GROUP 2302 – Restricted Boltzmann Machines (RBM)

Andrea Marchetti, Luca Menti, Giovanni Merlin, and Alberto Saretto
(Dated: April 2, 2023)

This report explores the application of Restricted Boltzmann Machines (RBMs) to the recognition of patterns in binary data sequences. In particular, we want to circumvent the possible presence of noise in the dataset, training the machine to classify the data correctly anyway. We employed the log-likelihood as the cost function and we tried to improve the performances of the machine in various ways, such as considering different minimisation algorithms (SGD and Adam) or varying the number of contrastive divergence steps. Our findings demonstrate that the RBM model can accurately discriminate binary data sequences according to the pattern they present, even in the presence of noise.

INTRODUCTION

In many scientific problems one has to deal with data whose structure is a relatively long series of binary figures, representing two different states of whatever physical quantity. Such series can comply with certain rules that allow them to be grouped into different cluster types. An example could be data describing two different types of proteins, made up of alternated blocks associated to polar or apolar behaviour. In particular, such a data will divide into two categories depending on whether each sample starts with a polar or an apolar sequence.

Nevertheless, due to structural or data collection errors, some sequences can be corrupted, making it more difficult to associate them to the correct cluster. Therefore, being able to recognise the sequences even if they present a significant error — eventually being able to denoise them — is of absolute interest for many scientific purposes.

This operation can be realised by means of a machine learning model called Restricted Boltzmann Machine (RBM). RBMs are unsupervised learning techniques that consist of two layers of nodes, visible and hidden, with the nodes of each layer fully connected to the nodes of the other layer, but with no connections between nodes of the same layer [1]. RBMs use a probabilistic approach to model the relationships between the visible and hidden layer.

The aim of this work is therefore to train an RBM to be able to recognise features in a dataset of binary series and hence to correctly cluster them even in presence of a significant amount of noise.

In our case the data consists of a collection of binary sequences (of values in either $\{0, 1\}$ or $\{-1, 1\}$) of 20 figures each. Each sequence is subdivided into blocks of 4 figures, each obtained through one-hot encoding — i.e., only one figure is set to 1 and the others to 0 (or -1). The generation algorithm ensures that sequences fall into one of two types. In the first type, one of the first two cells of the first block is set to 1, chosen randomly with equal probability (hence the first block will appear as 1000 or 0100). In the second type, the same happens for the last two cells (0010 or 0001). The subsequent blocks of the sequence are then generated as follows: if the previous block is 1000 or 0100 they are set to 0010 or 0001 (always chosen randomly and with equal probability), and vice versa.

To this core structure some noise is applied, specifically each block can mutate to any of the four possibilities with a probability of 10%.

The aim is then to discriminate between sequences of the first and of the second type, independently of the random variations caused by noise.

METHODS

We have implemented an RBM from scratch using the Python programming language. The RBM comprises a visible layer with $N_v = 20$ neurons and a hidden layer with $N_h = 6$ neurons, each characterised by a bias: a_i for visible units and b_μ for hidden units. Each visible neuron is linked to a hidden neuron by a weight $W_{i\mu}$. During the training phase, the values of the hidden units are computed from those of the visible units, and vice versa. However, the two layers are calculated through two different methods. Specifically, the values of the hidden neurons are set to 1 according to the following probability:

$$p(h_\mu = 1|v) = \sigma\left(b_\mu + \sum_i W_{i\mu} v_i\right) \tag{1}$$

Here, σ is the activation function, given by the sigmoid:

$$\sigma(x) = \frac{1}{1 + \exp(-gx)} \tag{2}$$

where g is the gap between the low and the high binary value ($g = 1$ if values are in $\{0, 1\}$ and $g = 2$ if they are in $\{-1, 1\}$).

