TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC MÁY TÍNH

BÁO CÁO ĐỒ ÁN MÁY HỌC NÂNG CAO

ĐỀ TÀI: NHẬN DIỆN CHỮ SỐ VỚI MẠNG NEURON NHÂN TẠO

Lớp: CS420.H11

GVHD: ThS. Hồ Long Vân

Nhóm thực hiện: Nightingale

- Trương Hoàng Lân – 11520201
- Trương Văn Lộc – 13520460
- Nguyễn Lê Minh Quí – 13520679

TP.Hồ Chí Minh, tháng 12 năm 2016

MỤC LỤC

LỜI NÓI ĐẦU

Mạng neuron nhân tạo ra đời từ những năm 1943, tuy nhiên vào thời điểm đó kiến trúc von Neumann vẫn chiếm ưu thế trong ngành khoa học máy tính nên việc nghiên cứu mạng neuron không được chú trọng đến. Thêm vào đó là hạn chế về mặt kỹ thuật, các máy tính lúc này vẫn còn rất đơn sơ, hầu hết chỉ là những bảng mạch điện.

Ý tưởng về một chiếc máy tính có thể tự lập trình chính nó đã thu hút rất nhiều nhà khoa học lao vào nghiên cứu, và xuất hiện trong không ít các bộ phim khoa học viễn tưởng của Hollywood.

Trải qua nhiều giai đoạn phát triển của khoa học máy tính và trí tuệ nhân tạo, ngày nay mạng neuron nhân tạo ngày càng tiến gần hơn tới ước mơ của con người: một cỗ máy có trí thông minh nhân tạo, nhận thức được thế giới xung quanh, vượt xa con người về khả năng học hỏi, tư duy, tính toán, ….

Hiện tại việc nghiên cứu mạng neuron hầu hết đều tập trung tại Google, thành công gần đây nhất của AI đó là việc trí tuệ nhân tạo có tên Alphago đánh bại một đại kiện tướng cờ vây hàng đầu thế giới. Thành công này đánh một dấu mốc lịch sự quan trọng trong việc nghiên cứu phát triển trí tuệ nhân tạo nói chung và mạng neuron nói riêng.

Trong phạm vi giới hạn của khóa học, chúng em quyết định chọn đề tài nhận diện chữ số (digits recognition), chúng em chọn đề tài này vì cảm thấy đây là một trong những ứng dụng phổ biến và rất thú vị của mạng neuron.

LỜI CẢM ƠN

Chúng em xin chân thành cảm ơn thầy Hồ Long Vân đã tận tình giảng dạy, giúp đỡ, chỉ bảo chúng em. Những kiến thức thầy truyền đạt đã giúp đỡ chúng em rất nhiều trong việc hoàn thành đồ án này.

NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

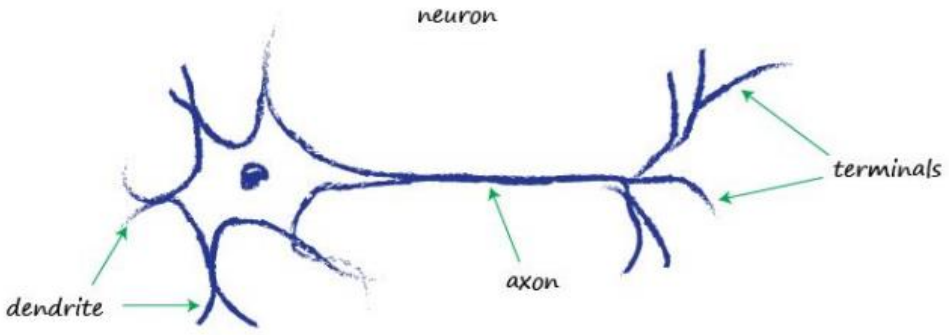
……, ngày …… tháng …… năm ……

Người nhận xét

1. Chương I: Tổng quan về mạng neuron

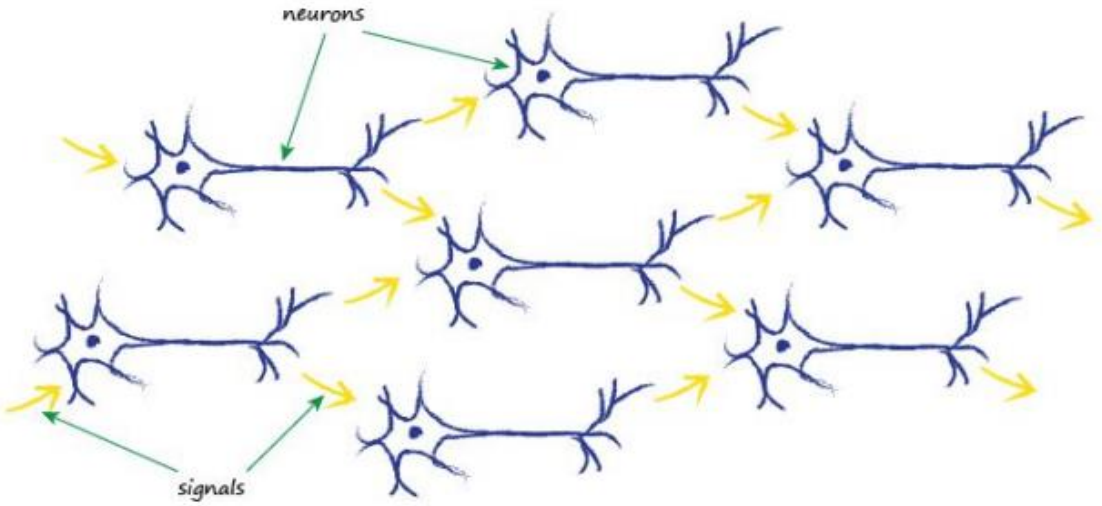
1.1. Giới thiệu về mạng neuron

Mạng neuron nhân tạo là một phương pháp tính toán dựa trên một tập hợp nhiều đơn vị (neuron) liên kết với nhau, mô phỏng việc giải quyết vấn đề của một bộ não sinh học.

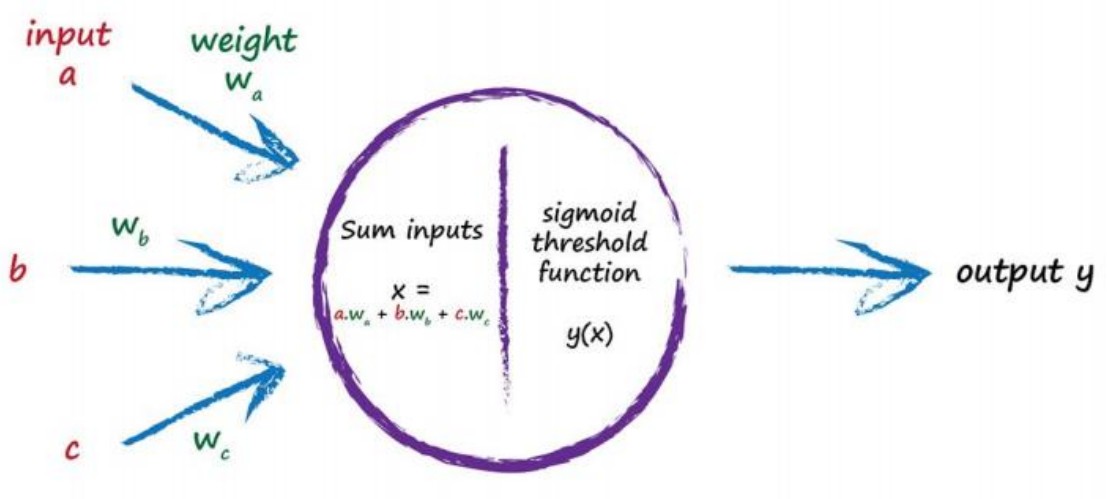


Cấu trúc của một neuron sinh học

Mỗi đơn vị neuron được liên kết bởi nhiều đơn vị neuron khác và ngược lại, và thường đi kèm với một hàm tính tổng các giá trị mà nó nhận từ các đơn vị neuron khác.



Các neuron liên kết với nhau

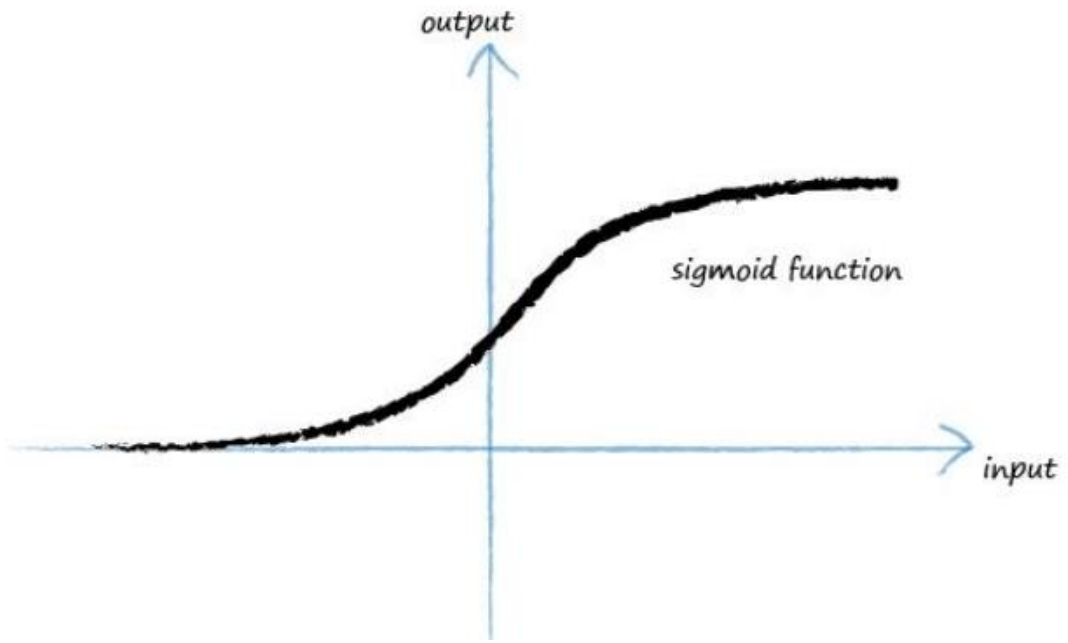


Các tín hiệu input của một neuron được tính tổng và sau đó đưa vào hàm sigmoid

Có nhiều hàm sigmoid, nhưng ta thường dùng :

$$y = \frac{1}{1 + e^{-x}}$$

Hàm này có đồ thị như sau:



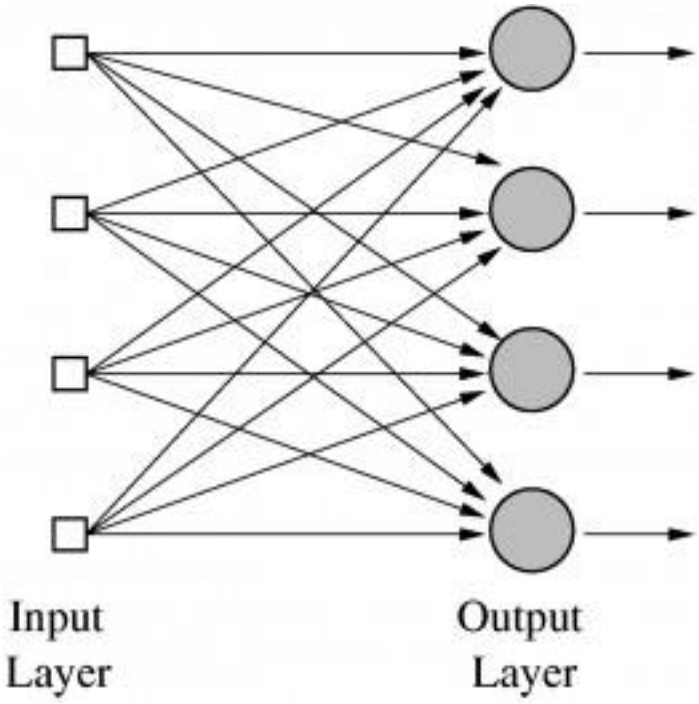
Với y nằm trong khoảng từ [0, 1]

Ở giữa mỗi liên kết của các neuron thường tồn tại một hàm trọng số (threshold), tín hiệu muốn truyền đi được từ neuron này sang neuron khác phải có giá trị lớn hơn hàm trọng số.

Mạng neuron nhân tạo tự học và tự huấn luyện chính nó. Khác với những chương trình được lập trình sẵn , mạng neuron có thể giải quyết được các vấn đề rất phức tạp mà các phương pháp lập trình truyền thống khó lòng có thể giải quyết được như nhận dạng, phân loại, v.v.

