Machine Learning HW3 – Neural Network

NCTU EE 0310128 游騰杰

Programming language: Python

Environment: Python 2.7.13 under Homebrew @ Mac OS X 10.11.6

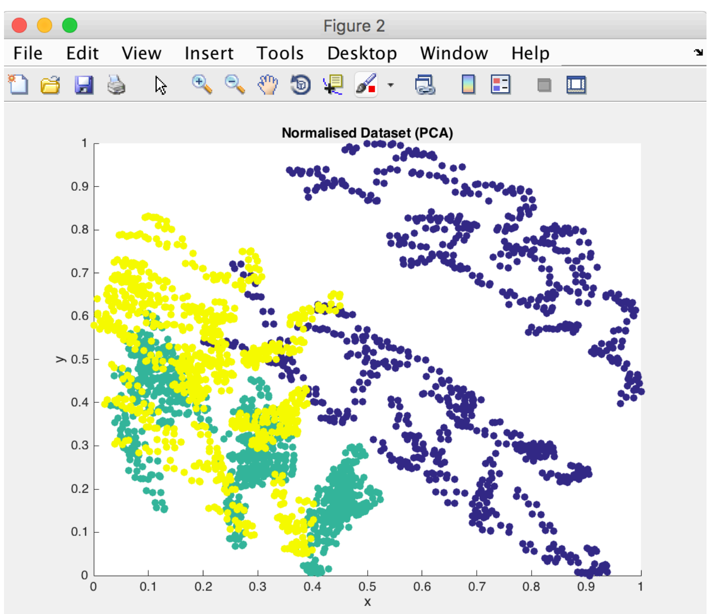
Python module used: numpy, pandas, math, Pillow, random, scikit-learn

Plot: Matlab

1. Use Principal component analysis (PCA) to map data down to 2 dimensions.

```python

def dimension_reduction(data, test, n, bLDA, target):
    if not bLDA:
        dimReduction = sklPCA(n_components=n)
        transformed_data = dimReduction.fit_transform(data)
        transformed_test = dimReduction.transform(test)
    else:
        dimReduction = sklLDA(n_components=n)
        transformed_data = dimReduction.fit_transform(data, target)
        transformed_test = dimReduction.transform(test)

    return transformed_data, transformed_test

    # Dimension Reduction
    transformed_data, transformed_test \
    = nn.dimension_reduction(cv_data, testing_data, reduce_dimension, \
                             bLDA, train_class)
    #nn.exportData3d("pca.csv", transformed_data[:, 0] \
    #                         , transformed_data[:, 1] \
    #                         , train_class)
```

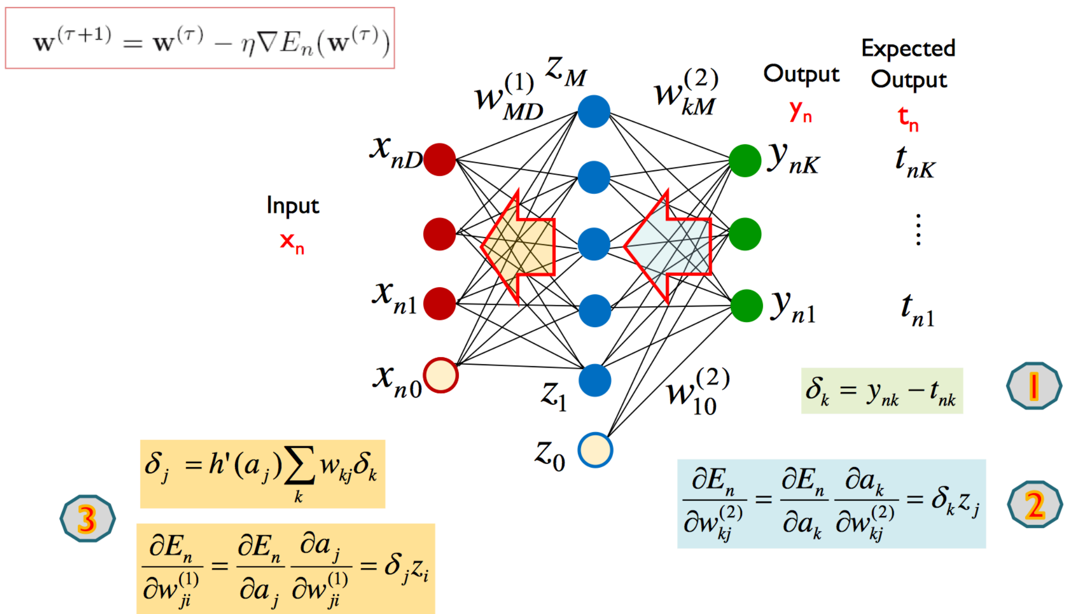
Dataset in feature subspace:



