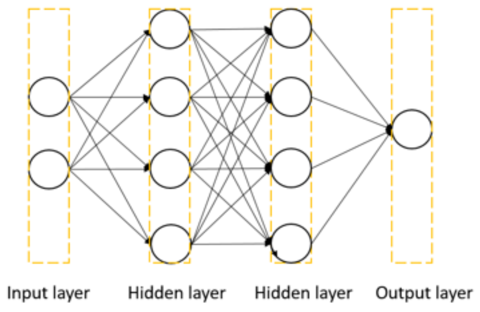
Introduction

In this lab, I will implement a simple vanilla neural network with **2** hidden layers by executing forward pass and backpropagation from scratch, using only the ***Numpy*** and other standard libraries in Python.



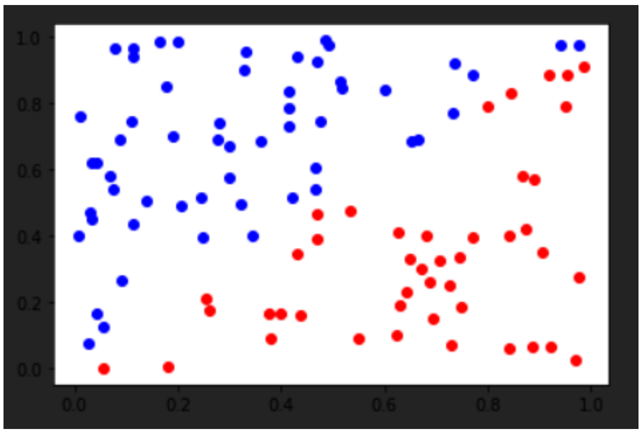
The data we're given can be classified into 2 groups, with their ***labels*** being **0** and **1**, respectively. Each data point has **2** features, namely *x-coordinate* and *y-coordinate*.
This neural network is aimed to try classfying the data into 2 groups. That is to say, we are solving a ***logistic regression problem***. Also, since we are solving such a binary classification problem, our loss function used in the output layer is the ***binary cross entropy loss***.

A. The Data Generator

They are 2 types of data generator used in this lab. The first one is a ***linear-seperated*** data generator. The code script is shown below

```python
def generate_linear(n = 100):
    pts = np.random.uniform(0, 1, (n, 2))
    inputs = []
    labels = []

    for pt in pts:
        inputs.append([pt[0], pt[1]])
        if pt[0] > pt[1]:
            labels.append(0)
        else:
            labels.append(1)

    return np.array(inputs), np.array(labels).reshape(n, 1)
```

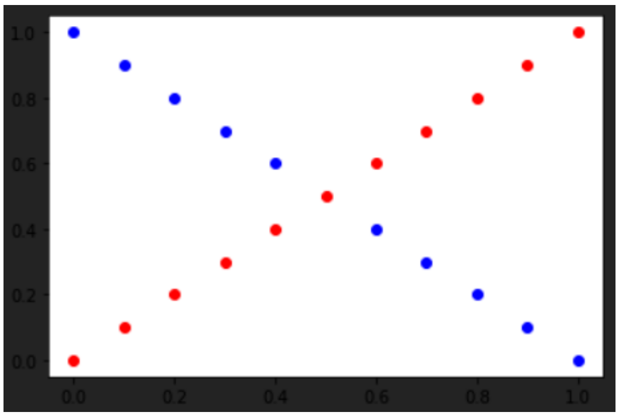
Each data points has an *x-coordinate* and a *y-coordinate* (**2 features**). The generator output 2 things: the data with shape $(100, 2)$ and their labels with shape $(100, 1)$. Both are 2d numpy array.
The data points scatter like so:



All the points have both their x and y coordinated lie within $(0,1)$ range, and the two groups are linearly seperated by the line $y=x$.
The second generator is a ***XOR*** data generator, which generates data in XOR-like pattern. The code script is as follows:

```python
def generate_XOR_easy():
    inputs = []
    labels = []

    for i in range(11):
        inputs.append([0.1 * i, 0.1 * i])
        labels.append(0)

        if 0.1 * i == 0.5:
            continue

        inputs.append([0.1 * i, 1 - 0.1 * i])
        labels.append(1)

    return np.array(inputs), np.array(labels).reshape(21, 1)
```

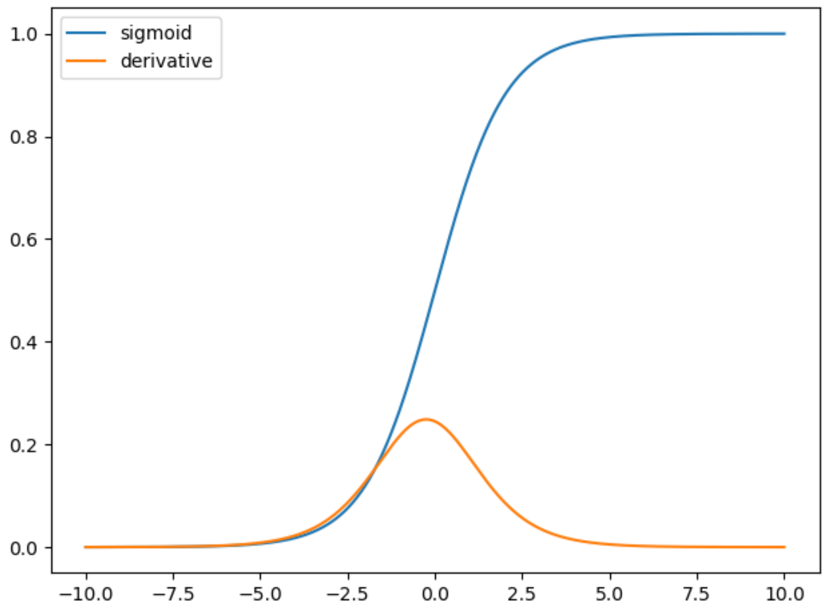
Similarly, each data point has **2** features. The generator outputs: the data with shape $(21,2)$ and their labels with shape $(21,1)$. Both are 2d numpy array. The data scatter like so,



All the points form a cross lay within the square defined by the 4 corners $(0,0),(0,1),(1,0),(1,1)$
The middle point with the coordinate $(0.5,0.5)$ belongs to the diagonal which spans from $(0,0)$ to $(1,1)$.

Experiment Setups

A. The Sigmoid Function



The ***sigmoid function*** is defined as

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

```
def sigmoid(x):
    sigmoid_vec = np.vectorize(lambda t: 1 / (1 + np.exp(-t)) if t > 0
    else 1 - 1 / (1 + np.exp(t)))
    return sigmoid_vec(x)
```

If the input x is **positive**, use the form $\frac{1}{1+e^{-x}}$. Otherwise, use the form $1 - \frac{1}{1+e^{x}}$ instead.
The reason why I seperate the input into 2 cases, is that sometimes x might be a **very small negative value**, such as *-4294967296* or something far smaller. Thus, $-x$ would then be very big, causing overflow problem. To prevent this, use the other form when x is **negative**.
Since the input x is likely a *vector* or even *matrix*, I use the ***vectorized*** function so as to consider each entry in x individually.
The ***first derivative*** of the sigmoid function can be obtained as

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = (\frac{1}{1+e^{-x}})(\frac{e^{-x}}{1+e^{-x}})$$
$$= \sigma(x)(1-\sigma(x))$$

```python
def derivative_sigmoid(x):
    return np.multiply(x, 1.0 - x)
```

Please note that **the input of function `derivative_sigmoid()` , x should be some output value from the sigmoid function**. $x \in (0, 1)$, that is.

B. Neural Network

In the fundamental settings, both **numbers of neurons** in the 2 **hidden layers** are set to **10**.

```python
self.n1 = n1 # 10
self.n2 = n2 # 10
```

The **learning rate** is set to be **0.1**.

```python
self.lr = lr # 0.1
```

Also, I implemented an easy **learning rate scheduler**

> The **later half** of epochs were trained with **0.5** times the initial learning rate.

```python
def train(self, X, y):
    for e in range(self.epochs):
        if e == round(self.epochs / 2):
            self.lr /= 2 # scheduler
        ......
```

The **activation function** is the **sigmoid function**, which we just introduced.
Since the data from both generators are pretty simple, we only consider ***weights*** in our network for simplicity.
However, if the data pattern becomes relatively complicated, it is dispensible that one include the ***bias terms*** in their networks for reaching the convergence.

Anyway, here are my network structure and model parameters.