On the other hand, the visible layer must be created such to preserve the one-hot encoding structure [2]. To

do so, after specifying the block's length A and the number of blocks N, we calculate the probability associated with each of the A possibilities for each block ($A = 4$ in our case), and generate it accordingly. Specifically, these probabilities are given by:

$$p(v,h) = \frac{e^{-\beta E(v,h)}}{Z} \tag{3}$$

where E is the energy and Z is the partition function, according to the following definitions:

$$E(v,h) = -\sum_i a_i v_i - \sum_\mu b_\mu h_\mu - \sum_{i,\mu} v_i W_{i\mu} h_\mu \tag{4}$$

$$Z = \sum_{\{v,h\}} p(v,h) = \sum_{\{v,h\}} e^{-\beta E(v,h)} \tag{5}$$

with Z calculated on all the possible configurations of v and h.

The computation makes use of contrastive divergence (CD) i.e., the calculation of the values of the hidden and visible layer is cyclic: at each step, the hidden layer is first calculated from the visible one, and then the visible is calculated from the hidden. If the number of such steps is n, we talk about CD-n.

In particular, the training was carried out by updating weights and biases to maximise the log-likelihood (LL):

$$\log \mathcal{L}(v,h) = -\langle E \rangle_{data} - \log Z \tag{6}$$

where — as before — E is the energy and Z is the partition function.

To make this calculation, we exploited the fact that the derivatives with respect to the parameters are given by:

$$\frac{\partial \log \mathcal{L}}{\partial W_{i\mu}} = \langle v_i h_\mu \rangle_{data} - \langle v_i h_\mu \rangle_{model} \tag{7}$$

$$\frac{\partial \log \mathcal{L}}{\partial a_i} = \langle v_i \rangle_{data} - \langle v_i \rangle_{model} \tag{8}$$

$$\frac{\partial \log \mathcal{L}}{\partial b_\mu} = \langle h_\mu \rangle_{data} - \langle h_\mu \rangle_{model} \tag{9}$$

as stated in [1], where the averages are calculated on both the real dataset (data) and on the dataset generated by the machine (model). In particular, the hidden layer for the real dataset was simply calculated from the visible in one step.

Initially in the program, we defined a function to split the dataset into training and validation sets, which randomly shuffles the data at each call to allow for multiple training sessions with different sets starting from a unique data file. We then defined the RBM architecture, initialising the weights and biases of the visible layer using a Gaussian distribution centred at 0, and setting to zero the bias of the hidden layer. The standard deviation of the Gaussian was chosen as $\sigma = 2/\sqrt{N_v + N_h}$ as suggested in [1]. Additionally, we developed tools to visualise the RBM's parameters at various stages of the training process (Fig. 1).

Next, we created functions to compute the energy, partition function and likelihood of the RBM, and to generate the hidden and visible layer of the network following the previously described procedure. Finally, we designed a training protocol to maximise the likelihood of the RBM using gradient descent.

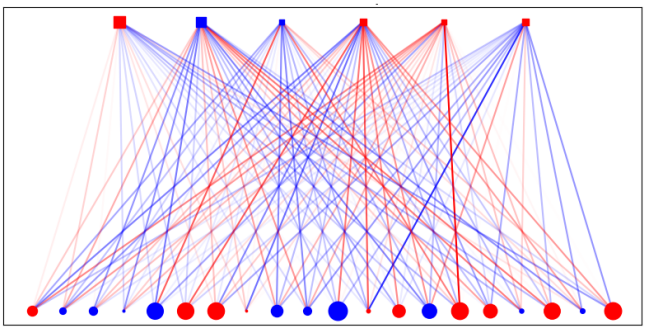

FIG. 1: **Initial RBM's parameters.** The top row corresponds the 6-neuron hidden layer, while the bottom to the 20-neuron visible layer. The dots and squares represent the values of the initial bias: the bigger the marker, the larger the value, with red associated to positive values and blue to negative values. The lines represent the weights: similarly to the markers, larger values are associated to thicker lines, and the colour indicates the sign.

After the definition of the RBM, we trained it in different conditions, employing the likelihood as a quality estimator and comparing its values — calculated on the validation set — obtained from different methods.

