A MATLAB implementation of a Restricted Boltzmann Machine and its application to a classification task

Beatrice Segalini
(Dated: February 5th, 2021)

Abstract: Restricted Boltzmann Machines (RBMs) are a class of generative models with many useful practical applications, ranging from supervised tasks such as classification, to unsupervised collaborative filtering (e.g. movie recommendations). In this work, the MNIST dataset of handwritten digits is used to train a RBM, coded in MATLAB, for the digit classification task. To do so, the general structure of RBM is defined and the main learning techniques are displayed, focusing especially on SGD methods. The RBM programmed is finally tested and its performance is compared with another Neural Network model.

INTRODUCTION

The emerging importance of Machine Learning in all scientific fields is undeniable. Restricted Boltzmann Machines (RBMs), in particular, sit at the intersection of physics and learning algorithms. They are a generative energy-based model, and can so be used to perform a classification task: a trained RBM, which has learned the general probability distribution of a dataset, can indeed be able to assign the proper label to a general sample belonging to the same type of data.

This is as a matter of fact the purpose of this work, in which a RBM is trained to recognise hand-written digits of the MNIST database.

Thus, RBMs, their core purpose and the main learning mechanism are first defined in section 1.

In section 2, the principal methods for improving the Stochastic Gradient Descent (SGD) algorithm, implemented in the training of the RBM, are displayed.

After this theoretical introduction, in section 3 the architecture of the network with some technical information is presented.

The performances for the classification task of the different RBMs analysed are therefore evaluated and reported in section 4, where the best performing RBM is also compared with a different network architecture.

Finally, conclusions are drawn in section 5.

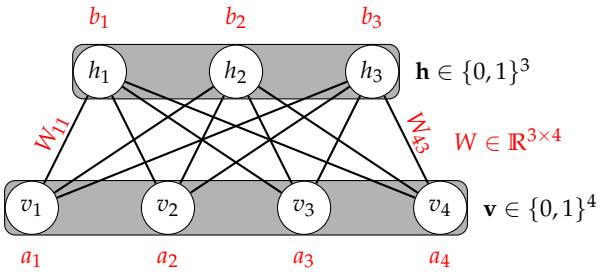

Figure (1) – The bipartite graph of a RBM. $\mathcal{E}(\boldsymbol{v}, \boldsymbol{h})$ is obtained by multiplying each parameter by its respective connected node or nodes, e.g. a_1 with v_1, b_1 with h_1 or W_{11} with $v_1 h_1$, and then summing all contributions. Image taken from [1].

1. RESTRICTED BOLTZMANN MACHINES

Given a dataset $\mathcal{D}$ of N d-binary variables $\{\boldsymbol{x_i}\}_{i=1,\dots,N}$, with $\boldsymbol{x_i} \in \{0,1\}^d$ each sampled from an unknown probability distribution $p(\boldsymbol{x})$, the aim of a Restricted Boltzmann Machine (RBM) [2] is usually to obtain a *good* parametrization $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ of $p(\boldsymbol{x})$.

In general, $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ cannot be factorized as $\prod_{i=1}^{d} p_i(x_i)$, because entries x_j in a sequence $\boldsymbol{x}$ can be highly correlated. This makes the search for the optimal parameters $\boldsymbol{\theta}$ (i.e., the *training*) hard, because changing any θ_j has a global effect on "all" $p_{\boldsymbol{\theta}}(\boldsymbol{x})$.

One possible solution is to consider M additional **hidden** binary variables $\boldsymbol{h}$, learn from data a joint pdf $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{h})$, and then produce samples according to its *marginalization*:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{\boldsymbol{h}\in\{0,1\}^M} p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{h}) \tag{1}$$

The $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{h})$ can be chosen to be factorizable for all $\boldsymbol{x}$ and $\boldsymbol{h}$, i.e. such that:

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{h}) = \prod_{i=1}^{d} p_{\boldsymbol{\theta}}(x_i|\boldsymbol{h}) p_{\boldsymbol{\theta}}(\boldsymbol{h}) \tag{2}$$

However, the resulting marginal distribution $p_{\boldsymbol{\theta}}(\boldsymbol{x})$ will not be, in general, factorizable. In fact, let for simplicity $d = 2$ and $M = 1$. Then marginalizing (2) leads to:

$$\begin{aligned} p_{\boldsymbol{\theta}}(x_1, x_2) &= \sum_{h_1} p_{\boldsymbol{\theta}}(x_1|h_1) p_{\boldsymbol{\theta}}(x_2|h_1) p_{\boldsymbol{\theta}}(h_1) \\ &\neq p_{\boldsymbol{\theta}}(x_1) p_{\boldsymbol{\theta}}(x_2) \sum_{h_1} p_{\boldsymbol{\theta}}(h_1) \end{aligned}$$

