Protein Folding Problem : A Reinforcement Learning Approach

Abstract : In this paper we tackle a simplified version of the protein folding problem, framed as a combinatorial optimization task. We propose a reinforcement learning based approach using the Q-Learning algorithm and show that, for protein sequences of moderate length, this method works well. We then analyze the effect of varying different hyper-parameters on the result. Lastly, we extend our Q-Learning algorithm using a neural network to approximate Q values, a technique known in the literature as Deep Q-Learning (DQN).

Index Terms : Protein Folding, Reinforcement Learning, Q-Learning, Combinatorial Optimization.

Author : Samy TAFASCA

I. Domain and motivation

1. Introduction

Proteins count as one of the major classes of biological macromolecules. They perform various tasks that are necessary for the body to function correctly. This includes regulating pH levels (balance between acids and bases), providing structure or even acting as a messenger between cells and organs. They are created as a linear sequence of amino-acids that folds in the space to form a three dimensional structure. Given their vital role, one can see them as a basic unit of life, and developing a deeper knowledge of their structure and functions can help researchers gain a better understanding of living organisms.

As soon as it is synthesized, a protein folds rapidly until it reaches a stable three dimensional structure referred to as the *Native State*. Once this stable form is reached, the protein can carry out its functions normally. The problem we aim to tackle in this work is called the Protein Folding Problem, where the goal is to predict the topological structure of the Native State, given the sequence of amino-acids of a protein. This was proved to be an NP-complete optimization problem. When solving it, we often assume that the protein folding process obeys the Thermodynamic Principe : the Native State is characterized by a minimum level of Gibbs Free Energy, and the information required to determine its structure is entirely encoded in the sequence of amino-acids [1]. This assumption implies the existence of a well defined function that maps a sequence of amino-acids onto its corresponding native state. Hence, our purpose is to approximate evaluations of this function.

The prediction of the 3D structure of a protein remains one of the biggest challenges in bioinformatics. It has attracted a great deal of attention by the research community over the years. This is mainly due to the important implications it has on various domains such as bioinformatics, biochemistry, genetic engineering, molecular biology, medicine, to name a few. This paper particularly focuses on the prediction task in a bi-dimensional space as opposed to the more general three-dimensional case. This simplification helps with the computation, but the methods presented herein can be easily extended to the 3D setting.

2. Conceptual representation

We first begin by introducing a framework that will help represent the protein folding process as a combinatorial optimization task. Combinatorial Optimization is the search of an optimal solution in a well defined discrete space. This is inherently a difficult problem as the search space is usually large, the parameters are linked together and the presence of local minima adds a degree of uncertainty to the optimality of solutions. In this context, we are interested in approaches that help achieve acceptable results using reasonable resources. Reinforcement Learning (RL) and Evolutionary Algorithms (EA) are two example fields that include common techniques used to solve combinatorial optimization problems.

One of the most important classes of abstract models for proteins is the lattice-based class model. In this setting, each amino-acid takes a position on a lattice (or a grid), and the energy of the conformation is computed based on these positions. Therefore, the goal can be reformulated as finding the set of amino-acid positions that minimizes the Gibbs Free Energy of the protein. Many lattice-based models were proposed in the literature, and in this work, we use Dill's Hydrophobic-Polar (HP) model [2]. This model classifies amino-acids based on their hydrophobicity, where two distinct classes are defined :

- Hydrophobic or non-polar (denoted by H) : class of amino-acids that are repelled by water.
- hydrophilic or Polar (denoted by P) : class of amino-acids that attracted to water.

This model relies on the observation that hydrophobicity is responsible for the generation of important forces that steer the folding process to the native state. Consequently, a protein of n amino-acids can be defined as the following sequence : $P = p_1 p_2 p_3 p_4 \dots p_n$ where $p_i \in \{H, P\}$ and $1 \leqslant i \leqslant n$ where we assume that $n \geq 3$. A conformation C of the protein is then represented as the ordered set of positions of the amino-acids on the lattice : $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$. Obviously, the sequence of amino-acids in the 2D space needs to form a path since amino-acids are chained together, but not all conformations constitute a path. For a conformation to be a path, the points on the grid need to form a chain where each two consecutive amino-acids are placed immediately next to one another (horizontally or vertically) on the given lattice. This definition can be formalized as follows :

$$C \text{ is a path} \iff (\forall\ 1 \leqslant i, j \leqslant n,\ |\ i - j\ | = 1 \implies |\ x_i - x_j\ | + |\ y_i - y_j\ | = 1)$$

Finally, a conformation is said to be valid if and only if it is a self-avoiding path. Self-avoidance means that different amino-acids in the sequence have to occupy different positions on the grid

(i.e no collisions in the chain). This definition can be formalized as follows :

$$C \text{ is self-avoiding} \iff (\forall\ 1 \leqslant i, j \leqslant n,\ i \neq j \implies (x_i, y_i) \neq (x_j, y_j))$$

Equipped with this formalism and these definitions, we are ready to dive into the problem.

II. Reinforcement learning

In this section, we aim to define our task from a reinforcement learning perspective in order to be able to use Q-learning to solve it. To do so, we need to represent the problem as a Markov Decision Process (MDP) which provides an abstract mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of the agent. MDPs are meant to frame the problem of learning from the interaction with an environment to achieve a desired goal.

For the purpose of this project, the experimentations carried out in later sections will focus on solving the folding problem for the following protein.

$$P = HPHPPHHPHPPHPH$$

This sequence contains a total of 14 amino-acids, 7 of which are Hydrophobic (H) and 7 are Polar (P). The goal is to find a solution that will minimize the energy of the structure in a 2 dimensional grid. A solution in this case, is an ordered list of coordinates referring to the position of each element of the sequence. At a later stage, we will see how to scale learning to larger sequences.

1. State transition function

There are many ways to define the problem of protein folding as a reinforcement learning task. We present here, a slightly modified approach to the one used in the paper by Gabriela Czibula et al. [3]. For a protein of n amino-acids, a ***State*** is defined as a path of length p where $p \leq n$, representing the positions on the lattice of the first p elements in the sequence.

