Computing Methods for Experimental Physics and Data Analysis

# Playing *Connect-4* with Deep Neural Networks

Luca Arnaboldi*

a.y. 2020-2021

**Abstract**

The goal of this project is to train a DNN to play *Connect 4*. Although the game has been completely solved many years ago, it still offers a really interesting environment where training and testing neural networks. First of all, I've implemented a game engine that allows to play games and tournaments between players based on many different classical strategies. after that, I've used the data collected from many games between known agent, in order to train a convolutional neural network with supervised learning. In the end I've explored reinforcement learning techniques to create a neural network that is able to play without relying on external agent for training.

*Connect-4* is game where two players challenge each other in making a connection of 4 checks in a $6 \times 7$ board[1]. Since it is a perfect information game and the number of states is relatively limited, the game can be completely solved through enchanted brute-force techniques. Nevertheless, it is interesting to address the problem using Deep Neural Networks, since the deterministic algorithms allows to evaluate exactly the performance of the machine learning ones.

The first thing to do is writing a game engine that allows to play the game, make sure that every plays allowed position and checks for the winner. I've chosen to code in `Python`. The 3 principal classes of the GameEngine are:

- `Board`: it handles the board and all the methods to modify and check it. It's not contained in `Game` class since some strategies need methods on the board, but not a complete game
- `Game`: it handles the alternation between players moves by calling the proper methods on `Player` objects at the right time.
- `Player`: it's an *abstract class* that must be base class of all players.

The second step is code some players that plays with a given strategy. There are four category of players:

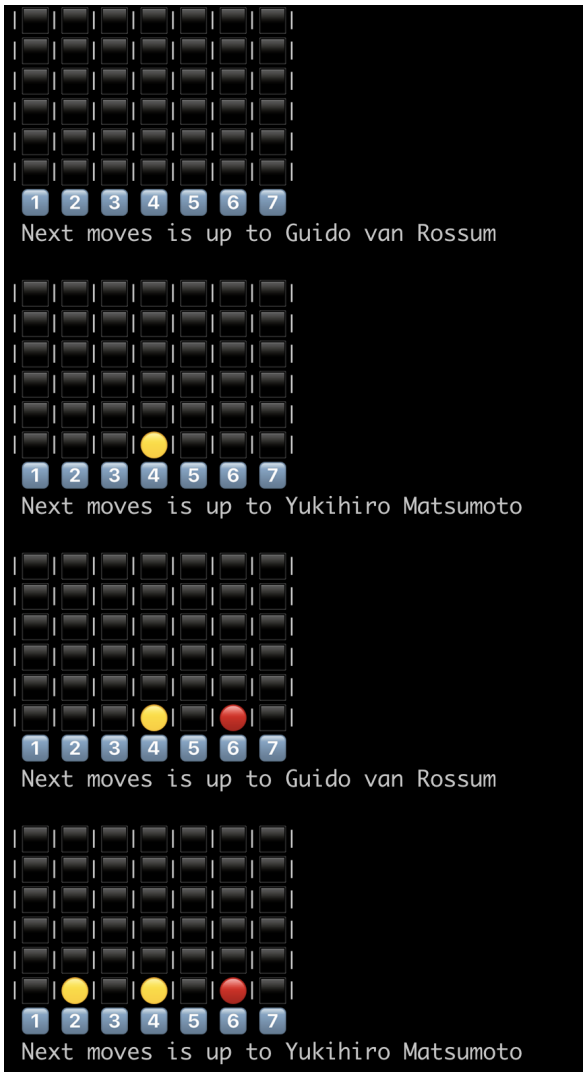
- *Basic Players*: they implements players without a real strategy. We have:
  - `RandomPlayer`: plays a random move between the allowed ones.
  - `ConsolePlayer`: read the move from `stdin`.
  - `FixedMovesPlayer`: Play a given sequence of moves. Used principally for unit testing.
- *Search Players*: these players are all based on *minimax algorithm* applied on the moves tree. Most of these players uses *alpha-beta pruning* technique to speed up the exploration. In addition to the classic implementations, there is a version that uses an heuristic on the move order, and another that adds up random noise to the moves score in order the strategy not deterministic.

  This category includes also the `PerfectPlayer`, that is nothing more than a wrapper for my engine of a `C++` implementation of a full search algorithm[Pon17b; Pon17a], that I've included in my repository as submodule.
- *Combination Players*: players that combines the strategy of two or more other player.
- *TensorFlow Players*: they play using board evaluation through a `tensorflow` model.

