Deep Belief Networks

Ashraf GHIYE

April 04, 2021

Contents

1 Introduction

The objective of this project is to study deep neural networks for the classification of handwritten digits. We will compare the performances, in terms of rate of good classifications, of a pre-trained network (using Deep Belief Networks) and of a randomly initialized network, as a function of the number of layers, the number the number of neurons per layer and finally the number of training data.

2 Data

2.1 AlphaDigits

This database from NYU[1], contains binary digits of size 20x16 that represent character "0" through "9" and capital "A" through "Z".

There are 39 examples of each class, thus the data frame will contain 1,404 rows each representing an image (39 examples of 36 classes) and 320 variables taking binary values 1 or 0 (black =1 and white = 0 for each pixel value of a 20x16 alpha-numeric image).

Below, we plot some examples to get a closer look into the dataset.

Figure 1: AlphaDigits examples.

[1] http://www.cs.nyu.edu/ roweis/data.html

2.2 MNIST

The MNIST[2] database (Modified National Institute of Standards and Technology database) is a large collection of handwritten digits. It has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image. They were also centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field.

In our case, we will binarize the images in order to get 0/1 pixels representing the background (in black) and the digit (in white).

Also, we will transform the labels into a one hot encoded vector, i.e. vector of size 10 for each image with all zeros except one at the position of the digit number.

Below, we show some examples from the dataset after binarization.

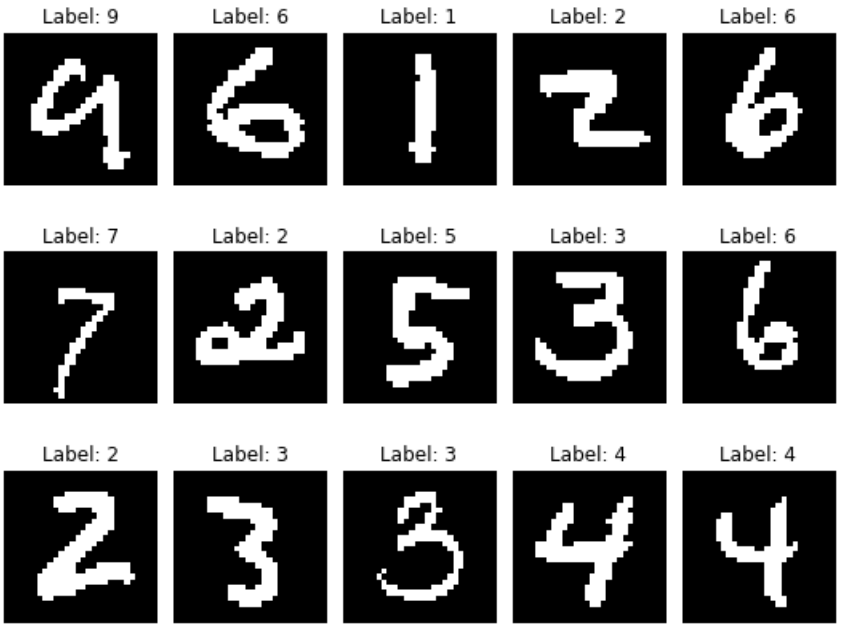


Figure 2: MNIST examples.

[2] http://yann.lecun.com/exdb

3 Preliminary Study

In this part, we will use RBM and DBN as generative networks. For that, we will train these models using AlphaDigits in an unsupervised fashion. We will then assess the quality of generated characters visually and compare their quality as a function of the number of training examples, the dimension of hidden variables and the number of Gibbs sampling iterations.

3.1 Restricted Boltzmann Machine

Restricted Boltzmann Machine is a stochastic shallow network with p input variables v_i and q output (or latent) variables h_j.

We will only focus on the case of binary variables.

$$v_i \in \{0,1\} \quad 1 \leq i \leq p$$

$$h_j \in \{0,1\} \quad 1 \leq j \leq q$$

RBM models the joint probability distribution between those variables as a function of a potential energy $E(v,h)$:

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)}$$

Where $Z = \sum_{v,h} e^{-E(v,h)}$ is a normalization constant.

The potential energy can be written as a polynomial function of v and h:

$$E(v,h) = -\sum_{i=1}^{p} a_i v_i - \sum_{j=1}^{q} b_j h_j - \sum_{i,j} w_{ij} a_i b_j$$

Hence, the RBM is a parametric model characterized with $\Theta = (a_i, b_j, w_{ij})$ and q.

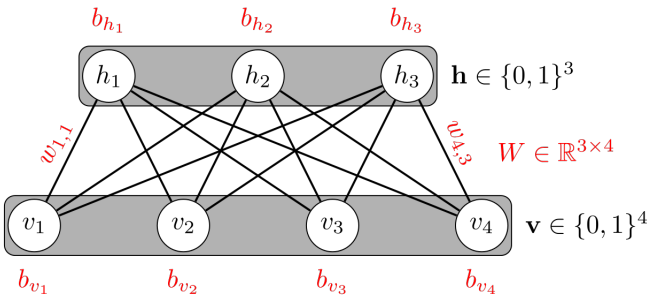


Figure 3: Restricted Boltzmann Machine with $p = 4$ and $q = 3$

Using Bayes formula, one can show easily that the conditional probability for a given output neuron $p(h_j|v)$ can be written as a simple sigmoid function:

$$p(h_j = 1|V) = \frac{1}{1 + e^{-(b_j + \sum_i w_{ij} v_i)}}$$

Conditioning on the input variables v, the latent variables h_j are independents from each other.

Boltzmann machines has shown to be universal approximators for probability distributions on **binary vectors**, this article [7] proves that an RBM with $2^{n1}1$ hidden units is a universal approximator of distributions on $0,1^n$.

RBM is a parametric statistical model, meaning that to learn the parameters Θ, we will need some statistical estimators such as Maximum Likelihood Estimation. It is for this reason they were widely used when backpropagation was infeasible.

However, the distribution is only known up to an unknown constant Z. Its calculation is intractable due to combinatorial complexity.

