SAPIENZA
UNIVERSITÀ DI ROMA

Restricted Boltzmann machines for pattern recognition

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Matematica Applicata

Candidate
Andrea Mantuano
ID number 1739874

Academic Year 2021/2022

Contents

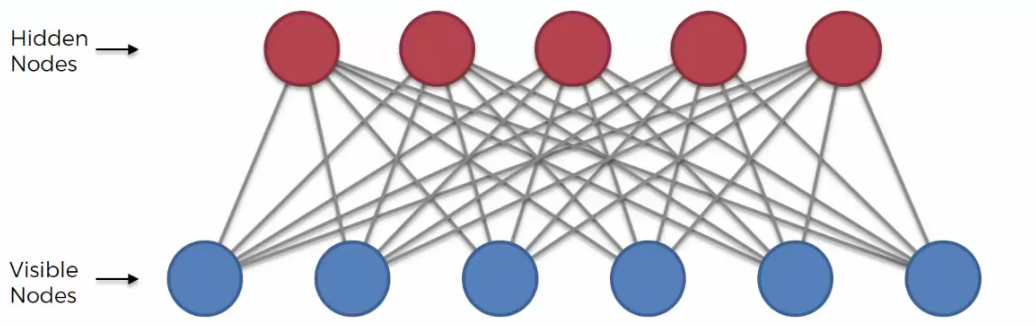
Chapter 1

Restricted Boltzmann Machines

1.1 Introduction

In this paper we will study a specific type of neural network called **Boltzmann machines**, a type of network popularized by Geoffrey Hinton in 1985. A Boltzmann machine is a model, inclusive of certain parameters, which can be used to capture important aspects of an unknown distribution (*target distribution*) starting from only a sample of it. Training a Boltzmann machine means to adapt its parameters in such a way that the probability distribution represented by it interpolates the training data as best as possible. Training is a process rather demanding from a computational point of view for this reason are often imposed restrictions on the network on which you are working. This brings us to the **Restricted Boltzmann machines**.

In Boltzmann machines there are two types of units (*neurons or nodes*), those **visible** and those **hidden**. The former constitute the components of an observation, for example as we will see later they can be associated with a single pixel of an image, and for this reason they are inserted by the user. The hidden units instead give us a model of the dependencies between the components of our observation, for this reason they are seen as detectors of the characteristics of the data.



As we can observe from the image, the restricted Boltzmann machines can be seen as complete bipartite graphs, where the first set of vertices is composed of the visible neurons,

the second set by the hidden neurons and the edges correspond to the connections synaptic connections.

Let $W = [w_{ij}]$ be the matrix of connections between units, where w_{ij} is the weight of the connection between the i-th visible node and the j-th hidden node, for simplicity we require that:

- $w_{ij} = w_{ji}$, i.e., that W is symmetric
- $w_{ij} = 0 \quad \forall i$, i.e., there are no self-interactions.

Moreover to every layer we associate a bias, we call $\mathbf{v}_0 = [v_0^i]$ the vector of the bias of the visible nodes and $\mathbf{h}_0 = [h_0^i]$ the vector of the biases of the hidden nodes.

Another key feature of the RBMs, is that they are models based on the energy of the joint configuration of the visible and hidden nodes. In order to capture the dependencies we associates a scalar energy scalar energy to each configuration of the variables, which serves as a measure of compatibility. A high energy means a bad compatibility. A model based on the energy always tries to reduce to minimize an energy function,this function for RBM is defined as:

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

The network then associates a probability with each possible pair (v,h) using this energy function:

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

where $Z = \sum_{v,h} e^{-E(v,h)}$ is the normalization function (*in physics this distribution is known as the Boltzmann distribution which gives the probability that a particle can be observed in the state with the energy E*). In addition the network assigns a probability to each visible vector v, which is obtained by summing over all the hidden state vectors:

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

The states of the two layers can take values in $\{1, 0\}$ which correspond to activated and non-activated states, respectively. The probability that a unit of the hidden layer is active knowing the states of the input layer (and vice versa) are:

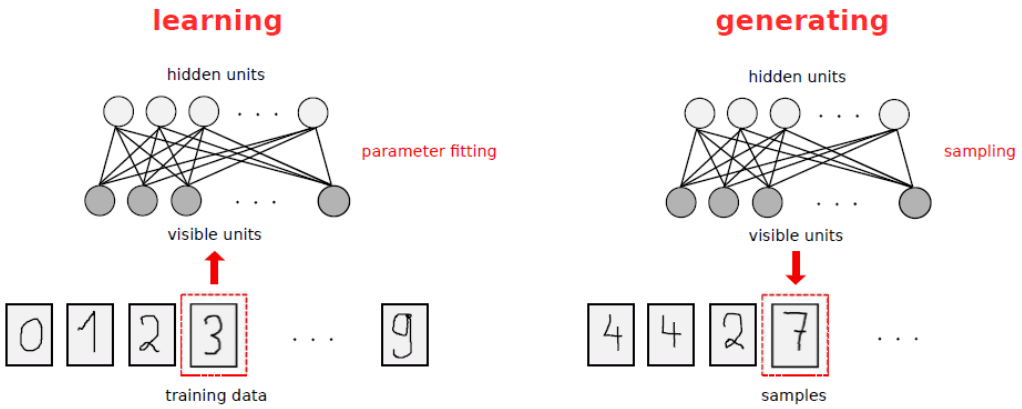
- $\mathbb{P}(h_i = 1 \,|\, v) = \sigma(v \cdot W + h_0^i)$
- $\mathbb{P}(v_i = 1 \,|\, h) = \sigma(h \cdot W^T + v_0^i)$

where $\sigma(x) = \frac{1}{1+e^{-x}}$ is the logistic function.

