Report of Homework3 IFT6390

Team Member

Qiang Ye (20139927), Lifeng Wan (20108546)

Coding Environment

python 3.5.2, numpy 1.14.2 matplotlib 2.2.0

---

1 THEORETICAL PART a (25 pts): derivatives and relationships between basic functions

Given
-- << logistic sigmoid >> $\text{sigmoid}(x) = \frac{1}{1+exp(-x)}$
-- << hyperbolic tangend >> $\tanh(x) = \frac{exp(x)-exp(-x)}{exp(x)+exp(-x)}$
-- << soft plus >> $\text{softplus}(x) = ln(1 + exp(x))$
-- << sign >> function sign which returns +1 if its arguments is positive, -1 if negative and 0 if 0.
-- $\mathbf{1}_S(x)$ is the indicator function which returns 1 if $x \in S$ or x respects condition S, otherwise returns 0
-- << rectifier >> function which keeps only the positive part of its argument: rect(x) returns x if $x \geq 0$ and returns 0 if $x < 0$. It is also named RELU (rectified linear unit): rect(x) = RELU(x) = $[x]_+$ = max(0,x) = $x \cdot \mathbf{1}_{\{x>0\}}(x)$

1. Show that $\text{sigmoid}(x) = \frac{1}{2}(\tanh(\frac{1}{2}x) + 1)$
   Answer

$$
\begin{aligned}
\frac{1}{2}\Big(\tanh(\frac{1}{2}x) + 1\Big) &= \frac{1}{2}\Big(\frac{exp(\frac{1}{2}x) - exp(-\frac{1}{2}x)}{exp(\frac{1}{2}x) + exp(-\frac{1}{2}x)} + 1\Big) \\
&= \frac{1}{2}\Big(\frac{2\ exp(\frac{1}{2}x)}{exp(\frac{1}{2}x) + exp(-\frac{1}{2}x)}\Big) \\
&= \frac{exp(\frac{1}{2}x)}{exp(\frac{1}{2}x) + exp(-\frac{1}{2}x)} \\
&= \frac{1}{1 + \frac{exp(-\frac{1}{2}x)}{exp(\frac{1}{2}x)}} \\
&= \frac{1}{1 + exp(-\frac{1}{2}x - \frac{1}{2}x)} \\
&= \frac{1}{1 + exp(-x)} \\
&= \text{sigmoid}(x)
\end{aligned}
$$

2. Show that $ln\ \text{sigmoid}(x) = -\text{softplus}(-x)$
Answer

$$\begin{aligned} ln\ \text{sigmoid}(x) &= ln\ \frac{1}{1+exp(-x)} \\ &= -ln\ (1+exp(-x)) \\ &= -\text{softplus}(-x) \end{aligned}$$

3. Show that the derivative of the sigmoid is: $\text{sigmoid}'(x) = \frac{d\,\text{sigmoid}}{d\,x}(x) = \text{sigmoid}(x)(1-\text{sigmoid}(x))$
Answer

$$\begin{aligned} \text{sigmoid}'(x) &= \frac{d\,\text{sigmoid}}{d\,x}(x) \\ &= \frac{d\ \frac{1}{1+exp(-x)}}{dx} \\ &= -(\frac{1}{1+exp(-x)})^2 \frac{d\,(1+exp(-x))}{dx} \\ &= \frac{1}{(1+exp(-x))^2} exp(-x) \\ &= \frac{1}{1+exp(-x)} \frac{exp(-x)}{1+exp(-x)} \\ &= \frac{1}{1+exp(-x)} \frac{1+exp(-x)-1}{1+exp(-x)} \\ &= \frac{1}{1+exp(-x)} \left(1-\frac{1}{1+exp(-x)}\right) \\ &= \text{sigmoid}(x)(1-\text{sigmoid}(x)) \end{aligned}$$

4. Show that the tanh derivative is: tanh'(x) = 1 - tanh$^2(x)$
Answer

$$\begin{aligned} tanh'(x) &= \frac{d\,(\frac{exp(x)-exp(-x)}{exp(x)+exp(-x)})}{d\,x} \\ &= \frac{(exp(x)+exp(-x))(exp(x)-exp(-x))' - (exp(x)-exp(-x))(exp(x)+exp(-x))'}{(exp(x)+exp(-x))^2} \\ &= \frac{(exp(x)+exp(-x))^2 - (exp(x)-exp(-x))^2}{(exp(x)+exp(-x))^2} \\ &= 1 - \frac{(exp(x)-exp(-x))^2}{(exp(x)+exp(-x))^2} \\ &= 1 - \tanh^2(x) \end{aligned}$$

5. Write the sign function using only indicator functions: $\text{sign}(x) = \ldots$
Answer $\text{sign}(x) = \mathbf{1}_{\{x>0\}}(x) - \mathbf{1}_{\{x<0\}}(x)$

6. Write the derivative of the absolute function $\text{abs}(x) = |x|$. Note: its derivate at 0 is not defined, but your function abs' can return 0 at 0. Note2: use the sign function: $\text{abs}'(x) = \ldots$
Answer $\text{abs}'(x) = \text{sign}(x)$

7. Write the derivative of the function rect. Note: its derivative at 0 is undefined, but your function can return 0 at 0. Note2: use the indicator function. $\text{rect}'(x) = \ldots$
Answer $\text{rect}'(x) = \mathbf{1}_{\{x>0\}}(x)$

8. Let the squared L_2 norm of a vector be: $\|\mathbf{x}\|_2^2 = \sum_i \mathbf{x}_i^2$. Write the vector of the gradient: $\frac{\partial \|\mathbf{x}\|_2^2}{\partial \mathbf{x}} = \dots$
   Answer
   $$\frac{\partial \|\mathbf{x}\|_2^2}{\partial \mathbf{x}} = \left(\frac{\partial \sum_i \mathbf{x}_i^2}{\partial \mathbf{x}_1}, \quad \frac{\partial \sum_i \mathbf{x}_i^2}{\partial \mathbf{x}_2}, \quad \cdots \right)^T = \left(2\mathbf{x}_1, \quad 2\mathbf{x}_2, \quad \cdots \right)^T = 2\mathbf{x}$$

9. Let the norm L_1 of a vector be: $\|\mathbf{x}\|_1 = \sum_i |\mathbf{x}_i|$. Write the vector of the gradient: $\frac{\partial \|\mathbf{x}\|_1}{\partial \mathbf{x}} = \dots$
   Answer
   $$\frac{\partial \|\mathbf{x}\|_1}{\partial \mathbf{x}} = \left(\frac{\partial \sum_i |x_i|}{\partial \mathbf{x}_1}, \quad \frac{\partial \sum_i |x_i|}{\partial \mathbf{x}_2}, \quad \cdots \right)^T = \left(\mathrm{sign}(\mathbf{x}_1), \quad \mathrm{sign}(\mathbf{x}_2), \quad \cdots \right)^T$$
   $$= \mathrm{sign}(\mathbf{x})$$

---

2 THEORETICAL PART b (25 pts): Gradient computation for parameters optimization in a neural net for multiclass classification

Let $D_n = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$be the dataset with $\mathbf{x}^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \{1, \dots, m\}$ indicating the class within m classes. For vectors and matrices in the following equations, vectors are by default considered to be column vectors.

Consider a neural net of the type $Multilayer\ perceptron$ (MLP) with only one hidden layer (meaning 3 layers total if we count the input and output layers). The hidden layer is made of d_h neurons fully connected to the input layer. We shall consider a non linearity of type **rectifier** (Rectified Linear Unit of **RELU**) for the hidden layer. The ouput layer is made of m neurons that are fully connected to the hidden layer. They are equipped with a **softmax** non linearity. The ouput of the j^{th} neuron of the output layer gives a score for the class j which is interpreted as the probability of $\mathbf{x}$ being of class j.

It is highly recomended that you draw the neural net as it helps understanding all the steps.

1. Let $\mathbf{W}^{(1)}$ a $d_h \times d$ matrix of weights and $\mathbf{b}^{(1)}$ the bias vector be the connections between the input layer and the hidden layer. What is the dimension of $\mathbf{b}^{(1)}$? Give the formula of the preactivation vector (before the non linearity) of the neurons of hidden layer $\mathbf{h}^a$ given $\mathbf{x}$ as input, first in a matrix form ($\mathbf{h}^a = \dots$), and then details on how to compute one element $\mathbf{h}^a_j = \dots$. Write the output vector of the hidden layer $\mathbf{h}^s$ with respect to $\mathbf{h}^a$.
   Answer
   $\mathbf{b}^{(1)} \in \mathbb{R}^{dh}$
   $\mathbf{h}^a = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$
   $\mathbf{h}^a_j = \mathbf{W}^{(1)}_j\mathbf{x} + \mathbf{b}^{(1)}_j = \sum_{i=1}^{d} \mathbf{W}^{(1)}_{ji}\mathbf{x}_i + \mathbf{b}^{(1)}_j$
   $\mathbf{h}^s = \mathbf{RELU}(\mathbf{h}^a)$

2. Let $\mathbf{W}^{(2)}$ a weight matrix and $\mathbf{b}^{(2)}$ a bias vector be the connections between the hidden layer and the output layer. What are the dimensions of $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$? Give the formula of the activation function of the neurons of the output layer $\mathbf{o}^a$ with respect to input $\mathbf{h}^s$ in a matrix form and then write in a detailed form for $\mathbf{o}^a_k$.
   Answer
   $\mathbf{W}^{(2)}$ is a matrix of $m \times d_h$, $\mathbf{b}^{(2)} \in \mathbb{R}^m$.
   $\mathbf{o}^a = \mathbf{W}^{(2)}\mathbf{h}^s + \mathbf{b}^{(2)}$
   $\mathbf{o}^a_k = \mathbf{W}^{(2)}_k\mathbf{h}^s + \mathbf{b}^{(2)}_k = \sum_{j=1}^{d_h} \mathbf{W}^{(2)}_{kj}\mathbf{h}^s_j + \mathbf{b}^{(2)}_k$

3. The output of the neurons at the output layer is given by:

$$\mathbf{o}^s = \text{softmax}(\mathbf{o}^a)$$

Give the precise equation for $\mathbf{o}_k^s$ using the softmax (formular with the exp). ***Show*** that the $\mathbf{o}_k^s$ are positive and sum to 1. Why is this important?

Answer

$\mathbf{o}_k^s = \frac{exp(\mathbf{o}_k^a)}{\sum_{i=1}^m exp(\mathbf{o}_i^a)}$

$\mathbf{o}_k^s$ is always positive becuase $exp(\mathbf{o}_k^a)$ and $\sum_k exp(\mathbf{o}_k^a)$ are always positive with $\mathbf{o}_k^a \in \mathbb{R}$.

$\sum_{k=1}^m \mathbf{o}_k^s = \sum_{k=1}^m \frac{exp(\mathbf{o}_k^a)}{\sum_{i=1}^m exp(\mathbf{o}_i^a)} = \frac{\sum_{k=1}^m exp(\mathbf{o}_k^a)}{\sum_{i=1}^m exp(\mathbf{o}_i^a)} = 1$

softmax function maps a score vector to a vector in which all elements sum to 1 such that each value in the vector can represent the probability with which an example belongs to a certain class. Therefore, it is crucial that all probabilities sum to 1 since an example should always be in one of the m classes.

4. The neural net computes, for an input vector $\mathbf{x}$, a vector of probability scores $\mathbf{o}^s(\mathbf{x})$. The probability, computed by a neural net, that an observation $\mathbf{x}$ belongs to calss y is given by the y^{th} output $\mathbf{o}_y^s(\mathbf{x})$. This suggests a loss function such as:

$$L(\mathbf{x}, y) = -log\ \mathbf{o}_y^s(\mathbf{x})$$

Find the equation of L as a function of the vector $\mathbf{o}^a$. It is easily achievable with the correct substitution using the equation of the previous question.

