

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

---

# ĐỒ ÁN CUỐI KÌ CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT



## ĐỀ TÀI: MẠNG NEURAL 3 LỚP

Giảng viên lý thuyết:

Nguyễn Thanh Sơn

Giảng viên hướng dẫn thực hành:

Nguyễn Đức Vũ

Người thực hiện:

Trương Thanh Minh – 21520064 – KHTN2021

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

## Mục lục

<b>I. Tổng quan đề tài.....</b>	<b>2</b>
1. Lý do chọn đề tài.....	2
2. Đối tượng và phạm vi đề tài.....	2
3. Mục tiêu đề tài.....	2
<b>II. Nội dung thực hiện đề tài .....</b>	<b>3</b>
1. Cơ sở lý thuyết.....	3
1.1. Mạng neural là gì?.....	3
1.2. Tại sao phải sử dụng mạng neural .....	4
1.3. Layer .....	4
1.4. Units .....	6
1.5. Weights và biases .....	7
1.6. Activation function .....	7
1.7. Feed forward .....	8
1.8. Loss function.....	10
1.9. Backpropagation .....	11
1.10. Cách mạng nơ ron làm việc .....	15
2. Giới thiệu bài toán .....	16
2.1. Phát biểu bài toán.....	16
2.2. Input .....	16
2.3. Output.....	18
3. Áp dụng mạng neural 3 lớp để giải quyết bài toán.....	18

3.1. Khởi tạo tham số .....	18
3.2. Hàm activation .....	19
3.3. Feed forward .....	19
3.4. Backpropagation .....	20
3.5. Optimize .....	20
3.6. Hàm mất mát .....	21
3.7. Train model .....	21
3.8. Tính toán độ chính xác của mô hình trên tập test .....	23
3.9. Dự đoán trên một ảnh chữ số viết tay .....	24
4. <i>Kết quả</i> .....	25
4.1. Nhận xét kết quả .....	25
4.2. So sánh với độ chính xác của mô hình với một số trường hợp khác nhau (về số iteration, hidden size) .....	26
5. <i>Ứng dụng của mạng neural</i> .....	28
<b>III. Kết luận</b> .....	<b>28</b>
<b>IV. Tài liệu tham khảo</b> .....	<b>29</b>

# **I. Tổng quan đề tài**

## **1. Lý do chọn đề tài**

Kể từ khi công nghệ phát triển, mạng neural đã phát triển một cách đáng kinh ngạc so với năm 1943 – năm mà mạng nơ ron nhân tạo đầu tiên được tạo ra bởi nhà sinh lý học Warren McCulloch và nhà logic Walter Pitts. Việc mạng nơ ron phát triển đã kéo theo DeepLearning bùng nổ và trở thành một hiện tượng như hiện nay. Mạng neural phát triển góp phần tăng khả năng phát triển về Deep Learning cũng như Trí tuệ nhân tạo.

Mạng neural mang lại hiệu quả khá cao trong các bài toán phân loại, ngay cả khi đó là mạng neural 3 lớp đơn giản, nó cũng mang lại độ chính xác khá cao, vượt trội hơn so với các phương pháp trước đó.

Chính vì điều đó, trong phạm vi đề tài này, em xin phép tìm hiểu về mạng neural 3 lớp.

## **2. Đối tượng và phạm vi đề tài**

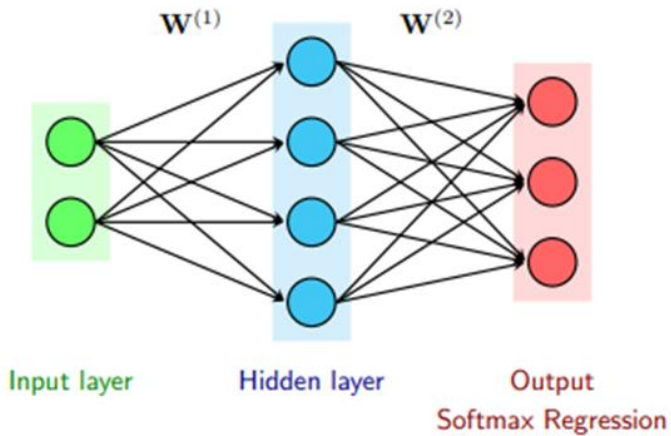
Trong phạm vi đề tài này, em chỉ tập trung vào việc xây dựng mạng nơ ron 3 lớp đơn giản để nhận diện các chữ số viết tay trong bộ dữ liệu phổ biến MNIST. Qua đó, làm rõ một số khái niệm, thuật toán quan trọng trong mạng nơ ron.

## **3. Mục tiêu đề tài**

- Hiểu được các khái niệm cơ bản của mạng nơ ron.
- Hiểu được cách hoạt động của mạng neural.
- Biết cách xây dựng một model phân loại chữ số viết tay.
- Hiểu được điểm mạnh, điểm yếu của mạng nơ ron.

## II. Nội dung thực hiện đề tài

### 1. Cơ sở lý thuyết



Hình 1: Hình ảnh mạng neural 3 lớp

#### 1.1. Mạng neural là gì?

Mạng neural là mạng sử dụng các mô hình toán học phức tạp để xử lý thông tin. Chúng dựa trên mô hình hoạt động của mạng nơ ron sinh học. Tương tự như mạng nơ ron con người, mạng nơ ron nhân tạo kết nối các node đơn giản được gọi là các neurons. Và một tập hợp các node như vậy tạo thành các layer.

Mạng neural nhân tạo sử dụng một loạt các thuật toán được sử dụng để xác định và nhận ra các mối quan hệ trong các tập dữ liệu. Mạng neural được sử dụng trên nhiều công nghệ và ứng dụng khác nhau như thị giác máy tính, nhận dạng giọng nói...

## 1.2. Tại sao phải sử dụng mạng neural

Khác với trong lập trình thi đấu, người dùng có các biến, có các quy tắc và việc của họ là lập trình để từ các quy tắc đó tìm ra output của bài toán. Tuy nhiên việc sử dụng mạng neural không phải thế. Mạng neural giúp ta giải quyết các bài toán mà chúng ta có bộ dữ liệu (đầu vào và đầu ra tương ứng), khi đó, mạng neural sẽ tìm ra quy tắc, các mối quan hệ giữa các tham số. Qua đó, có khả năng dự đoán với độ chính xác khá cao để có thể dự đoán với những dữ liệu chưa nhìn thấy bao giờ. Đây là một bước tiến mới trong nền khoa học của nhân loại, góp phần nâng cao, phát triển cuộc sống của con người.

## 1.3. Layer

Layer là nơi chứa các neurons, do đó, đây cũng là nơi các mạng neural thực hiện quá trình “học” của mình. Mỗi layer có những mục đích và nhiệm vụ khác nhau.

