CSE 5526 Programming Assignment 1

Summary Report

In this project, I implemented a two-layer perceptron with the backpropagation algorithm to solve the parity problem. As required, my backpropagation neural network consists of 4 input nodes, 4 hidden nodes as the first layer and one output node for the second layer. The weights and biases of the neural network are initialized to random numbers between -1 and 1. And logistic sigmoid function with a=1 is used as the activation function for all units.

Additionally, online update is applied in my neural network, in which weight adjustment occurs after the presentation of each pattern.

The input nodes only have two possible values, 0 or 1. So, for this problem, there are 2.^4=16 input patterns in total and each one corresponds to a desired output value of 0 or 1 according to the number of 1 in the input pattern. These 16 pairs of input and output data are used here as the training data.

The simplified structure of the neural network can be shown like this:

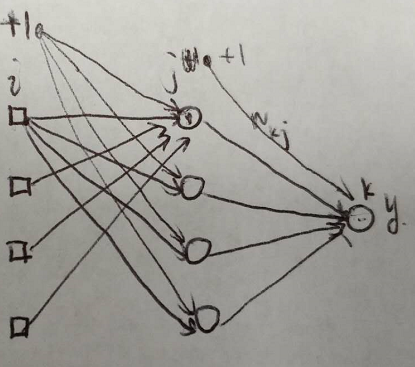


Figure 1. Simplified neural network structure

In this project, I used the learning rate from 0.05 to 0.5 with increment of 0.05 to conduct the experiments. In general, we can see that the number of epochs for training is decreasing as the value of learning rate increases, which means that the neural network with large learning rate can converge faster than the network with small learning rate. The results are shown in the following Table 1.

There is a function Weight\_update() in my code which has a parameter, alpha. It is used for adding momentum learning rate control for the neural network. I also did the experiment with momentum (alpha=0.9) in this project for the 10 values of learning rate from 0.05 to 0.5. The results are shown in Table 2 below. Similarly, it can be seen that the number of epochs before stopping decreases with the increasing of learning rate value in general. And comparing with the results of experiments without momentum, the number of epochs needed for training with momentum is much smaller than the one without momentum.

Table 1. Experiment results for different learning rate without momentum

|  |  |
| --- | --- |
| Value of learning rate | Number of epochs |
| 0.05 | 918300 |
| 0.1 | 515251 |
| 0.15 | 407117 |
| 0.2 | 9220 |
| 0.25 | 7648 |
| 0.3 | 55524 |
| 0.35 | 34082 |
| 0.4 | 28556 |
| 0.45 | 25098 |
| 0.5 | 22785 |

Table 2. Experiment results for different learning rate with momentum (alpha=0.9)

|  |  |
| --- | --- |
| Value of learning rate | Number of epochs |
| 0.05 | 162251 |
| 0.1 | 81329 |
| 0.15 | 54327 |
| 0.2 | 40690 |
| 0.25 | 31597 |
| 0.3 | 26453 |
| 0.35 | 2198 |
| 0.4 | 22702 |
| 0.45 | 20648 |
| 0.5 | 169182 |

One thing needed to be noticed is that, there are several outliers of number of epochs in the two tables, for example, the number of epochs for 0.2 learning rate in table 1 and the number for 0.35 learning rate for table 2. I think that is because the randomness of online learning. The online learning rate updates weights after the presentation of all of the 16 input patterns and the order of input patterns are shuffled, which brings uncertainty to the process of converge.

Source Program:

#### CSE 5526 Neural Networks Programming Assignment 1 ####

### import basic package in python ###

import random

import math

import numpy as np

import itertools

### Define the classes needed for the project ###

class InputNode:

def \_\_init\_\_(self,data,WeightList):

# data is the input data

# weightlist is the weight for each of the next layer node

self.data=data

self.WeightList=WeightList

def \_\_getdata\_\_(self): # get the input data of this input node

return self.data

def \_\_getWeight\_\_(self,i):

# get weight between the node and ith node in next layer

return self.WeightList[i]

class HiddenNode:

# for this project, there is only one hidden layer

# no need to specify the index

def \_\_init\_\_(self,bias,WeightList):

self.bias=bias

self.WeightList=WeightList

def \_\_getbias\_\_(self): # get the input data of this input node

return self.bias

def \_\_getWeight\_\_(self,i):

# get weight between the node and ith node in next layer

return self.WeightList[i]

class OutputNode:

def \_\_init\_\_(self,bias,output=None):

self.bias=bias

self.output=output

def \_\_getbias\_\_(self): # get the input data of this input node

return self.bias

def \_\_getoutput\_\_(self):

