Supporting Information for "Self-adaptive Learning in Memristor Convolutional Neural Networks"

Mingqiang Huang[1], Wei Zhang[1,2], Lunshuai Pan[1], Xuelong Yan[2], Guangchao Zhao[3], Hong Chen[1], Xingli Wang[3], Beng Kang Tay[3], Gaokuo Zhong[1], and Jiangyu Li[1,4]

[1]Research Center for Medical AI, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China
[2]Guangxi Key Laboratory of Automatic Detecting Technology and Instrument, Guilin University of Electronic Technology, Guilin 541004,China
[3]CNRS-International-NTU-THALES-Research-Alliance, Nanyang Technological University, Singapore 639798, Singapore
[4]Department of Materials Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, Guangdong, China

March 8, 2021

Abstract

This Supporting Information includes the simulation code on the training of the nonlinear memristor neural network using the self-adaptive learning method discussed in the main text. The demonstrated network is LeNet-5 and can be easily extended to be larger scale networks. The dataset in this work is the typical Modified National Institute of Standards and Technology handwriting digits dataset that is available at http://yann.lecun.com/exdb/mnist/.

Corresponding Authors

Email: mq.huang2@siat.ac.cn; gk.zhong@siat.ac.cn; lijy@sustech.edu.cn;

Overview of Code (in Python)

Requirements

IDE: PyCharm

Project interpreter: Python3.7 (anaconda3)

Step0: Data preparation.

Download the MNIST data from "http://yann.lecun.com/exdb/mnist/". In order to visualize the data element, we decode the original binary file and save it into .txt format files (see below).

Step1: Load the training images.(In the main function: line 7)

```
training_images = np.loadtxt("training_images60000.txt") # or the 1000 image file
training_labels = np.loadtxt("training_labels.txt")
test_images = np.loadtxt("test_images1000.txt")
test_labels = np.loadtxt("test_labels.txt")
```

Step2: Define the memristor parameters. (In the main function: line 18)

```
Gmax1 = 2               # normalized max conductance of device w1
Gmin1 = 0               # normalized min conductance of device w1
Gmax2 = 1               # normalized max conductance of device w2
Gmin2 = Gmin1*Gmax2/Gmax1   # make sure that Gmin1:Gmax1 == Gmin2:Gmax2
bit = 8             # The conductance state is thus Pmax== 2**bit - 1
LTP_Ap = 1              # non-linearity of the device at LTP
LTP_Ad = -1             # non-linearity of the device at LTD
```

Step3: Construct the memristor neural network (LeNet-5 is used in this demo).

Step4: Define the updating algorithm. (In the main function: line 63)

```
# the updating method can be changed
optim.updating_mode1()
# optim.updating_mode2()
# optim.updating_mode3_W2_random()
# optim.updating_mode4_W2_eq_R()
# The name of "mode1"/"mode2"/"mode3"/"mode4" is the same as that in the main text.
```

Step5: Initialization of the memristor conductance.

Step6: Updating the network with the proposed algorithm.

Copy the following codes (including one supporting part and the main function) into a new python file, copy the train/test image files (namely the prepared TXT files *"training_images60000.txt", "training_labels.txt ","test_images1000.txt","test_labels.txt"*) into the same folder, then click and run. Due the uploading document size limitation, here we only provide the 1000 images file for demostration"

Hosted file

`training_labels.txt` available at https://authorea.com/users/399299/articles/511844-supporting-information-for-self-adaptive-learning-in-memristor-convolutional-neural-networks

Hosted file

`training_images1000.txt` available at https://authorea.com/users/399299/articles/511844-supporting-information-for-self-adaptive-learning-in-memristor-convolutional-neural-networks

Hosted file

`test_labels.txt` available at https://authorea.com/users/399299/articles/511844-supporting-information-for-self-adaptive-learning-in-memristor-convolutional-neural-networks

