Machine de Boltzmann Restreinte

Gestion Informatique et Python

Amaury Maros

9 février 2025

Ce rapport s'inscrit dans le cadre de l'UE "Gestion informatique et Python" du Master 1 Physique à distance de l'Université Aix-Marseille. On se propose ici d'étudier une machine de Boltzmann restreinte (ou RBM, pour Restricted Boltzmann Machine) à l'apprentissage de chiffres manuscrits en utilisant la base de données MNIST.

Table des matières

Introduction

Les machines de Boltzmann restreintes (RBM) sont des modèles de réseaux de neurones dérivés du modèle de Hopfield [1][2] et utilisés pour l'apprentissage non supervisé. Leur fonctionnement repose sur des principes issus de la physique statistique, notamment le calcul des probabilités et la minimisation de l'énergie. Une RBM est composée de deux couches : une couche visible, correspondant aux données observables, notée **v** dans la figure 1, et une couche cachée, notée **h**, représentant les caractéristiques sous-jacentes de ces données. Les neurones au sein d'une même couche sont indépendants les uns des autres, et les connexions entre les neurones de la couche visible et ceux de la couche cachée suivent une relation « one-to-many », caractérisant la structure restreinte du modèle.

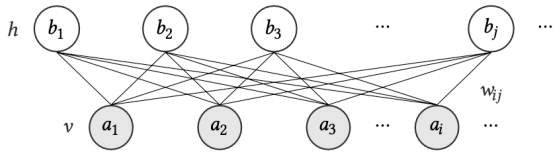


FIGURE 1 – Schéma d'une Machine de Boltzmann Restreinte

Chaque neurone dans les couches visible et cachée est associé à un biais, noté respectivement a_i et b_i, et chaque connexion entre un neurone de la couche visible et un neurone de la couche cachée est caractérisée par un poids w_{ij}. Une RBM peut être modélisée comme un système physique basé sur l'énergie donnée par la relation suivante [3] :

$$E(v,h) = -\sum_i \sum_j w_{ij} v_i h_j - \sum_i a_i v_i - \sum_j b_j h_j$$

que l'on peut réécire :

$$E(v,h) = -a^T v - b^T h - v^T W h$$

où l'exposant T indique la transposée et W est la matrice de poids, dont les coefficients sont w_{ij} et dont les dimensions sont $\dim(v) \times \dim(h)$. Cette énergie est fonction des biais a et b et du poids. L'optimisation du modèle revient donc à minimiser cette énergie en déterminant les paramètres de biais et de poids optimaux. La probabilité d'avoir la RBM dans cet état d'énergie est donnée par :

$$p(v,h) = \frac{e^{\frac{-E(v,h)}{k_B T}}}{Z}, \quad Z = \sum_{v,h} e^{\frac{-E(v,h)}{k_B T}}$$

On pose par convention $k_B T = 1$. La probabilité d'une configuration de la couche visible est donnée par la somme, sur toutes les configurations possibles de la couche cachée, de la probabilité totale. On a alors :

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

Minimiser l'énergie revient alors a maximiser cette probabilité. On peut également passer en échelle logarithmique. Comme les neurones au sein d'une même couche sont indépendants, la probabilité d'avoir une configuration donnée de la couche visible ou cachée est donnée par le produit des probabilités associées à chaque neuronne de la couche considérée. En échelle logarithmique, ce produit devient une somme.

1 Méthodes

1.1 Préparation des données

Le jeu de données utilisé, MNIST, comporte 70 000 images de chiffres manuscrits allant de 0 à 9. Chaque image est en niveaux de gris, de taille 28x28 pixels. Les images peuvent être représentées sous forme de vecteurs de longueur 784 contenant des valeurs numériques décrivant l'intensité de chaque pixels, chaque valeur étant comprise entre 0 et 255. Les données sont ensuite normalisées entre 0 et 1 en divisant chaque vecteur par 255.

Le jeu de données est séparé en deux sous-jeux : un jeu d'entraînement et un jeu de test au moyen de la fonction `train_test_split` de scikit-learn.

```
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, y_dat, test_size=10000,
↪  random_state=42, stratify=y_dat)
```

Le jeu de test étant de 10 000 objets, cela représente une proportion de donnée d'entrainement par rapport au données de test d'environ 85% pour 15%.

1.2 Implémentation du modèle

L'implémentation de la RBM se déroule en deux phases : une phase d'entraînement, durant laquelle le modèle apprend à partir des données, et une phase de test, où l'on évalue la qualité de l'apprentissage sur des données inconnues. L'apprentissage se fait de manière itérative et repose sur l'optimisation des poids et des biais via une procédure de descente de gradient. Une itération correspond au passage complet du modèle sur les données d'entrainement divisées en mini-lots ("batches") de taille L. L'implémentation du modèle est réalisée en Python en définissant une classe `RBM` à l'intérieure de laquelle plusieurs méthodes sont définies :

1. **Initialisation** :
   La classe `RBM` est initialisée avec les paramètres suivants :
   - `Nv` : le nombre d'unités visibles (784 pour des images de 28x28 pixels issues de MNIST),
   - `Nh` : le nombre d'unités cachées (valeur utilisée : 30),
   - `gamma` : le taux d'apprentissage (par défaut 0.1),
   - `T` : le nombre d'itérations pour l'entraînement,
   - `L` : la taille des lots (par défaut 16),
   - `W` : les poids initiaux, initialisés selon une distribution normale de moyenne 0 et d'écart-type 0.01,
   - `a` : les biais des neurones de la couche visible, initialisés selon une distribution normale de moyenne 0 et d'écart-type 1,
   - `b` : les biais des neurones de la couche cachée, également initialisés selon une distribution normale de moyenne 0 et d'écart-type 1.
2. **Méthode `sigmoid`** :
   La fonction sigmoïde $\sigma(x) = \frac{1}{1+e^{-x}}$ est utilisée pour estimer les probabilités d'activation des neurones cachés et visibles. Elle est définie dans la méthode `sigmoid`.
3. **Méthode `optimizeRBM`** :
   Pendant l'entraînement, les données d'apprentissage sont divisées en mini-lots. Pour chaque mini-lot, l'algorithme effectue les étapes suivantes basées sur le processus de "contrastive divergence" utilisé dans les machines de Boltzmann restreintes :

— **Phase positive** : Dans cette phase, on "connait" la couche visible et on calcule la probabilité d'activation des neurones cachés :

$$p(h_j = 1|\mathbf{v}) = \sigma\left(\sum_i w_{ij} v_i + b_j\right)$$

Ensuite, le produit scalaire $\langle \mathbf{v}, p(h = 1|\mathbf{v})\rangle$ est calculé. Il représente la contribution de la passe positive à l'erreur et à la mise à jour des paramètres a, b et W.

— **Phase négative** : Cette phase consiste à reconstruire les données visibles à partir des activations des neurones cachés, puis à recalculer les probabilités cachées à partir des données reconstruites. Les étapes sont les suivantes :

(a) Calcul des données visibles reconstruites $\mathbf{v}_{\text{mean}}$:

$$\mathbf{v}_{\text{mean}} = p(v_i = 1|\mathbf{h}) = \sigma\left(\sum_j p(h_j = 1|\mathbf{v}) w_{ij} + a_i\right)$$

(b) Calcul des probabilités d'activation des neurones cachés à partir des données reconstruites :

$$p(h_j = 1|\mathbf{v}_{\text{mean}}) = \sigma\left(\sum_i w_{ij} v_{\text{mean},i} + b_j\right)$$

(c) Calcul du produit scalaire $\langle \mathbf{v}_{\text{mean}}, p(h = 1|\mathbf{v}_{\text{mean}})\rangle$, utilisé pour évaluer la contribution de la passe négative.