1.2 Generative model

Performing a well successful training, a RBM provides a very good representation of the distribution that subtends the data of training, in this way our network can be seen as a

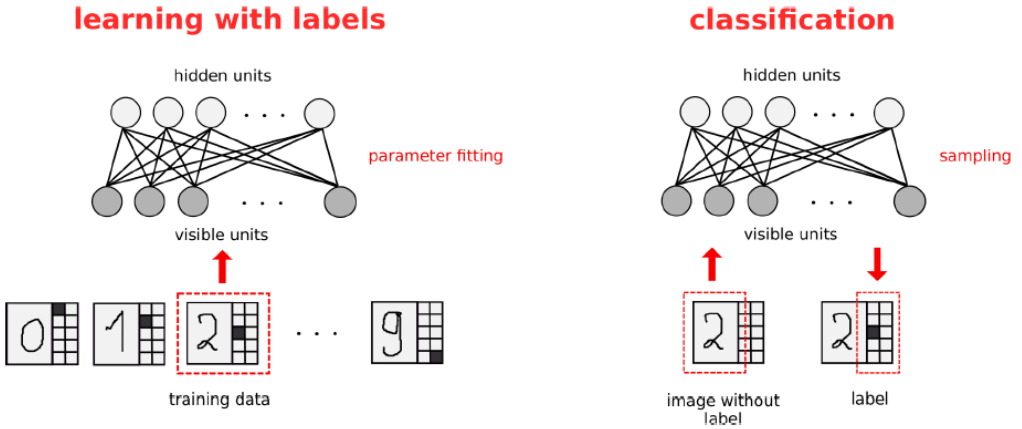
generative model that allows us to sample new data from the learned distribution. Having a generative model makes possible some useful applications, for example we can think to integrate some visible units corresponding to a partial observation partial observation (i.e. a corrupted image) and subsequently one can try to complete the observation.



The task of this particular RBM model is to learn a distribution $q(x)$, namely we want to adjust the parameters in order to to obtain $p_{W,\theta}(v) \sim q(x)$.

1.3 Classification model

Due to their being generative models, RBMs can be used as **classifiers**. Consider an application of this type: the restricted Boltzmann machine is trained to learn the joint probability distribution $q(x, y)$ of the input data (explanatory variables) and the corresponding labels, both represented by the visible units of the RBM. Next, 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.



1.4 The learning process

Our goal now is to adjust the parameters, weights and bias, so that the represented distribution is the best possible approximation of the target distribution. How can we measure the

distance between the two distributions? The approach we will use takes advantage of the **Kullback-Leibler** distance, so let's therefore try to minimize the function:

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

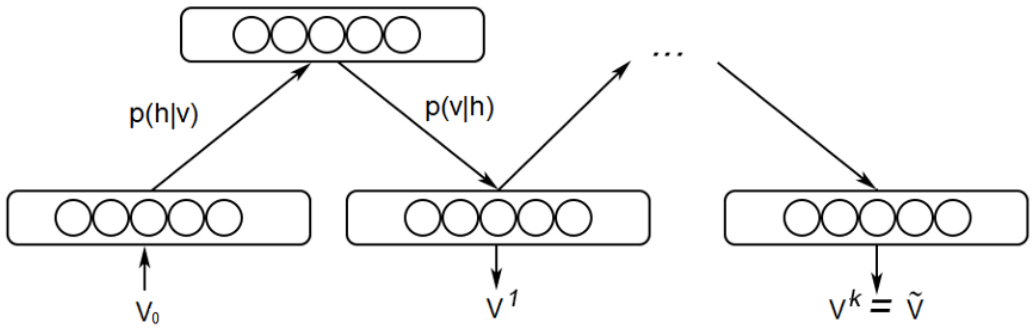
Putting it all together we get a formula for the gradient of the

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

from which we can derive the learning rule of the training algorithm. We observe that the first term to be calculated is immediate while the second is much more complicated, for this we will use a method statistical mechanics to obtain it.

1.5 Contrastive divergence

Monte Carlo methods based on Markov chains are algorithms used for sampling a probability distribution, based on the construction of Markov chains with the desired distribution as the stationary distribution. In fact after having simulated a great number of steps of the chain you can use the values extracted like sample of the wished distribution.



So approximating the second term of the gradient would require us to go through a Markov chain until the stationary distribution is reached (*where I reach the minimum energy and therefore the equilibrium*), but this requires a great effort computationally speaking. To overcome this problem we will use the Gibbs sampling which is always a Monte Carlo method based on Markov chains, but which is exploited in particular to obtain a sequence of observations that are approximated by a specific distribution when direct sampling is difficult.

The Gibbs chain is initialized with an example of the training set v_0 and after k-steps the sample v_k is obtained. Each step consists of two part:

1. sampling of $h(t)$ by exploiting the known distribution $p(h|v(t))$
2. sampling of $v(t+1)$ using the distribution just calculated $p(h|v(t))$.

the question that might arise is how many steps we need to take before to obtain a good sample, G. Hinton has shown that even a single step is sufficient to obtain a good approximation.

We are now able to calculate all the terms of the learning rule, so we can also update the parameters, the algorithm that takes care of this is called **Contrastive divergence**.

