Acknowledgments

I would first like to thank my supervisor, Professor Christoph Feinauer, who gave me the opportunity to be involved in this formative research project and patiently guided me throughout it. Without his steady and patient support, and without his insightful feedback, this work would not have been possible.

I dedicate this work to my family, in particular to my parents, Gabriella and Giuseppe. They have always done everything they could to contribute to my personal and academic growth, and they have always pushed me to never give up. This work is also dedicated to my brother, Simone, who has been tolerating me for 24 years, and whose valuable suggestions in the field of medicine have been crucial for the preparation of this project. I thank him also for being kind enough to share his Amazon prime account with me, and for preventing me from giving money to the company that is now paying my salary.

Special thanks to my "compagno di banco" Mario Russo. We have constantly supported each other during this journey, and we shared joy and pain. A deep friendship has developed between us and it will not stop upon completion of our studies. Even if we are now living and working in two different countries, our constructive exchange of memes will bridge the gap.

I would also like to thank my roommate, Ania, who has been my main point of reference during quarantine. She turned the "stay at home" period into an authentic learning experience in terms of gastronomy, TV series and quarantine games. Most importantly, she has always been there when I needed tips concerning professional attitude, effective writing, and any other aspect that contributed to my personal growth. Some of the sentences you will read in this dissertation are the result of hours and hours of intensive review sessions with her.

I am grateful to my friends: Mael, Sette, Bomba, and all the people that have shared part of their lives with me, in Italy and around the world. Unfortunately, I cannot squeeze all the names in one page, but if you are reading this, your name is probably in my head while I am writing these few lines.

Merci beaucoup to the new friends I met now that I started my new adventure in Luxembourg. "Sorry, I have to work on my thesis" has been my favorite excuse to turn down every invitation in the last seven months. Now I am proud that this sacrifice was all worth it, and it is time to finally celebrate.

Last but not least, I would like to thank Bocconi University for the immense amount of opportunities this institution has offered me throughout these five years, and I would like to thank the big Amazon family, which has warmly welcomed me and has given me the chance to begin an exciting new chapter in my life.

Contents

1 Introduction

1.1 The amino acids

Proteins are biological macro-molecules which derive from the chemical union of simpler molecules called *amino acids*. These amino acids are linked with each other thanks to a link called *peptide bond*. In nature, there exist 20 different amino acids that can build thousands and thousands of proteins of different size and functionalities. The smallest proteins contain at least 20 amino acids, but they are more often composed by hundreds of them.

Most of the amino acids are made of carbon, hydrogen, oxygen and nitrogen, and some of them also contain sulfur. They are characterized by the presence of an amino group ($-NH_2$) and an acid group ($-COOH$), and this is why they are called amino acids.

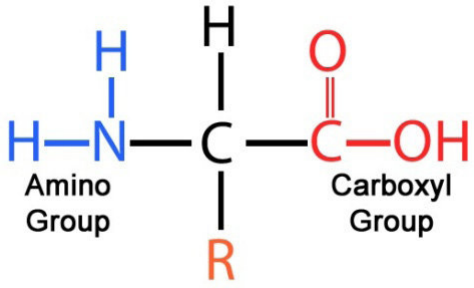

Figure 1.1: Amino acids general formula (source: `https://www.researchgate.net/publication/332440930_PROTEIN_ACIDIC_HYDROLYSIS_FOR_AMINO_ACIDS_ANALYSIS_IN_FOOD_-PROGRESS_OVER_TIME_A_SHORT_REVIEW`)

As we can see from Figure 1.1, the amino and the acid groups are connected to the same carbon atom, which is in turn linked to a hydrogen atom and to a variable amino acid residual (R). What makes every amino acid different from each other is this side chain R. In the simplest amino acid, the R group is only made of one hydrogen atom, while all the other amino acids have more complex groups, that can contain carbon, hydrogen, oxygen, nitrogen and sulfur atoms.

1.2 Protein structure and shape

In biology, we call dipeptides, tripeptides, oligopeptides and polypeptides molecules that are made of 2, 3, few and many amino acids, respectively. The peptide bond occurs between the carbon in the acid group of one amino acid and the nitrogen in the amino group of the other amino acid. This process also leads to the elimination of a molecule of water.

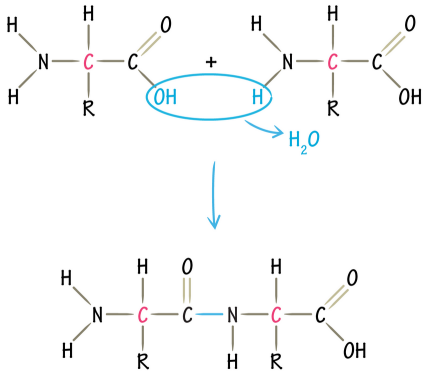

Figure 1.2: Peptide bond formation

Given the complexity of the protein structure, in order to be able to thoroughly describe a protein, we talk about primary, secondary, tertiary, and, for some proteins, quaternary structure. Each of these structures refers to a different level of molecular organization. The primary structure is simply given by the amino acid sequence in the polypeptide chain. The secondary structure is derived from the interactions between the different amino acids, that give a particular shape to the distinct parts of the chain. In the secondary structure, the two most common configurations are α Helix and β Sheet. The tertiary structure comes from the configuration of the protein in the space and from the combination of multiple regions linked with each other. This structure is stabilized by additional bonds between the lateral chains of the amino acids. The quaternary structure is related to the disposition, within the protein, of polypeptide subunits that are in turn characterized by a primary, a secondary and a tertiary structure. In this case, the protein is basically made of multiple chains, linked

by the same type of bonds that stabilize the tertiary structure. Not all the proteins have a quaternary structure.

1.3 Protein families

During evolution, proteins can mutate their structure, and this enables them to perform different functions. The outcome of this process is that today we can group most of the proteins into protein families. A protein family is a representation of all those proteins which are originated from one ancestral protein. Each family member has a sequence of amino acids that is similar to the one of the other proteins belonging to the same family. More specifically, the secondary structure of proteins belonging to the same family is very similar, and this means that these proteins have almost identical functions, even if their primary structure can differ. In many cases, the evolution has brought the amino acid sequences of some proteins to diverge so much that it is hard to assess if two proteins belong to the same family until we determine their three-dimensional structures.

Furthermore, given that this evolutionary process is random, sometimes these mutations can be deleterious and harm the protein, rather than improving its functionalities. However, in nature, only fit proteins exist, since natural selection gets rid of the faulty proteins that result from a bad genetic mutation. When we are dealing with proteins within human cells, it is crucial to predict if a certain mutation can be potentially pathogenic, and this is why the interest in modeling genetic mutations to determine their effects has been increasing among bio-engineering and human health researchers. Being able to successfully model such mutations can lead to correctly predict the effect of a genetic change on the human body and, ultimately, save lives.

1.4 Beta-lactamase TEM-1 proteins

The proteins we will be analyzing are the ones belonging to the beta-lactamase family. Beta-lactamases are enzymes that bacteria produce in order to increase their resistance to antibiotics. The main function of beta-lactamases is in fact to break the structure of the antibiotics. The antibiotic resistance of these proteins is measured through the calculation of the MIC values. MIC stands for Minimum Inhibitory Concentration, and it is the lowest concentration of a drug that is needed to prevent visible growth of bacteria [14]. In our case, the MIC value basically tells us how well the protein works: the protein in question is involved in making the bacterium resistant against antibiotics, and the MIC values measure this resistance.

The most common beta-lactamase bacterium is the TEM-1. The term TEM is derived from the name of the patient, Temoniera, from which the isolate was recovered in 1963 [17]. This specific type of beta-lactamase bacterium is the one we will be exploring throughout this study. The final goal is to build a statistical model that is able to assign a low probability to those genetic mutations that are potentially pathogenic, and vice versa. In other words, given a genetic mutation, we want the output of our model to be strongly correlated with the MIC values calculated in laboratories. If a mutation is bad for the protein, the MIC values for that protein will be lower, since the protein will have a worse resistance to antibiotics. This will need to be reflected in our model: if the genetic mutation is predicted to be bad for the protein, then our model will assign a lower probability to the mutated sequence. We will go more into the details as we proceed with this paper.