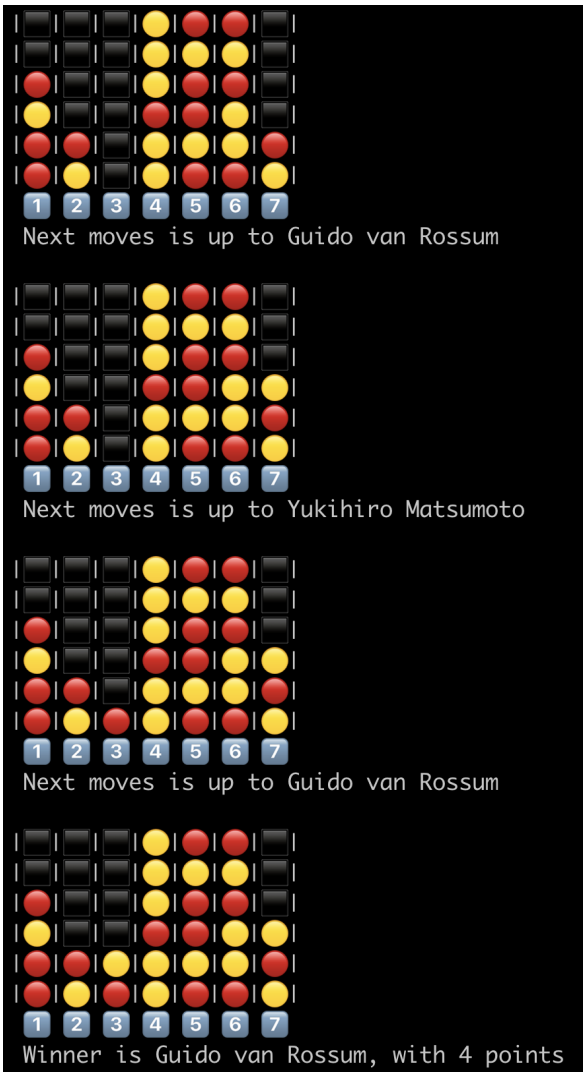
A screenshot of the game engine running a game it's reported in Figure 1. Full documentation of game engine and players can be found on GitHub Pages branch of the project.

*Id. Number: 565650, Email: l.arnaboldi@studenti.unipi.it

[1] More on *Connect4* rules at Wikipedia.




(a)




(b)

```
from connect4.GameEngine import Game
from connect4.Players import NoisyAlphaBetaPlayer, PerfectPlayer

g = Game(
  NoisyAlphaBetaPlayer(depth = 7, noise = 0.5, name = 'Yukihiro Matsumoto'),
  PerfectPlayer('./perfect-player', name = 'Guido van Rossum'),
  starter = P2
)

while not g.finish:
    print(g)
    g.next()
print(g)
```

(c)

Figure 1: example of a game played with `connect4.GameEngine`. (a) beginning; (b) ending; (c) code that has generated the game.

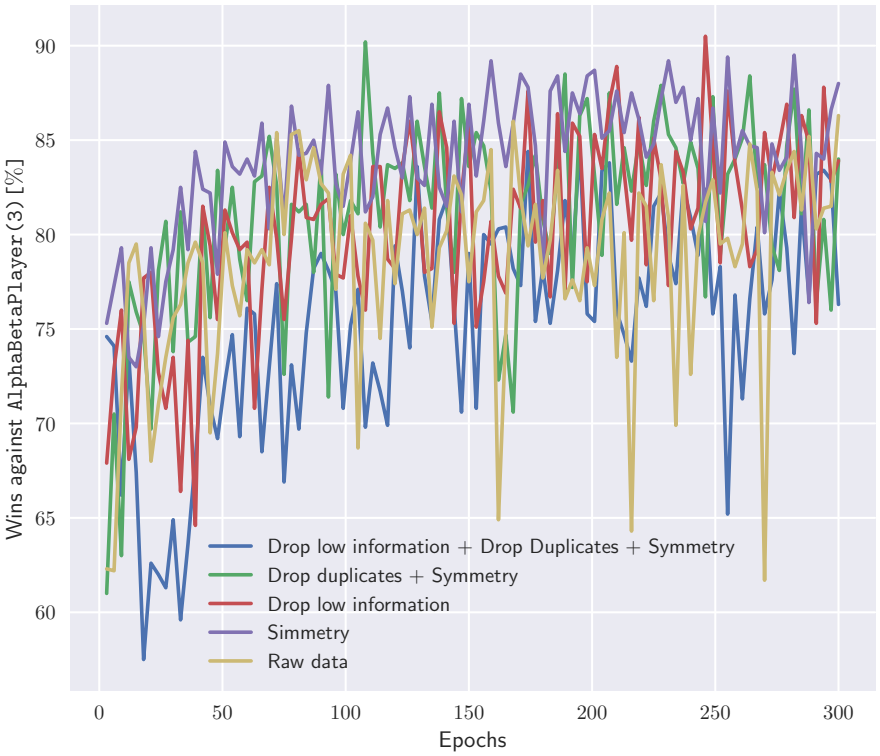



Figure 2: result of supervised training.