Reference: How to Implement the Backpropagation Algorithm From Scratch In Python

http://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/

2. Use stochastic gradient descent in back propagation.



Reference: (Lecture Notes) Neural Networks P. 21

I begin calculating all the deltas in every neuron backward-wise, starting with the output layer prediction with formula

$$\delta_k = y_k - t_k \tag{5.54}$$

[Formula] Reference: Textbook P. 243 (5.54)

all the preceding layers have the delta of the form

$$\delta_j = h'(a_j)\sum_k w_{kj}\delta_k \tag{5.56}$$

[Formula] Reference: Textbook P. 244 (5.56)

and I can obtain all the deltas, next combining those delta with the learning rate (η) and the layer inputs z_i' s or z_j' s using the Stochastic Gradient Descent formula to update the weight in each iteration.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)}). \tag{5.43}$$

[Formula] Reference: Textbook P. 240 (5.43)

Implementation:

```python

def training_nn(network, data, target, l_rate, n_epoch, bReLU, bMiniBatch, batch_iter):
    # target shape: numData x numClass (in one-of-k format)
    numData, numClass = target.shape
    numLayer = len(network)
    batch_count = int(numData / batch_iter)
    iterations = batch_iter if bMiniBatch else numData
    # Randomise the sequence of data
    bundle = np.random.permutation(np.hstack([data, target]))
    suffle_data = bundle[:, :data.shape[1]]
    suffle_targ = bundle[:, -1 * numClass:]
    #
    for epoch in range(n_epoch):
        error = np.array([0.0] * n_epoch)
        if not bMiniBatch:
            for itr_Data in range(numData):
                reset_delta_gradE(network)
                #probability = forward_propagate(network, data[itr_Data], bReLU)
                probability = forward_propagate(network, suffle_data[itr_Data], bReLU)
                prediction = softmax(probability)
                # Current class target
                #curr_target = target[itr_Data, :]
                curr_target = suffle_targ[itr_Data, :]
                error[epoch] = sum([pow(curr_target[itr] - prediction[itr], 2) \
                                    for itr in range(numClass)])
                calError_backprop(network, curr_target, bReLU)
                update_weights(network, suffle_data[itr_Data], l_rate, bMiniBatch, False)

```

Result:

1 Hidden Layer

```
epoch: 499, lrate: [0.1, 0.1], error: 0.383
epoch: 500, lrate: [0.1, 0.1], error: 0.383

===== Summary =====
Neural Network:
- Input Layer     :  2 neurons
- Hidden Layer #1:  4 neurons
- Output Layer    :  3 neurons
Error rate:   11.50% (   69 of   600)

In-Class error rate:
- Class 1:     9.00% (   18 of   200)
- Class 2:     0.50% (    1 of   200)
- Class 3:    25.00% (   50 of   200)
```

2 Hidden Layers

```
epoch: 200, lrate: [0.1, 0.1, 0.1], error: 0.384

===== Summary =====
Neural Network:
- Input Layer     :  2 neurons
- Hidden Layer #1:  5 neurons
- Hidden Layer #2:  7 neurons
- Output Layer    :  3 neurons
Error rate:   12.00% (   72 of   600)

In-Class error rate:
- Class 1:     9.50% (   19 of   200)
- Class 2:     3.50% (    7 of   200)
- Class 3:    23.00% (   46 of   200)
```

