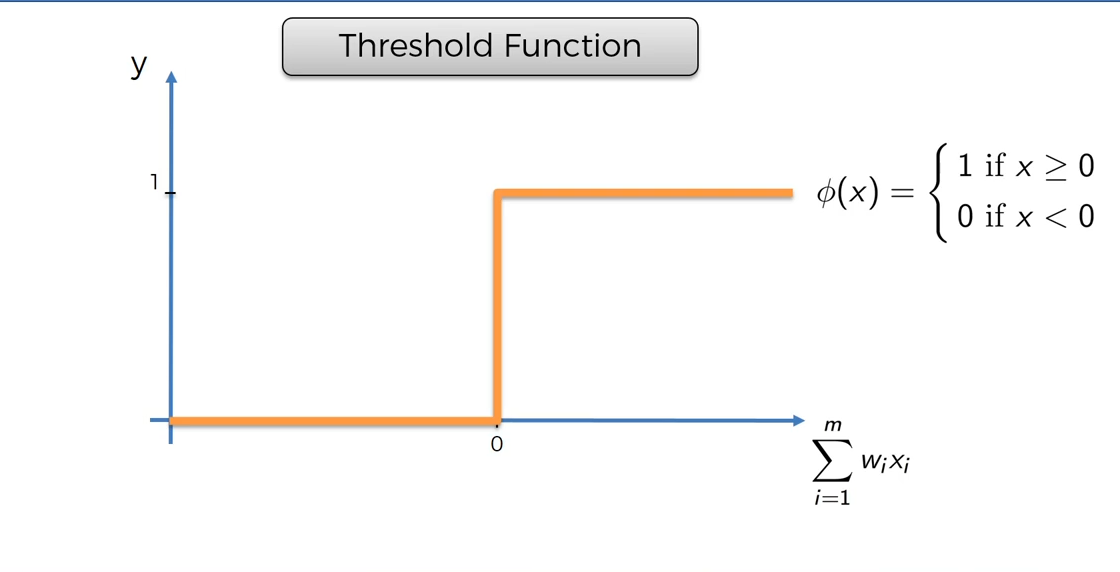
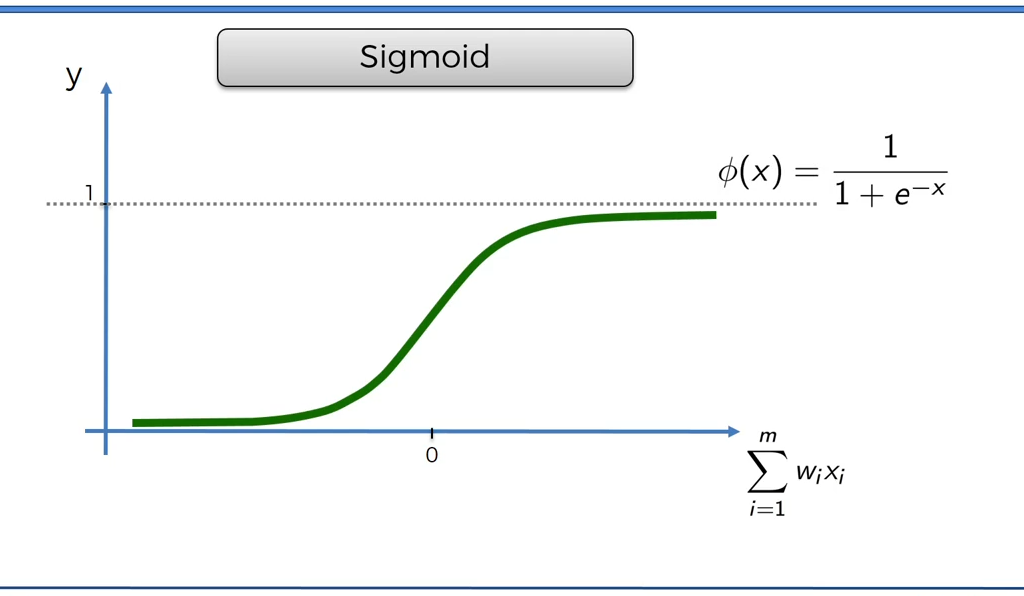


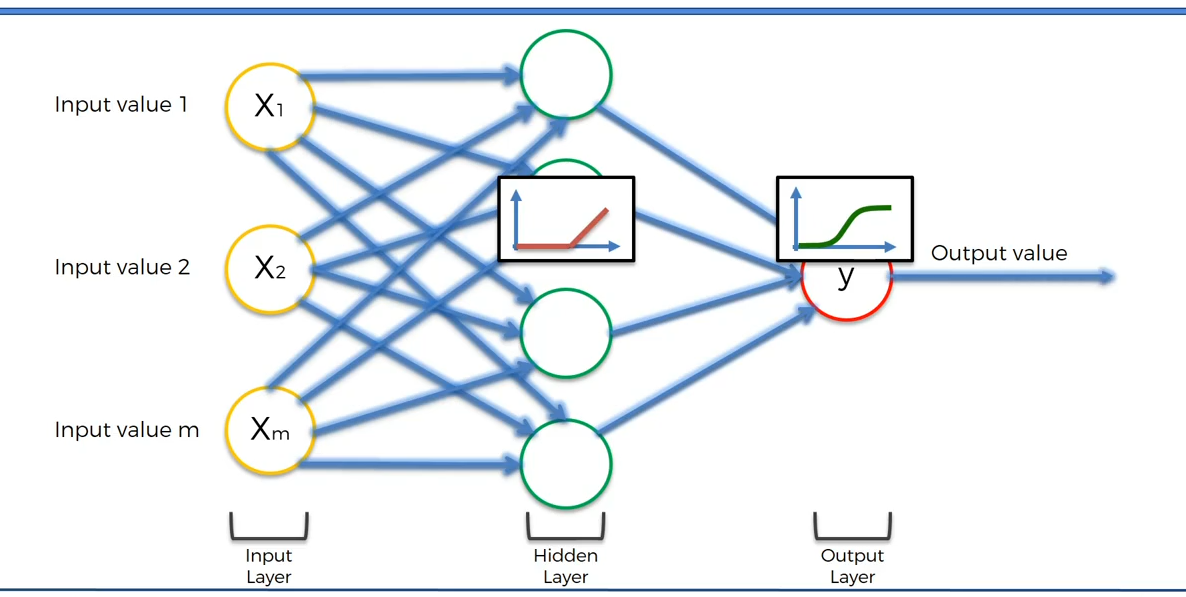
Activation functions(hàm kích hoạt)

