ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA KHOA HỌC MÁY TÍNH

TRẦN HOÀNG BẢO LY – 21521109

Cấu trúc dữ liệu và giải thuật

IT003.M21.KHTN

Nhận diện chữ số

Mạng Nơ ron 3 lớp

(Neural network L3)

TP. HỒ CHÍ MINH, 2022

Mục Lục

Chương 1. **Lý do chọn đề tài**

Bắt đầu từ những năm 50 của thế kỷ trước, những ý tưởng về một mạng học sâu đã được bắt đầu, thế nhưng đến tận 5 năm gần đây, deep learning nổi lên mạnh mẽ, như là một phong trào của cộng đồng khoa học máy tính. Câu hỏi đặt ra là, tại sao lại có sự trỗi dậy mạnh mẽ đó?

Câu trả lời không phải nằm ở hạn chế về mặt kỹ thuật, về mặt thiết bị, mà nằm ở Dữ liệu, Thời đại công nghệ thông tin đã bắt đầu , số người sử dụng internet ngày càng tăng, thời gian sử dụng internet cũng tăng, dẫn đến lượng dữ liệu tăng một cách khủng khiếp. Lấy ví dụ, năm 2016 cho thấy hơn 3,5 triệu tin nhắn văn bản được gửi đi mỗi phút. Năm 2017, con số này là 15.200.000 tin nhắn, tăng 334% (nguồn: VnExpress). Chính sự bùng nổ về mặt dữ liệu là sự thúc đẩy cho Deep Learning phát triển nhanh chóng đến như vậy.

Hiểu được điều đó, em muốn nhắm vào lĩnh vực này để phát triển trong hiện tại, cũng như tương lai gần, đó chính là lý do em chọn đề tài "Mạng nơ ron 3 lợp".

Với 10 nơ ron ở lớp đầu ra, Em quyết định xây dựng một mạng nơ ron 3 lớp đơn giản để có thể nhận diện được được 10 chữ số đầu trong hệ thập phân (0-9). Rất may là, Kaggle có luôn vấn đề này, và cung cấp sẵn một nguồn dữ liệu để train và test. Ý tưởng đã có, dữ liệu cũng đã có, vậy nên dự án được bắt đầu.

Chương 2. **Mô tả đề tài**

Ở đề tài này chúng ta sẽ phải nhận diện được các chữ số từ 0 đến 9, là các chữ số viết tay từ nhiều người khác nhau. Sau đó sắp xếp chúng vào các thư mục được đánh số đúng với số mà ảnh biểu diễn.

Để giải quyết đề bài trên, chúng ta sẽ xây dựng một mạng nơ ron 3 lớp (XX-XX-10)

Vì sao ở lớp đầu vào là XX? Vì bài toán ở đây là một ảnh nên chúng ta chưa biết được lớp đầu vào chính xác của mình là bao nhiêu. (Mở rộng hơn so với yêu cầu đặt ra là 32).

Mô tả dữ liệu từ Kaggle: Kaggle cung cấp cho chúng ta 2 bộ ảnh, một bộ để train và một bộ để test:

- Bộ train gồm 42.000 ảnh gray 28x28px và mỗi pixel sẽ có giá trị từ 0 đến 255 với không là màu đen hoàn toàn và 255 là màu trắng hoàn toàn, và một giá trị đại diện cho con số mà ảnh đó biểu diễn. Bộ ảnh này được lưu trữ chung trên một file .csv (Comma Separated Values).
- Bộ test gồm 28.000 ảnh gray 28x28px và mỗi pixel sẽ có giá trị từ 0 đến 255 với không là màu đen hoàn toàn và 255 là màu trắng hoàn toàn, và không có giá trị mà chúng ta chỉ sử dụng nó để test (đây là bộ mà chúng ta sẽ thực hiện phân chia). Bộ ảnh này được lưu trữ chung trên một file .csv (Comma Separated Values).

Chương 3. **Cơ sở toán học**

Trước hết ta xác định rằng đây là một bài toán nhị phân, kết quả đầu ra của bài toán chính là, có hay không bức ảnh đầu vào của chúng ta biểu thị cho một con số cụ thể nào đó, Vì vậy ở lớp đầu ra của chúng ta, node thứ i chính là khả năng ảnh đó có biểu thị cho số i hay không? Con số nầy nằm trong khoảng (0,1) với không là chắc chắn không thể, và 1 là hoàn toàn chắc chắn.

Với bài toán như phân loại nhị phân chúng ta áp dụng Hồi quy logistic (Logistic regression), để ước tính các tham số cho mô hình chúng ta, sao cho kết quả đạt được là tốt nhất.

3.1. Bài toán đặt ra

Cho x ϵR^{n_x}, đặt $\hat{y} = P(y = 1 \mid x)$

Chúng ta đặt các tham số sau: $w \in R^{nx}$ và $b \in R$

Đầu ra lúc này sẽ là: $\hat{y} = w^T x + b$. Tuy nhiên $\hat{y}$ là một hàm xác suất nên kết quả của nó phải nằm trong khoảng [0,1]

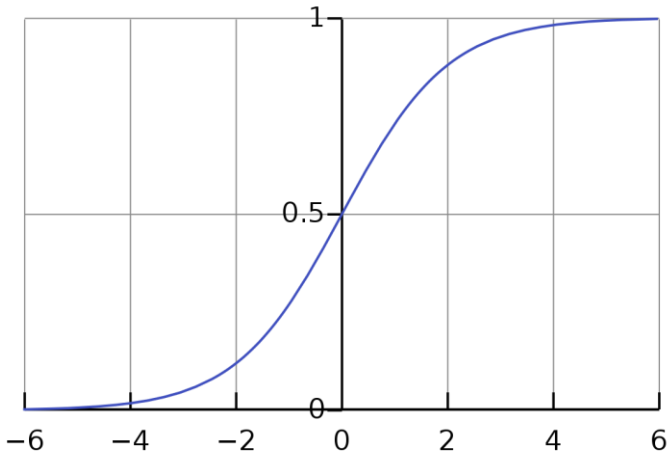
Nên đầu ra mong muốn của chúng ta sẽ sử dụng một hàm đặc biệt gọi là sigmoid,

