Lab1

鄭余玄

1 Introduction

In this lab, we are going to implement a fully connected (dense) artificial neural network with two hidden layers. However, if we directly hard code these gradients formula of a two-hidden-layer neural network may lack of generality. A fully connected neural network could be treated more generally if breaking down to fundamental building blocks.

In this report, I present an extensible length fully-connected neural network with variable depth to generalize a two-hidden-layer neural network. I will describe the training process in more detail in Section 2.

2 Method

This lab is mainly focus on the backpropogation process: calculating the gradient for each weights. I attempt to formulate the gradient of each layers in a system of recursive relations.

A fully connected neural network has a basic building block shown in figure 1, and in addition with a loss function structure. The input layer and output layer plays a role for stopping criterion for the recursive relation.

2.1 Sigmoid function

The sigmoid function of $y \in \mathbb{R}$ is defined as

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

, and its derivative is

$$\sigma'(y) = \sigma(y)(1 - \sigma(y))$$

To apply the sigmoid function on a row vector $\mathbf{y} \in \mathbb{R}^n$, denoted as $\sigma(\mathbf{y})$, we assign the sigmoid function on each component, namely,

$$\sigma(\mathbf{y}) = (\sigma(y_1), \ldots, \sigma(y_n)).$$

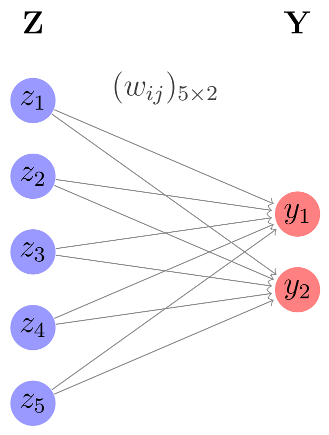


Figure 1: two (hidden) layers

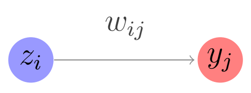


Figure 2: two (hidden) neurons

Similarly, its derivative is

$$\begin{aligned}\sigma'(\mathbf{y}) &= (\sigma(y_1)(1-\sigma(y_1)),\ldots,\sigma(y_n)(1-\sigma(y_n)))\\ &= (\sigma(y_1),\ldots,\sigma(y_n)) \odot (\mathbf{1} - (\sigma(y_1),\ldots,\sigma(y_n)))\\ &= \sigma(\mathbf{y}) \odot (\mathbf{1} - \sigma(\mathbf{y}))\end{aligned}$$

where $\odot$ is an element-wise product and $\mathbf{1}$ is a vector of ones.

Notably, if $\mathbf{y} = \sigma(\widetilde{\mathbf{y}})$, then

$$\begin{aligned}\sigma'(\widetilde{\mathbf{y}}) &= \sigma(\widetilde{\mathbf{y}}) \odot (\mathbf{1} - \sigma(\widetilde{\mathbf{y}}))\\ &= \mathbf{y} \odot (\mathbf{1} - \mathbf{y}).\end{aligned} \tag{1}$$

Therefore, we can intuitively define $s(\mathbf{y}) = \mathbf{y} \odot (\mathbf{1} - \mathbf{y})$ as an alternative for calculating $\sigma'(\widetilde{\mathbf{y}})$, where s could be view as a function of implicit $\widetilde{\mathbf{y}}$. This equation with be used in Section 2.3.

2.2 Neural network

A neural network needs to implement the following functions:

1. weights initialization
2. forward pass (prediction)
3. backward pass (backpropagation + weights update)

First, the weights are initialized to small random values, more specifically, a uniform random value between $\pm 4\sqrt{\frac{6}{\mathrm{dim}_{\mathrm{in}}+\mathrm{dim}_{\mathrm{out}}}}$. These weights are chosen due to the sigmoid loss function according to the result of Glorot and Bengio [1].

Then, in the forward pass, values of neurons in the forward layer are produced by applying activation on the product of the result of previous layers and the weights between. Take Figure 1 as an example, $\mathbf{Y} = \sigma(\mathbf{ZW})$ where σ is the sigmoid activation function used in this lab. In other words, from the vintage of each neuron shown in Figure 2, $y_j = \sigma(\sum_i w_{ij} z_i) = \sigma(\widetilde{y_j})$ where $\widetilde{y_j} = \sum_i w_{ij} z_i$ denotes the linear combination of the previous layer before activation.