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

Chapter 2

Implementation

Inside the function *__init__* all the parameters that need to be optimized are initialized. In particular the **learning rate** and **the number of iterations** of the training. We observe that the **number of visible nodes** and the **number of labels**, are prefixed a priori when the machine is created, since we will only be able to work with images with 28x28 pixels and the number of patterns to be recognized is fixed in advance. In addition, it is important to point out that the weights matrix in the function was initialized using a *normal distribution* with mean 0 and variance 1, while the bias vectors of the visible and hidden nodes, respectively hidden nodes, $\mathbf{v}_0$ and $\mathbf{h}_0$, respectively, are initially equal to the null vector.

Finally, among the basis functions that the model must necessarily have, we have *__train* that allows us to train our network by exploiting the Contrastive Divergence, an algorithm that we will see in detail later.

```
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)
```

Let's focus on the algorithmic part that operates the sampling that will be used in the Contrastive Divergence function.

- *_mean_hiddens*: this is the function that calculates the probability of activation of the *i-th hidden neuron* $\mathbb{P}(h_i = 1 \,|\, v) = \sigma(v \cdot W + h_0^i)$
- *_sample_hiddens*: is the function that after calculating the probability of activation of hidden neurons, performs the sample taking as reference a uniform distribution on the interval $[0, 1]$
- *_sample_visibles*: is the function that, after having calculated the activation probability of visible neurons, $\mathbb{P}(v_i = 1 \,|\, h) = \sigma(h \cdot W^T + v_0^i)$, samples them by taking reference to a uniform distribution over the interval $[0, 1]$.

Finally, since we are working with states $v, h \in \{0, 1\}$, the function *_sample* randomly extracts a number from $[0, 1]$, compares it with the number calculated by the previous compares it with the one calculated by the previous functions and assigns to the node in question the value 1 or 0 if the neuron is activated or not.

```
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)
```

As a last function we analyze the one dedicated to the optimization of the parameters, the

Contrastive Divergence. As seen earlier in the introductory part, this algorithm consists primarily of two phases: the first, called **positive phase**, in which is calculated the initial phase of the Gibbs step, this is $\mathbb{E}_{s_{(0)}}(s_i s_j)$ and subsequently the second step, called the **negative phase**, in which is calculated the much more difficult $\mathbb{E}_{j,\theta}(s_i s_j)$. Let us analyze the code in detail.

Let's look at what happens in the innermost for loop, initially we divide the dataset in batches that at each iteration will take the role of the vector v_0, after which we are going to calculate in order, the expected value of the distribution of hidden nodes conditional to the observation of the values of our batch (*positive_hidden*), the positive correlation between the states (*positive_associations*) and subsequently the state of the hidden neurons is calculated according to the dynamics previously illustrated. With these steps the positive phase can be considered concluded.

We execute the first step of the Gibbs simulation and calculate v_1 (*negative_visible*), which will be the starting point for the negative step. Then, in the same way as the previous step, we calculate the expected value of the distribution of hidden nodes conditioned upon observation of the values of the vector v_1 (*negative_hidden*) and finally the negative correlation between the states (*negative_associations*).

Now that both phases, positive and negative, are concluded we can update the neural network:

- $\Delta W = \Delta W + \epsilon\,(positive\_associations - negative\_associations)$
- $\Delta h_0 = \Delta h_0 + \epsilon\,(positive\_hidden - negative\_hidden)$
- $\Delta v_0 = \Delta v_0 + \epsilon\,(v_0 - v_1)$

where ϵ is the learning rate.

```
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
```

Chapter 3

Results

Now that we have seen how to create a Boltzmann machine, let's put it to the test. The work has been divided into two parts, the first in which the neural network will be tested with Rademachers variables, that are random vectors such that $v_i \in 0, 1 \quad \forall i$ and subsequently with a more structured dataset such as MNIST.

3.1 Rademachers variable

To proceed with the analysis I decided to work on 5 random patterns whose individual pixels have been extracted from a distribution of Bernoulli distribution with parameter $p = 0.5$, to each archetype I have then associated a label.

```
from utils_function import *

n_label = 5
plt.figure(figsize=(10, 10))
archetype = generateArchetype(n_label)

for i in range(len(archetype)):
    ax = plt.subplot(1, len(archetype), i+1)
    plt.imshow(archetype[i][0].reshape((28, 28)), cmap='gray')
    plt.title(archetype[i][1])
    plt.axis('off')
```

Once the archetypes were generated it was necessary to create the actual proper dataset.

Using the following function I created 2000 copies of each archetype flipping at each iteration the 5% of the pixels, in order to obtain the dataset to work on.

```
def myDataset(a):
    copies = []
    label = []
    for i in range(len(a)):
        for _ in range(2000):
            copies.append(flipped(5,a[i][0]))
            label.append(a[i][1])
    return pd.DataFrame({'Copies':copies, 'Label':label})

df = myDataset(archetype)
data = df.sample(frac=1).reset_index(drop=True)
```

Subsequently I have operated a subdivision of the data, the 70% is used as a training set while the remaining 30% as a test set.