— **Calcul de l'erreur** : L'erreur sur un mini-lot est mesurée comme la moyenne des erreurs quadratiques entre les données visibles originales (v_i) et leurs reconstructions ($v_{\text{mean},i}$) :

$$\text{erreur} = \frac{1}{L}\sum_{i=1}^{L}(v_i - v_{\text{mean},i})^2$$

Cette erreur est calculée individuellement pour chaque mini-lot, puis suivie tout au long de l'entraînement. Elle permet d'évaluer la qualité de la reconstruction effectuée par le modèle.

— **Mise à jour des paramètres** : Les poids W et les biais a et b (initialisés dans la définition de la classe `RBM`) sont incrémentés de leurs valeurs pour chaque mini-lots en fonction de la différence entre les contributions des phases positive et négative, selon les règles suivantes :

- Mise à jour des poids :

$$\mathbf{W} \leftarrow \mathbf{W} + \frac{\gamma}{L} \cdot \Big(\langle \mathbf{v}, p(h = 1 \mid \mathbf{v})\rangle - \langle \mathbf{v}_{\text{mean}}, p(h = 1 \mid \mathbf{v}_{\text{mean}})\rangle\Big)$$

- Mise à jour des biais des neurones visibles :

$$\mathbf{a} \leftarrow \mathbf{a} + \gamma \cdot \overline{(\mathbf{v} - \mathbf{v}_{\text{mean}})}$$

- Mise à jour des biais des neurones cachées :

$$\mathbf{b} \leftarrow \mathbf{b} + \gamma \cdot \overline{\big(p(h = 1 \mid \mathbf{v}) - p(h = 1 \mid \mathbf{v}_{\text{mean}})\big)}$$

— **Calcul de l'erreur globale** : Au début de chaque itération, l'erreur globale (error_globale) est initialisée à zéro. Lors du traitement de chaque mini-lot, l'erreur associée (erreur_batch)

est ajoutée à l'erreur globale. Une fois tous les mini-lots traités pour une itération, l'erreur globale est mise à jour en calculant la moyenne des erreurs sur tous les mini-lots :

$$\text{erreur\_globale} = \frac{\sum \text{erreur\_batch}}{N_{\text{batches}}}$$

où N_{batches} représente le nombre total de mini-lots dans l'itération.

La méthode `optimizeRBM` se renvoie elle-même : lorsque toutes les itérations T ont été effectuées, le modèle renvoie un objet entraîné avec les paramètres qui lui ont été fournis. Ce modèle peut alors être testé sur de nouvelles données, inconnues.

4. **Méthode `identify_X` :**

   La méthode `identify_X` utilise le modèle RBM entraîné pour reconstruire les données visibles à partir des données de test. Elle prend en entrée les données de test : ce sont les données visibles que l'algorithme doit reconstruire à partir des informations qu'il a apprises dans la méthode `optimizeRBM`. La méthode renvoie un objet `state_v`, représentant l'état visible reconstruit.

   Description :

   (a) **Calcul de la probabilité de l'état caché** : La méthode calcule la probabilité $p(h = 1|X)$, où h est l'état de la couche cachée et X les données visibles de test. Cette probabilité est obtenue en appliquant une fonction sigmoïde sur le produit scalaire entre X et les poids W, additionné au biais b :

   $$p(h = 1|X) = \sigma(XW + b)$$

   où σ est la fonction sigmoïde.

   (b) **Détermination de l'état caché** : On tire un nombre aléatoire entre 0 et 1. Si ce nombre est inférieur à la probabilité cachée, le neurone est activé ($h = 1$), sinon il reste inactif ($h = 0$). On obtient ainsi `state_h`.

   (c) **Reconstruction de l'état visible** : À partir de `state_h`, on calcule la probabilité de l'état visible reconstruit `prob_v` en appliquant une sigmoïde au produit scalaire entre h et W^T, additionné au biais a :

   $$p(v = 1|h) = \sigma(hW^T + a)$$

   (d) **Détermination de l'état visible** : L'état visible reconstruit `state_v` est obtenu similairement à `state_h` avec la probabilité visible.

1.3 Utilisation du modèle

Un objet `rbm` de la classe `RBM` est instancié sur le jeu d'entrainement. Un timer est initialisé pour calculer le temps de calcul associé à la phase d'optimisation. Le modèle est ensuite appliqué au jeu de test. Un exemple d'utilisation est donné dans le code ci-après (voir figure 2).

```python
# Instanciation d'un objet de la classe RBM
rbm = RBM(Nv=X_train.shape[1], Nh=30, T=20)

# Start timer
start_time = time.time()

# Optimisation du modèle
rbm.optimizeRBM(X_train)

# End timer
end_time = time.time()

# Temps de calcul
runtime = end_time - start_time
print(f"Training completed in {runtime:.2f} seconds.")

# Test du modèle sur le jeu de test
test_labels = [np.where(Y_test == str(i))[0][0] for i in range(10)]
test_images = X_test[test_labels]
generated = rbm.identify_X(test_images)
```

FIGURE 2 – Utilisation du modèle RBM

2 Résultats

2.1 Erreurs et temps de calcul

Nous commençons notre étude avec les paramètres suivants : le nombre d'atome visible (N_v) est de 784, le nombre de neurones de la couche cachée (N_h) est de 30, le taux d'apprentissage γ est fixé à 0,1 et nous utilisons des mini-lots de taille $L = 16$. Lors de l'entraînement et de l'optimisation d'un modèle d'apprentissage tel que notre machine de Boltzmann restreinte, il est pertinent de s'intéresser au coût de l'optimisation en termes de temps de calcul par rapport aux performances obtenues par le modèle. Différentes valeurs ont été testée pour le paramètre T, représentant le nombre d'itération effectuées par l'algorithme, intimement lié au temps de calcul. Ces valeurs vont de 10 à 100 par pas de 10. La figure 3 représente l'évolution du temps de calculs pour chaque valeur du paramètre T ainsi qu'un aperçu de la distribution des erreurs associées.

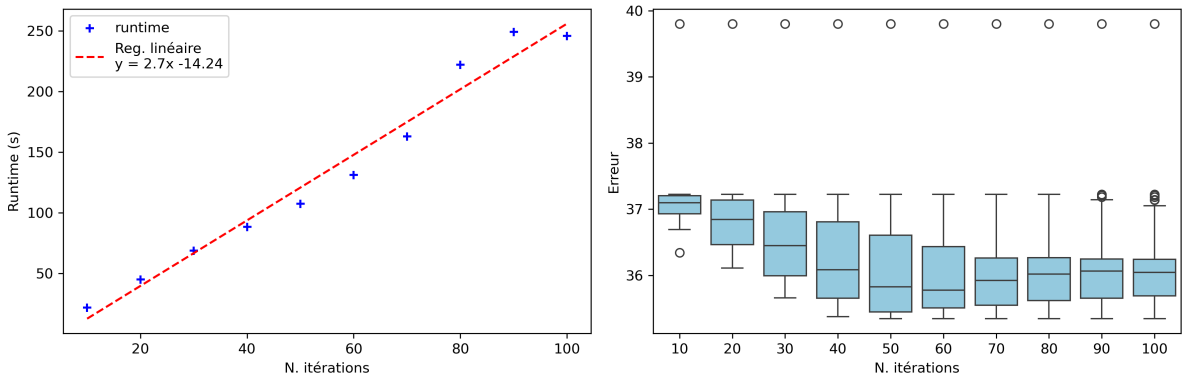


FIGURE 3 – Temps de calculs et distribution des erreurs pour différentes valeurs d'itération

Dans notre cas, nous constatons que le temps de calcul évolue linéairement avec le nombre d'itérations dans notre processus d'apprentissage. Ainsi, plus le nombre d'itérations augmente, plus le temps de calcul associé à notre modèle devient important, avec un facteur multiplicatif d'environ 2,7. Lorsque nous examinons la distribution des erreurs pour différentes valeurs d'itération, nous observons que l'erreur moyenne décroît jusqu'à atteindre un plateau à partir de $T = 60$, pour rester relativement stable entre $T = 70$ et $T = 100$. Par conséquent, l'erreur moyenne n'est pas significativement réduite pour des itérations supérieures à 60, bien que le temps de calcul associé puisse doubler : runtime$(T = 60) = 125\,\text{s}$ contre runtime$(T = 100) = 250\,\text{s}$.