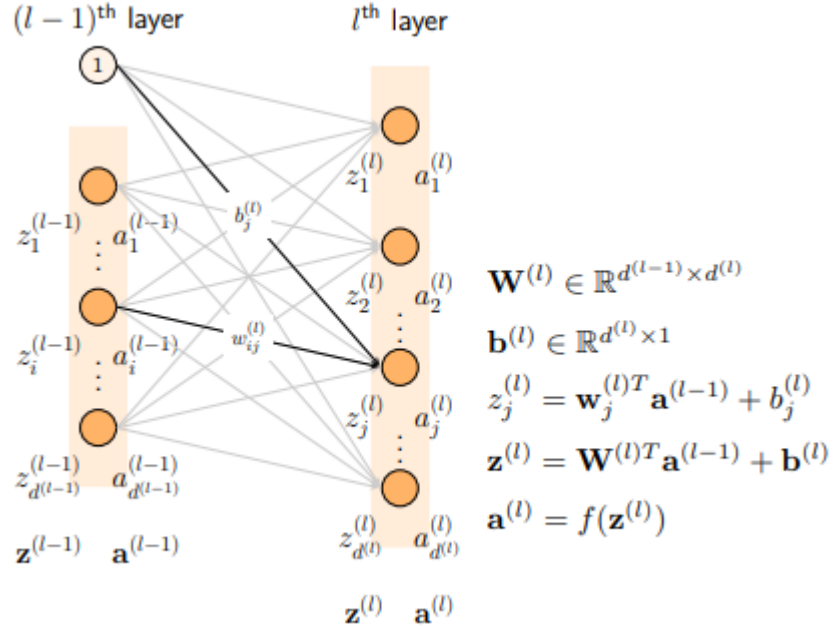
Trong mạng neural, có 2 layer bắt buộc phải có đó là **input layer** và **output layer**. Ngoài ra, nó còn có thể có thêm lớp **hidden layer**.

- Input layer là lớp đầu tiên của mạng (cụ thể trong hình 1, input layer chính là lớp có màu xanh lá). Đây là lớp chịu trách nhiệm cho việc nhận đầu vào của mạng. Thực hiện các phép tính thông qua các neurons của nó và sau đó đầu ra ở input layer sẽ được truyền làm đầu vào cho lớp kế tiếp nó.
- Output layer là lớp cuối cùng của mạng (lớp có màu đỏ), có nhiệm vụ đưa ra kết quả cuối cùng của mạng. Lớp output nhận đầu vào là kết quả

của lớp trước đó, thực hiện các phép tính thông qua các neurons của nó, sau đó, kết quả bài toán sẽ được tính toán. Với các bài toán khác nhau, hàm activation cũng sẽ khác nhau.

- Hidden layer – đây có thể nói sự ra đời của nó làm cho mạng neural vượt trội hơn hầu hết các thuật toán học máy khác. Hidden layer là lớp nằm giữa lớp đầu vào và lớp đầu ra (lớp có màu xanh dương). Đây cũng chính là lý do chính tại sao chúng được gọi là hidden. Từ “hidden” ngụ ý rằng chúng không hiển thị với các hệ thống bên ngoài và là “riêng tư” đối với mạng nơ ron. Thông thường, mỗi lớp ẩn chứa cùng một số lượng neurons. Số lượng lớp ẩn trong mạng nơ ron càng lớn thì thời gian mạng nơ ron tạo ra đầu ra càng lâu và càng có nhiều vấn đề phức tạp mà mạng nơ ron có thể giải quyết và ngược lại. Các nơ ron chỉ đơn giản là tính toán tổng weights, weights, bias và thực hiện hàm kích hoạt.

## 1.4. Units



Hình 2: Các ký hiệu sử dụng trong mạng Neural

Mỗi node hình tròn trong một layer được gọi là một unit hay neuron. Một layer gồm các node xếp dọc với nhau. Các neurons trong mỗi lớp của mạng neural thực hiện cùng một chức năng. Chúng chỉ đơn giản là tính toán tổng trọng số của đầu vào và weights, thêm giá trị bias và thực hiện một hàm activation.

Mỗi neuron sẽ gồm hai thông số chính là  $z, a$ . Trong đó, đầu vào của hidden layer thứ  $l$  được ký hiệu bởi  $z(l)$ , đầu ra của mỗi unit thường được ký hiệu là  $a(l)$  (thể hiện activation, tức giá trị của mỗi unit sau khi ta áp dụng activation function lên đầu vào  $z(l)$ ).



## 1.5. Weights và biases

Weights và bias là hai tham số quan trọng góp phần rất lớn vào việc liệu model của mình có thành công hay không.

Tham số weights được dùng để thể hiện rằng kết nối nào quan trọng hơn kết nối nào. Tham số này được thêm vào model nhằm làm cho model đánh giá đúng vai trò của từng biến trong bộ dữ liệu để từ đó, đưa ra kết quả dự đoán phù hợp nhất.

Biases được thêm vào model nhằm làm cho model trở nên linh hoạt hơn trong việc thể hiện mối quan hệ giữa các điểm dữ liệu.

Kí hiệu:

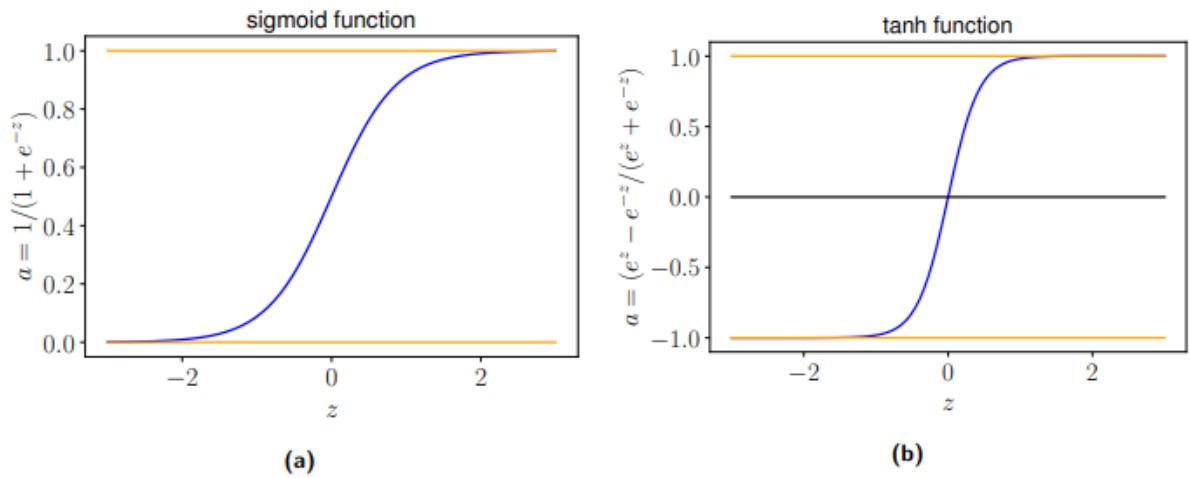
- $L$ : tổng số layer (trừ input layer) trong model
- Các ma trận weights được kí hiệu là  $W^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$ ,  $l = 1, 2, \dots, L$  trong đó  $W^{(l)}$  thể hiện các kết nối từ layer thứ  $l - 1$  tới layer thứ  $l$ . Cụ thể hơn, phần tử  $w_{ij}^{(l)}$  thể hiện kết nối từ node thứ  $i$  của layer thứ  $(l - 1)$  tới node  $j$  của layer thứ  $(l)$ . Các biases của layer thứ  $(l)$  được ký hiệu là  $b^{(l)} \in \mathbb{R}^{d^{(l)}}$ .