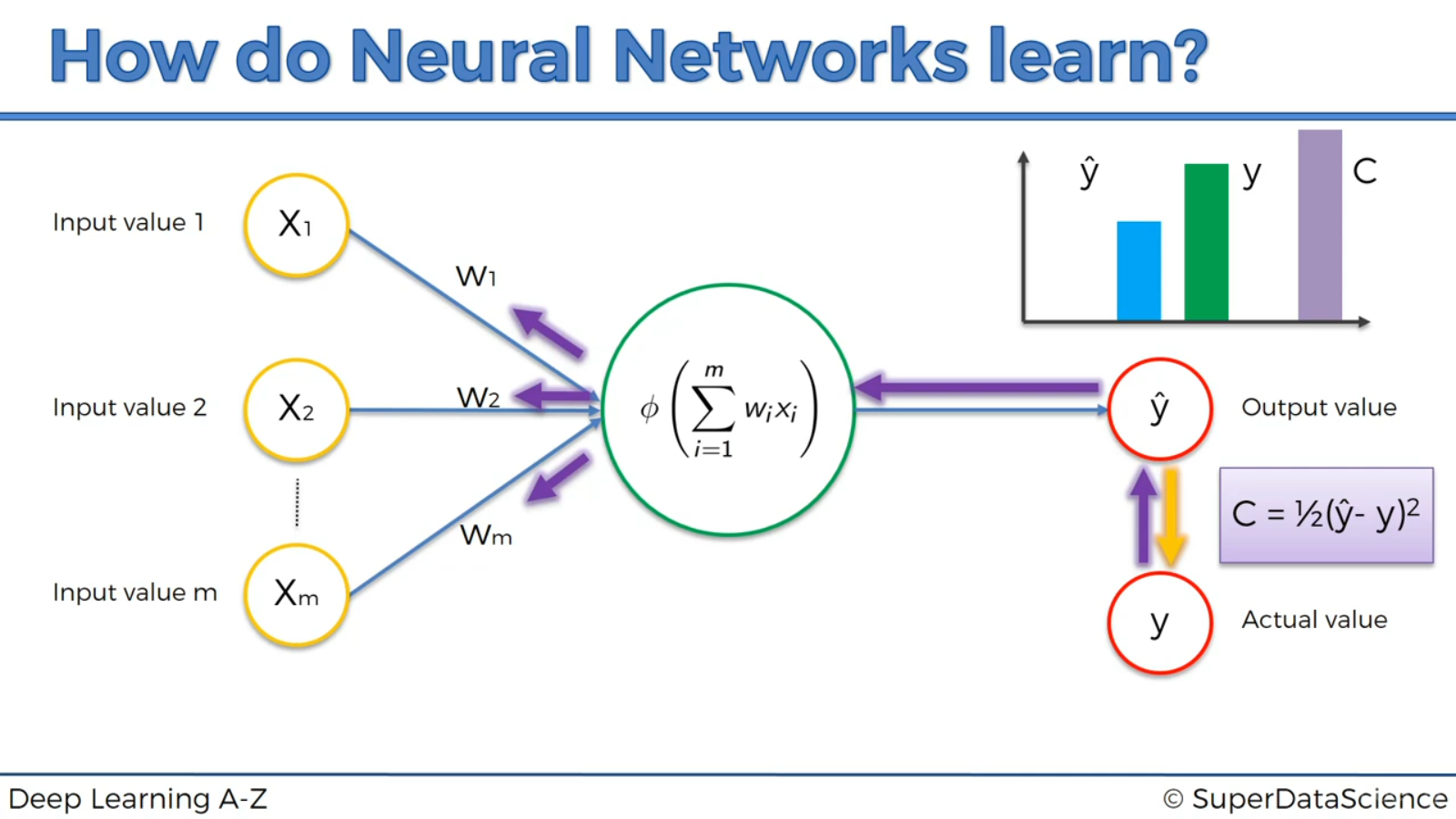


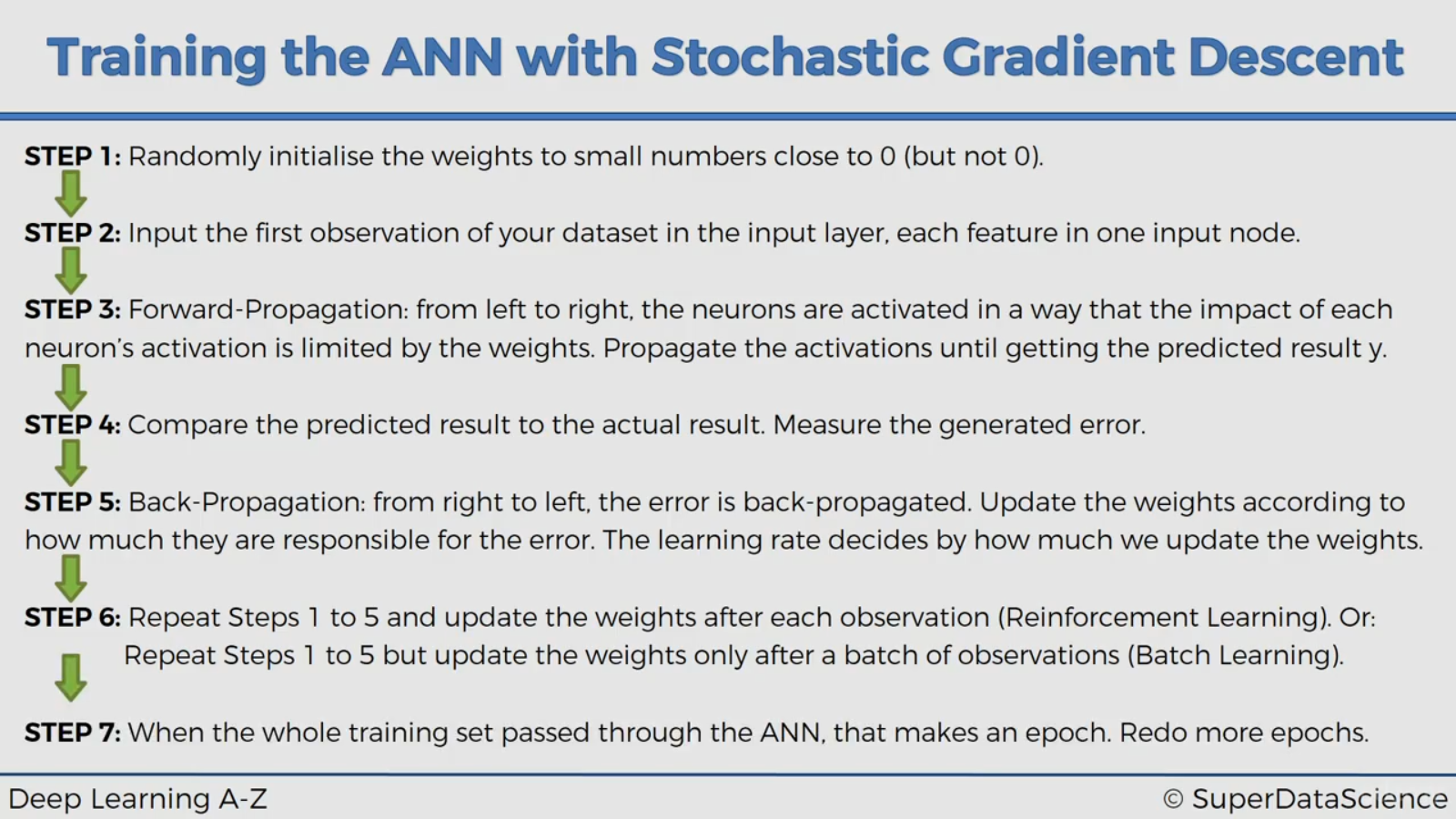










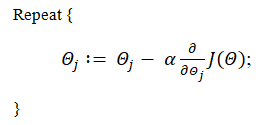


* Deep Learning sử dụng một mạng nơ ron để bắt chước trí thông minh của động vật (giống hành động của vật).
* Có 3 loại layer chính của các nơ ron trong mạng nơ ron là: **Input layer**, **Các hidden layer**, **Output layer**.
* **Input layer** nhận các dữ liệu đầu vào, Các **hidden layer** thực hiện các phép tính toán cho các đầu vào. Thử thách lớn nhất trong việc tạo mạng nơ ron là quyết định số lượng các hidden layer này, cũng như số các nơ ron cho mỗi layer, **Output layer** trả về dữ liệu đầu ra.
* Mối liên kết giữa nơ ron được kết hợp với một trọng số, nó chỉ ra được tầm quan trọng của giá trị đầu vào.
* Các nơ ron áp dục một Hàm kích hoạt (Activation function) trên dữ liệu để chuẩn hóa đầu ra cho nơ ron.
* Khi ta đã hoàn thành với tập dữ liệu, ta có thể tạo một hàm hiển thị độ sai của đầu ra của AI so với đầu ra thực tế. Hàm này được gọi là **Cost Function** . Một cách hiểu đơn giản rằng, ta muốn Cost Function của ta sẽ trả về 0, khi đó đầu ra của AI cũng sẽ giống với đầu ra thực tế từ tập dữ liệu.
* Để huấn luyện một mạng nơ ron, bạn cần một tập dữ liệu lớn.
* Việc lặp lại tập dữ liệu và so sánh các đầu ra sẽ sinh ra cost function giúp chỉ ra sai sót của AI so với đầu ra thực tế.
* Sâu mỗi vòng lặp trong tập dữ liệu, trọng số weight giữa nơ ron sẽ được điều chỉnh bằng Gradient Descent để giảm cost function.
* **Làm thế nào để làm giảm cost function?**

Chúng ta sẽ sử dụng một công nghệ gọi là Gradient Descent.

Gradient Descent là công nghệ cho phép ta tìm giá trị nhỏ nhất của một hàm. Trong trường hợp của ta, chúng ta đang tìm giá trị nhỏ nhất cho cost function. Nó hoạt động bằng cách thay đổi một giá trị rất nhỏ cho trọng số sau mỗi lần lặp trong tập dữ liệu. Bằng cách tính toán đạo hàm của cost function ở một tập trọng số, ta có thể tìm được hướng của cực tiểu.

Để tìm cực tiểu của *J*(Θ) ta áp dụng thuật toán Gradient Descent.



Với α là learning rate.  
Để thực hiện được thì cần phải tính được  ∂/Θ*j*​∂​*J*(Θ),để tính được đạo hàm này là việc tương đối khó và ta cần thực hiện một thuật toán được gọi là backpropagation để tính.

Mục tiêu của back propagation(**lan truyền ngược** ) là đi tính ∂/Θ*j*​∂​*J*(Θ)

Như vậy nhờ vào back propagation ta đã có thể tính được ∂/Θ*j*​∂​*J*(Θ) từ đó giúp thuật toán Gradient descent có thể hoạt động và ta có thể tìm ra tập Θ sao cho Cost function là nhỏ nhất.