Therefore, training such models will be done using the Contrastive-Divergence 1 algorithm. G. Hinton proposed this algorithm in 2002 [4] to avoid the difficulty in computing the log-likelihood gradient. It uses MCMC methods to sample from the unknown distribution an estimator to be used in the gradient descent step.

3.1.1 RBM Analysis

After we train the RBM successfully on the AlphaDigits, it can provide a closed-form representation of the distribution underlying the training data. It can be used then as a generative model that allows sampling from the learned distribution (Fig.4).

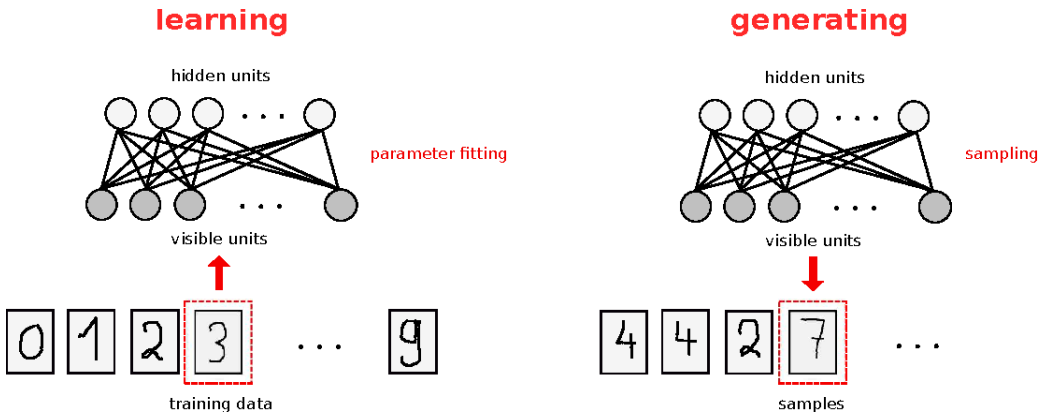
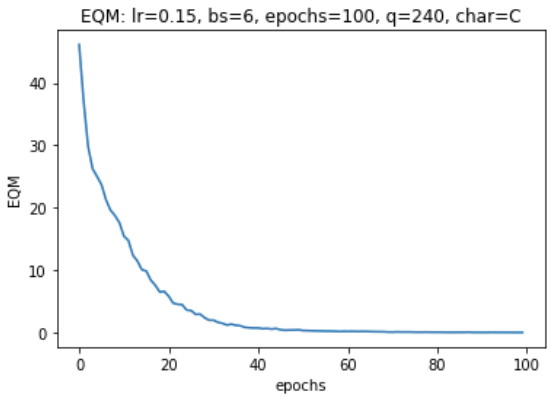


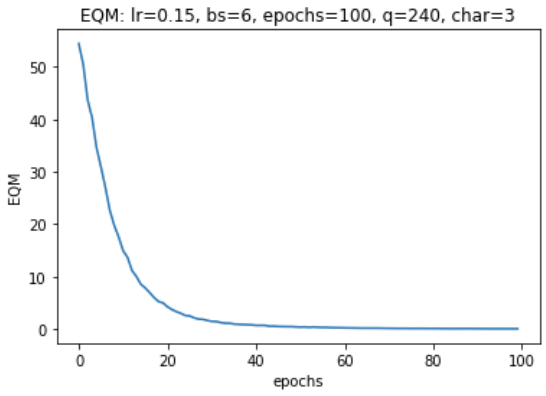
Figure 4: RBM for character generation.

To test the convergence of the learning algorithm, we compute the reconstruction error after each epoch which consist of comparing all the original samples X with reconstructed versions X' with respect to the ecludian distance, EQM = $||X - X'||^2$.

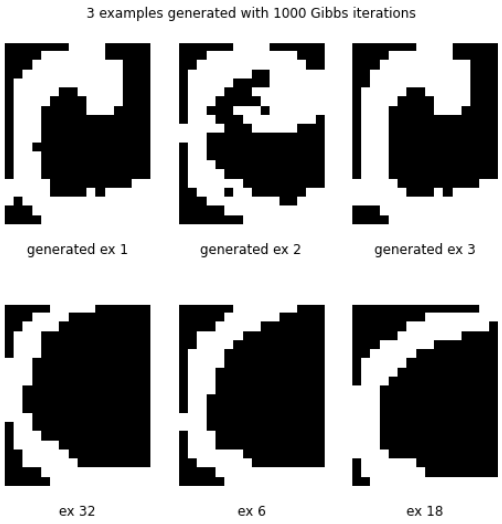
Here are the learning curves after training with the whole examples for two characters, i.e. we used the 39 examples for each of the character C and digit 3 to train two RBMs using CD-1:



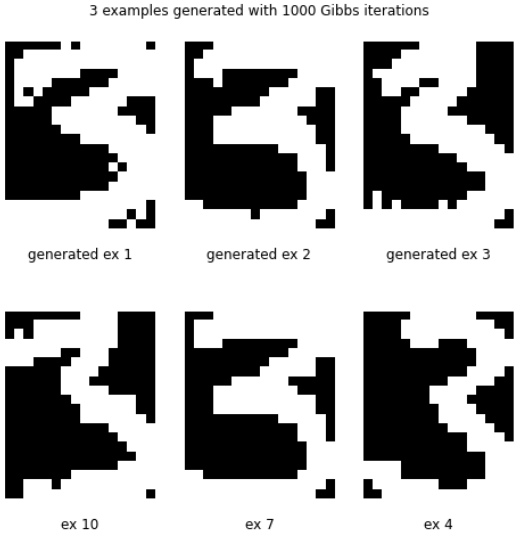
(a) Learning Curve 'C'



(b) learning Curve '3'



(c) Generating 'C' examples



(d) Generating '3' examples

Figure 5: RBM Experimental Results.