## 1.6. Activation function

Mỗi lớp output của một layer (trừ input layer) được tính dựa vào công thức

$$a^{(l)} = f^{(l)}(W^{(l)T} a^{(l-1)} + b^{(l)})$$

Trong đó,  $f^{(l)}$  là một hàm kích hoạt phi tuyến tính. Nếu hàm kích hoạt tại một layer là một hàm tuyến tính, layer này và layer tiếp theo có thể rút gọn thành một layer vì hợp của các hàm tuyến tính là một hàm tuyến tính.

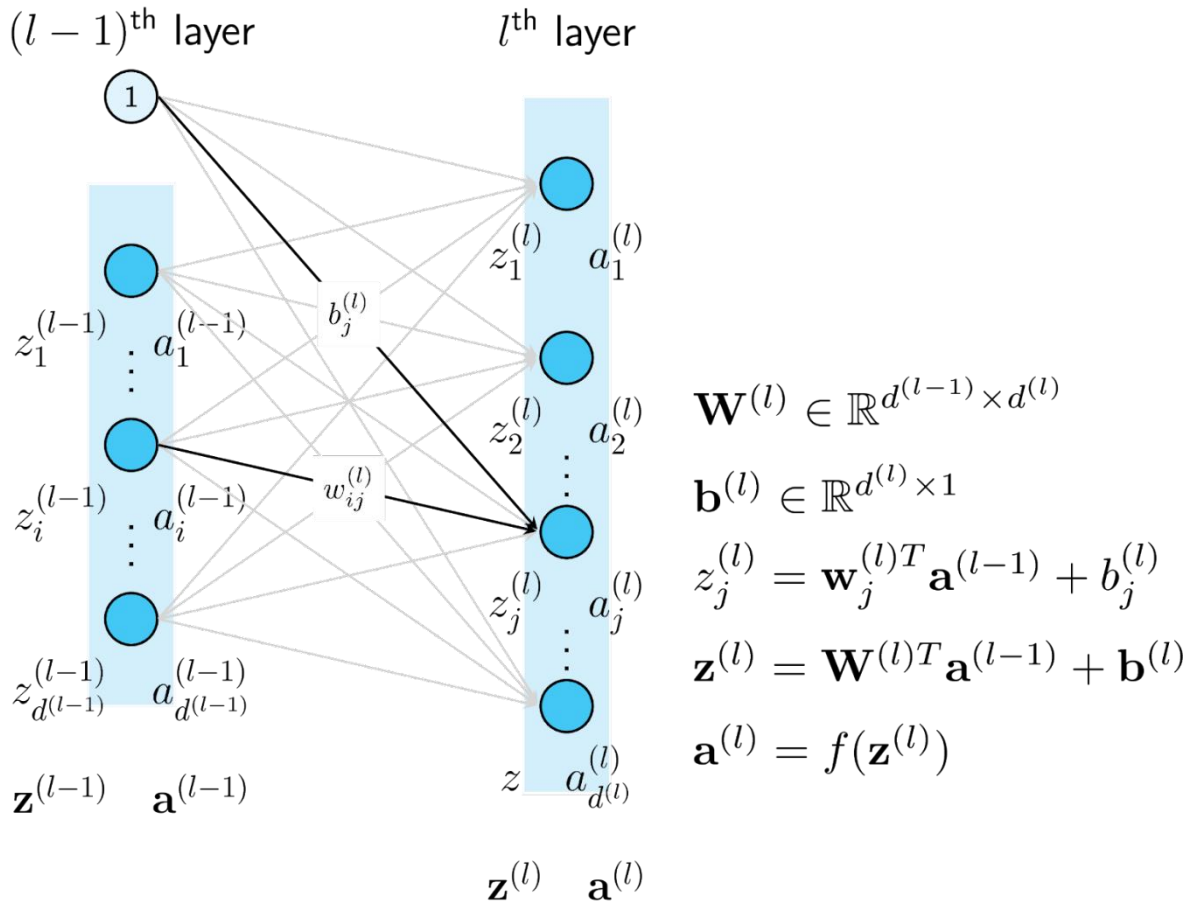


Hình 3: Ví dụ về đồ thị của hàm (a) sigmoid và (b) tanh

## 1.7. Feed forward

Feed forward là thuật toán lan truyền thẳng, nghĩa là các thông tin ở mỗi node sẽ được truyền qua các lớp tiếp theo thông qua các hàm activation.

Để hiểu rõ hơn, ta nhìn vào hình sau:



Hình 4: Hình ảnh minh họa cách feed forward hoạt động

Hình ảnh trên minh họa một cách khá rõ ràng cách feedforward hoạt động. Và cứ như thế, thông tin đầu vào sẽ được lan truyền qua các lớp, rồi cuối cùng là đến lớp output và cho ta kết quả mà mô hình dự đoán.

$$\mathbf{a}^{(0)} = \mathbf{x}$$

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)T} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}, l = 1, 2, \dots, L$$

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{z}^{(l)}), l = 1, 2, \dots, L$$

$$\hat{\mathbf{y}} = \mathbf{a}^{(L)}$$

## 1.8. Loss function

Trong học máy, có khá nhiều hàm mất mát, tuy nhiên, Cross entropy là một trong những hàm được sử dụng phổ biến nhất trong bài toán phân loại.

