Betting Agent for NCAA Basketball games

Igor Sadoune, Jeff Sylvestre-Decary
Department of Mathematics
Polytechnique Montral
igor.sadoune@polymtl.ca, jeff.sylvestre-decary@polymtl.ca

April 26, 2019

Contents

1 Introduction

Reinforcement learning (RL) methods have recently made some noise in the financial sphere. While very promising, for nearly 20 years now [1], deep RL in finance still largely unknown. However, tasks such as automated trading and portfolio selection seem very well suited for RL implementation [2]. In this report we study and discuss the application of deep RL algorithms on a sport betting problem. For that, we designed a gym-like environment in which the agent has to make profit in betting on sport event. We chose, as an example, to concentrate on NCAA men's basketball collegiate league, for which we use a related dataset from march madness Kaggle [3] competition. This is a financial environment, namely, a stochastic time-dependent state space, where each action can be seen as a stock trading decision. For instance, for a certain team on a given game, bet on the tuple $(game, team)$ is equivalent to buy a stock. However, for the sake of simplicity, the agent can't hold or short the stock, but the return is observed and perceived in its totality just after the occurrence of the match. This fact makes the bet closer to buy a debt rather than an equity stake. The core of our experiment is developed according to the following structure.
In a first time we present the main features of our environment, and we also discuss the reasons behind the choices we have made. In the second part we focus on implementation and the two main approaches we tried in order to tackle this problem. Then we discuss the different strategies that could be applied into further work, and finally we conclude on what we learned from this experiment but also the difficulties that we had regarding the structure of the problem.

2 Environment (Vanilla)

During our experimentation we tested out several configurations, in particular different reward functions. From this, two main versions emerged, each one was designed to be suitable for a specific algorithm given the fact that we use different formulation of the action space.
We constructed the environment with the naive assumption that there is exactly one match per time step, in which the agent has the choice of betting on one of the two teams or not (skip the bet). Originally, i.e. from the data, we observed that the number of match differ from day to day. Also, the agent always consider the information he has relatively to the match itself, namely, team 1 versus team 2, but never regarding a particular team, i.e. the labels team 1 and team 2 are assigned randomly to each team that are playing at a given step. In other words, the agent doesn't consider as a feature that a particular team is playing. A major challenge is that the reward are drawn stochastically since they depend on the outcomes of matchs, which are (the outcomes) stochastic since we assume the winning team win 90% and loose 10% to add some noise in our data.
An episode is structured as follow: the agent start with a given amount of money (the quantity doesn't matter since it is proportional with the cost of betting and the rewards) and in the vanilla version of the environment, the agent dies when the current amount of money is smaller than the cost of betting. In such case the current episode ends. In the episodic case, we also limited the number of step per episode to 3300 games.

2.1 Dataset

We used two datasets in this project. The first one was provided in a Kaggle [3] competition. It was build by the NCAA itself and Google Cloud. This dataset contains the results of every NCAA matchs since 1985, but detailed results only for matchs which occurs after 2003. There is over 90 000 matchs in total in this database. The different variables they provide can be find in Table 1.
As we describe in the first section, we want our agent to be able to place some bets on NCAA matchs. To be able to illustrate how much money he makes on an investment, we wanted to use the real moneyline for a given match. The moneyline can be positive or negative. In the first case, it represents the amount of money you receive by investing 100$ on the given team, and

Variable	min	max
field goals made	19	30
field goals attempted	41	64
three pointers made	4	11
three pointers attempted	12	32
free throws made	10	20
free throws attempted	13	29
offensive rebounds	6	15
defensive rebounds	20	29
assists	10	18
turnovers committed	10	17
steals	4	10
blocks	2	6
personal fouls committed	14	23

Table 1: NCAA's Database description

if it is negative it represents the amount of money you need to invest on the team to receive 100$. Free historical data regarding those informations are difficult to find, but we found sportsbookreviewsonline.com. This website provided us with the moneyline of most matchs since 2008. This results in a training set of almost 45 000 NCAA matchs.

