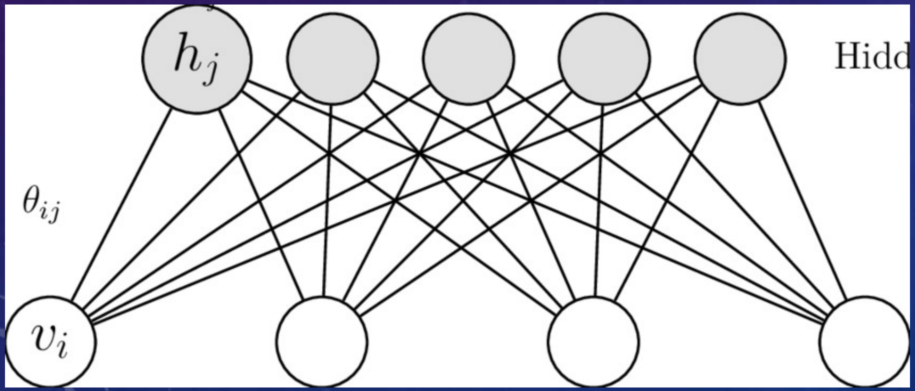
IMAGE PROCESSING ASSIGNMENTS - RESTRICTED BOLTZMANN MACHINE

IOMMI ANDREA 578212 - M.SC. IN ARTIFICIAL INTELLIGENCE

RESTRICTED BOLTZMANN MACHINE

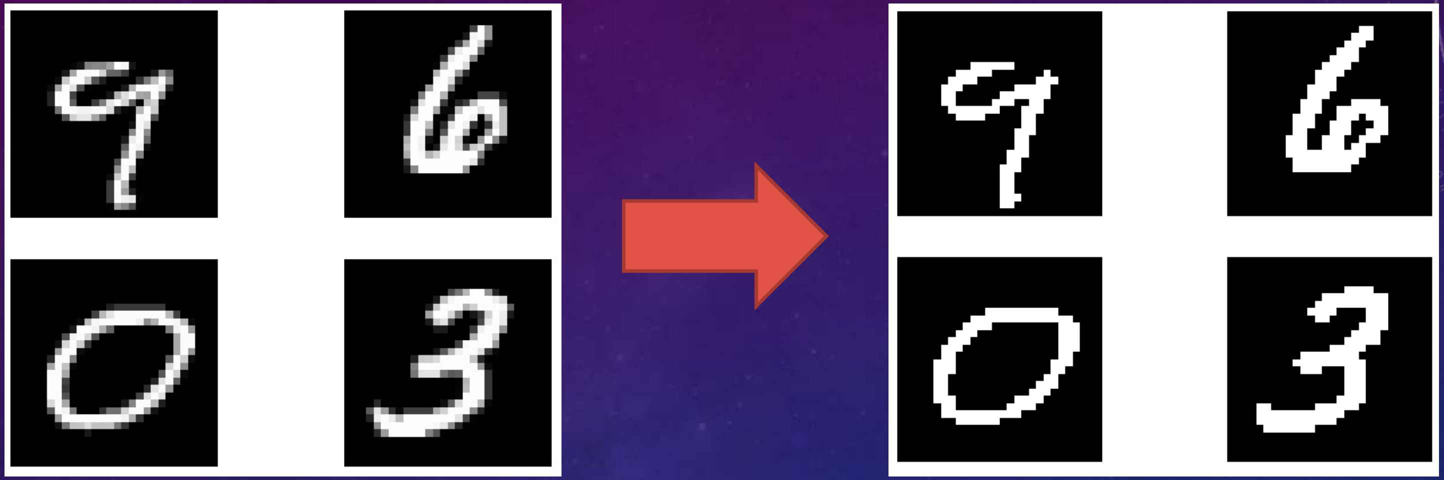
- The Restricted Boltzmann Machine (RBM) is a generative stochastic neural network that can learn a probability distribution over its set of inputs.
- A RBM is a particular version of general Boltzmann machines, with the particularity that their neurons must form a bipartite graph.
- There are two groups of units (referred to as the "visible" and "hidden" units respectively), It’s important that there are not connections between nodes within a group.



- We can see the “Visible units” as the input and the “hidden units” a probabilistic state.

