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

ĐỒ ÁN CUỐI KÌ
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

ĐỀ TÀI:

XÂY DỰNG MẠNG NEURAL 3 LỚP

Giảng viên lý thuyết: Nguyễn Thanh Sơn
Giảng viên hướng dẫn thực hành: Nguyễn Đức Vũ
Người thực hiện: Lê Chí Cường

--TP. Hồ Chí Minh, 2 tháng 6 năm 2022--

Mục lục

I. Tổng quan đề tài

1. Lý do chọn đề tài

Mô hình mạng Neural đã được đề xuất và nghiên cứu từ những thập niên 40, 50 của thế kỉ trước. Tuy nhiên, phải đến những năm gần đây, với sự nâng cấp mạnh mẽ của dung lượng bộ nhớ và tốc độ xử lí máy tính, Neural Network mới được ứng dụng để giải quyết các vấn đề trong thực tế và đem lại kết quả rất tốt.

Nhận thấy sự phổ biến và tính hiệu quả của việc sử dụng mạng Neural để giải quyết các bài toán trong lĩnh vực Trí tuệ nhân tạo, trong đồ án môn học CTDL&GT này, em xin phép tìm hiểu và xây dựng một mô hình mạng Neural đơn giản nhất – Mạng Neural 3 lớp.

2. Mục tiêu của đề tài

Xây dựng một mạng Neural 3 lớp cho bài toán **Multiclass – Classification** và kiểm tra tính hiệu quả của nó trong việc giải quyết bài toán phân loại các hình ảnh chữ số viết tay từ bộ dữ liệu **MNIST**

II. Nội dung thực hiện đề tài

1. Cơ sở lý thuyết

Trong phần này em sẽ đề cập đến các khái niệm cơ bản và cơ sở toán học cần thiết của mạng Neural 3 lớp.

1.1. Các khái niệm cơ bản cần thiết:

a. Tập dữ liệu (dataset), điểm dữ liệu (data point) và vector đặc trưng:

- **Dataset**: là tập dữ liệu chưa qua xử lí mà ta thu thập được. Một **dataset** có thể chứa nhiều **data point**.
- **Data point**: là một đơn vị thông tin độc lập trong **dataset** (ví dụ **dataset** là một tập hợp nhiều ảnh chân dung thì một **data point** là một bức ảnh trong số đó). Mỗi điểm dữ liệu gồm nhiều đặc trưng (*feature*) khác nhau, mỗi *feature* thường được biểu diễn dưới dạng một con số.
- Trong Machine Learning, ta thường biểu diễn một điểm dữ liệu như một vector $\mathrm{x} \in R^d$ trong đó mỗi phần tử x_i là một đặc trưng. Vector này gọi là **vector đặc trưng (*feature vector*)** của điểm dữ liệu.

b. Bài toán Classification (phân loại/phân lớp):

- Trong bài toán Classification, chương trình sẽ được yêu cầu chỉ ra *nhãn (label)* hay *lớp* (*class*) của một điểm dữ liệu (ví dụ nếu ta có bức ảnh một con mèo thì nhãn của bức ảnh này có thể là *mèo*, và chương trình sẽ được yêu cầu chỉ ra nhãn cho bức ảnh này là *mèo* nếu ta đưa cho chương trình bức ảnh này nhưng không dán nhãn).
- Nhãn thường là một phần tử trong tập hợp có *C* phần tử. Mỗi phần tử trong tập hợp gọi là một *class* hay một *label* và thường được đánh số từ 1 đến *C*.
- Để giải bài toán này, ta cần tìm một hàm số $f: R^d \rightarrow \{1, 2, \dots, C\}$ thỏa mãn điều kiện khi ta cho $y = f(\mathrm{x})$ với x là vector đặc trưng mô tả điểm dữ liệu thì *y* sẽ là nhãn của điểm dữ liệu đó.
- Trong nhiều bài toán, *y* có thể không phải là một con số mà là một vector thuộc R^C. Khi đó phần tử y_i của *y* biểu diễn xác suất điểm dữ liệu x có nhãn là i.

c. Supervised learning:

Là thuật toán dự đoán đầu ra của một hoặc nhiều điểm dữ liệu mới dựa trên các cặp [đầu ra, đầu vào] đã biết trước

Cụ thể, ta có một tập hợp biến đầu vào $X = \{x_1, x_2, x_3, \dots, x_N\}$ và tập đầu ra tương ứng $Y = \{y_1, y_2, y_3, \dots, y_N\}$ đã biết trước (trong đó x_i, y_i là các vector; y_i cũng có thể là một số vô hướng). Các cặp dữ liệu đã biết trước $(x_i, y_i) \in X \times Y$ tạo nên tập huấn luyện. Nhiệm vụ của ta là từ tập huấn luyện này tìm được một hàm số ánh xạ mỗi phần tử của tập *X* với một phần tử tương ứng của tập *Y* (có thể xấp xỉ với một độ chính xác nào đó):

$$y_i \approx f(x_i), \forall i = 1,2, \dots, N$$

Mục tiêu của ta là xấp xỉ hàm số f tốt nhất có thể để khi có một dữ liệu x mới, ta có thể tính được nhãn tương ứng y của nó với độ chính xác cao.

Trong phạm vi đề tài này, em sẽ sử dụng thuật toán **Supervised learning** để giải quyết bài toán phân loại chữ số viết tay sau khi đã xây dựng được mạng Neural 3 lớp.