Hosted file

`test_images1000.txt` available at https://authorea.com/users/399299/articles/511844-supporting-information-for-self-adaptive-learning-in-memristor-convolutional-neural-networks

```
import random
import numpy as np
import time


def im2col(input_data, Ky, Kx, stride=1, pad=0):
    N, CHin, Hin, Win = input_data.shape  # N=batch number
    Hout = (Hin + 2 * pad - Ky) // stride + 1  # output height
    Wout = (Win + 2 * pad - Kx) // stride + 1  # output width
    img = np.pad(input_data, [(0, 0), (0, 0), (pad, pad), (pad, pad)], 'constant')
    col = np.zeros((N, CHin, Ky, Kx, Hout, Wout))  # init (N, C, filter_h, filter_w, out_h, out_w)

    for y in range(Ky):
        y_max = y + stride * Hout
        for x in range(Kx):
            x_max = x + stride * Wout
            col[:, :, y, x, :, :] = img[:, :, y:y_max:stride, x:x_max:stride]

    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N * Hout * Wout, -1)  # image to col transfer
    return col


def col2im(col, input_shape, Ky, Kx, stride=1, pad=0):
    N, CHin, Hin, Win = input_shape
    Hout = (Hin + 2 * pad - Ky) // stride + 1
    Wout = (Win + 2 * pad - Kx) // stride + 1
    col = col.reshape(N, Hout, Wout, CHin, Ky, Kx).transpose(0, 3, 4, 5, 1, 2)
```

```
    img = np.zeros((N, CHin, Hin + 2 * pad + stride - 1, Win + 2 * pad + stride - 1))
    for y in range(Ky):
        y_max = y + stride * Hout
        for x in range(Kx):
            x_max = x + stride * Wout
            img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]

    return img[:, :, pad:Hin + pad, pad:Win + pad]


def MakeOneHot(Y, N, CHout):
    Z = np.zeros((N, CHout))
    Z[np.arange(N), Y] = 1
    return Z


def get_batch(X, Y, N, batch_size):
    i = random.randint(1, N - batch_size)
    return X[784 * i: 784 * (batch_size + i)], Y[i:i + batch_size]


def NLLLoss(Y_pred, Y_true):
    """
    Negative log likelihood loss
    """
    loss = 0.0
    N = Y_pred.shape[0]
    M = np.sum(Y_pred * Y_true, axis=1)
    for e in M:
        # print(e)
        if e == 0:
            loss += 500
        else:
            loss += -np.log(e)
    return loss / N


class Conv:

    def __init__(self, CHin, CHout, Hin, Win, Kx, Ky, stride, pad, Gmax1, Gmin1, Gmax2, Gmin2):
        self.stride = stride
        self.pad = pad
        self.x = None
        self.col = None
        self.col_W = None
        self.Hin = Hin
        self.Win = Win
        self.CHin = CHin
        self.Gmax1 = Gmax1
        self.Gmax2 = Gmax2
        self.Gmin1 = Gmin1
```

```
self.Gmin2 = Gmin2

self.W = {'dat': np.random.normal(0, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)), 'grad': 0,
          'P_dat': np.random.normal(0.5, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          'N_dat': np.random.normal(-0.5, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          'P_matrix': np.random.normal(0, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          'N_matrix': np.random.normal(0, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          'matrix': np.random.normal(0, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          'P_w1_dat': np.random.normal(1, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          'P_w2_dat': np.random.normal(0.5, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          'N_w1_dat': np.random.normal(1, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          'N_w2_dat': np.random.normal(0.5, np.sqrt(2 / CHin), (CHout, CHin, Kx, Ky)),
          }
self.b = {'dat': np.random.normal(0, np.sqrt(2 / CHout), CHout), 'grad': 0,
          'P_dat': np.random.normal(0.5, np.sqrt(2 / CHout), CHout),
          'N_dat': np.random.normal(-0.5, np.sqrt(2 / CHout), CHout),
          'P_matrix': np.random.normal(0, np.sqrt(2 / CHout), CHout),
          'N_matrix': np.random.normal(0, np.sqrt(2 / CHout), CHout),
          'matrix': np.random.normal(0, np.sqrt(2 / CHout), CHout),
          'P_w1_dat': np.random.normal(1, np.sqrt(2 / CHout), CHout),
          'P_w2_dat': np.random.normal(0.5, np.sqrt(2 / CHout), CHout),
          'N_w1_dat': np.random.normal(1, np.sqrt(2 / CHout), CHout),
          'N_w2_dat': np.random.normal(0.5, np.sqrt(2 / CHout), CHout)
          }

self.W['P_matrix'] = np.where(self.W['dat'] >= 0, 1, 0)
self.W['N_matrix'] = np.where(self.W['dat'] < 0, -1, 0)
self.W['matrix'] = self.W['P_matrix'] + self.W['N_matrix']
self.W['P_w1_dat'] = np.clip(self.W['P_w1_dat'], self.Gmin1, self.Gmax1)\
                     * self.W['P_matrix']