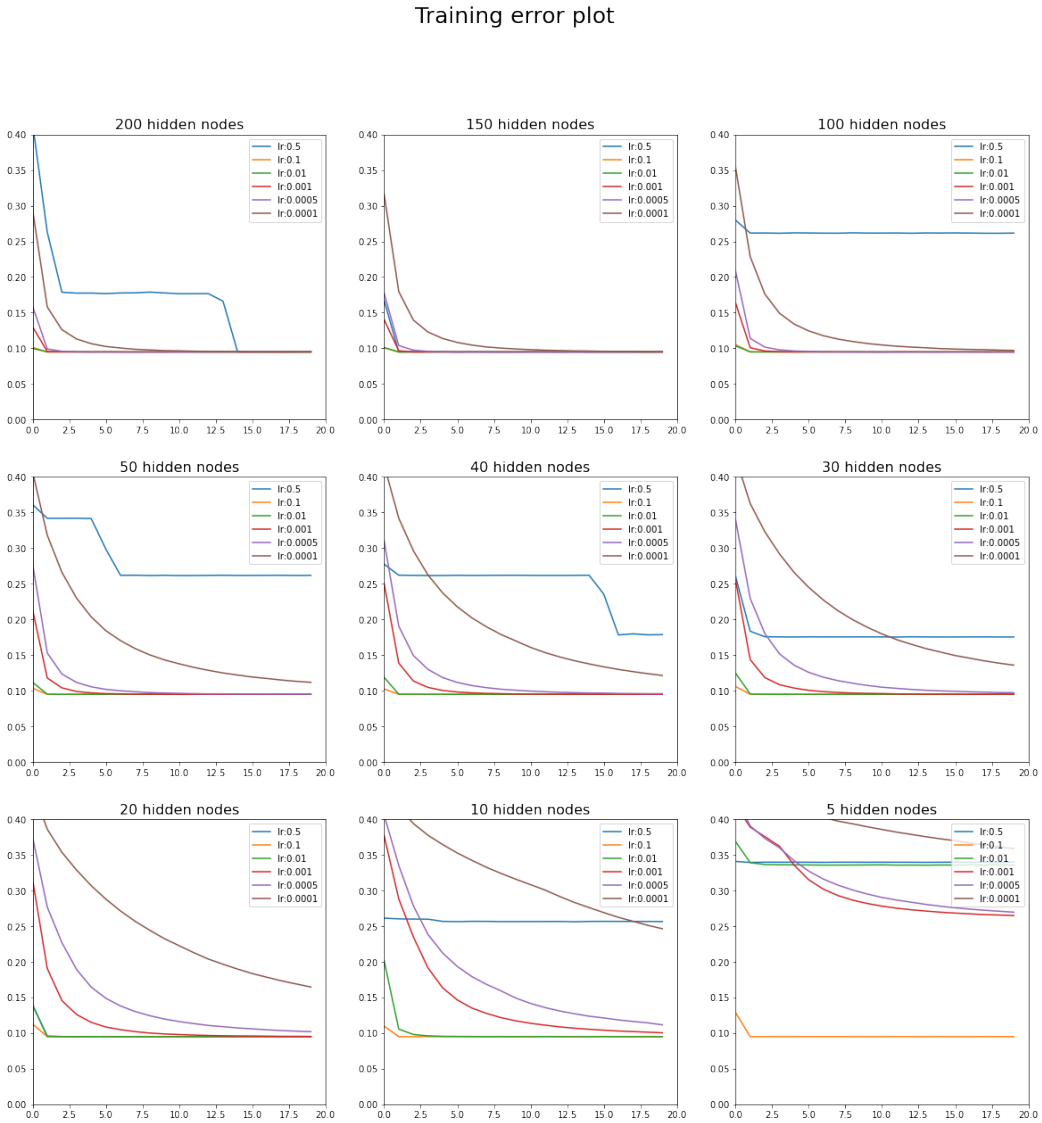
```
traindata = data.Copies[:int(0.7*len(df))]
testdata = data.Copies[int(0.7*len(df)):]

trainlabel = data.Label[:int(0.7*len(df))].tolist()
testlabel = data.Label[int(0.7*len(df)):].tolist()

traindata = np.array([traindata[i] for i in range(len(traindata))])
testdata = np.array([testdata[int(0.7*len(df))+i] for i in range(len(testdata))])

learning_rate = (0.5, 0.1, 0.01, 0.001, 0.0005, 0.0001)
n_hidden = (200, 150, 100, 50, 40, 30, 20, 10, 5)
```

Once obtained the training set on which to train our neural network, we proceed with the creation of the model and contextually we see the course of the error committed to every iteration of the learning, to the variation of the varying the parameters.

The error displayed indicates the MSE between the vector of visible nodes and the one computed with the Gibbs sampling.

```
for h in n_hidden:
    for lr in learning_rate:
        rbm = RBM(n_visible = 28*28, n_hidden = h, n_iterations = 20, learning_rate = lr)
        rbm._train(traindata, output = True)
        rbm._save(label = "Rad_")
```

```
# Training error plot
training_error_fixed_h(20, n_hidden, learning_rate, label = "Rad_")
```

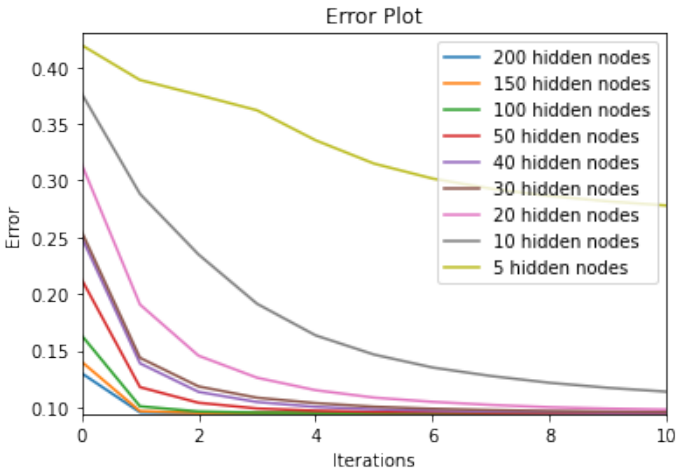


From these graphs it is evident as the network responds well to the process of training, in fact it is seen as the MSE iteration after iteration tends to decrease. In particular, by fixing the number of neurons and changing only the learning parameter, the error decreases as it decreases.