Answer

$$\begin{aligned} L(\mathbf{x}, y) = -log\ \mathbf{o}_y^s(\mathbf{x}) &= -log \frac{exp(\mathbf{o}_y^a(\mathbf{x}))}{\sum_{k=1}^m exp(\mathbf{o}_k^a(\mathbf{x}))} \\ &= log \sum_{k=1}^m exp(\mathbf{o}_k^a(\mathbf{x})) - log\ exp(\mathbf{o}_y^a(\mathbf{x})) \\ &= log \sum_{k=1}^m exp(\mathbf{o}_k^a(\mathbf{x})) - \mathbf{o}_y^a(\mathbf{x}) \end{aligned}$$

5. The training of the neural net will consist of finding parameters that minimize the empirical risk $\hat{R}$ associated with this loss function. What is $\hat{R}$? What is precisely the set θ of parameters of the network? How many scalar parameters n_θ are there? Write down the optimization problem of training the network in order to find the optimal values for these parameters.

Answer

$\hat{R} = \frac{1}{n} \sum_{i=1}^n L(\mathbf{x}^{(i)}, y^{(i)})$

$\theta = \{(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}), (\mathbf{W}^{(2)}, \mathbf{b}^{(2)})\}$

$n_\theta = d_h \times (d+1) + m \times (d_h + 1)$

The optimization process is to find an optimal $\theta^\star$:

$\theta^\star = \underset{\theta}{arg\ min}\ \hat{R} = \underset{\theta}{arg\ min}\ \frac{1}{n} \sum_{i=1}^n L(\mathbf{x}^{(i)}, y^{(i)})$

6. To find a solution to this optimization problem, we will use gradient descent. What is the (batch) gradient descent equation for this problem?

Answer

$\theta = \theta - \alpha \frac{\partial \hat{R}_\theta(D_n)}{\partial \theta}$

where α is a learning rate (step size) when performing gradient descent.

7. We can compute the vector of the gradient of the empirical risk $\hat{R}$ with respect to the parameters set θ this way

$$\begin{pmatrix} \frac{\partial \hat{R}}{\partial \theta_1} \\ \vdots \\ \frac{\partial \hat{R}}{\partial \theta_{n_\theta}} \end{pmatrix} = \frac{1}{n} \sum_{i=1}^{n} \begin{pmatrix} \frac{\partial L(\mathbf{x}_i, y_i)}{\partial \theta_1} \\ \vdots \\ \frac{\partial L(\mathbf{x}_i, y_i)}{\partial \theta_{n_\theta}} \end{pmatrix}$$

This hints that we only need to know how to compute the gradient of the loss L with an example $(\mathbf{x}, y)$ with respect to the parameters, defined as followed:

$$\frac{\partial L}{\partial \theta} = \begin{pmatrix} \frac{\partial L}{\partial \theta_1} \\ \vdots \\ \frac{\partial L}{\partial \theta_{n_\theta}} \end{pmatrix} = \begin{pmatrix} \frac{\partial L(\mathbf{x}, y)}{\partial \theta_1} \\ \vdots \\ \frac{\partial L(\mathbf{x}, y)}{\partial \theta_{n_\theta}} \end{pmatrix}$$

We shall use **gradient backpropagation**, starting with loss L and going to the output layer $\mathbf{o}$ then down the hidden layer $\mathbf{h}$ then finally at the input layer $\mathbf{x}$.

Show that

$$\frac{\partial L}{\partial \mathbf{o}^a} = \mathbf{o}^s - \mathrm{onehot}_m(y)$$

Note: Start from the expression of L as a function of $\mathbf{o}^a$ that you previously found. Start by computing $\frac{\partial L}{\partial \mathbf{o}^a_k}$ for $k \neq y$ (using the start of the expression of the logarithm derivate). Do the same ting for $\frac{\partial L}{\partial \mathbf{o}^a_y}$.

Answer

$$\frac{\partial L}{\partial \mathbf{o}^a} = \frac{\partial}{\partial \mathbf{o}^a}\left(log \sum_{k=1}^{m} exp(\mathbf{o}^a_k) - \mathbf{o}^a_y\right)$$

$$= \begin{pmatrix} \frac{\partial}{\partial \mathbf{o}^a_1}\left(log \sum_{k=1}^{m} exp(\mathbf{o}^a_k) - \mathbf{o}^a_y\right) \\ \vdots \\ \frac{\partial}{\partial \mathbf{o}^a_m}\left(log \sum_{k=1}^{m} exp(\mathbf{o}^a_k) - \mathbf{o}^a_y\right) \end{pmatrix}$$

when $k \neq y$,

$$\begin{aligned} \frac{\partial}{\partial \mathbf{o}^a_k}\left(log \sum_{k=1}^{m} exp(\mathbf{o}^a_k) - \mathbf{o}^a_y\right) &= \frac{\partial}{\partial \mathbf{o}^a_k}\left(log \sum_{k=1}^{m} exp(\mathbf{o}^a_k)\right) \\ &= \frac{1}{\sum_{k=1}^{m} exp(\mathbf{o}^a_k)} \frac{\partial}{\partial \mathbf{o}^a_k} \sum_{k=1}^{m} exp(\mathbf{o}^a_k) \\ &= \frac{exp(\mathbf{o}^a_k)}{\sum_{k=1}^{m} exp(\mathbf{o}^a_k)} \\ &= \mathbf{o}^s_k \end{aligned}$$

when $k = y$,

$$\begin{aligned} \frac{\partial}{\partial \mathbf{o}^a_k}\left(log \sum_{k=1}^{m} exp(\mathbf{o}^a_k) - \mathbf{o}^a_y\right) &= \frac{\partial}{\partial \mathbf{o}^a_k}\left(log \sum_{k=1}^{m} exp(\mathbf{o}^a_k)\right) - 1 \\ &= \frac{1}{\sum_{k=1}^{m} exp(\mathbf{o}^a_k)}\left(\frac{\partial}{\partial \mathbf{o}^a_k} \sum_{k=1}^{m} exp(\mathbf{o}^a_k)\right) - 1 \\ &= \frac{exp(\mathbf{o}^a_k)}{\sum_{k=1}^{m} exp(\mathbf{o}^a_k)} - 1 \\ &= \mathbf{o}^s_k - 1 \end{aligned}$$

Combine two above equations, we can conclude that:

$$\frac{\partial L}{\partial \mathbf{o}^a} = \mathbf{o}^s - \text{onehot}_m(y)$$

8. What is the numpy equivalent expression (it can fit 2 operations)?
   grad_oa = …
   …
   Answer
   numpy form:

```
# for one example
grad_oa = np.copy(os) # (n_output, 1)
np.put(grad_oa, y, grad_oa[y,0]-1) # (n_output, 1)

# for batch_size
def one_hot(m, y):
    y = y.reshape(-1, ) # (batch_size, )
    batch_size = y.shape[0]
    result = np.zeros((batch_size, m))
    result[np.arange(batch_size), y] = 1
    return result
grad_oa = o_s - one_hot(m, y) # (batch_size, n_output)
```

   IMPORTANT: From now on when we ask to "compute" the gradients or partial derivates, you only need to write them as function of previously computed derivates (**do not substitute the whole expressions already computed in the previous questions!**)

9. Compute the gradients with respect to parameters $\mathbf{W}^{(2)}$ and $\mathbf{b}^{(2)}$ of the output layer. Since L depends on $\mathbf{W}^{(2)}_{kj}$ and $\mathbf{b}^{(2)}_k$ only through $\mathbf{o}^a_k$ the result of the chain rule is:

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}_{kj}} = \frac{\partial L}{\partial \mathbf{o}^a_k} \frac{\partial \mathbf{o}^a_k}{\partial \mathbf{W}^{(2)}_{kj}}$$

   and

$$\frac{\partial L}{\partial \mathbf{b}^{(2)}_k} = \frac{\partial L}{\partial \mathbf{o}^a_k} \frac{\partial \mathbf{o}^a_k}{\partial \mathbf{b}^{(2)}_k}$$

   Answer

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}_{kj}} = \frac{\partial L}{\partial \mathbf{o}^a_k} \frac{\partial}{\partial \mathbf{W}^{(2)}_{kj}} \Big(\sum_{j=1}^{d_h} \mathbf{W}^{(2)}_{kj} \mathbf{h}^s_j + \mathbf{b}^{(2)}_k \Big) = \frac{\partial L}{\partial \mathbf{o}^a_k} \mathbf{h}^s_j$$

$$\frac{\partial L}{\partial \mathbf{b}^{(2)}_k} = \frac{\partial L}{\partial \mathbf{o}^a_k} \frac{\partial}{\partial \mathbf{b}^{(2)}_k} \Big(\sum_{j=1}^{d_h} \mathbf{W}^{(2)}_{kj} \mathbf{h}^s_j + \mathbf{b}^{(2)}_k \Big) = \frac{\partial L}{\partial \mathbf{o}^a_k}$$

10. Write down the gradient of the last question in matrix form and define the dimensions of all matrix or vectors involved.
    (What are the dimensions?)
    Take time to understand why the above equalities are the same as the equations of the last question.
    Give the numpy form:

grad_b2 = ...
grad_W2 = ...

Answer

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \frac{\partial L}{\partial \mathbf{o}^a}(h^s)^T$$

$$\frac{\partial L}{\partial \mathbf{b}^{(2)}} = \frac{\partial L}{\partial \mathbf{o}^a}$$

where,
$\mathbf{o}^a$ is a vector of $(m \times 1)$
$\mathbf{h}^s$ is a vector of $(d_h \times 1)$
$\mathbf{W}^{(2)}$ is a matrix of $(m \times d_h)$
$\mathbf{b}^{(2)}$ is a vector of $(m \times 1)$
$\frac{\partial L}{\partial \mathbf{W}^{(2)}}$ is a matrix of $(m \times d_h)$
$\frac{\partial L}{\partial \mathbf{b}^{(2)}}$ is a vector of $(m \times 1)$
$\frac{\partial L}{\partial \mathbf{o}^a}$ is a vector of $(m \times 1)$

numpy form:

```
# for one example
grad_W2 = np.dot(grad_oa, hs.T) # (n_output, n_hidden)
grad_b2 = grad_oa # (n_output, 1)

# for batch_size
grad_W2 = np.dot(grad_oa.T, hs) # (n_output, n_hidden)
grad_b2 = grad_oa # (batch_size, n_output)
grad_b2 = np.mean(grad_b2, axis = 0, keepdims = True).T #(n_output, 1)
```

11. What is the partial derivate of the loss L with respect to the output of the neurons at the hidden layer? Since L depends on $\mathbf{h}_j^s$ only through the activations of the output neurons $\mathbf{o}^a$ the chain rule yields:

$$\frac{\partial L}{\partial \mathbf{h}_j^s} = \sum_{k=1}^{m} \frac{\partial L}{\partial \mathbf{o}_k^a} \frac{\partial \mathbf{o}_k^a}{\partial \mathbf{h}_j^s}$$

Answer

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{h}_j^s} &= \sum_{k=1}^{m} \frac{\partial L}{\partial \mathbf{o}_k^a} \frac{\partial \mathbf{o}_k^a}{\partial \mathbf{h}_j^s} \\
&= \sum_{k=1}^{m} \frac{\partial L}{\partial \mathbf{o}_k^a} \frac{\partial}{\partial \mathbf{h}_j^s}\left(\sum_{j=1}^{d_h} \mathbf{W}_{kj}^{(2)}\mathbf{h}_j^s + \mathbf{b}_k^{(2)}\right) \\
&= \sum_{k=1}^{m} \frac{\partial L}{\partial \mathbf{o}_k^a} \mathbf{W}_{kj}^{(2)}
\end{aligned}$$

12. Write down the gradient of the last question in matrix form and define the dimensions of all matrix or vectors involved.
(What are the dimensions?)