2.1.1 Preprocessing

Once we found these datasets, we first had to preprocess data and use an ID for every team to link the information of both datasets. Secondly, NCAA dataset gives us detailed results of every games, however to be able to make a prediction on a match, we needed information on both teams playing before the game starts. To do so, we built a dynamic database which illustrate the performance of a team based on the observed statistics (Table 1 variables) of last matchs. For the first match of the season, the performance of a team is not yet known, so we used the average performance of the previous year. As a team played matchs in a given year, we used a weighted average from the observed matchs and last year average statistics to have the most accurate information on a team on a given day. Moreover, we also computed the number of win and loss a team had since the beginning of the season on a given day, and the second dataset gave us the moneyline for every team of

most matchs. We hope that this preprocessing of the data allows us to have a global idea of the latest performance of a team. Finally, for any given match

2.1.2 Data Evaluation

Before trying an RL implementation we worked around the dataset to verify its integrity. For that, we have used the data in order to verify if our variables were able to predict the winning team. We applied a non linear support vector machine with polynomial kernel of degree three on a binary classification problem, in which the targets were 0 (winner = team 1) and 1 (winner = team 2). We deduce that the dataset is relatively hard to learn, and the

Table 2: Poly3 SVM on pre-processed data.

Accuracy	**Recall**	
	Team1	Team2
0.72	0.51	0.84

table above lets predict a substantial stochastic component. As we plan to use at this stage deep RL, it is interesting to know how the data can help the learning agent since in both the implementation we try below a neural network will directly or indirectly predict the winner to help the agent's decision.

2.2 State Space

The state space is a continuous vector of size 16, generated by the dataset directly. Each time the environment is reset, the dataset is shuffled. After each transition from a state to another, the agent is virtually iterating over the rows of the data. The reward signal is computed in different ways depending on the version of the environment (cf.: section 2.4 and section 3), it is always based on the outcome of the previous match. The row of the dataset, namely the states, or assume to be independent, hence the shuffle at the start of the episode doesn't interfere with the reward signal.

2.3 Action Space

In the episodic implementation, we tried few set of actions to evaluate if there was any modification in the behavior of our agent. The first set of actions offers our agent the choice of betting 20$ on any of the two teams playing a given game. Secondly, we offered our agent the opportunity to choose between investing 20$ on any team or skipping the match if the agent is not confident enough. Finally, we combined these two sets of actions to create a third set with 5 options for our agent.

2.4 Rewards

Once again, the set of rewards change depending on the implementation. For the episodic implementation, we wanted our agent to not loose all of his money in the first place, so if the agent loose all his money, it receives -4000 points. In the other case where our agent survives 3300 matches and has over 3000$ in his wallet, the agent receieves +5000. Due to the lack of time, we helped our agent by rewarding it +1 for every step he made, +200 for every correct prediction (or good bet) and +15 for every step with where it has more than 3000$ in his wallet. Finally, when the agent decide to invest 10$ or 20$, he receives +1 or +5 respectively whatever the outcome of the game, and for every game the agent doesn't place a bet, he received a reward of -1.

2.5 Environment Evaluation with Hard-Coded Policies

In order to try our environment, we proceed with two hard-coded policies. This way we have constructed a benchmark, showing the reality of our environment. We thus tried the random policy (the agent takes an action at random between skip, bet on team1 and bet on team2)and an heuristic policy in which the agent bet on the team that holds the better chances to win. We compared their survivabilities and performances.