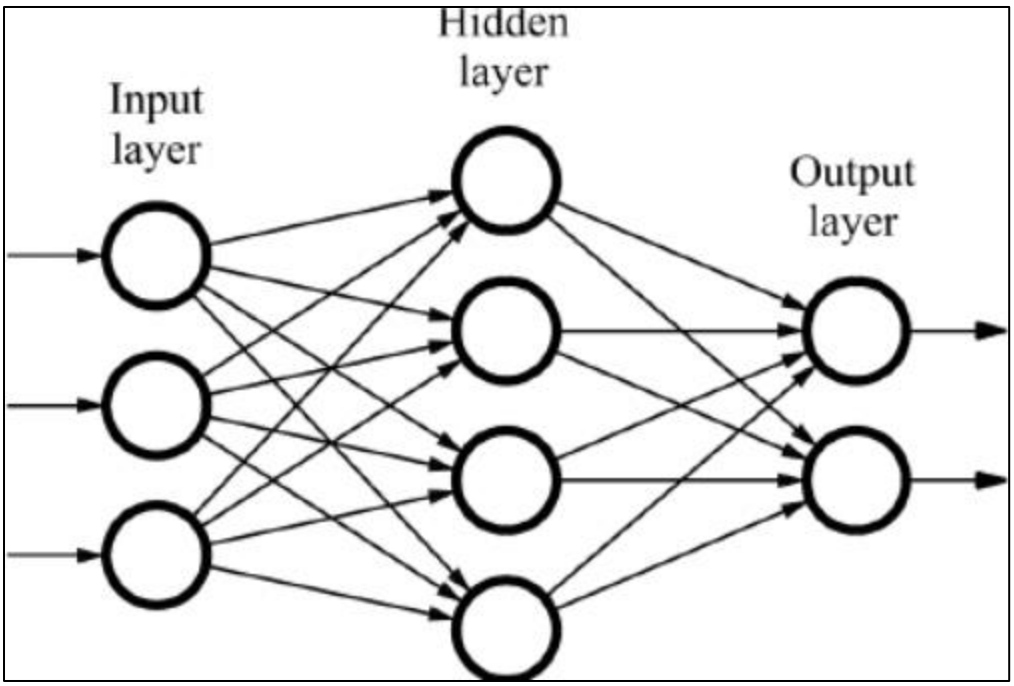
1.2. Mạng Neural 3 lớp, các khái niệm, kí hiệu và cơ sở toán học cần thiết

a. Mạng Neural 3 lớp:

- Mạng Neural 3 lớp là một mô hình tính toán làm nhiệm vụ xấp xỉ mối quan hệ giữa các cặp **điểm dữ liệu đầu vào – nhãn tương ứng** (x,y) trong tập huấn luyện bằng một hàm số có dạng:

$$y \approx g^{(2)}\left(g^{(1)}(x)\right) (1)$$

- Lý do mô hình này có tên **mạng Neural 3 lớp** là do nó thực hiện việc xấp xỉ hàm số ở công thức (1) thông qua 3 lớp là **input layer, hidden layer** và **output layer** như hình bên dưới:



Hình 1: Cấu trúc mạng Neural 3 lớp

b. ***Các khái niệm, kí hiệu:***

b.1. Layer:

- Trong mạng Neural, các Layer đóng vài trò là các hàm:

$$\mathrm{a}^{(l)} = g^{(l)}\left(\mathrm{a}^{(l-1)}\right)$$

Trong đó $\mathrm{a}^{(l)}$ là vector đầu ra của Layer thứ l ($l \in \{1,2,\dots,L\}$) tính từ Hidden Layer đầu tiên. Với quy ước $\mathrm{a}^{(0)} = \mathrm{x}$ là vector đầu ra của Input Layer và cũng là điểm dữ liệu đầu vào.

- Trong phạm vi đề tài này, em sẽ chỉ xét và sử dụng những hàm có dạng như dưới đây để xây dựng mạng Neural 3 lớp:

$$g^{(l)}\left(\mathrm{a}^{(l-1)}\right) = f\left(\mathrm{W}^{(l)T}\mathrm{a}^{(l-1)} + \mathrm{b}^{(l)}\right)$$

Trong đó $\mathrm{W}^{(l)}$ là ma trận trọng số (**Weight)** và $\mathrm{b}^{(l)}$ là vector **bias** của Layer thứ *l.* Hàm số *f* được gọi là hàm số kích hoạt (*activation function*) và là một hàm phi tuyến tính.

- Trong một số tài liệu, khi nói đến số lớp của một mạng Neural, người ta sẽ không tính lớp Input Layer. Tức là số lớp của mạng Neural sẽ bằng (số Hidden

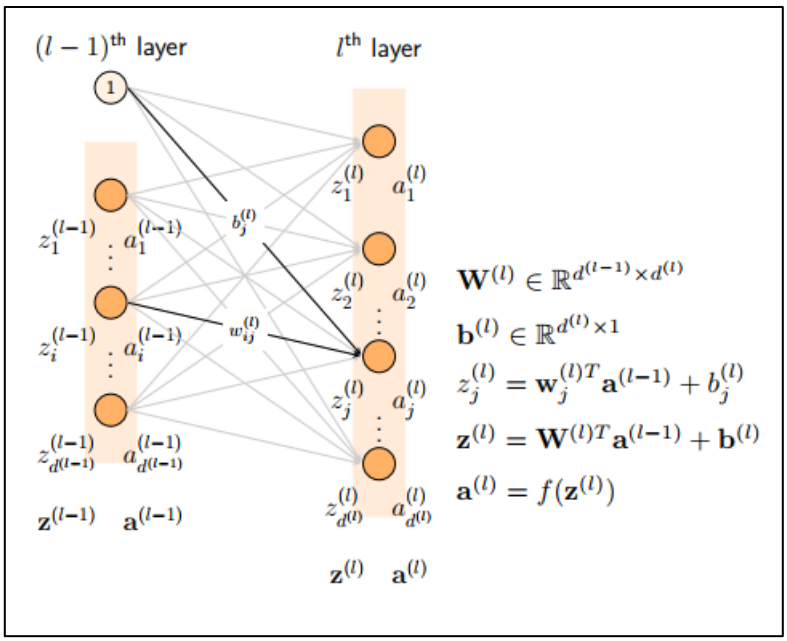
Layer + 1). Tuy nhiên trong đề tài này, em sẽ vẫn tính Input Layer khi nói đến số lớp của một mạng Neural.