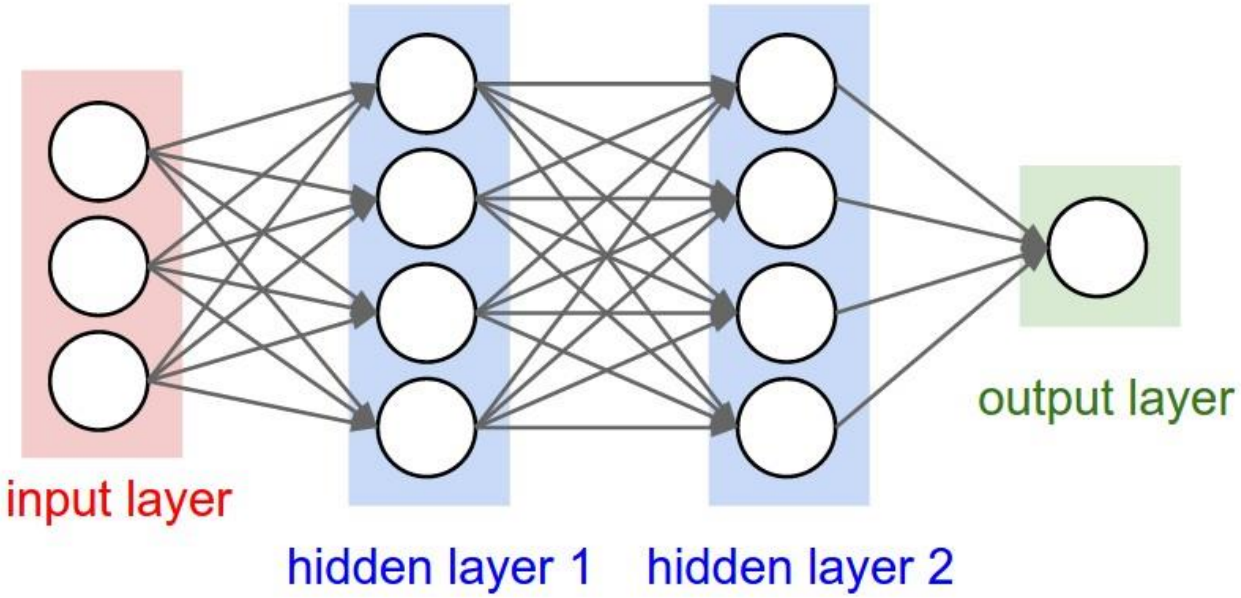
Mục đích của một mạng neuron nhân tạo là giải quyết các vấn đề tương tự với não người. Một mạng neuron thường bao gồm từ hàng ngàn tới hàng triệu các neuron và liên kết giữa các neuron, tuy nhiên con số này vẫn còn rất nhỏ bé nếu so với não người, và khả năng tính toán thậm chí còn chưa so được với bộ não của một con sâu.

Một mạng neuron nhân tạo gồm ít nhất 2 lớp, lớp input (input layer) và lớp output.



Một mạng neuron nhân tạo 2 lớp

Tuy nhiên, 2 lớp không phải là cấu trúc thường có của một mạng neuron. Một mạng neuron thường có từ 3 lớp trở lên, 1 lớp input, 1 lớp output, và 1 hay nhiều lớp ẩn (hidden layer).

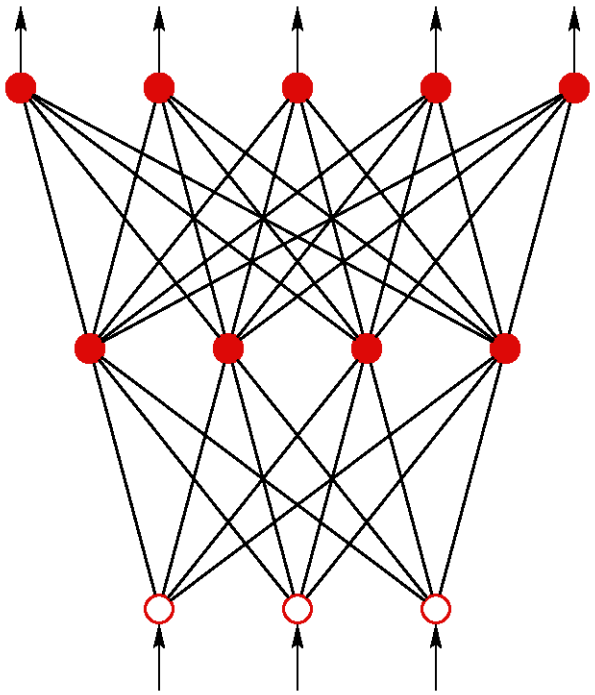


1.2. Một số cấu trúc phổ biến của mạng neuron nhân tạo

1.2.1. Mạng neuron Feedforward (FNN)

Một mạng neuron feedforward (Feedforward Neural Network) là một thuật toán phân lớp. Thường gồm một lượng lớn các đơn vị xử lý đóng vai trò như một neuron, được sắp xếp thành từng lớp. Mỗi đơn vị trong một lớp được liên kết với tất các các đơn vị ở lớp kế tiếp. Những liên kết này không giống nhau: mỗi liên có độ mạnh (strength) hay giá trị (weight) khác nhau. Các giá trị (weights) trên những liên kết này hình thành nên tri thức của mạng neuron. Các đơn vị trong một mạng neuron còn có thể được gọi là node.

Dữ liệu đầu vào được truyền vào mạng, đi qua từng lớp, và sau đó xuất ra các giá trị output. Không có sự truyền ngược giữa các lớp. Đây là lý do mà người ta gọi mạng neuron này là mạng neuron feedforward.



Một mạng neuron feedforward gồm 3 lớp, lớp input có 3 nodes, lớp ẩn có 4 nodes và lớp output có 5 nodes.

Trong đó lớp input chỉ đóng vai trò biểu diễn dữ liệu đầu vào, không được dùng trong tính toán và không biểu thị tri thức của một mạng neuron feedforward.

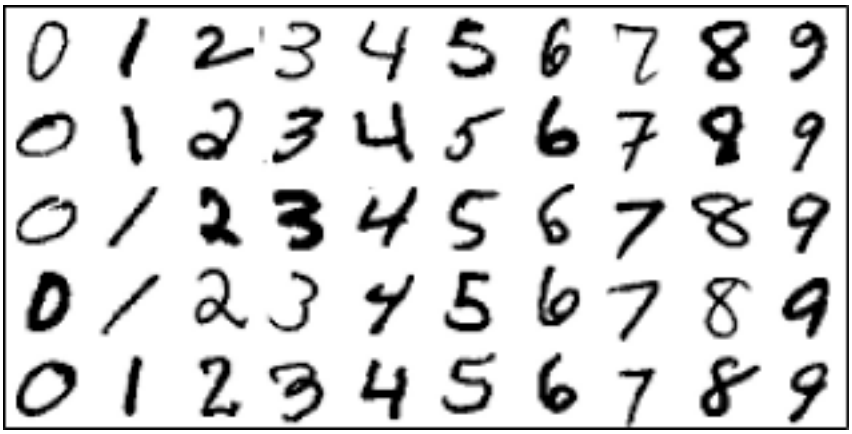
- Ứng dụng:

Mạng neuron feedforward thường được dùng để phân loại các khuôn mẫu (pattern).

Ví dụ:

+ Nhận dạng biển báo giao thông:

+ Nhận diện chữ số:



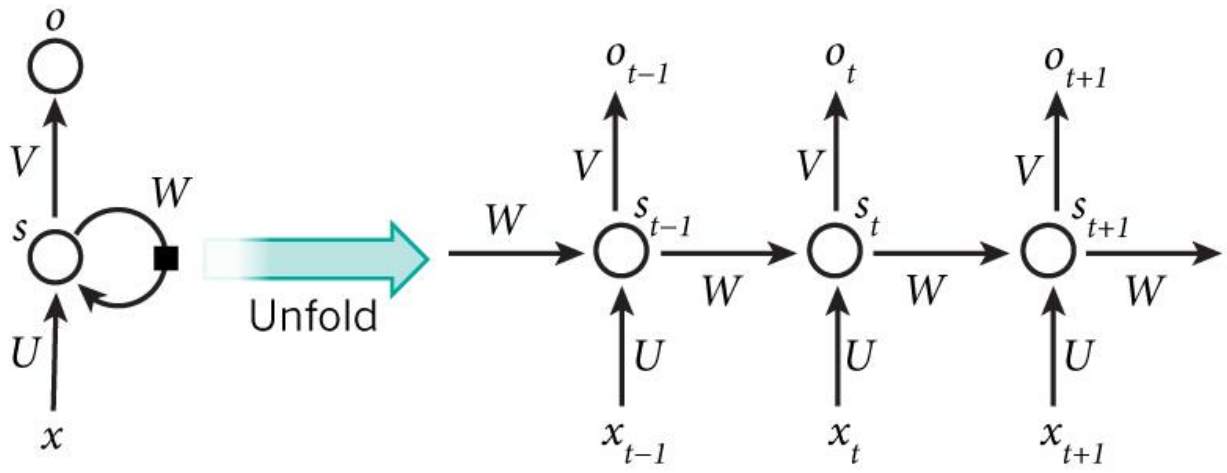
- Huấn luyện một mạng neuron feedforward:

Mạng neuron feedforward sử dụng thuật toán học có giám sát (supervised learning algorithm). Các khuôn mẫu (pattern) được biểu diễn dưới dạng input, sau đó được truyền qua các layer bên trong mạng cho tới khi tới lớp output. Sau đó các giá trị này được so sánh với giá trị thật mong muốn. Nếu như giá trị output khớp với giá trị mong muốn thì các trọng số giữa các liên kết của các nodes được giữ nguyên. Ngược lại, một (hoặc nhiều) giá trị lỗi sẽ được truyền ngược lại (back propagate) vào trong mạng. Dựa vào các giá trị này, giá trị liên kết (độ mạnh liên kết) giữa các nodes sẽ được cập nhật tương ứng. Cũng vì vậy mà mạng feedforward còn có tên gọi là mạng backpropagation.

1.2.2. Mạng neuron truy hồi (RNN)

Đối với mạng neuron thông thường, giá trị của các input và output thường không phụ thuộc vào nhau. Tuy nhiên điều này không phải lúc nào cũng có lợi. Ví dụ nếu muốn dự đoán mộ từ sẽ xuất hiện tiếp theo trong một câu thì ta phải biết các từ xuất hiện trước đó. Loại này được gọi là mạng neuron truy hồi (Recurrent Neural Network – RNN) vì với một chuỗi các input bất kì, các input này đều được xử lý như nhau, và giá trị output phụ thuộc vào output được tính trước đó. Có thể nói mạng neuron truy hồi có "trí nhớ" về những gì nó đã tính toán trước đó. Về lý thuyết, mạng neuron truy hồi có thể nhớ được một chuỗi các bước tính có độ dài bất kỳ. Tuy nhiên trong thực tế mạng neuron truy hồi chỉ có thể nhớ được một vài bước tính toán trước đó.

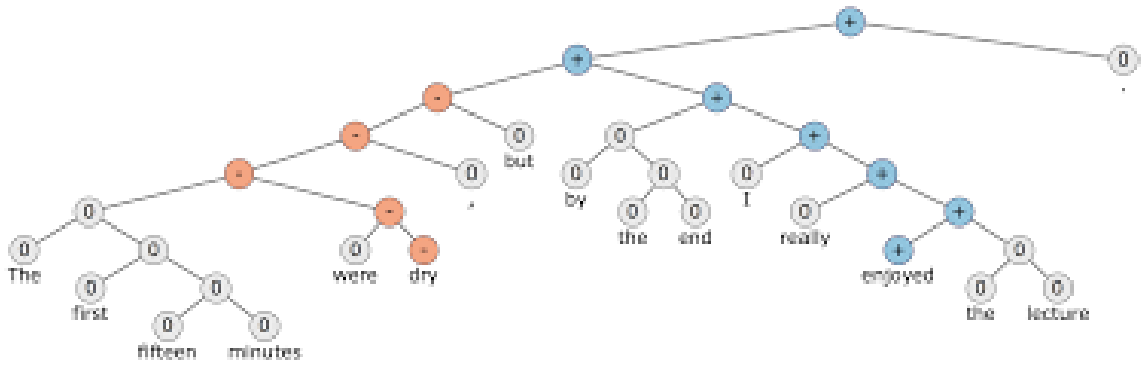
Cấu trúc phổ biến của mạng neuron truy hồi:



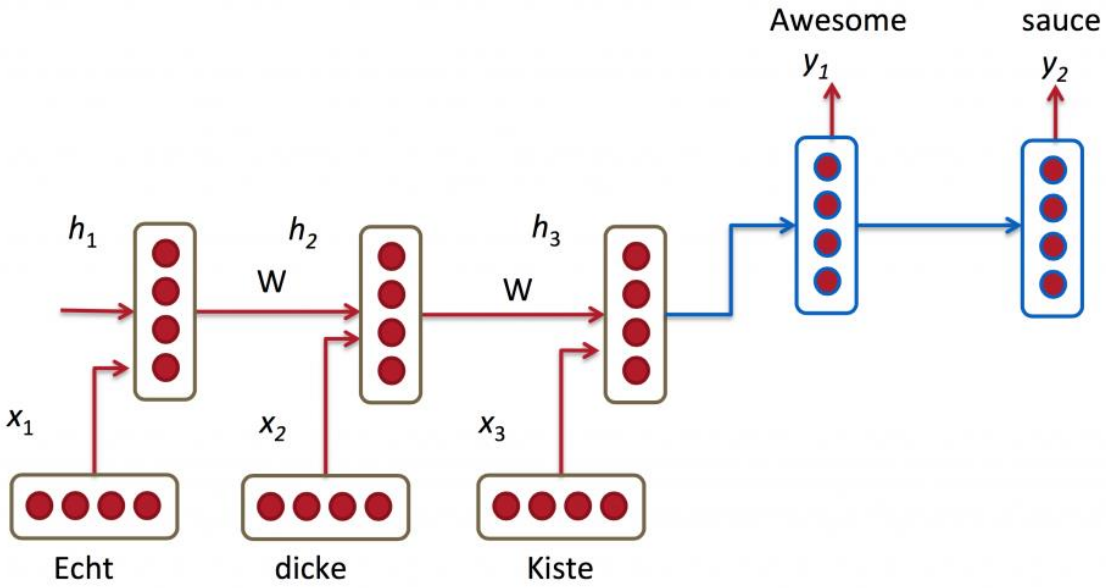
Ví dụ trên mô phỏng một mạng neuron truy hồi được unfold (hoặc unrolled) thành một mạng neuron hoàn chỉnh. Nghĩa là viết ra mạng neuron cho cả một câu. Ví dụ, nếu input nhập vào là một cầu gồm 5 từ, thì mạng neuron truy hồi sẽ được unrolled (hoặc unfold) thành 5 lớp, mỗi lớp tương ứng với một từ.