Take time to understand why the above equalities are the same as the equations of the last question.
Give the numpy form: grad_hs = …

Answer

$$\frac{\partial L}{\partial \mathbf{h}^s} = (\mathbf{W}^{(2)})^T \frac{\partial L}{\partial \mathbf{o}^a}$$

where,
$\mathbf{h}^s$ is a vector of $d_h \times 1$
$\mathbf{W}^{(2)}$ is a matrix of $m \times d_h$
$\frac{\partial L}{\partial \mathbf{o}^a}$ is a vector of $m \times 1$
$\frac{\partial L}{\partial \mathbf{h}^s}$ is a vector of $d_h \times 1$

numpy form:

```
# for one example
grad_hs = np.dot(W2.T, grad_oa)

# for batch_size
grad_hs = np.dot(grad_oa, self.W2) # (batch_size, n_hidden)
```

13. What is the partial derivate of the loss L with respect to the activation of the neurons at the hidden layer? Since L depends on the activation $\mathbf{h}^a_j$ only through $\mathbf{h}^s_j$ of this neuron, the chain rule gives:

$$\frac{\partial L}{\partial \mathbf{h}^a_j} = \frac{\partial L}{\partial \mathbf{h}^s_j} \frac{\partial \mathbf{h}^s_j}{\partial \mathbf{h}^a_j}$$

Note $\mathbf{h}^s_j = \mathrm{rect}(\mathbf{h}^a_j)$: the rectifier function is applied elementwise. Start by writing the derivate of the rectifier function $\frac{\partial \mathrm{rect}(z)}{\partial z} = \mathrm{rect}'(z) = \ldots$

Answer

$$\frac{\partial L}{\partial \mathbf{h}^a_j} = \frac{\partial L}{\partial \mathbf{h}^s_j} \frac{\partial \mathbf{h}^s_j}{\partial \mathbf{h}^a_j} = \frac{\partial L}{\partial \mathbf{h}^s_j} \mathbf{1}_{\{\mathbf{h}^a_j > 0\}}(\mathbf{h}^a_j)$$

14. Write down the gradient of the last question in matrix form and define the dimensions of all matrix or vectors involved. Give the numpy form.

Answer

$$\frac{\partial L}{\partial \mathbf{h}^a} = \frac{\partial L}{\partial \mathbf{h}^s} \odot \mathbf{1}_{\{\mathbf{h}^a_j > 0\}}(\mathbf{h}^a_j)$$

where $\odot$ denotes the elementwise multiplication. Each variable in the above equation is a vector of $d_h \times 1$.

numpy form:

```
# for one example
grad_ha = np.multiply(grad_hs, np.maximum(np.sign(ha),0)) # (n_hidden,
 1)

# for batch_size:
grad_ha = np.multiply(grad_hs, np.maximum(np.sign(ha),0)) # (batch_size,
 n_hidden)
```

15. What is the gradient with respect to the parameters $\mathbf{W}^{(1)}$ and $\mathbf{b}^{(1)}$ of the hidden layer?
    Note: same logic as a previous question
    Answer

$$\frac{\partial L}{\partial \mathbf{W}^{(1)}_{ji}} = \frac{\partial L}{\partial \mathbf{h}^a_j}\frac{\partial \mathbf{h}^a_j}{\partial \mathbf{W}^{(1)}_{ji}} = \frac{\partial L}{\partial \mathbf{h}^a_j}\frac{\partial}{\partial \mathbf{W}^{(1)}_{ji}}\Big(\sum_{i=1}^{d} \mathbf{W}^{(1)}_{ji}\mathbf{x}_i + \mathbf{b}^{(1)}_j\Big) = \mathbf{x}_i \frac{\partial L}{\partial \mathbf{h}^a_j}$$

$$\frac{\partial L}{\partial \mathbf{b}^{(1)}_{j}} = \frac{\partial L}{\partial \mathbf{h}^a_j}\frac{\partial}{\partial \mathbf{b}^{(1)}_{j}}\Big(\sum_{i=1}^{d} \mathbf{W}^{(1)}_{ji}\mathbf{x}_i + \mathbf{b}^{(1)}_j\Big) = \frac{\partial L}{\partial \mathbf{h}^a_j}$$

16. Write down the gradient of the last question in matrix form and define the dimensions of all matrix or vectors involved. Give the numpy form.
    Note: same logic as a previous question
    Answer

$$\frac{\partial L}{\partial \mathbf{W}^{(1)}} = \frac{\partial L}{\partial \mathbf{h}^a}\,\mathbf{x}^T$$

$$\frac{\partial L}{\partial \mathbf{b}^{(1)}} = \frac{\partial L}{\partial \mathbf{h}^a}$$

where,
$\mathbf{h}^a$ is a vector of $(d_h \times 1)$
$\mathbf{x}$ is a vector of $(d \times 1)$
$\mathbf{W}^{(1)}$ is a matrix of $(d_h \times d)$
$\mathbf{b}^{(1)}$ is a vector of $(d_h \times 1)$
$\frac{\partial L}{\partial \mathbf{W}^{(1)}}$ is a matrix of $(d_h \times d)$
$\frac{\partial L}{\partial \mathbf{b}^{(1)}}$ is a vector of $(d_h \times 1)$
$\frac{\partial L}{\partial \mathbf{h}^{a}}$ is a vector of $(d_h \times 1)$

numpy form:

```
# for one example
grad_W1 = np.dot(grad_ha, x.T) # (n_hidden, n_input)
grad_b1 = grad_ha # (n_hidden, 1)

# for batch_size
grad_W1 = np.dot(grad_ha.T, X) # (n_hidden, n_input)
grad_b1 = grad_ha # (batch_size, n_hidden)
grad_b1 = np.mean(grad_b1, axis = 0, keepdims = True).T # (n_hidden, 1)
```

17. What are the partial derivates of the loss L with respect to $\mathbf{x}$?
    Note: same logic as a previous question
    Answer

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{x}_i} &= \sum_{j=1}^{d_h} \frac{\partial L}{\partial \mathbf{h}^a_j} \frac{\partial \mathbf{h}^a_j}{\partial \mathbf{x}_i} \\ &= \sum_{j=1}^{d_h} \frac{\partial L}{\partial \mathbf{h}^a_j} \frac{\partial}{\partial \mathbf{x}_i} \Big(\sum_{i=1}^{d} \mathbf{W}^{(1)}_{ji} \mathbf{x}_i + \mathbf{b}^{(1)}_j \Big) \\ &= \sum_{j=1}^{d_h} \frac{\partial L}{\partial \mathbf{h}^a_j} \mathbf{W}^{(1)}_{ji}\end{aligned}$$

Matrix form:

$$\frac{\partial L}{\partial \mathbf{x}} = (\mathbf{W}^{(1)})^T \frac{\partial L}{\partial \mathbf{h}^a}$$

where,
$\mathbf{x}$ is a vector of $d \times 1$
$\mathbf{W}^{(1)}$ is a matrix of $d_h \times d$
$\frac{\partial L}{\partial \mathbf{h}^a}$ is a vector of $d_h \times 1$
$\frac{\partial L}{\partial \mathbf{x}}$ is a vector of $d \times 1$
numpy form:

```
# for one example
grad_x = np.dot(W1.T, grad_ha) # (n_input, 1)

# for batch_size
grad_x = np.dot(grad_ha, self.W1) # (batch_size, n_input)
```

18. We will now consider a **regularized** empirical risk: $\tilde{R} = \hat{R} + \mathcal{L}(\theta)$, where θ is the vector of all the parameters in the network and $\mathcal{L}(\theta)$ describes a scalar penalty as a function of the parameters θ. The penalty is given importance according to a prior preferences for the values of θ. The L_2(quadratic) regularization that penalizes the square norm (norm L_2) of the weights (but not the biases) is more standard, is used in ridge regresion and is sometimes called "weight-decay". Here we shall consider a double regularization L_2 and L_1 which is sometimes named "elastic net" and we will use different **hyperparameters** (positive scalars $\lambda_{11}, \lambda_{12}, \lambda_{21}, \lambda_{22}$) to control the effect of the regularization at each layer

$$\begin{aligned}\mathcal{L}(\theta) &= \mathcal{L}(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}) \\ &= \lambda_{11} \|\mathbf{W}^{(1)}\|_1 + \lambda_{12} \|\mathbf{W}^{(1)}\|_2^2 + \lambda_{21} \|\mathbf{W}^{(2)}\|_1 + \lambda_{22} \|\mathbf{W}^{(2)}\|_2^2 \\ &= \lambda_{11} \Big(\sum_{i,j} |\mathbf{W}^{(1)}_{ij}| \Big) + \lambda_{12} \Big(\sum_{i,j} (\mathbf{W}^{(1)}_{ij})^2 \Big) + \lambda_{21} \Big(\sum_{i,j} |\mathbf{W}^{(2)}_{ij}| \Big) + \lambda_{22} \Big(\sum_{i,j} (\mathbf{W}^{(2)}_{ij})^2 \Big)\end{aligned}$$

We will in fact minimize the regularized risk $\tilde{R}$ instead of $\hat{R}$. How does this change the gradient with respect to the different parameters?

Answer

$$\begin{aligned}\frac{\partial \tilde{R}}{\partial \mathbf{W}^{(1)}_{ij}} &= \frac{\partial \hat{R}}{\partial \mathbf{W}^{(1)}_{ij}} + \lambda_{11} \mathrm{sign}(\mathbf{W}^{(1)}_{ij}) + 2\lambda_{12} \mathbf{W}^{(1)}_{ij} \\ \frac{\partial \tilde{R}}{\partial \mathbf{W}^{(2)}_{ij}} &= \frac{\partial \hat{R}}{\partial \mathbf{W}^{(2)}_{ij}} + \lambda_{21} \mathrm{sign}(\mathbf{W}^{(2)}_{ij}) + 2\lambda_{22} \mathbf{W}^{(2)}_{ij}\end{aligned}$$

$$\frac{\partial \tilde{R}}{\partial \mathbf{b}^{(2)}_i} = \frac{\partial \hat{R}}{\partial \mathbf{b}^{(2)}_i}$$

$$\frac{\partial \tilde{R}}{\partial \mathbf{b}_i^{(1)}} = \frac{\partial \hat{R}}{\partial \mathbf{b}_i^{(1)}},$$

PRACTICAL PART (50 pts) : Neural net-work implementation and experiments

We ask you to implement a neural network where you compute the gradients using the formulas derived in the previous part (including elastic net type regularization). You must not use an existing neural network library, but you must use the derivation of part 2 (with corresponding variable names, etc). Note that you can reuse the general learning algorithm structure that we used in the demos, as well as the functions used to plot the decision functions.

Useful details on implementation : (cutted here. See homework description for detail)

Brief Description of the Answer / Code

We frist went through all the experiments demanded. After knowing what we will challenge, we decided first to do some basic works that all experiments will need, including a complete implementation of our `MLP` class who performs some basic functions for the experiments. After that, for each specific experiment, we first gave a detail explanation about related member functions of `MLP` we used for that experiment, then we wrote extra codes based on the member functions, run it to show the results. So, the whole practical part is divided into two main parts: **Preparation of the experiments** and **Experiments and Results**.

Preparation of the experiments

1. Implementation of two basic functions

We carefully implemented two functions: `my_softmax(z)` and `one_hoe(m, y)` to do the **softmax** prediction and **one_hot** encoding, both of which are implemented based on Matrix operation with one example also supported as an bath_size of 1.