Figure taken from:
https://www.researchgate.net/figure/An-example-restricted-Boltzmann-machine-RBM-consisting-of-two-layers-one-hidden-H_fig1_311430406

MNIST - HANDWRITTEN DIGITS



- The first phase that I executed is the preprocessing of dataset.
- I transformed the original input composed of an array (28x28 pixel) with value between 0 and 255 into a boolean matrix with only 0 and 1 values applying a threshold function.
- Thus the RBM receives to input vectors of 768 binary values (a flat version of threshold images).

PARAMETERS & TECHNIQUES

Techniques

- It was adopted a mini batch approach to update the weights and biases.
- Fixed learning rate.
- I used all (unlabeled) examples to train the RBM but in the Logistic regression, the "***Reconstructed dataset***" it was split in train and test set.
- "***Reconstructed dataset***" refers to a dataset with the same number of examples, but less feature (only 81) instead of the original dataset (784).

Parameters

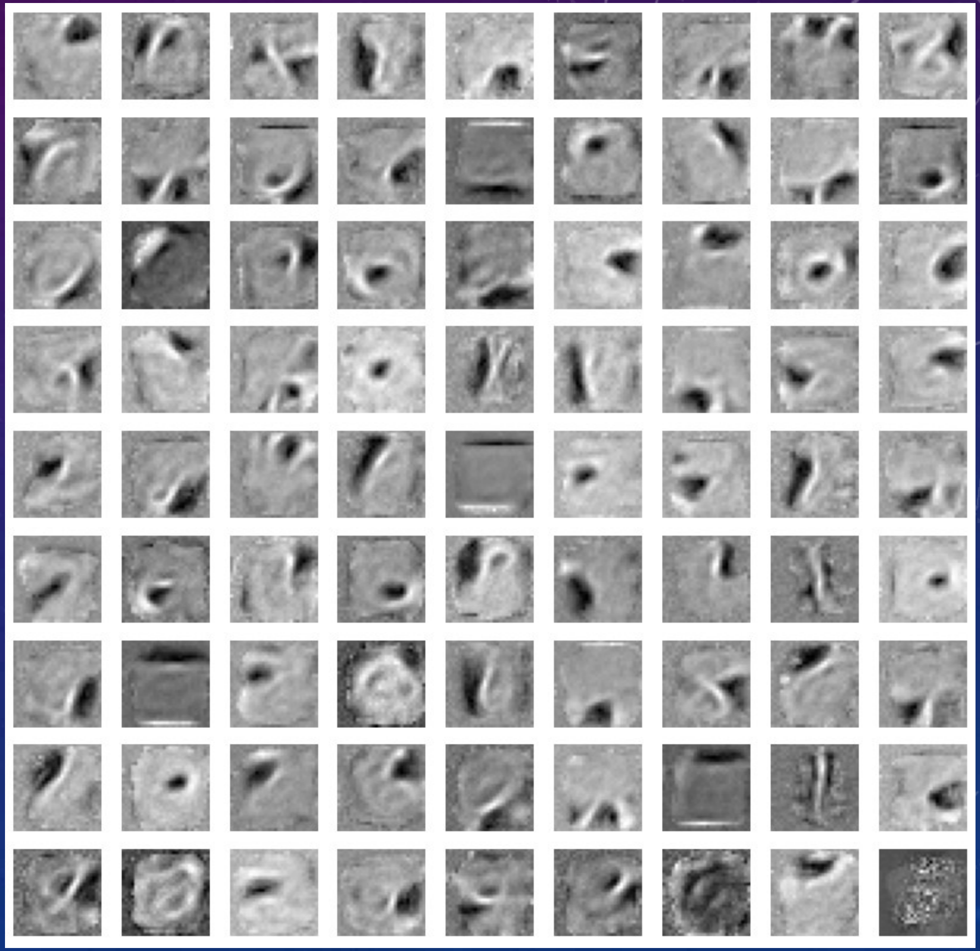
- **Number of epoch** : 50
- **Number of examples** : 60 000
- ***Visible node*** : 784
- **Hidden node** : 81
- **Learning rate** : 80%
- **Batch size**: 32
- **Activation function** : Sigmoid
- **Training set(example)** : 48 000
- **Test set(example)** : 12 000

Training and Test set refer to number of examples used in the Logistic Regression (final phase of the assignment).

