

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH



AI PROJECT ASSIGNMENT (CO3101)

Hand-Written Digit Recognition

GVHD: Nguyễn Tiến Thịnh
Lớp: L05
SVTH: Lý Gia Huy 2010289



Mục lục

1 Giới thiệu đề tài	2
1.1 Khái quát	2
1.2 Ý nghĩa khoa học và thực tiễn	2
1.3 Phạm vi ứng dụng của bài toán	3
2 Mô tả bài toán	3
2.1 Giới thiệu	3
2.2 Thách thức	3
3 Cơ sở lý thuyết	4
3.1 Neural Network	4
3.1.1 Giới thiệu	4
3.1.2 Khái niệm	4
3.1.3 Nơ ron	4
3.1.4 Hàm kích hoạt - Activation Function	5
3.1.5 Kiến trúc mạng	6
3.2 Convolutional Neural Networks - CNNs	6
3.2.1 Giới thiệu	6
3.2.2 Lớp tích chập - Convolutional Layer	6
3.2.3 Bước nhảy - Stride	9
3.2.4 Thủ thuật đệm - Padding	10
3.2.5 Lớp phi tuyến - Non-linear Layer	11
3.2.6 Lớp gộp - Pooling Layer	11
3.2.7 Fully Connected Layer	12
3.2.8 Hàm Softmax	13
3.2.9 Hiện tượng Overfitting	14
3.2.10 Validation	15
3.2.11 Tăng cường dữ liệu - Data Augmentation	15
3.2.12 Bỏ học - Dropout	15
3.3 Thuật toán Gradient Descent	16
3.3.1 Giới thiệu	16
3.3.2 Định nghĩa	16
3.3.3 Stochastic Gradient Descent - SGD	16
3.3.4 Batch Gradient Descent	17
3.3.5 Tối ưu Gradient Descent với Momentum	17
3.3.6 Tối ưu Gradient Descent với Nesterov	18
3.3.7 Điều chỉnh learning rate cho kết quả tốt hơn	18
4 Thiết kế và hiện thực	20
4.1 Tập dữ liệu dataset	20
4.2 Môi trường lập trình, công cụ hỗ trợ	20
4.3 Các bước tiến hành	20
4.3.1 Import các thư viện cần thiết	20
4.3.2 Load tập dữ liệu MNIST	21
4.3.3 Chuẩn bị dữ liệu - Tiền xử lý	22
4.3.4 Thiết kế model Neural Network	23
4.3.5 Question 1	24
4.3.6 Question 2	25
4.3.7 Question 3	27
4.3.8 Question 4	28
4.4 Dánh giá kết quả	30
4.5 Chạy thử chương trình	30
Tài liệu tham khảo	31

1 Giới thiệu đề tài

1.1 Khái quát

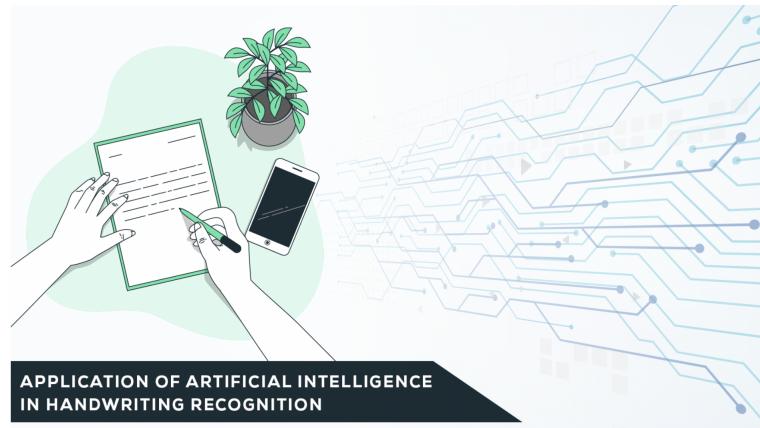
Trong công việc thường nhật, mặc dù các tài liệu, hồ sơ, đơn thư, chứng từ bằng giấy rất tiện lợi trong giao dịch, chúng thường dễ bị thất lạc và hư hỏng. Vì vậy, các doanh nghiệp thường phải bỏ ra một chi phí lớn để bảo quản, lưu trữ. Ngoài ra, việc tìm kiếm, sao chép, chỉnh sửa một phần hoặc toàn bộ văn bản cũng mất rất nhiều thời gian.

Vì những lí do trên, việc số hóa tài liệu giấy trở thành một nhu cầu bức thiết của nhiều người dân, doanh nghiệp và các tổ chức trong nước. Để làm được điều này, chúng ta cần đến công nghệ nhận dạng chữ viết tay.

Công nghệ nhận dạng chữ viết tay không chỉ được áp dụng trên giấy, mà còn có thể nhận dữ liệu từ các đầu vào khác như bảng nhựa, màn hình cảm ứng cũng như nhiều thiết bị khác.

Lợi ích của công nghệ nhận dạng chữ viết tay:

- Nhận dạng và trích xuất thông tin nhanh chóng với số lượng lớn tài liệu trong thời gian ngắn.
- Tiết kiệm thời gian nhập liệu thủ công.
- Giảm thiểu nhân lực nhập liệu.
- Tăng năng suất và hiệu quả công việc.
- Bảo quản, lưu trữ, trích xuất các tài liệu tốt hơn.



Phạm vi nghiên cứu của chúng ta là **nhận dạng chữ số viết tay** - một khía cạnh của đề tài **Handwriting Recognition**

1.2 Ý nghĩa khoa học và thực tiễn

- Đề xuất giải pháp đơn giản để thực hiện bài toán nhận diện chữ số viết tay.
- Áp dụng học sâu vào việc giải quyết bài toán nhận dạng chữ số viết tay.

1.3 Phạm vi ứng dụng của bài toán

Các ứng dụng của nhận dạng chữ số viết tay rất đa dạng, đơn cử như: đọc số CCCD/CMND, số BHYT, số tài khoản ngân hàng, biển số xe Nhận dạng chữ số viết tay còn giúp quản lý, lưu trữ các số liệu, sổ khám bệnh, giấy tờ hành chính một cách hiệu quả.



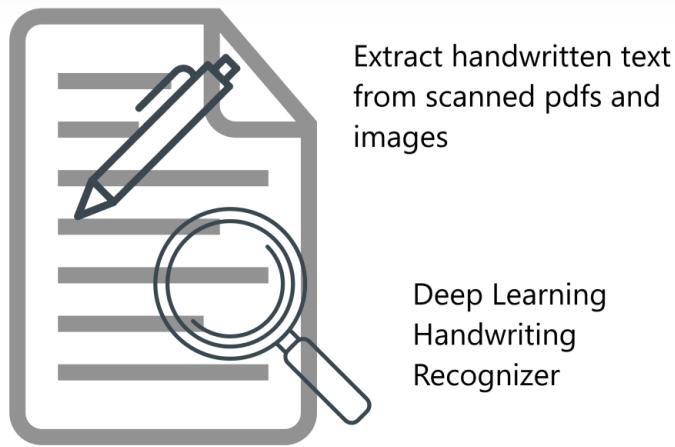
Nhận dạng chữ số viết tay cũng đóng vai trò quan trọng đối với thư viện số, cho phép nhập thông tin dạng văn bản hình ảnh vào máy tính bằng các phương pháp số hóa, phục hồi hình ảnh và nhận dạng.

2 Mô tả bài toán

2.1 Giới thiệu

Nhận dạng chữ số viết tay là khả năng máy tính nhận và diễn dịch tài liệu viết tay từ giấy, ảnh, màn hình cảm ứng và các thiết bị khác. Với nhận dạng chữ số viết tay, thiết bị sẽ diễn giải các số viết tay của người dùng sang định dạng mà máy tính hiểu được.

Có nhiều cấp độ nhận dạng chữ số, bắt đầu từ nhận dạng từng số riêng lẻ đến nhận dạng một chuỗi các số viết tay.



Ở đây, chúng ta có tập dữ liệu mẫu các bản viết tay của chữ số riêng lẻ. Yêu cầu đặt ra là hệ thống cần nhận diện được các số kiểm thử dựa trên tập dữ liệu cho trước. Để làm được điều này, chúng ta sẽ dùng tới thuật toán Deep Learning.

2.2 Thách thức

- Có sự khác biệt lớn giữa nét chữ của những người khác nhau.
- Phong cách viết tay của mỗi người cũng không nhất quán, thay đổi theo thời gian.
- Chất lượng kém của tài liệu / hình ảnh nguồn do xuống cấp có thể gây cản trở.
- Các số viết tay thường không thẳng hàng như văn bản đánh máy.

- Thu thập một tập dữ liệu tốt để học không dễ dàng.

3 Cơ sở lý thuyết

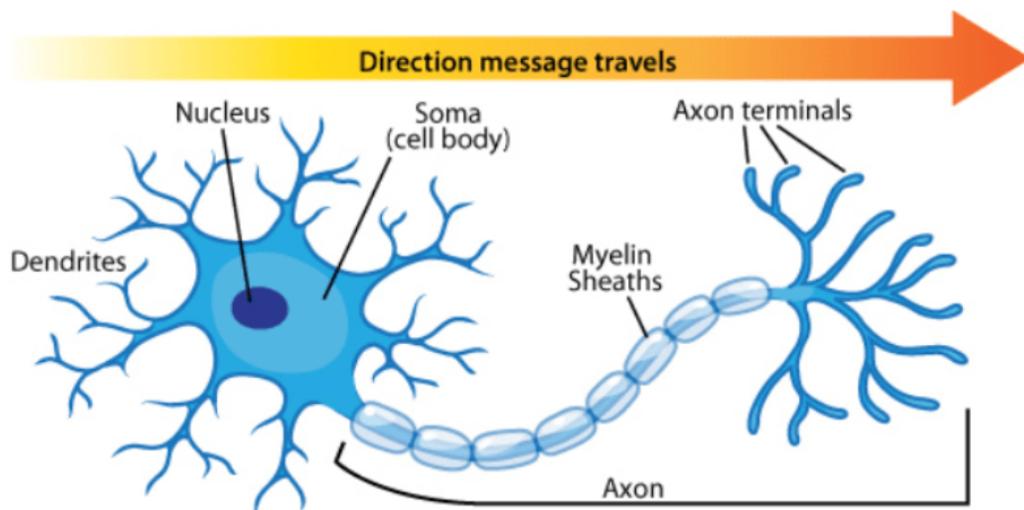
3.1 Neural Network

3.1.1 Giới thiệu

Deep Learning là một tập hợp con của Machine Learning dựa trên mạng thần kinh nhân tạo (gọi tắt là mạng nơron), có sự khác biệt ở một số khía cạnh quan trọng so với Học Máy “nông”. Nó cho phép máy tính giải quyết một loạt các vấn đề phức tạp trong thực tế mà các mô hình cũ không thể giải quyết được. Nền tảng trung tâm của Học Sâu là mạng nơron (Neural Network), mô phỏng tế bào nơron nằm sâu trong não con người.

Các tế bào thần kinh được gọi là nơron có chức năng dẫn truyền xung động thần kinh. Mỗi nơron nhận xung động thần kinh vào từ nơron đứng trước nó, truyền dọc theo axon và truyền ra nơron đứng sau nó thông qua synapse.

Neuron Anatomy



Hình 1: Cấu trúc của một nơron

Xung động thần kinh truyền từ nơron này sang nơron khác trong mạng nơron tạo nên mọi chức năng thần kinh của não bộ.

Chúng ta đều biết, bộ não của con người là một trong những nơi phức tạp nhất, tinh vi nhất. Mô phỏng được cách não bộ học tập luôn là ước mơ của các nhà khoa học. Thuật toán mạng nơron xuất phát từ tham vọng đó.

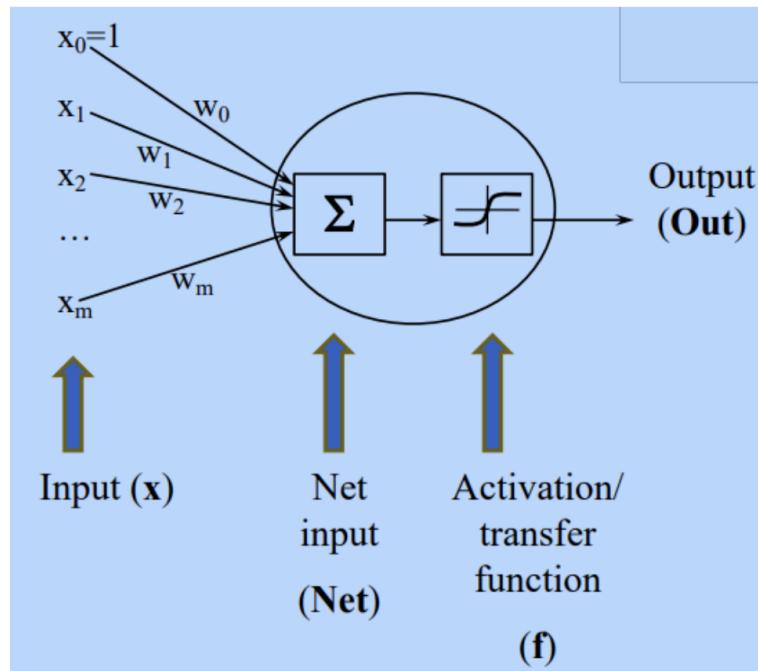
3.1.2 Khái niệm

Mạng nơron nhân tạo (Artificial Neural Networks) mô phỏng lại mạng nơron sinh học là một cấu trúc khống gồm các đơn vị tính toán đơn giản được liên kết chặt chẽ với nhau, trong đó các liên kết giữa các nơron quyết định chức năng của mạng. Đơn vị cơ bản của mạng nơron nhân tạo là các nơron.