From the Fig.5, it is clear that the network was capable of regenerating good examples, meaning that we were able to learn the distribution of the input space (AlphaDigits). After learning the join distribution of input and output spaces using the RBM we were able to re-generate characters and digits through Gibbs sampling.

Note that, as for any learning algorithm, the hyper-parameters play a key role in the shape and speed of the learning curve. The optimal choice of these parameters will requires a grid-search to obtain the best configuration that minimizes the reconstruction loss and yields the best results.

However, we have run a sequential search of the best parameters, i.e. by fixing other parameters and selecting the best parameter one at a time. We started from a convenient configuration and ended up with the parameters shown in Fig.5 (a) and (b).

Here is the studied effects of the various hyper-parameters (for the learning algorithm and for the model itself):

- **Learning rate**: The magnitude of updates controls the speed of convergence. If the learning rate is set too low, training will progress very slowly towards the optimal solution. Nevertheless, too large steps can cause the model to converge too quickly to a suboptimal solution or worth: to diverge. Thus, this parameter requires a trade-off between the rate of convergence and overshooting.

- **Batch size**: It controls the accuracy of the estimate of the error gradient when training with a stochastic gradient descent. It is a trade-off between the speed of computation and the quality of estimation (of the gradient).

- **epochs**: The number of effective pass over the whole dataset, e.g. for a small learning rate, we need more epochs to have more time to learn more. The choice of this parameter depends on the batch size and learning rate. Also, on the dynamics in the learning curve.

- **q**: The dimension of latent space. Lower values will make the learning process more complex, while large values will make learning easier. But this will eliminate the benefits of dimensionality reduction. If we have an input dimension p of, say 320, then, would like to summarize the information in a much smaller dimension. Therefore we want q, the output dimension, to be equal to 240. Only then we will be able to compress with a factor of 1.5 the original information.

- **Gibbs iterations**: The number of sampling iterations to generate an image, the bigger this number is the better quality of reconstructed image we get. It is also a trade-off between the quality and the efficiency of the algorithm.

- **Training examples**: The number of training examples plays an essential role in the learning process, in our case, we used the whole examples for each character (39 examples) to learn its representation. However, a break-point exist, meaning that below a given number of samples the network won't be able to learn the distribution.

3.2 Deep Belief Network

The basic idea underlying these deep architectures is that the hidden neurons of a trained RBM represent relevant features of the observations, and that these features can serve as input for another RBM, see Fig.6 for an illustration. By stacking RBMs in this way, one can learn features from features in the hope of arriving at a high-level representation.

Thus a Deep Belief Network (DBN) is a multi-layer generative graphical model. DBNs have bi-directional connections (RBM-type connections) on the top layer while the bottom layers only have top-down connections. Mathematically put, a DBN with L layers has p_0 input units and $p_1 ... p_L$ units for each of the following layers.

$$v_i = h_i^{(0)} \in \{0,1\}^{p_0}$$

$$h_i^{(1)} \in \{0,1\}^{p_1}; \quad h_i^{(k)} \in \{0,1\}^{p_k}; \quad h_i^{(L)} \in \{0,1\}^{p_L}$$

Note that only the last layer can be considered as RBM:

$$p(h^{L-1}, h^L) = \frac{1}{Z} e^{-E(h^{L-1}, h^L)}$$

and the relation between the other layers are given by:

$$p(h^{i-1}, h^i) = \prod_{j=1}^{p_i} p(h_j^{i-1}, h^i)$$

and $p(h_j^{i-1} = 1 | h^i) = \text{sigmoid}(b_j^{i-1} + \sum_k w_{jk}^i h_k^i)$.

On a side note, a Deep Belief Network is not a Deep Boltzmann Machine. In which the latter is a concatenation of undirectional RBM networks.

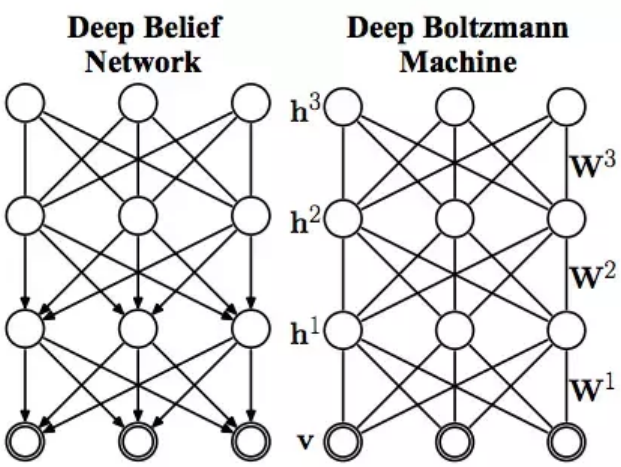


Figure 6: Comparison between DBN and DBM.

A DBN can be trained in a greedy unsupervised way, by training separately each RBM from it, in a bottom to top fashion, and using the hidden layer as an input layer for the next RBM.

This greedy approach consider a DBN as a stack of RBM, we then iterate over each layer and run a CD-1 algorithm to update its parameters a, b and W (Fig.7).

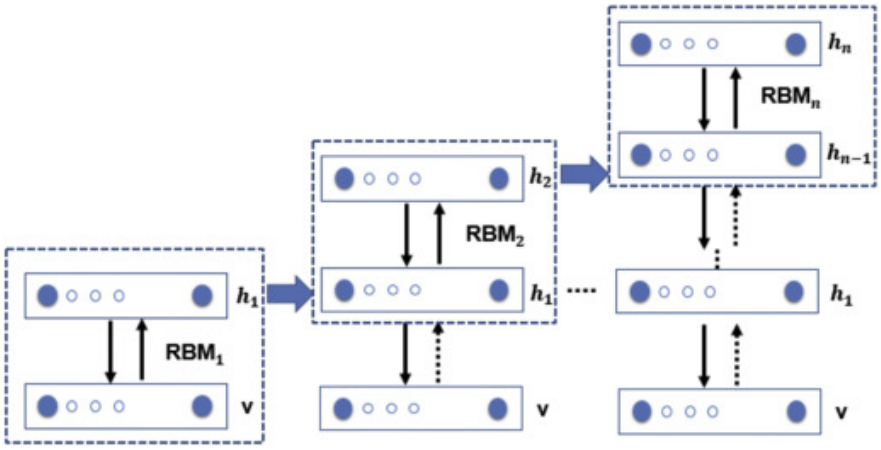


Figure 7: Layer-wise training: DBN.

