Neural Network

Homework 2

Ning Ma, A50055399

1 Linear Rgression

1. Since $\epsilon \sim \mathcal{N}(\mu = 0, \sigma^2)$, we have $\mathrm{P}(t|x;\theta) = \mathcal{N}(f(x,\theta), \sigma^2)$. Thus, for iid samples, we have

$$\mathrm{P}(T|X;\theta) = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(t^{(i)} - f(x^{(i)},\theta))^2}{2\sigma^2}\right) \tag{1}$$

The optimal parameter θ should maximize the above posterior. Since **log** is a monotonically increasing function, it is equivalent to maximize $\log \mathrm{P}(T|X;\theta)$

$$\log \mathrm{P}(T|X;\theta) \sim -\sum_{i=1}^{N} (t^{(i)} - f(x^{(i)},\theta))^2 \tag{2}$$

which is equivalent to minimize $\sum_{i=1}^{N}(t^{(i)} - f(x^{(i)},\theta))^2$ which is the SEE.
In conclusion, in this special problem, minimize the SSE is equivalent to maximize the posterior of observation.

2 Multilayer Perceptron

(a) The cross entropy loss function for softmax regression is

$$E = -\sum_{l=0}^{K-1} 1_{\{label=l\}} \log\, y_l \tag{3}$$

where $y_l = \frac{\exp(a_l)}{\sum_{m=0}^{K-1}\exp(a_m)}$

For the output layer, we have

$$\delta_k = -\frac{\partial E}{\partial a_k} = -\sum_l \frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial a_k} \tag{4}$$

$$= -\sum_l -\frac{1_{\{label=l\}}}{y_l}(y_l\delta_{lk} - y_l y_k) \tag{5}$$

$$= \sum_l 1_{\{label=l\}}(\delta_{lk} - y_k) \tag{6}$$

$$= \sum_l \delta_{lk} 1_{\{label=l\}} - y_k \sum_l 1_{\{label=l\}} \tag{7}$$

$$= 1_{\{label=k\}} - y_k = t_k - y_k \tag{8}$$

For the hidden layer, $y_j = g(a_j)$, we have

$$\delta_j = -\frac{\partial E}{\partial a_j} = -\sum_k \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{9}$$

$$= \sum_k \delta_k \frac{\partial a_k}{\partial a_j} = \sum_k \delta_k \frac{\partial a_k}{\partial y_j}\frac{\partial y_j}{\partial a_j} \tag{10}$$

$$= \sum_k \delta_k \frac{\partial \sum_l w_{lk} y_l}{\partial y_j} y_j^{'} = \sum_k \delta_k w_{jk} y_j^{'} \tag{11}$$

$$= y_j^{'} \sum_k \delta_k w_{jk} \tag{12}$$

where δ_k has been computed from the output layer.
(b) For the output layer, we have

$$w_{jk} = w_{jk} - \alpha \frac{\partial E}{\partial w_{jk}} = w_{jk} - \alpha \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial w_{jk}} \tag{13}$$

$$= w_{jk} + \alpha \delta_k \frac{\partial \sum_l w_{lk} y_l}{\partial w_{jk}} \tag{14}$$

$$= w_{jk} + \alpha \delta_k y_j \tag{15}$$

For the hidden layer, we have

$$w_{ij} = w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}} = w_{ij} - \alpha \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} \tag{16}$$

$$= w_{ij} + \alpha \delta_j \frac{\partial \sum_l w_{lj} x_l}{\partial w_{ij}} \tag{17}$$

$$= w_{ij} + \alpha \delta_j x_i \tag{18}$$

where we have already computed the δ_k and δ_j in part (a).

(c) For the output layer, since $w_{jk} = w_{jk} + \alpha \delta_k y_j$, we have

$$W_{HO} = W_{HO} + \alpha y^{(j)} \otimes \delta^{(k)} \tag{19}$$

where $y^{(j)}$ is a column-vector output from hidden layer, $\delta^{(k)}$ is a column-vector δ from the output layer, and $\otimes$ is an outer product operator.

Similarly, for the hidden layer, since $w_{ij} = w_{ij} + \alpha \delta_j x_i$, we have

$$W_{IH} = W_{IH} + \alpha x^{(i)} \otimes \delta^{(j)} \tag{20}$$

where $x^{(i)}$ is a column-vector input, $\delta^{(j)}$ is a column-vector δ from the hidden layer, and $\otimes$ is an outer product operator.

Since $\delta_j = y_j^{'} \sum_k \delta_k w_{jk}$, we have

$$\delta^{(j)} = (y^{'})^{(j)} \odot (W_{HO} \bullet \delta^{(k)}) \tag{21}$$

where $\odot$ is an element-wise multiplication operator. Thus, we have

$$W_{IH} = W_{IH} + \alpha x^{(i)} \otimes \left((y^{'})^{(j)} \odot (W_{HO} \bullet \delta^{(k)})\right) \tag{22}$$

(d)

i. See code 'DataProcess.py' for detail

ii. I choose a few parameters and the difference is smaller than 10^{-5}. Since I also get fine results in the following questions, I think my gradient is correct.

iii. We can see from Figure 1 that the training and test accuracy increases rapidly with number of iteration.

(e) From Figure 2, we can see that with regularization, the accuracy also increases rapidly with number of iterations. Large regularization can prevent from overfitting but make the model less powerful. With regularization, the gradient converges faster than the one without any trick. However, as regularization parameter increases, the accuracy decreases a little bit because the model is less powerful.

(f) From Figure 3, the accuracy also increases with number of iterations but increases slower than previous cases. This is because momentum intends to dump big jumps during gradient descent process.

