Lab1 : Back-propagation

contributed by < `tl32rodan` >

Introduction

- Deep learning technique is trending nowadays. One of the reasons being powerful is the ***gradient-based optimization***.

> Most deep learning algorithms involve optimization of some sort. Optimization refers to the task of either minimizing or maximizing some function $f(x)$ by altering
> x. … We can reduce $f(x)$ by moving x in small steps with opposite sign of the derivative. This technique is called **gradient descent**.
> – *Deep Learning* (https://www.deeplearningbook.org/), Goodfellow et-al., MIT Press, 2016.
> *Section 4.3*

- To have a comprehensive knowledge of gradient-based optimization, this lab is for the purpose of implementing backpropagation of a simple two-layered feedforward network without using deep learning frameworks.

Lab Objective

- The purpose of this lab is to understand and implement simple neural networks with forwarding pass and backpropagation using two hidden layers.
- Only Numpy and the python standard libraries are available, any other framework (ex : Tensorflow、PyTorch) is not allowed in this lab.

Requirements

- Implement simple neural networks with two hidden layers.
- Use backpropagation in this neural network and can only use Numpy and other python standard libraries to implement.
- Plot comparison figure that show the predict result and the ground-truth

Experiment Setups

Sigmoid functions

Activation function

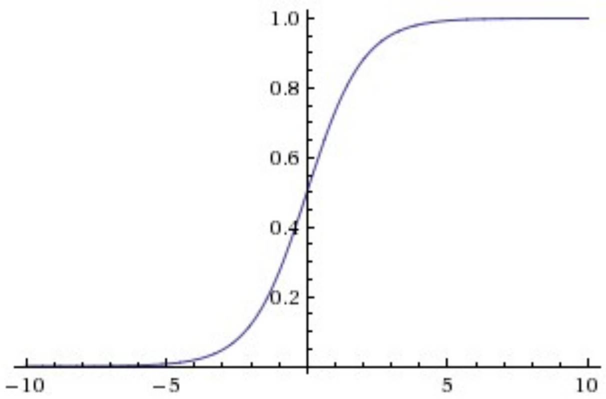
- A layer of a feedforward network is just a **linear model** of input x: $f(x;w,b)=w^T x+b$
- If 2 linear layers form a network, then it is just another linear model, meaning that it’s still not able to solve XOR problem (https://medium.com/@jayeshbahire/the-xor-problem-in-neural-networks-50006411840b)

> Suppose the network now contains two functions chained together: $h=f^{(1)}(x;W,c)$ and $y=f^{(2)}(h;w,b)$, with the complete model being $f(x;W,c,w,b)=f^{(2)}(f^{(1)}(x))$. … Ignoring the intercept terms for the moment, suppose $f^{(1)}(x)=W^T x$ and $f^{(2)}(h)=h^T w$. Then $f(x)=w^T W^T x$. We could represent this function as $f(x)=x^T w'$, where $w'=Ww$ – *Deep Learning* (https://www.deeplearningbook.org/), Goodfellow et-al., MIT Press, 2016. *Section 6.1*

- Activation function 能將 ***nonlinear*** 引入模型中, 使得模型能夠實現 nonlinear 的學習

Sigmoid function

- One of the most common activation functions



- $\sigma(x)=\frac{1}{1+\exp(-x)}$
    - Pros:
        - 能將 layer output $w^T x+b$ 轉為 $[0,1]$ 之間的數值
        - 處處可微; 代表將來做 gradient decent (https://hackmd.io/i9wkp1OZSsamwb7hJuO44w?view#Introduction) 時，不會遇到無法更新的情況
        - Sigmoid function 的微分為 $\frac{d}{dx}\sigma(x)=\sigma(x)(1-\sigma(x))$，所以在計算 backpropagation 時，有不錯的計算效率
            - sigmoid function 微分推導 (https://math.stackexchange.com/questions/78575/derivative-of-sigmoid-function-sigma-x-frac11e-x)
    - Cons:

  - 由於要計算 $\exp(-x)$, 所以運算相對複雜
  - 有 **saturation problem**

    > The sigmoid function **saturates** when its argument is very positive or very negative, meaning that the function becomes very flat and insensitive to small changes in its input – *Deep Learning* (https://www.deeplearningbook.org/), Goodfellow et-al., MIT Press, 2016. *Section 3.10*

  - Not zero-centered

- 實現

```
class Sigmoid(Layer):
    def __init__(self):
        self.y = None

    def forward(self, x):
        # 計算 sigmoid(x)
        self.y = 1.0 /(1.0 + np.exp(-x))
        return self.y

    def backward(self, prev_grad,lr=0.1):
        # 計算 sigmoid(x) 的微分，並且做 backpropagation
        return prev_grad * np.multiply((1.0 - self.y) , se
```

Neural network

Deep feedforward networks

- According to ***Chapeter 10*** of the book ***Deep Learning*** (https://www.deeplearningbook.org/), **Goodfellow et-al.**

> Deep feedforward networks, also often called **feedforward neural networks**, or **multilayer perceptrons (MLPs)**, are the quintessential deep learning models. The goal of a feedforward network is to approximate some function f^*. A feedforward network defines a mapping $y=f(x;\theta)$ and learns the value of the parameters θ that result in the best function approximation.
> They are called ***feedforward*** because information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are **no feedback connections** in which outputs of the model are fed back into itself.
> When feedforward neural networks are extended to include feedback connections, they are called **recurrent neural networks**

Implementing Details

Implementation Details:

