MNIST Autoencoding with Restrict Boltzmann Machine

Linyang He (15307130240)

December 23, 2018

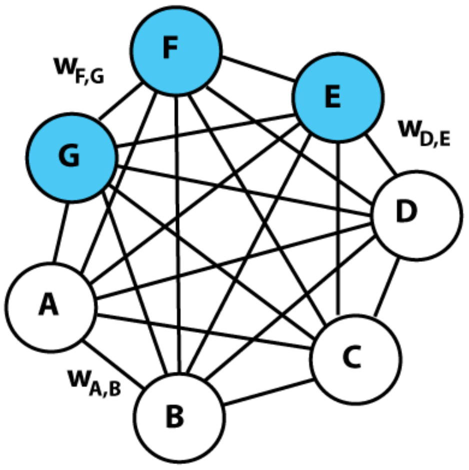
Abstract

In this project, we use Restricted Boltzamann Machine(RBM) to embedding some certian of information of the hand-wriiting pictures. In some way, . Contrastive divergence is the algorithm we use for parameters updaiting. Besides, we use Gipps sampling in the training to enhance the effienecy.

1 Background

1.1 Boltzmann Machine

A Boltzmann machine (also called stochastic Hopfield network with hidden units) is a type of stochastic recurrent neural network. A Boltzmann machine is a network of units with an "energy" defined for the overall network. Its units produce binary results. As you can see from the figure below, the nodes can be divided into two category: hidden nodes and visibible nodes.



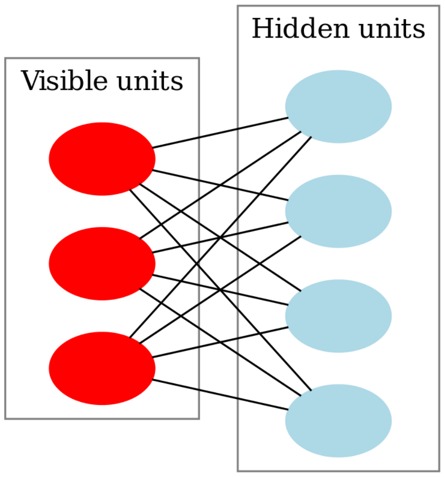
While the global energy, can be computed as:

$$E = -\left(\sum_{i<j} w_{ij}\, s_i\, s_j + \sum_i \theta_i\, s_i\right)$$

In this equation, $w_{ij}$is the weight between unit j and unit i. And s_i is the state value with respect to the node. Beside, $s_i \in \{0, 1\}$. 1 denotes that the node is activated, while 0 denotes the node closed. θ_i is the bias of the unit i.

1.2 Restricted Boltzmann Machine

If you turn the complete graph in general Boltzmann machine into a bipartite graph, you can get the so-called restricted Boltzamann Machine.



A restricted Boltzmann machine (RBM) is a generative stochastic artificial neural network that can learn a probability distribution over its set of inputs. RBMs were initially invented under the name Harmonium by Paul Smolensky in 1986 and rose to prominence after Geoffrey Hinton and collaborators invented fast learning algorithms for them in the mid-2000. RBM is essentially a tool for Unsupervised Learning because it can be used for dimensionality reduction (less hidden layers), learning Features (hidden layer output is a feature), self-encoder (automatic encoder) and deep belief network (multiple RBM stacked). Actually, in this project, what we do is using the RBM as an autoencoder considering that we don't use the labeling information at all.

In this situation, we can get the energy for the RBM as:

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

The denotion is the same as above, while a_i and b_i is the specific bias for visible layer nodes and hidden layer nodes. Besides, this energy function is analogous to that of a Hopfield network. As for the probability distributions over hidden and/or visible vectors, they can be defined in terms of the energy function

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

Here Z denotes a partition function. It is a normalizing constant to ensure the probability distribution sums to 1. In this way, the marginal probability of a visible (input) vector of booleans is the sum over all possible hidden layer configurations:

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

The individual activation probabilities are given by

$$P(h_j = 1|v) = \sigma\left(b_j + \sum_{i=1}^{m} w_{i,j} v_i\right)$$

and

$$P(v_i = 1|h) = \sigma\left(a_i + \sum_{j=1}^{n} w_{i,j} h_j\right)$$

where $\sigma\sigma$ denotes the logistic sigmoid.