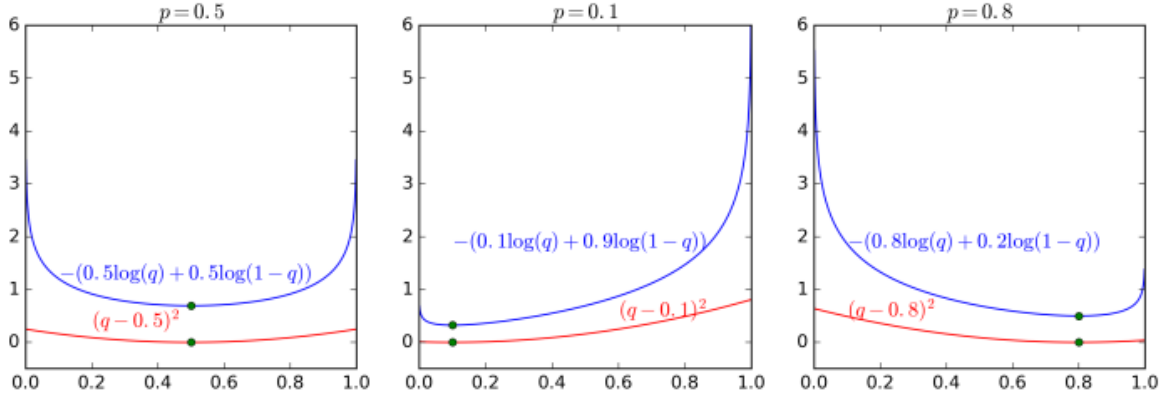
Khi đầu ra dự đoán và đầu ra thực sự của nó là hai vector thể hiện xác suất, khoảng cách của chúng thường được đo bằng một đại lượng được gọi là cross entropy. Một đặc điểm nổi bật của đại lượng này là nếu cố định một vector xác suất, giá trị của nó đạt giá trị nhỏ nhất khi hai vector xác suất bằng nhau, và rất lớn khi hai vector đó lệch nhau nhiều.

Cross entropy giữa hai vector phân phối  $p$  và  $q$  rời rạc được định nghĩa bởi

$$H(p, q) = - \sum_{i=1}^C p_i \log q_i$$

( $C$  là chiều dài vector  $p, q$ )

Cross entropy được sử dụng rộng rãi vì nó thể hiện rõ ràng sự chênh lệch giữa hai điểm dữ liệu. Nghĩa là nó sẽ cho nghiệm gần với giá trị đầu ra thực sự hơn vì những nghiệm ở xa sẽ có loss rất cao. Điều đó làm cho cross entropy được sử dụng rộng rãi. Hình ảnh dưới sẽ thể hiện rõ hơn khi so sánh giữa cross entropy và MSE – một hàm loss khá phổ biến trong các bài toán học máy.



Hình 5: So sánh giữa hàm cross entropy và hàm bình phương khoảng cách.

## 1.9. Backpropagation

Backpropagation là thuật toán được dùng để cập nhật từng trọng số của  $w$ ,  $b$  trong mạng để đầu ra thực tế gần với đầu ra mục tiêu hơn, do đó giảm thiểu lỗi cho từng nơ ron đầu ra và toàn mạng.

Phương pháp phổ biến nhất để tối ưu hàm mất mát là gradient descent (GD). Để áp dụng GD, ta cần tính được đạo hàm của hàm mất mát theo từng trọng số  $W^{(l)}$  và vector bias  $b^{(l)}$ .

Trước khi thực hiện backpropagation, ta gọi đến feedforward để thực hiện tính toán ra giá trị  $\hat{y}$ .

$$a^{(0)} = x$$

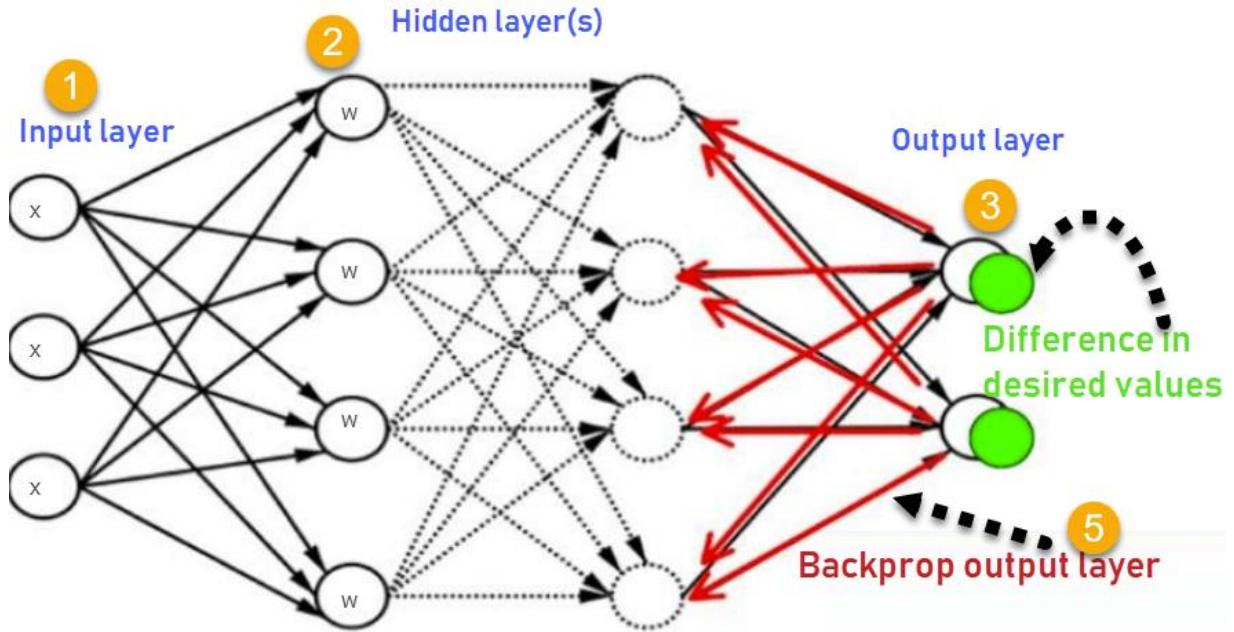
$$z^{(l)} = W^{(l)T} a^{(l-1)} + b^{(l)}, l = 1, 2, \dots, L$$

$$a^{(l)} = f^{(l)}(z^{(l)}), l = 1, 2, \dots, L$$

$$\hat{y} = a^{(L)}$$

Sau đó, ta tiến hành gọi backpropagation để điều chỉnh weight, bias nhằm làm giảm hàm loss. Phương pháp backpropagation giúp tính đạo hàm ngược

từ layer cuối cùng về layer đầu tiên. Layer cuối cùng được tính toán trước thì nó gần gũi với đầu ra dự đoán và hàm mất mát. Việc tính toán đạo hàm của ma trận hệ số trong các layer trước được thực hiện dựa trên một quy tắc chuỗi quen thuộc cho đạo hàm của hàm hợp.



