ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN CUỐI KỲ
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

ĐỀ TÀI:

NEURAL NETWORK 3 LỚP

Giảng viên lý thuyết: Nguyễn Thanh Sơn
Giảng viên hướng dẫn thực hành: Nguyễn Đức Vũ
Người thực hiện: Tô Anh Phát – 21520085 – KHTN2021

--TP. Hồ Chí Minh, 1 tháng 6 năm 2022--

Mục lục

I. Tổng quan đề tài

1. Giới thiệu đề tài

Mạng nơ-ron nhân tạo (*Neural Network*) là một mô hình lập trình tuyệt vời lấy cảm hứng từ mạng neural thần kinh. Cùng với với các kĩ thuật học sâu (Deep Learning), neural network đang trở thành một công cụ rất mạnh mẽ mang lại hiệu quả tốt nhất cho nhiều bài toán khó như nhận dạng ảnh, giọng nói hay xử lý ngôn ngữ tự nhiên.

2. Đối tượng và phạm vi đề tài

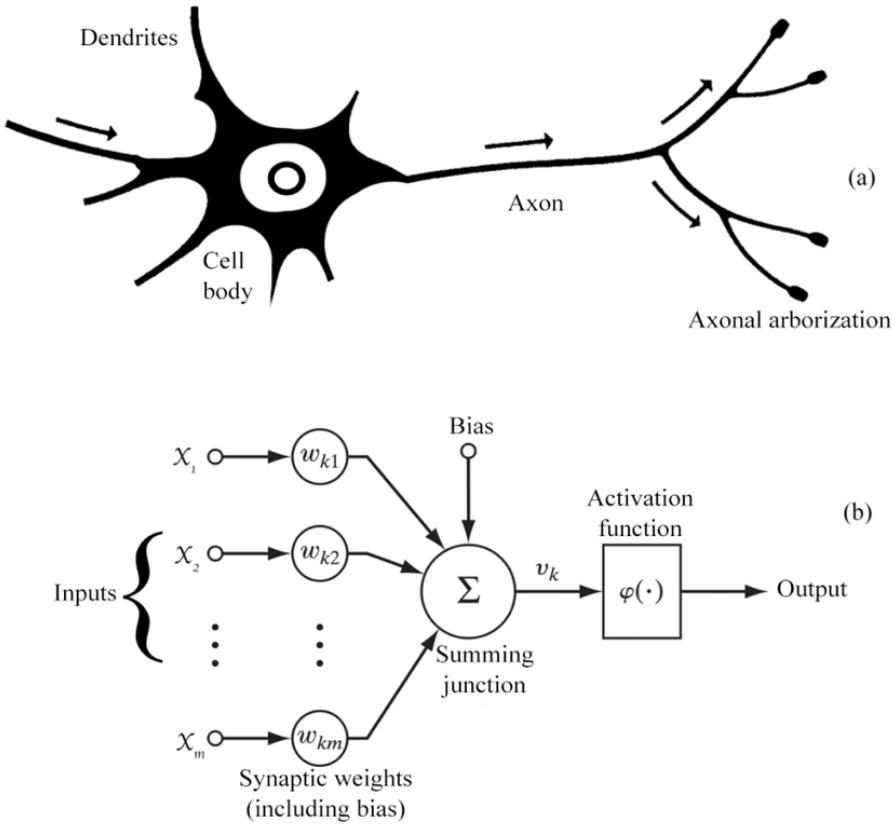
Trong phạm vi đề tài này, để tìm hiểu về cách mạng neural hoạt động, em sẽ xây dựng mạng neural 3 lớp đơn giản để nhận diện chữ số viết tay trên tập dữ liệu Mnist.

II. Nội dung đề tài

1. Cơ sở lý thuyết

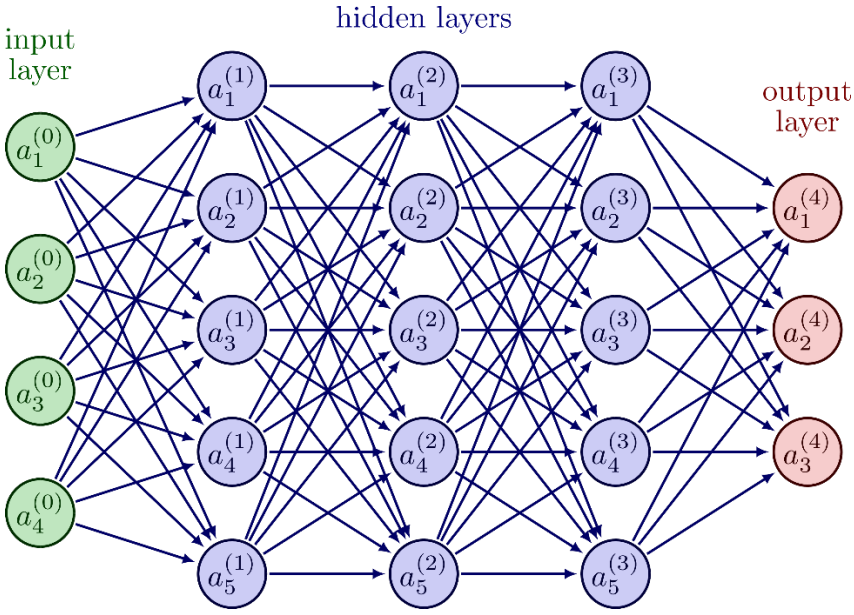
a. Mạng neural là gì?

Mạng neural nhân tạo được truyền cảm hứng bởi mạng thần kinh trong bộ não con người. Cả hai đều có những tế bào (nút) liên kết với nhau và có thể kích hoạt được. Tuy nhiên quá trình xử lý cơ bản là khác nhau.



Hình 1: Mạng neural nhân tạo và mạng neural sinh học

Một tế bào đơn lẻ về cơ bản là vô dụng, nhưng khi kết hợp với hàng trăm hoặc hàng ngàn tế bào khác thì lại vượt trội hơn bất cứ phương pháp máy học nào.



Hình 2: Liên kết giữa các tế bào

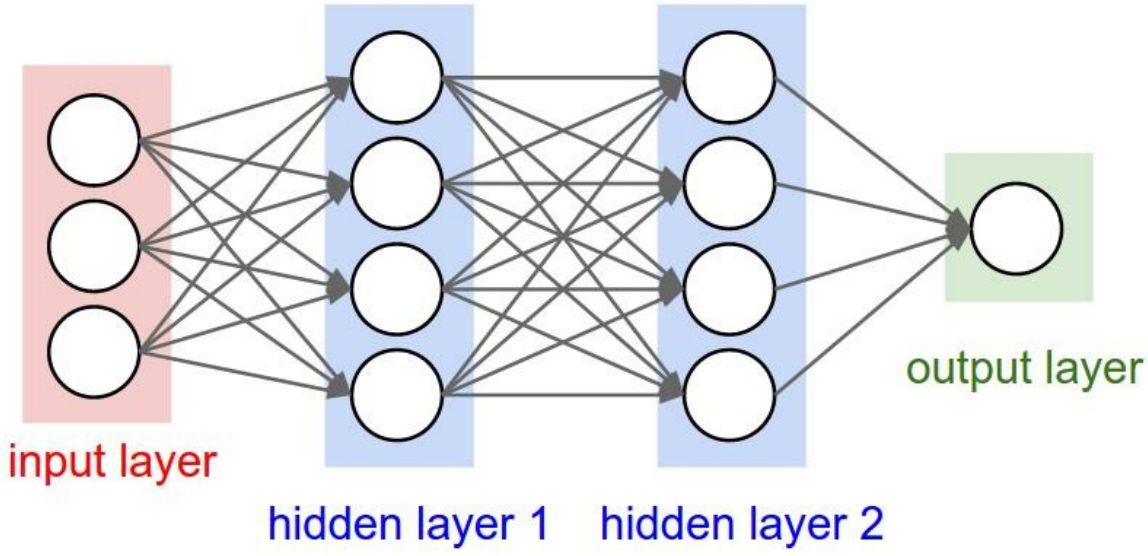
b. Layer

Sự liên kết giữa các nút trong *Hình* 2 là liên kết theo lớp (*layer*).