In [1]:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

def my_softmax(z):
    return np.exp(z) / np.sum(np.exp(z), axis=1, keepdims=True)

def one_hot(m, y):
    y = y.reshape(-1, ) # (batch_size, )
    y = y.astype(int)
    batch_size = y.shape[0]
    result = np.zeros((batch_size, m))
    result[np.arange(batch_size), y] = 1
    return result

DEBUG = True
def debug(*kargs):
    global DEBUG
    if DEBUG:
        print(*kargs)
```

2. Implementation of MLP class

Note:

1. Initialization of the network parameters(`W1, W2, b1, b2`) is not performed in the function `__init__` as we might not know the number of features of input and output until we know the X and y.
2. Two set of methods (loop and matrix expression) about the forward and backward propagation are implemented as required by the experiments which will be explained in detail in the answer of related experiments.

The completed implementation of class `MLP` is as follows:

In [2]:

```
class MLP():
    """one hidden layer with RELU and Softmax
    """
    def __init__(self,
                 n_hidden = 2,
                 lambdas = [0, 0, 0, 0],
                 learning_rate = 1e-3,
                 epochs = 200,
                 batch_size = 10,
                 verbose = False,
                 n_input = None,
                 n_output = None,
                 early_stop = False
                ):
        self.n_i = n_input
        self.n_o = n_output
        self.n_h = n_hidden
        self.lambdas = lambdas
        self.lr = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size
        self.early_stop = early_stop
        self.verbose = verbose


    def _init_weights_biases(self, X, y, n_output = None):
        """number of input and output neurons are decided when dataset
        is known, initialization of parameters are performed here.
        Params
            X: input of training data ndarray (sample_size, n_features)
            y: true class labels of X ndarray (sample_size, 1)
            n_output: number of true classes, is None, can be calculated
        from y.
        Returns
            None
        """
        if n_output is None:
            y = y.reshape(-1, )
            self.n_o = int(np.max(y) + 1)
        else:
            self.n_o = n_output
        np.random.seed(0) # for compare
        sample_size, n_input = X.shape
        self.n_i = n_input
        _high = 1 / np.sqrt(self.n_i)
        self.W1 = np.random.uniform(-1 * _high, _high,
                                    (self.n_h, self.n_i))
        self.b1 = np.zeros((self.n_h, 1)) # column vector

        _high = 1 / np.sqrt(self.n_h)
        self.W2 = np.random.uniform(-1 * _high, _high,
                                    (self.n_o, self.n_h))
        self.b2 = np.zeros((self.n_o, 1)) # column vector


    def _init_grads(self):
        """Initialization of gradient of parameters
        Params:None
        Returns
```

```
            grads: zero gradient of all parameters dict
        """
        grad_W1 = np.zeros_like(self.W1)
        grad_W2 = np.zeros_like(self.W2)
        grad_b1 = np.zeros_like(self.b1)
        grad_b2 = np.zeros_like(self.b2)
        grads = {"grad_W1":grad_W1, "grad_W2":grad_W2,
                 "grad_b1":grad_b1, "grad_b2":grad_b2,
                }
        return grads


    def _fprop_one_example(self, x):
        """forward propagation of the network using one example.
        Params
            x: example ndarray (n_features, 1)
        Returns
            cache: a dict variable including computation results used for
        back propagation.
        """
        x = x.reshape(-1, 1) # column vector (n_input, 1)
        ha = np.dot(self.W1, x) + self.b1 # (n_hidden, 1)
        hs = np.maximum(ha, 0) # (n_hidden, 1)

        oa = np.dot(self.W2, hs) + self.b2 # (n_output, 1)
        os = my_softmax(oa.T).T # (n_output, 1) softmax

        cache = {"ha": ha, "hs": hs, "oa": oa, "os": os, "x":x}
        return cache


    def _bprop_one_example(self, cache, y):
        """backward propagation of the network using one example.
        Params
            cache: a dict variable from the output of forward propagation.
            y: int, true class index of the example
        Returns
            grads: gradient of the parameters of the network.
        """
        ha = cache["ha"] # (n_hidden, 1)
        hs = cache["hs"] # (n_hidden, 1)
        oa = cache["oa"] # (n_output, 1)
        os = cache["os"] # (n_output, 1)
        x = cache["x"]   # (n_input, 1)
        grad_oa = np.copy(os) # (n_output, 1)
        np.put(grad_oa, y, grad_oa[y,0]-1)

        grad_W2 = np.dot(grad_oa, hs.T) # (n_output, n_hidden)
        grad_b2 = grad_oa # (n_output, 1)

        grad_hs = np.dot(self.W2.T, grad_oa) # (n_hidden, 1)
        grad_ha = np.multiply(grad_hs, np.maximum(np.sign(ha),0))
        # (n_hidden, 1)

        grad_W1 = np.dot(grad_ha, x.T) # (n_hidden, n_input)
        grad_b1 = grad_ha # (n_hidden, 1)

        # grad_x = np.dot(self.W1.T, grad_ha) # (n_input, 1)
        lambdas = self.lambdas
        if not lambdas[0] == 0:
            grad_W1 += lambdas[0] * np.sign(self.W1)
```

```
        if not lambdas[1] == 0:
            grad_W1 += 2 * lambdas[1] * self.W1
        if not lambdas[2] == 0:
            grad_W2 += lambdas[2] * np.sign(self.W2)
        if not lambdas[3] == 0:
            grad_W2 += 2 * lambdas[3] * self.W2

        grads = {"grad_W1":grad_W1, "grad_W2":grad_W2,
                 "grad_b1":grad_b1, "grad_b2":grad_b2,
                 #"grad_hs":grad_hs, "grad_ha":grad_ha,
                }
        return grads


    def fit_non_matrix_expression(self, X, y):
        """fit X to y without using matrix expression. parameters
        are updated after every batch_size. In each batch_size,
        grads are accumulated by a looping over each example.
        Params
            X: training X ndarray (batch_size, n_features)
            y: training labels ndarray (batch_size, )
        Returns
            losses: list of loss of each epoch [float]
            errors: list of error rate of each epoch [float]
        """
        self._init_weights_biases(X, y)

        sample_size, _ = X.shape
        batches = int(np.ceil(sample_size / self.batch_size))
        losses, errors = [], [] # store loss and error each epoch
        for epoch in range(self.epochs):
            loss, err = 0.0, 0.0
            for j in range(batches):
                b_start = j * self.batch_size
                b_end = min(sample_size, (j+1) * self.batch_size)
                batch_X = X[b_start:b_end, :]
                batch_y = y[b_start:b_end]
                grads = self._init_grads()
                for k in range(b_end - b_start):
                    cache = self._fprop_one_example(batch_X[k])
                    grads_tmp = self._bprop_one_example(cache, batch_y[k])
                    for key in grads:
                        # compute mean grads[key]
                        grads[key] += (grads_tmp[key] - grads[key])/(k + 1)
                    loss += self._compute_one_loss(cache["os"], batch_y[k])
                self._update_params(grads)
            # end of mini_batch

            error = self.compute_error(self.predict(X), y)
            errors.append(error)
            loss /= sample_size
            loss += self._compute_regular_loss()
            losses.append(loss)
        # end of one epoch
        return losses, errors


    def _fprop(self, X):
        """
        forward propagation using Matrix expression
        params
```

```
            X: input data (batch_size, n_input)
        returns
            os: output of the network (batch_size, n_output)
        """
        ha = np.dot(X, self.W1.T) + self.b1.T # (batch_size, n_hidden)
        hs = np.maximum(ha, 0) # relu (batch_size, n_hidden)

        oa = np.dot(hs, self.W2.T) + self.b2.T # (batch_size, n_output)
        os = my_softmax(oa) # (batch_size, n_output)

        cache = {"ha": ha, "hs": hs, "oa": oa, "os": os, "X":X}
        return cache


    def _bprop(self, cache, y):
        """
        backward propagation using Matrix expression
        params
            os: output of network (batch_size, n_output)
            y: true label class (batch_size, 1)
        returns:
            grads: gradients( grad_W1, grad_W2, grad_b1, grad_b2)
        """
        ha = cache["ha"] # (batch_size, n_hidden)
        hs = cache["hs"] # (batch_size, n_hidden)
        oa = cache["oa"] # (batch_size, n_output)
        os = cache["os"] # (batch_size, n_output)
        X = cache["X"]   # (batch_size, n_input)

        batch_size, n_o = os.shape # (batch_size, n_output)
        grad_oa = os - one_hot(n_o, y) #(batch_size, n_output)
        # grad_oa = np.mean(grad_oa, axis = 0, keepdims = True).T
        # (n_output, 1)

        grad_W2 = np.dot(grad_oa.T, hs) # (n_output, n_hidden)
        grad_b2 = grad_oa # (batch_size, n_output)

        grad_hs = np.dot(grad_oa, self.W2) # (batch_size, n_hidden)
        grad_ha = np.multiply(grad_hs, np.maximum(np.sign(ha),0))
        # (batch_size, n_hidden)

        grad_W1 = np.dot(grad_ha.T, X) # (n_hidden, n_input)
        grad_b1 = grad_ha # (batch_size, n_hidden)

        # grad_x = np.dot(grad_ha, self.W1) # (batch_size, d)

        # mean grad
        grad_W2 /= batch_size
        grad_W1 /= batch_size
        grad_b2 = np.mean(grad_b2, axis = 0, keepdims = True).T
        grad_b1 = np.mean(grad_b1, axis = 0, keepdims = True).T

        lambdas = self.lambdas
        if not lambdas[0] == 0:
            grad_W1 += lambdas[0] * np.sign(self.W1)
        if not lambdas[1] == 0:
            grad_W1 += 2 * lambdas[1] * self.W1
        if not lambdas[2] == 0:
            grad_W2 += lambdas[2] * np.sign(self.W2)
        if not lambdas[3] == 0:
            grad_W2 += 2 * lambdas[3] * self.W2
```

```

        grads = {"grad_W1":grad_W1, "grad_W2":grad_W2,
                 "grad_b1":grad_b1, "grad_b2":grad_b2,
                 #"grad_hs":grad_hs, "grad_ha":grad_ha,
                }
        return grads


    def fit(self, X, y):
        """fit X to y. Matrix operation
        Params
            X: training X ndarray (sample_size, n_features)
            y: training labels ndarray (sample_size, )
        Returns
            losses: list of loss of each epoch [float]
            errors: list of error rate of each epoch [float]
        """
        self._init_weights_biases(X, y)
        sample_size, _ = X.shape
        batches = int(np.ceil(sample_size / self.batch_size))
        losses, errors = [], []

        for epoch in range(self.epochs):
            for j in range(batches):
                b_start = j * self.batch_size
                b_end = min(sample_size, (j+1) * self.batch_size)
                batch_X = X[b_start:b_end, :]
                batch_y = y[b_start:b_end]
                cache = self._fprop(batch_X)
                grads = self._bprop(cache, batch_y)
                self._update_params(grads)
            # end batch_size
            os = self._fprop(X)["os"]
            loss = self.compute_empirical_risk(os, y)
            losses.append(loss)

            oy = np.argmax(os, axis = 1)
            error = self.compute_error(oy, y)
            errors.append(error)
        # end epoch
        return losses, errors


    def predict_probs(self, X):
        """predict probabilities (os) of given data
        Params
            X: examples. ndarray (certain_size, n_input)
        Returns
            os: probabilities of X belong to every class.
        ndarray (certain_size, n_output)
        """
        return self._fprop(X)["os"]


    def predict(self, X):
        """predict class labels of given data
        Params
            X: examples ndarray (certain_size, n_input)
        Returns
            y_output: predicted label of given data. ndarray
        (certain_size, )
```

```
        """
        os = self.predict_probs(X)
        return np.argmax(os, axis = 1)


    def _backup_params(self):
        """save current network parameters
        Params: None
        Returns:
            parameters saved. dict.
        """
        old_params = {"W1":np.copy(self.W1),
                      "W2":np.copy(self.W2),
                      "b1":np.copy(self.b1),
                      "b2":np.copy(self.b2)
                          }
        return old_params


    def _restore_params(self, old_params = None):
        """restore network parameters by change network's parameters
        Params
            old_params: parameters restored
        Returns:
            None
        """
        if old_params is None:
            return
        self.W1 = old_params["W1"]
        self.W2 = old_params["W2"]
        self.b1 = old_params["b1"]
        self.b2 = old_params["b2"]


    def _update_params(self, grads, learnig_rate = None):
        """update network parameters using given gradients and larning
        rate
        Params
            grads: gradients of parameters. dict
            learning_rate: learning rate. float
        Returns
            None
        """
        lr = learnig_rate
        if lr is None:
            lr = self.lr
        self.W1 -= lr * grads["grad_W1"]
        self.W2 -= lr * grads["grad_W2"]
        self.b1 -= lr * grads["grad_b1"]
        self.b2 -= lr * grads["grad_b2"]


    def compute_error(self, oy, y):
        """compute mis classification rate
        Params
            oy: output of a classifier ndarray (size, )
            y: true class labels ndarray (size, )
        Returns
            error: misclassified error. float
        """
        return  float(1 - np.sum(oy == y) / len(oy))
```