(g) Figure 4 shows how accuracy changes with number of iteration for different functions. We can see that *sigmoid* activation function converges more quickly than *tanh* and *ReLu*, and *tanh* converges quicker than *ReLu*. This is because *ReLu* is a linear increasing function, while *sigmod* and *tanh* are nonlinear functions and become saturated when input is large

(h)

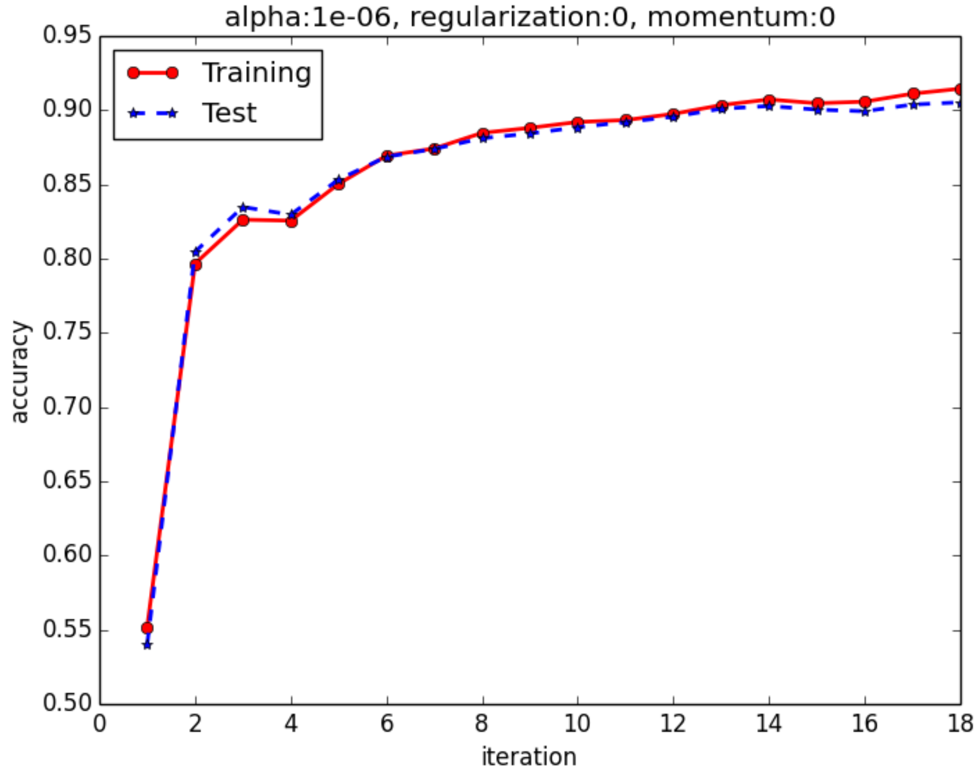
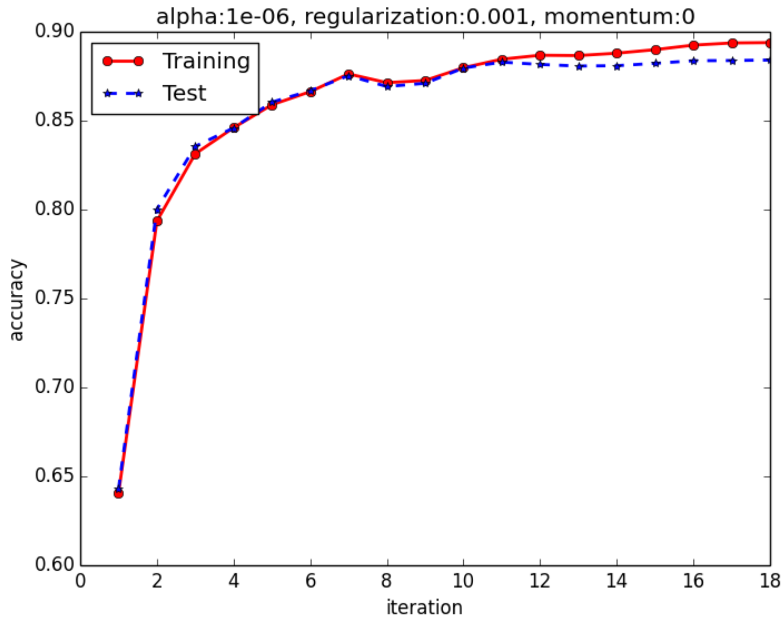


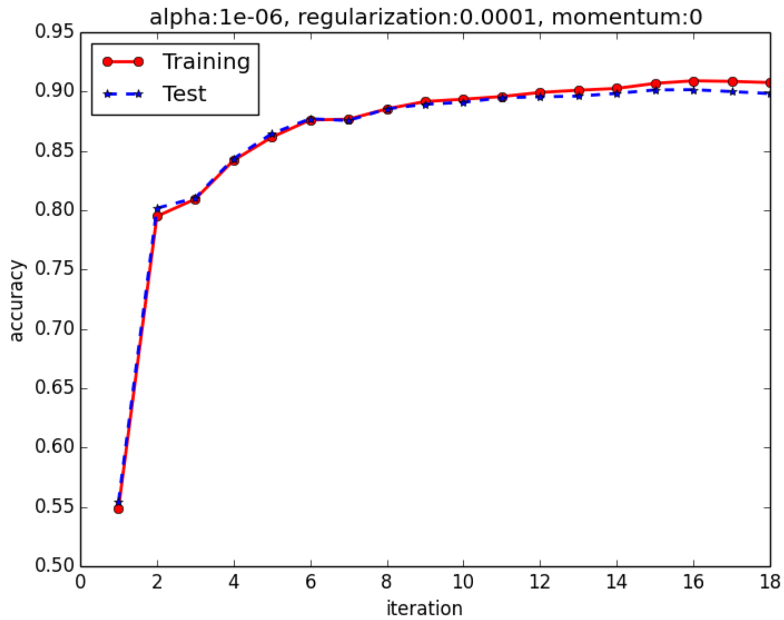
Figure 1: (d).iii. Sigmoid without trick

i. From Figure 5, we can see that the accuracy increases with number of hidden units. It makes sense because the neural network becomes more powerful when the number of hidden units is larger. However, we can see that the accuracy improvement is very small. So, it is not always necessary to have a very large number of hidden units to increase slight accuracy but sacrifice a lot of computational time.

ii. From Figure 6, we can see that, though the number of weights are similar, the accuracy increases with number of layers. Although, this double hidden layer network has similar number of weight, it has more complicated structure, and thus provides more powerful representation. So, when increasing hidden units does not increase the accuracy significantly, we can consider changing the structure of the network.