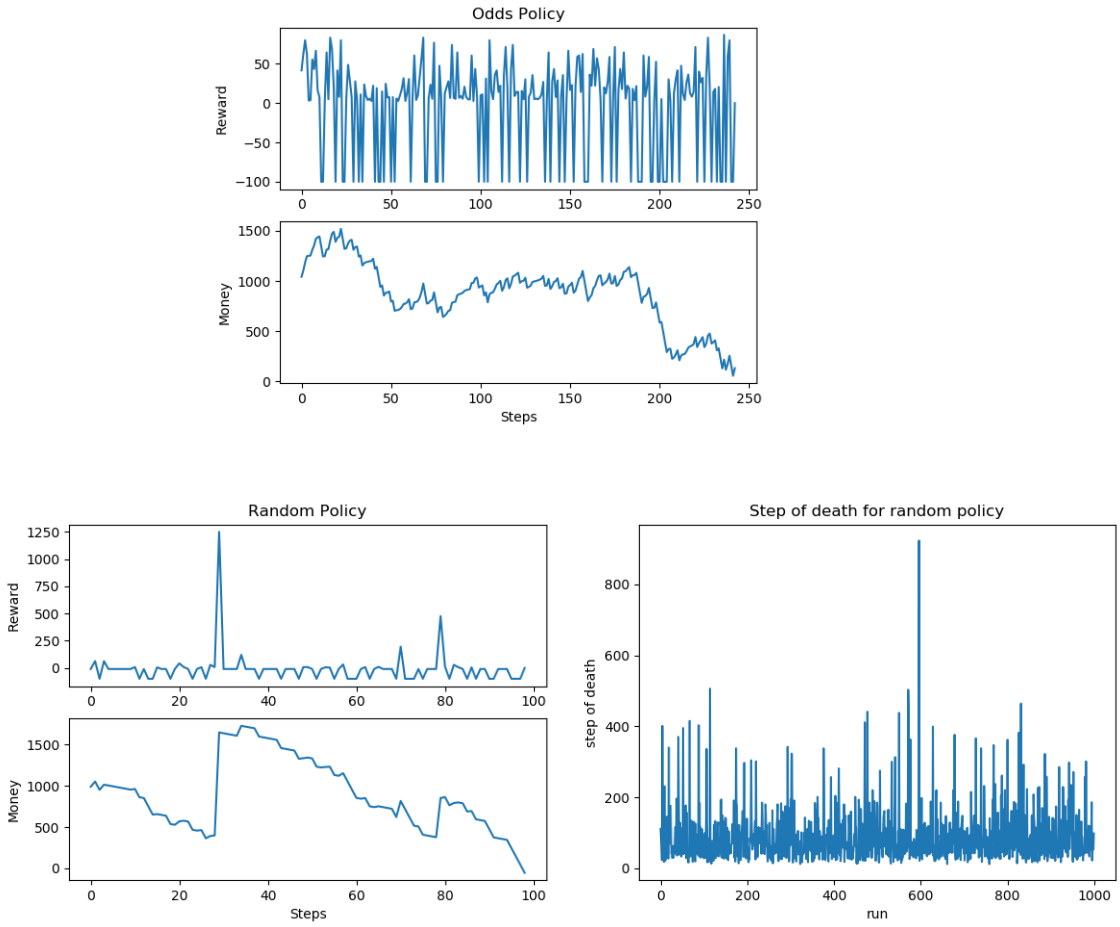

Figure 1: Hard-Coded Policies

There are no real surprises to expect from the random policy, however, we observe that the agent following the heuristic policy, called "odds policy" in the figure, dies really soon, completing merely 250 steps. This denotes the difficulty to predict the winner for each match.

3 Episodic approach

3.1 Double Deep Q-Network

Deep Q-Network is a transformation of the well-known Q-learning algorithm which tries to find the optimal action to take at any states. Q-learning is an algorithm that take an action at a given state and receive a reward. These rewards are then use to update the state-action value function by using the equation below. Once we reach the optimal value function, we can take an argmax over the states-actions pairs to determine what is the optimal policy.

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

In the case of Deep Q-network (DQN), it uses a function approximation to estimate the $Q(S_t, A_t)$ values. As we can see in Table 1, the state parameters are continuous over a large range of values which makes it very difficult to solve this problem in the tabular case. That being said, we decided to use a neural network with two hidden layers as the function approximation. We believe that the flexibility of a neural network will perform better inside our stochastic environment. Moreover, we used experience replay to train our algorithm over a mini batch of size n rather than training over only one observation. The size of the mini batch become an hyperparameter to be tune, but it allows greater flexibility in this model and it will find a better optimize solution. Because of this function approximation and this experience replay, our update rule has now become

$$w_{t+1} \longleftarrow w_t + \alpha[G_t - \hat{v}(S_\tau, w)]\nabla\hat{v}(S_\tau, w)$$

where w represents the parameters of the neural network, $\hat{v}(S_\tau, w)$ represents the output of the current neural network, $\nabla\hat{v}(S_\tau, w)$ is the gradient over the parameters of the neural network and G_t is the observed return after n actions. We started our project by implementing a Deep Q-Network, but we ended up having outputs of the neural network becoming too big. So we turned ourselves to the Double Deep Q-Network [4] which modify the DQN by considering one action-value function Q' to select the right action and another one Q to evaluate the selected action. In other words, the target value we have in the Q-learning update rule is

