TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI

**KHOA CÔNG NGHỆ THÔNG TIN**



**BÁO CÁO**

**THỰC TẬP TỐT NGHIỆP**

**NGÀNH: KĨ THUẬT PHẦN MỀM**

**NGHIÊN CỨU DEEP LEARNING, ỨNG DỤNG NHẬN DẠNG CHỮ SỐ VIẾT TAY**

|  |  |
| --- | --- |
| Sinh viên thực hiện: | Bùi Ngọc Minh  Lớp ĐH KTPM1 K11 |
| Giảng viên hướng dẫn: | TS. Nguyễn Bá Nghiễn |

***Hà Nội, 03/2020***

# LỜI NÓI ĐẦU

Ngày nay với thời đại công nghệ phát triển, AI – Artificial Intelligence và cụ thể hơn Machine Learning đang là xu thế và nổi lên như một bằng chứng của cuộc cách mạng công nghiệp lần thứ tư. Trí tuệ nhân tạo đang len lỏi và từng ngõ ngách trong cuộc sống của chúng ta và trở nên quá quen thuộc như Siri của Apple, hệ thống tự nhận diện khuôn mặt của Facebook, xe tự lái, giảm thiểu tai nạn của Google..

Machine Learning là một tập con của AI. Là một lĩnh vực nhỏ của ngành Khoa Học Máy Tính, nó có khả năng tự học hỏi dựa trên dữ liệu đầu vào mà không cần phải lập trình cụ thể.

Những năm gần đây, khi mà khả năng tính toán của máy tính đã được nâng lên một tầm cao mới, dữ liệu ngày một lớn đã được các công ty công nghệ lớn thu thập được. Lĩnh vực mới được ra đời để phục vụ nhu cầu đó là Deep Learning. Deep Learning đã giúp máy tính thực thi những công việc tưởng chừng như không thể như phân loại các vật thể trong một bức ảnh, tự tạo chú thích cho ảnh, giả giọng người, ...

Bắt kịp theo xu hướng của thế giới hiện tại, và tại Việt Nam cũng đang rất thiếu nguồn nhân lực về ngành này nên em đã quyết định chọn đề tài “Nghiên cứu Deep Learning, ứng dụng nhân dạng chữ số viết tay” do thầy Nguyễn Bá Nghiễn hướng dẫn.

Với bài toán nhận dạng chữ số viết tay thì vấn đề trở nên phức tạp vì các lí do như không có khái niệm về kích cỡ, font chữ, cùng 1 số mỗi người viết một khác.

Nội dung bài báo cáo gồm 3 chương

* Chương 1: Tổng quan về đề tài
* Chương 2: Deep Learning
* Chương 3: Cài đặt thực nghiệm và đánh giá

# LỜI CẢM ƠN

Em xin chân thành cảm ơn các thầy, các cô khoa Công nghệ Thông tin – Trường Đại học Công Nghiệp Hà Nội đã tận tình dạy dỗ và truyền đạt cho em nhiều kiến thức quý báu trên ghế nhà trường.

Đặc biệt em xin tỏ lòng biết ơn sâu sắc đến thầy Ts. Nguyễn Bá Nghiễn đã tận tình giúp đỡ và truyền đạt nhiều kinh nghiệm về mảng Deep Learning để em có thể thực hiện và hoàn thành tốt đề tài đã chọn.

Ngoài ra em cũng xin chân thành cảm ơn các bạn lớp Kĩ Thuật Phần Mềm 1 – K11, trường Đại học Công Nghiệp Hà Nội, những đồng nghiệp ở công ty hiện đang đang đi làm đã luôn sát cánh, động viên và giúp đỡ mình rất nhiều trong quá trình thực hiện đề tài này.

Em xin chân thành cảm ơn!

**Mục lục**

[LỜI NÓI ĐẦU 2](#_Toc38280278)

[LỜI CẢM ƠN 3](#_Toc38280279)

[CHƯƠNG I: TỔNG QUAN VỀ ĐỀ TÀI 5](#_Toc38280280)

[1. Giới thiệu về bài toán nhận dạng chữ số viết tay 5](#_Toc38280281)

[2. Đặt vấn đề và hướng giải quyết 6](#_Toc38280282)

[CHƯƠNG II: DEEP LEARNING 7](#_Toc38280283)

[1. Features Engineering 7](#_Toc38280284)

[1.1. Binning 8](#_Toc38280285)

[1.2. One-hot encoding 8](#_Toc38280286)

[1.3. Scaling 9](#_Toc38280287)

[2. Network Architecture 9](#_Toc38280288)

[2.1. Từ Perceptron đến Neural Network 9](#_Toc38280289)

[2.2. Convolunation Neural Network - CNN 18](#_Toc38280290)

[3. Optimization 30](#_Toc38280291)

[3.1. Overfitting/underfitting 30](#_Toc38280292)

[3.2. Optimizer 33](#_Toc38280293)

[3.3. Activation Functions 37](#_Toc38280294)

[3.4. Initialization 37](#_Toc38280295)

[CHƯƠNG III: CÀI ĐẶT THỰC NGHIỆM VÀ ĐÁNH GIÁ 39](#_Toc38280296)

# CHƯƠNG I: TỔNG QUAN VỀ ĐỀ TÀI

## Giới thiệu về bài toán nhận dạng chữ số viết tay

Nhận dạng chữ số viết tay là 1 dạng trong bài toán Phân loại hình ảnh – Image Classification. Đây là một trong những bài toán phân loại phổ biến nhất trong lĩnh vực Computer Vision và Deep Learning.

Bộ cơ sở dữ liệu **MNIST – Modified National Institure of Standards and Technology** là bộ cơ sở dữ liệu lớn nhất về chữ số viết tay và được sử dụng trong hầu hết các thuật toán nhận dạng hình ảnh (Image Classification). MNIST bao gồm 2 tập dữ liệu, tập dữ liệu dùng để training có 60000 các ví dụ khác nhau về chữ số viết tay từ 0 đến 9, tập dữ liệu để test thì gồm 10000 ví dụ khác nhau tương tự. Tất cả đều đã được gán nhãn (label). Bộ cơ sở dữ liệu này có thể được tải xuống từ trang chủ: <http://yann.lecun.com/exdb/mnist/>. Dưới đây là hình ảnh ví dụ về 1 số hình ảnh trong cơ sở dữ liệu MNIST.

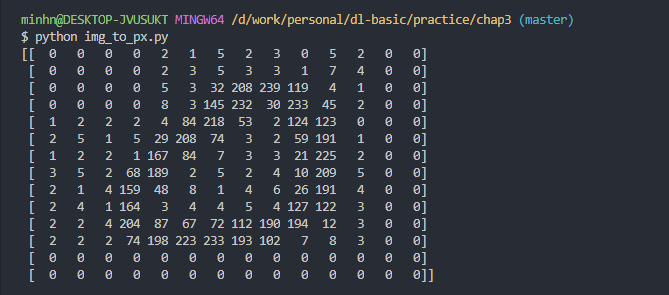


Bộ cơ sở dữ liệu chữ số viết tay MNIST

Mỗi bức ảnh là một ảnh đen trắng có kích thước 28x28 pixel (tổng 784 pixel). Mỗi pixel mang một giá trị là số từ 0 đến 255. Các pixel càng gần 0 thì càng đen, càng gần 255 thì càng trắng. Mỗi ảnh đều được chuẩn hóa về kích thước 28x28 và dạng ảnh xác (gray scale) cũng như đã được căn giữa cố định vị trí. Dưới đây là ví dụ về ảnh chữ số 0 và giá trị của các pixel trong ảnh.

D:\work\personal\dl-basic\practice\chap3\training\0\img_153.jpg

Một ví dụ về số 0 trong CSDL MNIST



Ma trận pixel của ảnh trên

Đây là tập dữ liệu rất phổ biến được sử dụng để thử nghiệm các thuật toán do tính chất **khá dễ** của nó. Có câu rằng **Thuật toán chạy tốt trên MNIST chưa chắc chạy tốt trên dataset khác nhưng không tốt trên MNIST thì đa phần không tốt trên các dataset khác**. Chính vì lý do đó nên MNIST được coi là dataset phổ biến nhất trong bài toán phân loại ảnh,

## Đặt vấn đề và hướng giải quyết

Mỗi ảnh đều là ảnh xám và có kích thước 28x28 và đều được đánh label cho từng ảnh từ 0 đến 9. Và chúng ta cần dự đoán ảnh là số mấy.

Trong các thuật toán cở bản Machine Learning thì để phân loại chúng ta có thuật toán Logistic Regression. Tuy nhiên output của thuật toán chỉ là giá trị nhị phân và là một mô hình Neural Network đơn giản. Do đó chúng ta sẽ cần áp dụng mô hình Neural Network cho bài toán phân loại ảnh.

Với mô hình Neural Network thì chúng ta có các loại như Neural Network cổ điển, CNN – Convolutional Neural Network và RNN – Recurrent Neural Network.

Do input của bài toán là ảnh nên chúng ta sẽ sử dụng mô hình Neural Network cổ điển hoặc CNN.

## Mục tiêu đề tài

Do đây là một bài toán đã phổ biến nên mục tiêu chúng ta sẽ tìm ra 1 model mà có độ chính xác > 90%.

Ở bài viết này tôi sẽ dùng thư viện Keras của Python để xử bài toán.

# CHƯƠNG II: DEEP LEARNING

## Features Engineering

Feature Engineering là quá trình chuyển đổi tập dữ liệu thô ban đầu thành các thuộc tính (features) có thể giúp biểu diễn tập dữ liệu ban đầu tốt hơn, để giải quyết các bài toán dễ dàng hơn, giúp tương thích với từng mô hình dự đoán cụ thể cũng như cải thiện độ chính xác của mô hình dự đoán hiện tại.

Feature Engineering là một giai đoạn không thể thiếu trong quá trình phát triển bất kì một hệ thống thông minh nào.

Phần lớn các bài toán Machine Learning có thể được thể hiện trong hình vẽ sau:



Nguồn: machinelearningcoban.com

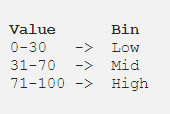
Với khối Feature Extraction (Feature Engineering) chúng ta cần phải thiết kế sao cho từ tạo ra một Feature Extractor biến dữ liệu thô ban đầu thành dữ liệu phù hợp với mục đích sử dụng.

Một số kỹ thuật thường được sử dụng:

### Binning

Là một kỹ thuật để chuyển các dữ liệu liên tục thành các nhóm dữ liệu. Thực hiện bằng cách nhóm các giá trị vào các “bin” đã được xác định trước. Các giá trị liên tục sau đó được thay thế bằng tên của “bin” đã được định nghĩa trước mà chứa giá trị đó.

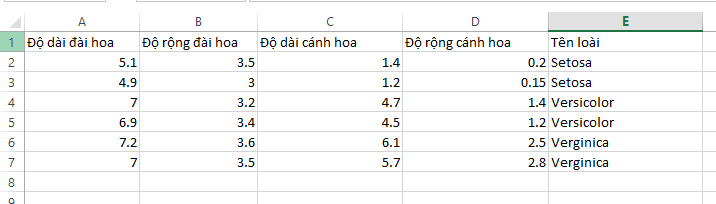
Ví dụ: Trong tập dữ liệu từ 0 -> 100. Ta chia thành 3 bin như sau



### One-hot encoding

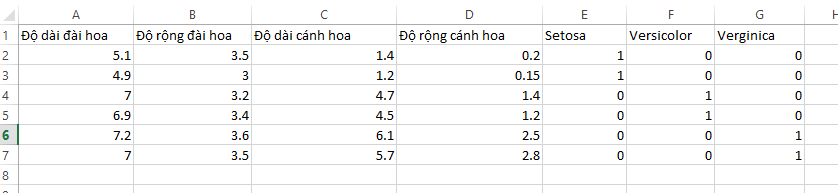
Là một trong những phương thức phổ biến nhất trong Machine Learning. Kỹ thuật này sẽ tách value của 1 cột thành nhiều cột và đánh số 0 hoặc 1 vào mỗi cột. Các giá trị nhị phân này biểu thị mối quan hệ giữa các cột đã tách ra với cột gốc.