3.1.3 Nơron

Mỗi nơron sẽ nhận tín hiệu vào từ các nơron phía trước hay một nguồn bên ngoài và sử dụng chúng để tính tín hiệu ra, sau đó sẽ lan truyền sang các nơron khác.

Cấu tạo của nơron:



Hình 2: Cấu tạo của một nơ ron

- x_i với $i = 1..m$ là các tín hiệu đầu vào
- w_i với $i = 1..m$ là các trọng số tương ứng với x_i
- w_0 là độ lệch (bias) với $x_0 = 1$
- Net Input là sự kết hợp của các tín hiệu đầu vào $Net(w, x)$, tính bằng công thức:

$$Net(w, x) = \sum_{i=0}^m w_i x_i$$

- f là hàm activation/transfer
- Đầu ra Output: $Out = f(Net(w, x))$

3.1.4 Hàm kích hoạt - Activation Function

Phần lớn các nơron chuyển net input bằng cách sử dụng một hàm vô hướng (scalar-to-scalar function) gọi là hàm kích hoạt, kết quả của hàm này là một giá trị gọi là mức độ kích hoạt của đơn vị (unit's activation). Một số hàm kích hoạt thường được sử dụng là:

1. Linear function

$$Out(Net) = max(0, Net)$$

- Được sử dụng phổ biến
- Linear function dễ tính toán
- Là một hàm liên tục

2. Sigmoid

$$Out(Net) = \frac{1}{1 + e^{-\alpha(Net+\theta)}}$$

- α là hệ số góc
- Linear function dễ tính toán
- Là một hàm liên tục

3.1.5 Kiến trúc mạng

Kiến trúc của mạng được định nghĩa bởi: số lớp (layers), số nơ ron trên mỗi lớp, và sự liên kết giữa các lớp như thế nào. Các mạng về tổng thể được chia thành hai loại dựa trên cách thức liên kết các nơ ron

1. Mạng truyền thẳng - Feed-forward neural network

Dòng dữ liệu từ đơn vị đầu vào đến đơn vị đầu ra chỉ được truyền thẳng. Việc xử lý dữ liệu có thể mở rộng ra nhiều lớp, nhưng không có các liên kết phản hồi. Nghĩa là, các liên kết mở rộng từ các đơn vị đầu ra tới các đơn vị đầu vào trong cùng một lớp hay các lớp trước đó là không được phép.

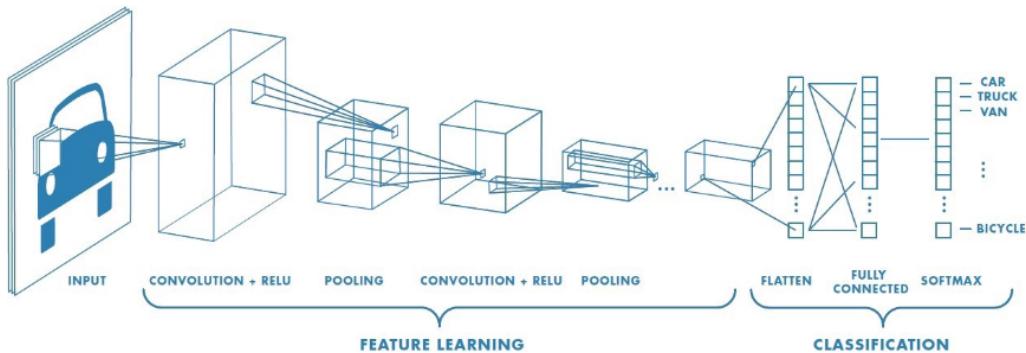
2. Mạng hồi quy - Recurrent neural network

Có chứa các liên kết ngược. Khác với mạng truyền thẳng, các thuộc tính động của mạng mới quan trọng. Trong một số trường hợp, các giá trị kích hoạt của các đơn vị trải qua quá trình nổi lồng (tăng giảm số đơn vị và thay đổi các liên kết) cho đến khi mạng đạt đến một trạng thái ổn định và các giá trị kích hoạt không thay đổi nữa. Trong các ứng dụng khác mà cách chạy tạo thành đầu ra của mạng thì những sự thay đổi các giá trị kích hoạt là đáng quan tâm.

3.2 Convolutional Neural Networks - CNNs

3.2.1 Giới thiệu

Convolutional Neural Networks (CNNs) là một cụm từ dùng để chỉ mạng nơ ron tích chập. Đây là mô hình Deep Learning tiên tiến cho phép chúng ta sử dụng được các hệ thống thông tin với độ chính xác vô cùng cao. CNN được ứng dụng phổ biến trong những bài toán nhận dạng object trong ảnh.



Hình 3: Kiến trúc CNNs phổ biến

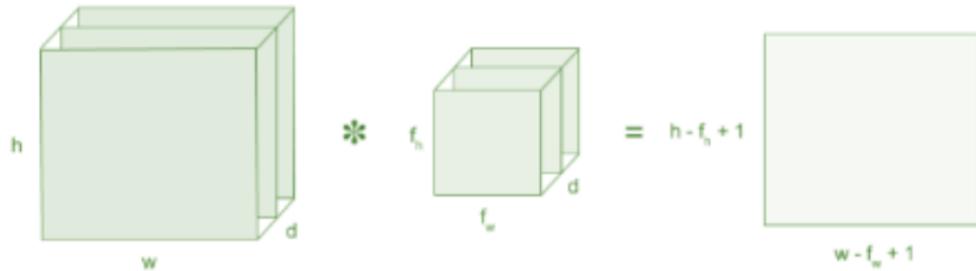
Kiến trúc của CNN tương tự như cấu trúc của các tế bào thần kinh trong não người và được lấy cảm hứng từ tổ chức của Vỏ não thị giác. Các tế bào thần kinh riêng lẻ chỉ phản ứng với các kích thước trong một vùng giới hạn của trường thị giác, được gọi là trường tiếp nhận. Tập hợp các trường tiếp nhận chồng lên nhau bao phủ toàn bộ vùng thị giác, cho phép con người thấy được toàn cảnh.

3.2.2 Lớp tích chập - Convolutional Layer

Lớp tích chập (gọi tắt là Conv Layer) là lớp quan trọng nhất và cũng là lớp đầu tiên của mô hình CNN. Lớp này có chức năng chính là phát hiện các đặc trưng có tính không gian hiệu quả.

Trong tầng này có 4 đối tượng chính là: ma trận đầu vào, bộ lọc, và trường tiếp nhận (receptive field) và feature map. Conv layer nhận đầu vào là một ma trận 3 chiều và một bộ lọc cần phải học. Bộ lọc này sẽ trượt qua từng vị trí trên bức ảnh để tính tích chập (convolution) giữa bộ lọc và phần tương ứng trên bức ảnh. Phần tương ứng này gọi là trường tiếp nhận, tức là vùng mà một nơ-ron có thể nhìn thấy để đưa ra quyết định, và ma trận cho ra bởi quá trình này được gọi là feature map.

- An image matrix (volume) of dimension $(h \times w \times d)$
- A filter $(f_h \times f_w \times d)$
- Outputs a volume dimension $(h - f_h + 1) \times (w - f_w + 1) \times 1$



Để cho dễ hình dung về chức năng cũng như cách tiến hành của lớp Conv, chúng ta sẽ cùng xem xét ví dụ đơn giản sau:

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0



5 x 5 – Image Matrix

1	0	1
0	1	0
1	0	1

3 x 3 – Filter Matrix

Khi xét image và bộ lọc như trên thì lớp tích chập của chúng sẽ gọi là *Feature Map*

1	1	1	0	0
0	1	1	1	0
0	0	1 _{x1}	1 _{x0}	1 _{x1}
0	0	1 _{x0}	1 _{x1}	0 _{x0}
0	1	1 _{x1}	0 _{x0}	0 _{x1}

Image

4	3	4
2	4	3
2	3	4

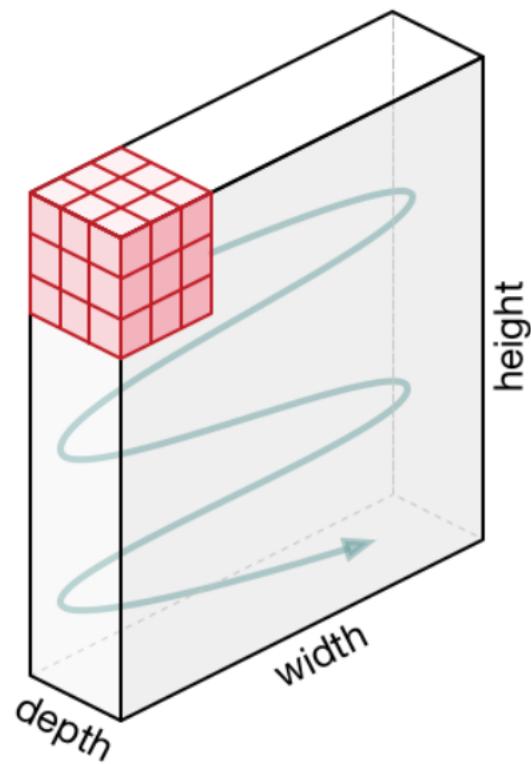
Convolved Feature

Sự kết hợp của 1 hình ảnh với các bộ lọc khác nhau có thể thực hiện các hoạt động như phát hiện cạnh, làm mờ và làm sắc nét bằng cách áp dụng các bộ lọc. Ví dụ dưới đây cho thấy hình ảnh tích chập khác nhau sau khi áp dụng các Kernel khác nhau.

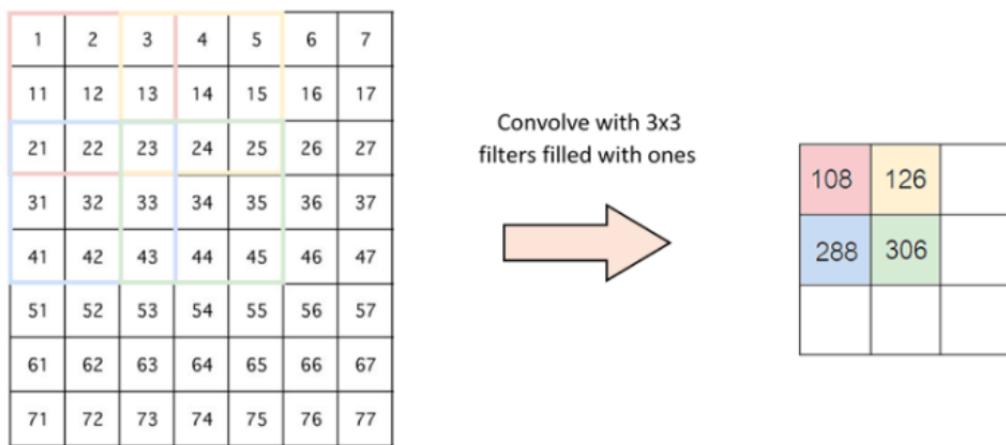
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

3.2.3 Bước nhảy - Stride

Bước nhảy (Stride) là số pixel thay đổi trên ma trận đầu vào. Khi bước nhảy là 1 thì ta di chuyển các kernel 1 pixel. Khi bước nhảy là 2 thì ta di chuyển các kernel di 2 pixel và tiếp tục như vậy.

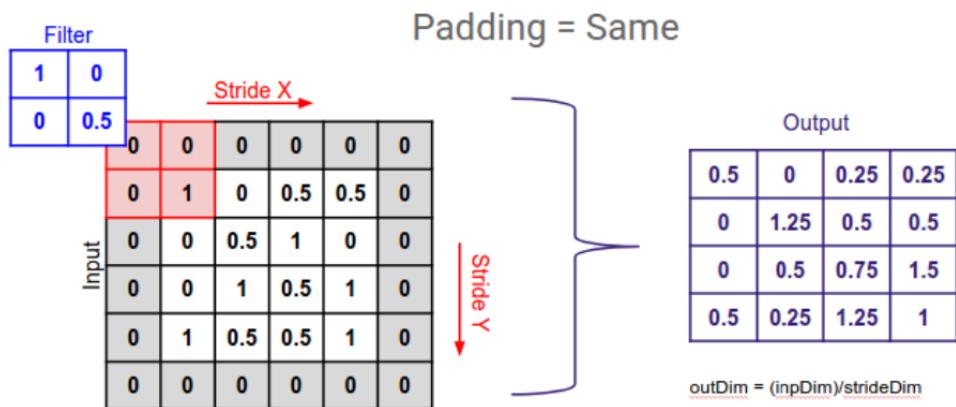


Khi ta cho stride = 2 thì các kernel hoạt động sẽ như sau:



3.2.4 Thủ thuật đệm - Padding

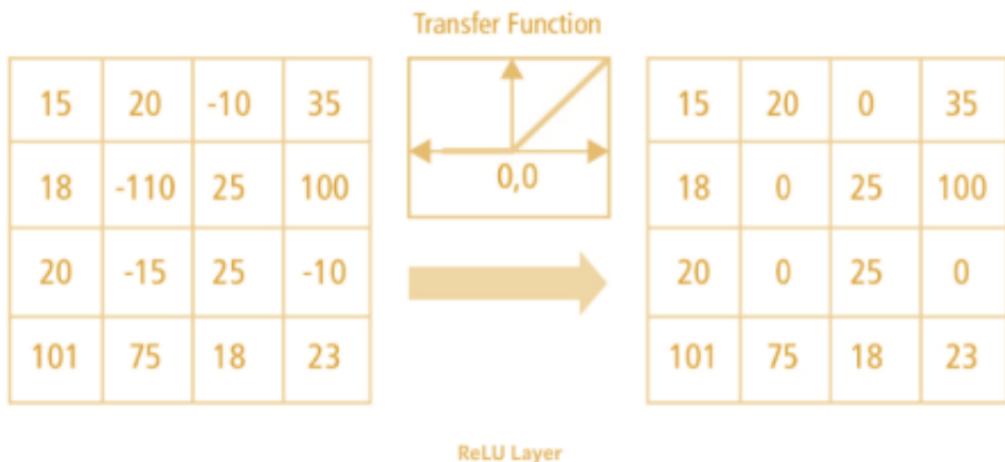
Khi áp dụng phép tích chập thì ma trận đầu vào sẽ nhỏ dần đi khi stride, do đó số lớp của mô hình CNN sẽ bị giới hạn, và chúng ta không thể xây dựng nơ ron như ý muốn. Để giải quyết tình trạng này, chúng ta cần đệm (padding) vào ma trận đầu vào để đảm bảo kích thước đầu ra sau mỗi tầng tích chập là không đổi, từ đó có thể xây dựng được mô hình với số tầng tích chập lớn tùy ý.



