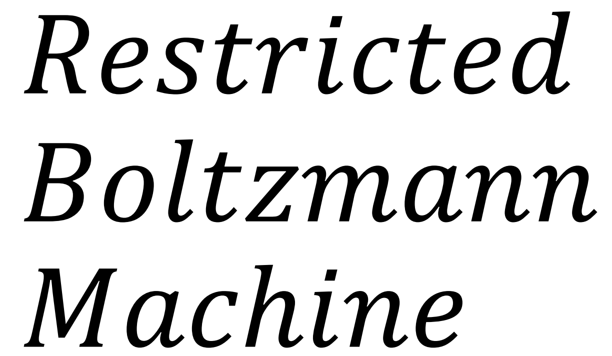


제한된 볼츠만 머신

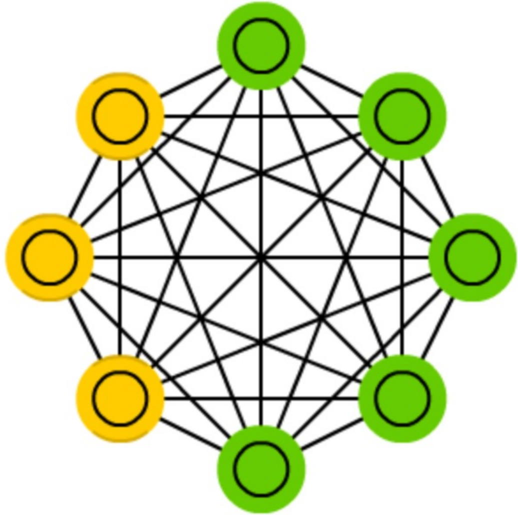
1. 선정 이유
2. 작동 원리
3. 코드

선정이유

1. 물리적 개념을 바탕으로 만들어진 모델
2. 생물의 기억을 생성하는 방법을 바탕으로 한 모델
3. 생성 모델에 대한 궁금증

작동원리: Boltzmann Machine

Boltzmann Machine (BM)



$Visible\ Unit(v)$: 입력에 해당하는 층으로 식별이 가능한 특성들(예상 변수)

$Hidden\ Unit(h)$: 식별이 불가능한 특성의 존재를 암시하기 위한 추가적인 특성 (숨
변수)

$Weights\ between\ Visible\ Units\ (r_{ij})\ and\ bias\ (b_i)$

$Weights\ between\ Hidden\ Units\ (t_{ij})\ and\ bias\ (c_i)$

$Weights\ between\ Visible\ and\ Hidden\ Unit\ (w_{ij})$

$Energy\ (E)$: 학습 정도를 수치적으로 판단하기 위한 $Cost\ Function$

입력된 특성과 숨겨진 변수들 모두 고려하여 가장 그럴듯한 값을 만들어 내는 모델.
에너지가 낮을수록 안정하다는 물리적 개념을 바탕으로, 에너지가 가장 낮은 상태를 가장 그럴듯한 상태로 판단

$Cost\ Function: Energy$

$$E(v,h) = -\left(\sum_{i,j} \left(r_{ij} v_i v_j + w_{ij} v_j h_i + t_{ij} h_i h_j \right) + \sum_i b_i v_i + \sum_j c_j h_j \right)$$

① visible interaction

② visible-hidden interaction

③ hidden interaction

④ bias of visible

⑤ bias of hidden

$$E(v,h) = -(v^T R\, v + h^T W\, v + h^T T\, h + b^T v + c^T h)$$

*Restricted Boltmann Machine*은
visible layer & hidden layer node 사이의 연결을 고려하지 않음 => 각 사건을 독립으로 볼 수 있음

$$r_{ij} = 0\,, \qquad t_{ij} = 0$$

$$E(v,h) = -h^T W\, v - b^T v - c^T h$$

제한된 볼츠만머신의 에너지 함수

(제한된) 볼츠만머신은 볼츠만 분포를 가정하여 확률분포를 생성한다.

(E가 낮은 패턴이 자주 관측되는 볼츠만 분포 가정)

$$p_{w,b,c}(v,h) = \frac{\exp(-E(v,h))}{Z}$$

$p_i \propto e^{-\epsilon_i/kT}$; Boltzman distribution

where $Z = \sum_{v,h} \exp(-E(v,h))$

관측 가능한 $visible\ unit, v$ 에 관한 확률은

$$p_{w,b,c}(v) = \sum_h \frac{1}{Z} e^{-E(v,h)} = \frac{1}{Z'} e^{-F(v)}$$

where $Z' = \sum_v \exp(-F(v))$ $and\ F(v) = -log \sum_h \exp(-E(v,h))$

이때 $F(v)$는 헬름홀츠 Free Energy, $F(v) = -k_B log Z$ 에서 왔다.