In our first test, we investigated the impact of using different input values, specifically values in $\{0, 1\}$ (spins off) or in $\{-1, 1\}$ (spins on). In particular, we trained two different machines on the same training set, but in one case we previously transformed all 0's in -1's.
In the second test, we compared the performance of two different minimisation algorithms, namely Stochastic Gradient Descent (SGD) and Adam, as described in [1]. During the training, the algorithm is repeated for a certain number of epochs, and at the end of each epoch the learning rate is slightly decreased following a power law. Lastly, we evaluated the effect of varying the number of steps of contrastive divergence.

RESULTS

In our study, we employed a dataset of 10,000 entries, which was split into a training set of 7,000 entries and a validation set of 3,000 entries. Moreover, the duration of training procedure was set to 50 epochs, which, as can be seen in all cases below, were sufficient for the log-likelihood to converge.

We first employed SGD on 500-entry minibatches with a learning rate $\eta_0 = 0.002$, which was decreased after every epoch as previously described. The results of the training in these conditions for spins on and off are shown in Fig. 2. We can note that there is no visible difference in the readability of the parameters in the two cases: the choice of values in {-1,1} just translates into weights of greater magnitude.

To quantitatively estimate the effectiveness of the learning, we evaluated the trend of the LL on the training set as a function of the epoch, expecting it to increase as an indicator of the success of the learning. The results are shown in Fig. 3. We can observe how the two cases, despite differing only by the data representation, display a considerable different behaviour: for values in {0,1} the LL steadily increases and tends to converge at a value of about -7, whereas for values in {-1,1} — albeit having a greater value — it tends to fluctuate without increasing, therefore suggesting an ineffective learning. Indeed, when evaluating the performance of the machine on the validation set, we obtained an LL of -11.65 for spins on and of -6.62 for spins off, suggesting a higher effectiveness of the latter configuration. For this reason, in the following tests we made use of the {0,1} encoding.

For the comparison between SGD and Adam, we increased the initial learning rate to $\eta_0 = 0.02$ to speed up the computation. We kept 500-entry minibatches for both minimisation algorithms. The results are shown in Fig. 3. As we can see, with an increased learning rate SGD appears to be less stable than before, but its convergence is not affected. On the other hand, Adam — despite being more stable — does not seem to improve the performances, reaching an LL convergence value lower than the one obtained by SGD. This could point to the fact that the algorithm has stabilised around a local minimum of the LL, from which it cannot escape. Nevertheless, when calculating the LL on the validation set we obtained -5.21 for SGD and -6.74 for Adam, hence suggesting that indeed SGD achieves a better performance.

Lastly, we studied the behaviour of the RBM at different values of the number of CD steps employing both SGD and Adam. In both case we employed 500-entry minibatches and initial learning rate $\eta_0 = 0.02$. The results are presented in Fig. 4. All trainings reached stability within 50 epochs, with SGD being again slightly slower and unstable than Adam, although reaching higher values of LL.