Pour chaque valeur du paramètre T, les erreurs ont été exprimées en fonction des intérations du modèle. La figure 4 suivante illustre les résultats obtenus. Les points de données sont représentés en bleus, les pointillés oranges représentent l'allure générale de la courbe et les droites en pointillés verts représentent les coordonnées du point représentant l'erreur minimale représentée par un point rouge.

Lorsque nous observons l'évolution de l'erreur au cours de chaque itération, nous constatons que l'erreur minimale obtenue dans nos modélisations est atteinte pour un nombre d'itérations de 42. Pour un nombre d'itérations supérieur, l'erreur augmente à nouveau avant de redescendre aux alentours de $T = 70$. Jusqu'à $T = 100$, l'erreur ne parvient pas à retrouver sa valeur minimale. Ainsi, dans une perspective de développement et de recherche des paramètres optimaux, il pourrait être judicieux de limiter l'entraînement de notre modèle à 40 itérations. Cela permettrait d'optimiser l'erreur tout en réduisant le temps de calcul associées à de multiples entraînements.

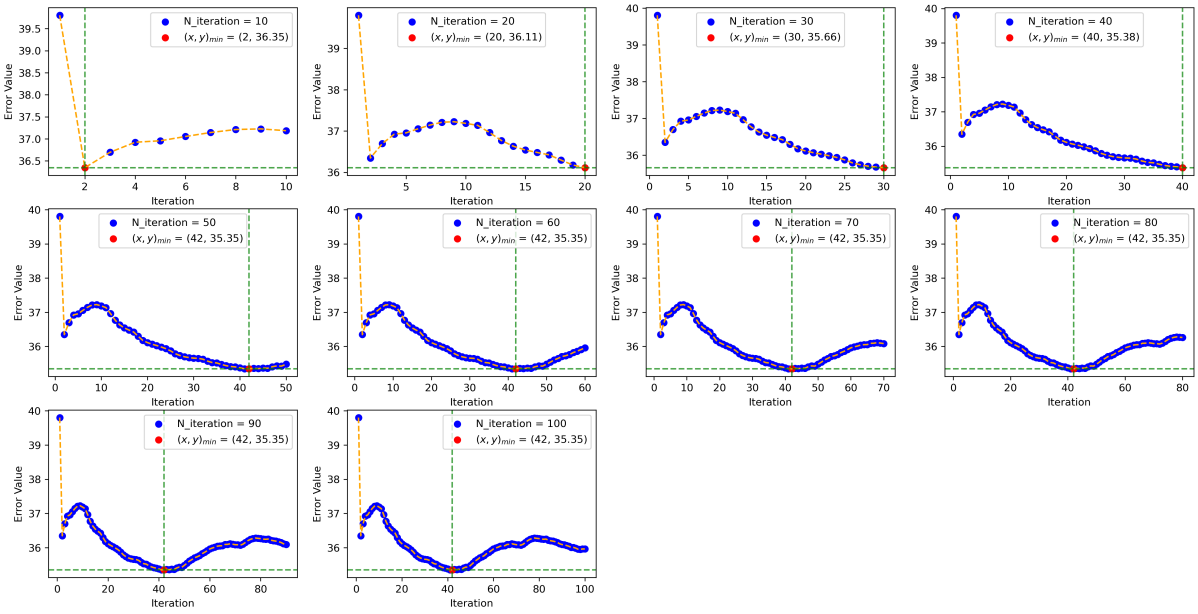


FIGURE 4 – Evolution de l'erreur au cours de chaque itération pour différentes valeur de T

2.2 Reconstruction des images

Bien que l'erreur soit acceptable (environ 35%), il est important de vérifier la robustesse du modèle sur le jeu de test d'un point de vue pratique. Pour ce faire, nous pouvons comparer les images reconstruites avec les images originales (figure 7). Pour toutes les valeurs de T, certains chiffres semblent plus difficiles à appréhender par le modèle ; c'est le cas notamment des chiffres "3", "5" et "8". Pour des valeurs de T supérieures à 40, on peut visualiser l'impact de l'augmentation de l'erreur du modèle, notamment sur les chiffres "5" et "8" qui sont moins bien reconstruits.

On pourrait se demander si, pour des temps d'itération plus longs, l'erreur diminuerait significativement. La figure 8 montre l'évolution de l'erreur pour 200 et 400 itérations. On constate que l'erreur se rapproche de sa valeur minimale pour les modèles entraînés sur 200 et 250 itérations, mais au-delà, l'erreur augmente. Il ne semble donc pas judicieux d'utiliser des nombres d'itérations aussi élevés. De plus, certaines reconstructions (figure 9) sont de qualité relativement médiocre : c'est le cas notamment du chiffre "0" pour T = 400, qui était jusque-là un chiffre relativement bien reconstruit, ainsi que des chiffres "5" et "8", pour T = 200.

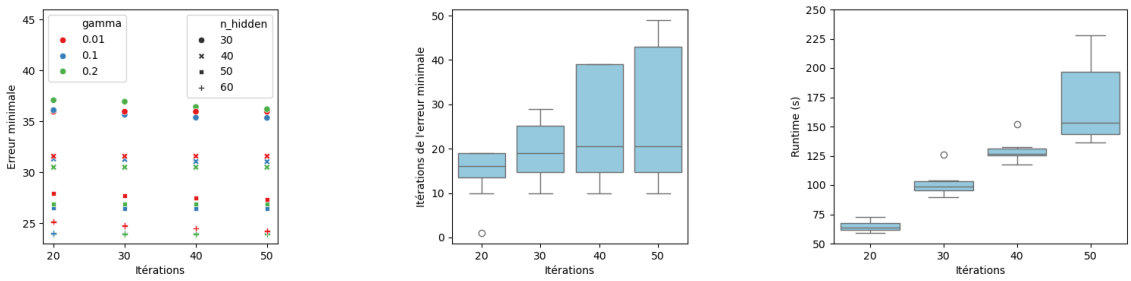
2.3 Recherche de paramètres optimaux

Des résultats précédents, nous avons constaté que 40 itérations semblaient suffisantes pour entrainer le modèle sans consommer trop de ressources. Nous décidons donc de tester plusieurs valeurs distribuées autour de cette valeur, à savoir 20, 30, 40 et 50. Nous testons 2 valeurs de γ supplémentaires, une significativement plus faible et une autre plus importante. Le nombre de neurones de la couche cachée est également étudié, avec des valeurs allant de 30 à 60 par pas de 10. L'ensemble des paramètres étudiés est listé dans le tableau 1.