Thus, (1) can capture correlations between different x_j, and the property (2) allows a fast learning algorithm, as shown thereafter. In the specific case of a RBM, $p_{\boldsymbol{\theta}}(\boldsymbol{x}, \boldsymbol{h})$ is the Boltzmann distribution [3]:

$$p_{\boldsymbol{\theta}}(\boldsymbol{v}, \boldsymbol{h}) \equiv \frac{1}{Z} \exp(-\mathcal{E}(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta})) \tag{3}$$

$$\mathcal{E}(\boldsymbol{v}, \boldsymbol{h}; \boldsymbol{\theta}) \equiv -\sum_{i=1}^{d} a_i v_i - \sum_{\mu=1}^{M} b_\mu h_\mu - \sum_{i\mu} W_{i\mu} v_i h_\mu \tag{4}$$

$$Z \equiv \sum_{\substack{v\in\{0,1\}^d \\ h\in\{0,1\}^M}} \exp\left(-\mathcal{E}(v,h;\theta)\right) \tag{5}$$

where v is used in place of x to denote the **visible** degrees of freedom, i.e. the ones directly derived from the input data. The **energy** $\mathcal{E}(v,h;\theta)$ can be visually schematised as a bipartite graph (Figure 1). Note that only interactions between a visible and a hidden node are allowed, meaning that the $\{v_i\}$ and $\{h_j\}$ don't interact "with themselves" [4], as prescribed by (2).

Training is done by maximising the "closeness" of $p_\theta(x)$ to $p(x)$, i.e. the average probability $\mathcal{L}$ (*likelihood*) of generating the observed samples $x_i \in \mathcal{D}$ from the parametrised distribution $p_\theta(x)$:

$$\theta\colon \max_\theta \mathcal{L}(\theta) = \max_\theta \langle \log p_\theta(x)\rangle_{\mathcal{D}}$$

To do so **gradient descent** is exploited, starting from a *random* choice of parameters θ_0 and iteratively "tweaking" it in the direction of the gradient $\nabla\mathcal{L}(\theta)$ until convergence, following this **update rule**:

$$\theta_{t+1} = \theta_t + \eta\nabla\mathcal{L}(\theta), \tag{6}$$

where $\eta \in (0,1]$ is the **learning rate**, adjusting how much change is allowed at each iteration. The gradients are given by [5]:

$$\begin{aligned}
&\frac{\partial}{\partial\theta}\mathcal{L} = \langle\frac{\partial}{\partial\theta}\log\big(p_\theta(v,h)\big)\rangle_{\mathcal{D}} = \\
&= \langle\frac{\partial}{\partial\theta}\Big[\log\sum_{g\in\{0,1\}^M}\exp(-\mathcal{E}(v,g;\theta)) - \log Z(\theta)\Big]\rangle_{\mathcal{D}} = \\
&\qquad = \langle G_\theta(v,\theta)\rangle_{\mathcal{D}} - \langle G_\theta(v,\theta)\rangle_{p_\theta} \qquad (7)\\
&G_\theta(x,\theta) = \sum_{h\in\{0,1\}^M} p_\theta(h|x)\frac{\partial}{\partial\theta}(-\mathcal{E}(x,h;\theta))
\end{aligned}$$

$G_\theta(x,\theta)$ are just the gradients of the energy $\mathcal{E}$ "marginalized" over the hidden units h. So (7) is directed towards values of θ for which the energy-gradients point on average in the same direction, both for $p(x)$ and $p_\theta(x)$.

Physically, gradient descent is adjusting the energy of configurations x, so that the ones *close* to $x_i \in \mathcal{D}$ have lower energy.

Substituting (4) in (7) leads to:

$$\begin{aligned}
G_{W_{ij}}(v;\theta) = v_i\cdot p_\theta(h_j=1|v);\quad G_{a_i}(v;\theta) = v_i\\
G_{b_j}(v;\theta) = p_\theta(h_j=1|v)\\
p_\theta(h_j=1|v) = \sigma\underbrace{\Big(b_j + \sum_{k=1}^d v_k W_{kj}\Big)}_{A_j} \qquad (8)
\end{aligned}$$

where σ is the **sigmoid** function. So $\langle G_\theta(v,\theta)\rangle_{\mathcal{D}}$ (the so-called *positive* phase) can be computed by setting $v = x_i$, calculating the hidden unit activations A_j, setting $h_j = 1$ with probability $\sigma(A_j)$, and finally averaging over $\mathcal{D}$. The second term, $\langle G_\theta(v,\theta)\rangle_{p_\theta}$ (the *negative* phase) cannot be directly evaluated in general, because $Z(\theta)$ (5) has no analytic form, and requires 2^{d+M} evaluations to be numerically computed, which usually is not feasible. So, **Gibbs sampling** is instead used. Thanks to (2):

$$p_\theta(v|h) = \prod_{i=1}^d p_\theta(v_i|h);\quad p_\theta(h|v) = \prod_{\mu=1}^M p_\theta(h_\mu|v)$$