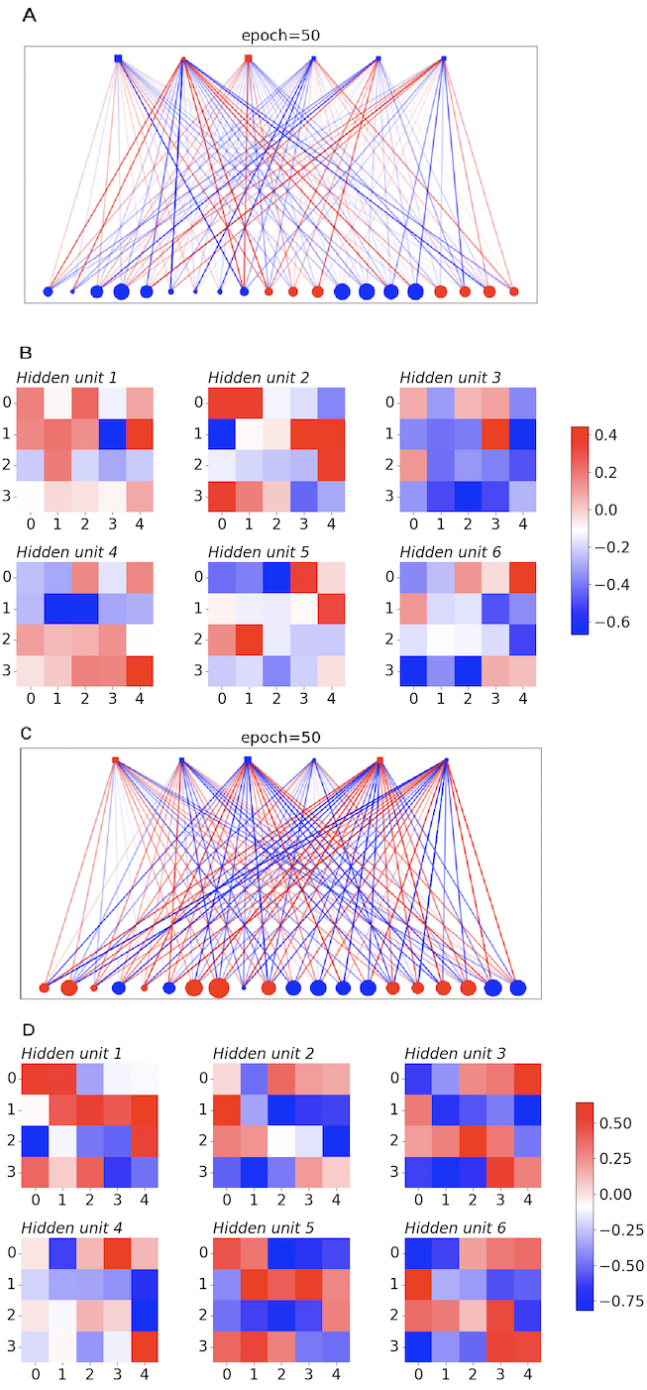

FIG. 2: **Final RBM's parameters.** Visual representation of RBM's parameters after 50 epochs, for spins off (A,B) and spins on (C,D). Panels A, C are as described in Fig. 1. Panels B, D show a colour-coded map of the weights $W_{i\mu}$ connecting the two layers at the end of the training. Each table is associated to a hidden unit, and encodes the relation with all the visible units. Namely, each column represents a block of 4 neurons, and the colour represents the value of the weights (red for positive and blue for negative).

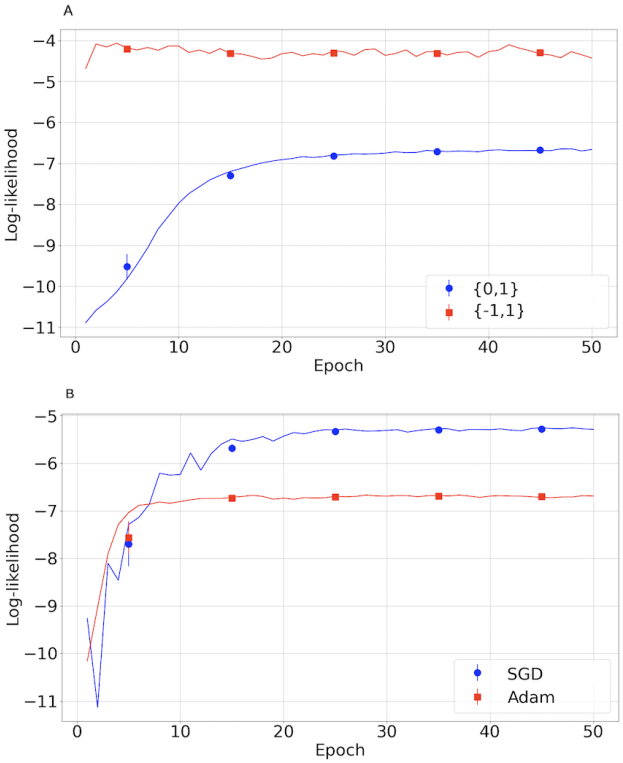

FIG. 3: **Log-likelihood on the training set**. LL as a function of the epoch for (A) spins on and off and (B) SGD and Adam. The values are averaged over windows of 10 consecutive epochs, with mean value and standard error represented by dots and error bars.