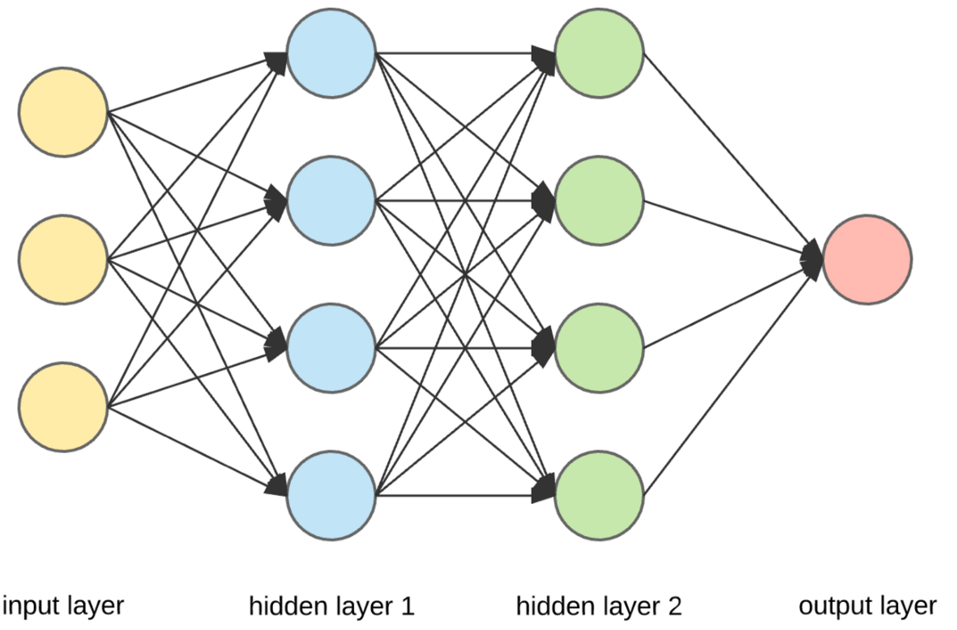


image source: **[https://r23456999.medium.com/何謂-artificial-neural-netwrok-33c546c94794](https://r23456999.medium.com/%E4%BD%95%E8%AC%82-artificial-neural-netwrok-33c546c94794)**

(https://r23456999.medium.com/%E4%BD%95%E8%AC%82-artificial-neural-netwrok-33c546c94794)

This image is only for reference! Its numbers of neurons are not the same as my network!

```
self.input = np.random.rand(2, 1)
self.W1 = np.random.rand(n1, 2)
self.W2 = np.random.rand(n2, n1)
self.W3 = np.random.rand(1, n2)
self.H1 = np.random.rand(n1, 1)
self.H2 = np.random.rand(n2, 1)
self.H3 = np.random.rand(1, 1)
self.Z1 = np.random.rand(n1, 1)
self.Z2 = np.random.rand(n2, 1)
self.output = np.random.rand(1, 1)
```

My ***input layer*** has only **2** neurons, since each data point has only **2** features, namely *x-coordinate* and *y-coordinate*. Thus, `self.input` has the shape of $(2, 1)$, a column vector.
The values of the neurons in ***hidden layer 1*** are stored in `self.H1` and `self.Z1`. Both of them have the shape of $(n_1, 1) = (10, 1)$, since we have **10** neurons in hidden layer 1. The difference between `self.H1` and `self.Z1` is that `self.H1` saves the *original values* while `self.Z1` saves the values *after passing the activation function* (**sigmoid function** in the basic settings).
Similarly, `self.H2` and `self.Z2` save the *original values* and the *values after activation function* in ***hidden layer 2***, respectively. Both of them have the shape of

$(n_2, 1) = (10, 1)$, since we have **10** neurons in hidden layer 2.
As for the ***output layer***, there is only **1** neuron. `self.H3` saves the *original value* while `self.output` saves the *values after sigmoid function*. Both of `self.H3` and `self.output` have the shape $(1, 1)$. The value in this layer is **the output value**, which will be the prediction once rounded (becomes either 0 or 1).
Finally, the ***weight matrixes*** between different layers.
`self.W1` is the weight matrix between *input layer* and *hidden layer 1*, with the shape of $(n_1, 2) = (10, 2)$.
`self.W2` is the weight matrix between *hidden layer 1* and *hidden layer 2*, with the shape of $(n_2, n_1) = (10, 10)$.
`self.W3` is the weight matrix between *hidden layer 2* and *output layer*, with the shape of $(1, n_2) = (1, 10)$.
All these model parameters are randomly initialized by `numpy.random.rand()` .

Since we are doing the binary classification problem, we chose the ***binary cross entropy*** as our loss function. Let $\vec{y}$ be the **ground truth labels** and $\vec{pred\_y}$ be the **predictions**. Then, the loss can be computed as

$$L(\vec{y}, \vec{pred\_y}) = -\vec{y}^T \log(\vec{pred\_y} + \epsilon \cdot \vec{1}) - (\vec{1} - \vec{y})^T \log(\vec{1} - \vec{pred\_y} + \epsilon \cdot \vec{1})$$

where $\epsilon = 10^{-9}$ in my code implementation. Note that this tiny value was added to prevent invalid values while taking logarithm operations.

```
self.epsilon = 1e-9
......
def loss_fn(self, y, pred_y):
    '''cross-entropy'''
    return -np.matmul(y.T, np.log(pred_y + self.epsilon)) - np.matmul((1-y).T,
    np.log(1 - pred_y + self.epsilon))
```

Forward Pass

Now, for the **forward pass**, we just need to *cascade* the model parameters in the same order as the predefined network structure.

```
def forward(self, X):
    self.input = X
    self.H1 = np.matmul(self.W1, self.input)
    if self.activation_fn:
        self.Z1 = sigmoid(self.H1)
    else:
        self.Z1 = self.H1
    self.H2 = np.matmul(self.W2, self.Z1)
    if self.activation_fn:
        self.Z2 = sigmoid(self.H2)
    else:
        self.Z2 = self.H2
    self.H3 = np.matmul(self.W3, self.Z2)
    self.output = sigmoid(self.H3)

    return self.output
```

Each time a single data point X is introduced in the model, we receive the 2 values by **the input neurons**, stored in `self.input` . And then, the n_1 size of *original* values right before passing to hidden layer 1, H_1, is calculated as

$$H_1 = W_1 X$$

And then after passing through the activation function, we get the values of neurons in **hidden layer 1**, stored in Z_1.

$$Z_1 = \sigma(H_1)$$

Note that in the **Discussion** session later on, we will also try training ***without activation function***. So in that case, Z_1 is just the same as H_1.

$$Z_1 = H_1$$

Likewise, for **hidden layer 2**, H_2 stores the n_2 size of values before activation function

$$H_2 = W_2 Z_1$$

If training with activation function, then the values of neurons in **hidden layer 2** are

$$Z_2 = \sigma(H_2)$$

Otherwise,

$$Z_2 = H_2$$

Finally, the *original value* before activation function in **output layer** is stored in H_3

$$H_3 = W_3 Z_2$$

After the sigmoid function, we get the **output value**, `self.output`

$$output = \sigma(H_3)$$

Note that whether we are training with the activation or not, **the sigmoid function of the output layer must not be changed or plugged out!**. This is because the function constraints the values into $(0, 1)$ interval, making the binary prediction possible.
If not, we are unable to make predictions, let alone computing loss and accuracy.

> As the for the ***backpropagation***, please refer to the next session.

Train and Test

```
def train(self, X, y):
    for e in range(self.epochs):
        if e == round(self.epochs / 2):
            self.lr /= 2 # scheduler
        for i in range(X.shape[0]):
            self.output = self.forward(X[i: (i + 1), :].T)
            self.backward(y[i: (i + 1), :], self.output)

        if e % self.interval == 0:
            print('Epochs {}: ,'.format(e), end = '')
            self.test(X, y)

    print('Training finished')
    self.test(X, y)
    show_loss(self.loss_arr, self.epochs, self.interval)

def test(self, X, y):
    error = 0.0
    pred_y = np.zeros_like(y, dtype = float)
    for i in range(X.shape[0]):
        output = self.forward(X[i: (i + 1), :].T)
        pred_y[i, 0] = output.item()
        result = np.round(output)
        error += abs(result - y[i: (i + 1), :])
    error /= X.shape[0]
    loss = self.loss_fn(y, pred_y)
    self.loss_arr.append(loss[0][0])
    print(f"accuracy: {((1 - error) * 100)[0][0]:.2f}%, loss: {loss[0][0]:.5f}")
    print('')
```

In the training session, accompany the initial learning rate being **0.1**, **epochs = 1000** for the data generated by the ***linearly-separated generator*** and **epoch = 2500** for the data generated by the ***XOR generator*** are the best choices, since the training will most likely to reach ***100%*** in these epochs. These 2 values are determined by several try and error.
In every epochs, *a single data point* from X is transposed into a column vector, and feed to the model. After the network output a value by ***forward pass***, we use the **output value** as well as the **groud truth label** to conduct ***backwardpropagation***.

> Compute the gradients and update the weights. Details will be described later in ***backpropagation*** part.

Every `self.interval` $= 100$ epochs, we will compute the current **loss** and **accuracy**. We will store the losses every 100 epochs as well, for the purpose of *drawing the **loss-to-epoch curve** by function* `show_loss` .

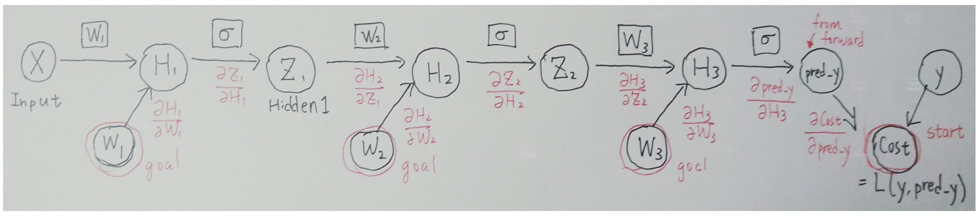
C. Backpropagation

```
def backward(self, y, pred_y):
    Cost_to_predy = (1-y) / (1-pred_y + self.epsilon) - y / (pred_y + self.epsilon
    predy_to_H3 = np.diag(derivative_sigmoid(self.output).reshape(-1))
    H3_to_W3 = self.Z2.T
    Cost_to_H3 = np.matmul(Cost_to_predy, predy_to_H3)
    Cost_to_W3 = np.matmul(Cost_to_H3, H3_to_W3)

    H3_to_Z2 = self.W3
    if self.activation_fn:
        Z2_to_H2 = np.diag(derivative_sigmoid(self.Z2).reshape(-1))
    else:
        Z2_to_H2 = np.identity(derivative_sigmoid(self.Z2).reshape(-1).shape[0])
    H2_to_W2 = np.zeros((self.H2.shape[0], self.W2.shape[0] * self.W2.shape[1]))
    n, m = self.H2.shape[0], self.W2.shape[1]
    for i in range(n):
        H2_to_W2[i, i * m: (i+1) * m] = self.Z1.reshape(-1)
    Cost_to_H2 = np.linalg.multi_dot([Cost_to_H3, H3_to_Z2, Z2_to_H2])
    Cost_to_W2 = np.matmul(Cost_to_H2, H2_to_W2)
    Cost_to_W2 = Cost_to_W2.reshape(n, m)

    H2_to_Z1 = self.W2
    if self.activation_fn:
        Z1_to_H1 = np.diag(derivative_sigmoid(self.Z1).reshape(-1))
    else:
        Z1_to_H1 = np.identity(derivative_sigmoid(self.Z1).reshape(-1).shape[0])
    H1_to_W1 = np.zeros((self.H1.shape[0], self.W1.shape[0] * self.W1.shape[1]))
    n, m = self.H1.shape[0], self.W1.shape[1]
    for i in range(n):
        H1_to_W1[i, i * m: (i + 1) * m] = self.input.reshape(-1)
    Cost_to_H1 = np.linalg.multi_dot([Cost_to_H2, H2_to_Z1, Z1_to_H1])
    Cost_to_W1 = np.matmul(Cost_to_H1, H1_to_W1)
    Cost_to_W1 = Cost_to_W1.reshape(n, m)

    self.W1 -= self.lr * Cost_to_W1
    self.W2 -= self.lr * Cost_to_W2
    self.W3 -= self.lr * Cost_to_W3
    return
```