Sigmoid

$$f(x) = \frac{1}{1+e^{-x}}$$

Ký hiệu là (σ) nên đầu ra lúc này của chúng ta sẽ là

$\hat{y} = \sigma(w^T x + b)$



Hình 3-1 Đồ thị hàm sigmoid

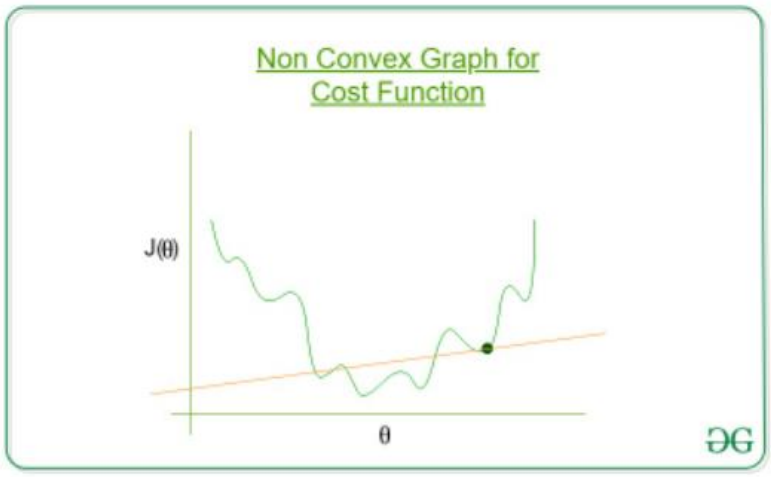
Hàm sigmoid sẽ ép các giá trị nằm trong khoảng (0,1)

Nhưng trong bài toán này chúng ta sẽ không sử dụng hàm sigmoid, thay vào đó chúng ta sẽ sử dụng 2 hàm khác, vấn đề này sẽ được để cập đến ở phần sau.

3.2. Hàm chi phí (cost function)

Cho một tập {(x1,y1), (x2,y2) , … , (xm,ym)}. Chúng ta muốn rằng $\hat{y} \approx y$

Vậy hàm log error function: L(y, $\hat{y}$) = $\frac{1}{2}(\hat{y} - y)^2$ hàm này có dạng:



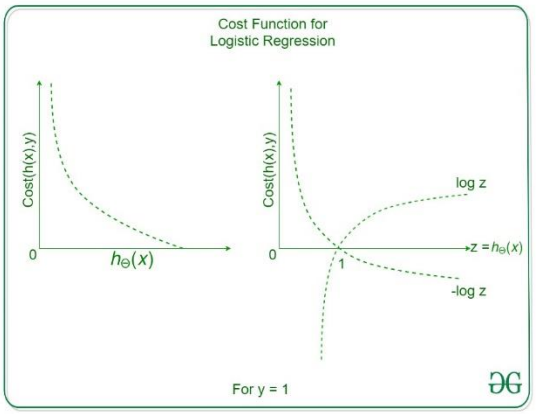
Tuy nhiên trong hồi quy logistic hàm không lồi này, sẽ ảnh hưởng đến thuật toán Gradinet Descent (là thuật toán mà chúng ta dùng để tối ưu tham số, sẽ được đề cập đến sau). Vậy nên chúng ta xây dựng một hàm log errors mới có dạng như sau

L(y, $\hat{y}$) = $-(y \log \hat{y} + (1 - y) \log(1 - \hat{y}))$

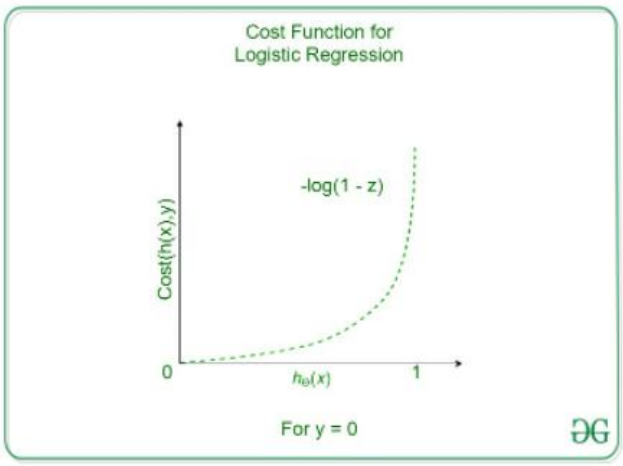
- Nếu y = 1, L(y, $\hat{y}$) = $- y \log \hat{y}$ muốn log$\hat{y}$ lớn => $\hat{y}$ $lớn$
- Nếu y = 0, L(y, $\hat{y}$) = $(1 - y) \log (1 - \hat{y})$ muốn log$\hat{y}$ lớn => $\hat{y}$ $nhỏ$

Hàm chi phí : J = $\frac{1}{m}\sum_{i=1}^{m} L(\hat{y}_i, y_i)$

Đồ thị: nếu y = 1

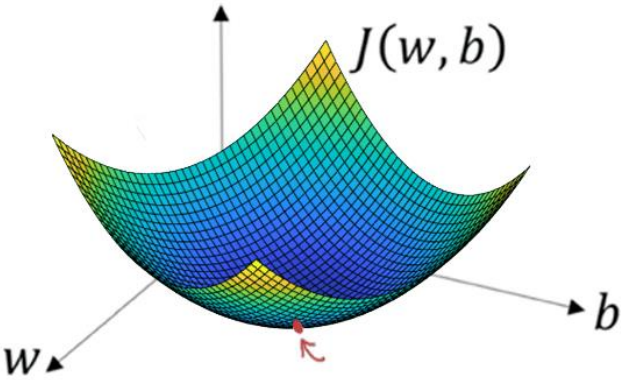


Đồ thị nếu y = 0:

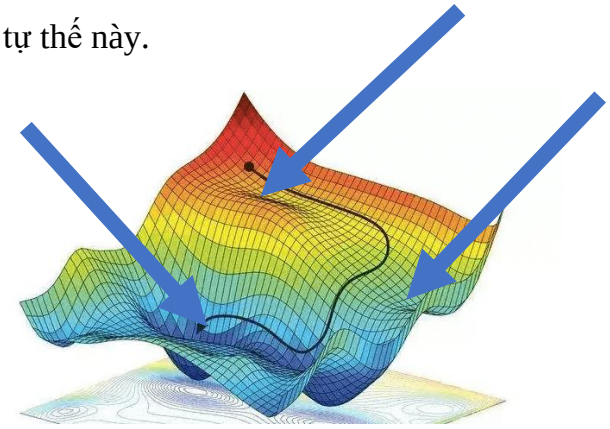


3.3. Gradien Descent

Khi đã xác định được hàm chi phí J, điều chúng ta cần tìm chính là tìm các tham số đầu vào w và b sao cho J(w,b) là bé nhất, đẫn đến một bài toán tìm cực tiểu, và Gradient Descent chính là thuật toán làm điều đó.

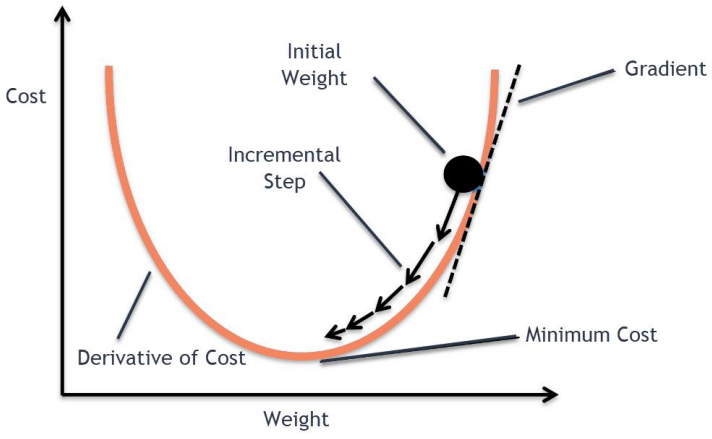


Và chúng ta muốn hàm điểm của chúng ta tìm được chính là hàm lồi có dạng như trên, đó là lý do vì sao hàm log error của chúng ta, không thể làm hàm bình phương vì nó sẽ có dạng tương tự thế này.



Các điểm được chỉ vào là các điểm trũng gọi là các điểm tối ưu cục bộ, cái chúng ta đang tìm là điểm tối ưu toàn cục nên việc có một hàm log errors mới là cần thiết.

Thuật toán Gradient Descent được trình bày như sau:



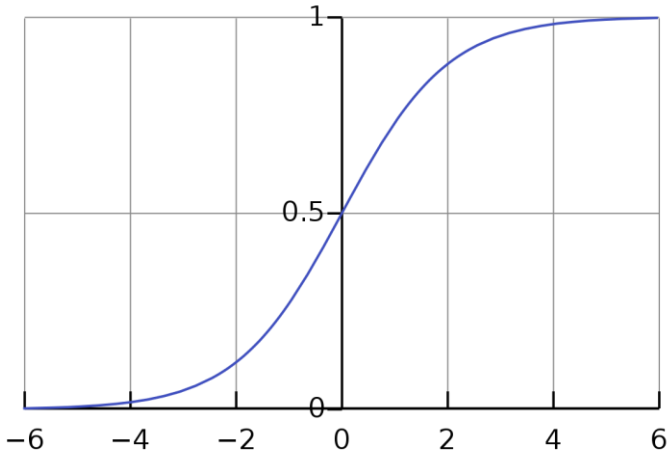
Bước 1: Từ một điểm w khởi tạo ban đầu:

Bước 2: $w = w - \boldsymbol{\alpha}\frac{\mathbf{dJ(w)}}{\mathbf{dw}}$

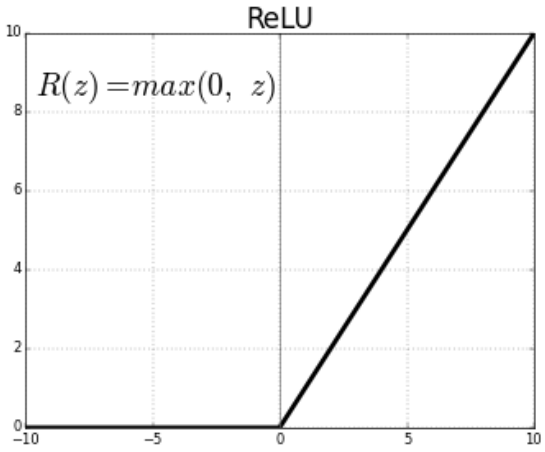
Bước 3: Lặp lại quá trình trên đến khi tìm được w tối ưu

Trong đó $\boldsymbol{\alpha}$ được gọi là learning rate, là tham số mà chúng ta đặt ra sao cho bài toán đạt được là tối ưu nhất.

$\frac{\mathbf{dJ(w)}}{\mathbf{dw}}$ đại diện cho đạo hàm của J(w) tại điểm w biểu thị cho độ dốc tại điểm đó. Độ dốc càng lớn, tốc độ trở về điểm tối ưu càng nhanh. Quay trở lại hàm signoid.