Finally, in the backward pass, all weights are update with regard to the gradient of the loss function. The details of backpropagation will be describe in Section 2.3. For the forward pass and backward pass, these process will iterate until stopping criterion is reached.

2.3 Backpropagation

We first calculate gradients of the building block shown in Figure 1. To be consistent, each layer is represented as a row vector, namely, $\mathbf{Z} = (z_1, \ldots, z_n)$ and $\mathbf{Y} = (y_1, \ldots, y_m)$, and the weights is $\mathbf{W} = (w_{ij})_{n \times m}$. In Section 2.2, we know that $\mathbf{Y} = \sigma(\mathbf{ZW})$ in the forward pass.

Then, we derive the loss function L with regard to the weights w_{ij} and neurons z_i, shown in Equation 2 and 3.

$$\begin{aligned}
\frac{\partial L}{\partial w_{ij}} &= \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial w_{ij}} && \text{(2a)} \\
&= \frac{\partial L}{\partial y_j} \frac{\partial y_j}{\partial \widetilde{y_j}} \frac{\partial \widetilde{y_j}}{\partial w_{ij}} = \frac{\partial L}{\partial y_j} \sigma'(\widetilde{y_j}) z_j && \text{(2b)} \\
&= \frac{\partial L}{\partial y_j} s(y_j) z_i && \text{(2c)}
\end{aligned}$$

From Equation 2a to 2b, we employ chain rule on the loss function L from an activated neuron y_j to a pre-activated neuron $\widetilde{y_j}$. Equation 2b enables us to substitute $\sigma'(\widetilde{y_j})$ as $s(y_j)$ mentioned in Section 2.1 Equation 1 such that the derivative is only related to activated neurons. Likewise, this technique also applies on Equation 3.

$$\frac{\partial L}{\partial z_i} = \sum_j \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial z_i} \tag{3a}$$

$$= \sum_j \frac{\partial L}{\partial y_j}\frac{\partial y_j}{\partial \widetilde{y_j}}\frac{\partial \widetilde{y_j}}{\partial z_i} = \sum_j \frac{\partial L}{\partial y_j}\sigma'(\widetilde{y_j})w_{ij} \tag{3b}$$

$$= \sum_j \frac{\partial L}{\partial y_j}s(y_j)w_{ij} \tag{3c}$$

In addition, we can organize $\frac{\partial L}{\partial w_{ij}}$ (Equation 2) into the matrix form $\frac{\partial L}{\partial \mathbf{W}}$ (Equation 4a), and $\frac{\partial L}{\partial z_i}$ (Equation 3) into $\frac{\partial L}{\partial \mathbf{Z}}$ (Equation 4b), where $\otimes$ denotes the Kronecker product.

$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{Z}^T \otimes (\frac{\partial L}{\partial \mathbf{Y}} \odot s(\mathbf{Y})) \tag{4a}$$

$$\frac{\partial L}{\partial \mathbf{Z}} = (\frac{\partial L}{\partial \mathbf{Y}} \odot s(\mathbf{Y}))\mathbf{W}^T \tag{4b}$$

Furthermore, in the matrix form (Equation 4), we can observe an occurring term of $\frac{\partial L}{\partial \mathbf{Y}} \odot s(\mathbf{Y})$, that is $\frac{\partial L}{\partial \widetilde{\mathbf{Z}}}$, which can be rewritten in Equation 5.

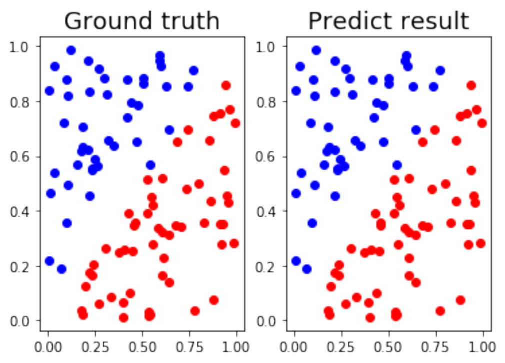
$$\frac{\partial L}{\partial \mathbf{W}} = \mathbf{Z}^T \otimes \frac{\partial L}{\partial \widetilde{\mathbf{Y}}} \tag{5a}$$

$$\frac{\partial L}{\partial \widetilde{\mathbf{Z}}} = \frac{\partial L}{\partial \mathbf{Z}} \odot s(\mathbf{Z}) = (\frac{\partial L}{\partial \widetilde{\mathbf{Y}}}\mathbf{W}^T) \odot s(\mathbf{Z}) \tag{5b}$$

