MSc Electronic and Computer Engineering

Data Mining and Machine Learning (2019)

Lab 4 – Neural Networks

Group 12

Miaoyu Niu (1893824)

Kai Chen (1948361)

2019-3-18

UNIVERSITY OF BIRMINGHAM

Neuron model

Before starting this lab, I will first review the basics, because I believe that the basic knowledge is solid, in order to improve and understand the knowledge faster. In 1943, McCulloch and Pitts proposed the "M-P neuron model" as shown below:

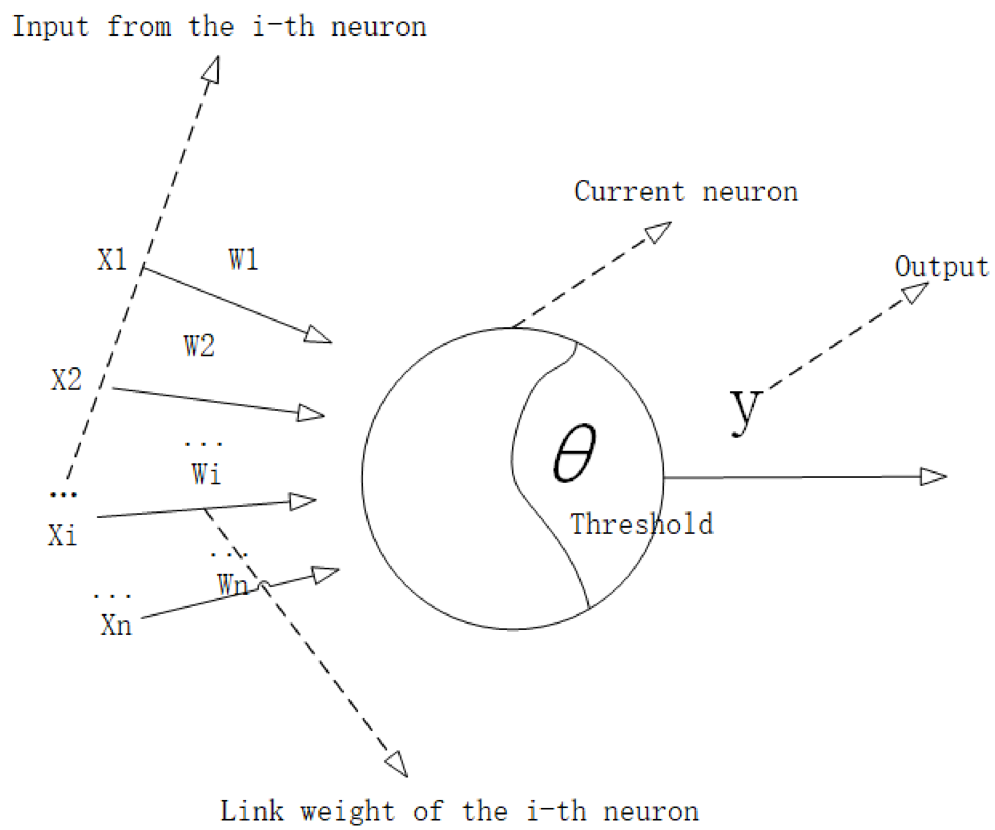

Figure.1 M-P neuron model

In this model, neurons receive input signals from n other neurons that are passed through a weighted connection. The total input value received by the neuron is compared to the threshold of the neuron. The comparison is then processed by the "activation function" to produce the output of the neuron.

Error Back-Propagation algorithm

The error back propagation algorithm is the most successful neural network algorithm to date. When using neural networks in the display task, most of them are trained using the BP algorithm. Not only for multi-layer feedforward neural networks, but also for other types of neural networks.

The input instance is first provided to the input layer neurons, and then the signal is forwarded layer by layer until the result of the output layer is generated; Then calculate the error of the output layer, and then propagate the error back to the hidden layer neurons; Finally, the connection weight and threshold are adjusted according to the error of the hidden layer neurons. The iterative process is repeated until some stop condition position is reached (for example, the training error has reached a small value).