Một cách đơn giản và phổ biến nhất để padding là sử dụng hàng số 0, ngoài ra cũng có thể sử dụng reflection padding hoặc là symmetric padding.

3.2.5 Lớp phi tuyến - Non-linear Layer

ReLU viết tắt của Rectified Linear Unit, là 1 hàm phi tuyến với đầu ra là: $f(x) = \max(0, x)$. *ReLU* sẽ gán những giá trị âm bằng 0 và giữ nguyên giá trị của đầu vào khi lớn hơn 0.



Một số hàm phi tuyến khác như hàm tanh, hàm *sigmoid* cũng có thể được sử dụng thay cho *ReLU*. Tuy vậy, người ta thường dùng *ReLU* bởi vì công thức đơn giản và hội tụ nhanh.

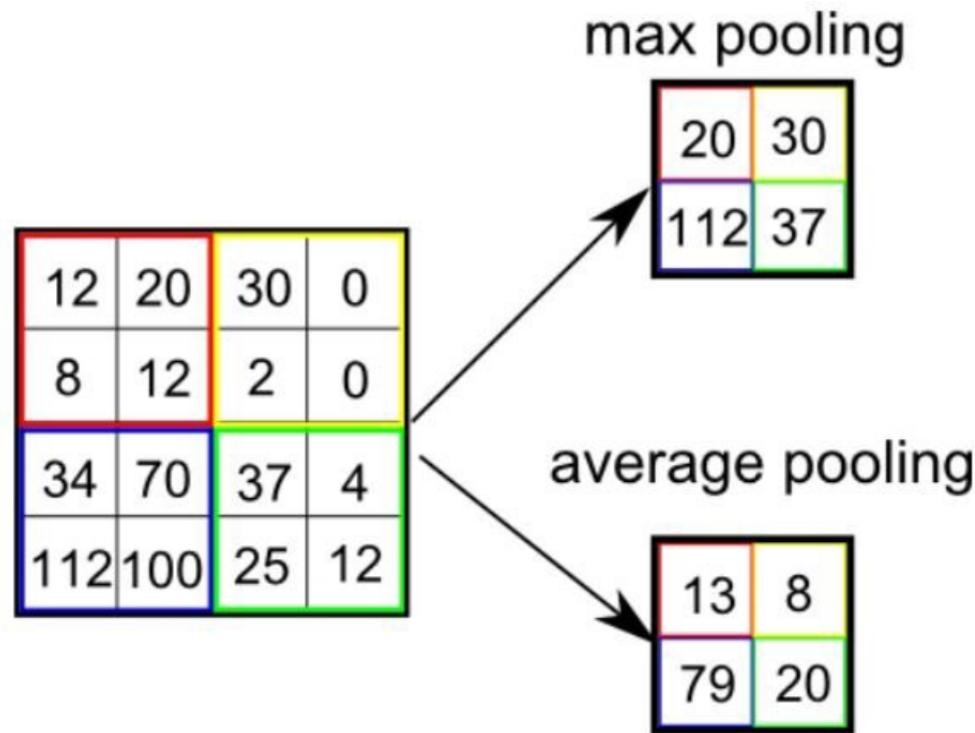
ReLU cũng có một số vấn đề tiềm ẩn như không có đạo hàm tại điểm 0, giá trị của hàm *ReLU* có thể lớn đến vô cùng và nếu chúng ta không khởi tạo trọng số cẩn thận, hoặc khởi tạo learning rate quá lớn thì những neuron ở tầng này sẽ rơi vào trạng thái chết, tức là luôn có giá trị < 0 .

3.2.6 Lớp gộp - Pooling Layer

Lớp gộp sẽ giảm bớt số lượng tham số khi hình ảnh quá lớn. Không gian pooling còn được gọi là lấy mẫu con hoặc lấy mẫu xuống làm giảm kích thước của mỗi map nhưng vẫn giữ lại thông tin quan trọng. Có nhiều loại thủ tục gộp khác nhau:

- Max Pooling: Lấy phần tử lớn nhất từ ma trận đối tượng
- Average Pooling: Lấy trung bình cộng các phần tử

- Sum Pooling: Lấy tổng các phần tử

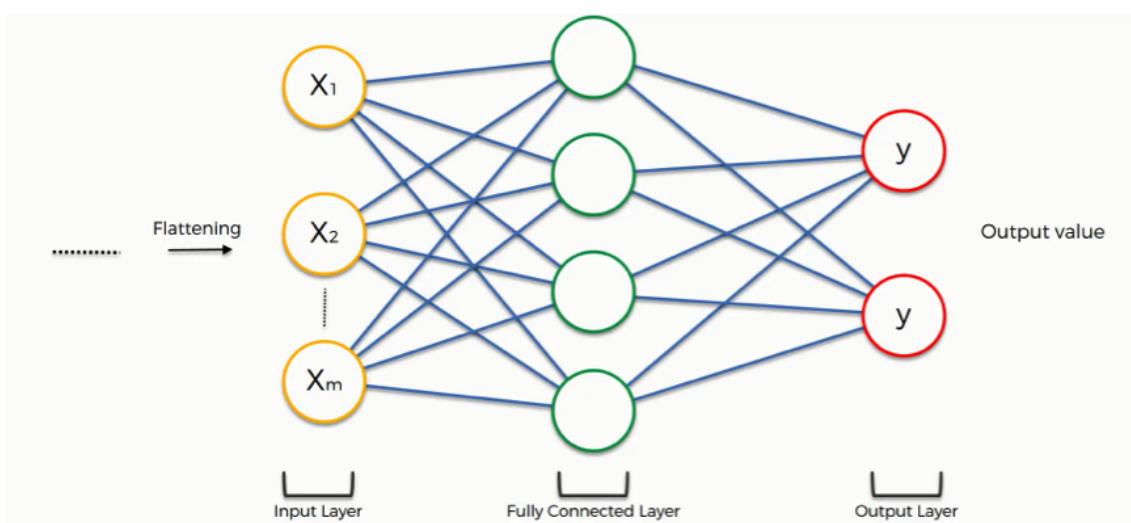


Hiện nay, người ta chủ yếu sử dụng Max Pooling, vì nó bảo toàn hình ảnh tốt hơn các thủ tục khác.

3.2.7 Fully Connected Layer

Sau khi ảnh được truyền qua nhiều convolutional layer và pooling layer thì model đã học được tương đối các đặc điểm của ảnh.

Tầng cuối cùng của mô hình CNNs trong bài toán phân loại ảnh là tầng fully connected layer. Tầng này có chức năng chuyển ma trận đặc trưng ở tầng trước thành vector chứa xác suất của các đối tượng cần được dự đoán. Sau đó ta dùng các fully connected layer để kết hợp các đặc điểm của ảnh để ra được output của mô hình.



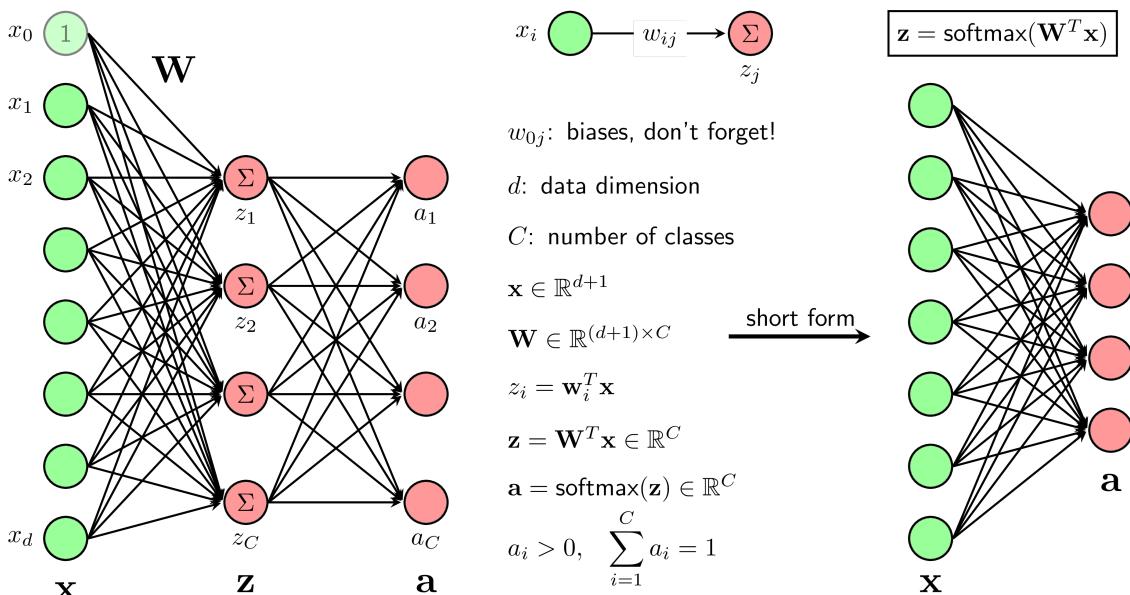
Trong bài toán phân loại số viết tay MNIST có 10 lớp tương ứng 10 số từ 0-1, tầng fully connected layer sẽ chuyển ma trận đặc trưng của tầng trước thành vector có 10 chiều thể hiện xác suất của 10 lớp tương ứng.

3.2.8 Hàm Softmax

Với bài toán nhận diện số viết tay có số class = 10, chúng ta không thể sử dụng **sigmoid** làm hàm activation được. Vì vậy cần phải chọn một hàm cho phép chúng ta dự đoán khả năng phân loại hơn 2 classes - **softmax**.

$$[g(y)]_i = \frac{\exp(y_i)}{\sum_{i=1}^{10} \exp(y_i)}, i = 1, \dots, 10$$

Hàm **softmax** sẽ chỉ ra xác suất của mỗi class. Nếu $[g(y)]_i$ là thành phần lớn nhất của $g(y)$ thì class được dự đoán sẽ ứng với index i trong tập label.



Mà class được dự đoán sẽ chưa chắc đã đúng. Kết quả dự đoán sẽ phụ thuộc trọng số \mathbf{W} mà chúng ta sử dụng để tính xác suất của mỗi class. Vì vậy nếu muốn kết quả dự đoán đúng với thực tế hơn thì chúng ta phải dạy cho model học để tìm ra được \mathbf{W} tối ưu cho bài toán.

Để làm được điều đó, chúng ta phải sử dụng loss function để so sánh độ chênh lệch giữa kết quả dự đoán và kết quả thực tế. Loss function $J(\mathbf{W}, \mathbf{b})$ thực chất là tổng số error của sự dự đoán xảy ra. Nên việc chúng ta cần làm là tìm làm giải sao cho giá trị loss function nhỏ nhất.

Các loss function có thể sử dụng cho bài toán này là:

1. Mean Squared Error

$$J(W, b) = \sum_{j=1}^m \sum_{i=1}^{10} (\{g[f(x_j)] - y_j\}_i)^2$$

2. Cross-Entropy

$$J(W, b) = - \sum_{i=1}^N \sum_{j=1}^{10} y_{ij} \log \left(\frac{\exp(\{g[f(x_j)]\}_i)}{\sum_{j=1}^{10} \exp(\{g[f(x_j)]\}_i)} \right)$$

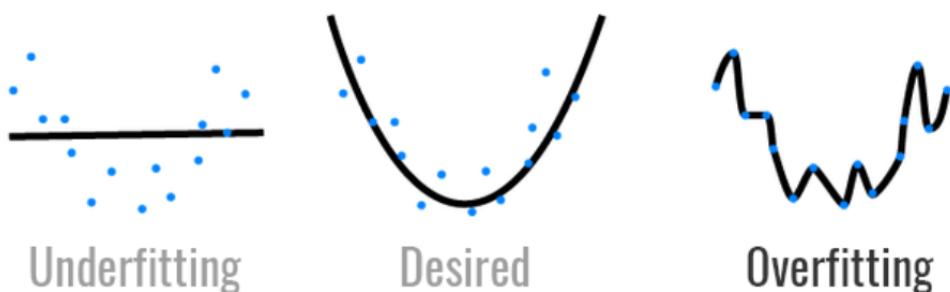
Việc tiếp theo là chúng ta sẽ sử dụng **Gradient Descent** để tìm ra cặp giá trị (\mathbf{W}, \mathbf{b}) để cho

$$\text{argmin} \mathbf{J}(\mathbf{W}, \mathbf{b})$$

3.2.9 Hiện tượng Overfitting

Overfitting là hiện tượng mô hình tìm được quá khớp với dữ liệu training. Việc quá khớp này có thể dẫn đến việc dự đoán nhầm nhiễu, và chất lượng mô hình không còn tốt trên dữ liệu test nữa. Dữ liệu test được giả sử là không được biết trước, và không được sử dụng để xây dựng các mô hình Machine Learning.

