E13.

1(a). $h \in \{-1,0,1\}^H \quad x \in \{0,1\}^d$

$$p(x,h|\theta) = \frac{1}{Z}\exp\left(\sum_{j=1}^{H} w_j^T x \cdot h_j + \sum_{j=1}^{H} h_j b_j\right)$$

$$p(x|\theta) = \sum_{h\in\{0,1\}^H} \frac{1}{Z}\exp\left(\sum_{j=1}^{H} w_j^T x h_j + \sum_{j=1}^{H} h_j b_j\right)$$

$$= \frac{1}{Z}\sum_{h\in\{0,1\}^H} \prod_{j=1}^{H} \exp\left((w_j^T x + b_j)\cdot h_j\right)$$

$$= \frac{1}{Z}\prod_{j=1}^{H}\sum_{h_j\in\{0,1\}} \exp\left((w_j^T x + b_j)\cdot h_j\right)$$

$$= \frac{1}{Z}\prod_{j=1}^{H} \exp(w_j^T x + b_j) + 1 + \exp(-w_j^T x - b_j)$$

$$= \frac{1}{Z}\prod_{j=1}^{H}\left(1 + 2\cosh(w_j^T x + b_j)\right)$$

where $\cosh(w_j^T x + b_j) = \dfrac{\exp(w_j^T x + b_j) + \exp(-w_j^T x - b_j)}{2}$.

So, $g_j(x,\theta_j) = 1 + 2\cosh(w_j^T x + b_j)$

1.(b) $$\frac{\partial \log p(x_n|\theta)}{\partial w_j} = \frac{\partial}{\partial w_j}\Big(\underbrace{\sum_{j=1}^{H}\log\left(1+2\cosh(w_j^T x_n + b_j)\right)}_{f(x_n,\theta)} - \log Z\Big)$$

$$= \frac{\partial}{\partial w_j}\Big(f(x_n,\theta) - \log\sum_{x\in\{0,1\}^d}\exp(f(x,\theta))\Big)$$

$$= \underbrace{\frac{2\sinh(w_j^T x_n + b_j)\,x_n}{1+2\cosh(w_j^T x_n + b_j)}}_{\psi_1(x_n,\theta_j)} - \sum_{x\in\{0,1\}^d}\frac{\exp(f(x,\theta))\cdot\psi_1(x,\theta_j)}{\sum_{x\in\{0,1\}^d}\exp(f(x,\theta))}$$

$\frac{\exp(f(x,\theta))}{\sum_{x\in\{0,1\}^d}\exp(f(x,\theta))} = p(x|\theta)$

$$\textcircled{1} = x_n\cdot\sigma(w_j^T x_n + b_j) - E_{x\sim p(x|\theta)}\left[x\cdot\sigma(w_j^T x + b_j)\right]$$

$$\frac{\partial \log p(x_n|\theta)}{\partial b_j} = \underbrace{\frac{2\sinh(w_j^T x_n + b_j)\cdot 1}{1+2\cosh(w_j^T x_n + b_j)}}_{\psi_2(x_n,\theta_j)} - \frac{\sum_{x\in\{0,1\}^d}\exp(f(x,\theta))\cdot\psi_2(x,\theta_j)}{\sum_{x\in\{0,1\}^d}\exp(f(x,\theta))}$$

$$\textcircled{2} = \sigma(w_j^T x_n + b_j) - E_{x\sim p(x|\theta)}\left[\sigma(w_j^T x + b_j)\right]$$

E2 a) $p(x|\theta) = \frac{1}{Z}\prod_{j=1}^{H}\sum_{k=1}^{C}\alpha_{jk}\frac{1}{(2\pi)^{d/2}}\exp\left(-\frac{1}{2}\|x-u_{jk}\|^2\right)$

$$= \frac{1}{Z}\sum_{k_1=1}^{C}\cdots\sum_{k_H=1}^{C}\prod_{j=1}^{H}\alpha_{jk_j}\frac{1}{(2\pi)^{d/2}}\exp\left(-\frac{1}{2}\|x-u_{jk_j}\|^2\right)$$

where inner Π is $\left(\prod_{j=1}^{H}\alpha_{jk_j}\right)\frac{1}{(2\pi)^{dH/2}}\exp\left(-\frac{1}{2}\sum_{j=1}^{H}\|x-u_{jk_j}\|^2\right)$

where inner exp is $-\frac{1}{2}\sum_{j=1}^{H}(x-u_{jk_j})^T(x-u_{jk_j})$

because x has no index, $\Rightarrow -\frac{H}{2}\left(x-\frac{1}{H}\sum_{j=1}^{H}u_{jk_j}\right)^T\left(x-\frac{1}{H}\sum_{j=1}^{H}u_{jk_j}\right)$