**Nơron Network**

mạng NN được hình thành từ các tầng nơ-ron nhân tạo. Mạng NN gồm 3 kiểu tầng chính là **tầng vào** (input layer) biểu diễn cho đầu vào, **tầng ra** (output layer) biểu diễn cho kết quả đầu ra và **tầng ẩn** (hidden layer) thể hiện cho các bước suy luận trung gian. Mỗi nơ-ron sẽ nhận tất cả đầu vào từ các nơ-ron ở tầng trước đó và sử dụng một **hàm kích hoạt dạng**(activation function) phi tuyến như sigmoid, ReLU, tanh để tính toán đầu ra.

Quá trình suy luận từ tầng vào tới tầng ra của mạng NN là quá trình **lan truyền tiến** (feedforward), tức là đầu vào các nơ-ron tại 1 tầng đều lấy từ kết quả các nơ-ron tầng trước đó mà không có quá trình suy luận ngược lại.

Để tối ưu được hàm lỗi *J*(W) này người ta sử dụng giải thuật **lan truyền ngược** (backpropagation) để tính được đạo hàm của hàm lỗi này.

 \* Hàm chuyển đổi phi tuyến được sử dụng phổ biến trong ANN là **sigmoid(**bởi nó dễ lấy đạo hàm, do đó có thể giảm đáng kể tính toán trong quá trình huấn luyện. Hàm này đƣợc ứng dụng cho các chƣơng trình ứng dụng mà các đầu ra mong muốn rơi vào khoảng [0,1].**) (logical activation) function.**

Kết quả của Sigmoid Function thuộc khoảng [0,1] nên còn gọi là hàm chuẩn hóa (Normalized Function).

**http://bis.net.vn/photos/storage/20110612161354813.png**

**2. Quá trình học (Learning Processing) của ANN**

ANN được huấn luyện (Training) hay được học (Learning) theo 2 kỹ thuật cơ bản đó là học có giám sát (Supervised Learning) và học không giám sát (Unsupervised Learning).

**Supervised learning**: Quá trình Training được lặp lại cho đến kết quả (output) của ANN đạt được giá trị mong muốn (Desired value) đã biết. Điển hình cho kỹ thuật này là mạng Neuron lan truyền ngược (**Backpropagation**).

**Unsupervised learning:** Không sử dụng tri thức bên ngoài trong quá trình học (Learning), nên còn gọi là tự tổ chức (Self – Organizing). Mạng Neuron điển hình được huấn luyện theo kiểu Unsupervised là Sefl – Organizing Map (SOM).

Trong quá trình huấn luyện mạng, thuật ngữ **“epoch”** được dùng để mô tả quá trình khi tất cả các input patterns của training set được đưa để huấn luyện mạng. Nói cách khách 1 epoch được hoàn thành khi tất cả các dữ liệu trong training set được đưa vào huấn luyện mạng. Vì vậy số lượng **“epoch”** xác định số lần mạng được huấn luyện (hay số lần đưa tất cả các dữ liệu trong training set vào mạng).