N_v	L	N_h	T	gamma
784	16	30, 40, 50, 60	20, 30, 40, 50	0.01, 0.1, 0.2

TABLE 1 – Ensemble des paramètres testés

Nous pouvons constater d'après la figure 5a ce que nous avions remarqué précédemment : un nombre d'itérations plus élevé n'apporte pas forcément une diminution significative de l'erreur. De plus, l'itération moyenne pour laquelle l'erreur est minimale reste la même et se situe autour de 20 (figure 5b). La figure 5a montre également qu'un facteur influençant la performance de notre modèle est le nombre de neurones dans la couche cachée. C'est d'ailleurs ce facteur qui a le plus grand impact sur l'évolution de l'erreur minimale. Cela est compréhensible, car la couche cachée dans notre modèle sert à "capturer" l'information, du moins la structure, de nos données visibles. En augmentant le nombre de neurones, on augmente les capacités d'apprentissage de la structure des données visibles. Le facteur γ est lui aussi important : les points verts ($\gamma = 0.2$) sont généralement reliés à des erreurs plus faibles que les points rouges ($\gamma = 0.01$). Cela est d'autant plus le cas que le nombre de neurones cachés est grand. Cela fait sens, car le facteur γ est lié à la mise à jour des biais et des poids dans notre algorithme : faire varier ce facteur permet de faire varier la "vitesse à laquelle l'algorithme apprend". La figure 5c montre que les temps de calcul pour de grandes valeurs d'itérations sont en moyenne 2 à 3 fois plus longs. Cela confirme donc les premières observations que nous avions faites. Nous pouvons extraire des ensembles de paramètres pour lesquels l'erreur est significativement réduite : pour $\gamma = 0.2$, $N_h = 60$ et $T = 20$, l'erreur est d'environ 24%. La figure 6 présente les images reconstruites avec ces paramètres. Nous constatons que les chiffres sont mieux représentés, notamment le "5" et le "8", où l'on reconnaît les formes caractéristiques de leur représentation (la boucle du bas pour le "5" et les deux boucles pour le "8").



(a) (b) (c)

FIGURE 5 – Recherche de paramètres optimaux. (a) Evolution de l'erreur minimale. (b) Itération pour laquelle l'erreur est minimale. (c) Evolution du temps de calcul.

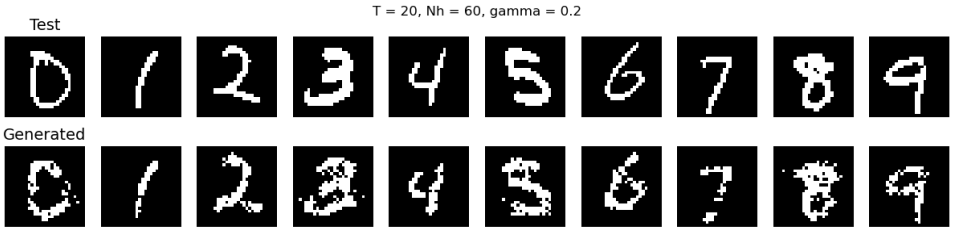


FIGURE 6 – Images reconstruites à partir de la RBM entrainée avec les paramètres optimaux

3 Conclusion

L'implémentation de notre machine de Boltzmann restreinte nous a permis de combiner les atouts du langage de programmation Python avec l'application de concepts de physique statistique à travers une thématique actuelle : les réseaux de neurones. Notre étude s'est concentrée sur la base de données MNIST et a consisté à générer des chiffres manuscrits à partir d'images d'entraînement. Le temps de calcul ainsi que le pourcentage d'erreur de l'algorithme ont été suivis, et plusieurs nombres d'itérations ont été testés. D'autres paramètres ont également été modifiés, notamment le taux d'apprentissage (γ) et le nombre de neurones dans la couche cachée. Les résultats montrent qu'un nombre d'itérations trop élevé entraîne une mauvaise reconstruction des chiffres, contrairement à des itérations plus courtes qui offrent de meilleures performances. Nous avons également mis en évidence l'importance du bon réglage des paramètres du modèle ainsi que leur signification : pour un nombre de neurones dans la couche visible fixé, les performances du modèle s'améliorent lorsque le nombre de neurones dans la couche cachée augmente, ce qui reflète une augmentation des capacités d'apprentissage de la structure des données visibles, particulièrement pour des valeurs de γ comprises entre 0.1 et 0.2. Une valeur trop faible de ce dernier entraîne une augmentation du temps de calcul et une vision "trop étroite" de la structure des données cachées.

4 Annexes

4.1 Reconstruction d'images

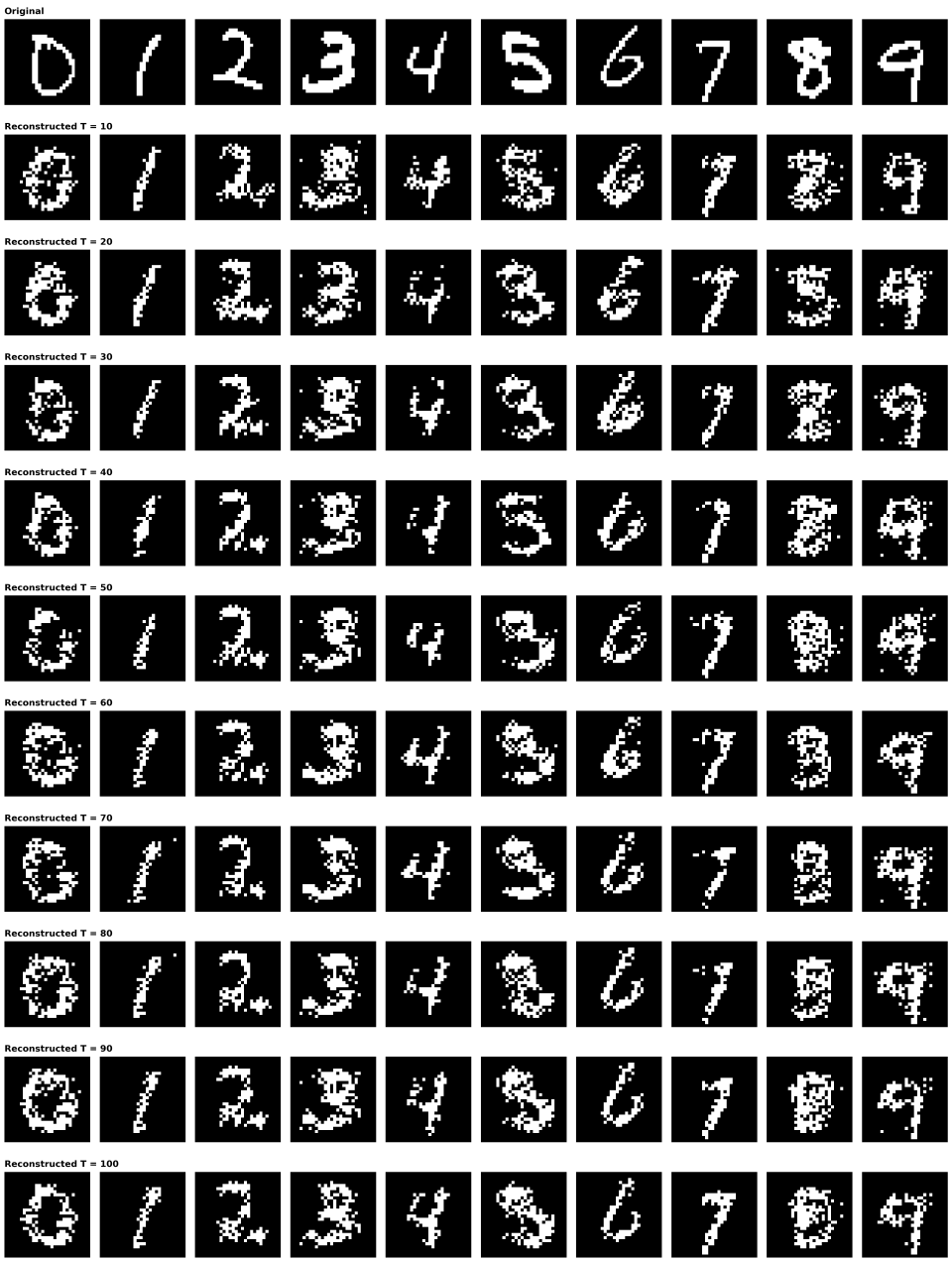
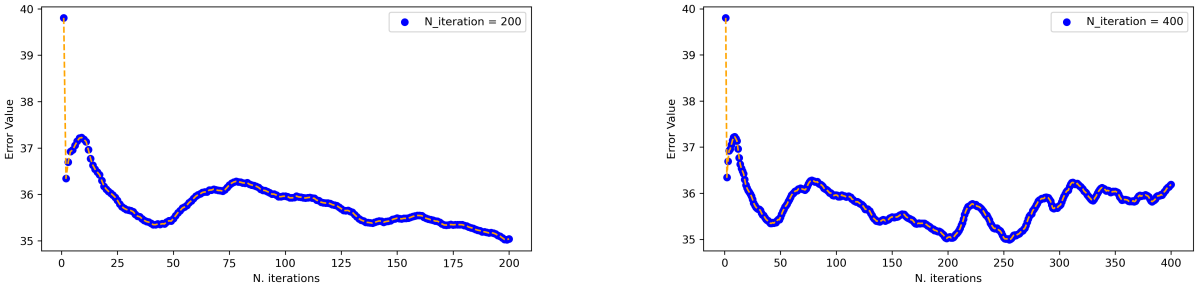