I implement backpropagation by the help of ***computational graph***. The reference material is this tutorial video: **https://youtu.be/-yhm3WdGFok**

(https://youtu.be/-yhm3WdGFok)

I will directly explain the gradient results without any derivation. If you are interested, you can refer to the video above.



The above figure is the **computational graph** of my neural network. From the **forward pass**, we got the **prediced** value $pred\_y$, and we have the **ground truth** value of the data y. Thus, we can calculate the **cost**, $L(y, pred\_y)$.

$$Cost = -y\log(pred\_y + \epsilon) - (1-y)\log(1 - pred\_y + \epsilon)$$
$$where\ \epsilon = 10^{-9}$$

To update the 3 weight matrixes W_1, W_2, W_3, what we need are the gradients of the **cost** with respect to **these matrixes**. For this, we can simply multiply all the gradients along side the path from the cost to these matrixes.

$$\frac{\partial Cost}{\partial W_3} = (\frac{\partial Cost}{\partial pred\_y})(\frac{\partial pred\_y}{\partial H_3})(\frac{\partial H_3}{\partial W_3})$$
$$\frac{\partial Cost}{\partial W_2} = (\frac{\partial Cost}{\partial pred\_y})(\frac{\partial pred\_y}{\partial H_3})(\frac{\partial H_3}{\partial Z_2})(\frac{\partial Z_2}{\partial H_2})(\frac{\partial H_2}{\partial W_2})$$
$$\frac{\partial Cost}{\partial W_1} = (\frac{\partial Cost}{\partial pred\_y})(\frac{\partial pred\_y}{\partial H_3})(\frac{\partial H_3}{\partial Z_2})(\frac{\partial Z_2}{\partial H_2})(\frac{\partial H_2}{\partial Z_1})(\frac{\partial Z_1}{\partial H_1})(\frac{\partial H_1}{\partial W_1})$$

Firstly, for $\frac{\partial Cost}{\partial pred\_y}$, it's just the derivative of the binary cross entropy loss w.r.t. $pred\_y$.

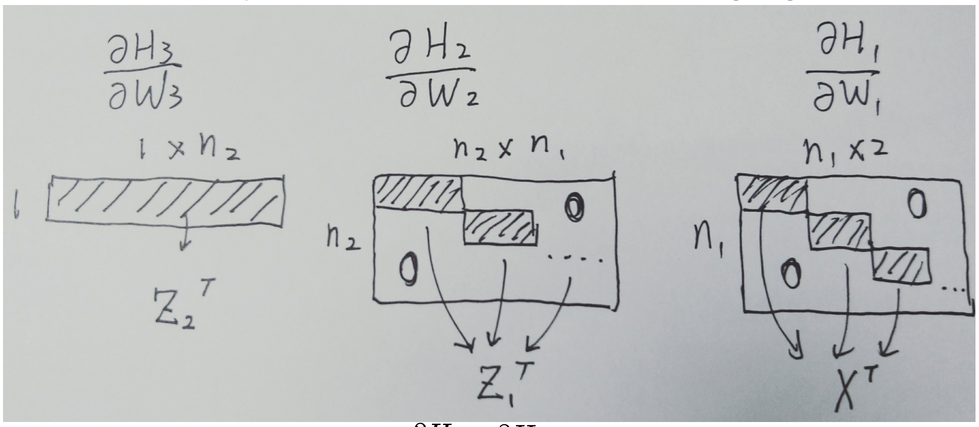
$$\begin{aligned}\frac{\partial Cost}{\partial pred\_y} &= \frac{\partial}{\partial pred\_y}(-y\log(pred\_y+\epsilon) - (1-y)\log(1-pred\_y+\epsilon)) \\ &= -\frac{y}{pred\_y+\epsilon} + \frac{1-y}{1-pred\_y+\epsilon}\end{aligned}$$

Note that we also add the tiny value $\epsilon = 10^{-9}$ here, to prevent dividing from 0.
A little reminder before we go on: for $\frac{\partial \vec{a}}{\partial \vec{b}}$, the result is actually a ***Jacobian Matrix*** where the row number is the length of $\vec{a}$ and the column number is the length of $\vec{b}$.
Now, for $\frac{\partial pred\_y}{\partial H_3}, \frac{\partial Z_2}{\partial H_2}, \frac{\partial Z_1}{\partial H_1}$, since the edges represent ***sigmoid*** operations, we need to introduce the derivative of the sigmoid function:

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$$

These 3 matrixes are all in fact, **diagonal matrixes** under the activation function being *sigmoid function*, with the size of $(1,1), (n_2, n_2), (n_1, n_1)$, respectively. ($n_1 = n_2 = 10$ by default)
The diagonal entries of these 3 matrixes are $pred\_y, Z_2, Z_1$ flatten, respectively.

The 3 gradients $\frac{\partial H_3}{\partial W_3}, \frac{\partial H_2}{\partial W_2}, \frac{\partial H_1}{\partial W_1}$ are somehow much more complicated. These 3 matrixes have the shape of $(1, 1 \times n_2), (n_2, n_2 \times n_1), (n_1, n_1 \times 2)$, respectively. Note that these matrixes should have been *3-dimensional*, but for easier understanding, we flatten the W_1, W_2, W_3 into vectors. Now, the Jacobian matrixes of these derivatives become 2d matrixes.
As for how one construct these matrixes, firstly, initialize **zero matrixes**. And then gradually fill in the correct vectors as the pattern shown in the following figure.



Lastly, for the gradients $\frac{\partial H_3}{\partial Z_2}, \frac{\partial H_2}{\partial Z_1}$, they are just the W_3 and W_2, respectively. Their shapes are $(1, n_2)$ and (n_2, n_1). Note that for the edges corresponding to **sigmoid operations**, namely $\frac{\partial pred\_y}{\partial H_3}, \frac{\partial Z_2}{\partial H_2}, \frac{\partial Z_1}{\partial H_1}$, besides $\frac{\partial pred\_y}{\partial H_3}$, the other two square matrixes will be just ***identity matrixes*** I if we are working **without activation function**. And again, the reason why $\frac{\partial pred\_y}{\partial H_3}$ was kept the same is because that we need to make a binary prediction at the output layer, so the sigmoid function is required.

Finally, after we computed all the gradients above, and got

$$\frac{\partial Cost}{\partial W_3}, \frac{\partial Cost}{\partial W_2}, \frac{\partial Cost}{\partial W_1}$$

these 3 gradients, we can **update the weight matrixes** W_1, W_2, W_3 as

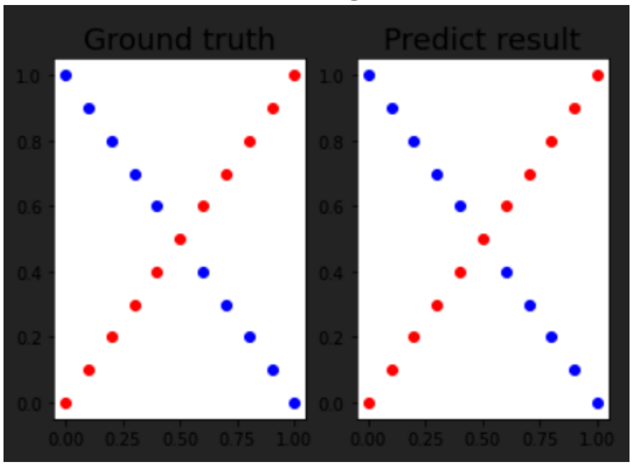
$$W_1^{(new)} = W_1 - lr \times \frac{\partial Cost}{\partial W_1}$$
$$W_2^{(new)} = W_2 - lr \times \frac{\partial Cost}{\partial W_2}$$
$$W_3^{(new)} = W_3 - lr \times \frac{\partial Cost}{\partial W_3}$$

where lr is the learning rate, set to be **0.1** in the fundamental settings.