Hình 6: Cách mạng neural làm việc

Gọi  $J$  là hàm mất mát. Đạo hàm của hàm mất mát chỉ theo một thành phần của ma trận trọng số ở output layer:

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

Trong đó,  $e_j^{(L)} = \frac{\partial J}{\partial z_j^{(L)}}$  là một đại lượng không quá khó để tính toán và

$\frac{\partial z_j^{(L)}}{\partial w_{i,j}^{(L)}} = a_i^{(L-1)}$  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:

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

Với đạo hàm theo hệ số ở các lớp  $l$  thấp hơn, ta làm tương tự như trên.

Quay lại với việc tính giá trị  $e_j^{(l)}$ . Ta có:

$$\begin{aligned} e_j^{(l)} &= \frac{\partial J}{\partial z_j^{(l)}} = \frac{\partial J}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \left( \sum_{k=1}^{d^{(l+1)}} \frac{\partial J}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial a_j^{(l)}} \right) f'(z_j^{(l)}) \\ &= \left( \sum_{k=1}^{d^{(l+1)}} e_k^{(l+1)} w_{jk}^{(l+1)} \right) f'(z_j^{(l)}) = (w_{j:}^{(l+1)} e^{(l+1)}) f'(z_j^{(l)}) \end{aligned}$$

Với  $e_j^{(l+1)} = [e_1^{(l+1)}, e_2^{(l+1)}, \dots, e_{d^{(l+1)}}^{(l+1)}]^T \in \mathbb{R}^{d^{(l+1)} \times 1}$  và  $w_{j:}^{(l+1)}$  được hiểu là hàng thứ  $j$  của ma trận  $W^{(l+1)}$ .

Trong đó,  $\sum$  xuất hiện ở hàng thứ hai trong phép tính trên xuất hiện vì  $a_j^{(l)}$  xuất hiện khi tính các  $z_k^{(l+1)}, k = 1, 2, \dots, d^{(l+1)}$ . Biểu thức  $f'$  xuất hiện là vì  $a_j^{(l)} = f(z_j^{(l)})$ .

Tương tự, có thể suy ra

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

Tóm lại, để dễ hình dung và quan sát, dưới đây, em xin tóm tắt lại các bước làm:

Bước feedforward: Với 1 giá trị đầu vào  $X$ , tính giá trị đầu ra của mạng neural, trong quá trình tính toán, lưu lại các activation  $a^{(l)}$  tại mỗi layer.

Với output layer, tính  $e_j^{(L)} = \frac{\partial J}{\partial z^{(L)}}$

Từ đó suy ra 
$$\begin{cases} \frac{\partial J}{\partial W^{(L)}} = a^{(L-1)} e^{(L)T} \\ \frac{\partial J}{\partial b^{(L)}} = e^{(L)} \end{cases}$$

Với  $l = L - 1, L - 2, \dots, 1$ , tính:  $e^{(l)} = (W^{(l+1)} e^{(l+1)}) \odot f'(z^{(l)})$

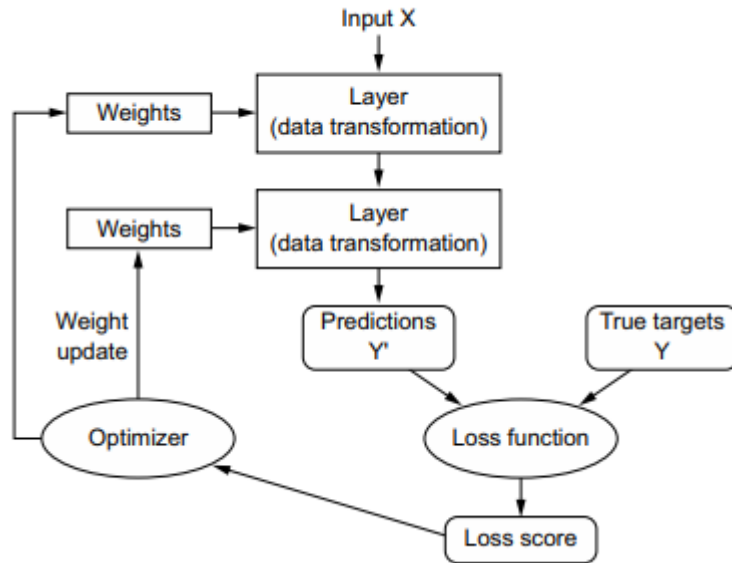
Trong đó,  $\odot$  là lấy từng thành phần của hai vector nhân với nhau để được vector kết quả.

Cập nhật đạo hàm cho ma trận trọng số và vector biases:

$$\begin{cases} \frac{\partial J}{\partial W^{(l)}} = a^{(l-1)} e^{(l)T} \\ \frac{\partial J}{\partial b^{(l)}} = e^{(l)} \end{cases}$$



## 1.10. Cách mạng nơ ron làm việc



Hình 7: Hình ảnh mô tả cách mạng neural làm việc

- **Bước 1:** Ban đầu người ta sẽ khởi tạo các tham số  $w$ ,  $b$ , số node trong input layer, output layer và hidden layer (nếu có).
- **Bước 2:** Gọi đến hàm feed forward để truyền tải thông tin và tính toán giá trị lỗi.
- **Bước 3:** Gọi hàm backpropagation để tính giá trị đạo hàm của  $W$ ,  $b$ .
- **Bước 4:** tối ưu thông qua quy tắc đi ngược đạo hàm.
- **Bước 5:** quay lại bước 2 để gọi feed forward để tiếp tục cho model “học”. Nếu đủ số iteration thì dừng.

## **2. Giới thiệu bài toán**

### **2.1. Phát biểu bài toán**

Bài toán của chúng ta là xây dựng một mô hình có thể nhận diện các chữ số viết tay. Bài toán cho chúng ta bộ dữ liệu MNIST để train model. Đây là bài toán kinh điển khi tìm hiểu về mạng nơ ron cũng như Deep Learning.

### **2.2. Input**

Dữ liệu đầu vào là các hình ảnh của bộ dữ liệu MNIST (Modified National Institute of Standards and Technology database). Đây là bộ dữ liệu lớn nhất về chữ số viết tay thường được dùng trong việc huấn luyện các hệ thống xử lý hình ảnh khác nhau.

Bộ dữ liệu gồm hai tập con:

- Tập dữ liệu huấn luyện (training set): có tổng cộng 60k ví dụ khác nhau về chữ số viết tay từ 0 đến 9.
- Tập dữ liệu kiểm tra (test set): có 10k ví dụ khác nhau.
- Tất cả các tập dữ liệu trên đều được gán nhãn.

Dưới đây là ví dụ về một số hình ảnh được trích ra từ MNIST.