Về cơ bản, overfitting xảy ra khi mô hình quá phức tạp để mô phỏng training data. Điều này đặc biệt xảy ra khi lượng dữ liệu training quá nhỏ trong khi độ phức tạp của mô hình quá cao. Hiện tượng Overfitting thường xảy ra trong các mô hình phi tham số hoặc phi tuyến, những mô hình có sự linh hoạt cao trong xây dựng hàm mục tiêu.



Một mô hình được coi là tốt (fit) nếu cả training error và testing error đều thấp. Nếu training error thấp nhưng testing error cao, ta nói mô hình bị overfitting. Nếu train error cao và test error cao, ta nói mô hình bị underfitting. Nếu train error cao nhưng test error thấp, ta đang gặp may mắn.

	Low Training Error	High Training Error
Low Testing Error	The model is learning!	Probably some error in your code. Or you've created a psychic AI.
High Testing Error	OVERFITTING	The model is not learning.

Overfitting thực sự là một vấn đề quan trọng bởi vì việc đánh giá mô hình học máy trên bộ dữ liệu huấn luyện sẽ khác biệt với việc đánh giá độ chính xác của tổng thể (những dữ liệu mà mô hình chưa gặp bao giờ). Hiện nay có nhiều cách để tránh Overfitting, ví dụ như:

- Validation



- Data Augmentation
- Dropout

3.2.10 Validation

Chúng ta thường quen với việc chia tập dữ liệu ra thành hai tập nhỏ: training data và test data. Tuy nhiên, khi xây dựng mô hình, ta không sử dụng test data. Vậy làm cách nào để biết được chất lượng của mô hình với unseen data (tức dữ liệu chưa nhìn thấy bao giờ)

Phương pháp đơn giản nhất là trích từ tập dữ liệu training ra một tập con nhỏ và thực hiện việc đánh giá mô hình trên tập con nhỏ này. Tập con nhỏ được trích ra từ tập training này được gọi là validation set. Lúc này, tập training mới là phần còn lại của tập training ban đầu. Training error được tính trên tập training mới này, và có một khái niệm nữa được định nghĩa tương tự như trên: validation error, tức error được tính trên tập validation.

Việc này giống như khi bạn ôn thi. Giả sử bạn không biết đề thi như thế nào nhưng có 10 bộ đề thi từ các năm trước. Để xem trình độ của mình trước khi thi thế nào, có một cách là bỏ riêng một bộ đề ra, không ôn tập gì. Việc ôn tập sẽ được thực hiện dựa trên 9 bộ còn lại. Sau khi ôn tập xong, bạn bỏ bộ đề đã để riêng ra làm thử và kiểm tra kết quả, như thế mới *khách quan*, mới giống như thi thật. 10 bộ đề ở các năm trước là *toute bộ* training set bạn có. Để tránh việc học lệch, học tủ theo chỉ 10 bộ, bạn tách 9 bộ ra làm training set thật, bộ còn lại là validation test. Khi làm như thế thì mới đánh giá được việc bạn học đã tốt thật hay chưa, hay chỉ là học tủ. Vì vậy, Overfitting còn có thể so sánh với việc Học tủ của con người.

Với khái niệm mới này, ta tìm mô hình sao cho cả train error và validation error đều nhỏ, qua đó có thể dự đoán được rằng test error cũng nhỏ. Phương pháp thường được sử dụng là sử dụng nhiều mô hình khác nhau. Mô hình nào cho validation error nhỏ nhất sẽ là mô hình tốt.

Thông thường, ta bắt đầu từ mô hình đơn giản, sau đó tăng dần độ phức tạp của mô hình. Khi nào validation error có chiều hướng tăng lên thì chọn mô hình ngay trước đó. Chú ý rằng mô hình càng phức tạp, train error có xu hướng càng nhỏ đi.

3.2.11 Tăng cường dữ liệu - Data Augmentation

Dữ liệu càng nhiều, càng đa dạng, hiện tượng overfitting càng khó xảy ra. Lý do là vì càng có nhiều dữ liệu, training error càng giảm.

Tuy nhiên, thu thập thêm ảnh số viết tay là một quá trình tốn kém và vô vị, do vậy người ta cần đến một kỹ thuật gọi là tăng cường dữ liệu (Data Augmentation) để dù có ít dữ liệu, thì bạn vẫn có thể tạo ra được nhiều dữ liệu hơn dựa trên những dữ liệu bạn đã có.

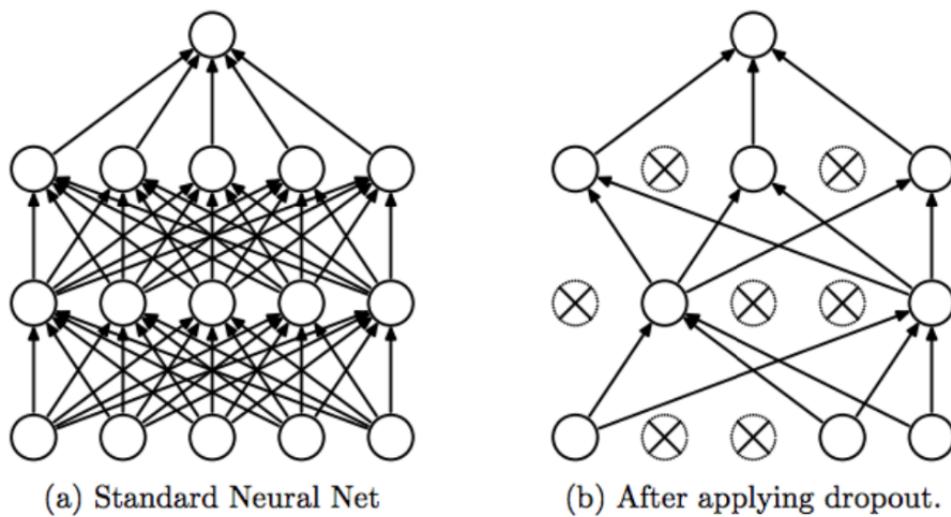
Có nhiều cách tăng cường dữ liệu cho việc nhận dạng số viết tay:

- Các phép biến đổi hình học (Geometry based) – lật, cắt, xoay hoặc dịch hình ảnh một cách ngẫu nhiên
- Chuyển đổi không gian màu (Color based) – thay đổi các kênh màu RGB, tăng hoặc giảm bất kỳ màu nào, tăng độ sắc nét, tăng tính tương phản,..
- Thêm nhiễu (Noise/occlusion) – thêm nhiễu cho ảnh như nhiễu ngẫu nhiên, nhiễu có mẫu, nhiễu do nén ảnh, ...
- Xóa ngẫu nhiên (Random crop) – xóa ngẫu nhiên một phần của hình ảnh ban đầu.

3.2.12 Bỏ học - Dropout

Khi chúng ta sử dụng full connected layer, các neural sẽ phụ thuộc “mạnh” lấn nhau trong suốt quá trình huấn luyện, điều này làm giảm sức mạnh cho mỗi nơ ron và dẫn đến bị overfitting tập training. Vì vậy, chúng ta cần kĩ thuật bỏ học.

Dropout (Bỏ học) là việc bỏ qua các đơn vị (tức là 1 nút mạng) trong quá trình đào tạo 1 cách ngẫu nhiên. Bằng việc bỏ qua này thì đơn vị đó sẽ không được xem xét trong quá trình forward và backward.



Về mặt kỹ thuật, tại mỗi giai đoạn huấn luyện, mỗi node có xác suất bị bỏ qua là $1-p$ và xác suất được chọn là p .

3.3 Thuật toán Gradient Descent

3.3.1 Giới thiệu

Trong Machine Learning nói riêng và Optimization Problem nói chung, chúng ta thường xuyên phải tìm min/max của một hàm số nào đó. Ví dụ như các loss function trong training model. Nhìn chung, việc tìm global minimum của các loss function trong Machine Learning là rất phức tạp, thậm chí là bất khả thi. Thay vào đó, người ta thường cố gắng tìm các điểm local minimum, và ở một mức độ nào đó, coi đó là nghiệm cần tìm của bài toán.

Các điểm local minimum là nghiệm của phương trình đạo hàm bằng 0. Nếu bằng một cách nào đó có thể tìm được toàn bộ (hữu hạn) các điểm cực tiểu, ta chỉ cần thay từng điểm local minimum đó vào hàm số rồi tìm điểm làm cho hàm có giá trị nhỏ nhất. Tuy nhiên, trong hầu hết các trường hợp, việc giải phương trình đạo hàm bằng 0 là bất khả thi. Nguyên nhân có thể đến từ sự phức tạp của dạng của đạo hàm, từ việc các điểm dữ liệu có số chiều lớn, hoặc từ việc có quá nhiều điểm dữ liệu.

Hướng tiếp cận phổ biến nhất là xuất phát từ một điểm mà chúng ta coi là gần với nghiệm của bài toán, sau đó dùng một phép toán lặp để tiến dần đến điểm cần tìm, tức đến khi đạo hàm gần với 0. Gradient Descent (viết gọn là GD) và các biến thể của nó là một trong những phương pháp được dùng nhiều nhất.

3.3.2 Định nghĩa

Giả sử ta cần tìm global minimum cho hàm $f(\theta)$, trong đó θ là một vector chỉ tập hợp các tham số của một mô hình cần tối ưu. Thuật toán GD bắt đầu bằng một điểm dự đoán θ_0 , sau đó, ở vòng lặp thứ t , quy tắc cập nhật là:

$$\theta_{t+1} = \theta_t - \eta \nabla_{\theta} f(\theta_t)$$

Khi cập nhật $\theta = \mathbf{w}$, chúng ta sử dụng tất cả các điểm dữ liệu \mathbf{x}_i . Cách làm này có một vài hạn chế đối với cơ sở dữ liệu có số điểm dữ liệu lớn. Việc phải tính toán lại đạo hàm với tất cả các điểm này sau mỗi vòng lặp trở nên cồng kềnh và không hiệu quả. Nếu làm theo Gradient Descent, tức tính lại đạo hàm của hàm mất mát tại tất cả các điểm dữ liệu, thì thời gian tính toán sẽ rất lâu.

3.3.3 Stochastic Gradient Descent - SGD

Trong thuật toán này, tại 1 thời điểm, ta chỉ tính đạo hàm của hàm mất mát dựa trên chỉ một điểm dữ liệu \mathbf{x}_i rồi cập nhật θ dựa trên đạo hàm này. Việc này được thực hiện với từng điểm trên toàn bộ dữ liệu, sau đó lặp lại quá trình trên. Thuật toán rất đơn giản này trên thực tế lại làm việc rất hiệu quả.

Mỗi lần duyệt một lượt qua tất cả các điểm trên toàn bộ dữ liệu được gọi là một epoch. Với GD thông thường thì mỗi epoch ứng với 1 lần cập nhật θ , với SGD thì mỗi epoch ứng với N lần cập nhật θ tương ứng N điểm dữ liệu trong training set. Nhìn vào một mặt, việc cập nhật từng điểm một như thế này có thể làm giảm đi tốc độ thực hiện 1 epoch. Nhưng nhìn vào một mặt khác, SGD chỉ yêu cầu một lượng epoch rất nhỏ (thường là 10 cho lần đầu tiên, sau đó khi có dữ liệu mới thì chỉ cần chạy dưới một epoch là đã có nghiệm tốt).

$$\theta = \theta - \eta \nabla_{\theta} J(\theta; \mathbf{x}_i; \mathbf{y}_i)$$

Dặc điểm:

- Sau mỗi epoch, chúng ta cần shuffle (xáo trộn) thứ tự của các dữ liệu để đảm bảo tính ngẫu nhiên. Việc này cũng ảnh hưởng tới hiệu năng của SGD.
- SGD phù hợp với các bài toán có lượng cơ sở dữ liệu lớn (chủ yếu là Deep Learning) và các bài toán yêu cầu mô hình thay đổi liên tục.

3.3.4 Batch Gradient Descent

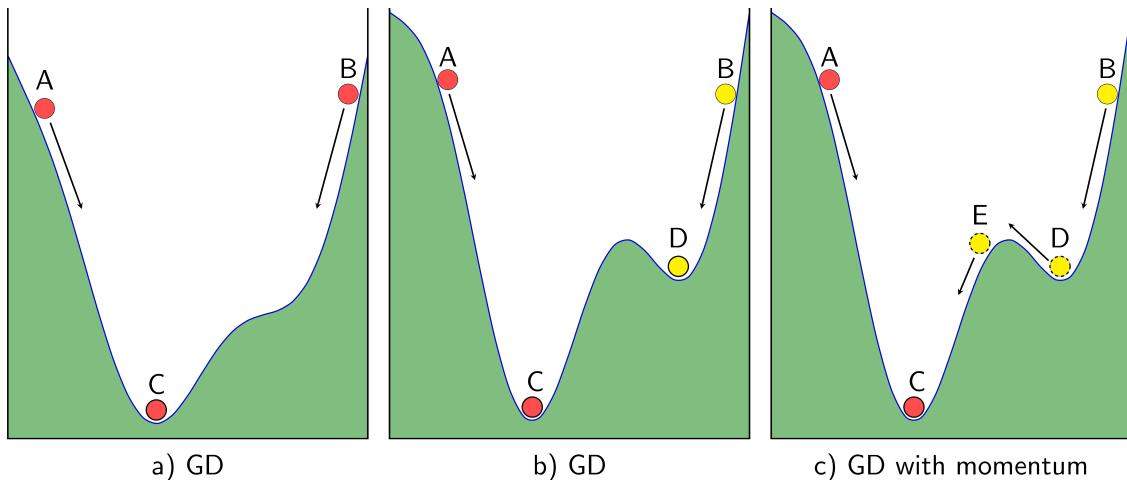
Khác với SGD, Batch Gradient Descent sử dụng một số lượng $n > 1$ (nhưng vẫn nhỏ hơn tổng số dữ liệu N rất nhiều). Giống với SGD, Batch Gradient Descent bắt đầu mỗi epoch bằng việc xáo trộn ngẫu nhiên dữ liệu rồi chia toàn bộ dữ liệu thành các batch, mỗi batch có n điểm dữ liệu (trừ batch cuối có thể có ít hơn nếu N không chia hết cho n). Mỗi lần cập nhật, thuật toán này lấy ra một batch để tính toán đạo hàm rồi cập nhật. Công thức có thể viết dưới dạng:

$$\theta = \theta - \eta \nabla_{\theta} J(\theta; \mathbf{x}_{i:i+n}; \mathbf{y}_{i:i+n})$$

Dặc điểm:

- Dữ liệu này sau mỗi epoch là khác nhau vì chúng cần được xáo trộn.
- Batch GD được sử dụng trong hầu hết các thuật toán Machine Learning, đặc biệt là trong Deep Learning. Giá trị n thường được chọn là khoảng từ 50 đến 100.