- Ứng dụng:

+ Xử lý ngôn ngữ tự nhiên (Natural Language Processing – NLP):

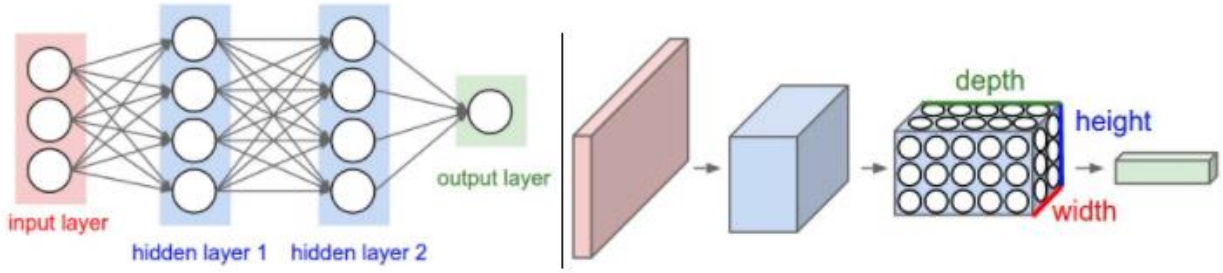


+ Dịch máy (Machine Translation):



1.2.3. Mạng neuron tích chập (CNN)

Mạng neuron tích chập (Convolutional Neural Network – CNN) cũng giống với các mạng neuron thông thường (như mạng feedforward), được cấu tạo với các đơn vị đóng vai trò như các neuron và có khả năng học dựa vào giá trị liên kết. Điểm khác biệt chính là mạng neuron tích chập được tạo nên trong không gian ba chiều (3D) gồm: cao, rộng, sâu.



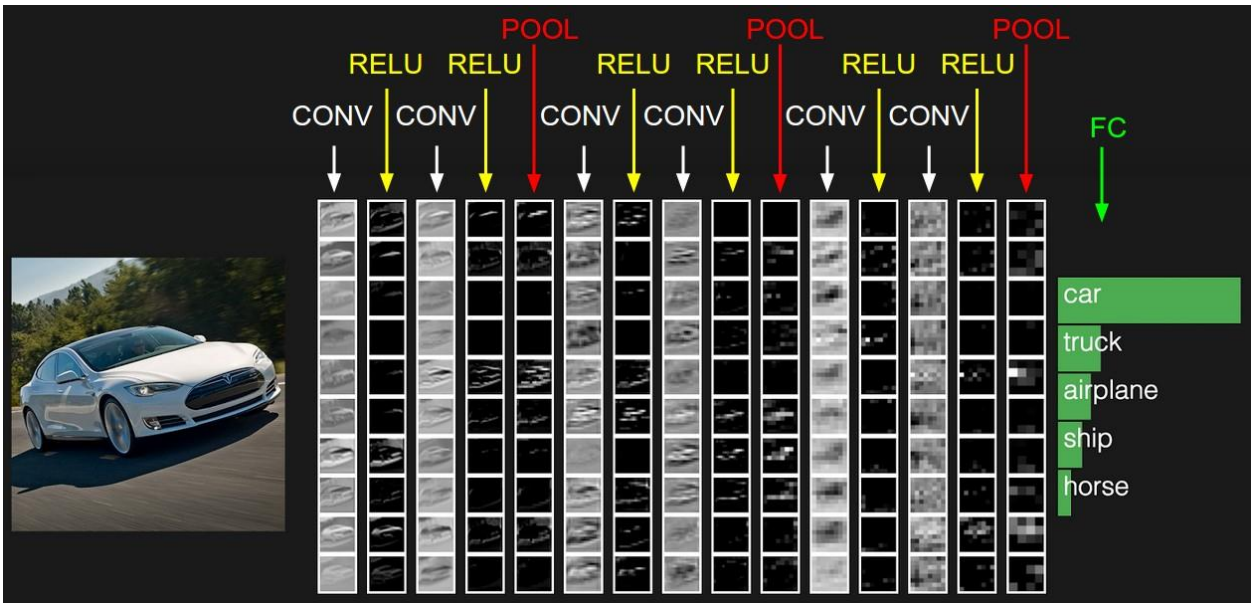
Một mạng neuron thông thường (trái), và mạng neuron tích chập (phải).

Mạng neuron tích chập gồm nhiều lớp, mỗi lớp chuyển 1 lượng kích hoạt (volume of activations) cho lớp khác thông qua một hàm d (differentiable function). Có 3 loại lớp chính được dùng trong mạng neuron tích chập: lớp tích chập, lớp pooling và lớp fully-connected (thường thấy trong các mạng neuron thông thường).

- Lớp tích chập: là lớp chính trong mạng neuron tích chập, hầu hết mọi tính toán đều được thực hiện ở lớp này.
- Lớp pooling: thường được thêm vào giữa 2 lớp tích chập nối tiếp nhau, nhằm giảm kích thước của các tham số, từ đó giảm bớt các tính toán cho các lớp tích chập.
- Lớp fully-connected: kết nối với tất các các kích hoạt (activations) của layer trước nó (thường thấy ở một mạng neural thông thường). Các kích hoạt được tính toán dựa bằng cách nhân ma trận với một giá trị bias (bias offset).

- Ứng dụng:

+ Nhận diện ảnh hoặc video



2. Chương II: Đề tài nhận diện chữ số

2.1. Lý do chọn đề tài

Nhóm em chọn đề tài nhận diễn chữ số vì nhiều lý do

- Qua tìm hiểu nhóm em cảm thấy nhận diện chữ số là một bài toán khá hay trong ứng dụng mạng neuron.
- Nhóm quyết định sử dụng kiến trúc mạng neuron feedforward vì tính đơn giản, dễ thực hiện, và mạng lại độ chính xác cao, và vì ứng dụng của mạng neuron feedforward phù hợp với yêu cầu của đề tài.
- Một lý do khác là datasets có sẵn rất nhiều.

2.2. Mục tiêu đặt ra

- Hoàn thành một chương trình có thể chạy và đưa ra kết quả nhận diện chữ số với độ chính xác cao.
- Hiểu rõ cách thức hoạt động của một mạng neuron nhân tạo.

2.3. Phương pháp thực hiện

- Nghiên cứu, đọc các bài báo khoa học cũng như các tài liệu hướng dẫn có liên quan đến các phương pháp, hướng tiếp cận bài toán nhận diện, mà cụ thể là nhận diện chữ số.
- Chọn ngôn ngữ lập trình và các thư viện / framework hỗ trợ tốt cho bài toán này để thực hiện.

3. Chương III: Chi tiết các bước thực hiện

Ta có thể chia các bước học của mạng feedforward thành 3 bước:

Bước 1: Nhận các input, feedforward chúng qua mạng neuron và xuất ra các output.

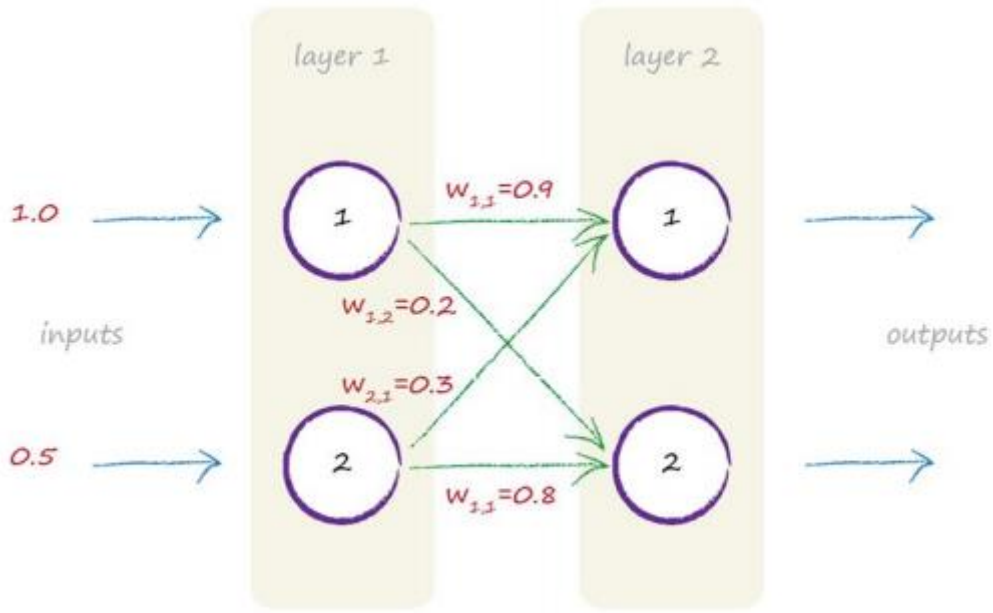
Bước 2: So sánh output với giá trị mong muốn, tính toán lỗi (nếu có), sau đó lan truyền ngược lỗi lại vào mạng.

Bước 3: Cập nhật giá trị của các liên kết giữa các node.

3.1. Bước 1

Giả sử ta có 1 mạng neuron vơi 2 lớp, lớp 1 gồm 2 node 1 và 2, tương tự với lớp 2.

Liên kết giữa các node ở lớp 1 và lớp 2 có thể được mô tả như sau:

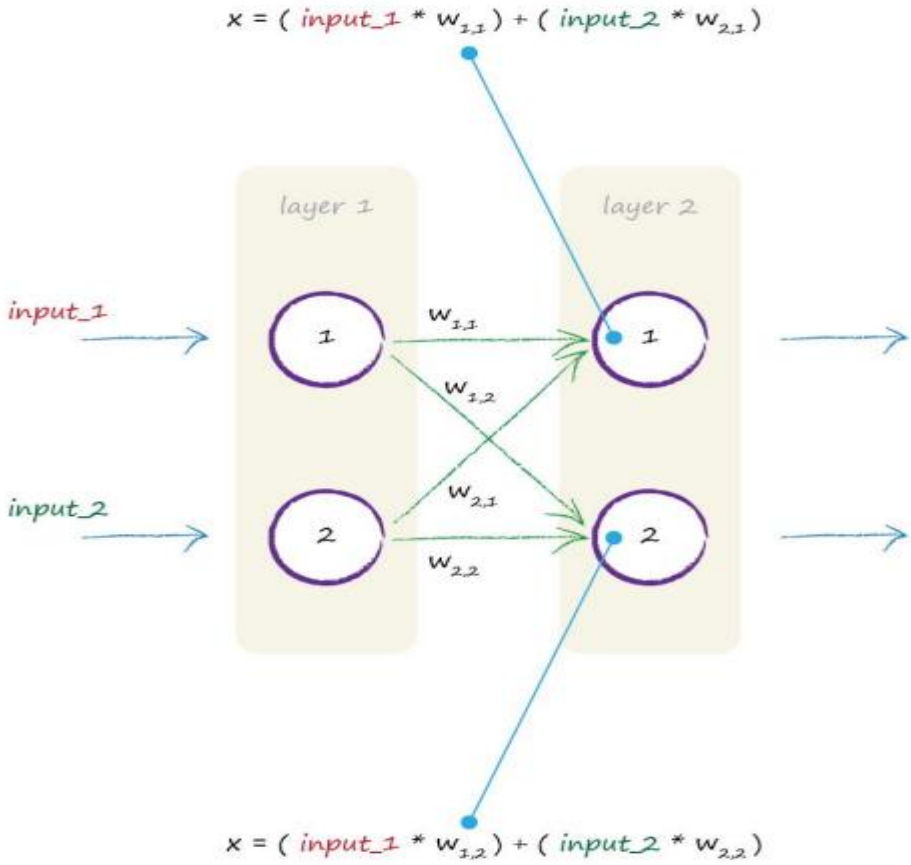


Trong đó $w_{1,1}$ là liên kết giữa node 1 ở lớp 1 với node 1 ở lớp 2, tương tự cho các liên kết còn lại. Và $i_1 = 1.0, i_2 = 0.5$ là ứng với 2 inputs .

Gọi x_1, x_2 là các output của node 1 và 2 ở layer 2 thì x_1, x_2 sẽ được tính như sau.

$$x_1 = w_{1,1} * i_1 + w_{2,1} * i_2$$

$$x_2 = w_{1,2} * i_1 + w_{2,2} * i_2$$



Nếu ta xếp $w_{1,1}, w_{1,2}\ w_{2,1}, w_{2,2}$ và i_1, i_2 vào 2 ma trận $Matrix_{weights}$ và I, ta có:

$Matrix_{weights} = \begin{pmatrix} w_{1,1} & w_{2,1} \\ w_{1,2} & w_{2,2} \end{pmatrix}$ và $I = \begin{pmatrix} i_1 \\ i_2 \end{pmatrix}$

Ta có $Matrix_{weights} * I = \begin{pmatrix} w_{1,1} * i_1 + w_{2,1} * i_2 \\ w_{1,2} * i_1 + w_{2,2} * i_2 \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$

x_1, x_2 là 2 outputs của node 1 và 2 ở lớp 1 đồng thời là 2 inputs của node 1 và 2 ở lớp 2.

