Restricted Boltzmann Machines for pattern retrieval

Corso: Modelli di reti neurali

Facoltà
di Scienze Matematiche
Fisiche e Naturali
Sapienza
Università di Roma

Candidato:
Andrea Mantuano

Matricola:
1739874

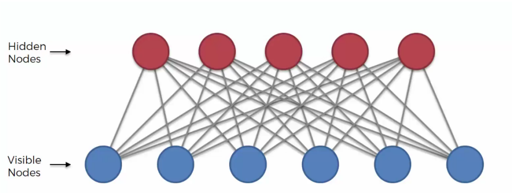
25 Marzo 2022

Table of contents

1. Introduction
2. Implementation
3. Results
4. Plans for further work

Restricted Boltzamann machines

- Recurrent neural network
- Complete bipartite graph
- Simmetric weights
- Energy based model



- Energy function

$$E(v,h) = -\sum_{i\in visible} v_0^i v_i - \sum_{j\in hidden} v_0^j h_j - \sum_{i,j} v_i h_j w_{ij} = -\mathbf{v}_O^T\mathbf{v} - \mathbf{h}_0^T\mathbf{h} - \mathbf{v}^T W\mathbf{h}$$

- Probability of (v,h)

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

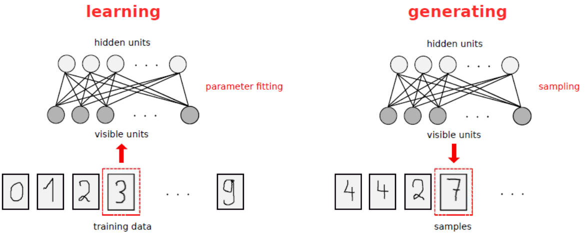
- Conditional distributions

$$\mathbb{P}(h_i = 1 \,|\, v) = \sigma(v \cdot W + h_0^i)$$

$$\mathbb{P}(v_i = 1 \,|\, h) = \sigma(h \cdot W^T + v_0^i)$$

Generative model

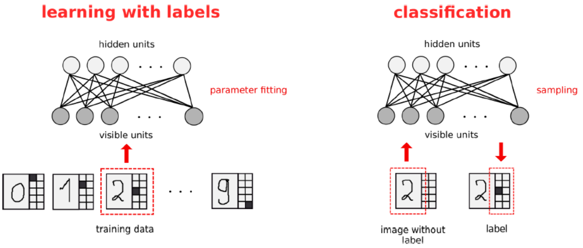
- Fitting RBM learn a distribution $q(x)$ such that to obtain $p_{W,\theta}(v) \sim q(x)$.
- Iteratively performs one step of Contrastive Divergence on dataset



- Allows us to sample new data from the learned distribution or reconstruct corrupted ones

Classification model

- Model a joint distribution on inputs x and taget classes y
- There are two weight matrices:
  - W between x and h
  - U between y and h



- A new example of the *pattern*, without label, is passed as input to the trained model and this will be able to predict the corresponding label.

The learning process

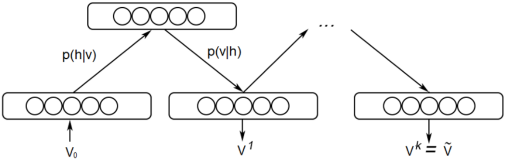
- Goal: adjust the parameters, weights and bias, so that the represented distribution is the best possible approximation of the target distribution
- **Kullback-Leibler**: $D(q||p_\lambda) = \sum q(v) \log \frac{p_\lambda(v)}{q(v)}$
- Minimize the function:

$$
\begin{aligned}
q(v)\frac{\partial}{\partial\lambda}\log Z \quad &= \frac{1}{Z}\sum q(v)\frac{\partial}{\partial\lambda}e^{-E(v,h)} = \sum q(v)\frac{e^{-E(v,h)}}{Z}\frac{\partial}{\partial\lambda}E(v,h) \\
&= \sum p_\lambda(v,h)\frac{\partial}{\partial\lambda}E(v,h) =< \frac{\partial}{\partial\lambda}E_\lambda(v,h) >_\lambda \\
q(v)\frac{\partial}{\partial\lambda}\log Z(v) \quad &= \frac{1}{Z(v)}\sum q(v)\frac{\partial}{\partial\lambda}e^{-E(v,h)} = \sum q(v)\frac{e^{-E(v,h)}}{Z(v)}\frac{\partial}{\partial\lambda}E(v,h) \\
&= \sum q(v)p_\lambda(h|v)\frac{\partial}{\partial\lambda}E(v,h) =< \frac{\partial}{\partial\lambda}E_\lambda(v,h) >_{v,\lambda}
\end{aligned}
$$

$$
\implies \frac{\partial}{\partial\lambda}D(q||p_\lambda) \propto \mathbb{E}_{v,h}\left(\frac{\partial}{\partial\lambda}E\lambda(v,h)\right) - \mathbb{E}_\lambda\left(\frac{\partial}{\partial\lambda}E(v,h)\right)
$$