Ví dụ: Bài toán phân loại hoa IRIS ta có 3 loài hoa là Setosa, Versicolor, Verginica



IRIS dataset

Sau khi áp dụng One-hot encoding thì chúng ta sẽ có được bảng mới như sau:



One-hot encoding IRIS

### Scaling

Khi các điểm dữ liệu chệnh lệch nhau quá lớn, một thành phần có giá trị trong khoảng 0-1, thành phần khác lại có giá trị từ 0-1000. Lúc này chúng ta cần chuẩn hóa dữ liệu trước khi thực hiện các bước tiếp theo.

Một vài phương pháp thường dùng:

* + 1. Rescaling

Phương pháp này thông thường sẽ đưa tất cả các thành phần dữ liệu về cùng 1 khoảng [0;1] hoặc [-1;1]. Ví dụ nếu muốn đưa một thành phần về khoảng [0;1] công thức sẽ là:

Trong đó:

|  |  |
| --- | --- |
| x: | Giá trị ban đầu |
| x’: | Giá trị sau khi chuẩn hóa |
| min(x), max(x): | Giá trị max và min được tính trên toàn bộ dữ liệu ở cùng một thành phần |

* + 1. Standardization

Phương pháp này giả sử mỗi thành phần đều có phân phối chuẩn với kỳ vọng là 0 và phương sai là 1. Khi đó, công thức chuẩn hóa sẽ là:

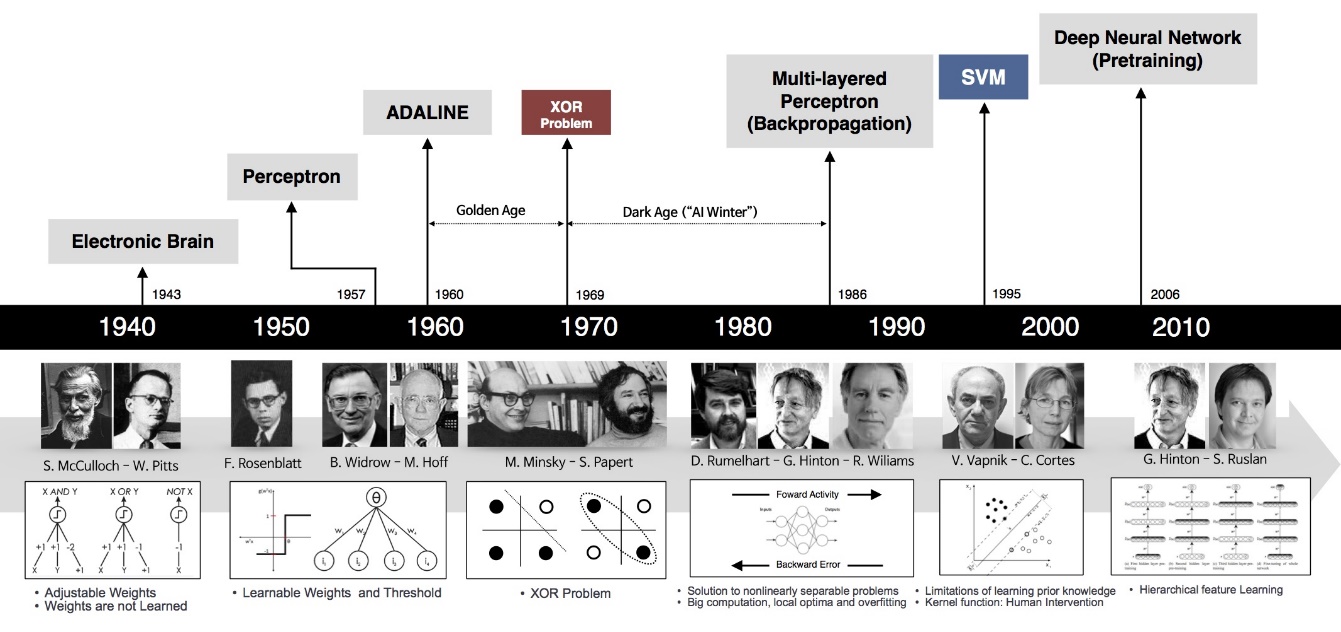
Trong đó:

|  |  |
| --- | --- |
| x: | Giá trị ban đầu |
| x': | Giá trị sau khi chuẩn hóa |
| µ: | Kỳ vọng |
| σ: | Độ lệch chuẩn |

## Network Architecture

### Từ Perceptron đến Neural Network

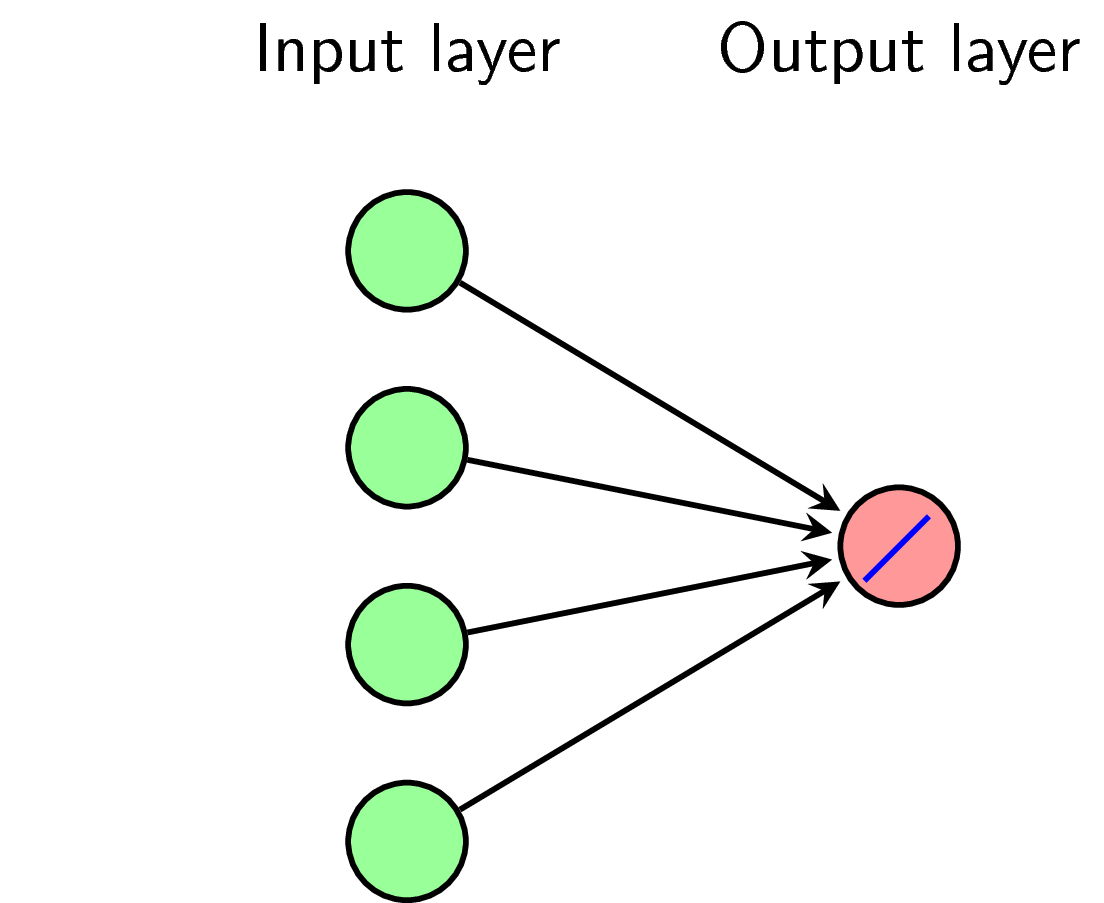
Trong những năm gần đây Deep Learning mới được nhắc nhiều đến, tuy nhiên những nền tảng cơ bản đầu tiên đã được xuất hiện từ rất lâu. Dưới đây là lịch sử hình thành của Deep Learning



Nguồn: machinelearningcoban.com

Phiên bản sơ khai của Neural Network là Perceptron Learning Algorithm (Perceptron) là một thuật toán supervised learning giúp giải quyết các bài toán phân lớp nhị phân.

Vào thời điểm đấy mọi người đều tin rằng thuật toán này sẽ làm được những việc tưởng chừng như không thể như có thể đi, nói chuyện, nhìn, nhận thức, tự sinh sản. Tuy nhiên vào năm 1969 Marvin Minsky và Seymour Papert đã chứng minh rằng perceptron **không thể** học được bài toán **XOR**.



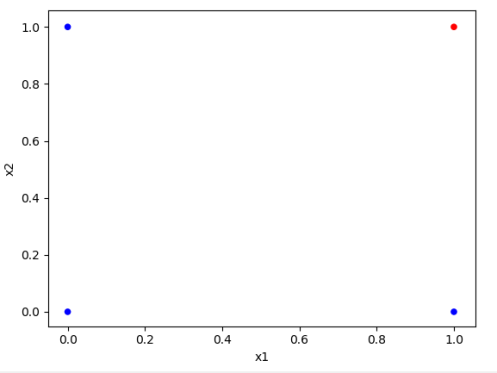
Mô hình Perceptron

Với thuật toán Perceptron chúng ta sẽ tìm ra đường thẳng chia dữ liệu thành 1 miền để cho ra kết quả.

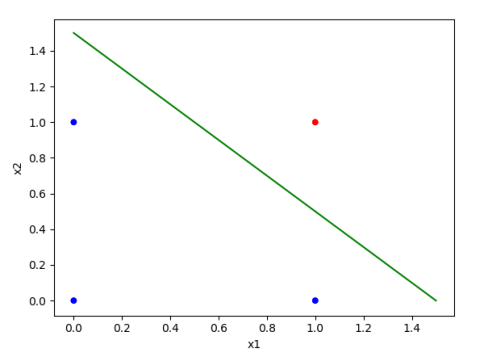
Ví dụ với bài toán AND, ta có bảng chân lý:

|  |  |  |
| --- | --- | --- |
| A | B | A AND B |
| 1 | 1 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 0 | 0 | 1 |

Biểu diễn dữ liệu dưới dạng đồ thị ta có:

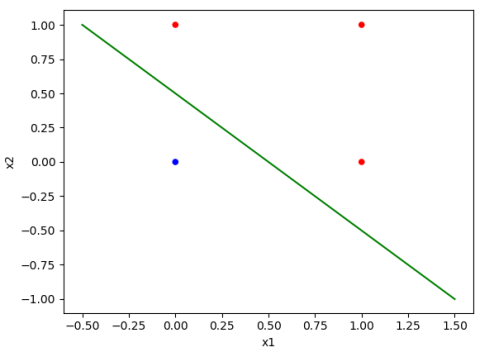


Áp dụng thuật toán Perceptron ta tìm ra được đường thẳng chia thành 2 miền dữ liệu như sau:



Tương tự với bài toán OR ta có bảng chân lý vào đường thẳng chia cắt 2 miền dữ liệu như sau:

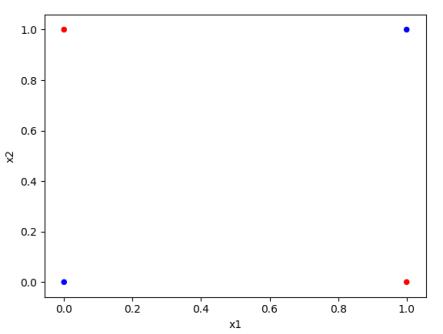
|  |  |  |
| --- | --- | --- |
| A | B | A OR B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



Đối với bài toán XOR, ta có bảng chân lý:

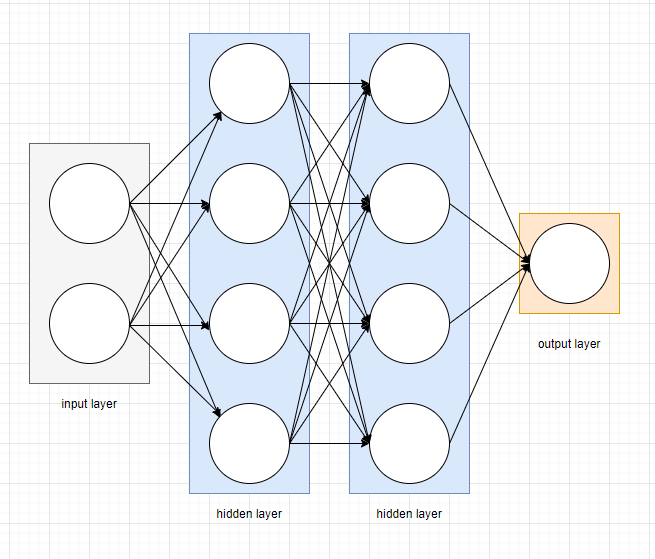
|  |  |  |
| --- | --- | --- |
| A | B | A XOR B |
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Ta có đồ thị đối với bài toán XOR:



Như vậy chúng ta có thể thấy không thể dùng 1 đường thẳng để chia dữ liệu bài toán XOR thành 2 miền được. Lúc này người ta mới cần 1 giải pháp mới đó là **Neural Network**.