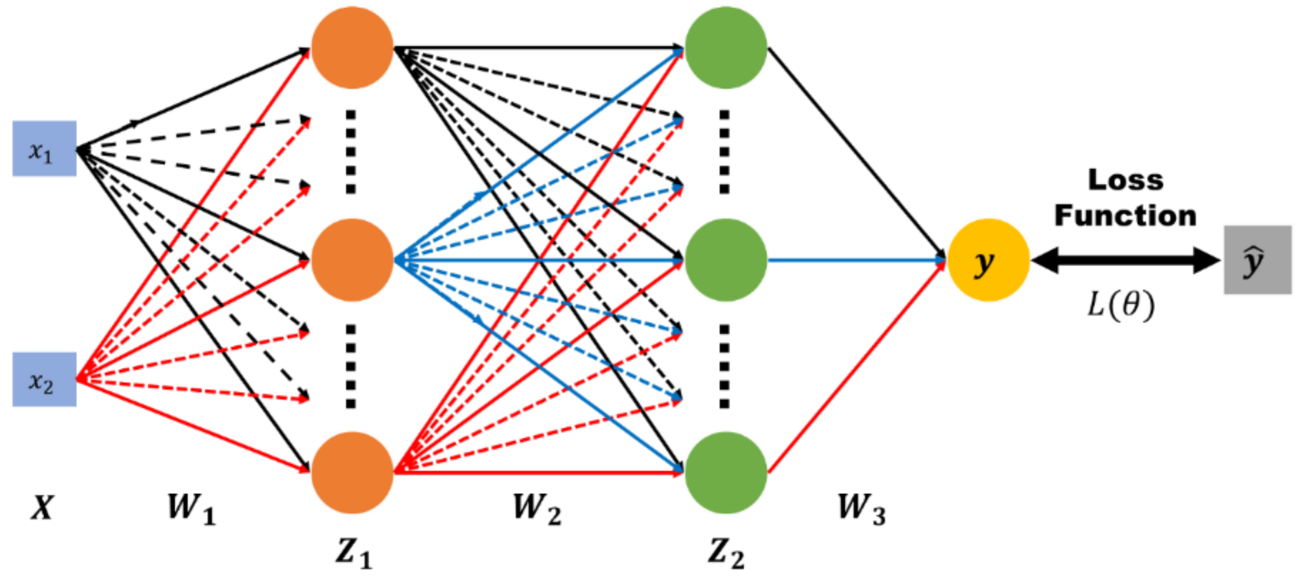


Figure 2. Forward pass

- In the figure 2, we used the following definitions for the notations:
  1. x_1, x_2 : *nerual network inputs*
  2. X : $[x_1, x_2]$
  3. y : *nerual network outputs*
  4. $\hat{y}$: *ground truth*
  5. $L(\theta)$: *loss function*
  6. W_1, W_2, W_3 : *weight matrix of network layers*

- 依據上面的架構圖，實現程式碼如下:

```
class TLNN(object):
    def __init__(self):
        # 用 collection.OrderedDict 依序存放 layer
        # 以便於 forward & backward呼叫
        self.layers = OrderedDict()
        self.layers['linear_1'] = Linear(2,4,bias=True)
        self.layers['sigmoid_1'] = Sigmoid()
        self.layers['linear_2'] = Linear(4,4,bias=True)
        self.layers['sigmoid_2'] = Sigmoid()
        self.layers['output'] = Linear(4,1,bias = False)
        self.layers['sigmoid_3'] = Sigmoid()

        self.loss_func = MSE()

    def forward(self, x):
        # 把計算的值"順向"傳遞 -> "forward"
        for layer in self.layers.values():
            x = layer.forward(x)
        return x

    def cal_loss(self, y, ground_truth):
        # 計算 loss
        return self.loss_func.forward(y,ground_truth)

    def backward(self,lr=0.05):
        dy = self.loss_func.backward()

        # Reverse the layers list for easily conducting ba
        back_layers = list(self.layers.values())
        back_layers.reverse()
        # 把 gradient "逆向"傳遞 -> "forward"
        for layer in back_layers:
            dy = layer.backward(dy,lr=lr)
```

說明

- Input 為二維的資料; output 僅為 `0` 或 `1`
- Hidden layers 皆為 `4` 個 units
- Activation function 皆為 sigmoid function (https://hackmd.io/i9wkp1OZSsamwb7hJuO44w#Sigmoid-function)
- Loss function 採用 MSE(Mean Square Error)

> In statistics, the mean squared error (MSE) or mean squared deviation (MSD) of an estimator (of a procedure for estimating an unobserved quantity) measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value. – *Wilipedia* (https://en.wikipedia.org/wiki/Mean_squared_error)

- ~~非常簡單~~

Backpropagation

- Definition:

> The back-propagation algorithm (Rumelhart et al., 1986a) (http://www.cs.toronto.edu/~hinton/absps/naturebp.pdf), often simply called
> **backprop**, allows the information from the cost to then flow backwards through
> the network, in order to compute the gradient.
> – *Deep Learning* (https://www.deeplearningbook.org/), Goodfellow et-al., MIT Press, 2016. *Section 6.5*

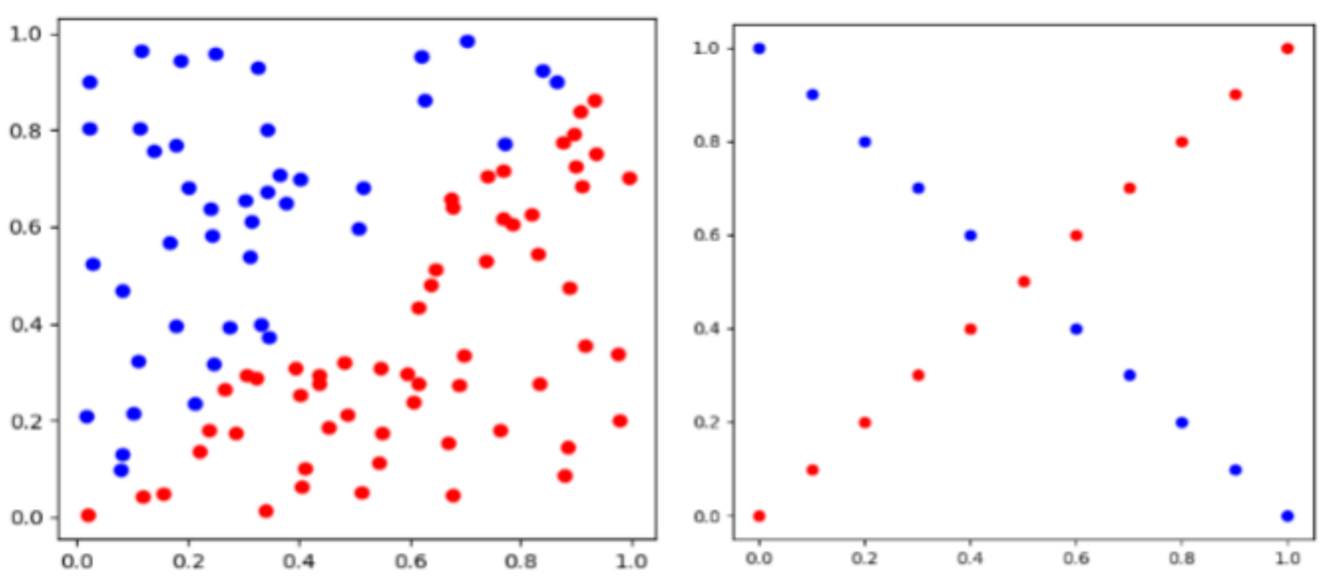
- 對於 backpropagation 的誤解
  1. Backpropagation 不是 neural network 的 learning algorithm
     - back-propagation refers only to the method for computing the gradient , while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient.
  2. Backpropagation 不是 只能用在 multi-layer neural network 上
     - 任何計算 gradient $\nabla_x f(x,y)$ for an arbitrary function f 的過程，皆可稱之。
       - x is a set of variables whose derivatives are desired
       - y is an additional set of variables that are inputs to the function but whose derivatives are not required