From the previous definition, we can define an ***Action*** as the transition from one state S_p to the next S_{p+1} where the element $(p+1)$ in the sequence is placed next to the previous one (as a horizontal or vertical neighbor). One obvious action space in this case is the set $\{\text{Left, Up, Right, Down}\}$ which was used in [3]. At a given step, the agent will decide to move in one of the 4 available directions to place the next element of its sequence. This said, we opted for a different Action Space composed of 3 elements $\{\text{Left, Forward, Right}\}$. The main reason behind this representation is to prevent the agent from going back on its track, thus avoiding obvious collisions (such as action Up followed by action Down). This action space also has the advantage of being smaller (size 3 instead of 4), thereby reducing computational and storage requirements during training. However, this representation also raises a few concerns that need to be addressed. Indeed, the idea of going forward implies some form of orientation : the agent needs to be facing a particular direction for the forward move to be meaningful. For example, an orientation to the "Right" would mean the agent will keep moving right if it picks the Forward action, up if it's the Left action and down for the Right action. When the agent has already started the episode, the orientation at a given step can be inferred from the previous actions. However, when the agent is in the initial state, there is no orientation, so one has to be given by the user to get the agent started. One way of going about this, is to pick an initial orientation randomly. This is a viable way to do it, but it suffers from the problem of duplicity of conformations. Figure 1 shows an example of a final state for a Protein defined by the sequence $P = PHHP$. All four conformations correspond to the series of actions $\{\text{Forward, Forward, Left}\}$, but with different initial orientations. For the first conformation on the left, the initial orientation is "Up", for the second one it's "Right" and so forth. In reality, however, these 4 dispositions are equivalent, and refer to the same structure. In order to remove this duplicity, we decided to impose the initial orientation as "Up". This constraint doesn't entirely solve the problem, since a forced initial orientation will still allow 3 out of the 4 equivalent conformations to be reached depending on the first action chosen by the agent. For this reason, we not only forced the initial orientation, but also the initial action as Forward. This means that the initial state of our agent will always be $\{(0,0), (0,1)\}$. By applying these constraints, we have effectively removed equivalent conformations (that are due to the grid's orientation), thereby reducing the number of possible states by a factor of 1/4. In the example of Figure 1, only the first conformation on the left is viable in this setting and will correspond to the series of actions $\{\text{Forward, Left}\}$, or equivalently, the terminal State $\{(0,0), (0,1), (0,2), (-1,2)\}$.

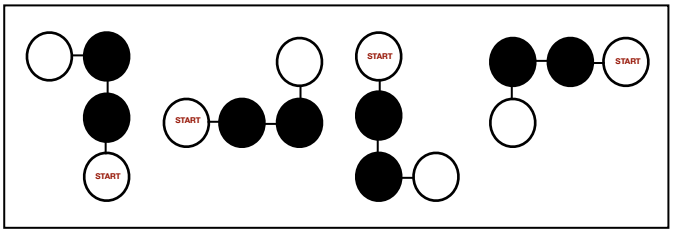

Figure 1 : Four equivalent conformations of Protein PHHP.

When the agent begins an episode, the two first elements of the protein are initially placed on the origin and the upper neighbor at the positions $(0, 0)$ and $(0, 1)$ respectively. At this point, it can pick its next action and place the following amino-acid accordingly. The episode terminates when the agent places all the amino-acids of the sequence on the lattice. Therefore, an episode has exactly $n-2$ steps, n being the number of elements in the Protein (we exclude the first two, since they correspond to the initial state). Table 1 below shows the possible transitions from the initial state.

INITIAL STATE S_0	ACTION a	NEXT STATE S_1
{ (0, 0), (0, 1) }	Left	{ (0, 0), (0, 1), (-1, 1) }
{ (0, 0), (0, 1) }	Forward	{ (0, 0), (0, 1), (0, 2) }
{ (0, 0), (0, 1) }	Right	{ (0, 0), (0, 1), (1, 1) }

2. Reward function

Now that we have our definitions of State Space and Action Space, we move to another important component of the reinforcement learning framework : the reward function. This function seeks to reward or penalize the agent for its actions depending on their contribution to the end goal. But before defining our reward function, we need to introduce the Energy function we wish to minimize, and how it can be computed.

The Gibbs Free Energy in the HP model reflects the fact that hydrophobic (H) amino-acids have a tendency to form a core (unite together in the center of the structure). Consequently, the energy function adds -1 for each pair of hydrophobic elements that

are neighbors (horizontally or vertically) in the grid, but not consecutive in the sequence defining the protein. Formally speaking, for a protein of sequence $P = p_1 p_2 p_3 p_4 \dots p_n$ where $p_i \in \{H, P\}$ and $1 \leqslant i \leqslant n$, we define an intermediate function I such that :

$$\forall\ 1 \leqslant i, j \leqslant n,\quad I(i,j) = \begin{cases} -1, & \text{if } p_i = p_j = H \text{ and } |x_i - x_j| + |y_i - y_j| = 1 \\ 0, & \text{otherwise} \end{cases}$$

The energy function for a valid conformation C is then defined as :

$$E(C) = \sum_{1 \leqslant i \leqslant j-2 \leqslant n} I(i,j)$$

Next, we define our reward function as the following :

- If the transition results in a valid path (meaning, the move does not generate a collision), then the reward received by the agent is 0.1.
- If the transition results in a collision (configuration that is not valid), the reward is 0.
- If the agent reaches the terminal state, and if it has not registered any collisions, the reward received by the agent is $-E(C)$, where C is the conformation achieved by the agent.

Since the state space typically includes terminal states that are not valid, as in, not self-avoiding paths, the role of this reward function is to enforce this constraint and steer the decision making of the agent towards valid solutions by punishing collisions. Moreover, the magnitude of the energy function ensures that the agent seeks the minimum of the energy function. Figure 2 shows an example of a conformation for a given sequence and the rewards gained by the agent at each step. In this case, the solution provided by the agent is the best possible solution, as -2 is the minimum achievable energy for this sequence of amino-acids.