# get weight between the node and ith node in next layer

return self.output

### Define the functions needed for the project ###

# activation function

def LogFunc(x):

try:

ans = 1.0/(1+math.exp(-x))

except OverflowError:

if x <-100:

return 0

elif x>100:

return 1

return ans

# derivative of activation function

def LogFuncDer(x):

return LogFunc(x) \* (1 - LogFunc(x))

# Cost/loss function

def CostFunction(act\_output,exp\_output):

ESquare=((act\_output-exp\_output)\*\*2)\*0.5

return ESquare

# get the expected output based on the input data

def Exp\_output(input\_vec):

input\_vec = input\_vec.reshape(4,)

exp\_output = list(input\_vec.reshape(4,)).count(1) % 2

return exp\_output

# BackPropogation between output layer and hidden layer

def BackProp\_op2hi(yj, w\_hi2op, exp\_output, act\_output):

vk = sum(np.multiply(w\_hi2op.reshape(4,), yj.reshape(4,)))

dw\_hi2op = -(exp\_output-act\_output) \* LogFuncDer(vk+b\_op) \* yj

db\_op = -(exp\_output-act\_output) \* LogFuncDer(vk + b\_op) \* 1

return (dw\_hi2op, db\_op)

# BackPropogation between hidden layer and input layer

def BackProp\_hi2in(input\_vec, w\_in2hi, b\_hi, yj, w\_hi2op):

vj = np.dot(w\_in2hi, input\_vec) + b\_hi

vj\_prime = np.array([-LogFuncDer(vj[0]), -LogFuncDer(vj[1]), -LogFuncDer(vj[2]), -LogFuncDer(vj[3])]).reshape(4,1)

vk = sum(np.multiply(w\_hi2op.reshape(4,), yj.reshape(4,)))

delta = (exp\_output-act\_output) \* LogFuncDer(vk + b\_op)

dw\_in2hi = np.outer(np.multiply(vj\_prime, delta\*w\_hi2op), input\_vec)

db\_hi = np.outer(np.multiply(vj\_prime, delta\*w\_hi2op), 1)

assert np.shape(dw\_in2hi) == np.shape(w\_in2hi) == (4,4)

assert np.shape(db\_hi) == np.shape(b\_hi) == (4,1)

return (dw\_in2hi, db\_hi)

# The whole backpropogation

def BackProp(input\_vec, act\_output, w\_in2hi, b\_hi, w\_hi2op, b\_op):

(\_, yj) = Forward\_in2hi(input\_vec, w\_in2hi, b\_hi)

act\_output = Forward\_hi2op(w\_hi2op, b\_op, yj)

Forward(input\_vec, w\_in2hi, w\_hi2op, b\_hi, b\_op)

(dw\_hi2op, db\_op) = BackProp\_op2hi(yj, w\_hi2op, exp\_output, act\_output)

(dw\_in2hi, db\_hi) = BackProp\_hi2in(input\_vec, w\_in2hi, b\_hi, yj, w\_hi2op)

return (dw\_in2hi, dw\_hi2op, db\_hi, db\_op)

# Forward Process between input and hidden layer

def Forward\_in2hi(input\_vec, w\_in2hi, b\_hi):

input\_vec=input\_vec.reshape(4,1)

tmp = np.dot(w\_in2hi, input\_vec) + b\_hi

tmp = tmp.reshape(4,)

result = [LogFunc(tmp[0]), LogFunc(tmp[1]), LogFunc(tmp[2]), LogFunc(tmp[3])]

result = np.array(result).reshape(4,1)

return (tmp, result)

# Forward Process between hidden and output layer

def Forward\_hi2op(w\_hi2op, b\_op, yj):

yj=yj.reshape(4,1)

w\_hi2op = w\_hi2op.reshape(4,)

yj = yj.reshape(4,)

tmp = np.sum(w\_hi2op \* yj) + b\_op

return LogFunc(tmp)

# The whole forward process

def Forward(input\_vec, w\_in2hi, w\_hi2op, b\_hi, b\_op):