$Cost\ Function: Energy\ to\ Free\ Energy$

$$F(v) = -log \sum_h \exp\big(-E(v,h)\big)$$

$$= -\ log \sum_h e^{-(-b^T v\ - c^T h\ - h^T W\ v)}$$

$$= -log \sum_h e^{(b^T v + c^T h + h^T W\ v)}$$

$$= -b^T v\ -\ log \sum_h e^{(c^T h + h^T W\ v)}$$

이 때, $h \in \{0, 1\}$ 상태만을 가질 때,
이를 'Bernoulli RBM' 라고 한다(이진분류).

$if\ \ h \in \{0, 1\}; Bernoulli\ RBM$

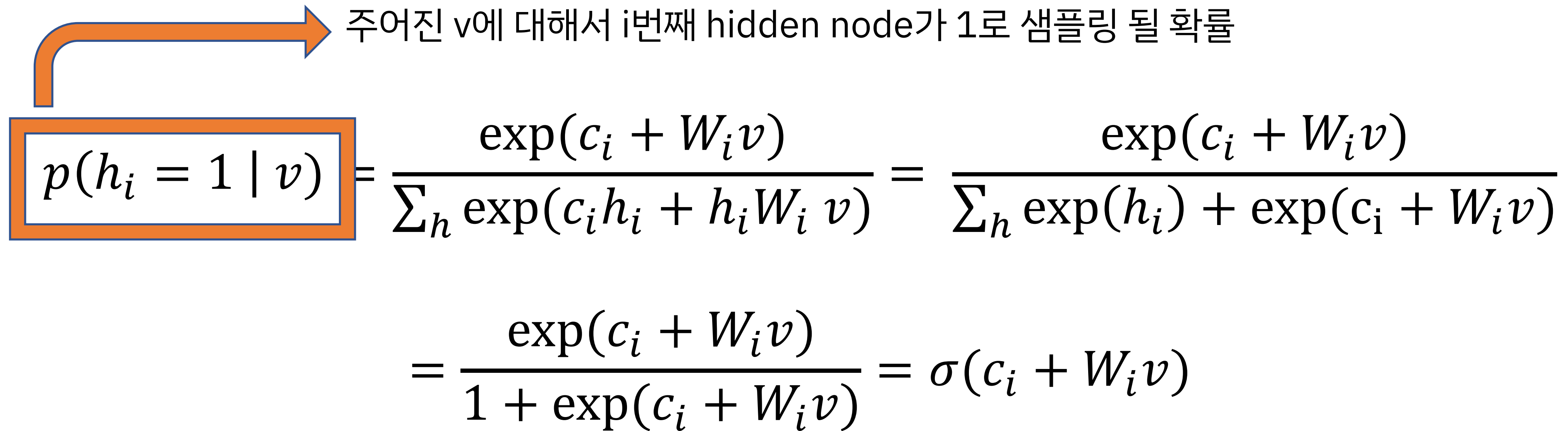
$$F(v) = -b^T v\ - \sum_{i=1}^{n} \log(e^0 + e^{c_i + W_i v}) = -b^T v\ - \sum_{i=1}^{n} \log(1 + e^{c_i + W_i v})$$

일반적인 입력 v에 대한 출력 h의 확률 분포; $p(h|v)$에 대해

$$p(h|v) = \frac{p(h,v)}{p(v)} = \frac{p(h,v)}{\sum_h p(h,v)} = \frac{\frac{1}{Z}\exp(-E(h,v))}{\sum_h \frac{1}{Z}\exp(-E(h,v))} = \frac{e^{c^T h + h^T W v}}{\sum_h e^{c^T h + h^T W v}}$$

로 주어진다.