**Quá trình học của Supervised ANN được mô tả như sau:**

1.Tính giá trị output .

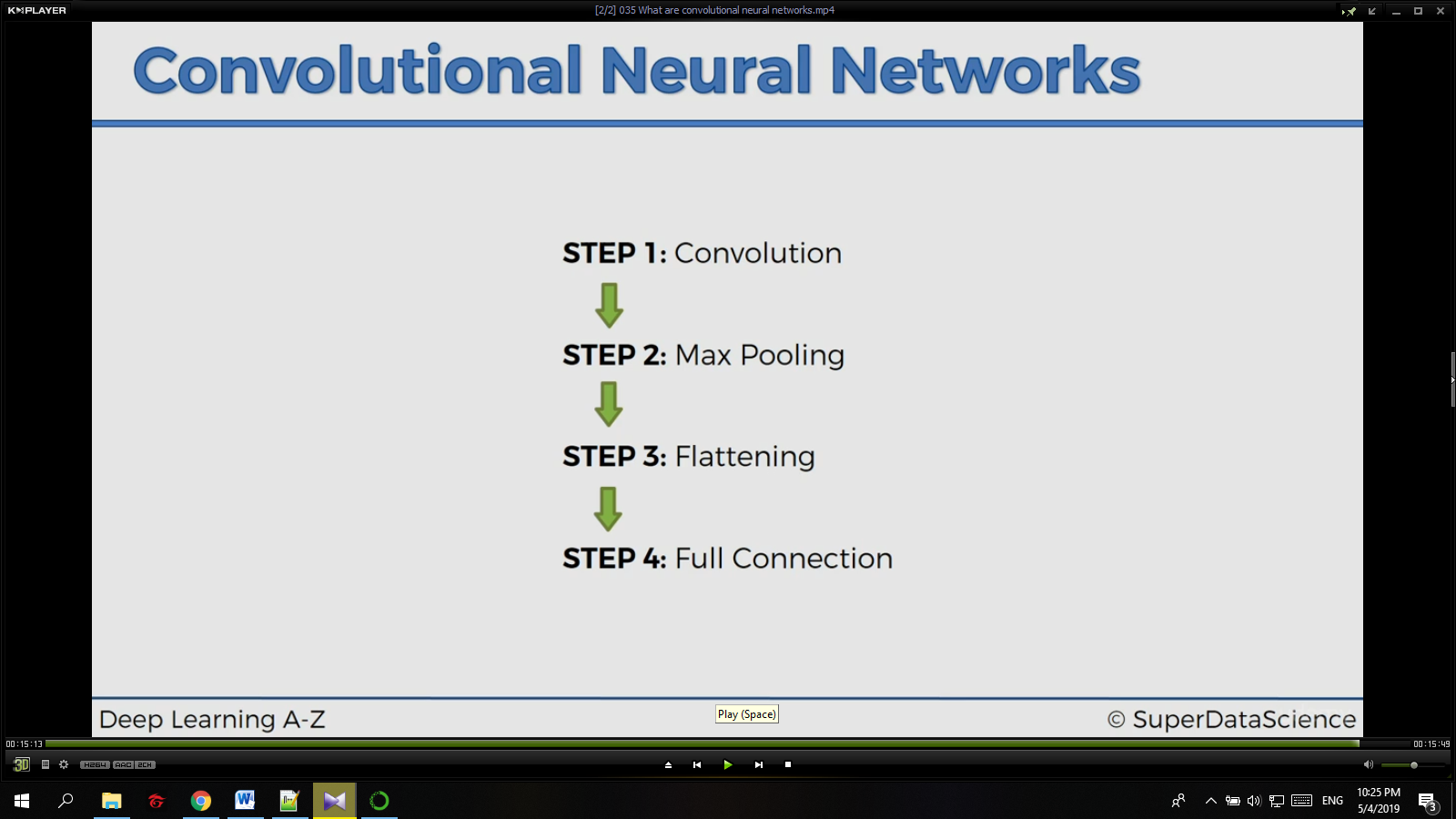
2. So sánh output với giá trị mong muốn (desired value).

3. Nếu chưa đạt được giá trị mong muốn thì hiệu chỉnh trọng số (weights) và tính lại output

**Stochastic Gradient Descent**

Tại một thời điểm ta chỉ tính đạo hàm trên một mẫu và cập nhật ma trận trọng số ν Mỗi lần duyệt qua tất cả các điểm trên toàn bộ dữ liệu ta gọi là epoch υ Với GD: mỗi lần cập nhật ma trận trọng số gọi là một epoch υ Với SGD: Mỗi epoch tương ứng với N lần cập nhật trọng số ν Việc cập nhật từng điểm có thể giảm tốc độ thực hiện 1 epoch. Tuy nhiên SGD chỉ yêu cầu một lượng epoch nhỏ và thường phù hợp với bài toán có dữ liệu lớn (deep learning). Thứ tự chọn điểm dữ liệu: sau mỗi epoch ta cần trộn thứ tự của các điểm dữ liệu để đảm bảo tính ngẫu nhiên ν Giải thuật SGD hội tụ nhanh hơn GD

CNN



## 

## 

## 

## CNNs so sánh hình ảnh theo từng mảnh. Các mảnh mà nó tìm được gọi là các feature. Bằng cách tìm ở mức thô các feature khớp nhau ở cùng vị trí trong hai hình ảnh, CNNsnhìn ra sự tương đồng tốt hơn nhiều so với việc khớp toàn bộ bức ảnh.

## Đóng vai trò nhỏ nhưng quan trọng trong quá trình này là Rectified Linear Unit hoặc ReLU. Toán của cái này cũng rất đơn giản -- bất cứ nơi nào có số âm, hoán đổi nó với 0. Điều này giúp CNN giữ vững sự tin cậy toán học bằng cách giữ các giá trị đã được học khỏi bị mắc kẹt gần 0 hoặc bị thổi bay về vô tận. Đó là thứ dầu mỡ bôi trơn CNNs- không hấp dẫn mấy, nhưng nếu không có nó, chúng sẽ không đi xa hơn được. Đầu ra của một layer ReLU có kích thước giống với đầu vào, chỉ là tất cả các giá trị âm được loại bỏ.

## Convolutional là gì?

Là một cửa sổ trượt (Sliding Windows) trên một ma trận

Các convolutional layer có các parameter(kernel) đã được học để tự điều chỉnh lấy ra những thông tin chính xác nhất mà không cần chọn các feature.

Sliding Window hay còn gọi là kernel, filter hoặc feature detect là một ma trận có kích thước nhỏ như trong ví dụ trên là 3x3. Convolution hay tích chập là nhân từng phần tử bên trong ma trận 3x3 với ma trận bên trái. Kết quả được một ma trận gọi là Convoled feature được sinh ra từ việc nhận ma trận Filter với ma trận ảnh 5x5 bên trái.