So $p(x|\theta) = \sum_{k_1=1}^{C}\cdots\sum_{k_H=1}^{C}\frac{1}{Z}\cdot\frac{1}{(2\pi)^{d(H-1)/2}}\left[\exp\left(-\frac{1}{2}\|x-\frac{1}{H}\sum_{j=1}^{H}u_{jk_j}\|^2\right)\right]^H$

$$p(x|\theta) = \sum_{k_1=1}^{C}\cdots\sum_{k_H=1}^{C}\frac{1}{Z}\cdot\frac{1}{2\pi^{d(H-1)/2}}\cdot\left[\exp\left(-\frac{1}{2}\|x-\frac{1}{H}\sum_{j=1}^{H}u_{jk_j}\|^2\right)\right]^{H-1}$$

$$\cdot\underbrace{\frac{1}{2\pi^{d/2}}\exp\left(-\frac{1}{2}\|x-\frac{1}{H}\sum_{j=1}^{H}u_{jk_j}\|^2\right)}_{\sim N}$$

where $Z = \frac{1}{2\pi^{d(H-1)/2}}\cdot\left[\exp\left(-\frac{1}{2}\|x-\frac{1}{H}\sum_{j=1}^{H}u_{jk_j}\|^2\right)\right]^{H-1}$ is a normalization term

So $p(x|\theta)$ is C^K term mixture of Gaussian.

and $m_k = \frac{1}{H}\sum_{j=1}^{H}u_{jk_j}$. ■

b). $m_k = \frac{1}{H}\sum_{j=1}^{H}u_{jk_j}$ $\quad k\in\{1,2\}^2$

So possible combinations are

$m_k = \frac{1}{2}\sum_{j=1}^{2}u_{jk_j}$ ⇒

$k=\{1,1\}$ $m_k = \frac{1}{2}(u_{11}+u_{21}) = \begin{pmatrix}1\\1\end{pmatrix}$

$k=\{1,2\}$ $m_k = \frac{1}{2}(u_{11}+u_{22}) = \begin{pmatrix}1\\2\end{pmatrix}$

$k=\{2,1\}$ $m_k = \frac{1}{2}(u_{12}+u_{21}) = \begin{pmatrix}2\\1\end{pmatrix}$

$k=\{2,2\}$ $m_k = \frac{1}{2}(u_{12}+u_{22}) = \begin{pmatrix}2\\2\end{pmatrix}$

above are four m_k of mixtures centers

sheet13

February 15, 2021

1 Training a Restricted Boltzmann Machine

In this exercise, we implement and train an RBM to model the distribution of MNIST digits 3. The following code loads some libraries, extracts the MNIST digits 3, and converts them to binary vectors.

[132]:
```
import matplotlib
%matplotlib inline
from matplotlib import pyplot as plt
import numpy,numpy.random
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original')
X = (mnist.data[mnist.target==3]>127.5)*1.0
X.shape,X.min(),X.max()
```

```
/home/jiayun/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:77:
DeprecationWarning: Function fetch_mldata is deprecated; fetch_mldata was
deprecated in version 0.20 and will be removed in version 0.22
  warnings.warn(msg, category=DeprecationWarning)
/home/jiayun/.local/lib/python3.6/site-packages/sklearn/utils/deprecation.py:77:
DeprecationWarning: Function mldata_filename is deprecated; mldata_filename was
deprecated in version 0.20 and will be removed in version 0.22
  warnings.warn(msg, category=DeprecationWarning)
```

[132]:
```
((7141, 784), 0.0, 1.0)
```

The RBM learning algorithm consists of several parts, which we will implement one after the other.