Lab task

The lab is to implement the EBP training algorithm for a multilayer perceptron 4-2-4 encoder. The magnitude of netj becomes large, but the values 0 and 1 are never realised. Hence for practical purposes it is better to replace, for example, 1, 0, 0, 0 in Table 1 with 0.9, 0.1, 0.1, 0.1. In this task, I choose python to implement my code.

```
def demo():
    # Teach network XOR function
    pat = [ [[0.9,0.1,0.1,0.1], [0.9,0.1,0.1,0.1]],
        [[0.9,0.1,0.1,0.1], [0.9,0.1,0.1,0.1]],
        [[0.9,0.1,0.1,0.1], [0.9,0.1,0.1,0.1]],
        [[0.9,0.1,0.1,0.1], [0.9,0.1,0.1,0.1]] ]

    # Create a network with four input, two hidden, and four output nodes
    n = NeuralNetwork(4, 2, 4)
    # Train it with some patterns
    n.train(pat)
    # Test it
    n.test(pat)
```

Step 1: Generate a matrix of I*J

```
# Make a matrix
def makeMatrix(I, J, fill=0.0):
    m = []
    for i in range(I):
        m.append([fill]*J)
    return m
```

Step 2: Define the activation function

```
# Sigmoid Function
def sigmoid(net):
    return 1.0/(1.0 + math.exp(-net))

# Derivative of Sigmoid Function
def dsigmoid(y):
    return y*(1.0 - y)
```

Step 3: Create a neural network

```python
class NeuralNetwork:
    def __init__(self, ni, nh, no):
        # ni,nh,no are the number of input, hidden, and output nodes separately
        self.ni = ni + 1    # +1 for bias node
        self.nh = nh + 1    # +1 for bias node
        self.no = no

        # Activations for nodes
        self.ai = [1.0]*self.ni
        self.ah = [1.0]*self.nh
        self.ao = [1.0]*self.no

        # Create weights
        self.wi = makeMatrix(self.ni, self.nh)
        self.wo = makeMatrix(self.nh, self.no)
        # Set them to random vaules
        for i in range(self.ni):
            for j in range(self.nh):
                self.wi[i][j] = random.gauss(0,0.2)
        for j in range(self.nh):
            for k in range(self.no):
                self.wo[j][k] = random.gauss(0,0.2)
        # Last change in weights for momentum
        self.ci = makeMatrix(self.ni, self.nh)
        self.co = makeMatrix(self.nh, self.no)
```

Step 4: Training with input data can get different weights

```python
    def update(self, inputs):
        if len(inputs) != self.ni-1:
            raise ValueError('Wrong number of inputs')

        # Input activations
        for i in range(self.ni-1):
            self.ai[i] = inputs[i]

        # Hidden activations
        for j in range(self.nh-1):
            sum = 0.0
            for i in range(self.ni):
                sum = sum + self.ai[i] * self.wi[i][j]
            self.ah[j] = sigmoid(sum)

        # Output activations
        for k in range(self.no):
            sum = 0.0
            for j in range(self.nh):
                sum = sum + self.ah[j] * self.wo[j][k]
            self.ao[k] = sigmoid(sum)

        return self.ao[:]
```

Error result:

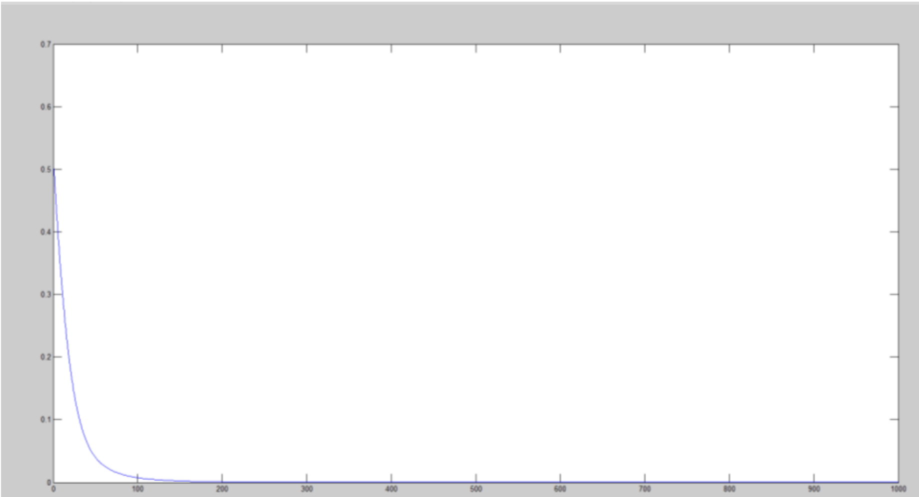

Figure.2 Error of output 1 as a function of n

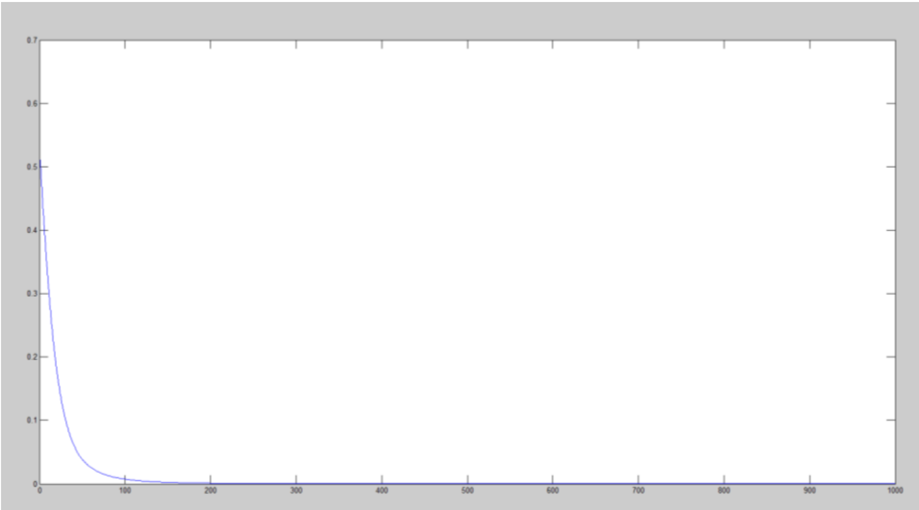

Figure.3 Error of output 2 as a function of n

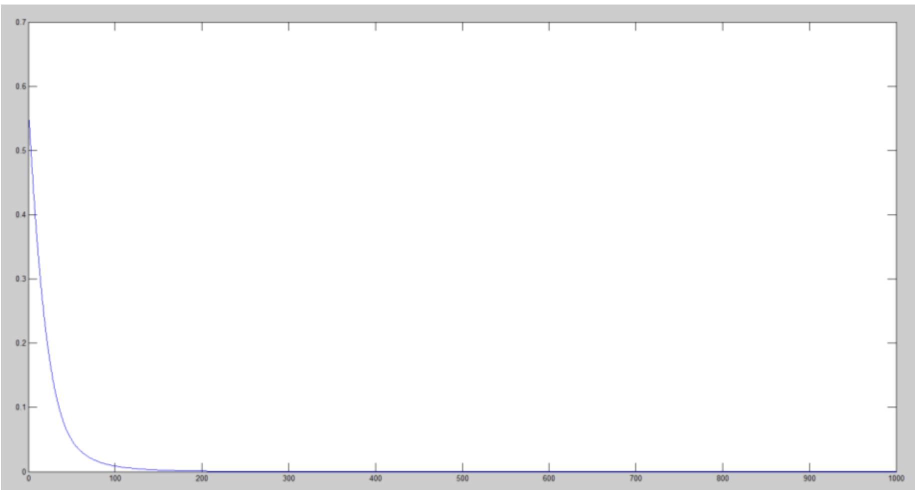

Figure.4 Error of output 3 as a function of n

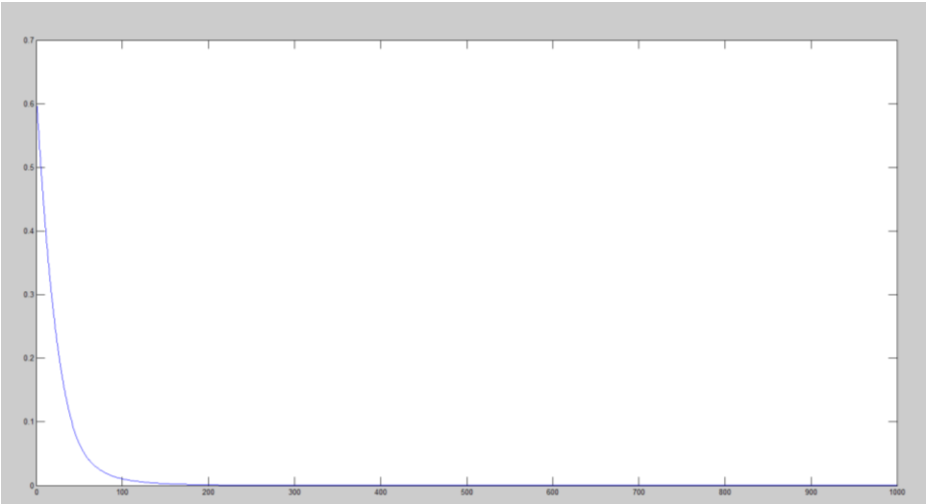

Figure.5 Error of output 4 as a function of n

Since the error plot shows that the error will be close to zero after 1,000 iterations, it validates the error backpropagation theory and proves that my code is working properly.

