MATLAB Deep Learning Notes VII

Key Words : Cost Function, Cross Entropy Function

1. Cost Function

1. So far, we all know that supervised learning of neural network is a process of adjusting the weights to reduce the error of the training data. In this context, we try to use the *cost function* to measure of the neural network's error. There are two types of cost functions for supervised learning. It is noteworthy that cost function can also be called *loss function* or *objective function.*

$$J = \sum_{i=1}^{M} \frac{1}{2}(d_i - y_i)^2 \tag{22}$$

$$J = \sum_{i=1}^{M} \{-d_i \ln(y_i) - (1 - d_i)\ln(1 - y_i)\} \tag{23}$$

where y_i is the output from the output node, d_i is the correct output from the training data, and M is the number of output nodes.

2. Consider the equation (22), This cost is the square of the difference between the neural network's output, y, and the correct output, d. A greater difference between the two values leads to a larger error. We use this cost function in our early notes.

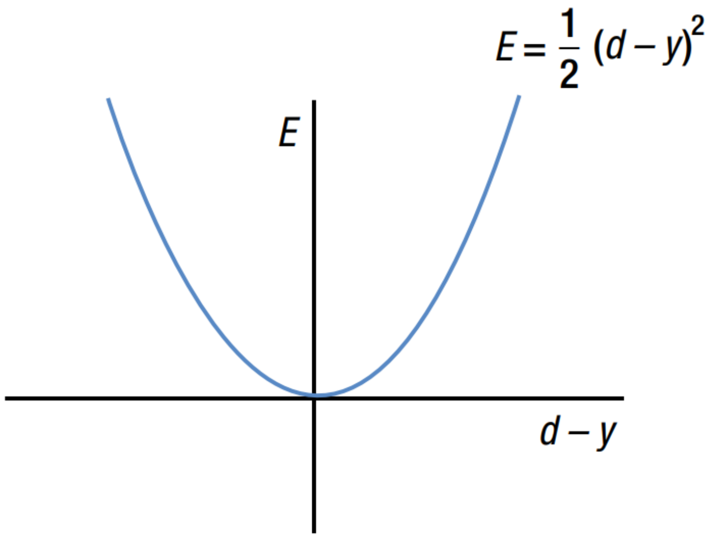

Figure 1: The greater the difference between the output and the correct output, the larger the error.

3. But it is not enough. So we introduce the cross entropy function (23), and the function can be rewritten as

$$E = \begin{cases} -\ln(y) & d = 1 \\ -\ln(1-y) & d = 0 \end{cases} \tag{24}$$

and we can easily verify that 24 is proportional to the error, which is a very important feature of cost function. See Fig.2

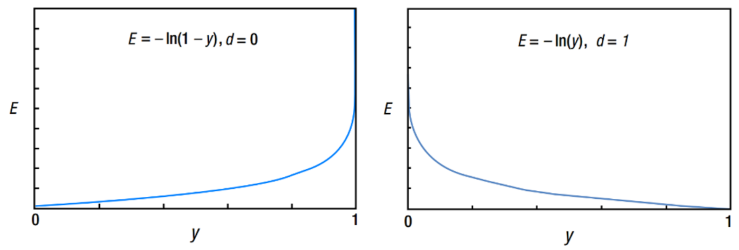

Figure 2: The cross entropy function is proportional to the error.

4. The primary difference of the cross entropy function from the quadratic function (22) is its geometric increase, that is to say, the cross entropy function is much more sensitive to the error, so the learning rules derived from the cross entropy function has a better performance.
5. It is important to know that if we use cross entropy function to replace quadratic function, the calculation of the delta at the output node need to be changed. Specifically,

$$\delta = \varphi'(v)e \to \delta = e$$

6. The key is the fact that the output and hidden layers employ the different formulas of the delta calculation when the learning rule is based on the cross entropy and the sigmoid function. See Fig.3

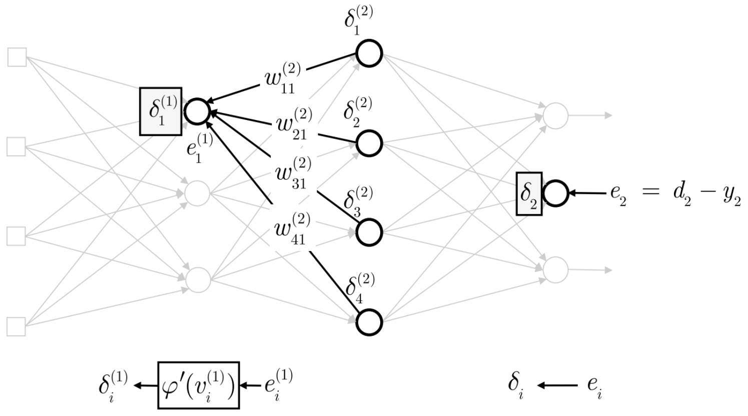

Figure 3: The output and hidden layers employ the different formulas of the delta calculation.

7. We can also use regularization to reduce overfitting(i.e., adding the sum of weights to the cost function).

$$J = \sum_{i=1}^{M} \frac{1}{2}(d_i - y_i)^2 + \lambda \frac{1}{2}\|w\| \tag{25}$$

$$J = \sum_{i=1}^{M} \{-d_i \ln(y_i) - (1 - d_i)\ln(1 - y_i)\} + \lambda \frac{1}{2}\|w\| \tag{26}$$

where λ is the coefficient that determines how much of the connection weight is reflected on the cost function. Keep in mind that applying a new cost function leads to a different learning rule formula.