1.0.1 Implement the sigmoid function (10 P)

Implement a function that receives a vector of values and applies the sigmoid function

$$\mathrm{sigm}(t) = \frac{\exp(t)}{1+\exp(t)}$$

element-wise to that vector. Note that a naive implementation the function is numerically instable. Instability can be avoided by using the hyperbolic tangent and applying some rescaling/offseting so

that it matches the sigmoid. Alternatively, one can distinguish between large values of t and small values of t and apply a different treatment to these two cases. In any case, your implementation should be numerically stable, and fast (i.e. use vector operations instead of loops).

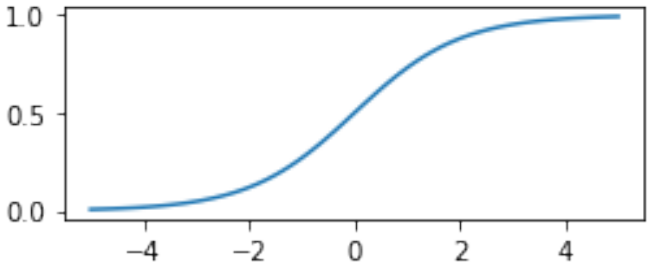
[133]:
```
def sigm(z):

    # -----------------------------------
    # TODO: replace by your code
    # -----------------------------------
    # It can be proved that tanh(x) = 2*sigmoid(2x) - 1
    p = (numpy.tanh(z/2) + 1)/2
    # -----------------------------------

    return p
```

Your implementation can be tested with the code below

[134]:
```
z = numpy.linspace(-5,5,100)
plt.figure(figsize=(4,1.5)); plt.plot(z,sigm(z)); plt.show()
```



Before implementing the RBM, we need a data structure to store its parameters. Here, we use a dictionary containing the weights and bias given as numpy arrays. Our RBM has 784 input dimensions (the number of pixels in a MNIST digit), and 100 hidden units (i.e. 100 experts).

[135]:
```
d = 784
H = 100

rbm = {
    'W': numpy.random.normal(0,0.1,[d,H]),
    'b': numpy.zeros([H]),
}
```

The initial weights of the RBM are set at random in order to break symmetries.

1.0.2 Implement the alternate Gibbs sampler (10 P)

We would like to implement a function that performs alternate Gibbs sampling to generate samples from the distribution modeled by the RBM. Recall: Each unit of the RBM is sampled according to a Bernoulli distribution of some parameter p depending on the other units:

$$\forall_{j=1}^{H}: \; h_j \sim \text{Bernoulli}(p = \text{sigm}(\boldsymbol{x}^\top \boldsymbol{w}_j + b_j))$$
$$\forall_{i=1}^{d}: \; x_i \sim \text{Bernoulli}(p = \text{sigm}(\boldsymbol{h}^\top \boldsymbol{w}_i))$$

where $\boldsymbol{w}_i$ denotes the vector of weights that connect to input feature i.

Hint: drawing from a Bernoulli distribution is equivalent to drawing from a binomial distribution with ***n=1****. Hence, you can make use of the function* ***numpy.random.binomial****, which supports vector operation by admits a vector of probabilities as input. Here is an example:*

[136]:
```
numpy.random.binomial(1,numpy.linspace(0,1,22))
```

[136]:
```
array([0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1])
```

**Task: Implement the function below that receives as input a variable `rbm` representing the dictionary containing the RBM parameters, a data point `x` given as a numpy array of shape `(784,)`, and an integer `k` corresponding to the number of alternate Gibbs sampling steps it should perform. Your function should return a sample of same shape as your data point.

Hint: You can use matrix-vector multiplication to compute several probabilities values in parallel.

[137]:
```
def gibbs(rbm,x,k):

    # ----------------------------------
    # TODO: replace by your code
    # ----------------------------------
    for index in range(k):
        ph = sigm(x@rbm['W'] + rbm['b'])
        h = numpy.random.binomial(1,ph)
        px = sigm(h@rbm['W'].T)
        x = numpy.random.binomial(1,px)
    xm=x
    return xm
```

The gibbs sampler you have implemented can be tested on some MNIST digit.

[138]:
```
xm = gibbs(rbm,X[0],3)

plt.figure(figsize=(4,1.5))
ax = plt.subplot(1,2,1); ax.set_title('before'); ax.imshow(X[0].
 ↪reshape(28,28),cmap='gray')
```

```
ax = plt.subplot(1,2,2); ax.set_title('after'); ax.imshow(xm.
 ↪reshape(28,28),cmap='gray')
plt.show()
```

The sample looks like random noise. This is because the RBM has not been trained yet.