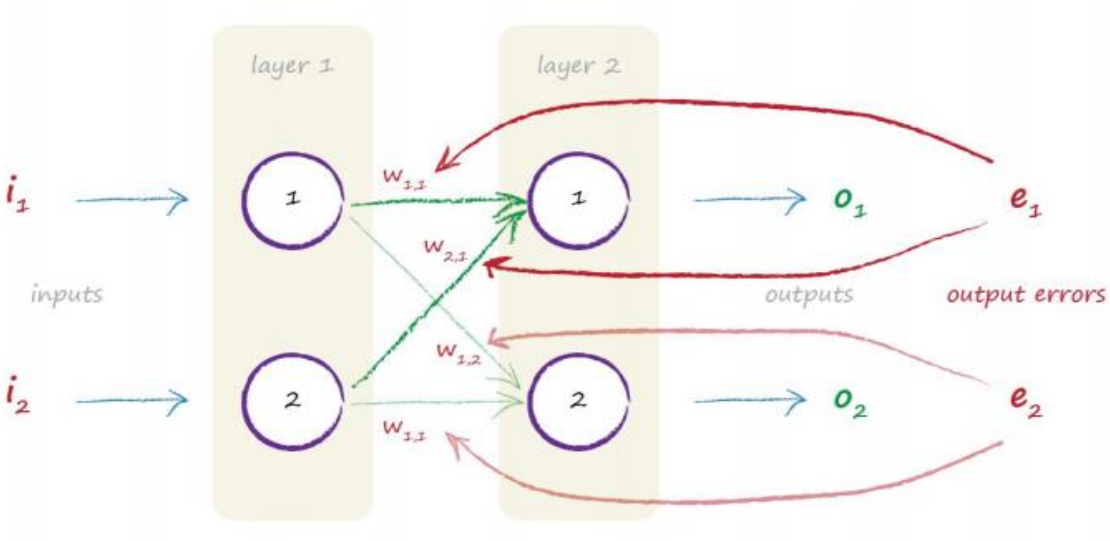
Từ đây ta có thể suy ra, để tính được outputs cho lớp tiếp theo:

$$Matrix_{output} = Matrix_{input} * Matrix_{weights}$$

Nếu có nhiều hơn 2 lớp, thì $Matrix_{output}$ của lớp phía trước là $Matrix_{input}$ của lớp phía sau .

3.2. Bước 2

Giá trị lỗi được tính dựa trên sự đóng góp của các giá trị liên kết.



Lỗi của các nodes ở lớp output có thể được tính trực tiếp bằng cách so sánh các outputs với các giá trị thực tế. Tuy nhiên ta không thể áp dụng cách tính như vậy cho các nodes ở lớp ẩn (hidden layer), vậy làm cách nào để tính giá trị lỗi cho các nodes ở lớp ẩn?

Dễ thấy, e_1 được tính đựa vào o_1, mà o_1 được tính dựa vào $w_{1,1}$ và $w_{2,1}$. Sự đóng góp nhiều hay ít dựa trên giá trị của $w_{1,1}$ và $w_{2,1}$.

+ Đối với e_1:

- Đóng góp của $w_{1,1} = \frac{w_{1,1}}{w_{1,1}+w_{2,1}}$
- Đóng góp của $w_{2,1} = \frac{w_{2,1}}{w_{1,1}+w_{2,1}}$

+ Đối với e_2:

- Đóng góp của $w_{1,2} = \frac{w_{1,2}}{w_{1,2}+w_{2,2}}$
- Đóng góp của $w_{2,2} = \frac{w_{2,2}}{w_{1,2}+w_{2,2}}$

Ta có:

$$E_{hidden} = \begin{matrix} \frac{w_{1,1}}{w_{1,1}+w_{2,1}} & \frac{w_{1,2}}{w_{1,2}+w_{2,2}} \\ \frac{w_{2,1}}{w_{1,1}+w_{2,1}} & w2,2w1,2 + w2,2 \end{matrix} * \begin{matrix} e_1 \\ e_2 \end{matrix}$$

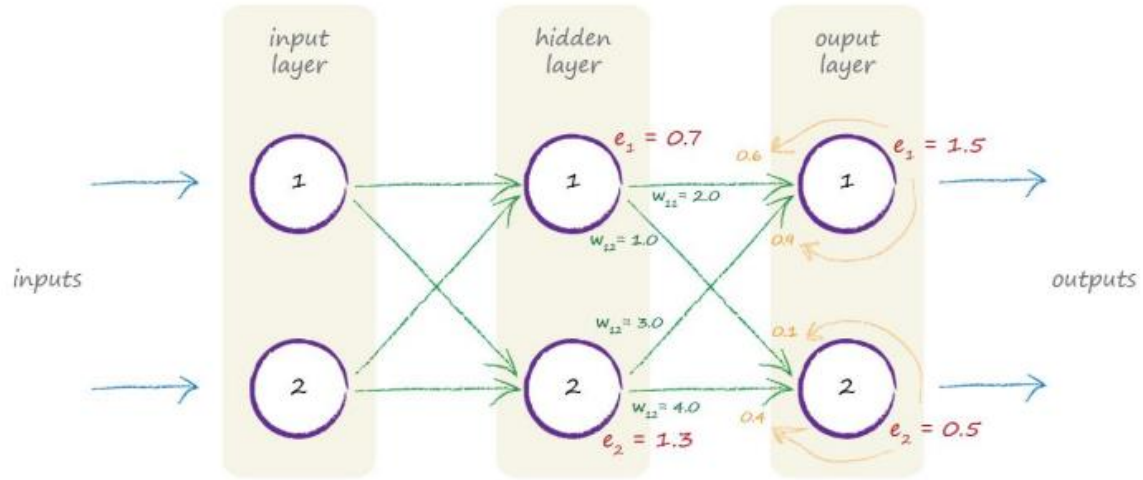
Làm gọn:

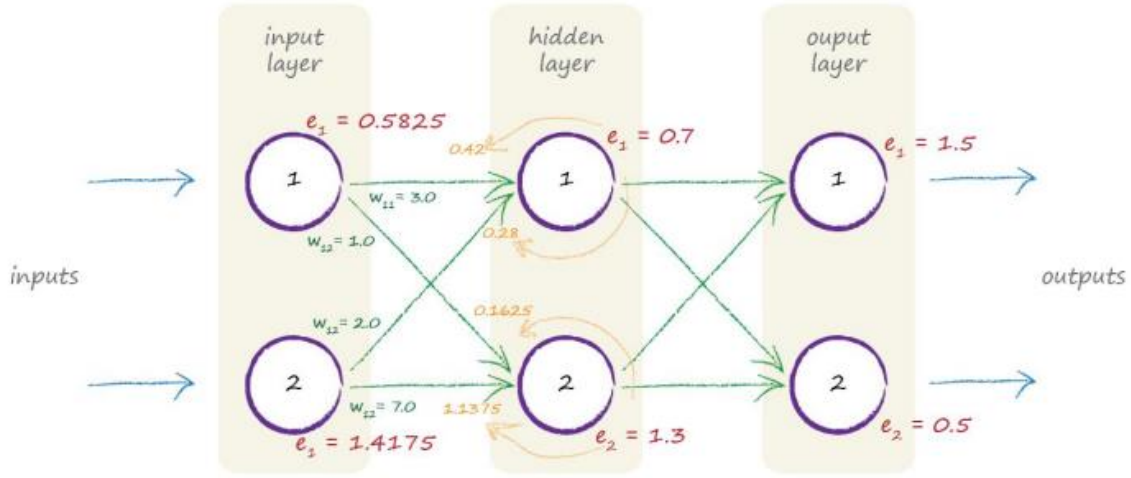
$$E_{hidden} = \begin{pmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{pmatrix} * \begin{pmatrix} e_1 \\ e_2 \end{pmatrix}$$

$$= Matrix_{weights\_hidden}{}^{T} * Matrix_{error\_output}$$

$$E_{hidden} = Matrix_{weights\_hidden}{}^{T} * Matrix_{error\_output}$$

Một số ví dụ:



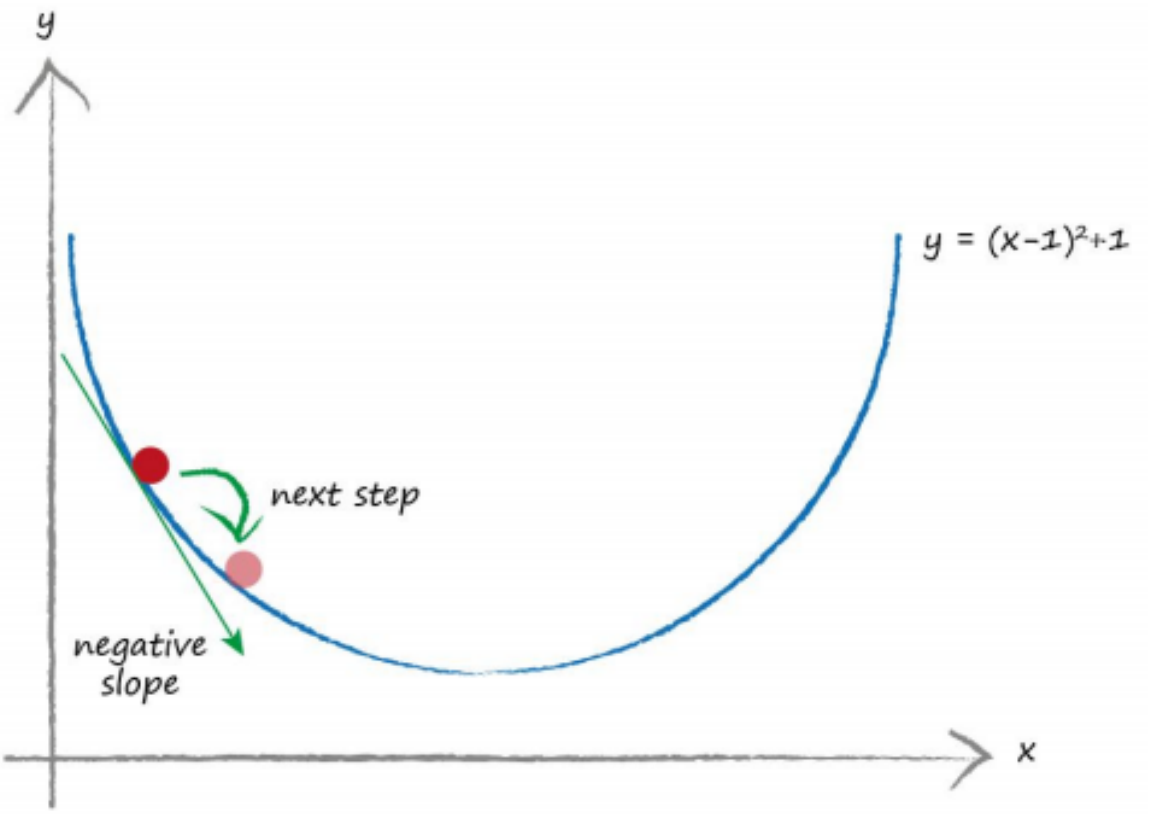


3.3. Bước 3

Gradient descent là phương pháp để tìm giá trị nhỏ nhất có thể (không nhất thiết phải là nhỏ nhất) của một hàm toán học phức tạp. Trong trường hợp này, ta muốn tối thiểu hoá (minimize) các giá trị lỗi, bản thân các giá trị lỗi được đóng góp bởi giá trị của các liên kết (ma trận liên kết). Nếu gọi E là hàm lỗi, thì hàm này phụ thuộc vào rất nhiều biến, không thể trực tiếp tìm min của hàm này bằng các phương pháp toán học thông thường.