- 實作想法
  - 對於每一層 layer of neural network, 都定義一個 memeber function `backward()`
  - 接收後面傳回來的 gradient `prev_grad` 為 input,
  - 計算 weight `w` 的權重
  - 回傳 gradient w.r.t 該 layer 的 input x, i.e. $\frac{\partial L(\theta)}{\partial x}$, 其中 $L(\theta)$ 為 loss function value
  - Example:

```
    def backward(self, prev_grad, lr = 0.1):
        '''
            Input:
                prev_grad: np.array that comes from "next"
                lr : Learning rate
        '''
        # Calculate gradient of w, and update it with (lea
        dw = np.dot(self.x.T,prev_grad)
        self.w -= lr * dw

        # Update bias if needed. Gradient of bias is the e
        if self.has_bias is True:
            db = np.sum(prev_grad,axis=0)
            self.b -= lr * db

        # Return gradient that prpogate to next layer
        return np.dot(prev_grad,self.w.T)
```

- 補充說明:
  - 若 $Y = XW + b$, 則 $\frac{\partial L(\theta)}{\partial X} = \frac{\partial Y}{\partial X}\frac{\partial L(\theta)}{\partial Y} = \frac{\partial L(\theta)}{\partial Y} \cdot W^T$
  - $\frac{\partial L(\theta)}{\partial W} = \frac{\partial Y}{\partial W}\frac{\partial L(\theta)}{\partial Y} = X^T \cdot \frac{\partial L(\theta)}{\partial Y}$
    - $\frac{\partial L(\theta)}{\partial Y}$ 即為後面傳回來的 gradient `prev_grad`

Training Dataset

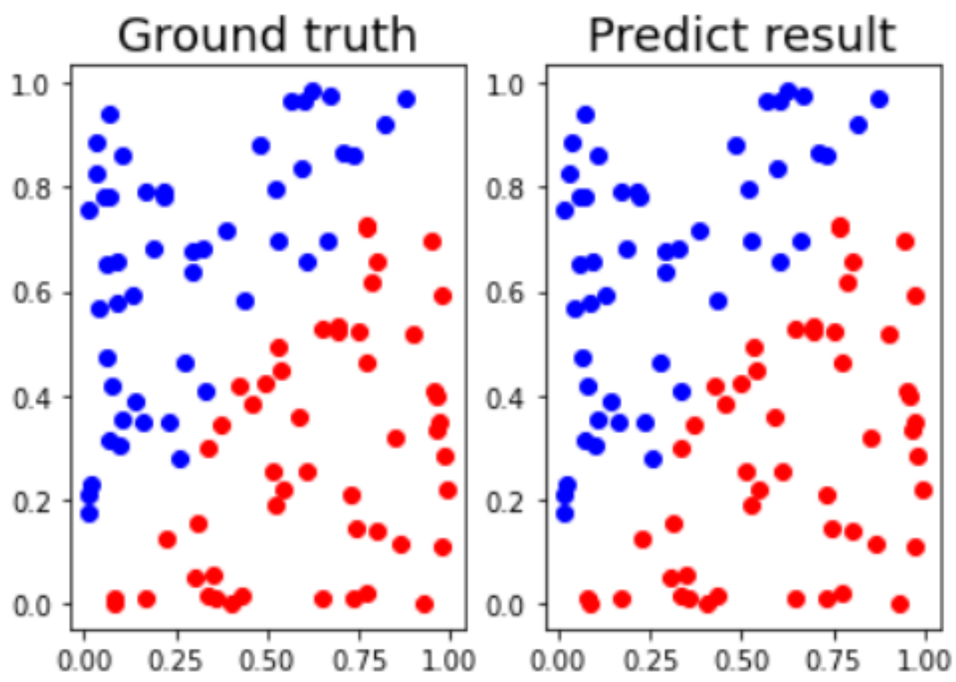


- Left: A set of linearly seperatable data. Easy
- Right: XOR data, which is a little more difficult to train

Linear data

- 預設訓練 `10000` 次

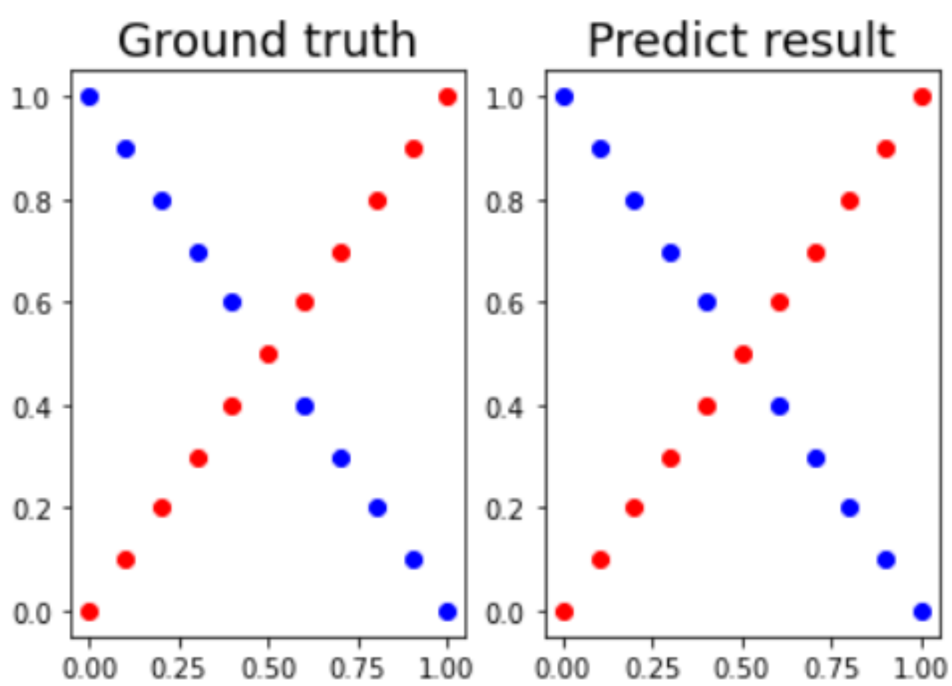
```
data_x, data_y = generate_linear(n=100)
pred_y, loss_list = run_Net(data_x,data_y)

# Comparison graph
show_result(data_x,data_y,pred_y)
```



XOR data

- 由於 XOR 較難訓練且資料不多，索性訓練 `500000` 次

```
data_x, data_y = generate_XOR_easy()
pred_y, loss_list = run_Net(data_x,data_y,\
                    num_epochs = 500000, print_freq = 10000)
show_result(data_x,data_y,pred_y)s
```



Prediction Accuracy

- 計算方法:
  - 由於我們的 neural network 的輸出未必為整數，所以簡單以 `0.5` 為分界: 大於 `0.5` 就算 `1`，小於 `0.5` 就算 `0`；簡單粗暴

- 程式碼:

```
def acc(ground_truth, y):
    print ('Accuracy = ',100* (ground_truth[ground_tru
```

- Linear data accuracy

```
pred_y[pred_y>0.5] = 1
pred_y[pred_y<0.5] = 0
acc(data_y,pred_y)
```

```
Accuracy =  100.0 %
```

- XOR data accuracy

```
pred_y[pred_y>0.5] = 1
pred_y[pred_y<0.5] = 0
acc(data_y,pred_y)
```