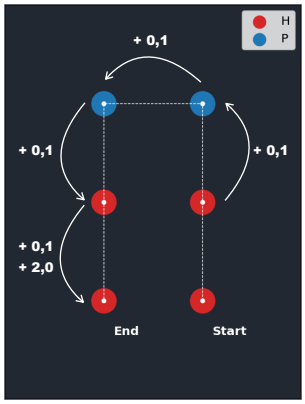

Figure 2 : An example solution for the sequence HHPPHH. At each valid step, a reward of 0.1 is gained. During the last transition, we also add the negative of the energy (which in this case is -2, as we have 2 pairs of H elements that are non consecutive neighbors). Notice during the first transition no reward is gained as the first two elements constitute the Initial State of the agent.

3. Learning Policy

A policy is another core component in reinforcement learning. It defines the learning agent's way of behaving in the environment at a given step. It can be seen as a mapping from the *State Space* to the *Action Space* that the agent relies on to pick the best action available given a perceived state. The psychological equivalent of a policy is the Stimulus-Response associations. Policies can either be deterministic or stochastic specifying probabilities for each action. Solving a reinforcement learning problem roughly translates to finding an optimal policy that achieves the best long term reward.

Learning algorithms can be either on-policy or off-policy. On-policy algorithms, such as SARSA, learn action values, not for the optimal policy but for a near optimal policy that still explores [4]. In Off-policy learning however, such as Q-learning, we distinguish two policies : an *estimation policy* that is learned about which will eventually become the optimal policy, and a *behavior policy* whose purpose is to generate behavior, and as such, is more exploratory in nature.

In this work, we opted for an ε-greedy policy as the learning policy of our agent. In this setting, the agent chooses the **optimal** action (as currently estimated from Q) with a probability $(1 - \epsilon)$, and a **random** action with probability ϵ. This is desirable as the element of randomness encourages the exploration of the state space while the optimal choice of action ensures exploitation of what was previously learned. Since our agent initially starts with no prior knowledge of action values, a good approach is to start with a high value of ϵ (1 for example) such that the agent mostly explores during the first episodes, and slowly decrease its value until some minimum threshold, denoted ϵ_{min}, to ensure the agent is mainly exploiting what it has learned by the end of the episode. The non-null minimum threshold, as opposed to a 0 value for ϵ (which would mean 100% exploitation), is helpful in the sense that it prevents the agent for being permanently stuck in a local minimum. For this reason, we also define a new variable called the *decay rate* λ, which controls the rate by which we decrease the value of ϵ after each episode. Below is the pseudo-code implementing the ε-greedy policy.

1. At the start of learning, set $\epsilon = 1$
2. During each episode, generate a number from a uniform distribution $\beta \sim \mathcal{U}(0,1)$
3. If $\beta \leq \epsilon$, then pick a random action from $\{\text{Left, Forward, Right}\}$
   Else, pick the best action (from the current estimation of Q) : $a_{max} = \arg\max_{a'} Q(s, a')$
4. After each episode, if $\epsilon \geq \epsilon_{min}$ set $\epsilon = \epsilon * \lambda$

4. Graphical Representation

In this section, we will use a toy example given by the Protein from Figure 2 defined by the following sequence : $P = HHPPHH$. It was necessary to pick a rather small sequence in order to keep the graphical representation manageable since the number of states is exponential with the size of the sequence $\mathcal{O}(3^n)$. Figure 3 illustrates the problem visually, where the states have been named incrementally from 0 to 120.

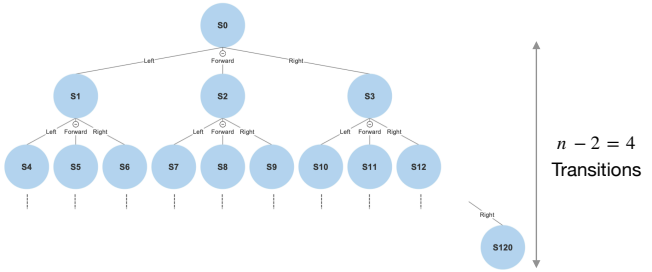

Figure 3 : Graphical representation of the HHPPHH Protein Folding Problem.

The rewards were not displayed in the graph. But for each transition, if it doesn't result in a collision, the agent gains 0.1, otherwise the reward is 0. The transition to the terminal states, if no collisions were registered, rewards the agent with $0.1 - E(C)$, which translates to 2.1 since the minimum energy for this protein is -2.

5. The R Matrix

The R Matrix describes the immediate reward that is available to the agent for taking action (a) when in state (s). The rows of this matrix represent the different possible states of the environment, while the columns denote the actions available to the agent. For the $P = HHPPHH$ protein, there is a total of 121 states (including the initial state). Consequently, the R-Matrix has 121 rows and 3 columns. It is also possible to represent the columns as the next states that will result from taking some action given the current state, but this will yield a 121x121 matrix where, for each row, only 3 values are registered and the rest is unavailable. This latter approach is obviously not efficient, hence our first choice. Table 2 reports the R-matrix for the first few rows (states). The states' identification numbers follow the same logic as in Figure 3. The transitions to the terminal states (not shown) will yield one of four possible reward values : 0 if the last transition results in a collision, 0.1 if the last transition is valid but there was at least 1 collision before, 1.1 if there are no collisions in the conformation and the energy function returns -1, and 2.1 if there are no collisions and the energy function returns -2 (the minimum energy for this sequence). The R Matrix was implemented as a method of the Agent's Class in python, which takes as input the current state and outputs the reward value based on the definition presented before.