$$R_{t+1} + \gamma \max_a Q(S_{t+1}, a),$$

and it becomes

$$R_{t+1} + \gamma \max_a Q(S_{t+1}, \arg\max_a Q'(S_{t+1}, a))$$

in DDQN. The important fact with the Double DQN is that we restrict the maximization bias that is responsible for overoptimistic value estimates. Thus, in the Double DQN two value functions are learned, leading to two sets of weights. Since we are working with function approximation, it is important to notice that these two Q functions will not have the same set of parameters because they are not updated at the same time. Since we now have two action-value function to train simultaneously, we added an hyperparameter to let us know when we should update the parameters of target Q values as well. Furthermore, We used an epsilon-greedy method with an exploration decay hyperparameter to provide a way to explore different action in different states, but the exploration decrease as we go through episodes to converge to the optimal value functions. Finally, we also had a learning rate as hyperparameter to optimize.

3.2 Implementation and Results

Unfortunately, we had couple issues due to the fact that we did not have good computational power. Also, we realized at the last minute the Deep Q-Network was producing Nan at some point for the prediction of the action-value function. We did not realize this because the selected action by the argmax function was behaving as expected (Choosing not betting in most case which is action 0 in the implementation). These late modifications limited the fine tuning for the Double Deep Q-Network model, but here's what we got!

We did an episodic implementation of the DDQN where an episode results in being able to survive to 3300 matchs (1 season). That being said, we wanted to see if our agent was first able to survive the environment and able to make money over these 3300 matchs.

As mentioned in section 2.3, we offered our agent 3 sets of actions. The first set of actions consist of only having the choice of betting 20$ on any team. To analyze if our agent was able to learn, we plotted learning curves over only 100 episodes due to time except for the 3 actions set which has 500 episodes. It is obvious that our agent is having trouble predicting who will be the winner of the match, because he always ended missing money and

unable to complete one season with money in his wallet. As we can see in Figure 2, the learning seems very difficult, but the agent slowly does more rewards as the agent approach 100 episodes. Obviously having more episodes would have help, but this high variance is due to the fact that we shuffle our dataset before every episode making it very difficult to learn. Moreover, Reinforcement Learning is known to be a technique which is data hungry, so 40 000 games might not be enough to be able to have a good prediction over these 3300 matchs. However, if we had time, tuning this algorithm would certainly have help considering that the SVM presented in section 2.5 seems significant.