## Cấu trúc của mạng CNN

Mạng CNN là một tập hợp các lớp Convolution chồng lên nhau và sử dụng các hàm nonlinear activation như ReLU và tanh để kích hoạt các trọng số trong các node. Mỗi một lớp sau khi thông qua các hàm kích hoạt sẽ tạo ra các thông tin trừu tượng hơn cho các lớp tiếp theo. Mỗi một lớp sau khi thông qua các hàm kích hoạt sẽ tạo ra các thông tin trừu tượng hơn cho các lớp tiếp theo.

Các layer liên kết được với nhau thông qua cơ chế convolution. Layer tiếp theo là kết quả convolution từ layer trước đó, nhờ vậy mà ta có được các kết nối cục bộ. Như vậy mỗi neuron ở lớp kế tiếp sinh ra từ kết quả của filter áp đặt lên một vùng ảnh cục bộ của neuron trước đó.

Mỗi một lớp được sử dụng các filter khác nhau thông thường có hàng trăm hàng nghìn filter như vậy và kết hợp kết quả của chúng lại. Ngoài ra có một số layer khác như pooling/subsampling layer dùng để chắt lọc lại các thông tin hữu ích hơn (loại bỏ các thông tin nhiễu). Trong quá trình huấn luyện mạng (traning) CNN tự động học các giá trị qua các lớp filter dựa vào cách thức mà bạn thực hiện.

Chúng ta gọi việc map từ input layer sang hidden layer là một feature map. Một convolutional layer bao gồm các feature map khác nhau. Mỗi một feature map giúp detect một vài feature trong bức ảnh. Lợi ích lớn nhất của trọng số chia sẻ là giảm tối đa số lượng tham số trong mạng CNN.

Trong mô hình CNN có 2 khía cạnh cần quan tâm là **tính bất biến** (Location Invariance) và **tính kết hợp** (Compositionality).

Pooling layer sẽ cho bạn tính bất biến đối với phép dịch chuyển (translation), phép quay (rotation) và phép co giãn (scaling). Tính kết hợp cục bộ cho ta các cấp độ biểu diễn thông tin từ mức độ thấp đến mức độ cao và trừu tượng hơn thông qua convolution từ các filter. Đó là lý do tại sao CNNs cho ra mô hình với độ chính xác rất cao. Cũng giống như cách con người nhận biết các vật thể trong tự nhiên.

### Lớp tổng hợp (pooling layer)

Lớp pooling thường được sử dụng ngay sau lớp convulational để đơn giản hóa thông tin đầu ra để giảm bớt số lượng neuron.

Thủ tục pooling phổ biến là max-pooling, thủ tục này chọn giá trị lớn nhất trong vùng đầu vào 2x2.

Như vậy qua lớp Max Pooling thì số lượng neuron giảm đi phân nửa. Trong một mạng CNN có nhiều Feature Map nên mỗi Feature Map chúng ta sẽ cho mỗi Max Pooling khác nhau. Chúng ta có thể thấy rằng Max Pooling là cách hỏi xem trong các đặc trưng này thì đặc trưng nào là đặc trưng nhất. Ngoài Max Pooling còn có L2 Pooling.

2 lớp cuối cùng của các kết nối trong mạng là một lớp đầy đủ kết nối (fully connected layer) . Lớp này nối mọi nơron từ lớp max pooled tới mọi nơron của tầng ra.

Input: Lấy cảm hứng từ xử lý ảnh nên input của CNN có dạng như một bức ảnh chứ không có dạng vector như ANN, cụ thể một bức ảnh sau khi số hoá có dạng **width** x **height** x **depth**(**width**: số lượng điểm ảnh trên chiều rộng, **height**: số lượng điểm ảnh trên chiều cao, **depth**: số lượng kênh chẳng hạn như RGB có 3 kênh đại diện cho mức độ của 3 màu Đỏ, Lục, Lam) nên input của CNN là 1 tensor 3 chiều (tensor là gì ta sẽ cùng tìm hiểu trong phần tiếp theo).

Layer: ngoài sử dụng layer fully-connected CNN còn sử dụng một số layer đặc biệt như “Convolution Layer” (đây chính là nơi mà các trọng số được share), “ReLU” , “Pool” (không chứa trọng số hay tham số nhưng lại giúp giảm kích thước của ảnh). 

### Pooling Layer:

Pooling layer trong mạng CNN thực hiện công việc loại bỏ bớt những thông tin không cần thiết sau khi thực hiện tích chập. Điều này rất hữu ích khi ta sử dụng mạng cho ảnh có kích cỡ lớn (dimentional reduction). Tuy nhiên nếu lạm dụng loại layer này cũng có thể khiến data đi qua bị mất dữ liệu.

### 3. Fully-Connected Layer:

Cũng như khi sử dụng mạng ANN truyền thống để xử lý những data có dạng matrix như image. Ta cần flatten data về dạng vector, sau đó đưa vào ANN như bình thường. Hay nói cách khác phần Fully-Connected Layer (FC Layer) chính là một mạng NN được gắn vào phần cuối của CNNs. Phần FC-Layer này chính là nơi từ các feature được extract bởi phần convolution và pooling tạo ra kết quả cuối cùng (Classification hoặc Regression).

**ReLU Layer:**

ReLU layer áp dụng các kích hoạt (activation function) **max(0,x)**lên đầu ra của Conv Layer, có tác dụng đưa các giá trị âm về thành 0. Layer này không thay đổi kích thước của ảnh và không có thêm bất kì tham số nào.

Mục đích của lớp ReLu là đưa ảnh một mức ngưỡng, ở đây là 0. Để loại bỏ các giá trị âm không cần thiết mà có thể sẽ ảnh hưởng cho việc tính toán ở các layer sau đó.