Once the setting is ready, we can start using DNN. The first approach I've tried is *supervised learning*. I've coded a derived class of `GameEngine.Game` called `SupervisedGame` where a third player watch the game between other two, and stores the move it would done in every situation. The collected data is the dataset used for the training. I've collected the data from 42000 games between `RandomPlayer` and `NoisyAlphaBetaPlayer` with different noise and depth, all supervised by `CenteredAlphaBetaPlayer(6)`. Three corrections could be applied to the raw data:

- *Symmetry*: the board is symmetric over vertical reflection. The symmetry can be applied to double the data in the dateset.
- *Duplicates*: some board configuration can appear more than once; duplicates can be removed.
- *Information Threshold*: the supervisor may not able to find the best moves in a configuration, resulting in a low information distribution that may be discarded.

Since the board has already a 2D structure I've decided to use a convolutional neural network, with kernel size equal to 4. After the convolutional layer there are 2 hidden layer with ReLU activation function, and finally the output layer has 7 (as the number of columns) neurons. Each output neuron value is the `logit` of the probability that the agent should move in that column. I've trained the network for 300 epochs, using binary cross entropy as loss function; the result of the training are in Figure 2.

Supervised Learning is a bit like cheating if the goal is to create a stand-alone player who can play using a neural network. This because we need a player which is able to play reasonably well to generate the dataset. To train a neural network without any external agent we have to use *reinforcement learning*. The approach I've decided to take is a revisited version of *deep Q-learning*: I train a DNN that is able to assign a score to each board configuration, and the agent makes the move that gives highest score. In each episodes the agent plays against itself and the configuration scores are calculated based on the game result when the match is finished. The *epsilon-gready strategy* is used to deal with the trade-off between learning and exploration. The loss function used is mean squared error. A few things can still be configured:

- *Epsilon decay*: the decay of $\varepsilon$ over episodes can be linear or exponential. The difference can be seen in Figure 3.
- *Flatten Neural Network*: it could be that a not convolutional neural networks learns better.
- *Multi-Traning*: it may happen that an agent adapt to play only against himself and does not leanr a good general strategy. This could be avoided by training 2 or more agent together.

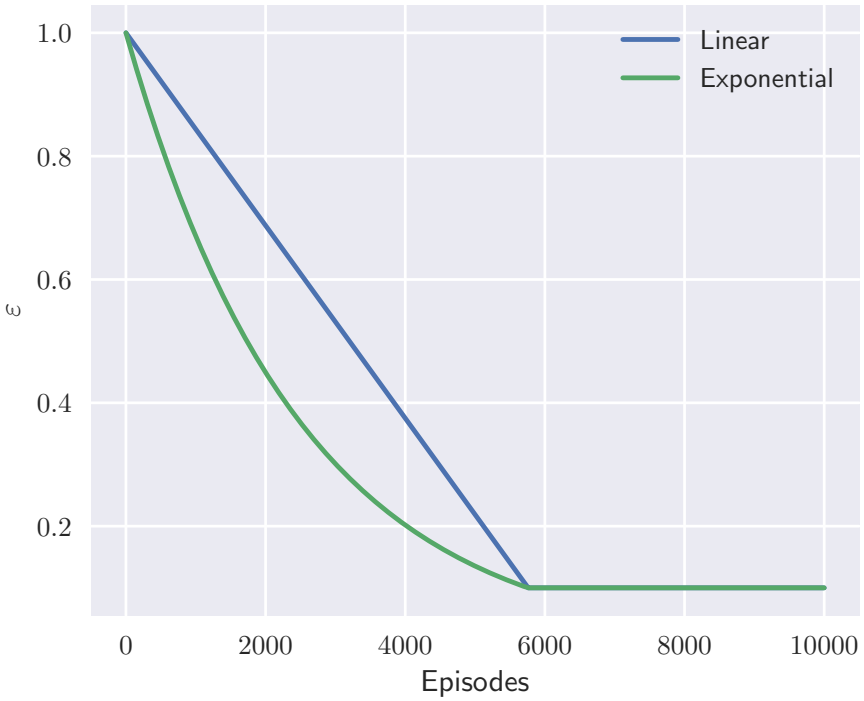



Figure 3: two possible $\varepsilon$-decays over 10000 episodes. A minimum value of $\varepsilon$ has been set since some tests showed that too low $\varepsilon$ are useless to learning.

I've tested all these hypothesis and result are shown in Figure 4. It's clear that the flatten neural network is worse than all the others, so the initial choice of using a convolutional one is correct. There is no appreciable difference between linear and exponential decay, but since the former make the learning faster, the linear one is preferable. *Multi-training* does not significantly boost learning, but it outputs three independent model that can be combined with `PoolPlayer`. Moreover, training more than one agent does not affect the performance too much since most of the job can be parallelized.

I've decided to go with a 50000 episodes multi-training of 3 agent, with linear decay. The result are plotted in Figure 5.

# References

[Pon17a] Pascal Pons. *Connect 4 Game Solver*. https://github.com/PascalPons/connect4. 2017.

[Pon17b] Pascal Pons. "Solving Connect 4: How to Build a Perfect AI". In: (2017). URL: http://blog.gamesolver.org/.


Figure 4: test different training strategy of Reinforcement Learning with 10000 episodes.


Figure 5: result of supervised training.