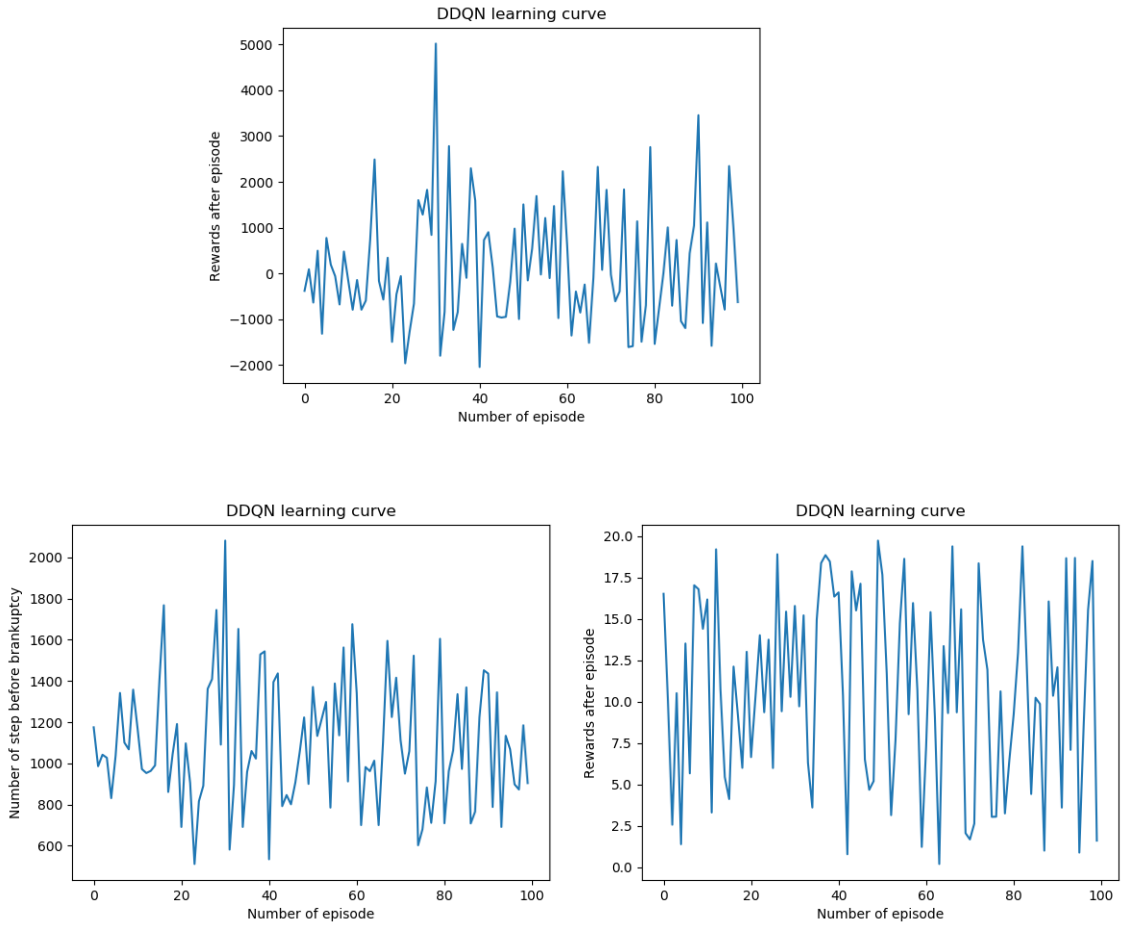

Figure 2: 2 actions: (Upper) Rewards by episode , (bottom left) Number of steps by episode, (Bottom right) Money at the end of each episode

Since our agent was having trouble succeeding in this environment, we offered him a third options which results in skipping the game. This third action is an interesting one because our agent should be able to learn to not bet if he is not sure of the outcome. As we can see in Figure 3, this is clearly what happen because our agent suddenly learn to survive in this environment. When we observe the action taken by the algorithm, we can easily notice the agent is picking more often the skipping action. Even if our agent is not able to make money, just like in this first case, he made some progress with this third action because we can observe an increasing trend in the

money he has in his wallet at the end of the episode. We can also observe by looking the output of the code that the agent choose more often team 2 than team 1 which is what it should do if we refer to the dataset. Finally, it is unfortunate that we only had 500 episodes cause he did not reach the big rewards of +5000 if he finish with more than 3000$, but we believe he would have reach it with more episodes cause all three graphs are increasing as we approach the 500 episodes.