3.3.5 Tối ưu Gradient Descent với Momentum



Trong GD, chúng ta cần tính lượng thay đổi ở thời điểm t để cập nhật vị trí mới cho nghiệm θ . Nếu chúng ta coi đại lượng này như vận tốc v_t trong vật lý, vị trí mới của hòn bi sẽ là $\theta_{t-1} = \theta_t - v_t$. Dẫu trừ thể hiện việc phải di chuyển ngược với đạo hàm. Công việc của chúng ta bây giờ là tính đại lượng v_t sao cho nó vừa mang thông tin của độ dốc (tức đạo hàm), vừa mang thông tin của đà, tức vận tốc trước đó v_{t-1} (chúng ta coi như vận tốc ban đầu $v_0 = 0$). Một cách đơn giản nhất, ta có thể cộng hai đại lượng này lại:

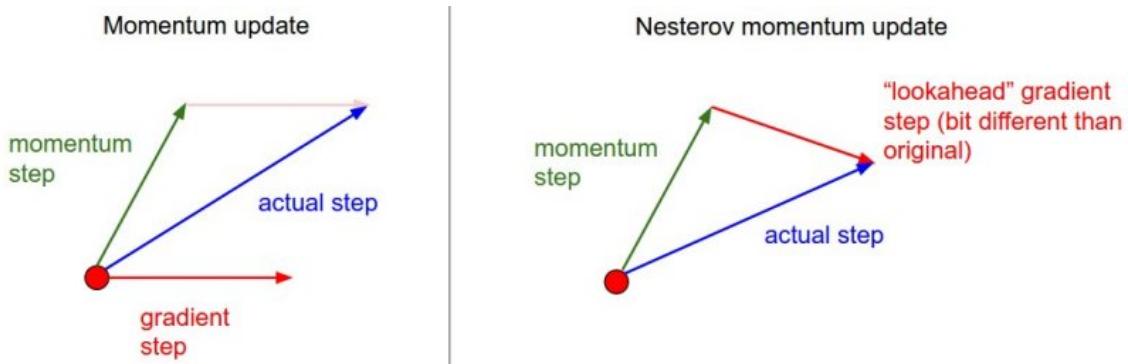
$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta)$$

Trong đó γ thường được chọn là một giá trị khoảng 0.9.

Thuật toán đơn giản này tỏ ra rất hiệu quả trong các bài toán thực tế.

3.3.6 Tối ưu Gradient Descent với Nesterov

Momentum giúp hòn bi vượt qua được dốc *locaminimum*, tuy nhiên, có một hạn chế chúng ta có thể thấy trong ví dụ trên: Khi tới gần đích, momentum vẫn mất khá nhiều thời gian trước khi dừng lại. Lý do lại cũng chính là vì có đà. Có một phương pháp khác tiếp tục giúp khắc phục điều này, phương pháp đó mang tên Nesterov accelerated gradient (NAG), giúp cho thuật toán hội tụ nhanh hơn.



Ý tưởng cơ bản là dự đoán hướng đi trong tương lai, tức nhìn trước một bước. Cụ thể, nếu sử dụng số hạng momentum $t-1$ để cập nhật thì ta có thể xấp xỉ được vị trí tiếp theo của hòn bi là θ_{t-1} . Vậy, thay vì sử dụng gradient của điểm hiện tại, NAG đi trước một bước, sử dụng gradient của điểm tiếp theo.

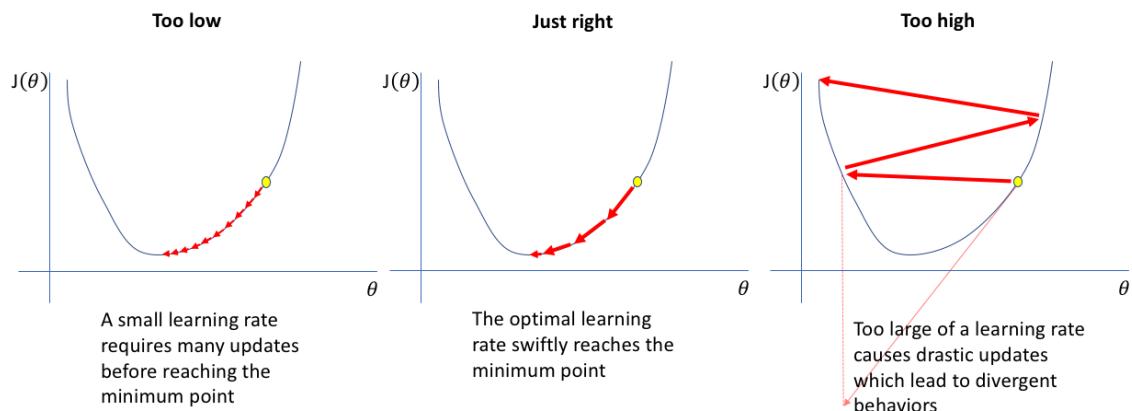
Công thức cập nhật nghiệm:

$$v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma v_{t-1}) \quad \theta = \theta - v_t$$

3.3.7 Điều chỉnh learning rate cho kết quả tốt hơn

Learning rate đóng một vai trò quan trọng khi train neural network với gradient descent. Tham số này sẽ scale kích thước của trọng số (\mathbf{W}, \mathbf{b}) khi update trạng thái để minimize hàm loss cho network.

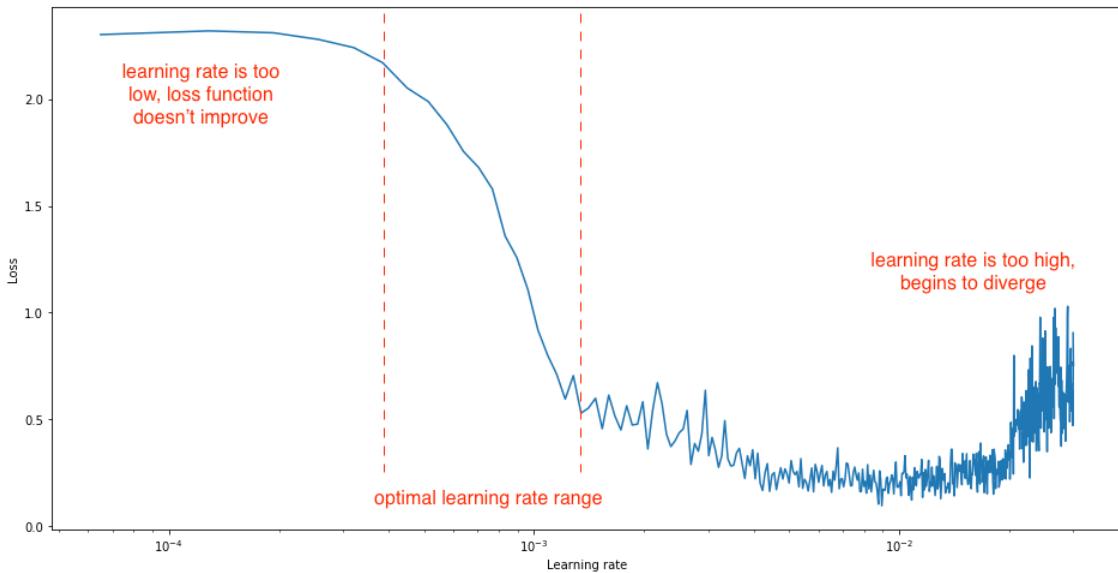
Nếu learning rate quá nhỏ sẽ dẫn đến quá trình train bị chậm bởi vì trọng số (\mathbf{W}, \mathbf{b}) scale một lượng rất nhỏ mỗi lần update. Còn nếu learning rate quá lớn, nó sẽ dẫn đến sự khác nhau không mong muốn trong các lần update.



Vì vậy, nếu chúng ta chọn được **learning rate** tối ưu phù hợp thì sẽ giúp neural network hội tụ đến kết globalminimal nhanh hơn, đồng thời hàm loss sẽ được minimize để cho kết quả bài toán tốt hơn - tức là độ chính xác cao hơn và loss thấp hơn.

1. Hướng tiếp cận để tìm learning rate tối ưu

Với learning rate nhỏ, thì loss sẽ giảm nhưng chỉ ở mức nông cạn. Khi tiến gần đến vùng tối ưu learning rate thì loss sẽ thay đổi rất nhanh. Còn tăng learning rate sẽ dần đến loss tăng theo bởi vì tham số được cập nhật có thể thuộc phía còn lại so với hướng đang đi.



Do đó, chúng ta nên schedule **dynamic** learning rate, tức là learning sẽ thay đổi liên tục trong quá trình training, nhưng chỉ thay đổi trong một phạm vi có thể chấp nhận được

2. Schedule learning rate với Triangular schedule

Triangular schedule là thuật toán sẽ thay đổi learning rate η giữa 2 giá trị bound η_{min} và η_{max} cho trước. Sự thay đổi này sẽ tuân thủ theo quy luật cập nhật có hình tam giác.

Công thức thay đổi learning rate η có dạng như sau:

$$\eta_t = \eta_{min} + (\eta_{max} - \eta_{min}) \cdot \max(0, 1 - x)$$

Với x là:

$$x = \left| \frac{\text{iterations}}{\text{stepsize}} - 2(\text{cycle}) + 1 \right|$$

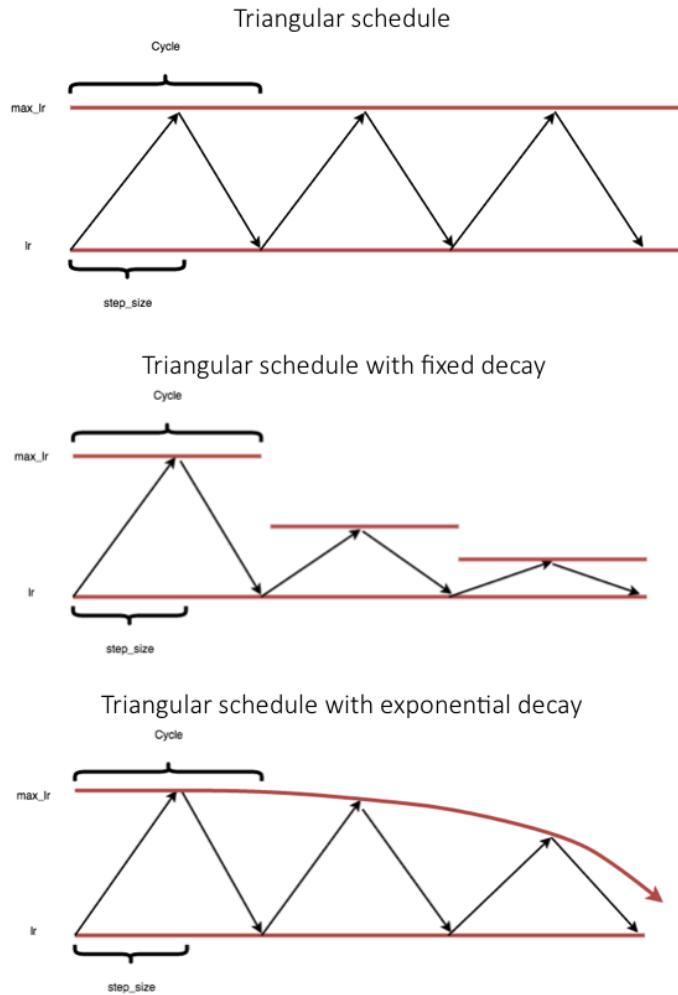
và $cycle$ được tính bởi công thức:

$$\text{cycle} = \text{floor}(1 + \frac{\text{iterations}}{2\text{stepsize}})$$

Đặc điểm:

- Chúng ta có thể đi nhanh được với những step đầu tiên để nhanh đến vùng tối ưu kết quả. Điều này sẽ giúp ta giảm thời gian training một cách đáng kể - một điều khá quan trọng trong các neural network phức tạp.
- Do learning rate thay đổi liên tục nên khi tới vùng tối ưu thì hàm loss sẽ có thể duyệt liên tục qua các vùng xung quanh đó bằng cách scale với learning rate khác nhau. Điều này sẽ giúp kết quả của chúng ta thường sẽ tối ưu hơn so với learning rate cố định.

Có ba dạng triangular schedule khác nhau:



4 Thiết kế và hiện thực

4.1 Tập dữ liệu dataset

Dữ liệu được lấy từ tập dataset MNIST. Tập dữ liệu bao gồm 70000 ảnh chữ số viết tay kích thước 28x28 pixel đã được đánh nhãn, trong đó các chữ cái viết tay nằm trong ô vuông chính giữa có kích thước 29x28 pixel. Mỗi pixel có một giá trị pixel duy nhất được liên kết với nó, cho biết độ sáng hoặc tối của pixel đó. Giá trị pixel là một số nguyên từ 0 đến 255.

