Optional Semester Project: Reinforcement Learning and Backgammon

Adrien Vandenbroucque
Supervisor: Prof. Rüdiger Urbanke

Fall 2020

1 Introduction

With the increase of computing power, it has become easier and easier to build more complex models that successfully implement the reinforcement learning paradigm. In particular, models such as neural networks have become the go-to to perform function approximation. This project aims at investigating how these ideas can be used in the context of learning how to play games, in particular the game of backgammon.

By reviewing previous work done in the 1990s, this helps settle the foundations for a potentially strong learning using TD learning [11]. Then, it is tested whether using ideas from recent work and implementing them to the system can improve the agent's performance. An example of successful learning in the last decade is that of AlphaGo [10], a program that learns to play the game of Go with no expert knowledge, and is able to defeat the world's best human players.

Overall, we aim to get a better understanding of how an agent can learn from his own experience without external intervention. By doing so, this gives the opportunity to review the basic reinforcement learning concepts, but also gives the freedom to experiment by using new techniques. A large portion of this project is dedicated to the implementation and the results achieved. Specifically, a comparison of the various models trained is given in order to evaluate which agent learns best.

In this report, a review of backgammon and previous work is presented first, explaining in more detail the different important concepts used in order to understand how to reconstruct a similar agent. Secondly, the concepts from reinforcement learning are described, such as TD learning or Monte Carlo tree search. Thirdly, an implementation of it is described and improvements are proposed together with the obtained results. Finally, a summary is given in the conclusion together with the general trends observed.

2 Problem statement

We consider the task of learning to play the game of backgammon with no previous knowledge except the game's rules.

In order to be able to do that, we follow the usual reinforcement learning setting. We first setup the notation and recall basic notions of reinforcement learning that will be used throughout the report. Most of the notation and definitions presented below are taken from [13].

2.1 The Reinforcement Learning Setup

The task here is the process of sequential decision-making. At each time step t, the agent is in a certain state S_t of the environment and has to choose an action A_t from a set of possible actions $\mathcal{A}(S_t)$ to take. The environment then responds by transitioning to a new state S_{t+1}, and also gives a reward signal R_{t+1} which is supposed to represent how well the agent is doing. This

process goes on over and over, and the goal of the agent is to maximize some function of these rewards, for example the cumulative reward $\sum_{t=0} R_{t+1}$.

One way of simply modeling such an environment is to use the concept of finite Markov decision processes, which we define below.

Definition 2.1. *A finite Markov decision process (MDP) is a tuple* $(\mathcal{S}, \mathcal{A}, \mathcal{R}, p, s_0)$*, where:*

- $\mathcal{S}$ *is the set of possible states, with* $|\mathcal{S}|$ *finite.*
- $\mathcal{A}$ *is the set of possible actions, with* $|\mathcal{A}|$ *finite. We write* $\mathcal{A}(s)$ *for the set of possible actions from state* $s \in \mathcal{S}$.
- $\mathcal{R}$ *is the set of possible rewards, with* $|\mathcal{R}|$ *finite.*
- p *is the function that defines the dynamics of the process, i.e., it represents the transition probabilities* $p(s', r|s, a) := \mathbb{P}(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a)$ *for all* $s', s \in \mathcal{S}, r \in \mathcal{R}$*, and* $a \in \mathcal{A}(s)$.
- $s_0 \in \mathcal{S}$ *is a starting state.*

Note that for a MDP, the random variables S_t and R_t have their own discrete probability distributions, and they depend only on the preceding state and action as we see in the definition of p. This is similar to the Markov property of stochastic processes and that is why we call these MDPs. We also see from definition 2.1 that the probabilities given by p completely describe the dynamics of the environment.

From p, we can for example get the probabilities $p(s'|s, a)$ by summing over possible rewards, or we can get the expected reward given that we are in a certain state and choose a certain action.

2.2 Rewards and Episodes

Now that we presented the setup in terms of random variables, we can restate the goal of the agent as maximizing the expected return, which is a function of the sequence of rewards. In our case, after some time step t, we will get a reward sequence $R_{t+1}, R_{t+2}, ..., R_T$. Here, T denotes a final time step, or the end of an episode. When considering games such as backgammon, an episode corresponds to one play of the game and T corresponds to the time step when the game ends, which is assumed to be finite.

For the purpose of this project, we define the return at step t as simply

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T = \sum_{k=t+1}^{T} R_k, \tag{1}$$

the sum of rewards after time t. We do this because for the game of backgammon, the number of moves that can be made until the end of a game is finite. A more general return can be given by introducing the concept of discounting. This is almost identical as before except that future rewards are discounted as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{2}$$

where $\gamma \in [0, 1]$ is a parameter called the discount rate. It sets the present value of future rewards, making for example a reward received k steps in the future only worth γ^{k-1} times what it would be worth if received immediately. For $\gamma = 1$, we retrieve the simple sum of rewards.

2.3 Policies and Value Functions

In order to go further and present useful definitions, we need to go back to how the agent interacts with the environment. So far, we said that it must choose some action when at a certain state S_t, and this choice is formally represented by a policy.

Definition 2.2. *The algorithm used by the agent in state $s \in \mathcal{S}$ to choose an action $a \in \mathcal{A}(s)$ can be viewed as a distribution $\pi(a|s)$, the probability that $A_t = a$ if $S_t = s$, and is called a policy.*

Observe that given a policy (that informs us how the agent interacts with the environment) and the transition probabilities p of an MDP (that informs us how the environment reacts to choices of the agent), we now have a whole defined process from a probabilistic point of view. We can state the following useful definitions.

Definition 2.3. *For a given MDP and under some policy π, the value function of a state s is defined as*

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k | S_t = s\right]. \tag{3}$$

Its purpose is to estimate how good it is for the agent which is using policy π to visit state s.

A value function v_π is useful, but it does not provide enough information by itself to reconstruct the underlying policy π when the transition probabilities of the process are unknown. Indeed, not knowing how the environment responds when choosing a certain action in a certain state makes knowing v_π unhelpful in finding π. Thus, we define below a function which captures this missing information by giving a value to not only a state, but the state coupled with a certain action.

Definition 2.4. *For a given MDP and under some policy π, the value of taking action $a \in \mathcal{A}(s)$ in state s is defined as*

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k | S_t = s, A_t = a\right], \tag{4}$$

the quality function. Its purpose is to estimate how good it is for the agent which is using policy π to visit state s and choose action a.

Since both v_π and q_π can be estimated from experience, this is what the agent usually aims to do in order to perform well. By having estimates of these functions, the agent can then indeed always choose to move according to these values by picking moves that yields the highest estimated value. If the agent is successful in having very accurate estimates, then its choices will be close to optimal.

The agent's goal is thus to find a a policy which is able to bring the highest returns over the long run. In the case of MDPs, one can define an optimal policy π_* as being the policy that maximizes the return, i.e., $\pi_* = \arg\max_\pi \mathbb{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k\right]$.

Note that as value and quality functions can be induced by any policies, we can consider the functions v_{π_*} and q_{π_*}, that we denote more succinctly as v_* and q_*. We have $v_*(s) = \max_\pi v_\pi(a)$ and $q_*(s,a) = \max_\pi q_\pi(s,a)$. An agent thus aims to find the optimal policy π_* of the MDP in order to make the sequence of decision that yields the highest sequence of rewards.

3 Reinforcement Learning and Backgammon

3.1 The Game of Backgammon

We first review what backgammon is and the useful rules. This section is heavily inspired from the rules presented on *https://www.bkgm.com/rules.html*.

3.1.1 Setup

Backgammon is a two-player game, played on a board which consists of twenty-four narrow triangles called points. The triangles alternate in color and are grouped into four quadrants of six triangles each. The quadrants are referred to as a player's home board and outer board, and the opponent's home board and outer board. The home and outer boards are separated from each other by a space called the bar. A representation is shown in figure 1.

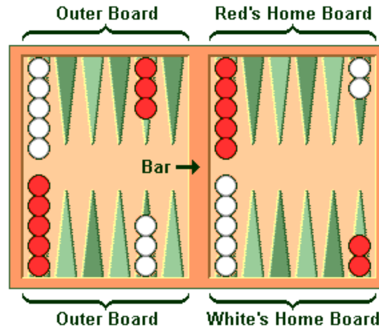

Figure 1: A backgammon board with checkers in the starting position

The points are number from 1 to 24 for either player, starting in that player's home board. Thus the last point is the twenty-four point, which corresponds to the opponent's one point. This can be seen more clearly in figure 2. Each player starts with fifteen checkers of his own color. The initial arrangement of checkers is displayed in figure 1 but can be described in words as follows: two on each player's twenty-four point, five on each player's thirteen point, three on each player's eight point, and five on each player's six point.