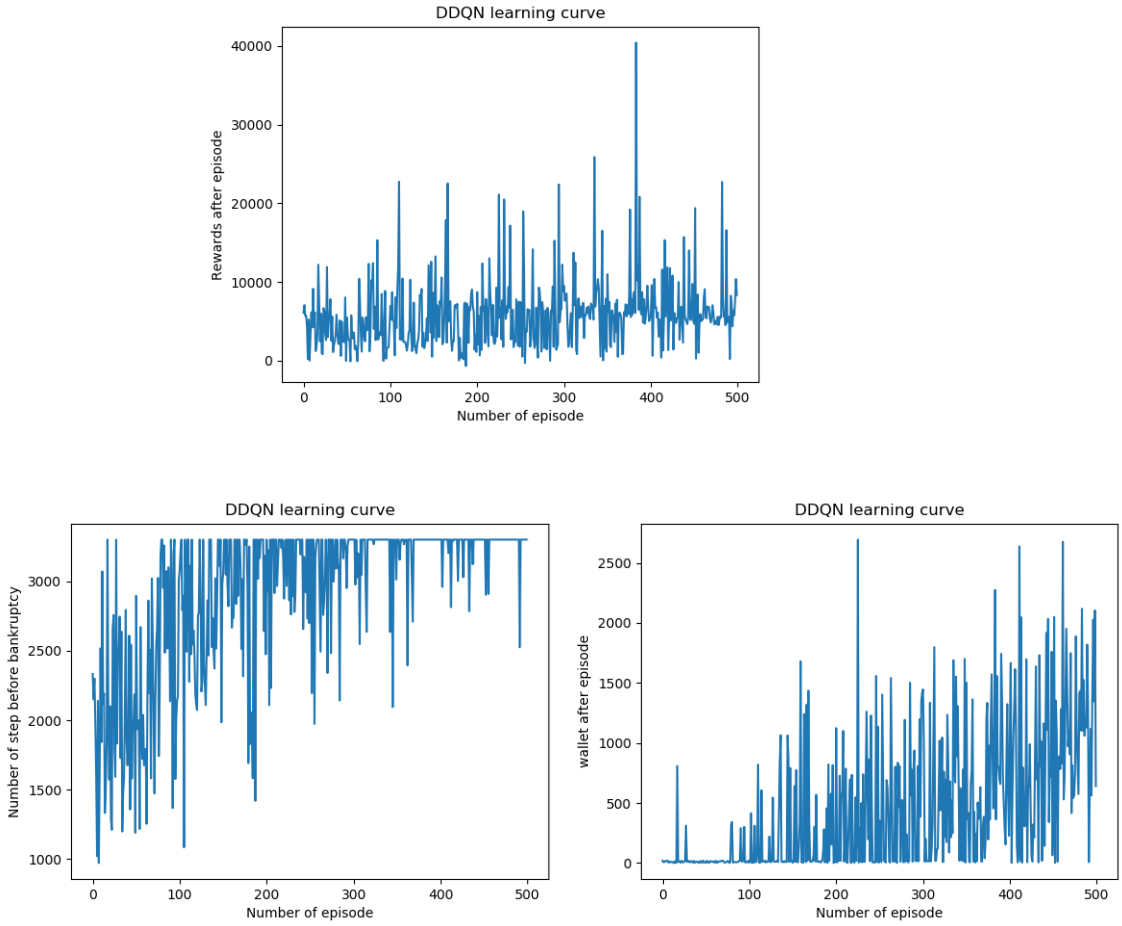

Figure 3: 3 actions: (Upper) Rewards by episode , (bottom left) Number of steps by episode, (Bottom right) Money at the end of each episode

We then offered our agent to bet 10$ rather than only 20$ in the case where

he is not sure of the outcome of the matchs, but only for 100 episodes as well. As we can see in Figure 4, this third set of actions does not succeed at completing 3300 matchs. Having more episodes might have solve this problem, however it mights have took more time than it did for 3 actions. This is because it is more difficult for our agent to learn to skip a match since he has two more actions to consider than in the previous case. It is interesting however to observe that our agent is having less variance in his wallet because of this new action. We can verify this by observing that our agent is choosing more often to bet 10$ than 20$. This is what we wanted because it illustrates that our agent tries to avoid loosing to much money if he isn't very sure, but he still need more episode to learn to skip a match. Once again, the variance can be explain by the fact that we always shuffle our data before every environment.

