BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT THÀNH PHỐ HỒ CHÍ MINH
60 NĂM XÂY DỰNG VÀ PHÁT TRIỂN

TRƯƠNG NGỌC SƠN
PHẠM NGỌC SƠN
LÊ MINH

HUẤN LUYỆN MẠNG NƠ-RON HỌC SÂU

(Sách chuyên khảo)

NHÀ XUẤT BẢN
ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

PGS.TS. TRƯƠNG NGỌC SƠN
TS. PHẠM NGỌC SƠN
ThS. LÊ MINH

HUẤN LUYỆN
MẠNG NƠ-RON HỌC SÂU

(Sách chuyên khảo)

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA
THÀNH PHỐ HỒ CHÍ MINH – 2022

LỜI NÓI ĐẦU

Máy học và trí tuệ nhân tạo đang phát triển mạnh và ngày càng được ứng dụng nhiều vào cuộc sống nhằm góp phần tạo ra các công nghệ có thể thay thế sức lao động của con người. Sự phát triển của trí tuệ nhân tạo không chỉ tạo ra các hệ thống tự động hỗ trợ con người mà nó còn hướng đến thay thế các công việc mà trước đây chỉ có con người mới thực hiện được. Trí tuệ nhân tạo đề cập đến một loại ứng dụng có khả năng thực hiện các công việc một cách thông minh như con người vì bộ não con người đóng vai trò quyết định trong các hoạt động. Chính vì thế, khi nói đến trí tuệ nhân tạo, chúng ta hướng đến việc thiết kế một mạng nơ-ron nhân tạo sao cho nó hoạt động tương tự như bộ não con người bởi vì nơ-ron là đơn vị tạo nên bộ não người. Ngày nay các lĩnh vực của trí tuệ nhân tạo không ngừng phát triển như mạng học sâu, xử lý ngôn ngữ tự nhiên,… Những mô hình này đều dựa trên các mạng nơ-ron nhân tạo nhiều lớp. Hiểu về cấu trúc và phương pháp huấn luyện các mạng nơ-ron nhân tạo, mạng học sâu là nền tảng cho việc nghiên cứu và phát triển các ứng dụng của mạng học sâu trong nhiều lĩnh vực khác nhau. Quyển sách này trình bày mô hình toán của các phương pháp huấn luyện được sử dụng trong mạng học sâu. Hơn nữa, quyển sách trình bày các nghiên cứu xuất phát từ mô hình toán, sau đó hướng đến triển khai các mô hình sử dụng ngôn ngữ Python. Các chương trình được nghiên cứu và viết sát theo mô hình toán mà không sử dụng các thư viện viết sẵn giúp người đọc hiểu được mô hình toán của quá trình huấn luyện và phương pháp lập trình. Trong các ứng dụng nâng cao, chúng ta đi sử dụng các thư viện hỗ trợ sau khi đã hiểu tường tận về các phương pháp huấn luyện, ý nghĩa các thông số mô hình.

Quyển sách **Huấn luyện mạng nơ-ron học sâu** được biên soạn bao gồm các lý thuyết về huấn luyện mạng nơ-ron dựa trên giải thuật Gradient Descent và các phương pháp cải tiến của giải thuật Gradient Descent. Các mô hình toán của quá trình huấn luyện được thực thi trên ngôn ngữ Python và thư viện *numpy*. Trong phần cuối của quyển sách, chúng ta sẽ phát triển một số ứng dụng thực tế áp dụng các mô hình mạng học sâu. Quyển sách được biên soạn nhằm phục vụ cho quá trình nghiên cứu và ứng dụng máy học, trí tuệ nhân tạo, mạng học sâu của sinh viên các ngành Kỹ thuật tại các trường Đại học.

Nội dung của quyển sách được tổ chức thành 6 chương:

Chương 1: Giới thiệu về mạng nơ-ron nhân tạo và các phương pháp huấn luyện mạng. Chương 1 bao gồm các mô hình toán cụ thể của quá trình huấn luyện mạng nơ-ron.

Chương 2: Giới thiệu các phương pháp huấn luyện mạng nơ-ron được sử dụng trong các mạng nhiều lớp (multi-layer neural network), mạng nơ-ron sâu (Deep neural network) và chương trình được xây dựng đi kèm cho mỗi phương pháp huấn luyện. Nội dung Chương 2 trình bày các phương pháp huấn luyện truyền thống và các phương pháp cải tiến được sử dụng phổ biến trong phát triển mạng học sâu hiện đại.

Chương 3: Giới thiệu các phương pháp tối ưu cho quá trình huấn luyện các mạng nơ-ron. Bên cạnh đó, Chương 3 cũng đồng thời giới thiệu các chương trình được xây dựng đi kèm cho mỗi phương pháp tối ưu.

Chương 4: Giới thiệu về mạng nơ-ron tích chập và phương pháp huấn luyện cho mạng nơ-ron tích chập. Chương này phân tích cấu trúc của các mạng nơ-ron tích chập, chức năng của các lớp, phương pháp huấn luyện và cập nhật trọng số cho mạng nơ-ron tích chập và ứng dụng minh họa được xây dựng kèm.

Chương 5: Trình bày phương pháp lượng tử hóa các thông số của mô hình mạng nơ-ron để tối ưu mạng nơ-ron cho mục đích thực thi trên các hệ thống phần cứng có tài nguyên giới hạn. Chương này cũng giới thiệu chi tiết về mạng nơ-ron nhị phân, phương pháp cập nhật trọng số trong mạng nơ-ron nhị phân và ứng dụng minh họa đi kèm.

Chương 6: Xây dựng các ứng dụng mạng nơ-ron sâu dựa trên thư viện Pytorch. Trong chương này, chúng ta tìm hiểu một số đặc điểm của thư viện Pytorch, xây dựng và huấn luyện các ứng dụng mạng nhiều lớp sử dụng thư viện Pytorch.

Cuối cùng, tuy đã rất cố gắng biên soạn và chỉnh sửa nhưng quyển sách khó tránh khỏi những thiếu sót, tác giả mong nhận được những đóng góp quý báu từ sinh viên và quý đồng nghiệp để quyển sách này được hoàn thiện hơn trong những lần tái bản tiếp theo. Mọi ý kiến đóng góp xin gửi về tác giả theo địa chỉ liên lạc:

Bộ môn Kỹ thuật Máy tính – Viễn thông, Khoa Điện - Điện tử,
Trường Đại học Sư phạm Kỹ thuật TP.HCM

Email: sontn@hcmute.edu.vn

Tác giả

MỤC LỤC

LỜI NÓI ĐẦU 3

Chương 1: GIỚI THIỆU 9
1.1 NƠ-RON NHÂN TẠO 9
1.2 MẠNG NƠ-RON NHÂN TẠO 13
1.3 CƠ SỞ CỦA QUÁ TRÌNH HUẤN LUYỆN MẠNG NƠ-RON 15
1.3.1 Đạo hàm và Gradient của hàm số 16
1.3.2 Gradient và chiều tăng cực đại của hàm số nhiều biến 17
1.3.3 Gradient Descent – Giảm dần theo độ dốc 19
1.3.4 Gradient Descent và bài toán hồi quy tuyến tính 20

Chương 2: HUẤN LUYỆN MẠNG NƠ-RON 25
2.1 HUẤN LUYỆN MẠNG NƠ-RON VỚI GRADIENT DESCENT 25
2.2 BATCH GRADIENT DESCENT VÀ STOCHASTIC GRADIENT DESCENT. 33
2.3 PHƯƠNG PHÁP LAN TRUYỀN NGƯỢC SAI SỐ (BACKPROPAGATION) 34
2.4 ỨNG DỤNG BACKPROPAGATION VÀ GRADIENT DESCENT CHO BÀI TOÁN NHẬN DẠNG KÝ TỰ VIẾT TAY 38

Chương 3: CÁC PHƯƠNG PHÁP TỐI ƯU QUÁ TRÌNH HUẤN LUYỆN MẠNG NƠ-RON 53
3.1 SỬ DỤNG MOMENTUM 53
3.2 ADAGRAD – ADAPTIVE GRADIENT DESCENT 59
3.3 RMSPROP (ROOT MEAN SQUARED PROPAGATION) 65
3.4 ADAM 71
3.5 SỬ DỤNG KỸ THUẬT DROPOUT 78

Chương 4: HUẤN LUYỆN MẠNG NƠ-RON TÍCH CHẬP 89
4.1 MẠNG NƠ-RON TÍCH CHẬP 89
4.2 HUẤN LUYỆN MẠNG NƠ-RON TÍCH CHẬP 92

Chương 5: LƯỢNG TỬ HÓA CÁC THÔNG SỐ MÔ HÌNH MẠNG NƠ-RON 106
5.1 MẠNG NƠ-RON VỚI TẬP TRỌNG SỐ NHỊ PHÂN 106
5.2 HUẤN LUYỆN MẠNG NƠ-RON NHỊ PHÂN 108

Chương 6: PHÁT TRIỂN ỨNG DỤNG MẠNG HỌC SÂU VỚI PYTORCH 120
6.1 GIỚI THIỆU 120
6.2 AUTOGRAD – AUTOMATIC DIFFIRENTIATION 123
6.3 SỬ DỤNG LỚP OPTIMIZATION CỦA PYTORCH. 131
6.4 PYTORCH NN MODULE 134
6.5 SỬ DỤNG NN.SEQUENTIAL 143

TÀI LIỆU THAM KHẢO 150

LIỆT KÊ THUẬT NGỮ VIẾT TẮT

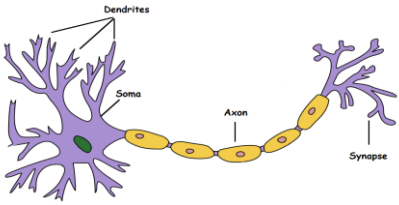
NN	Neural Network
AI	Artificial Intelligence
NLP	Natural Language Processing
CPU	Central Processing Unit
GPU	Graphic Processing Unit
RNN	Recurent Neural Network
CNN	Convolutional Neural Network
GD	Gradient Descent
SGD	Stochastic Gradient Descent
ReLU	Rectified Linear Unit
MNIST	Modified National Institute of Standards and Technology
MSE	Mean Squared Error
FM	Feature Map
API	Application Programming Interface
DNN	Deep Neural Network

Chương 1
GIỚI THIỆU

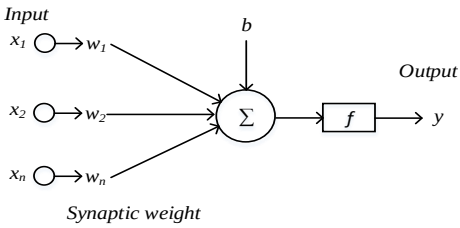
1.1 NƠ-RON NHÂN TẠO

Nơ-ron là đơn vị cơ bản trong bộ não con người. Các nơ-ron kết nối với nhau tạo nên một mạng lưới các nơ-ron trong não giúp con người thực hiện các chức năng từ nhận thức, học, ghi nhớ đến hành động. Mạng nơ-ron nhân tạo (Artificial Neural Network) là một hệ thống tính toán được lấy ý tưởng từ mạng nơ-ron sinh học trong não bộ con người. Một mạng nơ-ron nhân tạo là một tập hợp của các đơn vị nơ-ron nhân tạo được kết nối với nhau. Tùy thuộc vào cách mà các nơ-ron kết nối với nhau mà chúng ta sẽ có các kiến trúc mạng khác nhau. Các kết nối trong mạng cho phép tín hiệu có thể lan truyền từ ngõ vào đến ngõ ra và có thể tồn tại các lan truyền ngược từ ngõ ra của một hay nhiều nơ-ron đến ngõ vào của một hay nhiều nơ-ron khác trong mạng.

Mỗi nơ-ron nhân tạo là một mô hình toán của một nơ-ron sinh học, cho phép nó có thể tạo ra giá trị ngõ ra từ các giá trị tín hiệu ngõ vào. Mô hình toán cho mỗi nơ-ron nhân tạo bao gồm 3 thành phần chủ yếu: bộ nhân (multiplication), bộ cộng (summation) và hàm kích hoạt (activation). Các tín hiệu ngõ vào được nhân với trọng số tương ứng (weight). Sau đó, kết quả của phép nhân được cộng lại và đưa qua hàm kích hoạt (activation function). Mô hình toán cho một nơ-ron được minh họa trong Hình 1.1.



(a) Cấu trúc nơ-ron



(a) Mô hình toán của nơ-ron

Hình 1.1: Nơ-ron sinh học và mô hình toán của nơ-ron sinh học.

Cấu trúc của một nơ-ron sinh học được minh họa trong Hình 1.1(a). Trong mạng nơ-ron, thông tin truyền đến tế bào thần kinh thông qua các nhánh thần kinh (dendrite). Soma sẽ xử lý các tín hiệu rồi truyền tín hiệu đó qua thân tế bào (Axon) và đến các tế bào khác thông qua các khớp nối thần kinh (Synapse). Nơ-ron sinh học được mô hình hóa như Hình 1.1(b). Các ngõ vào (input) đóng vai trò như các nhánh thần kinh (dendrite) nhận

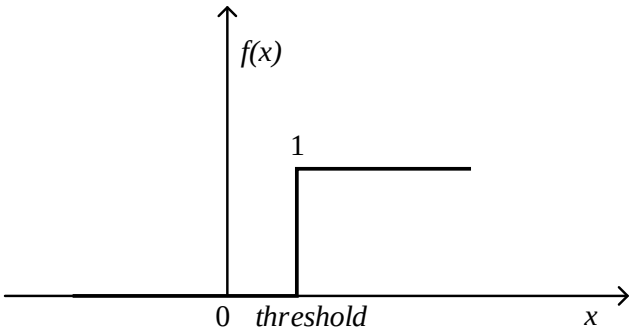
các tín hiệu từ các nơ-ron khác. Các tín hiệu được khuếch đại lớn hơn hay bị làm suy giảm nhỏ hơn phụ thuộc vào các trọng số (weight). Trong đó, các trọng số được xem như một tác nhân có khả năng thay đổi tín hiệu nhận được từ các tế bào thần kinh ở lớp trước. Tín hiệu đến nơ-ron từ tất cả các ngõ vào sẽ được cộng lại với nhau để quyết định ngõ ra. Ví dụ, nếu tổng các tín hiệu nhận được nhỏ hơn một mức ngưỡng, ngõ ra sẽ ở mức 0. Ngược lại, nếu tổng các tín hiệu nhận được lớn hơn mức ngưỡng, ngõ ra sẽ lên mức 1. Trong các tế bào thần kinh sinh học, khi tổng các tín hiệu đạt lớn hơn mức ngưỡng, nơ-ron sẽ phát ra một xung truyền đến các nơ-ron tiếp theo. Trong mô hình toán của nơ-ron, việc quyết định ngõ ra được thực hiện bởi hàm kích hoạt (activation function). Đơn giản trong trường hợp này hàm kích hoạt là hàm ngưỡng. Nếu ngõ vào hàm ngưỡng lớn hơn một giá trị đặt trước thì ngõ ra được thiết lập là 1, ngược lại ngõ ra sẽ được xoá về 0. Mô hình toán của một nơ-ron với các ngõ vào và trọng số được mô tả bằng phương trình (1.1):

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1.1}$$

Trong đó y là ngõ ra của nơ-ron, f là hàm kích hoạt, x_i là các ngõ vào và w_i là các trọng số của các ngõ vào tương ứng. Các trọng số w_i có giá trị thay đổi trong quá trình huấn luyện nơ-ron. Nếu w_i và x_i là các giá trị dương, giá trị ngõ vào x_i góp phần làm cho ngõ ra y tăng. Trong trường hợp này, các khớp nối được gọi là các khớp kích thích (Excitatory synapse). Trong trường hợp giá trị của w_i là số âm, các trọng số w_i sẽ làm đảo giá trị nhận được từ x_i. Các khớp nối có giá trị trọng số âm được gọi là các khớp nối hạn chế (Inhibitory synapse). Trong công thức (1.1), b là giá trị độ lệch (Bias). Bias được xem như là một mặt phẳng được thêm vào trong phương trình tuyến tính để hiệu chỉnh ngõ ra. Bias là một hằng số cho phép mô hình khớp với dữ liệu được huấn luyện.

Các hàm truyền *f*, hay còn được gọi là hàm kích hoạt, là các phương trình toán học xác định ngõ ra của các nơ-ron. Trong mô hình nơ-ron sinh học, hàm truyền quyết định các nơ-ron có tích cực hay không, tức là tạo ra xung ngõ ra cho các nơ-ron kết nối sau nó hay không khi ngõ vào đạt đến một mức nào đó. Trong trường hợp tương tự, nếu sử dụng hàm truyền là hàm ngưỡng thì ngõ ra của nơ-ron sẽ có giá trị 1 khi hàm tổng vượt qua mức ngưỡng. Các mô hình mạng khác nhau sử dụng các hàm truyền khác nhau.

Hàm ngưỡng là hàm kích hoạt đơn giản cho các đơn vị nơ-ron cơ bản. Đồ thị hàm ngưỡng được mô tả trong Hình 1.2.

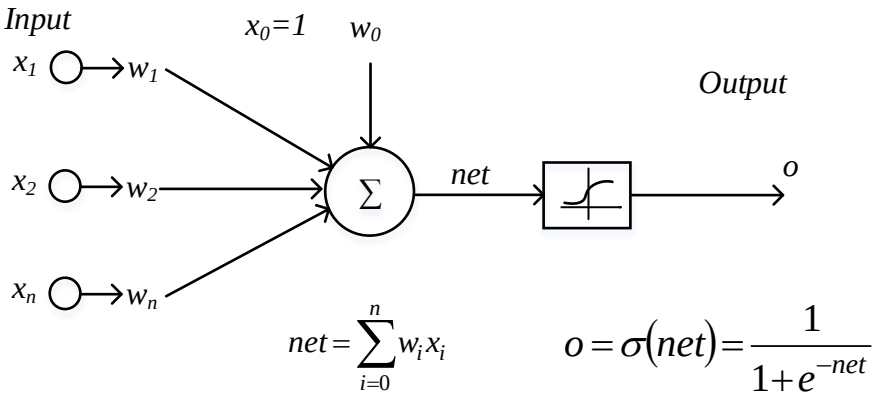


Hình 1.2: Hàm ngưỡng.

Ngõ ra của nơ-ron sử dụng hàm ngưỡng làm hàm kích hoạt được xác định bởi phương trình sau:

$$y = \begin{cases} 1 \text{ nếu } \sum_{i=1}^{n} w_i x_i + b \geq threshold \\ 0 \text{ nếu } \sum_{i=1}^{n} w_i x_i + b < threshold \end{cases} \quad (1.2)$$

Hàm ngưỡng được sử dụng trong trường hợp các đơn vị nơ-ron cơ bản như Perceptron. Các nơ-ron trong mạng nhiều lớp sử dụng một số hàm kích hoạt phi tuyến như hàm *Sigmoid, Tanh, ReLU*. Các nơ-ron ở các lớp ngõ ra trong các bài toán phân lớp thường sử dụng hàm tạo ra phân bố xác suất như hàm *Softmax*.

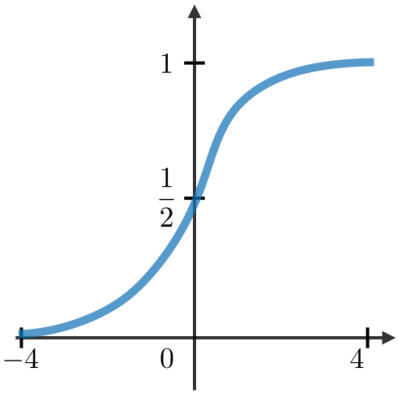


Hình 1.3: Mô hình nơ-ron với hàm kích hoạt phi tuyến.

Hình 1.3 mô tả một đơn vị nơ-ron với hàm kích hoạt là hàm phi tuyến. Hàm *Sigmoid* được sử dụng phổ biến cho các nơ-ron ở các lớp ẩn. Hàm *Sigmoid* cho ngõ ra luôn dương và không tuyến tính từ trong khoảng (0, 1).

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{1.3}$$

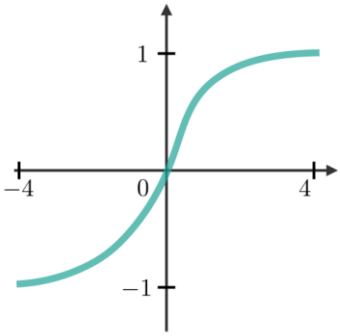
Hàm *Sigmoid* có mức độ tính toán phức tạp. Khi được triển khai trên các hệ thống có cấu trúc vi xử lý nhỏ, hàm *Sigmoid* tiêu tốn nhiều năng lượng và thời gian cho quá trình tính toán. Hơn nữa, một trong những vấn đề của các nơ-ron sử dụng hàm *Sigmoid* là việc các Gradient bị khử dẫn đến các trọng số không được cập nhật. Đồ thị hàm *Sigmoid* được biểu diễn trong Hình 1.4.



Hình 1.4: Đồ thị hàm Sigmoid.

Hàm *Tanh* cũng là một hàm kích hoạt phi tuyến tương tự như hàm Sigmoid. Hàm *Tanh* có thể ít thấy trong các mạng nơ-ron nhiều lớp truyền thẳng nhưng lại xuất hiện nhiều trong các mạng nơ-ron hồi qui. Hàm *Tanh* tạo ra ngõ ra có giá trị từ -1 đến 1 thay vì từ 0 đến 1 như hàm *Sigmoid*.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{1.4}$$

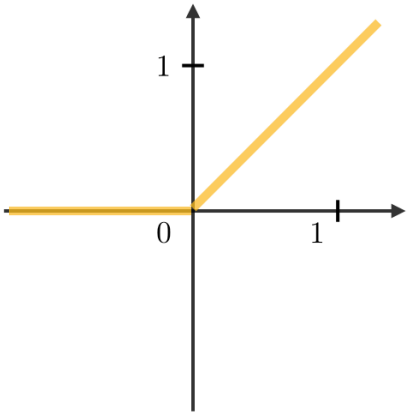


Hình 1.5: Đồ thị hàm tanh.

Trong các mạng nhiều lớp hàm các hàm phi tuyến như hàm *Tanh* và hàm *Sigmoid* có thể làm cho giá trị Gradient bị triệt tiêu trong quá trình lan truyền qua nhiều lớp mạng. Một trong những phương pháp cải thiện vấn đề Gradient bị triệt tiêu là là sử dụng các hàm chỉnh lưu tuyến tính, *ReLU* (Rectified linear unit). *ReLU* được sử dụng phổ biến trong các mạng nhiều lớp vì nó giúp khắc phục được một số nhược điểm của hàm phi tuyến. Hàm *ReLU* là một hàm tuyến tính khi giá trị ngõ vào dương và luôn bằng 0 khi giá trị ngõ vào âm. *ReLU* được thể hiện bằng phương trình toán sau:

$$ReLU(x) = \begin{cases} 0, & \text{nếu } x < 0 \\ x, & \text{nếu } x \geq 0 \end{cases} \tag{1.5}$$

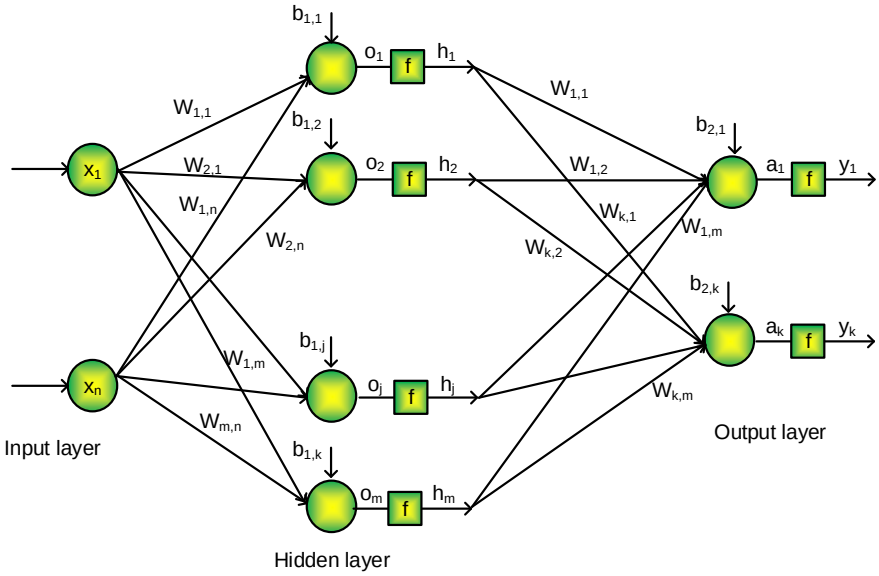
Hàm *ReLU* được thực hiện đơn giản hơn so với các hàm phi tuyến vì nó không cần phép toán phức tạp, cho phép tiết kiệm tài nguyên và thời gian xử lý trong các mạng nơ-ron nhân tạo nhiều lớp. Sử dụng hàm *ReLU* giúp cho quá trình học hội tụ nhanh hơn. Hơn nữa nó khắc phục được tình trạng triệt tiêu Gradient của hàm *Sigmoid* và hàm *Tanh*.