This allows to set $v^{(0)}$ to x_i, sample $h^{(0)}$ from $p(h|v^{(0)})$, and then $v^{(1)}$ from $p(v|h^{(0)})$. If this process is reiterated for an infinite time, $v^{(\infty)}$ will be a "fantasy" sample generated by $p_\theta(x)$, that can be used for calculating the average $\langle G_\theta(v,\theta)\rangle_{p_\theta}$.

Computationally, only k back-and-forth steps are done, leading to the **Contrastive Divergence** CD-k technique. When gradients are computed over minibatches of data, as in **Stochastic Gradient Descent**, $k = 1$ usually suffices, as training can tolerate some noise in the gradients. That is also the case of the RBM chosen for the classification task presented in this work.

2. STOCHASTIC GRADIENT DESCENT TECHNIQUES

In this paper, the basic implementation of a RBM is improved by using come common Stochastic Gradient (SGD) techniques in order to make the learning process easier and faster, and to improve the performances of the learned model in the classification task. These procedures are also necessary to ensure the training convergence: in fact, since the CD-1 technique is chosen for the learning process, the approximation error due to the truncation of the Gibbs sampling chain can lead to divergence and to wrong parameters estimates. The main methods implemented are the **momentum** and the **weight decay**.

Momentum

One of most commonly implemented techniques for speeding the learning process is adding *momentum* in the parameters update phase. Instead of using only the estimated gradient times the learning rate, as shown in (6), the momentum method completes the update rule with an additional term, regularised by the hyper-parameter $\alpha \in (0,1]$:

$$\theta_{t+1} = \theta_t + \eta\nabla\mathcal{L}(\theta) + \alpha\Delta\theta_{t-1}. \tag{9}$$

In this way, the updated parameters depend on both the gradient and their change at the previous iterative step.

As one can deduce, this method is particularly effective in speeding the learning process when the profile of the target function is characterised by long, narrow valleys. In this situation, steepest descent methods are particularly slow, since the system is forced to oscillate back and forth on the direction orthogonal to the main

axis of the valley, moving towards convergence with very little steps. The momentum term helps average out the oscillation, forcing the update to move not towards the direction of steepest descent, while at the same time adds up contributions along the long axis [6].

The effect of the momentum is particularly significant at the beginning of the learning process, where the random initialisation of parameters may lead to big fluctuation in the updates of weights and biases and generally gradients are bigger. As a consequence, smaller values of α are preferred. After some iterations, the training process stabilise and hence increasing the value of the momentum parameter is necessary to speed up the learning [7].

Weight-decay and error computation

As already stated, CD$-k$ algorithm naturally produces an approximation error due to the truncation at the k-th step (with $k = 1$ in the examined case). This can lead to a distortion of the learning process and to divergence, with the network predicting a model whose parameters do not reach the maximum likelihood.

However, this effect can be prevented by introducing an additional term in the update rule for weights update, the so-called **weight-decay** term:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \eta \boldsymbol{\nabla} \mathcal{L}(\boldsymbol{w}) + \alpha \Delta \boldsymbol{w}_{t-1} - \lambda \boldsymbol{w}_t. \qquad (10)$$

The extra term is the derivative of a function that penalises large weights, the *L2 cost function*, and it helps not only to solve the divergence problem, but also to prevent overfitting and to ease weights interpretation, by eliminating large weights which warp the visual representation at the end of the learning process.

The L2 metric is also used to compute the training error for each training epoch, simply calculating the norm of the difference between the reconstructed samples and the original ones, normalised by the number of samples.

3. RBM ARCHITECTURE

Main parameters initialisation

Given the structure of the input data, the number of visible units `Ni` of the RBM will always be fixed to 784, that is the number of pixels in any sample taken from the MNIST database.