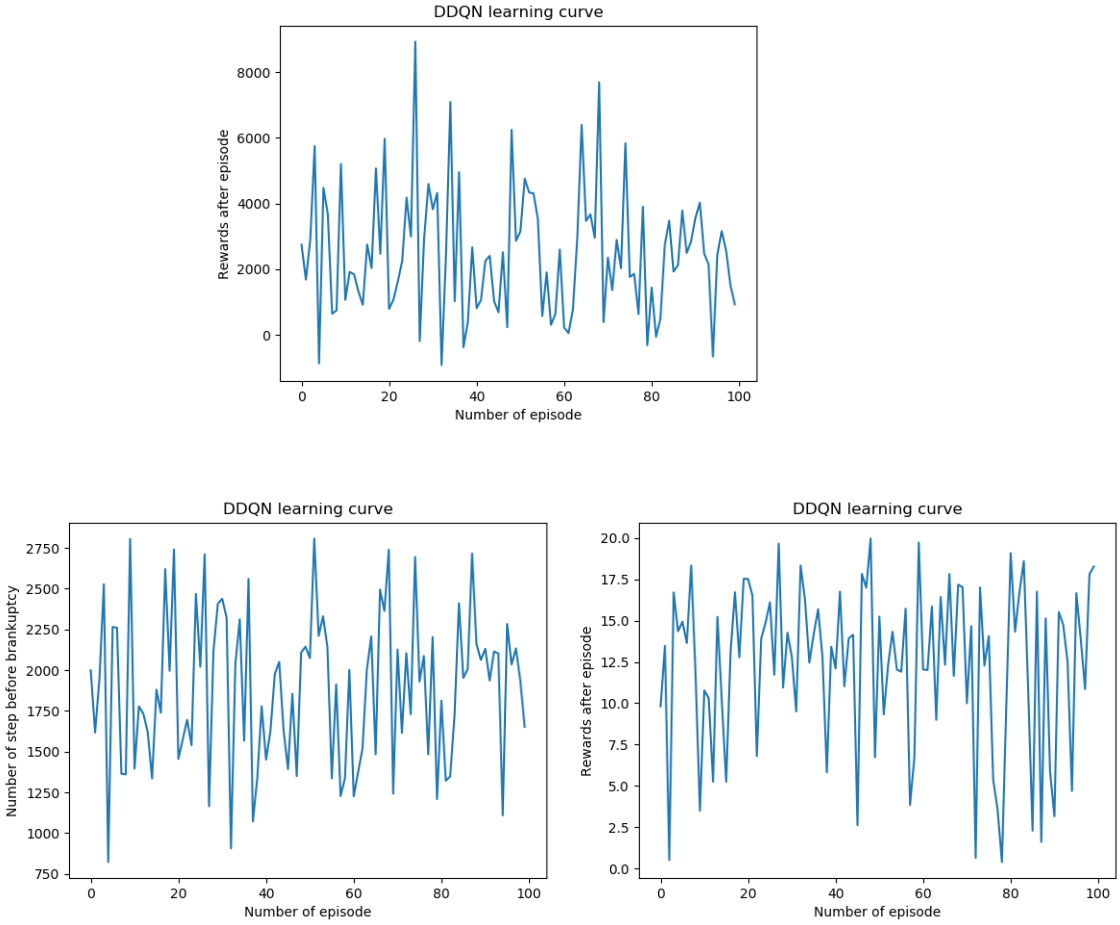

Figure 4: 5 actions: (Upper) Rewards by episode , (bottom left) Number of steps by episode, (Bottom right) Money at the end of each episode

4 Parametrized Action Space and DDPG

In contrast with the first approach, we have intended another strategy based on two major modifications. First, we introduced a continuous treatment for the amount of money the agent can bet, through a parameterized action space form [5]. More precisely, the action space is now a mixed tuple, the agent takes a discrete action: skip, bet1 or bet2 and a continuous parameter is associated to the action bet1 ad bet2. Now the agent is capable of decide

how much money it bets, when it does so.
Furthermore, such a change in the action space motivates the use of another algorithm, namely, the DDPG [6]. This is the strategy used in [5]. Unfortunately, we were unsuccessful at running the scripts until the end, since the computation appears heavy enough to outcast the computational means that we had at our disposal.

4.1 Modifications to the Environment

We defined the action space for a step t as follow:

$$A_t = \cup_{a \in A^d} \{(a, x_t)\}, \quad \forall t = 1, \dots, T$$

where T is a terminal state, x_t the continuous parameter and $A^d = \{a_0, a_1, a_2\}$ the set of discrete action.
The reward setting has also been modified. Lets P_t represents the portfolio, r_t the reward, x_t the amount bet if $a_0 = 0$ and s_{it} a score representing the odds of victory for team $i = \{1, 2\}$ at time t.

$$\begin{aligned} r_t &= \Delta P_t \\ &= P_t - P_{t-1} \\ &= P_t + [(s_{it} - 1)x_t]\mathbb{1}_{win} - x_t[1 - \mathbb{1}_{win}] \end{aligned}$$

where the indicator function indicates whether or not the agent bet on the winning team. The agent is also allowed to reach a negative portfolio value (we interpret that as allowing the agent to borrow money). In other words, the agent can't die, and episodes have now a fixed number of step of 1000 (in the first approach the number of step per episode was set by the death of the agent).
Furthermore, we can define the action space with the following polyhedron,

$$
\begin{aligned}
a_0 + \sum_{i=1}^{2} a_i &= 1 \\
x_i &\geq a_i, \quad \forall i \in \{1,2\} \\
x_i &\leq P_t a_i, \quad \forall i \in \{1,2\} \\
x_i &\in \Re_+, \quad \forall i \in \{1,2\} \\
P_t &\in \Re_+, \quad \forall t \in \{1,\dots,T\} \\
a &\in \mathbb{B}^3,
\end{aligned}
$$

Meaning that the agent can take one discrete action at the time and the value of the parameter is either 0 when $a_0(skip) = 1$ or $x_t \in [1, P_t]$ o.w.