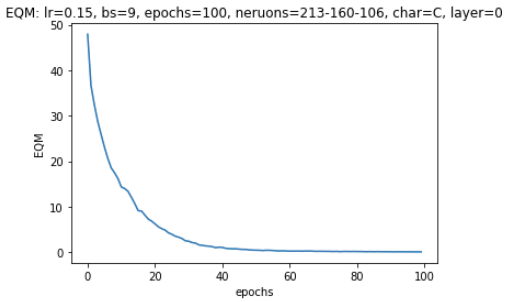
The advantage of a DBN over RBM is that via deep architecture, we are able to approximate complex joint distributions in an efficient manner comparing to a simple RBM. However, it comes with a computational cost (more parameters to estimate) and thus it needs more data to be learnt efficiently.

3.2.1 DBN Analysis

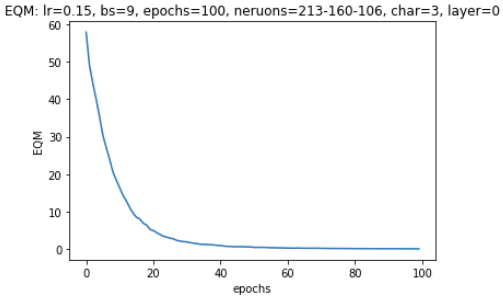
Figure 8 shows the experimental results of learning a DBN on AlphaDigits and using the learned model to generate some examples.

As in the case of an RBM, we used heuristics in the choice of hyper-parameters by keeping the best learning parameters from the previous experiments and choosing a three-layer DBN with $\frac{2}{3}p$, $\frac{1}{2}p$ and $\frac{1}{3}p$ neurons respectively.

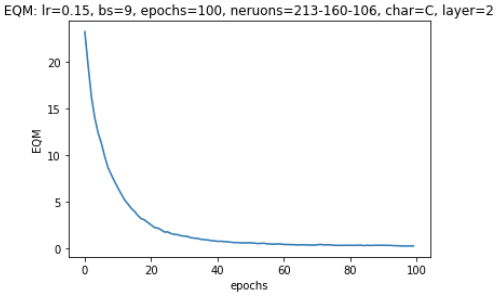
By comparing the quality of generated examples between a DBN and an RBM, we can see that the DBN has generated more realistic digits and characters. The RBM also gave good results, yet with more noise. Moreover, the DBN has learnt a lower-dimensional representation ($\frac{p}{3}$) of the input space (which can be regarded as higher-abstract features) compared to the RBM ($\frac{p}{1.5}$).



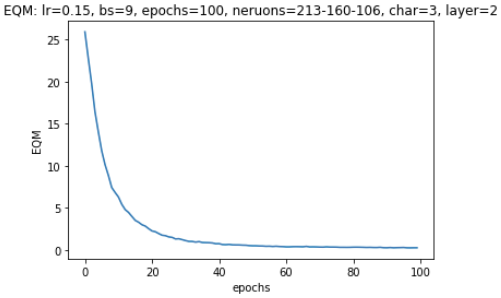
(a) Learning Curve layer 0



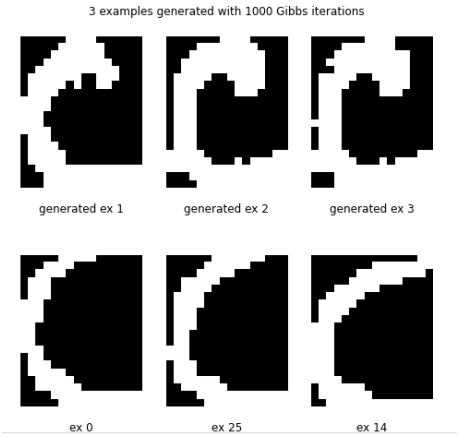
(b) Learning Curve layer 0



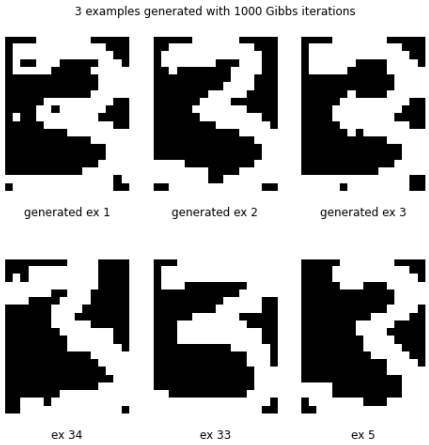
(c) Learning Curve layer 2



(d) Learning Curve layer 2



(e) Generating 'C' examples



(f) Generating '3' examples

Figure 8: DBN Experimental Results.

4 Primary Work

It is an important property that single as well as stacked RBMs can be reinterpreted as deterministic feed-forward neural networks. When viewed as neural networks they are used as functions mapping the observations to the expectations of the latent variables in the top layer. These can be interpreted as the learned features, which can, for example, serve as inputs for a supervised learning system. Furthermore, the neural network corresponding to a trained RBM or DBN can be augmented by an output layer where the additional units represent labels (e.g., corresponding to classes) of the observations. Then we have a standard neural network for classification that can be further trained by the standard supervised learning algorithms (backpropagation).

It has been argued that this initialization (or unsupervised pre-training) of the feed-forward neural network weights based on a generative model helps to overcome some of the problems that have been observed when training multi-layer neural networks [5].

We will prove these claims ourselves using the MNIST dataset. For that we will consider two neural networks, one is pre-trained as a DBN and the other is randomly initialized. We will then compare both models in the task of handwritten digit classification.