Besides, the training will be always performed on 60000 digits taken from the so-called **training set**, while the final testing is carried out on the 10000 digits of the **test set**. Since the number of data is not negligible, to speed up the process the samples are divided into smaller chunks of data (*minibatches*), on which the CD-1 algorithm is implemented in parallel. This leads to a better gradient evaluation and significantly improves the execution time. For all the following implementations, the `batch_size` is set to 600.

For some of the other parameters initialisation, the guide [7] is taken as reference. In fact:

- both hidden and visible unit's biases are initialised as all equal to zero;
- initial values of the weights are drawn from a normal distribution with zero-mean and standard deviation equal to 0.01;
- the momentum parameter, α, is set to 0.5 for the first 15 epochs, then to 0.9 for the rest of the training;
- the maximum number of training epochs, `max_epoch`, is fixed to 100.

However, the RBM structure is yet to be defined. Indeed, three core parameters are still missing a proper initialisation: the number of hidden units `Nhidden`, the learning rate η and the weight-decay rate λ.

The most important one is `Nhidden`, which defines the architecture of the network and the number of parameters to be optimised in the learning process. To study different RBM structures, `Nhidden` will be changed between 16, 100 and 196.

For each of these architectures, the learning rate η is varied between 0.01, 0.001 and 0.0001. In addition, taking as inspiration the study on the convergence of the CD-1 algorithm in [8], the weight-decay parameter λ is chosen from the following set: $[10^{-5}; 5 \cdot 10^{-5}; 10^{-4}]$. In fact, λ cannot assume a much bigger value or the weights will be completely annihilated after the update phase, and thus the final prediction given by the network would be distorted.

Softmax layer

Finally, after having trained the RBM, the classification task is left to a **Softmax** layer, defined briefly in this section and more in detail in [9], [10].

Given some training samples $\boldsymbol{x}$ to be classified into k classes, the Softmax Regression model first computes a score $\boldsymbol{s}(x)$ for each class k:

$$s_k(\mathbf{x}) = \mathbf{x}^\top \boldsymbol{\theta}^{(k)} \qquad (11)$$

where $\boldsymbol{\theta}^{(k)}$ is the parameter vector of class k. Then, by maximising the quantity (12):

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp\left(s_k(\mathbf{x})\right)}{\sum_{j=1}^{K} \exp\left(s_j(\mathbf{x})\right)}, \qquad (12)$$

the Softmax Regression classifier is able to predict the class k with the highest estimated probability for $\boldsymbol{x}$, given the scores of each class for that sample.

Minimising the *cross entropy* (13) is the most common way to perform such task, as doing so penalises the models when it estimates a low probability for a target class.

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right) \tag{13}$$

In (13), $y_k^{(i)}$ represents the target probability that the i-th sample belongs to class k.

The Softmax Regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as the MNIST digits, making it perfect for the purpose of this implementation.

Therefore, the MATLAB `trainSoftmaxLayer` function is used with the default cross-entropy loss function and the maximum number of iteration fixed to 1000.

The digits and their labels, properly encoded with the sigmoid function and the RBM parameters, are set as input for training the Softmax layer, which then is able to assign new labels as output.

4. DISCUSSION

Choosing the best network

In this section, the best set of parameters [`Nhidden`, η, λ] for the classification task is assessed. The goal is to obtain a network that is able to assign the proper label to any random handwritten digit.

To evaluate the quality of the results, the **accuracy** is defined as the percentage of correctly classified samples over the total number of samples of the dataset and it is computed for both training and test set.

In the following, the performances are analysed using such metric. In table (I), (II), (III), one can find the accuracy result over the several network and parameter configurations, for both training and test set.

It is important to state that, since each run took approximately 15 minutes to produce the accuracy results, only one repetition of training loop and accuracy evaluation is performed for each combination of parameters. Of course, this is just a rough estimate used to understand the general properties of the network, since the training process is in any case subject to random fluctuations. As a consequence, these results are just broadly taken as reference to understand the overall behaviour for each set of parameters.

After this preliminary consideration, it is now possible to notice that the number of hidden units heavily impacts on the performances: in fact, for `Nhidden`$= 16$, none of the parameter combinations reach good results, failing to properly recognise the digits $\approx 40 - 20\%$ of the times, if not more.

On the other hand, the two bigger networks show outstanding results, with accuracy levels between 90.7% and 94.1% for `Nhidden`$= 100$, and between 92.5% and 95.8% for `Nhidden`$= 196$.

Table (I) – Train and test classification accuracy, for each combination of λ, η, with `Nhidden`$= 16$.