1.0.3 Implementing the gradient (10 P)

Now we will create a function that implements the gradient of the log-likelihood the RBM associates to a given data point $\boldsymbol{x}$. Recall: The components of the gradient are given by:

$$\nabla_{\boldsymbol{w}_j} \log p(\boldsymbol{x}|\theta) = \boldsymbol{x} \cdot \text{sigm}(\boldsymbol{w}_j^\top \boldsymbol{x} + b_j) - \mathbb{E}_{p(\boldsymbol{x}|\theta)}[\boldsymbol{x} \cdot \text{sigm}(\boldsymbol{w}_j^\top \boldsymbol{x} + b_j)]$$
$$\nabla_{b_j} \log p(\boldsymbol{x}|\theta) = \text{sigm}(\boldsymbol{w}_j^\top \boldsymbol{x} + b_j) - \mathbb{E}_{p(\boldsymbol{x}|\theta)}[\text{sigm}(\boldsymbol{w}_j^\top \boldsymbol{x} + b_j)]$$

The rightmost term will be approximated using CD-k, i.e. we start from the data point and apply k steps of Gibbs samping using the function implemented above. We set the number of steps to k=3.

Task: Implement a function that computes the proposed approximation of the gradient for a given data point $\boldsymbol{x}$ given as an array of shape `(784,)`. Your function should return a tuple `(dW,db)` where `dW` and `db` are arrays of same size as the parameters `W` and `b` of the RBM, but containing the approximated gradients.

[139]:
```
def gradient(rbm,x):

    # ------------------------------------
    # TODO: replace by your code
    # ------------------------------------
    x_r = gibbs(rbm, x, 3) #x realization
    x_r = x_r[:,numpy.newaxis]
    x = x[:,numpy.newaxis]
    sig_r = sigm(rbm['W'].T @ x_r + rbm['b'][:,numpy.newaxis]) # 100,1
    sig = sigm(rbm['W'].T @ x + rbm['b'][:,numpy.newaxis]) # 100,1
```

```
    Ex_in_w = x_r @ sig.T
    Ex_in_b = sig_r
    dW = (x @ sig.T - Ex_in_w).squeeze()
    db = (sig - Ex_in_b).squeeze()
    # ------------------------------------

    return dW,db
```

1.0.4 Implementing gradient ascent (10 P)

We now would like to learn the parameters of the RBM by performing a gradient ascent procedure. The function to implement should loop for the number of iterations `nbit` specified as argument, at each iteration pick one data point at random, compute the gradient using the method implemented above, and preform the update step using the learning rate `lr` specified as argument. Recall: the update step is given by:

$$\boldsymbol{w}_j \leftarrow \boldsymbol{w}_j + \gamma \cdot \nabla_{\boldsymbol{w}_j} \log p(\boldsymbol{x}|\theta)$$
$$b_j \leftarrow b_j + \gamma \cdot \nabla_{b_j} \log p(\boldsymbol{x}|\theta)$$

where γ is the learning rate.

Task: Implement a function that performs the gradient ascent procedure using the learning rate and number of iterations specified as arguments. The function receives a dataset `X` given as an array of shape `(N,784)` where `N` is the number of data points. The function terminates after all training iterations have been performed. The function does not need to return anything.

[140]:
```
def train(rbm,X,lr=0.005,nbit=25000):

    # ------------------------------------
    # TODO: replace by your code
    # ------------------------------------
    for index in range(nbit):
        random_index = numpy.random.choice(X.shape[0])
        x = X[random_index,:]
        dW, db = gradient(rbm, x)
        rbm['W'] += lr * dW
        rbm['b'] += lr * db
    # ------------------------------------
```

1.0.5 Training and Inspecting the RBM

We now train the RBM. If the RBM has been implemented correctly, the code above should run in approximately 1-2 minutes. For debuging purposes, you may reduce the number of iterations