2 The Boltzmann distribution

As outlined in the paper Mutation effects predicted from sequence co-variation, by Thomas A. Hopf et al. [11], each protein family is often modeled as a probability distribution that is defined as follows:

$$p(\sigma) = \frac{1}{Z} \exp\{E(\sigma)\} \tag{2.1}$$

where p is the probability that an evolutionary process generates the sequence σ. E is a function that can take different forms, we will go more into the details in the next section. On the other hand, Z is a partition function that normalizes the distribution, we can define it as

$$Z = \sum \underbrace{\exp\{E(\sigma)\}}_{\text{Boltzmann factor}} \tag{2.2}$$

and it is the sum of the so called Boltzmann factors of all the possible sequences σ.

The idea behind a well functioning model is, intuitively, to assign high probability to fit, healthy protein sequences, whereas other sequences are expected to have a lower probability. Let us consider a protein sequence $\sigma = (a_1, ..., a_{i-1}, a_i, a_{i+1}, ..., a_L)$, where $(a_1, ..., a_L)$ is a sequence consisting of L amino acids. We will define $\sigma' = (a_1, ..., a_{i-1}, b, a_{i+1}, ..., a_L)$ the protein sequence that is obtained as a result of a genetic mutation where the amino acid a_i at position i is substituted by the amino acid b. We expect $p(\sigma) - p(\sigma') > 0$ if this mutation is bad for the protein, and $p(\sigma) - p(\sigma') \leq 0$ otherwise.

2.1 Definition of $\mathbf{E}(\sigma)$

So far, so good. The tricky part, which is also the main objective of this work, is to find a good definition for the "energy" function $E(\sigma)$. Hopf et al. [11] define this function as

$$E(\sigma) = \sum_i \mathbf{h}_i(\sigma_i) + \sum_{i<j} \mathbf{J}_{ij}(\sigma_i, \sigma_j) \tag{2.3}$$

where $\mathbf{J}$ is the coupling term between a pair of residues (i, j) while $\mathbf{h}$ is the site-wise bias term. A model that combines equation 2.1 and 2.2 with equation 2.3 is also known as Markov random field, or Potts model, in statistical physics, or pairwise graphical model, in computer science.

The estimation of the two terms $\mathbf{h}$ and $\mathbf{J}$ that define the energy function is not an easy task, although many methods have been implemented for this scope, as explained by Cocco et al. [3]. In this paper, the energy function will be substituted by a neural network. The energy of a protein sequence will therefore be defined as

$$E(\sigma) = MLP(\sigma) \tag{2.4}$$

where MLP is a multi-layer perceptron that will be described in details in section 4.

2.2 Multiple Sequence Alignment

Proteins are basically sequences of amino acids. Before exploring all the details of the multi-layer perceptron and how we are going to train it, we need to understand how we can translate this sequence of amino acids into a format that the computer can read.

Proteins are divided into families, and a family is composed by proteins that are similar with each other and have usually the same structure and function. The amino acid sequences

of these proteins can change during evolution, but the structure and the function of the protein typically remains the same. As mentioned at the beginning of this dissertation, we will work with beta-lactamase TEM-1.

Throughout known life, there exist 20 amino acids (e.g. Alanine, Arginine, Asparagine, etc.), and a combination of these amino acids generates the protein. Each of these amino acids is also identified by a letter code: the Alanine is identified with the letter A, the Arginine with letter R, Asparagine with N, and so on. This means that we can simply treat the protein sequence as a string of characters, and then encode this string into a tensor that we will feed to the neural network.

Since these amino acid sequences change during evolution, proteins belonging to the same family can have different lengths, and some amino acids that appear in one sequence may not appear in another one, and vice versa. In order to feed the sequences to the neural network, we need them to be aligned and have the same length. In order to do so, we insert a placeholder, called gap, which is indicated by a "-" symbol. Once all the sequences are cleaned up and are all of the same length, we can represent the sequences as a single matrix that is called *Multiple Sequence Alignment* (MSA).

2.3 Encoding and representation of the sequences

We said that we can represent the protein sequence as a string of characters. If, for instance, a protein had length 30, including amino acids and gaps, we could represent it as follows:

$$\sigma = (a_1, ..., a_{30}) = (-, -, A, A, Q, L, C, V, I, R, N, G, R, ..., G) \tag{2.5}$$

The sequence shown at 2.5 has two gaps in position 1 and 2, and twenty-eight between amino acids and gaps in the remaining positions.

If we want to represent the same sequence in matrix form, we end up with an $n \times m$ table with n equal to the number of all the possible amino acids and m equal to the length of the sequence. In this specific case, we will have a matrix with 21 rows (the 20 amino acids plus the gap symbol) and 30 columns .

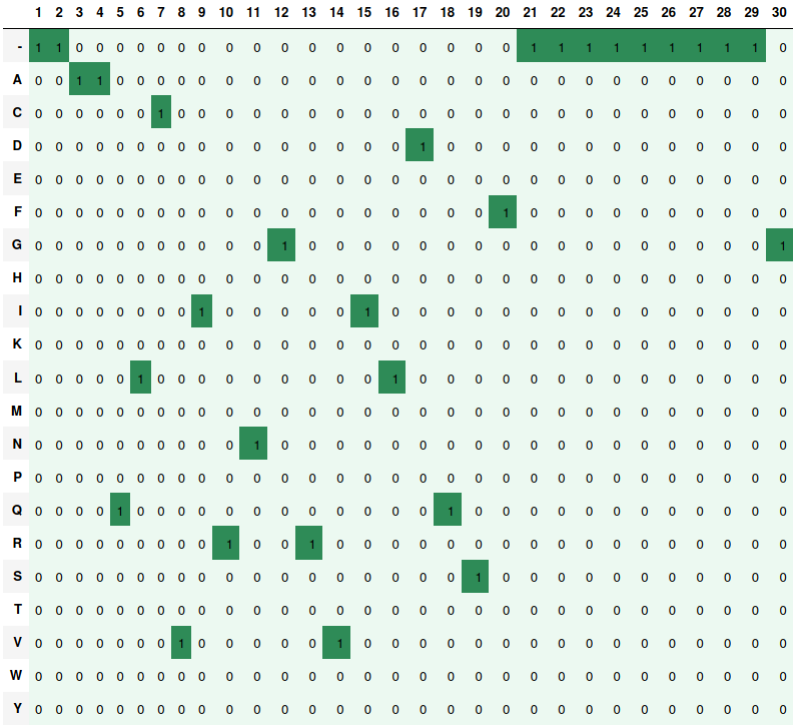

Figure 2.1: Representation of the Multiple Sequence Alignment matrix

The matrix represented in Figure 2.1 clearly shows that in position 1 and 2 we have a gap, in position 3 and 4 we have alanine (A), in position 4 we have glutamine (Q), and so on.

In order to finally encode the protein sequence, we just need to flatten this matrix and turn it into a one-dimensional vector. We will therefore end up with a vector with $30 \times 21 = 630$ elements that we can represent like this:

```
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
```

0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ..., 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

After cleaning up our beta-lactamase TEM-1 proteins as described in section 2.2, they all become of length 330. If we encode them as described in this section, we will have a vector of length $330 \times 21 = 6,930$ for each of these sequences. This vector will be the input that we will feed to our neural network, which will therefore have $6,930$ input neurons.

3 Why unsupervised learning

We need to begin this section with a trivial, almost Darwinian statement, which is however the main and most obvious reason why it is so complex to model the behavior of genetic mutations and why so much work has been and keeps being done on this topic: in nature, only the fittest survives. If this statement were not true, we would have data from "healthy" and "unhealthy" proteins, and we could just build a relatively simple supervised learning algorithm in order to determine if a certain genetic mutation is potentially bad for the future of the protein.

The first step to overcome this issue is to use an energy-based model (EBM) by taking inspiration from what Yilun Du and Igor Mordatch describe in their paper Implicit Generation and Modeling with Energy-Based Models [5]. The real challenge here is to implement from scratch an EBM that works for our purpose, since the existing models, including the one developed by Du and Mordatch, are mostly written for image recognition.

3.1 Generating images with energy-based models

Generative models are widely used in the field of computer vision and image generation. This is one of the most interesting topics in the field of data science and, for this reason, scientists have developed many different strategies that allow this kind of models to keep improving over time. Today, we are able to generate images that are almost indistinguishable from real images. Energy-based models are a subset of these techniques, and they come really handy also for our purpose.

As described by Du and Mordatch, also these models have the aim of learning an energy function $E(x)$. When dealing with images, neural networks are often the best choice, and this is why $E_\vartheta(x) \in \mathbb{R}$ is represented in their paper as a neural network parameterized by weights ϑ. This energy function is then used to define a probability distribution that is very

similar to what we mentioned in section 2

$$p_\vartheta(x) = \frac{\exp\{-E_\vartheta(x)\}}{\int \exp\{-E_\vartheta(x)\}dx} \tag{3.1}$$

The issue is that, when working with images, generating data from this distribution by relying on Markov Chain Monte Carlo (MCMC) methods can be tough, since the mixing times of those methods can be long, meaning that it could be humanly impossible for the Markov chain to get "close" to its steady state distribution in a reasonable time.