Step 5: Add bias units to the input and hidden layers to get the error close to zero.

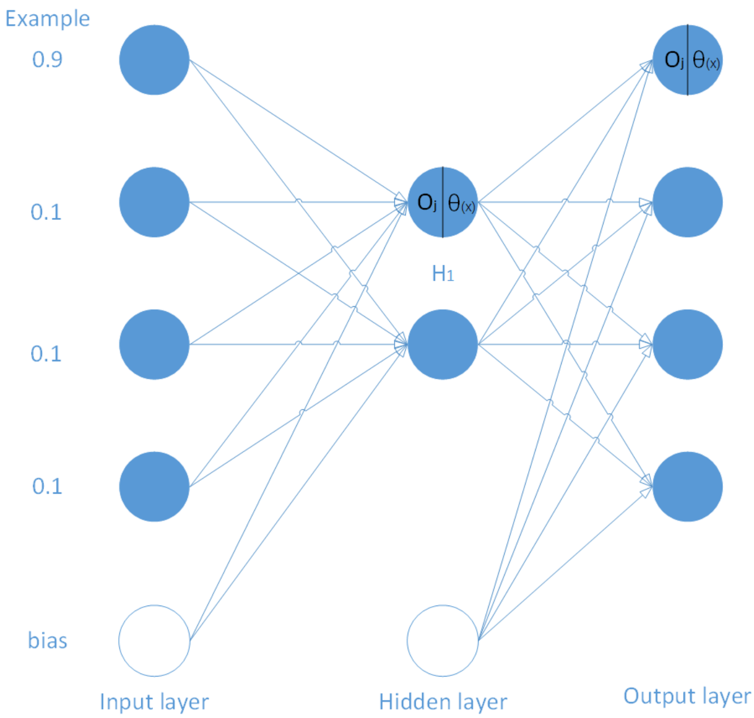

Figure.6 MLP structure for 4-2-4 encoder with bias units

```
class NeuralNetwork:
    def __init__(self, ni, nh, no):
        # ni,nh,no are the number of input, hidden, and output nodes separatel
        self.ni = ni + 1    # +1 for bias node
        self.nh = nh + 1    # +1 for bias node
        self.no = no
```

Before adding bias units:

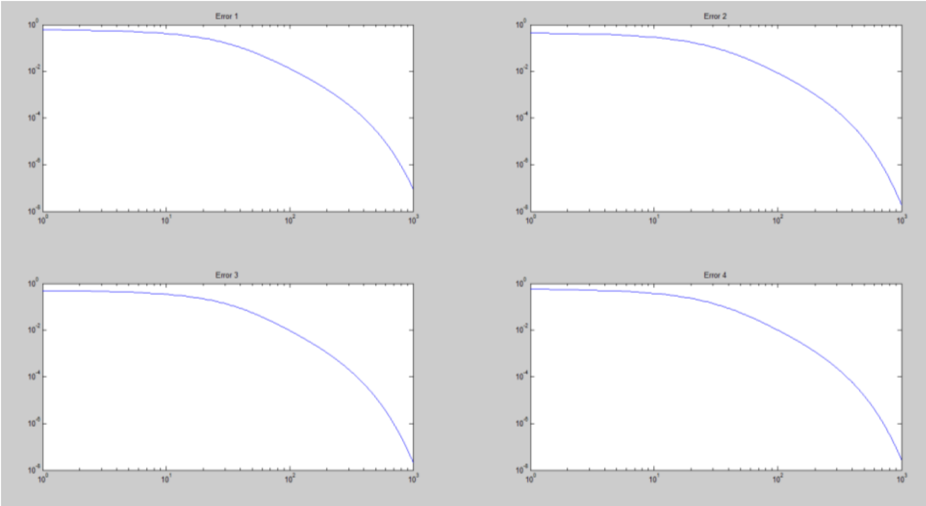

Figure.7 Error plot before adding bias units

After adding bias units:

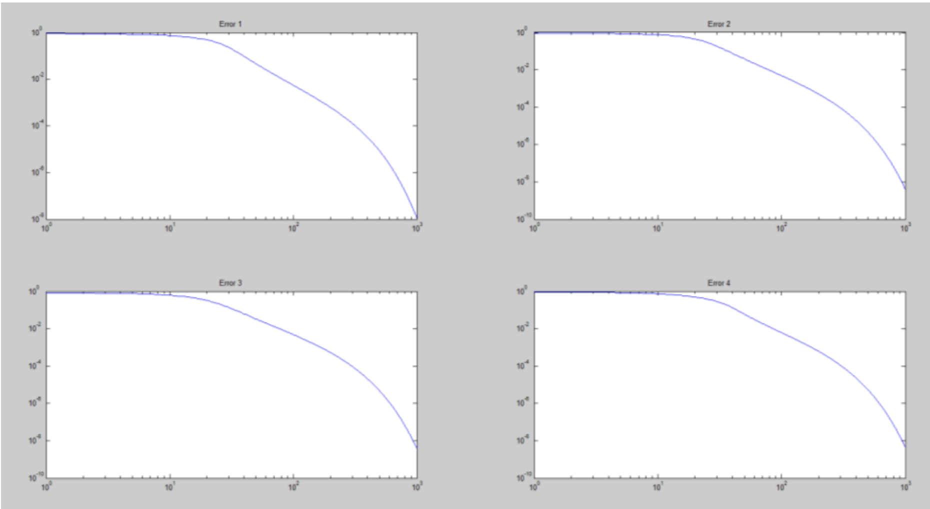

Figure.8 Error plot after adding bias units

As you can see from the plot above, the error becomes smaller as we add the bias units to the input and hidden layers.

The BP neural network has the following advantages: First, it can realize arbitrarily complex nonlinear functions, with strong learning ability and certain promotion and generalization capabilities. At the same time, it is simple to implement and has been used in many large systems. While having the above advantages, BP neural network still has its shortcomings: the convergence speed is slow. It is easy to fall into the local minimum and is prone to over-fitting. The choice of network structure can only be determined by experience.

Appendix:

The task code is following:

```
import math
import random

random.seed(0)

# Make a matrix
def makeMatrix(I, J, fill=0.0):
    m = []
    for i in range(I):
        m.append([fill]*J)
    return m

# Sigmoid Function
def sigmoid(net):
    return 1.0/(1.0 + math.exp(-net))

# Derivative of Sigmoid Function
def dsigmoid(y):
    return y*(1.0 - y)

class NN:
    def __init__(self, ni, nh, no):
        # ni,nh,no are the number of input, hidden, and output nodes separately
        self.ni = ni + 1      # +1 for bias node
        self.nh = nh + 1      # +1 for bias node
        self.no = no

        # Activations for nodes
        self.ai = [1.0]*self.ni
        self.ah = [1.0]*self.nh
        self.ao = [1.0]*self.no

        # Create weights
        self.wi = makeMatrix(self.ni, self.nh)
        self.wo = makeMatrix(self.nh, self.no)
        # Set them to random vaules
        for i in range(self.ni):
            for j in range(self.nh):
                self.wi[i][j] = random.gauss(0,0.2)
        for j in range(self.nh):
            for k in range(self.no):
                self.wo[j][k] = random.gauss(0,0.2)
```

```
        # Last change in weights for momentum
        self.ci = makeMatrix(self.ni, self.nh)
        self.co = makeMatrix(self.nh, self.no)

    def update(self, inputs):
        if len(inputs) != self.ni-1:
            raise ValueError('Wrong number of inputs')

        # Input activations
        for i in range(self.ni-1):
            self.ai[i] = inputs[i]

        # Hidden activations
        for j in range(self.nh-1):
            sum = 0.0
            for i in range(self.ni):
                sum = sum + self.ai[i] * self.wi[i][j]
            self.ah[j] = sigmoid(sum)

        # Output activations
        for k in range(self.no):
            sum = 0.0
            for j in range(self.nh):
                sum = sum + self.ah[j] * self.wo[j][k]
            self.ao[k] = sigmoid(sum)

        return self.ao[:]


    def EBP(self, targets, N, M):
        if len(targets) != self.no:
            raise ValueError('Wrong number of target values')

        # Calculate error terms for output
        output_deltas = [0.0] * self.no
        for k in range(self.no):
            error = targets[k]-self.ao[k]
            output_deltas[k] = dsigmoid(self.ao[k]) * error

        # Calculate error terms for hidden
        hidden_deltas = [0.0] * self.nh
        for j in range(self.nh):
            error = 0.0
            for k in range(self.no):
                error = error + output_deltas[k]*self.wo[j][k]
            hidden_deltas[j] = dsigmoid(self.ah[j]) * error
```

```
        # Update output weights
        for j in range(self.nh):
            for k in range(self.no):
                change = output_deltas[k]*self.ah[j]
                self.wo[j][k] = self.wo[j][k] + N*change + M*self.co[j][k]
                self.co[j][k] = N*change + M*self.co[j][k]

        # Update input weights
        for i in range(self.ni):
            for j in range(self.nh):
                change = hidden_deltas[j]*self.ai[i]
                self.wi[i][j] = self.wi[i][j] + N*change + M*self.ci[i][j]
                self.ci[i][j] = N*change + M*self.ci[i][j]

        # Calculate error
        error = 0.0
        for k in range(len(targets)):
            error = error + 0.5*(targets[k]-self.ao[k])**2
        return error


    def test(self, patterns):
        for p in patterns:
            print(p[0], '->', self.update(p[0]))

    def weights(self):
        print('Input weights:')
        for i in range(self.ni):
            print(self.wi[i])
        print()
        print('Output weights:')
        for j in range(self.nh):
            print(self.wo[j])
    def train(self, patterns, N=0.1, M=0.9):
  # N is learning rate, and M is momentum factor
        for i in range(1000):    # iterations=1000
            error = 0.0
            for p in patterns:
                inputs = p[0]
                targets = p[1]
                self.update(inputs)
                error = error + self.EBP(targets, N, M)

def demo():
    # Teach network XOR function
```

```
    pat = [ [[0.9,0.1,0.1,0.1], [0.9,0.1,0.1,0.1]],
        [[0.9,0.1,0.1,0.1], [0.9,0.1,0.1,0.1]],
        [[0.9,0.1,0.1,0.1], [0.9,0.1,0.1,0.1]],
        [[0.9,0.1,0.1,0.1], [0.9,0.1,0.1,0.1]] ]

    # Create a network with four input, two hidden, and four output nodes
    n = NN(4, 2, 4)
    # Train it with some patterns
    n.train(pat)
    # Test it
    n.test(pat)


if __name__ == '__main__':
    demo()
```