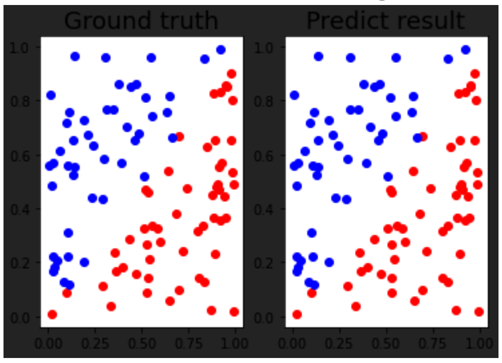
Results of Testing

A. Screenshot and Comparison Figure

Left subplots are the **ground truths**, while the **right** subplots are the **predict results**.
Blue and **red** colors denote 2 different groups.
The data from XOR generator

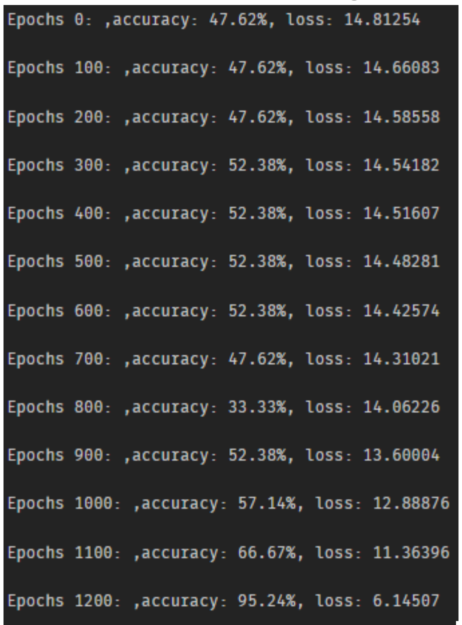


The data from linear generator

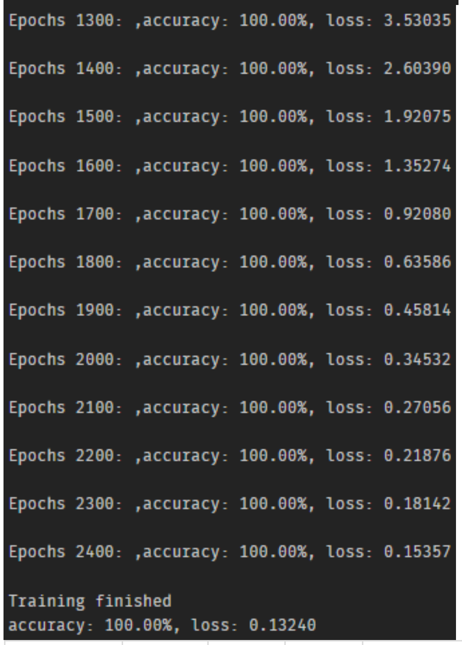


B. The Accuracy of prediction

The data from XOR generator



```
Epochs 0: ,accuracy: 47.62%, loss: 14.81254

Epochs 100: ,accuracy: 47.62%, loss: 14.66083

Epochs 200: ,accuracy: 47.62%, loss: 14.58558

Epochs 300: ,accuracy: 52.38%, loss: 14.54182

Epochs 400: ,accuracy: 52.38%, loss: 14.51607

Epochs 500: ,accuracy: 52.38%, loss: 14.48281

Epochs 600: ,accuracy: 52.38%, loss: 14.42574

Epochs 700: ,accuracy: 47.62%, loss: 14.31021

Epochs 800: ,accuracy: 33.33%, loss: 14.06226

Epochs 900: ,accuracy: 52.38%, loss: 13.60004

Epochs 1000: ,accuracy: 57.14%, loss: 12.88876

Epochs 1100: ,accuracy: 66.67%, loss: 11.36396

Epochs 1200: ,accuracy: 95.24%, loss: 6.14507
```



```
Epochs 1300: ,accuracy: 100.00%, loss: 3.53035

Epochs 1400: ,accuracy: 100.00%, loss: 2.60390

Epochs 1500: ,accuracy: 100.00%, loss: 1.92075

Epochs 1600: ,accuracy: 100.00%, loss: 1.35274

Epochs 1700: ,accuracy: 100.00%, loss: 0.92080

Epochs 1800: ,accuracy: 100.00%, loss: 0.63586

Epochs 1900: ,accuracy: 100.00%, loss: 0.45814

Epochs 2000: ,accuracy: 100.00%, loss: 0.34532

Epochs 2100: ,accuracy: 100.00%, loss: 0.27056

Epochs 2200: ,accuracy: 100.00%, loss: 0.21876

Epochs 2300: ,accuracy: 100.00%, loss: 0.18142

Epochs 2400: ,accuracy: 100.00%, loss: 0.15357

Training finished
accuracy: 100.00%, loss: 0.13240
```

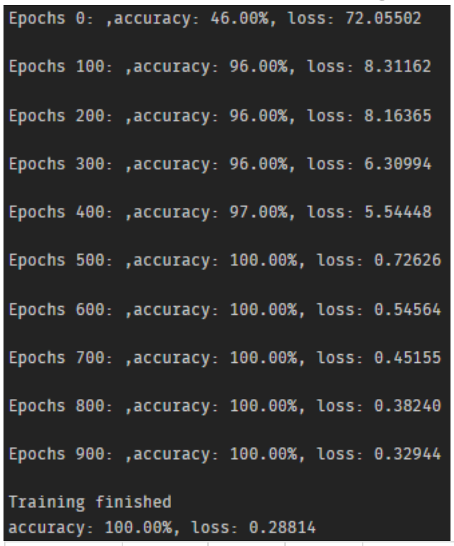
epochs	**lr**	n_1	n_2	**loss**	**accuracy**
2500	0.1	10	10	0.13240	100%

lr: initial learning rate

n_1: number of neurons in hidden layer 1

n_2: number of neurons in hidden layer 2

The data from linear generator



```
Epochs 0: ,accuracy: 46.00%, loss: 72.05502

Epochs 100: ,accuracy: 96.00%, loss: 8.31162

Epochs 200: ,accuracy: 96.00%, loss: 8.16365

Epochs 300: ,accuracy: 96.00%, loss: 6.30994

Epochs 400: ,accuracy: 97.00%, loss: 5.54448

Epochs 500: ,accuracy: 100.00%, loss: 0.72626

Epochs 600: ,accuracy: 100.00%, loss: 0.54564

Epochs 700: ,accuracy: 100.00%, loss: 0.45155

Epochs 800: ,accuracy: 100.00%, loss: 0.38240

Epochs 900: ,accuracy: 100.00%, loss: 0.32944

Training finished
accuracy: 100.00%, loss: 0.28814
```

epochs	lr	n_1	n_2	loss	accuracy
1000	0.1	10	10	0.28814	100%

lr: initial learning rate

n_1: number of neurons in hidden layer 1

n_2: number of neurons in hidden layer 2

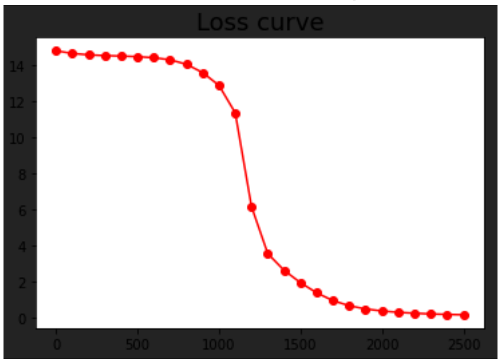
Our network reached 100% accuracy on both data!

C. Learning Curve

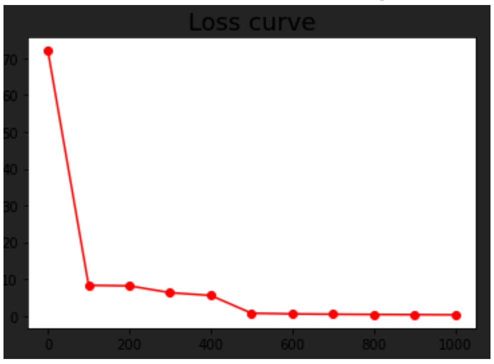
X-axis: *Epochs*

Y-axis: *Loss*

The data from XOR generator



The data from linear generator



D. Anything You Want to Discuss

> **XOR data** takes *longer* epochs to reach 100% accuracy than **linearly-separated data**.

I think there are 2 reasons causing this consequence.

1. The default generators generate different number of data. In **XOR** generator, the data size is fixed to **21**, while in **linear** generator, the data size is **100** and can even go higher.
   The greater amount of data might be one of the reason to faster convergence in **linear data**.
2. From the figures above, one can see that **linearly-separated data** can be easily separated by a straight line, while **XOR data** needs **more than 1 line** to separate 2 different groups.
   This might be another reason leading to longer training epochs.

From the learning curves show above, we can validate this phenomenon. Another thing I would like to mention, is that

> **Backpropagation** through ***computational graph*** might lead to *longer training epochs* as well as *more memory consuming*, if compared to ordinary backpropagation method.

This is because backpropagation through **computational graph** needs to construct several ***sparse matrixes***. If you refer to the backpropagation introduction section above, you might find out that these Jacobian matrixes are indeed sparse, with **most of the entries being zeros**. Despite the fact that backpropagation through computational graph is more time and memory consuming, we still choose the method instead of the ordinary one to implement since this method is way straight-forward to understand the propagation paths.

