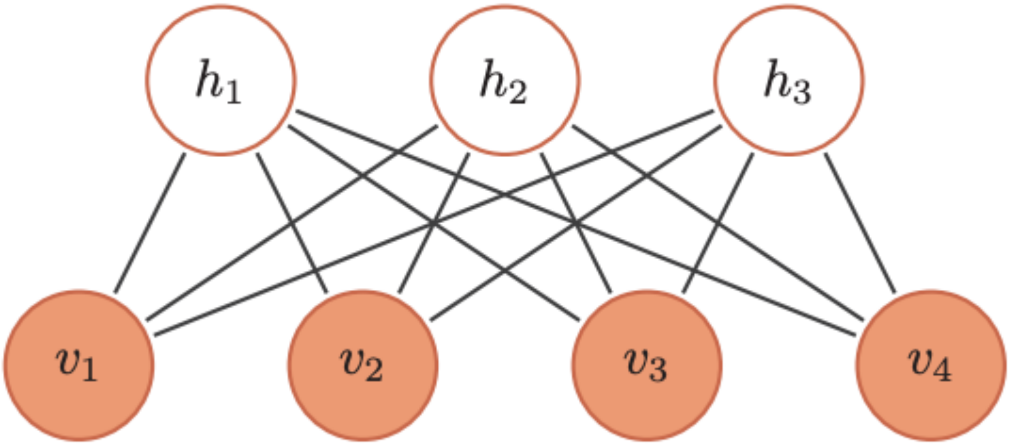
Restricted Boltzmann Machine

Introduction of RBM

Restricted Boltzmann Machine is a model based on Boltzmann machine。 Since the complexity of BN model, it can not reduce to the balance state in the reasonable time, so we added some constrains on the BN in the practice, then we have RBM.



In general, RBM is bipartial undirected model, as shown in the picture below. The pic described a RBM with 3 hidden nodes and 4 visual nodes, and each hidden nodes fully connects with each visual nodes, and vice versa. at the first glimpse, the RBM is similar to the 2-layer full connected neural network. In the next part, we will formalize the model in the mathematical way.

Model Definition

Suppose there is a RBM with m visible variables and n hidden variables, so we can define the paramter of the RBM:

- visible vector $v \in \{0, 1\}^m$
- hidden vector $h \in \{0, 1\}^n$
- Weight matrix $W \in R^{m \times n}$
- visible bias $a \in R^m$
- hidden bias $b \in R^n$

Given a dataset $\{x_i\}_i^N$, where x_i is a vector of m dimension. In our task

CD Algorithm

The goal of the CD algorithm is to learn the paramter of the RBM, the detailed is followed:

Given the input vector as the visible input v_0, and we provided 2 formula to calculate the probability of generated vector p(**h** = 1 | **v**) = $\sigma(W^T v + b)$ (1)

p(**v** = 1 | **h**) = $\sigma(Wh + a)$ (2)

for i in 1......T epochs:

```
for j in 1.....N samples:
```

1.Compute the **hidden variable distribution** $p(h_0 = 1|v_0)$ with (1)

2. Sample h_0 from $p(h_0 = 1|v_0)$
3. Compute the visible variable distribution $p(v_1 = 1|h_0)$ with (2)
4. Sample reverse visible variable v_1 from $p(v_1 = 1|h_0)$
5. Compute $p(h_1 = 1|v_1)$ with (1)
6. Sample h_1 from $p(h_1 = 1|v_1)$
7. Update W with W + $\alpha(v_0 h_0^T - v_1 h_1^T)$
8. update a with a + $\alpha(v_0 - v_1)$
9. Update b with $b + \alpha(h_0 - h_1)$

Gibbs Sampling

1. Define the initial visible input v_0

   for t in 0....T times

   ```
   Compute hidden variable distribution $$p(h_{t} = 1 | v_{t}) $$ with
   (1)

   Sample $$ h_0 $$ from $$ p(h_{t} = 1 |v_{t}) $$

   Compute visible variable distribution   $$p(v_{t+1} = 1| h_{t})$$ with
   (2)

   Sample $$v_t$$ from  $$p(v_{t+1} = 1| h_t)$$

   return $$v_{t+1}$$ and $$ h_{t}$$
   ```

Experiment

The given dataset in our task is mnist dataset, which contains 60000 images of shape 28*28, indicating the manually written digit from 0 to 9. We flatten the data into 60000 * 784 format, so the hidden layer's size is 784, while the visible layer is uncertain, and the learning rate variable is also uncertain. After training the RBM based on CD algorithm, we use the model to sample the data, than we reshpae the output vector to 28*28 images.

Analysis & Conclusion

To fit our model, we need to adjust our parameters to find the best one. Here are some examples and the outcomes of our analysis.

Learning rate α

We adjust the learning rate ranging from 0.00001-0.1, we found that the best value to fit the model is 0.01

The size of hidden layer

The size of the hidden layer is adjustable, so we change it from 2 to 1000, adn we found that the best value is about 100. If the number of layers is small we can not generate the graph we wanted, while the number of layers is large, we need to spend more time for convegence of RBM.