4.2 DDPG

The Deep Deterministic Policy Gradient algorithm was first proposed by google deep mind in [6]. It belongs to the Actor-Critic class. We apply it to the parameterized action space of the latest version of our environment, however, we didn't implement the modification that [5] brings, namely by bounding action space gradient, even if it is recommended in the case of bounded continuous space, such as this one.
Following [6] we use an actor-critic architecture. The *actor* takes as input the state s_t and holds two output layers, one predicting through a softmax function a stochastic vector $\pi^d : S \rightarrow P(A^d)$. The next discrete action to take is then the argmax of the output, giving $a' = \mu(s'|\theta^\mu)$. The second layer, a sigmoid function, output the continuous parameter $\pi : S \rightarrow x$. We define therefore the policy:

$$\pi(a, x|s) = \pi^d(a|s)\pi^a(x|s)$$

The actor minimizes

$$L_\mu(s'|\theta^\mu) = (a - a*)^2 = (\mu(s|\theta^Q) - a*)^2$$

. The critic takes as input the sate action pair and output with a linear terminal layer, a predicted q-value: $\pi^a : (S \times A) \rightarrow \hat{Q(s_t, a_t)}$. The corresponding loss is:

$$L_Q(s, a|\theta^Q) = \big(Q(s, a|\theta^Q) - (r + \gamma Q(s', \mu(s'|\theta^\mu)'|\theta^Q))\big)^2$$

. The critic provides with a single backward pass through the network the gradients with respect to the input (not the parameters): $\nabla_a Q(s, a|\theta^Q)$. These gradients serve then as a target for the actor network. In other words, the critic evaluate an action produced by the actor and the resulting gradients are then used to update the actor:

$$\nabla_{\theta\mu}\mu(s) = \nabla_a Q(s, a|\theta^Q)\nabla_{\theta\mu}\mu(s|\theta^\mu).$$

Finally the parameters of the two network are (softly) updated.
In our implementation we used two hidden layers activated by a ReLU function. Each intermediate layer is followed by a 1D batchnorm layer. In the critic network we use also a "Golrot uniform" weight initialization for the second hidden layer [7]

5 Further Work

We should have fine tuned DDQN to get better results. It would also be interesting to apply this model to other league such as NBA where it might be easier to collect more data and moneyline than it was for NCAA basketball. In the continuity of the parameterized action space with DDPG, it would be natural to implement the full version described in [5] in bounding the gradients.
It could be also interesting to try a continuous exclusive action space with only one parameter. It would be a bounded continuous action for which a positive value means to bet on team1 and a negative value on team2. A region defined around 0 would be the action skip.
A lot of alternatives are available, but the hope of better results still on the side of the reward shaping. Also, we could have consider to drop the action skip and thus to force the agent to bet. In such case, the goal is to survive as long as possible rather than trying to make profit.

6 Conclusion

In conclusion, we observed that this environment is a difficult one to learn for an agent and our agent need more data to learn an optimal stochastic policy. Even though we were unable to find the optimal policy, its interesting to observe that our agent is able to have a learning curve which informs us

that he learned how to not go bankrupt, but there's still some progress to be made to be able to have an agent which can learn a profitable policy!

References

[1] Saffel. M Moody. J. Learning to trade via direct reinforcement.

[2] Kearns. M Nevmyvaka. y, Feng. Y. Reinforcement learning for optimized trade execution.

[3] Google Cloud and NCAA ML Competition 2019-Men's kernel description. `https://www.kaggle.com/c/mens-machine-learning-competition-2019/data`. Accessed: 2019-04-24.

[4] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[5] Stone. P Hausknecht. M. Deep reinforcement learning in parameterized action space.

[6] al Lillicrap. T. Continuous control with deep reinforcement learning.

[7] Bengio. Y Glorot. X. Understanding the difficulty of training deep feedforward neural network.

[8] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.