1.3 Contrastive Divergence

For training the model, we use the contrastiva divergence algorithm to update the parameters. This is how the algorithm works:

1. Take a training sample v, compute the probabilities of the hidden units and sample a hidden activation vector h from this probability distribution.

2. Compute the outer product of v and h and call this the forward gradient.

3. From h, sample a reconstruction v' of the visible units, then resample the hidden activationsh' from this. (This is actually the Gibbs sampling step.)

4. Compute the outer product of v' and h' and call this the backward gradient.

5. Update the weight matrix.

$$W \leftarrow W + \alpha(\hat{\mathbf{v}}^{(n)}\mathbf{h}^{\mathrm{T}} - \mathbf{v}'\mathbf{h}'^{\mathrm{T}})$$

6. Update the bias vector.

$$\mathbf{a} \leftarrow \mathbf{a} + \alpha(\hat{\mathbf{v}}^{(n)} - \mathbf{v}') \ ;$$
$$\mathbf{b} \leftarrow \mathbf{b} + \alpha(\mathbf{h} - \mathbf{h}') \ ;$$

1.4 Gibbs Sampling

Gibbs sampling or a Gibbs sampler is a Markov chain Monte Carlo (MCMC) algorithm for obtaining a sequence of observations which are approximated from a specified multivariate probability distribution, when direct sampling is difficult. The algorithm is showing as follows.

Algorithm 1 Gibbs sampler

```
Initialize $x^{(0)} \sim q(x)$
for iteration $i = 1, 2, \ldots$ do
    $x_1^{(i)} \sim p(X_1 = x_1 | X_2 = x_2^{(i-1)}, X_3 = x_3^{(i-1)}, \ldots, X_D = x_D^{(i-1)})$
    $x_2^{(i)} \sim p(X_2 = x_2 | X_1 = x_1^{(i)}, X_3 = x_3^{(i-1)}, \ldots, X_D = x_D^{(i-1)})$
    $\vdots$
    $x_D^{(i)} \sim p(X_D = x_D | X_1 = x_1^{(i)}, X_2 = x_2^{(i)}, \ldots, X_D = x_{D-1}^{(i)})$
end for
```

1.5 Deep belief network

Deep belief network is a stack of RBM. Deep belief network (DBN) is a generative graphical model, or alternatively a class of deep neural network, composed of multiple layers of latent variables ("hidden units"), with connections between the layers but not between units within each layer. When trained on a set of examples without supervision, a DBN can learn to probabilistically reconstruct its inputs. The layers then act as feature detectors. After this learning step, a DBN can be further trained with supervision to perform classification. DBNs can be viewed as a composition of simple, unsupervised networks such as restricted Boltzmann machines (RBMs) or autoencoders, where each sub-network's hidden layer serves as the visible layer for the next.

Since the parameters is very hard to learn in DBN, we might consider DBN to be trained greedily, one layer at a time, led to one of the first effective deep learning algorithms.

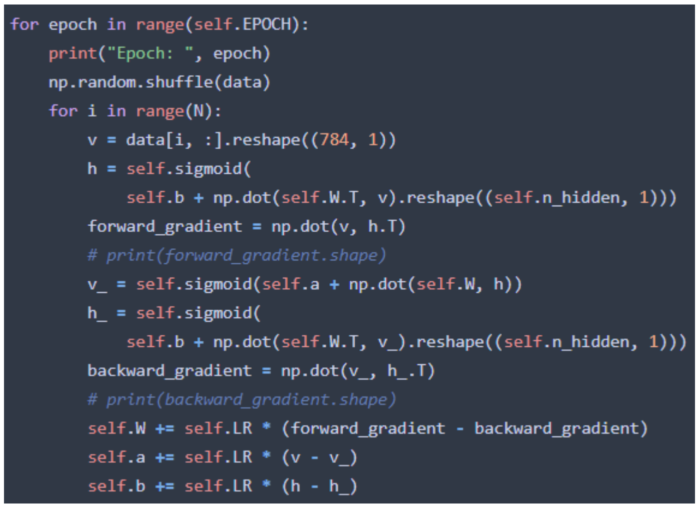
2 Experiment

2.1 Data

We use PyTorch's Torchvision to download the MNIST dataset. Besides, we just use the training data without any labeling information when we run the RBM.