Hình 8: Một số hình ảnh về bộ dữ liệu MNIST

Mỗi bức ảnh là một ảnh trắng đen (có 1 channel), có kích thước 28x28 pixel (tổng cộng 784 pixels). Mỗi pixel mang giá trị là một số tự nhiên từ 0 đến 255. Các pixel màu đen có giá trị bằng 0, các pixel càng trắng thì có giá trị càng cao (nhưng không quá 255). Dưới đây là một ví dụ về chữ số 5 và các giá trị pixel của nó được lấy từ bộ dữ liệu MNIST. (để dễ quan sát ma trận pixel của nó, tôi đã resize ảnh trên thành 14x14).



Hình 9: Một chữ số trong bộ dữ liệu MNIST

```
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,  0,  0,  5,  9, 66, 50,105, 94,  0,  0,
        0],
       [ 0,  0,  0,12,139,189,232,253,253,143,158, 75,  0,  0,
        0],
       [ 0,  0,  0,  5,177,217,241, 98,171,  0,  0,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,  4, 74,197,  1,  0,  0,  0,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,  0,  3,180,114, 27,  0,  0,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,  0,  0,20,181,220, 51,  0,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,  0,  0,  0,  4,149,236,16,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,  0,  0,47,165,236,223,  1,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,22,151,245,239,134, 20,  0,  0,  0,  0,
        0],
       [ 0,  0,57,167,245,251,148, 22,  0,  0,  0,  0,  0,  0,
        0],
       [ 0,  0,97,127, 87, 37,  0,  0,  0,  0,  0,  0,  0,  0,
        0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
        0]], dtype=uint8)
```

Hình 10: Giá trị pixel của chữ số ở Hình 9

## 2.3. Output

- Output của bài toán là nhãn của chữ số đó.
- Ví dụ input của bài toán của là hình 2 thì khi đó, output của bài toán sẽ là 5.

## 3. Áp dụng mạng neural 3 lớp để giải quyết bài toán

Xây dựng mạng neural 3 lớp trên Python

### 3.1. Khởi tạo tham số

Như đã nói ở trên, ban đầu, ta sẽ khởi tạo giá trị của mạng  $W$ ,  $b$  và số node trong input layer, hidden layer và output layer để phục vụ cho các bước phía sau. Cụ thể  $W^{(1)}$  chính là `first_layer['weight']` thể hiện trọng số nối từ lớp input đến lớp hidden.  $W^{(2)}$  chính là `second_layer['weight']` thể hiện trọng số nối từ lớp hidden đến lớp output.  $b^{(1)}$  chính là `first_layer['bias']` thể hiện giá trị bias của lớp hidden.  $b^{(2)}$  chính là `second_layer['bias']` thể hiện giá trị bias của lớp output.

```
def __init__(self, inputs, hidden, outputs):
    self.first_layer['weight'] = np.random.randn(inputs, hidden) /
np.sqrt(inputs)
    self.first_layer['bias'] = np.random.randn(hidden, 1) / np.sqrt(hidden)
    self.second_layer['weight'] = np.random.randn(hidden, outputs) /
np.sqrt(hidden)
    self.second_layer['bias'] = np.random.randn(outputs, 1) / np.sqrt(outputs)

    self.input_size = inputs
    self.hid_size = hidden
    self.output_size = outputs
```

### 3.2. Hàm activation

Trong hàm này, ta sẽ ghi công thức của hàm, và cách tính đạo hàm của nó nếu có yêu cầu. Hàm này sẽ được dùng chủ yếu trong feedforward và backpropagation. Ở đây, em nêu ra một vài hàm phổ biến.

```
def activfunc(self, Z, type = 'ReLU', deri = False):
    if type == 'ReLU':
        if deri == True: # neu co dao ham
            return np.array([1 if i > 0 else 0 for i in np.squeeze(Z)])
        else:
            return np.array([i if i > 0 else 0 for i in np.squeeze(Z)])
    elif type == 'Sigmoid':
        if deri == True:
            return 1/(1+np.exp(-Z))*(1-1/(1+np.exp(-Z)))
        else:
            return 1/(1+np.exp(-Z))
    elif type == 'tanh':
        if deri == True:
            return
        else:
            return 1-(np.tanh(Z))**2
    else:
        raise TypeError('Invalid type!')
```

### 3.3. Feed forward

Cách tính tương tự giống như đã nói ở phần trên

```
def feedforward(self, x, y):
    Z1 = np.dot(self.first_layer['weight'].T, x).reshape((self.hid_size,1)) +
    self.first_layer['bias'] # Z[1] = W[1].T * X + b[1]
    A1 = np.array(self.activfunc(Z1)).reshape((self.hid_size,1)) # A[1] =
    ReLU(Z[1])
    Z2 = np.dot(self.second_layer['para'].T, A1).reshape((self.output_size,1))
    + self.second_layer['bias'] # Z[2] = W[2].T * A[1] + b[2]
    y_hat = np.squeeze(self.Softmax(Z2))
    error = self.cross_entropy_error(y_hat,y)
    para = {
```

```

        'Z1': Z1,
        'A1': A1,
        'Z2': Z2,
        'y_hat': y_hat,
        'error': error
    }
    return para

```

### 3.4. Backpropagation

Về việc tính backpropagation cũng thực hiện theo tư tưởng ở trên. Ở *dtmp*, nó đại diện cho giá trị đạo hàm của hàm loss.

```

def back_propagation(self,x,y,f_result):
    # onehot encoder
    E = np.array([0]*self.output_size).reshape((1, self.output_size))
    E[0][y] = 1
    dtmp = (f_result['y_hat'] - E).reshape((self.output_size,1))
    db2 = dtmp
    dW2 = np.dot(dtmp, f_result['A1'].T)
    delta = np.dot(self.second_layer['weight'], dtmp) *
self.activfunc(f_result['Z1'], deri = True).reshape(self.hid_size, 1)
    db1 = delta
    dW1 = np.dot(db1.reshape((self.hid_size, 1)),x.reshape((1, 784)))

    grad = {
        'dW2':dW2,
        'db2':db2,
        'db1':db1,
        'dW1':dW1
    }
    return grad

```

### 3.5. Optimize

Phần optimize là phần tính toán dựa trên giá trị đạo hàm đã tính toán ở phần backpropagation. Đây là phần gọi nôm na là “đi ngược đạo hàm”. Nghĩa là nếu giá trị đạo hàm đang dương thì sẽ cập nhật giá trị theo chiều âm và

ngược lại. Mục tiêu của nó là cập nhật lại giá trị sao cho nó dần tiếp cận đến điểm cực tiểu.

```
def optimize(self, b_result, learning_rate):
    self.second_layer['weight'] -= learning_rate*b_result['dw2'].T
    self.second_layer['bias'] -= learning_rate*b_result['db2']
    self.first_layer['weight'] -= learning_rate*b_result['dw1'].T
    self.first_layer['bias'] -= learning_rate*b_result['db1']
```