`Nhidden`$= 16$	$\eta = 0.01$	$\eta = 0.001$	$\eta = 0.0001$
$\lambda = 10^{-5}$	**Train:** 52.5% **Test:** 51.9%	**Train:** 68.6% **Test:** 68.9%	**Train:** 70.4% **Test:** 74.8%
$\lambda = 10^{-4}$	**Train:** 62.8% **Test:** 63.7%	**Train:** 68.8% **Test:** 62.8%	**Train:** 74.8% **Test:** 76%
$\lambda = 5 \cdot 10^{-4}$	**Train:** 73.6% **Test:** 74.2%	**Train:** 77.6% **Test:** 77.6%	**Train:** 78.1% **Test:** 78.4%

Table (II) – Train and test classification accuracy, for each combination of λ, η, with `Nhidden`$= 100$.

`Nhidden`$= 100$	$\eta = 0.01$	$\eta = 0.001$	$\eta = 0.0001$
$\lambda = 10^{-5}$	**Train:** 90.7% **Test:** 90.5%	**Train:** 93.1% **Test:** 93.2%	**Train:** 93.3% **Test:** 92.9%
$\lambda = 10^{-4}$	**Train:** 94.1% **Test:** 94.1%	**Train:** 93% **Test:** 93.1%	**Train:** 93% **Test:** 92.7%
$\lambda = 5 \cdot 10^{-4}$	**Train:** 93.5% **Test:** 93.4%	**Train:** 92.9% **Test:** 93.2%	**Train:** 93.4% **Test:** 92.9%

Table (III) – Train and test classification accuracy, for each combination of λ, η, with `Nhidden`$= 196$.

`Nhidden`$= 196$	$\eta = 0.01$	$\eta = 0.001$	$\eta = 0.0001$
$\lambda = 10^{-5}$	**Train:** 92.5% **Test:** 92.6%	**Train:** 94.9% **Test:** 94.4%	**Train:** 95.1% **Test:** 94.4%
$\lambda = 10^{-4}$	**Train:** 94.8% **Test:** 94.5%	**Train:** 95.2% **Test:** 94.7%	**Train:** 95.1% **Test:** 94.3%
$\lambda = 5 \cdot 10^{-4}$	**Train:** 95.8 % **Test:** 95.3 %	**Train:** 95% **Test:** 94.5%	**Train:** 95.1% **Test:** 94.2%

One can notice that generally the accuracy value for the test set is lower than the one of the training set (as expected) and that they do not heavily depend on the choice of η, λ, although a certain growing tendency in the accuracy value can be observed when η decrease and λ increase (moving from the left to the right, from top to bottom of the tables). Nevertheless, this trend seems to be reverted for the last line of both tables (II), (III): in fact, when η decrease, the network, despite still predicting correctly over 93% of the labels, seems to be less effective.

In conclusion, the best set of parameters is the one highlighted in Table (III):

- `Nhidden`$= 196$;
- $\eta = 0.01$;
- $\lambda = 5 \cdot 10^{-4}$.

This final network configuration is retrained with more training epochs (`max_epoch`= 500) and its final results are displayed in the following section.

Final performance on the classification task

The final results for the classification task are here displayed. In Figure (2), the plot of the evolution of the training is shown. The error is computed with the L2 definition (as explained in section 2).

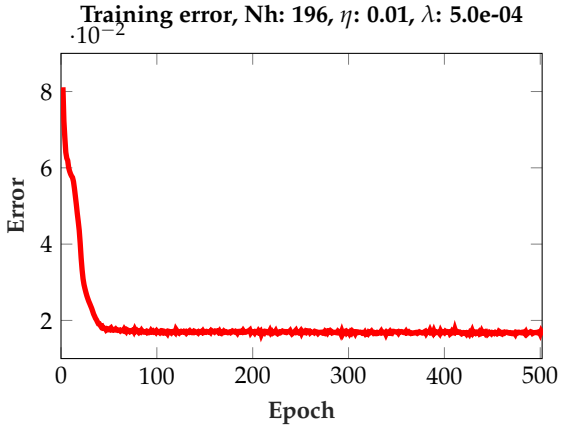

Figure (2) – Training error as a function of the epoch number, for RBM with 196 hidden units, learning rate $\eta = 0.01$ and weight decay parameter $\lambda = 5 \cdot 10^{-4}$. Graph produced with [11].

Observing the graph, it can be clearly seen that, after an initial phase of fast decay, the training error stabilises around the value 0.01, with few fluctuations. It is also remarkable that at about epoch 15 there is a slight change in the tendency of the graph: this is due to the change in the momentum parameter α which impacts on the learning process, as expected.

The RBM classification results are exhibited with 2 confusion matrices, one for the training set (3) and one for the test set (4), produced with the MATLAB function `plotconfusion`.