However, we must be careful that too small values of the learning parameter could not make converge the process of the gradient descent or make the convergence extremely slow.

Having chosen 0.001 as the optimal value for the learning rate, we proceed with the tuning of the number of hidden nodes.

```
# Training error plot fixed lr
best_lr = 0.001
training_error_fixed_lr(best_lr, n_hidden, label="Rad_")
```



Analyzing the graph we can state that the best RBM that we can train on this dataset has the following parameters:

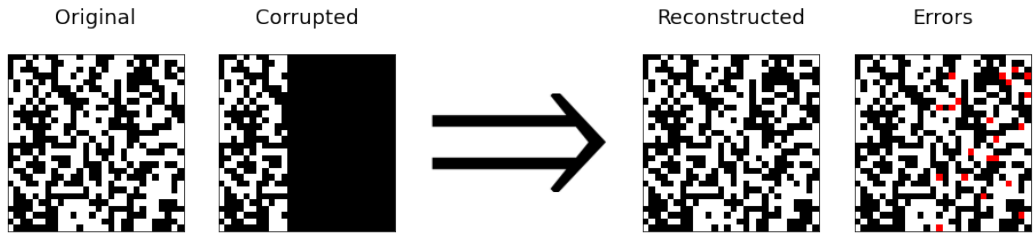
- learning rate: 0.001
- n° hidden nodes: 100

```
best_h = 100
bestRBM = RBM(n_visible = 28*28, n_hidden = best_h, n_iterations = 100, learning_rate =␣
 ↪best_lr)
bestRBM._train(traindata)
```

Now that the tuning of the parameters has been effected and therefore we have obtained the machine with better performance, let's put it to the test in the reconstruct a corrupted pattern.

```
# Pick another random image and set some parts of the image to zero.
im_ind = 800
rest = 11
X_missing = testdata[im_ind].copy().reshape(28,28)
X_missing[:,rest:] = 0

# Image Reconstruction
rec,_,_ = reconstruction(X_missing, bestRBM , corrupted = rest)
# Apply a threshold and complete the image
X_recon = np.where(rec > 0.6*np.max(rec), 1, 0)
X_complete = X_missing.copy()
X_complete[:,rest:] = X_recon

# Highlight the differences
diff = differences(testdata[im_ind],X_complete, tot = 28*(28-rest))
# Plot the figures
plotReconstructed(testdata[im_ind], X_missing, X_complete, diff)
```

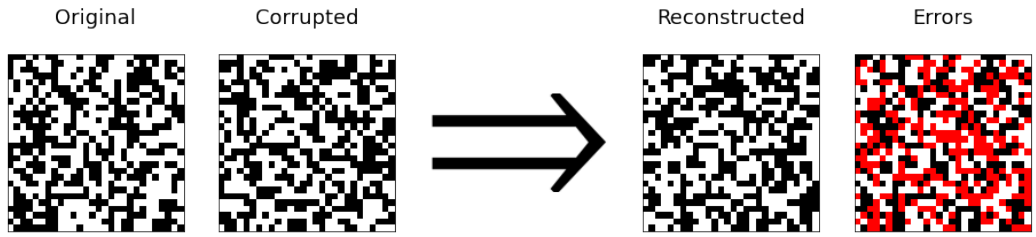
```
Accuracy of:  95.59 %
```



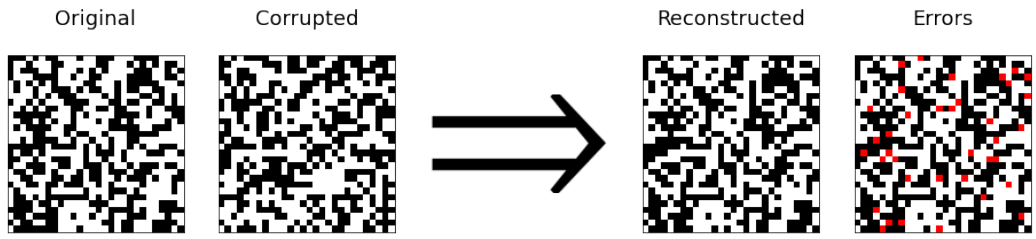
Always using the previously trained machine, let's test it in recreating a pattern after the pixels of an element of the test set have been flipped.

```
# Pick another random image and set some parts of the image to zero.
im_ind = 800

for perc_flip in (55,40):

    X_missing = flipped(perc_flip,testdata[im_ind].copy()).reshape(28,28)
    # Image Reconstruction
    rec,m,i = reconstruction(X_missing, bestRBM, archetype)
    # Apply a threshold and complete the image
    X_recon = np.where(rec > 0.5*np.max(rec), 1, 0)
    # Highlight the differences
    diff = differences(testdata[im_ind],X_recon,perc_flip)
    # Plot the figures
    plotReconstructed(testdata[im_ind], X_missing, X_recon, diff)
```