이진 분류문제에서는 $h \in \{0, 1\}$ 이므로

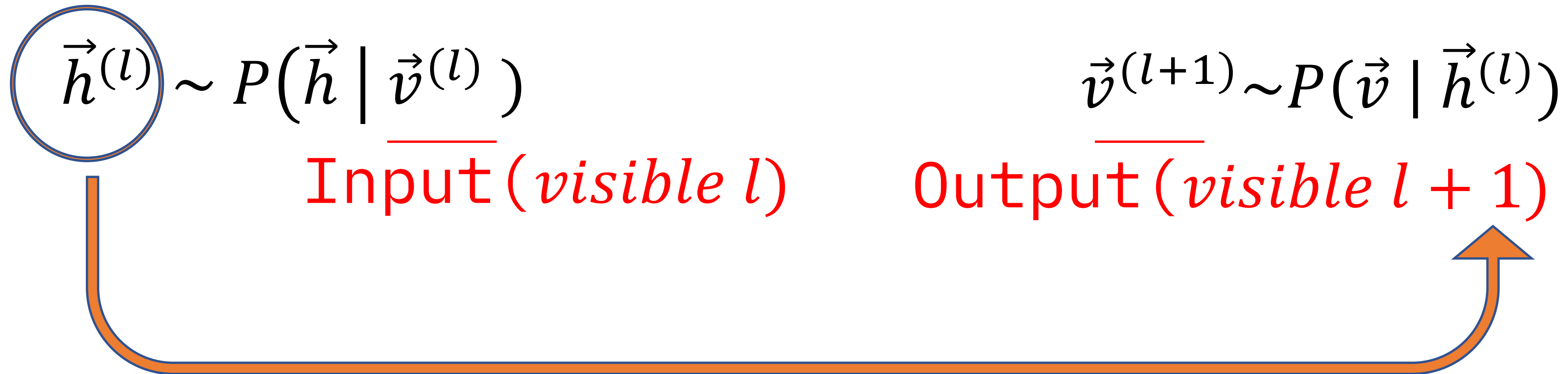


$\sigma(\cdot)$: *Sigmoid function*

same as, $\quad p\left(v_j = 1 \,\middle|\, h\right) = \sigma(b_j + W_j^T v)$

흥미롭게도, 이진분류 문제에서는 Sigmoid 함수를 얻을 수 있다.

학습 과정 : with *Gradient Descent*



제한된 볼츠만 모델로 얻는 $\boldsymbol{p(v,h)}$ 와 실제 데이터의 확률 분포 $\boldsymbol{p_o}$ 가 비슷해지도록 학습을 진행!!

p & p_o 확률을 비교할 때, 일반적으로 쿨백-라이블러 거리 $D_{KL}(P_o|P)$나 Log *Likelihood* 등을 사용함

$$D_{KL}(P|P_o) = \Sigma P(i) log \frac{P(i)}{P_o(i)}$$

-> P_o를 P 대신 사용했을 때, 엔트로피의 차이

학습 과정 : with $Gradient\ Descent$

RBM의 파라미터 b, c, W에 대하여 $likelihood$의 곱$(=\min(-\log L))$이 최대가 되도록 학습!
이하, 파라미터 b, c, W를 θ로 표기함

$$-\frac{\partial}{\partial\theta}\ln p(v) = -\frac{\partial}{\partial\theta}\ln\frac{\exp(-F(v))}{Z'} = -\frac{\partial}{\partial\theta}(-F(v) - \ln(Z'))$$

$$= \frac{\partial}{\partial\theta}F(v) + \frac{\partial}{\partial\theta}\ln(\sum_{\tilde{v}}\exp(-F(\tilde{v}))$$

$$= \frac{\partial}{\partial\theta}F(v) - \sum_{\tilde{v}}\frac{\exp(-F(\tilde{v}))}{Z'}\frac{\partial}{\partial\theta}F(\tilde{v})$$

$$= \frac{\partial}{\partial\theta}F(v) - \sum_{\tilde{v}}p(\tilde{v})\frac{\partial}{\partial\theta}F(\tilde{v})$$

From Expected value(기대값) $E(x) = \sum xP(x)$

$$= \frac{\partial}{\partial\theta}F(v) - E_p\{\frac{\partial F(\tilde{v})}{\partial\theta}\}$$

학습 과정 : with $Gradient\ Descent$

$$-\frac{\partial}{\partial\theta}\ln p(v) = \frac{\partial}{\partial\theta}F(v) - \sum_{\tilde{v}} p(\tilde{v})\frac{\partial}{\partial\theta}F(\tilde{v})$$

해당 값을 정확히 얻기 위해서는
h, v의 조합에 대해 값을 모두 계산해야 함

Gibbs Sampling을 이용해 계산할 수 있지만,
높은 계산량을 필요로 함

$$\frac{\partial}{\partial\theta}F(v) - E_p\{\frac{\partial F(\tilde{v})}{\partial\theta}\}$$

$$\approx \frac{\partial}{\partial\theta}F(v) - \frac{1}{N}\sum_{\tilde{v}\in N}\frac{\partial F(\tilde{v})}{\partial\theta}$$

$$\rightarrow \frac{\partial}{\partial\theta}[\,F(v) - \frac{1}{N}\sum_{\tilde{v}\in N}F(\tilde{v})\,]$$

$\boldsymbol{Contrastive\ Divergence(CD)}$

p(v, h)가 수렴할 때까지 구하는 것이 아닌
일정 횟수만 반복한 후 $p(v, h)$를 근사해서 사용

Gibbs Sampling을 k 번만 수행해서 $gradient$ 값을 얻는다!