```


    def _compute_one_loss(self, os, y):
        """loss of one example
        Params
            os: column vector output (n_output, 1)
            y: scalar int
        Returns loss of one example. float
        """
        os = os.reshape(-1, 1)
        os_y = os[y, 0] + 1e-100 # avoid np.log(0)
        return float(-np.log(os_y))


    def _compute_regular_loss(self):
        """regular loss
        Params
            None
        Returns
            None
        """
        loss = 0.0
        lambdas = self.lambdas
        if not lambdas[0] == 0:
              loss += lambdas[0] * np.sum(np.abs(self.W1))
        if not lambdas[1] == 0:
              loss += lambdas[1] * np.sum(np.power(self.W1, 2))
        if not lambdas[2] == 0:
              loss += lambdas[2] * np.sum(np.abs(self.W2))
        if not lambdas[3] == 0:
              loss += lambdas[3] * np.sum(np.power(self.W2, 2))
        return float(loss)


    def compute_empirical_risk(self, os, y):
        """
        compute $\tilde R = \hat R + \mathcal L(\theta)$
        params
            os: output of network (sample_size, n_output)
            y: true label (sample_size, class_index)
        """
        batch_size, n_output = os.shape
        y = y.reshape(-1, )
        y = y.astype(int)
        os_y = os[np.arange(batch_size), y]
        loss = np.mean(-np.log(os_y)) # might cause warning from np.log()
        loss += self._compute_regular_loss()
        return float(loss)
```

3. Loading two datasets: Circle dataset and Fashion MNIST dataset

Two datasets will be loaded by following codes, each of which will be shuffled and then divided into 3 parts for training, validating, and testing with a proportion about 7:2:2 and 5:1:1 respectively.

See the following codes for details.

In [3]:

```
from time import time
from datetime import datetime
from tqdm import tqdm
import utils.mnist_reader as mnist_reader

# load circles data
data = np.loadtxt(open('circles.txt','r'))
np.random.shuffle(data)
X = data[:,0:2]
y = data[:,2].reshape(-1)
y = y.astype(int)

total_size = len(X) # 1100
train_size = 700
valid_size = 200
test_size = total_size - train_size - valid_size

# split dataset to three parts: train, valid, test
train_X = X[0: train_size,:]
train_y = y[0: train_size]
valid_X = X[train_size: train_size + test_size,:]
valid_y = y[train_size: train_size + test_size]
test_X = X[train_size + test_size:,:]
test_y = y[train_size + test_size:]

debug(train_X.shape, train_y.shape)
debug(valid_X.shape, valid_y.shape)
debug(test_X.shape, test_y.shape)

# load Fashion MNIST dataset
fshn_train_X0, fshn_train_y0 = mnist_reader.load_mnist('data/fashion',
                                                       kind='train')
fshn_test_X, fshn_test_y = mnist_reader.load_mnist('data/fashion',
                                                   kind='t10k')
#plt.scatter(X[:,0], X[:,1], c = y)
fshn_train_X = fshn_train_X0[0:50000,:]
fshn_train_y = fshn_train_y0[0:50000]
fshn_valid_X = fshn_train_X0[50000:,:]
fshn_valid_y = fshn_train_y0[50000:]

debug(fshn_train_X.shape, fshn_train_y.shape)
debug(fshn_valid_X.shape, fshn_valid_y.shape)
debug(fshn_test_X.shape, fshn_test_y.shape)
```

```
(700, 2) (700,)
(200, 2) (200,)
(200, 2) (200,)
(50000, 784) (50000,)
(10000, 784) (10000,)
(10000, 784) (10000,)
```

Experiments and Results

1.

As a beginning, start with an implementation that computes the gradients for a single example, and check that the gradient is correct using the finite difference method described above.

Answer / Code

The gradient computed by forward and backward propagation relies on the methods in `MLP` class:

```
cache = mlp._fprop_one_example(x)
grads1 = mlp._bprop_one_example(cache, c)
```

The first function `_fprop_one_example(x)` receives an example `x` , reshapes it to a column vector, computes the hidden layer activation (ha), hidden layer output (ho), output layer activation (oa) and output layer output (os) in order. All these variables are cached into a dictionary `cache` as a return variable for the use of back propagation. The codes are are as follows:

```
def _fprop_one_example(self, x):
    x = x.reshape(-1, 1) # to column vector
    ha = np.dot(self.W1, x) + self.b1
    hs = np.maximum(ha, 0)

    oa = np.dot(self.W2, hs) + self.b2
    os = my_softmax(oa.T).T # to row vector for softmax

    cache = {"ha": ha, "hs": hs, "oa": oa, "os": os, "x":x}
    return cache
```

The second function `mlp._bprop_one_example(cache, c)` receives the return value of the first function (`cache`) and the true class label of the example (`c`). By the equations derived in theory part and the following codes, the gradients of the netwrok parameters are then claculated:

```
    def _bprop_one_example(self, cache, y):
        ha = cache["ha"] # (n_hidden, 1)
        hs = cache["hs"] # (n_hidden, 1)
        oa = cache["oa"] # (n_output, 1)
        os = cache["os"] # (n_output, 1)
        x = cache["x"]   # (n_input, 1)
        grad_oa = np.copy(os)
        np.put(grad_oa, y, grad_oa[y,0]-1)

        grad_W2 = np.dot(grad_oa, hs.T) # (n_output, n_hidden)
        grad_b2 = grad_oa # (n_output, 1)

        grad_hs = np.dot(self.W2.T, grad_oa) # (n_hidden, 1)
        grad_ha = np.multiply(grad_hs, np.maximum(np.sign(ha),0)) # (n_hidden,
 1)

        grad_W1 = np.dot(grad_ha, x.T) # (n_hidden, n_input)
        grad_b1 = grad_ha # (n_hidden, 1)

        # grad_x = np.dot(self.W1.T, grad_ha) # (n_input, 1)
        lambdas = self.lambdas
        if not lambdas[0] == 0:
            grad_W1 += lambdas[0] * np.sign(self.W1)
        if not lambdas[1] == 0:
            grad_W1 += 2 * lambdas[1] * self.W1
        if not lambdas[2] == 0:
            grad_W2 += lambdas[2] * np.sign(self.W2)
        if not lambdas[3] == 0:
            grad_W2 += 2 * lambdas[3] * self.W2

        grads = {"grad_W1":grad_W1, "grad_W2":grad_W2,
                 "grad_b1":grad_b1, "grad_b2":grad_b2,
                 #"grad_hs":grad_hs, "grad_ha":grad_ha,
                }
        return grads
```

Two other functions `is_grad_equal` and `is_grads_equal` are implemented to check if the grads by finite difference method and by back propagation of the network are equal. Only when two values of the gradient with respect to each parameter (`W1, W2, b1, b2`) from two methods are almost equal (difference between 0.01), will the functions give a `True` result and print "All grads are equal".

See the following codes for details.

In [4]:

```
def grad_finite_diff_one_example(mlp, x, c, epsilon = 1e-5):
    grads = mlp._init_grads()
    params = {"grad_W1":mlp.W1, "grad_W2":mlp.W2,
              "grad_b1":mlp.b1, "grad_b2":mlp.b2,
             }
    for grad_key in grads:
        grad = grads[grad_key]
        param = params[grad_key]
        m, n = grad.shape
        for i in range(m):
            for j in range(n):
                old_value = param[i, j]

                value_plus = old_value + epsilon
                param[i, j] = value_plus
                cache = mlp._fprop_one_example(x)
                loss_plus = mlp._compute_one_loss(cache["os"], c)
                loss_plus += mlp._compute_regular_loss()

                value_minor = old_value - epsilon
                param[i, j] = value_minor
                cache = mlp._fprop_one_example(x)
                loss_minor = mlp._compute_one_loss(cache["os"], c)
                loss_minor += mlp._compute_regular_loss()

                grad[i, j] = (loss_plus - loss_minor)/(2.0 * epsilon)
                param[i, j] = old_value
    return grads


def is_grad_equal(grad1, grad2, epsilon = 1e-6):
    """check if all element values of two grad array are equal"""
    if not grad1.shape == grad2.shape:
        return False
    m, n = grad1.shape
    for i in range(m):
        for j in range(n):
            if abs(grad1[i,j] - grad2[i,j]) > epsilon:
                return False
    return True


def is_grads_equal(grads1, grads2, epsilon = 1e-6):
    for grad_key in grads1:
        if not is_grad_equal(grads1[grad_key], grads2[grad_key], epsilon):
            print("Failure in gradient check of param: {}".format(grad_key))
            return False
    print("All gradients are equal")
    return True
```

We randomly chose an example from `circle` dataset, calculated gradients from two methods respectively, and checked them are equal by the following codes:

In [5]:

```
# create a mlp with certain neurons in hidden layer
mlp = MLP(n_hidden = 2)
mlp._init_weights_biases(X, y) # initialize weights and bias

for _ in range(1): # check times
    index = np.random.randint(0, len(X)) # randomly select an example
    x, c = X[index, :], y[index]

    cache = mlp._fprop_one_example(x)
    grads1 = mlp._bprop_one_example(cache, c)

    grads2 = grad_finite_diff_one_example(mlp, x, c)
    is_grads_equal(grads1, grads2)

```

```
All gradients are equal
```

2.

Display the gradients for both methods (direct computation and finite difference) for a small network (e.g. d = 2 and d_h = 2) with random weights and for a single example.

Answer / Code

The following codes printed the values of the gradients.

In [6]:

```
def print_grads(grads1, grads2, desript=["",""]):
    for grad_key in grads1:
        print(grad_key + " by " + desript[0] + ":")
        print(grads1[grad_key])
        print(grad_key + " by " + desript[1] + ":")
        print(grads2[grad_key])
        print()

print_grads(grads1, grads2, ["back propagation",
                             "finite difference method"])
```

```
grad_b2 by back propagation:
[[-0.51004331]
 [ 0.51004331]]
grad_b2 by finite difference method:
[[-0.51004331]
 [ 0.51004331]]

grad_W2 by back propagation:
[[ 0.         -0.05893403]
 [ 0.          0.05893403]]
grad_W2 by finite difference method:
[[ 0.         -0.05893403]
 [ 0.          0.05893403]]

grad_W1 by back propagation:
[[ 0.          0.        ]
 [ 0.16704075 -0.05960002]]
grad_W1 by finite difference method:
[[ 0.          0.        ]
 [ 0.16704075 -0.05960002]]

grad_b1 by back propagation:
[[0.        ]
 [0.17735494]]
grad_b1 by finite difference method:
[[0.        ]
 [0.17735494]]
```

3.

Add a hyperparameter for the minibatch size K to allow compute the gradients on a minibatch of K examples (in a matrix), by looping over the K examples (this is a small addition to your previous code).

Answer / Code

In this experiment, two member function of class `MLP` were used:

```
_compute_one_loss(self, os, y)
_compute_regular_loss(self)
```

The loss of one example is calculated by the following equation:

$$L(\mathbf{x}, y) = -log\ \mathbf{o}^s_y(\mathbf{x})$$

which we have already seen in question 2.4 of theoretical part. The corresponding codes are:

```
    def _compute_one_loss(self, os, y):
        """loss of one example
        params
            os: column vector output (n_output, 1)
            y: scalar int
        """
        os = os.reshape(-1, 1)
        os_y = os[y, 0] + 1e-100 # avoid np.log(0)
        return float(-np.log(os_y))
```

The loss of regularization is also a part of the loss on either one example or batch examples, and they can be derived by following codes:

```
    def _compute_regular_loss(self):
        loss = 0.0
        lambdas = self.lambdas
        if not lambdas[0] == 0:
              loss += lambdas[0] * np.sum(np.abs(self.W1))
        if not lambdas[1] == 0:
              loss += lambdas[1] * np.sum(np.power(self.W1, 2))
        if not lambdas[2] == 0:
              loss += lambdas[2] * np.sum(np.abs(self.W2))
        if not lambdas[3] == 0:
              loss += lambdas[3] * np.sum(np.power(self.W2, 2))
        return float(loss)
```

For the method of back propagation, the gradient with respect to a parameter on a batch size of K examples can be considered as the mean of the gradient on each sample in the batch, which is implemented in the function: `grad_back_prop_batch(mlp, X, y)` .

For the method of finite difference method, the gradient relies on the tiny change of average loss by the tiny change of the parameter. So, the key is to compute the average loss on the minibatch. See the implementation of the function `grad_finite_diff_batch(mlp, X, y, epsilon = 1e-5)` for details.

In [7]:

```
def grad_back_prop_batch(mlp, X, y):
    batch_size, n_input = X.shape
    y = y.reshape(-1, )
    grads1 = mlp._init_grads()
    for i in range(batch_size):
        x, c = X[i,:], y[i]
        cache = mlp._fprop_one_example(x)
        grads = mlp._bprop_one_example(cache, c)
        for key in grads1:
            # compute mean grads1[key]
            grads1[key] += (grads[key] - grads1[key])/(i + 1)
    return grads1


def grad_finite_diff_batch(mlp, X, y, epsilon = 1e-5):
    grads = mlp._init_grads()
    params = {"grad_W1":mlp.W1, "grad_W2":mlp.W2,
              "grad_b1":mlp.b1, "grad_b2":mlp.b2,
             }
    batch_size, _ = X.shape
    y = y.reshape(-1, )
    y = y.astype(int)
    for grad_key in grads:
        grad = grads[grad_key]
        param = params[grad_key]
        m, n = grad.shape
        for i in range(m):
            for j in range(n):
                old_value = param[i, j]

                value_plus = old_value + epsilon
                param[i, j] = value_plus
                loss_plus = 0
                for k in range(batch_size):
                    cache = mlp._fprop_one_example(X[k,:])
                    loss_plus += mlp._compute_one_loss(cache["os"], y[k])
                loss_plus /= batch_size
                loss_plus += mlp._compute_regular_loss()


                value_minor = old_value - epsilon
                param[i, j] = value_minor
                loss_minor = 0
                for k in range(batch_size):
                    cache = mlp._fprop_one_example(X[k,:])
                    loss_minor += mlp._compute_one_loss(cache["os"], y[k])
                loss_minor /= batch_size
                loss_minor += mlp._compute_regular_loss()
                grad[i, j] = (loss_plus - loss_minor)/(2.0 * epsilon)
                param[i, j] = old_value
    return grads
```

The graients from two methods should be equal. Again, we can check them by the functions used in the experiment 1.

Luckly, We got the same gradients.

In [8]:

```
# create a mlp with certain neurons in hidden layer
mlp = MLP(n_hidden = 2)
mlp._init_weights_biases(X, y) # initialize weights and bias
batch_size = 10
start = np.random.randint(0, len(X)-batch_size)
batch_X = X[start:start + batch_size,:]
batch_y = y[start:start + batch_size]

grads1 = grad_back_prop_batch(mlp, batch_X, batch_y)
grads2 = grad_finite_diff_batch(mlp, batch_X, batch_y)
is_grads_equal(grads1, grads2)
```

```
All gradients are equal
```

Out[8]:

```
True
```

4.

Display the gradients for both methods (direct computation and finite difference) for a small network (e.g. $d = 2$ and $d_h = 2$) with random weights and for a minibatch with 10 examples (you can use examples from both classes from the two circles dataset).

Answers / Code

Just use the function implemented in experiment 2 to display the two gradients.

In [9]:

```
print_grads(grads1, grads2, ["back propagation", "finite difference method"])
```

```
grad_b2 by back propagation:
[[-0.2040551]
 [ 0.2040551]]
grad_b2 by finite difference method:
[[-0.2040551]
 [ 0.2040551]]

grad_W2 by back propagation:
[[ 0.0170263  -0.00454214]
 [-0.0170263   0.00454214]]
grad_W2 by finite difference method:
[[ 0.0170263  -0.00454214]
 [-0.0170263   0.00454214]]

grad_W1 by back propagation:
[[-9.75817411e-05 -1.08023074e-03]
 [ 2.85217777e-02 -4.04200060e-02]]
grad_W1 by finite difference method:
[[-9.75817427e-05 -1.08023074e-03]
 [ 2.85217777e-02 -4.04200060e-02]]

grad_b1 by back propagation:
[[-0.0009252 ]
 [ 0.01873617]]
grad_b1 by finite difference method:
[[-0.0009252 ]
 [ 0.01873617]]
```

5.

Train your neural network using gradient descent on the two circles dataset. Plot the decision regions for several different values of the hyperparameters (weight decay, number of hidden units, early stopping) so as to illustrate their effect on the capacity of the model.

Answer / Code

We specifically implemented the following method to train our network on different hyperparameters and plot the corresponding decision regions.

As to early stopping, we set the tolerant epoch times `n_patience = 20` . When error rate on validate dataset starts to increase, we observe the consequesnt `n_patience` number of epochs before stopping training.

Several member functions of class `MLP` are used in this experiments:

```
_init_weights_biases(train_X, train_y)
_fprop_one_example(batch_X[k])
_bprop_one_example(cache, batch_y[k])
_update_params(grads)
_compute_regular_loss()
predict_probs(train_X)
compute_error(train_oy, train_y)
compute_empirical_risk(valid_os, valid_y)
_backup_params()
_restore_params((best_params))
```

The purposes of these functions are just as simple as the names of the functions tell, and they are also easy to implement. No further explanation for these functions. See the implementation of the class `MLP` for details.

In [10]:

```
def train_by_looping(train_X, train_y, valid_X, valid_y,
           n_hidden = 8, early_stop = False, weight_decay = [0.0, 0.0, 0.0, 0.0]
           learning_rate = 1e-3, epochs = 1000, batch_size = 10
          ):
    mlp = MLP(n_hidden = n_hidden,
              learning_rate = learning_rate,
              epochs = epochs,
              batch_size = batch_size,
              early_stop = early_stop,
              lambdas = weight_decay)

    mlp._init_weights_biases(train_X, train_y)

    # for early stopping
    best_params = None # params that performs best in valid set
    n_patientce = 200 # maximun numbers of epoch tolerated when mlp
                      # with current params doesn't perform better than
                      # with best_params.
    n_epochs_bad = 0 # epochs which passed yet didn't bring better params
    best_valid_loss = float('inf')
    best_valid_error = float('inf')


    sample_size, _ = train_X.shape
    batches = int(np.ceil(sample_size / mlp.batch_size))
    losses = [[], []] # losses[0] is for trainng, 1 for valid
    errors = [[], []] # [0] for training, 1 for valid

    for epoch in range(mlp.epochs):
        train_loss = 0.0
        for j in range(batches):
            b_start = j * mlp.batch_size
            b_end = min(sample_size, (j+1) * mlp.batch_size)
            batch_X = X[b_start:b_end, :]
            batch_y = y[b_start:b_end]
            grads = mlp._init_grads()
            for k in range(b_end - b_start):
                cache = mlp._fprop_one_example(batch_X[k])
                grads_tmp = mlp._bprop_one_example(cache, batch_y[k])
                for key in grads:
                    # compute mean grads[key]
                    grads[key] += (grads_tmp[key] - grads[key])/(k + 1)
                train_loss += mlp._compute_one_loss(cache["os"], batch_y[k])
                mlp._update_params(grads)
            # end of mini_batch

        train_loss /= sample_size
        train_loss += mlp._compute_regular_loss()
        losses[0].append(train_loss)

        if early_stop == True:
            train_os = mlp.predict_probs(train_X)
            train_oy = np.argmax(train_os, axis = 1)
            train_error = mlp.compute_error(train_oy, train_y)
            errors[0].append(train_error)

            valid_os = mlp.predict_probs(valid_X)
            valid_oy = np.argmax(valid_os, axis = 1)
            valid_loss = mlp.compute_empirical_risk(valid_os, valid_y)
```

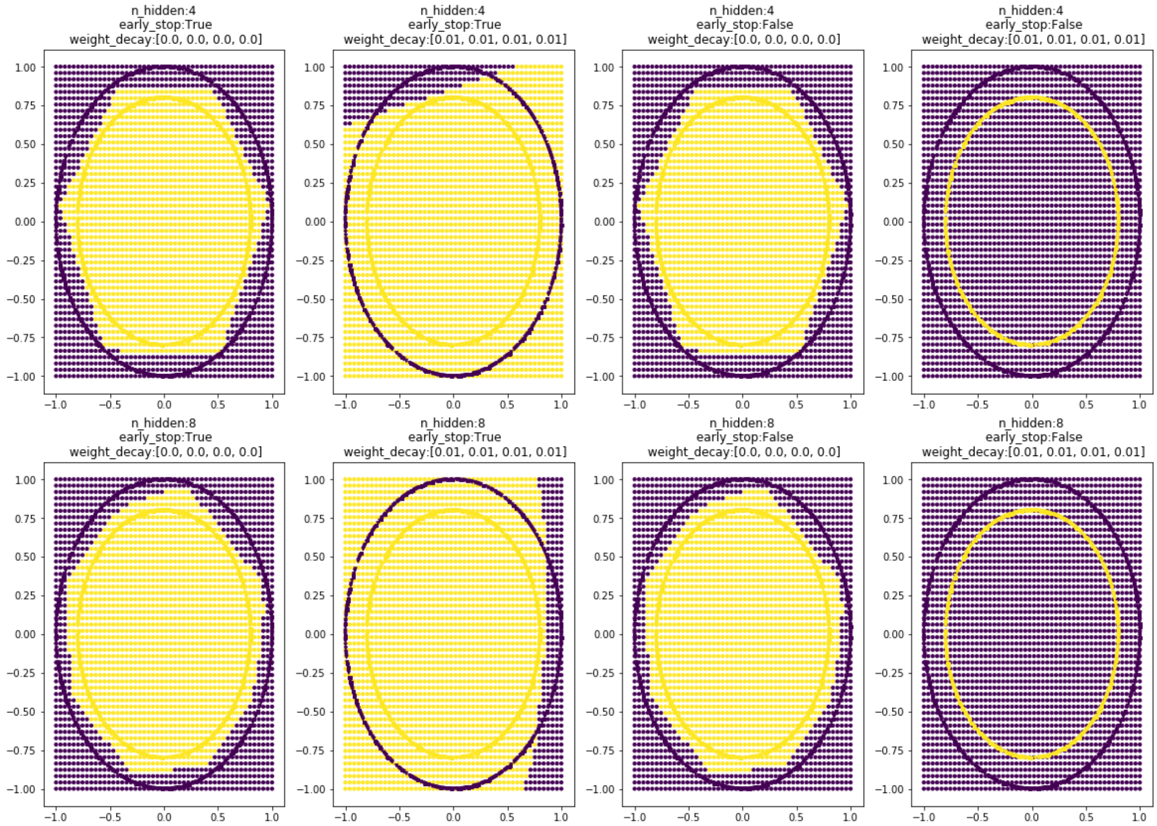
```
            losses[1].append(valid_loss)
            valid_error = mlp.compute_error(valid_oy, valid_y)
            errors[1].append(valid_error)

            if epoch == 0: # first epoch completed
                best_params = mlp._backup_params()
                best_valid_error = valid_error
            else:
                if valid_error > best_valid_error: #
                    n_epochs_bad += 1
                    if n_epochs_bad == n_patientce: # stop
                        mlp._restore_params(best_params)
                        return mlp, losses, errors, best_valid_error
                else: # loss is decressing
                    best_params = mlp._backup_params()
                    best_valid_error = valid_error
                    n_epochs_bad = 0
    # end of one epoch
    return mlp, losses, errors, best_valid_error
```

We trained the network using different combination of the following values of the hyperparameters.

1. number of hidden layers: 4 and 8
2. early stop: True and False
3. weight decays(λs): [0.0, 0.0, 0.0, 0.0] and [0.01, 0.01, 0.01, 0.01]

In [11]:

```
# hyper parameters selected.
n_hiddens = [4, 8]
early_stops = [True, False]
weight_decays = [[0.0]*4, [0.01]*4]

train_valid_X = np.vstack((train_X, valid_X))
train_valid_y = np.vstack((train_y.reshape(-1,1), valid_y.reshape(-1,1)))
train_valid = np.hstack((train_valid_X, train_valid_y))

min_x1, max_x1 = min(train_valid[:,0]), max(train_valid[:,0])
min_x2, max_x2 = min(train_valid[:,1]), max(train_valid[:,1])

n_points = 50
grid_x1 = np.linspace(min_x1, max_x1, num = n_points)
grid_x2 = np.linspace(min_x2, max_x2, num = n_points)