Một mạng neural nhân tạo được tổ chức theo lớp, với mỗi lớp gồm các nút xếp chồng lên nhau. Có 3 loại lớp là:

- Lớp vào (*input layer*): Lớp này nằm bên trái cùng của mạng, thể hiện cho các đầu vào của mạng.
- Lớp ẩn (*hidden layer*): Lớp này nằm giữa lớp vào và lớp ra nó thể hiện cho quá trình suy luận của mạng.
- Lớp ra (*output layer*): Là lớp bên phải cùng và nó thể hiện cho những đầu ra của mạng.

Mỗi mô hình luôn có 1 input layer, 1 output layer, có thể có hoặc không các hidden layer.

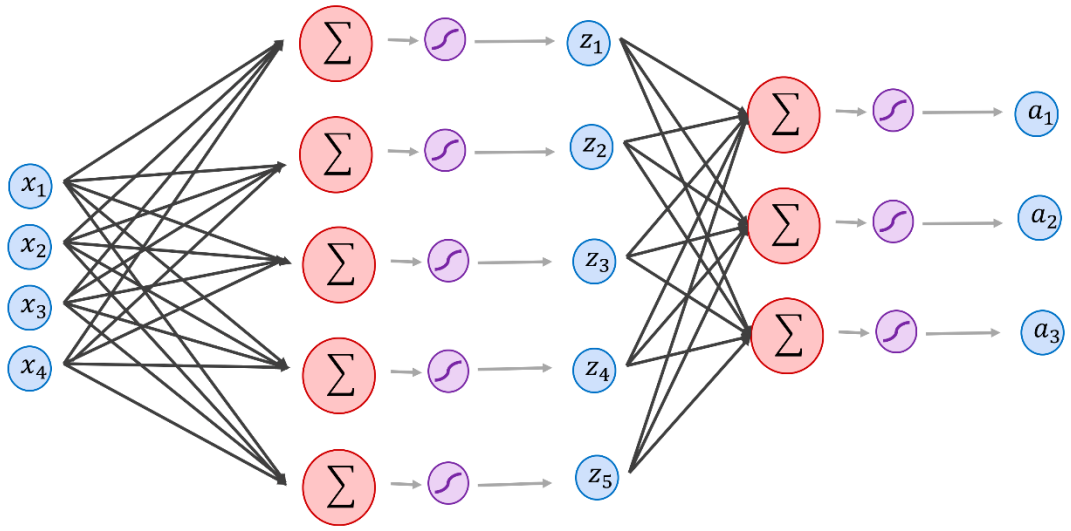


Hình 3: Các lớp trong mạng neural

c. Node

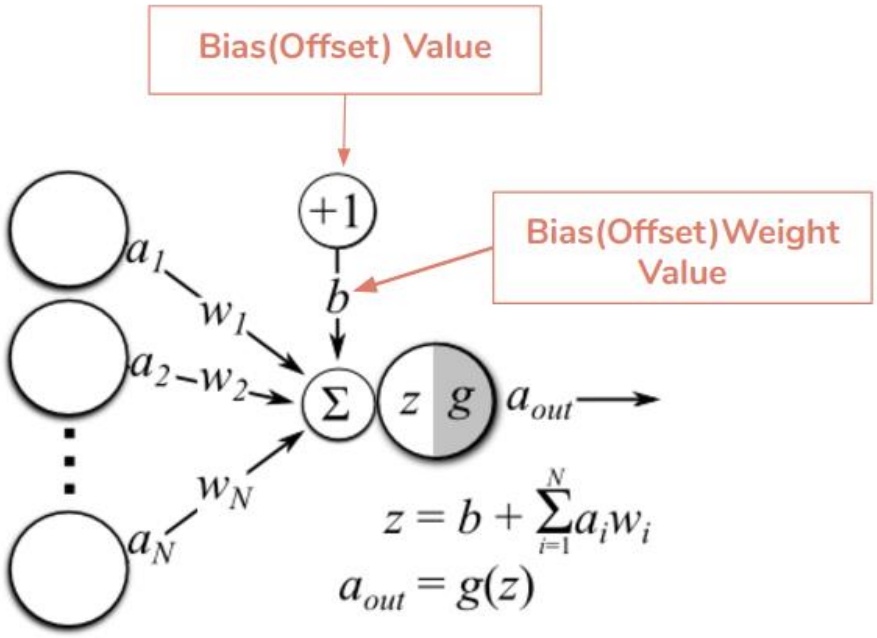
Mỗi hình tròn trong mỗi layer ở *Hình* 3 được gọi là một nút (*node*).

Với mỗi node trong hidden layer và output layer sẽ liên kết với tất cả các node ở layer trước đó. Node ở layer phía sau nhận thông tin từ các node ở layer phía trước qua tính toán ma trận và kích hoạt qua một hàm toán học.



Hình 4: Truyền thông tin giữa layer phía trước và layer phía sau

d. Weights và bias

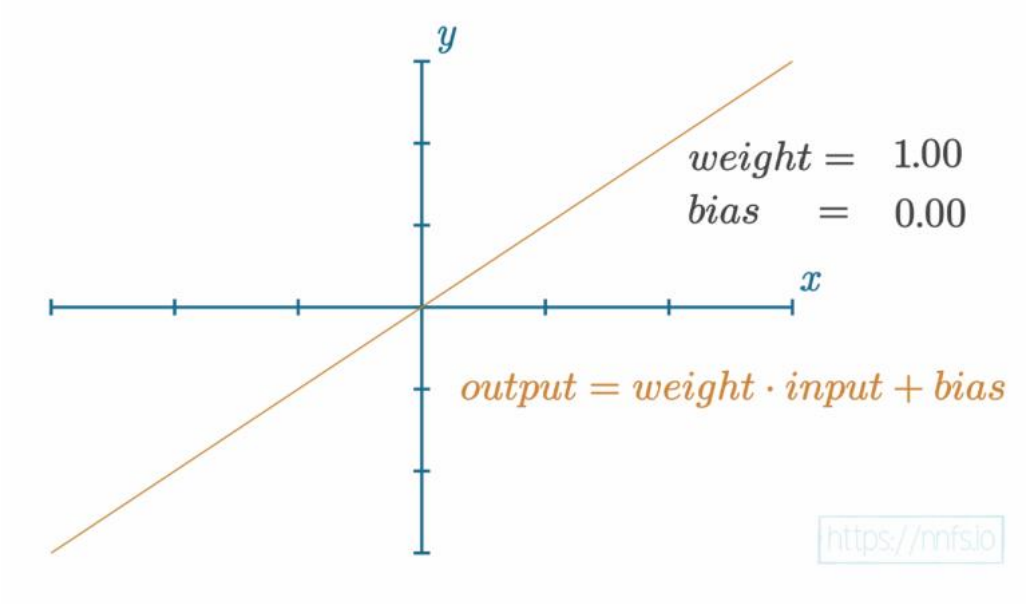


Hình 5: Weights và bias trong neural network

Trên mỗi liên kết giữa các nút có một *weight* gắn liền với nó. *Weight* là một hệ số có thể huấn luyện được và nó được nhân với với dữ liệu đầu vào. Một