(a) regularization: 0.001



(b) regularization: 0.0001

Figure 2: (e). Different regularization

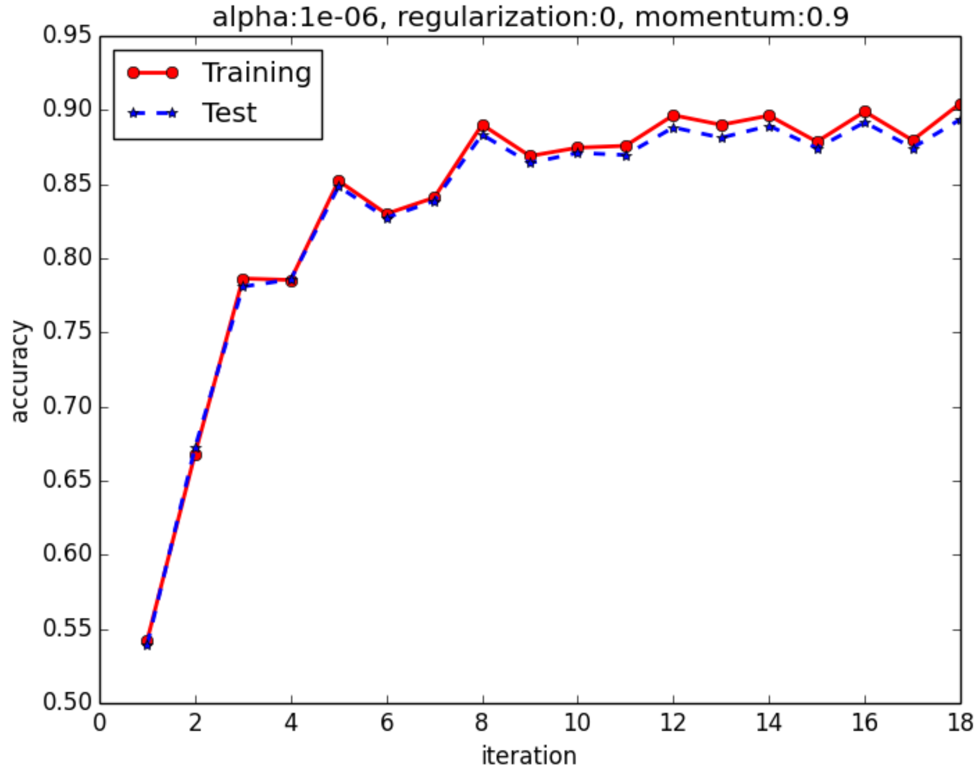
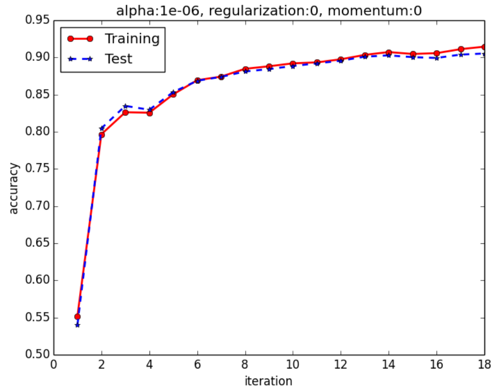


Figure 3: (f). With momentum

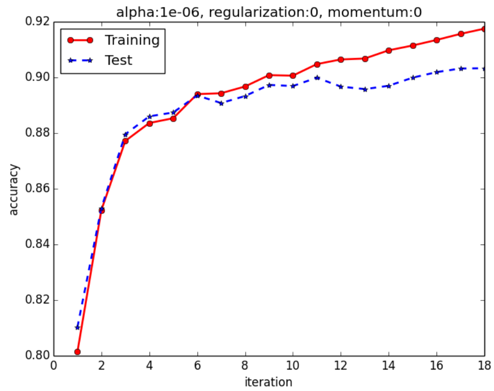
3 Appendix

The following is the source code

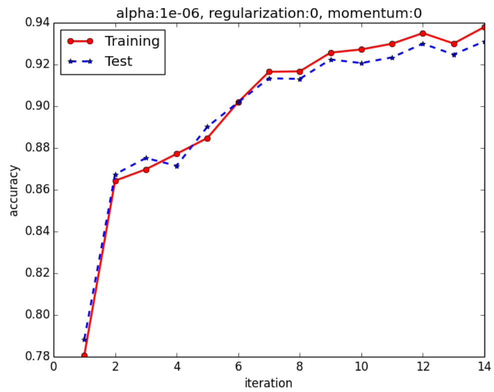
```
#################### DataProcess.py ##################
import os, struct
import numpy as np
from array import array as pyarray
from numpy import append, array, int8, uint8, zeros
from scipy import stats

def load_mnist(dataset="training", digits=None, path=None,
    asbytes=False, selection=None,
      return_labels=True, return_indices=False, zscore=True,
          appendOne=True, isShuffle=True):
```