(\_, yj) = Forward\_in2hi(input\_vec, w\_in2hi, b\_hi)

act\_output = Forward\_hi2op(w\_hi2op, b\_op, yj)

return act\_output

# Update the weights based on several parameters

def Weight\_update(w\_in2hi, b\_hi, w\_hi2op, b\_op, dw\_in2hi, dw\_hi2op, db\_hi, db\_op, dw\_in2hi\_, dw\_hi2op\_, db\_hi\_, db\_op\_, gama=0.05,alpha=0.9):

dw\_in2hi = gama \* dw\_in2hi + alpha \* dw\_in2hi\_

dw\_hi2op = gama \* dw\_hi2op + alpha \* dw\_hi2op\_

db\_hi = gama \* db\_hi + alpha\* db\_hi\_

db\_op = gama \* db\_op + alpha \* db\_op\_

return (w\_in2hi-dw\_in2hi, b\_hi-db\_hi, w\_hi2op-dw\_hi2op, b\_op- db\_op, dw\_in2hi, db\_hi, dw\_hi2op, db\_op)

# Form the format of input and output data pair following the defined order

def Data\_pair():

data = []

for input\_vec in itertools.product([0, 1], [0, 1], [0, 1], [0, 1]):

input\_vec = np.array(input\_vec)

data +=[(input\_vec.reshape(4,1), Exp\_output(input\_vec))]

return data

# Form the format of input and output data pair randomly

def Data\_pair\_shuffle():

data = []

for input\_vec in itertools.product([0, 1], [0, 1], [0, 1], [0, 1]):

input\_vec = np.array(input\_vec)

data +=[(input\_vec.reshape(4,1), Exp\_output(input\_vec))]

random.shuffle(data)

return data

def initialize\_gradient():

dw\_in2hi = np.zeros((4, 4))

db\_hi = np.zeros((4, 1))

dw\_hi2op = np.zeros((4, 1))

db\_op = 0

return (dw\_in2hi, db\_hi, dw\_hi2op, db\_op)

### Main program ###

# Initialize the data\_pair which is used for error test

data\_pair=Data\_pair()

gama\_list=[(0.5-0.05\*i) for i in range(10)]

for gama in gama\_list:

np.random.seed(42)

input\_vec = np.random.choice([0,1],4).reshape(4,1)

w\_in2hi = np.random.rand(4,4)\*2-1

b\_hi = np.random.rand(4,1)\*2-1

w\_hi2op = np.random.rand(4,1)\*2-1

b\_op = np.random.rand()\*2-1

exp\_output = Exp\_output(input\_vec)

new\_update = initialize\_gradient()

index\_epoch=0

while True:

cost\_total = 0

dw\_in2hi = np.zeros((4, 4))

db\_hi = np.zeros((4, 1))

dw\_hi2op = np.zeros((4, 1))

db\_op = 0

data\_pair\_shuffle=Data\_pair\_shuffle()

# randomize the order of input patterns for online learning

for (input\_vec, exp\_output) in data\_pair\_shuffle:

act\_output = Forward(input\_vec, w\_in2hi, w\_hi2op, b\_hi, b\_op)

(dw\_in2hi, dw\_hi2op, db\_hi, db\_op) = BackProp(input\_vec, exp\_output, w\_in2hi, b\_hi, w\_hi2op, b\_op)

old\_update = new\_update

(dw\_in2hi\_, db\_hi\_, dw\_hi2op\_, db\_op\_) = old\_update

# determine the value of alpha, when alpha is 0.9, the learning rate is controlled by momentum

# when alpha is 0, the learning rate is not controlled by momentum

(w\_in2hi, b\_hi, w\_hi2op, b\_op, dw\_in2hi, db\_hi, dw\_hi2op, db\_op) = Weight\_update(w\_in2hi, b\_hi, w\_hi2op, b\_op, dw\_in2hi, dw\_hi2op, db\_hi, db\_op, dw\_in2hi\_, dw\_hi2op\_, db\_hi\_, db\_op\_, gama=gama,alpha=0)

new\_update=(dw\_in2hi, dw\_hi2op, db\_hi, db\_op)

# calculate the value of cost function for each of the input pattern.

ek=np.array([])

for (input\_vec, exp\_output) in data\_pair:

act\_output = Forward(input\_vec, w\_in2hi, w\_hi2op, b\_hi, b\_op)

cost\_total += CostFunction(act\_output, exp\_output)

ek=np.append(ek,[exp\_output-act\_output])

index\_epoch=index\_epoch+1

is\_abs\_err\_small=abs(ek)<=0.05

if not (False in is\_abs\_err\_small):

print('Learning rate is %s, number of epochs %s\n '%(gama, index\_epoch))

print(ek)

break

# if index\_epoch % 100 == 0:

# print('Epoch is %s: Total Loss is %s' % (int(index\_epoch), cost\_total))