Action / State	Left	Forward	Right
S0 = {(0,0), (0,1)}	0.1	0.1	0.1
S1 = {(0,0), (0,1), (-1,1)}	0.1	0.1	0.1
S2 = {(0,0), (0,1), (0,2)}	0.1	0.1	0.1
S3 = {(0,0), (0,1), (1,1)}	0.1	0.1	0.1
S4 = {(0,0), (0,1), (-1,1), (-1,0)}	0	0.1	0.1
S5 = {(0,0), (0,1), (-1,1), (-2, 1)}	0.1	0.1	0.1
S6 = {(0,0), (0,1), (-1,1), (-1, 2)}	0.1	0.1	0.1
S7 = {(0,0), (0,1), (0,2), (-1,2)}	0.1	0.1	0.1
S8 = {(0,0), (0,1), (0,2), (0,3)}	0.1	0.1	0.1
S9 = {(0,0), (0,1), (0,2), (1,2)}	0.1	0.1	0.1
S10 = {(0,0), (0,1), (1,1), (1, 2)}	0.1	0.1	0.1
S11 = {(0,0), (0,1), (1,1), (2,1)}	0.1	0.1	0.1
S12 = {(0,0), (0,1), (1,1), (1,0)}	0.1	0.1	0

Table 2 : Selected rows from the R Matrix.

6. Parameters for Q-Learning

The Q-learning algorithm depends on a few hyper-parameters that were presented before. The ones that were not mentioned, are defined next.

- **Learning Rate** (α) **:** Q-learning is an iterative process where the agent interacts with the environment, gains new information, and incorporates that information into its estimation of the action values Q. In order to do so, at each step, the update rule computes the difference between the Temporal Difference Target (TD-target, which is an estimated return) and the current estimate of the action value. This difference is equivalent to the gradient in gradient descent. We then multiply (α) by this difference and add the result to the current action value estimate. In this way, (α) controls the magnitude of the update and consequently the speed and convergence of the learning process. (α) takes on values between 0 and 1, and can either be fixed or dynamic, usually decaying over time.

- **Discount Factor** (γ) **:** This parameter controls the importance of future rewards in the value function. It usually takes values between 0 and 1 (the upper bound is excluded if the task is continual, included if the task is episodic). The reason for this range is to ensure convergence of the infinite sum of future rewards used to compute the expected *discounted return*. If the value is 0, the agent is "myopic" in the sense that it only tries to maximize immediate rewards. When (γ) approaches 1, the agent becomes more "farsighted" in that it assigns higher weights to future rewards.

For the first run of our Q-learning algorithm, we had to manually set the values for each of the hyper-parameters. Before doing the experiments however, it is difficult to set the right values as they are usually problem dependent and require proper tuning. That said, we can default to values that are recommended in the literature. In reality, we also ran the algorithm a few times with different combinations and retained the one that seemed to work best.

Parameter Name	Parameter Symbol	Parameter Value
Learning Rate	α	1.0
Discount Factor	γ	0.9
Exploration Factor	ϵ	1.0
Decay Factor	λ	0.999925
Minimum Exploration Factor	ϵ_{min}	0.05
Policy	π	ϵ-greedy

Table 3 : Default Parameters of Q-Learning.

Henceforth, unless stated otherwise, parameters for the Q-learning algorithms will take the default values presented in Table 3. In order to run our experiments, a few more parameters relative to the training procedure needed to be set. The first one being the number of training episodes, which we set to 50,000. Second, we repeated each experiment 20 times and averaged the outcomes to stabilize the results. Also, recall that these experiments will be ran on the protein :

$$P = HPHPPHHPHPPHPH.$$

7. Update of the Q-matrix

In this project, instead of initializing a static zero matrix representing the Q values and filling it during the training, we instead opted for a dynamic python dictionary based approach, where we start with a dictionary containing the initial state alone, and each time we discover a new state, it is appended to the dictionary. This approach is efficient both in terms of computation and storage. The key in the dictionary is the state (the ordered list of positions of the k first elements, packaged as a python tuple data structure). The values are python lists of length 3 that contain the Q values for each one of the 3 actions (in the order Left, Forward, Right) given a state. This programming optimization was necessary otherwise Q-learning would never run on larger sequences since the number of states grows exponentially and we don't need to record them all.

Given the nature of this implementation, no Q-Matrix is available. However, we can show how new entries are being added to the dictionary and how their Q values are being updated after each step for one learning episode.

Time step	Q-Dictionary	Action Chosen	Next State
t_0	{ ((0, 0), (0, 1)) : [0, 0, 0] }	Right	((0, 0), (0, 1), (1, 1))
t_1	{ ((0, 0), (0, 1)) : [0, 0, 0.1], ((0, 0), (0, 1), (1, 1)) : [0, 0, 0] }	Forward	((0, 0), (0, 1), (1, 1), (2, 1))
t_2	{ ((0, 0), (0, 1)) : [0, 0, 0.1], ((0, 0), (0, 1), (1, 1)) : [0, 0.1, 0] , ((0, 0), (0, 1), (1, 1), (2, 1)) : [0, 0, 0] }	Right	((0, 0), (0, 1), (1, 1), (2, 1), (2, 0))
t_3	{ ((0, 0), (0, 1)) : [0, 0, 0.1], ((0, 0), (0, 1), (1, 1)) : [0, 0.1, 0], ((0, 0), (0, 1), (1, 1), (2, 1)) : [0, 0, 0.1], ((0, 0), (0, 1), (1, 1), (2, 1), (2, 0)) : [0, 0, 0]}	Right	((0, 0), (0, 1), (1, 1), (2, 1), (2, 0), (1, 0))
t_4	{ ((0, 0), (0, 1)) : [0, 0, 0.1], ((0, 0), (0, 1), (1, 1)) : [0, 0.1, 0], ((0, 0), (0, 1), (1, 1), (2, 1)) : [0, 0, 0.1], ((0, 0), (0, 1), (1, 1), (2, 1), (2, 0)) : [0, 0, 1.1], ((0, 0), (0, 1), (1, 1), (2, 1), (2, 0), (1, 0)) : [0, 0, 0] }	-	-

Table 4 : Q-Dictionary updates during the first episode for the sequence HHPPHH.