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

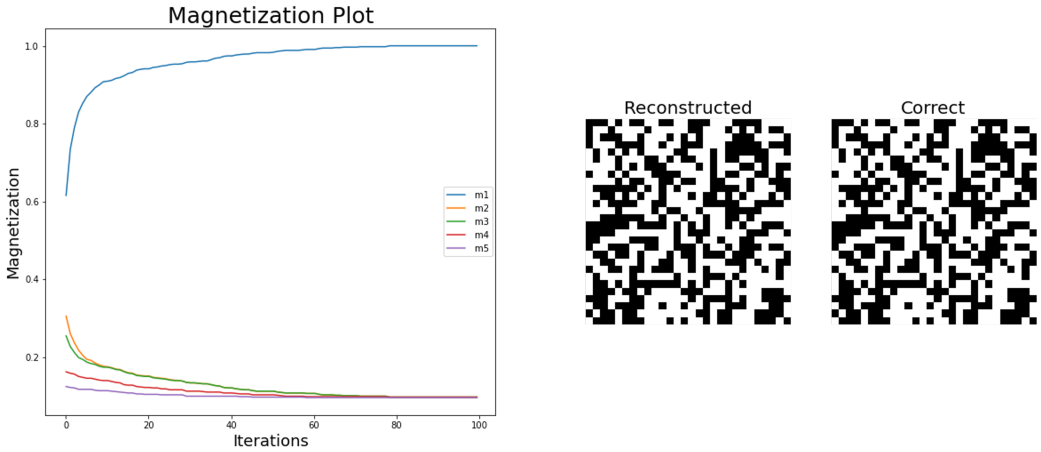


```
After flipping 40 % of pixels the accuracy is:  95.28 %
```



Following these two reconstructions we can affirm that the neural network performs very well its job so much to be able to reconstruct the right pattern in spite of having been flipped more than one third of pixels. Exceeded this threshold we see that it is no longer able and struggles to recreate the right archetype.

```
plot_m(m,i, archetype, X_recon)
```



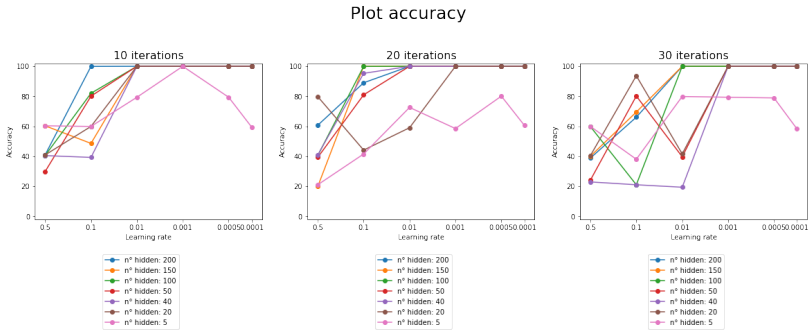
In this graph we can observe the trend of magnetization for each archetype. We note that not being completely orthogonal the magnetization with respect to the wrong pattern does not tend to 0, nevertheless the magnetization with higher value is the one related to the right pattern recognized.

3.1.1 Classification

We also see the behavior of our neural network in classifying patterns.

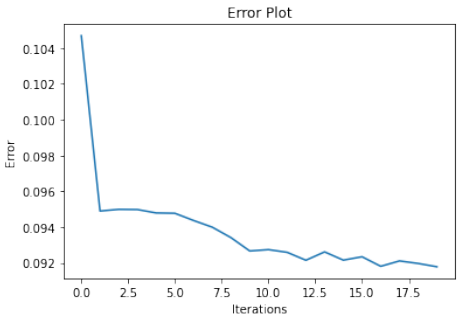
```
for n in [10,20,30]:
    for h in n_hidden:
        acc = []
        for lr in learning_rate:
            rbm = RBM(n_visible = 28*28, n_hidden = h, n_iterations = n, learning_rate =␣
 ↪lr, classifier = True, n_label = 5)
            rbm._train(traindata, trainlabel, classifier = True, output = True)
            rbm._save(classifier = True, label = str(n)+"_Rad_")
```

```
plotAccuracy(testdata, testlabel, learning_rate,n_hidden,n_it)
```




```
rbm = RBM(n_visible = 28*28, n_hidden = 100, n_iterations = 30, learning_rate = 0.01,␣
 ↪classifier = True, n_label = 5)
rbm._train(traindata, trainlabel, classifier = True)
```

```
# Training error plot
plt.plot(range(len(rbm.training_errors)), rbm.training_errors, label = "Training Error")

plt.title("Error Plot")

plt.ylabel('Error')
plt.xlabel('Iterations')

plt.show()
```



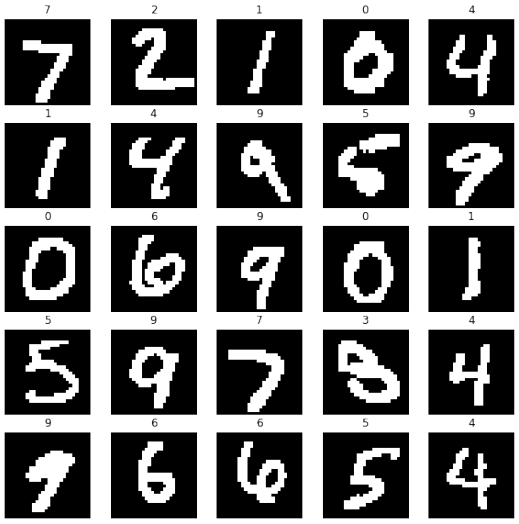
We measure the accuracy of the just trained machine by comparing the labels predicted by the network with those of the test set and then we print out the results.

```
acc = 0
k = rbm._predict(testdata)
for i in range(len(testlabel)):
    if k[i] == testlabel[i]:
        acc += 1

print("Accuracy: ",round(acc/len(testlabel)*100,2),"%")
```

```
Accuracy:  100.0 %
```