References
[0] https://www.cs.toronto.edu/~hinton/absps/guideTR.pdf
[1] https://medium.com/machine-learning-researcher/boltzmann-machine-c2ce76d94da5

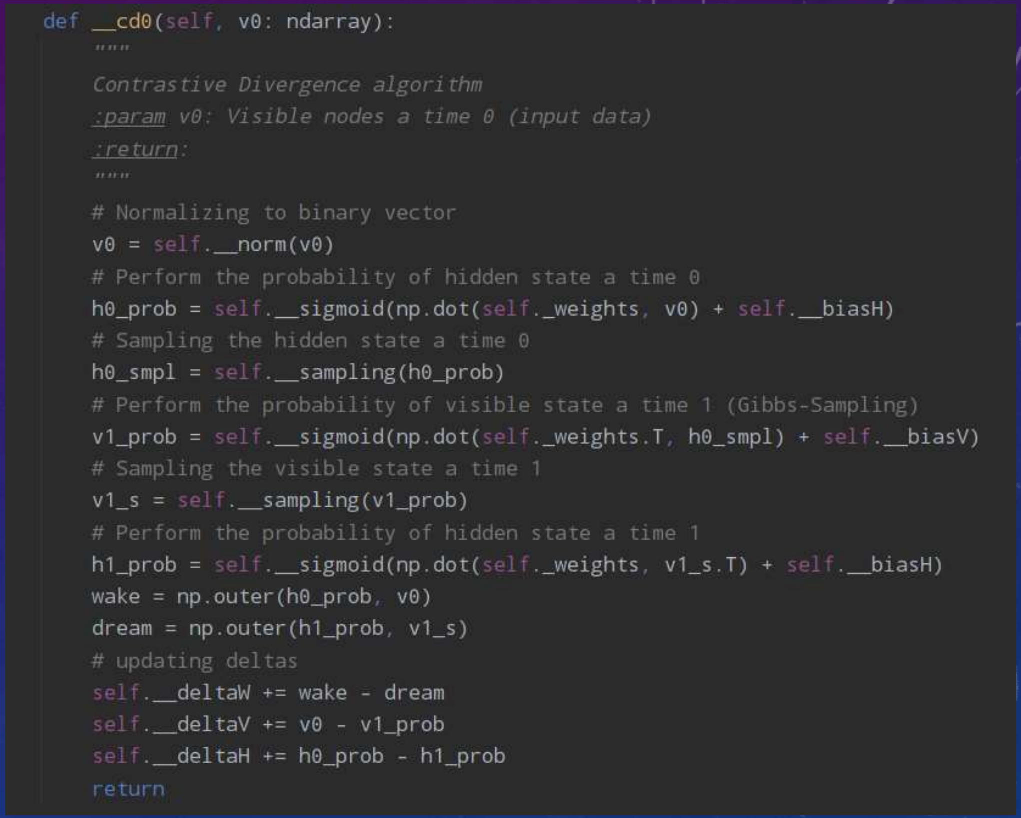
RESTRICTED BOLTZMANN MACHINE FILTERS

- It was used 81 hidden nodes (displayed in 9 x 9 grid).
- A square represents the weights learnt by CD0 for one hidden node.
- The filters are rescaled but the “darkness” of pixel identifies the intensity (relevance).



CONTRASTIVE DIVERGENCE ALGORITHM

- Gibbs-sampling is an approximate method, it used when the sampling is difficult.
- The gradient descent procedure exploits Gibbs-sampling approach to compute the weights update.
- In our case I used a CD-0, so just one iteration of Gibbs method, it’s enough to approximate well.
- Oss. “*Normalizing*” means applying a threshold function.