Mô hình **Neural Network** tổng quát:



Mô hình Neural Network

Mô hình Neural Network với nhiều hidden layer (được gọi là multiple layer perceptron – MLP) là 1 mô hình nâng cấp của Perceptron có thể huấn luyện 1 cách hiệu quả dựa trên một quy trình đơn giản gọi là **Backpropagation (lan truyền ngược)**. Việc tính đạo hàm của các hàm số phức tạp mô tả quan hệ giữa input vào output của bài toán là rất quan trọng vì hầu hết các thuật toán tối ưu đều được thực hiện thông qua việc tính đạo hàm. Việc này giúp Neural Network thoát khỏi những hạn chế tồn đọng của Perceptron.

Trong mô hình Neural Network, layer đầu tiên là input layer, các layer nằm giữa là các hidden layers và layer cuối cùng là output layer. Mỗi hình tròn được gọi là node của mỗi layer.

Mỗi mô hình Neural Network của từng bài toán luôn có 1 input layer, 1 output layer và có hoặc không các hidden layer. Tổng số layer của mô hình được quy ước là tổng số layer – 1 (không tính input layer).

Ví dụ trong hình trên thì chúng ta có 1 input layer với 2 nodes, 2 hidden layers mỗi hidden layer có 4 nodes và output layer có 1 node. Tổng số lượng layer của mô hình là 3 layer.

Chúng ta sẽ quy ước số node trong hidden layer thứ i là .

Ma trận kích thước là ma trận hệ số giữa layer (k-1) và layer k, trong đó là hệ số kết nối từ node thứ i của layer k-1 đến node thứ j của layer k.

Với node thứ i trong layer ta thực hiện 2 bước:

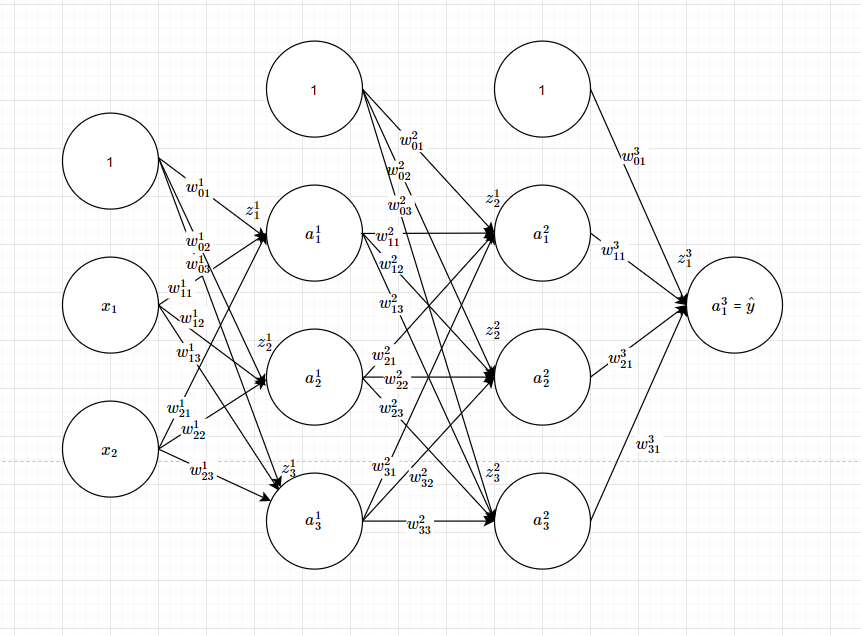
* Tính tổng linear: , là tổng tất cả các node trong layer trước đấy nhân với hệ số w tương ứng.
* Áp dụng hàm activation:

Vector có kích thước là giá trị các node của layer k sau bước tính tổng các linear

Vector có kích thước là giá trị của các node trong layer k sau bước áp dụng hàm activation cho vector

Lưu ý: tại mọi layer l

Quay trở lại bài toán XOR, giả sử chúng ta áp dụng mô hình Neural Network cổ điển vào bài toán, gồm 2 hidden layers, mỗi layer 3 node như sau.



Mô hình Neural Network cho bài toán XOR

Giá trị ở các node được tính như sau:

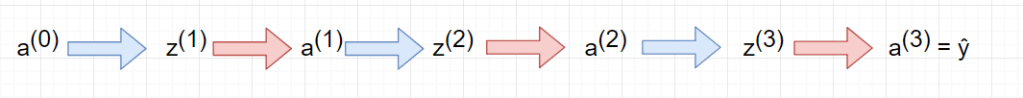
Tại node thứ nhất layer 1 ta có:

Hay tại node thứ 3 layer 2 ta có:

Tương tự tính ra được giá trị của các node còn lại.

Gọi là ma trận hệ số của input layer. Viết 1 cách ngắn gọn công thức tính ma trận hệ số a của các layer ta có:

Tóm lại chúng ta có quá trình Feedforward như sau:



Feedforward

Từ input đầu vào chúng ta có thể tính ra được giá trị dự đoán , việc cần làm ở đây chính là đi tìm hệ số W. Chúng ta có thể áp dụng thuật toán Gradient Descent, đi tìm đạo hàm của các hệ số đối với loss function. Và việc tính đạo hàm của các hệ số trong bài toán này sẽ áp dụng thuật toán Backpropagation.

**Backpropagation**

Gọi X là ma trận hệ số input của bài toán XOR và Y là ma trận hệ số output. Ta có 2 ma trận X, Y như sau:

**Độ chính xác của phương trình (Loss function)**

Với mỗi điểm ta có hàm mất mát:

Hàm mất mát của thuật toán Neural Network tổng quát trên toàn bộ dữ liệu sẽ như sau:

**Gradient Descent**

Để tìm điểm cực tiểu của hàm mất mát, ta có thể dùng thuật toán Gradient Descent để tìm đạo hàm của các hệ số W với hàm mất mát. Nhưng hàm mất mát lại quá phức tạp khiến việc tìm đạo hàm riêng với từng là rất khó. Thuật toán Backpropagation giúp chúng ta dễ dàng tìm được các đạo hàm riêng này.

Với mỗi điểm ta có hàm mất mát:

trong đó là giá trị mà model dự đoán còn là giá trị thực tế của dữ liệu.

Đạo hàm riêng của giá trị dự đoán trên hàm mất mát:

Tính đạo hàm riêng của , áp dụng chain rule chúng ta có:

Do đó :

Tương tự chúng ta có đạo hàm riêng của các :

Đạo hàm riêng của và theo L, áp dụng chain rule ta có công thức tổng quát như sau:

Một lưu ý khi sử dụng Gradient Descent với Neural Network đó là ta không nên khởi tạo tất cả các giá trị vì nếu làm vậy sẽ làm giá trị các nút trong cùng 1 lớp khác lớp input sẽ bị giống nhau, dẫn đến việc thuật toán không thể kết thúc được. Thay vào đó chúng ta nên khởi tạo là các giá trị ngẫu nhiên, thông thường sẽ khởi tạo giá trị ngẫu nhiên nằm trong vùng lân cận của 0.

Mô hình chung thuật toán Neural Network:

* Chọn các giá trị ngẫu nhiên và giá trị learning rate α
* Liên tiếp lặp các phép biến đổi

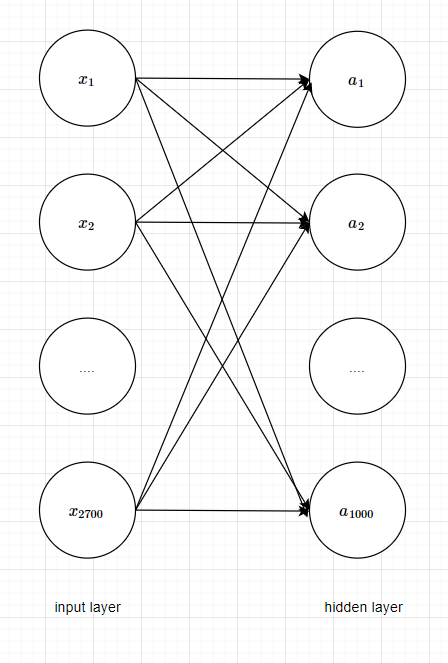
với các đạo hàm riêng được tính bằng thuật toán Backpropagation.

* Thuật toán dừng lại khi giá trị hàm mất mát thay đổi rất nhỏ hoặc trị tuyệt đối các đạo hàm riêng rất nhỏ. Nếu thuật toán không thể kết thúc thì chọn lại giá trị α sao cho hợp lý và chạy lại bước trên.

### Convolunation Neural Network - CNN

Trong mô hình Neural Network cổ điển, mỗi hidden layer là một fully connected layer, nghĩa là mỗi node được kết nối với tất cả các node trong layer trước đấy. Cả mô hình thì được gọi là fully connected neural network (FCN).

Lấy ví dụ với bài toán phân loại ảnh mỗi ảnh có kích thước là 30x30 pixel. Nếu để biểu thị hết nội dung của bức ảnh thì cần truyền vào input layer tất cả các pixel của ảnh với tổng số node là 30x30x3 = 2700 nodes.



Giả sử ta khởi tạo số lượng nodes trong hidden layer là 1000 nodes. Khi đấy số lượng W được sinh ra là 2701\*1000=2701000 weights. Số lượng tham số rất lớn, nếu đối với bài toán xử lý ảnh 256x256 khi đấy số lượng tham số được sinh ra là rất lớn. Khi đó chúng ta cần phải pháp tốt hơn đó là mô hình Convolutional Neural Network (CNN).

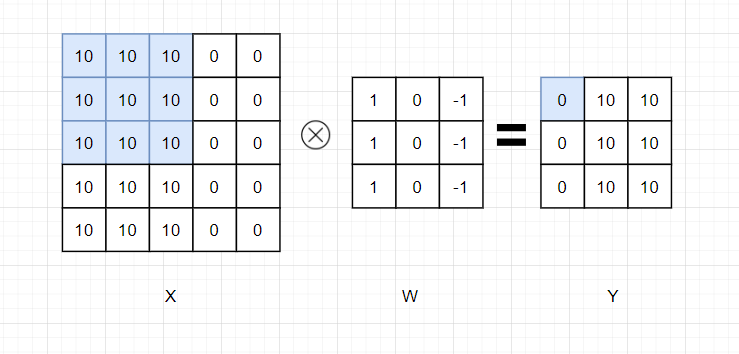
Trong 1 bức ảnh thì các pixel ở gần nhau sẽ có mối liên hệ với nhau hơn là các pixel ở cách xa nhau. Nếu chúng ta sử dụng mô hình Neural Network cổ điển thì khi đấy sẽ bị mất đi tính chất này của ảnh.

**Phép tính convolution**

Ví dụ trên ảnh xám, ta có ma trận A có kích thước m\*n.