Hình 1.6: Đồ thị hàm ReLU.

1.2 MẠNG NƠ-RON NHÂN TẠO

Mạng nơ-ron nhân tạo được hình thành từ việc kết nối các nơ-ron nhân tạo lại với nhau theo một thể thức nào đó. Việc kết nối sẽ hình thành nên sự khác biệt về cấu trúc cũng như tính năng của các mạng nơ-ron. Kiểu kết nối đơn giản và truyền thống là kết nối lan truyền thẳng như mô tả trong Hình 1.7.



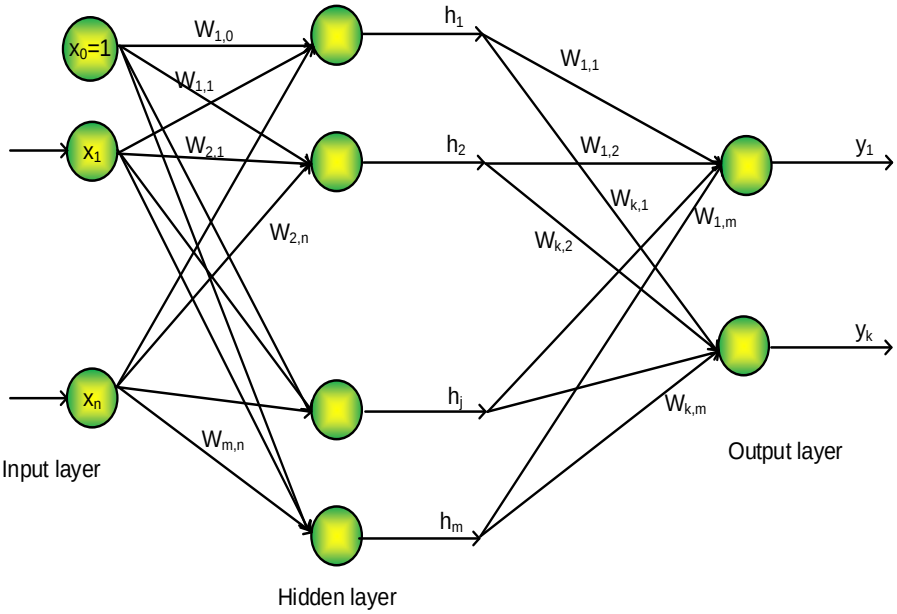
Hình 1.7: Mạng nơ-ron lan truyền thẳng phổ biến với các trọng số kết nối.

Giá trị ngõ ra cuối cùng của mạng được xác định thông qua quá trình tính toán cho từng nút mạng từ ngõ vào đến các lớp ẩn và đến ngõ ra.

$$
\begin{aligned}
o_1 &= x_1W_{1,1} + \ldots + x_nW_{1,n} + b_{1,1} \\
h_1 &= f(o_1) \\
o_2 &= x_1W_{2,1} + \ldots + x_nW_{2,n} + b_{1,2} \\
h_2 &= f(o_2) \\
a_1 &= h_1W_{1,1} + \ldots + h_mW_{1,m} + b_{2,1} \\
y_1 &= f(a_1) \\
a_k &= h_1W_{k,1} + \ldots + h_mW_{k,m} + b_{2,k} \\
y_k &= f(a_k)
\end{aligned}
\tag{1.6}
$$

Trong đó $W_{i,j}$ là trọng số kết nối từ ngõ ra nơ-ron thứ j đến nơ-ron thứ i ở lớp sau nó. Các hàm kích hoạt f có thể là một trong các hàm phi tuyến hoặc hàm chỉnh lưu tuyến tính, ReLU.

Trong một số trường hợp, để đơn giản chúng ta có thể xem độ bias như một trọng số đặc biệt kết nối ngõ vào có giá trị hằng là 1. Các hàm kích hoạt không được thể hiện riêng lẻ mà được tích hợp trong mỗi đơn vị nơ-ron. Một mạng nơ-ron như Hình 1.7 có thể được thể hiện dưới dạng đơn giản như Hình 1.8.



Hình 1.8: Mạng nơ-ron lan truyền thẳng trong đó bias được xem như trọng số của một ngõ vào với giá trị luôn bằng 1, hàm kích hoạt được tích hợp trong mỗi nơ-ron đơn vị.

Mạng nơ-ron nhân tạo là một mô hình toán học trong đó tín hiệu được lan truyền và tính toán qua mỗi đơn vị trong mạng. Các giá trị trọng số có vai trò quyết định đến chức năng của mạng cũng như giá trị các ngõ ra. Trong mô hình trên, chúng ta có thể xem mỗi nơ-ron là một hàm đa biến, trong đó các biến là các giá trị ngõ vào và các trọng số đóng vai trò là các hằng số tương ứng cho mỗi biến ngõ vào. Nếu xem mỗi nơ-ron là một hàm, trong đó ánh xạ từ các ngõ vào và một ngõ ra thì giá trị các trọng số quyết định qui luật của ánh xạ này. Hay nói cách khác, giá trị trọng số quyết định đến chức năng của mỗi nơ-ron trong mạng. Quá trình huấn luyện nơ-ron là quá trình đi tìm tập trọng số của các nơ-ron trong mạng sao cho mạng nơ-ron thực thi một hàm số đa biến đáp ứng được yêu cầu của bài toán.

1.3 CƠ SỞ CỦA QUÁ TRÌNH HUẤN LUYỆN MẠNG NƠ-RON

Một mạng nơ-ron nhiều lớp được mô hình hóa như một hàm đa biến tạo ra một ánh xạ từ một vector ngõ vào đến vector ngõ ra. Quá trình huấn luyện mạng là một quá trình tối ưu hàm đa biến sao cho hàm số đáp ứng được với tập dữ liệu. Hay nói cách khác là hàm số tạo ra ngõ ra mong muốn đối với từng vector ngõ vào. Mặc dù cho đến nay mạng nơ-ron nhân tạo được phát triển với nhiều mô hình và cấu trúc khác nhau, tuy nhiên các quá trình huấn

luyện vẫn dựa trên các phương pháp tối ưu hàm nhiều biến phổ biến. Cơ sở toán học hỗ trợ cho mạng nơ-ron nhân tạo, mạng học sâu và các lĩnh vực thuộc máy học, trí tuệ nhân tạo bao gồm đại số tuyến tính, lý thuyết tối ưu, lý thuyết đồ thị, xác suất thống kê. Trong quyển sách này, tác giả không trình bày lại hết các kiến thức cơ sở toán học như trên mà chỉ tập trung các mô hình toán cơ bản và phổ biến được dùng để tối ưu hàm nhiều biến và ứng dụng vào giải quyết một số bài toán về huấn luyện mạng nơ-ron nhiều lớp. Trong các bài tập ứng dụng, tác giả trình bày mô hình toán cho các quá trình, các phương pháp và giải thuật huấn luyện cho mạng nơ-ron. Sau đó, quyển sách hướng đến việc giải thích phương pháp ứng dụng các phép toán trong đại số tuyến tính để thực thi các mô hình toán cho quá trình huấn luyện. Cuối cùng quyển sách trình bày phương pháp lập trình cho các bài toán huấn luyện cụ thể dựa vào mô hình toán.

1.3.1 Đạo hàm và Gradient của hàm số

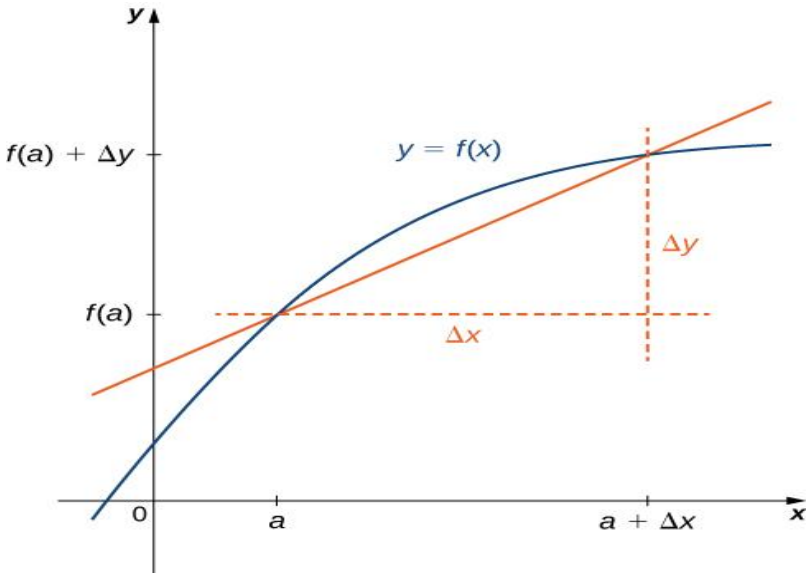
Đạo hàm của hàm số tại một điểm cho biết tốc độ thay đổi của hàm số tại điểm đó. Khái niệm đạo hàm được ứng dụng nhiều trong các bài toán tối ưu. Giả sử hàm y là hàm số theo biến x và được biểu diễn bởi $y = f(x)$. Đạo hàm của hàm số y theo biến x được định nghĩa như sau:

$$f'(x) = \lim_{\Delta x \to o} \frac{f(x + \Delta x) - f(x)}{\Delta x} \tag{1.7}$$

Từ định nghĩa trên, chúng ta thấy đạo hàm của hàm số đo mức độ thay đổi của hàm y so với mức độ thay đổi của giá trị biến x. Đạo hàm của hàm số có thể được thể hiện dưới dạng đơn giản (1.8).

$$\begin{aligned} y' &= \lim_{\Delta x \to o} \frac{\Delta y}{\Delta x} \\ \frac{dy}{dx} &= \lim_{\Delta x \to o} \frac{\Delta y}{\Delta x} \end{aligned} \tag{1.8}$$

Giá trị đạo hàm cho biết chiều thay đổi và tốc độ thay đổi của hàm số theo biến số x. Đạo hàm còn được xem như độ dốc của hàm số xét trong một khoảng nhỏ. Trong ví dụ trên, y là hàm một biến theo x. Như vậy độ dốc của hàm số tại một điểm có thể được tính thông qua tính đạo hàm của hàm số tại điểm đó. Mô tả hình học của đạo hàm được minh họa trong Hình 1.9.



Hình 1.9: Mô tả hình học của đạo hàm $\frac{dy}{dx} = \lim_{\Delta x \to o} \frac{\Delta y}{\Delta x}$.

1.3.2 Gradient và chiều tăng cực đại của hàm số nhiều biến

Đối với hàm một biến, chúng ta dùng khái niệm đạo hàm để tính độ dốc của hàm số cũng như xác định chiều biến thiên của hàm số theo biến số. Trong các ứng dụng máy học và trí tuệ nhân tạo, các hệ thống thường được mô hình hóa bằng các hàm đa biến. Đối với hàm đa biến, chúng ta cũng có thể tính đạo hàm của hàm số thông qua tính đạo hàm từng phần của hàm số theo từng biến số cụ thể. Như vậy, độ dốc của hàm số tại một điểm phụ thuộc vào tất cả các đạo hàm từng phần của hàm số tại vị trí đang xét. Gradient được hiểu như độ dốc của hàm số xét tại một điểm xác định. Trong quyển sách này, chúng ta có thể sử dụng khái niệm Gradient và độ dốc đang xen lẫn nhau. Gradient của hàm nhiều biến là một vector mà thành phần của nó là các đạo hàm từng phần theo các biến. Giả sử chúng ta có hàm số $f(x, y, z)$ là một hàm với 3 biến x, y, z, Gradient của hàm f được ký hiệu ∇f và được định nghĩa như sau:

$$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}\right]$$
$$\nabla f = \left[\frac{\partial f}{\partial x} i + \frac{\partial f}{\partial y} j + \frac{\partial f}{\partial z} k\right] \tag{1.9}$$

Trong đó i, j, k là vector đơn vị ứng với các trục x, y, z.

Vector Gradient có chiều và độ lớn xác định bởi giá trị đạo hàm của hàm số theo các biến thành phần x, y, z. Điều này nói lên rằng giá trị

độ lớn và chiều của vector Gradient cho biết tỷ lệ thay đổi của hàm số theo các biến thành phần x, y, z.

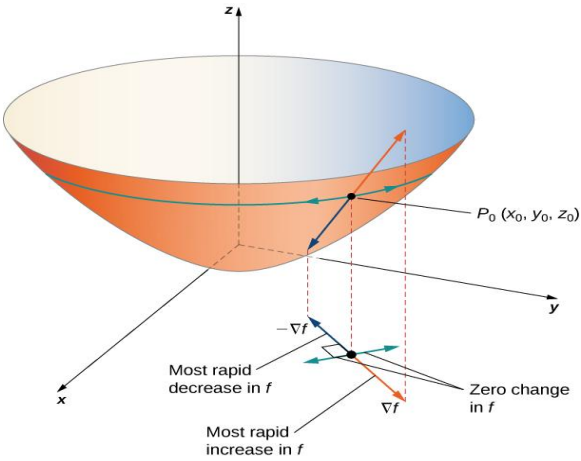
Đối với hàm số $f(x, y, z)$, đạo hàm từng phần của hàm số đối với biến x cho biết tỷ lệ thay đổi của hàm f theo chiều của x, đạo hàm từng phần của hàm f đối với y cho biết tỷ lệ thay đổi của f theo hướng y và đạo hàm từng phần của hàm f đối với z cho biết tỷ lệ thay đổi của f theo hướng z. Một bài toán đặt ra là làm thế nào để tính tỷ lệ thay đổi của hàm f theo một hướng cụ thể nào đó. Trong trường hợp này, chúng ta có thể sử dụng khái niệm đạo hàm có hướng của hàm số f. Đạo hàm có hướng của hàm số f là một giá trị vô hướng cho biết tỷ lệ thay đổi của hàm số theo một hướng của một vector b nào đó.

$$\mathrm{D}_{\vec{b}} f = \nabla f . \vec{b} \tag{1.10}$$

Trong đó $\mathrm{D}_{\vec{b}} f$ là đạo hàm có hướng của hàm f theo hướng vector b. Phép nhân vô hướng của vector Gradient và vector b được thể hiện ở (1.11).

$$\nabla f . \vec{b} = \|\nabla f\| \, \|b\| cos\theta \tag{1.11}$$

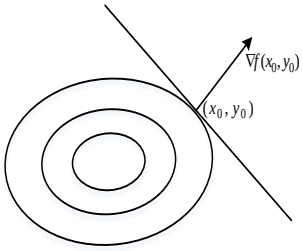
Trong (1.11), θ là góc tạo bởi vector b và vector Gradient của hàm f. Đồng thời, giá trị của đạo hàm có hướng đạt lớn nhất khi góc $\theta = 0$. Đạo hàm có hướng đạt giá trị lớn nhất khi hướng của vector Gradient trùng với hướng của vector b. Có thể thấy rằng chiều tăng lớn nhất của hàm số f chính là chiều của vector Gradient. Gradient được sử dụng để xác định chiều tăng lớn nhất của hàm số. Thông qua Gradient, các phương pháp tối ưu sẽ dựa vào chiều của vector Gradient mà thay đổi các biến số thích hợp sao cho hàm số đạt đến giá trị mong muốn nhanh nhất.



Hình 1.10: Gradient cho biết giá trị lớn nhất và bé nhất của đạo hàm có hướng tại một điểm.

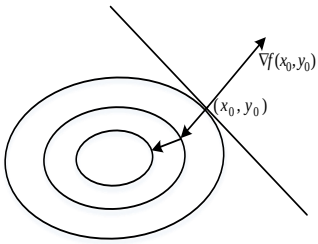
1.3.3 Gradient Descent – Giảm dần theo độ dốc

Gradient escent (Giảm dần theo độ dốc) là một thuật toán tối ưu được sử dụng phổ biến nhằm cực tiểu hóa một hàm số bằng cách lặp lại việc di chuyển theo hướng dốc nhất được xác định bằng giá trị âm của Gradient. Trong máy học và trí tuệ nhân tạo, phương pháp này được sử dụng để tính toán cho quá trình cập nhật trọng số của mô hình sao cho hàm mất mát (loss function) đạt giá trị cực tiểu mong muốn. Trước hết, chúng ta biết rằng Gradient là vector mà các thành phần của nó là đạo hàm từng phần của hàm số theo từng biến số tương ứng. Chiều và độ lớn của vector Gradient phụ thuộc vào chiều và độ lớn của các đạo hàm từng phần. Vector Gradient giúp xác định chiều và tốc độ thay đổi của các biến thành phần thích hợp sao cho hàm f sớm đạt giá trị cực tiểu mong muốn nhất.



Hình 1.11: Gradient của hàm f tại (x_0, y_0).

Giả sử chúng ta có hàm $f(x, y)$ là hàm mục tiêu trong các bài toán tối ưu. Quá trình tối ưu được thực hiện bằng cách điều chỉnh các thông số sao cho hàm mục tiêu đạt giá trị cực tiểu. Tại mỗi điểm có tọa độ (x, y), vector Gradient có chiều trùng với chiều tăng cực đại của hàm số. Do đó, để đạt đến giá trị cực tiểu của hàm số một cách nhanh nhất, chúng ta đi theo chiều ngược lại với chiều của vector Gradient. Phương pháp này được gọi là Gradient Descent.



Hình 1.12: Gradient Descent – giảm dần theo độ dốc.

Trong phương pháp Gradient Descent, việc chọn điểm bắt đầu (x_0, y_0) là ngẫu nhiên. Sau đó, chúng ta sẽ thực hiện các bước thay đổi (cập nhật) theo hướng ngược chiều với vector Gradient.

$$\vec{x}_0 = (x_0, y_0)$$
$$\vec{x}_{n+1} = \vec{x}_n - \eta \nabla f(\vec{x}_n) \quad (1.12)$$

Quá trình cập nhật được lặp lại cho đến khi f đạt giá trị cực tiểu. Bằng cách thay đổi theo hướng ngược lại với hướng của vector Gradient, hàm f sẽ tiến dần đến điểm cực tiểu. η là tỷ lệ cập nhật và quyết định về độ lớn cho mỗi lần cập nhật. Tỷ lệ cập nhật hay trong một số trường hợp còn được xem như là tốc độ cập nhật. Tỷ lệ cập nhật quá nhỏ có thể làm cho quá trình đạt đến điểm cực tiểu (còn gọi là quá trình hội tụ) kéo dài hơn. Ngược lại, tốc độ cập nhật lớn sẽ làm cho hàm số vượt qua giá trị cực tiểu và trở nên dao động tại vị trí cực tiểu.

1.3.4 Gradient Descent và bài toán hồi quy tuyến tính

Hồi quy tuyến tính (linear regression) là một giải thuật phổ biến trong học máy và được sử dụng trong các bài toán dự đoán. Kết quả dự đoán của bài toán hồi quy tuyến tính dựa vào giá trị của tập trọng số đạt được trong quá trình huấn luyện. Các bài toán hồi quy tuyến tính được xem như là tổ hợp tuyến tính của các ngõ vào. Nó cũng được xem như là ngõ ra của các đơn vị nơ-ron tuyến tính. Khác với một đơn vị nơ-ron trong các mạng học sâu, ngõ ra của hồi quy tuyến tính không sử dụng hàm kích hoạt phi tuyến nào. Về cơ bản, chúng ta có thể xem bài toán hồi quy tuyến tính là một dạng của mạng nơ-ron.

Bài toán hồi quy tuyến tính đơn giản bao gồm một ngõ ra y và một ngõ vào x được thể hiện thông qua mối quan hệ tuyến tính như sau:

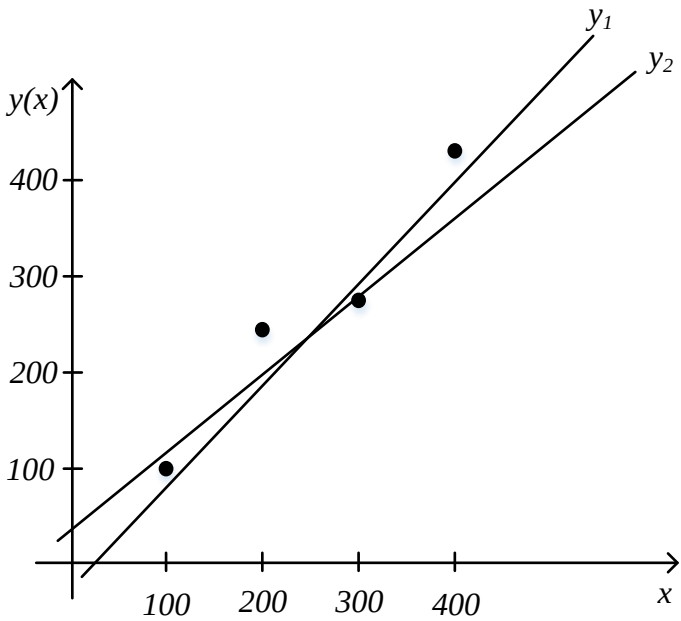
$$y = ax + b \quad (1.13)$$

Trong đó, chúng ta biết rằng y là đường thẳng được xác định bởi độ dốc a và giá trị độ lệch b. Một ví dụ cơ bản cho bài toán hồi quy tuyến tính đơn biến là bài toán dự đoán giá nhà dựa vào kích thước nhà. Trong đó, các số liệu đã có thể hiện mối quan hệ tương đối giữa giá nhà và kích thước nhà được thể hiện trong Bảng 1.1.

Bảng 1.1: Ví dụ của bài toán hồi quy tuyến tính - Giá nhà và kích thước nhà.

Kích thước (m^2)	Giá (triệu)
100	100
200	250
300	290
400	420

Trong bảng trên, chúng ta thấy mối quan hệ của giá nhà và kích thước nhà được thể hiện thông qua 4 tập số liệu được thu thập. Mỗi tập số liệu bao gồm một kích thước nhà với một giá nhà tương ứng. Sau khi có các số liệu như trong Bảng 1.1, chúng ta cần tìm mối quan hệ của giá nhà và kích thước thể hiện thông qua một hàm số và từ đó có thể dự đoán giá của một nhà khi chúng ta biết được kích thước của nó. Bài toán hồi quy tuyến tính đơn giản trong ví dụ này là chúng ta xem kích thước nhà như một biến x, giá là hàm số y. Chúng ta tìm mối quan hệ của x và y, thể hiện thông qua mối quan hệ tuyến tính $y = ax + b$. Trong bài toán hồi quy tuyến tính, chúng ta đi tìm giá trị a và b sao cho hàm số y mô tả được mối quan hệ giữa kích thước và giá nhà. Các đại lượng a và b tìm được thông qua các giá trị được cho trước trong Bảng 1.1.



Hình 1.13: Bài toán hồi quy tuyến tính đơn biến.

Các đường thẳng biểu diễn mối quan hệ của x và y có thể được tìm ra như trong Hình 1.13. Trong trường hợp này chúng ta giả sử có thể tồn tại đường thẳng y_1 và y_2. Chúng ta thấy mỗi đường sẽ cho mức độ chính xác khác nhau. Trong quá trình tối ưu cho bài toán hồi quy tuyến tính, chúng ta lựa chọn đường thẳng sao cho tổng sai số là nhỏ nhất. Sau khi lựa chọn được đường thẳng tối ưu, chúng ta có thể dự đoán giá trị của y

cho một giá trị bất kỳ của x. Trong các bài toán hồi quy tuyến tính, khi các thông số cho kết quả sai số nhỏ nhất đối với tập dữ liệu cho trước, nó được kỳ vọng là sẽ cho kết quả dự đoán chính xác nhất đối với một giá trị mới. Tuy nhiên điều này không phải luôn đúng. Một số mô hình sau khi tối ưu cho kết quả sai số rất thấp đối với tập dữ liệu đã cho nhưng đối với tập dữ liệu mới nó lại không đáp ứng được. Vấn đề này sẽ được cải tiến thông qua một số phương pháp được sử dụng trong kỹ thuật huấn luyện mạng nơ-ron nhiều lớp.