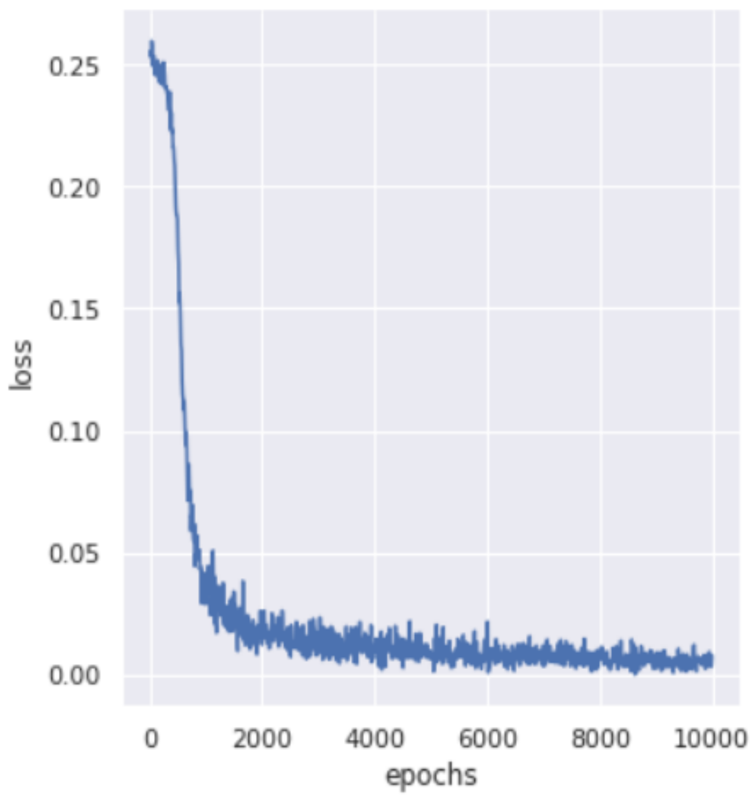
```
Accuracy =  100.0 %
```

Learning curves

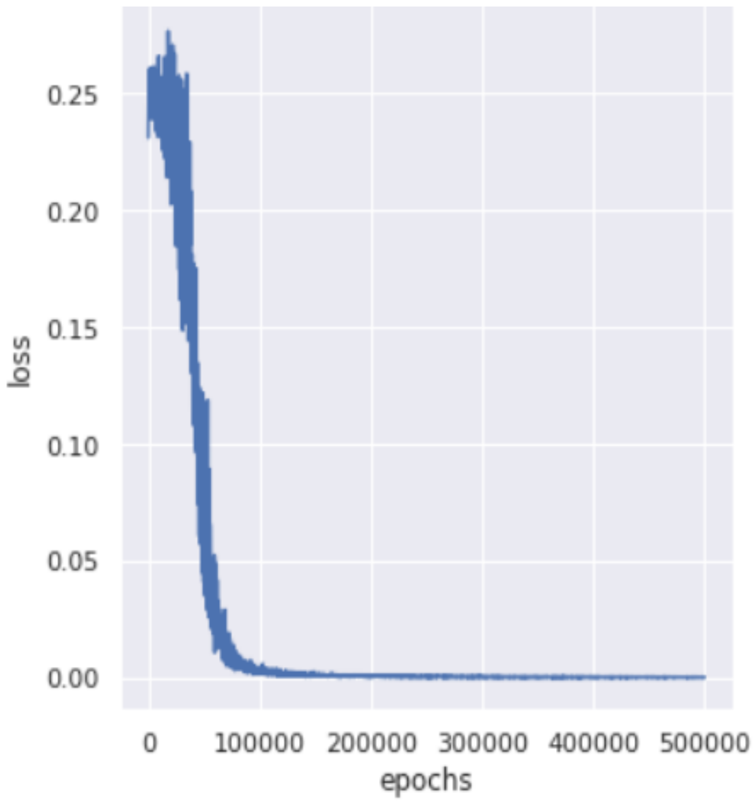
loss/epoch curve

- 透過 loss 對 epoch 做的 curve, 我們可以觀察訓練過程中, loss function 收斂的情況

- Linear data



- XOR data

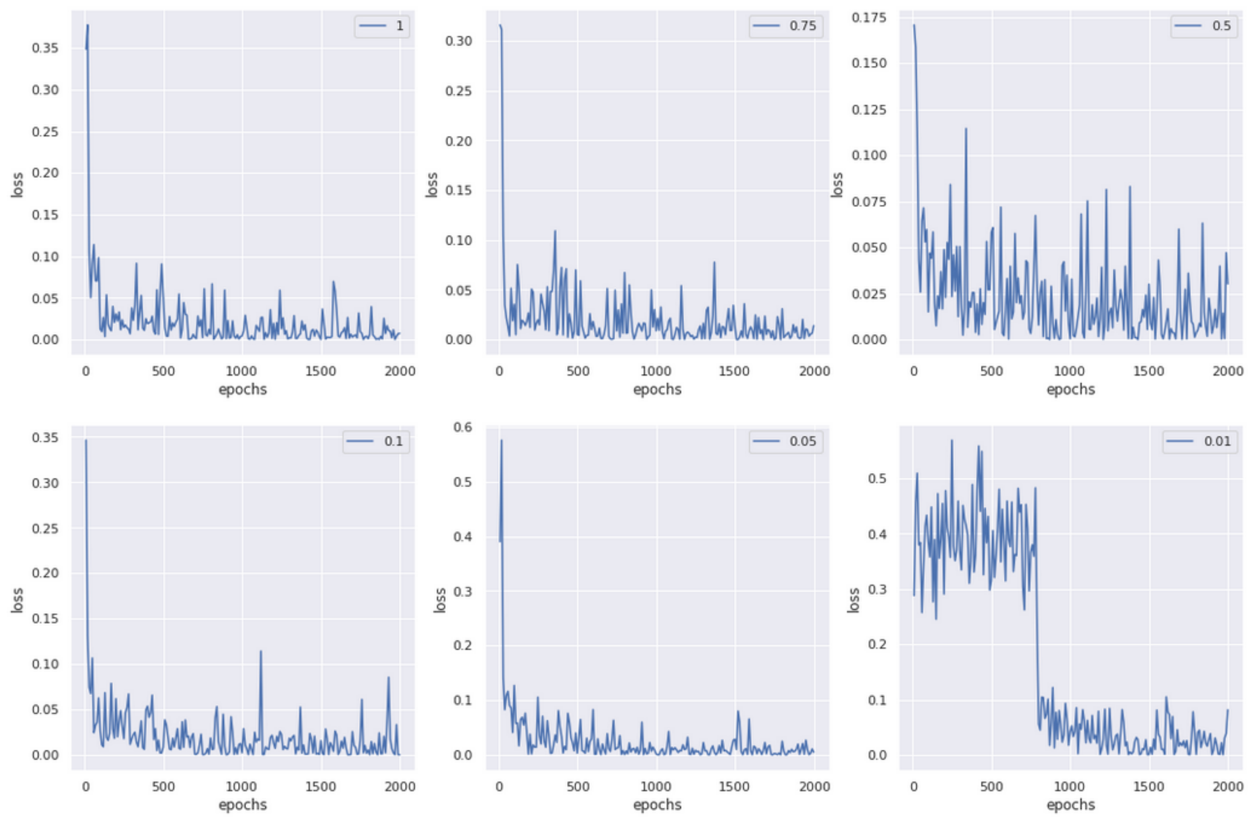


- 顯然，XOR 需要更多的 epochs 來使loss 收斂, 這大致上表示 XOR dataset

Discussion

Train with different learning rates

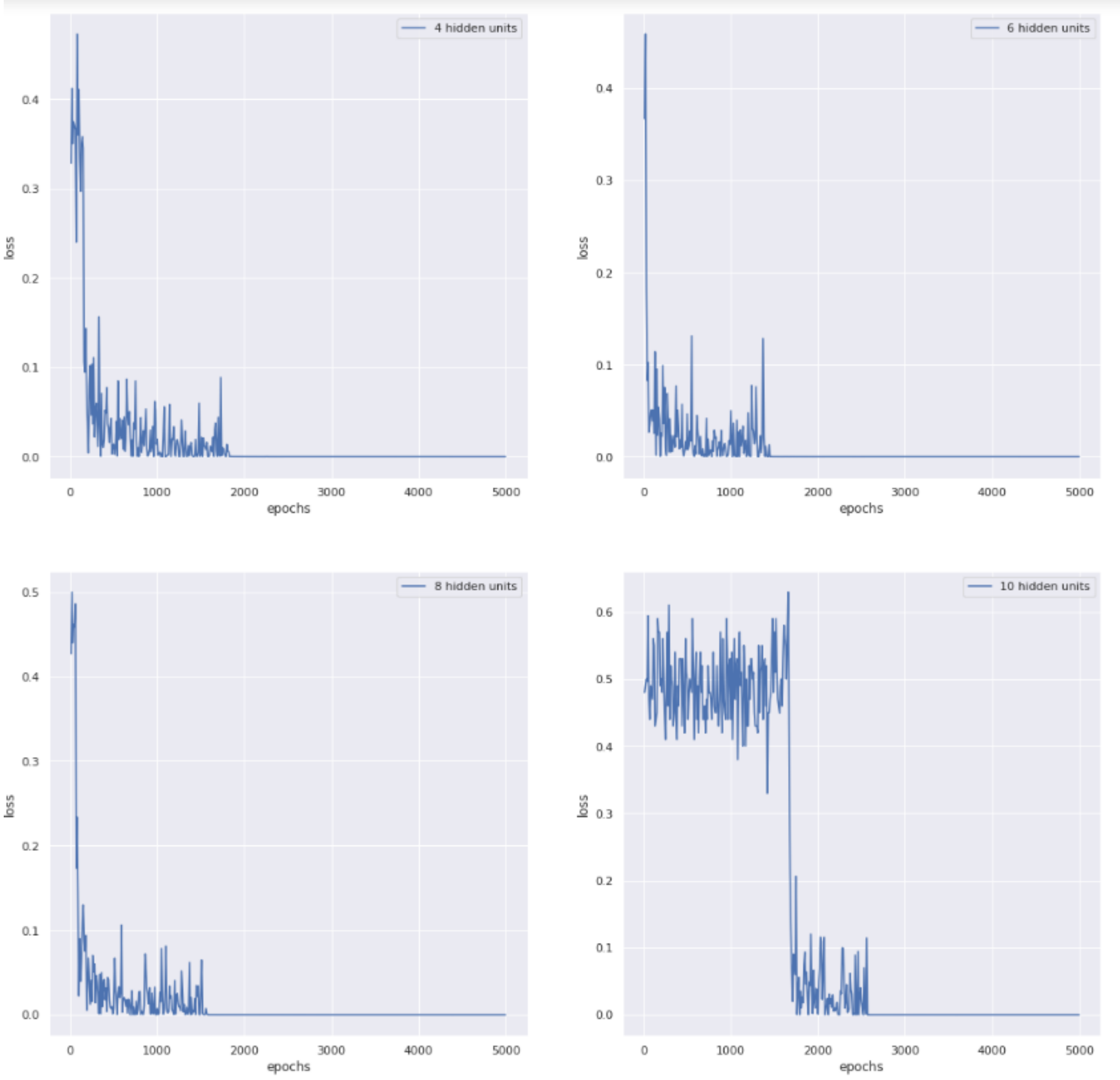
- The choose of learning rate is essential in training neural network
- We will train with lr = `1` , `0.75` , `0.5` , `0.1` , `0.05` , `0.01` , and see how they affect the training process:
  - 此實驗以 linear data 為例



- According to the figure above, when the learning rate is too small(for example, `0.01` in this case), the loss may not converge quick enough.

Train with different numbers of hidden units

- 上述的實驗皆僅用了 2 層 4 units 的 hidden layers
- 接下來我們嘗試不同的 hidden layer units 數來訓練,觀察其對訓練過程的影響
- Linear data



  - 可以看到, 雖然增加到 6 或 8 units 可以加快收斂速度, 但並不是越多越好. 一個可能的原因是題目太簡單, 使用 capacity 較高的 model, 反而可能要花更多時間達到收斂

Train without sigmoid function

- 在課堂中及 ***Deep Learning*** (https://www.deeplearningbook.org/), **Goodfellow et-al.** 書中皆有提到, neural network 的強大, 有一部份是來自於 activation function 能 fit nonlinear function; 若把 activation function 移除, 則神經網路只能學到 **linear function**; 這裡我們就試著把 sigmoid 層拿到, 來檢驗是否如此
- 程式碼

```
class TLNN(object):
    def __init__(self, layer_1_units = 4, layer_2_units =
        self.layers = OrderedDict()
        self.layers['linear_1'] = Linear(2,layer_1_units,b
        #self.layers['sigmoid_1'] = Sigmoid()
        self.layers['linear_2'] = Linear(layer_1_units,lay
        #self.layers['sigmoid_2'] = Sigmoid()
        self.layers['output'] = Linear(layer_2_units,1,bia
        self.layers['sigmoid_3'] = Sigmoid()

        self.loss_func = MSE()
        ...
```