2.2 Training

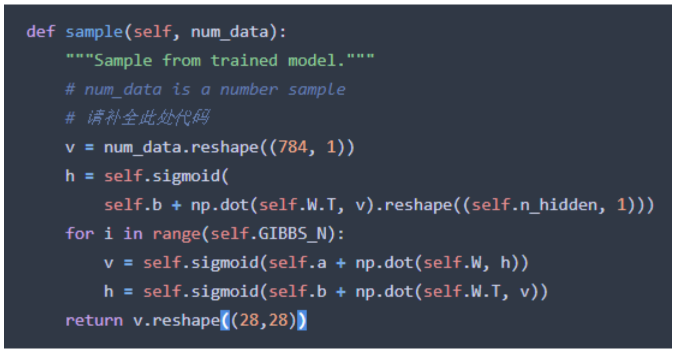
Just follow the algorithm in contrastive divergence, we can get the parameters updated. Here's the key part of the training stage:



```python
for epoch in range(self.EPOCH):
    print("Epoch: ", epoch)
    np.random.shuffle(data)
    for i in range(N):
        v = data[i, :].reshape((784, 1))
        h = self.sigmoid(
            self.b + np.dot(self.W.T, v).reshape((self.n_hidden, 1)))
        forward_gradient = np.dot(v, h.T)
        # print(forward_gradient.shape)
        v_ = self.sigmoid(self.a + np.dot(self.W, h))
        h_ = self.sigmoid(
            self.b + np.dot(self.W.T, v_).reshape((self.n_hidden, 1)))
        backward_gradient = np.dot(v_, h_.T)
        # print(backward_gradient.shape)
        self.W += self.LR * (forward_gradient - backward_gradient)
        self.a += self.LR * (v - v_)
        self.b += self.LR * (h - h_)
```

2.3 Sampling

We use the gibbs sampling. Besides, we would run through the RBM many times to make sure the sample stable. Here's the key part of the sampling stage:

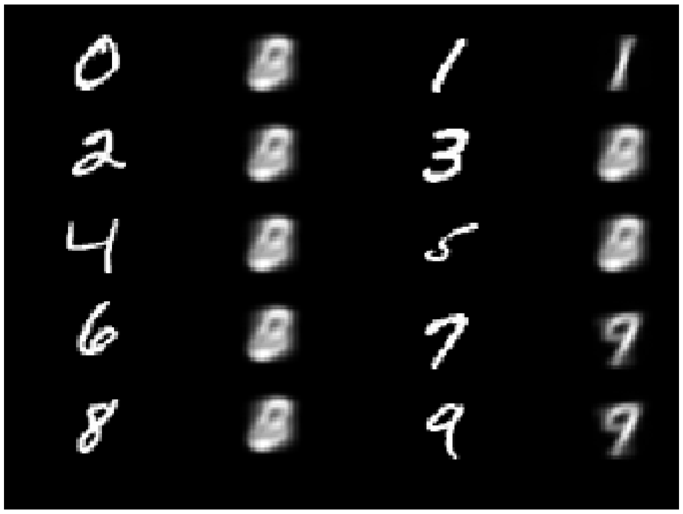


```python
def sample(self, num_data):
    """Sample from trained model."""
    # num_data is a number sample
    # 请补全此处代码
    v = num_data.reshape((784, 1))
    h = self.sigmoid(
        self.b + np.dot(self.W.T, v).reshape((self.n_hidden, 1)))
    for i in range(self.GIBBS_N):
        v = self.sigmoid(self.a + np.dot(self.W, h))
        h = self.sigmoid(self.b + np.dot(self.W.T, v))
    return v.reshape((28,28))
```

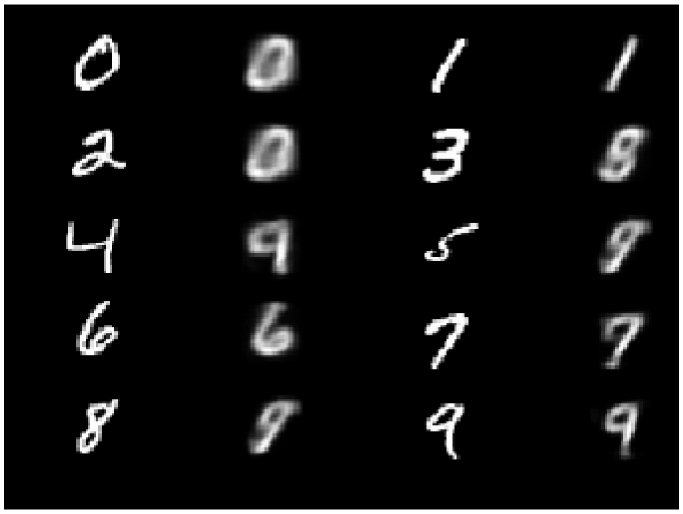
3 Result

We will see different parameters' sampling result. All the results below are based on the learning rate as 0.01, GIBBS_N as 3, and EPOCH as 3.