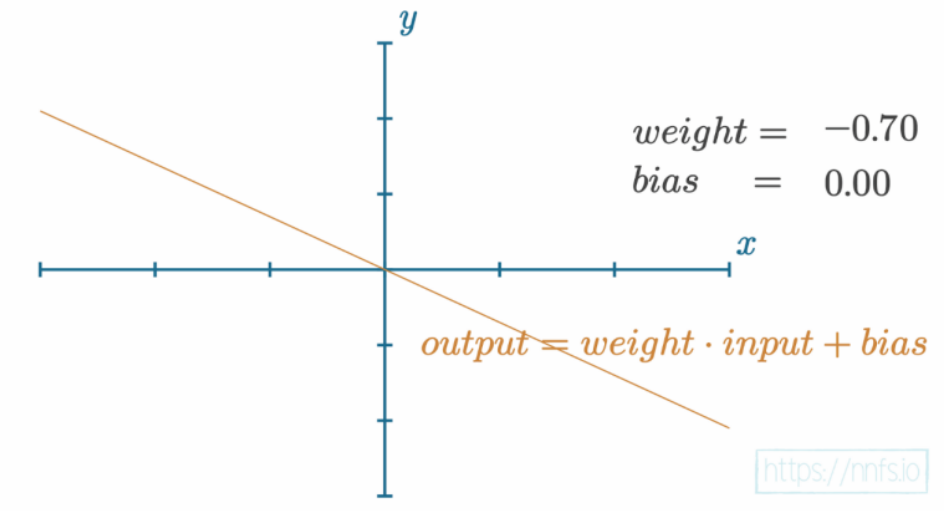
khi tất cả *inputs* · *weights* được truyền vào nút, nó sẽ được lấy tổng và cộng thêm *bias*, một hệ số có thể được huấn luyện khác.

Mục đích của *weights* và *bias* có thể được xem như là những phần tử giúp ta có thể điều chỉnh mô hình phù hợp với dữ liệu. Trong một mạng neural, ta thường có hàng ngàn hoặc hàng triệu những tham số có thể huấn luyện được như thế này. Cả *Bias* và *weights* đều là những tham số có thể điều chỉnh và nó sẽ tác động đầu ra của mạng, nhưng theo cách khác nhau. Khi *weights* được nhân, thì nó sẽ chỉ thay đổi độ lớn và dấu từ âm sang dương hoặc ngược lại. *Output* = *weight* · *input* + *bias* tương đồng với phương trình đường thẳng $y = mx + b$.



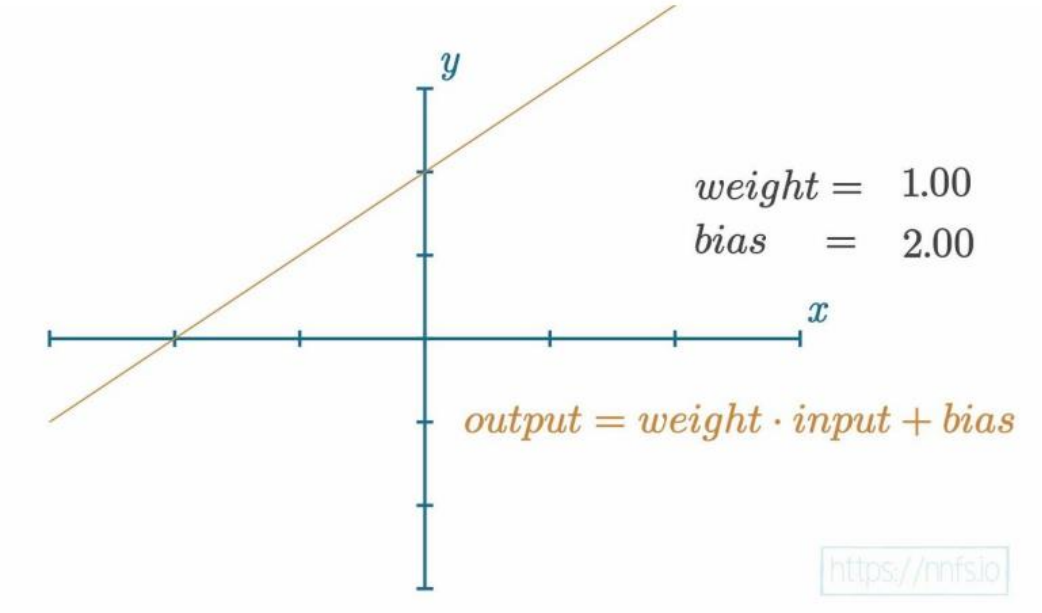
Hình 6: Kết quả của nút một đầu vào với weight = 1, bias = 0 và đầu vào x

Khi ta giảm *weight*, độ dốc sẽ giảm. Nếu *weight* âm thì độ dốc sẽ âm.



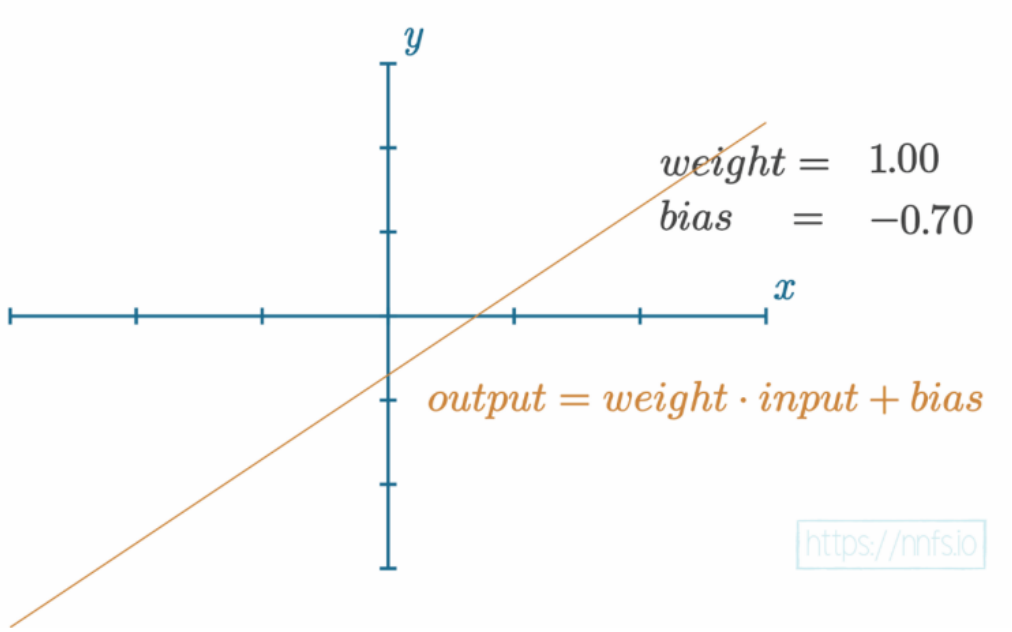
Hình 7: Kết quả của nút một đầu vào với weight = -0.7, bias = 0 và đầu vào x