self.W['N_w1_dat'] = np.clip(self.W['N_w1_dat'], self.Gmin1, self.Gmax1)\
                     * (-1 * self.W['N_matrix'])
self.W['P_w2_dat'] = np.clip(self.W['P_w2_dat'], self.Gmin2, self.Gmax2)\
                     * self.W['P_matrix']
self.W['N_w2_dat'] = np.clip(self.W['N_w2_dat'], self.Gmin2, self.Gmax2)\
                     * (-1 * self.W['N_matrix'])
self.W['P_dat'] = (self.W['P_w1_dat'] - self.W['P_w2_dat'])
self.W['N_dat'] = (self.W['N_w2_dat'] - self.W['N_w1_dat'])
self.W['dat'] = self.W['P_dat'] + self.W['N_dat']

self.b['P_matrix'] = np.where(self.b['dat'] >= 0, 1, 0)
self.b['N_matrix'] = np.where(self.b['dat'] < 0, -1, 0)
self.b['matrix'] = self.b['P_matrix'] + self.b['N_matrix']
self.b['P_w1_dat'] = np.clip(self.b['P_w1_dat'], self.Gmin1, self.Gmax1)\
                     * self.b['P_matrix']
self.b['N_w1_dat'] = np.clip(self.b['N_w1_dat'], self.Gmin1, self.Gmax1)\
                     * (-self.b['N_matrix'])
self.b['P_w2_dat'] = np.clip(self.b['P_w2_dat'], self.Gmin2, self.Gmax2)\
                     * self.b['P_matrix']
self.b['N_w2_dat'] = np.clip(self.b['N_w2_dat'], self.Gmin2, self.Gmax2)\
                     * (-self.b['N_matrix'])
```

```
        self.b['P_dat'] = (self.b['P_w1_dat'] - self.b['P_w2_dat'])
        self.b['N_dat'] = (self.b['N_w2_dat'] - self.b['N_w1_dat'])
        self.b['dat'] = self.b['P_dat'] + self.b['N_dat']

    def _forward(self, x):
        x = x.reshape(x.shape[0], self.CHin, self.Hin, self.Win)
        FN, CHin, Ky, Kx = self.W['dat'].shape
        N, CHin, Hin, Win = x.shape
        Hout = 1 + int((Hin + 2 * self.pad - Ky) / self.stride)
        Wout = 1 + int((Win + 2 * self.pad - Kx) / self.stride)
        col = im2col(x, Ky, Kx, self.stride, self.pad)
        col_W = self.W['dat'].reshape(FN, -1).T
        out = np.dot(col, col_W) + self.b['dat']
        out = out.reshape(N, Hout, Wout, -1).transpose(0, 3, 1, 2)
        self.x = x
        self.col = col
        self.col_W = col_W
        return out

    def _backward(self, dout):
        FN, CHin, Ky, Kx = self.W['dat'].shape
        dout = dout.transpose(0, 2, 3, 1).reshape(-1, FN)
        db = np.sum(dout, axis=0)
        dW = np.dot(self.col.T, dout)
        dW = dW.transpose(1, 0).reshape(FN, CHin, Ky, Kx)

        self.W['grad'] = dW
        self.b['grad'] = db
        dcol = np.dot(dout, self.col_W.T)
        dx = col2im(dcol, self.x.shape, Ky, Kx, self.stride, self.pad)

        return dx


class MaxPool:
    def __init__(self, Ky, Kx, stride=2, pad=0):
        self.Ky = Ky
        self.Kx = Kx
        self.stride = stride
        self.pad = pad
        self.x = None
        self.arg_max = None

    def _forward(self, x):
        N, CHin, Hin, Win = x.shape
        Hout = int(1 + (Hin - self.Ky) / self.stride)
        Wout = int(1 + (Win - self.Kx) / self.stride)
        col = im2col(x, self.Ky, self.Kx, self.stride, self.pad)
        col = col.reshape(-1, self.Ky * self.Kx)
        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)
```

```
        out = out.reshape(N, Hout, Wout, CHin).transpose(0, 3, 1, 2)
        self.x = x
        self.arg_max = arg_max
        return out

    def _backward(self, dout):
        dout = dout.transpose(0, 2, 3, 1)
        pool_size = self.Ky * self.Kx
        dmax = np.zeros((dout.size, pool_size))
        dmax[np.arange(self.arg_max.size), self.arg_max.flatten()] = dout.flatten()
        dmax = dmax.reshape(dout.shape + (pool_size,))
        dcol = dmax.reshape(dmax.shape[0] * dmax.shape[1] * dmax.shape[2], -1)
        dx = col2im(dcol, self.x.shape, self.Ky, self.Kx, self.stride, self.pad)

        return dx


class FC():
    """
    Fully connected layer
    """

    def __init__(self, CHin, CHout, Gmax1, Gmin1, Gmax2, Gmin2):
        self.cache = None
        self.Gmax1 = Gmax1
        self.Gmax2 = Gmax2
        self.Gmin1 = Gmin1
        self.Gmin2 = Gmin2

        self.W = {'dat': np.random.normal(0, np.sqrt(2 / CHin), (CHin, CHout)), 'grad': 0,
                  'P_dat': np.random.normal(0.5, np.sqrt(2 / CHin), (CHin, CHout)),
                  'N_dat': np.random.normal(-0.5, np.sqrt(2 / CHin), (CHin, CHout)),
                  'P_matrix': np.random.normal(0, np.sqrt(2 / CHin), (CHin, CHout)),
                  'N_matrix': np.random.normal(0, np.sqrt(2 / CHin), (CHin, CHout)),
                  'matrix': np.random.normal(0, np.sqrt(2 / CHin), (CHin, CHout)),
                  'P_w1_dat': np.random.normal(1, np.sqrt(2 / CHin), (CHin, CHout)),
                  'P_w2_dat': np.random.normal(0.5, np.sqrt(2 / CHin), (CHin, CHout)),