The solution proposed by Du and Mordatch is to apply Langevin dynamics in the sampling process: given a data point $\tilde{x}^{k-1}$, the following data point $\tilde{x}^k$ is generated according to the following process

$$\tilde{x}^k = \tilde{x}^{k-1} - \frac{\lambda}{2}\nabla_x E_\vartheta(\tilde{x}^{k-1}) + \omega^k \tag{3.2}$$

with $\omega^k \sim N(0, \lambda)$. This strategy works fine since images have continuous state, and, as $K \to \infty$ and $\lambda \to 0$, the samples generated by this procedure come from the distribution defined by the energy function, as proven by Welling and Teh [20]. Unfortunately, proteins have discrete state, therefore we will need to follow a slightly different strategy for our purpose.

3.2 Sampling protein sequences with energy-based models

As mentioned, implementing Langevin dynamics does not work in our case, since we are dealing with discrete data. As an alternative, we can implement a standard Metropolis-Hastings algorithm to sample protein sequences. As described in section 2.2, our protein sequences are represented as strings of 330 characters, where each character is either an amino acid or a gap "-". The sampling algorithm works by starting from a random protein sequence $\sigma = (a_1, ..., a_{i-1}, a_i, a_{i+1}, ..., a_{330})$ and proceeding as follows:

1. sample a random position i and a random amino acid b;

2. generate a new sequence by exchanging a_i (the amino acid in position i) with the amino acid b;

3. calculate the difference in energy ΔE between the original sequence and the newly generated sequence;

4. "accept" the new sequence with probability $p = \exp\{-\Delta E\}$;

5. take the latest "accepted" sequence as new input and repeat the process.

The mathematical details of this algorithm are described in Appendix A.

4 Training the neural network

So far, we have treated our energy function E as a known, magic function that we feed to our model in order to get the results we want. In reality, we do not know yet how this energy function is composed, and this is actually the objective of this study. Given $n = 21$ the number of amino acids (plus the gap symbol), and m positions in the protein sequence, the energy function maps each protein sequence to a number $E(\sigma) : \mathbb{R}^{n \times m} \mapsto \mathbb{R}$, with $m = 330$ in our case. As described in section 2.1, $E(\sigma)$ is often defined as the sum between a coupling term $\mathbf{h}$ and a site-wise bias term $\mathbf{J}$.

We want, on the other hand, to define this energy function as a neural network and, in order to do so, we need to train it.

4.1 Network details

The more complex the model, the higher the computational time. Given that the sampling process is already time consuming, we can start with a relatively simple neural network. We will use a simple fully connected multi-layer perceptron with one hidden layer consisting of 100 hidden units, and Rectified Linear Unit (ReLU) activation function, which is the most widely used. In order to feed a protein sequence to this network, we need to represent the sequence as a matrix, as described in section 2.2; we then flatten this matrix into a one-dimensional vector with $330 \times 21 = 6,930$ elements that the perceptron can take as input.

In the end, our perceptron will be composed as follows: $6,930$ neurons in the input layer, 100 neurons in the hidden layer, and 1 output neuron.

4.2 Training process

Everyone knows the Turing test: a human being C communicates with a machine A and another human being B. C exchanges messages with A, and B and tries to decide who is the machine and who is the human being. If C cannot decide within a sufficient time, the machine can be said to be intelligent.

In our case, we do not have two different models for generation and discrimination, but we can still devise a similar test. We have a sampling model, described in section 3.2, and we have a set of real sequences belonging to the beta-lactamase TEM-1 family. The neural network implied in the sampling process is also used to assign a score to each sequence and, ultimately, distinguish good proteins from bad proteins. Since the neural network is then used both for sampling and for discriminating, once it learns to distinguish real proteins from the generated ones, it will also be able to sample protein sequences whose distribution is closer to the real distribution. Better generating proteins will allow the model to better learn what is a good protein and what is a bad one, and we expect this virtuous circle to develop a model that will be able to predict the effect of genetic mutations on real sequences.

5 Maximum Likelihood training

In order to train our model, we can take advantage of what Du and Mordatch have implemented when dealing with images [5]. Although we are not dealing with the same kind of data, their training algorithm can work well in our case too. However, before diving into the details, we need to take a step back and recall how maximum likelihood and gradient descent work.

One of the biggest problems in statistics is the estimation of the parameters of the distribution of a population, given a sample extracted from the population itself. This estimation problem can have two distinct forms. If we assume that the analytical form of the distribution function of the population is well known, and we only need to estimate one or more parameters of this distribution, then we are talking about *parametric estimation*. An example of parametric estimation is the estimation of the parameters μ and σ^2 when we know that our population is distributed according to a Normal distribution. If, instead, we do not assume any analytical form for the distribution function, we are in the field of *non-parametric estimation*. In our particular case, we are dealing with a parametric problem, since we are assuming a particular distribution function, and we want to find the best set of parameters ϑ for our neural network.

5.1 Maximum Likelihood estimation

One of the most ancient methods for the calculation of the estimators is the moment method, proposed by Karl Pearson at the end of the XIX century [9]. This strategy allows to obtain in a simple manner reasonable estimators, but the estimators obtained in this way often do not benefit from good properties. The technique that we are instead using to infer our parameters is the maximum likelihood estimation.

Before mentioning what is the principle that maximum likelihood estimators rely upon,

we need to define what a likelihood function is.

Definition 5.1. Let X be a population with probability function $f(x;\vartheta)$, depending on the parameter ϑ. If we extract a sample $(X_1, ..., X_n)$ from our population X, we call *likelihood* of the realization $(x_1, ..., x_n)$ of that sample, the quantity

$$\mathcal{L}(\vartheta; x_1, ..., x_n) = \prod_{i=1}^{n} f(x_i; \vartheta) \tag{5.1}$$

We can examine this example in order to better understand this concept. Let us consider a coin, we want to infer if it is a trick coin or a normal one. Let us therefore assume that the probability that the coin lands on heads (ϑ) is either $\vartheta = \frac{1}{2}$, if the coin is normal, or $\vartheta = \frac{2}{3}$, if it is a trick coin. We toss the coin 10 times, and we want to choose what is the most likely value of ϑ, between the two proposed values. Let us suppose that our realization is $\bar{x} = (H, T, T, H, H, T, H, H, H, T)$, meaning that we have 6 heads and 4 tails. Given the two possible values of ϑ, the likelihood of obtaining this result is

$$\begin{aligned} \mathcal{L}\left(\frac{1}{2}; \bar{x}\right) &= \left(\frac{1}{2}\right)^{10} = 0.000977 \\ \mathcal{L}\left(\frac{2}{3}; \bar{x}\right) &= \left(\frac{2}{3}\right)^{6}\left(\frac{1}{3}\right)^{4} = 0.0011 \end{aligned}$$

Since $\mathcal{L}\left(\frac{1}{2}; \bar{x}\right) < \mathcal{L}\left(\frac{2}{3}; \bar{x}\right)$, and since we assume that the observed realization is more likely to occur, then we can conclude that our estimate of ϑ is $\hat{\vartheta} = \frac{2}{3}$. The most likely estimate is $\frac{2}{3}$, since the value of $\vartheta = \frac{2}{3}$ leads to a higher probability of obtaining the observed realization.

Definition 5.2. The estimator $\hat{\vartheta}(x_1, ..., x_n)$ is called *maximum likelihood estimator* if $\hat{\vartheta}$ is the maximum of the likelihood function, that is, if $\hat{\vartheta}$ is such that

$$\mathcal{L}(\hat{\vartheta}; x_1, ..., x_n) \geq \mathcal{L}(\vartheta; x_1, ..., x_n) \tag{5.2}$$

for all possible ϑ.