Confusion matrices are specific table layouts that allow visualisation of the performance of an algorithm, in which each row of the matrix represents the instances in a predicted class, while each column represents the instances in an actual class.

The green diagonal cells correspond to observations that are correctly classified. The off-diagonal cells, instead, correspond to incorrectly classified observations. Both the number of observations and the percentage of the total number of observations are shown in each cell.

The column on the far right of the plot shows the percentages of all the examples predicted to belong to each class that are correctly and incorrectly classified. These metrics are often called the **precision** and **false discovery rate**, respectively. The row at the bottom of the plot shows the percentages of all the examples belonging to each class that are correctly and incorrectly classified. These metrics are often called the **true positive rate** and **false negative rate**, respectively. The cell in the bottom right of the plot shows the overall accuracy.

Confusion matrix - Training, Nh: 196, η: 0.01, λ: 5.0e-04

Predicted labels \ True labels	0	1	2	3	4	5	6	7	8	9	
0	5793 9.7%	1 0.0%	23 0.0%	14 0.0%	7 0.0%	32 0.1%	23 0.0%	7 0.0%	22 0.0%	16 0.0%	97.6% 2.4%
1	0 0.0%	6656 11.1%	25 0.0%	12 0.0%	10 0.0%	7 0.0%	6 0.0%	15 0.0%	15 0.0%	8 0.0%	98.5% 1.5%
2	6 0.0%	31 0.1%	5638 9.4%	82 0.1%	28 0.0%	26 0.0%	27 0.0%	40 0.1%	36 0.1%	8 0.0%	95.2% 4.8%
3	11 0.0%	7 0.0%	68 0.1%	5740 9.6%	4 0.0%	106 0.2%	3 0.0%	19 0.0%	64 0.1%	58 0.1%	94.4% 5.6%
4	9 0.0%	5 0.0%	31 0.1%	2 0.0%	5564 9.3%	22 0.0%	24 0.0%	43 0.1%	25 0.0%	102 0.2%	95.5% 4.5%
5	21 0.0%	6 0.0%	18 0.0%	108 0.2%	7 0.0%	5088 8.5%	42 0.1%	9 0.0%	52 0.1%	33 0.1%	94.5% 5.5%
6	25 0.0%	3 0.0%	34 0.1%	9 0.0%	29 0.0%	50 0.1%	5771 9.6%	2 0.0%	27 0.0%	1 0.0%	97.0% 3.0%
7	10 0.0%	15 0.0%	44 0.1%	32 0.1%	26 0.0%	7 0.0%	1 0.0%	6012 10.0%	5 0.0%	108 0.2%	96.0% 4.0%
8	35 0.1%	15 0.0%	60 0.1%	85 0.1%	27 0.0%	52 0.1%	20 0.0%	15 0.0%	5561 9.3%	34 0.1%	94.2% 5.8%
9	13 0.0%	3 0.0%	17 0.0%	47 0.1%	140 0.2%	31 0.1%	1 0.0%	103 0.2%	44 0.1%	5581 9.3%	93.3% 6.7%
	97.8% 2.2%	98.7% 1.3%	94.6% 5.4%	93.6% 6.4%	95.2% 4.8%	93.9% 6.1%	97.5% 2.5%	96.0% 4.0%	95.0% 5.0%	93.8% 6.2%	**95.7%** **4.3%**

Figure (3) – Confusion matrix for the training test. Graph produced with [11].