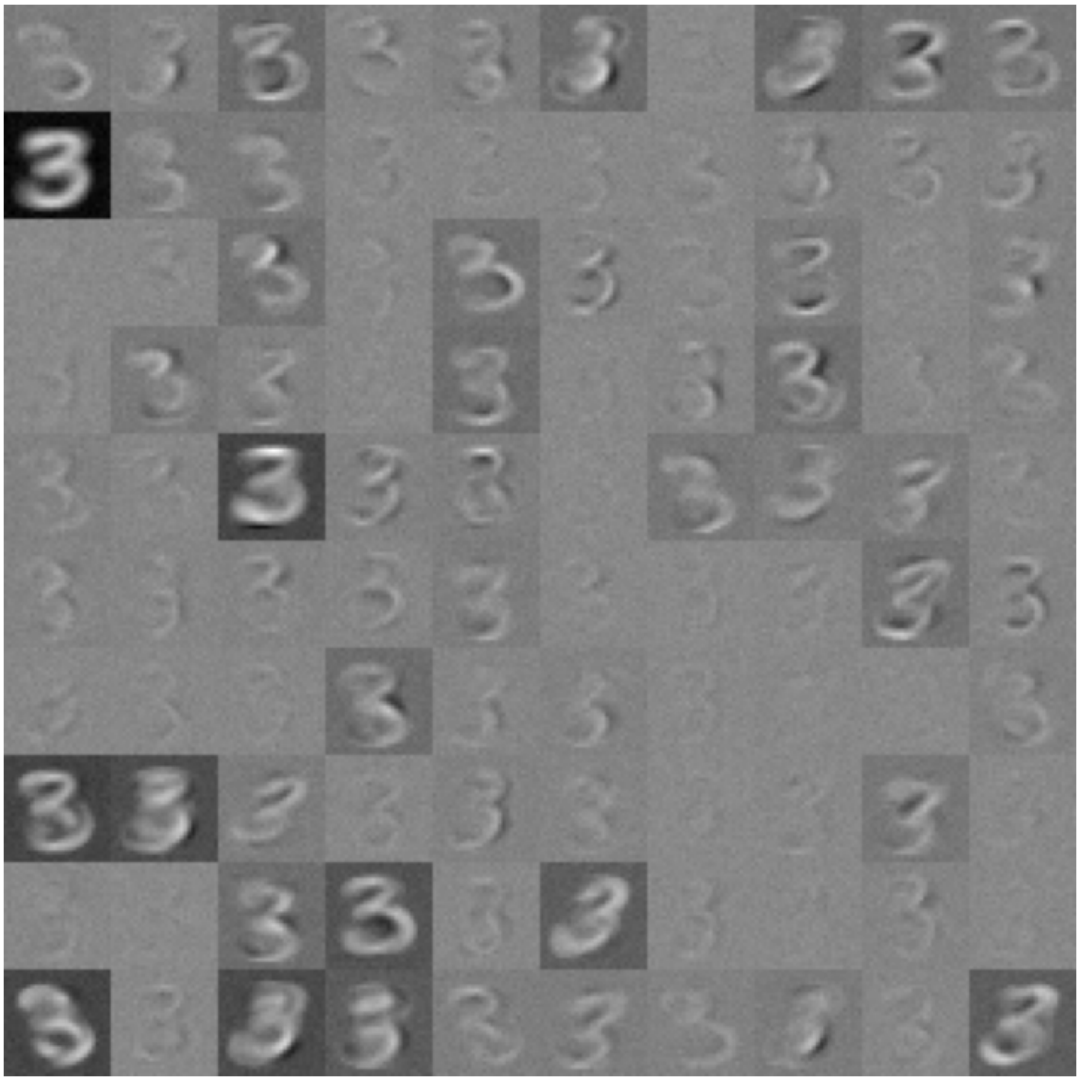
initially to 1000, and switching back to the original number of iterations for the final run.

[141]:
```
train(rbm,X)
```

Once the RBM has been trained, the result of training can be visualized. The following code renders the weights of the RBM as a mosaic where each tile corresponds to one latent variable (or expert).

[142]:
```
plt.figure(figsize=(10,10))
plt.axis('off')
plt.subplots_adjust(left=0,right=1,bottom=0,top=1)
plt.imshow(rbm['W'].reshape(28,28,10,10).transpose(2,0,3,1).
 ↪reshape(280,280),cmap='gray')
```

[142]:
```
<matplotlib.image.AxesImage at 0x7fa85259f8d0>
```



We observe that the RBM has learned a rich set of features, some of which are localized in pixel space, and therefore, apply to a large region of the input space. These features are statistically robust and also transferrable to other digits.