Với *bias* khác 0:



Hình 8: Kết quả của nút một đầu vào với weight = 1, bias = 2 và đầu vào x

Khi ta giảm *bias*, phương trình sẽ di chuyển xuống.



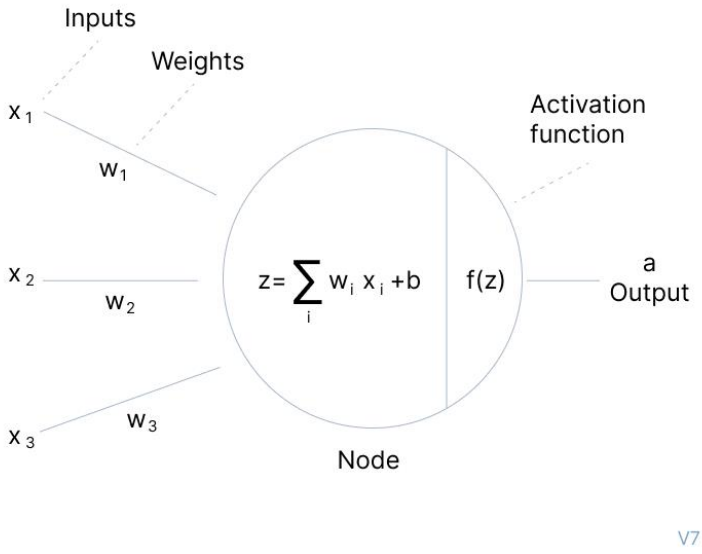
Hình 9: Kết quả của nút một đầu vào với weight = 1, bias = -0.7 và đầu vào x

e. Activation function

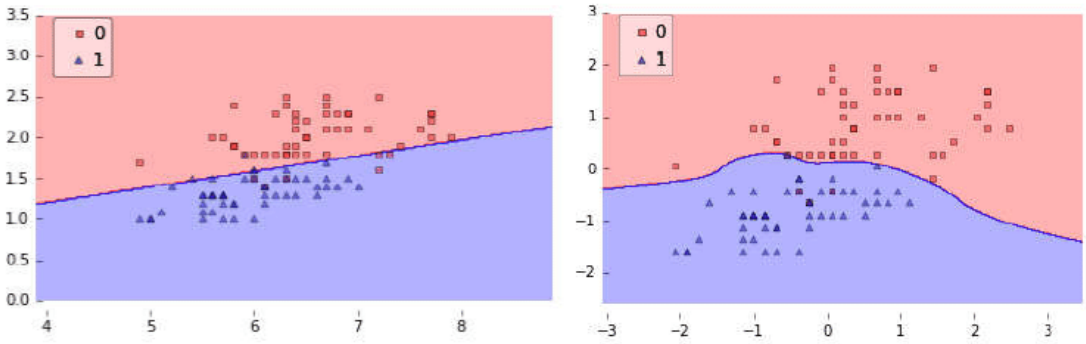
Ta tiếp tục lấy *output* ở bước *output* = *weight* · *input* + *bias* và cho qua hàm kích hoạt (*activation*):

$$Output = Activation(Output)$$

Mục đích của việc cho dữ liệu đi qua các hàm kích hoạt nhằm cho mô hình có được tính phi tuyến. Nếu các hàm kích hoạt là tuyến tính, thì ta có thể dễ dàng rút gọn lại chỉ còn một hàm tuyến tính. Do đó, nó sẽ gặp khó khăn trong việc giải quyết các bài toán phức tạp, phi tuyến tính.



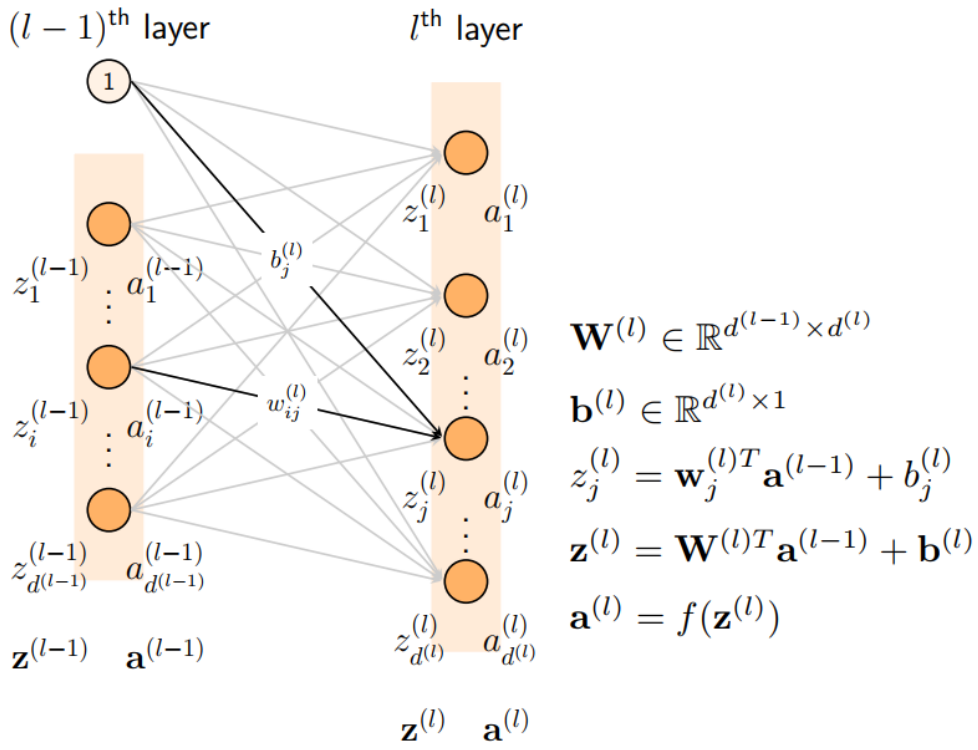
Hình 10: Hàm kích hoạt sau mỗi nút



Hình 11: Trước và sau khi áp dụng hàm activation

f. Forward propagation (Feed forward)

Forward propagation là thuật toán lan truyền xuôi, thực hiện truyền dữ liệu theo thứ tự từ lớp đầu vào đến lớp đầu ra.