Trong bài toán trên chúng ta chỉ có một biến x. Trong đa phần các trường hợp chúng ta bắt gặp các bài toán hồi quy đa biến, trong đó hàm số f là một tổ hợp tuyến tính của nhiều biến ngõ vào, ví dụ:

$$f(x, y, z) = w_1 x + w_2 y + w_3 z \tag{1.14}$$

Với x, y, z là các biến ngõ vào và w_1, w_2, w_3 là các trọng số tương ứng. Trong mô hình hồi quy tuyến tính đa biến, chúng ta cũng thực hiện phương pháp tối ưu tương tự hồi quy đơn biến. Quá trình tối ưu thông qua tìm kiếm các giá trị trọng số sao cho hàm số tạo ra giá trị ngõ ra mong muốn.

Trong mô hình hồi quy tuyến tính, các thông số được lựa chọn sao cho mô hình có khả năng trả về các giá trị khớp với giá trị mong muốn trong tập dữ liệu được dùng để tối ưu hóa mô hình. Để đánh giá mức độ chính xác của mô hình, chúng ta tính sai số giữa ngõ ra dự đoán và ngõ ra mong muốn. Chúng ta định nghĩa ngõ ra thực hay còn gọi là ngõ ra dự đoán là kết quả đáp ứng của mô hình đối với một dữ liệu ngõ vào. Chúng ta cũng định nghĩa ngõ ra mong muốn là ngõ ra được kỳ vọng là chính xác nhất đối với từng dữ liệu ngõ vào. Một số từ ngữ được dùng để chỉ ngõ ra mong muốn như nhãn (label) hay đích đến (target).

Xét mô hình hồi quy tuyến tính đơn biến ở ví dụ trước, $y = ax + b$. Tập dữ liệu là các cặp (x, y) được dùng để lựa chọn giá trị cho a và b sao cho mô hình có thể đáp ứng tốt nhất đối với tất cả các cặp (x, y) được dùng để huấn luyện mô hình. Sai số giữa ngõ ra thực tế và ngõ ra mong muốn trên một tập dữ liệu được xác định thông qua hàm sai số. Hàm sai số được định nghĩa là giá trị sai số giữa ngõ ra thực và ngõ ra mong muốn được xét trên một tập dữ liệu cụ thể. Trong nhiều mô hình, để đánh giá sai số này chúng ta còn dùng khái niệm hàm mất mát (loss function) hoặc hàm chi phí (cost function). Mặc dù các hàm được dùng với các tên khác nhau và cũng có một số sự khác nhau giữa các hàm, tuy nhiên về ý nghĩa các hàm này đều cho phép ước lượng mức độ chính xác của mô hình từ phương pháp so sánh giá trị dự đoán

và giá trị mong muốn. Phương pháp tính sai số cơ bản là sử dụng sai số bình phương trung bình (Mean Squared Error - MSE). MSE đo giá trị sai số giữa ngõ ra dự đoán và ngõ ra mong muốn của mô hình trên toàn bộ tập dữ liệu.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b))^2 \tag{1.14}$$

Trong đó y_i là giá trị mong muốn, $(ax_i + b)$ là giá trị ngõ ra dự đoán của mô hình đối với ngõ vào x_i, n là tổng số mẫu.

Hàm sai số (1.14) cho biết giá trị sai lệch giữa giá trị thực và giá trị dự đoán từ mô hình hồi quy tuyến tính. Quá trình tối ưu là quá trình đi tìm các thông số mô hình sao cho giá trị hàm sai số là nhỏ nhất. Trong bài toán tối ưu cho mô hình hồi quy tuyến tính, chúng ta sử dụng phương pháp Gradient Descent đã giới thiệu ở phần trước để thay đổi giá trị các trọng số sao cho hàm sai số tiến về điểm cực tiểu. Để cập nhật giá trị của a và b sao cho hàm sai số tiến dần về điểm cực tiểu, chúng ta dựa vào các cặp giá trị cho trước (x_i, y_i). Các cặp giá trị này còn được gọi là tập dữ liệu huấn luyện cho mô hình. Phương pháp huấn luyện ở đây là huấn luyện có giám sát như đã giới thiệu ở phần trước. Trong phương pháp này, chúng ta có giá trị ngõ vào x_i và chúng ta có giá trị mong muốn tương ứng cho ngõ vào x_i là y_i. Như vậy, dựa vào sai số cho mỗi cặp (x_i, y_i), chúng ta điều chỉnh a, b theo chiều ngược với chiều của vector Gradient để hàm sai số đạt đến điểm cực tiểu nhanh nhất. Với mỗi cặp giá trị (x_i, y_i), chúng ta cập nhật a và b dựa theo phương pháp Gradient Descent như sau:

$$\begin{gathered} C = \frac{1}{n}\sum_{i=1}^{n}(y_i - (ax_i + b))^2 \\ [a, b] \leftarrow [a, b] - \eta\nabla C \\ \nabla C = \left[\frac{\partial C}{\partial a}, \frac{\partial C}{\partial b}\right] \end{gathered} \tag{1.15}$$

Trong đó η là tốc độ cập nhật. C là hàm sai số sử dụng sai số bình phương trung bình.

Từ phương trình (1.15) chúng ta có thể tính giá trị cập nhật a, b đối với n cặp giá trị (x_i, y_i) để hàm sai số tiến về giá trị cực tiểu nhanh nhất dựa vào phương pháp Gradient Descent.

$$\frac{\partial C}{\partial a} = -\frac{2}{n}\sum_{i=1}^{n}\big(y_i - (ax_i + b)\big)x_i$$

$$\frac{\partial C}{\partial b} = -\frac{2}{n}\sum_{i=1}^{n}\big(y_i - (ax_i + b)\big)$$

$$a \leftarrow a + \eta\frac{2}{n}\sum_{i=1}^{n}\big(y_i - (ax_i + b)\big)x_i \qquad (1.16)$$

$$b \leftarrow b + \eta\frac{2}{n}\sum_{i=1}^{n}\big(y_i - (ax_i + b)\big)$$

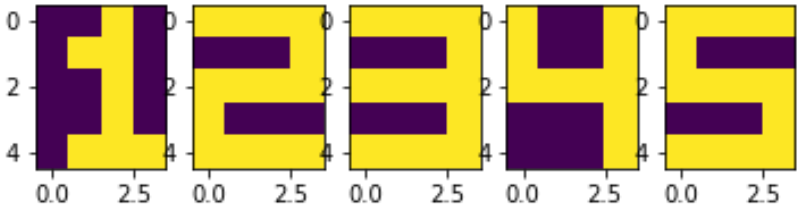
Quá trình huấn luyện được lặp lại cho đến khi hàm sai số đạt giá trị nhỏ mong muốn. Công thức (1.16) là công thức cập nhật giá trị cho các trọng số theo phương pháp giảm dần theo độ dốc cho bài toán hồi quy tuyến tính.

Chương 2
HUẤN LUYỆN MẠNG NƠ-RON

2.1 HUẤN LUYỆN MẠNG NƠ-RON VỚI GRADIENT DESCENT

Trong bài toán hồi quy tuyến tính, chúng ta đã định nghĩa hàm sai số để ước lượng mức độ sai số giữa ngõ ra thực và ngõ ra mong muốn. Trong mạng nơ-ron, để đánh giá mô hình chúng ta cũng tính toán mức độ khớp của ngõ ra dự đoán và ngõ ra mong muốn thông qua định nghĩa một số hàm sai số. Trong các mạng nơ-ron nhiều lớp, sai số được tính thông qua các hàm mất mát (loss function) hoặc hàm chi phí (cost function). Hàm mất mát hay làm chi phí đều cho phép ước lượng mức độ khớp của mô hình đối với tập dữ liệu được sử dụng để huấn luyện. Trong quyển sách này, chúng ta có thể gọi chung các hàm này là hàm sai số hoặc sử dụng các thuật ngữ tiếng Anh của chúng như loss function, cost function.

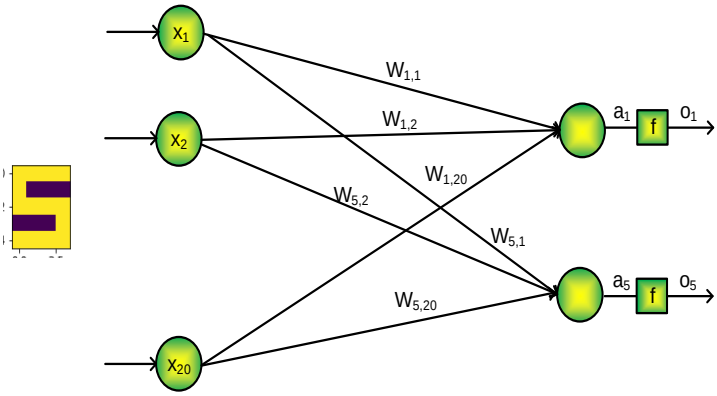
Gradient Descent là phương pháp hiệu quả được sử dụng để tối ưu hàm sai số trong mạng nơ-ron. Trong phần trước, chúng ta đã khảo sát ứng dụng của Gradient Descent để tối ưu hàm sai số của mô hình hồi quy tuyến tính. Phương pháp áp dụng Gradient Descent tối ưu hàm sai số của mạng nơ-ron tương tự như cách chúng ta làm với mô hình hồi quy tuyến tính. Có thể xem mỗi ngõ ra của nơ-ron là một mô hình hồi quy tuyến tính, tuy nhiên điểm khác biệt ở đây là đối với mỗi nơ-ron chúng sử dụng thêm các hàm kích hoạt phi tuyến (activation function) như đã trình bày trong Chương 1. Giả sử chúng ta khảo sát một mạng nơ-ron đơn giản được sử dụng để phân lớp hình ảnh 5 ký tự như trong Hình 2.1. Mỗi ký tự là một ảnh có kích thước 5×4. Mỗi ký tự được mã hóa dưới dạng 1 vector có kích thước là 20.



Hình 2.1: Ảnh cách ký tự sử dụng cho bài toán phân lớp đơn giản.

Mạng nơ-ron ứng dụng trong các bài toán phân lớp (classification) có nhiều ngõ ra, mỗi ngõ ra đại diện cho 1 lớp khác nhau. Trong ví dụ đầu tiên, chúng ta giả sử một mạng nơ-ron gồm có 5 nơ-ron

ở lớp ngõ ra cho phép phân lớp 5 ký tự được minh họa trong Hình 2.1. Mỗi nơ-ron có 20 ngõ vào và các nơ-ron kết nối chung các ngõ vào như Hình 2.2.



Hình 2.2: Ví dụ mạng nơ-ron đơn giản cho bài toán phân lớp.

Mỗi nơ-ron tích cực mức cao đối với 1 ngõ vào. Mỗi một mẫu huấn luyện sẽ bao gồm một vector ngõ vào và một vector ngõ ra. Vector ngõ ra được gọi là giá trị mong muốn và được mã hóa sử dụng mã one-hot. One-hot là kỹ thuật mã hóa các ngõ ra của mạng nơ-ron được sử dụng phổ biến trong các mạng nhiều lớp cho các bài toán phân lớp. Cụ thể, mã one-hot là một vector có độ dài bằng với số lượng nơ-ron ngõ ra. Mỗi vector one-hot ngõ ra tương ứng với một vector ngõ vào. Trong mỗi vector mã hóa one-hot, chỉ có một phần tử có giá trị 1 cho nơ-ron tích cực, các giá trị còn lại bằng 0 cho các nơ-ron không tích cực. Mã one-hot cho vector các ngõ ra được minh họa trong Bảng 2.1.

Một mạng nơ-ron đơn giản gồm có 5 nơ-ron ở lớp ngõ ra. Các nơ-ron tích cực ở mức cao cho lần lượt từng ảnh ngõ vào. Mạng nơ-ron được thiết kế như Hình 2.2. Như vậy, giá trị ngõ ra mong muốn (target) cho mỗi ngõ ra đối với các ngõ vào được thể hiện trong Bảng 2.1.

Bảng 2.1: Giá trị ngõ ra mong muốn.

	Ngõ ra được mã hóa one-hot				
Mẫu	O_1	O_2	O_3	O_4	O_5
1	1	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	1

Trong Bảng 2.1, chúng ta có các mẫu và các giá trị ngõ ra mong muốn tương ứng. Bảng 2.1 là tập dữ liệu huấn luyện được sử dụng trong quá trình tối ưu hàm sai số. Tương ứng với mỗi mẫu, chúng ta có giá trị ngõ vào là một vector ngõ vào và một vector giá trị ngõ ra mong muốn cho các nơ-ron ngõ ra. Tập trọng số được khởi tạo ngẫu nhiên nhỏ. Việc khởi tạo tập trọng số cũng có vai trò quyết định đến quá trình huấn luyện. Có nhiều phương pháp khởi tạo tập trọng số. Trong ví dụ này, chúng ta chỉ khởi tạo các giá trị ngẫu nhiên và nhỏ hơn 1. Lần lượt từng mẫu được đưa đến ngõ vào để tính giá trị ngõ ra dự đoán của cả 5 nơ-ron. Sai số sau đó được tính và các trọng số được cập nhật sao cho hàm sai số tiến dần về điểm cực tiểu. Giả sử trong trường hợp này, các nơ-ron đều sử dụng hàm Sigmoid làm hàm kích hoạt ngõ ra. Chúng ta xây dựng công thức cho quá trình cập nhật trọng số sử dụng phương pháp Gradient Descent. Từ Hình 2.2 chúng ta có thể tính giá trị ngõ ra của các nơ-ron:

$$a_j = \sum_{i=1}^{20} x_i w_{j,i} + b_j$$
$$o_j = \sigma(a_j) = \frac{1}{1 + e^{-a_j}} \tag{2.1}$$

Trong đó $w_{j,i}$ là trọng số của ngõ vào thứ i đến nơ-ron thứ j. σ là hàm Sigmoid.

Hàm sai số được tính trên tổng sai số các ngõ ra và đối với tổng các mẫu huấn luyện, sử dụng sai số bình phương trung bình, chúng ta có:

$$C = \frac{1}{5}\sum_{t=1}^{5}\sum_{k=1}^{5}(y_k^t - o_k^t)^2 \tag{2.2}$$

Trong đó y_k^t là giá trị mong muốn của ngõ ra thứ k và o_k^t là giá trị ngõ ra dự đoán của ngõ ra thứ k đối với mẫu thứ t trong tập huấn luyện. Công thức (2.2) tính tổng sai số của tất cả 5 nơ-ron ngõ ra đối với 5 mẫu. Các trọng số $w_{j,i}$ kết nối từ ngõ vào i đến nơ-ron thứ j được cập nhật theo phương pháp Gradient Descent.

$$w_{j,i} \leftarrow w_{j,i} - \eta \frac{\partial C}{\partial w_{j,i}} \tag{2.3}$$

Phương pháp Gradient Descent hướng đến việc cập nhật trọng số theo ngược chiều với chiều tăng cực đại của hàm số. Do đó, $w_{j,i}$ được cập nhật bằng cách tính Gradient của hàm sai số đối với $w_{j,i}$.

$$\frac{\partial C}{\partial w_{j,i}} = \frac{1}{5}\sum_{t=1}^{5}\sum_{k=1}^{5} 2.(y_k^t - o_k^t)\left(-\frac{\partial o_k^t}{\partial w_{j,i}}\right)$$

$$\frac{\partial C}{\partial w_{j,i}} = -\frac{2}{5}\sum_{t=1}^{5}\sum_{k=1}^{5}(y_k^t - o_k^t)\left(\frac{\partial o_k^t}{\partial a_k^t}\frac{\partial a_k^t}{\partial w_{j,i}}\right) \tag{2.4}$$

Chúng ta có $\frac{\partial a_k^t}{\partial w_{j,i}} = 0$, khi $k \neq j$. Phương trình (2.4) được viết lại như sau:

$$\frac{\partial C}{\partial w_{j,i}} = -\frac{2}{5}\sum_{t=1}^{5}\left(y_j^t - o_j^t\right)\left(\frac{\partial o_j^t}{\partial a_j^t}\frac{\partial a_j^t}{\partial w_{j,i}}\right) \tag{2.5}$$

Các đạo hàm trong (2.5) được tính như dựa vào hàm Sigmoid và công thức (2.1).

$$o_j^t = \sigma\left(a_j^t\right) = \frac{1}{1+e^{-a_j^t}}$$

$$\frac{\partial o_j^t}{\partial a_j^t} = \left(\frac{1}{\left(1+e^{-a_j^t}\right)^2}\right)e^{-a_j^t}$$

$$\frac{\partial o_j^t}{\partial a_j^t} = \left(\frac{1}{\left(1+e^{-a_j^t}\right)}\frac{e^{-a_j^t}}{\left(1+e^{-a_j^t}\right)}\right) \tag{2.6}$$

$$\frac{\partial o_j^t}{\partial a_j^t} = \left(\frac{1}{\left(1+e^{-a_j^t}\right)}\frac{1+e^{-a_j^t}-1}{\left(1+e^{-a_j^t}\right)}\right)$$

$$\frac{\partial o_j^t}{\partial a_j^t} = \left(\frac{1}{\left(1 + e^{-a_j^t}\right)} \left(1 - \frac{1}{\left(1 + e^{-a_j^t}\right)} \right) \right)$$

$$\frac{\partial o_j^t}{\partial a_j^t} = o_j^t (1 - o_j^t)$$

$$\frac{\partial a_j^t}{\partial w_{j,i}} = x_j^t$$

Trong đó x_j^t là ngõ vào thứ j của mẫu thứ t.

Từ (2.4), (2.5), (2.6), chúng ta có công thức cập nhật cho các trọng số.

$$\frac{\partial C}{\partial w_{j,i}} = -\frac{2}{5} \sum_{t=1}^{5} \left(y_j^t - o_j^t\right) o_j^t (1 - o_j^t) x_j^t$$

$$w_{j,i} \leftarrow w_{j,i} + \Delta w_{j,i} \tag{2.7}$$

$$\Delta w_{j,i} = \eta \frac{2}{5} \sum_{t=1}^{5} \left(y_j^t - o_j^t\right) o_j^t (1 - o_j^t) x_j^t$$

Các trọng số được cập nhật theo công thức (2.7). Giá trị bias được xem như là một trọng số đặc biệt cho các ngõ vào luôn có giá trị bằng 1. Do đó, các bias cũng được thiết lập các công thức cập nhật tương tự như các trọng số.

$$\frac{\partial C}{\partial b_j} = -\frac{2}{5} \sum_{t=1}^{5} \left(y_j^t - o_j^t\right) \left(\frac{\partial o_j^t}{\partial a_j^t} \frac{\partial a_j^t}{\partial b_j} \right)$$

$$\frac{\partial C}{\partial b_j} = -\frac{2}{5} \sum_{t=1}^{5} \left(y_j^t - o_j^t\right) o_j^t (1 - o_j^t) \tag{2.8}$$

$$b_j \leftarrow b_j + \Delta b_j$$

$$\Delta b_j = \eta \frac{2}{5} \sum_{t=1}^{5} \left(y_j^t - o_j^t\right) o_j^t (1 - o_j^t)$$

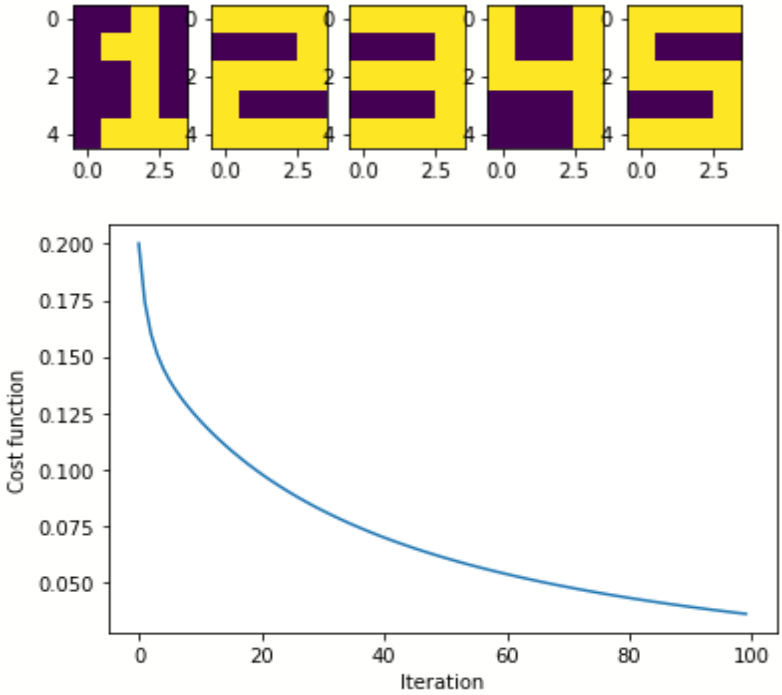
Các trọng số và bias được cập nhật theo công thức (2.7) và (2.8). Phương pháp Gradient tính toán giá trị cập nhật cho các trọng số dựa trên sai số khi tính cho tất cả các mẫu.

Chương trình huấn luyện cho 5 nơ-ron phân loại 5 mẫu được viết đơn giản sử dụng ngôn ngữ Python với thư viện numpy được minh họa bên dưới. Các mẫu ký tự được tạo ra bằng ma trận.

```
import numpy as np
import matplotlib.pyplot as plt
Char = np.matrix([[0,0,1,0,
                   0,1,1,0,
                   0,0,1,0,
                   0,0,1,0,
                   0,1,1,1],
                  [1,1,1,1,
                   0,0,0,1,
                   1,1,1,1,
                   1,0,0,0,
                   1,1,1,1],
                  [1,1,1,1,
                   0,0,0,1,
                   1,1,1,1,
                   0,0,0,1,
                   1,1,1,1],
                  [1,0,0,1,
                   1,0,0,1,
                   1,1,1,1,
                   0,0,0,1,
                   0,0,0,1],
                  [1,1,1,1,
                   1,0,0,0,
```

```
                    1,1,1,1,
                    0,0,0,1,
                    1,1,1,1]],dtype = np.int8)
Target = np.matrix([[1,0,0,0,0],
                      [0,1,0,0,0],
                      [0,0,1,0,0],
                      [0,0,0,1,0],
                      [0,0,0,0,1]])

plt.subplot(151)
imgplot = plt.imshow(Char[0,:].reshape(5,4))
plt.subplot(152)
imgplot = plt.imshow(Char[1,:].reshape(5,4))
plt.subplot(153)
imgplot = plt.imshow(Char[2,:].reshape(5,4))
plt.subplot(154)
imgplot = plt.imshow(Char[3,:].reshape(5,4))
plt.subplot(155)
imgplot = plt.imshow(Char[4,:].reshape(5,4))
plt.show()
def sigmoid(x):
  return 1./(1.+np.exp(-x))Cost = np.zeros(100)
W =  np.matrix(np.random.uniform(-0.1,0.1,(5,20)))
n=0.1 # learning rate
for j in range(100):
  dW = np.zeros_like(W)
  for i in range (5):
    X = Char[i,:]
    t= Target[i,:]
    a=np.dot(X,W.transpose())
    o=sigmoid(a)
    d = np.multiply(np.multiply((t-o),o),(1-o))
    dW += d.T*X
  #update
  W=W+n*dW
```

```
  #Check the output, calculate cost
  A= sigmoid(Char*W.T)
  Cost[j]=np.mean(np.power((Target-A),2))
plt.plot(Cost,)
plt.ylabel('Cost function')
plt.xlabel('Iteration')
plt.show()
```



Hình 2.3: Đồ thị hàm chi phí qua các lần lặp.

Trong chương trình trên, chúng ta thực hiện tính toán giá trị cập nhật trên từng mẫu và tổng chúng lại, sau đó thực hiện cập nhật 1 lần khi kết thúc tính toán cho 5 mẫu. Các giá trị Gradient và giá trị cập nhật cho trọng số được tính thông qua các phép nhân ma trận. Để hiểu hơn chương trình, người đọc cần xem lại lý thuyết về ma trận và vector.