III. Experiments

1. Performance vs Episodes

In this section we run Q-learning on the environment of the sequence $P = HPHPPHHPHPPHPH$. The optimal energy we are looking to achieve is -6 which translates to a total episode reward of 7.2. The experiment is ran only once this time since the purpose of this section is just to show that the agent is indeed learning (we will use multiple runs later on when we optimize the parameters). The default values presented in a previous section are used for the hyper-parameters of Q-learning. For comparison purposes, we also run the experiment with a random agent, that is, an agent that randomly picks an action at each step without any form of learning whatsoever.

Performance measures in Q-learning (and reinforcement learning in general) can vary depending on the context and the problem being solved. For this work, we use two graphs to depict the performance of learning. The first and main one is the reward achieved by the agent per episode. This visualization however has too much variance, so instead, we use a moving average of window size 500 to stabilize the curve and better see the patterns. In this graph, the main feature we're interested in is the level of the curve, which represents the overall reward being achieved at a certain point in time. The second graph being used is the cumulative reward by episode. The main advantage of this second measure as opposed to the first one is the natural smoothness of the curve which is due to the small incremental steps after each episode. The main characteristic we will be looking at in this graph, is the slope of the curve, which reflects the rate of change of the reward achieved in each episode : the bigger the slope, the better our learning is. Figure 4 reports the results obtained.

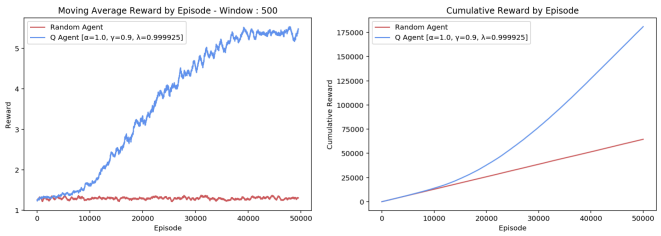

Figure 4 : Performance comparison between Q-learning and a random agent.

It is rather clear from the figure that our Q-learning agent is learning during his training as the reward by episode is increasing until it reaches a plateau of 5.5 from 40,000 episodes and onward. The random agent however, remains at the same level, roughly 1.3, throughout the 50,000 episodes. In a sense, the value of 1.3 represents the average reward that can be obtained by randomly choosing actions. It is worth noting that during the first few episodes, both the Q Agent and the Random Agent are at the same level, which is normal given that early on, Q-learning is exploring most of the time (exploration rate is 100% and decreases slowly). The second graph reflects the same result as the slope of the Q Agent changes over time, and becomes much higher than that of the random agent which remains constant. Finally, it is interesting to inspect the solution produced by our learning. Figure 5 shows the structure achieved by our Q-agent. This is constructed by placing the agent in an initial state and placing the next elements of the sequence according to a greedy policy (at each step, we pick the action with the highest "estimated' Q value). This looks like a good solution, however it's not the best one. Although the agent managed to avoid collisions, the energy achieved by this structure is -5, as opposed to -6 which we were hoping for. This partly explains why the first graph in Figure 4 was converging to a value of 5.5 even though the maximum is 7.2. We said partly because there is a second reason which has to do with the minimum exploration rate. Since the exploration rate never reaches 0 (the minimum is 5%), we will always have a chance to do some more exploration, and hence potentially reduce the reward. That said, we can probably improve our solution by better tuning the hyper-parameter values which we will attempt to do shortly.

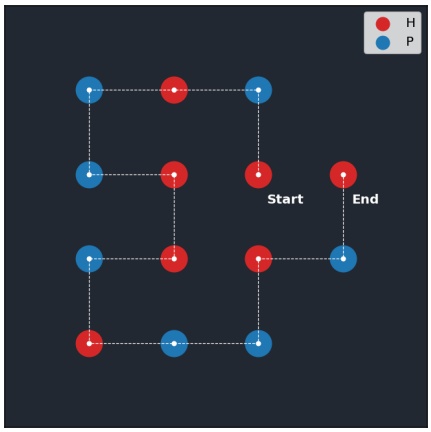

Figure 5 : Solution achieved by Q-learning.

2. Learning rate α

In this experiment, we explore the effect of the learning rate on the training of the agent. We try 5 different values spread throughout the range of [0, 1], and for each value, run the experiment 20 times and average the results to reduce the variance. Then, we visualize the two performance graphs in Figure 6.

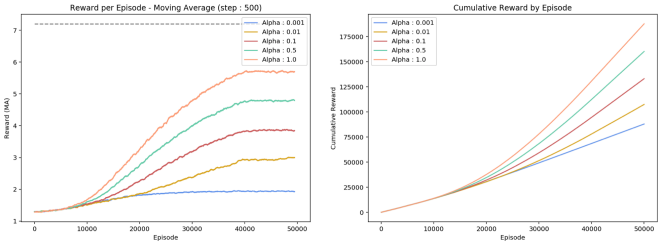

Figure 6 : Training with different learning rates.

The impact of the learning rate is rather clear from the figure, higher learning rates tend to result in a better convergence with 1.0 being the best value, all else equal. It is worth noting that for the two lowest values of 0.001 and 0.01, the curves seem to be still increasing at the final episode, indicating that the algorithm has yet to converge which is only natural given that the learning rate controls the speed of convergence. The dashed line shown as a reference, represents the best possible reward for an episode, which is 7.2.

3. Discount factor γ

We repeat the same experiment carried out in the previous section except this time, we vary the *Discount Factor* (gamma) instead and maintain everything else constant. Figure 7 reports the result obtained.

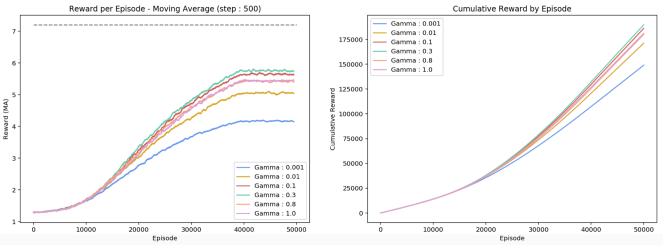

Figure 7 : Training with different Discount Factors.

