Homework 1.5 XOR with Backpropagation

Paden Rumsey

October 24, 2018

1 Introduction

In the last homework we saw that perceptrons are interesting tools that can solve relatively simplistic linear problems. When combined together they can provide enough non-linearity to solve non-linear problems. However, due to problems like translation in groups, simple perceptron learning is inefficient, and unable to solve some issues. This is why we introduce the multi-layer perceptron with backpropagation. Backpropagation involves using partial and chain derivatives to calculate loss in a system with respect to the weights so we can minimize the error between our target output and the output from our network. Using multiple perceptrons with backpropagations allows us to solve complex non-linear problems. XOR isn't necessarily complicated, but it is non-linear.

2 Some Initial Math

Learning Rate

$\eta = 0.5$

Input

$$\begin{bmatrix} I_1 & I_2 & B \\ 0 & 1 & 1 \end{bmatrix}$$

Original Weights (Hidden & Output Layers)

$$\begin{bmatrix} -0.72132677 & 0.10280051 \\ -0.11470404 & 0.84444676 \\ -0.18124214 & 0.57317879 \end{bmatrix} \quad \begin{bmatrix} 0.0571348 \\ -0.69255019 \\ 0.45870247 \end{bmatrix}$$

Feedforward with Activation (First Layer)

$$0 * -0.72132677 + 1 * -0.11470404 + 1 * -0.18124214 = -0.29594618$$

$$0 * 0.10280051 + 1 * 0.84444676 + 1 * 0.57317879 = 1.41762554$$

$$\frac{1}{1 + e^{-(-0.29594618)}} = 0.42654877$$

$$\frac{1}{1 + e^{-(1.41762554)}} = 0.80496591$$

Hidden Layer

$$\begin{bmatrix} 0.42654877 \\ 0.80496591 \\ 1 \end{bmatrix}$$

Feed-Forward Hidden Layer

$$0.42654877 * 0.0571348 + 0.80496591 * (-0.69255019) + 1 * 0.45870247 = (-0.07440603)$$

$$\frac{1}{1 + e^{-(-0.07440603)}} = 0.48140707$$

Backpropagation

$$d2 = 0.48140707 * (1 - 0.48140707) * (0 - 0.48140707) = 0.12946896$$

$$d1 = \begin{bmatrix} 0.42654877 \\ 0.80496591 \\ 1 \end{bmatrix} * (1 - \begin{bmatrix} 0.42654877 \\ 0.80496591 \\ 1 \end{bmatrix}) * (d2 * \begin{bmatrix} 0.0571348 \\ -0.69255019 \\ 0.45870247 \end{bmatrix} = \begin{bmatrix} -0.00290237 \\ 0.02008515 \\ 0 \end{bmatrix})$$

$$dT1 = \begin{bmatrix}0 & 1 & 1\end{bmatrix} * \begin{bmatrix}-0.00290237 \\ 0.02008515 \\ 0\end{bmatrix} = \begin{bmatrix}-0.00290237 \\ 0.02008515\end{bmatrix} = \begin{bmatrix}0 & 0 \\ 0.00180939 & -0.01407683 \\ 0.00180939 & -0.01407683\end{bmatrix}$$

$$dT2 = \begin{bmatrix}0.42654877 \\ 0.804965911 \\ 1\end{bmatrix} * 0.12946896 = \begin{bmatrix}0.05522482 \\ 0.1042181 \\ 0.12946896\end{bmatrix}$$

$$weights2 = \begin{bmatrix}0.0571348 \\ -0.69255019 \\ 0.45870247\end{bmatrix} + (\eta * \begin{bmatrix}0.05522482 \\ 0.1042181 \\ 0.12946896\end{bmatrix})$$

$$weights1 = \begin{bmatrix}-0.72132677 & 0.10280051 \\ -0.11470404 & 0.84444676 \\ -0.18124214 & 0.57317879\end{bmatrix} + (\eta * \begin{bmatrix}0 & 0 \\ 0.00180939 & -0.01407683 \\ 0.00180939 & -0.01407683\end{bmatrix}$$

$$weights2 = \begin{bmatrix}0.08474721 \\ -0.6404411 \\ 0.523436957\end{bmatrix}$$

$$weights1 = \begin{bmatrix}-0.72132677 & 0.10280051 \\ -0.11379935 & 0.83740834 \\ -0.18033744 & 0.56614037\end{bmatrix}$$

3 Running the XOR with Different Learning Rates

- $\eta = .5$ - I initially ran the network on this learning rate for 1000 epochs. Needless to say it didn't really converge to *any* particular values. Instead, the result for all four samples hovered around 0.5. I then ran it for 5000 epochs. This time, the network started to converge on the right values but didn't reach peak convergence. The samples that should have been 1 were at 0.8 and the samples that were supposed to be 0 were 0.1 and 0.2. 8,000 epochs were required to start seeing samples that converged to values of .9 and under .1 for the values of 1 and 0, respectively.

- $\eta = .75$ - I decided to try 1000 epochs again to see if maybe it would converge in the right direction, even only slightly. But it seemed to perform just as well as the .5 learning rate did. I then tried 5000 epochs which gave clear convergence with values above .9 and below .1. Since it converged at 5000 I decided to try 2500 epochs. This value was much more unreliable and didn't give a semblance of convergence.

- $\eta = .9$ - It seems 1000 epochs will never be enough as I tried it with this, and it didn't converge at all either. The values hovered around .5. Using 2500 epochs was, again, unreliable and didn't give me the results I sought. Running it with 4000 epochs was enough to give me values above .9 and below .1 again.

- $\eta = 0.01$ - Knowing this rate was extremely small. I figured I would start out large and try 5000 epochs to begin with. This resulted in no convergence for this extremely small learning rate. So, I went bigger by a factor of 10: 50,000 epochs. This still did not converge. Now, I would try it with 100,000 but I believe it won't converge there either. This is an extremely small learning rate and I'm not sure when, or even if, it would converge.

4 Appendix A. Code

```
    import numpy as np

def activate(h):
    return 1/(1 + np.exp(-h))

#Inputs with their respective biases
x = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
y = np.array([0,1,1,0])
#Randomize the weight matrix
w = 2 * np.random.rand(3,2) - 1
w2 = 2* np.random.rand(3,1) - 1

#learning rate or (eta)
learn = .01

j = 0

while (j < 50000):

    #0 and 0
    print("")
    for i in range(x.shape[0]):
        #Select a singular input from the list
        input = x[i]
        input = np.reshape(input, (1,3))

        #Feedforward
        h = np.dot(input, w)
        hidden = activate(h)
        hidden = np.append(hidden, [1])
        hidden = np.reshape(hidden, (1,3))
        output = np.dot(hidden,w2)
        result = activate(output)
        result = result[0]
        r = result
```

```
#Compute the backpropagation using the derivative of
#the sigmoid and the error from the actual value

d2 = ((1 - result) * result) * (y[i] - result)
d1 = (hidden * (1 - hidden)) * d2.dot(w2.T)
d2 = np.reshape(d2, (1,1))

update2 = (hidden.T.dot(d2))
update1 = (input.T.dot(d1[:,:-1]))

#Update the weights

w2 += learn * update2
w += learn * update1

print(str(input[0][0]) + " " + str(input[0][1]))
print("result " + str(r[0]))

j = j + 1
```