Hình 12: Minh họa thuật toán forward propagation

Trong đó:

- Z là các giá trị của node khi chưa qua hàm activation.
- A là các giá trị của node sau khi qua hàm activation.
- f là hàm activation.
- W, B là ma trận *weight* và *bias* giữa 2 hai lớp.

g. Backpropagation

Sau bước forward propagation, ta có được kết quả dự đoán là đầu ra của lớp cuối cùng ($\hat{y} = a^L$). Để đánh giá được mức độ sai lệch (*loss*) của kết quả dự đoán với dữ liệu thực, ta có hàm mất mát. Ví dụ ta có hàm mất mát Mean Square Error (MSE):

$$J(\mathbf{W}, \mathbf{b}, \mathbf{X}, \mathbf{Y}) = \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{y}_n - \hat{\mathbf{y}}_n\|_2^2 = \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{y}_n - \mathbf{a}_n^{(L)}\|_2^2$$

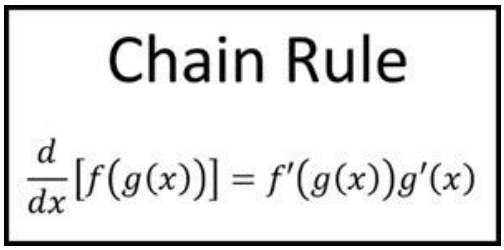
Hình 13: Hàm mất mát MSE

Trong đó:

- W, *b* là tập hợp tất cả ma trận trọng số giữa các layer và vector *bias* giữa các layer.
- X, Y là cặp dữ liệu huấn luyện với mỗi cột tương ứng với một điểm dữ liệu.

Do đó mục tiêu của ta bây giờ là phải tối thiểu hàm mất mát. Ta cần tính được $\nabla_{W^{(l)}}J, \nabla_{B^{(l)}}J$ với $\forall l = 1, 2, \dots, L$. Tuy nhiên, việc tính toán trực tiếp các giá trị đạo hàm là cực kỳ phức tạp vì hàm mất mát không phụ thuộc trực tiếp vào các ma trận hệ số và vector *bias*.

Phương pháp phổ biến nhất được dùng có tên là *Backpropagation* giúp tính đạo hàm ngược từ layer cuối cùng đến layer đầu tiên. Layer cuối cùng được tính toán trước vì nó gần gũi hơn với đầu ra dự đoán và hàm mất mát. Việc tính toán đạo hàm của các ma trận hệ số trong các layer trước được thực hiện dựa trên một quy tắc quan thuộc cho đạo hàm của hàm hợp.



Hình 14: Đạo hàm của hàm hợp

Đạo hàm của hàm mất mát theo chỉ một thành phần của ma trận trọng số của output layer:

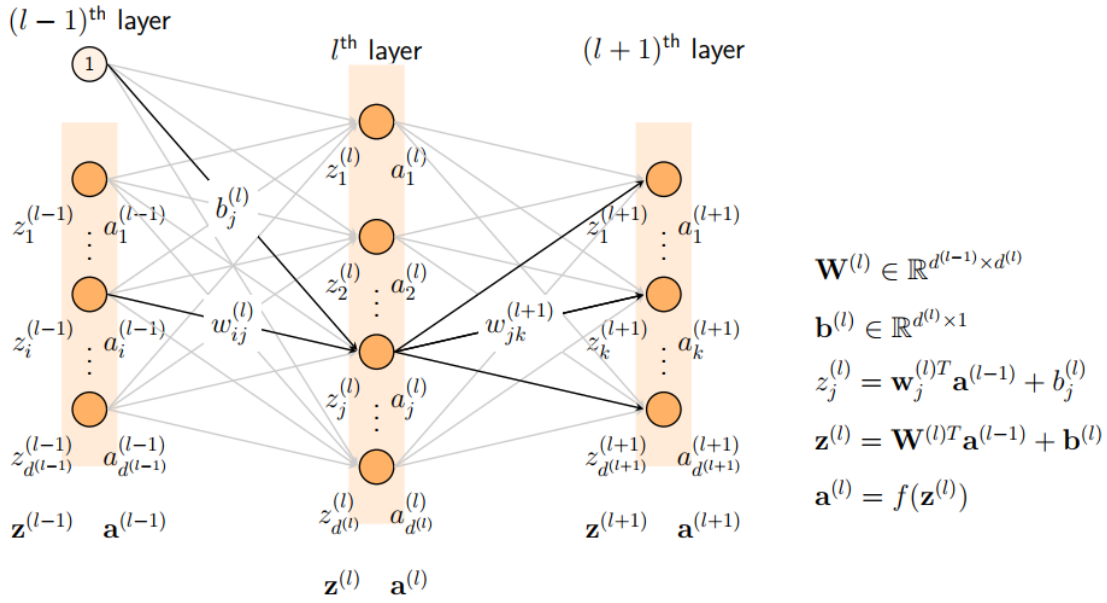
$$\frac{\partial J}{\partial w_{ij}^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} \cdot \frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = e_j^{(L)} a_i^{(L-1)}$$

Trong đó:

- $e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$ thường là một đại lượng *không quá khó để tính toán.*
- $\frac{\partial z_j^{(L)}}{\partial w_{ij}^{(L)}} = a_i^{(L-1)}$ vì $z_j^{(L)} = w_j^{(L)T} a^{(L-1)} + b_j^{(L)}$.

Tương tự, đạo hàm của hàm mất mát theo *bias* của layer cuối cùng là:

$$\frac{\partial J}{\partial b_j^{(L)}} = \frac{\partial J}{\partial z_j^{(L)}} . \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} = e_j^{(L)}$$



Hình 15: Quá trình trình truyền dữ liệu

Dựa vào *Hình 15*, ta có thể tính được:

$$\frac{\partial J}{\partial w_{ij}^{(l)}} = \frac{\partial J}{\partial z_j^{(l)}} . \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} = e_j^{(l)} a_i^{(l-1)}$$

với

$$e_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial a_j^{(l)}} . \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$$
$$= \left(\sum_{k=1}^{d^{(l+1)}} \frac{\partial J}{\partial z_k^{(l+1)}} . \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \right) f^{(l)'}(z_j^{(l)}) = \left(\sum_{k=1}^{d^{(l+1)}} e_k^{(l+1)} w_{jk}^{(l+1)} \right) f^{(l)'}(z_j^{(l)})$$

Trong đó:

- $e^{(l+1)} = \left[e_1^{(l+1)}, e_2^{(l+1)}, \dots, e_{d^{(l+1)}}^{(l+1)} \right]^T \in R^{d^{(l+1)} * 1}$.
- $w_{j:}^{(l+1)}$: được hiểu là *hàng* thứ j của ma trận $W^{(l+1)}$.
- Do $a_j^{(l)} = f^{(l)}(z_j^{(l)})$ nên $\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = f^{(l)'}(z_j^{(l)})$.

Tới đây, ta có thể thấy rằng việc activation function có đạo hàm đơn giản sẽ có ích rất nhiều trong việc tính toán.