(a) sigmoid

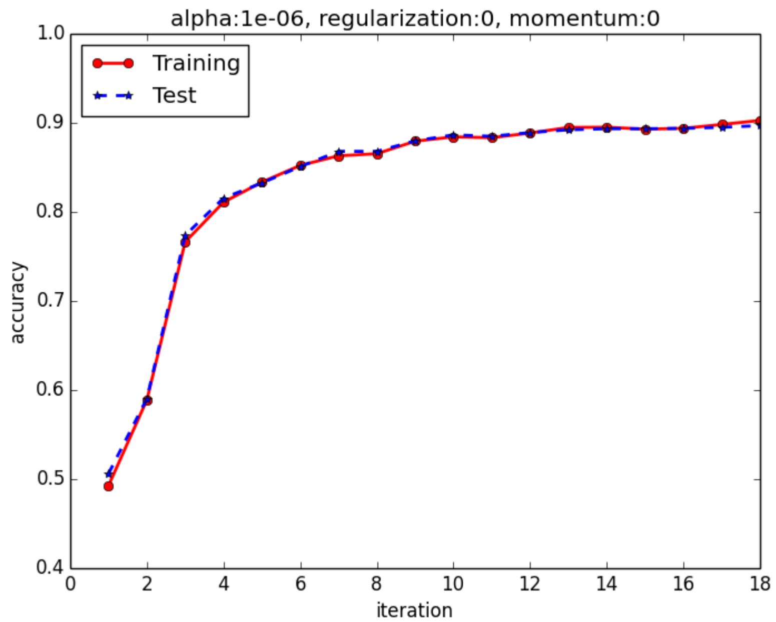


(b) tanh

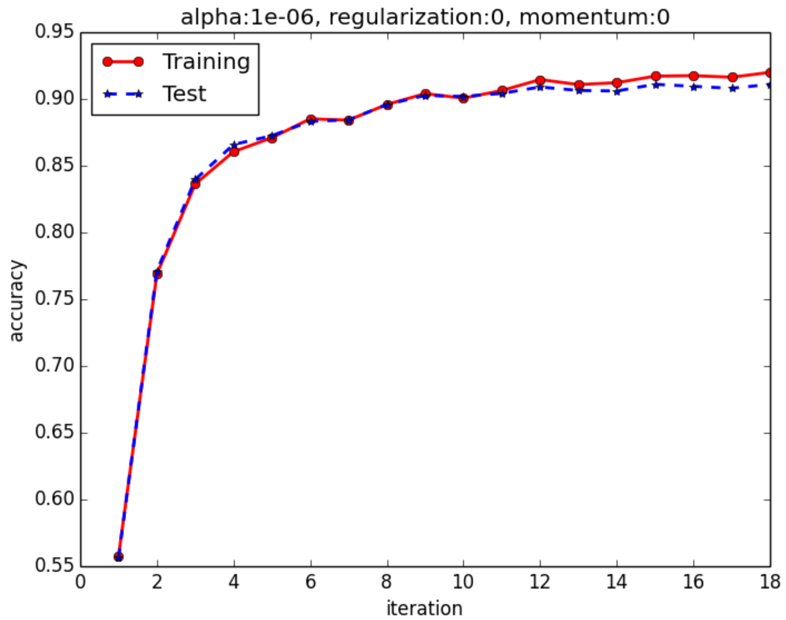


(c) ReLu

Figure 4: (g). Different activation functions



(a) Half hidden units



(b) Double hidden units

Figure 5: (h).i. Different number of hidden units

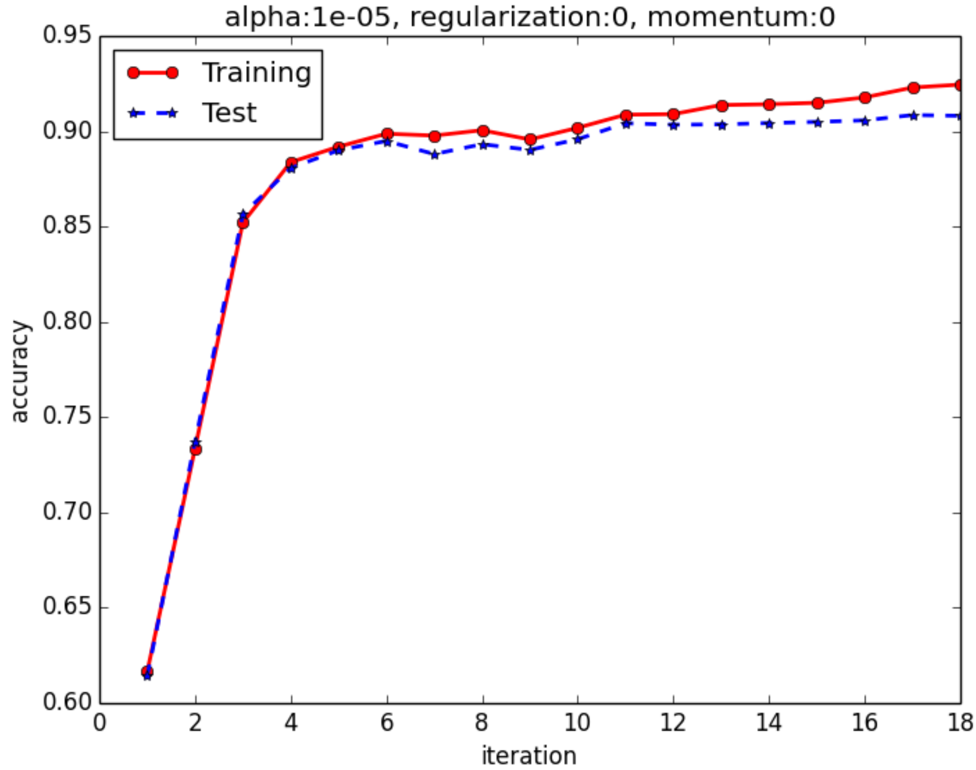


Figure 6: (h).ii. Two Hidden Layer

```
"""
Loads MNIST files into a 3D numpy array.

You have to download the data separately from [MNIST]_.
    It is recommended
to set the environment variable ``MNIST`` to point to
   the folder where you
put the data, so that you don't have to select path. On
    a Linux+bash setup,
this is done by adding the following to your ``.bashrc
   ``::

    export MNIST=/path/to/mnist

Parameters
----------
```

```
dataset : str
    Either "training" or "testing", depending on which
       dataset you want to
    load.
digits : list
    Integer list of digits to load. The entire database
        is loaded if set to
    ``None``. Default is ``None``.
path : str
    Path to your MNIST datafiles. The default is ``None
       ``, which will try
    to take the path from your environment variable ``
       MNIST``. The data can
    be downloaded from http://yann.lecun.com/exdb/mnist
       /.
asbytes : bool
    If True, returns data as ``numpy.uint8`` in [0,
       255] as opposed to
    ``numpy.float64`` in [0.0, 1.0].
selection : slice
    Using a `slice` object, specify what subset of the
       dataset to load. An
    example is ``slice(0, 20, 2)``, which would load
       every other digit
    until--but not including--the twentieth.
return_labels : bool
    Specify whether or not labels should be returned.
       This is also a speed
    performance if digits are not specified, since then
        the labels file
    does not need to be read at all.
return_indicies : bool
    Specify whether or not to return the MNIST indices
       that were fetched.
    This is valuable only if digits is specified,
       because in that case it
    can be valuable to know how far
    in the database it reached.
zscore : boolean
```

```
        if True, do the zscore transformation for the raw
            image data. Default is true.
    appendOne : boolean
        if True, append one in the front of the image data
            representing the input to the bias node
    isShuffle : boolean
        if True, shuffle the data set.

    Returns
    ---------
    images : ndarray
        Image data of shape ``(N, rows, cols)``, where ``N
            `` is the number of images. If neither labels
            nor inices are returned, then this is returned
            directly, and not inside a 1-sized tuple.
    labels : ndarray
        Array of size ``N`` describing the labels. Returned
             only if ``return_labels`` is `True`, which is
            default.
    indices : ndarray
        The indices in the database that were returned.

    Examples
    ----------
    Assuming that you have downloaded the MNIST database
        and set the
    environment variable ``$MNIST`` point to the folder,
        this will load all
    images and labels from the training set:

    >>> images, labels = ag.io.load_mnist('training') #
        doctest: +SKIP

    Load 100 sevens from the testing set:

    >>> sevens = ag.io.load_mnist('testing', digits=[7],
        selection=slice(0, 100), return_labels=False) #
        doctest: +SKIP

    """
```

```
# The files are assumed to have these names and should
    be found in 'path'
files = {
    'training': ('train-images-idx3-ubyte', 'train-
        labels-idx1-ubyte'),
    'testing': ('t10k-images-idx3-ubyte', 't10k-labels-
        idx1-ubyte'),
}

if path is None:
    try:
        path = os.environ['MNIST']
    except KeyError:
        raise ValueError("Unspecified path requires
            environment variable $MNIST to be set")

try:
    images_fname = os.path.join(path, files[dataset
        ][0])
    labels_fname = os.path.join(path, files[dataset
        ][1])
except KeyError:
    raise ValueError("Data set must be 'testing' or '
        training'")

# We can skip the labels file only if digits aren't
    specified and labels aren't asked for
if return_labels or digits is not None:
    flbl = open(labels_fname, 'rb')
    magic_nr, size = struct.unpack(">II", flbl.read(8))
    labels_raw = pyarray("b", flbl.read())
    flbl.close()

fimg = open(images_fname, 'rb')
magic_nr, size, rows, cols = struct.unpack(">IIII",
    fimg.read(16))
images_raw = pyarray("B", fimg.read())
fimg.close()
```

```
indices = range(size)
N = len(indices)

images = zeros((N, rows*cols), dtype=uint8)

if return_labels:
    labels = zeros((N), dtype=int8)
for i, index in enumerate(indices):
    images[i] = array(images_raw[ indices[i]*rows*cols
        : (indices[i]+1)*rows*cols ]).reshape(1,rows*
        cols)
    if return_labels:
        labels[i] = labels_raw[indices[i]]

if not asbytes:
        images = images.astype(float)/255.0

if zscore:
    images = zscoreTransform(images);
#numpy.insert(arr, index, values, axis=None)
if appendOne:
    images = np.insert(images, 0, 1, axis=1)

#shuffle the images
if isShuffle:
    np.random.shuffle(indices)
    images = images[indices]
    labels = labels[indices]