1 Hidden Layer (Cross-Validation)

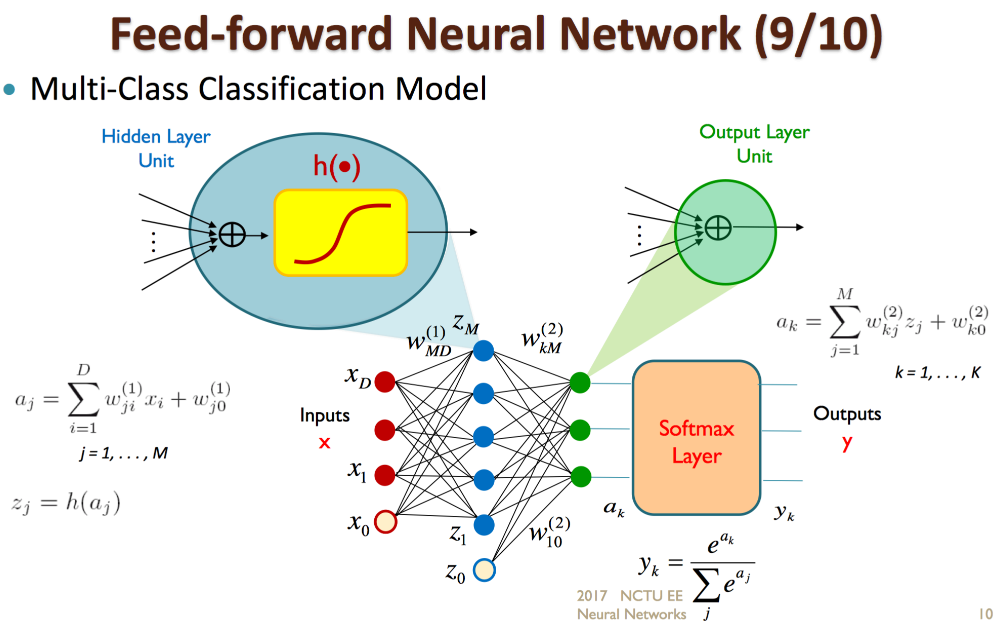
```
epoch: 500, lrate: [0.1, 0.1], error: 0.419
Iteration: #4
Error rate:   16.83% ( 101 of   600)

=== Cross-Validation Summary ===
Iterations          :  4
Epoch               : 500
Average error rate:    14.58%
Maximum error rate:    16.83%
Minimum error rate:    10.83%
```

2 Hidden Layers (Cross-Validation)

```
epoch: 199, lrate: [0.1, 0.1, 0.1], error: 0.434
epoch: 200, lrate: [0.1, 0.1, 0.1], error: 0.437
Iteration: #4
Error rate:   25.83% ( 155 of   600)

=== Cross-Validation Summary ===
Iterations          :  4
Epoch               : 200
Average error rate:    16.54%
Maximum error rate:    25.83%
Minimum error rate:    11.83%
```

3. Implement the neural network model with 1-hidden layer. Choose the sigmoid function as the activation function.

Design Structure:



Reference: (Lecture Notes) Neural Networks P. 10

First connect the inputs (red nodes) to the hidden layer (blue nodes) using the formula

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \qquad (5.2)$$

[Formula] Reference: Textbook P. 227 (5.2)

where a_j denotes the "activation" of the node, in addition, I choose sigmoid function as the activation function (shown below, equation 5.5)

```
def sigmoid_transferFcn(activation):
    # Logistic Sigmoid function
    return 1.0 / (1.0 + math.exp(-activation))

def sigmoid_derivative(x):
    # sigmoid function derivative = output * (1.0 - output)
    return x * (1.0 - x)
```

$$y_k = \sigma(a_k) \qquad (5.5)$$

$$\sigma(a) = \frac{1}{1 + \exp(-a)}. \qquad (5.6)$$

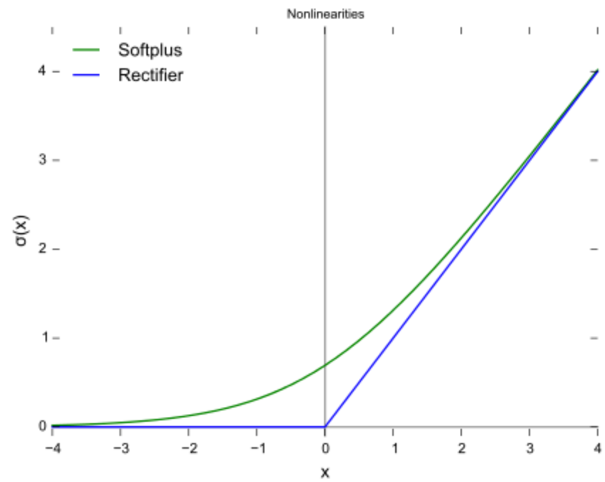
$$\frac{d\sigma}{da} = \sigma(1 - \sigma). \qquad (4.88)$$