Both players are supposed to have their own pair of dice and a dice cup used for shaking, but this will of course not be of importance here.

The goal of the game for each player is to move all his checkers into his home board and bear them off (which means to remove them from the board). The first player who is able to do so is the one who wins the game.

3.1.2 Moving Checkers on the Board

At each turn, the current player first throws his two dice and then move the checkers according to the numbers rolled.

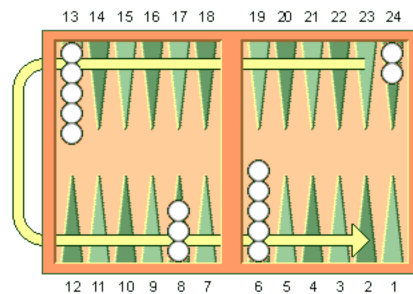

Figure 2: A backgammon board with points numbered from white player's perspective, showing the direction in which checkers should move

From the current player's perspective, his checkers should always move counter-clockwise, and the following rules apply:

1. A checker can only move to a point which is not already occupied by two or more of the opponent's checkers.

2. The numbers appearing on the dice should be thought as two moves. As such, the current player can choose to move one or two checkers, depending on the valid moves that can be played.
3. When the player rolls doubles, he can play the number shown twice. In a sense in this case, he has to move checkers four times instead of the usual two moves allowed.
4. If possible, the player has to use the two numbers shown on the dice. If only one move is possible, the player should use it. If either number allows the player to move but not both, he should play the the larger one. In the case where no move can be made, the player skips his turn. If a double was made, the player should use as many numbers as possible out the four ones.

3.1.3 Hitting and Entering

In previous section, we presented how checkers can move on the board, and learned that a checker can still move to a point where there is a single checker of the opponent. A configuration where there is a single checker is called a blot. If a player is able to move a checker on a blot where the opponent stands, we say that the blot is "hit" and it is then placed on the bar.

When a player has one or more checkers on the bar, he must first enter these checkers back in the board and for that, he has to enter the checkers into the opponent's home board. In order to do so, a checker must be entered by moving it to an open point corresponding to one of the numbers on the rolled dice. As such, if a checker can only land on a point where there are already two or more checkers of the opponent, then the move is not possible.

If none of the points is open in the opponent's home board, then the player loses his turn. If only some checkers but not all can be entered, the player must enter as many as he can and then skip the rest of his turn.

When a player has entered all of his checkers, he can then start moving his other checkers if there is any unused numbers on the dice. The player can choose to move a checker that was just entered, or any of his other checkers.

3.1.4 Bearing Off

We mentioned that the goal of backgammon is for each player to remove his checkers from the board, which is called bearing off. In order to start do so, a player must have moved all of his checkers into his home board. A checker can be removed if a number on the dice is the same as the point on which the checker is positioned. So by rolling a 5 and a 3, the player is allowed to remove the checkers at points 3 and 5 if he has checkers at these points.

If there is no checker on the point indicated by the roll, the player cannot simply remove one of the checker on a lower-numbered point. He must make a legal move using a checker on a higher-numbered point. However if there are no checkers on higher-numbered points, then he is able (and required) to remove a checker from the highest point on which one of his checkers is. Note that a player is not forced to remove his checkers, he is allowed to do any other legal move. For example, if a player rolls a 2 and has a checker on point 5, he is allowed to move it to point 3.

But once again, for a player to bear off his checkers, he must have all of them in his home board. As soon as it is not the case, he must first bring back all his checkers to his home board. This can happen if a checker is hit during the bear-off process. In that case, since the checker enters into the opponent's home board, it has to move all the way back to the player's home board before the process of bearing off can start again. The first player who is able to bear off all fifteen checkers wins the game.

3.1.5 Additional Rules

There are usually additional rules that we decide no to take into account as it only complexifies the problem.

For example, the concept of doubling is used in backgammon. A game is worth one point, but players can raise the stakes and double the number of points if they feel that they have the advantage.

There are also possible additional rules that can be applied at the end of the game, depending on the losing player's number of the checkers left on the board and their positions. This introduces concepts such as "gammons" and "backgammons". For more details, please check the rules on *https://www.bkgm.com/rules.html*.

3.1.6 Backgammon in the Reinforcement Learning Setting

Let us briefly frame the game of backgammon into the reinforcement learning setting. Here, states corresponds to board configurations. There is also full knowledge of the environment, i.e., we know what the transition probabilities are (they simply depend on the outcome of the dice).

In such a case where everything is known, being able to estimate what v_* is is enough to find a policy close to optimal. Indeed, there is no need to add the extra information contained in q_* about the states since we already know the transitions. Thus, the goal for the agent is to be able to estimate v_* as precisely as possible.

For backgammon, with the choice of rewards we make (see section 4.1.3), the value function evaluated at some state s approximates the probability of player white winning.

3.2 Previous work

This project is based on the groundbreaking work from Tesauro [16], who in the 1990s designed a program called TD-Gammon which used a combination of TD learning and nonlinear function approximation to learn how to play backgammon. A neural network was trained using the backpropagation algorithm with the TD errors.

In fact, prior to this, Tesauro already implemented an artificial intelligence called Neurogammon [14] that already played backgammon using neural networks. However, this program only relied on expert knowledge and used it to be trained in a supervised way. Still, after training, Neurogammon was able to play at the level of intermediate human players, but it relied completely on the expert knowledge. Later, TD-Gammon would be able to learn from his own experience with no human knowledge.

The TD-Gammon programs evolved with the years as Tesauro implemented more advanced versions. One can see in table 1 the evolution of his programs.

Program	Hidden Units	Training Games
TD-Gammon 0.0	40	300,000
TD-Gammon 1.0	80	300,000
TD-Gammon 2.0	40	800,000
TD-Gammon 2.1	80	1,500,000
TD-Gammon 3.0	160	1,500,000

Table 1: The evolution of the TD-Gammon program

In the first version TD-Gammon 0.0, there was initially no expert backgammon knowledge fed into it, but it was able to perform as well as other computer programs for playing the game but which had extensive backgammon knowledge, which was really impressive at the time. To learn, TD-Gammon 0.0 played around 300,000 games against itself, and used this knowledge to update its neural network used to approximate the value of board positions (which is the value function v_* we mentioned before).

Later, TD-Gammon 1.0 was revealed and it was using the same ideas as the initial version, but also incorporated expert knowledge to its learning which improved its level of play. The number of hidden units also increased to 80. The later 2.0 and 2.1 versions were upgraded by using what is called a two-ply search when playing a game. That is, instead of looking at the possible moves from a certain state and evaluating them, we look two moves ahead and pick a

move which gives us the highest expected reward. So the agent would look at what the opponent could play also after its own play.

The last version TD-Gammon 3.0 was performing even better by using a three-ply search when playing and 160 hidden units. It is interesting to note that each of these versions played against the world best human backgammon players at the time they were created. And it turned out that the programs were able to reach and even exceed the performance of these top players.

In the next sections, we will go into more detail about how this algorithm works and understand which concepts are used.

4 Concepts

4.1 Learning to Play Backgammon

We have seen in section 2.3 that the goal of an agent is to find an optimal policy, one that yields the highest return in the long run.

There are two main collections of algorithms used to solve this problem, one called Dynamic Programming (DP) and the other Monte Carlo Methods (MCM). DP is useful when the agent has full knowledge of the environment, that is it knows the transition probabilities, while when using MCM we do not assume complete knowledge of the environment. For MCM, the advantage is that the agent learns from experience, by interacting with the environment and exploring it. For DP, the advantage is that it updates its estimates of value functions in part by using other estimates, thus not requiring to wait for the end of an episode. One can find a great explanation of both techniques in chapters 3 and 4 of [13].

Is there a way to combine these ideas and get the best of both worlds? Luckily, Temporal-Difference (TD) learning [11] precisely does this.

4.1.1 Temporal-Difference Learning

We present in this section the TD learning algorithm, and how this is used in our setup.

For most reinforcement learning techniques, when making updates to estimates, the general update rule looks as follows:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \alpha(\text{Target} - \text{OldEstimate}).$$

That is, an estimate is updated by making it move towards the direction of some target value, that we believe is a more accurate value for our estimate. The parameter α is a step-size, and allows to control how much we move towards the direction of the target.