Contrastive Divergence

- Gradient is hard to compute $\Rightarrow$ run Markov chain to approximate
  - sampling of $h(t)$ by exploiting the known distribution $p(h|v(t))$
  - sampling of $v(t+1)$ using the distribution just calculated $p(h|v(t))$.



- One step of Gibbs Sampling is sufficient

$$CD_k(\lambda, v^{(0)}) = \sum_h p(h|v^{(k)}) \frac{\partial}{\partial \lambda} E(v^{(k)}, h) - \sum_h p(h|v^{(0)}) \frac{\partial}{\partial \lambda} E(v^{(0)}, h)$$

Table of contents

1. Introduction
2. Implementation
3. Results
4. Plans for further work

```python
def __init__(self, n_visible, n_hidden, learning_rate = 0.1, batch_size = 1,
             n_iterations = 100, classifier = False, n_label = 0):

    self.n_visible = n_visible
    self.n_hidden = n_hidden
    self.lr = learning_rate
    self.batch_size = batch_size
    self.n_iterations = n_iterations

    self.training_errors = []
    self.training_reconstructions = []
    self.W = np.zeros(shape = (n_visible, self.n_hidden))
    self.v0 = np.zeros(n_visible)       # Bias visible
    self.h0 = np.zeros(self.n_hidden)   # Bias hidden

    if classifier:
        self.n_label = n_label
        self.U = np.zeros(shape = (n_label, self.n_hidden))
        self.z0 = np.zeros(n_label)

def _initialize_weights(self, classifier):
    self.W = np.random.normal(scale = 1, size = (self.n_visible, self.n_hidden))

    if classifier:
        self.U = np.random.normal(scale = 1, size = (self.n_label, self.n_hidden))

def _train(self, X, y = None, classifier = False):

    self._initialize_weights(classifier)
    if classifier:
        self._CD1_Classification(X,y)
    else: self._CD1_Generative(X)
```

```python
def _mean_hiddens(self, v):
    #Computes the probabilities P(h=1|v).

    return sigma(v.dot(self.W) + self.h0)

def _sample_hiddens(self, v):
    #Sample from the distribution P(h|v).

    p = self._mean_hiddens(v)
    return self._sample(p)

def _sample_visibles(self, h):
    #Sample from the distribution P(v|h).

    p = sigma(h.dot(self.W.T) + self.v0)
    return self._sample(p)

def _sample(self, X):

    return X > np.random.random_sample(size = X.shape)
```

```python
def _CD1_Generative(self, X):

    for _ in tqdm(range(self.n_iterations)):
        batch_errors = []

        for batch in batch_iterator(X, batch_size = self.batch_size):
            #v_0 = batch

            # Positive phase ---> E_v,j,o[sisj] = E_s(0)[sisj]
            positive_hidden = self._mean_hiddens(batch)  # E(h)_0=1*P(h=1|v) -0*P(h=-1|v)
            positive_associations = batch.T.dot(positive_hidden) #E(h)_0*v_0
            hidden_states = self._sample_hiddens(batch) # hidden to use in the second part of sample

            # Negative phase ---> E_j,o[sisj] = E_s(inf)[sisj]
            negative_visible = self._sample_visibles(hidden_states) # v_k
            negative_hidden = self._mean_hiddens(negative_visible)   # E(h)_k=1*P(h=1|v) -0*P(h=-1|v)
            negative_associations = negative_visible.T.dot(negative_hidden) #E(h)_k*v_k

            self.W  += self.lr * (positive_associations - negative_associations)
            self.h0 += self.lr * (positive_hidden.sum(axis = 0) - negative_hidden.sum(axis = 0))
            self.v0 += self.lr * (batch.sum(axis = 0) - negative_visible.sum(axis = 0))

            batch_errors.append(np.mean((batch - negative_visible) ** 2))

        self.training_errors.append(np.mean(batch_errors))

        # Reconstruct a batch of images from the training set
        self.training_reconstructions.append(self._reconstruct(X[:25]))
        if np.mean(batch_errors)*100 < 1: return
```