[Formula] Reference: Textbook P. 205 (4.88), P. 228 (5.5) and (5.6)

Continue this process in the output layer (green nodes), but here the sigmoid function is changed to Softmax function for the multi-class classification model.

4. Implement the neural network model with 2-hidden layer. Choose the rectified function as the activation function.

```python
def rectify_transferFcn(activation):
    # Rectify function
    alpha = -0.01
    return activation if activation > 0.0 else \
            alpha * activation

def rectify_derivative(x):
    # Rectify function = 0 if x <= 0 else 1
    alpha = -0.01
    #print x
    return 1. if x > 0.0 else alpha
```



$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

Reference: Rectifier (neural networks) - Wikipedia
https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

The process and structure is basically the same as the previous one, but the number of hidden layer is added to two, also the activation function, which is sigmoid function in the previous case, is implemented with rectify function, or rectify linear unit (ReLU) function.

At first I found it difficult to train the model with the rectify function, after searching for some possible causes, it turned out that the rectify region (x < 0) might suffer from the dying ReLU problem in some applications, so here I use leaky ReLU for rectify function instead.

> **Leaky ReLU.** Leaky ReLUs are one attempt to fix the "dying ReLU" problem. Instead of the function being zero when x < 0, a leaky ReLU will instead have a small negative slope (of 0.01, or so). That is, the function computes $f(x) = \mathbb{1}(x < 0)(\alpha x) + \mathbb{1}(x >= 0)(x)$ where α is a small constant. Some people report success with this form of activation function, but the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron, as seen in PReLU neurons, introduced in Delving Deep into Rectifiers, by Kaiming He et al., 2015. However, the consistency of the benefit across tasks is presently unclear.

Reference: CS231n Convolutional Neural Networks for Visual Recognition
http://cs231n.github.io/neural-networks-1/

