Playing "Bisca" with Deep Reinforcement Learning

Andre Faria
Instituto Superior Tecnico
Av. Rovisco Pais
Lisbon, Portugal
andre.lince.faria@tecnico.ulisboa.pt

Francisco Loio
Instituto Superior Tecnico
Av. Rovisco Pais
Lisbon, Portugal
franciscoloio@tecnico.ulisboa.pt

Vasco Fernandes
Instituto Superior Tecnico
Av. Rovisco Pais
Lisbon, Portugal
vasco.bailao@tecnico.ulisboa.pt

ABSTRACT

We present a deep reinforcement learning model to successfully learn control policies from sensory input data using reinforcement learning. The model is a multilayer perceptron (a specific type of feed-forward neural network), trained with a modified Q-learning algorithm. We applied this method to a popular card game called ”Bisca”, whose only rule is to play the same trump in a given turn. This approached lead to some improvements (results below).

CCS Concepts

•Computing methodologies → Multi-agent systems;

Keywords

AASMA, Q-learning, neural network, deep learning, deep reinforcement learning

1. INTRODUCTION

Learning to control agents directly from sensory data is one of the big challenges of Reinforcement Learning (RL). The quality of such models is directly proportional to the quality of data that is collected by the system. When addressing RL problems in real world applications, one faces the problem of estimating the value function for a particular domain. Neural networks, especially feed-forward neural networks, are a powerful tool to estimate such functions, given an appropriate state-space and sufficient data points. Even so, weight change (the weights are what is estimated by the neural network) induced by an update in a certain part of the state space might influence the values in arbitrary other regions - and therefore destroy the effort done so far in other regions. If these weights are not saved after each cycle of the neural network, this could lead to very long learning times, or no learning at all. This problem is addressed giving to the neural network, at update time, the previous history of updates. This leads to a more balanced estimation.

Appears in: *Proceedings of the 16th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2017), S. Das, E. Durfee, K. Larson, M. Winikoff (eds.), May 8–12, 2017, São Paulo, Brazil.*

2. BACKGROUND

In this section, we are going to explain the theoretical foundations of our implementations and our train-of-thought throughout the project. We begin with explaining Markovian Decision Processes (MDP), then we proceed to explain the classic Q-Learning algorithm. After this, we explain how to apply the classic Q-Learning algorithm to neural networks and then we go further and talk about the ”Neural Fitted Q-Iteraction” algorithm, referencing to the original scientific paper by Martin Riedmiller.

2.1 Markov Decision Processes

The control problems considered in this paper can be described as Markov Decision Processes (MDPs). An MDP is described by a set of states S, a set A of actions, a stochastic transition function $p(s, a, s')$ describing the stochastic system behaviour and an immediate reward or cost function $c : S \times A \to \mathbf{R}$. Besides this variables, an MDP is also described by a discount factor, $\gamma \in [0, 1]$. The discount factor accounts for the difference in importance between future rewards and present rewards. The objective is to estimate the value function V_π, that is given by:

$$V_\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)|s_0 = s, \pi]$$

$$\pi^* = argmax V_\pi(s)$$

2.2 Classic Q-Learning Algorithm

In classical Q-learning, the update rule is given by:

$$Q_{k+1}(s, a) := (1 - \alpha)Q(s, a) + \alpha(c(s, a) + \gamma min_b Q_k(s', b))$$

where s denotes the state where the transition starts, a is the action that is applied, and s' is the resulting state. α is a learning rate that has to be decreased in the course of learning in order to fulfill the conditions of stochastic approximation and γ is a discounting factor. It can be shown, that under mild assumptions Q-learning converges for finite state and action spaces, as long as every state action pair is updated infinitely often. Then, in the limit, the optimal Q-function is reached. Typically, the update is performed on-line in a sample-by-sample manner, that is, every time a new transition is made, the value function is updated.