b.2. Weight và bias:

- Ma trận trọng số (**Weight**) của layer *l* kí hiệu là $W^{(l)}$ ($W^{(l)} \in R^{d^{(l-1)} \times d^{(l)}}$ với $d^{(l)}$ là số Units (*nodes*) ở layer thứ *l*). Ma trận này thể hiện các kết nối hay biến đổi từ layer thứ *l*-1 sang layer thứ *l*. Cụ thể hơn, phần tử $w_{i,j}^{(l)}$ thể hiện kết nối từ *node* thứ *i* của layer *l*-1 đến *node* thứ *j* của layer *l*.
- ***Bias*** của lớp *l* là một vector được kí hiệu là $b^{(l)}$ ($b^{(l)} \in R^{d^{(l)}}$)
  - Để có thể xấp xỉ được mối quan hệ giữa các cặp điểm (x,y) trong tập huấn luyện đã nói ở trước, nhiệm vụ của Neural Network chính là tìm các **Weight** và **bias** này.

b.3. Units:

- Mỗi *node* hình tròn trong mỗi layer gọi là một *unit* (xem *Hình 1*).
- Ở layer thứ *l*, vector biểu diễn đầu vào kí hiệu là $z^{(l)}$, vector biểu diễn đầu ra kí hiệu là $a^{(l)}$ (đã đề cập ở ***b.1***). Đầu vào và đầu ra của *unit* thứ *i* trong layer *l* lần lượt được kí hiệu là $z_i^{(l)}$ và $a_i^{(l)}$ ($z_i^{(l)}, a_i^{(l)} \in R$)
  - Mối quan hệ giữa các đại lượng đã đề cập được thể hiện ở hình dưới đây (*nguồn* **[1]** *trang 201*):



Hình 2: Mối quan hệ giữa các đại lượng trong Neural Netwok

c. ***Cơ sở toán học cần thiết:***

c.1. ***Activation function:***

- ***Activation function*** f là một hàm phi tuyến tính. Hàm f này là cần thiết do ta không thể chỉ sử dụng hàm tuyến tính (Wx+b) để xấp xỉ mối qua hệ giữa các cặp điểm (x,y) được (vì các mối quan hệ này hầu hết không phải là quan hệ tuyến tính).
- Dưới đây là một số hàm phi tuyến tính thường được sử dụng làm *activation function*:
  - Hàm xác định dấu $sgn(x) = \begin{cases} 1 \; khi \; x > 0 \\ 0 \; khi \; x \leq 0 \end{cases}$
  - Hàm $sigmoid(x) = \frac{1}{1+e^{-x}}$
  - Hàm $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
  - Hàm $\text{ReLU}(x) = \max(0, x) = \begin{cases} x \; khi \; x > 0 \\ 0 \; khi \; x \leq 0 \end{cases}$
  - Hàm $softmax$ của một vector z ϵ R^C được định nghĩa như sau:

$$\text{a} = softmax(z) \Leftrightarrow a_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

- Trong phạm vi đề tài, em sẽ xây dựng mạng neural 3 lớp sử dụng hàm ReLU làm *activation function* cho Hidden Layer và hàm *softmax* làm *activation function* cho Output Layer để giải quyết bài toán phân loại chữ số viết tay.

c.2. FeedForward:

- Tính toán *FeedForward* (lan truyền tới) là bước thực hiện các phép toán trên các layer của network một cách lần lượt từ layer đầu (input layer) đến layer cuối (output layer).

- Cụ thể đối với mạng neural 3 lớp mà em sẽ xây dựng để giải quyết bài toán phân loại chữ số viết tay, bước *FeedForward* thực hiện các phép tính theo trình tự sau **[1]** :

$$\begin{aligned}\mathbf{Z}^{(1)} &= \mathbf{W}^{(1)T}\mathbf{X} + \mathbf{B}^{(1)}\\ \mathbf{A}^{(1)} &= \max(\mathbf{Z}^{(1)}, \mathbf{0})\\ \mathbf{Z}^{(2)} &= \mathbf{W}^{(2)T}\mathbf{A}^{(1)} + \mathbf{B}^{(2)}\\ \hat{\mathbf{Y}} &= \mathbf{A}^{(2)} = \text{softmax}(\mathbf{Z}^{(2)})\end{aligned}$$

  Trong đó, các điểm dữ liệu x_i là các vector cột được xếp cạnh nhau tạo thành ma trận đầu vào X, các đầu ra tương ứng y_i cũng là các vector cột và được xếp cạnh nhau tạo thành ma trận đầu ra tương ứng Y (các ma trận A, Z, B cũng được định nghĩa tương tự). $\hat{Y}$ là ma trận kết quả nhận được (kết quả dự đoán) sau khi thực hiện bước *FeedForward*.