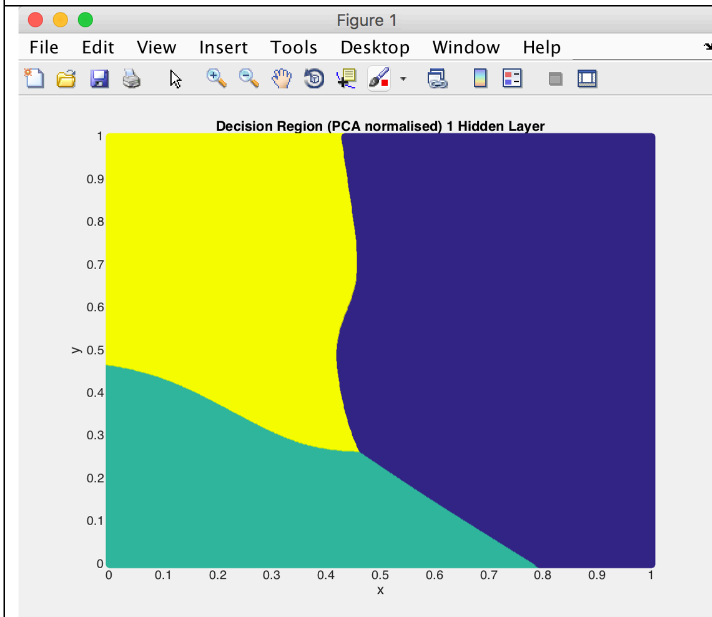
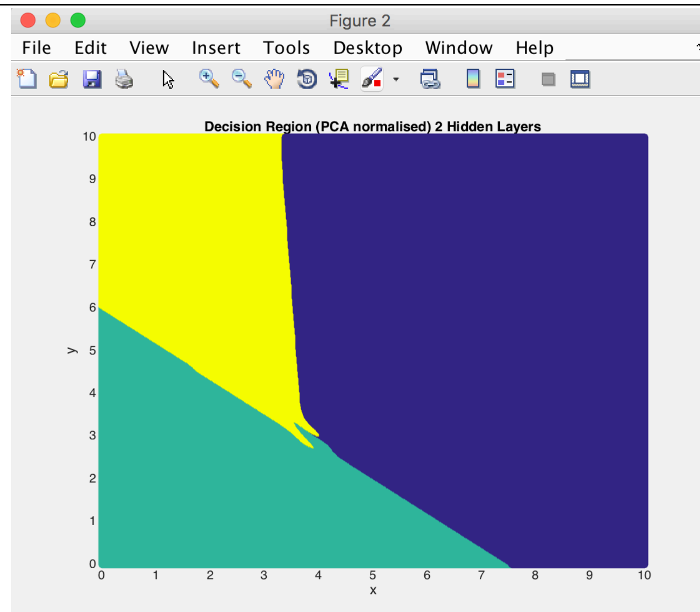
Implementing on Python:

```python
# Constants / Parameter
cv_part = 3
cv_enable = False
cv_count = cv_part if cv_enable else 1

suffle = True
reduce_dimension = 2

bLDA = False

hidden = 5
bReLU = False

bMiniBatch = True
batch_size = 5
batch_iter = batch_size if bMiniBatch else 1

reserved = [0.20, 0.20, 0.20]
l_rate = 0.10
n_epoch = 200
scaling = 1.0

Train_Path = "Data_Train/"
Demo_Path  = "Demo/"
```

```python
# Constants / Parameter
cv_part = 20
cv_enable = False
cv_count = cv_part if cv_enable else 1

suffle = True
reduce_dimension = 2

bLDA = False

hidden_1 = 5
hidden_2 = 7
bReLU = False

bMiniBatch = False
batch_size = 50
batch_iter = batch_size if bMiniBatch else 1

reserved = [0.10, 0.10, 0.10]
l_rate = 0.05
n_epoch = 25
scaling = 10.0

Train_Path = "Data_Train/"
Demo_Path  = "Demo/"
```

Name	Function
cv_part	Determine the amount of n in n-fold cross validation.
cv_enable	Enable or disable the cross-validation functioning in the code.
cv_count	Determine the times of iteration based on previous two values.
suffle	Determine whether to shuffle the input data sequence.
reduce_dimension	Determine the dimension of PCA reduces to.
bLDA	Determine whether use LDA instead of PCA.
hidden/hidden1/ hidden2	Set the number of neurons in the hidden layers.
bReLU	Determine whether to use Rectify Linear Function (ReLU) as activation function.
bMiniBatch	Determine whether to use mini-Batch Gradient Descent.
batch_size	Set the size of each mini-Batch.
batch_iter	Set the size of each mini-Batch according to bMiniBatch.
reserved_part	Define the portion of the data for each class that saved for testing (the number * 100 equal percentage of given data).
l_rate	Learning rate.
n_epoch	Set the number of epochs.
Train_Path	A list of iterations that a **w** of each class required to converge.
Demo_Path	The order of polynomial basis function.

5. Plot decision regions.

1 Hidden Layer	2 Hidden Layers
	

6. Compare and discuss their performances with homework 2.

Comparison:

	Linear Model		Neural Networks		
	Generative	Discriminative	SGD w/ sigmoid (1 hidden layer)	SGD w/ ReLU (2 hidden layers)	mBGD w/ sigmoid (1 hidden layer)
Error Rate	18.33%	11.00%	11.50%	12.00%	12.33%