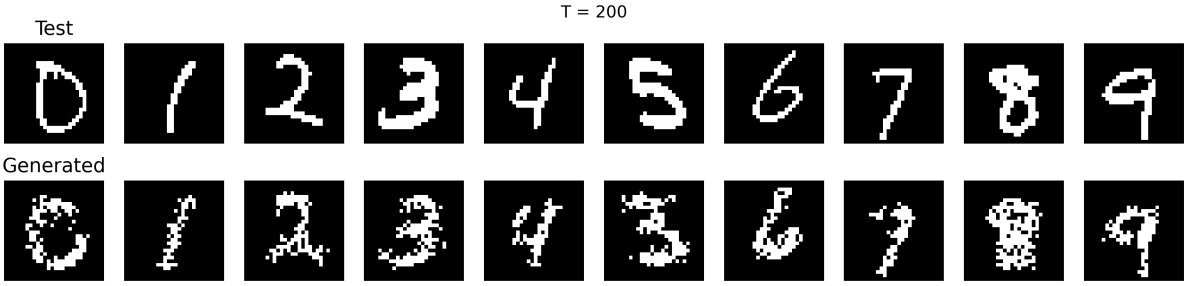
FIGURE 7 – Images originale issues de MNIST et images générées par la RBM pour différentes valeurs d'itération

4.2 Figures supplémentaires pour T = 200 et T = 400

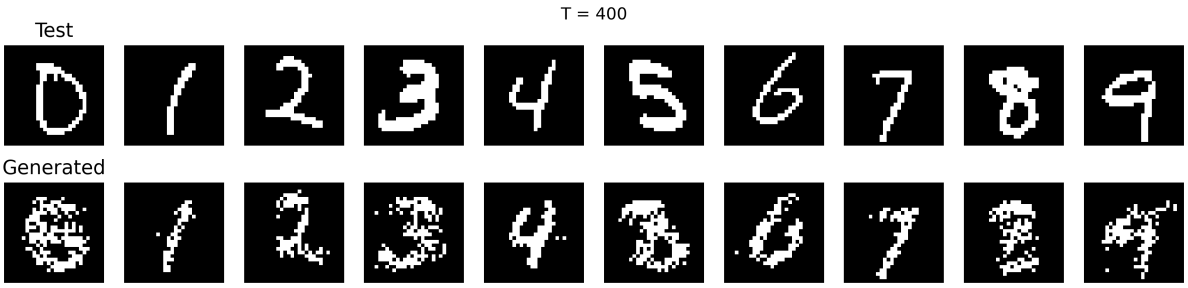


(a) (b)

FIGURE 8 – Erreurs pour un nombre d'itérations de (a) 200 et (b) 400



(a)



(b)

FIGURE 9 – Images reconstruites pour un nombre d'itérations de (a) 200 et (b) 400

4.3 Code

code

February 9, 2025

[1]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.model_selection import train_test_split
from sklearn.utils import gen_batches
from sklearn.datasets import fetch_openml
from sklearn.linear_model import LinearRegression

import time

import pickle
import joblib
import os

import re
```

1 MNIST

[2]:
```
# Download data
X, y = fetch_openml("mnist_784", version=1, return_X_y=True, as_frame=False)
# Save data in file 'Xy.npz'
np.savez('Xy', X=X, y=y)
```

2 Restricted Boltzmann Machine

[15]:
```
class RBM():
    """
    Restricted Boltzmann machine

    Reference:
        Hinton, G. E., Osindero, S. and Teh, Y.
        *A fast learning algorithm for deep belief nets*.
        Neural Computation 18, pp 1527-1554.
```

```
        pdf: https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf
    """
    def __init__(self, Nv, Nh, T, L, gamma):
        self.Nv = Nv              # number of visible units (28^2)
        self.Nh = Nh              # number of hidden units (30)
        self.gamma = gamma        # learning rate (0.1)
        self.T = T                # number of iterations
        self.L = L                # batch data size
        self.errors = []          # to extract errors values for one iteration

        # random number generator
        rng = np.random.default_rng(50419) # arbitrary seed

        # initial weights
        self.W = rng.normal(0, 0.01, size=(Nv, Nh))

        # initial biases
        self.a = rng.normal(0, 1, size=Nv)
        self.b = rng.normal(0, 1, size=Nh)


    def sigmoid(self, x):
        """
        Sigmoid function
        """
        return 1 / (1 + np.exp(-x))

    def optimizeRBM(self, X):
        """
        Training the RBM using the contrasted divergence of Hinton

        Parameters
        ----------
        X: training data

        Returns
        -------
        self: the trained RBM with the optimal set of parameters
        """
        rng = np.random.default_rng(12) # uniform
        for t in range(self.T):
            error_t = 0
            batches = list(gen_batches(X.shape[0], self.L))
            # constrasted divergence
            for batch in batches:
                # Positive sweep (data average)
                v = X[batch.start:batch.stop] # dim = (L, Nv)
```

```
                # Lv = len(v)
                prob_hPos = self.sigmoid(np.dot(v, self.W) + self.b)  #␣
↪p(h=1|v) dim = (L,Nh)
                hvPos = np.dot(v.T, prob_hPos) # scalar product <v prob_hPos>␣
↪dim = (Nv,Nh)

                # Negative sweep (model average)
                vmean = self.sigmoid(np.dot(prob_hPos, self.W.T) + self.a)  #␣
↪p(v=1|h) dim = (L,Nv)
                prob_hNeg = self.sigmoid(np.dot(vmean, self.W) + self.b)  #␣
↪p(h=1|vmean) dim = (L, Nh)

                hvNeg = np.dot(vmean.T, prob_hNeg) # scalar product <vmean␣
↪prob_hNeg> dim = (Nv, Nh)
                error = np.sum((v-vmean)**2)/self.L