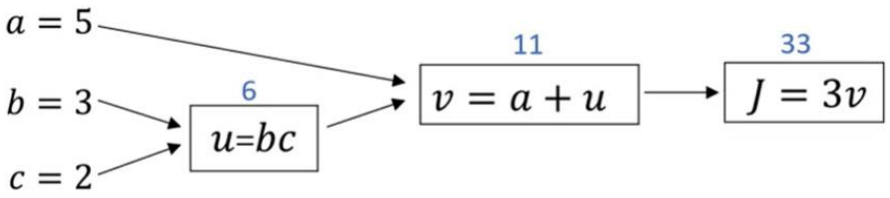
Như đã đề cập ở trên, hàm signoid không tối ưu cho thuật toán của cúng ta. Nhìn vào đồ thị trên ta có thể thấy khi x, càng lớn, f'(x) sẽ càng tiến về không, nghĩa là tốc độ của thuật toán sẽ giảm. Vậy nên, để khắc phục nhược điểm trên một hàm mới được sinh ra đó chính là hàm ReLU



Trong bài này, chúng ta cũng sẽ sử dụng hàm ReLU.

3.4. Đồ thị tính toán (Computation Graph)

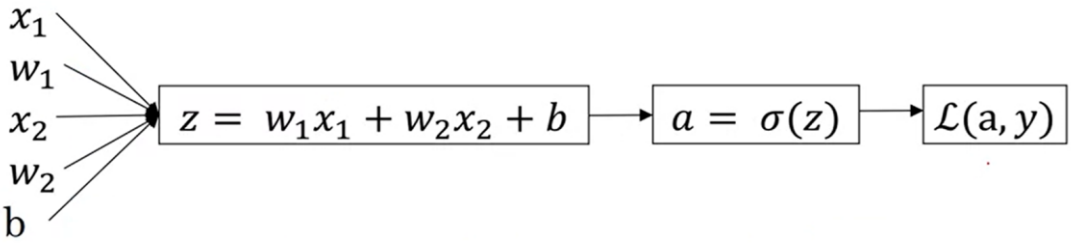
Ví dụ về một đồ thị tính toán:



Với đồ thị tính toán trên ta có thể lấy được các đạo hạm bất kỳ ví dụ

$\frac{dJ}{dv} = 3$, $\frac{dJ}{du} = \frac{dJ}{dv} \cdot \frac{dv}{da}$ Tương tự, ta có thể tính các đạo hàm trước, dựa vào cá đạo hàm sau đã được tính rồi.

Dựa vào mô hình trên ta có sơ đồ sau



Theo cách truyền ngược như trên ta có:

$$\partial a = \frac{dL(a,y)}{da} = -\frac{\boldsymbol{y}}{\boldsymbol{a}} + \frac{\boldsymbol{1-y}}{\boldsymbol{1-a}}$$

$$\partial z = \frac{\boldsymbol{d}_L}{\boldsymbol{da}} \cdot \frac{\boldsymbol{da}}{\boldsymbol{dz}} = a - y \left(với \frac{\boldsymbol{da}}{\boldsymbol{dz}} = a(1-a) \right)$$

Việc triển khai Gradient Descent sẽ trên m mẫu thử có thể thực hiện bằng cách khởi tạo các biến J, dw, db, sau đó lần lượt tính toán trên từng giá trị, x ∈ R^n chúng ta buộc phải thực hiện thêm một vòng lặp để có được giá trị đó. Độ phức tạp lúc này đã là O(m*n) , điều này là không tốt chút nào với deep learning nơi mà lượng dữ liệu đầu vào cực lơn.

Câu hỏi đặt ra là liệu có cách nào để chúng ta có thể tính toán các giá trị đó một lượt hay không?

3.5. Vector hóa(Vectorization)

Vector hóa là kỹ thuật quan trọng trong Deep learning, và để thực hiện được điều này, chúng ta sẽ cần đến một môn học khác ngoài giải tích, chính là đại số tuyến tính.

Với x ϵR^n chúng ta sẽ biểu diễn x dưới dạng một vector n chiều, lúc này chúng ta đã loại bỏ được một vòng lặp phía trong, tiếp đến, chúng ta sẽ sếp m vector x vào các cộ, có tổng cộng là m vector x, nên ta sẽ được ma trận là nxm

Ma trận có dạng như sau

$$\begin{Vmatrix} | & | & | & | \\ X1 & X2 & \dots & Xm \\ | & | & | & | \end{Vmatrix}$$

Câu hỏi đặt ra là, liệu có cách nào để tính toán ma trận trên nhanh hơn cách thông thường không?

Rất may là thư viện numpy của python đã có các hàm dựng sẵn (built-in function) dùng để tính toán các ma trận cực kỳ nhanh chóng. Hãy làm một phép đo đơn giản.

```
[8] import time
    import numpy as np

    a = np.random.rand(1000000)
    b = np.random.rand(1000000)
    tic = time.time()
    c = np.dot(a,b) # c = a*b in matrix
    toc = time.time()
    print('Vector: ' + str((toc-tic)*1000) + 'ms')

    Vector: 2.504110336303711ms

    tic = time.time()
    c = 0
    for i in range(1000000):
        c += a[i] * b[i]
    toc = time.time()
    print('Normal way: ' + str((toc-tic)*1000) + 'ms')

    Normal way: 485.72421073913574ms
```

Với thư viện numpy tính toán trên vector, thời gian thực hiện phép nhân là 2.5ms

Với cách tính thông thường, thời gian thực hiện là 485.7ms

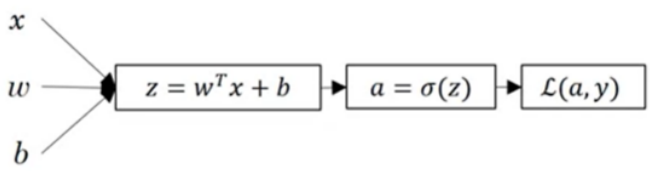
Cách tính theo vector cho hiệu suất tốt hơn đến ≈ 19328%

Như vậy chúng ta đang đi đúng hướng.