### 3.6. Hàm mất mát

Ở đây, em sử dụng hàm cross entropy để tính toán (lí do đã được trình bày ở phần trên).

```
def cross_entropy_error(self, y_hat, y):
    return -np.log(y_hat[y])
```

Hàm ở trên chỉ áp dụng cho một ảnh. Để tính loss cho một tập dữ liệu ta phải thực hiện tính tổng và sau đó lấy giá trị trung bình.

```
def loss(self, X_train, Y_train):
    loss = 0
    for n in range(len(X_train)):
        y = Y_train[n]
        x = X_train[n][:]
        loss += self.feedforward(x,y)['error']
    return loss / len(X_train)
```

### 3.7. Train model

Để model hoạt động hiệu quả, ta random các ảnh để đưa vào train.

```
rand_indices = np.random.choice(len(X_train), num_iterations, replace=True)
```

Với mỗi epoch, đầu tiên, ta sẽ truyền thông tin và tính giá trị loss thông qua feedforward. Sau đó, tính đạo hàm weight, bias thông qua backpropagation. Và cuối cùng là cập nhật lại giá trị cho chúng thông qua hàm optimize.

Sau khi kết thúc một epoch, ta sẽ tính toán hàm loss cho tập train và tính độ chính xác cho tập test.

```

def train(self, X_train, Y_train, num_iterations = 1000, learning_rate = 0.5):
    # generate random list index for train
    rand_indices = np.random.choice(len(X_train), num_iterations,
replace=True)

    def l_rate(base_rate, ite, num_iterations, schedule = False):
        # determine whether to use the learning schedule
        if schedule == True:
            return base_rate * 10 ** (-np.floor(ite/num_iterations*5))
        else:
            return base_rate

    count = 1
    loss_dict = {}
    test_dict = {}
    num_epochs = 1
    for i in rand_indices:
        f_result = self.feedforward(X_train[i], Y_train[i])
        b_result = self.back_propagation(X_train[i], Y_train[i], f_result)
        self.optimize(b_result, l_rate(learning_rate, i, num_iterations,
True))

        if count % 1000 == 0:
            print('Trained for {} times,'.format(count))
            if count % 5000 == 0:
                loss = self.loss(X_train,Y_train)
                test = self.testing(x_test, y_test)
                print('Trained for {} epoch(s),' .format(num_epochs), 'loss =
{}, test = {}'.format(loss,test))
                loss_dict[str(count)]=loss
                test_dict[str(count)]=test
                num_epochs += 1

        count += 1

    print('Training finished!')
    return loss_dict, test_dict

```



```
Trained for 11 epoch(s), loss = 0.12855110326055666, test = 0.9603
Trained for 56000 times,
Trained for 57000 times,
Trained for 58000 times,
Trained for 59000 times,
Trained for 60000 times,
Accuracy Test: 0.9619
Trained for 12 epoch(s), loss = 0.12026835961516827, test = 0.9619
Trained for 61000 times,
Trained for 62000 times,
Trained for 63000 times,
Trained for 64000 times,
Trained for 65000 times,
Accuracy Test: 0.956
Trained for 13 epoch(s), loss = 0.13294948251587416, test = 0.956
Trained for 66000 times,
Trained for 67000 times,
Trained for 68000 times,
Trained for 69000 times,
Trained for 70000 times,
Accuracy Test: 0.9627
Trained for 14 epoch(s), loss = 0.11071839164981077, test = 0.9627
Trained for 71000 times,
Trained for 72000 times,
Trained for 73000 times,
Trained for 74000 times,
Trained for 75000 times,
Accuracy Test: 0.9674
Trained for 15 epoch(s), loss = 0.10326325270310269, test = 0.9674
Trained for 76000 times
```

Hình 11: Quá trình train

### 3.8. Tính toán độ chính xác của mô hình trên tập test

Sau khi train xong, làm thế nào để đánh giá được độ chính xác của mô hình? Đó chính là lý do tại sao người ta lại cần dùng đến bộ test. Để tính toán được độ chính xác của model, ta gọi đến feedforward để tính ra giá trị dự đoán

của từng ảnh trong tập test và mang nó ra so với kết quả thực. Từ đó tính được độ chính xác của mô hình.


```
def testing(self, X_test, y_test):
    total_correct = 0
    for n in range(len(X_test)):
        y = y_test[n]
        x = X_test[n][:]
        prediction = np.argmax(self.feedforward(x, y)['y_hat'])
        if (prediction == y):
            total_correct += 1
    print('Accuracy Test: ', total_correct / len(X_test))
    return total_correct/np.float(len(X_test))
```

### 3.9. Dự đoán trên một ảnh chữ số viết tay

Hàm này khá đơn giản, bởi vì nó đã được tính ở trong feedforward. Tuy nhiên, ta viết thêm để dễ dàng sử dụng.

```
def predict(self, x):
    return np.argmax(self.feedforward(x, 1)['y_hat'])
```

Vì hàm feedforward yêu cầu phải truyền vào hai tham số là đầu vào và label tương ứng của nó. Nhưng khi predict lại không có label, do đó, ta có thể mặc định nó là một số bất kỳ (vì nó không ảnh hưởng tới kết quả dự đoán).

```
model.predict(x_test[11])  
6  
  
y_test[11]  
6  
  
cv2.imshow(x_test[11])  

```

Hình 12: Minh họa việc predict

## 4. Kết quả

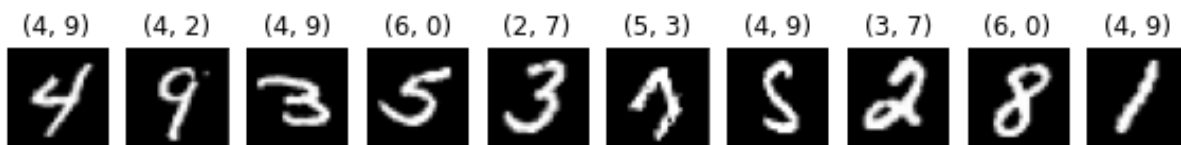
### 4.1. Nhận xét kết quả

Với việc xây dựng mạng neural 3 lớp đơn giản, ta đã có thể dự đoán chữ số viết tay trong bộ dữ liệu MNIST. Cụ thể, model đã đạt được **accuracy** trên bộ test là **0.9736** sau **200000** iterations với số node trong lớp hidden là **256**.

```
Trained for 40 epoch(s), loss = 0.05860740894576413, test = 0.9736  
Training finished!
```

Hình 13: Độ chính xác của mô hình

Vì sau khi train, độ chính xác không tuyệt đối, nên có trường hợp sai. Do đó, em visualize 1 số trường hợp sai ra để quan sát.