This time again, results suggest there is a strong relationship between the Discount Factor and learning performance. Unlike what we believed at first, 0.8 (which is closer to the 0.9 we set as a default before) is not the best value here, although together with 1.0 their performances are still acceptable. The best gamma value seems to be 0.3 with 0.1 closely behind it (roughly moderate values), while the worst cases are those closer to 0. This confirms our previous statement that hyper-parameters are problem dependent and there are no universal values that fit all scenarios. Convergence however, although attained, is not optimal with final stable reward values closer to 6.

4. Decay rate λ

As we chose Epsilon-greedy to be our policy, we have 3 parameters to consider. Exploration Rate, Minimum Exploration Rate and Decay Rate. We shall keep the initial Exploration Rate at 1.0 (100%) and Minimum Exploration Rate at 0.05 (5%) and only vary the decay. The reason for this is because we need the exploration to be maximum at the beginning, and it needs to be non null around the end. So we're left with the decrease speed to tune. Figure 8 below depicts the results obtained.

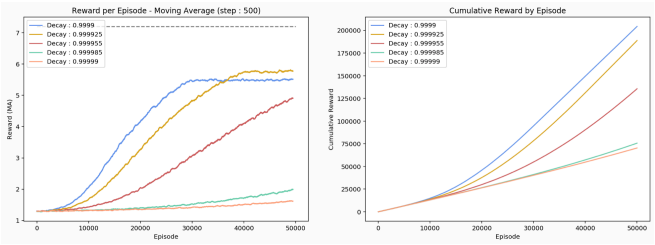

Figure 8 : Training with different Decay Rates.

From the graphs above, we note that 0.999925 (yellow curve) converges to the best value, which is also the default value we picked before. The value 0.9999 (blue curve) converges to a slightly lower value, but is much faster in its convergence rate than the yellow curve. This trade-off is interesting because there are situations in which it is desirable to parametrize the model to converge to a slightly sub-optimal value but with much less resources. Finally all the other curves seem to have yet to converge, with the red curve (0.999955) being the most promising one.

5. Hyper parameter optimization

What we have done so far is inspect the relationship between each hyper-parameter and the learning process independently. This is great for identifying how a particular parameter affects the outcome, but it doesn't necessarily help us find the best combination of values to use in Q-learning in order to reach the best performance. Indeed, the best gamma value found earlier for example, where we fixed the other parameters, doesn't make it the best gamma value for the overall problem. For finding the overall best gamma parameter, one would need to search the entire space of parameters allowing all of them to vary as well. In this context, a few approaches were proposed in the literature. By far, the most used one is *Grid Search*, where we predefine a grid of values and try each combination. However, this method is not efficient, and so we used a different approach called *Random Search* [5]. The idea here is to randomly draw values for each parameter from pre-defined probability distributions, and train with the combination. We repeat this process a certain number of times that our resources allow. We conducted Random Search 100 times, where parameters are drawn from the following distributions : $\alpha, \gamma \sim \mathcal{U}(0, 1)$, $\lambda \sim \mathcal{U}(0.9999, 0.99999)$. The reason we restrict the range of the decay rate is because we know from the above experiments that values far outside that range are either not going to converge, or converge to a sub-optimal solution very fast. Moreover, since we have generated 100 curves, we couldn't visualize them all due to clutter issues. Instead we selected the top 5 ones to compare. However, how do you compare two curves to determine that one is better than the other ? This generally depends on the problem, but in this work, we tried two methods. The first selects curves based on the cumulative reward at the end of the training (sum of all rewards gained during training), which basically ranks curves based on the overall score of the training. The second method however, is more concerned with the convergence level (whether we converge to a high reward value or not) irrespective of how much time it took to get there. To do so, we compute the average reward obtained in the last 100 episodes. Figure 9 shows the different curves for the two methods.

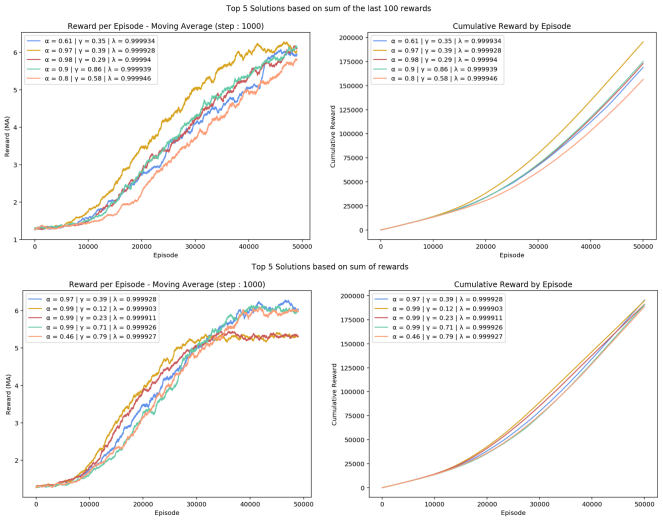

Figure 9 : Random Search for Hyper-Parameter Optimization with 100 iterations. We keep the top 5 curves accordingly to two criteria.

There is a noticeable difference between the two methods of curve comparison. Although the best curve seems to have been identified by both methods (yellow curve in the upper graph, blue in the lower one), the curves are generally different. In this case, we are more interested in solving the problem than maximizing the reward on the way there, thus, the upper graph is more appropriate. Interestingly, we have reached higher reward values than before, hitting the 6.2 mark. Figure 10 shows the structure found after training the agent using the optimal parameters from Figure 9. This is an optimal solution as it achieves an energy of -6 instead of the -5 obtained previously.