                # update parameters and error
                self.W += self.gamma*(hvPos - hvNeg)/self.L
                self.a += self.gamma*np.mean(v - vmean)
                self.b += self.gamma*np.mean(prob_hPos - prob_hNeg)
                error_t += error

            # Error
            error_t = error_t/len(batches)
            self.errors.append(error_t)
            print(f"iteration: {1+t}/{self.T} \t{'error:'} {error_t}")

    def identify_X(self, X):
        """
        Use the trained RBM to reconstruct test data

        Parameters
        ----------
        X: test data

        Returns
        -------
        self: visible state with the identified data
        """
        rng = np.random.default_rng(42)  # initialize a random number generator␣
↪with a fixed seed
        v = X  # initialize visible with the selected test data

        prob_h = self.sigmoid(np.dot(v, self.W) + self.b)  # p(h=1|v)
        state_h = (prob_h > rng.random(prob_h.shape)).astype(int)
```

```
        # Reconstruct visible state from hidden state
        prob_v = self.sigmoid(np.dot(state_h, self.W.T) + self.a)  # p(v=1|h)
        state_v = (prob_v > rng.random(prob_v.shape)).astype(int)

        return state_v
```

3 Data overview

[3]:
```
# MNIST data base of hand written digits

file_Xy = np.load('Xy.npz', allow_pickle=True)
X_dat = file_Xy['X'] # (70000, 784) images 28x28
y_dat = file_Xy['y'] # (70000) labels
```

[ ]:
```
print(len(X_dat[0]))
X_dat[0]
```

[ ]:
```
def X_image(N_img, X, y):
    _, axes = plt.subplots(nrows=1, ncols=N_img, figsize=(10, 3))
    for ax, image, label in zip(axes, X[:5].reshape((N_img, 28, 28)), y[:
 ↪N_img]):
        ax.set_axis_off()
        ax.imshow(1 - image, cmap='gray')
        ax.set_title("Training:"+label)

# transform data to binary

X = X_dat/255 # normalize to (0,1)
X = X > 0.5
X_image(5, X, y_dat)
```

[6]:
```
# Train test split

X_train, X_test, Y_train, Y_test = train_test_split(X, y_dat, test_size=10000,␣
 ↪random_state=42, stratify=y_dat)
```

[ ]:
```
list(gen_batches(X_train.shape[0], 16))
```

[ ]:
```
batch = list(gen_batches(X_train.shape[0], 16))[0]

X_train[batch.start:batch.stop]
```

4 Training the model

[ ]:
```
rbm = RBM(Nv=X_train.shape[1], Nh=30, T=40, L=16, gamma=0.1)

# Start timer
start_time = time.time()
rbm.optimizeRBM(X_train)
end_time = time.time()
# End timer

# Calculate and display the runtime
runtime = end_time - start_time
print(f"Training completed in {runtime:.2f} seconds.")
```

[17]:
```
# from the test set get images with labels {0, 1, ..., 9)

test_labels = [np.where(Y_test == str(i))[0][0] for i in range(10)] # labels␣
 ↪indices
test_images = X_test[test_labels] # 10 images with the digits
```

[18]:
```
# RBM reconstruction of the test images

generated = rbm.identify_X(test_images)
```

[ ]:
```
# Plot the original and reconstructed images

fig, ax = plt.subplots(nrows=2, ncols=10, sharey=True, figsize=(12, 3))
for c in range(10):
    # Original test images
    ax[0, c].imshow(test_images[c].reshape(28, 28), cmap='gray')  # Reshape to␣
 ↪28x28 for MNIST
    ax[0, c].set_axis_off()

    # Reconstructed images
    ax[1, c].imshow(generated[c].reshape(28, 28), cmap='gray')
    ax[1, c].set_axis_off()

ax[0, 0].set_title("Test", fontsize=14)
ax[1, 0].set_title("Generated", fontsize=14)


fig.subplots_adjust(left=0.01, right=0.99, bottom=0.01, top=0.91, wspace=0.2,␣
 ↪hspace=0.02)
plt.suptitle(f"T = {rbm.T}")
plt.show()
```

[99]:
```
# Test multiple values for T

runtime_list = []
generated_list = []
errors_list = []

for n_iter in np.arange(10,110,10):
    rbm = RBM(Nv=X_train.shape[1], Nh=30, T=n_iter, gamma=0.1)

    # Start timer
    start_time = time.time()
    rbm.optimizeRBM(X_train)
    end_time = time.time()
    # End timer

    # Calculate and display the runtime
    runtime = end_time - start_time
    print(f"Training completed in {runtime:.2f} seconds.")

    # RBM reconstruction of the test images
    generated = rbm.identify_X(test_images)

    runtime_list.append(runtime)
    generated_list.append(generated)
    errors_list.append(rbm.errors)

# Save as dictionary with pickle
tp_rbm = {'runtime':runtime_list,
          'generated':generated_list,
          'errors':errors_list}

try:
    with open('models/tp_rbm.pkl', 'wb') as f:
        pickle.dump(tp_rbm, f)
    print("Pickle file saved successfully.")
except Exception as e:
    print(f"Error saving pickle file: {e}")
```

[18]:
```
# Read in pickle file

with open('models/tp_rbm.pkl', 'rb') as f:
    tp_rbm = pickle.load(f)
```

[ ]:
```
plt.figure(figsize=(12, 4))

# Perform linear regression
y_values = tp_rbm['runtime'].copy()
```

```
x_values = np.array([i for i in np.arange(10,110,10)]).reshape(-1, 1)  #␣
 ↪Reshape for sklearn
model = LinearRegression()
model.fit(x_values, y_values)
y_pred = model.predict(x_values)

# Fig 1 : Runtime vs N. iterations
plt.subplot(121)
plt.scatter(np.arange(10, 110, 10), tp_rbm['runtime'], marker='+',␣
 ↪color='blue', label='runtime')
plt.plot(x_values, y_pred, '--', color='red', label=f"Reg. linéaire\ny =␣
 ↪{round(model.coef_[0],2)}x {round(model.intercept_, 2)}")
plt.xlabel("N. itérations")
plt.ylabel("Runtime (s)")
plt.legend()
# plt.title("Runtime vs N. iterations")

# Fig 2 : Errors boxplot
plt.subplot(122)
sns.boxplot(data=tp_rbm['errors'], color='skyblue')
plt.xlabel("N. itérations")
plt.xticks(ticks=np.arange(10), labels=[str(i) for i in np.arange(10, 110, 10)])
plt.ylabel("Erreur")
# plt.title("Error Distribution")

plt.tight_layout()

plt.savefig(f"figures/runtime_error_n_iter.png", dpi=300)

plt.show()
```

[ ]:
```
fig, axes = plt.subplots(3,4, figsize=(20, 10))

axes = axes.flatten()

for idx, (j, ax) in enumerate(zip(tp_rbm['errors'], axes)):

    # Generate x-axis values for the current array
    iterations = np.arange(1, len(j) + 1)

    # Plot the data
    ax.scatter(iterations, j, color='blue', label=f"N_iteration = {(idx + 1) *␣
 ↪10}")
    ax.plot(iterations, j, linestyle='--', color='orange')

    # Minimum error value and its corresponding iteration
    min_error = min(j)
```

```
    min_iteration = iterations[j.index(min_error)]