Hình 14: Các trường hợp sai

Hình trên là một số trường hợp sai với title của mỗi hình có dạng  $(x, y)$  với  $x$  là true label,  $y$  là predict.

## 4.2. So sánh với độ chính xác của mô hình với một số trường hợp khác nhau (về số iteration, hidden size)

- So sánh các iteration khác nhau

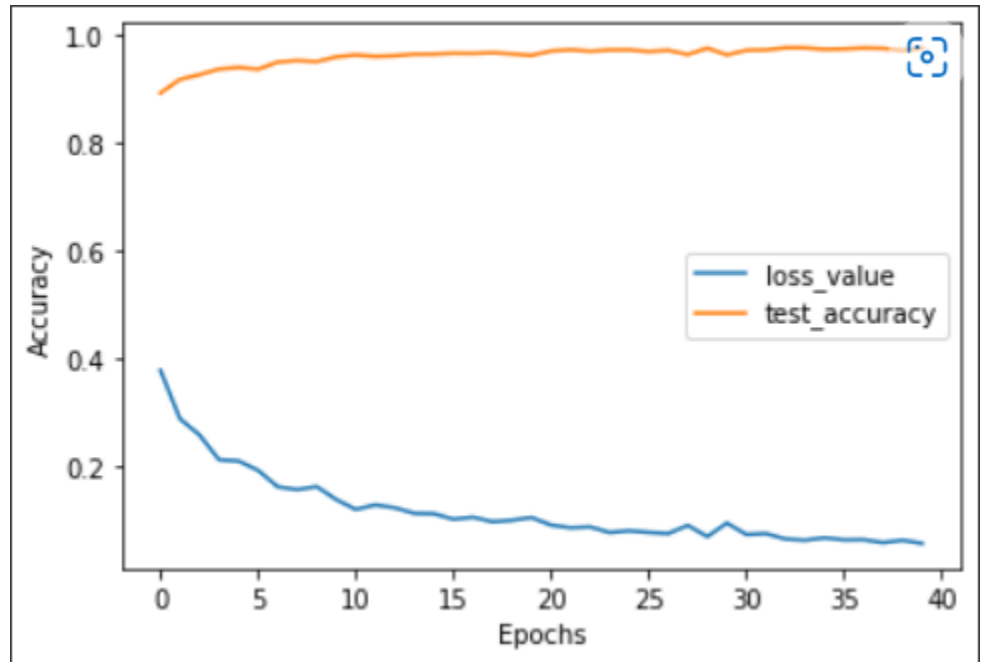
Để chọn được số iteration phù hợp để train, em đã thử nhiều giá trị khác nhau để chọn ra đáp án phù hợp.

Sau đây là kết quả biểu diễn accuracy và giá trị loss của bài toán ở các số lần thử **10000, 20000, 100000 và 200000**.

```
iteration = 10000: loss = 0.288, accuracy = 0.917%
iteration = 20000: loss = 0.212, accuracy = 0.936%
iteration = 100000: loss = 0.105, accuracy = 0.961%
iteration = 200000: loss = 0.057, accuracy = 0.975%
```

Hình 15: Loss và accuracy ở một số iteration

Qua đó, ta thấy rằng, khi cho model chạy qua **200000 iterations**, model cho kết quả khá tốt. Để biểu diễn rõ hơn loss và accuracy, ta visualize kết quả qua các epochs.



Hình 16: Visualize kết quả qua các epochs

- So sánh số node khác nhau trong hidden layer

Giống như ở trên, em cũng thử qua một số số node khác nhau trong hidden layer **32, 64, 128, 256, 512** em thấy rằng độ chính xác của mô hình từ khoảng 256 node không ảnh hưởng quá nhiều đến kết quả.

```
num_iterations = 200000
learning_rate = 0.01
num_inputs = 28*28
num_outputs = 10
hidden_size = [32, 64, 128, 256, 512]
acc_per_hidden_size = []
# data fitting, training and accuracy evaluation
for i in hidden_size:
    model = NN(num_inputs, i, num_outputs)
    cost_dict, tests_dict = model.train(x_train, y_train,
num_iterations=num_iterations, learning_rate=learning_rate)
    tmp, _ = model.testing(x_test, y_test)
    acc_per_hidden_size.append(tmp)
```

```
for i in range(len(hidden_size)):
    print("Accuracy for %d hidden size: %.3f" % (hidden_size[i], acc_per_hidden_size[i]))

Accuracy for 32 hidden size: 0.962
Accuracy for 64 hidden size: 0.968
Accuracy for 128 hidden size: 0.972
Accuracy for 256 hidden size: 0.974
Accuracy for 512 hidden size: 0.974
```

Hình 17: Accuracy cho số node khác nhau ở hidden size

Từ kết quả trên, em chọn số node trong hidden là 256.

## 5. Ứng dụng của mạng neural

Mạng neural có khá nhiều ứng dụng, ở đây em xin phép trình bày một số ứng dụng phổ biến của mạng neural.

- Nhận diện chữ số viết tay
- Dự đoán giá nhà
- Nhận diện khuôn mặt
- Dự đoán thị trường chứng khoán
- Chăm sóc sức khỏe
- ...

## III. Kết luận

Bài báo cáo trên cùng với file colab (ở link github đính kèm bên dưới) đã trình bày về mạng neural và áp dụng mạng neural lên một bài toán cụ thể (Nhận diện chữ số viết tay trên bộ dữ liệu MNIST).

Qua đó, ta thấy được các ưu, nhược điểm của mạng neural 3 lớp

- Ưu điểm

- Dễ dàng thấy, mạng neural có thể thực hiện các tác vụ mà một chương trình tuyến tính không thể thực hiện được (cụ thể ở đây là việc phân loại ảnh).
- Có nhiều ứng dụng trong thực tế như phân loại ảnh, nhận diện,...
- Dễ code, đơn giản.
- Có thể tự điều chỉnh các tham số để tăng độ chính xác của model.
- Nhược điểm
  - Cần phải có dữ liệu khá khá thì quá trình “học “ mới hiệu quả.
  - Với các mạng neural có nhiều node trong lớp ẩn, nó tốn rất nhiều thời gian.

Mọi file liên quan đến bài báo cáo: [trthminh/NeuralNetwork \(github.com\)](https://github.com/trthminh/NeuralNetwork)

## IV. Tài liệu tham khảo

[Machine Learning cơ bản \(machinelearningcoban.com\)](http://machinelearningcoban.com)

[MNIST database - Wikipedia](https://en.wikipedia.org/wiki/MNIST_database)

[8 Applications of Neural Networks / Analytics Steps](#)