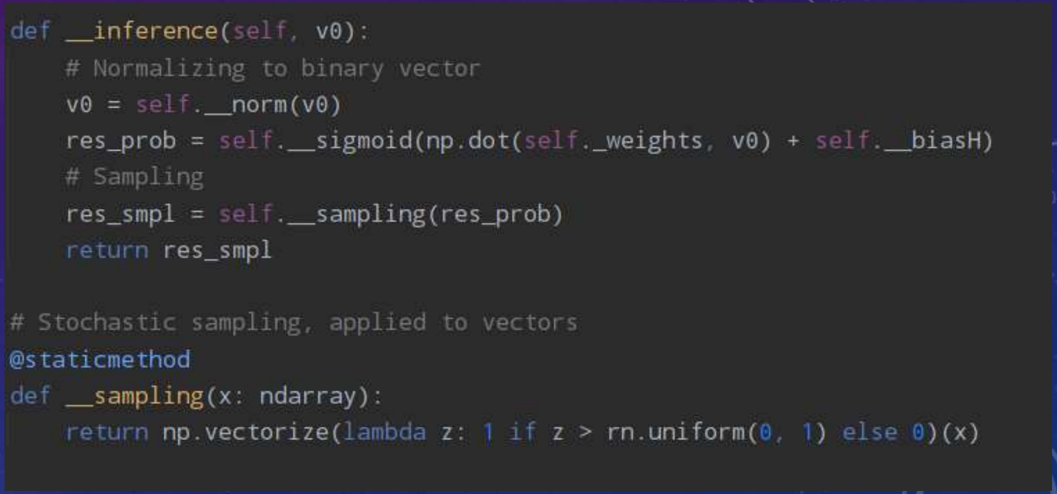


```python
def __cd0(self, v0: ndarray):
    """
    Contrastive Divergence algorithm
    :param v0: Visible nodes a time 0 (input data)
    :return:
    """
    # Normalizing to binary vector
    v0 = self.__norm(v0)
    # Perform the probability of hidden state a time 0
    h0_prob = self.__sigmoid(np.dot(self._weights, v0) + self.__biasH)
    # Sampling the hidden state a time 0
    h0_smpl = self.__sampling(h0_prob)
    # Perform the probability of visible state a time 1 (Gibbs-Sampling)
    v1_prob = self.__sigmoid(np.dot(self._weights.T, h0_smpl) + self.__biasV)
    # Sampling the visible state a time 1
    v1_s = self.__sampling(v1_prob)
    # Perform the probability of hidden state a time 1
    h1_prob = self.__sigmoid(np.dot(self._weights, v1_s.T) + self.__biasH)
    wake = np.outer(h0_prob, v0)
    dream = np.outer(h1_prob, v1_s)
    # updating deltas
    self.__deltaW += wake - dream
    self.__deltaV += v0 - v1_prob
    self.__deltaH += h0_prob - h1_prob
    return
```

***h0_smpl** refer to a sampled hidden probability vector*

INFERENCE

- The inference method gets “visible” nodes (input data) and return the "hidden“ state sampled (binary vector).
- The core of RBM is the ability to apply the **representation of learning**, indeed it has learnt by unlabelled examples how to transform the huge visible state in a smaller representation. (Feature reduction).

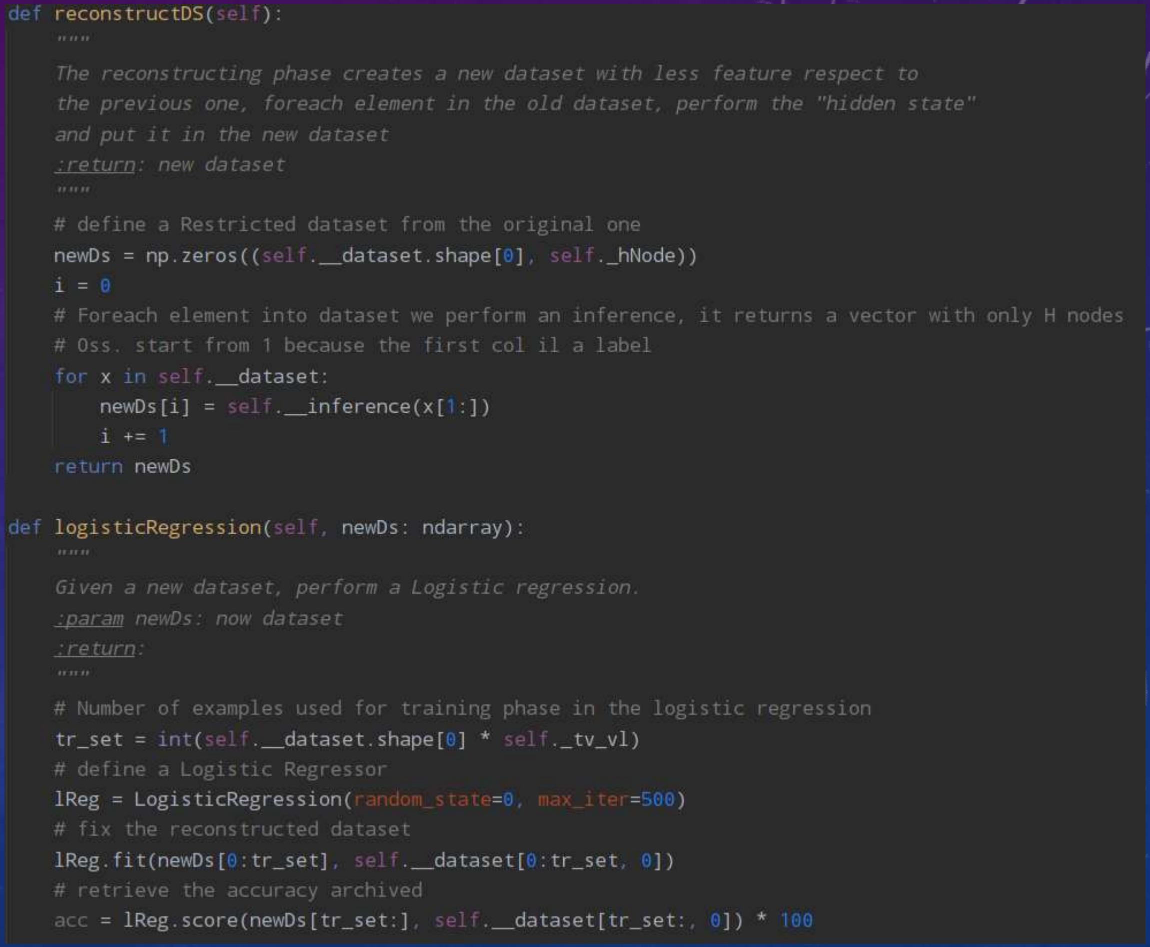


```python
def __inference(self, v0):
    # Normalizing to binary vector
    v0 = self.__norm(v0)
    res_prob = self.__sigmoid(np.dot(self._weights, v0) + self.__biasH)
    # Sampling
    res_smpl = self.__sampling(res_prob)
    return res_smpl

# Stochastic sampling, applied to vectors
@staticmethod
def __sampling(x: ndarray):
    return np.vectorize(lambda z: 1 if z > rn.uniform(0, 1) else 0)(x)
```