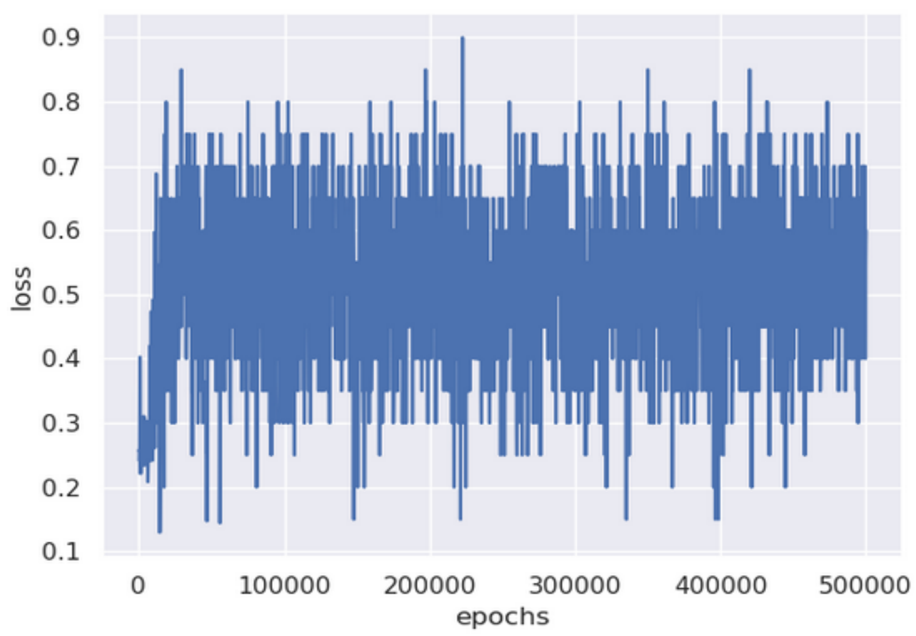
- 我們留下了最後一層的 sigmoid function, 否則似乎會變成無法更新 weight: 剛開始訓練就出現 loss 為 `nan` 的情況

- Training result - First Try
    - 使用各 `6` 個 hidden units 來訓練

```
(dl) ubuntu@ec037-003:~/dl/Lab1$ python3 train_XOR.py
/home/ubuntu/dl/Lab1/TLNN.py:84: RuntimeWarning: overflow encountered in squar
  return np.mean((self.y-self.ground_truth)**2)
train_XOR.py:20: RuntimeWarning: invalid value encountered in greater
  pred_y[pred_y>0.5] = 1
train_XOR.py:21: RuntimeWarning: invalid value encountered in less
  pred_y[pred_y<0.5] = 0
Accuracy =  0.0 %
```

        - ~~由於我們使用 MSE 作為 loss function, 在計算的時候出現了 overflow, 導致輸出為 `0`~~

- Training result - Second Try



    - 顯然, 即使我們訓練了 `500000` 次, 仍不見我們的神經網路有"學到東西"
    - 總結第一次沒成功的原因, 應該是 batch size 過大, 導致 MSE 計算產生 overflow (https://en.wikipedia.org/wiki/Integer_overflow) (雖然我們只設定 batch size = `20`)

Train with different loss functions

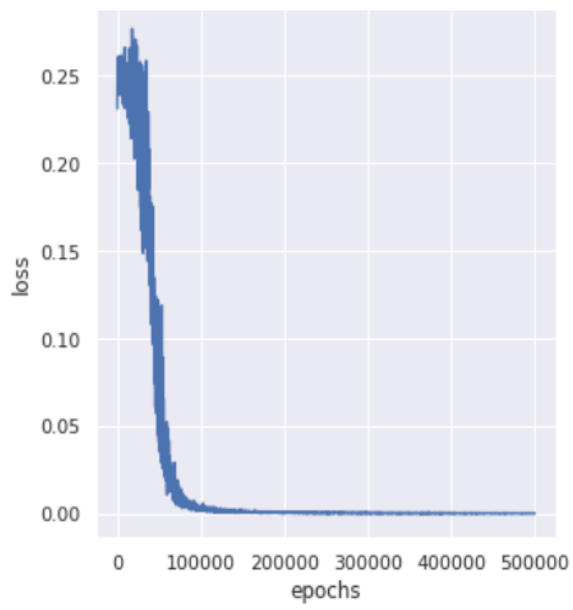
- Cross-Entropy

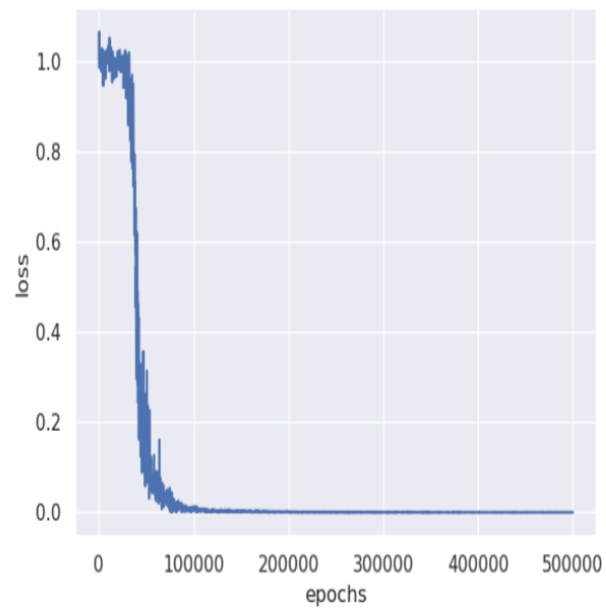
> In information theory, the ***cross entropy*** between two probability distributions p and q over the same underlying set of events measures the average number of bits needed to identify an event drawn from the set if a coding scheme used for the set is optimized for an estimated probability distribution q, rather than the true distribution p.
> The cross entropy of the distribution q relative to a distribution p over a given set is defined as follows:
> $H(p,q) = -E_p[\log q]$
>
> –*Wikipedia* *(https://en.wikipedia.org/wiki/Cross_entropy)*

  - 建議搭配 softmax 作為 output function 服用; Binary cross-entropy 則適合搭配~~溫開水~~ sigmoid output function

```
class Binary_Cross_Entropy(Layer):
def __init__(self):
    self.y = None
    self.ground_truth = None

def forward(self, y, ground_truth):
    # 使此 function 可以處理 batch ,也可以處理 single data
    if y.ndim == 1:
        ground_truth = ground_truth.reshape(1, ground_
        y = y.reshape(1, y.size)

    batch_size = y.shape[0]
    self.y = y
    self.ground_truth = ground_truth

    # To avoid -inf; 重要！！
    delta = 1e-7

    return (-1/ batch_size)*np.sum(self.ground_truth *

def backward(self,prev_grad=1,lr=0.1):
    '''
        prev_grad, lr: pseudo parameters
    '''
    batch_size = self.y.shape[0]

    # To avoid -inf;
    delta = 1e-7
    dx = - (np.divide(self.ground_truth, self.y + delt
    return dx
```

- 實驗設定
  - Dataset: XOR data
  - Activation function: `sigmoid()`
  - Loss function : Binary cross-entropy
- 實驗結果





  - 左圖是以 MSE 作為 loss function; 而右圖則是以 binary cross-entropy 作為 loss function
  - 可以見到兩個的收斂範圍都在 `100000` 次左右達到穩定,