SGD: Stochastic Gradient Descent

mBGD: mini-Batch Gradient Descent

ReLU: Rectify Linear Unit function

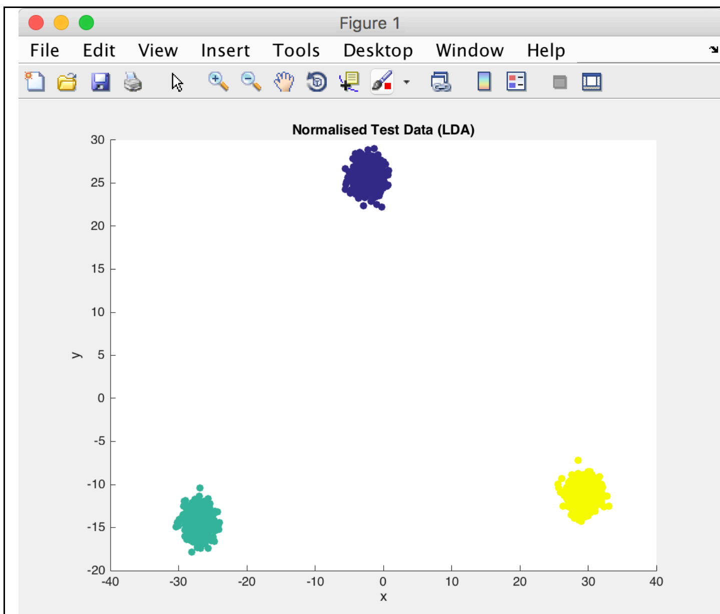
Discussion:

Starting from the training time between these models, both of the Linear Models have the least training time and the result is moderate; Neural Networks are models with higher computational complexity, but the performance and Cross-Validation result showed of high stability in the performance and high accuracy. However, the difference is not way too big, perhaps the problem itself is not complicated enough for the Neural Networks to stand out from those models, also, increasing the number of neurons in each layer may result in better fitting to the data.

Mini-Batch Gradient Descent (mBGD) is aimed to eliminate the variance between consecutive data, which leads to more stable change in the process of Gradient Descent, in the case of Stochastic Gradient Descent (SGD), the model would be too sensitive to the outliers.

Determining the size of each mini-Batch is of minor importance, I tried several times and found that 5 is a moderate number, since my training data is of a random class sequence, so if size of the mini-Batch is too large, the variance of the mini-Batch would be too small for training, or perhaps more epochs is needed. In my case, a mini-Batch with size of 5 entries of data converges quickly with around 50 epochs.

Additionally, I found the PCA is unable to seek the maximum separation between classes, so I tried performing Linear Discriminant Analysis (LDA) on the dataset, which is frequently used in supervised learning, the distribution of the dataset is shown below, both the distribution of the dataset and the result is outstanding.




```
 epoch:   9, lrate: [0.1, 0.1], error: 0.271
 epoch:  10, lrate: [0.1, 0.1], error: 0.271

 ===== Summary =====
Neural Network:
 - Input Layer    :  2 neurons
 - Hidden Layer #1:  4 neurons
 - Output Layer   :  3 neurons
 Used LDA instead of PCA.
 Error rate:    0.00% (   0 of  600)

 In-Class error rate:
 - Class 1:    0.00% (   0 of  200)
 - Class 2:    0.00% (   0 of  200)
 - Class 3:    0.00% (   0 of  200)
```

7. Explain and compare the following nouns with words:

(1) Batch Gradient Descent (BGD):

Use entire the dataset as training set, take average of the gradient of error function within the entire training set, and then update the weights once each epoch.

$$\boldsymbol{w}^{(old)} = \boldsymbol{w}^{new} + \eta \cdot \nabla E_{avg\ (whole\ trainingset)}(\boldsymbol{w})$$

(2) mini-Batch Gradient Descent (mBGD):

Divide the entire dataset to smaller dataset, take average of the gradient of error function within the mini-batch dataset, and then update the weights according to the average.

$$\boldsymbol{w}^{(old)} = \boldsymbol{w}^{new} + \eta \cdot \nabla E_{avg\ (mini-batch)}(\boldsymbol{w})$$

(3) Stochastic Gradient Descent (SGD):

Calculate the gradient of error function for each data in the training set and update the weights after each propagation of data.

$$\boldsymbol{w}^{(old)} = \boldsymbol{w}^{new} + \eta \cdot \nabla E_{stochastic}(\boldsymbol{w})$$

(4) Online Gradient Descent (OGD):

Calculate the gradient of error function after gaining real-time data and update the weights after the data propagates through the network. The big difference is that OGD discards the all the data after training.

$$\boldsymbol{w}^{(old)} = \boldsymbol{w}^{new} + \eta \cdot \nabla E_{online}(\boldsymbol{w})$$

OGD is used mostly in applications involving the analysis the trend of a phenomenon, which requires real-time accuracy, collecting data to a certain amount and update the model would be impractical for some applications, e.g. analyzing the click through rate (CTR) of an advertisement, if the data is collected (batch learning) and update the model once a fixed interval, say once a day, it would be relatively time-costing and the model cannot reflect the real-time trend of the CTR.