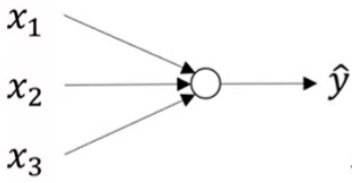
Chương 4. Mạng nơ ron (Neural network)

4.1. Ý tưởng

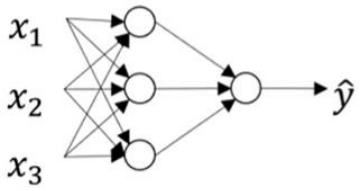
Từ mô hình toán học được đề cập ở trên



Chúng ta mô hình hóa nó như sau:



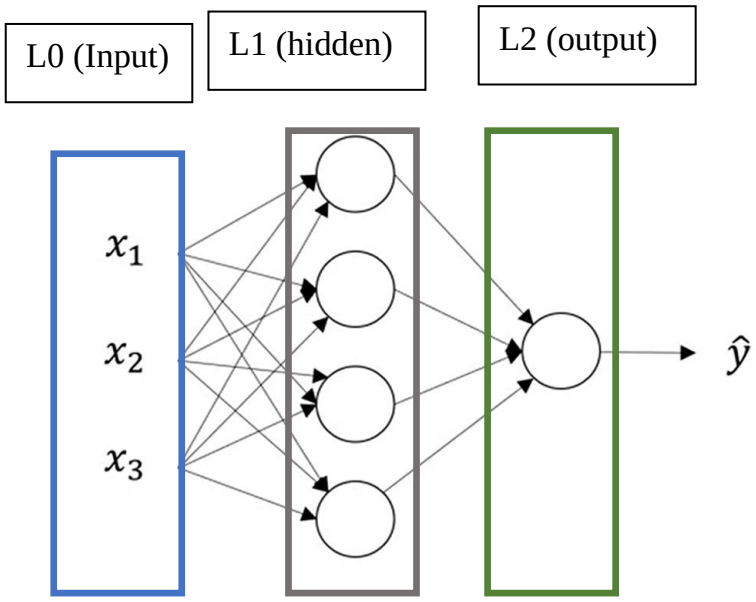
Để tính toán trên mạng này chúng ta cần mở rộng phần lớp thứ 2 gọi là lớp ẩn, Ta có mạng nơ ron đơn giản sau.



Tính toán theo chiều xuôi, chúng ta sẽ được như sau

x	Layer[1]		Layer[2]		Log
W	z[1] =W[1]*x+b	a[1] = g(z[1])	z[2]=W[2]*a[1]+b	a[2] = g(z[2])	L(a[2],y)
b	**Chiều xuôi** ⟶				

4.2. Trình bày vè mạng nơ ron



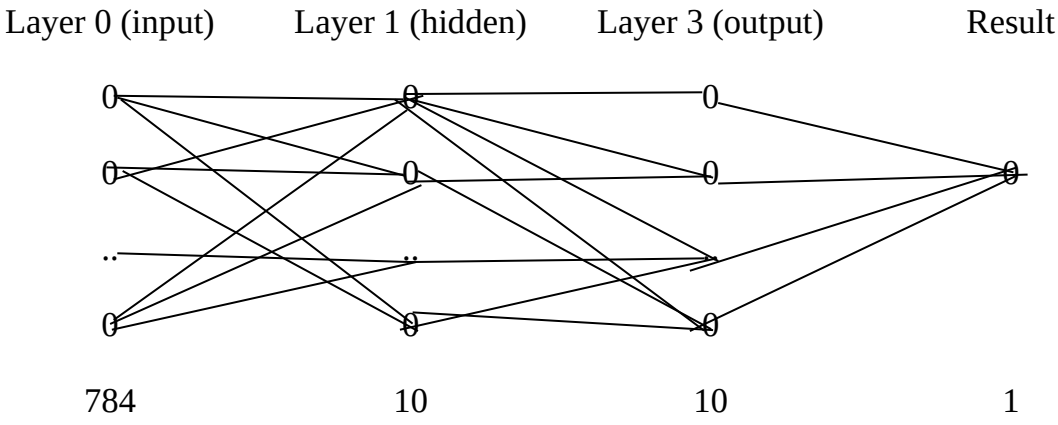
4.3. Mạng nơ ron trong bài toán nhận diện số

Trong bài toán này, chúng ta sẽ xây dựng mạng nơ ron để hiện thực bài toán nhận diện số. Ảnh đầu vào của chúng ta là ảnh 28x28 pixel mỗi pixel có giá trị từ 0-255.

Nên chúng ta sẽ xếp các pixel thành một cột duy nhất có 28x28 = 784

- Vậy Layer 0 của chúng ta sẽ có 784 nơ ron
- Để xử lý bài toán trên ở Layer 1 chúng ta cũng có 10 nơ ron
- Tất nhiên Layer 2 (output) sẽ có 10 nơ ron đại diện cho kết quả ta vừa tính được.