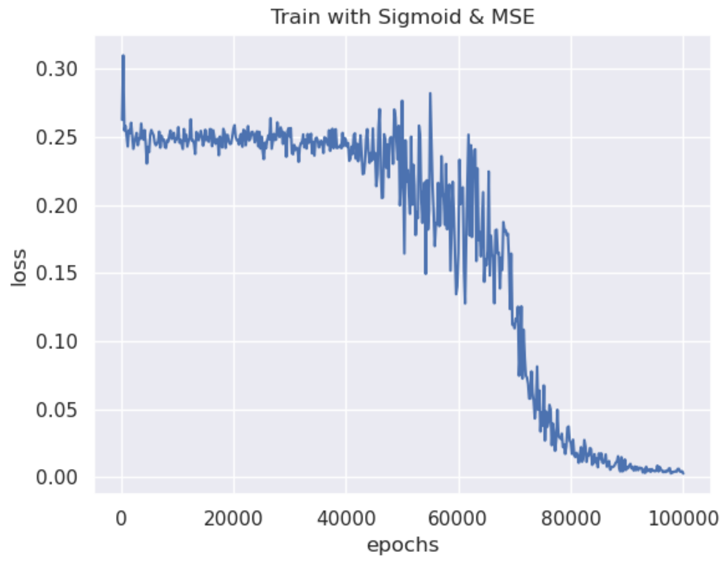
Train with different activation function

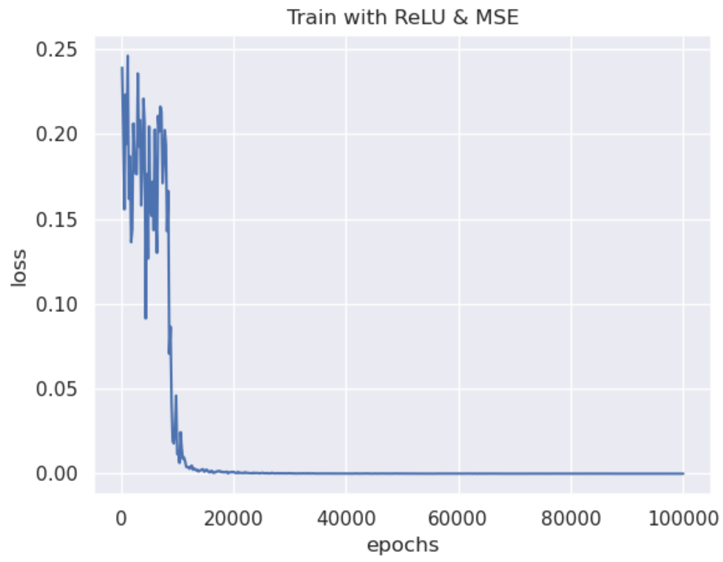
- ReLU
  - A simple function : $f(x)=\max(0,x)$
  - 其微分為 $\frac{\partial f(x)}{\partial x}=\begin{cases}1, & f(x)>0\\0, & \text{otherwise}\end{cases}$
  - 程式碼實做

```
class ReLU(Layer):
    def __init__(self):
        self.mask = None
        self.y = None

    def forward(self, x):
        self.mask = (x <= 0)
        self.y = x.copy()
        self.y[self.mask] = 0
        return self.y

    def backward(self, prev_grad,lr=0.1):
        # Return prev_grad * derivative of sigmoid fur
        grad = prev_grad
        grad[self.mask] = 0
        return grad
```

  - 實驗設定

    - Dataset: XOR data
    - Activation function: `ReLU()`
    - Loss function : Mean square error
  - 實驗結果

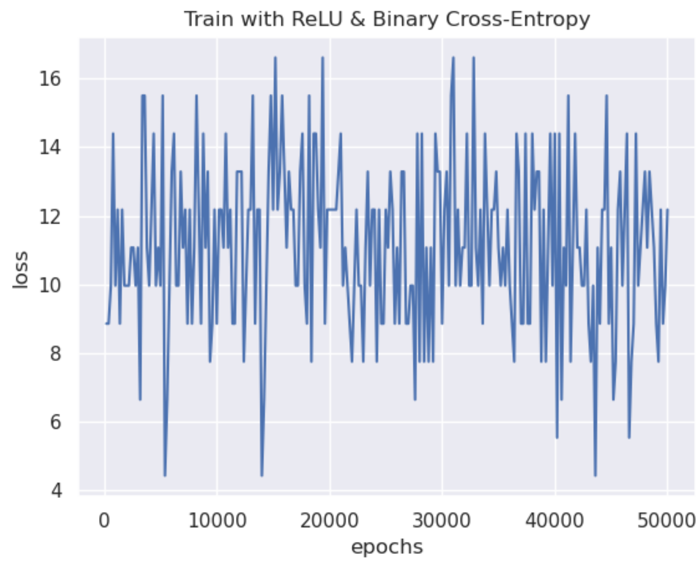




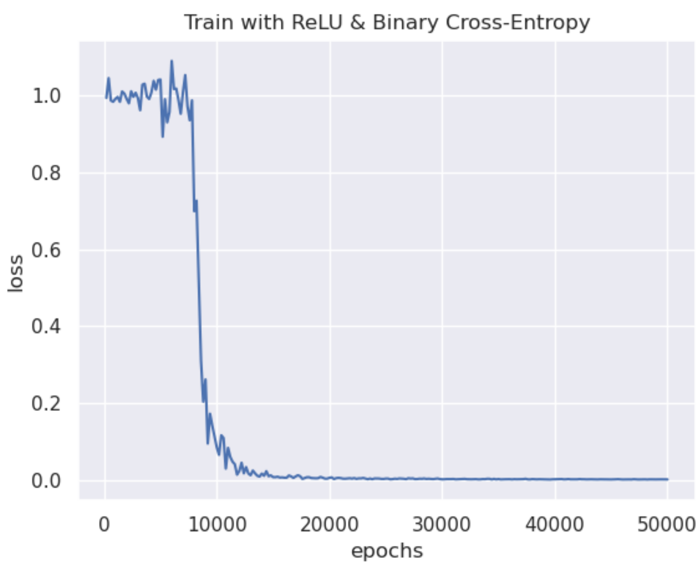
  - 從圖中我們可以很明顯地發現, 使用 `ReLU` function 在這個條件下, loss function 的收斂速度確實較快

Train with ReLU & Binary Cross-Entropy

- 接著, 我們把 activation function 跟 loss function 都換掉試試看
- 實驗設定
  - Dataset: XOR data
  - Activation function: `ReLU()`
  - Loss function : Binary cross-entropy
  - Learning rate: 0.01
- 第一次實驗結果



  - 未達收斂！
  - 分析原因: 即使視同一個 dataset, 不同的 loss function, 適用的 learning rate 可能也不同. 可能因此而導致 model 訓練未果
  - 進行第二次實驗
- 第二次實驗設定
  * Dataset: XOR data
  * Activation function: `ReLU()`
  * Loss function : Binary cross-entropy
  * Learning rate: 0.001
- 第二次實驗結果