In usual applications, finding the maximum likelihood estimator is not an easy task. Since we are basically trying to solve an optimization problem, the most straightforward solution is to compute the first order conditions, in order to find the value of ϑ that maximizes our likelihood function $\mathcal{L}$. In mathematical terms, we want to find ϑ such that

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial \vartheta} = 0 \\ \frac{\partial^2 \mathcal{L}}{\partial \vartheta^2} < 0 \end{cases} \tag{5.3}$$

Since the logarithmic function is monotone, it is usually more convenient to substitute to the likelihood function with the log-likelihood function. By doing so, equation 5.1 takes the following form

$$\frac{\partial}{\partial \vartheta} \ln \mathcal{L}(\vartheta; x_1, ..., x_n) = \sum_{i=1}^{n} \frac{\partial}{\partial \vartheta} \ln f(\vartheta; x_i) = 0 \tag{5.4}$$

Finally, optimization mostly assumes that we want to minimize a function, rather than maximizing it, and many software libraries implement only minimization. For this reason, this optimization problem is solved by minimizing the negative log-likelihood, rather than maximizing the likelihood. In addition, maximizing the likelihood is the same as minimizing the Kullback-Leibler divergence between the data distribution and the model distribution. Therefore, the optimization problem to find the maximum likelihood estimator often takes the following form

$$\min - \sum_{i=1}^{n} \ln f(x_i; \vartheta) \tag{5.5}$$

5.2 Gradient descent

The concept of optimization plays a crucial role when we talk about machine learning and deep learning in particular. The main goal of deep learning algorithms is to build an op-

timization model that, through an iterative process, minimizes an objective function $J(\vartheta)$ called *loss function* (or cost function).

The most popular optimization methods can be divided into two groups: the first order optimization methods, that include gradient descent, and the second (or higher) order optimization methods, that include, for instance, Newton's method. We will focus our attention on the former group, which is the one that applies to our case.

5.2.1 Batch gradient descent

The main idea behind the batch gradient descent algorithm is to iteratively update a parameter ϑ by moving it towards the opposite direction of our objective function $J(\vartheta)$. This update is developed in such a way as to gradually converge towards the optimum value of the objective function.

One of the parameters that we need to tune when performing this operation is the learning rate η, which basically determines the speed rate at which our parameter ϑ will move towards the optimum value. A high value of η allows ϑ to move faster towards the optimum, but it has the risk of missing the optimum point and diverging. On the other hand, a low value of η implies that ϑ will take much smaller steps, at each iteration. In this case, it is much more likely that ϑ will end up in the optimum point, but there is a high risk of ending up in a local minimum and, in addition, the learning process will be much slower. The solution is often to tune an additional parameter, called decay. The decay allows the learning rate to be big, in the initial steps of the optimization process, and progressively become smaller and smaller. This allows to avoid local minima and, in the same time, to prevent from diverging from the global minimum.

Summing up, our parameter ϑ_t at time t will be updated at time $t+1$ according to the

following equation

$$\vartheta_{t+1} = \vartheta_t - \eta \nabla_\vartheta J(\vartheta) \tag{5.6}$$

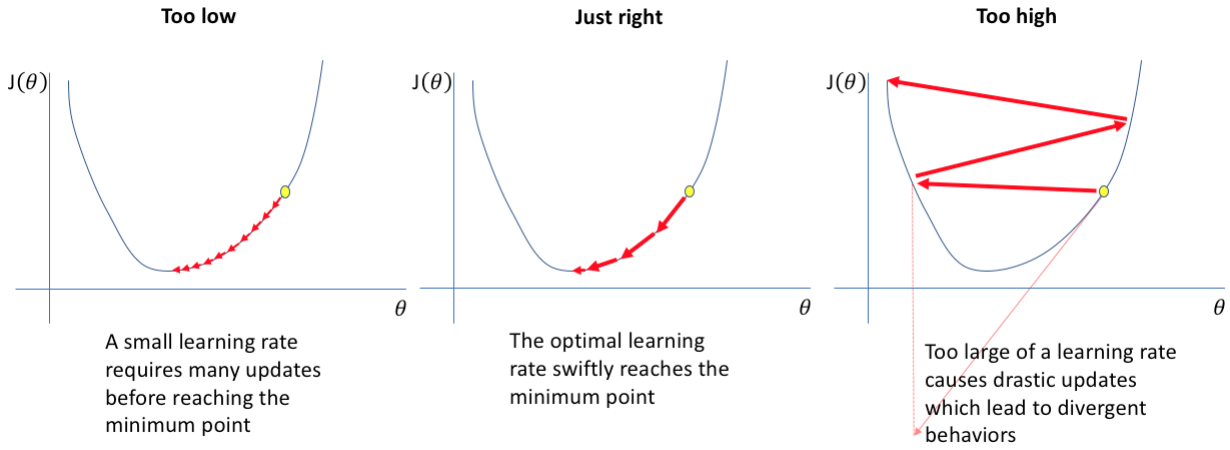

Figure 5.1: Setting the optimal learning rate of a neural network (source: `https://www.jeremyjordan.me/nn-learning-rate/`)

5.2.2 Stochastic gradient descent

In the previously described method, the gradient $\nabla_\vartheta J(\vartheta)$ is calculated by taking the whole training set at each iteration. This determines a high computational complexity, which can be hard to control if the dataset is too large. In order to avoid this issue, one of the possible solutions is to calculate the gradient on a single sample, which changes randomly at each iteration. This is the main idea behind stochastic gradient descent

$$\vartheta_{t+1} = \vartheta_t - \eta \nabla_\vartheta J(\vartheta, x^{(i)}) \tag{5.7}$$

Not only does this process reduce the computational complexity, but it also prevents the final solution to remain "trapped" in a local minimum of $J(\vartheta)$.

5.2.3 Mini-batch gradient descent

The shortcoming of using a single random sample from the training set is that the direction of the gradient can become too oscillating. A solution to reduce the variance of the gradient is to introduce a variation called mini-batch gradient descent, which is basically a mixture of batch and stochastic gradient descent. Instead of one single sample, at each iteration we take n samples from the training set, and we use them to calculate the gradient $\nabla_\vartheta J(\vartheta)$. This way, we reduce the variance of the gradient and we also stabilize the convergence

$$\vartheta_{t+1} = \vartheta_t - \eta \nabla_\vartheta J(\vartheta, x^{(i:i+n)}) \tag{5.8}$$

5.2.4 The momentum method

The concept of momentum derives from mechanical physics, and it takes into account a speed parameter ν. The idea behind this method is to calculate, at each iteration, an exponential moving average of the historical gradients and use this value as the direction to follow to update ϑ. An additional parameter $\beta \in [0, 1)$ determines the decay rate of the contribution of these historical gradients

$$\begin{aligned} \nu_t &= \beta \nu_{t-1} - \eta \nabla_\vartheta J(\vartheta) \\ \vartheta_{t+1} &= \vartheta_t + \nu_t \end{aligned}$$

5.2.5 The Nesterov momentum

A variation of the momentum method is the so called *Nesterov accelerated gradient*. The idea is the following: instead of evaluating the gradient in ϑ, we evaluate it at a more advanced

position $\vartheta + \beta\nu_{t-1}$, towards the direction of the momentum ν_{t-1}

$$\begin{aligned} \hat{\vartheta}_t &= \vartheta_t + \beta\nu_{t-1} \\ \nu_t &= \beta\nu_{t-1} - \eta\nabla_{\vartheta} J(\hat{\vartheta}) \\ \vartheta_{t+1} &= \vartheta_t + \nu_t \end{aligned}$$

5.2.6 Adaptive learning rate

What we mentioned at the beginning of this section is that, in order to speed up the convergence and avoid being trapped in local minima, one solution is to regulate the learning rate. The AdaGrad method [6] implements a dynamic regulation of the learning rate, based on the historical values of the gradient.

During the updating process of the parameter ϑ, the learning rate will not be constant, as in the previously described methods, but it will be recalculated by taking into account the historical values of the gradients. The disadvantage of this method is that, if the training process is very long, the learning rate will tend to 0, in the long run, and, as a consequence, ϑ will stabilize around the wrong stationary value.

$$\begin{aligned} g_t &= \nabla_{\vartheta_t} J(\vartheta_t) \\ \nu_t &= \sqrt{\sum_{i=1}^{t} (g_i)^2 + \varepsilon} \\ \vartheta_{t+1} &= \vartheta_t - \eta\frac{g_t}{\nu_t} \end{aligned}$$

This problem of ϑ tending to the wrong stationarity has been solved by Geoffrey Hilton with the RMSProp (Root Mean Square Propagation) method [10]. This method only takes a time window of the historical gradients and calculates, at each iteration, the cumulative moment

of the second order

$$\begin{aligned} \nu_t &= \beta\nu_{t-1} + (1-\beta)g_t^2 \\ \vartheta_{t+1} &= \vartheta_t - \eta\frac{g_t}{\nu_t} \end{aligned}$$

where $\beta \in [0, 1)$ is usually fixed around 0.9, as also suggested by Hilton himself, and it determines the importance of the memory in the past iterations in the calculation of the moving average. If β is set to 0, only the gradient of the current iteration is used.