$$\therefore \boldsymbol{loss = F(v) - F(v^{(k)})}$$

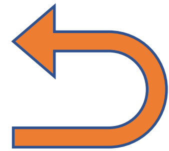
학습 과정 : with $Gradient\ Descent$

Hinton 교수님이 $CD(Contrastive\ Divergence)$를 제안하였고,
이후, 알고리즘의 결과가 Local Optimum으로 수렴한다고 이론적으로 증명됨

Geoffrey Everest Hinton
인공지능(AI) 분야를 개척자: 오차 역전파법, 딥러닝, 힌턴 다이어그램 등을 발명

$\boldsymbol{Gibbs\ Sampling}$ & $\boldsymbol{CD}$

$\boldsymbol{Step\ 1.\ Sample\ \vec{h}^{l}\sim P(\vec{h}|\vec{v}^{l})}$



$\boldsymbol{Step\ 2.\ Sample\ \vec{v}^{l+1}\sim P(\vec{v}|\vec{h}^{l})}$

$\boldsymbol{k(=1)}$ 번 반복 수행

작성코드

가장 익숙한 손글씨 데이터 **MNIST**를 실습으로 사용함

RBM Codes- Prepare Dataset :: MNIST

```
csian@172 RBM % tree
.
├── MNIST
│   └── raw
│       ├── t10k-images-idx3-ubyte
│       ├── t10k-images-idx3-ubyte.gz
│       ├── t10k-labels-idx1-ubyte
│       ├── t10k-labels-idx1-ubyte.gz
│       ├── train-images-idx3-ubyte
│       ├── train-images-idx3-ubyte.gz
│       ├── train-labels-idx1-ubyte
│       └── train-labels-idx1-ubyte.gz
├── mnist_load.py
└── rbm_fit.py

2 directories, 10 files
```

mnist_load.py를 통해 다운로드

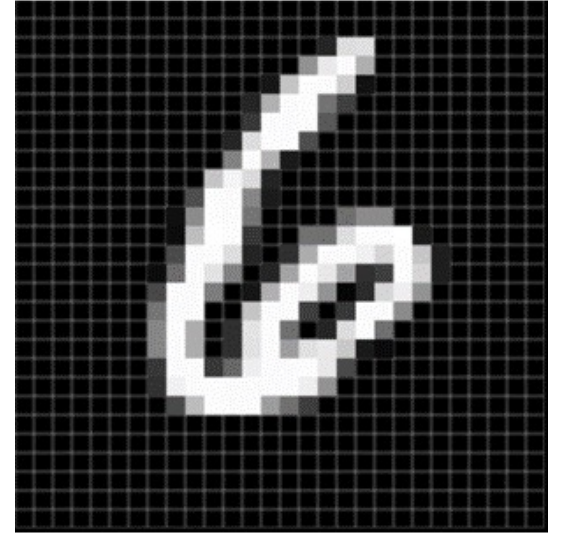
RBM Codes- Prepare Dataset :: MNIST

RBM — Vim mnist_load.py ▸ Python — 125×30

1: mnist_load.py   buffers

```
19 #!/opt/homebrew/Caskroom/miniforge/base/envs/tf_38/bin/python
18 import torch
17 from torchvision.datasets import MNIST
16 import torchvision.transforms as transforms
15
14 # Normalize data with mean=0.5, std=1.0
13 mnist_transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (1.0,))])
12
11 download_root="./"
10
 9 train_dataset=MNIST(download_root, transform=mnist_transform, train=True, download=True)
 8 valid_dataset=MNIST(download_root, transform=mnist_transform, train=False, download=True)
 7 test_dataset=MNIST(download_root, transform=mnist_transform, train=False, download=True)
 6
 5 DataLoader=torch.utils.data.DataLoader
 4 # dataset loading
 3 batch_size = 64
 2 train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
 1 valid_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)
20 test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)
```

RBM Codes- model class and main codes



Input: (28x28)(0 or 1)=728
Output: (28x28)(0 or 1)=728

1: rbm_fit.py

```
1  #!/opt/homebrew/Caskroom/miniforge/base/envs/tf_38/bin/python
 1 import numpy as np
 2 import torch
 3 import torchvision
 4 import matplotlib.pyplot as plt
 5
 6 def save_img(file_name, img)
 7   f="./pic/%s.png"%file_name
 8   img=np.transpose(img.numpy(), (1, 2, 0))
 9   plt.imsave(f, img)
```