Để dễ hiểu hơn, ta xét ví dụ sau:

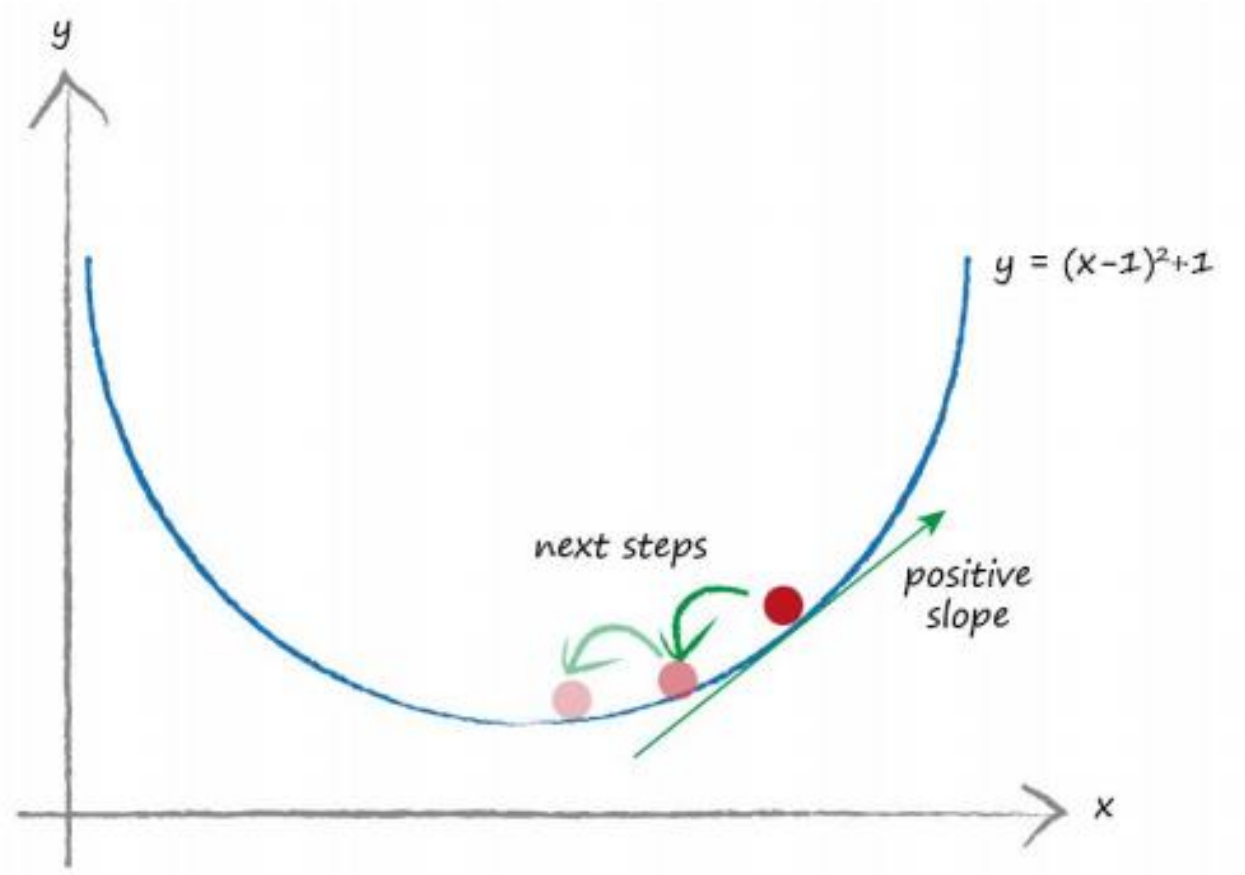
Hàm $y = (x-1)^2 + 1$ được biểu diễn t rong không gian 2 chiều như hình bên dưới



Giả sử y là hàm lỗi, ta muốn tìm x để y nhỏ có giá trị nhỏ nhất, và y là một hàm rất phức tạp không thể tìm được min bằng phương pháp toán học thông thường.

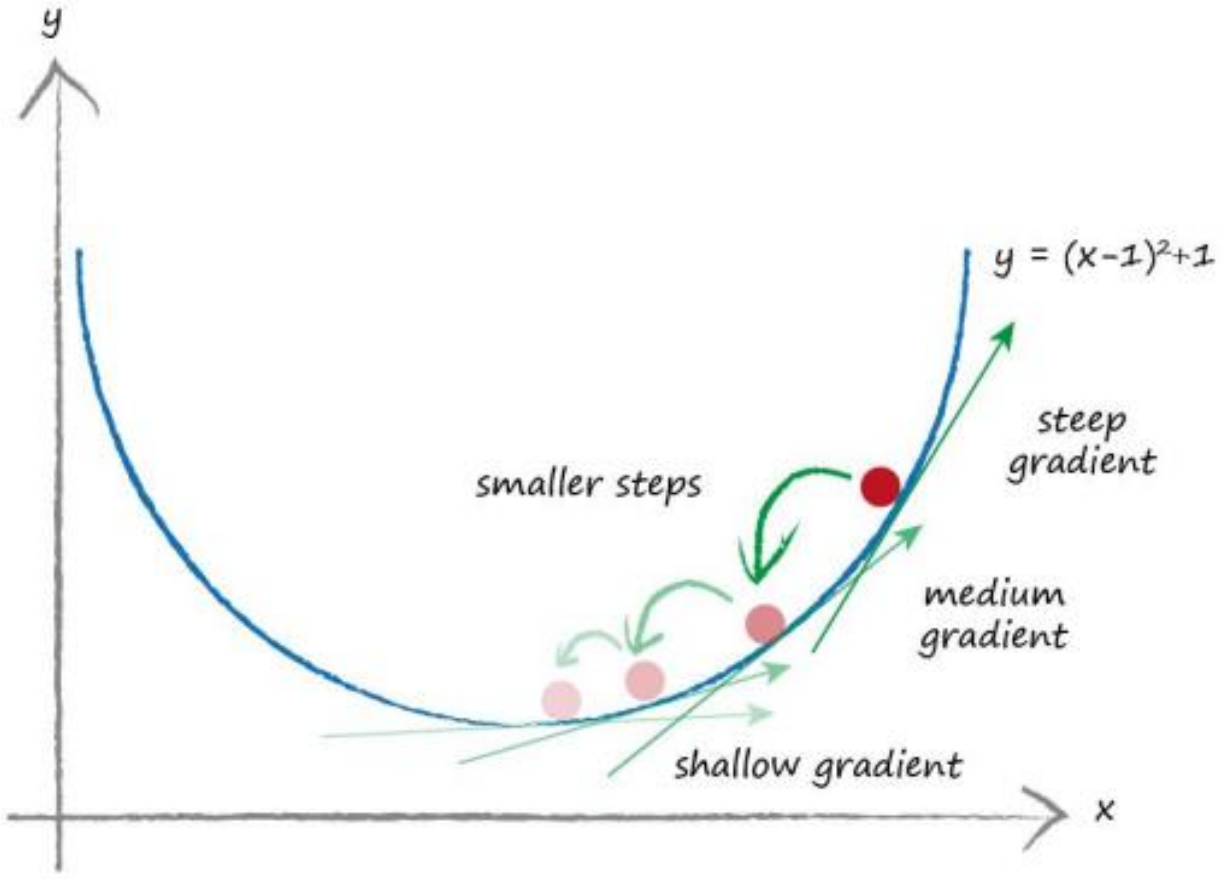
Đầu tiên, chọn một điểm bắt đầu (hình tròn màu đỏ). Giống như việc leo đồi, để leo xuống chân đồi, ta thường nhìn xung quanh nơi ta đang đứng để xem hướng nào có thể đi xuống. Vì ta đang đi xuống, nên độ dốc (slope) mang giá trị âm. Càng di chuyển (theo trục x) ta càng đến gần điểm min thật sự.

Lần này ta chọn một điểm bắt đầu khác, độ dốc mang giá trị dương (hướng lên) nên ta chọn hướng di chuyển về bên trái.

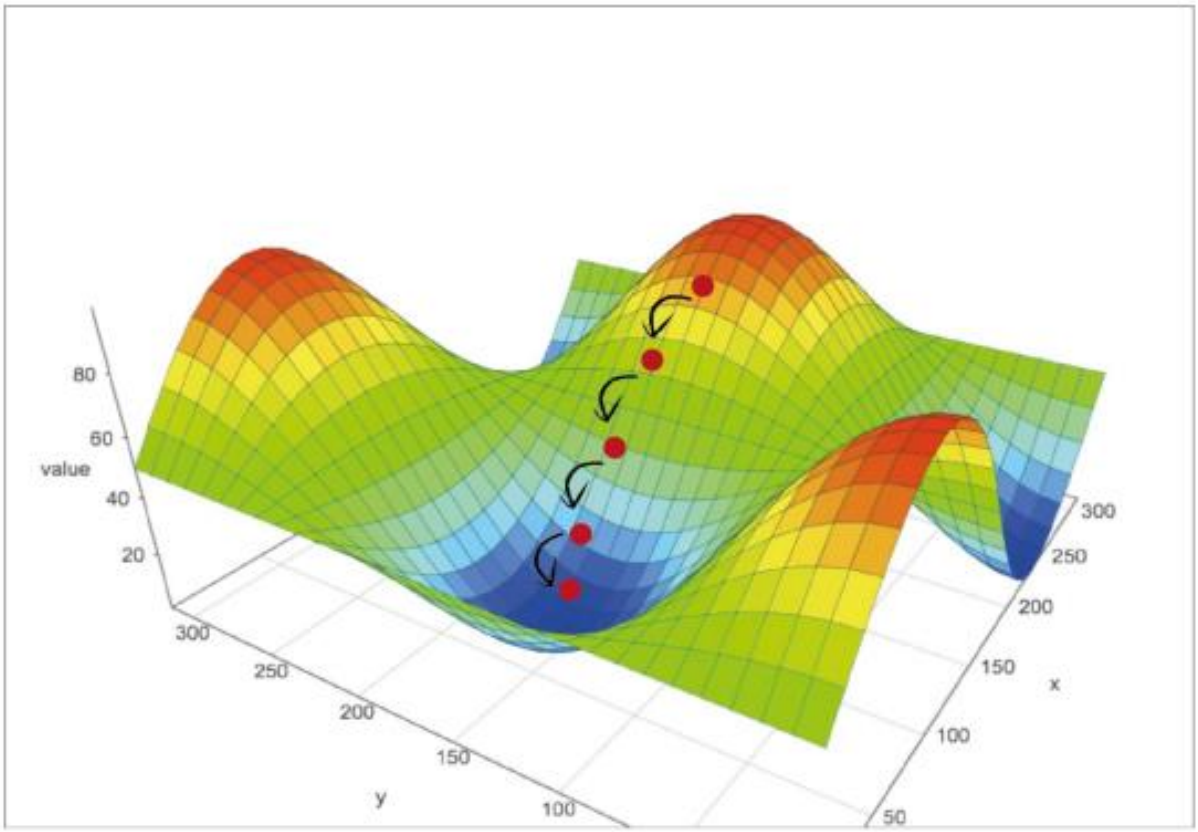


Có thể thấy, càng di chuyển, ta càng tiến gần tới điểm min thực sự. Tuy nhiên có thể xảy ra trường hợp ta đi quá điểm min thật sự (overstep). Lý do chính là do giá trị mà ta chọn cho mỗi step, nếu giá trị này quá lớn, thì có thể xảy ra overstep, còn giá trị quá nhỏ thì ta phải mất rất nhiều bước mới có thể tới được.

Ta có thể giải quyết vấn đề này bằng cách chọn giá trị cho mỗi bước dựa trên giá trị của độ dốc (slope hoặc gradient). Độ dốc càng nhỏ, thì giá trị mỗi bước càng nhỏ. (Vì trong thực tế khi ta đi xuống đồi, càng tới gần chân đồi, độ dốc sẽ càng nhỏ). Ý tưởng trên được mô phỏng như sau:



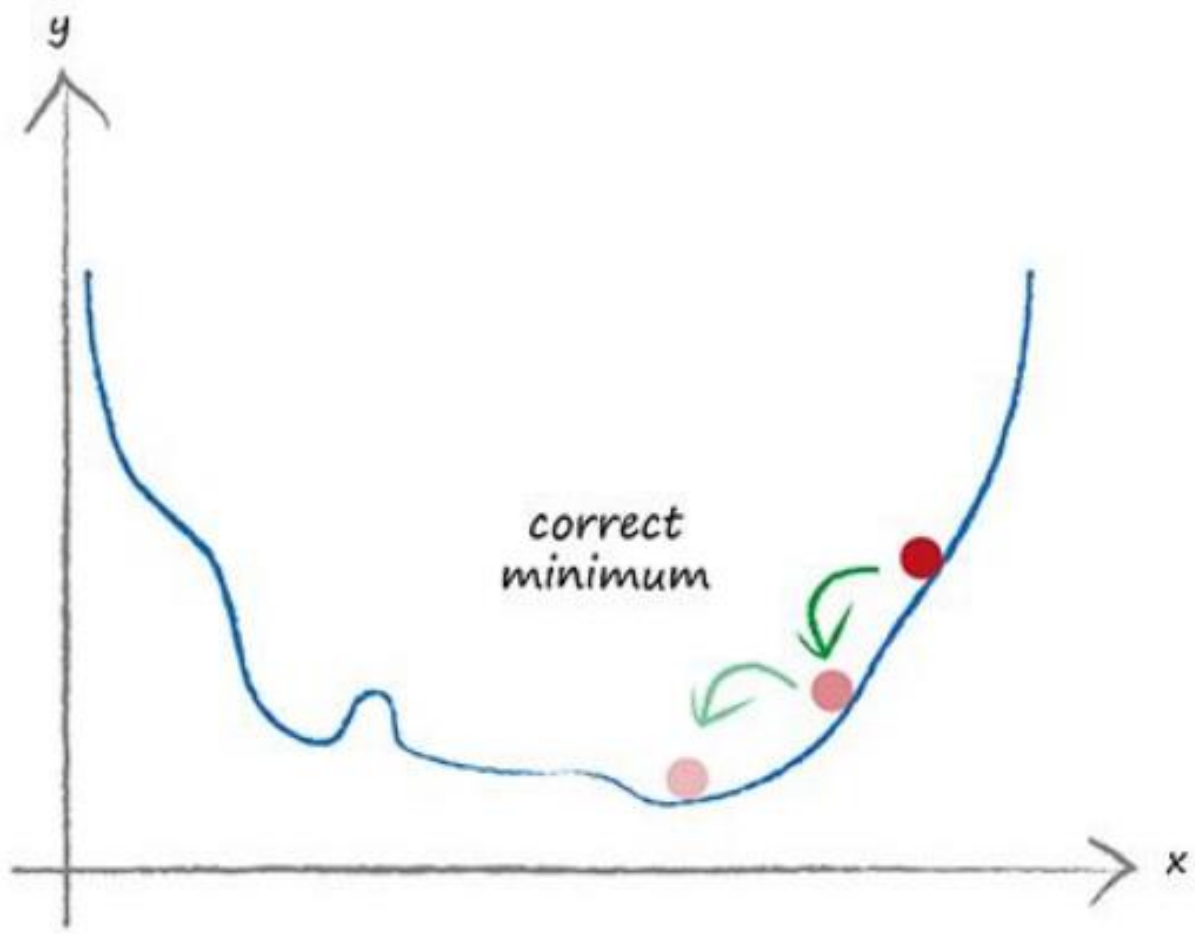
Phương pháp này không những chỉ có thể áp dụng cho hàm 1 biến, mà còn có thể áp dụng cho hàm nhiều biến (trong trường hợp mạng neuron là hàm lỗi). Một hàm phụ thuộc vào 2 biến, được mô phỏng trong không gian 3 chiều như hình sau:



Có thể thấy rằng thay vì chọn hướng đi được vẽ bên trên, ta có thể chọn hướng đi khác, tuy nhiên đích đến lại là một thung lũng (valley) khác. Vì xung quanh một ngọn đồi thường không chỉ có một hướng đi xuống, và có thể có nhiều điểm trũng (hay thung lũng) khác.