LOGISTIC REGRESSION

- After train, I used the RBM to build the "***Reconstructed dataset***"
- I used this new dataset in another (*supervisionated*) task.
- I chose a Logistic Regression from *scikit*, it takes: ***"hidden node" features*** as input and the corresponding ***label*** of handwritten digit as output.
- **It was archived a 92,35% of accuracy**



```
def reconstructDS(self):
    """
    The reconstructing phase creates a new dataset with less feature respect to
    the previous one, foreach element in the old dataset, perform the "hidden state"
    and put it in the new dataset
    :return: new dataset
    """
    # define a Restricted dataset from the original one
    newDs = np.zeros((self.__dataset.shape[0], self._hNode))
    i = 0
    # Foreach element into dataset we perform an inference, it returns a vector with only H nodes
    # Oss. start from 1 because the first col il a label
    for x in self.__dataset:
        newDs[i] = self.__inference(x[1:])
        i += 1
    return newDs

def logisticRegression(self, newDs: ndarray):
    """
    Given a new dataset, perform a Logistic regression.
    :param newDs: now dataset
    :return:
    """
    # Number of examples used for training phase in the logistic regression
    tr_set = int(self.__dataset.shape[0] * self._tv_vl)
    # define a Logistic Regressor
    lReg = LogisticRegression(random_state=0, max_iter=500)
    # fix the reconstructed dataset
    lReg.fit(newDs[0:tr_set], self.__dataset[0:tr_set, 0])
    # retrieve the accuracy archived
    acc = lReg.score(newDs[tr_set:], self.__dataset[tr_set:, 0]) * 100
```

PYTHON AND C ++

- The final training and reconstructing phases were not done in python, but in C++, I have written a (fast) version of the RBM implementation (even from scratch) to speed up the learning phase. Anyway, I'll attach all the materials to the presentation folder.

- I used C++ implementation to learn the weights and build the reconstructed dataset, then the program exports all data into C.S.V files witch are opened by python in order to plotting filter or execute the Logistic regression.

- I also deploy a python RBM implementation (previous snippet) much slower than C++ (obliviously ;))