                  'N_w1_dat': np.random.normal(1, np.sqrt(2 / CHin), (CHin, CHout)),
                  'N_w2_dat': np.random.normal(0.5, np.sqrt(2 / CHin), (CHin, CHout)),
                  }
        self.b = {'dat': np.random.normal(0, np.sqrt(2 / CHout), CHout), 'grad': 0,
                  'P_dat': np.random.normal(0.5, np.sqrt(2 / CHout), CHout),
                  'N_dat': np.random.normal(-0.5, np.sqrt(2 / CHout), CHout),
                  'P_matrix': np.random.normal(0, np.sqrt(2 / CHout), CHout),
                  'N_matrix': np.random.normal(0, np.sqrt(2 / CHout), CHout),
                  'matrix': np.random.normal(0, np.sqrt(2 / CHout), CHout),
                  'P_w1_dat': np.random.normal(1, np.sqrt(2 / CHout), CHout),
                  'P_w2_dat': np.random.normal(0.5, np.sqrt(2 / CHout), CHout),
                  'N_w1_dat': np.random.normal(1, np.sqrt(2 / CHout), CHout),
                  'N_w2_dat': np.random.normal(0.5, np.sqrt(2 / CHout), CHout),
                  }
```

```
        self.W['P_matrix'] = np.where(self.W['dat'] >= 0, 1, 0)
        self.W['N_matrix'] = np.where(self.W['dat'] < 0, -1, 0)
        self.W['matrix'] = self.W['P_matrix'] + self.W['N_matrix']
        self.W['P_w1_dat'] = np.clip(self.W['P_w1_dat'], self.Gmin1, self.Gmax1)\
                             * self.W['P_matrix']
        self.W['N_w1_dat'] = np.clip(self.W['N_w1_dat'], self.Gmin1, self.Gmax1)\
                             * (-self.W['N_matrix'])
        self.W['P_w2_dat'] = np.clip(self.W['P_w2_dat'], self.Gmin2, self.Gmax2)\
                             * self.W['P_matrix']
        self.W['N_w2_dat'] = np.clip(self.W['N_w2_dat'], self.Gmin2, self.Gmax2)\
                             * (-self.W['N_matrix'])
        self.W['P_dat'] = (self.W['P_w1_dat'] - self.W['P_w2_dat'])
        self.W['N_dat'] = (self.W['N_w2_dat'] - self.W['N_w1_dat'])
        self.W['dat'] = self.W['P_dat'] + self.W['N_dat']

        self.b['P_matrix'] = np.where(self.b['dat'] >= 0, 1, 0)
        self.b['N_matrix'] = np.where(self.b['dat'] < 0, -1, 0)
        self.b['matrix'] = self.b['P_matrix'] + self.b['N_matrix']
        self.b['P_w1_dat'] = np.clip(self.b['P_w1_dat'], self.Gmin1, self.Gmax1)\
                             * self.b['P_matrix']
        self.b['N_w1_dat'] = np.clip(self.b['N_w1_dat'], self.Gmin1, self.Gmax1)\
                             * (-self.b['N_matrix'])
        self.b['P_w2_dat'] = np.clip(self.b['P_w2_dat'], self.Gmin2, self.Gmax2)\
                             * self.b['P_matrix']
        self.b['N_w2_dat'] = np.clip(self.b['N_w2_dat'], self.Gmin2, self.Gmax2)\
                             * (-self.b['N_matrix'])
        self.b['P_dat'] = (self.b['P_w1_dat'] - self.b['P_w2_dat'])
        self.b['N_dat'] = (self.b['N_w2_dat'] - self.b['N_w1_dat'])
        self.b['dat'] = self.b['P_dat'] + self.b['N_dat']

    def _forward(self, X):
        out = np.dot(X, self.W['dat']) + self.b['dat']
        self.cache = X
        return out

    def _backward(self, dout):
        X = self.cache
        dX = np.dot(dout, self.W['dat'].T).reshape(X.shape)
        self.W['grad'] = np.dot(X.reshape(X.shape[0], np.prod(X.shape[1:])).T, dout)
        self.b['grad'] = np.sum(dout, axis=0)
        return dX


class ReLU():
    """
    ReLU activation layer
    """

    def __init__(self):
        self.cache = None
```

```
    def _forward(self, X):
        out = np.maximum(0, X)
        self.cache = X
        return out

    def _backward(self, dout):
        X = self.cache
        dX = np.array(dout, copy=True)
        dX[X <= 0] = 0
        return dX


class Softmax():
    """
    Softmax activation layer
    """

    def __init__(self):
        self.cache = None

    def _forward(self, X):
        maxes = np.amax(X, axis=1)
        maxes = maxes.reshape(maxes.shape[0], 1)
        Y = np.exp(X - maxes)
        Z = Y / np.sum(Y, axis=1).reshape(Y.shape[0], 1)
        self.cache = (X, Y, Z)
        return Z

    def _backward(self, dout):
        X, Y, Z = self.cache
        dZ = np.zeros(X.shape)
        dY = np.zeros(X.shape)
        dX = np.zeros(X.shape)
        N = X.shape[0]
        for n in range(N):
            i = np.argmax(Z[n])
            dZ[n, :] = np.diag(Z[n]) - np.outer(Z[n], Z[n])
            M = np.zeros((N, N))
            M[:, i] = 1
            dY[n, :] = np.eye(N) - M
        dX = np.dot(dout, dZ)
        dX = np.dot(dX, dY)
        return dX


class CrossEntropyLoss():
    def __init__(self):
        pass