RBM Codes- model class

```python
# RBM model with torch
class RBM(torch.nn.Module):
  def __init__(self, n_vis=784, n_hid=500, k=5):
    super(RBM, self).__init__()
    # weigths rand initialize
    self.W=torch.nn.Parameter(torch.randn(n_hid, n_vis)*1e-2)
    # bias 0 initialize
    self.v_bias=torch.nn.Parameter(torch.zeros(n_vis))
    self.h_bias=torch.nn.Parameter(torch.zeros(n_hid))
    # Number of Divergence
    self.k=k

  def sampling(self, p):
    # from Uniform distribution: achieve probability
    _p=p-torch.autograd.Variable(torch.rand(p.size()))
    # if _p>p return 1 else return -1;
    p_sign=torch.sign(_p)
    # with Relu-> if _p>p return 1 else return 0;
    return torch.nn.functional.relu(p_sign)

  def v_to_h(self, v):
    # P(h_i=1|v)=sigmoid(c_i+W_i*v)
    p_h=torch.sigmoid(torch.nn.functional.linear(v, self.W, self.h_bias))
    sample_h=self.sampling(p_h)
    return p_h, sample_h

  def h_to_v(self, h):
    # P(v_j=1|h)=sigmoid(b_j+W'_j*h)
    p_v=torch.sigmoid(torch.nn.functional.linear(h, self.W.t(), self.v_bias))
    sample_v=self.sampling(p_v)
    return p_v, sample_v

  def forward(self, v):
    p_h0, h0=self.v_to_h(v)
    _h=h0
    for _ in range(self.k): # Divergence count
      _p_v, _v=self.h_to_v(_h)
      _p_h, _h=self.v_to_h(_v)

    return v, _v    # return initial v and after k times divergenced _v

  def freeEnergy(self, v):
    # F(v)=-b'v-SUM_i[ log(1+exp(c_i+W_iv)) ]
    vbias=v.mv(self.v_bias) # b'v
    wx_b=torch.nn.functional.linear(v, self.W, self.h_bias) # c+Wv
    sum_i=torch.log(1+torch.exp(wx_b))
    ssum=torch.sum(sum_i, dim=1)
    # F(v)^(1)
    return (-ssum-vbias).mean()
```

class RBM{

Instance variables(클래스 인스턴스 변수){

- w : weight(visible node # x hidden node #)
  w[i][j]=visible[i]와 hidden[j] 사이의 가중치를 의미
- v_bias: bias of Visible
- h_bias: bias of Hidden
- k: Contrastive Divergence 횟수

}

Methods(클래스 메서드){

- sampling(p)
- v_to_h(v)
- h_to_v(h)
- forward(v)
- freeEnergy(v)

}};

RBM class : Methods

v(parameter):
Model의 input으로 (batch_size x 784)

$$F(v) = -b^T v - \sum_{i=1}^{n} \log(1 + e^{c_i + W_i v})$$

```python
def freeEnergy(self, v):
  # F(v)=-b'v-SUM_i[ log(1+exp(c_i+W_iv)) ]
  vbias=v.mv(self.v_bias) # b'v
  wx_b=torch.nn.functional.linear(v, self.W, self.h_bias) # c+Wv
  sum_i=torch.log(1+torch.exp(wx_b))
  ssum=torch.sum(sum_i, dim=1)
  # F(v)^(1)
  return (-ssum-vbias).mean()
```

sum_i.size()=(Batch_size x hidden #)

Batch FreeEnergy의 평균

이 후, main에서 Loss를 계산할 때 사용됨!

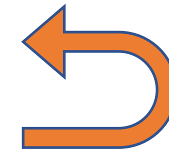
$$loss = F(v) - F(v^{(k)})$$

```python
v, v1=rbm(_sample_data)
_loss=rbm.freeEnergy(v)-rbm.freeEnergy(v1)
loss_append(_loss.data.item())
```

RBM class : Methods

$$Step\ 1.\ Sample\ \vec{h}^{l} \sim P(\vec{h}|\vec{v}^{l})$$

$$Step\ 2.\ Sample\ \vec{v}^{l+1} \sim P(\vec{v}|\vec{h}^{l})$$



$k(=1)$ 번 반복 수행

v(parameter):
Model의 input으로 (batch_size x 784)

```python
def forward(self, v):
  p_h0, h0=self.v_to_h(v)          # Step 1
  _h=h0
  for _ in range(self.k): # Divergence count
    _p_v, _v=self.h_to_v(_h)       # Step 2
    _p_h, _h=self.v_to_h(_v)
...
  return v, _v    # return initial v and after k times divergenced _v
```