Let us take an example. Suppose that our agent follows some policy π and we wish to estimate the value function $v_\pi(s)$ for all states $s \in \mathcal{S}$ with some estimate $\hat{v}(s)$. In the case of MCM, we will interact with the environment for one episode and get some return G_t, and make the update

$$\hat{v}(S_t) \leftarrow \hat{v}(S_t) + \alpha(G_t - \hat{v}(S_t)).$$

Note that in order to make the update of our estimate for S_t we need G_t, that we can compute only at the end of an episode since it consists of the sum of the rewards until the end of the episode. One may also wonder why G_t is the target here. This is because here, we are estimating the value function, and recall that we have by definition $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$, so G_t could be a good estimate of the value function.

Now TD learning sees it differently, by noting that

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s].$$

TD learning would like as target $R_{t+1} + \gamma v_\pi(S_{t+1})$, which makes it a bit like a MCM. But since we do not know v_π, it uses instead $R_{t+1} + \gamma \hat{v}(S_{t+1})$. That is, its target is based on other estimates, which makes it also a bit like DP. Hence, TD learning performs the following update

$$\hat{v}(S_t) \leftarrow \hat{v}(S_t) + \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}) - \hat{v}(S_t)).$$

The upside here is that we can make the update as soon as we receive the reward R_{t+1} and make the transition to the new state S_{t+1}, there is no need to wait until the end of an episode. This particular update is called the TD(0) update, a special case of the more general TD(λ) algorithm that we present later.

4.1.2 Value Function Approximation and Semi-Gradient Methods

Up until now, every method presented had simple update schemes in order to compute estimates of value functions. But is it feasible in practice? Let us take the example of the game of backgammon. There are approximately 10^{20} possible different states, which would mean that we would need (in case of value function estimation) to keep track of that many estimates. This is of course out of question. In such cases, we cannot hope to find an optimal policy as there are just too many computations to be done. We present in this section a way to find a good approximation of a value function while using much less computational power.

For that, instead of having a table that record the estimates of $v_\pi(s)$ for every possible state s, we assume that the value function can be parametrized with a weight vector $\mathbf{w} \in \mathbb{R}^d$. We thus write $\hat{v}(s, \mathbf{w})$ for our approximation of the value function using $\mathbf{w}$. The way the parametrization can be done is arbitrary. For example, if one believes that a linear function of the weights $\mathbf{w}$ can describe the value function well, then $\hat{v}$ will be a linear. More complex functions of the weights can be used, and in particular the weights can be one of a neural network as it is the case in this project.

So far, all updates have been described as a shift of an estimated value function towards some target that is believed to be a better estimate. Each time, there is a state s, and we shift its estimate towards some target u as explained in the previous section. This pair (s, u) can be seen as the "right behavior" of the value function in the sense that when at state s, the function value should be close to u. This resembles then a supervised learning framework, where (s, u) is an example-prediction pair, and this allows us to perform more complex updates. Here, the objective function or loss function can be defined as $\ell(\mathbf{w}) = \sum_{s \in \mathcal{S}} \frac{1}{2} \left[v_\pi(s) - \hat{v}(s, \mathbf{w})\right]^2$. We refer to the term in the loss corresponding to state s as $\ell_s(\mathbf{w}) := \frac{1}{2} \left[v_\pi(s) - \hat{v}(s, \mathbf{w})\right]^2$. So the closer our approximation $\hat{v}$ is to the true value function v_π we wish to estimate, the smaller the loss function is.

The goal is thus to minimize this loss function $\ell(\cdot)$ and find a set of weights $\mathbf{w}^*$ such that $\ell(\mathbf{w}^*) \leq \ell(\mathbf{w})$ for all $\mathbf{w}$, i.e., the global minimum. Out of all possible methods to update the weights, we use the *stochastic gradient descent* algorithm. The update done on the weights from step t to $t+1$ when at state S_t is

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \alpha \nabla \ell_{S_t}(\mathbf{w}_t) \\ &= \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right] \nabla \hat{v}(S_t, \mathbf{w}_t), \end{aligned}$$

where α is still a positive step-size parameter.

The issue here is that this update assumes that when we get a pair (S_t, U_t), the target U_t is the true value of $v_\pi(S_t)$. This is not the case here of course (otherwise the problem would be solved already since we would know v_π exactly), and we have instead only an approximation of it. Recall for example the MCM target was $U_t = G_t$. Because of this, we use that approximate target in the update to perform

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t)\right] \nabla \hat{v}(S_t, \mathbf{w}_t). \tag{5}$$

Note that the TD(0) target $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$ is dependent on the weights $\mathbf{w}_t$, which makes the target biased. In such a case, we refer to the method as being a *semi-gradient method*.

In the next section, we will see how TD-Gammon uses a more general update rule for the weights which is still based on the same ideas.

4.1.3 How TD-Gammon Learns to Play Backgammon

The algorithm used by TD-Gammon is a combination of what was presented previously. Specifically, it uses a semi-gradient form of the TD algorithm, and uses a neural network to compute the value function. Moreover, it uses a more general version of TD learning known as TD(λ) [11]. TD-Gammon also sets the discounting factor γ to 1, which is why it will not appear anymore below. We represent the weights of the neural network at step t with $\mathbf{w}_t \in \mathbb{R}^d$ and the output of the network for state S_t is written $\hat{v}(S_t, \mathbf{w}_t)$.

Previously, we always presented the problem of estimating v_π when an agent follows policy π. However, what we truly want is to find the optimal policy π_*. To do so, we still perform the updates as before, but actions are chosen in a greedy manner using the estimates $\hat{v}$ to evaluate states values. That is, the agent picks the action which moves it to a state s which has highest value according to $\hat{v}$. This method is called *SARSA* (see chapter 10 in [13]) and it usually aims at estimating the optimal q-values q_* to derive the optimal policy, but recall from section 3.1.6 that in our setting it is sufficient for us to estimate the values v_*.

The update made by TD(λ) to the weights from step t to $t+1$ is the following:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)\right] \sum_{k=1}^{t+1} \lambda^{t+1-k} \nabla \hat{v}(S_k, \mathbf{w}_k), \tag{6}$$

where $\lambda \in [0, 1]$ is a parameter called the *trace decay rate*. Note that when setting $\lambda = 0$, we recover equation (5). The advantage using this technique is that even though when doing update at time $t+1$ we technically need to have access to all the gradients $\nabla \hat{v}(S_k, \mathbf{w}_k)$ for $k = 1, \ldots, t+1$ to compute the sum in equation (6), we can actually build this sum incrementally. To see this, let

$$\mathbf{z}_t := \sum_{k=1}^{t} \lambda^{t-k} \nabla \hat{v}(S_k, \mathbf{w}_k)$$

which contains as many components as $\mathbf{w}_t$, and note that we have

$$\begin{aligned}
\mathbf{z}_{t+1} &= \sum_{k=1}^{t+1} \lambda^{t+1-k} \nabla \hat{v}(S_k, \mathbf{w}_k) \\
&= \nabla \hat{v}(S_{t+1}, \mathbf{w}_{t+1}) + \sum_{k=1}^{t} \lambda^{t+1-k} \nabla \hat{v}(S_k, \mathbf{w}_k) \\
&= \nabla \hat{v}(S_{t+1}, \mathbf{w}_{t+1}) + \lambda \mathbf{z}_t.
\end{aligned}$$

So starting with $\mathbf{z}_0 = \mathbf{0}$, the update in (6) can be done in two steps by updating first

$$\mathbf{z}_{t+1} = \nabla \hat{v}(S_{t+1}, \mathbf{w}_{t+1}) + \lambda \mathbf{z}_t$$

and then

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)\right] \mathbf{z}_{t+1}.$$

The vectors $\mathbf{z}_t$ are also known as *eligibility traces*, which is why the parameter λ that comes with it is called the *trace decay rate*. More information on eligibility traces can be found in chapter 12 of [13].

The complete general learning procedure is described in algorithm 1.