Program 15: BackPropCE

Listing 1: BackPropCE.m

```
function [weight1, weight2] = BackPropCE(weight1, weight2, data_input,...
correct_output)

alpha = 0.9;
N = 4;

for k = 1 : N
    x = data_input(k, :)';
    d = correct_output(k);

    v1 = weight1 * x;
    y1 = sigmoid(v1);
    v = weight2 * y1;
    y = sigmoid(v);

    e = d - y; % calculate the error of the output to the correct output
    delta = e; % different delta calculation !

    e1 = weight2' * delta; % calculate the error of the hidden layer
    delta1 = y1 .* (1 - y1) .* e1;

    % Adjust the weights
    dw1 = alpha * delta1 * x';
    weight1 = weight1 + dw1;

    dw2 = alpha * delta * y1';
    weight2 = weight2 + dw2;
end
end
```

Program 16: testBackPropCE

Listing 2: testBackPropCE.m

```
clear
data_input = [ 0, 0, 1; 0, 1, 1; 1, 0, 1; 1, 1, 1]; % training data
correct_output = [0; 1; 1; 0]; % correct outputs(i.e., labels)
weight1 = 2 * rand(4, 3) - 1;
weight2 = 2 * rand(1, 4) - 1;


for epoch = 1:10000 % train
[weight1, weight2] = BackPropCE(weight1, weight2, data_input,...
correct_output);
end

N = 4; % inference
for k = 1 : N
x = data_input(k, :)';
v1 = weight1 * x;
y1 = sigmoid(v1);
v = weight2 * y1;
y = sigmoid(v)
end
```

Output 16:

```
>> testBackPropCE

y =

   3.4883e-05

y =

    0.9999

y =

    0.9998

y =

   3.3885e-04
```

Program 17: CEVsXOR

Listing 3: CEVsXOR.m

```
clear

data_input = [ 0, 0, 1; 0, 1, 1; 1, 0, 1; 1, 1, 1 ]; % training data
correct_output = [ 0; 0; 1; 1 ]; % correct outputs
weight11 = 2 * rand(4, 3) - 1;
weight12 = 2 * rand(1, 4) - 1;
weight21 = weight11;
weight22 = weight12;

E1 = zeros(1000, 1);
E2 = E1;

for epoch = 1 : 1000 % the number of epoch = 1000
    [weight11, weight12] =  BackPropCE(weight11, weight12, data_input,
       correct_output);
    [weight21, weight22] =  BackPropXOR(weight21, weight22, data_input,
       correct_output);

    es1 = 0; es2 = 0;
    N = 4;
    for k = 1 : N
        x = data_input(k, :)';
        d = correct_output(k);
        % CE
        v1_CE = weight11 * x;
        y1_CE = sigmoid(v1_CE);
        v_CE = weight12 * y1_CE;
        y_CE = sigmoid(v_CE);
        es1 = es1 + (d - y_CE)^2;
        % XOR
        v1_XOR= weight21 * x;
        y1_XOR = sigmoid(v1_XOR);
        v_XOR = weight22 * y1_XOR;
        y_XOR = sigmoid(v_XOR);
        es2 = es2 + (d - y_XOR)^2;
    end
    E1(epoch) = es1 / N;
    E2(epoch) = es2 / N;

end

plot(E1, '--b', 'Linewidth',1);
```

```
hold on
plot(E2, '—.r', 'Linewidth',1);
xlabel('the number of Epoch')
ylabel('Average of Training error')
legend('Cross Entropy', 'Sum of Squared Error')
```

Output 17:

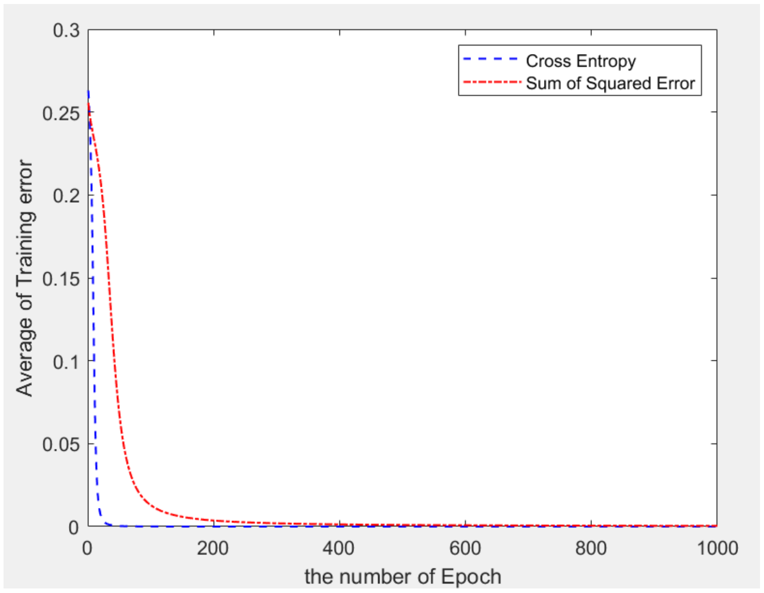

Figure 4: Cross Entropy has a better performance!