Với cách làm tương tự, ta có thể suy ra:

$$\frac{\partial J}{\partial b_j^{(l)}} = e_j^{(l)}$$

Nhận thấy rằng trong các công thức trên đây, việc tính các $e_j^{(l)}$ đóng một vai trò quan trọng. Hơn nữa, để tính được giá trị này, ta cần tính được các $e_j^{(l+1)}$. Nói cách khác, ta cần tính *ngược* các giá trị này từ cuối. Cái tên *Backpropagation* cũng xuất phát từ việc này.

Thuật toán *Backpropagation* tới $W^{(l)}$ và vector *bias* $b^{(l)}$ cho mini-batch:

1. *Bước feedforward: Với toàn bộ dữ liệu (batch) hoặc một nhóm dữ liệu (mini-batch) đầu vào* $\mathbf{X}$*, tính giá trị đầu ra của network, trong quá trình tính toán, lưu lại các activation* $\mathbf{A}^{(l)}$ *tại mỗi layer. Mỗi cột của* $\mathbf{A}^{(l)}$ *tương ứng với một cột của* $\mathbf{X}$*, tức một điểm dữ liệu đầu vào.*
2. *Với output layer, tính*

$$\mathbf{E}^{(L)} = \nabla_{\mathbf{Z}^{(L)}} J; \quad \nabla_{\mathbf{W}^{(L)}} J = \mathbf{A}^{(L-1)} \mathbf{E}^{(L)T}; \quad \nabla_{\mathbf{b}^{(L)}} J = \sum_{n=1}^{N} \mathbf{e}_n^{(L)}$$

3. *Với* $l = L-1, L-2, ..., 1$*, tính:*

$$\mathbf{E}^{(l)} = \left(\mathbf{W}^{(l+1)} \mathbf{E}^{(l+1)}\right) \odot f'(\mathbf{Z}^{(l)})$$

trong đó $\odot$ *là element-wise product hay Hadamard product tức lấy từng thành phần của hai ma trận nhân với nhau để được ma trận kết quả.*
4. *Cập nhật đạo hàm cho ma trận trọng số và vector biases:*

$$\nabla_{\mathbf{W}^{(l)}} J = \mathbf{A}^{(l-1)} \mathbf{E}^{(l)T}; \quad \nabla_{\mathbf{b}^{(l)}} J = \sum_{n=1}^{N} \mathbf{e}_n^{(l)}$$

Hình 16: Thuật toán Backpropagation

Sau đó, ta tiến hành cập nhật *weights* và *bias* của từng lớp:

$$W^{(l)} -= lr * \nabla_{W^{(l)}} J$$

$$b^{(l)} -= lr * \nabla_{b^{(l)}} J$$

Trong đó:

- *lr* một trong số những siêu tham số quan trọng nhất của mô hình. Nó được hiểu là một phần tỷ lệ của một bước dịch chuyển trọng số mô hình được cập nhật theo các mini-batch truyền vào. Độ lớn của learning rate sẽ ảnh hưởng trực tiếp tới tốc độ hội tụ của hàm mất mát tới điểm cực trị toàn cục.

2. Xây dựng mạng neural 3 lớp bằng Python

a. Class Input layer

```
1. class Input_Layer:
2.     def __init__(self, X, Y, output_size):
3.         self.X = X
```

```
4.          self.Y = Y
5.
6.          self.output_size = output_size
```

Lớp này là lớp đầu có chức năng nhận dữ liệu và truyền cho các lớp phía sau.

Trong đó:

- X là dữ liệu đầu vào.
- Y là nhãn thực tương ứng với dữ liệu đầu vào X.
- *output_size* là số nút của input layer.

b. Class Dense layer

Để dễ dàng cho việc cài đặt, ta sẽ tách hàm Activation trong các nút ra riêng thành lớp activation.

```
1. class Dense_Layer:
2.     def __init__(self, input_size=None, output_size=None):
3.         self.input_size = input_size
4.         self.output_size = output_size
5.         self.W = None
6.         self.B = None
7.
8.     def forward_propagation(self, inputs):
9.         self.inputs = inputs
10.        self.outputs = np.dot(inputs, self.W) + self.B
11.
12.    def back_propagation(self, E):
13.        self.dW = np.dot(self.inputs.T, E)
14.        self.dB = np.sum(E, axis=0, keepdims=True)
15.
16.        self.dInputs = np.dot(E, self.W.T)
```

Lớp này thực hiện chức năng đưa ra *outputs* = *inputs* · W + B.

Trong đó:

- *input_size*, *output_size* lần lượt là số lượng đầu vào và số nút của layer hiện tại. *input_size* và *output_size* dùng để khởi tạo ma trận trọng số W và vector *bias* B giữa layer phía trước và layer hiện tại.

- Phương thức *forward_propagation* thực hiện chức năng đưa ra *outputs* = *inputs* · W + B, với *inputs* là đầu vào mà nút nhận được, *outputs* là đầu ra của nút sau khi thực hiện tính toán.

- Phương thức *back_propagation* thực hiện chức năng tính đạo hàm của hàm mất mát so với W và B ở lớp hiện tại lần lượt là *d*W, *d*B thông qua *E* ở các lớp phía sau truyền đến. *dInputs* là E mới được tính ở lớp hiện tại và truyền tiếp cho lớp phía trước. Cách cài đặt được thực hiện theo ý tưởng *Hình 16* và được điều chỉnh để dễ dàng thực hiện trên máy tính.

c. Class Activation layer

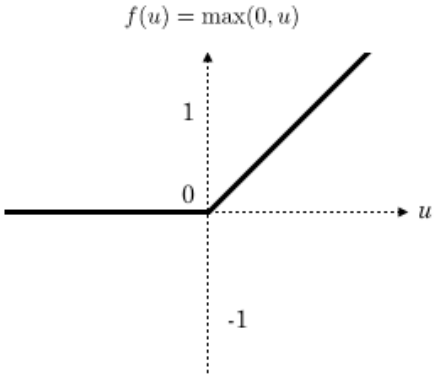
```
1. class Activation_Layer:
2.     def __init__(self, activation=None):
3.         self.activation = activation
4.         self.input_size = None
5.         self.output_size = None
6.
7.     def forward_propagation(self, inputs):
8.         self.inputs = inputs
9.
10.        if self.activation == 'relu':
11.            self.outputs = np.maximum(0, inputs)
12.        elif self.activation == 'softmax':
13.            exp_values = np.exp(inputs - np.max(inputs, axis=1,
14.                                                keepdims=True))
15.            # Normalize them for each sample
16.            probabilities = exp_values / np.sum(exp_values, axis=1,
17.                                                keepdims=True)
18.
19.            self.outputs = probabilities
20.
21.    def back_propagation(self, E):
22.        if self.activation == 'relu':
23.            self.dInputs = E.copy()
24.            self.dInputs[self.inputs <= 0] = 0
25.        elif self.activation == 'softmax':
26.            self.dInputs = np.empty_like(E)
27.                # Enumerate outputs and gradients
28.            for index, (single_output, single_dvalues) in \
29.            enumerate(zip(self.outputs, E)):
30.                # Flatten output array
```

```
                single_output = single_output.reshape(-1, 1)
                # Calculate Jacobian matrix of the output
                jacobian_matrix = np.diagflat(single_output) - \
                                  np.dot(single_output,
                                         single_output.T)

                # Calculate sample-wise gradient
                # and add it to the array of sample gradients
                self.dInputs[index] = np.dot(jacobian_matrix,
                                             single_dvalues)
```

Trong đó:

- *activation* là hàm activation mà ta muốn chọn.
- Phương thức *forward_propagation* nhận đầu vào *inputs* từ layer phía trước và cho ra *outputs* là *inputs* sau khi đã áp hàm activation. Trong phạm vi đề tài, ta chỉ cần 2 hàm activation là: *relu* và *softmax*.