Ta định nghĩa 1 ma trận vuông kích thước k\*k gọi là **kernel** trong đó k là số lẻ, k có thể bằng 1,3,5,7,9,... Ví dụ ta có ma trận **kernel** kích thước 3\*3

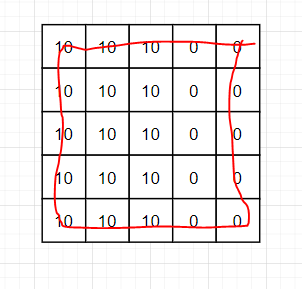
Với mỗi phần tử trong ma trận X có kích thước bằng kích thước của kernel W có phần tử làm trung tâm gọi là ma trận A. Sau đó tính tổng các phần tử của ma trận A với ma trận W thông qua phép tính element-wise rồi viết vào ma trận Y ta có:⊗⊗



Ví dụ khi tính ma trận A (nền xanh trong ma trận X) có cùng kích thước của ma trận W. Ta tính được . Làm tương tự với các phần tử còn lại ta tính được ma trận Y như hình trên.

Từ ví dụ trên chúng ta có thể thấy được là ma trận Y có kích thước nhỏ hơn ma trận X và có kích thước là (m-k+1)\*(n-k+1) với m\*n là kích thước ma trận X, k là kích thước của ma trận W.

Vừa rồi chúng ta mới chỉ xử lý những điểm nằm trong ma trận X. Tuy nhiên đối với những điểm nằm ở viền ma trận X như thì đang bị bỏ qua.



Để xử lý vấn đề trên thì chúng ta có khái niệm **Padding.**

**Padding**

Như ví dụ bên trên mỗi lần thực hiện phép tính convolution thì ta thu được ma trận Y có kích thước đều nhỏ hơn X. Tuy nhiên nếu giờ ta muốn thu được ma trận Y có kích thước bằng ma trận X thì chúng ta cần giải quyết các phần tử ở viền bằng cách thêm giá trị 0 ở viền ngoài ma trận X như sau:



Ma trận X khi thêm padding

Như vậy với cách này chúng ta đã xử lý được các phần tử ngoài viền của X và thu được ma trận Y có kích thước giống với ma trận X. Phép tính này gọi là convolution với **padding=1**. Ta nói padding=k nghĩa là thêm k vector 0 vào mỗi phía của ma trận.

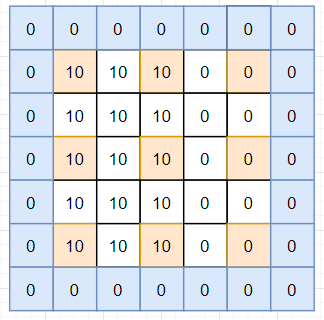
**Stride**

Như ví dụ bên trên ta thực hiện tuần tự các phàn tử trong ma trận X với ma trận W ta thu được ma trận Y có cùng kích thước với ma trận X ta gọi đó là convolution với **stride=1**



stride=1, padding=1

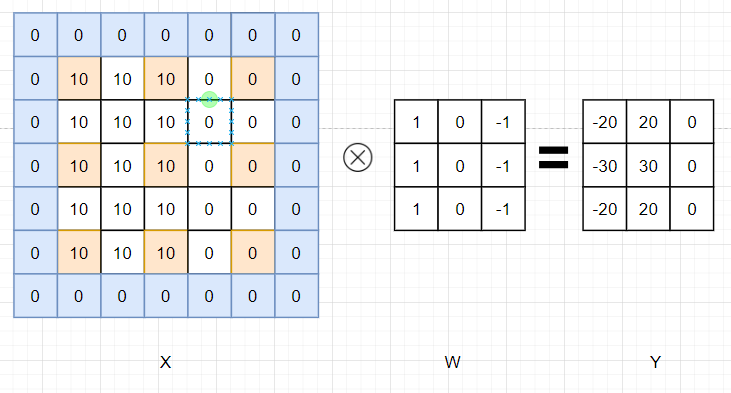
Tuy nhiên nếu **stride=k** (k > 1) thì ta chỉ thực hiện phép convolution trên các phần tử . Ví dụ với k = 2 ta có:



stride=2, padding=1

Để dễ hiểu thì chỉ đơn giản là chúng ta bắt đầu từ vị trí sau đó nhảy k bước theo chiều dọc và ngang cho đến khi hết ma trận X (các vị trí màu đỏ trong hình trên).

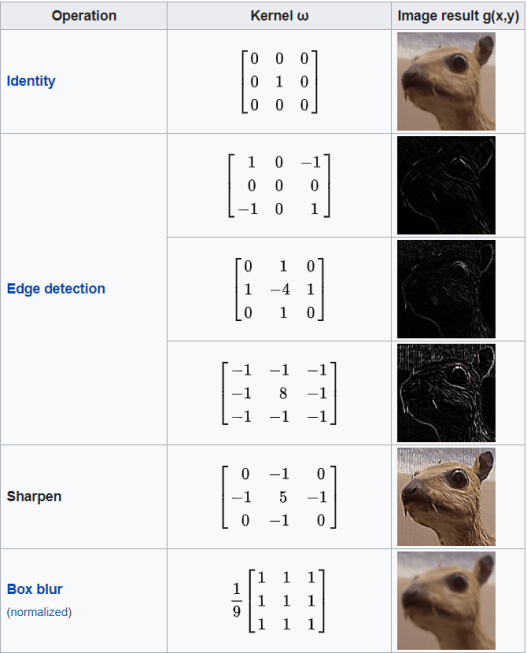
Khi đó ta thu được ma trận Y có kích thước là 3\*3 giảm đi đáng kể so với ma trận X.



Công thức tổng quát cho phép tính convolution của ma trận X có kích thước m\*n với kernel có kích thước k\*k, stride=s, padding=p ra ma trận Y có kích thước là

**Ý nghĩa của phép tính convolution**

Mục đích của phép tính convolution trên ảnh là làm mờ, làm nét ảnh, xác định các đường chính trong ảnh,... Mỗi kernel khác nhau sẽ có những ý nghĩa khách nhau. Ví dụ:



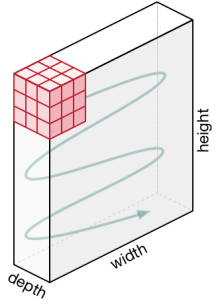
Nguồn: Kernel image processing – Wiki

**Convolutional Neural Network**

Như đã nói ở trên khi xử lý ảnh thì mô hình Neural Network cổ điển sẽ khiến cho số lượng parameters được tạo ra là rất lớn và không có được các đặc trưng của từng ảnh.

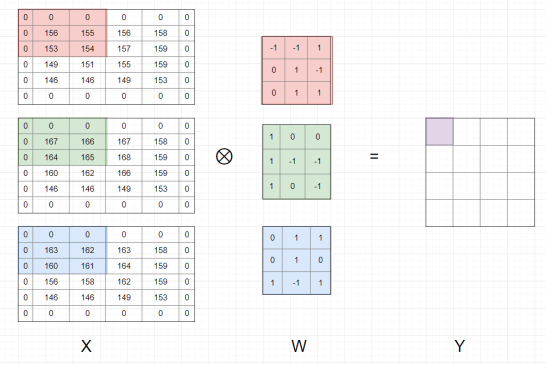
Áp dụng convolution vào layer trong Neural Network ta có thể giải quyết được vấn đề có số lượng lớn parameters mà vẫn lấy ra được các đặc trưng của ảnh.

Ví dụ bên trên là ví dụ convolution với ảnh xám. Tuy nhiên đối với ảnh màu có 3 channel red, green, blue thì khi biểu diễn chúng ta sẽ phải biểu diễn dưới dạng tensor 3 chiều. Nên khi định nghĩa kernel cũng phải là 1 kernal 3 chiều có kích thước k\*k\*3.

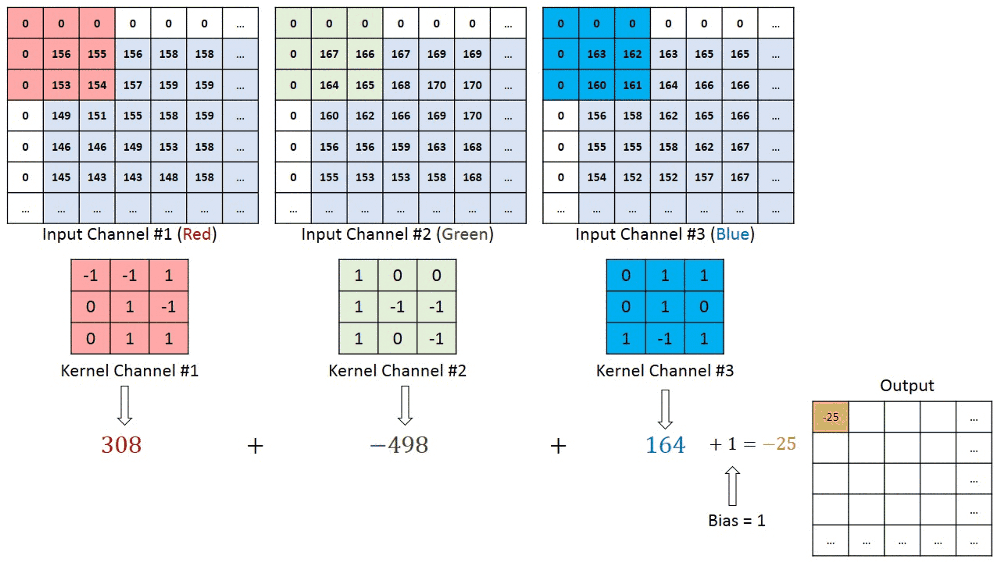


Phép tính convolution trên ảnh màu k = 3

Ta định nghĩa kernel có cùng độ xâu (depth) với input và thực hiện di chuyển khối kernel tương tự như trên ảnh xám.

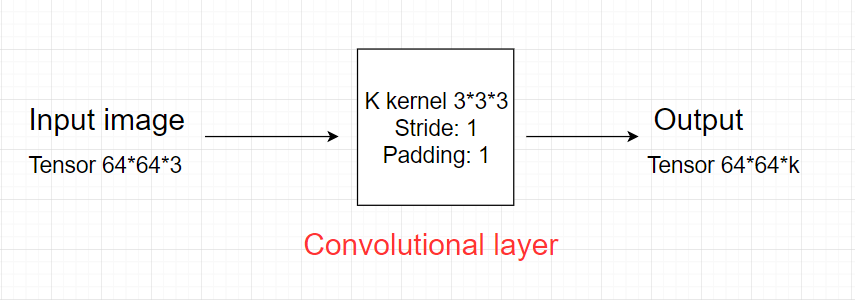


Khi biểu diễn tensor ngoài 2 chỉ số là i và j thì ta cần thêm chỉ số chiều xâu k. Nên mỗi phần tử trong ma trận X sẽ là .



Tương tự với các phần tử còn lại ta sẽ thu được ma trận output.

Với mỗi kernel khác nhau ta sẽ học được những đặc trưng khác nhau của ảnh, nên trong mỗi convolutional layer ta sẽ sử dụng nhiều kernel để có thể “học” được nhiều thuộc tính của ảnh như các nét dọc, nét ngang, nét cong,.. Vì mỗi kernel cho ra output là 1 matrix nên k kernel sẽ cho ra k output matrix. Ta kết hợp k output matrix này lại tạo thành 1 tensor 3 chiều có chiều sâu k.



Output của convolutional sẽ tương ứng là input của convolutional tiếp theo.

**Tổng quát**

Giả sử ta có input của 1 convolutional layer là 1 tensor có kích thước H \* W \* D. Kernel có kích thước F \* F \* D, stride S và padding P.

Ta có convolutional layer áp dụng K kernal sẽ sinh ra output là 1 tensor 3 chiều có kích thước: .