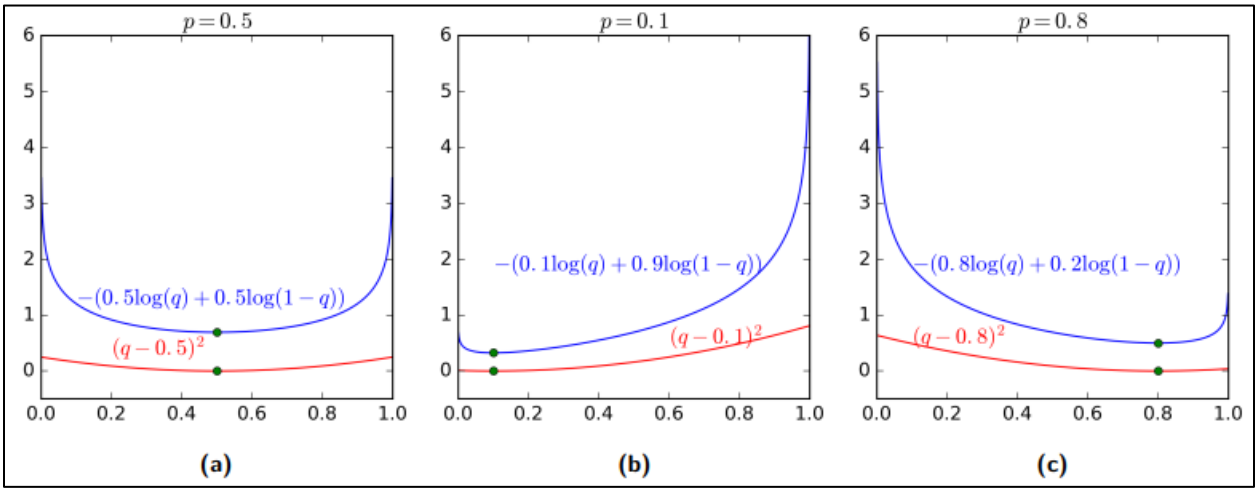
c.3. Backpropagation:

- Trước hết, ta cần nói về hàm mất mát (*loss function*): là hàm đánh giá khoảng cách (hay sai số) giữa kết quả dự đoán ($\hat{Y}$ nhận được sau bước *FeedForward*) và kết quả thực tế Y. Hàm này sẽ có kết quả càng nhỏ khi $\hat{Y}$ càng gần với Y và ngược lại. Các loại hàm mất mát phổ biến là:
  - Hàm MSE $= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$. Đây là một hàm mất mát đơn giản lấy ý tưởng từ khoảng cách Euclid nhưng đem lại hiệu quả cao trong rất nhiều trường hợp.

- Hàm cross-entropy giữa 2 vector phân phối rời rạc p và q có chiều C được định nghĩa bởi công thức sau:

$$-\sum_{i=1}^{C} p_i \log q_i \quad (2)$$

Hàm cross-entropy cho đánh giá tốt hơn MSE đối với bài toán phân phân loại chữ số viết tay mà em sẽ thực hiện do hàm này nhận giá trị rất cao khi khoảng cách giữa 2 xác suất p và q lớn. Hình dưới đây minh họa cho việc so sánh hàm cross-entropy và MSE:



Hình 3: So sánh cross-entropy và MSE [1]

- Sau khi nhận được $\hat{Y}$ từ bước *FeedForward*, mạng neural phải thực hiện thêm một bước nữa đó là điều chỉnh **Weights** và **bias** sao cho kết quả đánh giá của *loss function* là càng thấp càng tốt. Có như vậy thì ta mới nhận được được một xấp xỉ tốt cho mỗi quan hệ giữa các cặp điểm (x,y) trong tập huấn luyện.
- Phương pháp phổ biến nhất để thực hiện việc này là ***Gradient descent*** (*có thể xem chi tiết về Gradient descent ở tài liệu tham khảo [1] , trang 140*). Để có thể áp dụng được ***Gradient descent,*** ta cần tính đạo hàm của hàm mất mát (gọi là hàm *J*) theo từng ma trận trọng số $W^{(l)}$ và từng bias $b^{(l)}$. Việc tính đạo hàm này nếu làm trực tiếp sẽ vô cùng phức tạp, tuy nhiên nếu ta tính gián tiếp bằng cách sử dụng công thức đạo hàm của hàm hợp $\left(f\left(g(x)\right)\right)' = f'(g).g'(x)$ thì độ phức tạp sẽ được giảm đi rất nhiều. Chi tiết quá trình tính đạo hàm em sẽ

không đề cập trong bài báo cáo này vì quá nặng về phần toán học (*thay vào đó mọi người có thể xem ở tài liệu tham khảo* **[1]** *trang 184*).

- Một điều đặc biệt của bước này là ta sẽ phải tính toán ngược từ layer cuối lên layer đầu, bởi vậy mà bước này có tên gọi là *Backpropagation* (lan truyền ngược).
- Dưới đây em sẽ chỉ đề cập đến các tính toán mà mô hình mạng neural 3 lớp của em sẽ thực hiện trong bước *Backpropagation* (sử dụng hàm loss là cross-entropy) **[1]** :

$$\mathbf{E}^{(2)} = \nabla_{\mathbf{Z}^{(2)}} = \frac{1}{N}(\mathbf{A}^{(2)} - \mathbf{Y})$$

$$\nabla_{\mathbf{W}^{(2)}} = \mathbf{A}^{(1)}\mathbf{E}^{(2)T}; \quad \nabla_{\mathbf{b}^{(2)}} = \sum_{n=1}^{N} \mathbf{e}_n^{(2)}$$

$$\mathbf{E}^{(1)} = \left(\mathbf{W}^{(2)}\mathbf{E}^{(2)}\right) \odot f'(\mathbf{Z}^{(1)})$$

$$\nabla_{\mathbf{W}^{(1)}} = \mathbf{A}^{(0)}\mathbf{E}^{(1)T} = \mathbf{X}\mathbf{E}^{(1)T}; \quad \nabla_{\mathbf{b}^{(1)}} = \sum_{n=1}^{N} \mathbf{e}_n^{(1)}$$