Hình 17: Hàm relu

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Hình 18: Hàm softmax

- Phương thức *back_propagation* thực hiện tính E để truyền cho lớp *Dense*. Theo ý tưởng *Hình 16*, ta có $E^{(l)} = \left(W^{(l+1)}E^{(l+1)}\right) \odot \mathrm{f}'\left(Z^{(l)}\right)$. Do $\left(W^{(l+1)}E^{(l+1)}\right)$

đã được truyền đi từ layer phía sau, nên lớp activation hiện tại thêm vào E hiện tại một đại lượng $f'(Z^{(l)})$ để truyền đến lớp *Dense* liền trước nó.

d. Class Neural network

```
class NeuralNetwork:
    def __init__(self, *args):
        self.numLayers = len(args)
        self.layers = []

        for layer in args:
            self.layers.append(layer)
            if len(self.layers) > 1:
                if type(self.layers[-1]).__name__ == \
                'Dense_Layer':
                    self.layers[-1].input_size = \
                    self.layers[-2].output_size
                elif type(self.layers[-1]).__name__ == \
                'Activation_Layer':
                    self.layers[-1].input_size = \
                    self.layers[-2].input_size
                    self.layers[-1].output_size = \
                    self.layers[-2].output_size


    def forward_propagation(self, X):
        for i in range(1, self.numLayers):
            if i == 1:
                self.layers[i].forward_propagation(X)
            else:
                self.layers[i].forward_propagation(\
                                        self.layers[i-1].outputs)

    def compute_loss(self, Y, Y_pred):
        if self.loss == 'cross_entropy':
            return cross_entropy(Y, Y_pred)

    def back_propagation(self, Y):
        if self.loss == 'cross_entropy':
            self.layers[-1].dInputs = (self.layers[-1].outputs - Y) / \
            (self.layers[-1].outputs.shape[0])

        for i in range(-2, -self.numLayers, -1):
            self.layers[i].back_propagation(self.layers[i+1].dInputs)

    def update_weights(self, lr):
```

```
        for i in range(1, self.numLayers):
            if type(self.layers[i]).__name__ == 'Dense_Layer':
                self.layers[i].W -= lr * self.layers[i].dW
                self.layers[i].B -= lr * self.layers[i].dB

    def predict(self, X):
        self.forward_propagation(X)
        return self.layers[-1].outputs

    def evaluate(self, Y, Y_pred):
        pred = np.asarray([Y_pred[i].argmax() for i in \
                           range(len(Y_pred))])
        truth = np.asarray([Y[i].argmax() for i in range(len(Y))])
        return np.mean(pred == truth)

    def compile(self, lr=None, loss=None, seed=None):
        self.lr = lr
        self.loss = loss
        self.seed = seed

        np.random.seed(seed)
        random.seed(seed)

        for i in range(1, self.numLayers):

            if type(self.layers[i]).__name__ == 'Dense_Layer':
                self.layers[i].W = \
                np.random.randn(self.layers[i].input_size, \
                                self.layers[i].output_size) / \
                np.sqrt(self.layers[i].input_size * \
                        self.layers[i].output_size)

                self.layers[i].B = \
                np.zeros((1, self.layers[i].output_size))

    def lr_exp_decay(self, epoch):
        decay = 0.1
        return self.lr * math.exp(-decay * epoch)

    def fit(self, epochs, batch_size=None):
        X = self.layers[0].X
        Y = self.layers[0].Y

        if batch_size == None:
            batch_size = X.shape[0]

        batches = math.ceil(X.shape[0] / batch_size)
```

```
89.
90.         for epoch in range(epochs):
91.             loss_accumulated = 0
92.             acc_accumulated = 0
93.
94.             for batch in range(batches):
95.                 batch_X = X[batch_size*batch:\
96.                             min(batch_size*(batch+1), X.shape[0])]
97.                 batch_Y = Y[batch_size*batch:\
98.                             min(batch_size*(batch+1), X.shape[0])]
99.
100.                self.forward_propagation(batch_X)
101.                self.back_propagation(batch_Y)
102.                self.update_weights(self.lr_exp_decay(epoch))
103.
104.                loss_score = self.compute_loss(batch_Y, \
105.                                               self.layers[-1].outputs)
106.
107.                Y_pred = self.predict(batch_X)
108.                acc = self.evaluate(batch_Y, Y_pred)
109.
110.                loss_accumulated += loss_score
111.                acc_accumulated += acc
112.
113.            loss_accumulated /= batches
114.            acc_accumulated /= batches
115.
116.            print(f'Epoch {epoch+1}/{epochs}: \
117.            loss: {loss_accumulated} \
118.            - acc: {acc_accumulated}')
```

Trong đó:

- Phương thức *__init__*, đối với lớp *Dense*, khởi tạo số nút trong các layer sao cho số nút của layer phía trước bằng với số lượng đầu vào của layer phía sau. Đối với lớp *activation*, sẽ có số lượng đầu vào và số nút tương ứng với lớp *Dense* phía trước.
- Phương thức *forward_propagation* thực hiện lan truyền dữ liệu từ layer đầu đến layer cuối. Đầu vào của layer sẽ là đầu ra của layer phía trước.

- Phương thức *compute_loss* tính *loss* dựa trên kết quả dự đoán (Y_*pred*) và kết quả thực (Y). Hàm mất mát sử dụng sẽ là *cross_entropy*.

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Hình 19: Hàm cross entropy

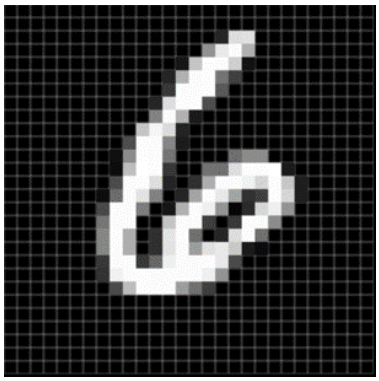
- Phương thức *back_propagation* lan truyền ngược dữ liệu từ layer cuối lên layer đầu. Với hàm mất mát là *cross_entropy* và có hàm activation là *softmax*, ta sẽ có $E^{(L)} = Y_{pred} - Y$.
- Phương thức *update_weights* cập nhật ma trận trọng số và vector *bias*.
- Phương thức *predict* dự đoán kết quả từ dữ liệu đầu vào X.
- Phương thức *evaluate* đánh giá mô hình qua kết quả dự đoán và kết quả thực.
- Phương thức *compile* khai báo các tham số cho mô hình và khởi tạo các ma trọng số W và vector *bias* B.
- Phương thức *lr_exp_decay* giảm *lr* qua từng epoch.
- Phương thức *fit* học dữ liệu theo từng *batch* lần lượt qua các bước *forward_propagation*, *compute_loss*, *back_propagation* và *update_weights*.