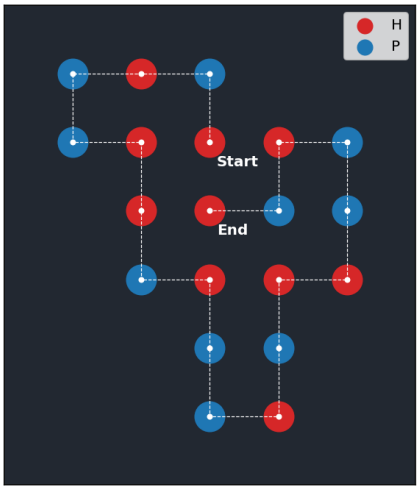

Figure 10 : Structure found by the agent after training with optimal hyper-parameter values.

6. Policy : Boltzmann Exploration

The last of our experiments concerns policies. So far, we have been using the epsilon-greedy policy, in this section we will compare it to the Boltzmann policy where we define its new parameters as : $T_{max} = 1.0$, $T_{decay} = 0.99991$, $T_{min} = 0.05$. The parameter T is the temperature which we anneal to 0.05 throughout the training. Figure 11 shows a comparison between the Boltzmann and the epsilon-greedy policies where the rest of the hyper-parameters are set according to our best achieved combination.

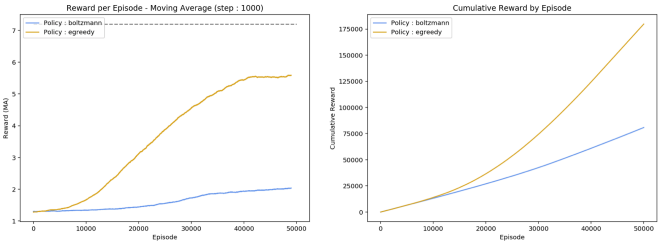

Figure 11 : Training with different Policies.

Surprisingly, the use of the Boltzmann policy yields poor results in this trial, which is likely due to the fact that the newly introduced hyper-parameters need proper tuning.

IV. Analysis

According to the first trial, the learning rate does indeed have a predictable effect on the learning process. The higher the learning rate, the faster we converge. High values for the learning rate are known to be risky in the sense that Q-learning may overshoot the update. In this case however, they seem to work best, with all values tried between 0.1 and 1.0 converging. But the higher the alpha value, the better the asymptotic behavior. This might indicate that the optimal Q values are far from the 0 initialization we chose form them. Thus, the update rule needs to make large changes in order to reach the optimal configuration.

The *Exploration Decay Rate* (λ) controls the speed by which we get to the minimum *Exploration Rate* (5%). If it's too low, we'll mostly be exploring in all of our 50,000 episodes, if it's too high, we reach the minimum exploration rate quickly, and stay relatively constant from there. The value that worked best for us (0.999925) reduces the exploration rate to its minimum by 4/5 of the training time (40,000 episodes), which was enough for the agent to explore the state space enough to get reasonable Q estimates. Afterwards, we're mostly exploiting our learned experience. This technique is usually used in the reinforcement learning literature where we start with a high exploration rate and turn it down to its minimum threshold (usually 1% or 5%) by half or three thirds of the training time. And from there, we mostly exploit what we learned, with a slight chance to explore, preventing the agent from ever getting permanently stuck.

One of the curious things about most of our experiments is that we never reach the optimal reward value of 7.2. This is understandable and is due to a few different reasons. First, since we have a minimum exploration rate of 5%, even when the action values converge to their optimal values, we will still explore in 5% of the remaining episodes leading to the best case scenario of a maximum average of 7.2 * 0.95 = 6.84 (assuming the reward is 0 in the episodes where we explore and when we exploit we get the full reward). But this is still above the values we were getting, which were between 5.8 and 6.2. The second reason is that the agent never gets to explore the entire state space, because 50,000 is not a high enough number of training epochs. Realistically, the agent explores around 45,000 different states in a training session which is far from the 265,720 states available (computed as the sum of a geometric series, with a first term a of 1 and a common ratio r of 3) for the protein being studied. Consequently, during each run of the experiment we usually converge to an energy of -4, -5 or -6 resulting in maximum rewards of 5.2, 6.2 or 7.2 after each episode which averages 6.2. Multiplying this value by 0.95 (instead of the unrealistic 7.2) gives 5.9 thereby explaining the results we were getting.

Hyper-parameter optimization is essential for achieving optimal performance in Machine Learning in general and Reinforcement Learning in particular. The motivation behind our *Random Search* comes from its original paper [5] hypothesizing that the different hyper-parameters of a model are not equally important for a given task. This makes the grid search approach not efficient and the *Random Search* superior. Furthermore, we could have used an even more sophisticated probabilistic approach called Bayesian Optimization [6] which was first introduced in the context of evaluating expensive cost function and was later shown to work well in machine learning hyper-parameter tuning. That said, when evaluating and comparing different models, it is important to have a single quantitative measure to rank the results accordingly. Curves however, are not a single measure, thus are difficult to compare. In Figure 9, we used two methods to convert the curves into single measures, and were more interested in the average reward value of the last few episodes because it was more indicative of the optimality of the structure we can expect from our trained Q-Agent. This optimization yielded positive results, exemplified by the two structures obtained in Figure 5 (with default values) and Figure 10 (after optimization).

The epsilon-greedy policy is by far the most prevalent one. However, despite its popularity, it is far from optimal as it only takes into account whether actions are the most rewarding or not. The Boltzmann policy was meant to correct that as it exploits all information contained in the estimated Q values. Instead of always picking the optimal action or a random action, we choose the appropriate action using probabilities as weights. The soft-max is introduced over the Q estimates for each action. This means the agent is more likely to pick the optimal action, but not all the time. The use of the relative value of each Q estimate is the biggest advantage the Boltzmann exploration has over the epsilon-greedy policy. Finally, we also use a Temperature parameter (T) which is annealed over time and controls the spread of the soft-max distribution. The poor result we got is likely reflecting a need for the temperature parameters to be optimized for our problem. Also, one of the shortcomings of the Boltzmann approach resides in the assumption that the soft-max over the estimated Q values provides a measure of the agent's confidence in each action. Nevertheless, this is not the case, since the agent is estimating optimality scores about each action and not how confident it is about this optimality.