Mạng nơ ron của chúng ta sẽ có dạng như sau:



Xếp các lớp này theo hàng dọc, ta được ma trận đầu vào (mxn) như sau:

$$\begin{Vmatrix} X11 & X21 & \dots & Xm1 \\ | & | & | & | \\ | & | & & | \\ | & | & | & | \\ X1n & X2n & \dots & Xmn \end{Vmatrix}$$

A = X (nxm)

Từ các công thức toán học ở trên ta rút ra được bộ công thức sau:

4.4. Công thức truyền xuôi:

Z1 = W1*X + b1;

A1 = g(Z1) (g là hàm hiệu chỉnh, trong trường hợp này ta sử dụng hàm ReLU)

Z2 = W2*A + b1;

A2 = g(Z2)

hàm hiệu chỉnh lúc này sẽ được thay đổi vì kết quả đầu ra mong muốn của chúng ta nằm trong khoảng 0 dến 1, tuy nhiên hàm signoid không phải là một sự lựa chọn tốt vậy nên chúng ta sẽ sử dụng một hàm mới gọi là hàm softmax có công thức như sau:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

4.5. Công thức truyền ngược:

dZ2 = A2 – Y

dW2 = 1/m * dZ2 * $A1^T$

db2 = 1/m = $\sum dZ2$

dZ1 = W^T dZ2*g’(Z1)

dW2 = 1/m * dZ1 * AX^T

db2 = 1/m = $\sum dZ1$

Chương 5. **Cài đặt với ngôn ngữ Python**

/* Đầu tiên, data được lấy từ Kaggle sang Google Colab, phần cài đặt và thử nghiệm được thực hiện trên Google Colab, sau đó, dữ liệu được tải về máy và cài đặt lại, để thực hiện trainning nhanh hơn và dễ dàng thực hiện phân chia ảnh.

Chi tiết cách lấy dữ liệu từ Kaggle đã có sẵn trong Colab, toàn bộ file liên quan đến dự án được tải lên github, link github ở cuối báo cáo này. */

Cài đặt thư viện:

Thư viện pandas và time dùng để đọc ảnh và đo thời gian training

```
from tkinter import W
import numpy as np
import pandas as pd
import time
```

Load và chuẩn bị dữ liệu

```
loadData = pd.read_csv('train/train.csv')
data = np.array(loadData)
m,n = data.shape
np.random.shuffle(data) # we dont know about data so we want to shuffle it
randomly

data_Train = data[: m].T #data (m,n) so we want it to (n,m)
Y_Train = data_Train[0] #first row is out Y label
X_Train = data_Train[1:n] # 1 to n is data image
X_Train = X_Train/255 # force if to [0,1]
```

Hàm khởi tạo W1,b1,W2,b2 (- 0.5 để hiệu chỉnh đẩy giá trị về khoảng [-0.5,0.5])

```
def Init_Pram():
  W1 = np.random.rand(10, n-1)  - 0.5
  b1 = np.random.rand(10, 1) - 0.5
  #We have 10 neural in L2 is our output layer
  W2 = np.random.rand(10,10) - 0.5
  b2 = np.random.rand(10,1) - 0.5
  return W1, b1, W2, b2
```

Ba hàm g (tuy nhiên chúng ta sẽ không dử dụng hàm signoid ở đây)

```
def ReLU(Z):
  return np.maximum(0,Z)
def Signoid(Z):
  return 1/(1+np.exp(-Z))
def SoftMax(Z):
  exp = np.exp(Z - np.max(Z))
  return exp / exp.sum(axis=0)
```

Z1 = W1*X + b1;

A1 = g(Z1) (g là hàm hiệu chỉnh, trong trường hợp này ta sử dụng hàm ReLU)

Z2 = W2*A + b1;

A2 = g(Z2) (softmax)

Theo công thức truyền xuôi ta có hàm truyền xuôi

```
def  Forward_Propagation(X, W1, b1, W2, b2):
  Z1 = W1.dot(X) + b1
  A1 = ReLU(Z1)
  Z2 = W2.dot(A1) + b2
  A2 = SoftMax(Z2)
  return Z1, A1, Z2, A2
```

dZ2 = A2 – Y

dW2 = 1/m * dZ2 * $A1^T$

db2 = 1/m = $\sum dZ2$

dZ1 = W^T dZ2*g’(Z1)

dW2 = 1/m * dZ1 * AX^T

db2 = 1/m = $\sum dZ1$

Để có thể tính toán tốt các công thức ở trên ta thực hiện khởi tạo lại ma trận pY thay vì m*1 sẽ là m*10, tại vị trí thứ pY[i, Y[i]] = 1

```
def init_Y(Y):
  # Create a Matrix(Y.size(), Y.max() +1) with all values are zero in this
case we have recognize 10 digits(from 0 to 9) so max must be 9 + 1 we have
10 colum
  aY = np.zeros((Y.size, Y.max() + 1))
  aY[np.arange(Y.size), Y] = 1 # Assign right number in each dataset is 1
  aY = aY.T
  return aY
```

Hàm lấy g'(Z1)

```
def sign_ReLU(Z):
  return Z>0
```

Hàm truyền ngược

```
def Back_Propagation(X,Y,Z1,A1,Z2,A2,W1,W2):
  preY = init_Y(Y)
  dZ2 = A2- preY
  dW2 = 1/m * dZ2.dot(A1.T)
  db2 = 1/m * np.sum(dZ2,1)

  dZ1 = W2.T.dot(dZ2) * sign_ReLU(Z1)
  dW1 = 1/m * dZ1.dot(X.T)
  db1 = 1/m * np.sum(dZ1,1)
  return dW1, db1, dW2, db2
```

Hàm cập nhật lại giá trị W1, b1, W2, b2 cho mỗi bước trong thuật toán Gradient Descent.

```
def Update_Parameter(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
  W1 = W1 - alpha*dW1
  b1 = b1 - alpha*db1.reshape(10,1)
  W2 = W2 - alpha*dW2
  b2 = b2 -  alpha*db2.reshape(10,1)
  return W1, b1, W2, b2
#Two function below to calculate Accurancy of your result (A / Y (input data))
def Get_Prediction(A):
  return np.argmax(A,0)
def Get_Accuracy(prediction, Y):
  print(prediction,Y)
  accuracy = np.sum(prediction == Y) / Y.size
  return accuracy
```

Hàm Gradient Descent

```
def Gradient_Descent(X, Y, alpha, number_Step):
  W1,b1,W2,b2 = Init_Pram()
  for i in range(number_Step):
    Z1, A1, Z2, A2 = Forward_Propagation(X, W1, b1, W2, b2)
    dW1, db1, dW2, db2 = Back_Propagation(X,Y,Z1,A1,Z2,A2,W1,W2)
    W1, b1, W2, b2 = Update_Parameter(W1,b1,W2,b2,dW1,db1,dW2,db2,alpha)
    if (i%10 == 0):
      print("step: ", i)
      prediction =  Get_Prediction(A2)
      print(Get_Accuracy(prediction,Y))
  return W1,b1, W2, b2
```

Cứ mỗi mười bước chúng ta sẽ in ra console độ chính xác đã đạt được hiện tại.