Một cách tổng quát, giả sử tập huấn luyện bao gồm n mẫu, mạng nơ-ron bao gồm m nơ-ron ở lớp ngõ ra. Hàm mất mát được định nghĩa:

$$C = \frac{1}{n}\sum_{t=1}^{n}\sum_{k=1}^{m}(y_k^t - o_k^t)^2 \qquad (2.9)$$

Trong một số trường hợp, để khi tính đạo hàm, giá trị 2 được tối giản người ta thêm 2 ở vị trí mẫu như sau:

$$C = \frac{1}{2n}\sum_{t=1}^{n}\sum_{k=1}^{m}(y_k^t - o_k^t)^2 \tag{2.10}$$

Việc thêm 2 ở mẫu số không ảnh hưởng khi chúng ta cực tiểu hóa hàm mất mát. Các trọng số được cập nhật sử dụng giải thuật giảm dần theo độ dốc:

$$\frac{\partial C}{\partial w_{j,i}} = -\frac{1}{n}\sum_{t=1}^{n}\left(y_j^t - o_j^t\right)o_j^t(1 - o_j^t)x_j^t$$

$$w_{j,i} \leftarrow w_{j,i} + \Delta w_{j,i} \tag{2.11}$$

$$\Delta w_{j,i} = \eta\frac{1}{n}\sum_{t=1}^{n}\left(y_j^t - o_j^t\right)o_j^t(1 - o_j^t)x_j^t$$

2.2 BATCH GRADIENT DESCENT VÀ STOCHASTIC GRADIENT DESCENT.

Phương pháp giảm dần theo độ dốc (Gradient Descent (GD)) được sử dụng phổ biến trong các mạng nơ-ron nhiều lớp để tối ưu hàm mất mát. Việc cập nhật các trọng số sử dụng công thức (2.11) thực hiện 1 lần khi kết thúc quá trình tính toán Gradient cho toàn bộ các mẫu trong tập huấn luyện. Tính Gradient và giá trị cập nhật cho các trọng số đối với cả tập mẫu được xem như phương pháp giảm dần theo độ dốc được thực thi trên toàn bộ mẫu (Batch Gradient Descent). Batch Gradient Descent sử dụng bộ nhớ lớn để lưu trữ giá trị Gradient tính được khi số lượng mẫu trong tập dữ liệu lớn. Hơn nữa quá trình huấn luyện có thể lâu hơn do quá trình cập nhật chỉ được thực hiện sau khi tính toán Gradient cho tất cả các mẫu.

Phương pháp giảm dần theo độ dốc ngẫu nhiên (Stochastic Gradient Descent - SGD) thực hiện quá trình tính toán và cập nhật cho từng mẫu trong tập huấn luyện và các mẫu này được lựa chọn ngẫu nhiên. Stochastic Gradient Descent giúp cho quá trình huấn luyện hội tụ nhanh hơn (khái niệm hội tụ được sử dụng để chỉ quá trình hàm mất mát tiến dần về điểm cực tiểu). Tuy nhiên, giá trị hàm sai số có thể không đạt giá trị cực tiểu như phương pháp Batch Gradient Descent. Hàm mất mát

và công thức cập nhật cho các trọng số được áp dụng cho từng mẫu trong tập dữ liệu như sau:

$$C = \sum_{k=1}^{m} (y_k^t - o_k^t)^2$$
$$w_{j,i} \leftarrow w_{j,i} + \Delta w_{j,i} \tag{2.12}$$
$$\Delta w_{j,i} = \eta\left(y_j^t - o_j^t\right) o_j^t (1 - o_j^t) x_j^t$$

Phương pháp được sử dụng phổ biến trong các mạng nhiều lớp hiện nay là thực hiện cập nhật trên các nhóm nhỏ (Mini-batch Stochastic Gradient Descent) thay vì cập nhật trên toàn bộ tập dữ liệu hoặc cập nhật trên từng mẫu dữ liệu. Mini-batch Stochastic Gradient Descent khắc phục được một số nhược điểm của Batch Gradient Descent và Stochastic Gradient Descent. Các nhóm được lựa chọn ngẫu nhiên trong tập dữ liệu. Hàm mất mát và công thức cho quá trình cập nhật với nhóm nhỏ được minh họa như công thức (2.13).

$$C = \frac{1}{2n} \sum_{t=1}^{n} \sum_{k=1}^{m} (y_k^t - o_k^t)^2$$
$$w_{j,i} \leftarrow w_{j,i} + \Delta w_{j,i} \tag{2.13}$$
$$\Delta w_{j,i} = \eta \frac{1}{n} \sum_{t=1}^{b} \left(y_j^t - o_j^t\right) o_j^t (1 - o_j^t) x_j^t$$

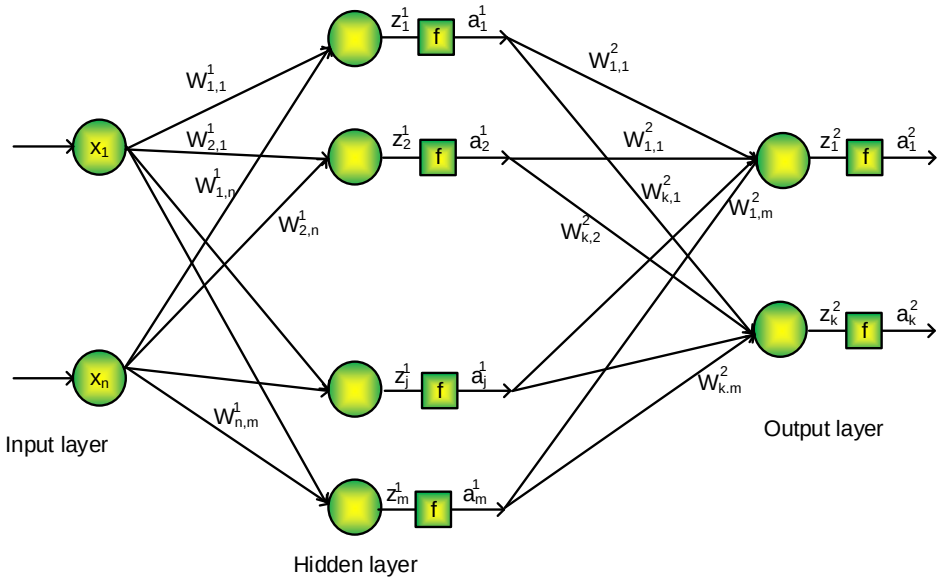
Trong đó b là kích thước của nhóm nhỏ ($b < n$).

2.3 PHƯƠNG PHÁP LAN TRUYỀN NGƯỢC SAI SỐ (BACKPROPAGATION)

Trong các mạng nơ-ron nhiều lớp (multi-layer neural network) và các mạng sâu (deep neural network), hàm mất mát được tính cho các nơ-ron ở lớp ngõ ra thông qua các phép ước lượng sai số giữa ngõ ra dự đoán và ngõ ra mong muốn. Cập nhật các trọng số cho các nơ-ron ở lớp ngõ ra được thực hiện thông qua Gradient Descent với hàm mất mát như đã được trình bày ở phần trước. Đối với các lớp ẩn, chúng ta không có giá trị ngõ ra mong muốn cho các nơ-ron ở lớp ẩn để xác định hàm mất mát cho các nơ-ron lớp ẩn. Sai số của mỗi nơ-ron ở lớp ẩn ảnh hưởng đến sai số của tất các các nơ-ron ở lớp sau đó mà nó kết nối đến. Do đó sai số của các nơ-ron ở lớp ẩn được tính bằng cách lan truyền ngược sai số từ các nơ-ron ở lớp sau mà nó kết nối

đến. Giải thuật lan truyền ngược sai số (backpropagation) được giới thiệu vào những năm 1970 nhưng chỉ được biết đến và áp dụng sau khi 3 tác giả David Rumelhart, Geo ffrey Hinton và Ronald Williams công bố trong bài báo khoa học "Learning representations by back-propagation" vào năm 1986.

Một mạng nơ-ron nhiều lớp được minh họa trong Hình 2.4. Mỗi nơ-ron trong một lớp bao gồm các kết nối từ ngõ ra của các nơ-ron lớp trước đó và một bias. Tuy nhiên để đơn giản, chúng ta có thể xem bias như một trọng số đặc biệt ứng với ngõ vào luôn có giá trị 1.



Hình 2.4: Mô hình mạng nơ-ron với 1 lớp ẩn và một lớp ngõ ra.

Trong Hình 2.4, chúng ta ký hiệu $w_{j,k}^{l}$ là trọng số kết nối ngõ vào thứ k của nơ-ron thứ j trong lớp thứ l. Hay nói khác, $w_{j,k}^{l}$ là trọng số của kết nối từ ngõ ra của nơ-ron thứ k trong lớp thứ $(l-1)$ đến nơ-ron thứ j trong lớp thứ l. a_j^l là ngõ ra của nơ-ron thứ j trong lớp thứ l, b_j^l là bias cho nơ-ron thứ j trong lớp thứ l. K là số lượng nơ-ron trong lớp $(l-1)$.

$$a_j^l = \sigma(z_j^l) = \sigma\left(\sum_{k=1}^{K} w_{j,k}^{l} a_k^{l-1} + b_j^l\right) \tag{2.14}$$

Trong đó z_j^l là tổ hợp tuyến tính của các ngõ vào với trọng số.

$$z_j^l = \sum_{k=1}^{K} w_{j,k}^{l} a_k^{l-1} + b_j^l \tag{2.15}$$

Hàm mất mát được tính cho các nơ-ron ở lớp ngõ ra, giả sử l là lớp cuối cùng trong mạng nơ-ron.

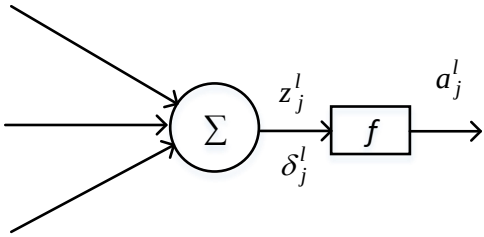
$$C = \frac{1}{2n}\sum_{i=1}^{n}\left(y(x_i) - a^l(x_i)\right)^2 \tag{2.16}$$

Trong đó n là số lượng mẫu được sử dụng để huấn luyện mạng, x_i là vector ngõ vào của mẫu thứ i, $y(x_i)$ là ngõ ra mong muốn đối với ngõ vào x_i. $a^l(x_i)$ là giá trị ngõ ra dự đoán của các nơ-ron ở lớp ngõ ra đối với ngõ vào của mẫu thứ i. Phương pháp tính sai số ở đây dựa vào giá trị ngõ ra của các nơ-ron ở lớp cuối cùng so với giá trị mong muốn của chúng. Để đơn giản cho quá trình tính toán, chúng ta có thể tính giá trị sai số đối với từng mẫu huấn luyện riêng lẻ. Hàm sai số tổng thực chất là giá trị trung bình của sai số trên tất cả các mẫu huấn luyện. Hàm sai số cho nơ-ron thứ j ở lớp ngõ ra như sau:

$$C = \frac{1}{2}\left(y_j - a_j^l\right)^2 \tag{2.17}$$

Trong phương pháp lan truyền ngược chúng ta lan truyền giá trị lỗi từ lớp cuối cùng về các lớp trước đó. Quá trình lan truyền lỗi và tính toán có thể thực hiện đối với từng mẫu riêng lẻ. Cập nhật trọng số có thể được thực hiện theo từng mẫu riêng lẻ hoặc theo nhóm hoặc cho cả tập dữ liệu.

Giả sử lớp ngõ ra là lớp thứ l, sai số được tính tại ngõ ra của nơ-ron lớp cuối cùng. Chúng ta sẽ tính sai số lan truyền ngược từ lớp ngõ ra về các lớp trước đó.

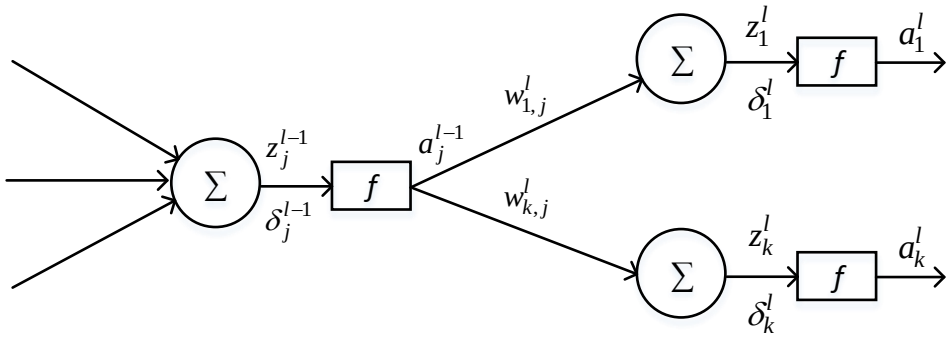


Hình 2.5: Nơ-ron ở lớp ngõ ra.

Đặt δ_j^l là sai số ngõ ra của nơ-ron thứ j ở lớp ngõ ra thứ l, chúng ta có thể tính δ_j^l thông qua hàm sai số ngõ ra như sau. Hàm kích hoạt được sử dụng là hàm Sigmoid.

$$\delta_j^l = -\frac{\partial C}{\partial z_j^l} = a_j^l\left(1 - a_j^l\right)\left(y_j - a_j^l\right) \tag{2.18}$$

Trong đó $a_j^l(1 - a_j^l)$ thu được từ đạo hàm của hàm Sigmoid như đã giới thiệu ở chương trước.



Hình 2.6: Nơ-ron lớp ẩn kết nối đến các nơ-ron ở lớp ngõ ra.

Tại lớp thứ $(l - 1)$ nơ-ron thứ j kết nối đến tất cả các nơ-ron ở lớp ngõ ra nên sự thay đổi của sai số của nơ-ron thứ j trong lớp $l - 1$ ảnh hưởng đến sai số ngõ ra của các nơ-ron ở lớp thứ l. Cụ thể giá trị ngõ ra của nơ-ron ở lớp thứ $l - 1$ ảnh hưởng đến sai số của tất các các nơ-ron ở lớp thứ l thông qua giá trị các trọng số kết nối từ nơ-ron thứ j trong lớp thứ $l - 1$ đến các nơ-ron trong lớp l. Do đó, sai số của nơ-ron thứ j trong lớp $l - 1$ được tính bằng tổng các sai số của các nơ-ron ở lớp l lan truyền ngược về nơ-ron thứ j trong lớp $l - 1$ thông qua các trọng số kết nối.

$$\delta_j^{l-1} = a_j^{l-1}\left(1 - a_j^{l-1}\right)\sum_{k=1}^{K} w_{k,j}^l\, \delta_k^l \tag{2.19}$$

Công thức (2.19) cho phép tính giá trị sai số tại các nơ-ron ở lớp ẩn bằng tổng các giá trị sai số lan truyền ngược về từ các nơ-ron ở lớp sau mà nó kết nối đến thông qua các trọng số tương ứng. Các sai số δ_j^l được tính tại ngõ ra các nơ-ron trước khi qua hàm kích hoạt nên chúng ta thấy có thành phần $a_j^{l-1}\left(1 - a_j^{l-1}\right)$ xuất hiện trong (2.19). Trong công thức (2.19), K là tổng số nơ-ron ở lớp ngõ ra. Chúng ta cập nhật các trọng số của nơ-ron ở lớp ẩn $l - 1$ sử dụng Gradient Descent sau khi xác định được giá trị sai số tại các nơ-ron lớp ẩn.

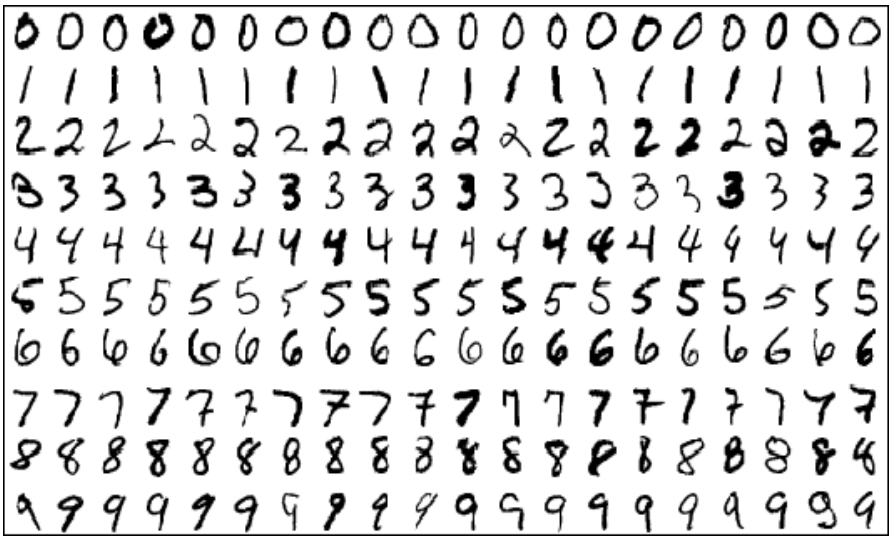
$$w_{j,i}^l = w_{j,i}^l + \eta\delta_j^l x_{j,i} \tag{2.20}$$

Trong công thức (2.20), chúng ta thấy trong phương pháp cập nhật có dấu "+" thay vì dấu "–" như phương pháp Gradient Descent đã giới thiệu ở chương trước. Trong (2.18), chúng ta đã đặt $\delta_j^l = -\frac{\partial C}{\partial z_j^l}$, do đó, giá trị δ_j^l ngược chiều với Gradient của hàm mất mát. Vì vậy trong

(2.20) xuất hiện dấu “+” thay vì dấu “–” để chỉ chiều cập nhật ngược chiều tăng cực đại của hàm sai số theo phương pháp giảm dần theo độ dốc.

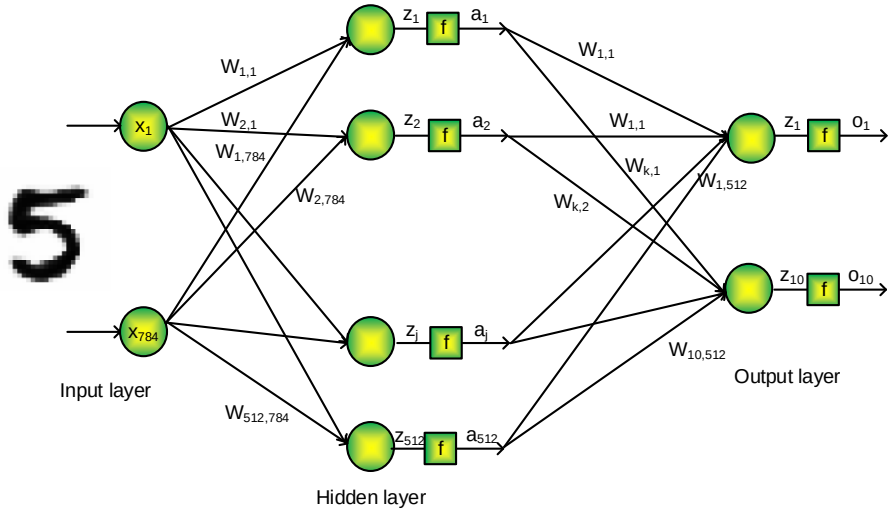
2.4 ỨNG DỤNG BACKPROPAGATION VÀ GRADIENT DESCENT CHO BÀI TOÁN NHẬN DẠNG KÝ TỰ VIẾT TAY.

MNIST (Modified National Institute of Standards and Technology) là một tập dữ liệu được sử dụng để huấn luyện cho các mạng nơ-ron trong các bài toán nhận dạng ký tự viết tay. Tập MNIST bao gồm 60,000 mẫu dùng cho huấn luyện (training sample) và 10,000 mẫu dùng cho việc kiểm thử mạng sau huấn luyện (testing sample). Mỗi mẫu là một ảnh 28×28 của một trong các ký tự từ 0 đến 9. Mỗi ảnh được chuyển sang một vector có độ dài 784. Các vector được ghép lại với nhau tạo thành một ma trận có kích thước 60,000×784 được dùng cho quá trình huấn luyện và tập kiểm thử có kích thước 10,000×784. Mỗi một ảnh có một nhãn để cho biết ảnh đó là của ký tự nào. Tương tự chúng ta có 60,000 nhãn cho tập huấn luyện và 10,000 nhãn cho tập kiểm thử. Các ký tự trong tập MNIST được minh họa trong Hình 2.7.



Hình 2.7: Tập các ký tự viết tay MNIST.

Tập dữ liệu MNIST có thể được tải trực tiếp khi thực thi mô hình thông qua các thư viện hỗ trợ hoặc có thể được lưu cục bộ dưới dạng tập tin “.csv”. Thiết kế mạng nơ-ron bao gồm 1 lớp ngõ vào có kích thước 784 tương ứng với số điểm ảnh của mỗi ảnh ký tự, 512 nơ-ron trong lớp ẩn và 10 nơ-ron lớp ngõ ra cho phép phân loại 10 ký tự viết tay. Mạng nơ-ron được minh họa trong Hình 2.8.



Hình 2.8: Mô hình mạng nơ-ron nhận dạng 10 ký tự viết tay.

Trong ví dụ này, chúng ta xây dựng và huấn luyện mô hình thông qua các mô hình toán đã phân tích trước đó. Các phép toán được thực hiện thông qua các phép tính ma trận. Trong trường hợp tập MNIST được lưu trên máy tính cục bộ dưới dạng tập tin ".csv", việc tải tập MNIST có thể được thực hiện thông qua chương trình như sau:

```
import numpy as np
data_path = "data/mnist/"
train_data=np.loadtxt(data_path                    +
"mnist_train.csv",
                        delimiter=",")
test_data =np.loadtxt(data_path + "mnist_test.csv",
                       delimiter=",")
x_train = np.asfarray(train_data[:, 1:])/255.0
x_test = np.asfarray(test_data[:, 1:])/255.0
y_train = np.asfarray(train_data[:, :1])
y_test = np.asfarray(test_data[:, :1])
```

Phương pháp đơn giản cho phép thử nghiệm các mô hình mạng nơ-ron với các tập dữ liệu có sẵn là tải trực tiếp thông qua các thư viện hỗ trợ. Ví dụ có thể tải thông qua thư viện Tensorflow hoặc Keras như sau:

```
import numpy as np
import tensorflow as tf
import datetime as dt
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)=
mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
```

Mỗi ảnh được lưu dưới dạng 1 vector với giá trị được hiệu chỉnh trong khoản $(0, 1)$. Ngõ ra số được mã hóa dưới dạng "one-hot" tương ứng cho 10 ngõ ra của nơ-ron. Tập MNIST được chia thành tập huấn luyện và tập kiểm thử. Trong tập huấn luyện chúng ta có ma trận x_train có kích thước 60,000×784 tương ứng cho 60,000 ảnh. Các ảnh được gán nhãn và được mã hóa trong ma trận y_train có kích thước 60,000×10. Tương tự như vậy, chúng ta cũng có 10,000 mẫu cho tập kiểm thử được lưu dưới dạng các ma trận.

Mỗi ảnh đầu vào được lưu dưới dạng một vector x, Wh là ma trận lưu trọng số kết nối của lớp ngõ vào đến lớp ẩn, Wo là ma trận lưu trọng số kết nối của lớp ẩn đến lớp ngõ ra, bh và bo là bias cho các nơ-ron lớp ẩn và lớp ngõ ra tương ứng.

$$
\begin{aligned}
x &= [x_1, x_2, x_3, x_4, \dots, x_{784}] \\
Wh &= \begin{bmatrix} w_{1,1} & \cdots & w_{1,784} \\ \vdots & \ddots & \vdots \\ w_{512,1} & \cdots & w_{512,784} \end{bmatrix} \\
bh &= [b_1, \dots, b_{512}] \\
Wo &= \begin{bmatrix} w_{1,1} & \cdots & w_{1,512} \\ \vdots & \ddots & \vdots \\ w_{10,1} & \cdots & w_{10,512} \end{bmatrix} \\
bo &= [b_1, \dots, b_{10}]
\end{aligned}
\tag{2.21}
$$

Giá trị ngõ ra đối với từng mẫu huấn luyện được đặt trong vector y, o là vector ngõ ra thực của các nơ-ron ở lớp ngõ ra. Chúng ta có thể tính giá trị ngõ ra của quá trình lan truyền thẳng đối với từng mẫu huấn luyện.

$$z_h = x.Wh^T + bh = [z_1, z_2, z_3, \dots, z_{512}]$$
$$a = \sigma(z_h)$$
$$z_o = a.Wo^T + bo = [z_1, z_2, z_3, \dots, z_{10}] \quad (2.22)$$
$$o = \sigma(z_o)$$

Vector sai số tại ngõ ra của các nơ-ron:

$$C = y - o = [c_1, c_2, c_3, \dots, c_{10}]$$

Trong đó: (2.23)

$$y = [y_1, y_2, y_3, \dots, y_{10}]$$

Sai số của các nơ-ron lớp ngõ ra đối với các nơ-ron sử dụng hàm Sigmoid:

$$d_o = o.(1 - o).(y - o) = [d_1, d_2, d_3, \dots, d_{10}] \quad (2.24)$$

Trong đó dấu (.) biểu diễn phép nhân từng phần tử (element-wise).