  - 結果非常成功
  - 由此實驗可知, learning rate 的調整是非常值得注意的環節; 這可能也是 optimizer 值得研究的原因之一

Bonus: Train with Different Learning Rate (II)

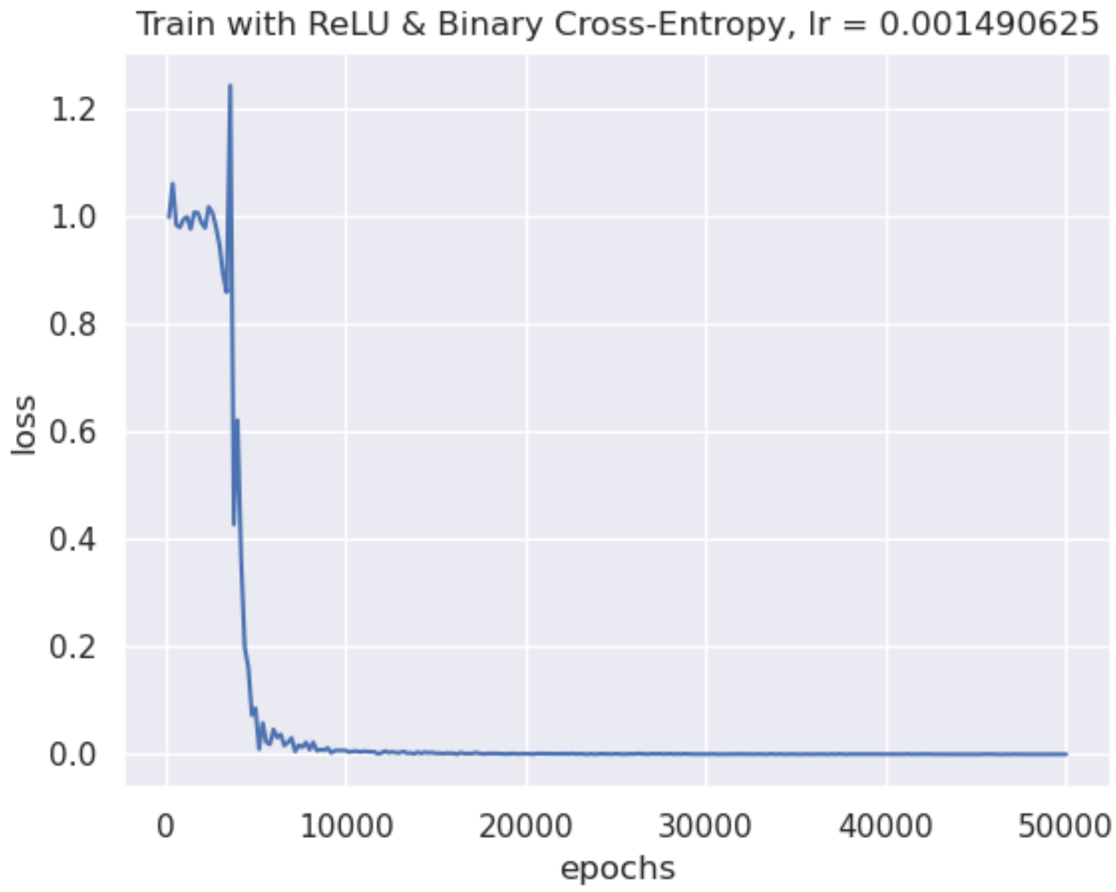
- 由於在上個實驗中, learning rate 確實扮演了重要的角色, 所以再度做了跟 learning rate 有關的實驗

- 我們手動調整 learning rate 的值, 希望找到接近發散與收斂的臨界值

- 實驗設定 (I)

- Dataset: XOR data
- Activation function: `ReLU()`
- Loss function : Binary cross-entropy
- Learning rate: 0.001490625

- 實驗結果 (I)



- 可以見到 loss 成功收斂, 沒有問題

- 實驗設定 (II)
  - Dataset: XOR data
  - Activation function: `ReLU()`
  - Loss function : Binary cross-entropy
  - Learning rate: 0.001490626

- 實驗結果 (II)

- 重複做了幾次, 也改變了 random seed, 但當 learning rate 增加 `1e-9` , 結果卻大不相同
- 猜測原因大概就是浮點數的限制 (https://hackmd.io/@sysprog/c-floating-point)
- 正因如此, 進行 neural network 實做的我們, 才更應該掌握這些計算機架構的背景知識, 以免徒耗心力
- 相關連結：軟體缺失導致的危害 (https://hackmd.io/@sysprog/software-failure)

Problem Discussions

Input & output dimension

- 一開始定義 Linear layer 的時候，我用的是 $y = WX + b$ 的形式
  - 在計算 $\frac{\partial L}{\partial W}$ 時，結果將會是 $\frac{\partial L}{\partial y} \cdot X^T$
  - 但當 X 是長度為 n 的vector, numpy 不會將其轉為 $(1, n)$ 的 matrix, 而是保留原長度
  - 解決方法:
    1. 可以透過判斷 X 的 shape 並手動更改
    2. 將上述式子改為 $y = XW + b$, 並計算 $\frac{\partial L}{\partial y} \cdot X^T$ 的結果
       - Pros: 可以無須對 X 作轉置，且為通常使用的方法
       - Cons: 跟 Lineaer Algebra 通常把 input 放後面的寫法不一樣了，個人私心不想這麼做(最後妥協了´`)
    3. 直接改為丟 mini batch 進去
       - 雖然成功解決了 X 為 vector 無法轉置的問題(因為變成 matrix 了)，但後續在計算 $y = WX + b$ 時, b 也遇到了相同的情形，而無法做 broadcast 加法

(https://numpy.org/doc/stable/user/basics.broadcasting.html)

- 結果: 放棄，改用上述 2. 的作法

RuntimeWarning: invalid value encountered in log2

- 發生在以 `ReLU` 作為 output function, binary cross-entropy 時產生的
    - `ReLU` 是一種 activation function, 並非用來當作 output function 使用
        - 原因： Output function 通常是根據問題的類別來定義的. 如：學習 Bernoulli distrubution (https://en.wikipedia.org/wiki/Bernoulli_distribution)(二元分類問題) 的 model 通常以 `sigmoid` 或 `tanh` 作為 output function, 因為 ground truth 為 `0` 或 `1` ; 而 Multinoulli Distribution (https://en.wikipedia.org/wiki/Categorical_distribution) 則常用 `softmax` 作為 output function
    - `np.log2` 的 input 值必須大於 `0` . 摁, 國中數學,但被我忘了

- 解決方法: 用 `sigmoid` 作為 output function, 只改變 hidden layer 的 activation function 為 `ReLU`

- 另外, 由於 $\log_2(0) = -\infty$, 所以實做時, 記得加上一個 `delta=1e-7` 來避免這種情況的產生