    def get(self, Y_pred, Y_true):
```

```
        N = Y_pred.shape[0]
        softmax = Softmax()
        prob = softmax._forward(Y_pred)
        loss = NLLLoss(prob, Y_true)
        Y_serial = np.argmax(Y_true, axis=1)
        dout = prob.copy()
        dout[np.arange(N), Y_serial] -= 1
        return loss, dout


class LeNet5():

    def __init__(self, Gmax1, Gmin1, Gmax2, Gmin2):
        self.Gmax1 = Gmax1
        self.Gmax2 = Gmax2
        self.Gmin1 = Gmin1
        self.Gmin2 = Gmin2
        self.conv1 = Conv(1, 6, 28, 28, 5, 5, 1, 0,
                          self.Gmax1, self.Gmin1, self.Gmax2, self.Gmin2)
        self.ReLU1 = ReLU()
        self.pool1 = MaxPool(2, 2)
        self.conv2 = Conv(6, 16, 12, 12, 5, 5, 1, 0,
                          self.Gmax1, self.Gmin1, self.Gmax2, self.Gmin2)
        self.ReLU2 = ReLU()
        self.pool2 = MaxPool(2, 2)
        self.FC1 = FC(16 * 4 * 4, 120, self.Gmax1, self.Gmin1, self.Gmax2, self.Gmin2)
        self.ReLU3 = ReLU()
        self.FC2 = FC(120, 84, self.Gmax1, self.Gmin1, self.Gmax2, self.Gmin2)
        self.ReLU4 = ReLU()
        self.FC3 = FC(84, 10, self.Gmax1, self.Gmin1, self.Gmax2, self.Gmin2)
        self.Softmax = Softmax()
        self.p2_shape = None

    def forward(self, X):
        h1 = self.conv1._forward(X)
        a1 = self.ReLU1._forward(h1)
        p1 = self.pool1._forward(a1)
        h2 = self.conv2._forward(p1)
        a2 = self.ReLU2._forward(h2)
        p2 = self.pool2._forward(a2)
        self.p2_shape = p2.shape
        fl = p2.reshape(X.shape[0], -1)
        h3 = 0.01 * self.FC1._forward(fl)
        a3 = self.ReLU3._forward(h3)
        h4 = 0.1 * self.FC2._forward(a3)
        a5 = self.ReLU4._forward(h4)
        h5 = 0.5 * self.FC3._forward(a5)
        # a5 = self.Softmax._forward(h5)
        return h5

    def backward(self, dout):
```

```
        # dout = self.Softmax._backward(dout)
        dout = 0.5 * self.FC3._backward(dout)
        dout = self.ReLU4._backward(dout)
        dout = 0.1 * self.FC2._backward(dout)
        dout = self.ReLU3._backward(dout)
        dout = 0.01 * self.FC1._backward(dout)
        dout = dout.reshape(self.p2_shape)  # reshape
        dout = self.pool2._backward(dout)
        dout = self.ReLU2._backward(dout)
        dout = self.conv2._backward(dout)
        dout = self.pool1._backward(dout)
        dout = self.ReLU1._backward(dout)
        dout = self.conv1._backward(dout)

    def get_params(self):
        return [self.conv1.W, self.conv1.b, self.conv2.W, self.conv2.b, self.FC1.W,
                self.FC1.b, self.FC2.W, self.FC2.b,self.FC3.W, self.FC3.b]

    def set_params(self, params):
        [self.conv1.W, self.conv1.b, self.conv2.W, self.conv2.b, self.FC1.W,
         self.FC1.b, self.FC2.W, self.FC2.b,self.FC3.W, self.FC3.b] = params


class SGD():
    def __init__(self, params, Ap, Bp, Ad, Bd, Pmax, Gmax1, Gmax2, Gmin1, Gmin2):
        self.parameters = params
        self.Ap = Ap
        self.Bp = Bp
        self.Ad = Ad
        self.Bd = Bd
        self.Pmax = Pmax
        self.Gmax1 = Gmax1
        self.Gmax2 = Gmax2
        self.Gmin1 = Gmin1
        self.Gmin2 = Gmin2
        self.G1 = self.Gmax1 - self.Gmin1
        self.G2 = self.Gmax2 - self.Gmin2
        self.D = self.Ad / self.Pmax
        self.P = self.Ap / self.Pmax

    def updating_mode1(self):
        for param in self.parameters:
            param['grad'][param['grad'] > 0] = 1
            param['grad'][param['grad'] < 0] = -1

            P_W_grad = param['P_matrix'] * param['grad']
            P_W_grad_p = np.where(P_W_grad > 0, 1, 0)
            P_W_grad_n = np.where(P_W_grad < 0, -1, 0)

            P_W1_dat_p_old = param['P_w1_dat'] * P_W_grad_p
            P_W1_dat_p_new = P_W1_dat_p_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6) \
```

```
                        * (self.G1 * self.Bd * P_W_grad_p - (P_W1_dat_p_old - self.Gmin1))
            P_W2_dat_p_old = param['P_w2_dat'] * P_W_grad_p
            P_W2_dat_p_new = P_W2_dat_p_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                        * (self.G2 * self.Bd * P_W_grad_p - (P_W2_dat_p_old - self.Gmin2))

            P_W1_dat_n_old = param['P_w1_dat'] * (-P_W_grad_n)
            P_W1_dat_n_new = P_W1_dat_n_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                        * (self.G1 * self.Bp * (-P_W_grad_n) - (P_W1_dat_n_old - self.Gmin1))
            P_W2_dat_n_old = param['P_w2_dat'] * (-P_W_grad_n)