X1, X2 = np.meshgrid(grid_x1, grid_x2)
grid_data = np.hstack((X1.reshape(-1, 1), X2.reshape(-1, 1)))
len_i = len(n_hiddens)
len_j = len(early_stops)
len_k = len(weight_decays)

plt.figure(figsize=(20, 14))
for i in range(len_i):
    for j in range(len_j):
        for k in range(len_k):
            mlp, _, _, _ = train_by_looping(train_X, train_y,
                                             valid_X, valid_y,
                                             n_hiddens[i],
                                             early_stops[j],
                                             weight_decays[k],
                                             epochs = 2000
                                            )
            predicts = mlp.predict(grid_data)
            plt.subplot(len_i, len_j * len_k,
                        i * (len_j * len_k) + (j * len_k) + k + 1)
            plt.scatter(grid_data[:,0], grid_data[:,1], c = predicts, s=10)
            #The training points
            plt.scatter(train_X[:,0], train_X[:,1], c = train_y,
                        marker = 'v', s=10)
            # The test points
            plt.scatter(valid_X[:,0], valid_X[:,1], c = valid_y,
                        marker = 's', s=10)
            plt.title("n_hidden:{}\n early_stop:{}\n weight_decay:{}"
                      .format(n_hiddens[i], early_stops[j], weight_decays[k]))
            # debug("{}".format(i * (len_j * len_k) + (j * len_k) + k + 1))
#plt.subplots_adjust()
plt.show()
```



6.

As a second step, copy your existing implementation to modify it to a new implementation that will use matrix calculus (instead of a loop) on batches of size K to improve efficiency. **Take the matrix expressions in numpy derived in the first part, and adapt them for a minibatch of size K. Show in your report what you have modified (describe the former and new expressions with the shapes of each matrices).**

Answer / Code

As we have seen before, forward and backward propagation computation are implemented using loop over single example in the following two functions:

```
_fprop_one_example(self, x)
_bprop_one_example(self, cache, y)
```

For this experiment, the above functions are accordingly implemented in matrix expressions in:

```
_fprop(self, X)
_bprop(self, cache, y)
```

The matrix expression are shown as follows with comments on each line shows the shape of the corresponding variable.

forward propagation on one example

```
x = x.reshape(-1, 1)                 # (n_input, 1)
ha = np.dot(self.W1, x) + self.b1    # (n_hidden, 1)
hs = np.maximum(ha, 0)               # (n_hidden, 1)
oa = np.dot(self.W2, hs) + self.b2   # (n_output, 1)
os = my_softmax(oa.T).T              # (n_output, 1)
```

backward propagation on one example

```
grad_oa = np.copy(os)                # (n_output, 1)
np.put(grad_oa, y, grad_oa[y,0]-1)
grad_W2 = np.dot(grad_oa, hs.T)      # (n_output, n_hidden)
grad_b2 = grad_oa # (n_output, 1)    # (n_output, 1)
grad_hs = np.dot(self.W2.T, grad_oa) # (n_hidden, 1)
grad_ha = np.multiply(grad_hs, np.maximum(np.sign(ha), 0)) # (n_hidden, 1)
grad_W1 = np.dot(grad_ha, x.T)       # (n_hidden, n_input)
grad_b1 = grad_ha                    # (n_hidden, 1)
```

forward propagation on batch examples

```
ha = np.dot(X, self.W1.T) + self.b1.T  # (batch_size, n_hidden)
hs = np.maximum(ha, 0)  # relu         # (batch_size, n_hidden)
oa = np.dot(hs, self.W2.T) + self.b2.T # (batch_size, n_output)
os = my_softmax(oa)                    # (batch_size, n_output)
```

backward propagation on batch examples

```
batch_size, n_o = os.shape      # (batch_size, n_output)
grad_oa = os - one_hot(n_o, y)  #(batch_size, n_output)
# grad_oa = np.mean(grad_oa, axis = 0, keepdims = True).T
                                # (n_output, 1)

grad_W2 = np.dot(grad_oa.T, hs) # (n_output, n_hidden)
grad_b2 = grad_oa               # (batch_size, n_output)
grad_hs = np.dot(grad_oa, self.W2) # (batch_size, n_hidden)
grad_ha = np.multiply(grad_hs, np.maximum(np.sign(ha),0))
                                # (batch_size, n_hidden)

grad_W1 = np.dot(grad_ha.T, X)  # (n_hidden, n_input)
grad_b1 = grad_ha               # (batch_size, n_hidden)
# grad_x = np.dot(grad_ha, self.W1) # (batch_size, n_input)

# mean grad
grad_W2 /= batch_size # shape keeps
grad_W1 /= batch_size # shape keeps
grad_b2 = np.mean(grad_b2, axis = 0, keepdims = True).T #(n_output, 1)
grad_b1 = np.mean(grad_b1, axis = 0, keepdims = True).T #(n_hidden, 1)
```

For both approches, the shapes of the parameters (`W1, W2, b1, b2`) do not change; they remain the same shape when they are initialized:

```
# W1 (n_hidden, n_input)
# W2 (n_output, n_hidden)
# b1 (n_hidden, 1)
# b2 (n_output, 1)
```

See the implementation of class `MLP` for the complete implementation of **forward propagation** and **backward propagation** on both one example and batch_size examples

7.

Compare both implementations (with a loop and with matrix calculus) to check that they both give the same values for the gradients on the parameters, first for K = 1, then for K = 10. Display the gradients for both methods.

Answer / Code

In this experiment, we use the function `grad_back_prop_batch(mlp, batch_X, batch_y)` implemented in **experiment 3** to calculate the gradient using loop on batch_size (K) examples for both $K = 1$ and $K = 10$, and use the functions:

```
_fprop(batch_X)
_bprop(cache, batch_y)
```

to calculate the gradient using matrix expression on batch_size (K) examples for both $K = 1$ and $K = 10$.

Here is the code and results:

In [12]:

```
Ks = [1, 10]
for K in Ks:
    print("Gradient check by loop and matrix on batch_size(K) = {}".format(K))
    mlp = MLP(n_hidden = 2, batch_size = K)
    mlp._init_weights_biases(X, y) # initialize weights and bias
    batch_size = K
    start = np.random.randint(0, len(X)-batch_size)
    batch_X = X[start:start + batch_size,:]
    batch_y = y[start:start + batch_size]

    grads_loop = grad_back_prop_batch(mlp, batch_X, batch_y)

    cache = mlp._fprop(batch_X)
    grads_matrix = mlp._bprop(cache, batch_y)

    is_grads_equal(grads_loop, grads_matrix)

    print_grads(grads_loop, grads_matrix, ["using loop", "matrix operation"])
```

```
Gradient check by loop and matrix on batch_size(K) = 1
All gradients are equal
grad_b2 by using loop:
[[-0.51004331]
 [ 0.51004331]]
grad_b2 by matrix operation:
[[-0.51004331]
 [ 0.51004331]]

grad_W2 by using loop:
[[ 0.         -0.05893403]
 [ 0.          0.05893403]]
grad_W2 by matrix operation:
[[ 0.         -0.05893403]
 [ 0.          0.05893403]]

grad_W1 by using loop:
[[ 0.          0.        ]
 [ 0.16704075 -0.05960002]]
grad_W1 by matrix operation:
[[ 0.          0.        ]
 [ 0.16704075 -0.05960002]]

grad_b1 by using loop:
[[0.        ]
 [0.17735494]]
grad_b1 by matrix operation:
[[0.        ]
 [0.17735494]]

Gradient check by loop and matrix on batch_size(K) = 10
All gradients are equal
grad_b2 by using loop:
[[-0.2040551]
 [ 0.2040551]]
grad_b2 by matrix operation:
[[-0.2040551]
 [ 0.2040551]]

grad_W2 by using loop:
```

```
[[ 0.0170263  -0.00454214]
 [-0.0170263   0.00454214]]
grad_W2 by matrix operation:
[[ 0.0170263  -0.00454214]
 [-0.0170263   0.00454214]]

grad_W1 by using loop:
[[-9.75817411e-05 -1.08023074e-03]
 [ 2.85217777e-02 -4.04200060e-02]]
grad_W1 by matrix operation:
[[-9.75817411e-05 -1.08023074e-03]
 [ 2.85217777e-02 -4.04200060e-02]]

grad_b1 by using loop:
[[-0.0009252 ]
 [ 0.01873617]]
grad_b1 by matrix operation:
[[-0.0009252 ]
 [ 0.01873617]]
```

8.

Time how long takes an epoch on fashion MNIST (1 epoch = 1 full traversal through the whole training set) for K = 100 for both versions (loop over a minibatch and matrix calculus).

Answer / Code

In this experiment, we use two member functions in our class `MLP` :

```
def fit_non_matrix_expression(self, X, y):
def fit(self, X, y)
```

Both functions using the hyper parameters direvied by function `__init__()` . Main parameters are:

```
n_hidden = 2,
lambdas = [0, 0, 0, 0],
learning_rate = 1e-3,
epochs = 200,
batch_size = 10,
early_stop = False
```

The first function tries to fit the dataset `(X, y)` by looping over each example, whereas the second does same thing by matrix calculus. See implementation of `MLP` for details of two functions.

To see how long **one** epoch using **batch_size** (K = 100) would take on two training methods, we first created an instance `mlp` by the code:

```
K = 100
mlp = MLP(n_hidden = 100, batch_size = K, epochs = 1)
```

Then, by execute the two function, we could calculate the time elapsed on each method.

Here is the complete codes and results:

In [13]:

```
K = 100
mlp = MLP(n_hidden = 100, batch_size = K, epochs = 1)
time_start = time()
mlp.fit_non_matrix_expression(fshn_train_X, fshn_train_y)
print("{:.2f} seconds for 1 epoch using loop over mini_batch (K={})."
      .format(time() - time_start, K))

time_start = time()
mlp.fit(fshn_train_X, fshn_train_y)
print("{:.2f} seconds for 1 epoch using matrix calculus on mini_batch (K={})."
      .format(time() - time_start, K))
```

```
18.67 seconds for 1 epoch using loop over mini_batch (K=100).
1.08 seconds for 1 epoch using matrix calculus on mini_batch (K=100).
```

9.

Adapt your code to compute the error (proportion of misclassified examples) on the training set as well as the total loss on the training set during each epoch of the training procedure, and at the end of each epoch, it computes the error and average loss on the validation set and the test set. Display the 6 corresponding figures (error and average loss on train/valid/test), and write them in a log file.

Answer / Code

Two functions are implemented perform this experiment:

1. `train` : train the network on training set, validation set, and test set; compute loss and error on three sets in each epoch; write them into a log file. Early stopping may also used in this experiment. In this function, loss and error on each epoch are calculated with the help of three member functions of class:

   ```
   predict_probs(X)
   compute_empirical_risk(os, y)
   compute_error(oy, y)
   ```

   See the class implementation for details.
2. `plot_learning_curves :` plot loss and error curves on the three data set.

In [14]:

```
def train(train_X, train_y, valid_X, valid_y, test_X, test_y,
          n_hidden = 100, early_stop = False, weight_decay = [0.0]*4,
          learning_rate = 1e-3, epochs = 2000,
          batch_size = 64, n_patience = 10,
          log_file_name = "mlp_training_log"):

    mlp = MLP(n_hidden = n_hidden,
              learning_rate = learning_rate,
              epochs = epochs,
              batch_size = batch_size,
              early_stop = early_stop,
              lambdas = weight_decay)

    mlp._init_weights_biases(train_X, train_y)
    # for early stopping
    best_params = None # params that performs best in valid set
    # n_patientce = 20 # maximun numbers of epoch tolerated when mlp
                      # with current params doesn't perform better than
                      # with best_params.
    n_epochs_bad = 0 # epochs which passed yet didn't bring better params
    best_valid_error = float('inf')

    sample_size, _ = train_X.shape
    batches = int(np.ceil(sample_size / mlp.batch_size))
    losses = [[], [], []] # [[train], [valid], [test]]
    errors = [[], [], []] # [[train], [valid], [test]]

    ISOTIMEFORMAT = '%Y-%m-%d %H:%M:%S'
    file_name = log_file_name + str(datetime.now().strftime(ISOTIMEFORMAT))
    file_name += ".txt"
    f = open(file_name, "w")
    f.write("training records of a MLP network\n")
    f.close()
    for epoch in range(mlp.epochs):
    # for epoch in range(mlp.epochs):
        # batch matrix forward and backward
        for j in range(batches):
            batch_start = j * mlp.batch_size
            batch_end = min(sample_size, (j+1) * mlp.batch_size)
            batch_X = train_X[batch_start:batch_end, :]
            batch_y = train_y[batch_start:batch_end]
            cache = mlp._fprop(batch_X)
            grads = mlp._bprop(cache, batch_y)
            mlp._update_params(grads)
        # end of mini_batch