Algorithm 1: Semi-Gradient TD(λ) to Approximate the Optimal Value Function

---

Input: a differentiable function $\hat{v}$, step-size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
$\mathbf{w} \leftarrow \mathbf{0}$
Loop for each episode:
  Initialize S
  $\mathbf{z} \leftarrow \mathbf{0}$
  Loop for each step of episode until S is terminal:
    Pick action A using the neural network with greedy method
    Observe reward R
    $\mathbf{z} \leftarrow \nabla \hat{v}(S, \mathbf{w}) + \lambda \mathbf{z}$
    $e \leftarrow R + \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha e \mathbf{z}$
    $S \leftarrow S'$
  end
end

In the case of backgammon, we follow the setup of [15] and rewards are always set to 0 except when a game ends, where they can then take the value 1 if player white wins. With this choice of rewards, the output of the neural network $\hat{v}(S_t, \mathbf{w}_t)$ is an estimate of player white's probability of winning from board configuration S_t. At each time step, one of the player plays and use the output of the neural network as an evaluation function using a greedy method. That is, if it is player white's turn, he will pick the action which leads to a state S_{t+1} which maximizes $\hat{v}(S_{t+1}, \mathbf{w}_t)$, its approximate probability of winning, while if it is the other player's turn, he will pick the action which minimizes $\hat{v}(S_{t+1}, \mathbf{w}_t)$.

Whoever the player is, the neural network is used as an aid to make a choice of an action, and this is why we say that the agent learns by self-play.

4.2 Improving the Agent's Playing

So far, we have seen the techniques and algorithms to learn how to play backgammon with self-play from a theoretical point of view. Assuming this works perfectly and we get to an optimal policy, then we are done since the agent always plays the optimal moves. Unfortunately, the policy to which the method described converges might in fact only be an approximate optimal policy. Indeed, recall from section 4.1.2 that we minimize a loss function in order to learn the weights of the neural network, but there might be many local minima and it is not impossible to be stuck at these minima. In such cases, the policy learned is not fully optimal. Are there any algorithms that could be used in order to improve the agent's decision making after it has been trained? It turns out that the answer is positive, and we present below two algorithms that leverage tree search and operate on the tree of game states. Figure 3 depicts such a tree in a game where at each state there are three possible actions.

4.2.1 Expectiminimax

The first algorithm that has the potential to improve the agent's playing is called the *minimax* algorithm. Here in particular, a variant called expectiminimax is used.

The minimax algorithm is a simple strategy for decision making in two-player games in order to find an optimal move, assuming that the opponent also plays in an optimal way. In order to make decisions, a value is associated to each state in the game, computed by an evaluation function which can be arbitrary. In our case for example, the function assigning scores to states will simply be the value function that the neural network estimates.

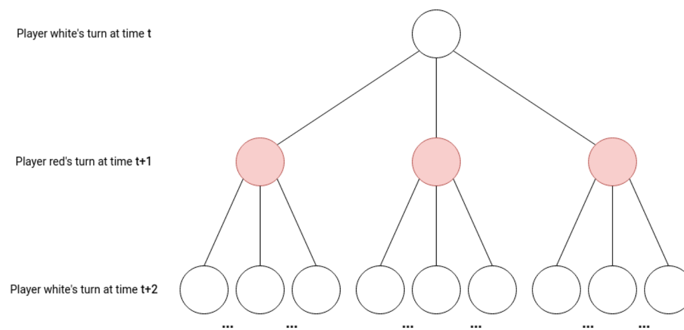

Figure 3: A tree showing the game states with the different players' turns

The minimax strategy then works as follows: One of the player always aims to maximize the score while the other aims to minimize it. For example, suppose that player white has to play. She wants to pick the next move which brings the maximum value. However player red plays also optimally, so she will make the next move so as to minimize player white's score. Let us show an example where at each state of the game there are three possible actions (A_1, A_2 and A_3), and we are at a state of the game where it is player white's turn and the game will be in a terminal state in two moves.

Figure 4 shows such a situation with the value of terminal states, and how the minimax strategy works. Suppose we are player white and it is our turn (root of the tree). We want to choose an action so as to maximize our score in the end. However, the next move is made by the opponent, who will make sure we get the worst possible score. This translates to player red always picking the action which leads to a terminal state with minimum value. In the figure, if player white takes action A_1, player red will then chooses action A_2 so as to minimize the score and get -1. The same applies when player white picks actions A_2 and A_3, and in these cases player red will choose actions which lead to scores 2 and -3, respectively. Thus, the optimal move for player white is to pick action A_2 which gives the maximum score of 2.

This example showed how the algorithm works when two considering two consecutive moves, but this can be extended to an arbitrary number of moves. At each level of the tree, one of the player always maximizes the score, while the other minimizes it. Such a technique is very powerful, but also quite costly. Indeed, even in a simple game with only three possible actions at each states, the number of states to be evaluated grows exponentially with the depth of the search. In practice, one often cuts the search at a certain depth, even though the chosen move may not be the most optimal one.

This algorithm is powerful, but how is it useful in a game like backgammon, where the possible actions depend on the roll of the dice, which obviously implies that there is randomness? The variant called *expectiminimax*, almost identical to the simple algorithm, is used to solve the problem. Here, randomness is included in the tree through *chance* nodes, whose children are states that can result from the various possible random events (see figure 5).

The algorithm works in the same manner as minimax, except that now the value of a chance node is computed as being the average of its children. The rest is exactly the same, each of the player has either the role of a maximizer or minimizer.

There exists ways to improve the running time of the minimax algorithm, notably by using the fact that one does not necessarily need to visit all states in the tree in order to make a min- or max-decision. For more information, the reader can refer to chapter 5 of [9], where the *alpha-beta* pruning technique is explained.

4.2.2 Monte-Carlo Tree Search

The downside of using a method such as minimax is that one needs an evaluation function to score the states. But as this function is arbitrary and chosen by us, there is no way to ensure it is actually a good way to estimate if a state is advantageous. Engineering a good evaluation

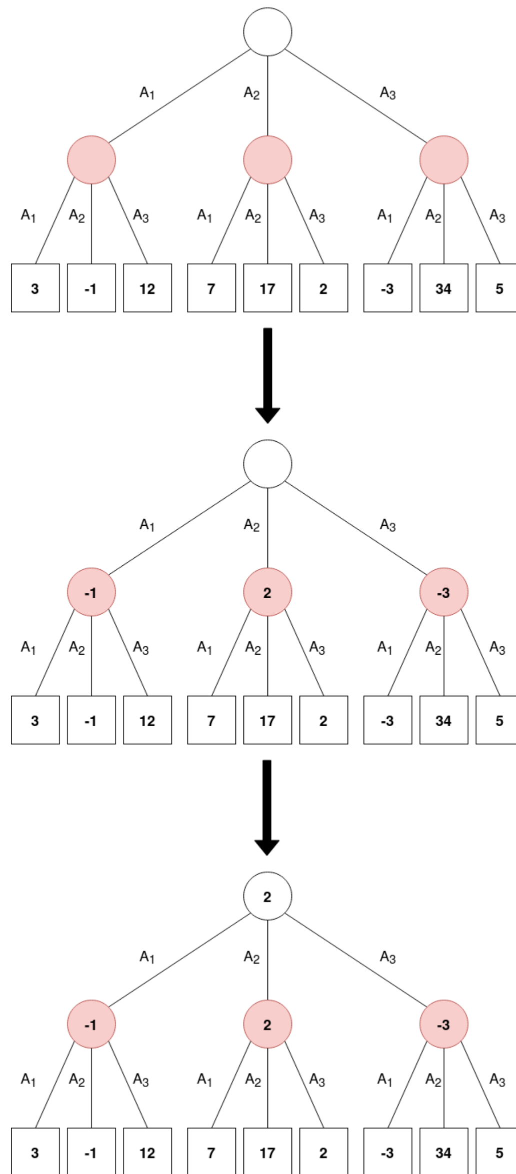

Figure 4: An example of how the minimax strategy works

function often requires a lot of thought and knowledge, not only from the game's perspective, but also knowledge from psychology for example.

However, one thing that can be leveraged to evaluate whether a state is good is randomness. The *Monte-Carlo tree search* (MCTS) algorithm searches the tree of game states (slightly different from before) to find a path which leads to a highest value state, without a direct need for an evaluation function. Indeed, suppose we are at a certain state and that we play thousands of random games starting from that state, and we notice that we end up winning 90% of the time. Then there is a high chance that this state is a good one to be in, since we end up winning in many cases. MCTS exactly does this kind of Monte-Carlo simulations in order to estimate how good a state is.