    # Add vertical and horizontal dashed lines at the minimum
    ax.axhline(y=min_error, linestyle='--', color='green', alpha=0.7)
    ax.axvline(x=min_iteration, linestyle='--', color='green', alpha=0.7)
    ax.scatter(min_iteration, min_error, color='red', label=f"$(x, y)_{{min}}$␣
 ↪= ({min_iteration}, {min_error:.2f})")

    # Set title and labels
    ax.set_ylabel('Error Value')
    ax.set_xlabel('Iteration')
    ax.legend()

for idx in range(len(tp_rbm['errors']), len(axes)):
    axes[idx].axis('off')

plt.savefig(f"figures/error_all.png", dpi=300, bbox_inches='tight')

plt.tight_layout()
plt.show()
```

[ ]:
```
fig, ax = plt.subplots(nrows=11, ncols=10, sharey=True, figsize=(15, 20))

# Original test images (row 0)
for c in range(10):
    ax[0, c].imshow(test_images[c].reshape(28, 28), cmap='gray')  # Reshape to␣
 ↪28x28 for MNIST
    ax[0, c].set_axis_off()

ax[0, 0].set_title("Original", fontsize=10, fontweight="bold", loc="left")

# Reconstructed images (rows 1 to 10)
for i, reconstructed_images in enumerate(tp_rbm['generated']):
    for c in range(10):
        ax[i + 1, c].imshow(reconstructed_images[c].reshape(28, 28),␣
 ↪cmap='gray')
        ax[i + 1, c].set_axis_off()

    # Add row title to the first subplot of each row
    ax[i + 1, 0].set_title(f"Reconstructed T = {10*(i + 1)}", fontsize=10,␣
 ↪fontweight="bold", loc="left")

plt.tight_layout()

plt.savefig("figures/generated_images_all.png", dpi=300, bbox_inches='tight')

plt.show()
```

[28]:
```
# Generate all plot individually

for i,j in zip(np.arange(10,110,10),tp_rbm['generated']):

    fig, ax = plt.subplots(nrows=2, ncols=10, sharey=True, figsize=(12, 3))
    for c in range(10):
        # Original test images
        ax[0, c].imshow(test_images[c].reshape(28, 28), cmap='gray')
        ax[0, c].set_axis_off()

        # Reconstructed images
        ax[1, c].imshow(j[c].reshape(28, 28), cmap='gray')
        ax[1, c].set_axis_off()

    ax[0, 0].set_title("Test", fontsize=14)
    ax[1, 0].set_title("Generated", fontsize=14)

    fig.subplots_adjust(left=0.01, right=0.99, bottom=0.01, top=0.91, wspace=0.
↪2, hspace=0.02)
    plt.suptitle(f"T = {i}")

    plt.savefig(f"figures/generated_images_T{i}.png", dpi=300)

    plt.show()
```

5 Train with T=200

[105]:
```
rbm = RBM(Nv=X_train.shape[1], Nh=30, T=200, gamma=0.1)

# Start timer
start_time = time.time()
rbm.optimizeRBM(X_train)
end_time = time.time()
# End timer

# Calculate and display the runtime
runtime = end_time - start_time
print(f"Training completed in {runtime:.2f} seconds.")

# from the test set get images with labels {0, 1, ..., 9)
test_labels = [np.where(Y_test == str(i))[0][0] for i in range(10)] # labels␣
↪indices
test_images = X_test[test_labels] # 10 images with the digits
generated = rbm.identify_X(test_images) # RBM reconstruction of the test images
```

```
try:
    with open('models/rbm200.pkl', 'wb') as f:
        pickle.dump(rbm, f)
    print("Pickle file saved successfully.")
except Exception as e:
    print(f"Error saving pickle file: {e}")


# Plot the input and output images
fig, ax = plt.subplots(nrows=2, ncols=10, sharey=True, figsize=(12, 3))
for c in range(10):
    # Original test images
    ax[0, c].imshow(test_images[c].reshape(28, 28), cmap='gray')  # Reshape to␣
 ↪28x28 for MNIST
    ax[0, c].set_axis_off()

    # Reconstructed images
    ax[1, c].imshow(generated[c].reshape(28, 28), cmap='gray')  # Reshape to␣
 ↪28x28 for MNIST
    ax[1, c].set_axis_off()

# Add titles
ax[0, 0].set_title("Test", fontsize=14)
ax[1, 0].set_title("Generated", fontsize=14)

# Adjust layout
fig.subplots_adjust(left=0.01, right=0.99, bottom=0.01, top=0.91, wspace=0.2,␣
 ↪hspace=0.02)
plt.suptitle(f"T = {rbm.T}")

plt.savefig(f"figures/generated_images_T200.png", dpi=300)

# Display the plot
plt.show()
```

[ ]:
```
with open('models/rbm200.pkl', 'rb') as f:
    rbm200 = pickle.load(f)

# Original y_values and append new value
y_values = tp_rbm['runtime'].copy()
y_values = y_values + [705.77]

# Define x-values including the new point
x_values = np.array([i for i in np.arange(10,110,10)] + [200]).reshape(-1, 1) ␣
 ↪# Reshape for LinearReg

# Perform linear regression
```

```
model = LinearRegression()
model.fit(x_values, y_values)

# Predict y-values using the linear regression model
y_pred = model.predict(x_values)

# Plot the scatter plot and regression line
plt.figure(figsize = (8,4))
plt.scatter(np.arange(1,201,1), rbm200.errors, color='blue',␣
 ↪label=f"N_iteration = 200")
plt.plot(np.arange(1,201,1), rbm200.errors, linestyle='--', color='orange')
plt.xlabel("N. iterations")
plt.ylabel('Error Value')
plt.legend()

plt.savefig(f"figures/error_T200.png", dpi=300,  bbox_inches='tight')

plt.show()
```

6 Train with T = 400

[107]:
```
rbm = RBM(Nv=X_train.shape[1], Nh=30, T=400, gamma=0.1)

# Start timer
start_time = time.time()
rbm.optimizeRBM(X_train)
end_time = time.time()
# End timer

# Calculate and display the runtime
runtime = end_time - start_time
print(f"Training completed in {runtime:.2f} seconds.")

# from the test set get images with labels {0, 1, ..., 9)
test_labels = [np.where(Y_test == str(i))[0][0] for i in range(10)] # labels␣
 ↪indices
test_images = X_test[test_labels] # 10 images with the digits
generated = rbm.identify_X(test_images) # RBM reconstruction of the test images


try:
    with open('models/rbm400.pkl', 'wb') as f:
        pickle.dump(rbm, f)
    print("Pickle file saved successfully.")
except Exception as e:
    print(f"Error saving pickle file: {e}")
```

```
# Plot the input and output images
fig, ax = plt.subplots(nrows=2, ncols=10, sharey=True, figsize=(12, 3))
for c in range(10):
    # Original test images
    ax[0, c].imshow(test_images[c].reshape(28, 28), cmap='gray')  # Reshape to␣
 ↪28x28 for MNIST
    ax[0, c].set_axis_off()

    # Reconstructed images
    ax[1, c].imshow(generated[c].reshape(28, 28), cmap='gray')  # Reshape to␣
 ↪28x28 for MNIST
    ax[1, c].set_axis_off()

# Add titles
ax[0, 0].set_title("Test", fontsize=14)
ax[1, 0].set_title("Generated", fontsize=14)

# Adjust layout
fig.subplots_adjust(left=0.01, right=0.99, bottom=0.01, top=0.91, wspace=0.2,␣
 ↪hspace=0.02)
plt.suptitle(f"T = {rbm.T}")

plt.savefig(f"figures/generated_images_T400.png", dpi=300)