Table of contents

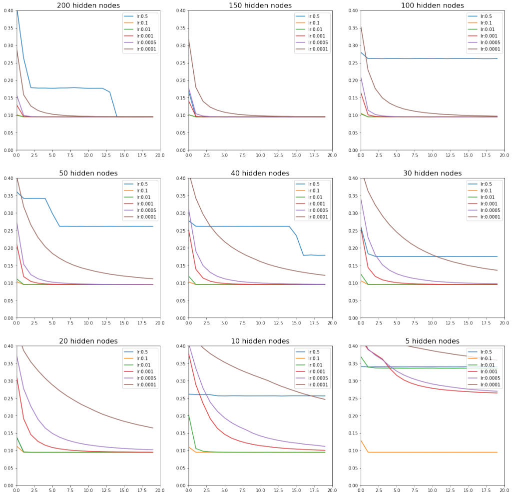
1. Introduction
2. Implementation
3. Results
4. Plans for further work

Rademachers variable

- 5 random patterns, pixel ~ $Be(0.5)$

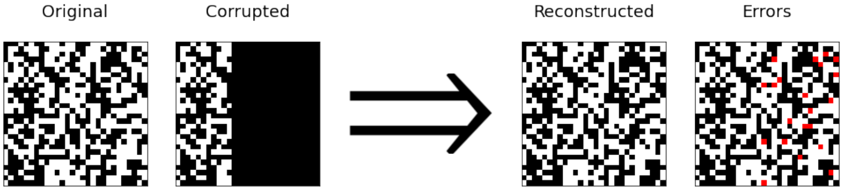
- dataset made of 2000 copies of each flipped archetype
- 70% trainset - 30% testset

Training error plot

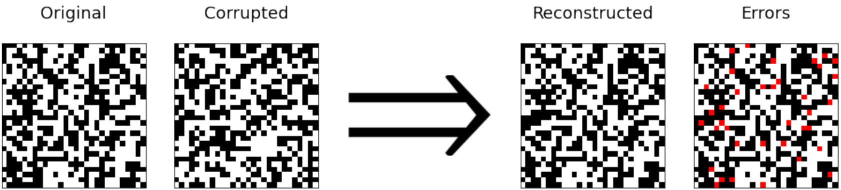


- n° hidden nodes: 100
- learning rate: 0.001

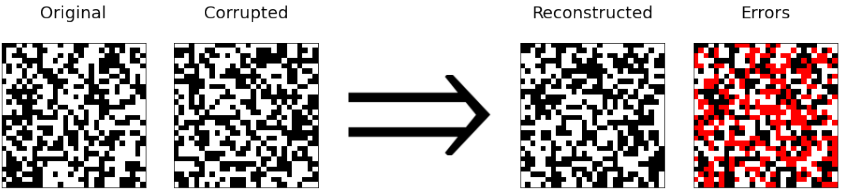
- only 11 columns over 28
- Accuracy of: 95.59 %



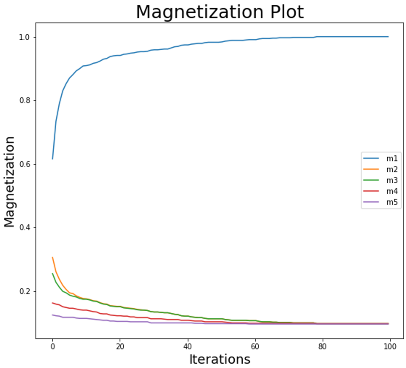
- After flipping 40 % of pixels the accuracy is: 95.28 %