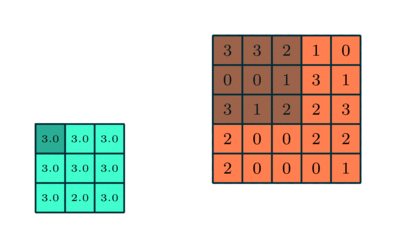


Output của convolutional layer sẽ phải qua áp dụng activation function trước khi trở thành input cho convolutional layer tiếp theo.

**Pooling layer**

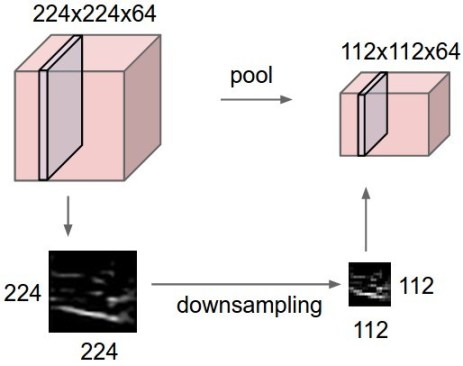
Pooling layer thường được sử dụng giữa các convolutional layer nhằm mục đích giảm kích thước dữ liệu nhưng vẫn giữ được các tính chất quan trọng của ảnh. Kích thước dữ liệu giảm giúp giảm tránh được việc tính toán trong model.

Gọi pooling size có kích thước K\*K. Input của pooling layer có kích thước H\*W\*D, ta tách ra làm D ma trận có kích thước H\*W. Với mỗi ma trận trên vùng kích thước K\*K ta tìm maximum hoặc average dữ liệu trong mỗi ma trận khi đấy ta có được ma trận output. Quy tắc về stride và padding giống như việc áp dụng kernel bên trên.



Max pooling layer

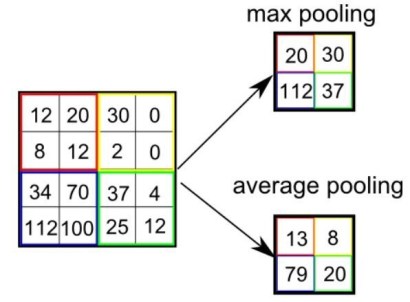
Tuy nhiên, phổ biến nhất thì người ta thường dùng pooling layer với size là (2,2), stride=2, padding=0. Khi đó output sẽ giảm đi 1 nửa và giữ nguyên chiều sâu.



Nguồn: nttuan8.com

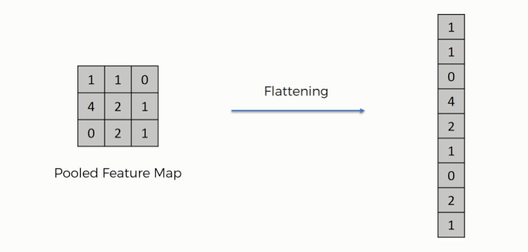
Có 2 loại pooling layer phổ biến nhất đấy là max pooling và average pooling.

* Max pooling: lấy ra phần tử lớn nhất trong ma trận
* Average pooling: lấy ra trung bình của các phần tử trong ma trận.



**Fully connected layer**

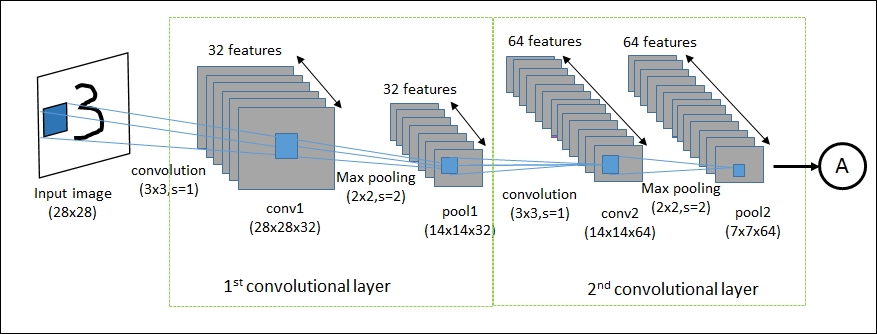
Sau khi ảnh được truyền qua nhiều convolutional layer và pooling layer thì model của chúng ta đã học được rất nhiều đặc điểm của ảnh thì khi ấy tensor của output layer cuối cùng có kích thước H\*W\*D sẽ được chuyển về 1 vector có kích thước là H\*W\*D.



Sau đó chúng ta sẽ dùng các fully connected layer để kết hợp các đặc điểm của ảnh để ra được output của model tương tự như với mô hình Neural Network cổ điển.

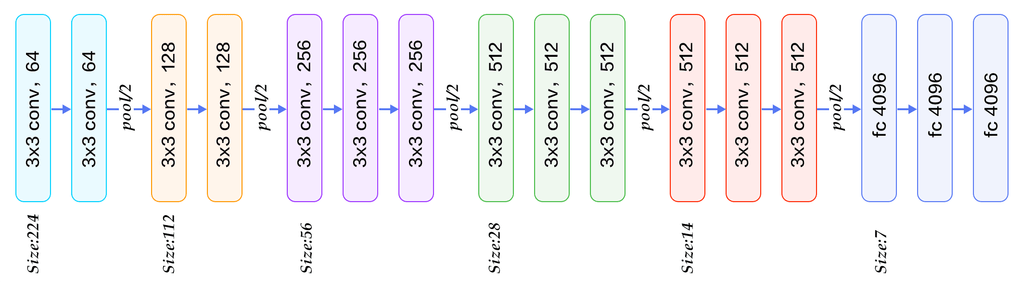
**Một số mô hình CNN phổ biến**

* **Mạng Lenet5**: là một mô hình được phát triển bởi Yann Lecunn (Director của AI Research Facebook) cùng với Le1on Bottou, Yoshua Bengio và Patrick Haffner



Nhận xét:

* + Ảnh đi qua 2 convolutional layer.
  + Từ input là ảnh 28x28 đi qua convolutional layer đầu tiên với 32 kernel 3x3, s=1 và max pooling 2x2, s=2 ta thu được tập các layer gồm 32 layer 14x14.
  + Đi qua convolutional layer thứ 2 với 64 kernel 3x3 ta thu được conv2 14x14x64. Sau đó áp dụng max pooling 2x2, s=2 ta thu được ouput là 64 layer 7x7.
  + Model lúc này đã học được rất nhiều đặc điểm và kích thước dữ liệu giờ cũng đã khá nhỏ, ta áp dụng fully connected layer và tìm ra output của bài toán.
* **Mạng VGG 16**: là mạng CNN được đề xuất bởi K. Simonyan và A. Zisserman, University of Oxford. Model sau khi train bởi mạng này đạt độ chính xác rất cao lên đến 92.7%, là mạng đạt top 5 test trong dữ liệu ImageNet gồm 14 triệu hình ảnh thuộc 1000 lớp khác nhau. Giờ áp dụng kiến thức ở trên để phân tích mạng VGG 16.



Nhận xét:

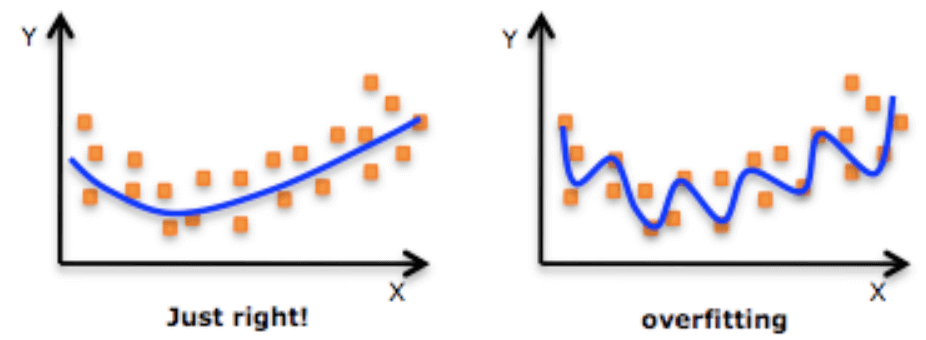
* + Convolutional layer kích tước 3x3. Mặc định khi không ghi config của stride và padding thì sẽ là padding=1, stride=1 để cho output dùng width và height với input.
  + Pool/2 là max pooling layer với size 2x2
  + 3x3 conv, 64 là 64 kernal đượ áp dụng trong layer đấy.
  + Càng các conv layer sau thì kích thước width, heigth càng giảm nhưng depth càng tăng.
  + Sau rất nhiều conv layer và pooling layer thì dữ liệu được flatten và cho vào fully connected layer để tìm ra output.

## Optimization

### Overfitting/underfitting

**Overfitting (sự quá khớp)**

Overfitting là một hiện tượng thường gặp phải trong quá trình training model. Khi mô hình chúng ta xây dựng được thể hiện được hết chi tiết của bộ dữ liệu training. Nghĩa là cả những dữ liệu nhiễu đều được chọn lọc và học để xây dựng model. Do đó bị ảnh hưởng rất nhiều đến độ chính xác của mô hình nói chung.

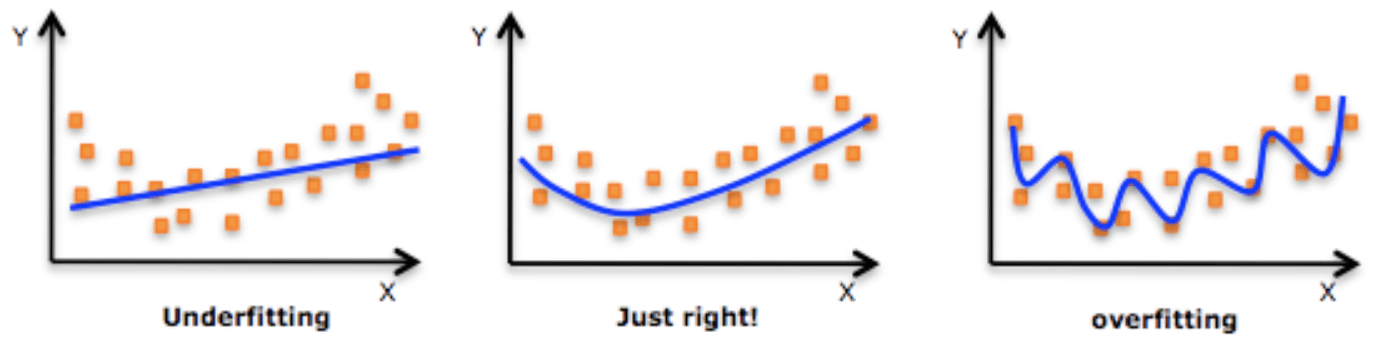


Overfitting

Để khắc phục tình trạng này thì chúng ta cần giảm số lượng feature không cần thiết hoặc áp dụng 1 số kĩ thuật như regularization, dropout hay early stopping.

**Underfitting (sự chưa khớp)**

Underfitting là hiện tượng khi xây dựng model chưa có độ chính xác cao trong tập dữ liệu training cũng như tổng quát hóa với toàn bộ dữ liệu. Để khắc phục tình trạng này thì chúng ta cần thay đổi thuật toán hoặc bổ xung thêm dữ liệu đầu vào.



Underfitting

**Regularization**

* **L1 norm (L1 regularization)**:

Phương pháp này cộng thêm 1 tham số vào hàm mất mát. Số hạng này càng lớn thì mô hình càng phức tạp. Hàm mất mát mới lúc này được gọi là **regularized loss function** và được định nghĩa như sau:

λ là 1 tham số mà ta sẽ tự chọn, nếu tham số này càng lớn thì hàm mất càng phụ thuộc vào biểu thức regularized, nếu tham số càng nhỏ thì hàm mất mát sẽ ít phụ thuộc vào biểu thức regularized. L1 norm sẽ làm cho các trọng số của các W không cần thiết giảm dần về 0. L1 còn có tên gọi khác là **Lasso regression**.