V. Advanced Reinforcement Learning

Q-learning has been great so far, but one of the main issues it suffers from is the natural constraints surrounding the Q matrix. For a huge State Space, such as the one we have, storing all Q values in a matrix or even a dictionary is not feasible from a computational and storage perspectives. Also, since the agent will probably not visit each and every state of the space during training, it is desirable to actually approximate the Q values using some estimator. This is how the idea of Deep Q-Learning came about [7] where a neural network was used as a function approximator to predict the Q values for each action given a state as an input. The model then tries to minimize the loss between its current estimation of the target Q values, and their true values computed as the estimated returns. Interestingly, the estimator does not need to be a neural network, as any other machine learning algorithm would work. However, neural networks enjoy a great deal of flexibility and complexity making them an attractive solution for this particular task. From there, a series of incremental and independant improvements were proposed by the research community, all of which supplement the idea of Deep Q-Learning. One of the major ones is Double Deep Q-Learning [8] which was proposed to correct an issue with the previous approach. Indeed, the same neural network being trained, that outputs predictions (estimations of Q values) in DQN, is also being used to generate the target (true values). This makes training the model like the math equivalent of hitting a moving target. In this sense, Double DQN is a stabilization measure that consists of duplicating the neural network. The first one then acts as the estimator that we train, referred to as the O*nline Network*, and the second one, T*arget Network*, is held still and used to generate the target values. We transfer the *Online Network*'s weights to the *Target Network*' every so often to keep it updated. Furthermore, we use a second stabilization measure called *Experience Replay* where we store experiences of the agent, an experience being a list of tuples $(\text{state}, \text{action}, \text{reward}, \text{next-state}, \text{done})$. This way, instead of the agent learning as he explores the environment, he adds the experiences to the buffer. Then we replay those experiences from the storage to the agent so he can learn from them and make better decisions. The last addition we're going to mention here is referred to as Prioritized Experience Replay [9]. The idea behind it is, instead of randomly picking experiences from the buffer to replay to the agent, we prioritize the experiences from which we can learn the most, by sampling them according to a probability distribution proportionate to the agent's temporal difference error the last time it trained on that particular experience.

We have used a Deep Q-Learning approach to solve our protein folding problem. We also make use of the experience buffer to store past experiences and replay them to the agent. Since Deep Reinforcement Learning requires heavy computational resources, we couldn't push the model further by incorporating other additions such as Double DQN (although we did program it, we just couldn't train it as long as it required). Here, we train a Q-Agent and a DQN-Agent (learning rate of 0.001) for 100,000 training episodes. The training time took around 4h30 on an Intel core i7 7th Gen CPU. Figure 12 below shows the result.

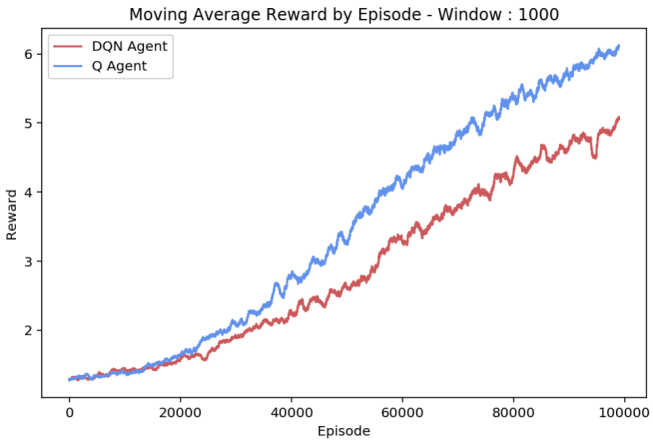

Figure 12 : Performance comparison between a Q-Agent and a DQN-Agent on the HPHPPHHPHPPHPH sequence.

The results show that the DQN Agent is clearly learning, however it has yet to converge. Thus, the 100,000 training episodes is clearly not enough with our current setting. With proper tuning, we expect the DQN Agent to perform at least as well as the Q Agent, while being a much more scalable approach to bigger Protein sequences (i.e bigger State Spaces).

VI. Conclusion

This work was an effort to showcase how reinforcement learning could be applied to solve a combinatorial optimization problem. In particular, we considered the Protein Folding problem, but this could generalize equally to other similar tasks such as the Traveling Salesman Problem (TSP). We used Q-Learning, as a

simple reinforcement learning technique belonging to the class of model-free, off-policy, value iteration based reinforcement algorithms. We also explored the use of Deep Q-Learning, an extension to Q-Learning that solves the issue of high dimensionality of the state space. However, even DQN is not state-of-the-art today, and we suspect that other more powerful RL algorithms could result in better and faster convergence. One such example is Asynchronous Advantage Actor-Critic (A3C), which combines both ideas of value iteration and policy iteration methods and boosts the learning via asynchronous parallel multi-agent training.

VII. References

[1] Fang, Yi. "Protein Folding: The Gibbs Free Energy." *arXiv preprint arXiv:1202.1358* (2012).

[2] Decatur, Scott E. "Protein folding in the generalized hydrophobic-polar model on the triangular lattice." *T echnical M emo M It-Lc S* (1996).

[3] Czibula, Gabriela, Maria-Iuliana Bocicor, and Istvan-Gergely Czibula. "A reinforcement learning model for solving the folding problem." *International Journal of Computer Technology and Applications* 2 (2011): 171-182.

[4] Sutton, Richard S., and Andrew G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[5] Bergstra, J., & Bengio, Y. (2012). Random search for hyper-parameter optimization. Journal of Machine Learning Research, 13(Feb), 281-305.

[6] Malkomes, G., Schaff, C., & Garnett, R. (2016). Bayesian optimization for automated model selection. In Advances in Neural Information Processing Systems (pp. 2900-2908).

[7] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

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

[9] Schaul, Tom, et al. "Prioritized experience replay." *arXiv preprint arXiv:1511.05952* (2015).