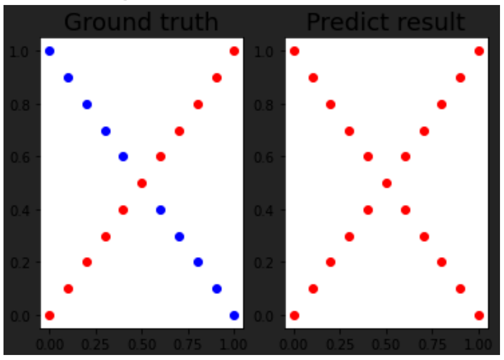
Discussion

A. Try Different Learning Rates

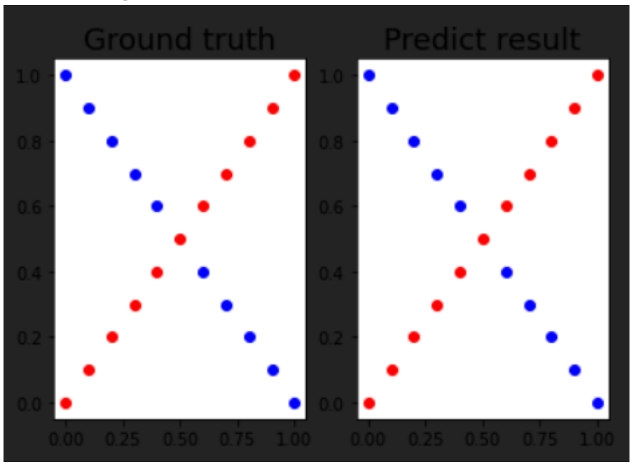
The data from XOR generator

1. Epochs: 2500
2. Number of neurons in hidden layer 1: 10
3. Number of neurons in hidden layer 2: 10

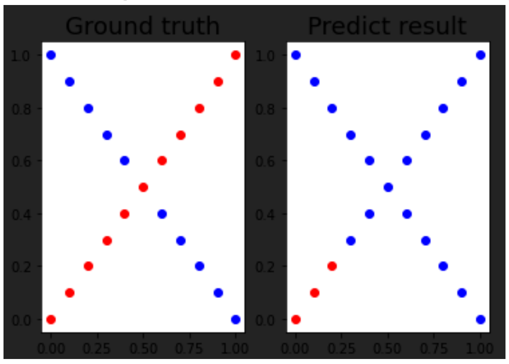
learning rate = **0.01**, final accuracy = **52.38%**



learning rate = **0.1 (default)**, final accuracy = **100%**

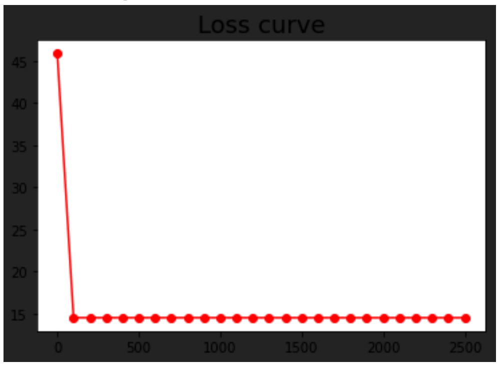


learning rate = **1**, final accuracy = **61.90%**

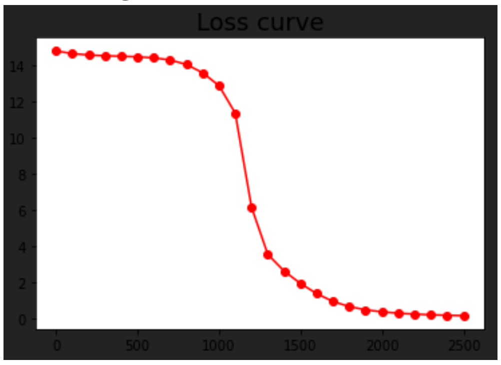


Learining Curves

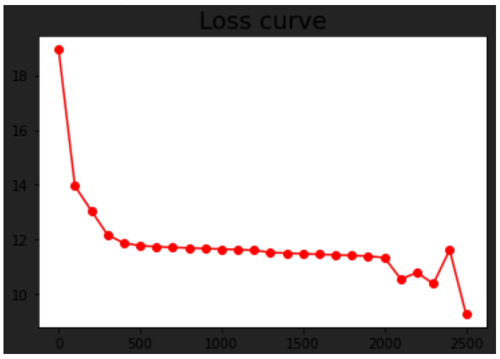
learning rate = **0.01**, final loss = **14.52753**



learning rate = **0.1 (default)**, final loss = **0.13240**



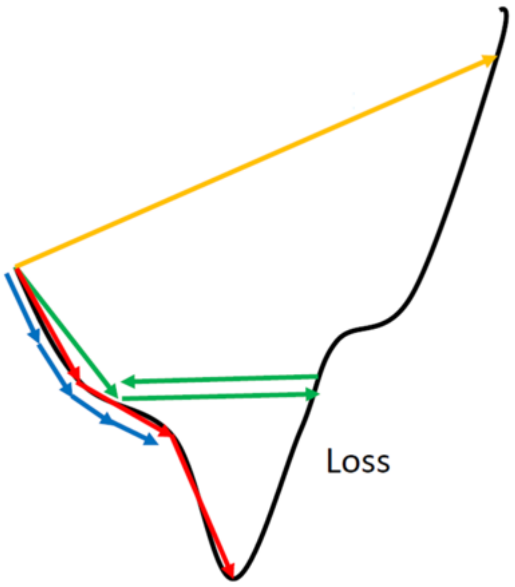
learning rate = **1**, final loss = **9.29162**



We can see that if the learning rate are set ***too small***, it would likely **fail to find local optimum**.

This is because that small learning rate means small updating stride. Even if we can still find the optimum theoritically, it will still take *very very long training epochs*.

On the other hand, if the learning rate are set ***too large***, it would likely **fail to converge**, since the updating stride is too wide to reach the local loss valley. One can see from the learning curve figure above. The losses at the last few epochs are *oscillating*.

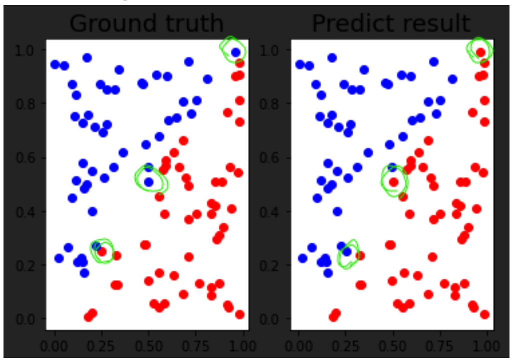


Small learning rate leads to the blue path, while large learning rate leads to the green path.

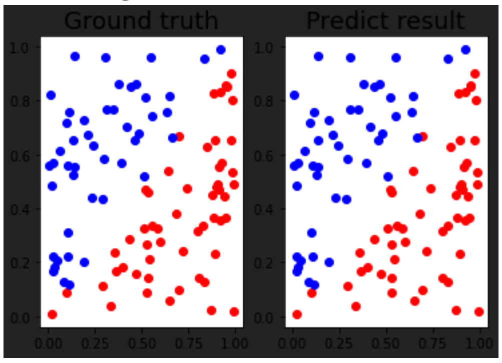
The data from linear generator

1. Epochs: 1000
2. Number of neurons in hidden layer 1: 10
3. Number of neurons in hidden layer 2: 10

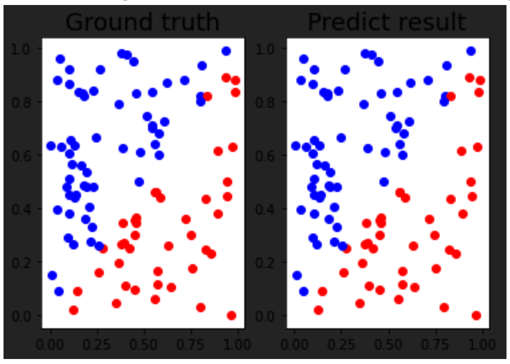
learning rate = **0.01**, final accuracy = **97%**



learning rate = **0.1 (default)**, final accuracy = **100%**

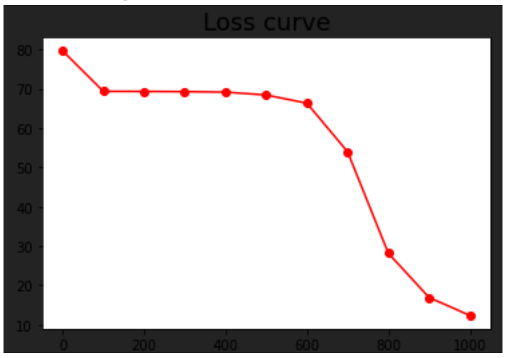


learning rate = **1**, final accuracy = **100%**

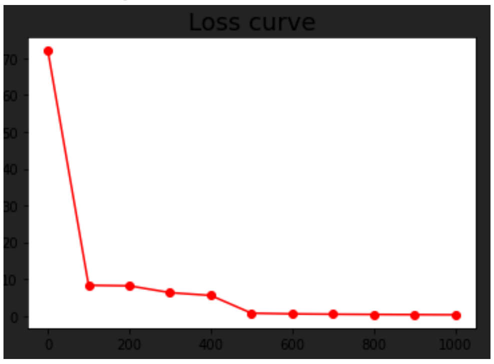


Learining Curves

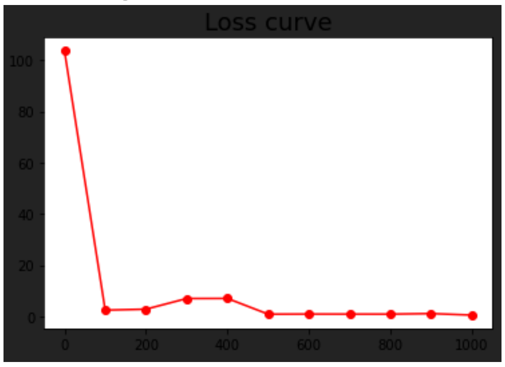
learning rate = **0.01**, final loss = **12.36974**



learning rate = **0.1 (default)**, final loss = **0.28814**



learning rate = **1**, final loss = **0.52719**



Since this generator generates more ($n = 100$) and easier(*can be separated by just a line*) data, The effect of learning rates is not very big.

Both default learning rate **0.1** and the larger learning rate **1** are able to reach **100%** accuracy. As for the smaller learning rate **0.01**, it only reaches **97%** accuracy, where **3**

points misclassified (the three data points within green circles in the above figure).
These 3 points are all border cases, and from the learning curve, one can see that the loss is in the middle of descent. So, I assume that maybe a little longer epochs can the network with the smaller learning rate reach 100% accuracy.