            P_W2_dat_n_new = P_W2_dat_n_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                        * (self.G2 * self.Bp * (-P_W_grad_n) - (P_W2_dat_n_old - self.Gmin2))

            N_W_grad = -1 * param['N_matrix'] * param['grad']
            N_W_grad_p = np.where(N_W_grad > 0, 1, 0)
            N_W_grad_n = np.where(N_W_grad < 0, -1, 0)

            N_W1_dat_p_old = param['N_w1_dat'] * N_W_grad_p
            N_W1_dat_p_new = N_W1_dat_p_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                        * (self.G1 * self.Bp * N_W_grad_p - (N_W1_dat_p_old - self.Gmin1))
            N_W2_dat_p_old = param['N_w2_dat'] * N_W_grad_p
            N_W2_dat_p_new = N_W2_dat_p_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                        * (self.G2 * self.Bp * N_W_grad_p - (N_W2_dat_p_old - self.Gmin2))

            N_W1_dat_n_old = param['N_w1_dat'] * (-N_W_grad_n)
            N_W1_dat_n_new = N_W1_dat_n_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                        * (self.G1 * self.Bd * (-N_W_grad_n) - (N_W1_dat_n_old - self.Gmin1))
            N_W2_dat_n_old = param['N_w2_dat'] * (-N_W_grad_n)
            N_W2_dat_n_new = N_W2_dat_n_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                        * (self.G2 * self.Bd * (-N_W_grad_n) - (N_W2_dat_n_old - self.Gmin2))

            param['P_w1_dat'] = P_W1_dat_p_new + P_W1_dat_n_new
            param['P_w1_dat'] = np.clip(param['P_w1_dat'], self.Gmin1, self.Gmax1)
            param['P_w2_dat'] = P_W2_dat_p_new + P_W2_dat_n_new
            param['P_w2_dat'] = np.clip(param['P_w2_dat'], self.Gmin2, self.Gmax2)
            param['N_w1_dat'] = N_W1_dat_p_new + N_W1_dat_n_new
            param['N_w1_dat'] = np.clip(param['N_w1_dat'], self.Gmin1, self.Gmax1)
            param['N_w2_dat'] = N_W2_dat_p_new + N_W2_dat_n_new
            param['N_w2_dat'] = np.clip(param['N_w2_dat'], self.Gmin2, self.Gmax2)
            param['N_dat'] = param['N_w2_dat'] - param['N_w1_dat']
            param['P_dat'] = param['P_w1_dat'] - param['P_w2_dat']

            param['dat'] = param['P_dat'] + param['N_dat']

    def updating_mode2(self):
        for param in self.parameters:
            param['grad'][param['grad'] > 0] = 1
            param['grad'][param['grad'] < 0] = -1

            P_W_grad = param['P_matrix'] * param['grad']
            P_W_grad_p = np.where(P_W_grad > 0, 1, 0)
            P_W_grad_n = np.where(P_W_grad < 0, -1, 0)
```

```
            P_W1_dat_p_old = param['P_w1_dat'] * P_W_grad_p
            P_W1_dat_p_new = P_W1_dat_p_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                             * (self.G1 * self.Bd * P_W_grad_p - (P_W1_dat_p_old - self.Gmin1))
            P_W2_dat_p_old = param['P_w2_dat'] * P_W_grad_p
            P_W2_dat_p_new = P_W2_dat_p_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                             * (self.G2 * self.Bp * P_W_grad_p - (P_W2_dat_p_old - self.Gmin2))

            P_W1_dat_n_old = param['P_w1_dat'] * (-P_W_grad_n)
            P_W1_dat_n_new = P_W1_dat_n_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                             * (self.G1 * self.Bp * (-P_W_grad_n) - (P_W1_dat_n_old - self.Gmin1))
            P_W2_dat_n_old = param['P_w2_dat'] * (-P_W_grad_n)
            P_W2_dat_n_new = P_W2_dat_n_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                             * (self.G2 * self.Bd * (-P_W_grad_n) - (P_W2_dat_n_old - self.Gmin2))

            N_W_grad = -1 * param['N_matrix'] * param['grad']
            N_W_grad_p = np.where(N_W_grad > 0, 1, 0)
            N_W_grad_n = np.where(N_W_grad < 0, -1, 0)

            N_W1_dat_p_old = param['N_w1_dat'] * N_W_grad_p
            N_W1_dat_p_new = N_W1_dat_p_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                             * (self.G1 * self.Bp * N_W_grad_p - (N_W1_dat_p_old - self.Gmin1))
            N_W2_dat_p_old = param['N_w2_dat'] * N_W_grad_p
            N_W2_dat_p_new = N_W2_dat_p_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                             * (self.G2 * self.Bd * N_W_grad_p - (N_W2_dat_p_old - self.Gmin2))

            N_W1_dat_n_old = param['N_w1_dat'] * (-N_W_grad_n)
            N_W1_dat_n_new = N_W1_dat_n_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                             * (self.G1 * self.Bd * (-N_W_grad_n) - (N_W1_dat_n_old - self.Gmin1))
            N_W2_dat_n_old = param['N_w2_dat'] * (-N_W_grad_n)
            N_W2_dat_n_new = N_W2_dat_n_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                             * (self.G2 * self.Bp * (-N_W_grad_n) - (N_W2_dat_n_old - self.Gmin2))