4.2 Môi trường lập trình, công cụ hỗ trợ

- Môi trường lập trình: Google Codelabs
- Ngôn ngữ sử dụng: Python
- Thư viện hỗ trợ: tensorflow

4.3 Các bước tiến hành

4.3.1 Import các thư viện cần thiết

Để training model, chúng ta phải import các thư viện hỗ trợ:

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import tensorflow as tf
4 from tensorflow.keras.utils import to_categorical
```

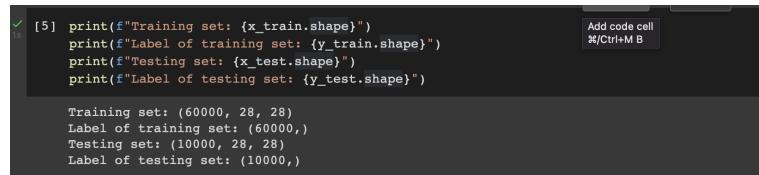
- **tensorflow**: Thư viện Tensorflow cho Deep Learning
- **numpy**: Cho các phép toán ma trận ngoài Tensorflow
- **matplotlib**: Vẽ các biểu đồ và hình ảnh cho tập training và validation data
- **to_categorical**: Để format label dùng cho training

4.3.2 Load tập dữ liệu MNIST

Ta có thể load MNIST dataset dễ dàng qua thư viện *tensorflow*:

```
1 mnist = tf.keras.datasets.mnist
2 (x_train, y_train), (x_test, y_test) = mnist.load_data()
```

1. Data Exploration



```
[5] print(f"Training set: {x_train.shape}")
      print(f"Label of training set: {y_train.shape}")
      print(f"Testing set: {x_test.shape}")
      print(f"Label of testing set: {y_test.shape}")

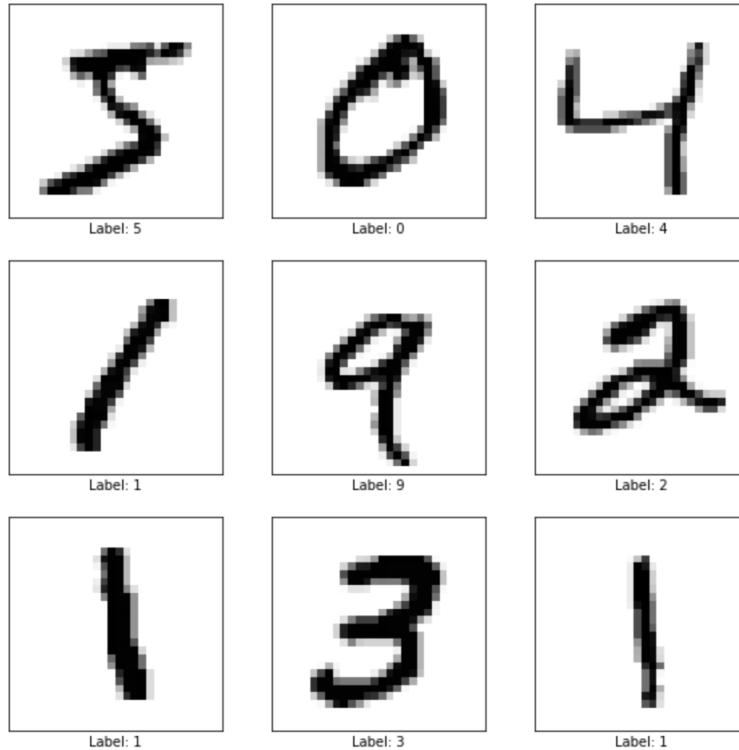
Training set: (60000, 28, 28)
Label of training set: (60000,)
Testing set: (10000, 28, 28)
Label of testing set: (10000,)
```

Tập dataset có 60000 samples dùng cho training, 10000 samples là dùng cho testing. Mỗi image sẽ có kích thước 28x28, được chứa tương ứng trong ma trận.

2. Data Visualization

Vẽ minh họa 9 ảnh đầu tiên trong tập dữ liệu training

```
1 plt.figure(figsize=(10,10))
2 for i in range(9):
3     plt.subplot(3,3,i+1)
4     plt.xticks([])
5     plt.yticks([])
6     plt.grid(False)
7     plt.imshow(x_train[i], cmap=plt.cm.binary)
8     plt.xlabel(f"Label: {y_train[i]}")
9 plt.show()
```



4.3.3 Chuẩn bị dữ liệu - Tiền xử lý

Thêm vào 1 chiều cho dữ liệu để thể hiện color channel gray:

```
[6] x_train = tf.keras.utils.normalize(x_train, axis=1)
    x_test = tf.keras.utils.normalize(x_test, axis=1)

    #expand 1 more dimension as 1 for colour channel gray
    x_train = x_train.reshape(x_train.shape[0], 28, 28,1)
    x_test = x_test.reshape(x_test.shape[0], 28, 28,1)
    print(f'{x_train.shape}')
    print(f'{x_test.shape}')

(60000, 28, 28, 1)
(10000, 28, 28, 1)
```

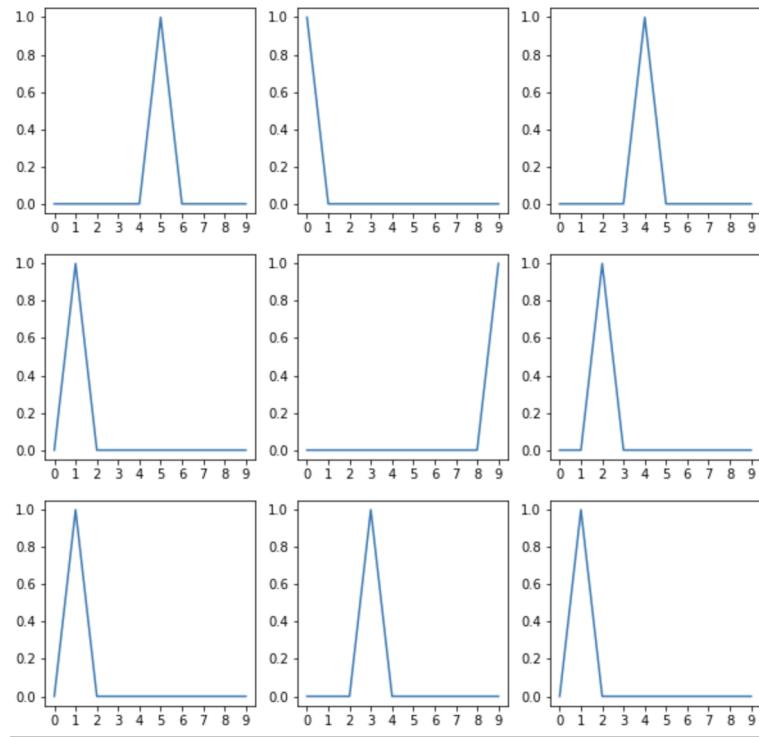
Tiêu chuẩn hoá feature là một bước quan trọng trong tiền xử lý. Nó được sử dụng để tập trung dữ liệu xung quanh trung vị zero và phương sai:

```
1 mean_px = x_train.mean().astype(np.float32)
2 std_px = x_train.std().astype(np.float32)
3
4 def standardize(x):
5     return (x-mean_px)/std_px
```

Tiếp theo, chúng ta cần format lại tập label theo dạng **a one-hot vector**:

```
1 y_train= to_categorical(y_train)
2 y_test= to_categorical(y_test)
```

Chúng ta cùng nhìn vào tập label sau khi format lại với 9 label đầu tiên



Cuối cùng, ta tạo hàm vẽ đồ thị để trực quan cho quá trình training của mỗi model. Đường màu xanh đại diện cho training set, còn màu cam đại diện cho validation set.

```
1 def plot_model_history(model_name, history):
2     plt.figure(figsize=(10,10))
3     # Plot accuracy of training and validation
4     plt.subplot(2,2,1)
5     plt.xticks(range(10))
6     plt.title(f'Accuracy of {model_name}')
7     plt.ylabel('Accuracy')
8     plt.xlabel('Epoch')
9     plt.grid(False)
10    plt.plot(history.history['accuracy'])
11    plt.plot(history.history['val_accuracy'])
12    plt.legend(['train', 'validation'], loc='lower right')
13    # Plot loss of training and validation
14    plt.subplot(2,2,2)
15    plt.xticks(range(10))
16    plt.title(f'Loss of {model_name}')
17    plt.ylabel('Loss')
18    plt.xlabel('Epoch')
19    plt.grid(False)
20    plt.plot(history.history['loss'])
21    plt.plot(history.history['val_loss'])
22    plt.legend(['train', 'validation'], loc='upper right')
23    plt.show()
```

4.3.4 Thiết kế model Neural Network

Chúng ta sẽ sử dụng các layer sau để xây dựng model trong suốt quá trình:

- Layer Conv2D: Conv2D filters có kernel (5x5) và tạo ra 64 convoluted images cùng kích thước với ảnh ban đầu. Nếu sử dụng padding = 'Same' thì layer giữ nguyên kích thước ban đầu của ảnh bằng zero padding.
- Layer MaxPooling2D: 64 output được thu nhỏ bằng max pooling.
- activation = 'relu': Đây là hàm kích hoạt phi tuyến, được sử dụng cho các Convolutional layer

- Layer Flatten: Đây là layer để "làm phẳng" hình ảnh trước khi đưa vào Dense layer. Layer này chuyển ảnh từ mảng 3 chiều thành mảng 1 chiều kích thước 784 pixels ($= 28 \times 28$). Layer này không có tham số để học mà chỉ định dạng lại dữ liệu.
- Layer Dense đầu tiên: Đây là fully connected layer có 128 neurons, lấy input từ layer trước của nó. Mỗi neuron lấy input từ tất cả output layer trước, điều chỉnh lại trọng số cần học và output ra 1 giá trị vào layer kế tiếp
- Layer output Dense: Đây là fully connected layer với 10 node, mỗi node tương ứng với 1 label. Layer này có input từ 128 nodes của layer ngay phía trước và output một giá trị trong khoảng [0..1]. Tổng của 10 nodes này bằng 1

4.3.5 Question 1

1. Hiện thực model full connected layers ở Chapter 1

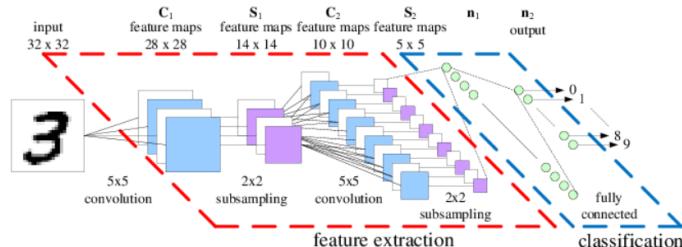
```

1 model = tf.keras.models.Sequential()
2 model.add(tf.keras.layers.Lambda(standardize, input_shape=(28,28,1)))
3
4 # Fully connected layers
5 model.add(tf.keras.layers.Flatten())
6 model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
7 model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
8
9 sgd = tf.keras.optimizers.SGD(learning_rate=0.01)
10 model.compile(
11     optimizer=sgd,
12     loss='mse',
13     metrics=['accuracy'])
14

```

2. Hiện thực model CNNs

Ta sẽ hiện thực model theo dạng sau:



```

1 model = tf.keras.models.Sequential()
2 model.add(tf.keras.layers.Lambda(standardize, input_shape=(28,28,1)))
3 # Convolutional layers
4 model.add(tf.keras.layers.Conv2D(64, (5,5), activation = 'relu', input_shape=(28,28,1), padding = 'Same'))
5 model.add(tf.keras.layers.MaxPooling2D(2,2))
6 model.add(tf.keras.layers.Conv2D(64, (5,5), activation = 'relu', padding = 'Same'))
7 model.add(tf.keras.layers.MaxPooling2D(pool_size = (2,2)))
8
9 # Fully connected layers
10 model.add(tf.keras.layers.Flatten())
11 model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
12 model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
13
14 sgd = tf.keras.optimizers.SGD(learning_rate=0.01)
15 model.compile(
16     optimizer=sgd,
17     loss='mse',
18     metrics=['accuracy'])
19

```

3. Đánh giá model

```

history_fc_mse = model.fit(x_train, y_train, batch_size=64, validation_split = 0.1, epochs=5)
loss, accuracy = model.evaluate(x_test, y_test)
print("The evaluation for test set: accuracy: (accuracy:.4f) - loss: (loss:.4f)")

Epoch 1/5
844/844 [=====] - 4s 4ms/step - loss: 0.0815 - accuracy: 0.3211 - val_loss: 0.0610 - val_accuracy: 0.5485
Epoch 2/5
844/844 [=====] - 3s 4ms/step - loss: 0.0503 - accuracy: 0.6534 - val_loss: 0.0361 - val_accuracy: 0.7807
Epoch 3/5
844/844 [=====] - 3s 4ms/step - loss: 0.0350 - accuracy: 0.7744 - val_loss: 0.0270 - val_accuracy: 0.8348
Epoch 4/5
844/844 [=====] - 3s 4ms/step - loss: 0.0286 - accuracy: 0.8170 - val_loss: 0.0225 - val_accuracy: 0.8598
Epoch 5/5
844/844 [=====] - 3s 4ms/step - loss: 0.0248 - accuracy: 0.8416 - val_loss: 0.0197 - val_accuracy: 0.8802
313/313 [=====] - 1s 2ms/step - loss: 0.0223 - accuracy: 0.8566
The evaluation for test set: accuracy: 0.8566 - loss: 0.0223

```

Hình 4: Quá trình training và evaluate testing data của model Full Connected Layers

```

history_cnn_mse = model.fit(x_train, y_train, batch_size=64, validation_split = 0.1, epochs=5)
loss, accuracy = model.evaluate(x_test, y_test)
print("The evaluation for test set: accuracy: (accuracy:.4f) - loss: (loss:.4f)")