More formally, the algorithm start with a tree which only has a root representing the current state. It then proceeds by iterating many times over the following steps:

1. **Selection**

   Start from the root of the current tree, and traverse the tree to select a leaf node using a

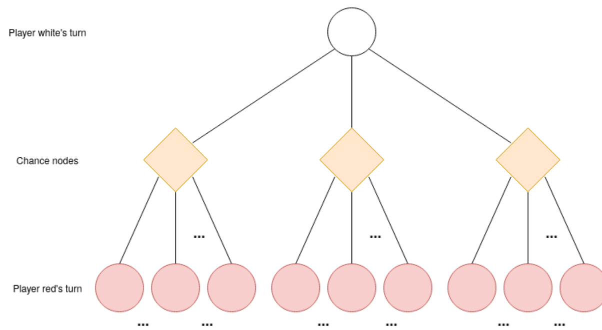

Figure 5: A tree when performing the expectiminimax algorithm

certain "tree policy".

2. **Expansion**

   The tree is expanded from the leaf node previously selected by adding a child node (if possible) reached from the selected node via an unexplored action .

3. **Simulation**

   Starting from the child node added in previous step, the algorithm now simulates a game until the end following a "rollout policy". This gives a score which depends on which player wins the game.

4. **Backup**

   The score is "backpropagated" up the tree by updating the value of each node that lead to the node from which the simulation started from.

These four steps are also shown visually in figure 6, originally figure 8.10 in [13].

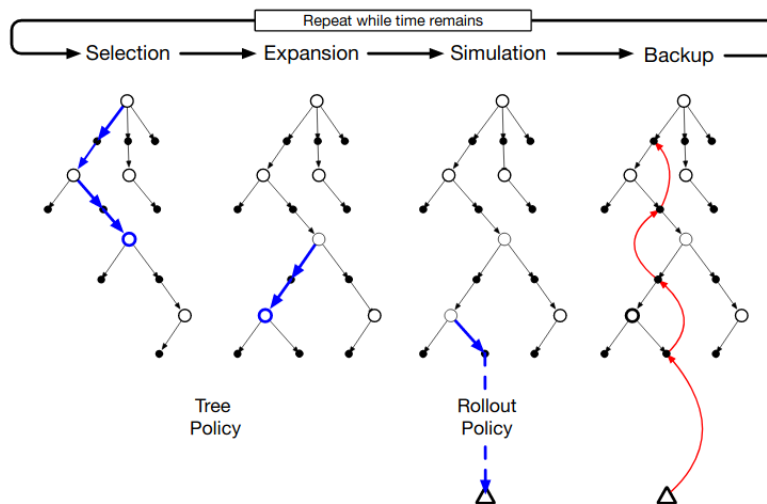

Figure 6: The four main steps on which the Monte-Carlo tree search algorithm iterates over.

Concretely, each node contains two variables that we will call w (number of wins) and n (number of visits), indexed by the iteration number i the MCTS is currently in. In the original form of MCTS, the tree policy used in the selection phase when traversing the tree is the *Upper Confidence Bound* (UCB) selection method adapted to tree [6]. The UCB algorithm simply gives a formula to give a score to a new state, and we then choose the one with highest score. The formula to compute the score for a node is

$$\frac{w_i}{n_i} + C\sqrt{\frac{\log(N)}{n_i}},$$

where N is the number of visits to the parent node, and C is a parameter controlling how much exploration we do.

In the simulation phase, the rollout policy used is to simply choose moves at random until the end of the game is attained. After the simulation phase, we reach the end of a game so we can give score 1 if the player running the MCTS has won, or 0 otherwise (just like the reward when training the neural network). This value is then backpropagated to all the nodes that lead to this terminal state simply by incrementing the w and n value for each of these nodes. This in turn updates the score given by the UCB algorithm in the selection phase for the next iteration of MCTS.

The nice advantage using MCTS is that time is not a restricting factor, in the sense that the process can be stopped at any time and we can return the best state to choose based on the current values. This allows in particular to adapt the number of simulations to the hardware that is at our disposal.

5 Implementation and Results

In this section, we describe concretely how the agent is implemented and what its performances are. We then present a few ways to potentially improve the learning and compare the learning with the simple agent.

To implement the agent, we used the Python [1] programming language as this is simple-to-use and efficient enough for our purposes. As a starting point, we used the code from *https://github.com/dellalibera/gym-backgammon* and *https://github.com/dellalibera/td-gammon*, which already implements the rules of backgammon in Python. We then built on it to implement the various agents and methods.

5.1 Implementation Details

5.1.1 Encoding of a board configuration

Recall that we use a neural network to evaluate a board position, however the neural network cannot take the board as is since it has an input layer with a certain number of units. So we should find a way to translate such positions into a sequence of numbers to be inputted to the network.

To encode a board configuration, we used the same method as Tesauro in his original TD-Gammon. The backgammon positions are in fact converted to a vector of numbers with 198 entries as follows:

- For each of the 24 points on the backgammon board, 4 numbers are used to encode the number of pieces of a certain color:
  - If there is only one piece then the four numbers are set to $[1, 0, 0, 0]$.
  - If there are two pieces then the four numbers are set to $[1, 1, 0, 0]$.
  - If there are $n \geq 3$ pieces then the four numbers are set to $[1, 1, 1, (n-3)/2]$.

  This accounts for a total of $24 \cdot 2 \cdot 4 = 192$ numbers.
- Two numbers are used to encode the number of white and black checkers on the bar. If there are n white checkers on the bar, the corresponding number takes value $n/2$.
- Two numbers are used to encode the number of white and black checkers already removed from the board. If there are n white checkers borne off, the corresponding number takes value $n/15$.

- To encode which player turn it is, two numbers are used in the following way:
  - If player white is playing, the two numbers are set to $[1, 0]$.
  - If player black is playing, the two numbers are set to $[0, 1]$.

Thus, the total number of numbers used to represent a board configuration is $192+2+2+2 = 198$. The encoding is done in the following way first because it is straightforward and the representation is still compact. Also, note that with the given choices, all numbers are all roughly scaled between 0 and 1.

5.1.2 Neural Network

The neural network used in the most basic version of TD-Gammon just has one hidden layer containing 40 units. For each hidden units and the output unit, it performs the weighted sum of its inputs and this result then goes through the *sigmoid* function. More formally, given the input vector S_t, the kth hidden unit $h(k)$ has output

$$h(k) = \sigma\left(\sum_{i=1}^{198} w_{ik}[S_t]_i\right),$$

where $\sigma(x) = \frac{1}{1-e^{-x}}$ is the sigmoid function, w_{ik} is the weight of the connection from input i to hidden unit k, and $[S_t]_i$ is the value of the ith input unit.

The same applies for the computations from hidden units to the output unit. Figure 7 depicts such a neural network, where a blue unit indicates that the application of a sigmoid is performed.

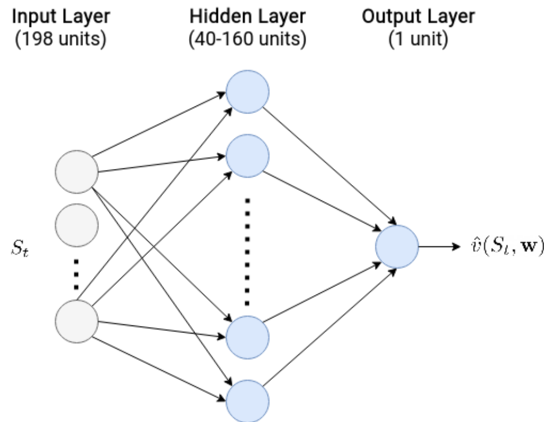

Figure 7: A typical neural network with a single hidden layer used in TD-Gammon (blue units apply the sigmoid function to their output)

Since the sigmoid "squashes" its input between 0 and 1, the output of the neural network can be interpreted as a probability. With the choice of reward mentioned in section 4.1.3, the output in this case can be interpreted as the probability of player white winning.

All the weights w_{ik} can in fact be collected into a big vector $\mathbf{w}$, which enables us to write the output of the neural net as $\hat{v}(S_t, \mathbf{w})$ given input S_t. These weights are what the algorithm "learns" when aiming at minimizing the loss function encountered in section 4.1.2.

In order to train the network and update the weights, the backpropagation algorithm [8] is used to compute the gradient $\nabla\hat{v}(S_t, \mathbf{w})$.

5.1.3 Methods in Modern Reinforcement Learning

As more and more reinforcement learning agents are implemented throughout the years, new techniques are used to potentially improve the learning. We briefly explain here two of these ideas.

The first one, called *opponent sampling*, is used in [2]. In that paper, they show that the skill of opponents encountered during training can have significant impact on the learning of the agents. In our case in the original setup, the agents always play using the most recent version of the neural network, but this can in fact lead to imbalance in training. Instead, they found that training using random old versions of the neural network works much better. As such, during training, we periodically store the current version of the neural network so that later on they can be used to be played against the most recent version of it.