Điều này nói lên một nhược điểm của gradient descent, đó là kết quả thu được không phải lúc nào cũng là nhỏ nhất, hay nói cách khác là bị kẹt ở một thung lũng không phải có độ cao thấp nhất.

Để tránh việc chọn nhầm thung lũng, hay min của một hàm, ta huấn luyện mạng neuron nhiều lần, và mỗi lần đều bắt đầu tại những điểm khác nhau trên đồi để tránh bị kẹt ở một thung lũng không phải thấp nhất. Chọn những điểm bắt đầu chính là chọn giá trị ban đầu cho các liên kết giữa các lớp trong mạng neuron.



Điều này nói lên một nhược điểm của gradient descent, đó là kết quả thu được không phải lúc nào cũng là nhỏ nhất, hay nói cách khác là bị kẹt ở một thung lũng không phải có độ cao thấp nhất.

Output của một mạng neuron là một hàm rất phức tạp gồm nhiều biến số, ma trận liên kết. Vậy ta có thể dùng gradient descent để tìm giá trị đúng cho các liên kết? Câu trả lời là đúng nêu ta chọn đúng hàm lỗi (error function).

Lỗi được tính bằng cách so sánh outputs của mạng neuron và outputs mong muốn. Ta có các cách tính lỗi như sau:

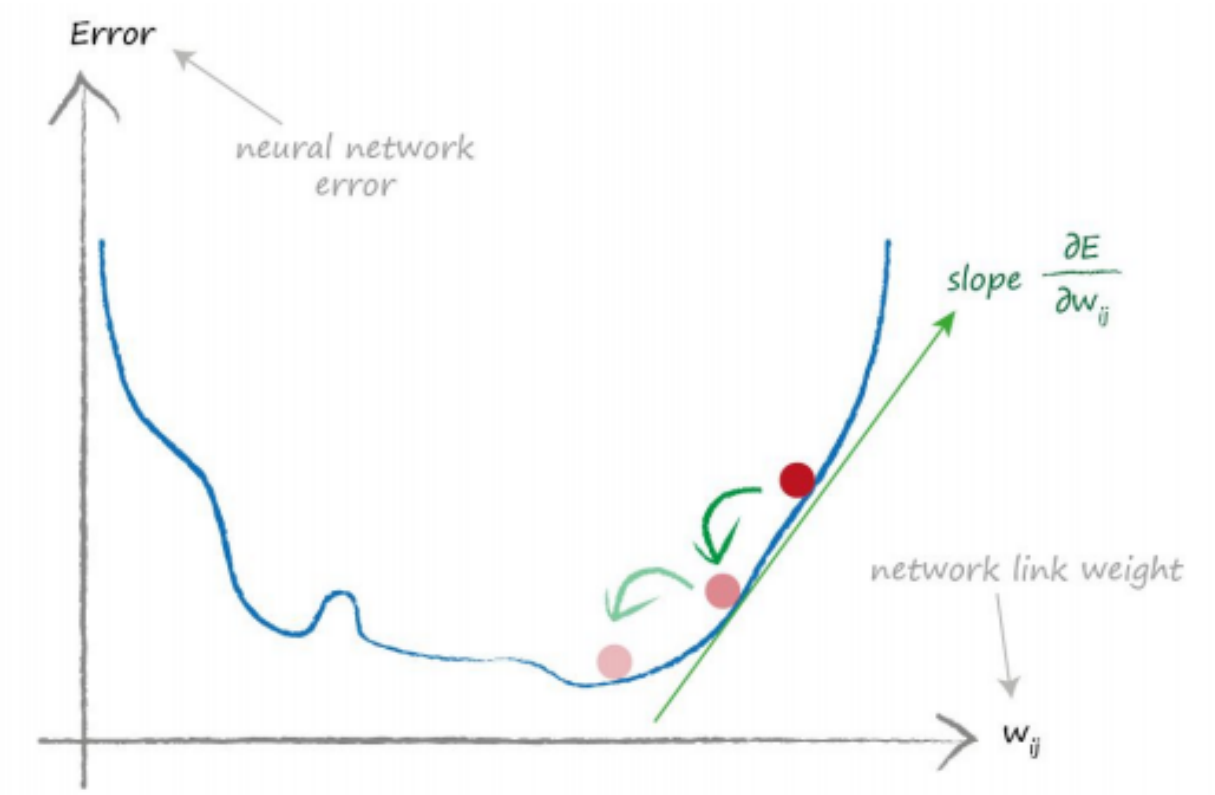
Network Output	Target Output	Error (target - actual)	Error |target - actual|	Error (target - actual)^2
0.4	0.5	0.1	0.1	0.01
0.8	0.7	-0.1	0.1	0.01
1.0	1.0	0	0	0
Sum		0	0.2	0.02

Cách đầu tiên (target – actual) nhìn có vẻ hợp lý, tuy nhiên khi tính tổng các nodes thì cho kết quả bằng 0. Khi tổng bằng 0 có nghĩa là không có lỗi, trong khi tồn tại 2 output không đúng với giá trị mong muốn.

Đối với cách thứ 2, giải quyết được vấn đề ở cách thứ nhất, cho ra kết quả tổng khác 0. Ta có thể chọn sử dụng cách này, tuy nhiên các giá trị lỗi không thể hiện được tính chất của gradient descent. Đó là 'giá trị của các bước và độ dốc giảm dần khi càng tiến gần về min".

Cách thứ 3, giá trị lỗi được tính bằng bình phương khác biệt giữa output và giá trị mong muốn. Cách này không cho ra kết quả âm, và giá trị thu được càng nhỏ khi sự khác biết giữa error và giá trị mong muốn càng nhỏ. Nên ta thường chọn cách thứ 3 là hàm lỗi.

Nếu xét y là lỗi, w_{ij} *là* x , thì độ dốc = $\frac{\partial E}{\partial w_{jk}}$ (đạo hàm của E theo w_{ij}).



Tóm lại, ta muốn tính:

$$\frac{\partial E}{\partial w_{jk}}$$

Biểu thức trên có nghĩa rằng, E thay đổi như thế nào khi w_{ij} thay đổi.

Hàm lỗi E là tổng bình phương khác biệt giữa các giá trị mong muốn và output của mạng neuron, cụ thể ở đây gồm n giá trị (n output nodes).

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial \sum_n (t_n - o_n)^2}{\partial w_{jk}}$$

Output tại node n, o_n chỉ phụ thuộc vào các liên kết tới nó. Nghĩa là tại node k, o_k chỉ phụ thuộc vào w_{jk} (các liên kết từ lớp j tới node k). Ta có thể thu gọn biểu thức trên lại như sau:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial (t_k - o_k)^2}{\partial w_{jk}}$$

Với t_k là hằng số, không phụ thuộc vào w_{jk}. Áp dụng 'chain rule' ta có:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial o_k} * \frac{\partial o_k}{\partial w_{jk}}$$

Mà $\frac{\partial E}{\partial o_k}$ là đạo hàm của hàm E theo o_k.

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) * \frac{\partial o_k}{\partial w_{jk}}$$

o_k là output tại node k, được tính bằng hàm sigmoid và tổng giá trị của các liên kết nhân với giá trị của các inputs.

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) * \frac{\partial sigmoid(\sum_j w_{jk} * o_j)}{\partial w_{jk}}$$

Lưu ý, o_j là output của lớp ẩn, không phải là output của lớp cuối cùng k.

Áp dụng công thức:

$$\frac{\partial sigmoid(x)}{\partial x} = sigmoid(x) * (1 - sigmoid(x))$$

Ta có:

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) * sigmoid\left(\sum_j w_{jk} * o_j\right) * \left(1 - sigmoid\left(\sum_j w_{jk} * o_j\right)\right) * \frac{\partial \sum_j w_{jk} * o_j}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial w_{jk}} = -2(t_k - o_k) * sigmoid\left(\sum_j w_{jk} * o_j\right) * \left(1 - sigmoid\left(\sum_j w_{jk} * o_j\right)\right) * o_j$$

Hằng số 2 trong biểu thức trên không quan trọng nên có thể được lược bỏ.

Tiến hành phân tích để rút gọn biểu thức trên:

- (target – actual) là giá trị lỗi được truyền ngược từ lớp output vào lớp ẩn, nên có thể viết gọn lại thành e_j
- Biểu thức tính tổng bên trong hàm sigmoid biểu thị tổng của tất cả inputs của node j. Đặt là i_j.
- Phần cuối cùng là output của các nodes o_i

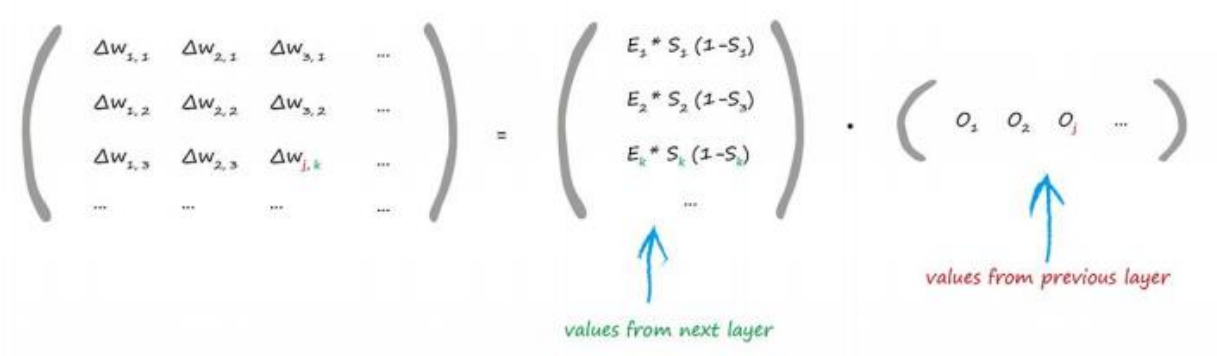
Hàm lỗi cho các liên kết giữa lớp input và lớp ẩn:

$$\frac{\partial E}{\partial w_{jk}} = -(e_j) * sigmoid\left(\sum_j w_{ij} * o_i\right) * \left(1 - sigmoid\left(\sum_j w_{ij} * o_i\right)\right) * o_i$$

Giá trị của các liên kết thay đổi ngược vời chiều của gradien (trong các ví dụ ở phần gradient). Và giá trị này cũng được thay đổi bởi một hằng số, tạm gọi là hằng số học α (learning factor), để tránh xảy ra hiện tượng overfitting. w_jk được cập nhật bằng công thức:

$$new\ w_{jk} = old\ w_{jk} - \alpha * \frac{\partial E}{\partial w_{jk}}$$

Biểu diễn dưới dạng nhân ma trận:

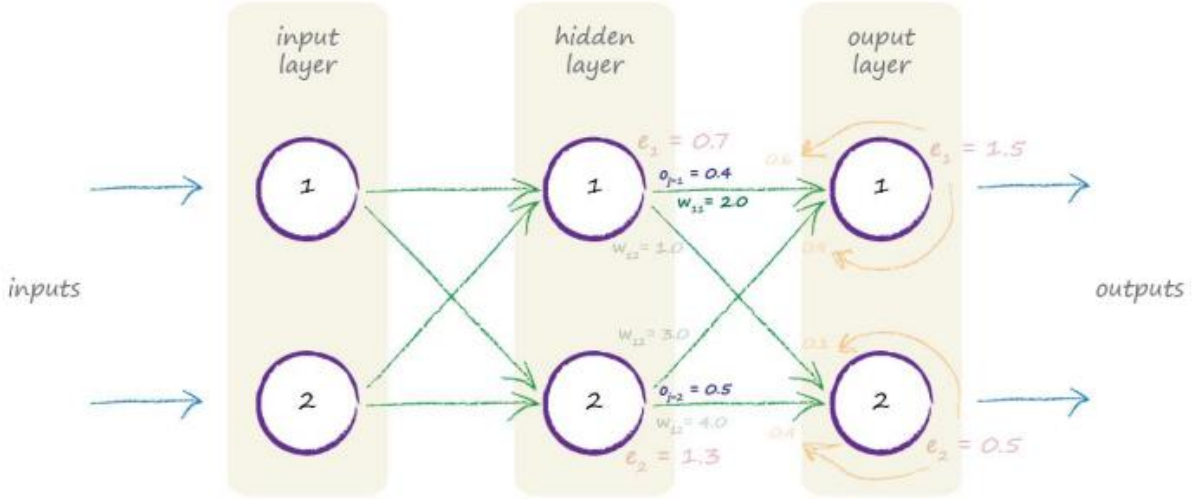


Gọi Δw_{jk} là giá trị update của các ma trận liên kết, ta có:

$$\Delta w_{jk} = \alpha * E_k * sigmoid(o_k) * \left(1 - sigmoid(o_k)\right) * {o_j}^T$$

Trong đó ${o_j}^T$ là ma trận xoay (transpose) của o_j.