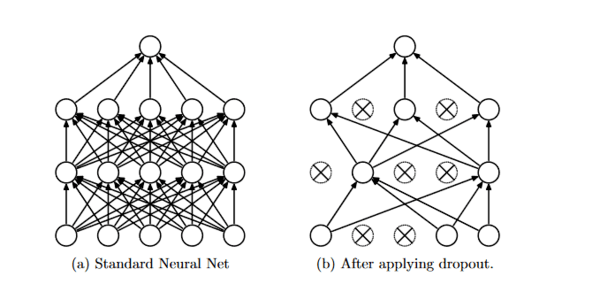
* **L2 norm (L2 regularization)**:

Ở L2 norm thì ta tính tổng bình phương các trọng số W được tính vào hàm mất mát. L2 sẽ hiệu quả hơn so với L1 ở mặt tính toán đối với các trường hợp có nhiều W khác 0. L2 còn có tên gọi khác là **Ridge regression**. Tính chất của λ tương tự đối với L1 norm. Đây là một kĩ thuật được sử dụng nhiều nhất trong Neural Networkd để tránh hiện tượng overfitting.

* **Dropout:**

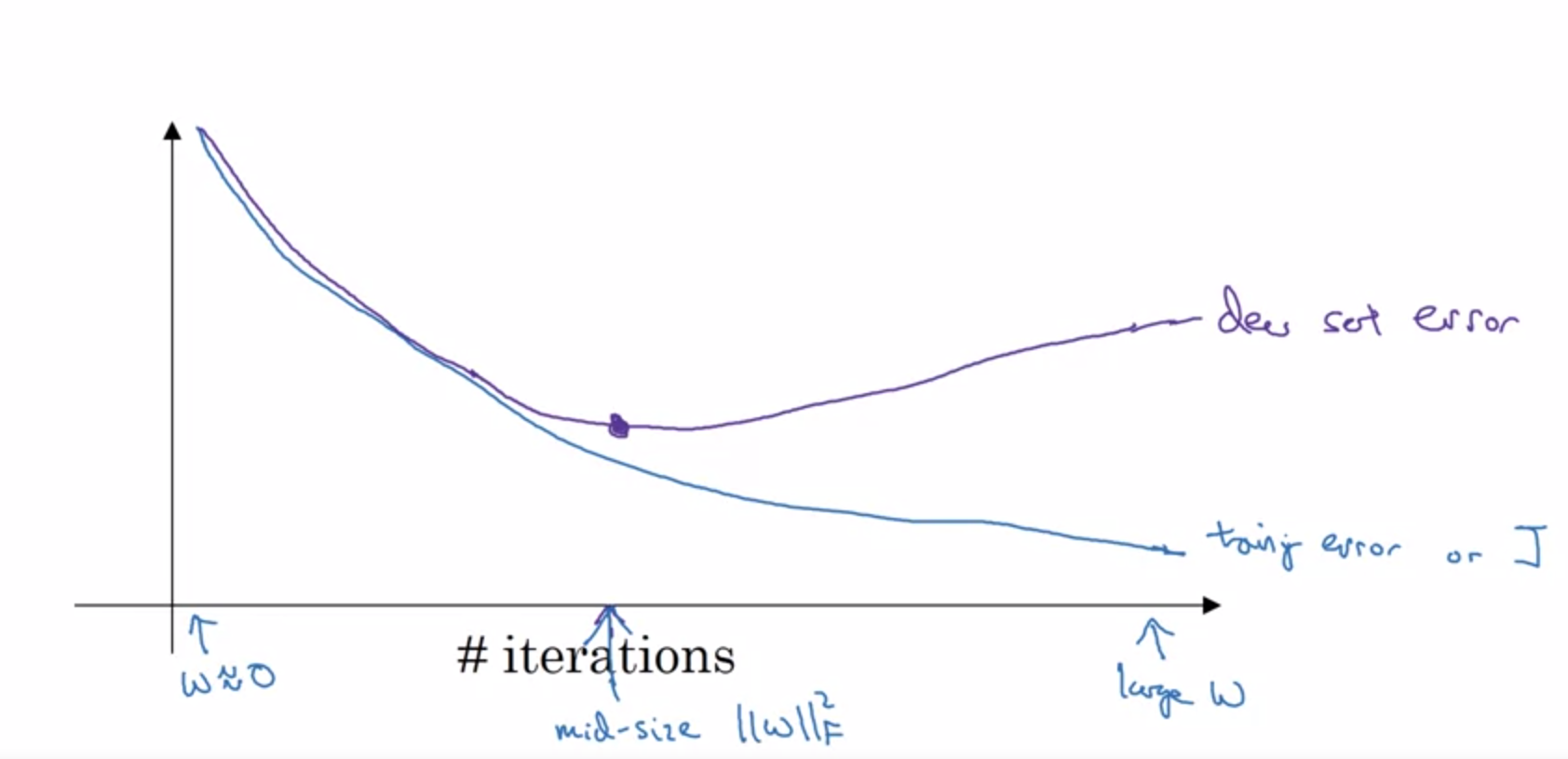
Đây là một phương pháp regularization rất độc đáo được giới thiệu bởi Srivastava et.al.(2014). Phương pháp này chúng ta sẽ bỏ qua một vài node ngẫu nhiên trong 1 lớp.

Mọi node sẽ có một tỉ lệ có thể bị bỏ qua, nếu ta nói drop=0.6 thì sẽ có 60% khả năng node đó sẽ bị bỏ qua trong quá trình huấn luyện. Tỉ lệ này sẽ làm cho mạng Neural bị thay đổi về kích thước, và ta sẽ có mạng mỗi qua mỗi lần train. Kỹ thuật này giải quyết vấn đề overfitting rất tốt tuy nhiên nếu tỉ lệ drop cao thì mô hình sẽ bị underfitting.



* **Early stopping:**

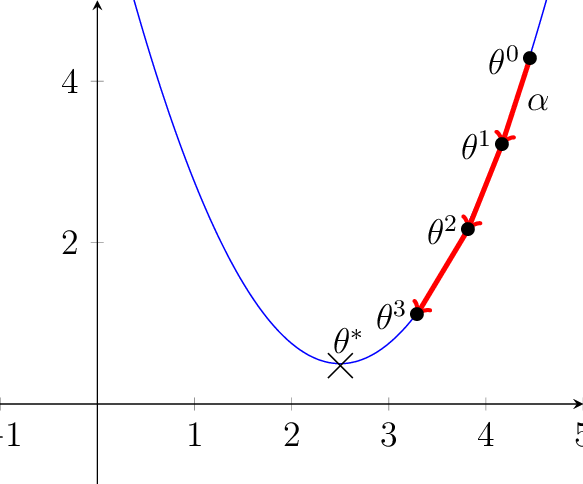
Phương pháp này chúng ta sẽ kiểm tra lại mô hình với dữ liệu training set và dev set (được tách ra từ 1 phần của training set) ở mỗi 1 chu kì mà chúng ta chọn. Khi training set vẫn có xu hướng giảm mà dev set lại có xu hướng tăng thì ta sẽ dừng mô hình lại ở đấy và xử dụng mô hình tương ứng với điểm mà dừng lại.



### Optimizer

**Gradient Descent**

Đây là một thuật toán rất quan trọng cho các bài toán tối ưu. Thuật toán này sẽ tìm ra giá trị nhỏ nhất của 1 hàm số dựa trên việc tính đạo hàm của nó.

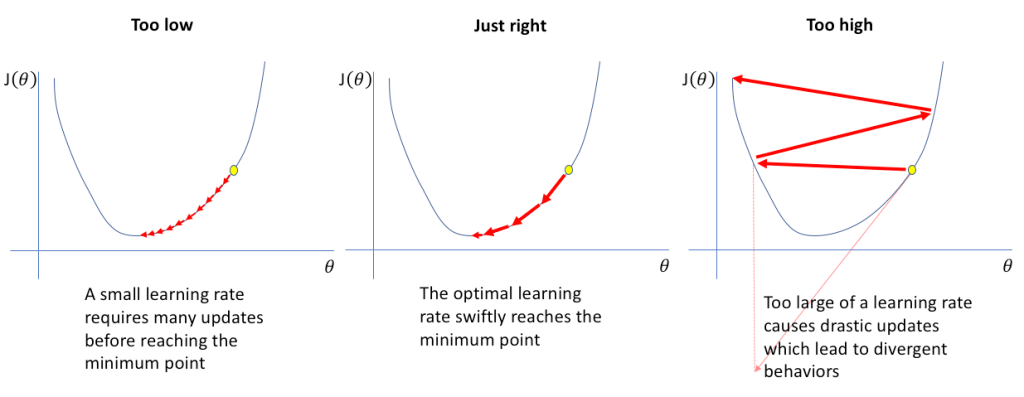


Như hình trên thì điểm cực tiểu của hàm số là vị trí . Gradient Descent có nhiều dạng khác nhau như Stochastic Gradient Descent (SGD), Mini-batch SDG nhưng về cơ bản thì để thực thi được thuật toán thì đều có các bước như sau:

* Bước 1: Khởi tạo giá trị tùy ý.
* Bước 2: Gán với được gọi là *learning rate* và là 1 hằng số không âm (ví dụ ).
* Bước 3: Tính lại . Nếu đủ nhỏ hoặc thỏa mãn điều kiện dừng thì dừng lại. Còn không thì quay ngược lại bước 2.

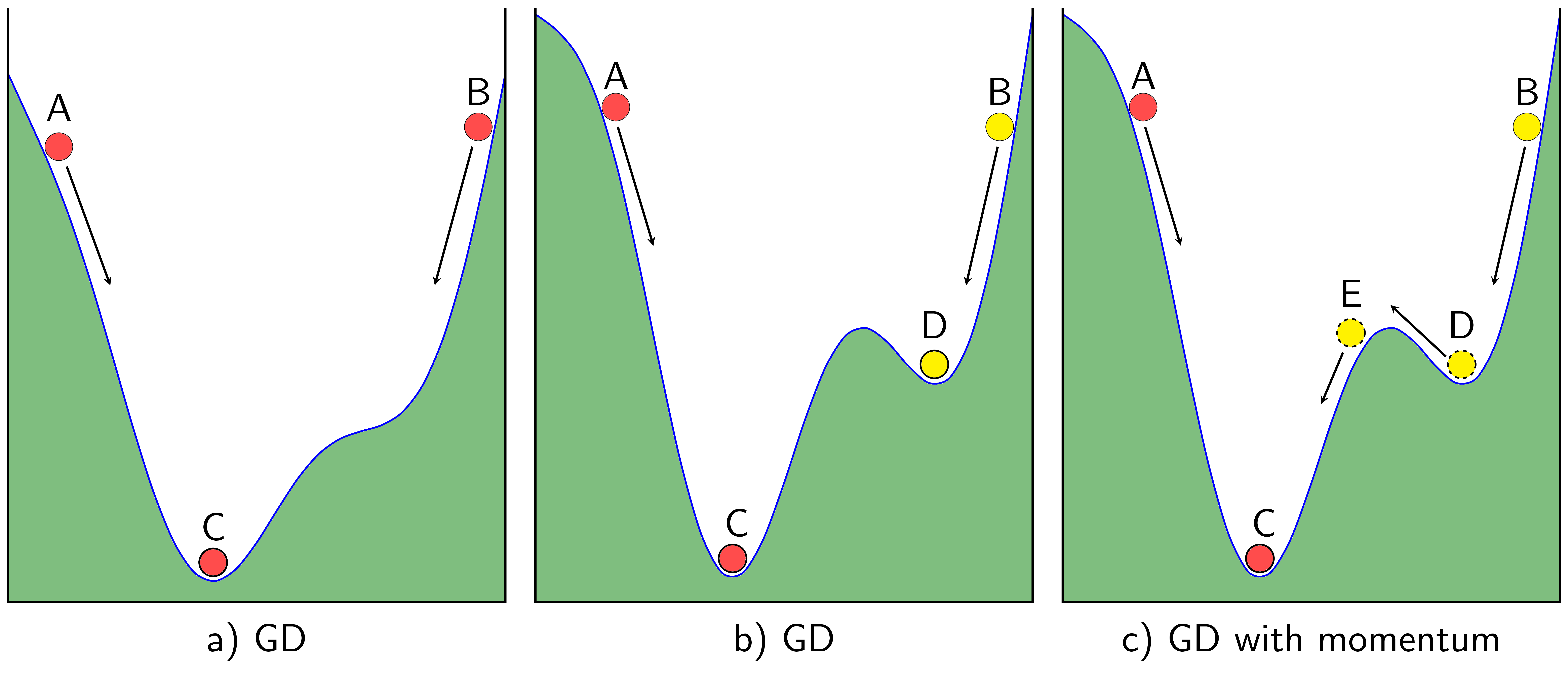
Việc chọn learning rate đóng vai trò rất quang trọng trong thuật toán này. Sẽ có 3 trường hợp xảy ra khi ta chọn learning rate:

* Nếu learning rate quá nhỏ: thì mỗi 1 vòng lặp hàm số sẽ giảm rất ít và cần phải thực hiện lại bước 2 rất nhiều lần để tìm được giá trị nhỏ nhất.
* Nếu learning rate quá lớn: thì sẽ gây ra hiện tượng *overshoot* và không thể tìm được giá trị nhỏ nhất của hàm.
* Nếu learning rate hợp lý: thì việc số lần lặp lại bước 2 sẽ ít và sẽ nhanh tìm được giá trị nhỏ nhất.