            param['P_w1_dat'] = P_W1_dat_p_new + P_W1_dat_n_new
            param['P_w1_dat'] = np.clip(param['P_w1_dat'], self.Gmin1, self.Gmax1)
            param['P_w2_dat'] = P_W2_dat_p_new + P_W2_dat_n_new
            param['P_w2_dat'] = np.clip(param['P_w2_dat'], self.Gmin2, self.Gmax2)
            param['N_w1_dat'] = N_W1_dat_p_new + N_W1_dat_n_new
            param['N_w1_dat'] = np.clip(param['N_w1_dat'], self.Gmin1, self.Gmax1)
            param['N_w2_dat'] = N_W2_dat_p_new + N_W2_dat_n_new
            param['N_w2_dat'] = np.clip(param['N_w2_dat'], self.Gmin2, self.Gmax2)
            param['N_dat'] = param['N_w2_dat'] - param['N_w1_dat']
            param['P_dat'] = param['P_w1_dat'] - param['P_w2_dat']

            param['dat'] = param['P_dat'] + param['N_dat']

    def updating_mode3_W2_random(self):
        for param in self.parameters:
            param['grad'][param['grad'] > 0] = 1
            param['grad'][param['grad'] < 0] = -1
```

```
            P_W_grad = param['P_matrix'] * param['grad']
            P_W_grad_p = np.where(P_W_grad > 0, 1, 0)
            P_W_grad_n = np.where(P_W_grad < 0, -1, 0)

            P_W1_dat_p_old = param['P_w1_dat'] * P_W_grad_p
            P_W1_dat_p_new = P_W1_dat_p_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                             * (self.G1 * self.Bd * P_W_grad_p - (P_W1_dat_p_old - self.Gmin1))
            P_W2_dat_p_old = param['P_w2_dat'] * P_W_grad_p
            P_W2_dat_p_new = P_W2_dat_p_old

            P_W1_dat_n_old = param['P_w1_dat'] * (-P_W_grad_n)
            P_W1_dat_n_new = P_W1_dat_n_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                             * (self.G1 * self.Bp * (-P_W_grad_n) - (P_W1_dat_n_old - self.Gmin1))
            P_W2_dat_n_old = param['P_w2_dat'] * (-P_W_grad_n)
            P_W2_dat_n_new = P_W2_dat_n_old

            N_W_grad = -1 * param['N_matrix'] * param['grad']
            N_W_grad_p = np.where(N_W_grad > 0, 1, 0)
            N_W_grad_n = np.where(N_W_grad < 0, -1, 0)

            N_W1_dat_p_old = param['N_w1_dat'] * N_W_grad_p
            N_W1_dat_p_new = N_W1_dat_p_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                             * (self.G1 * self.Bp * N_W_grad_p - (N_W1_dat_p_old - self.Gmin1))
            N_W2_dat_p_old = param['N_w2_dat'] * N_W_grad_p
            N_W2_dat_p_new = N_W2_dat_p_old

            N_W1_dat_n_old = param['N_w1_dat'] * (-N_W_grad_n)
            N_W1_dat_n_new = N_W1_dat_n_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                             * (self.G1 * self.Bd * (-N_W_grad_n) - (N_W1_dat_n_old - self.Gmin1))
            N_W2_dat_n_old = param['N_w2_dat'] * (-N_W_grad_n)
            N_W2_dat_n_new = N_W2_dat_n_old

            param['P_w1_dat'] = P_W1_dat_p_new + P_W1_dat_n_new
            param['P_w1_dat'] = np.clip(param['P_w1_dat'], self.Gmin1, self.Gmax1)
            param['P_w2_dat'] = P_W2_dat_p_new + P_W2_dat_n_new
            param['P_w2_dat'] = np.clip(param['P_w2_dat'], self.Gmin2, self.Gmax2)
            param['N_w1_dat'] = N_W1_dat_p_new + N_W1_dat_n_new
            param['N_w1_dat'] = np.clip(param['N_w1_dat'], self.Gmin1, self.Gmax1)
            param['N_w2_dat'] = N_W2_dat_p_new + N_W2_dat_n_new
            param['N_w2_dat'] = np.clip(param['N_w2_dat'], self.Gmin2, self.Gmax2)
            param['N_dat'] = param['N_w2_dat'] - param['N_w1_dat']
            param['P_dat'] = param['P_w1_dat'] - param['P_w2_dat']

            param['dat'] = param['P_dat'] + param['N_dat']

    def updating_mode4_W2_eq_R(self):
        for param in self.parameters:
            param['grad'][param['grad'] > 0] = 1
            param['grad'][param['grad'] < 0] = -1

            P_W_grad = param['P_matrix'] * param['grad']
```

```
            P_W_grad_p = np.where(P_W_grad > 0, 1, 0)
            P_W_grad_n = np.where(P_W_grad < 0, -1, 0)

            P_W1_dat_p_old = param['P_w1_dat'] * P_W_grad_p
            P_W1_dat_p_new = P_W1_dat_p_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                             * (self.G1 * self.Bd * P_W_grad_p - (P_W1_dat_p_old - self.Gmin1))

            P_W1_dat_n_old = param['P_w1_dat'] * (-P_W_grad_n)
            P_W1_dat_n_new = P_W1_dat_n_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6)\
                             * (self.G1 * self.Bp * (-P_W_grad_n) - (P_W1_dat_n_old - self.Gmin1))

            N_W_grad = -1 * param['N_matrix'] * param['grad']