3. Huấn luyện mạng neural 3 lớp đã xây dựng trên tập dữ liệu Mnist và đánh giá kết quả

a. Bài toán

- Input: Bức ảnh 28 x 28 pixels từ bộ dữ liệu Mnist.
- Output: Số tương ứng với số có trong bức ảnh.
- Ví dụ:

- Input:



- Output: 6

b. Giải quyết bài toán

Trước khi đưa phải mạng, ta phải tiền xử lý dữ liệu:

```
1. from keras.datasets import mnist
2. (train_X, train_y), (test_X, test_y) = mnist.load_data()
3.
4. def to_one_hot(labels, dimension):
5.     result = np.zeros((len(labels), dimension), dtype = np.uint8)
6.     for i, label in enumerate(labels):
7.         result[i, label] = 1
8.     return np.asarray(result)
9.
10.   print('X_train: ' + str(train_X.shape))
11.   print('Y_train: ' + str(train_y.shape))
12.
13.   # Normalize
14.   train_X = (train_X / 255).astype(np.float64)
15.   test_X = (test_X / 255).astype(np.float64)
16.   train_size = train_X.shape[0]
17.   test_size = test_X.shape[0]
18.
19.   # Flatten data
20.   trainX = np.asarray([train_X[i].flatten() for i in range(train_size)])
21.   testX = np.asarray([test_X[i].flatten() for i in range(test_size)])
22.
23.   # Encode label to one hot
24.   trainY = to_one_hot(train_y, 10)
25.   testY = to_one_hot(test_y, 10)
26.   print(trainY.shape)
```

Đầu tiên ta normalize dữ liệu nhằm làm cho dữ liệu gần gũi với máy tính bằng cách chia các phần tử trong dữ liệu cho 255 để cho giá trị về đoạn [0, 1].

Sau đó, ta duỗi các ma trận ảnh sang vector có số chiều là 28 * 28 để có thể truyền vào mạng.

Cuối cùng, đối với từng nhãn ta sẽ tiến hành mã hóa *one hot*. Ví dụ: có nhãn là 5 thì dạng *one hot* của nó sẽ là [0, 0, 0, 0, 0, 1, 0, 0, 0, 0], vector đó gồm 10 phần tử và phần tử tại vị trí 5 có giá trị là 1, biểu thị cho xác suất nhãn là số 5 là 100%.

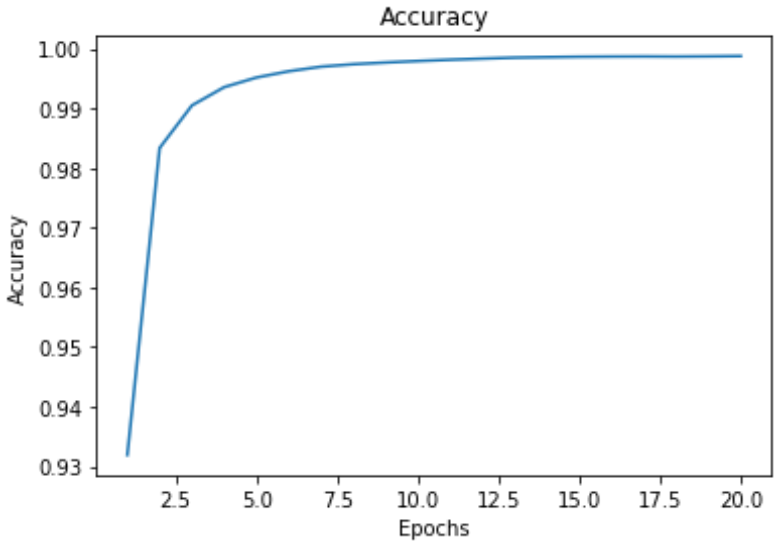
Để giải quyết bài toán, ta sử dụng mạng 3 lớp đã xây dựng ở trên:

```
1. model = NeuralNetwork(Input_Layer(trainX, trainY, output_size=28*28),
2.
3.                        Dense_Layer(output_size=256),
4.                        Activation_Layer('relu'),
5.
6.                        Dense_Layer(output_size=10),
7.                        Activation_Layer('softmax'))
8.
9.    model.compile(lr=0.5, loss='cross_entropy', seed=123)
10.   model.fit(epochs=20, batch_size=128)
```

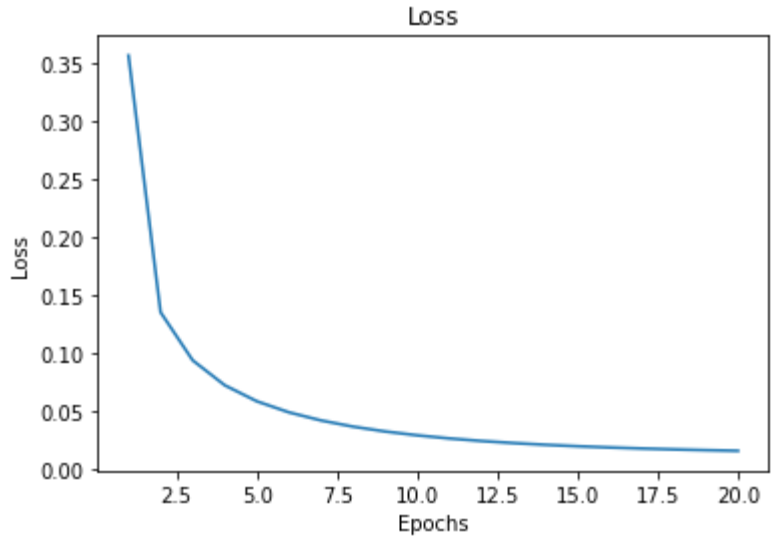
Trong đó:

- Lớp đầu tiên là input layer có số nút là 28 * 28.
- Lớp thứ hai gồm 256 nút và có activation là *relu*.
- Lớp cuối cùng là output layer gồm 10 nút (biểu thị cho 10 chữ số) và có activation là *softmax*.

Ta tiến hành train mô hình với số *epoch* là 20, *batch_size* là 128, mục đích của chia dữ liệu ra thành từng batch nhằm tránh tràn bộ nhớ khi load toàn bộ dữ liệu, thay vào đó ta chỉ load từng phần nhỏ đưa vào.



Hình 20: Sự thay đổi qua các epoch của accuracy



Hình 21: Sự thay đổi qua các epoch của loss

Kết quả dự đoán trên tập test sau khi huấn luyện như trên là: 0.9817.

III. Kết luận

Mạng neural 3 lớp mà em xây dựng đã hoạt động và có được kết quả tốt trong việc nhận diện chữ số viết tay. Qua đồ án lần này, em đã biết được cách mạng neural vận hành và giải quyết được nhiều bài toán khó mà mô hình tuyến tính không giải quyết được.

Link github: https://github.com/pattan99/NeuralNetwork

IV. Tài liệu tham khảo

Vũ Hữu Tiệp. “Machine Learning cơ bản”:
https://github.com/tiepvupsu/ebookMLCB/blob/master/book_ML_color.pdf
Neural networks form sratch in Python:
https://nnfs.io