4.1 Deep Neural Network

A multilayer perceptron (MLP) is a class of feed-forward artificial neural network. A MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training. Mathematically speaking, it is approximating a function f, from $\mathbb{R}^p$ (input space) to $\mathbb{R}^q$ (output space):

$$f : \mathbb{R}^p \rightarrow \mathbb{R}^q$$

A neural network can approximate any continuous function, provided it has at least one hidden layer and uses non-linear activation functions. This has been proven by the universal approximation theorem [2].

For the classification task, we will consider a neural network, which can be seen as a deep belief network with an extra layer on the output. We will apply a softmax function on the output neurons to turn the vector of numbers into a vector of probabilities.

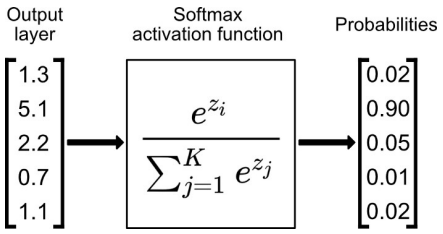


Figure 9: Softmax activation function.

Training these networks with backpropagation begin with an initialization of the weights matrices W^k between each layer and the bias vectors b^k, where k is the layer index. Then, through stochastic gradient descent steps, we can update these parameters based on a feedback signal from the output (a derivative of a loss function w.r.t the output) and a learning rate α.

It is well-known that the optimization landscape of a neural network is highly non-convex. Therefore the learning process is regarded as a hard optimization task. Injecting randomness in the gradient steps might help avoid a stuck in local minima. However, this latter still needs a large training corpus to be trained effectively and reach an optimal solution. They also suffer from slow learning and are highly sensitive to parameters selection.

For these reasons, initialization plays an essential role in the convergence of these algorithms. Random initialization methods were proposed by [3] to overcome this issue. In our case, we will consider another approach: initializing the weights with an unsupervised algorithm and considering the network (without the softmax) as a DBN. This unsupervised pre-training sets the stage for a final training phase where the deep architecture is fine-tuned with respect to a supervised training criterion with gradient-based optimization.

In the next section, we will run some experiments to compare the classification accuracy of both models. Note that we will use only half the training size (i.e. 30k examples) to train the network.

We will treat three scenarios separately. Later on, and based on the previous analysis, we will propose a network to achieve a good performance using the whole dataset (60k examples). Finally, we will draw some conclusions in the final section.

4.2 Results

4.2.1 The number of layers

Currently, there is no absolute answer of how many hidden layers should be stacked for best results. It depends on the types and structures of datasets and it is no exception to our datasets. A few hidden layers can be trained in a relatively short period of time, but result in poor performance as the system cannot fully store all the features of training datasets. Too many layers may result in over-fitting and slow learning time.

Therefore, we will compare both networks' accuracy **as a function of the number of layers**. For that, we initialized two identical networks each time with a different number of layers (from 1 to 5 layers). We keep other hyper-parameters fixed for each time.

The first row in Fig.10 (a) and (b) show that a pre-trained network outperforms a randomly initialized network as soon as we have more than 2 hidden layers. The reason behind this behaviour is that as we grow the depth of the network, we also increase the complexity of our model and hence, more data are needed to achieve good performance with a normal DNN. The pre-training phase works as a regularization procedure, in the sense that instead of start-

ing from a completely random point in the optimization landscape, we start from a place that reflects the marginal distribution of the input space. This will helps the learning algorithm to converge faster to an optimal solution.

4.2.2 The number of hidden units

The number of hidden units in each layer corresponds to the features of input data stored in the system. Similar to the number of hidden layers, too little or too many hidden units result in slow learning and poor performance.

We fixed the number of layers to 2 hidden layers and tested a different number of units, i.e. 100, 200, 300, 500 and 1000 hidden units. All other hyper-parameters held fixed.

The results in the second row, Fig.10 (c) and (d) show that increasing the number of units for a fixed number of layers does not yield a noticeable improvement. It is more efficient to add extra layers and keep a relatively small number of hidden units (between 200 and 500).

However, note that the pre-trained network out-performed its counterpart except for two cases when the number of units were too many (1000 hidden units) or too few (100 hidden units).

4.2.3 The number of training examples

Usually, neural networks have a huge number of parameters. Thus, they need a lot of training data in order for them to be robust and usable in the real world.

In this third experiment, we fixed all the hyper-parameters as before. We trained both models using a different set of training examples each time, i.e. using 1000, 5k, 7k, 10k, 30k and 60k.

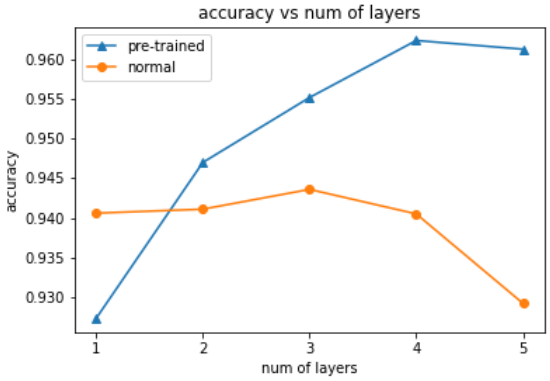
Fig.10 (e) and (f) show that a pre-trained network will always have averagely better accuracy than normal DBN, especially when the proportion of training data is smaller than 50%. It is one of the main reason behind pre-training a neural network to overcome the necessity of large datasets, for example in semi-supervised tasks we can pre-train the network with the unlabeled data and fine-tune on the few annotated examples.

4.3 Final Model

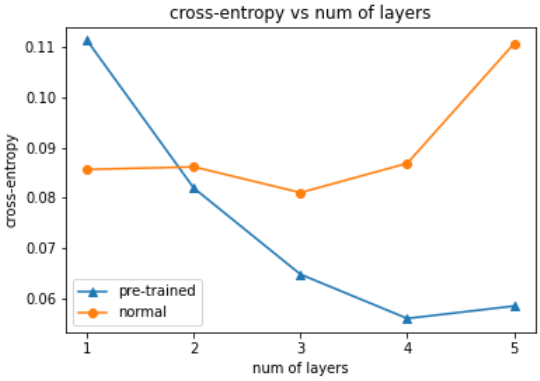
We choose an architecture with 3-hidden layers of 300 hidden units each. We pre-train the model with 60 epochs as a DBN, we then backpropagation with 100 epochs. We use a learning rate of 0.2 and a batch size of 100.