Contrastive Divergence k 번 반복

RBM class : Methods

$p(h_i = 1 \mid v)$

```python
def v_to_h(self, v):
  # P(h_i=1|v)=sigmoid(c_i+W_i*v)
  p_h=torch.sigmoid(torch.nn.functional.linear(v, self.W, self.h_bias))
  sample_h=self.sampling(p_h)
  return p_h, sample_h
```

$$p(h_i = 1 \mid v) = \sigma(c_i + W_i v)$$

$p(v_j = 1 \mid h)$

```python
def h_to_v(self, h):
  # P(v_j=1|h)=sigmoid(b_j+W'_j*h)
  p_v=torch.sigmoid(torch.nn.functional.linear(h, self.W.t(), self.v_bias))
  sample_v=self.sampling(p_v)
  return p_v, sample_v
```

$$p(v_j = 1 \mid h) = \sigma(b_j + W_j^T v)$$

```python
def sampling(self, p):
  # from Uniform distribution: achieve probability
  _p=p-torch.autograd.Variable(torch.rand(p.size()))
  # if _p>p return 1 else return -1;
  p_sign=torch.sign(_p)
  # with Relu-> if _p>p return 1 else return 0;
  return torch.nn.functional.relu(p_sign)
```

sample $\vec{h}^l$ *or sample* $\vec{v}^{l+1}$

$$if(\,rand(1) < p(h_i = 1|v)\,):\ h_j = 1;$$
$$else: h_j = 0;$$

$$if(\,rand(1) < p(v_i = 1|h)\,):\ v_i = 1;$$
$$else: v_i = 0;$$

p 의 확률로 1, $(1-p)$의 확률로 0 출력

학습된 분포 p에 대한 출력을 생성

RBM Codes- main codes

```
if __name__=="__main__":
  # k=1 conversion 1 times
  rbm=RBM(k=1)
  # optimizer Adam
  optimizer=torch.optim.Adam(rbm.parameters(), 1e-3)

  batch_size=64
  train_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST('./', train=True, download=True, transform=torchvision.transforms.Compose([torchvision.transforms.ToTensor()])), batch_size=batch_size)

  test=torchvision.datasets.MNIST('./', train=False, transform=torchvision.transforms.Compose([torchvision.transforms.ToTensor()]))
  # test with 32 data
  sample_data=test.data[:32, :].view(-1, 784)
  sample_data=sample_data.type(torch.FloatTensor)/255.

  v, v1=rbm(sample_data)
  save_img("label", torchvision.utils.make_grid(v.view(32, 1, 28, 28).data))
  save_img("before_train", torchvision.utils.make_grid(v1.view(32, 1, 28, 28).data))

  for epoch in range(100):
    loss=[]
    for _, (data, label) in enumerate(train_loader):
      data=torch.autograd.Variable(data.view(-1, 784))
      # we consider bernoulli h
      _sample_data=data.bernoulli()
      # forward
      v, v1=rbm(_sample_data)
      _loss=rbm.freeEnergy(v)-rbm.freeEnergy(v1)
      loss.append(_loss.data.item())
      optimizer.zero_grad()
      _loss.backward()
      optimizer.step()
    print("EPOCH: %d\r" %(epoch+1), flush=True, end="")
    save_img("epoch%d" %(epoch+1), torchvision.utils.make_grid(v1.view(32, 1, 28, 28).data))

  v, v1=rbm(sample_data)
  save_img("after_train", torchvision.utils.make_grid(v1.view(32, 1, 28, 28).data))
  plt.plot(loss, "co")
  plt.show()
RMAL  rbm_fit.py[+]                    freeEnergy  python  utf-8[unix]  100%
```

Training (the `for epoch in range(100):` loop)

Testing (the final `rbm(sample_data)` call and the `save_img("after_train", …)` line)

main :: training

```python
for epoch in range(100):
    loss=[]
    for _, (data, label) in enumerate(train_loader):
        data=torch.autograd.Variable(data.view(-1, 784))
        # we consider bernoulli
        _sample_data=data.bernoulli()
        # forward
        v, v1=rbm(_sample_data)
        _loss=rbm.freeEnergy(v)-rbm.freeEnergy(v1)
        loss.append(_loss.data.item())
        optimizer.zero_grad()
        _loss.backward()
        optimizer.step()
    print("EPOCH: %d\r" %(epoch+1), flush=True, end="")
    save_img("epoch%d" %(epoch+1), torchvision.utils.make_grid(v1.view(32, 1, 28, 28).data))
```

Epoch: 100

sample_data(batch_size x 784)(0 or 1): data[i]의 확률로 1, (1-data[i])의 확률로 0

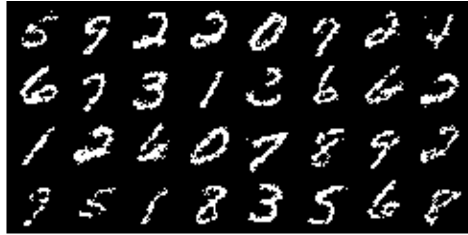
$loss = F(v) - F(v^{(k)})$

$w := arg\min_{w} \log(L(v))$

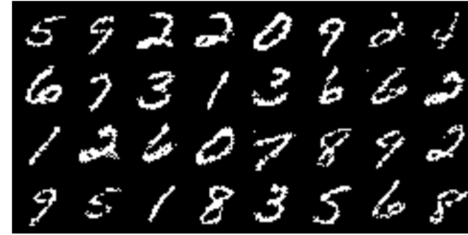
작성코드

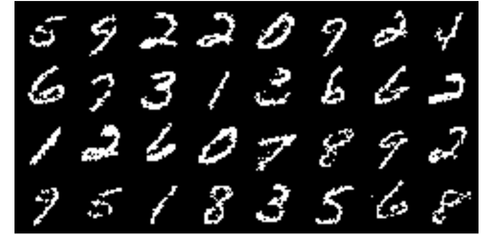
RBM Codes- Result: training epoch(Last batch 32)