Tính giá trị cập nhật cho các trọng lớp số ngõ ra như sau:

$$\Delta Wo = {d_o}^T a = \begin{bmatrix} d_1 a_1 & \cdots & d_1 a_{512} \\ \vdots & \ddots & \vdots \\ d_{10} a_1 & \cdots & d_{10} a_{512} \end{bmatrix}$$
$$Wo \leftarrow Wo + \eta \Delta Wo \quad (2.25)$$
$$bo \leftarrow bo + \eta d_o$$

Sai số của các nơ-ron lớp ẩn thu được bằng cách lan truyền ngược sai số của các nơ-ron từ lớp ngõ ra về các nơ-ron trong lớp ẩn. Giá trị sai số của mỗi nơ-ron ở lớp ẩn được tính bằng tổng các sai số lan truyền từ các nơ-ron ở lớp ngõ ra thông qua các trọng số trong Wo. Chúng ta có sai số của nơ-ron thứ 1 của lớp ẩn được tính theo (2.26).

$$d_{h1} = a_1(1 - a_1)\left(d_1 w_{1,1} + d_2 w_{2,1} + d_3 w_{3,1} + \cdots + d_{10} w_{10,1}\right) \quad (2.26)$$

Tính giá trị sai số cho tất cả các nơ-ron ở lớp ẩn thông qua các phép toán ma trận như sau:

$$d_h = a.(1 - a).(d_o * Wo) = [d_{h1}, d_{h2}, \dots, d_{h512}] \quad (2.27)$$

Trong đó, phép nhân từng phần tử vector được biểu diễn bởi dấu (.) và phép nhân ma trận được biểu diễn bởi dấu (*).

Cuối cùng chúng ta cập nhật trọng số cho lớp ẩn. Các trọng số có thể được cập nhật sau mỗi mẫu huấn luyện với Stochastic Gradient Descent hoặc sau mỗi nhóm mẫu huấn luyện với Mini-batch Stochastic Gradient Descent hoặc sau khi tính toán cho tất cả các mẫu trong tập

huấn luyện (Batch Gradient Descent). Trong ví dụ này chúng ta cập nhật trọng số sau mỗi mẫu huấn luyện.

$$\begin{aligned} \Delta Wh &= {d_h}^T x \\ Wh &\leftarrow Wh + \eta \Delta Wh \\ bh &\leftarrow bh + \eta d_h \end{aligned} \tag{2.28}$$

Chương trình được viết bằng Python với thư viện numpy.

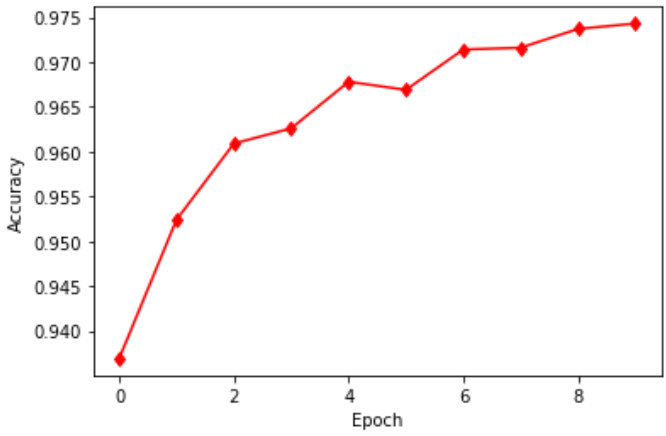
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
import datetime as dt
def sigmoid(x):
    return 1./(1.+np.exp(-x))
def AccTest(outN,labels):
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,Wh,bh,Wo,bo):
    OutH1=sigmoid(np.dot(samples,Wh.T)+bh)
    OutN=sigmoid(np.dot(OutH1,Wo.T)+bo)
    return OutN
# define network parameters
learningRate = 0.1
```

```
BatchSize= 100
Epoch=10
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
0.5,0.5,(NumHidden,NumInput)))
bh= np.random.uniform(0,0.5,(1,NumHidden))
dWh= np.zeros((NumHidden,NumInput))
dbh= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-0.5,0.5,(NumOutput,NumHidden))
bo= np.random.uniform(0,0.5,(1,NumOutput))
dWo= np.zeros((NumOutput,NumHidden))
dbo= np.zeros((1,NumOutput))
# Train the network with back propagation,SGD
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
Acc=np.zeros(Epoch)
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the training samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample):
        x=np.matrix(x_train[SampleIdx[i],:])
        y=np.matrix(y_train [SampleIdx[i],:])
          # Feedforward propagation
        a=sigmoid(np.dot(x,Wh.T)+bh)
        o =sigmoid(np.dot(a,Wo.T)+bo)
        do=np.multiply(np.multiply(o,(1-o)),(y-o))
        dWo=np.matrix(np.dot(do.T,a))
        dbo=np.mean(do,0)
```

```
        Wo=Wo + learningRate*dWo
        bo=bo + learningRate*dbo
        #back propagate error
        dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        dWh=np.dot(dh.T,x)
        dbh=np.mean(dh,0)
        # Update weights
        Wh=Wh + learningRate*dWh
        bh=bh + learningRate*dbh
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
#test the model
    RealOutN=feedforward(x_test,Wh,bh,Wo,bo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=t1=dt.datetime.now()
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

```
Load MNIST Dataset
----------------------------------
(60000, 784)
(60000, 10)
0:02:51.549678
Training epoch 0
Accuracy: 0.936900
0:03:00.683985
Training epoch 1
Accuracy: 0.952400
0:02:48.225638
```

```
Training epoch 2
Accuracy: 0.960900
0:02:50.006515
Training epoch 3
Accuracy: 0.962600
0:02:53.940777
Training epoch 4
Accuracy: 0.967800
0:02:57.980184
Training epoch 5
Accuracy: 0.966900
0:02:48.780461
Training epoch 6
Accuracy: 0.971400
0:02:50.924861
Training epoch 7
Accuracy: 0.971600
0:02:55.575684
Training epoch 8
Accuracy: 0.973700
0:02:49.130588
Training epoch 9
Accuracy: 0.974300
Total time :  0:28:51.365144
```



Ngõ ra các nơ-ron sử dụng hàm Sigmoid tạo ra giá trị không tuyến tính từ 0 đến 1. Các nơ-ron ngõ ra sử dụng hàm Sigmoid không thể hiện mối tương quan lẫn nhau. Trong hầu hết các bài toán phân lớp các ngõ ra được biểu diễn thông qua phân bố xác suất sử dụng hàm Softmax thay cho hàm Sigmoid. Hàm Softmax chuyển đổi giá trị các ngõ ra sang phân bố xác suất. Ngõ ra sau hàm Softmax của 1 nơ-ron được tính theo công thức sau:

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.29}$$

Trong đó K là số lượng các ngõ ra.

Khi các ngõ ra là phân bố xác suất, để tính mức độ tương quan của phân bố xác suất ngõ ra so với giá trị mong muốn, chúng ta có thể sử dụng phương pháp cross-entropy thay cho phương pháp sai số bình phương trung bình. Giả sử phân bố xác suất ngõ ra các nơ-ron được thể hiện dưới dạng $p = \{p_1, p_2, p_3, \dots, p_n\}$, ngõ ra mong muốn được thể hiện bởi vector $q = \{q_1, q_2, q_3, \dots, q_n\}$, hàm sai số sử dụng cross-entropy được định nghĩa như sau:

$$H(p, q) = -\sum_{i=1}^{n} p_i \log q_i \tag{2.30}$$

Trong ví dụ trên, chúng ta sử dụng phương pháp Stochastic Gradient Descent, cập nhật trọng số cho mỗi mẫu huấn luyện ngẫu nhiên. Phương pháp Stochastic Gradient Descent thực hiện trên từng mẫu huấn luyện làm cho quá trình tiến đến điểm cực tiểu của hàm sai số lâu hơn. Phương pháp Gradient Descent ngược lại cập nhật trọng số sau khi thực hiện tính toán cho toàn bộ mẫu trong tập huấn luyện. Gradient Descent đòi hỏi nhiều bộ nhớ lớn để cộng dồn và lưu các giá trị cập nhật cho các mẫu trong tập huấn luyện. Phương pháp hiệu quả có thể giải quyết nhược điểm của cả 2 phương pháp là cập nhật trọng số đối với từng nhóm nhỏ mẫu (mini-batch Stochastic Gradient Descent). Mini-batch Stochastic Gradient Descent cho phép quá trình hội tụ nhanh hơn đồng thời giảm thiểu việc hội tụ tại các điểm cực tiểu cục bộ. Sử dụng các phép toán ma trận cho phép tính toán giá trị cập nhật trên mỗi nhóm nhỏ dễ dàng hơn. Giả sử chúng ta thiết kế cập nhật cho mỗi nhóm 100 mẫu. Ngõ vào x được định nghĩa là một ma trận 2 chiều có thể chứa dữ liệu cho 100 mẫu:

$$x = \begin{pmatrix} x_1^1 & \cdots & x_{784}^1 \\ \vdots & \ddots & \vdots \\ x_1^{100} & \cdots & x_{784}^{100} \end{pmatrix} \tag{2.31}$$

Trong đó $x_1^1 \dots x_{784}^1$ và $x_1^{100} \dots x_{784}^{100}$ là ngõ vào cho mẫu thứ 1 và mẫu thứ 100 trong nhóm.

Ma trận trọng số lớp ẩn và lớp ngõ ra được định nghĩa tương tự ví dụ trước.

$$\begin{aligned} Wh &= \begin{bmatrix} w_{1,1} & \cdots & w_{1,784} \\ \vdots & \ddots & \vdots \\ w_{512,1} & \cdots & w_{512,784} \end{bmatrix} \\ bh &= [b_1, \dots, b_{512}] \\ Wo &= \begin{bmatrix} w_{1,1} & \cdots & w_{1,512} \\ \vdots & \ddots & \vdots \\ w_{10,1} & \cdots & w_{10,512} \end{bmatrix} \\ bo &= [b_1, \dots, b_{10}] \end{aligned} \tag{2.32}$$

Ngõ ra các nơ-ron lớp ẩn được tính 1 lần cho 100 mẫu:

$$z_h = x.Wh^T = \begin{bmatrix} zh_1^1 & \cdots & zh_{512}^1 \\ \vdots & \ddots & \vdots \\ zh_1^{100} & \cdots & zh_{512}^{100} \end{bmatrix} \tag{2.33}$$

Trong đó zh_1^1là ngõ ra nơ-ron thứ 1 trong lớp ẩn đối với mẫu huấn luyện thứ 1. zh_1^{100} là ngõ ra nơ-ron thứ 1 trong lớp ẩn đối với mẫu huấn luyện thứ 100.

Tương tự chúng ta tính ngõ ra cho 10 nơ-ron đối với 100 mẫu trong nhóm:

$$o = a.Wo^T = \begin{bmatrix} o_1^1 & \cdots & o_{10}^1 \\ \vdots & \ddots & \vdots \\ o_1^{100} & \cdots & o_{10}^{100} \end{bmatrix} \tag{2.34}$$

Trong đó $a = \sigma(z_h)$.

Hàm sai số sử dụng cross-entropy, ma trận giá trị sai số cho 10 nơ-ron ngõ ra với 100 mẫu:

$$d_o = y - o = \begin{bmatrix} d_{o1}^1 & \cdots & d_{o10}^1 \\ \vdots & \ddots & \vdots \\ d_{o1}^{100} & \cdots & d_{o10}^{100} \end{bmatrix} \tag{2.35}$$

Tính giá trị cập nhật cho trọng số lớp ngõ ra đối với 100 mẫu:

$$\Delta Wo = \frac{1}{100} d_o^T.a = \frac{1}{100} \begin{bmatrix} \Delta w_{1,1} & \cdots & \Delta w_{1,512} \\ \vdots & \ddots & \vdots \\ \Delta w_{10,1} & \cdots & \Delta w_{10,512} \end{bmatrix} \tag{2.36}$$

Trong đó:

$$\Delta w_{i,j} = \frac{1}{100}\left(d_{oi}^{1}a_{j}^{1} + d_{oi}^{2}a_{j}^{2} + \cdots + d_{oi}^{100}a_{j}^{100}\right)$$

Như vậy phép nhân ma trận cho phép cộng dồn các giá trị cập nhật cho 100 mẫu, do đó bằng cách lấy trung bình 100 mẫu ra có giá trị cập nhật cho trọng số. Sai số ngõ ra sau đó được lan truyền ngược về các nơ-ron ở lớp ẩn để tính giá trị cập nhật cho các nơ-ron ở lớp ẩn. Chương trình huấn luyện mạng nơ-ron sử dụng Mini-batch Stochastic Gradient Descent được viết lại như sau:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
import datetime as dt
def sigmoid(x):
    return 1./(1.+np.exp(-x))
def softmax(x):
    return np.divide(np.matrix(np.exp(x)),np.mat(np.sum(
np.exp(x),axis=1)))
def AccTest(outN,labels):
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,Wh,bh,Wo,bo):
```

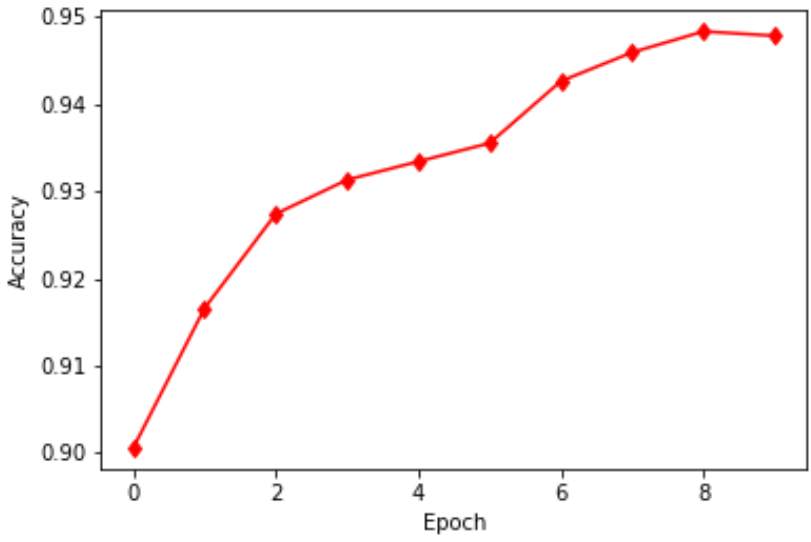
```
    bo=0
    bh=0
    OutH1=sigmoid(np.dot(samples,Wh.T)+bh)
    OutN=softmax(np.dot(OutH1,Wo.T)+bo)
    return OutN
# define network parameters
learningRate = 0.1
BatchSize= 100
Epoch=10
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
0.5,0.5,(NumHidden,NumInput)))
bh= np.random.uniform(0,0.5,(1,NumHidden))
dWh= np.zeros((NumHidden,NumInput))
dbh= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-0.5,0.5,(NumOutput,NumHidden))
bo= np.random.uniform(0,0.5,(1,NumOutput))
dWo= np.zeros((NumOutput,NumHidden))
dbo= np.zeros((1,NumOutput))
# Train the network with back propagation,mini-batch SGD
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
Acc=np.zeros(Epoch)
Cost_Entropy=np.zeros(Epoch)
IdxCost=0;
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSize)
))
for ep in range(Epoch):
    t1=dt.datetime.now()
```

```
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]
        x=np.matrix(x_train[Batch_sample,:])
        y=np.matrix(y_train [Batch_sample,:])
          # Feedforward propagation
        a=sigmoid(np.dot(x,Wh.T)+bh)
        o =softmax(np.dot(a,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=-
np.sum(np.multiply(y,np.log10(o)))
        IdxCost+=1;
        #calculate loss function
        do=(y-o)
        dWo=np.matrix(np.dot(do.T,a)/BatchSize)
        dbo=np.mean(do,0)
        Wo=Wo + learningRate*dWo
        bo=bo + learningRate*dbo
        #back propagate error
        dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        dWh=np.dot(dh.T,x)/BatchSize
        dbh=np.mean(dh,0)
        # Update weight
        Wh=Wh + learningRate*dWh
        bh=bh + learningRate*dbh
    Cost_Entropy[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
    print("Cross-Entropy %f" % Cost_Entropy[ep])
```

```
    #test the model
    RealOutN=feedforward(x_test,Wh,bh,Wo,bo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=dt.datetime.now() - t_start
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

```
Load MNIST Dataset
----------------------------------
(60000, 784)
(60000, 10)
0:00:09.079240
Training epoch 0
Cross-Entropy 25.796434
Accuracy: 0.889700
0:00:09.214800
Training epoch 1
Cross-Entropy 13.767033
Accuracy: 0.913900
0:00:09.157976
Training epoch 2
Cross-Entropy 11.673910
Accuracy: 0.922100
0:00:09.155567
Training epoch 3
Cross-Entropy 10.417808
Accuracy: 0.926900
0:00:09.415203
Training epoch 4
Cross-Entropy 9.524795
Accuracy: 0.934000
0:00:09.281436
```

```
Training epoch 5
Cross-Entropy 8.829437
Accuracy: 0.940600
0:00:09.081069
Training epoch 6
Cross-Entropy 8.244635
Accuracy: 0.940900
0:00:09.172462
Training epoch 7
Cross-Entropy 7.773770
Accuracy: 0.945800
0:00:09.211736
Training epoch 8
Cross-Entropy 7.357701
Accuracy: 0.944300
0:00:09.257006
Training epoch 9
Cross-Entropy 6.988373
Accuracy: 0.948000
Total time :  0:01:41.978292
```

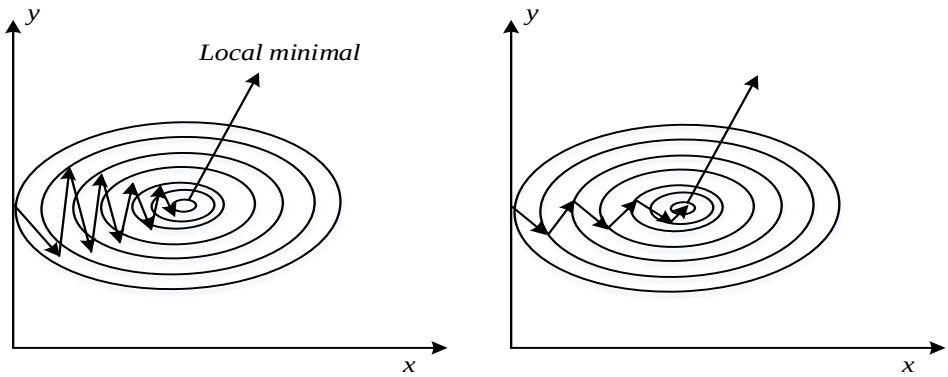


Chương 3
CÁC PHƯƠNG PHÁP TỐI ƯU QUÁ TRÌNH HUẤN LUYỆN MẠNG NƠ-RON

Trong chương trước chúng ta đã nghiên cứu mô hình toán của phương pháp Gradient Descent và ứng dụng vào bài toán nhận dạng ký tự viết tay. Kết quả của quá trình huấn luyện phụ thuộc vào các yếu tố như mô hình và đặc điểm của tập huấn luyện. Trong phần này chúng ta nghiên cứu một số phương pháp tối ưu dựa trên phương pháp Gradient Descent nhằm tăng tốc quá trình huấn luyện và hạn chế vấn đề hội tụ tại các điểm cực tiểu cục bộ. Các phương pháp đều được ví dụ minh họa trên cùng một ứng dụng là bài toán nhận dạng ký tự viết tay nhằm giúp người đọc dễ dàng nắm được phương pháp ứng dụng mô hình toán của các phương pháp vào quá trình lập trình với ngôn ngữ Python và thư viện numpy.

3.1 SỬ DỤNG MOMENTUM

Momentum được hiểu như "động lượng" hay "đà" trong vật lý học. Trong phương pháp Gradient Descent, tại mỗi thời điểm chúng ta tính giá trị cần cập nhật cho trọng số ngược chiều với chiều tăng cực đại của hàm mất mát. Momentum được thêm vào công thức cập nhật sao cho đại lượng momentum mang cả chiều và độ lớn của lần cập nhật trước đó. Điều này được hiểu như việc cập nhật tại thời điểm t chịu ảnh hưởng bởi lần cập nhật tại thời điểm $t - 1$. Việc thêm momentum vào quá trình cập



(a) Không sử dụng momentum *(a) Sử dụng momentum*

Hình 3.1: So sánh quá trình tối ưu không sử dụng momentum và sử dụng momentum.

nhật sẽ làm cho quá trình tiến đến điểm cực tiểu của hàm mất mát diễn ra nhanh hơn. Một cách trực quan chúng ta có thể xem việc tối ưu hàm sai số sử dụng Gradient Descent như quá trình dao động lên xuống dọc theo trục y. Nếu biên độ dao động càng lớn dọc theo trục y thì sẽ dẫn đến quá trình di chuyển theo trục x đến điểm cục bộ chậm hơn. Do đó ý tưởng để sử dụng momentum và làm giảm biên độ dao động nhằm đẩy nhanh quá trình tiến đến điểm cục bộ như minh họa trong Hình 3.1.