3.2 MNIST dataset

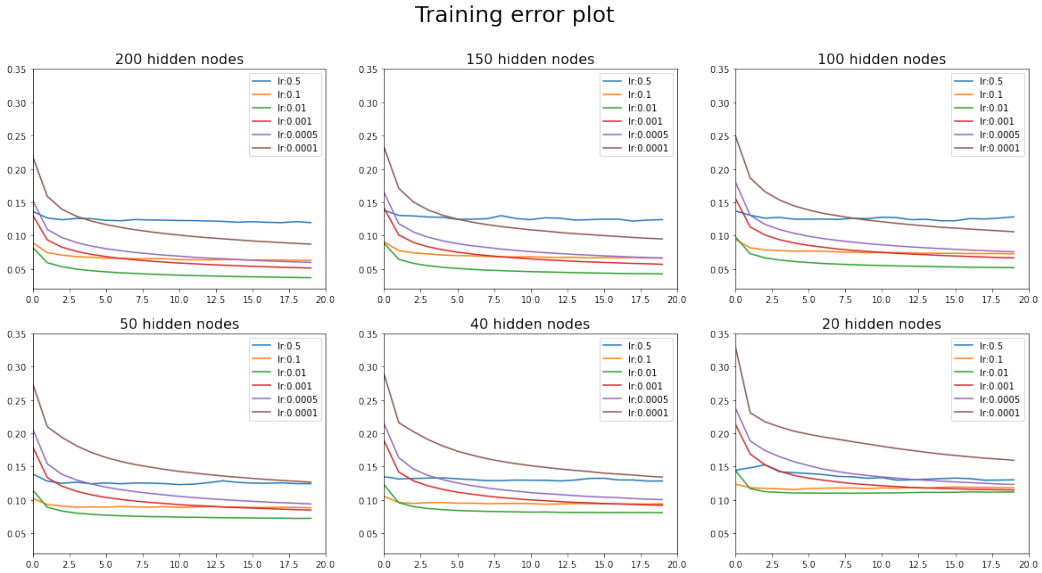
Having successfully concluded the analysis carried out on the Rademacher variables, we try the same approach with a more articulated dataset such as the MNIST, a database containing 70 thousand images of 28x28 pixels of handwritten figures.



```
train_data = pd.read_csv('C:/Users/CASA-PC/Desktop/ANDREA/Universita/Lezioni/Primo␣
 ↪semestre2/ADM/Homework/RBM/data/mnist_test.csv', delimiter = ',')
test_data = pd.read_csv('C:/Users/CASA-PC/Desktop/ANDREA/Universita/Lezioni/Primo␣
 ↪semestre2/ADM/Homework/RBM/data/mnist_test.csv', delimiter = ',')

label = 10

traindata = np.array([grayToBlack(img,0.4) for img in np.asfarray(train_data)[:,1:]])
testdata = np.array([grayToBlack(img,0.4) for img in np.asfarray(test_data)[:,1:]])

trainlabel = np.array(train_data)[:,0]
testlabel = np.array(test_data)[:,0]
```

```
for h in n_hidden:
    for lr in learning_rate:
        rbm = RBM(n_visible = 28*28, n_hidden = h, n_iterations = 20, learning_rate = lr)
        rbm._train(traindata, output = True)
        rbm._save(label = "MNIST_")
```

```
# Training error plot
training_error_fixed_h(20,n_hidden,learning_rate,label="MNIST_")
```



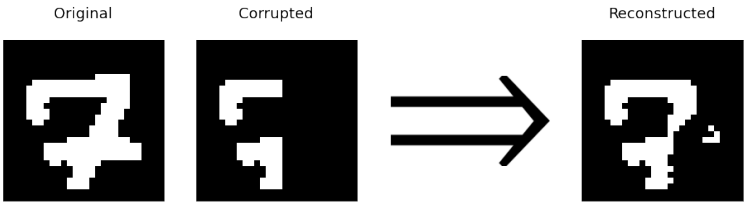
Analyzing the graph we can state that the best RBM that we can train on this dataset has the following parameters:

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

```
bestRBM_M = RBM(n_visible = 28*28, n_hidden = 200, n_iterations = 100, learning_rate = 0.
 ↪01)
bestRBM_M._train(traindata, output=False)
```

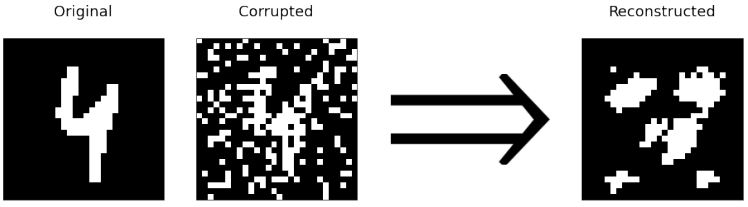
```
# Pick another random image and set some parts of the image to zero.
im_ind = 810
rest = 15
X_missing = testdata[im_ind].copy().reshape(28,28)
X_missing[:,rest:] = 0

# Image Reconstruction
rec = reconstruction(X_missing, bestRBM_M , corrupted = rest)
# Apply a threshold and complete the image
X_recon = np.where(rec > 0.6*np.max(rec), 1, 0)
X_complete = X_missing.copy()
X_complete[:,rest:] = X_recon
```