B. Try Different Number of Hidden Units

Let n_1 be the **number of neurons in hidden layer 1** and n_2 be **that of hidden layer 2**. The default setting in my implementation is

$$n_1 = n_2 = 10$$

In this part, I would like to see how the sizes of hidden units effect the performances. Thus, I will only consider more extreme cases. There are **4** cases I would like to try examining.

1. ***small*** to ***small***

$$n_1 = n_2 = 5$$

2. ***small*** to ***big***

$$n_1 = 5, n_2 = 100$$

3. ***big*** to ***small***

$$n_1 = 100, n_2 = 5$$

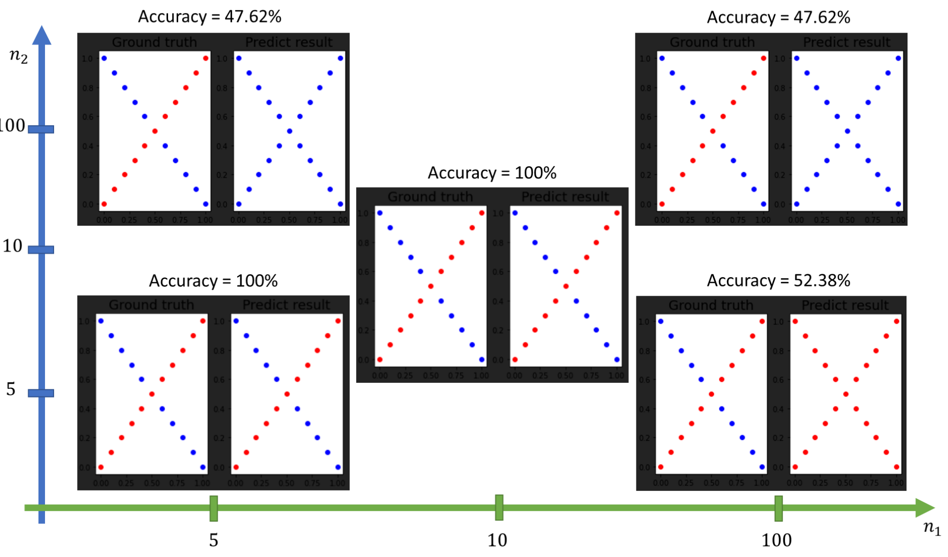
4. ***big*** to ***big***

$$n_1 = n_2 = 100$$

Other hyper-parameters are kept as default settings.

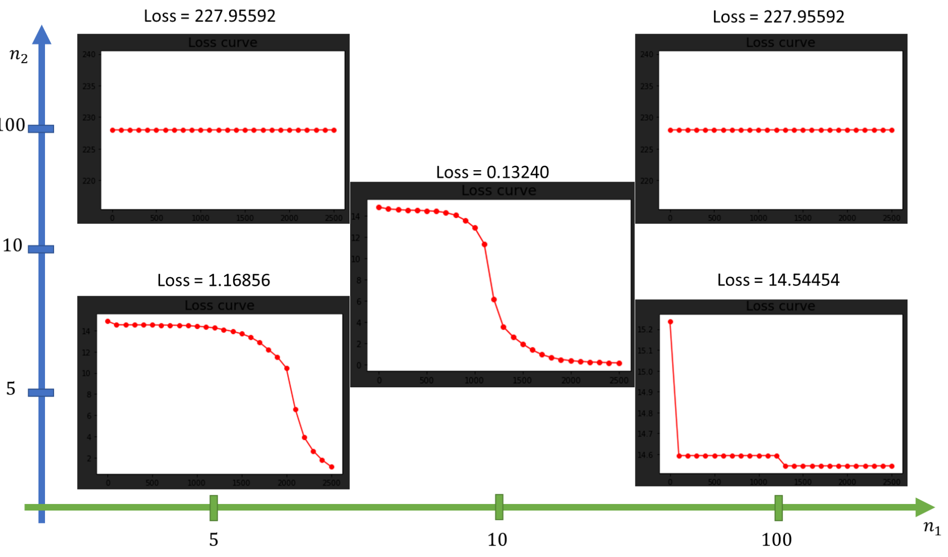
1. learning rate: 0.1
2. Epochs: 2500 for *XOR data*, 1000 for *linear data*

The data from XOR generator



Asides from $(n_1 = 5, n_2 = 5)$ and the default $(n_1 = 10, n_2 = 10)$, in the other 3 cases, the networks stupidly guessing all the points belonging to a single group. This might be due to the fact that these data are in a relatively easy pattern. Few neurons are sufficient to do the classification work. **Too many extra neurons will contribute many unwanted noises instead,** no matter in hidden layer 1 or hidden layer 2.

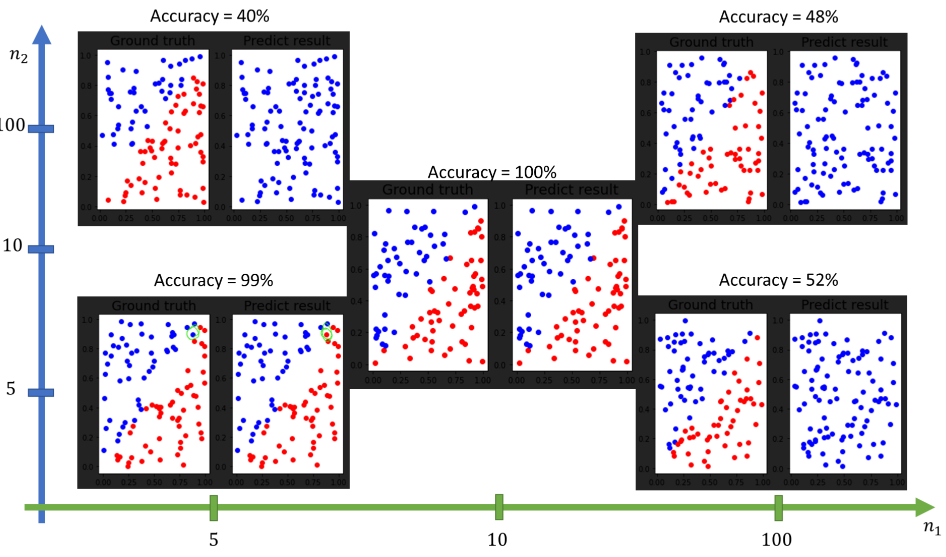
Learning curves



From the learning curves, we can see that **if neurons in hidden layer 2 are too many, the losses just don't decrease.** On the other hand, even if the curve looks unhealthy, when n_1 is big but n_2 is small, the losses are somehow decreasing. I think this might be owing to the case that the **output layer is more sensitive to the**

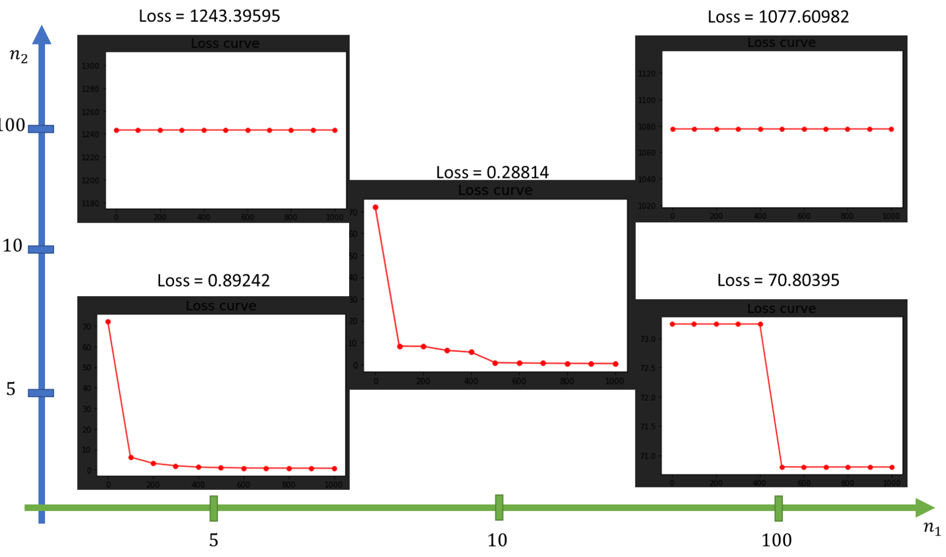
signals/noises from hidden layer 2 than from hidden layer 1, since hidden layer 2 is right before the output layer.

The data from linear generator



In the linear generator case, it's more obvious that **easier task does not need too many unnecessary hidden units**, no matter which hidde layer these extra units are added to. For the pairs $(n_1, n_2) = (5, 100), (100, 5), (100, 100)$, the networks just simply failed to classfy the data but assign them all in a single group.

Learning curves



Similar to the conclusion made in **XOR data**, the learning curves of **linear data** also presented the phenomenon that **the output performance is more sensitive to the**

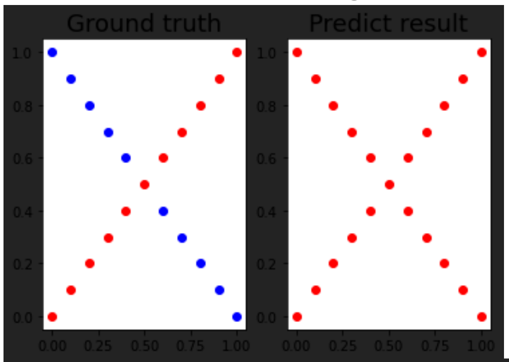
change made in hidden layer 2. Too many noises have cause the loss curves calm and smooth.