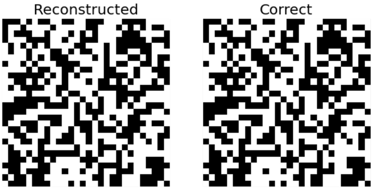


- After flipping 55% of pixels the accuracy is: 66.33 %

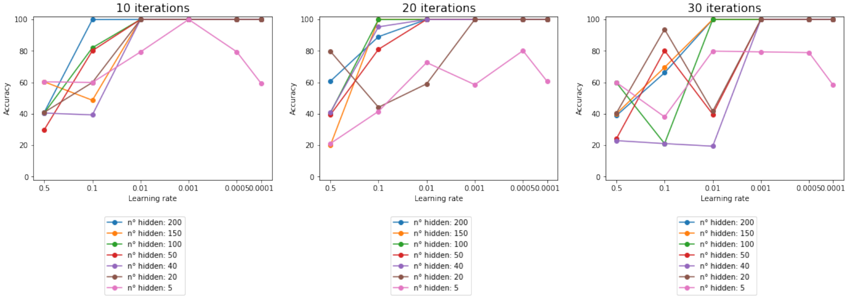


- $m(\sigma) = \frac{1}{N}\sum_i \sigma_i \xi_i$





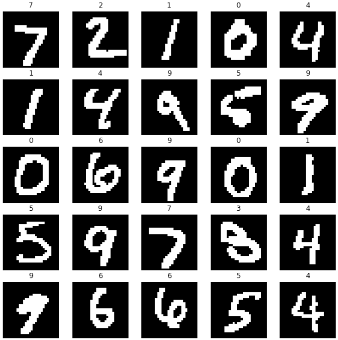
Plot accuracy

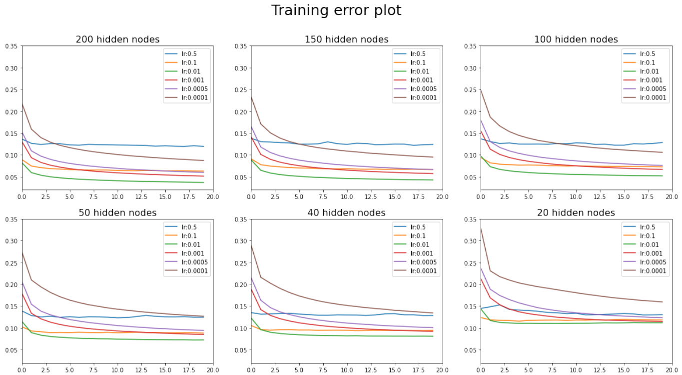


- Accuracy: 100%

MNIST dataset

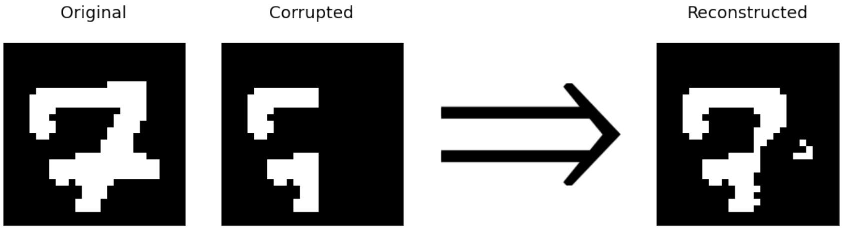
- 28x28 pixels images of digits



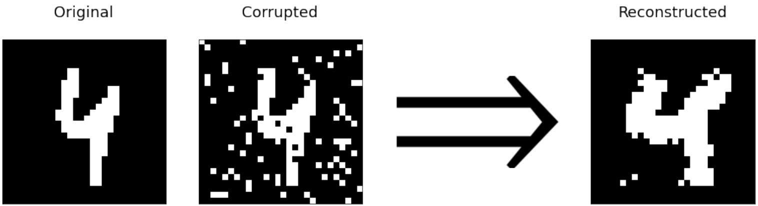


- n° hidden nodes: 200
- learning rate: 0.01

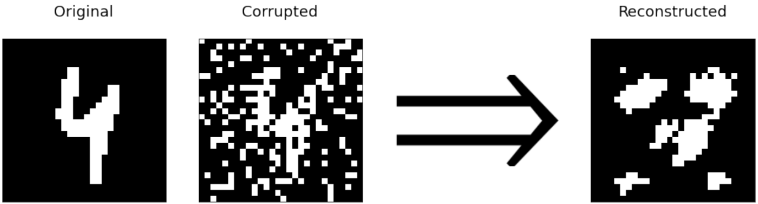
- only 14 columns over 28



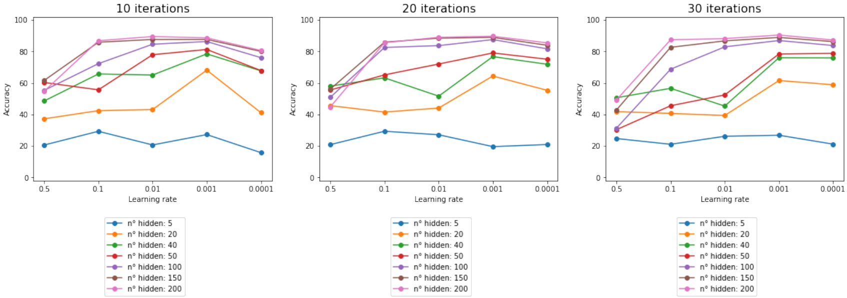
- After flipping 10 % of pixels:



- After flipping 20 % of pixels:



Plot accuracy



- Accuracy: 88.79%

Table of contents

1 Introduction

2 Implementation

3 Results

4 Plans for further work

- Try to put connections between hidden nodes to get better result
- Optimizing algorithms for best performance
- Testing on anonther datase, i.e. CIFAR