2.3 Q-Learning for Neural Networks

In principle, the above Q-learning rule can be directly implemented in a neural network. Since no direct assignment of Q-values like in a table based representation can be made, instead, an error function is introduced, that aims to measure the difference between the current Q-value and the new value that should be assigned. For example, a squared-error measure like the following can be used:

$$error = (Q(s,a) - (c(s,a) + \gamma min_b Q(s',b)))^2$$

2.4 Neural Fitted Q Iteration (NFQ)

The basic idea underlying NFQ is the following: Instead of updating the neural value function on-line (which leads to the problems described in the previous section), the update is performed off-line considering an entire set of transition experiences. Experiences are collected in triples of the form (s, a, s') by interacting with the (real or simulated) system. Here, s is the original state, a is the chosen action and s' is the resulting state. The set of experiences is called the sample set D. The consideration of the entire training information instead of on-line samples, has an important further consequence: It allows the application of advanced supervised learning methods, that converge faster and more reliably than online gradient descent methods. Here we use Rprop [RB93], a supervised learning method for batch learning, which is known to be very fast and very insensitive with respect to the choice of its learning parameters. The latter fact has the advantage, that we do not have to care about tuning the parameters for the supervised learning part of the overall (RL) learning problem. [1]

3. "BISCA" GAME

The ”Bisca” game is a card game played by two intervinients that play against each other. The objective of the game is to maximize the number of points achieved in a game. The trump card is defined in the beginning of each game. The cards are called ”trump cards” if they are of the same suit as the trump card. The cards of the suit of the trump (trump cards) can win any turn regardless of the cards played unless if two trump cards are played. In this case, the player that played the highest trump card wins the turn. A game is a set of turns. A turn is a smaller part of the game that, in our implementation, consists in each player playing a card. The number of turns of a game depends on the number of cards of the game. The game begins with each player holding 3 cards each. Every turn, there is a winner – the winner is the player that achieves the maximum number of points. The winner of a game is the player that the sum of the points achieved in each turn or when plays the highest trump. In our implementation, the only rule that the players have to comply is that each player has to ”assist”. ”Assisting” means that the players has to play, when they have it, a card of the same trump of the card that was played before.

4. IMPLEMENTATION

We choose to use Python to implement the domain of the problem (game logic) as well the neural network. The packages used for the neural network were: PyBrain and Numpy. We thought about using TensorFlow, but given the time that we needed to study and get acquainted with this tool, we decided to use PyBrain.

4.1 State of the game

The state of the system is characterised by two vectors (1 row by 40 columns): the carts that the player holds and the belief-state (characterised below). The state of the game is of the following form:

$$S := [[\mathbb{P}[card_1], \mathbb{P}[card_2], \ldots, \mathbb{P}[card_n]]|[\beta[card_1], \beta[card_2], \ldots, \beta[card_n]]]$$

4.2 Domain

Using the Python language, we implemented the domain of the game using object-oriented programming. We implemented the following classes: Game and Player. The Player class in then extended to implement the different kind of players: Minimax, Human and NFQ player.

4.3 Belief State

The belief state is a vector, organised by value and not by trump of the card. This information is of probabilistic nature. The values of this vector are updated before playing a card and the agent verifies that a change on the environment occurred. It begins as a vector full of $\frac{1}{37}$ values, given that each player has 3 cards (there are 37 cards left on the deck), and the beliefs is what a player thinks the opposite player could have in a given turn. In our code, the update of the beliefs was completely implemented. This could be verified in our implementation checking the belief-state after the game is finished. The expected result is a vector full of 0's (given that all the card had already been played).

4.4 Neural Network

Using the Python language and the PyBrain package, we implemented the neural network. Achieving a good aproximation of the value-function is the objective of the neural network. The type of neural network used is a Multilayer Perceptron. The neural network receives as input the belief-state (further explanation of the next subsection) and the cards that the player holds. This information is codified in vector (Numpy array) form. These vectors hold probabilist information. The vector corresponding to the cards that the players hold is of binary nature (i.e., the values are ”0” or ”1”). The neural network is trained with the state and the value-function of the previous state. The value-function, when pushed to its limits is described by:

$$V_\pi(s^l) = R_t + \gamma V(s^{l+1})$$

where s^l is the present state and s^{l-1} is the previous state. We apply the value-function to the previous state. The neural network that we implemented has 80 input layers, 1 hidden layer (with 100 nodes) and 1 output layer (Value-Function). The number of input layers is in line with the belief state definition above. Given that the belief state is composed by two vectors with 40 columns and 1 rows each, it makes sense to each node get a value.