Gọi hàm Gradient_Descent với alpha = 0.1 và số bước là 1000 và ghi nhận thời gian train

/*Về alpha, sau khi hiệu chỉnh nhiều lần thì con số 0.1 mang lại kết quả chính xác và ổn định hơn các số khác, tốc độ thực hiện tương đối cao. */

/*Về số bước, khi đến gần 1000 bước, tốc độ tăng độ chính xác dã giảm đi rất nhiều, nên con số 1000 là vừa đủ tốt*/

```
time_Log = 0
tic = time.time()
W1, b1, W2, b2 = Gradient_Descent(X_Train, Y_Train,  0.1, 1000)
toc = time.time()
time_Log = toc-tic
print(time_Log)
```

Lưu các tham số W1,b1,W2, b2 vào file để tái sử dụng

```
np.save('AfterTrainData/W1trained', W1)
np.save('AfterTrainData/b1trained', b1)
np.save('AfterTrainData/W2trained', W2)
np.save('AfterTrainData/b2trained', b2)
_, _, _, A2 = Forward_Propagation(X_Train, W1, b1, W2, b2)
predictions =  np.argmax(A2,0)
accuracy_Log = np.sum(predictions==Y_Train)/Y_Train.size
#Write TrainLog
import os
f = os.__file__
if (os.path.exists('TrainLog.txt')):
  f = open("TrainLog.txt", "w")
else:
  f =  open("TrainLog.txt", "x")
f.write("Time to train is: " + str(time_Log))
f.write("\nAccuracy on training set: "+ str(accuracy_Log))
f.close()
```

Lưu lại độ chính xác cuối cùng và thời gian thực hiện vào file TrainLog.txt

Time to train is: 108.89846014976501

Accuracy on training set: 0.8801190476190476

Kết quả ghi nhận trên file TrainLog

Thời gian thực hiện là 108.898s

Và độ chính xác đạt được là 0.88

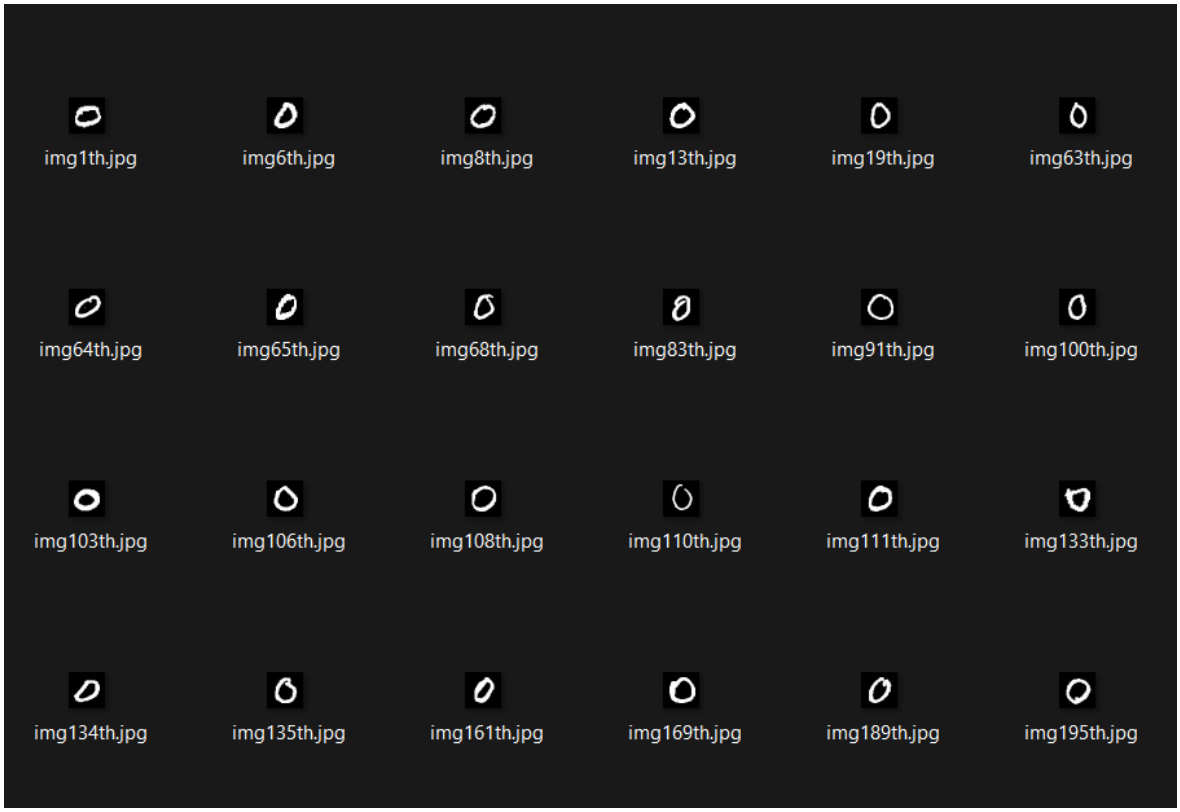
Thực hiện phân chia ảnh sử dụng dữ liệu vừa train:

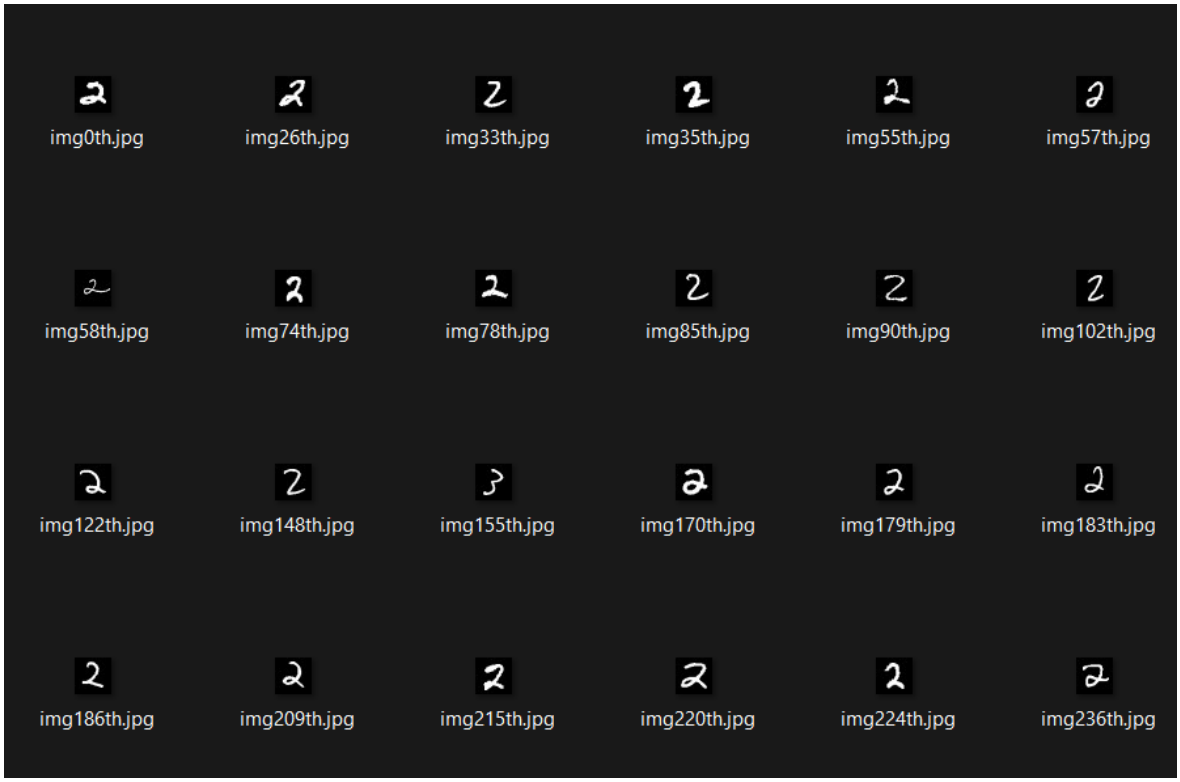
```
from cgi import test
from operator import index
from tkinter import Image, image_names
import numpy as np
import pandas as pd
import cv2 as cv
import os
#Create Folder to classify images
if (os.path.exists('Classify') == 0):
  os.mkdir('Classify')
if (os.path.exists('Classify/WrongPredictions') == 0):
  os.mkdir('Classify/WrongPredictions')
for i in range(10):
   path = 'Classify/Number_' + str(i)
   if (os.path.exists(path) == 0):
```

```
    os.mkdir(path)

test_Data =  pd.read_csv('train/test.csv')
data = np.array(test_Data)

m,n = data.shape
data = data.T
X = data
X = X/255
def ReLU(Z):
  return np.maximum(0,Z)
def SoftMax(Z):
  exp = np.exp(Z - np.max(Z))
  return exp / exp.sum(axis=0)
def  Forward_Propagation(X, W1, b1, W2, b2):
  Z1 = W1.dot(X) + b1
  A1 = ReLU(Z1)
  Z2 = W2.dot(A1) + b2
  A2 = SoftMax(Z2)
  return Z1, A1, Z2, A2
W1 = np.load('AfterTrainData/W1trained.npy')
W2 = np.load('AfterTrainData/W2trained.npy')
b1 = np.load('AfterTrainData/b1trained.npy')
b2 = np.load('AfterTrainData/b2trained.npy')
def Make_Predictions(X,W1,b1,W2,b2):
    _,_,_,A2 = Forward_Propagation(X,W1,b1,W2,b2)
    predictions = np.argmax(A2,0)
    return predictions
def Classify_Image(Index, W1,b1, W2, b2):
  img = X[:,Index,None]
  prediction = Make_Predictions(img,W1,b1,W2,b2)
  img = img.reshape(28,28) * 255
  numberPrediction = int(prediction)
  image_Path = 'Classify/Number_' + str(numberPrediction)
  image_Name = '/img' + str(Index) + 'th.jpg'
  cv.imwrite(image_Path + image_Name, img)

  #predictions = Make_Predictions(img, W1,b1,W2,b2)
  #img = img.reshape(28,28)*255
  #print(predictions)
for i in range(m):
  Classify_Image(i, W1, b1, W2, b2)
  print(i)
```

Đoạn mã trên chỉ đơn giản là load ảnh từ file test của kaggle, đưa ra dự đoán từ các tham số ta nhận được qua thời gian 109s train, và chia chúng vào các folder đúng với số mà chúng ta dự đoán bức ảnh sẽ hiển thị.

Chương 6. **Kết quả**





Trên đây là một vài hình ảnh đã được phân chia vào folder, có một số kết quả sai, vì độ chính xác của chúng ta chỉ là 0.88. Toàn bộ file kết quả cũng đã được cập nhập trên Github:

Chương 7. **Liên kết**

Github: click here

Colab: click here