Ví dụ:



Ta có:

$o_{j=1} = 0.4, o_{j=2} = 0.5$
$w_{11} = 2.0, w_{21} = 3.0$
$w_{12} = 1.0, w_{22} = 4.0$
$e_1 = 1.5, e_2 = 0.5$

Giá trị của w_11 sau khi được cập nhật là bao nhiêu?

⇨ Giải:
Từ công thức:

$$\frac{\partial E}{\partial w_{jk}} = -(t_k - o_k) * sigmoid\left(\sum_j w_{jk} * o_j\right) * \left(1 - sigmoid\left(\sum_j w_{jk} * o_j\right)\right) * o_j$$

- $(t_k - o_k) = e_1 = 1.5$
- $\sum_j w_{jk} * o_j = (w_{11} * o_{j=1}) + (w_{21} * o_{j=2}) = (2.0 * 0.4) + (3.0 * 0.5) = 2.3$
- $Sigmoid(2.3) = \frac{1}{1-e^{-2.3}} = 0.909$

- Sigmoid(2.3)*(1-Sigmoid(2.3)) = 0.083
- $o_j = o_{j=1} = 0.4$
- $-\frac{\partial E}{\partial w_{jk}} = -(1.5 * 0.083 * 0.4) = -0.0498$
- $new\ w_{jk} = old\ w_{jk} - \alpha * \frac{\partial E}{\partial w_{jk}} = 2.0 - (0.1 * -0.0498) = 2.0 + 0.00498 = 2.00498$

⇨ Giá trị sau khi được cập nhật của $w_{11} = 2.00498$

4. Chương IV: Xây dựng chương trình

4.1. Ngôn ngữ, nền tảng và các công cụ hỗ trợ

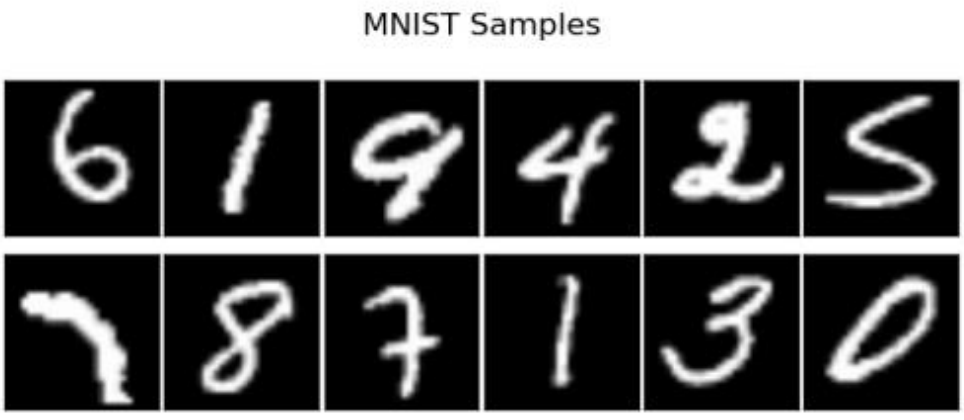
+ Ngôn ngữ sử dụng: Python 2.7

+ IDE: PyCharm Community Edition 2016.3

+ Các module/thư viện: Numpy, Matplotlib, OpenCV, Tkinter, Scipy

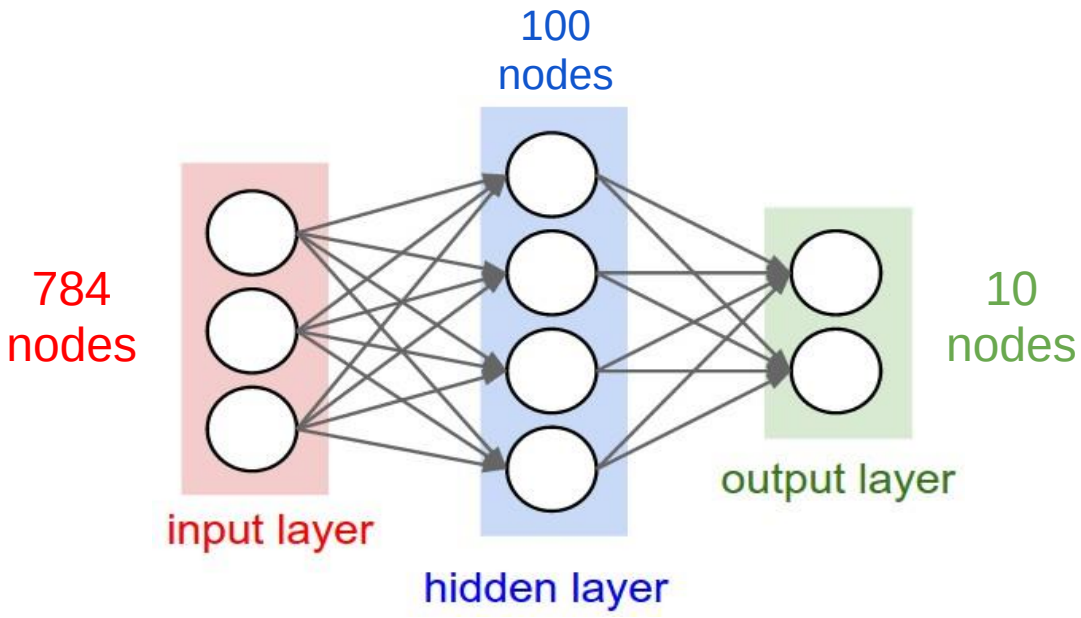
+ Datasets: MNIST data set

- Training set: http://pjreddie.com/media/files/mnist_train.csv - 60000 labels
- Test set: http://pjreddie.com/media/files/mnist_test.csv - 10000 labels
- Training set được mở rộng lên 180000 labels (alignment)



Một số mẫu từ dữ liệu MNIST

4.2. Cấu trúc của mạng neuron



Chương trình gồm 3 lớp:

- GUI.py: giao diện chương trình.
- NeuralNetwork.py: phần code mạng neuron.
- DigitDetection.py: thuật toán detect số.

4.3. Hiện thực chương trình

NeuralNetwork.py

```
import numpy
# scipy.special for the sigmoid function expit()
import scipy.special
import time
import scipy.ndimage

class Neural_Network:
    """
    inodes = so luong nodes trong lop input
    hnodes = so luong nodes trong lop an
    onodes = so luong nodes trong lop output
    wih = ma tran lien ket giua lop input va lop an
    who = ma tran lien ket giua lop an
```

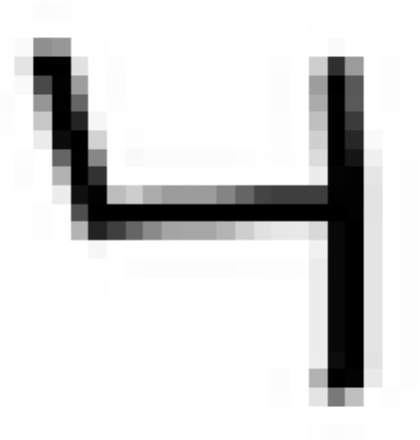
```
lr = learning rate
activation_function = ham sigmoid

# cach thuc hoat dong:
# buoc 1: nhan input, tinh ra output
# buoc 2: lay output so sanh voi gia tri mong muon, tu do update gia tri cua cac ma tran lien ket
"""
def __init__(self, inputnodes, hiddennodes, outputnodes, learningrate):
    self.inodes = inputnodes
    self.hnodes = hiddennodes
    self.onodes = outputnodes

    self.lr = learningrate

    # w_i_j la ma tran cac gia tri cua cac lien ket tu  node i to inode j
    # ma tran lien ket co dang
    # w11 w21
    # w12 w22
    # w13 w23 etc
    # gia tri khoi tao ban dau trong khoang -0.5, +0.5
    self.wih = (numpy.random.rand(self.hnodes, self.inodes) - 0.5)
    self.who = (numpy.random.rand(self.onodes, self.hnodes) - 0.5)
    #print(self.wih)
    #print(self.who)

    # ham activation: ham sigmoid
    self.activation_function = lambda x: scipy.special.expit(x)



def train(self, input_list, target_list):
    """
    huan luyen mang neuron
    :return:
    """
    # chuyen doi input va output thanh mang 2 chieu
    inputs = numpy.array(input_list, ndmin=2).T
    targets = numpy.array(target_list, ndmin=2).T

    #### 1. feed forward  gia tri input
    # tinh toan gia tri input cho lop an
    hidden_inputs = numpy.dot(self.wih, inputs)
    # tinh toan gia tri output cho lop an
    hidden_outputs = self.activation_function(hidden_inputs)
    # tinh toan gia tri input cho lop output
    final_inputs = numpy.dot(self.who, hidden_outputs)
    # tinh toan gia tri output cho lop xuat
    final_outpus = self.activation_function(final_inputs)


    #### 2. Truyen nguoc loi
```

```
        # tinh  error cuar lop output = (target - actual)
        output_erros = targets - final_outpus
        # tinh errors cua lop an
        hidden_erros = numpy.dot(self.who.T, output_erros)


        #### 3. Cap nhat gia tri cua cac lien ket
        # cap nhat gia tri cua cac lien ket giua lop an va lop output
        # cong thuc:  Wjk += lr * Ek * sigmoid(Ok)*(1-sigmoid(Ok)) * OjT
        self.who += self.lr * numpy.dot((output_erros * final_outpus*(1 - final_outpus)),
numpy.transpose(hidden_outputs))
        # cap nhat gia tri cua cac lien ket giua lop input va lop an
        self.wih += self.lr * numpy.dot((hidden_erros * hidden_outputs*(1 - hidden_outputs)),
numpy.transpose(inputs))


    def query(self, input_list):
        """
        truy van mang neuron
        tra ve ket qua ung voi input_list
        :return:
        """

        # chuyen doi ve mang 2 chieu
        inputs = numpy.array(input_list, ndmin=2).T
        # tinh toan input cho lop an
        hidden_inputs = numpy.dot(self.wih, inputs)
        # tinh toan output cho lop an
        hidden_outputs = self.activation_function(hidden_inputs)
        # tinh toan input cho lop output
        final_inputs = numpy.dot(self.who, hidden_outputs)
        # tinh toan output cho lop output
        final_outputs = self.activation_function(final_inputs)
        return final_outputs
```

Cách thức hoạt động

+ B1: vẽ số lên grid

+ B2: Phát hiện vùng chứa số, crop vùng đó và resize về 28 x 28 pixels



+ B3: convert về dạng ảnh xám (grayscale)

+ B4: chuyển ảnh về dạng ma trận 28x28

```
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   2.   0.  55. 169. 208. 197. 127.
 11.   1.   1.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   3.   0. 187.
255. 255. 255. 255.  66.   0.   3.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   3.   0. 180. 252. 250. 249. 252.  66.   0.   3.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   3.   0. 180. 255. 252. 252. 255.
 66.   0.   3.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   3.   0. 184.
255. 252. 252. 255.  64.   0.   3.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   3.   0. 157. 255. 252. 252. 255. 107.   0.   3.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   3.   0.  81. 255. 252. 252. 255.
177.   0.   3.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   3.   0.  64.
255. 252. 252. 255. 182.   0.   3.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   3.   0.  67. 255. 252. 252. 255. 181.   0.   3.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   3.   0.  65. 255. 252. 252. 255.
180.   0.   3.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   3.   0.  73.
255. 252. 252. 255. 186.   0.   3.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   3.   0. 163. 255. 252. 252. 255. 118.   0.   3.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   3.   0. 185. 255. 252. 252. 255.
 60.   0.   3.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   3.   0. 179.
255. 252. 252. 255.  69.   0.   3.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   3.   0. 184. 255. 252. 252. 255.  60.   0.   3.   0.
  0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.
  0.   0.   0.   0.   0.   3.   0. 168. 252. 249. 250. 252.
```

+ B5: chuẩn hóa giá trị của ma trận về range (0, 1)