#choose images from shuffled images
indices = range(size)
if digits:
    indices = [k for k in range(size) if labels[k] in
        digits]

if selection:
    indices = indices[selection]

images = images[indices]
```

```
    labels = labels[indices]




    ret = (images,)
    if return_labels:
        ret += (labels,)
    if return_indices:
        ret += (indices,)
    if len(ret) == 1:
        return ret[0] # Don't return a tuple of one
    else:
        return ret

def zscoreTransform(data):
    """
    Do a z-score transformation for the data such that the
       mean is 0 and variance is 1.
    """
    zscoreData = stats.zscore(data);
    #convert NaN to zero. NaN mean this feacture is a
       constant. So this feature is zero after centering
    zscoreData[np.isnan(zscoreData)] = 0;
    return zscoreData

if __name__ == '__main__':
    myTrainData = load_mnist(dataset="training", path='../'
       , digits = None, selection = None, zscore = True,
       isShuffle = True)
    import matplotlib.pyplot as plt

    images = myTrainData[0]
    for i in range(4):
        print i
        image = images[i]
        image = image[1:].reshape(28,28)
        plt.imshow(image, cmap=plt.get_cmap('gray'))
        plt.show()
```

```
################### FunctionGradient ####################
import numpy as np

def SoftMaxFunc(inputValue):
        return np.exp(inputValue) / sum(np.exp(inputValue))

def tanh(inputValue):
        return np.tanh(inputValue)

def tanhGradient(inputValue):
        return 1 - np.multiply(tanh(inputValue),tanh(
           inputValue))

def sigmoid(inputValue):
        return 1.0 / (1 + np.exp(-inputValue))

def sigmoidGradient(inputValue):
        return np.multiply(sigmoid(inputValue), 1 - sigmoid
           (inputValue))

def ReLu(inputValue):
        return np.maximum(0, inputValue)

def ReLuGradient(inputValue):
        slope = np.ones(inputValue.shape)
        slope[inputValue <= 0] = 0
        return slope

######################### multilayer.py ###################
import numpy as np
from FunctionGradient import SoftMaxFunc, tanh,
   tanhGradient, sigmoid, sigmoidGradient, ReLu,
   ReLuGradient
from DataProcess import load_mnist
import matplotlib.pyplot as plt
import timeit


class SoftmaxTopLayer(object):
        def __init__(self, rng, n_in, n_out):
                self.W = np.asarray( rng.uniform(
```

```
                low = -np.sqrt(6.0 / (n_in + n_out)
                    ),
                high = np.sqrt(6.0 / (n_in + n_out)
                    ),
                size = (n_in, n_out)
                )
        )
        #W(t) - W(t - 1)
        self.deltaW = np.zeros((n_in, n_out))

        self.delta = np.zeros(n_out)
        #\partial (E) / \partial (W)
        self.weightGradient = np.zeros((n_in, n_out
            ))
        self.inputValue = np.zeros(n_in)
        self.y = SoftMaxFunc(np.dot(self.inputValue
            , self.W))
        self.predict = np.argmax(self.y)
        self.params = self.W



class HiddenLayer(object):
    def __init__(self, rng, n_in, n_out, activation,
        activationGradient):
        self.W = np.asarray( rng.uniform(
                low = -np.sqrt(6.0 / (n_in + n_out)
                    ),
                high = np.sqrt(6.0 / (n_in + n_out)
                    ),
                size = (n_in, n_out)
                )
        )
        self.W[:,0] = 0
        #W(t) - W(t - 1)
        self.deltaW = np.zeros((n_in,n_out))

        #print self.W
        self.activation = activation
```

```
        self.activationGradient =
           activationGradient

        #the derivative at bias unit is zero
        self.delta = np.zeros(n_out)
        self.weightGradient = np.zeros((n_in, n_out
           ))
        self.inputValue = np.zeros(n_in)
        self.linearOutput = np.dot(self.inputValue,
             self.W)
        #add bias unit
        self.y = activation(self.linearOutput)
        self.y[0] = 1

        self.yPrime = activationGradient(self.
           linearOutput)
        self.yPrime[0] = 0

        self.params = self.W


class MultiLayerPerceptron(object):
    def __init__(self, rng, n_in, n_hidden, n_out,
                activation = tanh,
                   activationGradient =
                   tanhGradient, learningRate =
                   10**(-6), regularization = 0,
                   momentum = 0):
        self.labelRange = np.array(range(10))

        self.hiddenLayer = HiddenLayer(
                rng = rng, n_in = n_in, n_out =
                   n_hidden + 1, activation =
                   activation, activationGradient =
                    activationGradient
        )

        self.softmaxTopLayer = SoftmaxTopLayer(
                rng = rng, n_in = n_hidden + 1,
                   n_out = n_out
```

```
        )

        self.params = [self.hiddenLayer.params] + [
            self.softmaxTopLayer.params]
        self.learningRate = learningRate
        self.regularization = regularization
        self.momentum = momentum

    def forwardPropagate(self, inputValue, output):
        outputVec = 1*(self.labelRange == output)

        #propagate to the hidden layer, bias
            corresponds to a unit with constant
            ouput 1