        # compute loss and error rate on train, valid, and test dataset
        dataset = [(train_X, train_y), (valid_X, valid_y), (test_X, test_y)]
        for i in range(len(dataset)):
            os = mlp.predict_probs(dataset[i][0])
            loss = mlp.compute_empirical_risk(os, dataset[i][1])
            losses[i].append(loss)
            oy = np.argmax(os, axis = 1)
            error = mlp.compute_error(oy, dataset[i][1])
            errors[i].append(error)

        # write to log file for current epoch
        f = open(file_name, "a+")
        log = "epoch {:<5}: ".format(epoch)
```

```
        log += "train_l:{:5.3f}, train_er:{:.3f}; ".format(
            losses[0][-1], errors[0][-1])
        log += "valid_l:{:5.3f}, valid_er:{:.3f}; ".format(
            losses[1][-1], errors[1][-1])
        log += "test_l:{:5.3f}, test_er:{:.3f}; ".format(
            losses[2][-1], errors[2][-1])
        log += "\n"
        f.write(log)
        f.close()

        # early stopping check and related process
        if early_stop == True:
            if epoch == 0: # first epoch completed
                best_params = mlp._backup_params()
                best_valid_error = errors[1][-1]  # using error
            else:
                latest_valid_error = errors[1][-1]  # using error
                if latest_valid_error > best_valid_error: # use > instead of >
                    n_epochs_bad += 1
                    if n_epochs_bad == n_patience: # stop
                        mlp._restore_params(best_params)
                        best_epoch = epoch - n_patience
                        f = open(file_name, "a+")
                        log = "Early stop at epoch {}. With".format(best_epoch
                        log += " valid_loss:{:.3f}, valid_error:{:.3f}".format
                            losses[1][best_epoch], errors[1][best_epoch])
                        log += " train_loss:{:.3f}, train_error:{:.3f}".format
                            losses[0][best_epoch], errors[0][best_epoch])
                        f.write(log)
                        f.close()
                        debug(log)
                        debug("log file: {}".format(file_name))
                        return mlp, losses, errors, best_valid_error
                else: # loss is decressing
                    best_params = mlp._backup_params()
                    best_valid_error = latest_valid_error
                    n_epochs_bad = 0
    # end of one epoch
    debug("training complete. log file: {}".format(file_name))
    f = open(file_name, "a+")
    log = "n_hidden:{:<5}\t".format(n_hidden)
    log += "learning_rate:{:.3f}\t".format(learning_rate)
    log += "batch_size:{:<3}\t".format(batch_size)
    log += "n_patience:{:<3}\t".format(n_patience)
    log += "weight_decay:{}\n".format(weight_decay)
    f.write(log)
    f.close()
    return mlp, losses, errors, best_valid_error


def plot_learning_curves(losses, errors, data_set_name = ""):
    support_x = np.arange(len(losses[0]))
    colors = ['g-','b-','r']
    set_groups = ["train_set", "valid_set", "test_set"]

    plt.figure(figsize=(13, 6))
    plt.grid(True) # add a grid

    plt.subplot(1, 2, 1)
    plt.title("Loss curves on {} dataset".format(data_set_name))
    legends = []
```

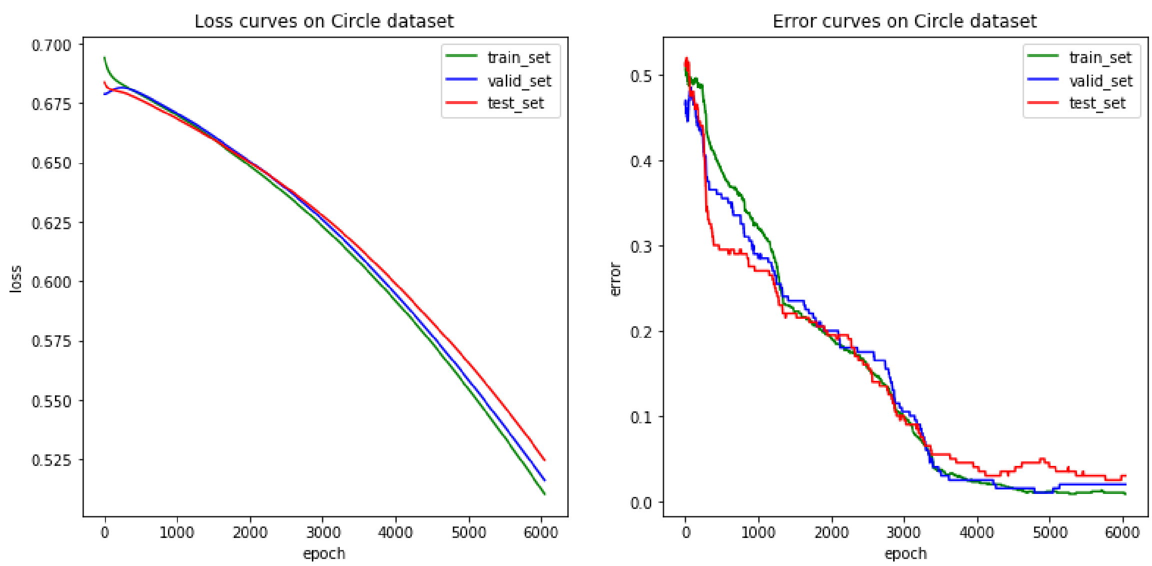
```python
    for i in range(3):
        plt.xlabel("epoch")
        plt.ylabel("loss")
        plt.plot(support_x, losses[i], colors[i])
        legends.append(set_groups[i])
    plt.legend(legends)

    plt.subplot(1, 2, 2)
    plt.title("Error curves on {} dataset".format(data_set_name))
    for i in range(3):
        plt.xlabel("epoch")
        plt.ylabel("error")
        plt.plot(support_x, errors[i], colors[i])
    plt.legend(legends)
    plt.show()
```

The following codes create an `MLP` instance with the hyper parameters given, train the network, give the best error rate on validation set, and plot the loss and error curves.

In [15]:

```python
n_hidden, early_stop, weight_decay = 16, True, [0.0]*4
mlp, losses, errors, best_valid_error = train(train_X, train_y,
                                valid_X, valid_y,
                                test_X, test_y,
                                n_hidden = n_hidden,
                                early_stop = early_stop,
                                weight_decay = weight_decay,
                                learning_rate = 1e-3,
                                epochs = 8000,
                                batch_size = 64,
                                n_patience = 1000,
                                log_file_name = "mlp_circle")

debug("best_valid_error: ", best_valid_error)
plot_learning_curves(losses, errors, "Circle")
```

```
Early stop at epoch 5048. With valid_loss:0.557, valid_error:0.010 tr
ain_loss:0.553, train_error:0.011
log file: mlp_circle2018-11-09 08:45:55.txt
best_valid_error:  0.010000000000000009
```



10

Train your network on the fashion MNIST dataset. Plot the training/valid/test curves (error and loss as a function of the epoch number, corresponding to what you wrote in a file in the last question). Add to your report the curves obtained using your best hyperparameters, i.e. for which you obtained your best error on the validation set. We suggest 2 plots : the first one will plot the error rate (train/valid/test with different colors, show which color in a legend) and the other one for the averaged loss (on train/valid/test). You should be able to get less than 20% test error.

Answer / Code

As for fashion MNIST dataset, function `train` implemented in last experiment is used. This time, the hyper parameters are set as follows:

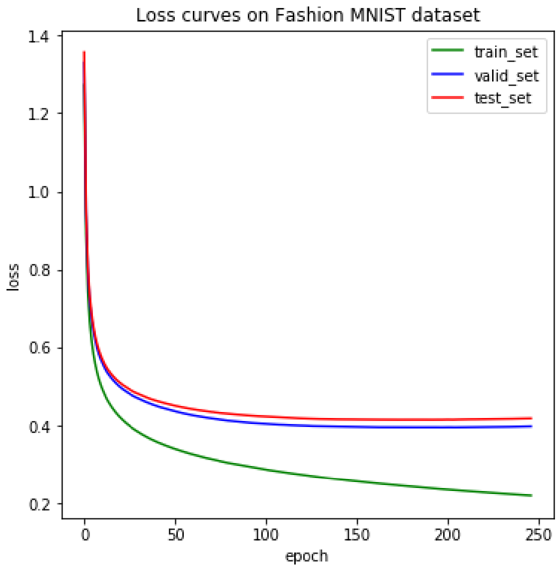
```
n_hidden = 128
early_stop = True
weight_decay = [0.0, 0.0, 0.0, 0.0]
learning_rate = 1e-4
batch_size = 128
```

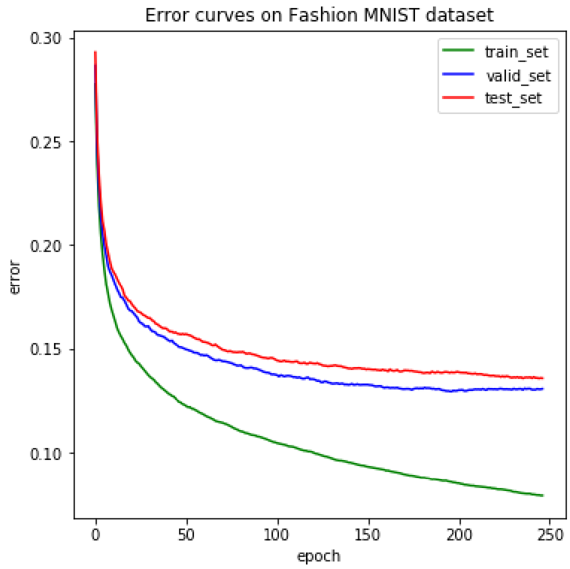
The best error rate on validation set is: 12.9%.

In [16]:

```
mlp, losses, errors, best_valid_error = train(fshn_train_X, fshn_train_y,
                                              fshn_valid_X, fshn_valid_y,
                                              fshn_test_X, fshn_test_y,
                                              n_hidden = 128,
                                              early_stop = True,
                                              weight_decay = [0.0]*4,
                                              learning_rate = 1e-4,
                                              epochs = 1000,
                                              batch_size = 128,
                                              n_patience = 50,
                                              log_file_name = "mlp_fashion_mnist")

debug("best_valid_error: {:.3f}".format(best_valid_error))
plot_learning_curves(losses, errors, "Fashion MNIST")
```

```
Early stop at epoch 196. With valid_loss:0.395, valid_error:0.129 tra
in_loss:0.237, train_error:0.086
log file: mlp_fashion_mnist2018-11-09 08:46:07.txt
best_valid_error: 0.129
```





The following codes are used to select the best from our hyper-parameter set：

```
n_hiddens = [64, 128]
early_stops = [True]
weight_decays = [[0.0]*4, [0, 0.001, 0.0, 0.001], [0.01]*4, [0.1]*4]
learning_rates = [1e-3, 1e-4]
batch_sizes = [64, 128]

for n_hidden in n_hiddens:
    for early_stop in early_stops:
        for weight_decay in weight_decays:
            for learning_rate in learning_rates:
                for batch_size in batch_sizes:
                    mlp, losses, errors, best_valid_error = train(
                                    fshn_train_X, fshn_train_y,
                                    fshn_valid_X, fshn_valid_y,
                                    fshn_test_X, fshn_test_y,
                                    n_hidden = n_hidden,
                                    early_stop = early_stop,
                                    weight_decay = weight_decay,
```

```
                                learning_rate = learning_rate,
                                epochs = 1000,
                                batch_size = batch_size,
                                n_patience = 20,
                                log_file_name = "mlp_fashion_mnist")
                        log = "best_v_error:{:.3f}\t".format(best_valid_error)
                        log += "n_hidden:{:<4}\t".format(n_hidden)
                        log += "early_stop:{}\t".format(early_stop)
                        log += "learning_rate:{:.4f}\t".format(learning_rate)
                        log += "batch_size:{:<4}\t".format(batch_size)
                        log += "lambdas:{}\n".format(weight_decay)
                        debug(log)
```

The end of the report of homework3 IFT6390

Team Member

Qiang Ye (20139927), Lifeng Wan (20108546)

Coding Environment

python 3.5.2, numpy 1.14.2 matplotlib 2.2.0

In [ ]:

```

```