Confusion matrix - Test, Nh: 196, η: 0.01, λ: 5.0e-04

Predicted labels \ True labels	0	1	2	3	4	5	6	7	8	9	
0	966 9.7%	0 0.0%	8 0.1%	3 0.0%	3 0.0%	8 0.1%	10 0.1%	0 0.0%	7 0.1%	8 0.1%	95.4% 4.6%
1	0 0.0%	1124 11.2%	5 0.1%	0 0.0%	0 0.0%	2 0.0%	3 0.0%	4 0.0%	3 0.0%	3 0.0%	98.3% 1.7%
2	1 0.0%	3 0.0%	962 9.6%	17 0.2%	5 0.1%	2 0.0%	4 0.0%	17 0.2%	9 0.1%	3 0.0%	94.0% 6.0%
3	2 0.0%	1 0.0%	9 0.1%	957 9.6%	0 0.0%	26 0.3%	0 0.0%	6 0.1%	19 0.2%	11 0.1%	92.8% 7.2%
4	1 0.0%	0 0.0%	7 0.1%	0 0.0%	932 9.3%	3 0.0%	5 0.1%	11 0.1%	6 0.1%	15 0.1%	95.1% 4.9%
5	1 0.0%	1 0.0%	6 0.1%	8 0.1%	1 0.0%	824 8.2%	10 0.1%	0 0.0%	11 0.1%	8 0.1%	94.7% 5.3%
6	4 0.0%	4 0.0%	7 0.1%	1 0.0%	4 0.0%	8 0.1%	921 9.2%	0 0.0%	3 0.0%	1 0.0%	96.6% 3.4%
7	1 0.0%	0 0.0%	7 0.1%	7 0.1%	7 0.1%	2 0.0%	1 0.0%	964 9.6%	9 0.1%	13 0.1%	95.4% 4.6%
8	3 0.0%	2 0.0%	16 0.2%	13 0.1%	7 0.1%	13 0.1%	4 0.0%	3 0.0%	898 9.0%	6 0.1%	93.1% 6.9%
9	1 0.0%	0 0.0%	5 0.1%	4 0.0%	23 0.2%	4 0.0%	0 0.0%	23 0.2%	9 0.1%	941 9.4%	93.2% 6.8%
	98.6% 1.4%	99.0% 1.0%	93.2% 6.8%	94.8% 5.2%	94.9% 5.1%	92.4% 7.6%	96.1% 3.9%	93.8% 6.2%	92.2% 7.8%	93.3% 6.7%	**94.9%** **5.1%**

Figure (4) – Confusion matrix for the test set. Graph produced with [11].

Of course, the most relevant picture is the test confusion matrix (4), since it evaluates the goodness of the RBM in the classification task on a dataset unknown to the machine.

Both figures show a very good global accuracy of about 95%. However, observing the tables, one can notice that some digits which have similar features (i.e. 4 and 9) have the highest chance of being mislabelled, and that other digits (8, 3) are more difficult to assign to a class. This is certainly a limit due to the prediction of the probability distribution of the pixels made by the RBM, which is probably more influenced by certain less distinguishable configurations, with high probability characterised by similar energy landscapes.

At last, in Figure (5), at the end of the paper, a representation of the weights of the network is presented. Each hidden unit is represented as a square matrix of $28 \times 28 = 784$ pixels, and each pixel represent the link between the hidden and the corresponding visible unit. The matrices are coloured in grey-scale according to the magnitude of the weight of each link. Consequently, by observing the colour pattern, it is possible to understand which units have learned better, if there are unused units or if particular features are detected by some precise hidden unit.

In this specific case, all the units present some kind of distribution, showing that the learning process worked properly.

Comparison with another neural network

The code of the other network considered can be found in [12]. The network analysed has two hidden layers with 80 and 60 neurons respectively. The units are activated with the *Elu* function (14):

$$R(z) = \begin{cases} z & z > 0 \\ \beta \cdot (e^z - 1) & z \leq 0 \end{cases} \tag{14}$$

with $\beta = 0.2$. It is trained over the same training set as the RBM, with 50 epochs and minibatches of size 10. The learning rate is $\eta = 0.0058$.

Test accuracy, with the aforementioned definition, is about 97.13%: this Neural network is indeed more effective than the one presented before, at the price of a more deep structure and a more time consuming implementation ($\approx$ 20 minutes for 50 epochs, versus $\approx$ 15 minutes per 100 epochs).

5. CONCLUSION

In this work, after properly introducing the RBMs and some of the training techniques used, a working implementation of a RBM has been developed for classifying handwritten digits. The architecture and the hyper-parameters chosen lead to an accuracy of 95%. The comparison with a different neural network clearly shows that the performances can be improved. Further studies can be lead, in order to tune better the hyper-parameters (especially η, λ) and generally upgrade the implemented network.

6. MATLAB CODE

The full MATLAB code can be found at https://github.com/SBea13/RBM-Matlab.

[1] M. Thoma, Latex-examples, https://github.com/MartinThoma/LaTeX-examples/tree/master/tikz/restricted-boltzmann-machine (2019).

[2] P. Mehta, M. Bukov, C.-H. Wang, A. G. R. Day, C. Richardson, C. K. Fisher, and D. J. Schwab, A high-bias, low-variance introduction to machine learning for physicists (2018), arXiv:1803.08823 [physics.comp-ph].

[3] ∧In general, a Boltzmann distribution naturally arises when searching the "most unbiased" distribution satisfying a set of constraints. This is the gist of the MaxEnt approach [13].

[4] ∧In a general (not restricted) Boltzmann Machine, these self-interactions are allowed, at the cost of making training harder. See [14] for more information.