            N_W_grad_p = np.where(N_W_grad > 0, 1, 0)
            N_W_grad_n = np.where(N_W_grad < 0, -1, 0)

            N_W1_dat_p_old = param['N_w1_dat'] * N_W_grad_p
            N_W1_dat_p_new = N_W1_dat_p_old + (self.P - (self.P ** 2) / 2 + (self.P ** 3) / 6) \
                             * (self.G1 * self.Bp * N_W_grad_p - (N_W1_dat_p_old - self.Gmin1))

            N_W1_dat_n_old = param['N_w1_dat'] * (-N_W_grad_n)
            N_W1_dat_n_new = N_W1_dat_n_old - (self.D + (self.D ** 2) / 2 + (self.D ** 3) / 6)\
                             * (self.G1 * self.Bd * (-N_W_grad_n) - (N_W1_dat_n_old - self.Gmin1))

            param['P_w1_dat'] = P_W1_dat_p_new + P_W1_dat_n_new
            param['P_w1_dat'] = np.clip(param['P_w1_dat'], self.Gmin1, self.Gmax1)
            param['P_w2_dat'] = 0.5

            param['N_w1_dat'] = N_W1_dat_p_new + N_W1_dat_n_new
            param['N_w1_dat'] = np.clip(param['N_w1_dat'], self.Gmin1, self.Gmax1)
            param['N_w2_dat'] = 0.5

            param['N_dat'] = param['N_w2_dat'] - param['N_w1_dat']
            param['P_dat'] = param['P_w1_dat'] - param['P_w2_dat']

            param['dat'] = param['P_dat'] + param['N_dat']


if __name__ == '__main__':
    print("start!")
    print("Loading the training images! The 12000 iamges may need 30 seconds, "
          "and the 60000 images may need 180 seconds!")
    print("Due the uploading document size limitation, "
          "here we provide the 1000 training images file for the demostration")
    start_time = time.time()

    training_images = np.loadtxt("training_images1000.txt")
    training_labels = np.loadtxt("training_labels.txt")
    test_images = np.loadtxt("test_images1000.txt")
    test_labels = np.loadtxt("test_labels.txt")
    training_images_numbers = int(len(training_images) / 784)
    test_images_numbers = int(len(test_images) / 784)
```

```
load_image_time = time.time()
print("load_image_time is %4f second" % (load_image_time - start_time))
print("*****************")

# ====== the following parameters can be changed
Gmax1 = 2
Gmin1 = 0
Gmax2 = 1
Gmin2 = Gmin1 * Gmax2 / Gmax1  # make sure that Gmin1:Gmax1 == Gmin2:Gmax2
bit = 8
LTP_Ap = 1  # non-linearity of the device at LTP
LTP_Ad = -1  # non-linearity of the device at LTD

batch_size = 128
iteration = 2000
# ====== the above parameters can be changed

LTP_Bp = 1 / (1 - np.exp(-LTP_Ap))
LTP_Bd = 1 / (1 - np.exp(-LTP_Ad))
Pmax = int(2 ** bit - 1)
print("Device: LTP_Ap= " + str(LTP_Ap) + ", LTP_Ad=" + str(LTP_Ad) + ", Pmax=" + str(Pmax))

# mnist.init()
X_train = training_images
Y_train = training_labels
X_test = test_images
Y_test = test_labels
X_train, X_test = X_train / float(255), X_test / float(255)

losses = []
model = LeNet5(Gmax1, Gmin1, Gmax2, Gmin2)
optim = SGD(model.get_params(),
            LTP_Ap, LTP_Bp, LTP_Ad, LTP_Bd, Pmax, Gmax1, Gmax2, Gmin1, Gmin2)
criterion = CrossEntropyLoss()

print("start training!")
print("the number of training images is", training_images_numbers)
print("total iteration is " + str(iteration))

for i in range(iteration + 1):
    # get batch, make onehot
    X_batch, Y_batch = get_batch(X_train, Y_train, training_images_numbers, batch_size)
    Y_batch = (Y_batch).astype(np.int)
    Y_batch = MakeOneHot(Y_batch, batch_size, 10)
    # forward, loss, backward, updating_mode
    X_batch = X_batch.reshape(batch_size, 784)
    Y_pred = model.forward(X_batch)
    loss, dout = criterion.get(Y_pred, Y_batch)
    model.backward(dout)
```

```
        # ====== the updating mode can be changed
        optim.updating_mode1()
        # optim.updating_mode2()
        # optim.updating_mode3_W2_random()
        # optim.updating_mode4_W2_eq_R()
        # ====== the above mode can be changed mode1,mode2,mode3,mode4

        if i % 10 == 0:
            losses.append(loss)
            test_size = int(test_images_numbers)
            x = 0
            for j in range(test_size):
                X_test_min = X_test[784 * j:784 * j + 784]
                X_test_min = X_test_min.reshape(1, 1, 28, 28)
                Y_test_min = Y_test[j]
                Y_pred_min = model.forward(X_test_min)
                result = np.argmax(Y_pred_min, axis=1)

                if (result == Y_test_min):
                    x = x + 1
            predict_rate = (x / test_images_numbers)
            print("iter: %s, \t loss: %5f, \t recognition_accuracy: %4f" % (i, loss, predict_rate))

    train_time = time.time()
    print("the total training time is %4f second" % (train_time - load_image_time))
```