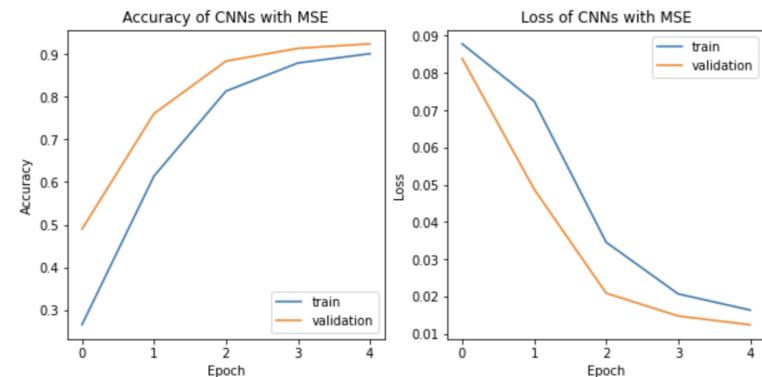
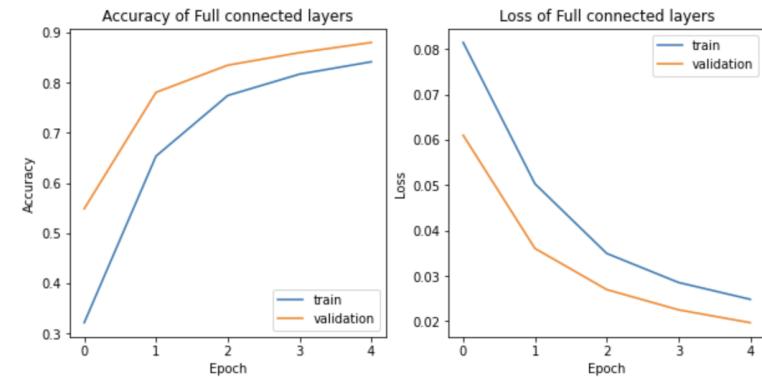
Epoch 1/5
844/844 [=====] - 232s 275ms/step - loss: 0.0878 - accuracy: 0.2654 - val_loss: 0.0839 - val_accuracy: 0.4895
Epoch 2/5
844/844 [=====] - 224s 255ms/step - loss: 0.0724 - accuracy: 0.6139 - val_loss: 0.0488 - val_accuracy: 0.7603
Epoch 3/5
844/844 [=====] - 225s 267ms/step - loss: 0.0345 - accuracy: 0.8131 - val_loss: 0.0209 - val_accuracy: 0.8835
Epoch 4/5
844/844 [=====] - 230s 273ms/step - loss: 0.0207 - accuracy: 0.8791 - val_loss: 0.0148 - val_accuracy: 0.9135
Epoch 5/5
844/844 [=====] - 227s 269ms/step - loss: 0.0163 - accuracy: 0.9009 - val_loss: 0.0124 - val_accuracy: 0.9240
313/313 [=====] - 10s 32ms/step - loss: 0.0140 - accuracy: 0.9135
The evaluation for test set: accuracy: 0.9135 - loss: 0.0140

```

Hình 5: Quá trình training và evaluate testing data của model CNNs

Từ kết quả trên, chúng ta dễ dàng nhận thấy được model CNNs sẽ cho kết quả tốt hơn nhiều. Bởi vì model Full Connected Layers sẽ bị miss một số feature quan trọng trong quá trình training, dẫn đến nó không nhận dạng tốt được trong trường hợp số được viết dễ gây nhầm lẫn với label khác.

Điều này sẽ được cải thiện đáng kể khi chúng ta thêm một số layer Conv và Pooling trước layer Full Connected. Độ chính xác của model CNNs sẽ đạt 91,35% sau khi train 5 epochs, cao hơn so với model Full Connected Layers 85,66% khi kiểm tra với 10000 sample trong tập test.



4.3.6 Question 2

Hàm MSE không phù hợp khi tính hàm loss có chứa xác suất dẫn đến độ chính xác của model thấp. Vì vậy chúng ta sẽ thay thế MSE bằng hàm Cross-Entropy - được sử dụng phổ biến trong các bài toán

tính loss function liên quan đến xác suất.

```

1 model = tf.keras.models.Sequential()
2 model.add(tf.keras.layers.Lambda(standardize, input_shape=(28,28,1)))
3 # Convolutional layers
4 model.add(tf.keras.layers.Conv2D(64, (5,5), activation = 'relu', input_shape=(28,28,1), padding = 'Same'))
5 model.add(tf.keras.layers.MaxPooling2D(2,2))
6 model.add(tf.keras.layers.Conv2D(64, (5,5), activation = 'relu', padding = 'Same'))
7 model.add(tf.keras.layers.MaxPooling2D(pool_size = (2,2)))
8
9 # Fully connected layers
10 model.add(tf.keras.layers.Flatten())
11 model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
12 model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
13
14 sgd = tf.keras.optimizers.SGD(learning_rate=0.01)
15 model.compile(
16     optimizer=sgd,
17     loss='categorical_crossentropy',
18     metrics=['accuracy']
19 )

```

The screenshot shows the command history for training a CNN model with MSE loss. It includes the training loop, validation metrics, and the final evaluation results.

```

history_cnn_mse = model.fit(x_train, y_train, batch_size=64, validation_split = 0.1, epochs=5)

loss, accuracy = model.evaluate(x_test, y_test)
print(f'The evaluation for test set: accuracy: {accuracy:.4f} - loss: {loss:.4f}')

Epoch 1/5
844/844 [=====] - 148s 175ms/step - loss: 0.3592 - accuracy: 0.8985 - val_loss: 0.1117 - val_accuracy: 0.9677
Epoch 2/5
844/844 [=====] - 149s 176ms/step - loss: 0.1171 - accuracy: 0.9655 - val_loss: 0.0789 - val_accuracy: 0.9768
Epoch 3/5
844/844 [=====] - 147s 175ms/step - loss: 0.0856 - accuracy: 0.9745 - val_loss: 0.0690 - val_accuracy: 0.9782
Epoch 4/5
844/844 [=====] - 149s 177ms/step - loss: 0.0696 - accuracy: 0.9793 - val_loss: 0.0685 - val_accuracy: 0.9823
Epoch 5/5
844/844 [=====] - 148s 175ms/step - loss: 0.0578 - accuracy: 0.9829 - val_loss: 0.0673 - val_accuracy: 0.9782
313/313 [=====] - 7s 21ms/step - loss: 0.0672 - accuracy: 0.9775
The evaluation for test set: accuracy: 0.9775 - loss: 0.0672

```

Hình 6: Quá trình training và evaluate testing data của model CNNs với MSE

The screenshot shows the command history for training a CNN model with Cross-Entropy loss. It includes the training loop, validation metrics, and the final evaluation results.

```

history_cnn_ce = model.fit(x_train, y_train, batch_size=64, validation_split = 0.1, epochs=5)

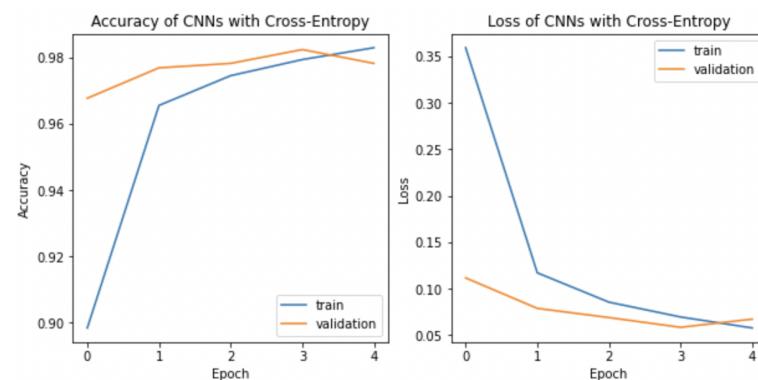
loss, accuracy = model.evaluate(x_test, y_test)
print(f'The evaluation for test set: accuracy: {accuracy:.4f} - loss: {loss:.4f}')

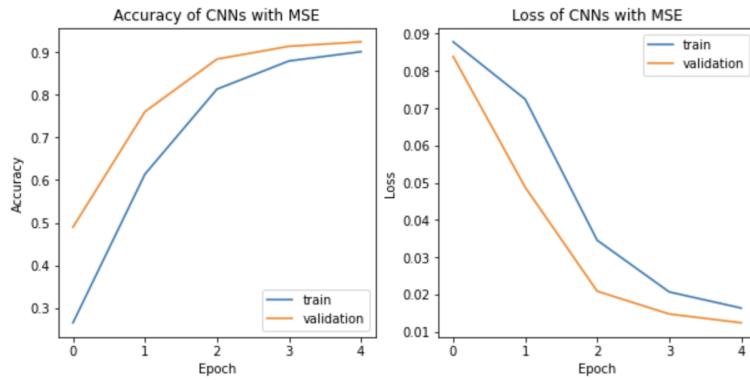
Epoch 1/5
844/844 [=====] - 232s 275ms/step - loss: 0.0878 - accuracy: 0.2654 - val_loss: 0.0839 - val_accuracy: 0.4895
Epoch 2/5
844/844 [=====] - 224s 265ms/step - loss: 0.0724 - accuracy: 0.6139 - val_loss: 0.0488 - val_accuracy: 0.7603
Epoch 3/5
844/844 [=====] - 225s 267ms/step - loss: 0.0345 - accuracy: 0.8131 - val_loss: 0.0209 - val_accuracy: 0.8835
Epoch 4/5
844/844 [=====] - 230s 273ms/step - loss: 0.0207 - accuracy: 0.8791 - val_loss: 0.0148 - val_accuracy: 0.9135
Epoch 5/5
844/844 [=====] - 227s 249ms/step - loss: 0.0163 - accuracy: 0.9009 - val_loss: 0.0124 - val_accuracy: 0.9240
313/313 [=====] - 10s 32ms/step - loss: 0.0140 - accuracy: 0.9135
The evaluation for test set: accuracy: 0.9135 - loss: 0.0140

```

Hình 7: Quá trình training và evaluate testing data của model CNNs với Cross-Entropy

Ta nhận thấy rõ rệt sự chênh lệch đồ chính xác khi sử dụng 2 hàm loss function trên. Với model sử dụng Cross-Entropy, độ chính xác của model ngay từ epoch đầu tiên đã gần 90%, sắp đạt tới kết quả train 5 epochs với MSE. Cuối cùng model này đạt được độ chính tới 97,75% khi kiểm tra trên tập test.





4.3.7 Question 3

Ở câu này, ta sẽ sử dụng model CNNs đã sử dụng:

```

1 model = tf.keras.models.Sequential()
2 model.add(tf.keras.layers.Lambda(standardize, input_shape=(28,28,1)))
3 # Convolutional layers
4 model.add(tf.keras.layers.Conv2D(64, (5,5), activation = 'relu', input_shape=(28,28,1), padding = 'Same'))
5 model.add(tf.keras.layers.MaxPooling2D(2,2))
6 model.add(tf.keras.layers.Conv2D(64, (5,5), activation = 'relu', padding = 'Same'))
7 model.add(tf.keras.layers.MaxPooling2D(pool_size = (2,2)))
8
9 # Fully connected layers
10 model.add(tf.keras.layers.Flatten())
11 model.add(tf.keras.layers.Dense(units=128, activation=tf.nn.relu))
12 model.add(tf.keras.layers.Dense(units=10, activation=tf.nn.softmax))
13
14 sgd = tf.keras.optimizers.SGD(learning_rate=0.01)
15 model.compile(
16     optimizer=sgd,
17     loss='categorical_crossentropy',
18     metrics=['accuracy'])
19

```

1. Sử dụng Stochastic Gradient Descent

```

history_sgd = model.fit(x_train, y_train, batch_size=1, validation_split = 0.1, epochs=5)
loss, accuracy = model.evaluate(x_test, y_test)
print("The evaluation for test set: accuracy: {accuracy:.4f} - loss: {loss:.4f}")

Epoch 1/5
54000/54000 [=====] - 480s 9ms/step - loss: 0.1345 - accuracy: 0.9599 - val_loss: 0.0560 - val_accuracy: 0.9840
Epoch 2/5
54000/54000 [=====] - 412s 8ms/step - loss: 0.0628 - accuracy: 0.9817 - val_loss: 0.0612 - val_accuracy: 0.9838
Epoch 3/5
54000/54000 [=====] - 436s 8ms/step - loss: 0.0520 - accuracy: 0.9851 - val_loss: 0.0734 - val_accuracy: 0.9793
Epoch 4/5
54000/54000 [=====] - 449s 8ms/step - loss: 0.0420 - accuracy: 0.9879 - val_loss: 0.0604 - val_accuracy: 0.9870
Epoch 5/5
54000/54000 [=====] - 452s 8ms/step - loss: 0.0462 - accuracy: 0.9871 - val_loss: 0.0931 - val_accuracy: 0.9825
313/313 [=====] - 7s 2ms/step - loss: 0.0707 - accuracy: 0.9850
The evaluation for test set: accuracy: 0.9850 - loss: 0.0707

```

Hình 8: Quá trình training và evaluate testing data sử dụng SGD

2. Sử dụng Mini-Batch Gradient Descent

```

history_minibgd = model.fit(x_train, y_train, batch_size=64, validation_split = 0.1, epochs=5)
loss, accuracy = model.evaluate(x_test, y_test)
print("The evaluation for test set: accuracy: {accuracy:.4f} - loss: {loss:.4f}")