**Pool Layer:**

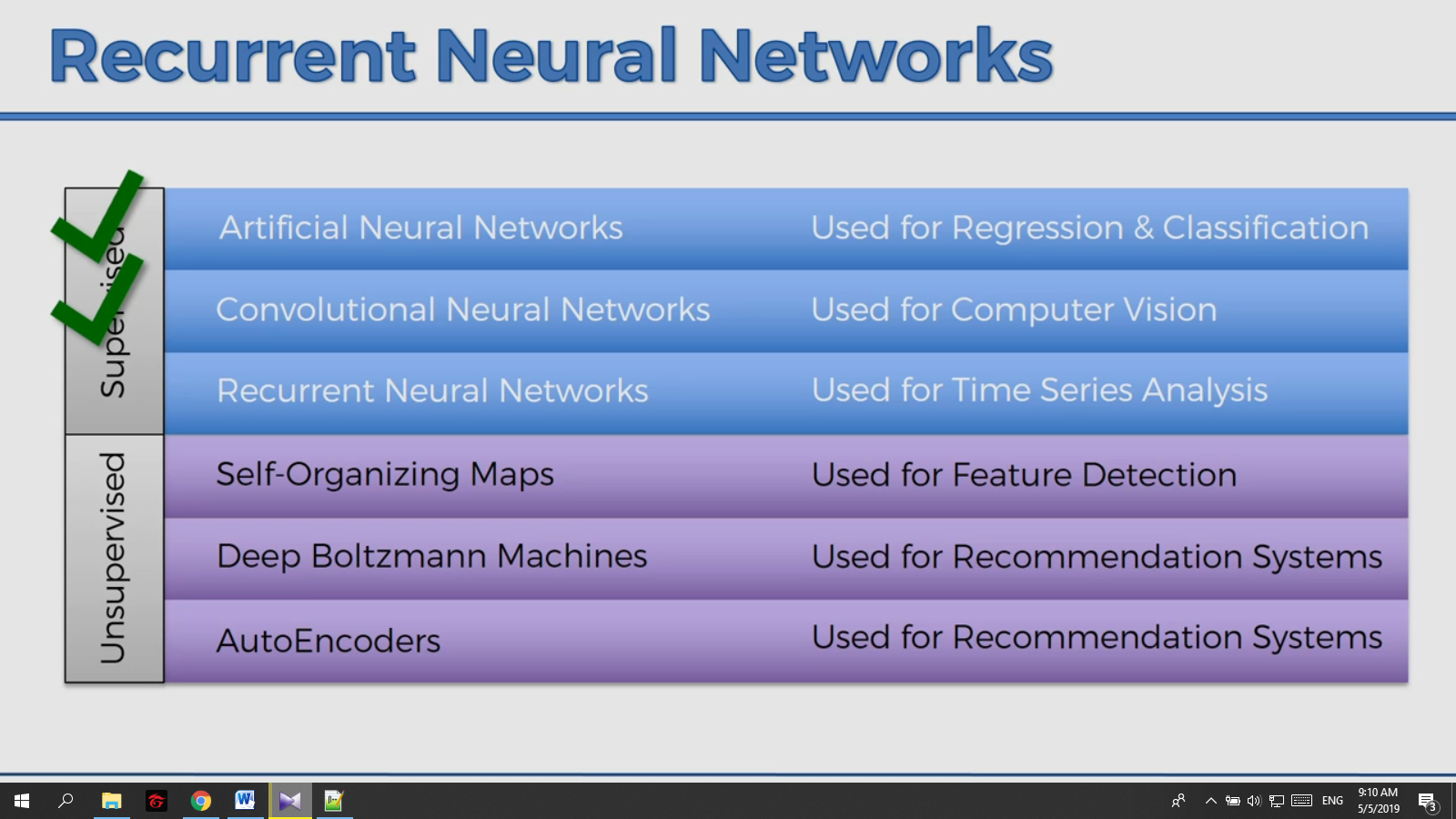
Pool Layer thực hiện chức năng làm giảm chiều không gian của đầu và giảm độ phức tạp tính toán của model ngoài ra Pool Layer còn giúp kiểm soát hiện tượng overffiting. Thông thường, Pool layer có nhiều hình thức khác nhau phù hợp cho nhiều bài toán, tuy nhiên Max Pooling là được sử dụng nhiều vào phổ biến hơn cả với ý tưởng cũng rất sát với thực tế con người đó là: **Giữ lại chi tiết quan trọng** hay hiểu ở trong bài toán này chính giữ lại **pixel có giá trị lớn nhất**.

**Fully\_Connected Layer (FC):**

Tên tiếng viết là Mạng liên kết đầy đủ. Tại lớp mạng này, mỗi một nơ-ron của layer này sẽ liên kết tới mọi nơ-ron của lớp khác. Để đưa ảnh từ các layer trước vào mạng này, buộc phải dàn phẳng bức ảnh ra thành 1 vector thay vì là mảng nhiều chiều như trước. Tại layer cuối cùng sẽ sử dụng 1 hàm kinh điển trong học máy mà bất kì ai cũng từng sử dụng đó là softmax để phân loại đối tượng dựa vào vector đặc trưng đã được tính toán của các lớp trước đó

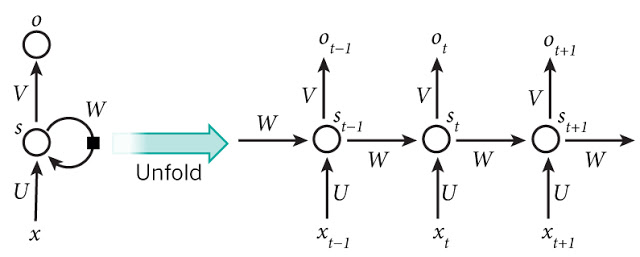
## Các parameter trong CNN

* Convolutional layer: số lượng layers được sử dụng trong mạng CNN
* Pooling: Lớp pooling thường được sử dụng ngay sau lớp convolational để đơn giản hóa thông tin đầu ra để giảm bớt số lượng neuron. Pooling phổ biến là max-pooling, thủ tục này chọn giá trị lớn nhất trong vùng đầu vào 2x2. Như vậy qua lớp Max Pooling thì số lượng neuron giảm đi phân nửa. Trong một mạng CNN có nhiều Feature Map nên mỗi Feature Map chúng ta sẽ cho mỗi Max Pooling khác nhau. Chúng ta có thể thấy rằng Max Pooling là cách hỏi xem trong các đặc trưng này thì đặc trưng nào là đặc trưng nhất. Ngoài Max Pooling còn có L2 Pooling. Cuối cùng ta đặt tất cả các lớp lại với nhau thành một CNN với đầu ra gồm các neuron với số lượng tùy bài toán. 2 lớp cuối cùng của các kết nối trong mạng là một lớp đầy đủ kết nối (fully connected layer). Lớp này nối mọi nơron từ lớp max pooled tới mọi nơron của tầng ra. Pooling parameter được chọn thường là 2x2
* Filter size: thường là ma trận 3x3 hoặc 5x5
* Epochs: số lượng chu kì lặp đi lặp lại của CNN
* Batch: để cải thiện việc tính toán trong CNN, ta sẽ dùng phương pháp chia nhỏ các hình ảnh ban đầu thành nhóm nhỏ hơn



RNN ra đời với ý tưởng chính là sử dụng một bộ nhớ để lưu lại thông tin từ từ những bước tính toán xử lý trước để dựa vào nó có thể đưa ra dự đoán chính xác nhất cho bước dự đoán hiện tại.