Finally, we have the Adam method (adaptive momentum estimation) [13] which, on top of memorizing the exponential moving average of the square of the gradients of the previous iterations, it also memorizes the exponential moving average of the gradients $m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t$

$$\begin{aligned} g_t &= \nabla_{\vartheta_t} J(\vartheta_t) \\ m_t &= \beta_1 m_{t-1} + (1-\beta_1)g_t \\ \alpha_t &= \alpha\frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \\ \vartheta_{t+1} &= \vartheta_t - \alpha_t\frac{m_t}{\sqrt{\nu_t}+\varepsilon} \end{aligned}$$

There are of course many other gradient descent techniques, but the ones listed here are enough to have a better view on how gradient descent works and what are the main risks and challenges. This will also help us understand what happens when, in the next sections, we will talk about the training algorithm of our model.

5.3 Training our model with maximum likelihood and gradient descent

In order to let the distribution defined by our energy function E model the data distribution, we need to maximize the likelihood of our data. In other words, we need to find the parameters $\hat{\vartheta}$ such that, given a likelihood function $\mathcal{L}$, $\mathcal{L}(\hat{\vartheta}; x_1, ..., x_n) \geq \mathcal{L}(\vartheta; x_1, ..., x_n)$ for any possible ϑ. In order to maximize the likelihood function, we can more easily minimize the negative log likelihood, as it usually happens in these cases. As shown by Du and Mordatch [5], we want to solve the following minimization problem

$$\min_{\vartheta} \mathcal{L}(\vartheta) = \mathbb{E}_{\boldsymbol{x}}[E_\vartheta(\boldsymbol{x}) - \log Z(\vartheta)] \tag{5.9}$$

We will minimize this function through a gradient descent iterative process, which will be described more in details in section 6.3. In order to perform gradient descent, however, we need of course the gradient of $\mathcal{L}(\vartheta)$ with respect to the parameters ϑ. As shown by Turner on his the CD notes he wrote in 2005 [19], the gradient of $\mathcal{L}$ takes the following form:

$$\nabla_\vartheta \mathcal{L} = \mathbb{E}_{\mathbf{x}^+}[\nabla_\vartheta E_\vartheta(\mathbf{x}^+)] - \mathbb{E}_{\mathbf{x}^-}[\nabla_\vartheta E_\vartheta(\mathbf{x}^-)] \tag{5.10}$$

In our case, the positive samples $\mathbf{x}^+$ are represented by the real protein sequences belonging to the beta-lactamase TEM-1 family, whereas the negative samples $\mathbf{x}^-$ come from the MCMC sampler described in section 3.2. As we can see from equation 5.10, at each gradient step we will decrease the energy of the positive data samples, and we will increase the energy of the negative ones. If the network is able to learn correctly, once trained it will assign lower energy to the real protein sequences. This is coherent with the probability function 3.1 described in section 3. Since protein sequences have discrete state, our probability function

takes a slightly different form with a sum, instead of the integral, at the denominator

$$p_\vartheta(x) = \frac{\exp\{-E_\vartheta(x)\}}{\sum \exp\{-E_\vartheta(x)\}} \tag{5.11}$$

If the model assigns lower energy to the real protein sequences, then $p_\vartheta(x)$ will be higher, which is exactly what we want.

6 Implementation

As mentioned in section 3.2, we will implement a standard Metropolis-Hastings algorithm to sample the negative samples that we will use to train our model. The candidates that will be proposed throughout the process will be accepted with probability $p = \exp\{-\Delta E\}$ and, of course, rejected with probability $1 - p$.

Since $\Delta E = E(\sigma\prime) - E(\sigma)$, we first need to check that sampling sequences using this multi-layer perceptron makes sense, meaning that we need to observe enough variability between the sampled protein sequences. Typically, these Monte Carlo chains need to equilibrate: they start off not being distributed according to the underlying distribution and slowly converge to it.

Calculating the exact distribution of our protein sequences is computationally infeasible, since $p(\sigma) = \frac{1}{Z}\exp\{E(\sigma)\}$, where Z is the sum of the energies of all the possible sequence, $Z = \sum \exp\{E(\sigma)\}$. Given that each sequence is composed by 330 elements, and since each element can be either a gap or one of the 20 amino acids, there are 21^{330} different combinations of sequences. Calculating the energy for all of them would require ages, with the computational power at our disposal. As a comparison, the total number of atoms in the universe is estimated to be between 10^{79} and 10^{81}.

6.1 Consistency check

In order to make sure that the sampled sequences converge to the underlying distribution, we can still make a consistency check. We can build a multi-layer perceptron similar to the one that we will use for the protein sequences, but we will tailor it to a smaller system. We will use a random MLP for sequences of size 3, meaning that the model will have just $21 \times 3 = 63$ input neurons. Calculating the exact distribution of sequences of length 3 is feasible, since there are only $21^3 = 9,261$ possible combinations, and we can calculate the energy for each

of them in a timely manner.

After calculating the true distribution, we will run a Markov Chain Monte Carlo simulation and we will compare the marginals between the simulation and the probabilities. This is to test the implementation: if the probabilities calculated with the simulation follow the probabilities calculated through the Boltzmann distribution, we can then proceed by working with the real sequences.

6.2 Results

We will sample protein sequences of length 3 using the same algorithm described in section 3.2 and running $n = 6,000,000$ iterations. The marginals that result from the MCMC simulation and the true probabilities are stored in a table that looks like the following:

σ_i	Sequence	$p(\sigma)$	$p_{MCMC}(\sigma)$
σ_1	YDF	0.000126	0.000125
σ_2	ECH	0.000126	0.000135
σ_3	YDH	0.000125	0.000124
σ_4	YCH	0.000124	0.000118
σ_5	YLF	0.000124	0.000106
...	...	...	...
σ_{9617}	AFC	0.000092	0.000096
σ_{9618}	HFA	0.000091	0.000093
σ_{9619}	A-A	0.000091	0.000092
σ_{9620}	AFN	0.000091	0.000086
σ_{9621}	AFA	0.000090	0.000095

Table 6.2: Distribution of $p(\sigma)$ and $p_{MCMC}(\sigma)$

In the $9,261$ rows we have all the possible sequences $\sigma_1, ..., \sigma_{9261}$. The values in column

$p(\sigma)$ are calculated using the probability distribution shown in equation 2.1, while the values in the column $p_{MCMC}(\sigma)$ are calculated as

$$p_{MCMC}(\sigma) = \frac{\sum_{i=1}^{n} \delta(\sigma, \sigma_i)}{n} \tag{6.1}$$

where $n = 6,000,000$, and $\delta(\sigma, \sigma_i)$ is the Kronecker delta that is defined to be 1 if $\sigma = \sigma_i$ and 0 otherwise. $p_{MCMC}(\sigma)$ is simply the number of times a certain sequence σ is generated, divided by the total number of generated sequences. As we can already see from the table, $p(\sigma)$ and $p_{MCMC}(\sigma)$ are pretty much aligned, which means that the sampling algorithm is behaving as expected, and that the Markov Chain is converging to the underlying distribution.

We can have a better view on this by plotting $p(\sigma)$ and $p_{MCMC}(\sigma)$ in the same plot.