Trong phương pháp Gradient Descent hoặc Stochastic Gradient Descent, cập nhật trọng số được thực hiện theo công thức (3.1).

$$\begin{aligned} w_t &\leftarrow w_t - \Delta w_t \\ \Delta w_t &= \eta \nabla \mathrm{C}(w) \end{aligned} \tag{3.1}$$

Momentum được thêm vào quá trình cập nhật trọng số như sau:

$$\Delta w_t = \eta \nabla \mathrm{C}(w) + \gamma \Delta w_{t-1} \tag{3.2}$$

Trong đó γ là momentum có giá trị trong khoảng $(0, 1)$; Δw_t là giá trị cập nhật cho trọng số ở thời điểm t, Δw_{t-1} là giá trị cập nhật cho trọng số ở thời điểm $(t-1)$. Chương trình huấn luyện mạng nơ-ron cho bài toán nhận dạng ký tự viết tay, trong đó thêm momentum vào quá trình cập nhật trọng số được bổ sung như sau:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
import datetime as dt
def sigmoid(x):
    return 1./(1.+np.exp(-x))
```

```
def softmax(x):
    return np.divide(np.matrix(np.exp(x)),np.mat(np.sum(
np.exp(x),axis=1)))
def AccTest(outN,labels): #calculate the matching score
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,Wh,bh,Wo,bo):
    OutH1=sigmoid(np.dot(samples,Wh.T)+bh)
    OutN=softmax(np.dot(OutH1,Wo.T)+bo)
    return OutN
# define network parameters
learningRate = 0.1
BatchSize= 100
Epoch=10
Momentum=0.95
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
0.5,0.5,(NumHidden,NumInput)))
bh= np.random.uniform(0,0.5,(1,NumHidden))
del_Wh= np.zeros((NumHidden,NumInput))
del_bh= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-0.5,0.5,(NumOutput,NumHidden))
bo= np.random.uniform(0,0.5,(1,NumOutput))
del_Wo= np.zeros((NumOutput,NumHidden))
del_bo= np.zeros((1,NumOutput))
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
```

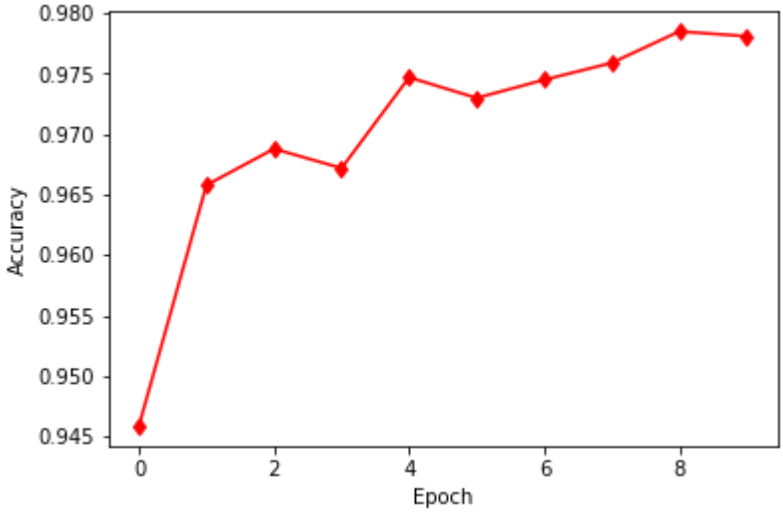
```
Acc=np.zeros(Epoch)
Cost_Entropy=np.zeros(Epoch)
IdxCost=0;
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSize)
))
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]

        x=np.matrix(x_train[Batch_sample,:])
        y=np.matrix(y_train [Batch_sample,:])
          # Feedforward propagation

        a=sigmoid(np.dot(x,Wh.T)+bh)
        o =softmax(np.dot(a,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=-
np.sum(np.multiply(y,np.log10(o)))
        IdxCost+=1;
        #calculate loss function
        do=(y-o)
        dWo=np.matrix(np.dot(do.T,a)/BatchSize)
        dbo=np.mean(do,0)
        WoUpdate=learningRate*dWo  + Momentum*del_Wo
        boUpdate= learningRate*dbo + Momentum*del_bo
        del_Wo=WoUpdate
        del_bo=boUpdate
        Wo=Wo + WoUpdate
        bo=bo + boUpdate
        #back propagate error
```

```
        dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        dWh=np.dot(dh.T,x)/BatchSize
        dbh=np.mean(dh,0)
        # Update weight
        WhUpdate=learningRate*dWh + Momentum*del_Wh
        bhUpdate=learningRate*dbh + Momentum*del_bh
        del_Wh=WhUpdate;
        del_bh=bhUpdate;
        Wh=Wh+ WhUpdate
        bh=bh+bhUpdate
    Cost_Entropy[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
    print("Cross-Entropy %f" % Cost_Entropy[ep])
    #test the model
    RealOutN=feedforward(x_test,Wh,bh,Wo,bo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=dt.datetime.now() - t_start
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

```
Load MNIST Dataset
----------------------------------
(60000, 784)
(60000, 10)
0:00:10.775514
Training epoch 0
Cross-Entropy 21.202618
```

```
Accuracy: 0.945900
0:00:10.931569
Training epoch 1
Cross-Entropy 5.990628
Accuracy: 0.965800
0:00:11.437919
Training epoch 2
Cross-Entropy 4.110325
Accuracy: 0.968800
0:00:10.659706
Training epoch 3
Cross-Entropy 2.871291
Accuracy: 0.967200
0:00:10.736743
Training epoch 4
Cross-Entropy 2.119727
Accuracy: 0.974700
0:00:10.657318
Training epoch 5
Cross-Entropy 1.727614
Accuracy: 0.973000
0:00:10.740984
Training epoch 6
Cross-Entropy 1.277120
Accuracy: 0.974500
0:00:10.604079
Training epoch 7
Cross-Entropy 0.957602
Accuracy: 0.975900
0:00:10.492816
Training epoch 8
Cross-Entropy 0.682725
Accuracy: 0.978500
0:00:10.529825
Training epoch 9
Cross-Entropy 0.528181
```

```
Accuracy: 0.978100
Total time :  0:01:52.230690
```



Bằng cách thêm momentum vào quá trình cập nhật trọng số, mô hình đạt độ chính xác cao hơn khi so sánh cùng số lần lặp lại quá trình huấn luyện.

3.2 ADAGRAD – ADAPTIVE GRADIENT DESCENT

Khái niệm tốc độ học thích nghi (Adaptive learning rate) được sử dụng khá nhiều trong các mô hình cải tiến của phương pháp Gradient Descent. Trong phương pháp huấn luyện sử dụng Gradient Descent, Stochastic Gradient Descent và mini-batch Stochastic Gradient Descent, tốc độ học là một hằng số có giá trị nhỏ và không thay đổi trong suốt quá trình huấn luyện mạng. Trong mô hình huấn luyện sử dụng tốc độ học thích nghi, tốc độ học giảm dần theo số lần lặp của quá trình huấn luyện và các trọng số ứng với ngõ vào có tần suất xuất hiện khác nhau sẽ có tốc độ học khác nhau. Phương pháp điều chỉnh tốc độ học thích nghi khác nhau dẫn đến các cải tiến của phương pháp huấn luyện sử dụng Gradient Descent.

Tốc độ học quyết định thời gian hội tụ của hàm mất mát. Tốc độ học quá lớn có thể làm cho quá trình học nhanh hơn, tuy nhiên nó dễ tạo ra dao động tại điểm cực tiểu. Trong các phương pháp huấn luyện mạng sử dụng tốc độ học thích nghi, tốc độ học giảm dần khi hàm mất mát tiến về gần điểm cực tiểu. Phương pháp Adagrad sử dụng giải thuật Gradient Descent, trong đó tốc độ học giảm tỷ lệ với lịch sử cập nhật của trọng số. Trong phương pháp Adagrad, mỗi trọng số được cập nhật sử dụng một

tốc độ khác nhau. Adagrad dựa trên nguyên lý các trọng số ứng với các đặc trưng với tần suất xuất hiện lớn sẽ có giá trị cập nhật nhỏ. Ngược lại, các trọng số ứng với các đặc trưng có tần suất xuất hiện nhỏ sẽ có giá trị cập nhật lớn hơn. Do đó, Adagrad có hiệu quả đối với các dữ liệu phân bố thưa.

Việc cập nhật trọng số với tốc độ học thích nghi, Adagrad, có thể được thực hiện như sau:

$$w_t = w_{t-1} - \eta' \frac{\partial C}{\partial w_{t-1}}$$

Trong đó:

$$\eta' = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}} \tag{3.3}$$

$$\alpha_t = \sum_{i=1}^{t} \left(\frac{\partial C}{\partial w_{t-1}} \right)^2$$

Với η là tốc độ học cố định, ε là giá trị đủ nhỏ nhằm đảm bảo mẫu số trong biểu thức (3.3) luôn khác 0. Có thể thấy mỗi một trọng số được cập nhật với một tốc độ học khác nhau phụ thuộc vào lịch sử cập nhật của chúng. Các giá trị tốc độ học giảm khi quá trình cập nhật được lặp lại liên tục. Chương trình huấn luyện mạng nơ-ron sử dụng Adagrad được viết lại như sau:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
import datetime as dt
```

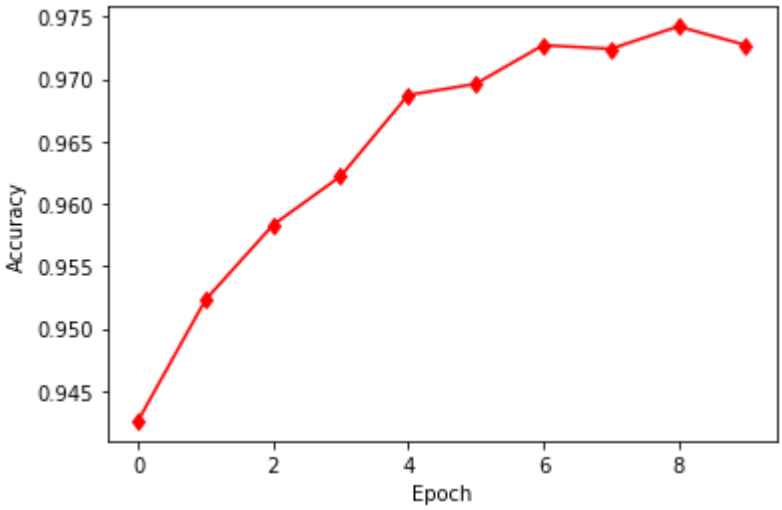
```
def sigmoid(x):
    return 1./(1.+np.exp(-x))
def softmax(x):
    return np.divide(np.matrix(np.exp(x)),np.mat(np.su
m(np.exp(x),axis=1)))
def AccTest(outN,labels):    # calculate the matching
score
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,Wh,bh,Wo,bo):
    bo=0
    bh=0
    OutH1=sigmoid(np.dot(samples,Wh.T)+bh)
    OutN=softmax(np.dot(OutH1,Wo.T)+bo)
    return OutN
# define network parameters
learningRate = 0.2
e = 0.01
BatchSize= 100
Epoch=10
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
0.5,0.5,(NumHidden,NumInput)))
bh= np.random.uniform(0,0.5,(1,NumHidden))
dWh= np.zeros((NumHidden,NumInput))
dWh_adagrad= np.zeros((NumHidden,NumInput))
dbh_adagrad= np.zeros((1,NumHidden))
#Output layer
```

```
Wo=np.random.uniform(-0.5,0.5,(NumOutput,NumHidden))
bo= np.random.uniform(0,0.5,(1,NumOutput))
dWo_adagrad= np.zeros((NumOutput,NumHidden))
dbo_adagrad= np.zeros((1,NumOutput))
# Train the network with
back propagation, stochastic Gradient descent
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
Acc=np.zeros(Epoch)
Cost_Entropy=np.zeros(Epoch)
IdxCost=0;
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSiz
e)))
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]
        #print(Batch_sample)
        x=np.matrix(x_train[Batch_sample,:])
        y=np.matrix(y_train [Batch_sample,:])
          # Feedforward propagation
        a=sigmoid(np.dot(x,Wh.T)+bh)
        o =softmax(np.dot(a,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=-
np.sum(np.multiply(y,np.log10(o)))
        IdxCost+=1;
        #calculate loss function
        do=(y-o)
        dWo=np.matrix(np.dot(do.T,a)/BatchSize)
        dbo=np.mean(do,0)
        dWo_adagrad+=np.multiply(dWo,dWo)
```

```
        dbo_adagrad+=np.multiply(dbo,dbo)
        #adaptive learning rate
        Wo=Wo + np.multiply((learningRate/np.sqrt(dWo_
adagrad+e)),dWo)
        bo=bo + np.multiply((learningRate/np.sqrt(dbo_
adagrad+e)),dbo)
        #back propagate error
        dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        dWh=np.dot(dh.T,x)/BatchSize
        dbh=np.mean(dh,0)
        # Update weight
        dWh_adagrad+=np.multiply(dWh,dWh)
        dbh_adagrad+=np.multiply(dbh,dbh)
        Wh=Wh + np.multiply((learningRate/np.sqrt(dWh_
adagrad+e)),dWh)
        bh=bh + np.multiply((learningRate/np.sqrt(dbh_
adagrad+e)),dbh)
    Cost_Entropy[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
    print("Cross-Entropy %f" % Cost_Entropy[ep])
    #test the model
    RealOutN=feedforward(x_test,Wh,bh,Wo,bo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=dt.datetime.now() - t_start
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

```
Load MNIST Dataset
----------------------------------
(60000, 784)
(60000, 10)
0:00:08.662139
Training epoch 0
Cross-Entropy 86.032728
Accuracy: 0.942600
0:00:08.999729
Training epoch 1
Cross-Entropy 7.461076
Accuracy: 0.952300
0:00:09.388296
Training epoch 2
Cross-Entropy 5.445465
Accuracy: 0.958300
0:00:08.954492
Training epoch 3
Cross-Entropy 4.347485
Accuracy: 0.962200
0:00:09.631177
Training epoch 4
Cross-Entropy 3.619225
Accuracy: 0.968700
0:00:08.862530
Training epoch 5
Cross-Entropy 3.112094
Accuracy: 0.969600
0:00:09.276593
Training epoch 6
Cross-Entropy 2.717460
Accuracy: 0.972700
0:00:08.777083
Training epoch 7
Cross-Entropy 2.350724
Accuracy: 0.972400
```

```
0:00:09.333739
Training epoch 8
Cross-Entropy 2.105085
Accuracy: 0.974200
0:00:08.700374
Training epoch 9
Cross-Entropy 1.856569
Accuracy: 0.972700
Total time :  0:01:34.834801
```



Phương pháp Adagrad cho phép cập nhật các trọng số sử dụng các giá trị tốc độ học khác nhau. Các giá trị tốc độ học giảm dần tỷ lệ với lịch sử cập nhật của từng trọng số. Trong chương trình trên chúng ta có thể bắt đầu với tốc độ học $\eta = 0.2$. Phương pháp Adagrad cũng cho kết quả tốt hơn, tương tự như sử dụng Momentum đối với tập dữ liệu MNIST.

3.3 RMSPROP (ROOT MEAN SQUARED PROPAGATION)

RMSProp cho phép cập nhật tốc độ học tại mỗi thời điểm tương tự như Adagrad. Tốc độ học giảm dần trong phương pháp Adagrad làm cho quá trình học chậm dần và tốc độ học có thể đạt đến giá trị rất nhỏ. RMSProp khắc phục nhược điểm này của Adagrad. Tốc độ học trong RMSProp thay đổi chậm hơn so với Adagrad.

$$w_t = w_{t-1} - \eta' \frac{\partial C}{\partial w_{t-1}} \tag{3.4}$$

Trong đó:

$$\eta' = \frac{\eta}{\sqrt{\alpha_t + \varepsilon}}$$

$$\alpha_t = \beta\alpha_{t-1} + (1-\beta)\left(\frac{\partial C}{\partial w_{t-1}}\right)^2$$

Chương trình sau được viết lại cho phép cập nhật trọng số với RMSProp.

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
import datetime as dt
def sigmoid(x):
    return 1./(1.+np.exp(-x))
def softmax(x):
    return np.divide(np.matrix(np.exp(x)),np.mat(np.sum(
np.exp(x),axis=1)))

def AccTest(outN,labels):    # calculate the matching sc
ore
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
```

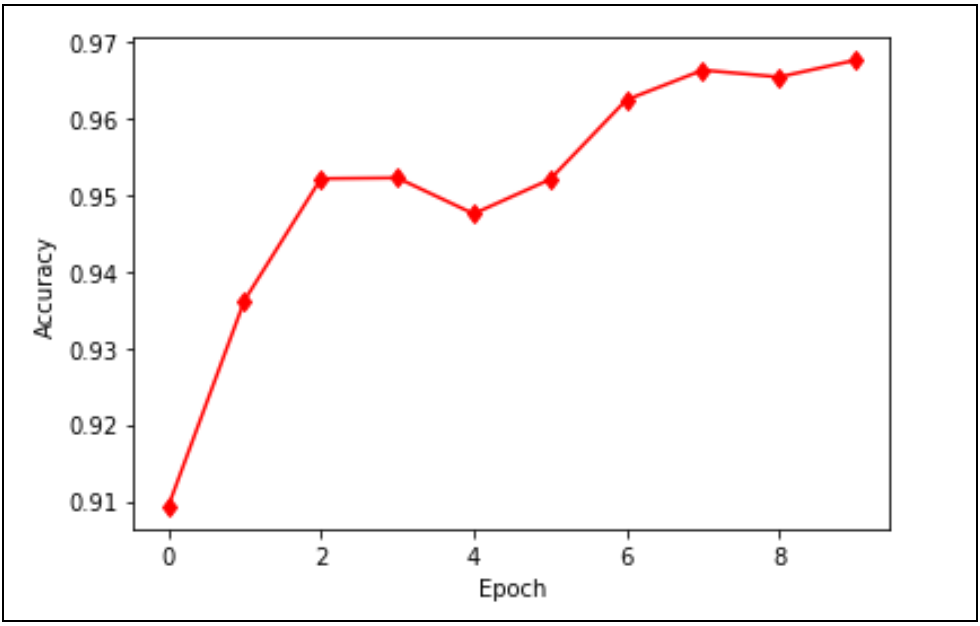
```
def feedforward(samples,Wh,bh,Wo,bo):
    OutH1=sigmoid(np.dot(samples,Wh.T)+bh)
    OutN=softmax(np.dot(OutH1,Wo.T)+bo)
    return OutN
# define network parameters
learningRate = 0.2
e = 0.1
b=0.9
BatchSize= 100
Epoch=10
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
0.5,0.5,(NumHidden,NumInput)))
bh= np.random.uniform(0,0.5,(1,NumHidden))
dWh= np.zeros((NumHidden,NumInput))
dWh_adagrad= np.zeros((NumHidden,NumInput))
dbh_adagrad= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-0.5,0.5,(NumOutput,NumHidden))
bo= np.random.uniform(0,0.5,(1,NumOutput))
dWo_adagrad= np.zeros((NumOutput,NumHidden))
dbo_adagrad= np.zeros((1,NumOutput))
# Train the network with
back propagation, stoshastic Gradient descent
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
Acc=np.zeros(Epoch)
Cost_Entropy=np.zeros(Epoch)
IdxCost=0;
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSize)
```

```
))
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]
        #print(Batch_sample)
        x=np.matrix(x_train[Batch_sample,:])
        y=np.matrix(y_train [Batch_sample,:])
          # Feedforward propagation
        a=sigmoid(np.dot(x,Wh.T)+bh)
        o =softmax(np.dot(a,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=-
np.sum(np.multiply(y,np.log10(o)))
        IdxCost+=1;
        #calculate loss function
        do=(y-o)
        dWo=np.matrix(np.dot(do.T,a)/BatchSize)
        dbo=np.mean(do,0)
        dWo_adagrad = b*dWo_adagrad + (1-
b)*np.multiply(dWo,dWo)
        dbo_adagrad = b*dbo_adagrad + (1-
b)*np.multiply(dbo,dbo)
        #adaptive learning rate
        Wo=Wo + np.multiply((learningRate/np.sqrt(dWo_ad
agrad+e)),dWo)
        bo=bo + np.multiply((learningRate/np.sqrt(dbo_ad
agrad+e)),dbo)
        #back propagate error
        dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        dWh=np.dot(dh.T,x)/BatchSize
        dbh=np.mean(dh,0)
```

```
        # Update weight
        dWh_adagrad = b*dWh_adagrad + (1-
b)*np.multiply(dWh,dWh)
        dbh_adagrad = b*dbh_adagrad + (1-
b)*np.multiply(dbh,dbh)
        Wh=Wh + np.multiply((learningRate/np.sqrt(dWh_ad
agrad+e)),dWh)
        bh=bh + np.multiply((learningRate/np.sqrt(dbh_ad
agrad+e)),dbh)
    Cost_Entropy[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
    print("Cross-Entropy %f" % Cost_Entropy[ep])
    #test the model
    RealOutN=feedforward(x_test,Wh,bh,Wo,bo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=dt.datetime.now() - t_start
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

```
Load MNIST Dataset
----------------------------------
(60000, 784)
(60000, 10)
0:00:09.680135
Training epoch 0
Cross-Entropy 470.174376
Accuracy: 0.909300
0:00:09.681680
Training epoch 1
```

```
Cross-Entropy 48.079397
Accuracy: 0.936300
0:00:09.819304
Training epoch 2
Cross-Entropy 27.524242
Accuracy: 0.952200
0:00:09.794219
Training epoch 3
Cross-Entropy 19.413944
Accuracy: 0.952300
0:00:09.732371
Training epoch 4
Cross-Entropy 17.579228
Accuracy: 0.947600
0:00:09.664412
Training epoch 5
Cross-Entropy 13.202434
Accuracy: 0.952100
0:00:09.685764
Training epoch 6
Cross-Entropy 13.426858
Accuracy: 0.962500
0:00:09.786811
Training epoch 7
Cross-Entropy 9.913321
Accuracy: 0.966400
0:00:09.937322
Training epoch 8
Cross-Entropy 10.441337
Accuracy: 0.965500
0:00:09.878648
Training epoch 9
Cross-Entropy 5.563022
Accuracy: 0.967700
Total time :  0:01:41.548008
```



3.4 ADAM

Adam là một phương pháp học dựa trên Stochastic Gradient Descent với tốc độ học thích nghi. Adam được xem như là sự kết hợp của phương pháp RMSProp và Stochastic Gradient Descent với momentum. Adam sử dụng bình phương Gradient để giảm tốc độ học tương tự như RMSProp và sử dụng ưu điểm của phương pháp Gradient Descent trong đó áp dụng momentum. Cập nhật trọng số được thực hiện theo phương pháp Adam được thực hiện theo bộ công thức (3.5).

$$
\begin{aligned}
v_{dW} &\leftarrow \beta_1 v_{dW} + (1-\beta_1)\frac{\partial C}{\partial w} \\
s_{dW} &\leftarrow \beta_2 s_{dW} + (1-\beta_2)\frac{\partial C}{\partial w} \\
\hat{v}_{dW} &= \frac{v_{dW}}{1-(\beta_1)^t} \\
\hat{s}_{dW} &= \frac{s_{dW}}{1-(\beta_2)^t} \\
W &\leftarrow W - \eta\frac{\hat{v}_{dW}}{\sqrt{\hat{s}_{dW}}+\varepsilon}
\end{aligned}
\tag{3.5}
$$

Chúng ta sửa lại chương trình để cập nhật trọng số theo phương pháp Adam. β_1 và β_2 thường được lựa chọn các các hệ số gần bằng 1 ($\beta_1 = 0.9$ và $\beta_2 = 0.99$).