RNN được gọi là hồi quy (Recurrent) bởi lẽ chúng thực hiện cùng một tác vụ cho tất cả các phần tử của một chuỗi với đầu ra phụ thuộc vào cả các phép tính trước đó. Nói cách khác, RNN có khả năng nhớ các thông tin được tính toán trước đó. Trên lý thuyết, RNN có thể sử dụng được thông tin của một văn bản rất dài, tuy nhiên thực tế thì nó chỉ có thể nhớ được một vài bước trước đó (ta cùng bàn cụ thể vấn đề này sau) mà thôi .Về cơ bản một mạng RNN có dạng như sau:



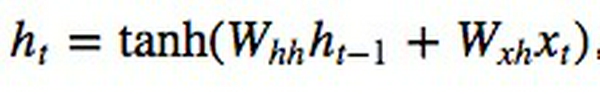
RNN được ứng dụng và thành công ở rất nhiều bài toán, đặc biệt là ở lĩnh vực NLP(xử lý ngôn ngữ tự nhiên).

### Ta sẽ cùng tìm hiểu một số lĩnh vực chính mà RNN cũng như LSTM(LSTM là Long short-term memory network. Mạng nhớ ngắn hạn dài !!!????? =)) Nó là 1 chuỗi các bộ nhớ ngắn hạn) được ứng dụng: Mô hình ngôn ngữ và tự động sinh văn bản, Dịch máy, Nhận dạng giọng nói, Mô tả hình ảnh

### Phương thức hoạt động

Mạng neural hồi quy nhận một vector đầu vào **x**và đưa ra vector đầu ra **y**. Để có thể lưu trữ được thông của các sự kiện trong quá khứ, mạng neural hồi quy lưu trữ trong chính nó một vector trạng thái ẩn **h**. Vector trạng thái này sẽ lưu giữ những thông tin của những sự kiện đã được xử lý bằng cách cập nhật lại giá trị mỗi khi một sự kiện mới được xử lý.

Vector trạng thái ẩn self.h được khởi tạo là một vector không. Hàm **np.tanh**là một hàm phi tuyến hyperbolic đưa giá trị của từng phần tử về khoảng [-1, 1]. Công thức toán học dùng để tính vector trạng thái **h**:



Các ma trận weights này sẽ được cập nhật trong quá trình huấn luyện để điều chỉnh hành vi của mạng neural hồi quy

**1.      Sơ lược về Self Organizing Map (SOM)**

Self Organizing Map (SOM) hay Self-Organizing Feature Map (SOFM) là một mạng Neuron nhân tạo (Artificial Neural Networks – ANN), được huấn luyện (trained) sử dụng kỹ thuật [*Unsupervised learning*](http://en.wikipedia.org/wiki/Unsupervised_learning) để biểu diễn dữ liệu với số chiều (dimension) thấp hơn nhiều (thường là 2 chiều) so với dữ liệu đầu vào nhiều chiều (thường số chiều lớn). Kết quả của SOM gọi là bản đồ (Map). SOM là một ANN, tuy nhiên SOM khác với các ANN là không sử dụng các lớp ẩn (hidden layers) chỉ sử dụng input và output layer. SOM sử dụng khái niệm láng giềng (neighborhood) để giữ lại đặc trưng của các dữ liệu đầu vào trên bản đồ (có nghĩa là các training sample tương tự nhau thì được đặt gần nhau trên bản đồ). Ưu điểm chính của SOM là biểu diễn trực quan dữ liệu nhiều chiều vào không gian ít chiều hơn (thường là 2 chiều) và đặc trưng của dữ liệu đầu vào được giữ lại trên bản đồ.

*SOM gồm các thành phần sau:*

**Output Layer:** Gồm các node (neurons) được bố trí trên một lưới (bản đồ) kích thước X xY. Mỗi neuron có vị trí xác định trên lưới, tại mỗi neuron lưu giữ một vector trọng số (weight vector) có số chiều bằng với số chiều của input vector.

**Input vector:**  Là các training sample có kích thước n

**Ma trận trọng số (weight matrix)** wijkết nối giữa input vector và các neurons

**ANN**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

# Importing the dataset

dataset=pd.read\_csv(r'C:\\Users\ADMIN\Desktop\Churn\_Modelling.csv')

X = dataset.iloc[:, 3:13].values

y = dataset.iloc[:, 13].values

# Encoding categorical data

from sklearn.preprocessing import LabelEncoder, OneHotEncoder

labelencoder\_X\_1 = LabelEncoder()

X[:, 1] = labelencoder\_X\_1.fit\_transform(X[:, 1])

labelencoder\_X\_2 = LabelEncoder()

X[:, 2] = labelencoder\_X\_2.fit\_transform(X[:, 2])

onehotencoder = OneHotEncoder(categorical\_features = [1])

X = onehotencoder.fit\_transform(X).toarray()

X=X[:,1:]

# Splitting the dataset into the Training set and Test set

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.2, random\_state = 0)

# Feature Scaling

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

#Part 2: Now let's make ANN

#importing keras libary and pakages

import keras

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Dropout

#initialising the ANN

classifier=Sequential()

#adding the input layer and the first hidden layer with dropout

classifier.add(Dense(output\_dim = 6, init = "uniform", activation = "relu", input\_dim = 11))

classifier.add(Dropout(p = 0.1))

#Add the second hidden layer

classifier.add(Dense(output\_dim = 6, init = "uniform", activation = "relu"))

classifier.add(Dropout(p = 0.1))

#Adding the output layer

classifier.add(Dense(output\_dim = 1, init = "uniform", activation = "sigmoid"))

#Compiling the ANN

classifier.compile(optimizer="adam", loss="binary\_crossentropy", metrics=['accuracy'])

#Fitting the ANN to the Training set

classifier.fit(X\_train,y\_train, batch\_size=10, nb\_epoch=100)

#Part3: Making the predictions and evaluating the model

# Predicting the Test set results

y\_pred = classifier.predict(X\_test)

y\_pred = (y\_pred > 0.5)

#Predicting a new single observation

"""Predict if the customer with the following infomation will leave the bank:

Geography: France

Credit Score: 600

Gender: Male

Age: 40

Tenure: 3

Balance: 60000

Number of Products: 2

Has credit card: yes

Estimated Salary: 50000

"""

new\_prediction=classifier.predict(sc.transform(np.array([[0.0, 0, 600, 1, 40, 3, 60000, 2, 1, 1, 50000]])))

new\_prediction = (new\_prediction > 0.5)

# Making the Confusion Matrix

from sklearn.metrics import confusion\_matrix

cm = confusion\_matrix(y\_test, y\_pred)

#Part : Evaluating, Improving and Turning the ANN

#Evaluating the ANN

from keras.wrappers.scikit\_learn import KerasClassifier

from sklearn.model\_selection import GridSearchCV

from keras.models import Sequential

from keras.layers import Dense

def build\_classifier():

classifier=Sequential()

classifier.add(Dense(output\_dim = 6, init = "uniform", activation = "relu", input\_dim = 11))

classifier.add(Dense(output\_dim = 6, init = "uniform", activation = "relu"))

classifier.add(Dense(output\_dim = 1, init = "uniform", activation = "sigmoid"))

classifier.compile(optimizer="adam", loss="binary\_crossentropy", metrics=['accuracy'])

return classifier

classifier=KerasClassifier(build\_fn = build\_classifier, batch\_size=10, nb\_epoch=100)

accuracies=cross\_val\_score(estimator = classifier, X=X\_train, y = y\_train, cv = 10, n\_jobs = 1)

mean=accuracies.mean()

variance = accuracies.std()