        self.hiddenLayer.inputValue = inputValue
        self.hiddenLayer.linearOutput = np.dot(self
            .hiddenLayer.inputValue, self.
            hiddenLayer.W)
        self.hiddenLayer.y = self.hiddenLayer.
            activation(self.hiddenLayer.linearOutput
            )
        self.hiddenLayer.y[0] = 1

        self.hiddenLayer.yPrime = self.hiddenLayer.
            activationGradient(self.hiddenLayer.
            linearOutput)
        self.hiddenLayer.yPrime[0] = 0

        #propagate to the top layer, bias
            corresponds to a unit with constant
            ouput 1
        self.softmaxTopLayer.inputValue = self.
            hiddenLayer.y
        self.softmaxTopLayer.linearOutput = np.dot(
            self.softmaxTopLayer.inputValue, self.
            softmaxTopLayer.W)
        self.softmaxTopLayer.y = SoftMaxFunc(self.
            softmaxTopLayer.linearOutput)
```

```
        self.softmaxTopLayer.predict = np.argmax(
            self.softmaxTopLayer.y)

        self.softmaxTopLayer.delta = outputVec -
            self.softmaxTopLayer.y
        self.softmaxTopLayer.weightGradient = self.
            softmaxTopLayer.weightGradient - \
                np.outer(self.softmaxTopLayer.
                    inputValue, self.softmaxTopLayer.
                    delta)


def backwardPropagate(self):
        self.hiddenLayer.delta = np.multiply(self.
            hiddenLayer.yPrime,
                np.dot(self.softmaxTopLayer.W, self
                    .softmaxTopLayer.delta))

        self.hiddenLayer.weightGradient = self.
            hiddenLayer.weightGradient - \
                np.outer(self.hiddenLayer.
                    inputValue, self.hiddenLayer.
                    delta)


def updateWeight(self):
        preW = self.softmaxTopLayer.W
        self.softmaxTopLayer.W = preW - \
                self.learningRate*self.
                    softmaxTopLayer.weightGradient +
                    \
                2*self.regularization*preW + self.
                    momentum*self.softmaxTopLayer.
                    deltaW
        self.softmaxTopLayer.deltaW = self.
            softmaxTopLayer.W - preW



        preHiddenW = self.hiddenLayer.W
```

```
            self.hiddenLayer.W = preHiddenW - \
                    self.learningRate*self.hiddenLayer.
                       weightGradient + \
                    2*self.regularization*preHiddenW +
                       self.momentum*self.hiddenLayer.
                       deltaW
            self.hiddenLayer.deltaW = self.hiddenLayer.
               W - preHiddenW


    def accuracy(self, INPUT, OUTPUT):
            count = 0
            for inputValue, output in zip(INPUT,OUTPUT)
               :
                    linearOutputHidden = np.dot(
                       inputValue, self.hiddenLayer.W)
                    yHidden = self.hiddenLayer.
                       activation(linearOutputHidden)
                    yHidden[0] = 1

                    linearOutput = np.dot(yHidden, self
                       .softmaxTopLayer.W)
                    y = SoftMaxFunc(linearOutput)
                    predict = np.argmax(y)

                    if predict == output:
                            count = count + 1

            return (float(count) / len(OUTPUT))

def test_MLP(MLP, trainImage, trainLabel, validImage,
   validLabel, testImage, testLabel,fileName, nEpochs=18):
        batchSize = 500
        numBatch = trainImage.shape[0] / batchSize

        trainingAccuracy = []
        testAccuracy = []
        preValidAccuracy = 0.05

        for k in range(nEpochs):
```

```
        start_time = timeit.default_timer()
        for i in range(numBatch):
            #miniBatch
            for j in range(batchSize):
                index = i*batchSize + j
                MLP.forwardPropagate(
                    trainImage[index,:],
                    trainLabel[index])
                MLP.backwardPropagate()

            MLP.updateWeight()
        validAccuracy = MLP.accuracy(validImage,
            validLabel)


        if validAccuracy > 0.95 and (abs(
            validAccuracy - preValidAccuracy) /
            preValidAccuracy < 0.0005):
                break
        preValidAccuracy = validAccuracy

        end_time = timeit.default_timer()
        print "one pass takes " + str(end_time -
            start_time) + 's'
        trainingAccuracy.append(MLP.accuracy(
            trainImage, trainLabel))
        testAccuracy.append(MLP.accuracy(testImage,
            testLabel))

plt.plot(range(1, len(trainingAccuracy) + 1,1),
    trainingAccuracy, 'ro-', linewidth=2)
plt.plot(range(1, len(testAccuracy) + 1,1),
    testAccuracy, 'b*--', linewidth=2)
plt.legend(["Training", "Test"],loc = 2 )
plt.xlabel('iteration')
plt.ylabel('accuracy')
plt.title('alpha:' + str(MLP.learningRate) + ", 
    regularization:" + str(MLP.regularization) + \
        ", momentum:" + str(MLP.momentum))
plt.savefig('./' + str(fileName))
```

```
        plt.show()


if __name__ == '__main__':
        allTrain = load_mnist(dataset="training", path='../
            ')

        trainImage = allTrain[0][0:49999,:]
        trainLabel = allTrain[1][0:49999]

        validImage = allTrain[0][50000:,:]