# Display the plot
plt.show()
```

[ ]:
```
with open('models/rbm400.pkl', 'rb') as f:
    rbm400 = pickle.load(f)

# Original y_values and append new value
y_values = tp_rbm['runtime'].copy()
y_values = y_values + [705.77, 958.39]

# Define x-values including the new point
x_values = np.array([i for i in np.arange(10,110,10)] + [200,400]).reshape(-1,␣
 ↪1)  # Reshape for sklearn

# Perform linear regression
model = LinearRegression()
model.fit(x_values, y_values)

# Predict y-values using the linear regression model
y_pred = model.predict(x_values)
```

```
# Plot the scatter plot and regression line
plt.figure(figsize = (8,4))
plt.scatter(np.arange(1,401,1), rbm400.errors, color='blue',␣
 ↪label=f"N_iteration = 400")
plt.plot(np.arange(1,401,1), rbm400.errors, linestyle='--', color='orange')
plt.xlabel("N. iterations")
plt.ylabel('Error Value')
plt.legend()

plt.savefig(f"figures/error_T400.png", dpi=300,  bbox_inches='tight')

plt.show()
```

6.1 Fine tuning

[ ]:
```
configurations = []

for n_iter in [20,30,40,50]:
    for n_hidden in [30,40,50,60]:
        for gamma_value in [0.01, 0.1, 0.2]:

            rbm = RBM(Nv=X_train.shape[1], Nh=n_hidden, T=n_iter, gamma =␣
 ↪gamma_value, L=16)

            start_time = time.time()

            rbm.optimizeRBM(X_train)
            end_time = time.time()
            # End timer

            # Calculate and display the runtime
            runtime = end_time - start_time

            # # Save the trained model to a file
            model_filename =␣
 ↪f'rbm_model_niter_{n_iter}_nhidden_{n_hidden}_gamma_{gamma_value}.pkl'
            joblib.dump(rbm, model_filename)

            # Save the configuration and runtime to the list
            configurations.append({
                'n_iter': n_iter,
                'n_hidden': n_hidden,
                'gamma': gamma_value,
                'runtime': runtime,
                'model_filename': model_filename  # Store the model filename
            })
```

```
# Convert the list of configurations into a DataFrame
config_df = pd.DataFrame(configurations)

# Optionally, save the DataFrame to a CSV file
config_df.to_csv("rbm_configurations.csv", index=False)
```

[21]:
```
df_runtime = pd.read_csv("rbm_configurations.csv")

models_dir = "models/"
models = {}

for i in os.listdir(models_dir):
    if i.startswith("rbm_model_niter"):
        models[i[:-4]] = joblib.load(os.path.join(models_dir, i))
```

[ ]:
```
models
```

[ ]:
```
df_metrics_header = ["n_iter", "n_hidden", "gamma", "min_error_idx",␣
 ↪"min_error"]

def extract_model_info(model):
    data = []
    data.append(model.T)
    data.append(model.Nh)
    data.append(model.gamma)
    data.append(np.array(model.errors).argmin())
    data.append(np.array(model.errors).min())
    return data


a = extract_model_info(models['rbm_model_niter_20_nhidden_30_gamma_0.01'])
b = extract_model_info(models['rbm_model_niter_20_nhidden_30_gamma_0.2'])

rows = [extract_model_info(models[i]) for i in models.keys()]

df_models = pd.DataFrame(rows, columns=df_metrics_header)

df_metrics = pd.merge(df_models, df_runtime, on=['n_iter', 'n_hidden',␣
 ↪'gamma'], how='left').sort_values('n_iter').reset_index(drop=True)
df_metrics
```

[ ]:
```
plt.figure(figsize=(4,4))

# Scatter plot
ax = sns.scatterplot(data=df_metrics, x='n_iter', y='min_error', hue='gamma',␣
 ↪style='n_hidden', palette='Set1')
```

```
plt.xticks([20,30,40,50])
plt.ylim((23,46))
plt.xlabel("Itérations")
plt.ylabel("Erreur minimale")

# Get handles and labels for both hue and style

handles, labels = ax.get_legend_handles_labels()

gamma_handles = handles[:1+len(df_metrics['gamma'].unique())]
gamma_labels = labels[:1+len(df_metrics['gamma'].unique())]

n_hidden_handles = handles[len(df_metrics['n_hidden'].unique()):]
n_hidden_labels = labels[len(df_metrics['n_hidden'].unique()):]

# Create new legend
legend1 = plt.legend(gamma_handles, gamma_labels, loc='upper left', fontsize=10)
legend2 = plt.legend(n_hidden_handles, n_hidden_labels, loc='upper right',␣
 ↪fontsize=10)

# Add both legends to the plot
ax.add_artist(legend1)

plt.savefig(f"figures/all_metrics.png", dpi=100, pad_inches=0.1)

plt.show()
```

[ ]:
```
plt.figure(figsize=(4,4))
sns.boxplot(df_metrics, x='n_iter', y='min_error_idx', color='skyblue')
plt.xlabel("Itérations")
plt.ylabel("Itérations de l'erreur minimale")
plt.savefig(f"figures/min_error_indice.png", dpi=100, pad_inches=0.1)
plt.show()
```

[ ]:
```
plt.figure(figsize=(5,4))
sns.boxplot(df_metrics, x='n_iter', y='runtime', color='skyblue')
plt.xlabel("Itérations")
plt.ylabel("Runtime (s)")
plt.ylim((50,250)) # remove outlier
plt.savefig(f"figures/runtime_boxplot.png", dpi=100, pad_inches=0.1)
plt.show()
```

[ ]:
```
# Training and test with "best_rbm"

best_rbm = RBM(Nv=X_train.shape[1], Nh=60, T=20, gamma = 0.2, L=16)

start_time = time.time()
```

```
best_rbm.optimizeRBM(X_train)
end_time = time.time()
# End timer

# Calculate and display the runtime
runtime = end_time - start_time

# Test
generated_best_rbm = best_rbm.identify_X(test_images)
```

[ ]:
```
# Plot
fig, ax = plt.subplots(nrows=2, ncols=10, sharey=True, figsize=(12, 3))
for c in range(10):
    # Original test images
    ax[0, c].imshow(test_images[c].reshape(28, 28), cmap='gray')  # Reshape to␣
 ↪28x28 for MNIST
    ax[0, c].set_axis_off()

    # Reconstructed images
    ax[1, c].imshow(generated_best_rbm[c].reshape(28, 28), cmap='gray')  #␣
 ↪Reshape to 28x28 for MNIST
    ax[1, c].set_axis_off()

# Add titles
ax[0, 0].set_title("Test", fontsize=14)
ax[1, 0].set_title("Generated", fontsize=14)

# Adjust layout
fig.subplots_adjust(left=0.01, right=0.99, bottom=0.01, top=0.91, wspace=0.2,␣
 ↪hspace=0.02)
plt.suptitle(f"T = {best_rbm.T}, Nh = {best_rbm.Nh}, gamma = {best_rbm.gamma}")
plt.savefig(f"figures/best_rbm_hyperparameters.png", dpi=300,␣
 ↪bbox_inches='tight')
plt.show()
```

Bibliographie

[1] J. Hoppfield. « Artificial Neural Networks ». In : *IEEE Circuits and Devices Magazine* (1988).

[2] S. Osindero et Y.-W. Teh G. E. Hinton. « A Fast Learning Algorithm for Deep Belief Nets ». In : *Neural Computation* (2006).

[3] J. Tubiana. « Restricted Boltzmann machines : from compositional representations to protein sequence analysis ». Thèse de doct. Ecole Doctorale n° 564, 2018.