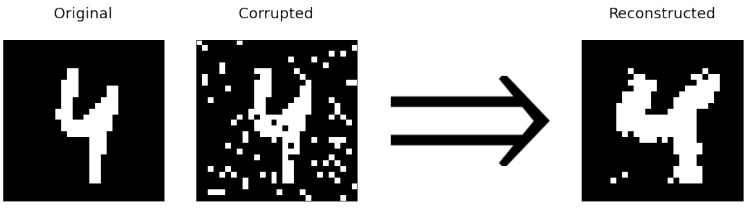
```
# Plot the figures
plotReconstructed(testdata[im_ind], X_missing, X_complete)
```



Let's see how again the newly trained machine manages to recreate the number 7 that was initially corrupted.

```
# Pick another random image and set some parts of the image to zero.
im_ind = 116

for perc_flip in (20,10):
    X_missing = flipped(perc_flip,testdata[im_ind].copy()).reshape(28,28)
    # Image Reconstruction
    rec = reconstruction(X_missing, bestRBM_M)
    # Apply a threshold and complete the image
    X_recon = np.where(rec > 0.5*np.max(rec), 1, 0)
    # Plot the result
    plotReconstructed(testdata[im_ind], X_missing, X_recon)
```





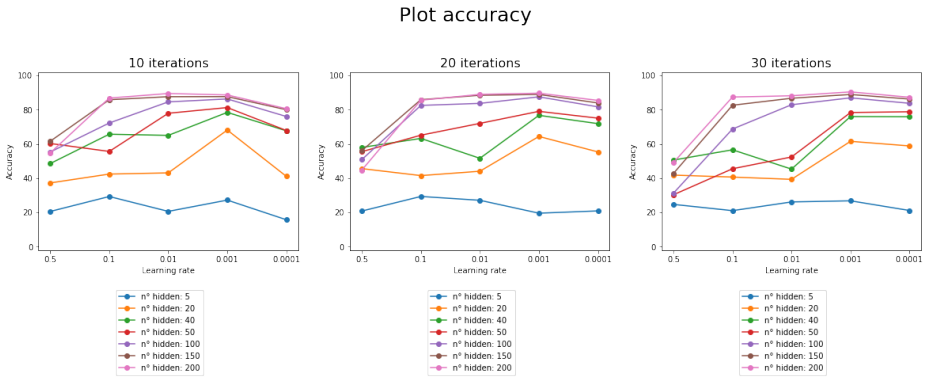
In these other two cases we observe that the network trained on the MNIST dataset performs worse than the one trained on Rademacher variables, in particular after having flipped only 20% of the pixels it is no longer able to recreate the original pattern.

3.2.1 Classification

Let's try using this machine as a classifier as well:

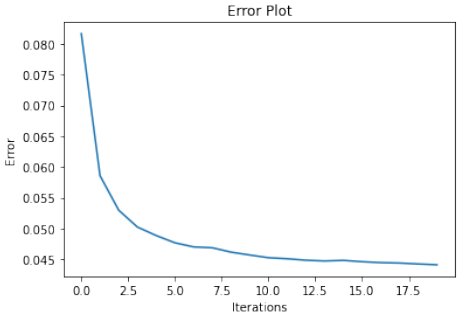
```
for n in n_it:
    for h in n_hidden:
        acc = []
        for lr in learning_rate:
            rbm = RBM(n_visible = 28*28, n_hidden = h, n_iterations = n, learning_rate =␣
 ↪lr, classifier = True, n_label = 10)
            rbm._train(traindata, trainlabel, classifier = True, output = True)
            rbm._save(classifier = True, label = str(n)+"_MNIST_")
```

```
plotAccuracy(testdata, testlabel,learning_rate,n_hidden,n_it,label = "_MNIST_")
```




```
rbm = RBM(n_visible = 28*28, n_hidden = 30, n_iterations = 100, learning_rate = 0.01,␣
 ↪classifier = True, n_label = label)
rbm._train(traindata, trainlabel, classifier = True)
```

```
# Training error plot
plt.plot(range(len(rbm.training_errors)), rbm.training_errors, label = "Training Error")

plt.title("Error Plot")

plt.ylabel('Error')
plt.xlabel('Iterations')

plt.show()
```




```
acc = 0
k = rbm._predict(testdata)
for i in range(len(testlabel)):
    if k[i] == testlabel[i]:
        acc += 1

print("Accuracy: ",round(acc/len(testlabel)*100,2),"%")
```

```
Accuracy:  88.79 %
```

Again, we observe that the accuracy of the MNIST machine is lower than that of the Rademacher machine. Most likely this is due to the nature of the data used in the training. In fact in the machine trained with Rademacher variables the correlation between different patterns is much lower than in the patterns of the MNIST dataset.

Chapter 4

Resources

4.1 Source Code

https://github.com/Mantuano-A/RBM

4.2 References

- E. Agliari, Dispense corso Modelli di Reti Neurali
- A. Fischer, C. Igel, An Introduction to Restricted Boltzmann Machines
- G. Hinton, A Practical Guide to Training Restricted Boltzmann Machines
- S. Cherla, S. N. Tran, T. Weyde and A. Garcez, Generalising the Discriminative Restricted Boltzmann Machine
- G. Hinton, T. Schmah, R. S. Zemel and S. L. Small, Generative versus discriminative training of RBMs for classification of fMRI images
- H. Larochelle, Y. Bengio, Classification using Discriminative Restricted Boltzmann Machines