C. Try Without Activation Function

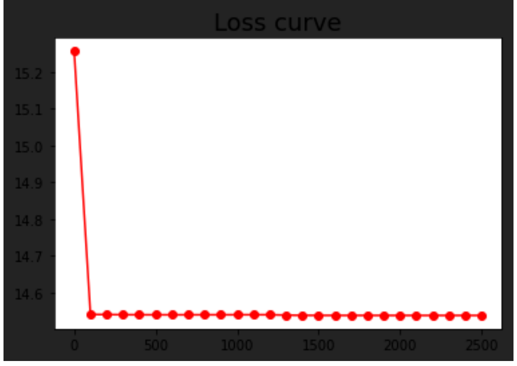
In this part, we would like to examine the effect if we plug out the activation function. Note that we only remove the activation function at the hidden layer 1 and hidden layer 2, but **not at the output layer**. Since the predict values need to be constrained within $(0, 1)$ to make binary classification, as well as loss and accuracy computations available.
All the other hyper-parameters are remained default

1. learning rate: 0.1
2. epochs: *2500* for *XOR data* and *1000* for *linear data*
3. number of neurons in hidden layer 1: 10
4. number of neurons in hidden layer 2: 10

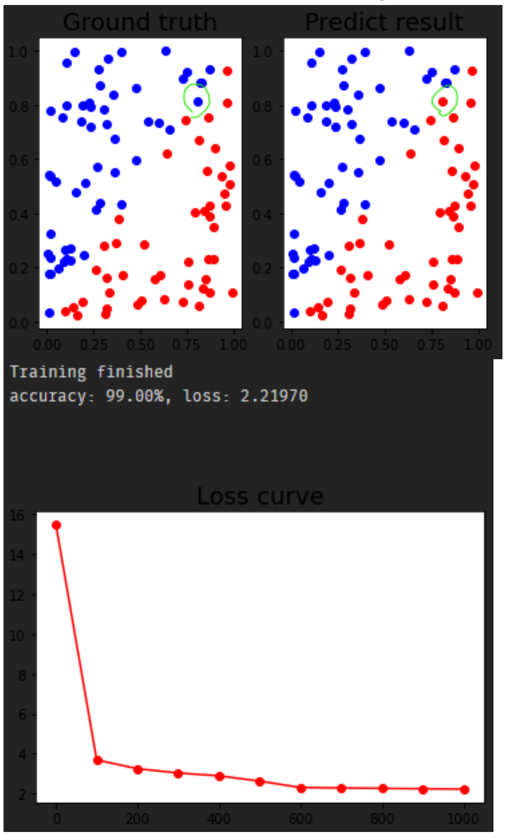
The data from XOR generator



```
Training finished
accuracy: 52.38%, loss: 14.53728
```



The data from linear generator



```
Training finished
accuracy: 99.00%, loss: 2.21970
```

No matter which patterns of data, the losses *plummet* real quick. For the **linearly-separated** data, since the pattern is quite easy, the classification ability is still pretty good (99% accuracy) even without the use of activation function.
In contrast, the network without activation functions failed to classified **XOR** data. I reckon that this might be on account of the **linearity property**. If we remove the *non-linear* activation function, the output signals from all the layers are just merely a **linear combination** of column vectors of the weight matrixes. **They cannot generate any signals that are out of the span of these vectors.** The linear property kinds of works as constraints here. If the data are ***not* linearly-separable**, such as **XOR** data in this case, it just won't work out without the non-linear transformation given by activation functions.

D. Anything You Want to Share

For these kind of easy data patterns, it is recommended that one use only few neurons to train. **5~10** units in each hidden layer is advised. The reasons are:

1. Redundant hidden units might generate noises, which severly interferes the output neuron making good predictions. I have described this point in ***B. Try Different Number of Hidden Units***.
2. Also, from the experiments, one can obviously observe that it takes much *longer time* in each epoch as the increase of hidden neurons. Since the weighted matrixes grow exponentially larger, it is normal to take more computation time and resources.
   Thus, finding a suitable number of neurons is indeed a big issue in deep learning tasks.

Extra

A. Different Optimizer

Reference material:
https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc
(https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc)
Original paper of **Adam**:
https://arxiv.org/abs/1412.6980 (https://arxiv.org/abs/1412.6980)
There are **2** hyper-parameters β_1 and β_2 in Adam optimizer. β_1 is the exponential decay rate for the **first moment** (i.e. *mean*) estimates for the ***gradient required***. β_2 is the exponential decay rate for the **second moment** (i.e. *uncentered variance*) estimates for the ***gradient required***. The literature values suggested by the original authors which work well with most datasets are

$$\beta_1 = 0.9,\ \beta_2 = 0.999$$

On a given iteration t, one can calculate the **moving averages** of ***mean*** and ***variance*** of the gradients, namely m_t and v_t, based on the current gradient g_t and the hyper-parameters β_1 and β_2 as follows:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(g_t)^2$$

β_1 is actually corresponding to the *momentum* part, and β_2 is corresponding to the *RMSProp* optimizer part.
The next step is called the ***bias correction*** step. Since most algorithms that depend on *moving averages* such as SGD and RMSProp are biased, we need this extra step to correct the bias.

$$m_t^{corr} = \frac{m_t}{1-(\beta_1)^t}$$
$$v_t^{corr} = \frac{v_t}{1-(\beta_2)^t}$$

Finally, we can update our weight w_t as follows:

$$w_t = w_{t-1} - \eta(\frac{m_t^{corr}}{\sqrt{v_t^{corr}} + \epsilon})$$

η is the learning rate, and $\epsilon = 10^{-9}$ is some arbitrary small value to prevent zero-division. $\eta = 0.1$ for the first half of epochs while $\eta = 0.05$ for the later half of epochs.

```
class Adam_optimizer():
    def __init__(self, beta1 = 0.9, beta2 = 0.999, epsilon = 1e-9):
        self.grad_mean = 0
        self.grad_var = 0
        '''momentum'''
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
    def step(self, t, w, dw, lr):
        # momentum beta 1
        self.grad_mean = self.beta1 * self.grad_mean + (1 - self.beta1) * dw
        # RMS beta 2
        self.grad_var = self.beta2 * self.grad_var + (1 - self.beta2) * (dw ** 2)
        # bias correction, for moving average based method (RMS)
        grad_mean_corr = self.grad_mean / (1 - self.beta1 ** t)
        grad_var_corr = self.grad_var / (1 - self.beta2 ** t)

        # update
        w -= lr * (grad_mean_corr / (np.sqrt(grad_var_corr) + self.epsilon))
        return w
```

In the `backward()` method of the neural network class `NN_optimizer`:

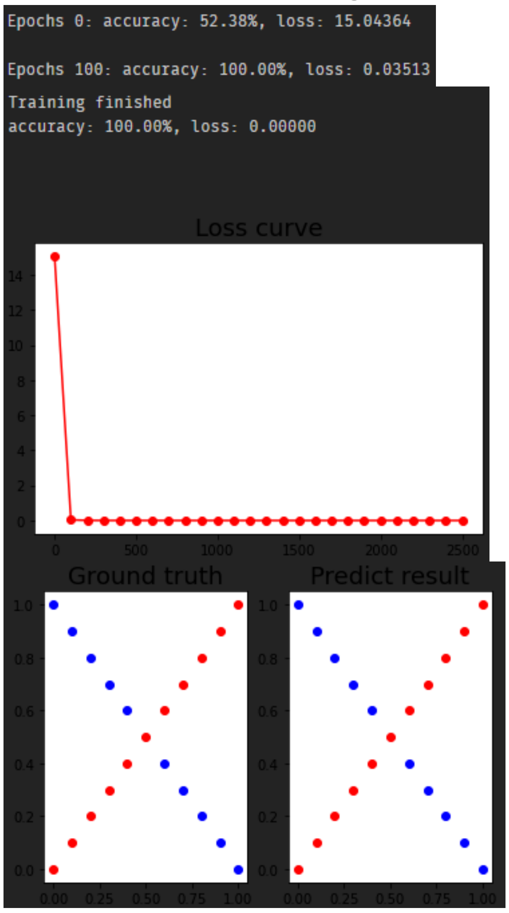
```
if self.optimizers:
    self.W1 = self.optimizers[0].step(t, self.W1, Cost_to_W1, self.lr)
    self.W2 = self.optimizers[1].step(t, self.W2, Cost_to_W2, self.lr)
    self.W3 = self.optimizers[2].step(t, self.W3, Cost_to_W3, self.lr)
```

Use 3 different Adam optimizers for each of the weight matrixes W_1, W_2, W_3.