        validLabel = allTrain[1][50000:]

        allTest = load_mnist(dataset="testing",path='../')
        testImage = allTest[0]
        testLabel = allTest[1]

        rng = np.random.RandomState(1234)

        #(D)
        MLP = MultiLayerPerceptron(rng, n_in = 28*28 + 1,
            n_hidden = 50, n_out = 10,
                    activation = sigmoid, activationGradient =
                        sigmoidGradient, learningRate = 10**(-6)
                        )
        test_MLP(MLP, trainImage, trainLabel, validImage,
            validLabel, testImage, testLabel,"sigmoid.png")

        #(E) /over flow occur
        MLPRegu1 = MultiLayerPerceptron(rng, n_in = 28*28 +
             1, n_hidden = 50, n_out = 10,
                    activation = sigmoid, activationGradient =
                        sigmoidGradient, learningRate = 10**(-6)
                        , regularization = 0.001)
        test_MLP(MLPRegu1, trainImage, trainLabel,
            validImage, validLabel, testImage, testLabel, "
            regularization1.png")

        MLPRegu2 = MultiLayerPerceptron(rng, n_in = 28*28 +
             1, n_hidden = 50, n_out = 10,
```

```
        activation = sigmoid, activationGradient =
            sigmoidGradient, learningRate = 10**(-6)
            , regularization = 0.0001)
test_MLP(MLPRegu2, trainImage, trainLabel,
    validImage, validLabel, testImage, testLabel,"
    regularization2.png")

#(F)
MLPMomentum = MultiLayerPerceptron(rng, n_in =
    28*28 + 1, n_hidden = 50, n_out = 10,
        activation = sigmoid, activationGradient =
            sigmoidGradient, learningRate = 10**(-6)
            , momentum = 0.9)
test_MLP(MLPMomentum, trainImage, trainLabel,
    validImage, validLabel, testImage, testLabel,"
    momentum.png")

#(G)
MLPTanh = MultiLayerPerceptron(rng, n_in = 28*28 +
    1, n_hidden = 50, n_out = 10,
        activation = tanh, activationGradient =
            tanhGradient, learningRate = 10**(-6))
test_MLP(MLPTanh, trainImage, trainLabel,
    validImage, validLabel, testImage, testLabel,"
    tanh.png")

MLPReLu = MultiLayerPerceptron(rng, n_in = 28*28 +
    1, n_hidden = 50, n_out = 10,
        activation = ReLu, activationGradient =
            ReLuGradient, learningRate = 10**(-6))
test_MLP(MLPReLu, trainImage, trainLabel,
    validImage, validLabel, testImage, testLabel,"
    ReLu.png", nEpochs=14)

#(H)
MLPHalf = MultiLayerPerceptron(rng, n_in = 28*28 +
    1, n_hidden = 25, n_out = 10,
        activation = sigmoid, activationGradient =
            sigmoidGradient, learningRate = 10**(-6)
            )
```

```
        test_MLP(MLPHalf, trainImage, trainLabel,
           validImage, validLabel, testImage, testLabel,"
           HalfHiddenUnits.png")

        MLPDouble = MultiLayerPerceptron(rng, n_in = 28*28
           + 1, n_hidden = 100, n_out = 10,
                activation = sigmoid, activationGradient =
                   sigmoidGradient, learningRate = 10**(-6)
                   )
        test_MLP(MLPDouble, trainImage, trainLabel,
           validImage, validLabel, testImage, testLabel,"
           DoubleHiddenUnits.png")

################ twoHiddenLayerNN.py ################
import numpy as np
from FunctionGradient import SoftMaxFunc, tanh,
   tanhGradient, sigmoid, sigmoidGradient, ReLu,
   ReLuGradient
from DataProcess import load_mnist
import matplotlib.pyplot as plt
import timeit
from multilayerNN import SoftmaxTopLayer,  HiddenLayer,
   test_MLP

class twoHiddenLayerPerceptron(object):
        def __init__(self, rng, n_in, n_hidden1, n_hidden2,
            n_out,
                        activation = sigmoid,
                           activationGradient =
                           sigmoidGradient, learningRate =
                           10**(-5), regularization = 0,
                           momentum = 0):
                self.labelRange = np.array(range(10))

                self.hiddenLayer1 = HiddenLayer(
                        rng = rng, n_in = n_in, n_out =
                           n_hidden1 + 1, activation =
                           activation, activationGradient =
                            activationGradient
                )
```

```
        self.hiddenLayer2 = HiddenLayer(
                rng = rng, n_in = n_hidden1 + 1,
                   n_out = n_hidden2 + 1,
                   activation = activation,
                   activationGradient =
                   activationGradient
        )

        self.softmaxTopLayer = SoftmaxTopLayer(
                rng = rng, n_in = n_hidden2 + 1,
                   n_out = n_out
        )


        self.params = [self.hiddenLayer1.params] +
           [self.hiddenLayer2.params] + [self.
           softmaxTopLayer.params]
        self.learningRate = learningRate
        self.regularization = regularization
        self.momentum = momentum

    def forwardPropagate(self, inputValue, output):
        outputVec = 1*(self.labelRange == output)

        #propagate to the hidden layer, bias
           corresponds to a unit with constant
           ouput 1
        self.hiddenLayer1.inputValue = inputValue
        self.hiddenLayer1.linearOutput = np.dot(
           self.hiddenLayer1.inputValue, self.
           hiddenLayer1.W)