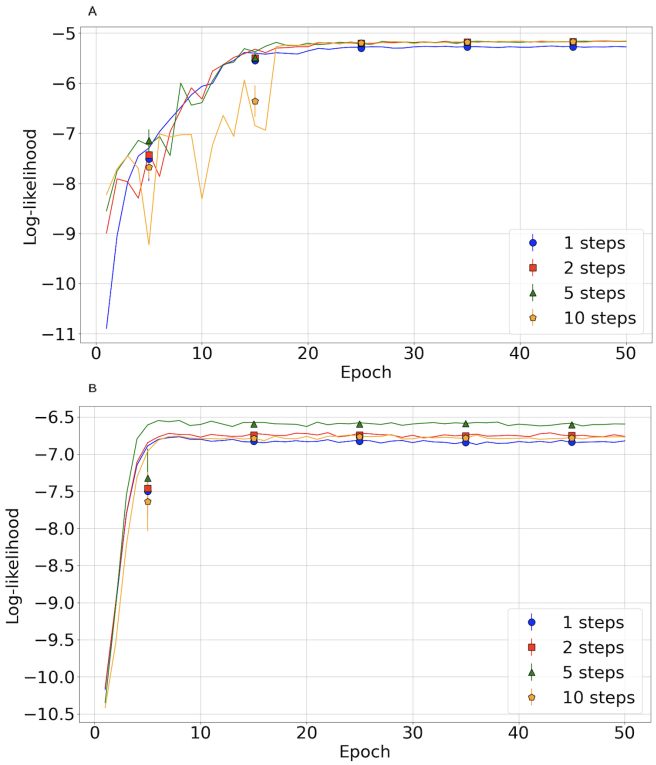

FIG. 4: **Log-likelihood on the training set: CD steps**. Plots of the LL on the training set employing (A) SGD and (B) Adam as a function of the epoch. The colours represent a different number of iterations of CD-n. Markers and bars as in Fig. 3.

In principle, in the limit of infinite iterations, the fantasy data are guaranteed to converge to the equilibrium distribution [1], which corresponds to the maximum ability of the RBM to learn. However, in both cases we obtained results for the LL very similar to each other for all the numbers of steps, both on the training and on the validation set (Tab. I). This result suggests that, in these conditions, a good performance can be obtained just after one or two CD steps, which permits to keep the training of the machine less computationally demanding.

CD steps	1	2	5	10
SGD	-5.30	-5.18	-5.21	-5.19
Adam	-6.75	-6.85	-6.65	-6.87

TABLE I: Log-likelihood on the validation set for various steps of CD, for both SGD and Adam.

CONCLUSIONS

In this report, we implemented an RBM from scratch using the Python programming language to develop an algorithm for binary sequence recognition, even in the presence of noise. We employed the LL as an estimator of the learning quality and initially maximised it through an SGD procedure. In this setup, we first investigated the impact of using different binary input values, specifically in {0,1} or {-1,1}. We discovered that using {0,1} resulted in a higher LL, while with {-1,1} the algorithm probably did not reach convergence. Therefore, we used values in {0,1} for the rest of the analysis. We then implemented the Adam gradient descent algorithm to try to improve RBM's performance, but the results did not show a significant improvement with respect to SGD. Indeed, even if Adam was more stable, the LL value was higher with SGD. Finally, we varied the number of CD steps while considering both Adam and SGD. In both cases, no significant difference among the different cases appeared in the goodness, suggesting that choosing a CD-1 or CD-2 could be a good compromise between RBM's performance and computational demand. The such created machine is able to recognise patterns in binary sequences and to generate new sequences accordingly. In further work, a possible use of the machine could be to denoise corrupted data retrieving the correct original sequence, with particular application to protein datasets.

[1] Mehta *et al.*, A high-bias, low-variance introduction to Machine Learning for physicists, Physics Reports **810**, 1–124 (2019). doi:10.1016/j.physrep.2019.03.001

[2] In principle, the machine could also be able to correctly produce one-hot encoding sequences autonomously, calculating the values of the visible neurons according to a formula analogous to (1). However, in practice this did not happen, and we had to force the sequences to be produced in one-hot encoding.