Refer to Fig.11 and 12 for performance evaluation, we obtain a final accuracy of 97.6% using a pre-trained network and 97.4% using the same configuration but without pre-training.

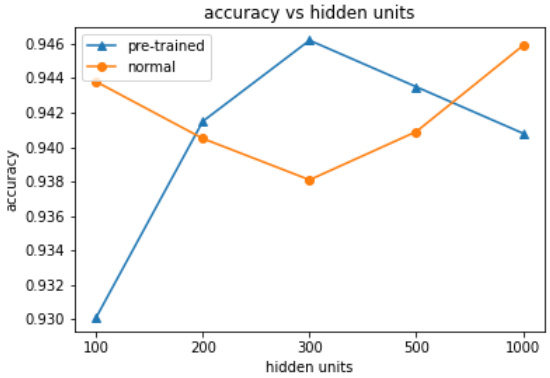
Also, Fig. 12 show clearly how the training loss of a pre-trained network start from 0.12 and have a smoother convergence over time compared to a randomly initialized network which start from a higher loss.



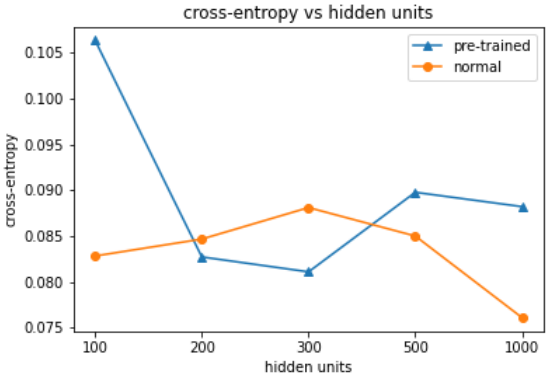
(a)



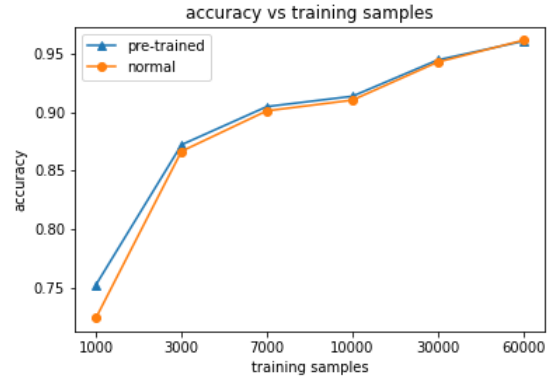
(b)



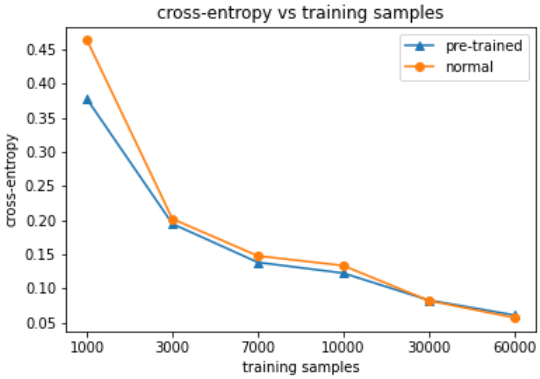
(c)



(d)

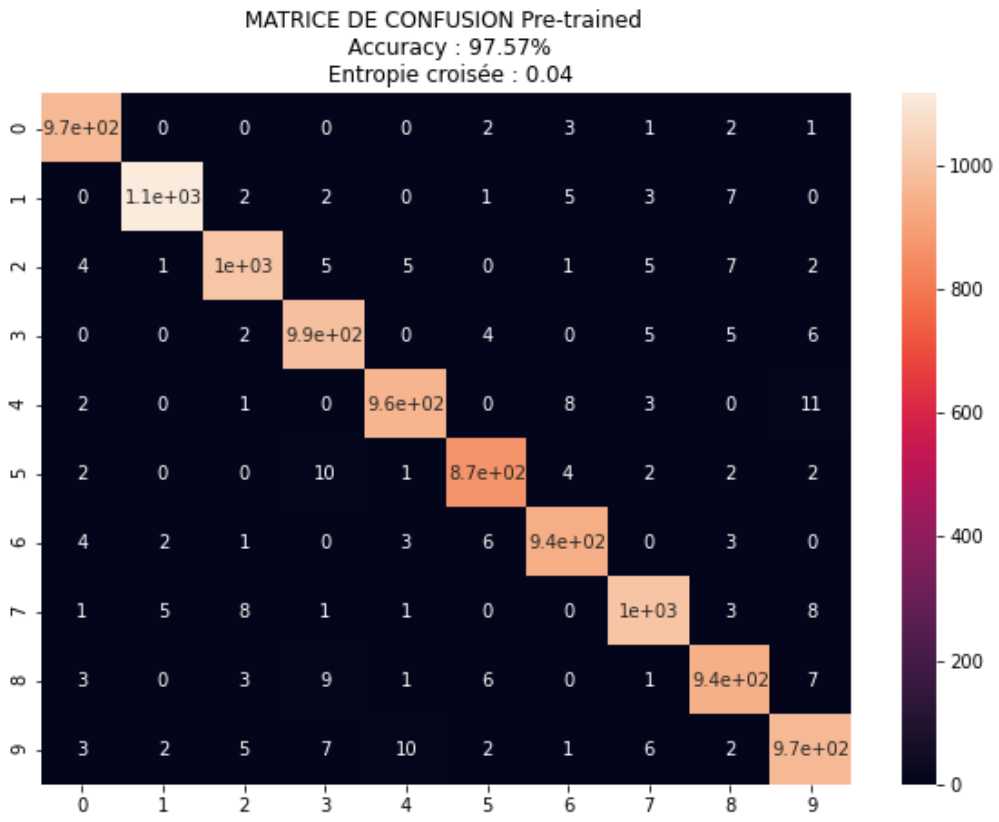


(e)



(f)

Figure 10: DNN Experimental Results on the test set.