3 trường hợp chọn learning rate

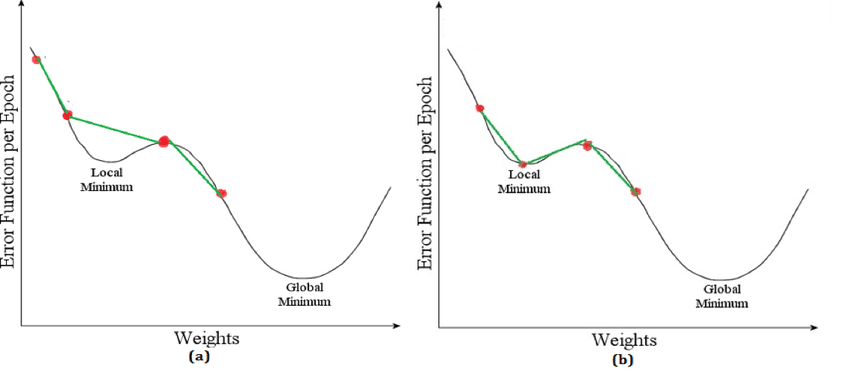
Nếu so sánh Gradient Descent dưới góc nhìn vật lý (theo hình dưới). Ở hình a thì nếu chúng ta thả vật từ vị trí A hay B thì cũng sẽ đều rơi xuống vị trí C. Tuy nhiên ở hình b và c khi thả vật ở vị trí A hay B thì chúng đều có thể rơi xuống 2 vị trí là C hoặc D. Vị trí D ở đây được gọi là *local mininum* là một điểm mà chúng ta không mong muốn.



Tuy nhiên nếu thả ở vị trí B với lực đủ mạnh thì sẽ có đà để lăn tới vị trí E và xuống vị trí C. Dựa trên hiện tượng này người ta ra đời một thuật toán nhằm khắc phục việc GD rơi vào trị trí local minimum không mong muốn. Đấy là thuật toán *Momentum*.

**Momentum**

Là một thuật toán tối ưu nhằm giúp việc triển khai Gradient Descent nhanh hơn. Ý tưởng của momentum là gia tốc học khi cùng hướng với chiều của gradient và giảm tốc học khi ngược hướng với gradient. Khi mommentum của Gradient Descent đủ lớn thì điểm tìm ra được sẽ vượt qua được các điểm local minimum để hướng tới điểm global minimum như hình dưới. Tham số trong thuật toán *momentum* là γ và trong thực nghiệm người ta khuyên nên chọn γ nằm trong khoảng 0.5 -> 0.9.



**RMSprop**

RMSprop khá giống với Gradient Descent with Momentum. Thuật toán RMSprop sẽ hạn chế sự biến thiên của đồ thị theo chiều dọc. Vì vậy, chúng ta có thể tăng tốc độ học và thuật toán này có thể thực hiện các bước nhảy lớn theo chiều ngang để có thể hội tụ nhanh hơn.

Điểm khác nhau giữa RMSprop và GD là việc tính toán độ dốc (gradients). Thay vì fix cứng learning rate như GD thì RMSprop coi learning rate như 1 tham số và thuật toán RMSprop chia learning cho trung bình của bình phương weight trong quá khứ. Nó làm cân bằng bước nhảy mỗi lần chạy, giảm bước nhảy khi gradient lớn và tăng bước nhảy khi gradient nhỏ.

Công thức của thuật toán này là:

Khi gần đến 0 và để tránh việc phương trình bị chia cho 0 thì người ta thường công thêm 1 giá trị ε và ε thường bằng .

**Adam (Adaptive Moment Estimation)**

Thuật toán Adam giống như sự kết hợp của thuật toán Momentum và RMSprop. Nó duy trì trung bình bình phương độ dốc quá khứ giống RMSprop và trung bình độ dốc quá khứ giống Momentum. Công thức của Adam là:

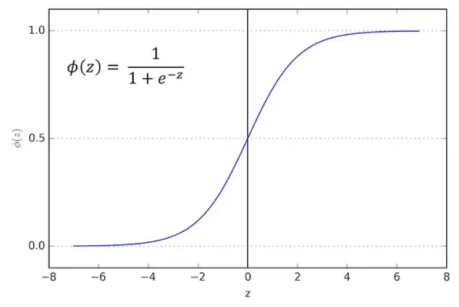
Với t là số lần lặp hiện tại. Thông thường với thuật toán Adam người ta sẽ lấy các hệ số , , ε như sau:

### Activation Functions

**Sigmoid**

Đối với những bài toán Logistic Regression thì output là các giá trị nhị phân như 0/1, true/false. Khi đó đầu ra dự đoán của bài toán thường được viết dưới dạng:

Trong đó Ɵ là logistic function, 1 trong những activation phổ biến khi áp dụng vào bài toán này là **Sigmoid**. Đây là 1 hàm số liên tục, nhận giá trị thực trong khoảng (0,1) và có đạo hàm tại mọi điểm.



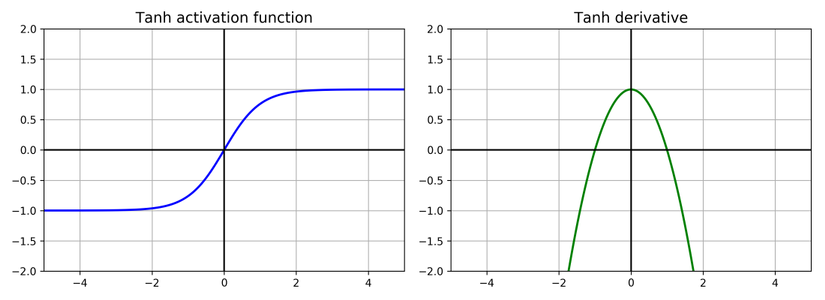
Đồ thị hàm Sigmoid

Phương trình hàm sigmoid khi đạo hàm là:

**Tanh**

Activation function Tanh khá giống với Sigmoid. Các giá trị của hàm sẽ biến thiên trong khoảng (-1,1). Ưu điểm của hàm này là giá trị của nó đã được *zero-centered* giúp cho quá trình optimization thuận tiện hơn, tuy nhiên nó vẫn không tránh được hiện tượng **Vanishing/Exploding gradients**.

Phương trình hàm Tanh và đạo hàm của nó là:

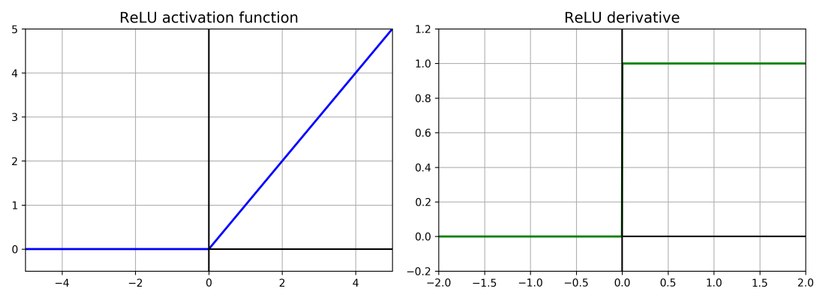


Đồ thị hàm Tanh

**ReLU**

ReLU và các biến thể của nó (Leaky ReLU, RreLU, PreLU, ELU, SELU) thường được sử dụng thay cho Sigmoid, do chúng hoạt động tốt hơn với Neural Network. Cụ thể, khi dùng ReLU chúng ta sẽ tránh được hiện tượng bão hòa đối với các giá trị dương và hàm này chạy khá nhanh trong quá trình tính toán do trong phương trình không tính toán luôn quan đến số thực.

Phương trình ReLU và đạo hàm của nó là:



Đồ thị hàm ReLU

### Initialization

**Xavier and He initialization Techniques**

Một trong những cách để tránh hiện tượng gradients không ổn định là thay đổi kỹ thuật weight initialization thay vì sử dụng random initialization. Chúng ta cần điều chỉnh inputs và outputs trong hai quá trình: **feedforward** khi predictvà **backforward** khi thực hiện backpropagation.

Để thực hiện được việc trên thì chúng ta cần 2 điều kiện chính:

* Với mỗi layer trong mạng, phương sai của inputs và outputs phải có giá trị như nhau.
* Phương sai của gradients trước và sau khi di chuyển qua một layer (quá trình backpropagation) cũng phải có giá trị như nhau.

Tuy nhiên cả 2 điều kiện trên sẽ xảy ra khi số lượng input và outputs cho một layer là như nhau. Trong thực tế chúng ta có một phương pháp khá hiệu quả đó là **Xavier Initialization**. Cụ thế như sau (xét trường hợp dùng sigmoid activation function):

* Xử dụng phân phối chuẩn với kỳ vọng là 0 và độ lệch chuẩn
* Sử dụng phân phối đều trong khoảng –r và –r trong đó

Khi số lượng inputs và outputs gần bằng nhau, hai đại lượng trên được viết gọn lại là:

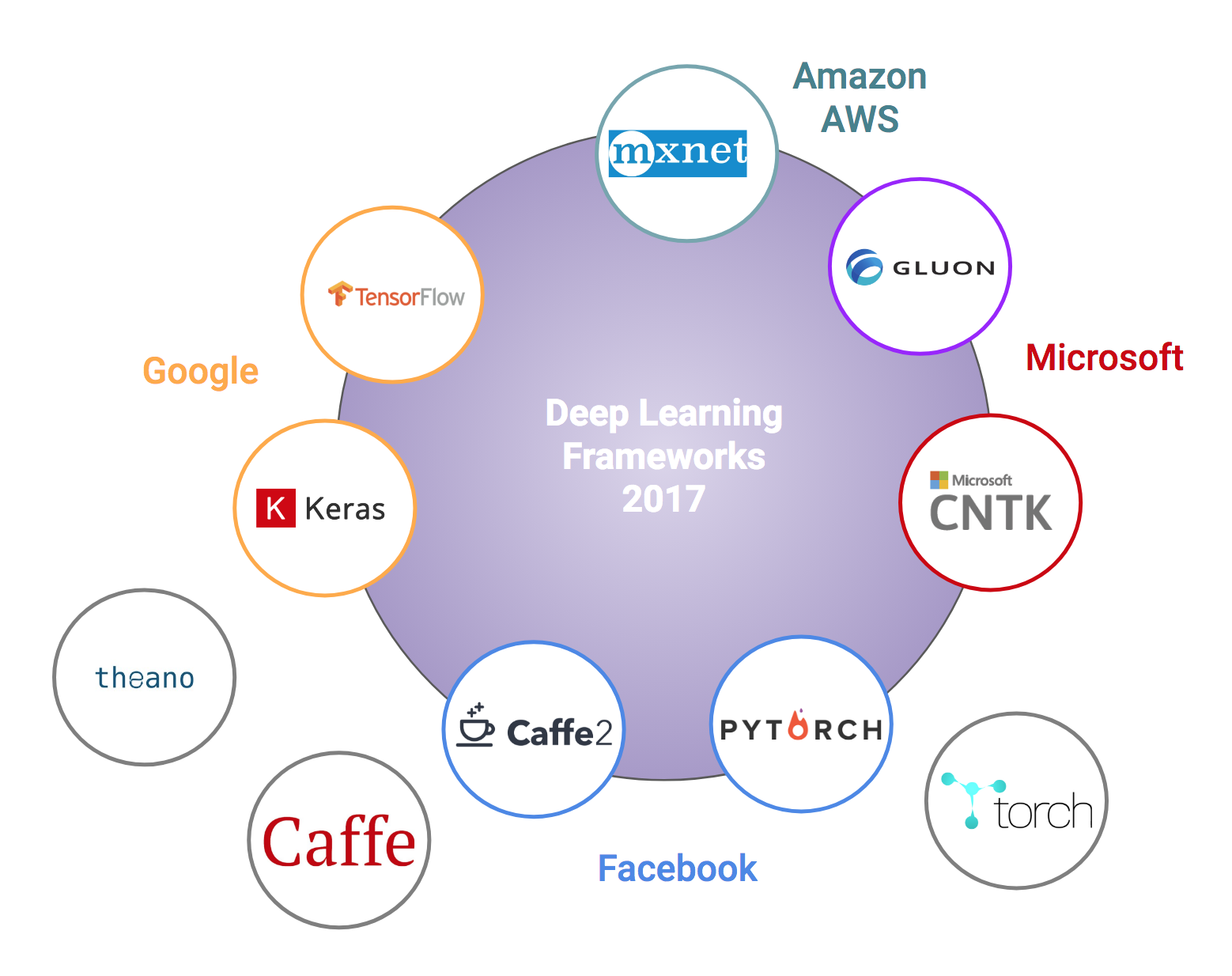
Nếu chúng ta xử dũng ReLU đôi khi kỹ thuật trên còn được gọi là **He** **Initialization**.

# CHƯƠNG III: CÀI ĐẶT THỰC NGHIỆM VÀ ĐÁNH GIÁ

## Giới thiệu sơ qua

### Keras

Kể từ lúc Deep Learning có những bước đột phá lớn, đã có hàng loạt các thư viện hỗ trợ Deep Learning ra đời. Một trong những thư viện đó mà chúng ta sẽ áp dụng vào bài toán là thư viện Keras của Google.



Các thư viện của các hãng công nghệ lớn

Lý do chúng ta chọn Keras là vì những đặc điểm sau (theo bài viết [Why use Keras?](https://keras.io/why-use-keras/)):

* + Keras ưu tiên trải nghiệm của người dùng
  + Keras đã được sử dụng rộng rãi trong doanh nghiệp và cộng đồng nghiên cứu
  + Keras giúp dễ dàng biến các thiết kế thành sản phẩm
  + Keras hỗ trợ training trên nhiều GPU phân tán
  + Keras hỗ trợ đa backend engines và không giới hạn vào một hệ sinh thái

## Hướng dẫn sử dụng

### Source code

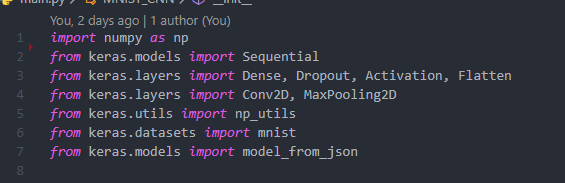
Mã nguồn của project được lưu trữ ở đây:

<https://github.com/noragalvin/dl-basic/tree/master/practice/mnist/python>

Trong đó: file **main.py** chứa toàn bộ code, 2 file **model.json** và **model.h5** là file lưu lại model cho lần training tiếp theo.

### Chi tiết và cách chạy project

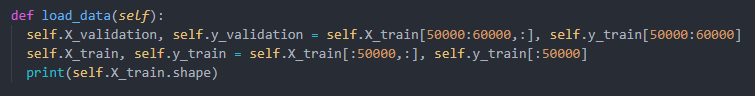
* Đầu tiên chúng ta cần import 1 số thư viện cần thiết như numpy, keras



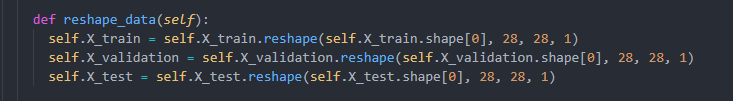
* Trong thư viện Keras có sẵn bộ dữ liệu về MNIST nên chúng ta chỉ cần load dữ liệu ra bằng cách:



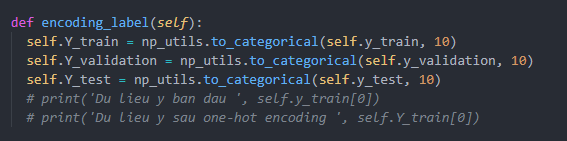
* Ở đây bộ dataset MNIST bao gồm 60.000 training set và 10.000 dataset, tôi sẽ chia ra 50.000 cho training set và 10.000 dữ liệu cho validation set.



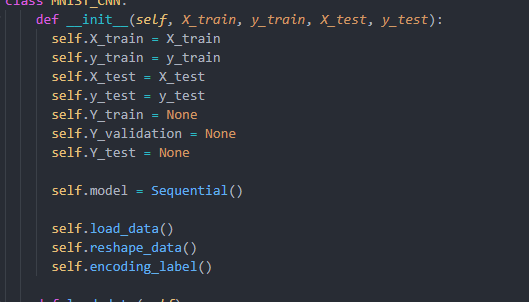
* Dữ liệu input cho mô hình CNN là 1 tensor 4 chiều (N, W, H, D), ở bài toán này là ảnh xám cho nên D = 1, và kích thước ảnh trong dataset là 28px nên W=H=28. Do dữ liệu ảnh có kích thước là 28px nên chúng ta sẽ cần reshape lại dữ liệu thành N\*28\*28\*1 để giống với kích thước mà Keras yêu cầu.



* Áp dụng one-hot encoding chúng ta chuyển đổi label của training set, validation set và test set. Ví dụ với label = 5 sau khi one-hot encoding chúng ta có được [0, 0, 0, 0, 0, 1, 0, 0, 0, 0].



* Sau đấy chúng ta định nghĩa model cho mô hình CNN. Đầu tiên tôi định nghĩa model là 1 Sequential() nghĩa là để cho Keras hiểu ta sẽ xếp các layer chồng lên nhau để tạo model. Ví dụ: input -> conv -> conv -> pool -> flatten -> fully connected layer -> output



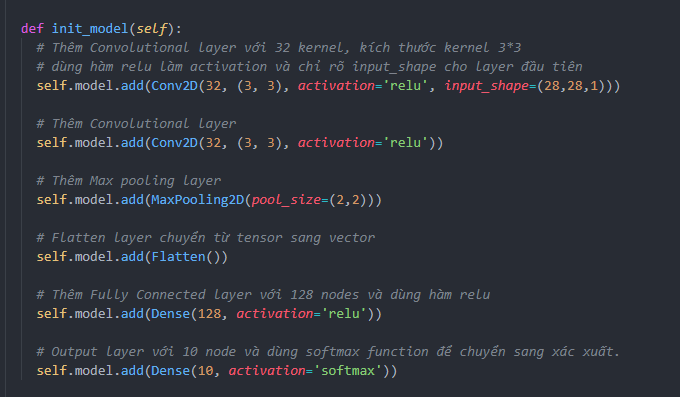
* Ở bài này sau quá trình training nhiều lần tôi quyết định dùng mô hình bao gồm conv -> conv -> max pool -> flatten -> fc -> output.
  + Ở layer đầu tiên cần chỉ rõ input\_shape của ảnh là (W, H, D). Do input là ảnh xám nên input\_shape = (28,28,1).
  + Do hàm relu có performance tốt hơn sigmoid nên tôi quyết định chọn relu làm activation function.
  + Khi thêm Convolutional Layer chúng ta cần chỉ rõ các tham số như số lượng layer K, kernel size (W, H), activation function với cấu trúc:

Conv2D(K, (W, H), actionvation=’activation\_function\_name’)

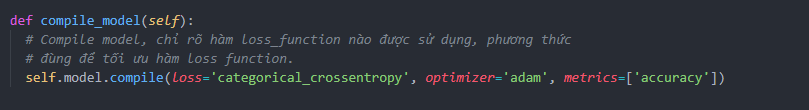
* + Khi thêm Max pooling Layer thì cần chĩ rõ kernel size bằng cách:

MaxPooling2D(pool\_size=(W,H))

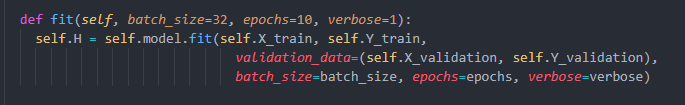
* + Bước Flatten là để chuyển từ tensor sang vector để fully connection. Ví dụ shape: 20x20 khi đi qua layer này sẽ là 400x1
  + Để thêm Fully Connected Layer ta cần chỉ rõ số lượng node trong layer và activation function được sử dụng trong layer.
  + Cuối cùng chúng ta dùng softmax activation function với output layer để chuyển output sang xác xuất.



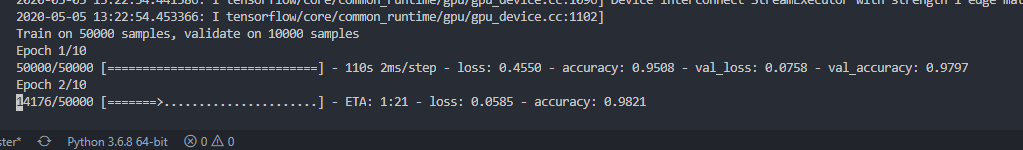
* Sau đấy chúng ta compile model, ở đây cần chỉ rõ loss function nào được sử dụng và phương thức dùng để tối ưu loss function. Ở đây tôi chọn loss function là categorical\_crossentropy và hàm tối ưu cho loss function là adam:
  + Categorical\_crossentropy: thường dùng trong classifier nhiều class.
  + Adam: là 1 thuật toán optimizer



* Tiếp theo thực hiện train model với data mnist đã lấy ra ở bên trên với batch\_size=32, epochs=10 và verbose=1

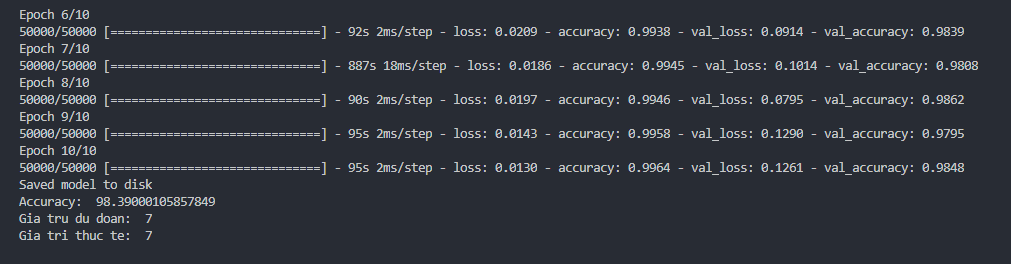


* Khi đó chương trình sẽ train model qua 10 epoch, ở mỗi epoch chúng ta có thể thấy được sự thay đổi của loss và accuracy như trong hình



## Kết quả đạt được

* Khi train model xong chúng ta thu được model có độ chính xác sấp xỉ 98%, và test thử predict 1 ảnh bất kì và kết quả cũng rất chính xác.



## Kết luận

Sau quá trình tìm hiểu về Deep Learning và các thuật toán cơ bản để có thể áp dụng được vào xử lý bài toán Nhận dạng chữ số viết tay. Từ đó tôi đã tìm được 1 model có độ chính xác 98%.

## Tài liệu tham khảo