Trong đó phép toán $\odot$ là tích chập của 2 ma trận (hoặc vector) có cùng kích thước, đây là phép tính nhân mỗi phần tử tương ứng của 2 ma trận (hoặc vector) với nhau, kết quả nhận được là một ma trận (hoặc vector) có kích thước giống với 2 ma trận (hoặc vector) ban đầu.

- Sau khi đã có các đạo hàm của hàm mất mát theo từng ma trận trọng số ($\nabla_{W^{(l)}}$) và bias ($\nabla_{b^{(l)}}$). Ta thực hiện bước ***Gradient descent*** để cập nhật lại ma trận trọng số và bias như sau (số ε được gọi là learning rate):

$$W^{(l)} \leftarrow W^{(l)} - \varepsilon\nabla_{W^{(l)}}$$

$$b^{(l)} \leftarrow b^{(l)} - \varepsilon\nabla_{b^{(l)}}$$

- Để có thể nhận được một xấp xỉ tốt của mối quan hệ giữa x và y, ta phải thực hiện 2 bước *FeedForward* và *Backpropagation* lặp đi lặp lại nhiều lần, mỗi lần sẽ cập nhật lại các ma trận trọng số và bias nhận được từ lần thực hiện trước. Quá trình lặp đi lặp lại này gọi là huấn luyện (***train***) mô hình.

2. Xây dựng mạng neural 3 lớp bằng Python

Em chọn Python để xây dựng mạng neural 3 lớp là do ngôn ngữ lập trình này có thư viện *numpy* hỗ trợ rất nhiều cho các phép toán trên ma trận và vector.

Sau đây em xin đi vào chi tiết xây dựng:

- Trước khi đi vào xây dựng, cần phải *import* thư viện *numpy* và các thư viện cần thiết:

```
import numpy as np
import matplotlib as plt
np.random.seed()
```

- Tiếp đến em sẽ xây dựng lại các hàm activation cần sử dụng và hàm loss cross-entropy (hàm d_relu là đạo hàm của hàm ReLU):

```
[57] def relu(s):
       m = 0 + s
       m[m < 0] = 0
       return m

     def d_relu(s):
       m = s + 0
       m[m > 0] = 1
       return m
```

```
[58] def softmax(s):
       e_s = np.exp(s - np.max(s, axis = 1, keepdims = True))
       result = e_s / e_s.sum(axis = 1, keepdims = True)
       return result

     #print(softmax(np.asarray([3,5,-1])))
```

```
[59] def cross_entropy_loss(Ypred, Y):
       loss = np.asarray([np.sum(np.multiply(Y[i], np.log(Ypred[i] + 1e-9))) for i in range(Y.shape[0])])
       return -np.mean(loss)
```

Ở đây, hàm cross_entropy_loss thực hiện tính toán trên hai ma trận gồm các vector cột là các vector phân phối rời rạc, chứ không phải chỉ với 2 vector phân phối rời rạc như ở công thức (2). Đây là hàm cross-entropy cho phương pháp **mini-batch** mà em sẽ đề cập ở phần tiếp theo.

- Tiếp theo là phần xây dựng lớp layer cho mạng Neural:

```python
class layer:
  def __init__(self, input_size, output_size, activation = "none"):
    self.input_size = input_size
    self.output_size = output_size
    if activation!="none":
      if activation == "relu":
        self.activation = relu
        self.derivative = d_relu
      elif activation == "softmax":
        self.activation = softmax
      else:
        raise ValueError('Activation not found')
      # initialize layer's Weight with random values belong to [-1, 1]
      self.W = np.random.randn(self.input_size, self.output_size)/np.sqrt(self.input_size*self.output_size)
      #initialize layer's bias with all zero
      self.b = 1.0*np.zeros((self.output_size), dtype = np.uint8)
    else:
      self.activation = "none"
      if (input_size!=output_size):
        raise ValueError('Input layer must have the same input_size and output_size')
  def __call__(self, input):
    if self.activation != "none": #if the layer is not Input Layer
      self.z = np.dot(input, self.W) + self.b # z = Wx + b
      self.a = self.activation(self.z) # a = f(x) where f is activation function
      return self.a
    else:
      return input
```

Ở dòng tính toán *self.z*, em có thay đổi một chút so với công thức (tính xW+b thay vì tính Wx+b vì vector b trong code của em là vector hàng)

- Cuối cùng là xây dựng lớp Neural Network (mạng neural 3 lớp) với các hàm *FeedForward* và *Backpropagation* thực hiện các bước tính toán em đã đề cập ở phần ***FeedForward:*** và ***Backpropagation:***:

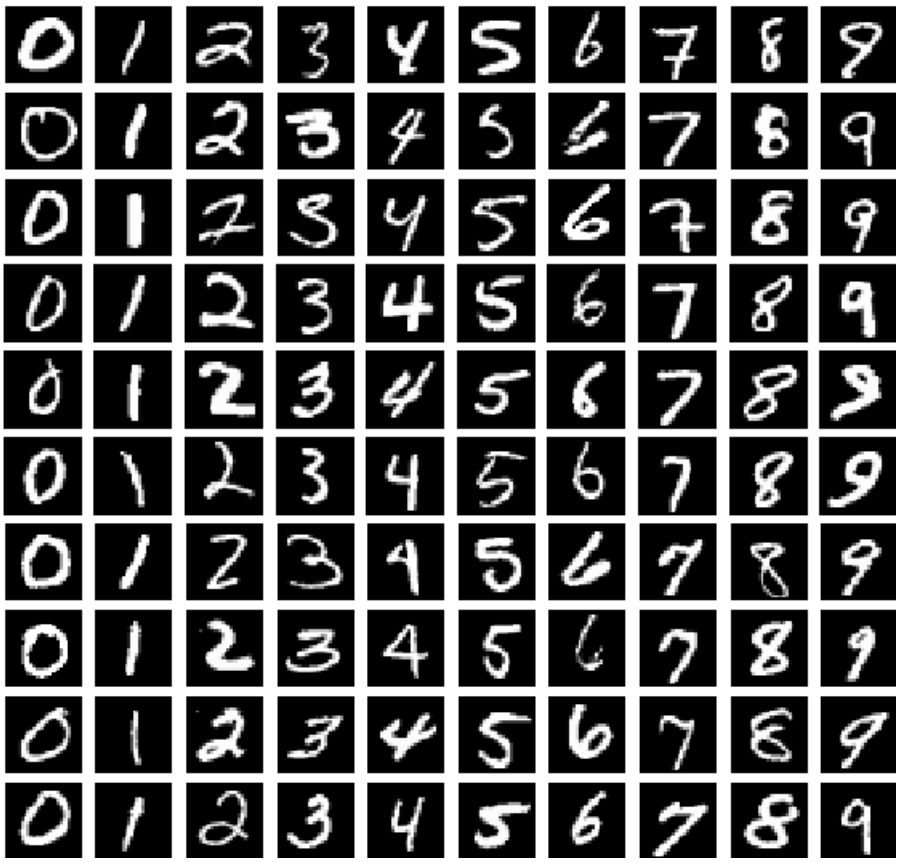
```
class Neural_network:
  def __init__(self, *args):
    self.Number_Of_Layers = len(args)
    self.L = []
    if len(args)!=3:
      raise ValueError(f'Number of layers must be 3')
    for arg in args:
      if len(self.L)!=0:
        if arg.input_size != self.L[len(self.L)-1].output_size:
          raise ValueError(f'input_size of layer {len(self.L)} must equal to output_size of layer {len(self.L)-1}')
      self.L.append(arg)
##########################################################################################################################
  def FeedForward(self, input):
    x = input
    for Layer in self.L:
      x = Layer(x) # Calculate output = g2(g1(x))
    return x
##########################################################################################################################
  def Backpropagation(self, X, Y):
    E = (self.L[-1].a - Y)/(self.L[-1].a.shape[0]) # E2 = (A2-Y)/N
    delta_W = []
    d_W = np.dot(self.L[-2].a.T, E) # delta_W2 = A1.E2^T
    delta_W.append(d_W)
    delta_b = []
    d_b = np.sum(E, axis = 0) # delta_b2 = sigma(e(2)_n)
    delta_b.append(d_b)

    E = np.multiply(np.dot(E, self.L[-1].W.T), self.L[-2].derivative(self.L[-2].z)) # E1 = (W2.E2) x f'(Z1)
    d_W = np.dot(X.T, E) # delta_W1 = X.E1^T
    delta_W.insert(0,d_W)
    d_b = np.sum(E, axis = 0) # delta_b1 = sigma(e(1)_n)
    delta_b.insert(0, d_b)

    return delta_W, delta_b
##########################################################################################################################
```

Tương tự như khi code phần layer, ở phần code của *Backpropagation* em đã phải điều chỉnh thứ tự các nhân tử trong phép nhân 2 ma trận cho phù hợp với với kích thước các ma trận **Weights** và vector **bias** mà em đã khởi tạo trước đó. Do đó các phép tính ở code có thể khác với các phép tính ở lý thuyết, nhưng không ảnh hưởng nghiêm trọng đến kết quả sau cùng (chỉ chuyển vị kết quả đầu ra, thay vì vector cột thì lại thành vector hàng).

3. Huấn luyện và kiểm tra mạng neural đã xây dựng trên bộ dữ liệu MNIST

a. Giới thiệu về bộ dữ liệu MNIST:

- Bộ dữ liệu MNIST là bộ dữ liệu lớn nhất về chữ số viết tay. MNIST bao gồm 2 tập: Tập huấn luyện (Training set) có tổng cộng 60000 bức ảnh về các chữ số viết tay từ 0 đến 9 và 60000 nhãn tương ứng cho từng bức ảnh. Tập dữ liệu kiểm tra (test set) có 10000 bức ảnh và 10000 nhãn tương ứng (*nguồn: https://machinelearningcoban.com/2017/01/04/kmeans2/*). Dưới đây là một vài bức ảnh trong bộ dữ liệu MNIST:



Hình 4: Một vài hình ảnh trong bô dữ liệu MNIST

- Mỗi bức ảnh trong bộ dữ liệu là một bức ảnh grayscale, có kích thước 28x28 pixel (tổng cộng có 784 pixels). Mỗi pixel mang một giá trị từ 0 đến 255 biểu thị "độ trắng" của pixel đó, pixel càng trắng thì giá trị càng cao và màu đen có giá trị là 0. Vì điều này mà mỗi bức ảnh trong MNIST có thể được lưu trong một mảng 2 chiều 28 hàng 28 cột.
- Trong thư viện keras.datasets của Python đã có sẵn bộ dữ liệu MNIST và dưới đây là cách để tải được bộ dataset này:

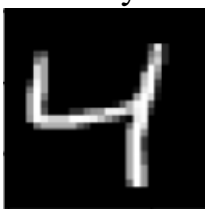
```
from keras.datasets import mnist
(train_X, train_y), (test_X, test_y) = mnist.load_data()
```

b. Huấn luyện và kiểm tra mạng neural đã xây dựng trên bộ dữ liệu MNIST (bài toán phân loại chữ số viết tay):

b.1. Huấn luyện mạng:

- Trước tiên em sẽ nói về input và output của bài toán phân loại chữ số viết tay:

- Input: bức ảnh 28x28 pixel từ bộ dữ liệu MNIST
- Output: Nhãn tương ứng cho bức ảnh đó, nhãn biểu thị số trên bức ảnh là số nào. Ví dụ với bức ảnh dưới đây thì nhãn của nó có thể là 4:



- Tiếp đến em xin trình bày cách giải quyết của mình cho bài toán này, sử dụng mạng neural 3 lớp mà mình đã xây dựng:
  - Vì bức ảnh có 784 pixels nên em sẽ khởi tạo mạng neural 3 lớp với input layer có 784 node, mỗi node theo thứ tự sẽ lưu một pixel của ảnh. Output layer của mạng sẽ có 10 node từ 0 đến 9, mỗi node *i* biểu thị xác suất bức ảnh sẽ có nhãn *i* là bao nhiêu. Hidden layer sẽ có 1024 node (số node ở hidden layer có thể thay đổi được).

```
model = Neural_network(
    layer(input_size = 28*28, output_size = 28*28),
    layer(input_size = 28*28, output_size = 1024, activation = "relu"),
    layer(input_size = 1024, output_size = 10, activation = "softmax")
)
```

Tuy nhiên, do bức ảnh là một mảng 2 chiều nhưng input layer lại nhận vào một vector 28x28 nên em phải vector hóa bức ảnh để có thể truyền vào input layer của mạng:

```
trainX = np.asarray([train_X[i].flatten() for i in range(train_size)]) #vectorize image
testX = np.asarray([test_X[i].flatten() for i in range(test_size)]) #vectorize image
```

Thêm một điều nữa là các nhãn tương ứng của bức ảnh là một số vô hướng (*scalar*) trong khi đầu ra của mạng là một vector có 10 phần tử biểu thị xác suất. Vì vậy, để tiện lợi cho việc tính toán sau này, em cũng sẽ chuyển các nhãn về các vector 10 phần tử, phần tử có chỉ mục tương ứng với con số mà nhãn biểu thị sẽ có giá trị 1, còn lại có giá trị 0 (ví dụ nếu nhãn là số 5 thì phần tử có chỉ mục 5 của vector sẽ có giá trị 1, còn lại có giá trị 0). Cách chuyển đổi này là hợp lí do số 1 tương ứng với xác suất 100%. Vector sau khi được chuyển đổi được gọi là một vector ở dạng **one-hot**.

Dưới đây là phần code thực hiện việc chuyển đổi một số vô hướng về một vector one-hot:

```
def to_one_hot(labels, dimension):
  result =np.zeros((len(labels), dimension), dtype = np.uint8)
  for i, label in enumerate(labels):
    result[i, label] = 1
  return np.asarray(result)
```

- Tiếp theo em sẽ trình bày về phương pháp huấn luyện (**train**) mô hình mà em sử dụng – **mini batch**:
    - Khác với phương pháp huấn luyện trên toàn bộ dữ training set (*full batch*), phương pháp *mini batch* chia training set thành nhiều *mini-batch* (lô) nhỏ hơn rất nhiều so với kích thước tập huấn luyện. Mỗi lần lặp sẽ lần lượt lấy ra và chỉ tính toán *Feedforward* và *Backpropagation* trên một mini batch để cập nhật các tham số của mô hình (Weights và bias). Việc này sẽ được thực hiện qua nhiều **epoch**, mỗi **epoch** là một lần duyệt qua hết tất cả các *mini batch*. Phương pháp huấn luyện này có một ưu điểm so với phương pháp huấn luyện *full batch* là khi tập huấn luyện lớn, các tính toán khi dùng *full batch* sẽ tốn rất nhiều thời gian, trong khi thời gian tính toán trên *mini batch* là rất nhỏ và hiệu quả nhận được rất cao.
    - Ngoài ra, để theo dõi tiến độ huấn luyện, em sẽ trích ra *mini-batch* cuối cùng trong tập huấn luyện để dự đoán (*predict*) và theo dõi độ chính xác, *mini-batch* được trích ra này gọi là tập kiểm thử (*validation set*)