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
import datetime as dt
def sigmoid(x):
    return 1./(1.+np.exp(-x))
def softmax(x):
    return np.divide(np.matrix(np.exp(x)),np.mat(np.sum(
np.exp(x),axis=1)))

def AccTest(outN,labels):    # calculate the matching sc
ore
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,Wh,bh,Wo,bo):
    OutH1=sigmoid(np.dot(samples,Wh.T)+bh)
    OutN=softmax(np.dot(OutH1,Wo.T)+bo)
    return OutN
# define network parameters
learningRate = 0.2
e = 1e-1
b1=0.9
```

```
b2=0.99
BatchSize= 100
Epoch=10
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
0.5,0.5,(NumHidden,NumInput)))
bh= np.random.uniform(0,0.5,(1,NumHidden))
dWh= np.zeros((NumHidden,NumInput))
vdWh= np.zeros((NumHidden,NumInput))
sdWh= np.zeros((NumHidden,NumInput))
dbh= np.zeros((1,NumHidden))
vdbh= np.zeros((1,NumHidden))
sdbh= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-0.5,0.5,(NumOutput,NumHidden))
bo= np.random.uniform(0,0.5,(1,NumOutput))
dWo= np.zeros((NumOutput,NumHidden))
vdWo= np.zeros((NumOutput,NumHidden))
sdWo= np.zeros((NumOutput,NumHidden))
dbo= np.zeros((1,NumOutput))
vdbo= np.zeros((1,NumOutput))
sdbo= np.zeros((1,NumOutput))
# Train the network with
back propagation, stoshastic Gradient descent
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
Acc=np.zeros(Epoch)
Cost_Entropy=np.zeros(Epoch)
IdxCost=0;
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSize)
```

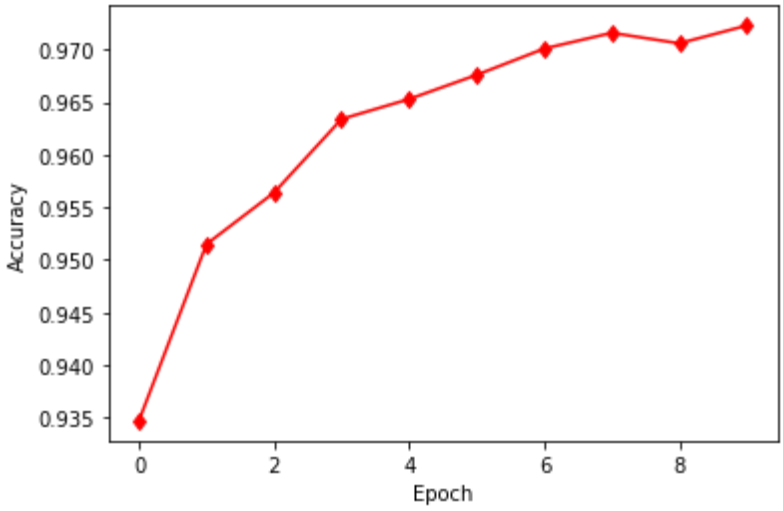
```
))
t = 0;
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]
        #print(Batch_sample)
        x=np.matrix(x_train[Batch_sample,:])
        y=np.matrix(y_train [Batch_sample,:])
          # Feedforward propagation
        a=sigmoid(np.dot(x,Wh.T)+bh)
        o =softmax(np.dot(a,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=-
np.sum(np.multiply(y,np.log10(o)))
        IdxCost+=1;
        #calculate loss function
        do=(y-o)
        dWo=np.matrix(np.dot(do.T,a)/BatchSize)
        dbo=np.mean(do,0)
        #Adam optimizer
        vdWo= b1*vdWo + (1-b1)*dWo
        sdWo= b2*sdWo + (1-b2)*np.multiply(dWo,dWo)
        vdWo_corrected =vdWo/(1-np.power(b1,t+1))
        sdWo_corrected =sdWo/(1-np.power(b2,t+1))
        vdbo= b1*vdbo + (1-b1)*dbo
        sdbo= b2*sdbo + (1-b2)*np.multiply(dbo,dbo)
        vdbo_corrected =vdbo/(1-np.power(b1,t+1))
        sdbo_corrected =sdbo/(1-np.power(b2,t+1))
        #adaptive learning rate
        Wo=Wo + learningRate*(vdWo_corrected/(np.sqrt(sd
Wo_corrected)+e))
```

```
        bo=bo + learningRate*(vdbo_corrected/(np.sqrt(sd
bo_corrected)+e))
        #back propagate error
        dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        dWh=np.dot(dh.T,x)/BatchSize
        dbh=np.mean(dh,0)
        # Update weight

        #Adam optimizer
        vdWh= b1*vdWh + (1-b1)*dWh
        sdWh= b2*sdWh + (1-b2)*np.multiply(dWh,dWh)
        vdWh_corrected =vdWh/(1-np.power(b1,t+1))
        sdWh_corrected =sdWh/(1-np.power(b2,t+1))
        vdbh= b1*vdbh + (1-b1)*dbh
        sdbh= b2*sdbh + (1-b2)*np.multiply(dbh,dbh)
        vdbh_corrected =vdbh/(1-np.power(b1,t+1))
        sdbh_corrected =sdbh/(1-np.power(b2,t+1))
        #adaptive learning rate
        Wh=Wh + learningRate*(vdWh_corrected/(np.sqrt(sd
Wh_corrected)+e))
        bh=bh + learningRate*(vdbh_corrected/(np.sqrt(sd
bh_corrected)+e))
        t+=1
    Cost_Entropy[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
    print("Cross-Entropy %f" % Cost_Entropy[ep])
    #test the model
    RealOutN=feedforward(x_test,Wh,bh,Wo,bo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=dt.datetime.now() - t_start
```

```
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

```
#Load MNIST Dataset
----------------------------------
(60000, 784)
(60000, 10)
0:00:16.841429
Training epoch 0
Cross-Entropy 34.697091
Accuracy: 0.934700
0:00:17.353949
Training epoch 1
Cross-Entropy 8.364542
Accuracy: 0.951500
0:00:17.455770
Training epoch 2
Cross-Entropy 6.467846
Accuracy: 0.956400
0:00:17.285357
Training epoch 3
Cross-Entropy 5.161616
Accuracy: 0.963400
0:00:17.733491
Training epoch 4
Cross-Entropy 4.374220
Accuracy: 0.965300
0:00:18.088458
Training epoch 5
Cross-Entropy 3.726710
Accuracy: 0.967600
0:00:17.539680
Training epoch 6
Cross-Entropy 3.223620
```

```
Accuracy: 0.970100
0:00:18.315199
Training epoch 7
Cross-Entropy 2.832344
Accuracy: 0.971600
0:00:17.610986
Training epoch 8
Cross-Entropy 2.488857
Accuracy: 0.970600
0:00:17.579738
Training epoch 9
Cross-Entropy 2.243875
Accuracy: 0.974300
Total time :  0:03:00.427781
```



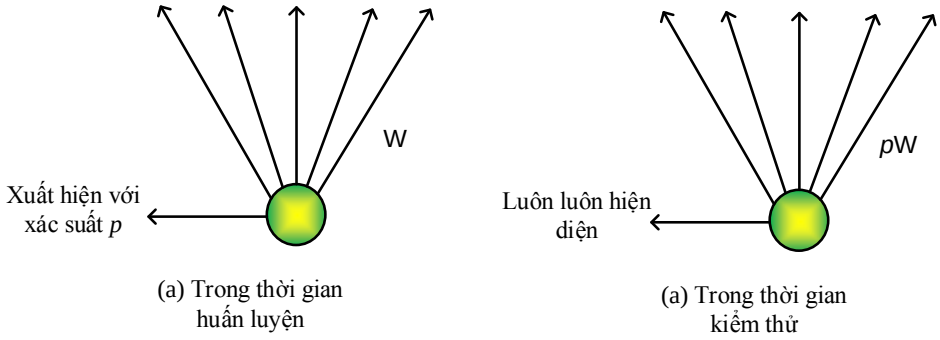
Huấn luyện với Adam được sử dụng phổ biến trong các mạng học sâu hiện đại bởi nó kế thừa được ưu điểm của các phương pháp học với tốc độ học thích nghi và sử dụng momentum.

Các chương trình ví dụ khi áp dụng các phương pháp huấn luyện mạng nơ-ron nhiều lớp được viết bằng ngôn ngữ Python và thư viện numpy. Trong các chương trình, các đoạn lệnh được viết sát theo mô hình toán học nhất mà không phải là được viết tối ưu nhất để người đọc có thể hiểu được các phương pháp huấn luyện mạng nơ-ron và ứng dụng vào quá trình lập trình huấn luyện các mô hình cụ thể.

Phương pháp huấn luyện với tốc độ học thích nghi dựa trên nguyên lý cập nhật trọng số của phương pháp Gradient Descent, tuy nhiên, trong các phương pháp sử dụng tốc độ học thích nghi, tốc độ học được giảm dần sau mỗi lần cập nhật trọng số. Trong các ví dụ, chúng ta đã khảo sát một số phương pháp bao gồm sử dụng Momentum, thay đổi tốc độ học với Adagrad, RMSProp. Một số phương pháp cũng dựa trên việc thay đổi tốc độ học như Adadelta, AdaMax…. Có thể thấy các phương pháp đều dựa trên mô hình toán tối ưu cho hàm mục tiêu (hàm mất mát) bằng cách cập nhật các trọng số theo chiều ngược với chiều tăng cực đại của hàm mất mát. Thông qua việc xây dựng các chương trình khá cơ bản từ ngôn ngữ Python và thư viện numpy, chúng ta sẽ hiểu hơn về các thuật toán, quá trình huấn luyện mạng và các phương pháp tối ưu hàm mục tiêu.

3.5 SỬ DỤNG KỸ THUẬT DROPOUT

Dropout là một kỹ thuật được sử dụng để cải thiện vấn đề quá khớp (overfitting) trong các mạng học sâu. Hiện tượng quá khớp là hiện tượng một mạng nơ-ron được huấn luyện và đáp ứng rất tốt với tập dữ liệu huấn luyện, tuy nhiên, mô hình lại cho kết quả không tốt trên tập dữ liệu khác. Hiện tượng quá khớp có thể được cải thiện bằng cách sử dụng kỹ thuật Dropout. Đây là kỹ thuật loại bỏ một số đơn vị nơ-ron trong các lớp mạng bằng cách ẩn nó đi. Bằng cách sử dụng Dropout, chúng ta hầu như loại bỏ tạm thời các nơ-ron này ra khỏi mạng nơ-ron trong quá trình huấn luyện. Việc này đồng nghĩa với các kết nối đến và các kết nối đi từ các nơ-ron bị loại bỏ cũng trở nên không có tác dụng. Dropout là kỹ thuật được sử dụng phổ biến trong các mạng sâu (Deep neural network) hiện nay. Trong các trường hợp đơn giản, mỗi đơn vị được giữ lại với một xác suất cố định *p* độc lập với các đơn vị nơ-ron khác. Trong đó *p* thông thường được chọn là 0.5.



Hình 3.2: Sử dụng Dropout trong quá trình huấn luyện và trong quá trình kiểm thử.

Trong quá trình huấn luyện, nơ-ron tồn tại với một xác suất p và kết nối đến các nơ-ron ở lớp tiếp theo thông qua tập trọng số w. Trong quá trình kiểm thử, các đơn vị nơ-ron luôn tồn tại và các trọng số được nhân bởi p. Ngõ ra trong quá trình kiểm thử được kỳ vọng luôn luôn bằng với ngõ ra trong quá trình huấn luyện. Như vậy kỹ thuật Dropout là biến ngẫu nhiên các nơ-ron với xác suất $1 - p$. Tuy nhiên trong quá trình kiểm thử cũng như quá trình hoạt động thì toàn bộ nơ-ron trong mạng luôn hiện diện. Chính vì số lượng nơ-ron trong quá trình huấn luyện đã giảm đi, do đó để có thể ngõ ra không thay đổi nhiều khi toàn bộ nơ-ron trong mạng tồn tại thì tập trọng số sẽ được nhân với p.

Giả sử mạng nơ-ron với L lớp ẩn. Trong đó $l \in \{1, \dots, L\}$ là lớp ẩn thứ l trong mạng, $z^{(l)}$ là vector ngõ vào của lớp thứ l, $y^{(l)}$ là vector ngõ ra từ lớp l ($y^{(0)} = x$ là ngõ vào). $w^{(l)}$ và $b^{(l)}$ là trọng số và độ lệch của lớp l. Quá trình lan truyền thẳng trong mạng được mô tả như sau:

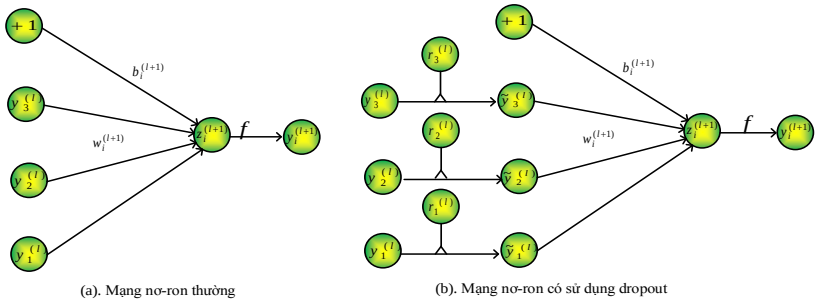
$$\begin{aligned} z_i^{(l+1)} &= w_i^{(l+1)} y^l + b_i^{(l+1)} \\ y_i^{(l+1)} &= f(z_i^{(l+1)}) \end{aligned} \tag{3.6}$$

Trong đó f là hàm kích hoạt, ví dụ $= 1/(1 + e^{-x})$. Áp dụng Dropout ta có:

$$\begin{aligned} r_j^{(l)} &\sim Bernoulli(p) \\ \tilde{y}^{(l)} &= r^{(l)} * y^{(l)} \\ z_i^{(l+1)} &= w_i^{(l+1)} \tilde{y}^l + b_i^{(l+1)} \\ y_i^{(l+1)} &= f(z_i^{(l+1)}) \end{aligned} \tag{3.7}$$

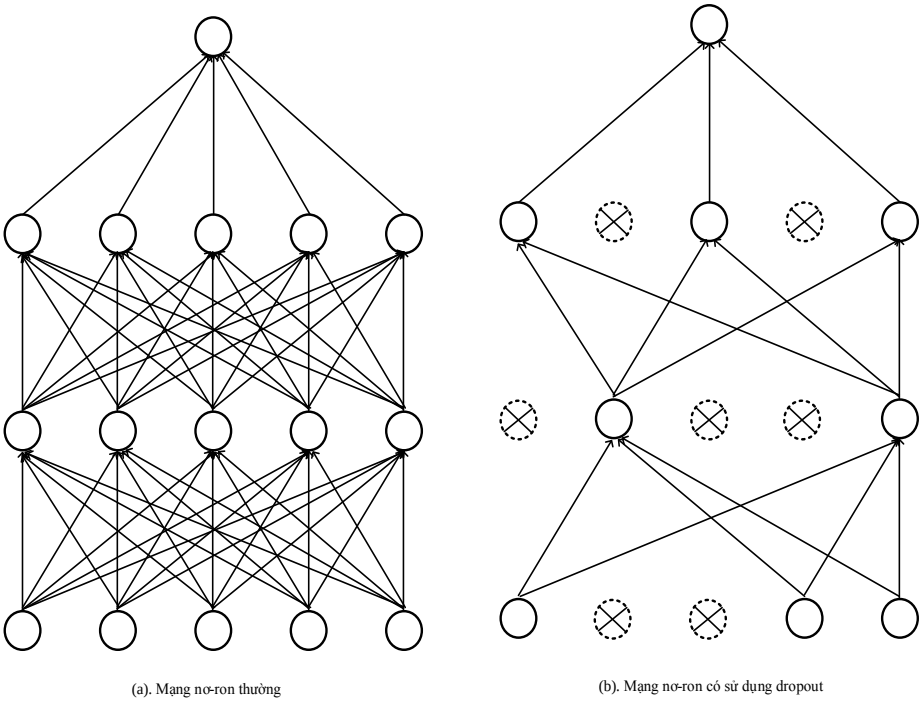
Trong đó phép $(*)$ là phép nhân từng phần tử ma trận (Element-wise product).

So sánh mạng nơ-ron thường và mạng nơ-ron có sử dụng Dropout được minh họa trong Hình 3.3.



Hình 3.3: So sánh mạng nơ-ron thường và mạng nơ-ron sử dụng Dropout.

Đối với bất kỳ lớp ẩn l nào, $r^{(l)}$ là vector của biến ngẫu nhiên Bernoulli với mỗi phần tử có xác suất bằng 1 là p. Vector này được nhân từng phần tử với ngõ ra của lớp l. Ngõ ra này sau đó trở thành ngõ vào của lớp tiếp theo. Như vậy nếu một phần tử trong $r^{(l)}$ có giá trị là 0 thì xem như nơ-ron tương ứng bị ẩn. Mô hình mạng nơ-ron với các kết nối đầy đủ và mạng nơ-ron có áp dụng Dropout trong quá trình huấn luyện được mô tả trong Hình 3.4.



Hình 3.4: Mô hình mạng nơ-ron với các kết nối đủ và mô hình mạng nơ-ron có áp dụng kỹ thuật Dropout.

Áp dụng Dropout được xem như tạo ra các mạng có cấu trúc đơn giản hơn trong quá trình huấn luyện. Trong quá trình huấn luyện, các Gradient của hàm mất mát dựa trên các cấu trúc mạng đơn giản này thay vì tính toán dựa trên mạng với kết nối đầy đủ. Trong quá trình kiểm thử và quá trình hoạt động, tập trọng số được thu nhỏ lại với $w_{test}^{(l)} = pw^{(l)}$ như được mô tả trong Hình 3.2.

Huấn luyện mạng nơ-ron có áp dụng Dropout tương tự như huấn luyện mạng nơ-ron nhiều lớp thông thường sử dụng Stochastic Gradient Descent. Điểm khác biệt duy nhất là trong mỗi trường hợp trong nhóm nhỏ (mini-batch), chúng ta thực hiện trên một mạng nơ-ron có kết nối đơn giản hơn vì một số lượng nơ-ron bị ẩn. Khi đó quá trình lan truyền

thẳng và quá trình lan truyền ngược đều được thực hiện trên mạng nơ-ron với các nơ-ron đã ẩn này.

Bây giờ, chúng ta sẽ áp dụng Dropout trên bài toán nhận dạng ký tự viết tay MNIST quen thuộc đã xây dựng ở các ví dụ trước. Trong mô hình này, chúng ta sẽ sử dụng kỹ thuật Dropout vơi xác suất $p = 0.5$. Trong quá trình huấn luyện, việc ẩn các nơ-ron được thực hiện trong cả quá trình lan truyền thẳng và quá trình lan truyền ngược. Các nơ-ron trong lớp ẩn sẽ được ẩn bằng cách nhân ngõ ra các phần tử ẩn với giá trị 0. Để thực hiện điều này chúng ta có thể tạo ra một vector với các giá trị ngẫu nhiên 0 và 1 với xác suất các phần tử có giá trị 1 là p. Các nơ-ron ẩn là ngẫu nhiên và khác nhau đối với mỗi mẫu trong nhóm nhỏ (mini-batch). Do đó, chúng ta có thể tạo một ma trận các giá trị 0, 1 với xác suất các giá trị 1 là p cho từng nhóm. Quá trình lan truyền thẳng được thực hiện với việc ẩn các nơ-ron ngẫu nhiên. Giả sử chiều dài nhóm (batch-size) là 100. Chúng ta có ngõ vào, trọng số, ngõ ra các nơ-ron lớp ẩn được xác định bởi các ma trận như sau:

$$x = \begin{pmatrix} x_1^1 & \cdots & x_{784}^1 \\ \vdots & \ddots & \vdots \\ x_1^{100} & \cdots & x_{784}^{100} \end{pmatrix} \tag{3.8}$$

Trong đó $x_1^1 \dots x_{784}^1$ và $x_1^{100} \dots x_{784}^{100}$ là ngõ vào cho mẫu thứ 1 và mẫu thứ 100 trong nhóm.

Ma trận trọng số lớp ẩn và lớp ngõ ra được định nghĩa tương tự ví dụ trước.

$$\begin{gathered} Wh = \begin{bmatrix} w_{1,1} & \cdots & w_{1,784} \\ \vdots & \ddots & \vdots \\ w_{512,1} & \cdots & w_{512,784} \end{bmatrix} \\ bh = [b_1, \dots, b_{512}] \\ Wo = \begin{bmatrix} w_{1,1} & \cdots & w_{1,512} \\ \vdots & \ddots & \vdots \\ w_{10,1} & \cdots & w_{10,512} \end{bmatrix} \\ bo = [b_1, \dots, b_{10}] \end{gathered} \tag{3.9}$$

Ngõ ra các nơ-ron lớp ẩn được tính 1 lần cho 100 mẫu.

$$\begin{gathered} z_h = x.Wh^T = \begin{bmatrix} zh_1^1 & \cdots & zh_{512}^1 \\ \vdots & \ddots & \vdots \\ zh_1^{100} & \cdots & zh_{512}^{100} \end{bmatrix} \\ a = \sigma(z_h) \end{gathered} \tag{3.10}$$

Ma trận Dropout có kích thước bằng với kích thước ma trận ngõ ra a để xóa giá trị ngõ ra các nơ-ron trong lớp ẩn nhằm ẩn các nơ-ron này. Các phần tử trong ma trận a được nhân với các phần tử tương ứng trong ma trận Dropout.

$$Dropout_{100\times512} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 & 0 \\ 1 & 1 & \cdots & 1 & 0 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 1 & \cdots & 1 & 0 & 1 \\ 1 & 1 & \cdots & 0 & 0 & 0 \end{bmatrix} \tag{3.11}$$

$$a = a.Dropout$$

Vị trí các nơ-ron bị ẩn được sử dụng trong quá trình lan truyền ngược. Các nơ-ron bị ẩn sẽ cho các Gradient bằng 0 trong quá trình tính cập nhật cho trọng số. Quá trình khởi tạo ma trận có giá trị 0 và 1 ngẫu nhiên với xác suất p và ẩn các nơ-ron trong lớp ẩn được thực hiện thông qua các đoạn chương trình minh họa bên dưới.

Khởi tạo ma trận các giá trị 0,1 ngẫu nhiên với p =0.5:

```
Dropout=np.random.binomial(size=(BatchSize,NumHidden), n
=1, p= 0.5)
a=sigmoid(np.dot(x,Wh.T)+bh)
```

Ẩn các nơ-ron trong quá trình lan truyền thẳng

```
a=np.multiply(a,Dropout)
o =softmax(np.dot(a,Wo.T)+bo)
```

Ẩn các nơ-ron trong quá trình lan truyền ngược bằng cách ẩn các Gradient của các nơ-ron ẩn tương ứng

```
dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-a)))
dh=np.multiply(dh,Dropout)
```

Trong kỹ thuật Dropout, các nơ-ron chỉ ẩn trong quá trình huấn luyện. Trong quá trình kiểm tra, tất cả các nơ-ron đều hiện diện, tuy nhiên, trọng số kết nối của các nơ-ron này đến lớp sau sẽ bị co lại bằng cách nhân các trọng số với giá trị xác suất p. Chương trình lan truyền thẳng được sửa lại như sau.

```
def feedforward(samples,Wh,bh,Wo,bo,Dropout):
    OutH1=sigmoid(np.dot(samples,Wh.T)+bh)
    OutN=softmax(np.dot(OutH1,Dropout*Wo.T)+bo)
```

Kỹ thuật Dropout được áp dụng cho tất cả các lớp trong mạng, các lớp ẩn được khuyến khích lựa chọn xác suất Dropout $p = 0.5$. Trong ví

dụ minh họa này chúng ta chỉ áp dụng Dropout cho các nơ-ron ở lớp ẩn. Chương trình huấn luyện mạng nơ-ron cho ứng dụng nhận dạng ký tự viết tay MNIST áp dụng Dropout cho lớp ẩn được viết lại như sau:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
import datetime as dt
def sigmoid(x):
    return 1./(1.+np.exp(-x))
def softmax(x):
    return np.divide(np.matrix(np.exp(x)),np.mat(np.sum(
np.exp(x),axis=1)))
def AccTest(outN,labels):#calculate the matching score
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,Wh,bh,Wo,bo,Dropout):
    bo=0
    bh=0
    OutH1=sigmoid(np.dot(samples,Wh.T)+bh)
    OutN=softmax(np.dot(OutH1,Dropout*Wo.T)+bo)
    return OutN
# define network parameters
```

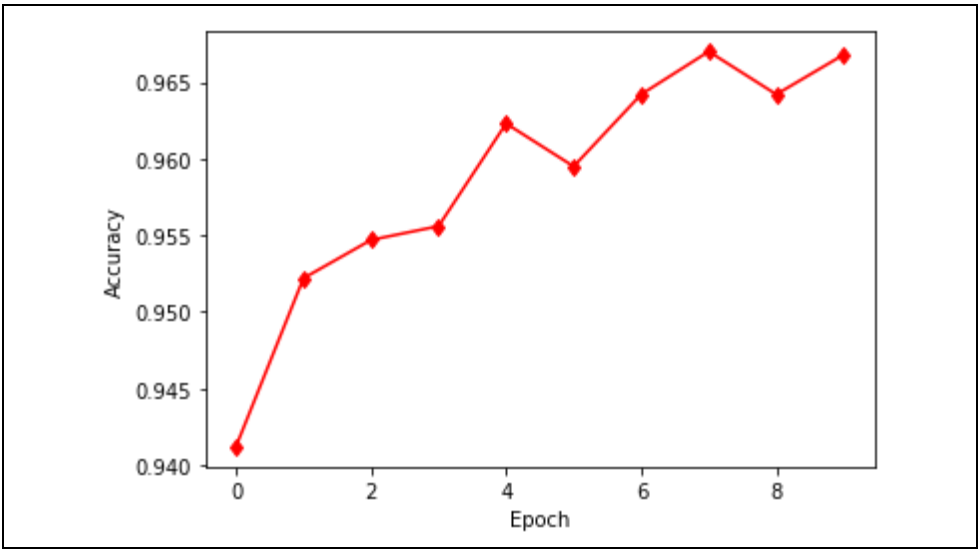
```
learningRate = 0.1
BatchSize= 100
Epoch=10
Momentum=0.95
p = 0.5 # dropout probability for hidden layer
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
0.5,0.5,(NumHidden,NumInput)))
bh= np.random.uniform(0,0.5,(1,NumHidden))
del_Wh= np.zeros((NumHidden,NumInput))
del_bh= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-0.5,0.5,(NumOutput,NumHidden))
bo= np.random.uniform(0,0.5,(1,NumOutput))
del_Wo= np.zeros((NumOutput,NumHidden))
del_bo= np.zeros((1,NumOutput))
# Train the network with
back propagation, stoshastic Gradient descent
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
Acc=np.zeros(Epoch)
Cost_Entropy=np.zeros(Epoch)
IdxCost=0;
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSize)
))
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
```