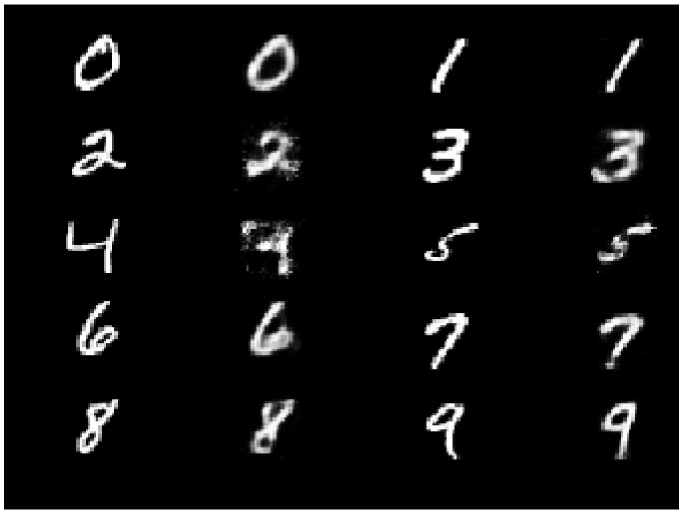
When the n_hidden is 2:



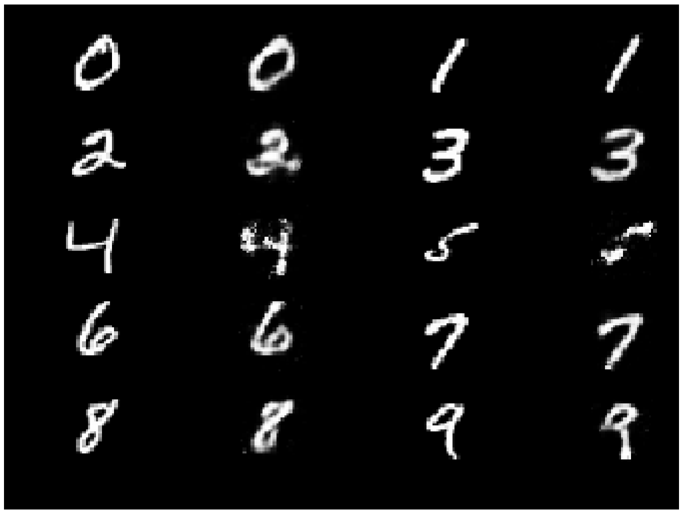
When the n_hidden is 10:



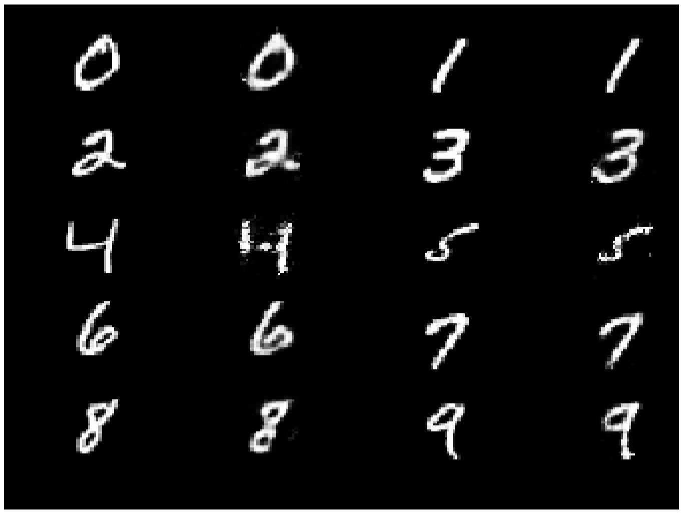
When the n_hidden is 100:



When the n_hidden is 300:



When the n_hidden is 500:



We could find that actually, when the n_hidden is 2, it won't work at all. If you want the autoencoder working, basically the n_hidden should be bigger than 100.