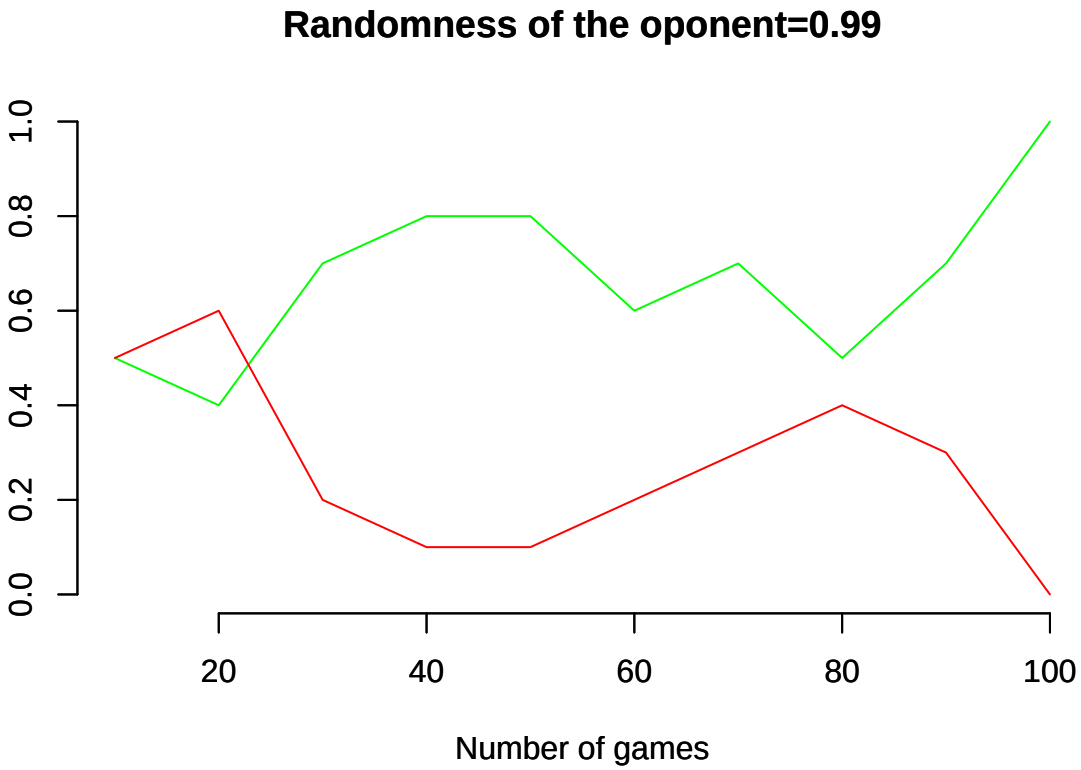
Third, the learning process, via neural networks, does not happen after each game but at regular intervals. This is purely for computational constraints. However, this does not affect in fine the performance of the learner, which is able to win up to 90% of the last phases' games.

5.3 Performance analysis

This section aims at assessing that the proposed algorithm leads to winning strategies.

The following plots show the evolution the learner's performance during the learning phase. We also study how the proportions of wins and losses (and draws by deduction) vary according to the levels of ϵ for each type of robot. Recall that the robot updates its parameters every 100 games, and can potentially improve at this moment.

The green line corresponds to the proportion of won games for the last 100 games The red line corresponds to the proportion of lost games for the last 100 games.



We see that a more eploratory learner improves its winning rate faster but is then relatively limited towards the last phases.

A more exploratory oponent helps the learner to win and thus discover what are the winning strategies. However, making the oponent too exploratory would limit the performance of the learner when playing against a trained human.

INCLUDE ADAPTATIVE LEARNING RATE?

Notice that a learner trained against a random oponent is able, after 100 games, to play easy moves, typically the last move to win, against a human player.

The last plot shows the average number of moves before winning for the last 100 games.

PLOT AVG MOVES

The learner can only win in three or four moves. Against a random player, we that the learner has an average number of moves to win closer to three with more games.

5.4 Improvements

Convolutional layer

Detect all illegal moves

6 Conclusion

The proposed algorithm behaved as expected with its tic-tac-toe implementation. More work has to be done concerning the proof of convergence for the regressor function and the proof that asymptotically the behaviour of the learner is optimal.

Now, we can think of more useful applications than this overkill tic-tac-toe player. For instance, a learner of this type, installed on augmented reality glasses could help people achieving precise tasks in a more optimal way. It could also be useful in online marketing to automatically identify set of actions leading to better sales, such as the sequence of clicks of customers. We also can imagine learning (thanks to vision) the way natural organisms evolve in their environment to build very adaptative machines.

8 References

Sutton and Barto, *Reinforcement learning: An introduction*, Vol. 2. No. 1, MIT press, 2012, available at people.inf.elte.hu/lorincz/Files/RL_2006/SuttonBook.pdf

http://www0.cs.ucl.ac.uk/staff/D.Silver/web/Teaching_files/FA.pdf

https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf

http://gaips.inesc-id.pt/~fmelo/pub/melo08icml.pdf

Human-level control through deep reinforcement learning http://www.readcube.com/articles/10.1038/nature14236?shared_access_token=Lo_2hFdW4MuqEcF3CVBZm9RgN0jAjWel9jnR3ZoTv0P5kedWvCy4SaOgpK5XXA6ecqo8d8J7l4EJsdjwai53GqKt-7JuioG0r3iV67MQIro74l6IxvmcVNKBgOwiMGi8U0izJ8Lw_A%3D%3D