Epoch 1/5
844/844 [=====] - 147s 173ms/step - loss: 0.3566 - accuracy: 0.8989 - val_loss: 0.1147 - val_accuracy: 0.9705
Epoch 2/5
844/844 [=====] - 148s 169ms/step - loss: 0.1123 - accuracy: 0.9657 - val_loss: 0.0804 - val_accuracy: 0.9775
Epoch 3/5
844/844 [=====] - 144s 171ms/step - loss: 0.0823 - accuracy: 0.9748 - val_loss: 0.0684 - val_accuracy: 0.9803
Epoch 4/5
844/844 [=====] - 144s 170ms/step - loss: 0.0670 - accuracy: 0.9793 - val_loss: 0.0613 - val_accuracy: 0.9818
Epoch 5/5
844/844 [=====] - 145s 172ms/step - loss: 0.0572 - accuracy: 0.9830 - val_loss: 0.0629 - val_accuracy: 0.9812
313/313 [=====] - 68 21ms/step - loss: 0.0564 - accuracy: 0.9811
The evaluation for test set: accuracy: 0.9811 - loss: 0.0564

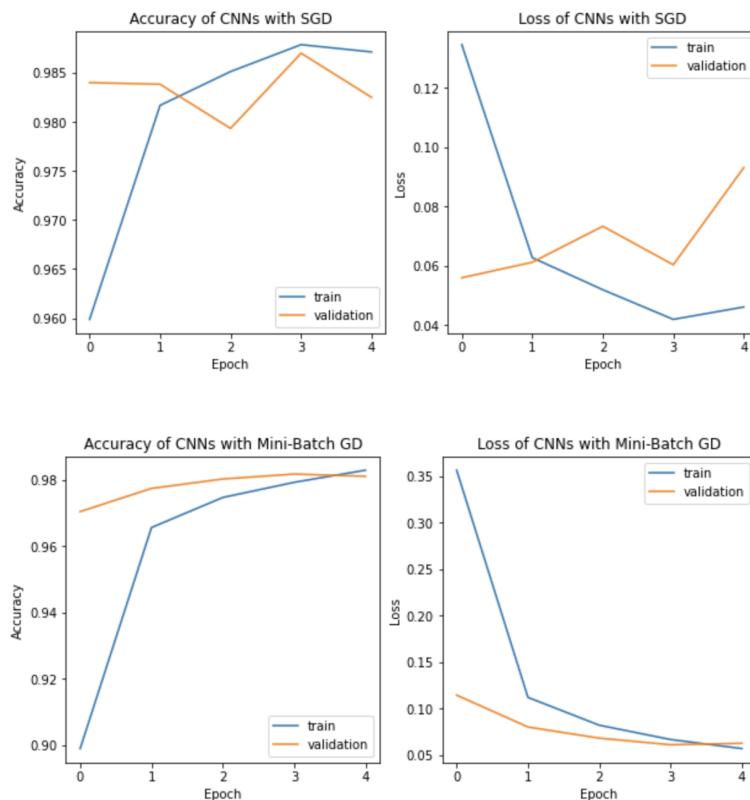
```

Hình 9: Quá trình training và evaluate testing data sử dụng Mini-Batch GD

3. Đánh giá model

Vì SGD là sẽ update trọng số với từng điểm dữ liệu trong epoch nên ngay từ đầu nó cho kết quả độ chính xác rất cao, hàm loss cũng rất nhỏ so với Mini-Batch. Nhưng có một điểm hạn chế là sẽ mất thời gian khá lâu để hoàn thành training. Bởi phải tính toán gradient của từng điểm dữ liệu. Điều này sẽ thấy rõ rệt hơn khi làm việc với tập training lớn.

Do đó, trong thực tế, chúng ta sẽ kết hợp Batch GD với SGD lại thành Mini-Batch để sử dụng training, vừa giảm bớt thời gian thực thi của SGD mà lại tăng độ chính xác hơn Batch GD.



4.3.8 Question 4

Để improve model cho kết quả tốt hơn, chúng ta implement callback **TriangularSchedule** để schedule sự thay đổi của learning rate. Ở đây chúng ta sẽ dùng thuật toán Triangular Schedule không sử dụng decay.

```

1 class TriangularSchedule(tf.keras.callbacks.Callback):
2     def __init__(self, base_lr, max_lr, step_size=1):
3         super(TriangularSchedule, self).__init__()
4         self.base_lr = 0.01
5         self.max_lr = 0.1
6         self.step_size = 64.
7         self.scale_fn = lambda x: 1.
8         self.clr_iterations = 0.
9         self.trn_iterations = 0.
10        self.history = {}
11
12    def clr(self):
13        cycle = np.floor(1+self.clr_iterations/(2*self.step_size))
14        x = np.abs(self.clr_iterations/self.step_size - 2*cycle + 1)
15        return self.base_lr + (self.max_lr-self.base_lr)*np.maximum(0, (1-x))*self.scale_fn(cycle)
16
17    def on_train_begin(self, logs={}):
18        logs = logs or {}
19        if self.clr_iterations == 0:
20            tf.keras.backend.set_value(self.model.optimizer.lr, self.base_lr)
21        else:
22            tf.keras.backend.set_value(self.model.optimizer.lr, self.clr())

```

```

23     def on_batch_end(self, epoch, logs=None):
24         logs = logs or {}
25         self.trn_iterations += 1
26         self.clr_iterations += 1
27
28
29         self.history.setdefault('lr', []).append(tf.keras.backend.get_value(self.model.optimizer.lr))
30         self.history.setdefault('iterations', []).append(self.trn_iterations)
31
32         for k, v in logs.items():
33             self.history.setdefault(k, []).append(v)
34
35         tf.keras.backend.set_value(self.model.optimizer.lr, self.clr())

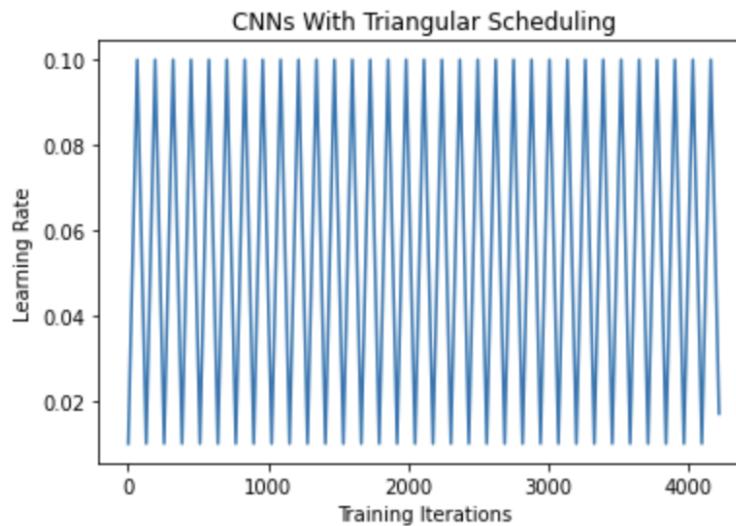
```

```

1 lr_schedule = TriangularSchedule(base_lr=0.01, max_lr=0.1, step_size=64.)

```

Learning rate trong quá trình training sẽ được thay đổi liên tục trong khoảng [0.01, 0.1]. Khi bắt đầu training thì learning rate sẽ được khởi tạo là 0.01.



Dánh giá

Ta cùng so sánh kết quả quá trình training khi thực hiện schedule learning rate với khi sử dụng learning rate cố định.

```

[68] history_lr = model.fit(x_train, y_train, batch_size=64, validation_split=0.1, epochs=5, callbacks=[lr_schedule])
      loss, accuracy = model.evaluate(x_test, y_test)
      print(f"The evaluation for test set: accuracy: {accuracy:.4f} - loss: {loss:.4f}")

Epoch 1/5
844/844 [=====] - 102s 121ms/step - loss: 0.2052 - accuracy: 0.9370 - val_loss: 0.1107 - val_accuracy: 0.9642
Epoch 2/5
844/844 [=====] - 100s 119ms/step - loss: 0.0576 - accuracy: 0.9819 - val_loss: 0.0468 - val_accuracy: 0.9857
Epoch 3/5
844/844 [=====] - 101s 120ms/step - loss: 0.0409 - accuracy: 0.9875 - val_loss: 0.0459 - val_accuracy: 0.9867
Epoch 4/5
844/844 [=====] - 100s 119ms/step - loss: 0.0390 - accuracy: 0.9906 - val_loss: 0.0521 - val_accuracy: 0.9847
Epoch 5/5
844/844 [=====] - 100s 119ms/step - loss: 0.0240 - accuracy: 0.9928 - val_loss: 0.0369 - val_accuracy: 0.9893
313/313 [=====] - 5s 16ms/step - loss: 0.0271 - accuracy: 0.9919
The evaluation for test set: accuracy: 0.9919 - loss: 0.2271

```

Hình 10: Quá trình training và evaluate testing data khi điều chỉnh LR

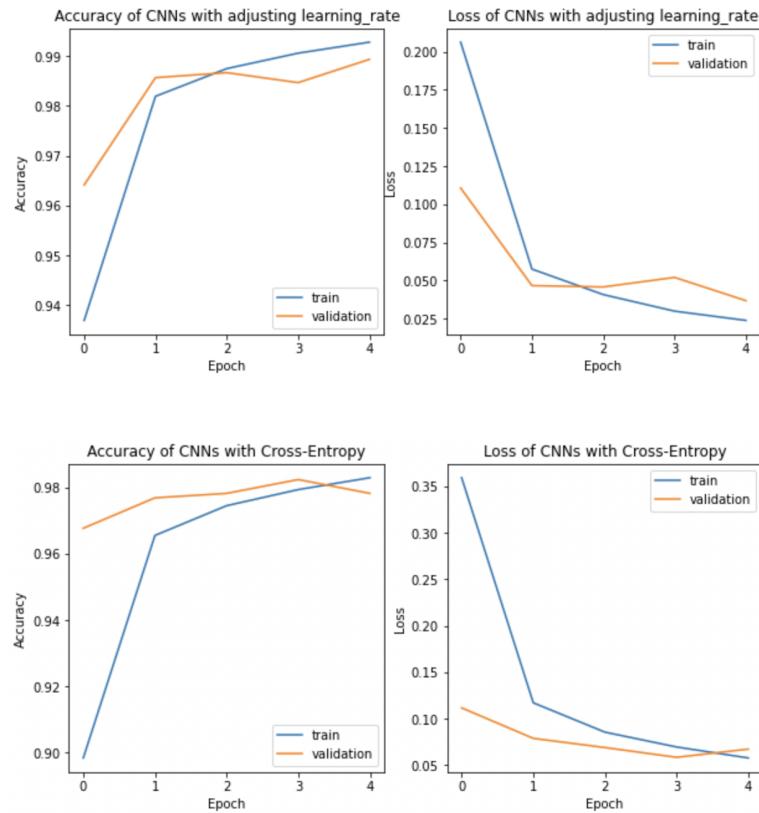
```

[68] history_cnn_cd = model.fit(x_train, y_train, batch_size=64, validation_split = 0.1, epochs=5)
      loss, accuracy = model.evaluate(x_test, y_test)
      print(f"The evaluation for test set: accuracy: {accuracy:.4f} - loss: {loss:.4f}")

Epoch 1/5
844/844 [=====] - 148s 175ms/step - loss: 0.3592 - accuracy: 0.8985 - val_loss: 0.1117 - val_accuracy: 0.9677
Epoch 2/5
844/844 [=====] - 149s 176ms/step - loss: 0.1171 - accuracy: 0.9655 - val_loss: 0.0789 - val_accuracy: 0.9768
Epoch 3/5
844/844 [=====] - 147s 175ms/step - loss: 0.0556 - accuracy: 0.9745 - val_loss: 0.0690 - val_accuracy: 0.9782
Epoch 4/5
844/844 [=====] - 149s 177ms/step - loss: 0.0596 - accuracy: 0.9793 - val_loss: 0.0585 - val_accuracy: 0.9823
Epoch 5/5
844/844 [=====] - 146s 175ms/step - loss: 0.0578 - accuracy: 0.9629 - val_loss: 0.0673 - val_accuracy: 0.9782
313/313 [=====] - 7s 21ms/step - loss: 0.0672 - accuracy: 0.9775
The evaluation for test set: accuracy: 0.9775 - loss: 0.0672

```

Hình 11: Quá trình training và evaluate testing data với LR cố định



Khi thực hiện schedule learning rate với *triangular rule*:

- Model đạt được kết quả tốt hơn - độ chính xác đạt được lên 99,19% khi kiểm tra với tập test mà chỉ train với 5 epochs.
- Kết quả loss sẽ thấp hơn qua các epochs, đồng nghĩa với việc trọng số (\mathbf{W}, \mathbf{b}) hội tụ đến điểm globalminimal nhanh hơn rất nhiều. Điều này dẫn đến thời gian train 1 epoch sẽ giảm 2/3 so với LR cố định.

4.4 Đánh giá kết quả

Khi đánh giá trên tập test của dataset, độ chính xác của model đến mức tối ưu với các thuật toán được áp dụng là **99,19%**.

Có thể thấy model chúng ta xây dựng có độ chính xác cao (trên 99%) và không gặp tình trạng overfitting hay underfitting.

4.5 Chạy thử chương trình

Source chương trình với model huấn luyện sau cùng, dùng để kiểm tra:
<https://github.com/huylys12/handwritten-digit-recognition.git>

Hướng dẫn chi tiết có thể xem ở link Google Codelab



Tài liệu

- [1] Adrian Rosebrock (2020). OCR:Hand writing recognition with OpenCV, Keras, and TensorFlow.
- [2] Savita Ahlawat (2020). Improved Handwritten Digit Recognition Using Convolutional Neural Networks (CNN).
- [3] Samay Pashine, Ritik Dixit, and Rishika Kushwah. Handwritten digit recognition using machine and deep learning algorithms.
- [4] <https://machinelearningcoban.com/2017/02/17/softmax/>.
- [5] <https://www.kaggle.com/c/digit-recognizer>.