5. CYCLE OF OPERATION

The cycle of operation of our system is described by the 3-state-graph below:

$$\begin{tikzcd}
\text{start} \arrow[r] & q_0 \arrow[rr] & & q_1 \arrow[dl] \\
& & q_3 \arrow[ul] &
\end{tikzcd}$$

5.1 State q_0

The state q_0 stands for the initilization of the Multilayer Perceptron. When we initialize the neural network, the value-function is poorly mapped. Given that, the return of the value-function equated above are going to be completely distorted given that the network has not been put through any training.

5.2 State q_1

The state q_1 stands for the generation of data samples that are going to be used to train the network. This dataset (set of data samples) is composed by the state of the game at an instant t.

5.3 State q_2

The state q_2 stands for the training phase of the cycle. In this phase, the neural network is provided the dataset generated in the previous state.

6. RESULTS

In Figure 1 is possible to check the performance of the NFQ-Agent *versus* the Random Agent. It can be seen that the majority of the games played, the NFQ-Agent is superior to the Random Agent.

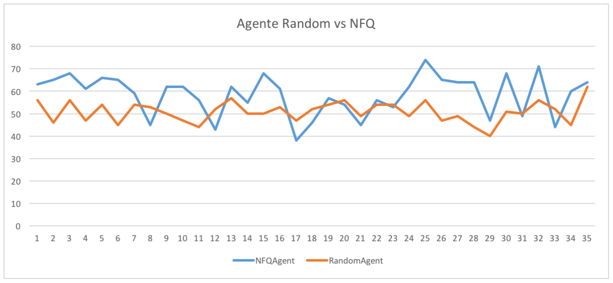

Figure 1: Random Player vs NFQ Agent

In Figure 2, its possible to check how the fitted error of the network decreases with training. In this graphic, the x-axis represents the cycle of training. A cycle of training consists in playing 50 games, then training the network and so on until reaching 100 cycles of training.

7. CONCLUSIONS

The paper proposes a modified version of NFQ algorithm, a memory based method to train V-value functions based on multi-layer perceptrons. By storing and reusing all transition experiences, the neural learning process can be made very data efficient and reliable. Additionally, by allowing for batch supervised learning in the core of adaptation, advanced supervised learning techniques can be applied that provide reliable and quick convergence of the supervised learning part of the problem. NFQ allows to exploit the positive effects of generalisation in multi-layer perceptrons while avoiding their negative effects of disturbing previously learned experiences. After a careful review of this paper, its obvious that we achieved all the proposed objectives defined on the project report delivered in the beginning of the semester. Its also obvious that this algorithm works as our implementation.

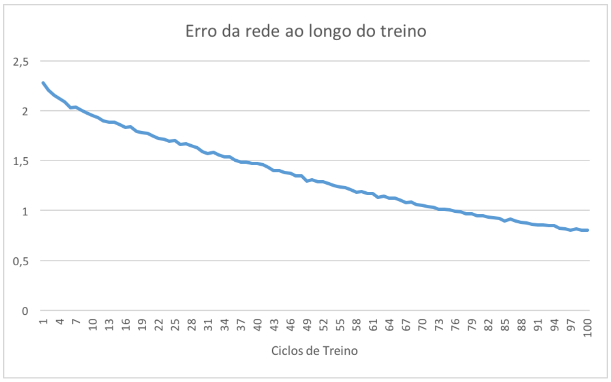

Figure 2: Fitted Error - Neural Network

Acknowledgments

We would like to thank professor Manuel Lopes, Filipa Correia and Ana Paiva for all the support in laboratory and theoretical classes along the whole semester. This work would not be possible without the precious help from each one of this teachers.

REFERENCES

[1] M. Riedmiller.