The second improvement comes from [10], where in order to ensure that the playing always generates the best qualit data, each neural network checkpoint is evaluated against the currently best performing network. If the new network is able to win by a margin of $> 55\%$, then the training continues from this new network, otherwise the weights are reverted to the previously best network and the training restart from there.

5.2 Results

In the following, we present the results of various experiments. We benchmark the models in two different ways.

The first one uses a program called GNUBG, which implements different "levels" of backgammon play. For our purpose, we will compare to the highest level which is supposed to play at a world-class level.

The second way of getting useful insights is by comparing new models to some other model that is used as a baseline. Here, the baseline is a TD-Gammon model with 80 units and trained for 1'500'000 games with parameters set to $\alpha = 0.1$ and $\lambda = 0.7$.

5.2.1 Experiment 1: Recreating TD-Gammon

The first experiment consists in recreating the original TD-Gammon and see how it performs against the GNUBG program.

The parameters are set to $\alpha = 0.1$ and $\lambda = 0.7$ just like Tesauro [15] did.

We compare the two basic models, which contain 40 and 80 units in the hidden layer, for different numbers of training games. We evaluate the models by making them play against GNUBG for 1'000 games. Result are displayed in table 2.

Hidden Units	Training Games	Winning Ratio
40	50'000	12.20%
80	50'000	12.50%
40	250'000	20.50%
80	250'000	20.70%
40	800'000	22.30%
80	800'000	22.70%

Table 2: Winning ratios depending on the number of units and number of training games

We observe first that even with a low number of training games like 50'000, the agent is still able to learn pretty well since it can defeat a world-class level player $\sim 12\%$ of the time. As the number of training games increases, so does the winning rate, but the more we train the less important the improvement is. Interestingly, we note that the difference between 40 units and 80 units is not that big.

5.2.2 Experiment 2: Changing the α and λ parameters

In his original paper, Tesauro found after some tuning that the learning rate $\alpha = 0.1$ and the trace decay rate $\lambda = 0.7$ were good parameters. In the following, we compare the learning plots of agents for different values of these parameters.

For α, we train with values 0.1 and 0.2 on a 80 units neural network. For λ, we train with values 0.7 and 0.1 on a 160 units neural network. Both results are shown in figures 8 and 9.

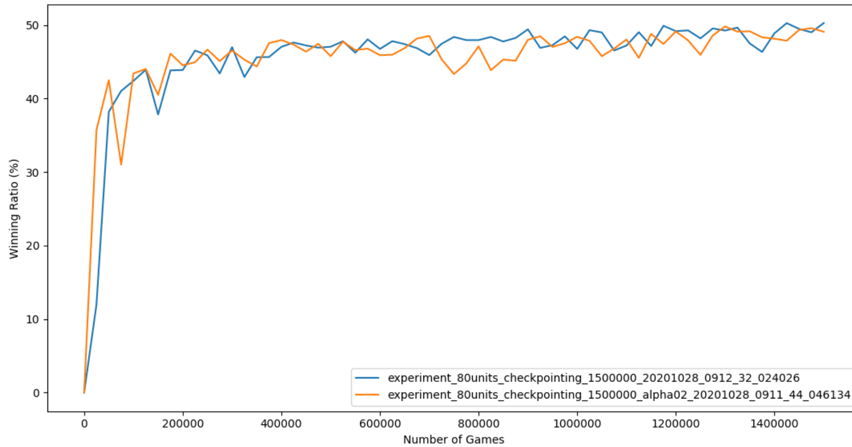

Figure 8: Learning plot for $\alpha = 0.1$ (in blue) and $\alpha = 0.2$ (in orange).

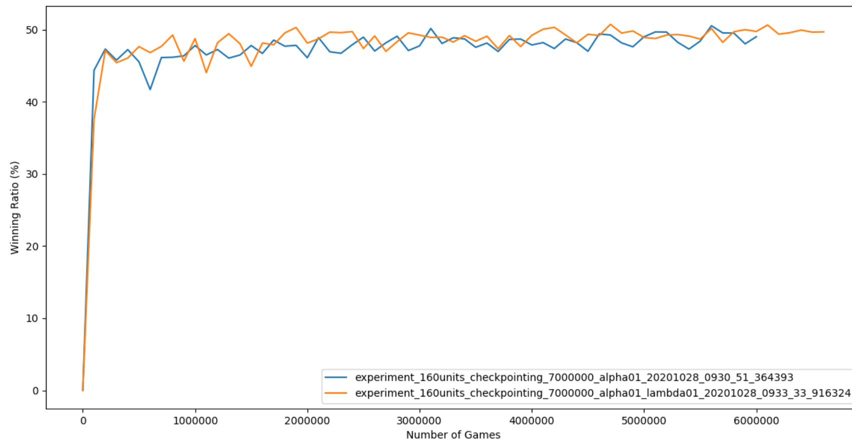

Figure 9: Learning plot for $\lambda = 0.7$ (in blue) and $\alpha = 0.1$ (in orange).

In the case of α, we can observe that the higher value of alpha (0.2) tends to improve the learning at the beginning, but soon after both learning curves look pretty much the same. In fact, they achieve about the same result in the end.

In the case of λ, the learning curves are also very similar. This is not entirely surprising, as other implementations observe the same learning behavior regardless of the λ value.

5.2.3 Experiment 3: Adding more layers

Does the number of layers affect the agent's learning? Intuitively, it would make sense that adding more layer, and thus more units in the neural network will lead to a more complex and thus potentially more accurate approximation of the value function. We compare here the learning process when using hidden layers with 40 (trained for 1'500'000 games) and 80 units (trained for 3'000'000 games). Each time, we compare the simple version with just one hidden layer with a version with two hidden layers. The training is otherwise done as usual with the usual parameters' values.

For the 40 units case, the results in figure 10 do not seem to show a big difference between the two methods. It even seems that the 2-layer neural network is slightly behind in terms of winning ratio. However on the last training games, both method reach the same winning ratio, and it seems that the 2-layer neural network might start getting better from there.

Thankfully, for the 80 units case, the number of training games is higher, so the difference can be better seen. In fact, we see from figure 11 that in this case, the 2-layer neural network

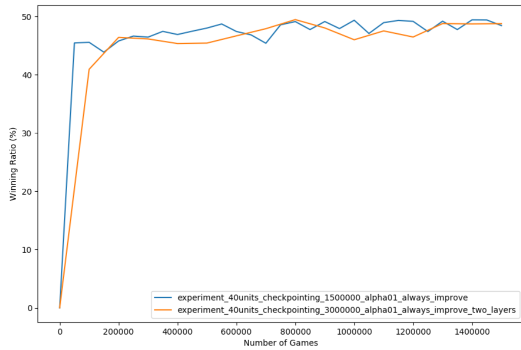

Figure 10: Learning plot with 40 units in the hidden layers for a 1-layer neural network (blue) and a 2-layer neural network (orange).

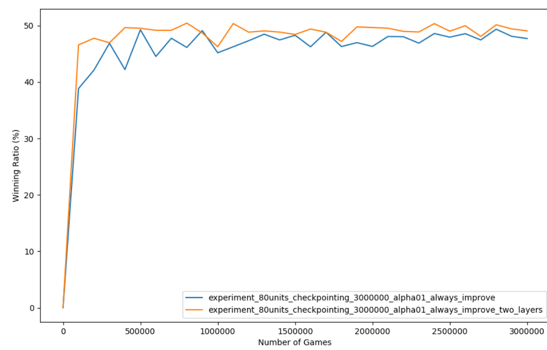

Figure 11: Learning plot with 80 units in the hidden layers for a 1-layer neural network (blue) and a 2-layer neural network (orange).

consistently performs better than the simple version, by a margin of $\sim$ 2-3%.

So for a high enough number of training games, adding layers seems to improve the performance of the agent. It would be interesting to dig deeper in this direction to test whether more complex neural network architectures can achieve an even better result.

5.2.4 Experiment 4: Adding more neurons

In section 3.2, we have seen the various versions of TD-Gammon implemented by Tesauro, and how the number of units used increased with the versions. This suggests of course that more units in hidden layers imply better performances. We verify this in this experiment, with results depicted in figure 12. Here, we compare when using 40, 80 and 160 units when training for 3'000'000 games.