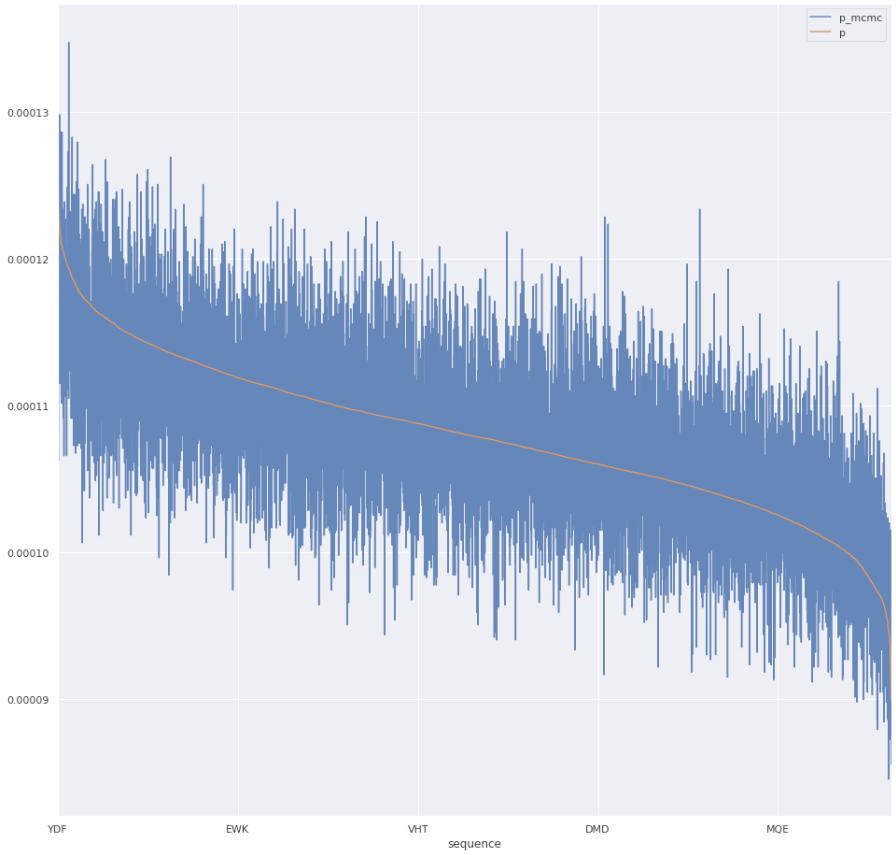

Figure 6.1: Distribution of $p(\sigma)$ and $p_{MCMC}(\sigma)$

On the x axis we have each of the $9,261$ sequences of length 3. The orange line is the true distribution, calculated for each sequence using equation 2.1. The blue line, instead, represents the frequency at which each sequence is generated using the algorithm described in section 3.2. As we can see, p_{MCMC} follows the distribution of p: proteins with a lower p have also a lower probability of being generated. This means that the system is healthy and works as expected; we can therefore proceed with the next steps.

6.3 MLP training

Since we passed the consistency check, we can fit the MLP the simple system described in section 4.1: an input layer of $6,930$ neurons, a hidden layer with 100 neurons and a single output neuron. We start our training by randomly selecting $1,000$ protein sequences from the beta-lactamase TEM-1 family. These will be our positive samples. The negative samples are sampled following the methodology described in Appendix A, by running $n = 3,000,000$ iterations and with $StepSize = 3,000$. This way, we will have the same number of positive and negative samples that we will feed to the neural network.

The parameters ϑ of the network E_ϑ are updated by performing a usual gradient descent

$$\vartheta^{t+1} = \vartheta^t - \lambda \nabla_\vartheta \mathcal{L} \tag{6.2}$$

where our loss function $\mathcal{L}$ and its gradient are defined in section 5.3. The calculation of the gradient of the loss function requires the calculation of the expected value of the gradient of the network $\mathbb{E}_{\mathbf{x}^+}[\nabla_\vartheta E_\vartheta(\mathbf{x}^+)]$ and $\mathbb{E}_{\mathbf{x}^-}[\nabla_\vartheta E_\vartheta(\mathbf{x}^-)]$. We will compute this expectation simply as the sum of the gradients of each parameter divided by the total number of samples

$$\mathbb{E}_{\mathbf{x}}[\nabla_\vartheta E_\vartheta(\mathbf{x})] = \frac{\sum_{i=1}^{n} \nabla_\vartheta E_\vartheta(x_i)}{n} \tag{6.3}$$

The learning rate λ in equation 6.2 is set to 0.001.

After the first iteration, we are moving from a model that randomly assigns energy to a protein sequence to a model that is supposed to assign lower energy (and, in turn, higher probability) to the positive samples. However, since neural networks learn well if they learn slowly, one iteration is of course not enough. We need to repeat the process described in this section multiple times to train our model correctly. In order to do so, we need to begin our second iteration by sampling a new set of positive and negative samples. Regarding the positive samples, we can just randomly select another $1,000$ sequences from our database and feed them to the network. As for the negative samples, instead, we have two alternatives: the first option is to take the last sequence that was generated in the first iteration, and use it as new input to feed to the algorithm described in Appendix A, in order to generate a new set of positive samples. The second choice is, instead, to sample the negative sequences starting from a new real sequence. This second possibility is called *contrastive divergence* [19] and it should be a better solution in our case: our sampling algorithm depends on our energy function $E(x)$, which gets better and better after each epoch. This means that, not only will the model be able to distinguish the positive samples from the negative ones, but it will also improve at sampling sequences that are distributed according to the underlying distribution. The contrastive divergence also gives us the possibility to parallelize the algorithm, by running several chains with different training sequences as starting point.

Summing up, this is how the training algorithm works:

1. randomly select $1,000$ sequences belonging to the beta-lactamase TEM-1 family (positive samples), and sample $1,000$ negative samples by using a real sequence as starting point and following the algorithm described in section 3.2;

2. encode each sequence as described in section 2.3 and pass them through the neural network in order to calculate their energy;

3. calculate the gradient and update the parameters as shown in equation 6.2;

4. repeat the process, by sampling the negative samples using a new real sequence as starting point. The parameters of the energy function used to calculate the acceptance probability $p = \exp\{-\Delta E\}$ change after each iteration.

6.4 Training results

As mentioned in section 5.3, the model works well if it is able to assign high probability to the real sequences. Since $p_\vartheta(x) = \frac{\exp\{-E_\vartheta(x)\}}{\sum \exp\{-E_\vartheta(x)\}}$, sequences with high probability will have lower energy. If we take a random multi-layer perceptron, and we feed it with real and random sequences, this is what we get:

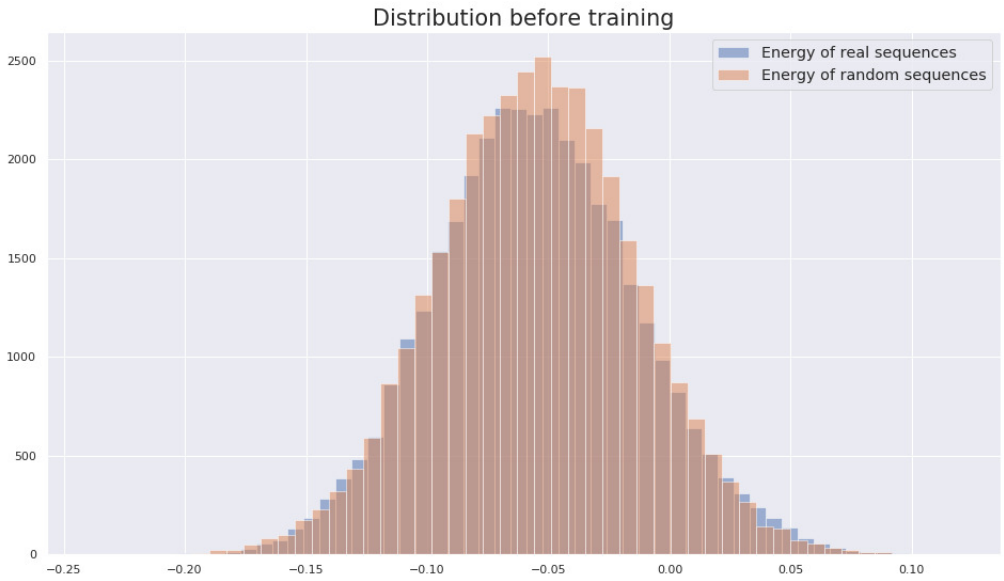

Figure 6.2: Energy of real and random sequences before training the model

The red histogram represents the distribution of the energy assigned to random sequences, whereas the blue one shows the distribution of the energy assigned to the real proteins. As we can see, the two distributions overlaps. If we calculate the energy of a protein sequence using this random network, we will not be able to say if the sequence is real or not. This is,

instead, what the model looks like after 100 epochs.