[5] J. Schlüter, Restricted boltzmann machine derivations, http://ofai.at/~jan.schlueter/pubs/2014_techrep_rbm.pdf (2014).

[6] N. Qian, Neural Networks **12**, 145 (1999).

[7] G. Hinton, A practical guide to training restricted boltzmann machines (version 1) (2010).

[8] A. Fischer and C. Igel, Pattern Recognition **47**, 25 (2014).

[9] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (O'Reilly Media, Inc., 2019).

[10] C. M. Bishop, *Pattern Recognition and Machine Learning* (Springer, 2006) pp. 196–203.

[11] N. Schlömer, matlab2tikz/matlab2tikz, https://github.com/matlab2tikz/matlab2tikz (2020).

[12] J. Langelaar, Mnist neural network training and testing, https://www.mathworks.com/matlabcentral/fileexchange/73010-mnist-neural-network-training-and-testing (2019).

[13] E. T. Jaynes, Phys. Rev. **106**, 620 (1957).

[14] T. Osogami, Boltzmann machines and energy-based models (2017), arXiv:1708.06008 [cs.NE].

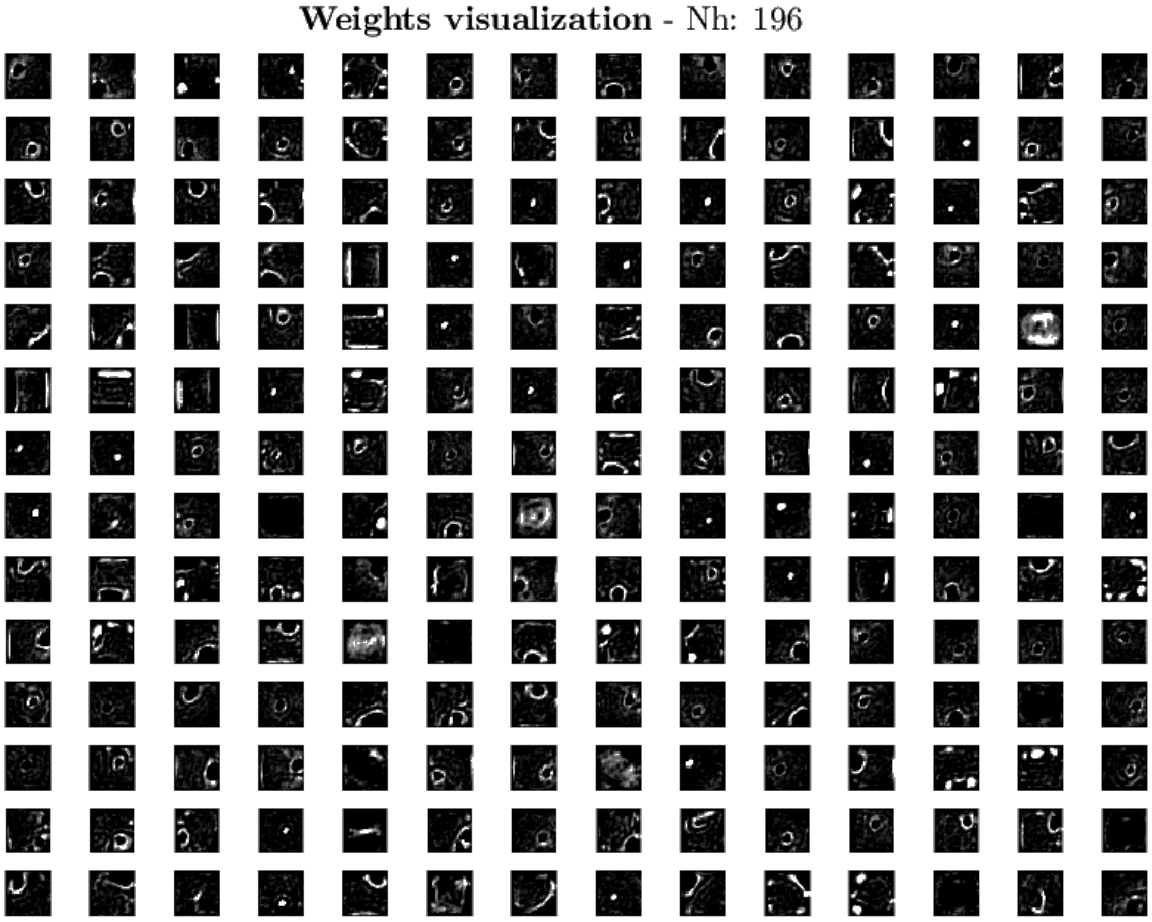

Figure (5) – Hidden units weights representation.