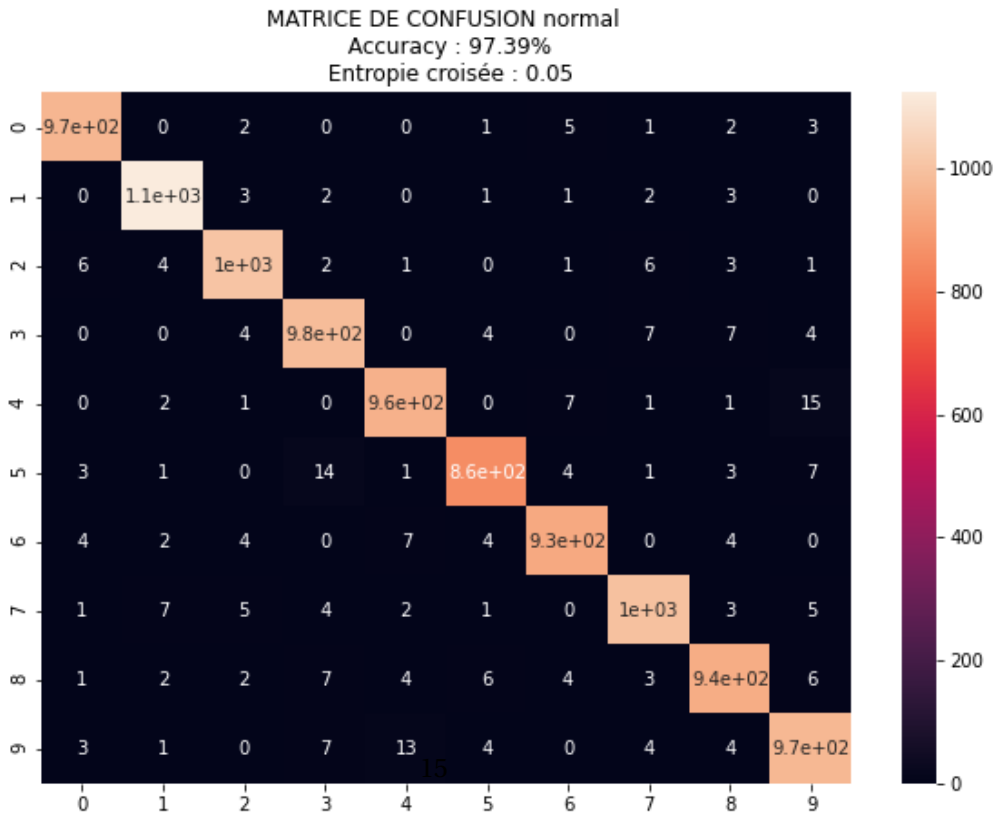
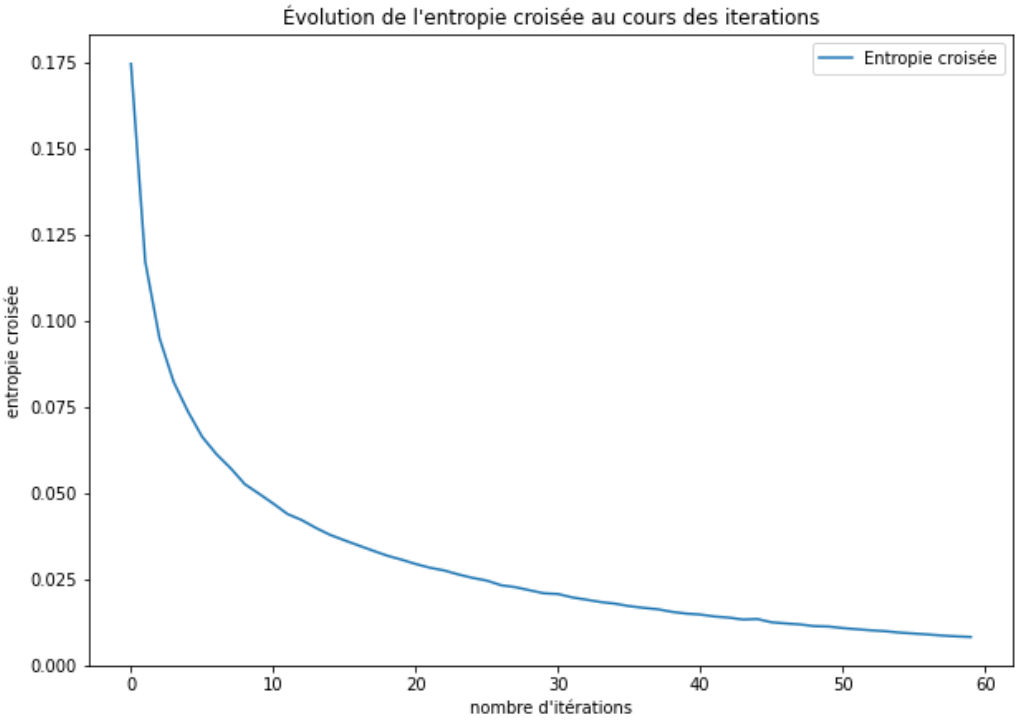


Figure 11: Confusion matrix on the 10k test set, y-axis: true labels, x-axis: predicted labels.