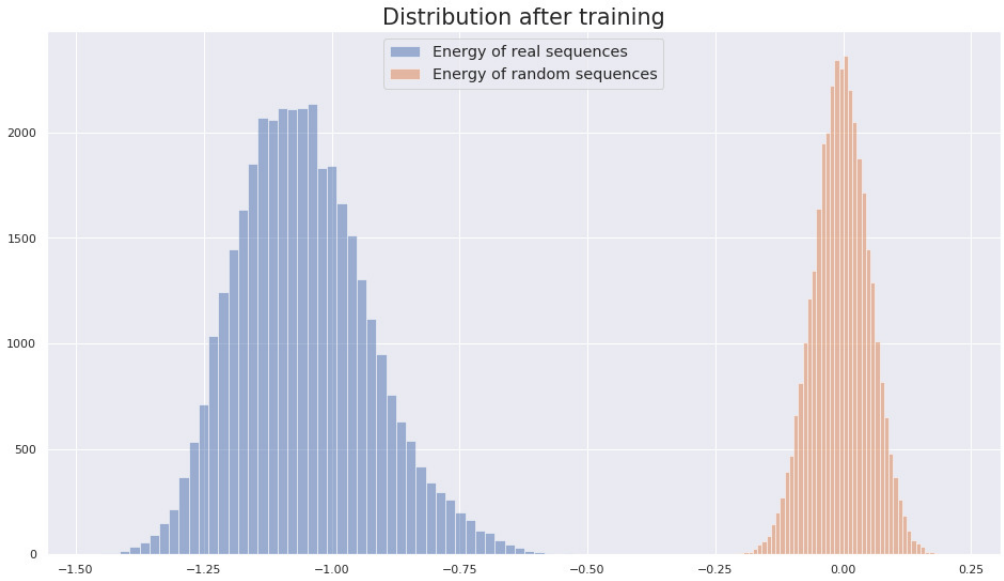

Figure 6.3: Energy of real and random sequences after training the model

After a hundred iterations, the model learned what is the amino acid sequence of a real protein and what is just a random sequence of amino acids. Of course, being able to distinguish real sequences from random sequences does not mean that the model will be able to perfectly predict the effect of a genetic mutation on a real sequence, which is a much harder task. We need to perform some additional analyses on real protein sequences in order to evaluate the true performances of our model.

7 Results

In order to check whether we can use this energy function to predict the effect of mutations, we can run a similar analysis to what Figliuzzi et al. have done in their paper Coevolutionary landscape inference and the context-dependence of mutations in beta-lactamase TEM-1 [8]. They use a different model for predicting the effects but, since we are anyway dealing with the same protein family, we can follow a similar procedure.

As mentioned at the beginning of this paper, the MIC (Minimum Inhibitory Concentration) values tell us how well the protein works, since beta-lactamase is involved in making the bacterium resistant against antibiotics. Some mutations in the protein sequence make the bacterium less resistant against antibiotics and, if this happens, the result of the mutation will be a negative MIC score, which means that the protein has lost some of its power. On the other hand, given the way our model is built, the energy difference calculated by our model should be positive if the mutation is bad for the protein. In conclusion, we can say that our model works as intended if we observe a negative correlation between the MIC scores calculated in laboratory and the energy difference provided by our model.

7.1 Preprocessing the real sequences

PNAS, Proceedings of the National Academy of Sciences of the United States of America, is one of the most famous scientific journals, and it belongs to the United States National Academy of Sciences. On their website, we can find a dataset containing some statistics that explain the result of different genetic mutations. This dataset is the result of an analysis conducted by Herve Jacquier et al., which is well described in their publication Capturing the mutational landscape of thebeta-lactamase TEM-1 [12]. They start from a real sequence belonging to the beta-lactamase TEM-1 family (more specifically, the protein in question is the Escherichia coli - Q6SJ61_ECOLX), they mutate one of the amino acids that compose

the protein, and they record some statistics regarding the result of such mutation. Here is a sample of the resulting dataset:

Residue	Wt_AA	Mutant_AA	Mutation	MIC_Score	ΔΔG_foldX	ΔΔG_PopMusic
34	A	D	A34D	−5.321928	2.119122807	2.01
34	A	G	A34G	−1	2.052719298	2.22
34	A	V	A34V	0	−1.013859649	0
35	E	D	E35D	−2.160964	1.346754386	1.21
35	E	V	E35V	−2.321928	2.414298246	−0.32
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 7.1: Sample outcome of genetic mutations

If we consider the first row, for example, mutation $A34D$ means that the amino acid A (Alanine) in position 34 in the wild type sequence has been replaced by amino acid D (Aspartic acid). The three columns on the right contain some statistics related to the outcome of such mutation. As mentioned, we are only interested in the column containing the MIC scores. In particular, this mutation shows a MIC score of -5.321928, which means that the protein has slightly lost its ability to make bacteria resistant against antibiotics.

In order to check if the result of our model is consistent with what we observe from this data, we need to take the same wild-type sequence and feed it to our neural network. The sequence is available in FASTA format on UniProt (Universal Protein), a freely accessible database of protein data, created by combining the SwissProt, TrEMBL and PIRPSD databases. The sequence we get from this database is not aligned, and the alignement process takes some additional steps that are described in Appendix B. Once the sequence is successfully aligned, we can feed it and its mutations to our model and evaluate the performances.

7.2 Results on real sequences

Once the wild-type sequence has been aligned, we just need to calculate its energy and the energy of the mutations. For each mutation we then calculate the energy difference between the original sequence and the mutated one. For instance, in the case of the $A34D$ mutation, we first calculate the energy of the wild-type sequence $E(\sigma)$, then we replace the amino acid A in position 34 with the amino acid D, and we calculate the energy of this new sequence $E(\sigma')$. Finally, we calculate the energy difference simply as $\Delta E = E(\sigma) - E(\sigma')$. We repeat this process for all mutations and, in the end, our table will look like this:

Residue	Wt_AA	Mutant_AA	Mutation	MIC_Score	Energy_Difference
34	A	D	A34D	-5.321928	-0.006542
34	A	G	A34G	-1	0.002583
34	A	V	A34V	0	0.002220
35	E	D	E35D	-2.160964	-0.001561
35	E	V	E35V	-2.321928	-0.005803
⋮	⋮	⋮	⋮	⋮	⋮

Table 7.2: Model prediction of the outcome of genetic mutations

The final step is the calculation of the correlation between the MIC scores and the energy differences. We will use the Spearman's rank correlation coefficient, or Spearman's ρ, which assesses how well the relationship between two variables can be described using a monotonic function, even if their relationship is not linear. In order to calculate this coefficient, we first need to rank the two variables that are the object of our study (MIC_Score and Energy_Difference, in the table). We rank the highest value in each column as 1, and we assign the lowest rank to the lowest value. After ranking the two columns, we calculate the

Spearman's rank correlation coefficient ρ with the following equation:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{7.1}$$

where d_i is the the difference between the ranks of the two variables in each observation, and n is the number of observations.

Our dataset consists of 239 observations and, by computing the calculations shown in equation 7.1 we obtain a Spearman's rank correlation coefficient of -0.26. The negative correlation shows that, when the MIC score decreases, our model recognizes that the mutation is bad for the protein, and it predicts that the mutation will result in an increase in the energy of the protein. As described in section 5.3, the model was trained to assign low energy to the positive samples (the real sequences) and high energy to the negative ones (the sequences sampled using the MCMC algorithm described in Appendix A).

Given the relative simplicity of our model, a rank correlation of -0.26 is a good result. We must not forget that our neural network has $6,930$ input neurons and one single hidden layer of just 100 neurons. This result shows that even a simpler model is able to capture a good signal from the data. Of course the size of the network is not large enough that one would expect it to be competitive with state-of-the-art models, but getting a significant correlation means that the model is working, and it opens the doors for future work. A more complex model will allow to reach an even more significant result.

7.3 Significance test

In order to complete this work, we need to make sure that the result we obtained is statistically significant. In other words, we need to show that the this result does not simply come from a lucky correlation that we would have obtained even with a random model. In order to

do so, we can compute a permutation test for significance: we randomly permutate the column containing the MIC scores in our dataset, and we calculate the rank correlation for each permutation. This, intuitively, shows how likely it is to obtain a correlation of -0.26 by randomly assigning a score to each genetic mutation. By doing so, we are also able to empirically calculate the p-value of our model. The p-value, in fact, is the probability of observing values that are more extreme than our test statistics.

By running a million permutations, we can check how many times we get a correlation greater than -0.26 in absolute magnitude, and have an estimate of our p-value.