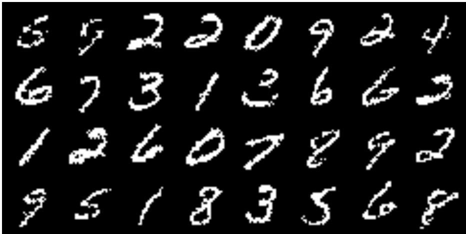
Epoch 1

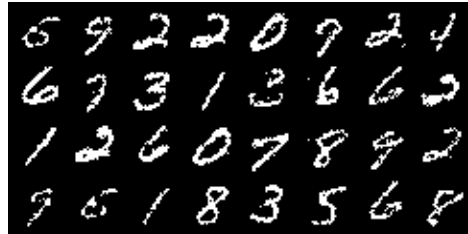

Epoch 10

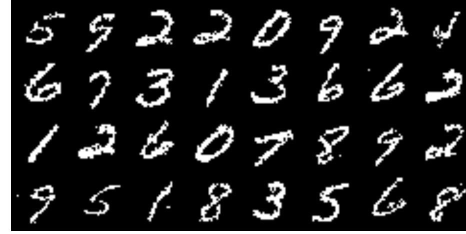
Epoch 20

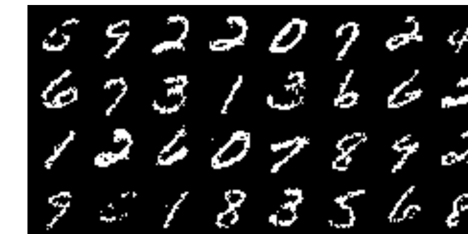

Epoch 30

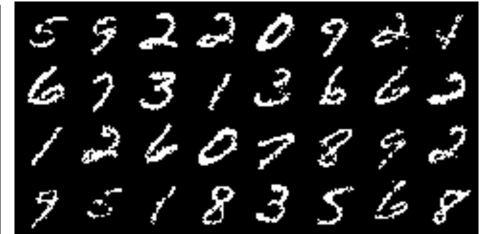

Epoch 40


Epoch 50


Epoch 60

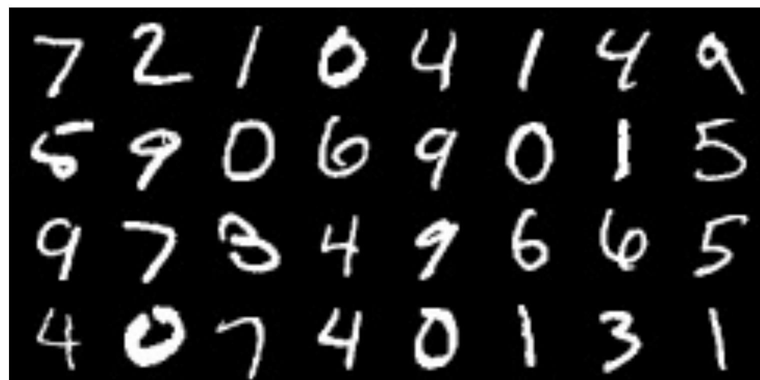

Epoch 70

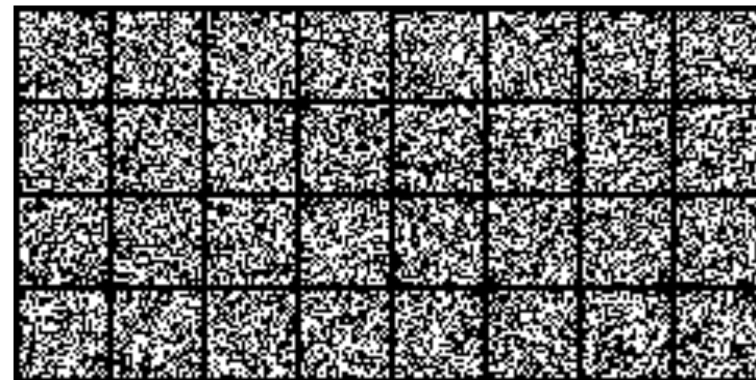

Epoch 90

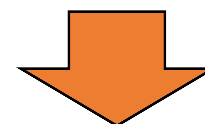

Epoch 100

첫 10 epoch 이후로는 인식이 가능할 정도의 샘플을 생성

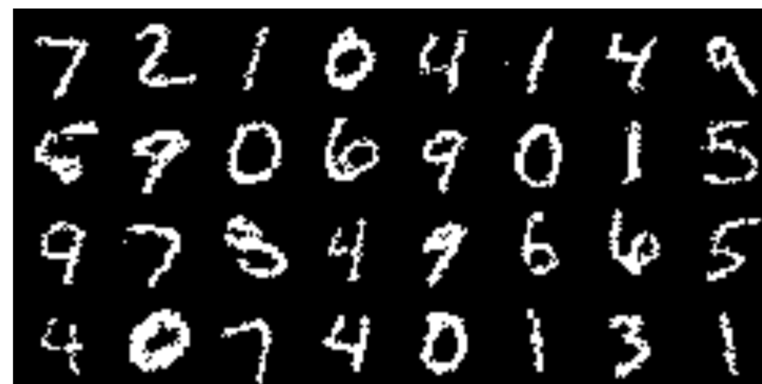
RBM Codes- Result: Test Dataset 32







After 100 Epochs



Implicit sampling & Explicit sampling

Energy Based Model(EBMs):
RBMs;1980

VAEs(Variational Autoencoders); 2014
GANs;2014

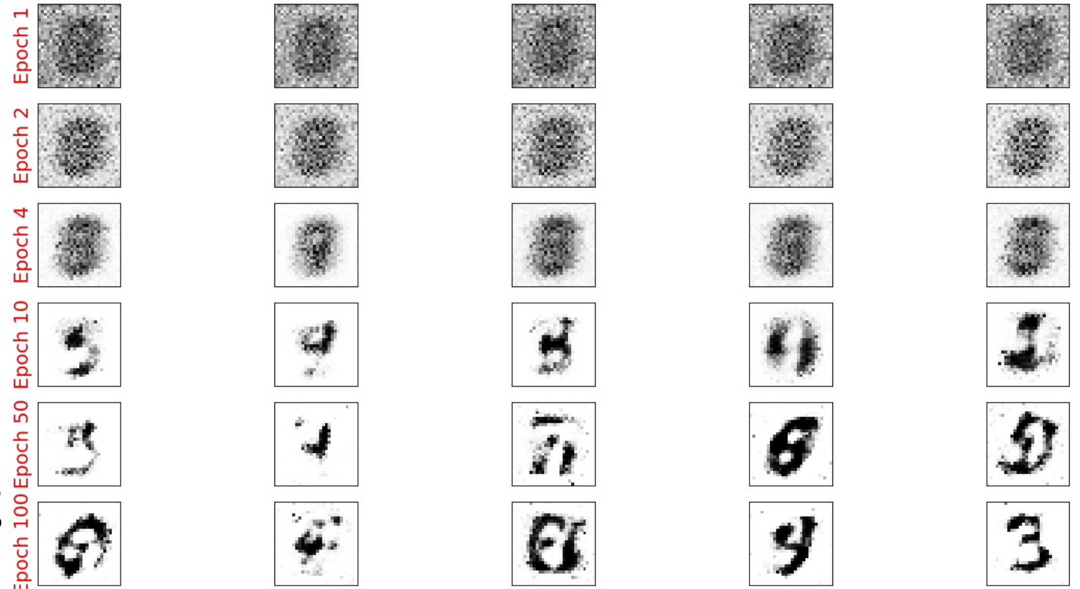
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 100)               2000
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 100)               0
_________________________________________________________________
dense_1 (Dense)              (None, 784)               79184
=================================================================
Total params: 81,184
Trainable params: 81,184
Non-trainable params: 0
_________________________________________________________________

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 100)               78500
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 100)               0
_________________________________________________________________
dropout (Dropout)            (None, 100)               0
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 101
=================================================================
Total params: 78,601
Trainable params: 78,601
Non-trainable params: 0
_________________________________________________________________
```

생성모델 역사의 초반부를 장식했던 RBM은 GAN등의 역전파를 이용하는 모델들에 밀려 많이 사용되지는 않지만,

- 단순성과 안정성
- 통계적 힘의 공유
- 소요시간 대비 높은 성능
- 유연하게 데이터를 생성(구조에 대한 구속이 적음)
- 학습이 더욱 쉬움

Implicit Generation and Generalization in Energy-Based Model(2019)

위와 같은 특징들로 새로운 생성모델을 더욱 향상시키기 위해 연구되고 사용되고 있다.



GAN MNIST generated img without CNN(not DCGAN)

감사합니다