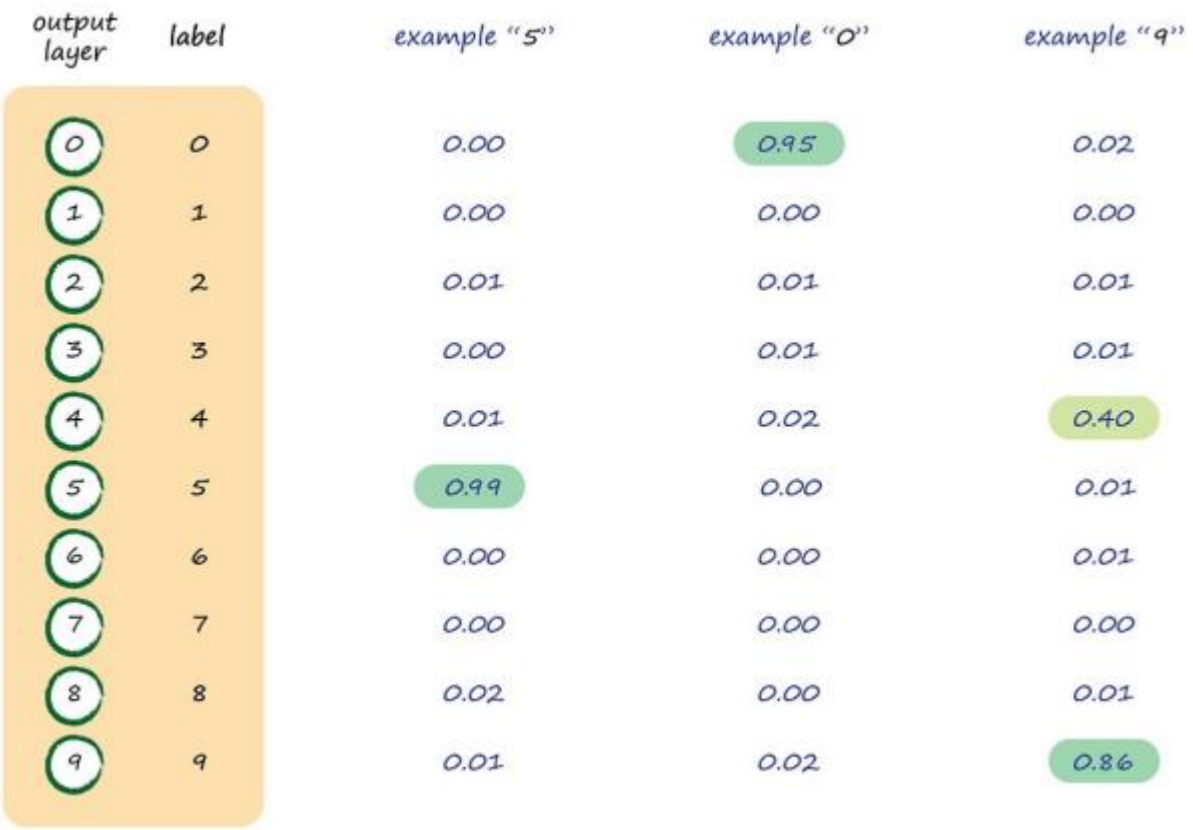
```
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.208       0.62729412  0.99223529  0.62729412  0.20411765
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.19635294  0.934       0.98835294  0.98835294  0.98835294
0.93011765  0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.01        0.01        0.01        0.01        0.01        0.01
0.01        0.21964706  0.89129412  0.99223529  0.98835294  0.93788235
0.91458824  0.98835294  0.23129412  0.03329412  0.01        0.01        0.01
```

+ B6: áp dụng mạng neuron.

+ B7: hiển thị kết quả.

```
[ 0.99  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01  0.01]
```

Nhận biết kết quả từ output của mạng neuron.

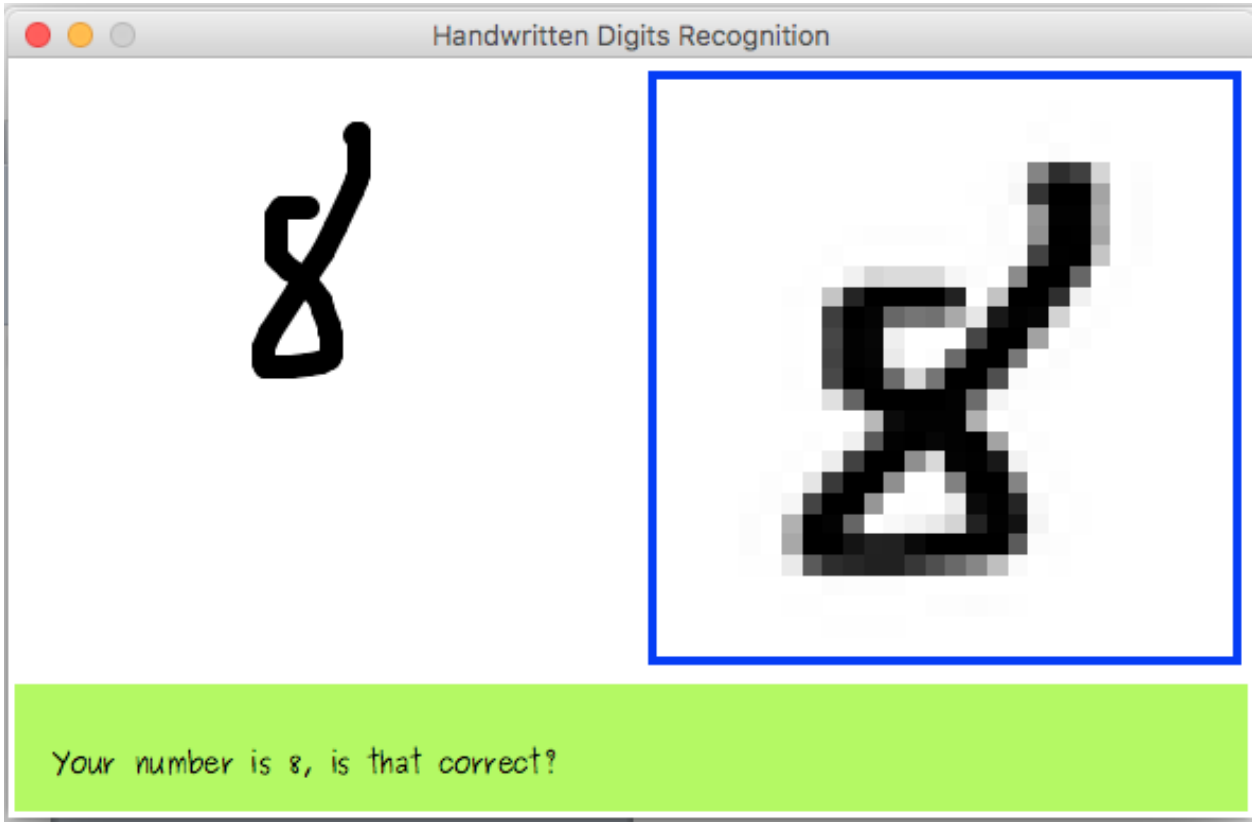


Ở cột "example 5", giá trị max nằm ở phần tử thứ 5 (tính từ 0), nghĩa là mạng neuron nhận dạng số đã viết là 5.

Ở cột "example 0", giá trị max nằm ở phần tử đầu tiên, nghĩa là mạng neuron nhận dạng số đã viết là 0.

Ở cột "example 9", ta có 2 phần tử có giá trị lớn, tuy nhiên phần tử có giá trị max là phần tử số 9, mạng neuron nhận dạng số đã viết là 9.

Giao diện chương trình

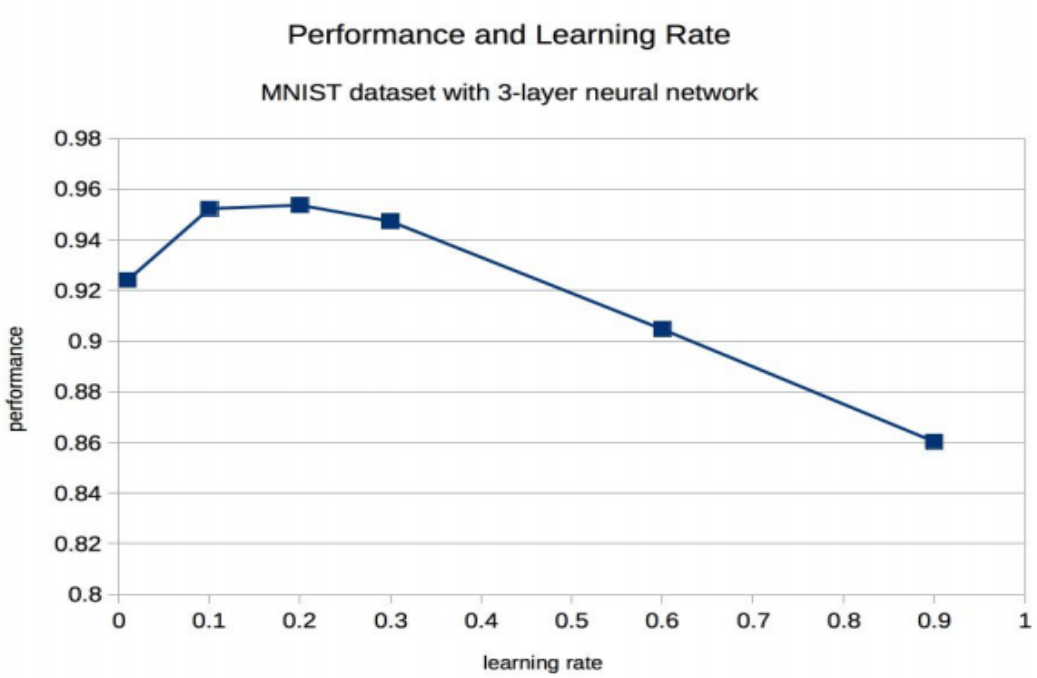


Kết quả chương trình

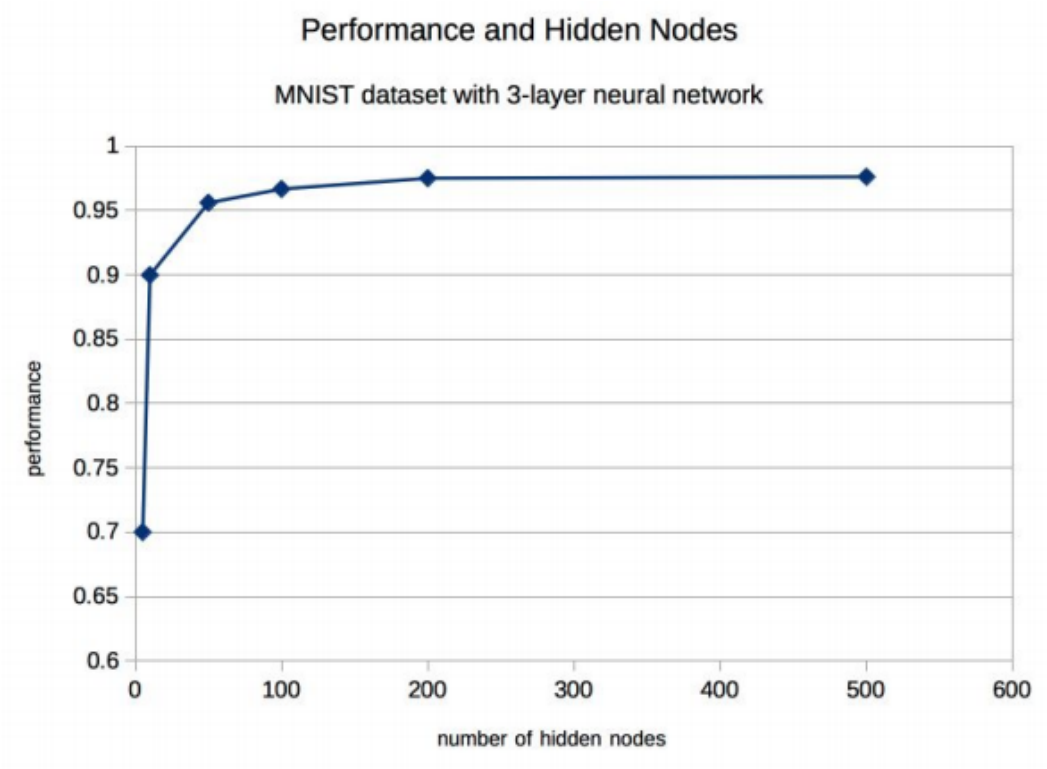
- Thời gian huấn luyện: >170s
- Độ chính xác (với 10000 test labels): ~95%

```
Finish training
Took 125.70317602157593 s
Start testing 10000 images
Finish testing
Took  2.3022420406341553 s
Accuracy: 95.28%
```

Độ chính xác khi thay đổi learning rate



Độ chính xác khi thay đổi số node ở lớp ẩn (hidden layer)



5. Chương V: Kết luận

5.1. Kết quả đạt được

- Xây dựng thành công một chương trình nhận diện chữ số sử dụng mạng neuron feedforward với độ chính xác khá cao.

- Chương trình có GUI thân thiện, dễ sử dụng.

- Nhóm đã học nhiều kiến thức về máy học cũng như xử lý ảnh thông qua đề tài này.

- Nâng cao kỹ năng làm việc nhóm.

- Nâng cao kỹ năng viết báo cáo.

5.2. Hạn chế

- Vì một số hạn chế của module Tkinter cũng như trình độ của nhóm còn giới hạn nên phần GUI chưa tốt cho lắm.

- Các số vẽ vào grid để test phải có dạng giống như dữ liệu training của MNIST.

5.3. Hướng phát triển

- Tìm hiểu thêm các phương pháp tiếp cận khác để cải thiện như là áp dụng CNN, kết hợp thêm nhiều kỹ thuật khử nhiễu nhằm xây dựng một chương trình có đầu vào là 1 ảnh ngẫu nhiên thay vì vẽ thẳng vào grid hay lấy ma trận ảnh test của MNIST như hiện nay.

- Cải thiện GUI và một số tính năng khác bằng cách tìm hiểu thêm về các module/thư viện hỗ trợ (như PyGame).

TÀI LIỆU THAM KHẢO

[1] Heaton-Introduction_to_Neural_Networks_with_Java - http://heatonresearch.com

[2] Python Machine Learning - Sebastian Raschka

[3] Make Your Own NeuralNetwork - Tariq Rashid

[4] Introduction to the Math of Neural Networks – Jeff Heaton

[5] Docs of OpenCV: http://docs.opencv.org/