        self.hiddenLayer1.y = self.hiddenLayer1.
           activation(self.hiddenLayer1.
           linearOutput)
        self.hiddenLayer1.y[0] = 1
        self.hiddenLayer1.yPrime = self.
           hiddenLayer1.activationGradient(self.
           hiddenLayer1.linearOutput)
        self.hiddenLayer1.yPrime[0] = 0
```

```
#propagate to the  second hidden layer,
    bias corresponds to a unit with constant
     ouput 1
self.hiddenLayer2.inputValue = self.
    hiddenLayer1.y
self.hiddenLayer2.linearOutput = np.dot(
    self.hiddenLayer2.inputValue, self.
    hiddenLayer2.W)
self.hiddenLayer2.y = self.hiddenLayer2.
    activation(self.hiddenLayer2.
    linearOutput)
self.hiddenLayer2.y[0] = 1
self.hiddenLayer2.yPrime = self.
    hiddenLayer2.activationGradient(self.
    hiddenLayer2.linearOutput)
self.hiddenLayer2.yPrime[0] = 0

#propagate to the top layer, bias
    corresponds to a unit with constant
    ouput 1
self.softmaxTopLayer.inputValue = self.
    hiddenLayer2.y
self.softmaxTopLayer.linearOutput = np.dot(
    self.softmaxTopLayer.inputValue, self.
    softmaxTopLayer.W)
self.softmaxTopLayer.y = SoftMaxFunc(self.
    softmaxTopLayer.linearOutput)
self.softmaxTopLayer.predict = np.argmax(
    self.softmaxTopLayer.y)

self.softmaxTopLayer.delta = outputVec -
    self.softmaxTopLayer.y
self.softmaxTopLayer.weightGradient = self.
    softmaxTopLayer.weightGradient - \
```

```
def backwardPropagate(self):
        self.hiddenLayer2.delta = np.multiply(self.
           hiddenLayer2.yPrime,
                 np.dot(self.softmaxTopLayer.W, self
                    .softmaxTopLayer.delta))

        self.hiddenLayer2.weightGradient = self.
           hiddenLayer2.weightGradient - \
                 np.outer(self.hiddenLayer2.
                    inputValue, self.hiddenLayer2.
                    delta)

        self.hiddenLayer1.delta = np.multiply(self.
           hiddenLayer1.yPrime,
                 np.dot(self.hiddenLayer2.W, self.
                    hiddenLayer2.delta))

        self.hiddenLayer1.weightGradient = self.
           hiddenLayer1.weightGradient - \
                 np.outer(self.hiddenLayer1.
                    inputValue, self.hiddenLayer1.
                    delta)


def updateWeight(self):
        preW = self.softmaxTopLayer.W
        self.softmaxTopLayer.W = preW - \
```

```
                self.learningRate*self.
                    softmaxTopLayer.weightGradient +
                    \
                2*self.regularization*preW + self.
                    momentum*self.softmaxTopLayer.
                    deltaW
        self.softmaxTopLayer.deltaW = self.
            softmaxTopLayer.W - preW



        preHiddenW2 = self.hiddenLayer2.W
        self.hiddenLayer2.W = preHiddenW2 - \
                self.learningRate*self.hiddenLayer2
                    .weightGradient + \
                2*self.regularization*preHiddenW2 +
                     self.momentum*self.hiddenLayer2
                    .deltaW
        self.hiddenLayer2.deltaW = self.
            hiddenLayer2.W - preHiddenW2

        preHiddenW1 = self.hiddenLayer1.W
        self.hiddenLayer1.W = preHiddenW1 - \
                self.learningRate*self.hiddenLayer1
                    .weightGradient + \
                2*self.regularization*preHiddenW1 +
                     self.momentum*self.hiddenLayer1
                    .deltaW
        self.hiddenLayer1.deltaW = self.
            hiddenLayer1.W - preHiddenW1


    def accuracy(self, INPUT, OUTPUT):
        count = 0
        for inputValue, output in zip(INPUT,OUTPUT)
            :
                linearOutputHidden1 = np.dot(
                    inputValue, self.hiddenLayer1.W)
                yHidden1 = self.hiddenLayer1.
                    activation(linearOutputHidden1)
```

```
                    yHidden1[0] = 1

                    linearOutputHidden2 = np.dot(
                        yHidden1, self.hiddenLayer2.W)
                    yHidden2 = self.hiddenLayer2.
                        activation(linearOutputHidden2)
                    yHidden2[0] = 1

                    linearOutput = np.dot(yHidden2,
                        self.softmaxTopLayer.W)
                    y = SoftMaxFunc(linearOutput)
                    predict = np.argmax(y)

                    if predict == output:
                            count = count + 1

            return (float(count) / len(OUTPUT))


if __name__ == '__main__':
        allTrain = load_mnist(dataset="training", path='../
            ')

        trainImage = allTrain[0][0:49999,:]
        trainLabel = allTrain[1][0:49999]

        validImage = allTrain[0][50000:,:]
        validLabel = allTrain[1][50000:]

        allTest = load_mnist(dataset="testing",path='../')
        testImage = allTest[0]
        testLabel = allTest[1]

        rng = np.random.RandomState(1234)

        #(D)
        MLP = twoHiddenLayerPerceptron(rng, n_in = 28*28 +
            1, n_hidden1 = 43, n_hidden2 = 43, n_out = 10,
                activation = sigmoid, activationGradient =
                    sigmoidGradient, learningRate = 10**(-5)
```

```
            )
test_MLP(MLP, trainImage, trainLabel, validImage,
    validLabel, testImage, testLabel,"twoHiddenLayer
    .png")
```