Implementation

Based on the framework provided by the TA, we just need to fill in the blank function. We use numpy to provide the matrix multiple calculation. The detailed code can be found in the attachment.

Argument learning

```
parser = argparse.ArgumentParser()
parser.add_argument('--hidden_layer', '-y', required=False, type=int,
default=100)
parser.add_argument('--learning_rate', '-l', required=False, type=float,
default=0.01)
args = parser.parse_args()
```

Util function

To utilize the map function on high-dimensional data, we need to define some helper functions.

```
def new_map(func, tensor):
    if type(tensor) != np.ndarray:
        return func(tensor)
    ret = []
    for iter in tensor:
        ret.append(new_map(func, iter))

    return np.array(ret)
```

To validate our model implementation, we need to monitor the energy as time goes by.

```
def _energy(self, visible, hidden):
        return -np.inner(self.a.flatten(), visible.flatten()) -
np.inner(self.b.flatten(), hidden.flatten()) \
            -np.matmul(np.matmul(np.transpose(visible), self.W), hidden)
```

CD algorithm

```
self.visible = data.reshape(-1, self.n_observe)
        for epoch in range(max_epoch):
            np.random.shuffle(data)
            for v in data:
                ## CD loss
                v = v.reshape(-1, 1)
                h_dist = new_map(sigmoid, \
                    np.matmul(np.transpose(self.W), v) + self.b)

                # h_sample = self._sample_binary(h_dist)
                h_sample = h_dist
                v_dist = new_map(sigmoid, \
                    np.matmul(self.W, h_sample) + self.a)

                # v_sample = self._sample_binary(v_dist)
                v_sample = v_dist
                h_dist2 = new_map(sigmoid, \
                    np.matmul(np.transpose(self.W), v_sample) + self.b)

                # h_sample2 = self._sample_binary(h_dist2)
                h_sample2 = h_dist2
                ## Update weight
                self.W += self.alpha * \
                    (np.matmul(v, np.transpose(h_sample)) -
                        np.matmul(v_sample, np.transpose(h_sample2)))

                self.a += self.alpha * (v - v_sample)
                self.b += self.alpha * (h_sample - h_sample2)

            print (self._energy(v, h_sample2))

            np.save("w.npy", W)
            np.save("a.npy", a)
            np.save("b.npy", b)
```

The implementation of CD is slightly different from the original version, there is no need to sample the data from probability of both hidden and visible layer, we just regard the probability as the sampled data. Then we use the `np.matmul()` and `np.transpose()` to finish all the matrix calculation. Pay attention that we save the parameters using `np.save()` , so that we can directly use it in the generation process.

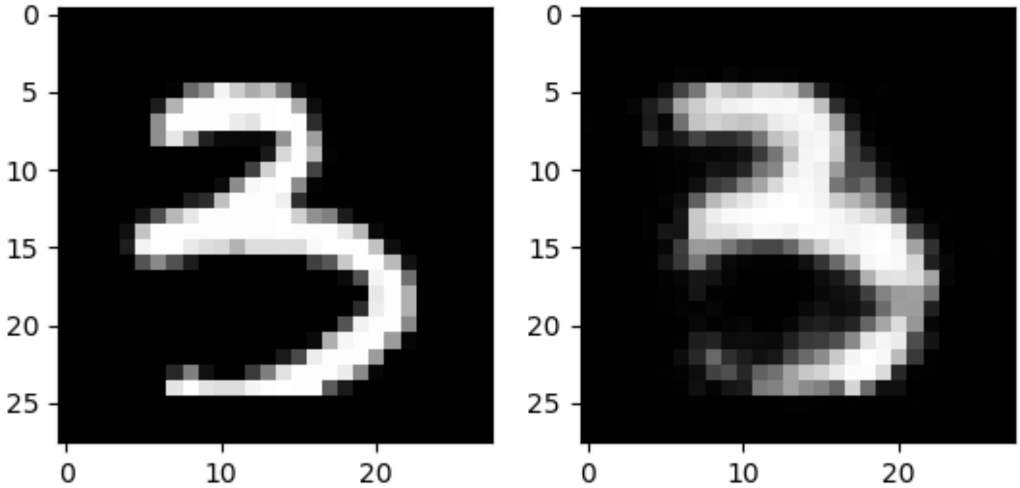
Sample data

```
self.W = np.load("w.npy")
self.a = np.load("a.npy").reshape(-1, 1)
self.b = np.load("b.npy").reshape(-1, 1)
for iter in range(iter_times):
p_h = new_map(sigmoid, np.matmul(np.transpose(self.W), new_v) + self.b)

# new_h = self._sample_binary(p_h)
new_h = p_h
p_v = new_map(sigmoid, np.matmul(self.W, new_h) + self.a)
# new_v = self._sample_binary(p_v)
new_v = p_v
# import pdb;pdb.set_trace()
return new_v
```

Generate new images

```
plt.subplot(1,2,1)
plt.imshow(raw_img.reshape((28, 28)), cmap="gray")
plt.subplot(1,2,2)
plt.imshow(gen_img.reshape((28, 28)), cmap="gray")
plt.savefig("result.png")
```


0
5
10
15
20
25
0
10
20
0
5
10
15
20
25
0
10
20