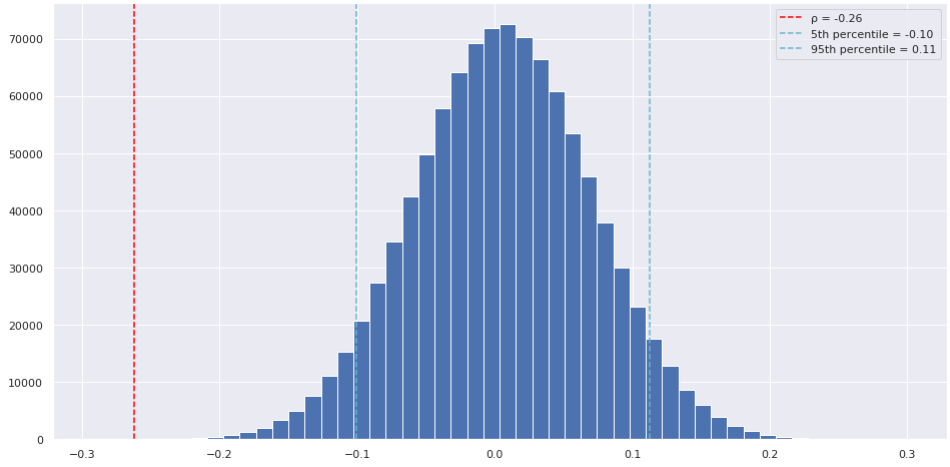

Figure 7.1: Statistical significance of the model

The histogram shows the rank correlation that results from each permutation. As expected, most of the realizations lie around 0, since we do not expect any correlation coming from a random model. The light blue dashed lines represent the fifth and the ninety-fifth percentile of the correlations resulting from the permutations, while the red vertical line is the realization of $\rho = -0.26$ that comes from our model. Out of the one million random permutations we computed, only 20 showed a correlation that was greater than -0.26 in absolute value. This means that our experimental p-value is $p = 2 \cdot 10^{-5} \simeq 0$, and this ultimately proves the statistical significance of our model.

8 Concluding remarks

Our dissertation has come to an end. We started this work from the very basics, by learning what proteins are, what they are made of, and how they work. We then tried to model them, by assigning a probability distribution to each protein, which could give us a measure of fitness. As we have seen, this is not an easy task, and many complex models have been developed by the research world in order to achieve this result. One of the goals of this work was to show that data science, machine and deep learning could play a crucial role in this, and that significant results can be achieved even with a relatively simpler model.

In order to gain resistance against antibiotics, real bacteria are under evolutionary pressure, which means that the only data we have comes from the proteins that survived. This arguably means that the proteins we have at our disposal and that we can analyze are the ones that are in good shape and well functioning. This, on the other hand, does not give us the possibility to train a relatively simpler supervised model by just fitting good and bad sequences to a neural network. We overtook this issue by developing a Markov Chain Monte Carlo sampling algorithm in order to generate protein sequences that come from the same underlying distribution.

We then trained our actual model by feeding real and generated sequences to a neural network, which learned to assign a different score to the real and the generated ones. This is again based on the assumption that, if a protein does not exist in nature, this is probably because its characteristics would not allow it to survive. After showing some promising intermediate results on the performances of the model, we ultimately tested it on real protein sequences. We showed that the outcome of our model is statistically significantly correlated with the results that come from real experiments performed in laboratory.

Given the relative simplicity of our model, this work is a successful proof of principle, and it provides many areas of improvements. More computational power and more availability

of resources will allow to build more complex deep learning models. More hidden layers and neurons will increase the number of parameters and will presumably improve the results of the model.

As shown, this result is a great starting point, and it gives us high hopes for what can be achieved in the future. Genetic mutations are often linked to human diseases, and many of these diseases can be responsible for death. Detecting potentially pathogenic mutations is a first step to prevent such diseases and to save lives. This work shows that statistics, biology, medicine, and all the different scientific fields can collaborate with each other to fight for and defend people's health. The world will never stop fighting for human health, and this works wants to add a tiny piece to the puzzle of the battle for life.

References

[1] B. Alberts, A. Johnson, J. Lewis, D. Morgan, M. Raff, K. Roberts, and P. Walter. *Molecular Biology of the Cell*. Garland Science, 2015.

[2] M. D. Cifarelli, C. Gigliarano, and M. Cozzi. Metodi statistici per l'economia. April 2016.

[3] S. Cocco, C. Feinauer, M. Figliuzzi, R. Monasson, and M. Weigt. Inverse statistical physics of protein sequences: A key issues review. 2017.

[4] X. Ding, Z. Zou, and C. III. Deciphering protein evolution and fitness landscapes with latent space models. *Nature Communications*, 10:5644, December 2019.

[5] Y. Du and I. Mordatch. Implicit generation and generalization in energy-based models. March 2019.

[6] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12:2121–2159, July 2011.

[7] C. Feinauer and M. Weigt. Context-aware prediction of pathogenicity of missense mutations involved in human disease. January 2017.

[8] M. Figliuzzi, H. Jacquier, A. Schug, O. Tenaillon, and M. Weigt. Coevolutionary landscape inference and the context-dependence of mutations in beta-lactamase tem-1. *Molecular Biology Evolution*, 33, August 2015.

[9] R. FISHER. Professor karl pearson and the method of moments. *Annals of Human Genetics*, 7:303 – 318, April 2011.

[10] G. Hinton, N. Srivastava, and K. Swersky. Overview of mini-batch gradient descent, lecture 6a coursera class, neural networks for machine learning.

[11] T. Hopf, J. Ingraham, F. Poelwijk, C. SchÃrfe, M. Springer, C. Sander, and D. Marks. Mutation effects predicted from sequence co-variation. *Nature Biotechnology*, 35, January 2017.

[12] H. Jacquier, A. Birgy, H. Nagard, Y. Mechulam, E. Schmitt, J. Glodt, B. BerÃ§ot, E. Petit, J. Poulain, G. Barnaud, P.-A. Gros, and O. Tenaillon. Capturing the mutational landscape of the beta-lactamase tem-1. *Proceedings of the National Academy of Sciences of the United States of America*, 110, July 2013.

[13] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2014.

[14] P. Mckinnon and S. Davis. Pharmacokinetic and pharmacodynamic issues in the treatment of bacterial infectious diseases. *European journal of clinical microbiology infectious diseases : official publication of the European Society of Clinical Microbiology*, 23:271–88, May 2004.

[15] G. Mustatea, E. Ungureanu, and E. Iorga. Protein acidic hydrolysis for amino acids analysis in food -progress over time: A short review. 26:81–88, April 2019.

[16] A. Riesselman, J. Ingraham, and D. Marks. Deep generative models of genetic variation capture mutation effects. *Nature Methods*, 15, October 2018.

[17] J. Ruiz. Etymologia: Tem. *Emerging Infectious Diseases*, 24:709 – 709, 2018.

[18] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. *30th International Conference on Machine Learning, ICML 2013*, pages 1139–1147, January 2013.

[19] R. Turner. Cd notes. 2005.

[20] M. Welling and Y. Teh. Bayesian learning via stochastic gradient langevin dynamics. pages 681–688, January 2011.

Appendix A: Protein sequence sampling algorithm

Algorithm 1: Protein sequence sampling algorithm

Inputs : $\sigma = (a_1, ..., a_{i-1}, a_i, a_{i+1}, ..., a_{330})$: protein sequence belonging to the beta-lactamase TEM-1 family

E: energy function

n: number of iterations

StepSize: store a newly generated sequence every *StepSize* iterations

Output: $\boldsymbol{\sigma'} = (\sigma'_1, ..., \sigma'_n)$: list of protein sequences generated starting from σ

repeat n times

sample a random position $i \in [0, 330]$

sample a random amino acid

$b \in \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y, -\}$, $b \neq a_i$

$\sigma' = \sigma(a_i \rightarrow b) = (a_1, ..., a_{i-1}, b, a_{i+1}, ..., a_{330})$

$\Delta E = E(\sigma') - E(\sigma)$

$p = \exp\{-\Delta E\}$

generate a uniform random number $u \in [0, 1]$

if $u < p$ **then**

  $\sigma = \sigma'$

if $n \mod StepSize = 0$ **then**

  add σ' to $\boldsymbol{\sigma'}$

Appendix B: Sequence alignment procedure

Our model is trained on protein sequences of length 330. When we look at the sequence in FASTA format available on UniProt, we can see that it only has 286 positions. In order to feed the sequence to our model, we first need to align it.

The alignment of the sequence is carried out by a hidden markov model provided by PFAM. In order to run this model, we we need to use a specific software package called hmmer. This can be easily done with one line of code in bash

hmmalign command to align protein sequences

```
$ hmmalign Beta-lactamase.hmm Q6SJ61.fasta
```

`hmmalign` is part of the hmmer software package, and `Beta-lactamase.hmm` is the hidden Markov model for the beta-lactamase family. From the output of this command we can then retrieve the correct aligned sequence.

In the dataset shown in Table 7.1, the residue indicates the position in the unaligned sequence. The aligned sequence might in fact not have an amino acid A in position 34. We therefore need to create a mapping between the aligned and unaligned positions. In order to correctly per form the mutation, we take for example the $A34D$ mutation, we check which aligned position the unaligned position corresponds to, and we mutate the aligned sequence. In this example, the amino acid A in position 34 in the unaligned sequence corresponds to the amino acid A in position 4, in the aligned one.

Index