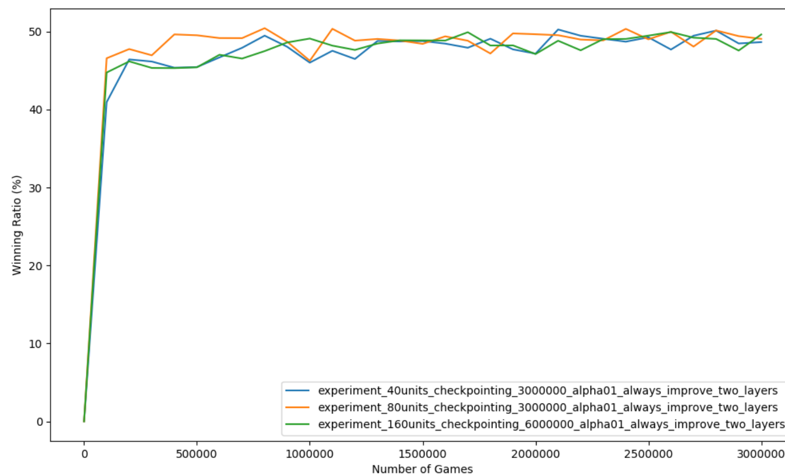

Figure 12: Plot of winning ratio when using 40 units (blue), 80 units (orange) and 160 units (green).

We can observe that in the first million of games, the neural network with 80 units performs significantly better than the others. However after that, the three variants seem to more or less get the same winning ratio, with no clear "winner" in that case.

This suggests that using a neural network with 80 units in the hidden layer should be sufficient and already brings really good results. The advantage is that it is faster to train compared to the 160 units version.

5.2.5 Experiment 5: Using opponent sampling and constant improvement

In this experiment, we want to determine whether the techniques presented in section 5.1.3 actually impact the learning process in the first training games.

We thus train a simple 40 units neural network for 100'000 games and compare the learning curves when using opponent sampling, constant improvement, both and none of them. More precisely, for opponent sampling we sample a random old neural network every 2'000 games, and we always keep track of at most 20 random old neural networks. For constant improvement, we compare the current network with previous best every 5'000 games, and keep the network if it wins by a margin of 55%. Moreover, if after reverting the learning two times already, we do not revert anymore and continue the learning until we reach another 5'000 games. Results are shown in figure 13.

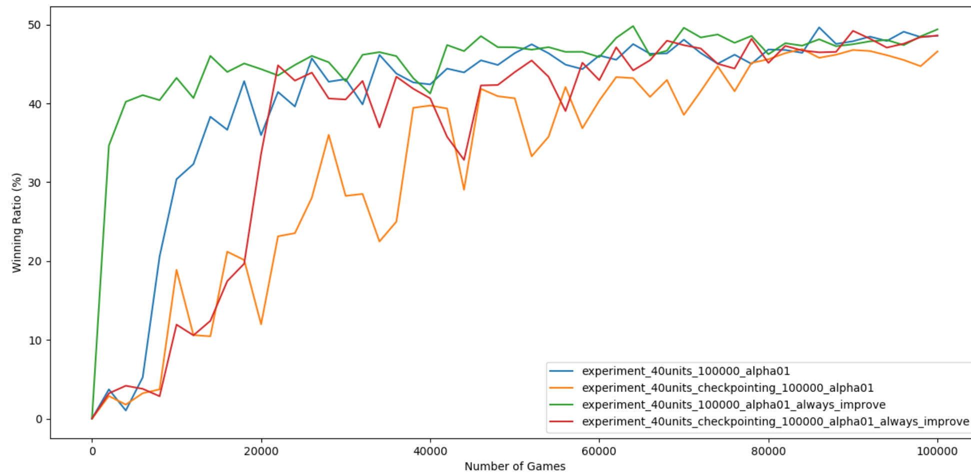

Figure 13: Learning curves when using the "modern methods". There are four curves: TD-Gammon original (blue), with opponent sampling (orange), with constant improvement (green) and with both (red).

The results here are really interesting as we observe that opponent sampling seems to actually worsen the learning at the start! Indeed, opponent sampling alone yields the worst curve, followed by opponent sampling combined with constant improvement.

The green curve indicates that constant improvement clearly has a positive impact on the learning of the agent. It is able to reach much higher winning ratio in much less iterations than any other curve. However after about 50'000 training games, there does not seem to be much difference between using constant improvement or not.

5.2.6 Experiment 6: Using Expectiminimax and Monte-Carlo tree search

This experiment aims to see whether using tree search techniques during play improves the agent's winning ratio against the GNUBG program. We compare the agent's performance when using no specific technique, when using the expectiminimax algorithm, and when using the Monte-Carlo tree search algorithm.

For the expectiminimax algorithm, we use a 2-ply search just like Tesauro used and prune the tree using alpha-beta pruning. For the MCTS algorithm, we simplify it in order to reduce the compute time. As such, we only run 500 simulations and when running a simulation, we do not go all the way until the end of the game but we stop at depth 5 and instead of returning the final reward we return the value of the state given by the neural network.

Note that MCTS is a lot more reliable when we can run thousands of simulations, so we can already expects its results to not be that incredible. Also, the number of games we use to evaluate the agent is 250 as the running time is particularly high when using tree search methods.

For the sake of having multiple examples, we show in table 3 the results for the 40-unit and 80-unit neural networks trained for 250'000 games already encountered in experiment 1.

Hidden Units	Simple	Expectiminimax	MCTS
40	20.50%	28.80%	21.80%
80	20.70%	29.10%	21.60%

Table 3: Winning ratio depending on the number of units and whether a tree search method is used

We directly observe how the expectiminimax improves the agent's performance, by almost 10%. This is verified for the two neural network considered. As for MCTS, it also brings a little bit of improvement (about 1%), but nothing much compared to the other algorithm. It would be interesting to use a lot more simulations during MCTS so that the full power of that method could be leveraged.

5.2.7 Experiment 7: Using Expectiminimax while learning

It is noted in [3] that in some games like chess or shogi, the minimax algorithm is an important to train an agent which achieves good performances.

We already mentioned minimax (and expectiminimax) previously but it was used only after training, as an aid to make better decisions. Here however, the idea would be to use the algorithm to make choices during the training time.

The only complication is the following: running the expectiminimax is costly, so much that it becomes a problem. Indeed, during the first training games since the neural network makes random moves, a game can last for thousands of moves, whereas a usual backgammon game lasts for about 60 moves.

To overcome this, we decide to train the network as usual for the first $10'000$ iterations, but then use the expectiminimax approach for the next iterations. We train a 40-unit neural network for $100'000$ iterations, results are shown in figure 14

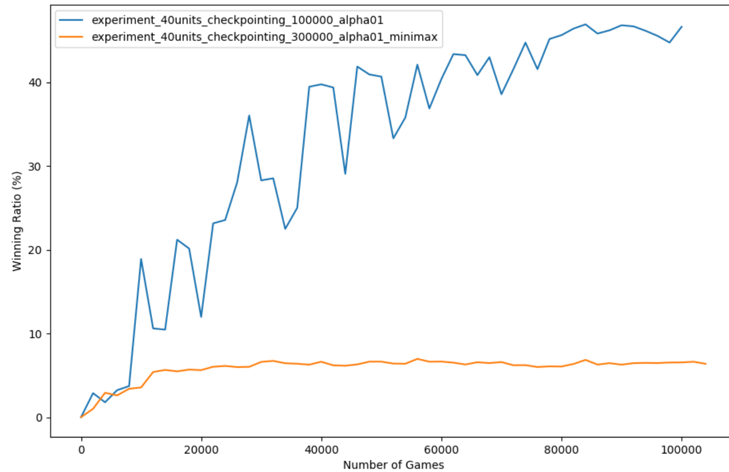

Figure 14: Learning curves when using the expectiminimax algorithm during training (in orange) and without using it (in blue).

Unfortunately, results are not as expected. The training when using expectiminimax starts as usual but very quickly stagnates at around 8% for the percentage of wins.

In fact, this is probably due to the fact that at the $10'000$th iteration, the method changes so that it uses expectiminimax to choose moves. This somehow brings a huge imbalance when training, where one of the player ends up always winning while the other always loses, which prevents learning from happening. It is not clear why this behavior kicked in, but the only way

to get satisfactory results would be to train only using expectiminimax decisions, even during the first random games. However this would take months to train with the current implementation and so we do not go further in that direction.