Dưới đây là phần code thực hiện việc chia và huấn luyện *mini-batch*:

```
def fit(self, X, Y, epoches, eta, batch_size):
  numberOfBatch = math.ceil(X.shape[0]/batch_size)
  loss_array = []
  lenLoss = 0
  random.seed()
  self.val_array = []
  self.acc_array = []
  for epoch in range(epoches):
    loss = 0
    LR = self.lr_exp_decay(epoch, eta) # adjust learning rate
    val_array = []
    acc_array = []
    for batch in range(numberOfBatch-1):
      TrainX = X[batch*batch_size : (batch+1)*batch_size - 1] #create mini-batchX
      TrainY = Y[batch*batch_size : (batch+1)*batch_size - 1] #create mini-batchY
      TestX = X[(numberOfBatch - 1)*batch_size : ] # use last mini-batch to validate
      TestY = Y[(numberOfBatch - 1)*batch_size : ]
      test_result = self.predict(TestX)
      val_array.append(self.val(test_result, TestY))
      ypred = self.FeedForward(TrainX)
      acc_array.append(self.val(ypred, TrainY))
      loss = cross_entropy_loss(ypred, TrainY)
      (dW, db) = self.Backpropagation(TrainX, TrainY)
      for i in range(1, self.Number_Of_Layers):
        self.L[i].W += -LR*dW[i-1]  # adjust W
        self.L[i].b += -LR*db[i-1]  # adjust b
    val_acc = np.mean(np.asarray(val_array)) # calculate accuracy on validation set of each epoch
    acc = np.mean(np.asarray(acc_array)) # calculate accuray on trained set of each epoch
    self.val_array.append(val_acc)
    self.acc_array.append(acc)
    print(f"epoch {epoch} - loss: {loss} - val_acc: {val_acc} - acc: {acc} - learning rate: {LR}")
    dist = abs(val_acc - acc)
```

Tiếp theo là phần huấn luyện mô hình: em huấn luyện mô hình với 40 epochs, *learning rate* khởi điểm là 0.5 (learning rate sẽ giảm dần theo hàm số mũ sau mỗi epoch, chi tiết xem tại: https://neptune.ai/blog/how-to-choose-a-learning-rate-scheduler), kích thước của *mini-batch* là 512:

```
Num_epochs = 40
model.fit(trainX, trainY, epochs = Num_epochs, eta = 0.5, batch_size = 512)
```

Và dưới đây là kết quả huấn luyện ở các epoch cuối cùng (việc huấn luyện diễn ra trong khoảng 8 phút):

```
epoch 35 - loss: 0.07619768933324834 - val_acc: 0.9895833333333334 - acc: 0.9830899693913395 - learning rate: 0.01509869171115925
epoch 36 - loss: 0.07607024019842626 - val_acc: 0.9895833333333334 - acc: 0.9831568735678327 - learning rate: 0.01366186122364628
epoch 37 - loss: 0.07595171403191234 - val_acc: 0.9895833333333334 - acc: 0.9831903256560793 - learning rate: 0.012361763235169694
epoch 38 - loss: 0.07584533267270006 - val_acc: 0.9895833333333334 - acc: 0.9832739558766959 - learning rate: 0.011185385928082795
epoch 39 - loss: 0.07574992323369839 - val_acc: 0.9895833333333334 - acc: 0.9833074079649424 - learning rate: 0.01012095572290219
```

b.2. Kiểm tra (predict) mạng đã huấn luyện trên test set:

Kết quả predict của mạng trên **test set** sau khi huấn luyện như trên là: `0.9746`

Em cũng đã kiểm tra sự ảnh hưởng của số node ở hidden layer và kích thước của mini-batch đến kết quả dự đoán sau cùng (cùng huấn luyện trong 40 epochs) và thu được kết quả sau:

- Số node = 512, batch-size = 256: acc = 0.9697, train trong 5p
- Số node = 512, batch-size = 512: acc = 0.9702, train trong 4p
- Số node = 512, batch-size = 1024: acc = 0.9661, train trong 4p
- Số node = 1024, batch-size = 256: acc = 0.9751, train trong 9p
- Số node = 1024, batch-size = 512: acc = 0.9746, train trong 8p
- Số node = 1024, batch-size = 1024: acc = 0.9674, train trong 8p

Từ kết quả trên, có thể rút ra nhận xét là độ chính xác của dự đoán sẽ cao hơn khi số node nhiều hơn và batch-size nhỏ hơn. Tuy nhiên các con số này phải thuộc một khoảng nào đó, không được quá lớn (sẽ gây hiện tượng overfit) cũng như quá nhỏ (sẽ làm cho các Weights và bias chưa được cập nhập đủ tốt), khiến cho độ chính xác của kết quả dự đoán trên test set giảm xuống một cách trầm trọng (*chi tiết về overfit có thể xem ở tài liệu tham khảo* **[1]** *, trang 91*).

III. Kết luận

Mạng Neural 3 lớp mà em xây dựng thể hiện khá tốt trong việc giải quyết bài toán phân loại chữ số viết tay. Tuy nhiên, mạng neural này còn tồn tại một hạn chế là khi chọn batch-size quá nhỏ (dưới 300) thì đôi khi hiện tượng overfit có thể làm cho kết quả đầu ra của bước *FeedForward* rất gần với 0, điều này làm cho khi tính toán hàm *cross-entropy* sẽ tính toán *log(0) = infinity* khiến cho chương trình bị lỗi. Dù em cộng thêm một số rất nhỏ là 1e-9 ở hàm *cross-entropy* để khắc phục sự cố này nhưng chưa thể khắc phục được hoàn toàn. Vì vậy, khi chạy chương trình của em, hy vọng người chạy chương trình sẽ đặt batch-size và số node phù hợp (tốt nhất là từ 512 đến 1024) để tránh hiện tượng lỗi xảy ra. Em sẽ tiếp tục khắc phục sự cố này và mở rộng mạng Neural lên nhiều lớp hơn.

Tài liệu tham khảo

[1] Vũ Hữu Tiệp. "Machine Learning cơ bản". Link sách trên github: https://github.com/tiepvupsu/ebookMLCB/blob/master/book_ML_color.pdf

[2] Francois Chollet. "Deep Learning with Python".

Code đầy đủ

https://github.com/LCCuong/LCCuong/blob/gh-pages/Three_layers_NeuralNetwork.ipynb