https://github.com/jacons/R.-Boltzmann-machine-Py-Cpp

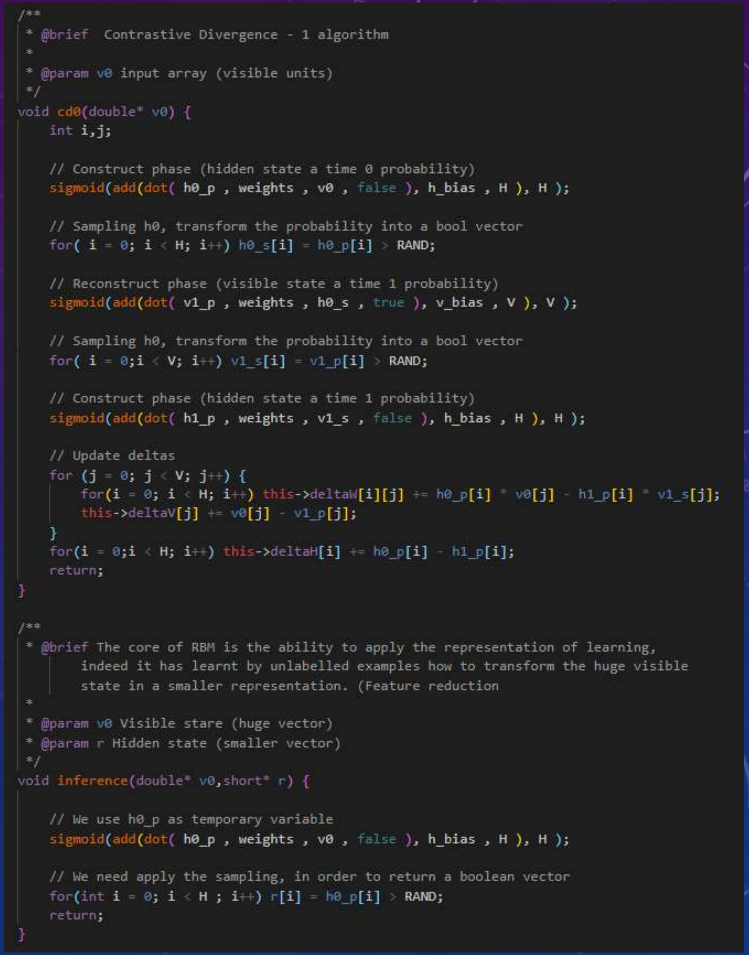
ENHANCE THE ANALYSIS & FUN THINGS

- There are several things we should do, maybe could be useful to improve the accuracy adding additional layer to RBM, or play with more complex technique such as learning rate decay or model selection and model assessment to pick the best RBM model.
- We could take into account to use another Machine learning model to solve the assignment, i.e. Auto-encoders.

Phase	Python	C++
Loading dataset	1868 ms	2242 ms
Learning	≈ 3h	707 s
Reconstructing	27 s	2918 ms

Logistic Regression (784 nodes)	Logistic Regression (81 nodes)
31 s	15 s

Both are made in python



```
/**
 * @brief  Contrastive Divergence - 1 algorithm
 *
 * @param v0 input array (visible units)
 */
void cd0(double* v0) {
    int i,j;

    // Construct phase (hidden state a time 0 probability)
    sigmoid(add(dot( h0_p , weights , v0 , false ), h_bias , H ), H );

    // Sampling h0, transform the probability into a bool vector
    for( i = 0; i < H; i++) h0_s[i] = h0_p[i] > RAND;

    // Reconstruct phase (visible state a time 1 probability)
    sigmoid(add(dot( v1_p , weights , h0_s , true ), v_bias , V ), V );

    // Sampling h0, transform the probability into a bool vector
    for( i = 0;i < V; i++) v1_s[i] = v1_p[i] > RAND;

    // Construct phase (hidden state a time 1 probability)
    sigmoid(add(dot( h1_p , weights , v1_s , false ), h_bias , H ), H );

    // Update deltas
    for (j = 0; j < V; j++) {
        for(i = 0; i < H; i++) this->deltaW[i][j] += h0_p[i] * v0[j] - h1_p[i] * v1_s[j];
        this->deltaV[j] += v0[j] - v1_p[j];
    }
    for(i = 0;i < H; i++) this->deltaH[i] += h0_p[i] - h1_p[i];
    return;
}

/**
 * @brief The core of RBM is the ability to apply the representation of learning,
 *      indeed it has learnt by unlabelled examples how to transform the huge visible
 *      state in a smaller representation. (Feature reduction
 *
 * @param v0 Visible stare (huge vector)
 * @param r Hidden state (smaller vector)
 */
void inference(double* v0,short* r) {

    // We use h0_p as temporary variable
    sigmoid(add(dot( h0_p , weights , v0 , false ), h_bias , H ), H );

    // We need apply the sampling, in order to return a boolean vector
    for(int i = 0; i < H ; i++) r[i] = h0_p[i] > RAND;
    return;
}
```

Contrastive divergence in C++