5.2.8 Experiment 8: Best performing model

With what we learned from the experiments, we can conlude the following:

- For the choice of parameters, keeping those found by Tesauro yields a good training. Here, we decide to train the agent for 2'000'000 games.
- Using two layers instead of one brings slightly better results.
- Using 80 units usually brings the best results.
- Using the "constant improvement" technique seems to help during training, while "opponent sampling" does not.
- When playing, using the expectiminimax algorithm improves the agent's performance by $\sim 10\%$.

Recall also that so far, the best performing agent against GNUBG is the one from experiment 6, which is able to win 29.1% of the time.

A comparison of the performance against GNUBG over 250 games when enabling expectiminimax or not is given in table 4.

Simple	Expectiminimax
22.70%	32.10%

Table 4: Winning ratio

We see that even though there is not much change when we do not use expectiminimax, using it allow to reach a winning ratio of 32.1%, the best overall!

In figure 15, we compare the learning curve of the best agent with another agent that uses an 80-unit neural network, but without all the tweaks.

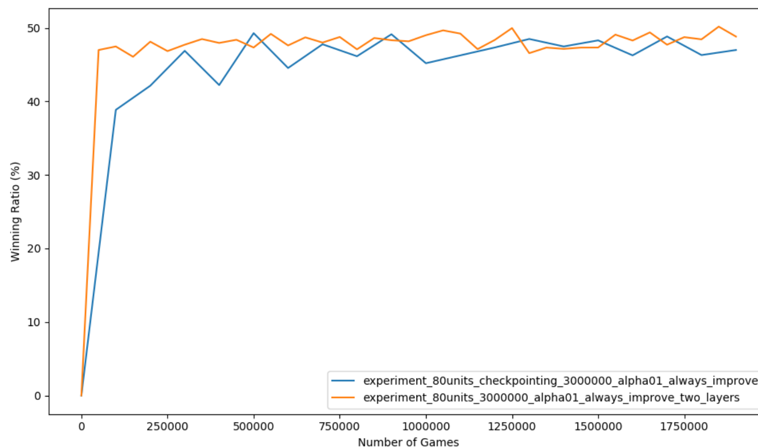

Figure 15: Comparison between best agent and another agent using 80-unit neural network

We can definitely see how right from the start, the learning happens much faster for the best agent, and the winning ratio is almost always consistently higher than the more simple agent.

Even though this shows how one can improve the performance of an agent quite a lot by selecting the right techniques, there is still a lot more that can be done to improve the program further.

5.3 Future Improvements

There are of course a myriad of things one could try to potentially improve the agents playing. What we have concluded from the experiments is that whatever learning strategy we decide to use, it always tend to achieve the same results in the end, with no method that has a clear advantage. Some actually do speed up the learning as we have seen, but it does not mean that they allow to reach a better level of play in the end. Thus, we present below two types of improvements: trivial and non-trivial.

Trivial improvements are easy to make (and most of them were implemented but not presented in this report), but they do not actually improve much or at all the agent's performance. These includes:

- Using more units in the hidden layer of the neural network.
- Using more hidden layers in the neural network.
- Using different activation functions in the units.
- Using different architectures for the neural network (convolutional neural network for example).

Examples of techniques that yield clear improvements of the agent are the use of the expectiminimax algorithm or Monte-Carlo tree search. However, other more complex ideas could be used, such as

- Performing MCTS not only when playing after the training, but during the training (as in [10]). We would then be able to use a different loss function involving the result of MCTS at each move. However, this would require to parallelize many of the computations in order to efficiently use MCTS, which was unfortunately undoable in the restricted amount of time we had.
- Training and using different neural networks depending on what the state of the game is. Indeed, as there are different strategies to use depending on the game's state, a single neural network may not capture all the best moves. However, using many specialized neural networks could yield substantial improvements.
- Searching in more depth the game tree when using the expectiminimax algorithm. This would require some kind of smart heuristic to prune parts of the tree since the number of states to consider could be huge. Some research in this directions has been done in [4].

These are only a few ideas, but one could also think about the learning algorithm itself. Is TD learning the best way to approach this problem?

Indeed, it is known that TD learning is not able to achieve as much success with other games such as chess or checkers. So it seems that there is something particular about backgammon that makes it "learnable" in a way that is quite unique. In [7], they show that some of the success of TD-Gammon was replicated without the use of backpropagation or TD learning, but simply using hill-climbing in a relative fitness environment. That is, they would still use a neural network, but they would make it play against a "mutated" version of it by adding Gaussian noise to the weights. If the mutant is able to win more than 50% of the time, then it is selected for the next generation, and so on. It thus seems that the success of Tesauro's program had more to do with the co-evolutionary structure of the learning task and the dynamics of backgammon itself.

But one could then ask, is there a method that would lead to a faster and more robust learning? One could then try plenty of methods and compare them to hopefully get concluding empirical results. It is also interesting to note that in order to improve TD-Gammon, Tesauro used expert features. But would it be possible to reach the same level of play only with self-play?

6 Conclusion

In this project we presented and reviewed the algorithm used to teach an agent how to play the game of backgammon through self-play. In the first three sections, we introduced the basic notions of reinforcement learning and defined what the goals were. With an idea of what backgammon is, we were able to state the goal of the agent as maximizing an expected reward. Moreover, we saw how a neural network can be helpful to approximate the value function that describes the utility of the states.

Section 4.1 was dedicated to explaining how TD Learning combined with a neural network was used to approximate the value function. We saw how simple the update rule turns out to be, and in particular how the TD(λ) algorithm gives a generalization of the simple TD(0) update.

In section 4.2, we presented how one could use various techniques when playing to make better predictions after training. In particular, we introduced the expectiminimax algorithm as a way to take into account the possible future opponent's moves and still maximize our reward in the worst-case scenario. We also saw a different idea which uses sampling at its core, the Monte Carlo tree search. It allows to search deeper in the tree of possible moves which can sometimes give good results.

Finally, the implementation presented in section 5 gave insights about the correctness of the entire learning procedure. By comparing the learning curves of various models, this allowed to gauge which architectures or techniques yield the best results. It is also in this section that we presented a few implemented ideas that come from the famous AlphaGo program like "constant improvement". It turns out that even though some techniques really speed up the convergence of the procedure, other methods like opponent sampling do not seem to have such a positive impact. In any case, there is plenty of room for improvement as we noted. One could go in many directions, taking inspiration not only from recent methods but also old ones that have proven to be effective. As more and more methods are discovered in the field of reinforcement learning, this gives the opportunity to constantly build on these programs and get closer to an optimal learning.

Acknowledgments

I would like to first thank Rüdiger Urbanke for helping me throughout the project. It was a lot of work at first to get a good understanding of the game of Backgammon and the TD algorithm, and he helped me push myself to better understand the core concepts. I would also like to thank my friend Damian Dudzicz, who gave me useful hints and helped me understand the basics of reinforcement learning as I was new to the field. Finally, I acknowledge the work of Alessio Della Libera who created a very simple-to-use Python code implementing the rules of Backgammon which can be found at *https://github.com/dellalibera/gym-backgammon*.

References

[1] Python Software Foundation python language reference version 3.7. http://www.python.org.

[2] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition. 2018.

[3] Donald F. Beal and Martin C. Smith. Temporal difference learning applied to game playing and the results of application to shogi. *Theoretical Computer Science*, 252(1):105 – 119, 2001.

[4] Thomas Hauk, Michael Buro, and Jonathan Schaeffer. *-minimax performance in backgammon. In *Computers and Games*, pages 51–66, Berlin, Heidelberg, 2006. Springer.

[5] Sergey Ivanov and Alexander D'yakonov. Modern deep reinforcement learning algorithms. *arXiv:1906.10025*, 2019.

[6] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer.

[7] Jordan B. Pollack and Alan D. Blair. Why did td-gammon work? In *Proceedings of the 9th International Conference on Neural Information Processing Systems*, page 10–16. MIT Press, 1996.

[8] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Representations by Back-Propagating Errors*, page 696–699. MIT Press, Cambridge, MA, USA, 1988.

[9] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, USA, 3rd edition, 2009.

[10] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[11] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Mach. Learn.*, page 9–44, August 1988.

[12] Richard S. Sutton. Implementation details of the td(λ) procedure for the case of vector predictions and backpropagation. *Tech. Rep.*, 1989.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, second edition, 2018.

[14] Gerald Tesauro. Neurogammon: a neural-network backgammon program. In *1990 IJCNN International Joint Conference on Neural Networks*, pages 33–39 vol.3, 1990.

[15] Gerald Tesauro. Practical issues in temporal difference learning. *Mach. Learn.*, 8(3–4):257–277, May 1992.

[16] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.