The other parameters are kept default:

1. learning rate: **0.1** in the first half and **0.05** in the later half
2. the number of hidden units of hidden layer 1 and 2: both **10**
3. epochs: **2500** for **XOR data** and **1000** for **linearly separated data**

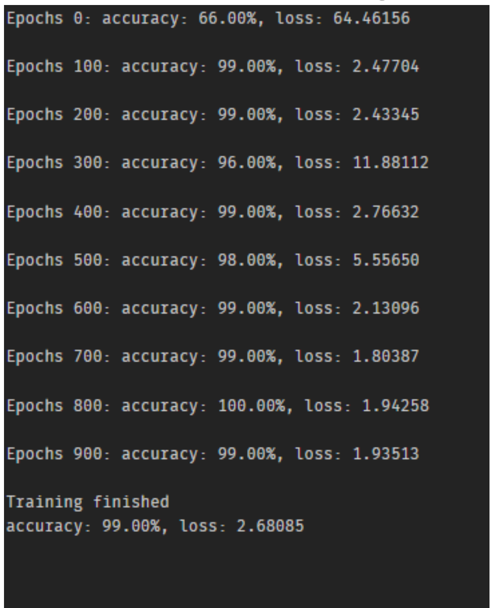
The data from XOR generator



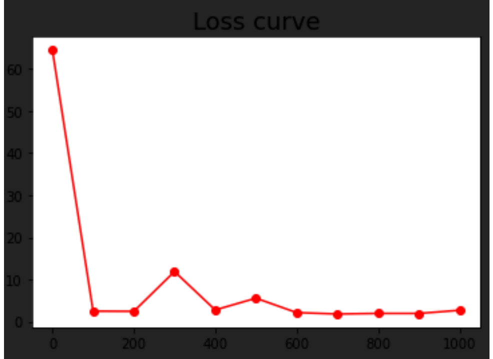
```
Epochs 0: accuracy: 52.38%, loss: 15.04364

Epochs 100: accuracy: 100.00%, loss: 0.03513
Training finished
accuracy: 100.00%, loss: 0.00000
```

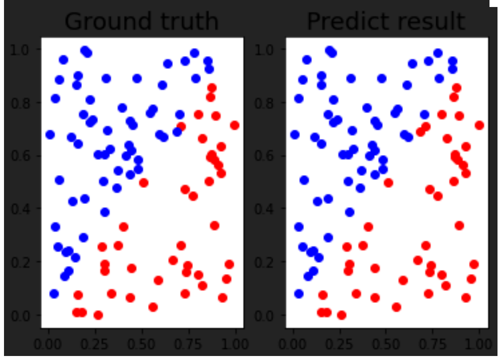
The predictive performance is still perfect with **100%** accuracy. However, it only takes **at most 100** epochs to reach 100% accuracy! Adding **Adam** optimizer does help accelerate the training processes. Also, you can see that after 2500 epochs, **the loss drops to 0**!

The data from linear generator



```
Epochs 0: accuracy: 66.00%, loss: 64.46156

Epochs 100: accuracy: 99.00%, loss: 2.47704

Epochs 200: accuracy: 99.00%, loss: 2.43345

Epochs 300: accuracy: 96.00%, loss: 11.88112

Epochs 400: accuracy: 99.00%, loss: 2.76632

Epochs 500: accuracy: 98.00%, loss: 5.55650

Epochs 600: accuracy: 99.00%, loss: 2.13096

Epochs 700: accuracy: 99.00%, loss: 1.80387

Epochs 800: accuracy: 100.00%, loss: 1.94258

Epochs 900: accuracy: 99.00%, loss: 1.93513

Training finished
accuracy: 99.00%, loss: 2.68085
```

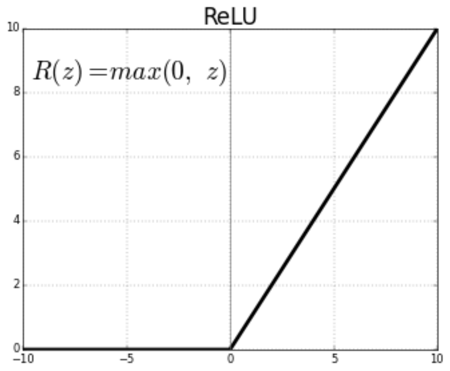




After applying the **Adam** optimizers, it still reaches **99%** accuracy. As one can see, the network actually reached **100%** once. The misclassified one data point is in fact on the separation line. One interesting phenomenon is that the loss curve has a little spike at around epoch 300, where the accuracy also drops to 96%. Nevertheless, Adam optimizer helps the network updates its weights back to a normal spot within the next 100 epochs.

B. Different Activation Function

I also try the ***Rectified Linear Unit (ReLU)*** activation function. The function looks like so



If the input z is **greater than zero**, the output is also z itself. Otherwise, the output will be **0**.

```
def ReLU(z):
    return np.maximum(z, 0)
```

The **derivative** of ReLU function can also be separated into 2 cases.

1. If the input $z > 0$, then the derivative is **1**.

$$R(z) = z, \ \therefore \ R'(z) = 1$$

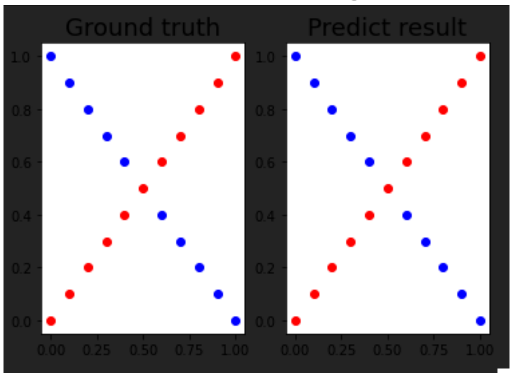
2. If the input $z \leq 0$, then the derivative is **0**.

$$R(z) = 0, \ \therefore \ R'(z) = 0$$

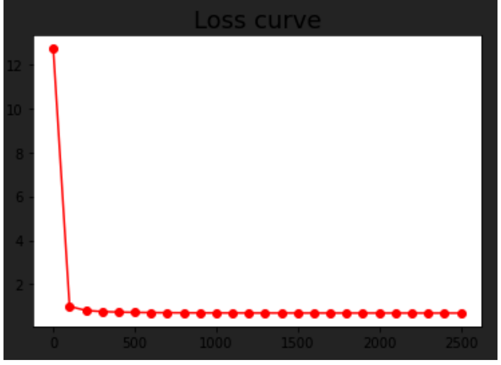
```
def derivative_ReLU(z):
    return (z > 0.0) * 1.0
```

The network structure and all the hyper-parameters settings are left the same.

The data from XOR generator



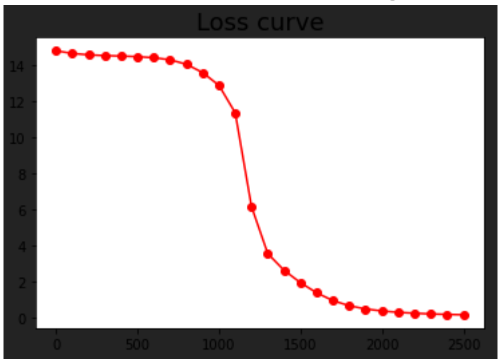
```
Training finished
accuracy: 100.00%, loss: 0.69876
```



The network reaches **100%** accuracy. Moreover, it actually did this **in at most 100 epochs only**!

```
Epochs 0: accuracy: 76.19%, loss: 12.70475

Epochs 100: accuracy: 100.00%, loss: 1.00398
```

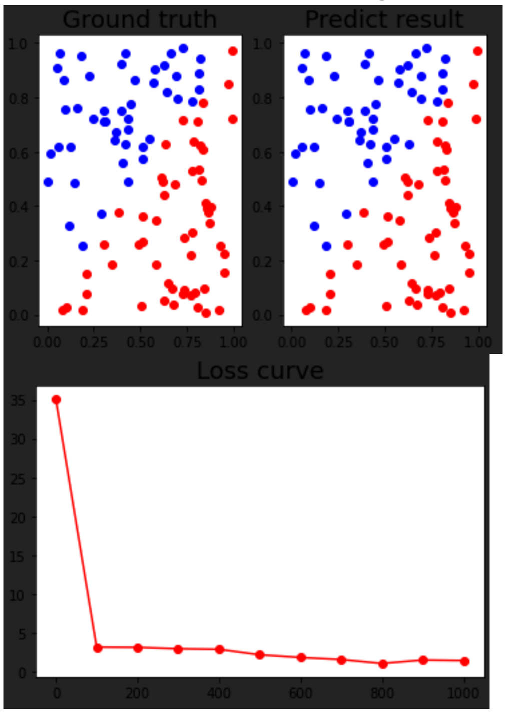
Compare to the learning curve of original **sigmoid** activation function, which clearly takes longer epochs to achieve low loss and high accuracy.



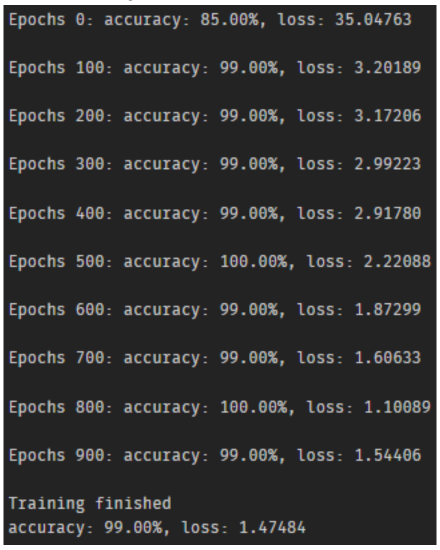
ReLU function is generally **more efficient** than sigmoid function, since the calculation operation is fairly simple and easy (*if negative, just set the output to 0. if positive, just set the output as the input*).

Another advantage of ReLU over sigmoid function is that in practice, networks with Relu tend to show better convergence performance. The statement was provided by ***Krizhevsky et al*** **(http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf)**.

The data from linear generator



As for the linearly separated data, the results and conclusion are the same as mentioned above. The accuracy is **99%**.



```
Epochs 0: accuracy: 85.00%, loss: 35.04763

Epochs 100: accuracy: 99.00%, loss: 3.20189

Epochs 200: accuracy: 99.00%, loss: 3.17206

Epochs 300: accuracy: 99.00%, loss: 2.99223

Epochs 400: accuracy: 99.00%, loss: 2.91780

Epochs 500: accuracy: 100.00%, loss: 2.22088

Epochs 600: accuracy: 99.00%, loss: 1.87299

Epochs 700: accuracy: 99.00%, loss: 1.60633

Epochs 800: accuracy: 100.00%, loss: 1.10089

Epochs 900: accuracy: 99.00%, loss: 1.54406

Training finished
accuracy: 99.00%, loss: 1.47484
```

From the training logs, one can see that at some points, the network does reach **100%**. The only 1 error is a point that is ***really*** close the separation line.