Comparison:

	BGD	mBGD	SGD	OGD
Training set	Fixed	Fixed	Fixed	Real-time update
Data in each iteration	Entire dataset	Subsets from the entire dataset	One entry in the entire dataset	Depends
Required computational power	High	Middle	Low	Low
Convergence (Stability)	Most stable	Comparatively stable	Not stable	Not stable

8. Bonus:

Use mini-batch method to retry the task 3 and add discussion into task 6.

```python
        else:
            offset = 0
            error = [0.0] * n_epoch
            mbatch_data, mbatch_targ \
                            = mini_batch_training(suffle_data, suffle_targ, batch_iter)
            for itr_batch in range(batch_count):
                reset_delta_gradE(network)
                # Load current batch data and target
                b_data = mbatch_data[itr_batch]
                b_targ = mbatch_targ[itr_batch]
                for itr_Data in range(batch_iter):
                    probability = forward_propagate(network, b_data[itr_Data], bReLU)
                    prediction = softmax(probability)
                    # Current class target
                    curr_target = b_targ[itr_Data]
                    error[epoch] += sum([pow(curr_target[itr] - prediction[itr], 2) \
                                        for itr in range(numClass)])
                    calError_backprop(network, curr_target, bReLU)
                    # miniBatch update grad(E)
                    update_weights(network, b_data[itr_Data], l_rate, bMiniBatch, True)

                # miniBatch update weights
                update_weights(network, b_data[0, :], l_rate, bMiniBatch, False)

            error[epoch] /= (batch_iter * batch_count)

        print " epoch: {:3d}, lrate: {:.3f}, error: {:.3f}" \
              .format(epoch + 1, l_rate, error[epoch])

```

```python
def update_weights(network, data, l_rate, bMiniBatch, updateGradE):
    # w(new) = w(old) + learning_rate * delta * input
    # mini-batch: w(new) = w(old) + learning_rate * avg(grad(E(w)))
    numLayer = len(network)
    for itr_layer in range(numLayer):
        if itr_layer != 0:
            # Inputs of current layer = outputs from previous layer
            inputs = [neuron['output'] for neuron in network[itr_layer - 1]]
        else:
            # Initlialize input with input data (bias not included)
            inputs = data

        numInput = len(inputs)
        for neuron in network[itr_layer]:
            for itr in range(numInput): # bias is added later
                if not bMiniBatch:
                    neuron['weights'][itr] += l_rate * neuron['delta'] * inputs[itr]
                elif updateGradE:
                    neuron['gradE'][itr] = np.append(neuron['gradE'][itr],
                                                     neuron['delta'] * inputs[itr])
                else:
                    neuron['weights'][itr] += l_rate * neuron['gradE'][itr].mean()

            # Update bias weight / gradient E
            if not bMiniBatch:
                neuron['weights'][-1] += l_rate * neuron['delta'] * 1.0
            elif updateGradE:
                neuron['gradE'][-1] = np.append(neuron['gradE'][-1],
                                                neuron['delta'] * 1.0)
            else:
                neuron['weights'][-1] += l_rate * neuron['gradE'][-1].mean()

```

Result:

```
epoch: 195, lrate: 0.100, error: 0.433
epoch: 196, lrate: 0.100, error: 0.433
epoch: 197, lrate: 0.100, error: 0.433
epoch: 198, lrate: 0.100, error: 0.433
epoch: 199, lrate: 0.100, error: 0.433
epoch: 200, lrate: 0.100, error: 0.433

===== Summary =====
Activation function        : Sigmoid Function
mini-Batch Gradient Descent: True
mini-Batch Size            :   5
Error rate:   12.33% (  74 of  600)

In-Class error rate:
- Class 1:   12.00% (  24 of  200)
- Class 2:    2.00% (   4 of  200)
- Class 3:   23.00% (  46 of  200)
```