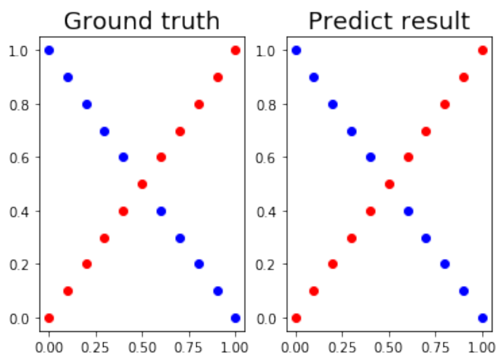
Finally, I use the squared error in the loss function layer. Then, the derivative of squared error is simply a subtraction.

3 Result

In this section, I evaluated two types of inputs, *linear n=100* and *XOR easy*, shown in Figure 3. Both experiments use the same settings: 3 hidden neurons for two hidden layers, learning rate equals to a constant (0.8), and training processes converge until all loss are less than a threshold (0.04).



(a) linear n=100



(b) XOR easy

Figure 3: Results

4 Discussion

During the training, I figure out the importance of the weighs initialization. If all weights are merely initialized to zeros, it could be harder to converge. On the other hand, I also find out the effects of learning rates, shown in Figure 4, where a better learning rate could converge much more faster.

References

[1] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

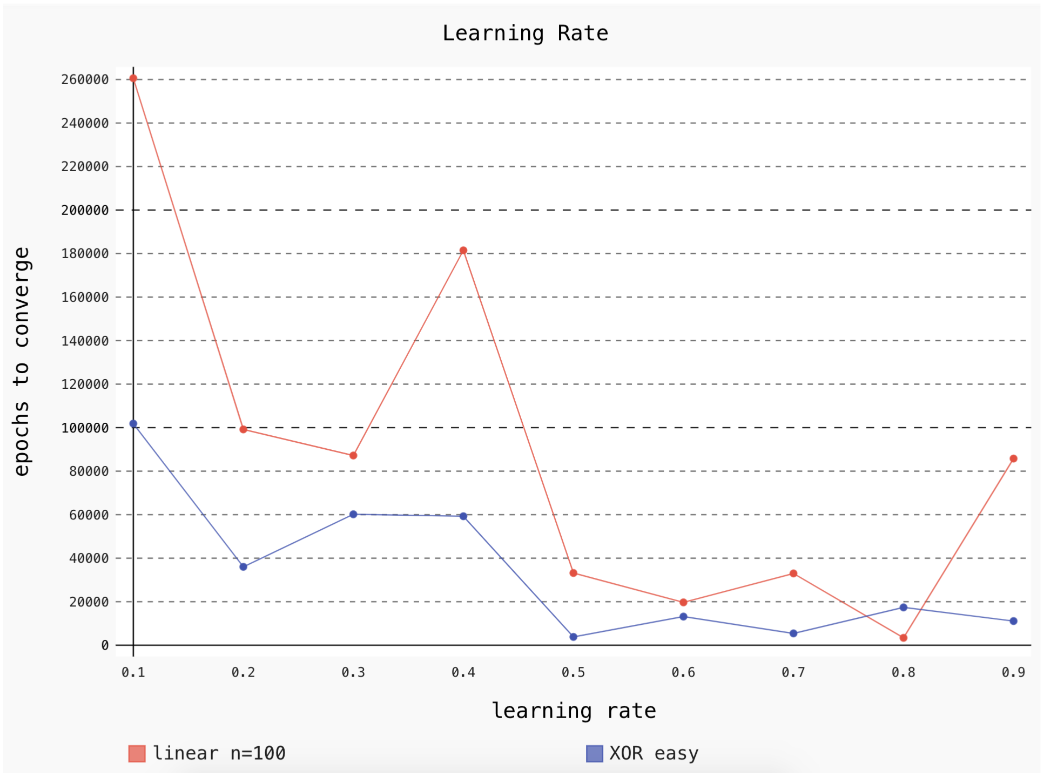


Figure 4: Learning Rate