#Improving the ANN

#Droput Regularization to reduce overfitting if needed

#Turning the ANN

from keras.wrappers.scikit\_learn import KerasClassifier

from sklearn.model\_selection import GridSearchCV

from keras.models import Sequential

from keras.layers import Dense

def build\_classifier(optimizer):

classifier=Sequential()

classifier.add(Dense(output\_dim = 6, init = "uniform", activation = "relu", input\_dim = 11))

classifier.add(Dense(output\_dim = 6, init = "uniform", activation = "relu"))

classifier.add(Dense(output\_dim = 1, init = "uniform", activation = "sigmoid"))

classifier.compile(optimizer=optimizer, loss="binary\_crossentropy", metrics=['accuracy'])

return classifier

classifier=KerasClassifier(build\_fn = build\_classifier)

parameters = {'batch\_size':[25, 32],

'nb\_epoch':[100, 500],

'optimizer': ['adam', 'rmsprop']}

grid\_search = GridSearchCV(estimator = classifier,

param\_grid = parameters,

scoring = 'accuracy',

cv=10 )

grid\_search = grid\_search.fit(X\_train, y\_train)

best\_parameters = grid\_search.best\_params\_

best\_accuracy = grid\_search.best\_score\_

**CNN**

#Part 1: Building the CNN

from keras.models import Sequential

from keras.layers import Convolution2D

from keras.layers import MaxPooling2D

from keras.layers import Flatten

from keras.layers import Dense

#Initialising the CNN

classifier = Sequential()

#Step 1: Convolution

classifier.add(Convolution2D(32, 3, 3, input\_shape = (64, 64, 3), activation = 'relu'))

#Step 2: Pooling

classifier.add(MaxPooling2D(pool\_size = (2, 2)))

#Step 3: Flatten

classifier.add(Flatten())

#Step 4: Full Conection

classifier.add(Dense(output\_dim = 128, activation = 'relu'))

classifier.add(Dense(output\_dim = 1, activation = 'sigmoid'))

#Compiling the CNN

classifier.compile(optimizer = 'adam', loss = 'binary\_crossentropy', metrics = ['accuracy'])

#Part 2: Fitting the CNN to the images

from keras.preprocessing.image import ImageDataGenerator

train\_datagen = ImageDataGenerator(

rescale = 1./255,

shear\_range = 0.2,

zoom\_range = 0.2,

horizontal\_flip = True)

test\_datagen = ImageDataGenerator(rescale=1./255)

training\_set = train\_datagen.flow\_from\_directory(r'C:\\Users\ADMIN\Desktop\dataset\training\_set',

target\_size = (64, 64),

batch\_size = 32,

class\_mode = 'binary'

)

test\_set = test\_datagen.flow\_from\_directory(r'C:\\Users\ADMIN\Desktop\dataset\test\_set',

target\_size = (64, 64),

batch\_size = 32,

class\_mode = 'binary'

)

classifier.fit\_generator(training\_set,

steps\_per\_epoch = 1000,

epochs = 25,

validation\_data = test\_set,

validation\_step = 2000

)

**RNN**

# Recurrent Neural Network

# Part 1 - Data Preprocessing

# Importing the libraries

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

# Importing the training set

dataset\_train = pd.read\_csv('Google\_Stock\_Price\_Train.csv')

training\_set = dataset\_train.iloc[:, 1:2].values

# Feature Scaling

from sklearn.preprocessing import MinMaxScaler

sc = MinMaxScaler(feature\_range = (0, 1))

training\_set\_scaled = sc.fit\_transform(training\_set)

# Creating a data structure with 60 timesteps and 1 output

X\_train = []

y\_train = []

for i in range(60, 1258):

X\_train.append(training\_set\_scaled[i-60:i, 0])

y\_train.append(training\_set\_scaled[i, 0])

X\_train, y\_train = np.array(X\_train), np.array(y\_train)

# Reshaping

X\_train = np.reshape(X\_train, (X\_train.shape[0], X\_train.shape[1], 1))

# Part 2 - Building the RNN

# Importing the Keras libraries and packages

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import LSTM

from keras.layers import Dropout

# Initialising the RNN

regressor = Sequential()

# Adding the first LSTM layer and some Dropout regularisation

regressor.add(LSTM(units = 50, return\_sequences = True, input\_shape = (X\_train.shape[1], 1)))

regressor.add(Dropout(0.2))

# Adding a second LSTM layer and some Dropout regularisation

regressor.add(LSTM(units = 50, return\_sequences = True))

regressor.add(Dropout(0.2))

# Adding a third LSTM layer and some Dropout regularisation

regressor.add(LSTM(units = 50, return\_sequences = True))

regressor.add(Dropout(0.2))

# Adding a fourth LSTM layer and some Dropout regularisation

regressor.add(LSTM(units = 50))

regressor.add(Dropout(0.2))

# Adding the output layer

regressor.add(Dense(units = 1))

# Compiling the RNN

regressor.compile(optimizer = 'adam', loss = 'mean\_squared\_error')

# Fitting the RNN to the Training set

regressor.fit(X\_train, y\_train, epochs = 100, batch\_size = 32)

# Part 3 - Making the predictions and visualising the results

# Getting the real stock price of 2017

dataset\_test = pd.read\_csv('Google\_Stock\_Price\_Test.csv')

real\_stock\_price = dataset\_test.iloc[:, 1:2].values

# Getting the predicted stock price of 2017

dataset\_total = pd.concat((dataset\_train['Open'], dataset\_test['Open']), axis = 0)

inputs = dataset\_total[len(dataset\_total) - len(dataset\_test) - 60:].values

inputs = inputs.reshape(-1,1)

inputs = sc.transform(inputs)

X\_test = []

for i in range(60, 80):

X\_test.append(inputs[i-60:i, 0])

X\_test = np.array(X\_test)

X\_test = np.reshape(X\_test, (X\_test.shape[0], X\_test.shape[1], 1))

predicted\_stock\_price = regressor.predict(X\_test)

predicted\_stock\_price = sc.inverse\_transform(predicted\_stock\_price)

# Visualising the results

plt.plot(real\_stock\_price, color = 'red', label = 'Real Google Stock Price')

plt.plot(predicted\_stock\_price, color = 'blue', label = 'Predicted Google Stock Price')

plt.title('Google Stock Price Prediction')

plt.xlabel('Time')

plt.ylabel('Google Stock Price')

plt.legend()

plt.show()