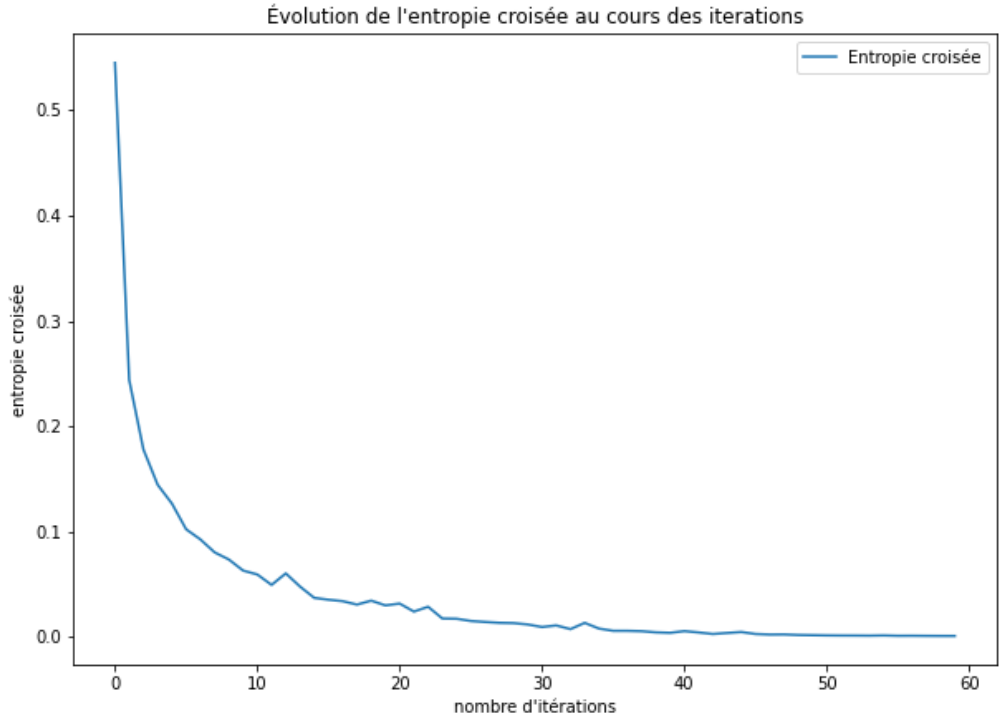


Figure 12: Training loss vs epochs, on top: pre-trained network.

5 Conclusion

Deep Belief Networks (DBNs) were invented as a solution for the problems encountered when using traditional neural networks training in deep layered networks, such as slow learning, becoming stuck in local minima due to poor parameter selection, and requiring a lot of training datasets. DBNs were initially introduced in (Larochelle, Erhan, Courville, Bergstra, Bengio, 2007)[6] as probabilistic generative models to provide an alternative to the discriminative nature of traditional neural nets. Generative models provide a joint probability distribution over input data and labels, facilitating the estimation of both $p(x|y)$ and $p(y|x)$, while discriminative models only use the last model $p(y|x)$.

Bengio and al.[1] showed that unsupervised pre-training adds robustness to a deep architecture. The same set of results also suggests that increasing the depth of an architecture that is not pre-trained increases the probability of finding poor apparent local minima. Pre-trained networks give consistently better generalization.

Finally, note that unsupervised pre-training is not only a way of getting a good initial marginal distribution, and that it captures more intricate dependencies between parameters. But, it can be regarded as a **regularization technique** (based on Markov Energy Fields), meaning that we are forcing the network to represent the latent structure of the input distribution and encourage hidden layers to encode that structure.

Intuitively, we are setting a constraint on the search region in the parameter space: **amongst all the neural networks that can do well on the classification, we want to favour neural networks that also understand why our input data is special**. In our case for the MNIST dataset, we want networks that understand digits, i.e. pixels intensities randomly drawn from a uniform distribution. But instead, they represent characters, and thus each character (class) has a proper distribution which assigns probability mass in the different region on the grid (like a heat-map).

This prior can enable models learn faster, use less training data and still be robust and accurate.

6 Code Execution

The code is available on **github.com/ashrafghiye/DBN**. The code folder can be divided into three parts:

1- **principal_RBM_alpha**, **principal_DBN_alpha** and **principal_DNN_MNIST** contain only function definitions with no runnable code. Another file, **utils.py** also contains some necessary functions to load datasets, etc.. These files have to be imported in any script that aims to use RBM, DBN or DNN as objects..
2- **preliminary_RBM_tests.py** and **preliminary_DBN_tests.py** are scripts that you can run immediately. They perform a quantitative and qualitative analysis on the AlphaDigits, their results are stored in images/RBM_analysis and images/DBN_analysis.
3- **main.py** contains the study case on MNIST dataset, you have to choose a number between 1, 2 or 3 and run the script to compare a pre-trained DBN with a randomly initialized DNN. The results are stored in images/DBN_analysis.

Finally, **two Jupyter notebooks** contains all the pieces of code (organized into sections) in one place. In the first notebook, implementation is done with numpy from scratch. The **second** is the same notebook, but with operations substituted with **torch operations**, We gained enormously in performance by replacing numpy arrays with torch tensors.

For more details, about how to run the code, and use it as standard objects, please refer my repository on GitHub which have some examples on how it works.

References

[1] Bengio Y. et al. "Why Does Unsupervised Pre-training Help Deep Learning?" In: *Journal of Machine Learning Research* (2010).

[2] G. Cybenko. "Approximation by superpositions of a sigmoidal function." In: *Math. Control Signal Systems 2* (1989). URL: https://doi.org/10.1007/BF02551274.

[3] Xavier Glorot and Y. Bengio. *Understanding the difficulty of training deep feedforward neural networks*. 2010.

[4] G. Hinton. "Training products of experts by minimizing contrastive divergence." In: *Neural Computation, 14(8)* (2002).

[5] G. E. Hinton and R. R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." In: *Science, 313(5786)* (2006).

[6] Hugo Larochelle and Yoshua Bengio et al. "An empirical evaluation of deep architectures on problems with many factors of variation." In: *Conf.Mach.Learn* (2007).

[7] Guido Montufar. "Restricted Boltzmann Machines: Introduction and Review". In: *arXiv:1806.07066* (2018).