```
        # Dropout
        # visible a number neurons in hidden layer
        Dropout=np.random.binomial(size=(BatchSize,NumHi
dden), n=1, p= p)
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]
        #print(Batch_sample)
        x=np.matrix(x_train[Batch_sample,:])
        y=np.matrix(y_train [Batch_sample,:])
          # Feedforward propagation
        a=sigmoid(np.dot(x,Wh.T)+bh)
        #apply dropout to feed-forward propagation
        a=np.multiply(a,Dropout)
        o =softmax(np.dot(a,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=-
np.sum(np.multiply(y,np.log10(o)))
        IdxCost+=1;
        #calculate loss function
        do=(y-o)
        dWo=np.matrix(np.dot(do.T,a)/BatchSize)
        dbo=np.mean(do,0)
        WoUpdate=learningRate*dWo  + Momentum*del_Wo
        boUpdate= learningRate*dbo + Momentum*del_bo
        del_Wo=WoUpdate
        del_bo=boUpdate
        Wo=Wo + WoUpdate
        bo=bo + boUpdate
        #back propagate error
        dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        #apply dropout to backpropagation
        dh=np.multiply(dh,Dropout)
        dWh=np.dot(dh.T,x)/BatchSize
        dbh=np.mean(dh,0)
```

```
        # Update weight
        WhUpdate=learningRate*dWh + Momentum*del_Wh
        bhUpdate=learningRate*dbh + Momentum*del_bh
        del_Wh=WhUpdate;
        del_bh=bhUpdate;
        Wh=Wh+ WhUpdate
        bh=bh+bhUpdate
    Cost_Entropy[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
    print("Cross-Entropy %f" % Cost_Entropy[ep])
    #test the model
    RealOutN=feedforward(x_test,Wh,bh,Wo,bo,p)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=dt.datetime.now() - t_start
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

```
Load MNIST Dataset
-----------------------------------
(60000, 784)
(60000, 10)
0:00:09.475472
Training epoch 0
Cross-Entropy 22.227583
Accuracy: 0.941200
0:00:09.427273
Training epoch 1
Cross-Entropy 11.078878
```

```
Accuracy: 0.952200
0:00:09.493466
Training epoch 2
Cross-Entropy 8.758152
Accuracy: 0.954700
0:00:09.478311
Training epoch 3
Cross-Entropy 7.516186
Accuracy: 0.955600
0:00:09.413973
Training epoch 4
Cross-Entropy 6.460323
Accuracy: 0.962300
0:00:09.470159
Training epoch 5
Cross-Entropy 5.934156
Accuracy: 0.959500
0:00:09.517908
Training epoch 6
Cross-Entropy 5.275725
Accuracy: 0.964200
0:00:09.555959
Training epoch 7
Cross-Entropy 4.980261
Accuracy: 0.967000
0:00:09.453716
Training epoch 8
Cross-Entropy 4.610047
Accuracy: 0.964200
0:00:09.495181
Training epoch 9
Cross-Entropy 4.271871
Accuracy: 0.966800
Total time :  0:01:39.306834
```

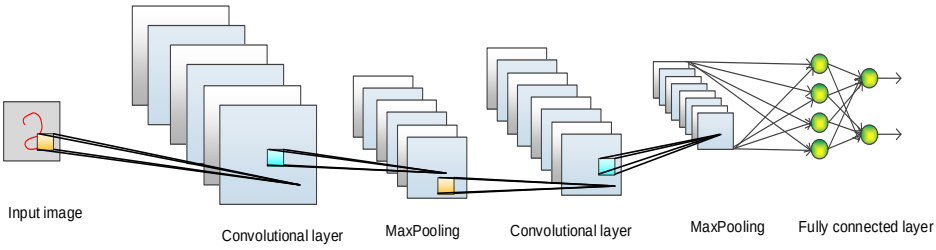

0.965
0.960
0.955
0.950
0.945
0.940
Accuracy
0
2
4
6
8
Epoch

Chương 4
HUẤN LUYỆN MẠNG NƠ-RON TÍCH CHẬP (Convolutional Neural Network)

4.1 MẠNG NƠ-RON TÍCH CHẬP

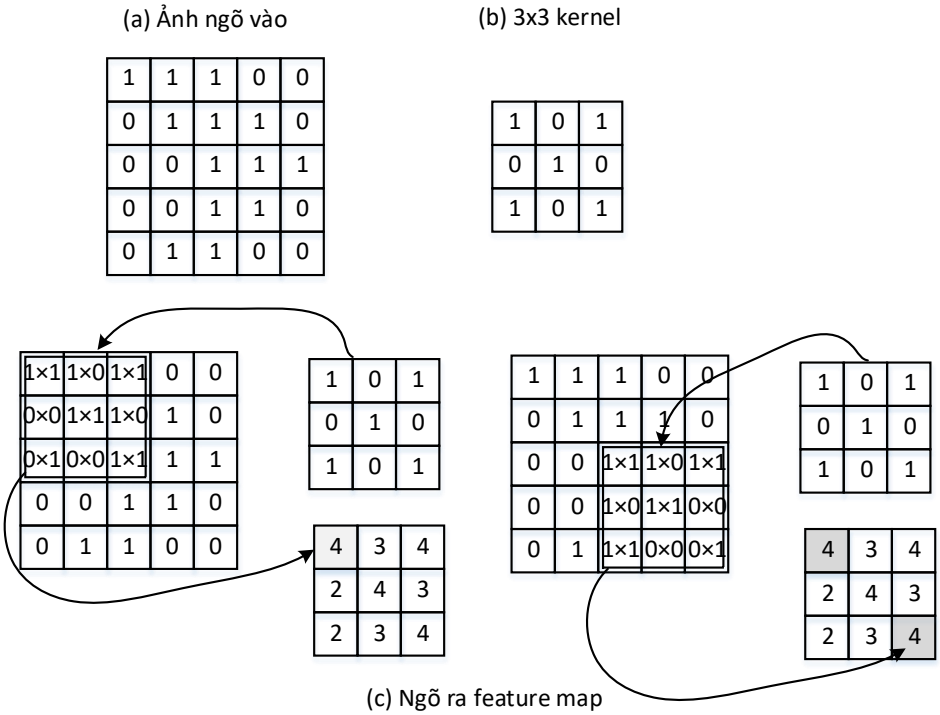
Mạng nơ-ron tích chập hay còn có thể gọi ngắn gọn là mạng nơ-ron chập (Convolutional Neural Network - CNN) dựa trên phép tích chập của ảnh với các bộ lọc thu được trong quá trình huấn luyện để tách các đặc điểm đặc trưng của ảnh. Trong mạng nơ-ron tích chập, chúng ta không hoặc ít khi sử dụng các bộ lọc đã biết trong xử lý ảnh để tách biên hay loại bỏ nhiễu. Các bộ lọc thu được sau quá trình huấn luyện có thể rất khác với các bộ lọc thông dụng trong xử lý ảnh. Trong giải thuật học có giám sát, các bộ lọc được cập nhật trọng số sao cho hàm mất mát tiến về điểm cực tiểu. Ví dụ, lớp tích chập đầu tiên được huấn luyện để phát hiện biên, trong khi các lớp sau được huấn luyện để phát hiện các đặc tính phức tạp hơn của các đối tượng trong ảnh. Một trong những ý tưởng hình thành nên mạng nơ-ron tích chập là việc chia sẻ các trọng số. Trong các lớp kết nối đầy đủ của mạng nhiều lớp, một nơ-ron ở lớp sau được kết nối đến tất cả các nơ-ron ở lớp trước nó. Tương tự như vậy, một nơ-ron sẽ kết nối đến tất cả các nơ-ron ở lớp sau nó. Kết quả là tạo ra số lượng kết nối và các trọng số tương ứng khá lớn trong mạng. Trong lớp chập, nơ-ron ở lớp sau chỉ kết nối đến một nhóm các nơ-ron ở lớp trước. Hơn nữa các nơ-ron kết nối đến các nhóm này sử dụng cùng một tập trọng số. Do đó mạng nơ-ron tích chập hiệu quả khi số lượng lớp mạng tăng lên. Mạng nơ-ron tích chập là một mô hình khá phổ biến trong các mạng sâu nhiều lớp (Deep neural network).

Mạng nơ-ron tích chập là sự kết hợp của mạng nơ-ron với các lớp kết nối đủ và mạng nơ-ron với các lớp chập. Một mạng nơ-ron tích chập có kiến trúc như Hình 4.1.



Hình 4.1: Mạng nơ-ron tích chập.

Mạng nơ-ron tích chập trong Hình 4.1 có kiến trúc bao gồm 2 lớp chập (convolutional layer), 2 lớp gộp (pooling layer) và lớp kết nối đầy đủ (fully-connected layer). Ảnh ngõ vào là các ma trận 2 chiều với các giá trị và giả sử có các giá trị nhị phân như trong Hình 4.2(a). Bộ lọc (filter) hay còn được gọi là "kernel" có kích thước 3×3 và giả sử có các giá trị nhị phân như Hình 4.2(b). Nếu chúng ta không thực hiện kết gán 0 vào biên ảnh ban đầu thì kích thước ma trận ngõ ra sẽ nhỏ hơn kích thước ảnh ma trận ảnh ngõ vào. Các phần tử của ma trận ngõ ra được gọi là các bản đồ đặc trưng (Feature map) thu được từ phép tính tổng của phép nhân các phần tử bộ lọc và ảnh khi trượt bộ lọc hết ảnh. Các feature map được tạo ra như minh họa trong Hình 4.2(c).

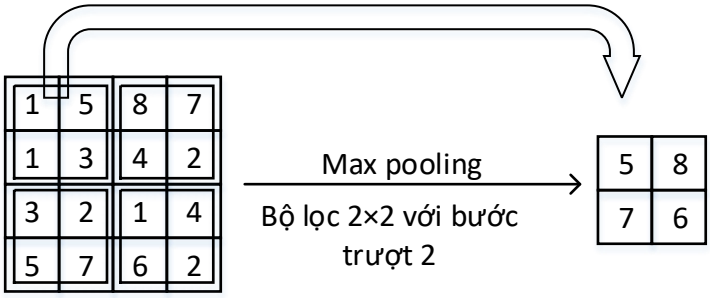


Hình 4.2: Mô tả tích chập ảnh với bộ lọc (kernel).

Các mạng nơ-ron tích chập điển hình có 3 lớp cơ bản. Lớp chập thực hiện phép tích chập tạo ra tập có giá trị tích cực tuyến tính trong các feature map. Các giá trị tuyến tính trong feature map được cho qua các hàm kích hoạt phi tuyến như hàm ReLU. Lớp này còn được gọi là tầng detector. Lớp thứ 2 trong mô hình là lớp gộp (pooling) được dùng để hiệu chỉnh ngõ ra. Các hàm Pooling đặt ngõ ra của lớp feature map trong mối liên hệ với các ngõ ra lân cận. Các hàm Pooling thường được sử dụng trong các mạng nơ-ron tích chập là hàm Max Pooling, Average Pooling

hoặc hàm Sum Pooling. Hàm Max Pooling được sử dụng phổ biến trong các mạng nơ-ron tích chập ứng dụng trong nhận dạng, phân loại ảnh. Trong tất cả các trường hợp, Pooling tạo ra các giá trị biểu diễn cho dữ liệu tương đối bất biến đối với sự dịch chuyển của dữ liệu ngõ vào. Tính bất biến đối với sự dịch chuyển được hiểu nếu chúng ta dịch chuyển một lượng nhỏ các giá trị đầu vào, các giá trị ngõ ra của lớp Pooling hầu như có thể được giữ không đổi. Tính bất biến đối với các dịch chuyển nội bộ có thể rất hữu ích nếu chúng ta quan tâm tới việc một vài đặc điểm có mặt hơn là xác định chính xác nó ở đâu. Ví dụ khi chúng ta xác định có sự tồn tại của khuôn mặt trong ảnh hay không, chúng ta không nhất thiết phải biết vị trí chính xác của mắt, chúng ta chỉ cần nhận biết có mắt ở vị trí bên trái và bên phải của khuôn mặt.

- Max Pooling: Hàm Max Pooling chọn giá trị cao nhất trong cửa sổ lân cận. Trong trường hợp này chúng ta không quan tâm vị trí giá trị cao nhất ở đâu. Chính vì thế, khi ngõ vào dịch chuyển nhỏ, ngõ ra vẫn không thay đổi. Ví dụ mô tả hoạt động của Max Pooling được trình bày trong hình 4.3.

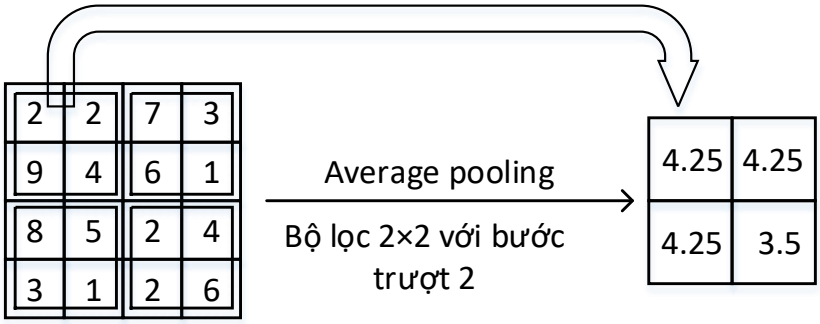


Hình 4.3: Max Pooling với của sổ 2×2 và bước trượt 2.

Hình 4.3 mô tả hoạt động của Max Pooling với cửa sổ 2×2 và bước trượt bằng 2. Khác với hoạt động tích chập, các bước trượt độc lập với kích thước của bộ lọc và thông thường chọn bước trượt là 1. Trong lớp Max Pooling, mục đích chúng ta đi tìm một số đặc điểm nào đó trong cửa sổ hơn là xác định vị trí của nó. Chúng ta tìm giá trị lớn nhất trong cửa sổ mà không quan tâm giá trị lớn nhất đó nằm ở vị trí nào. Bước trượt cho lớp Max Pooling thường được chọn bằng với kích thước của cửa sổ, tức là không có sự trùng lắp. Do đó, kết quả thu được ở Hình 4.3 là một ma trận có kích thước bằng một nửa ma trận ban đầu.

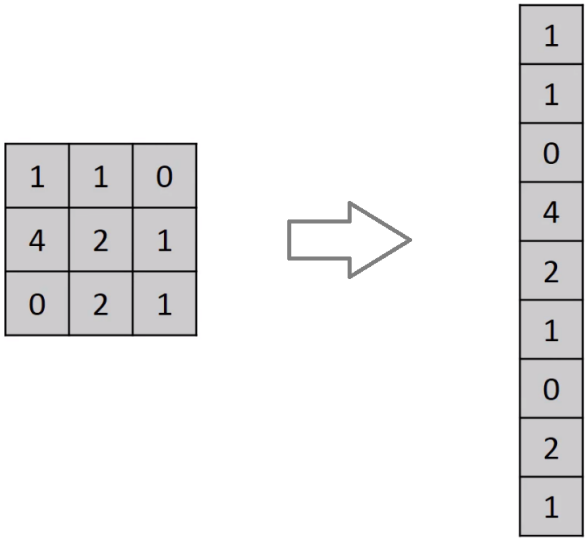
- Average Pooling: Hàm trung bình được sử dụng cho lớp Pooling tương tự như hàm Max. Trong đó, giá trị lựa chọn là giá trị trung bình

của các phần tử trong của sổ. Hoạt động của Average Pooling được mô tả trong Hình 4.4.



Hình 4.4: Average Pooling.

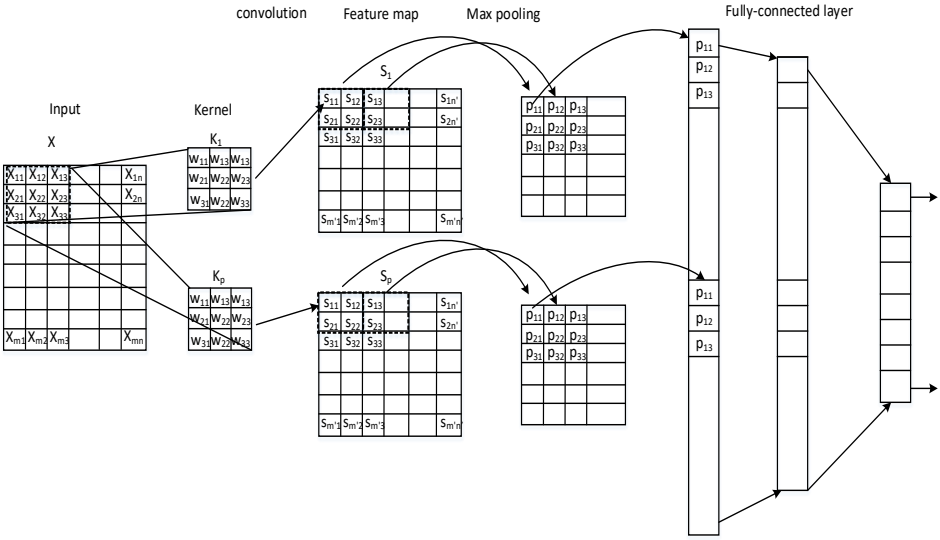
Lớp cuối cùng trong các mạng nơ-ron tích chập là các lớp kết nối đủ sử dụng mạng nhiều lớp với hàm tích cực Softmax cho lớp nơ-ron ngõ ra. Ngõ vào của các lớp kết nối đủ là các vector một chiều. Do đó, các ma trận lớp pooling cuối cùng được chuyển thành vector và đưa đến lớp kết nối đủ như được minh họa trong hình 4.5.



Hình 4.5: Các ma trận được chuyển thành vector ngõ vào cho lớp kết nối đủ.

4.2 HUẤN LUYỆN MẠNG NƠ-RON TÍCH CHẬP

Một mạng nơ-ron tích chập với 1 lớp chập, 1 lớp gộp và lớp kết nối đầy đủ được minh họa chi tiết trong Hình 4.6.



Hình 4.6: Mô tả chi tiết một mạng nơ-ron tích chập với 1 lớp chập, 1 lớp gộp và lớp kết nối đủ.

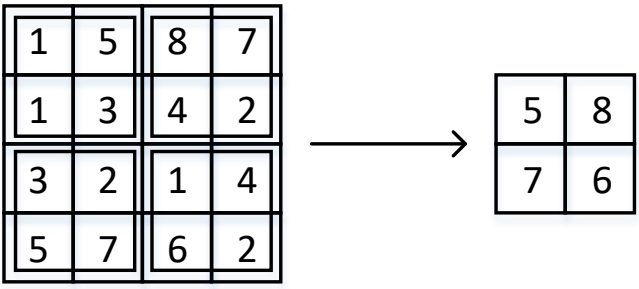
Ngõ vào được biểu diễn dưới dạng các ma trận 2 chiều. Phép tính tích chập được thực hiện trên các ảnh sử dụng các bộ lọc để tạo ra các bản đồ đặc trưng (Feature map) tương ứng. Các Feature map có cùng kích thước với ảnh ngõ vào nếu như các ảnh ngõ vào được thêm các biên trước khi thực hiện phép toán tích chập hoặc có kích thước nhỏ hơn. Phép tích tích chập thực hiện trên các ma trận ngõ vào với các bộ lọc (filter hay kernel) trong mạng CNN thực chất là phép tương quan chéo (Cross-correlation). Tuy nhiên, sự khác biệt giữa phép tương quan chéo và phép tích chập chỉ đơn thuần là bộ lọc được lật (flipped kernel) hoặc giữ nguyên trước khi thực hiện phép toán. Trong phép tương quan chéo, chúng ta tính tổng của phép nhân các phần tử của bộ lọc và ảnh khi bộ lọc trượt hết ảnh, trong khi đó với phép tích chập chúng ta lật bộ lọc 180 độ trước khi thực hiện tính tổng của phép nhân các phần tử bộ lọc và ảnh khi bộ lọc trượt hết ảnh. Trong các mạng nơ-ron tích chập, các bộ lọc thu được trong quá trình huấn luyện, do đó thông thường chúng ta sử dụng tương quan chéo (cross-correlation) trong các mạng CNN thay vì tích chập (convolution). Việc sử dụng tích chập hay tương quan chéo có thể phụ thuộc vào quá trình huấn luyện, do nó sử dụng tích chập hay tương quan chéo không làm thay đổi quá trình thực hiện của mô hình. Việc sử dụng phép tương quan chéo làm cho chương trình đơn giản hơn.

Ma trận bản đồ đặc trưng (Feature map) được tính theo công thức tổng quát như sau:

$$s_{ij} = \sum_{m=1}^{2} \sum_{n=1}^{2} w_{(m)(n)} \cdot a_{(i-1+m)(j-1+n)} \tag{4.1}$$

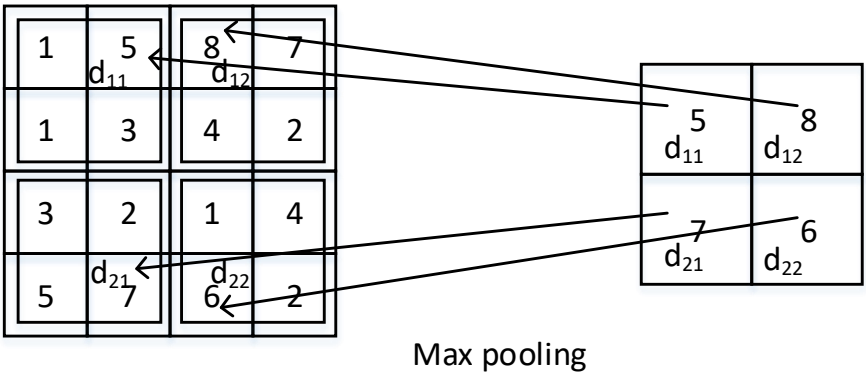
Trong các mạng CNN, các nơ-ron trong lớp chập sử dụng hàm kích hoạt là các đơn vị tuyến tính chỉnh lưu (ReLU) như đã được giới thiệu trong Chương 1.

Lớp gộp (Pooling) làm giảm kích thước Feature map đi một nửa bằng cách sử dụng các cửa sổ 2×2. Trong lớp gộp có thể sử dụng phương pháp lấy giá trị trung bình các phần tử trong cửa sổ (Average pooling) hoặc lấy giá trị lớn nhất của các phần tử trong cửa sổ (Max pooling). Ngõ ra lớp Max pooling được tính như sau:



Hình 4.7: Ngõ ra lớp Max pooling.

Các giá trị ma trận thu được sau khi thực hiện gộp được chuyển thành vector và làm ngõ vào cho mạng kết nối đủ. Quá trình huấn luyện mạng CNN cũng dựa trên các phương pháp giảm dần theo độ dốc của hàm mất mát. Hàm mất mát được xác định dựa vào lớp ngõ ra của lớp kết nối đủ. Các sai số được lan truyền ngược qua lớp kết nối đủ, về lớp gộp, và lớp chập để thực hiện việc cập nhật trọng số. Hàm mất mát và quá trình lan truyền ngược trong lớp kết nối đủ đã được thảo luận ở chương trước. Trong phần này chúng ta giả sử giá trị sai số đã được lan truyền về đến ngõ vào lớp kết nối đủ. Các sai số này sau đó được sắp xếp lại dưới dạng các ma trận 2 chiều phù hợp với các ma trận ngõ ra của lớp gộp và tiếp tục lan truyền về lớp Feature map. Quá trình lan truyền ngược qua lớp gộp được minh họa trong Hình 4.8.



Hình 4.8: Quá trình lan truyền ngược qua lớp gộp.

Các phần tử ngõ ra của lớp gộp là phần tử lớn nhất trong từng cửa sổ khi trượt hết Feature map. Trong mỗi cửa sổ trượt, chỉ có phần tử có giá trị lớn nhất ảnh hưởng đến sai số ngõ ra của mạng. Do đó, các sai số trong lớp gộp sẽ lan truyền về lớp chập trong từng cửa sổ tại những vị trí có giá trị lớn nhất. Các vị trí còn lại sẽ nhận giá trị sai số bằng 0.

Việc lan truyền các sai số trong lớp gộp có thể thực hiện bằng ngôn ngữ Python như sau với thư viện numpy. Giả sử fm là feature map sau khi thực hiện tích chập, mp là ma trận thu được tại lớp gộp, vị trí giá trị lớn nhất trong fm được tô đậm trong (4.2), dmp là ma trận các sai số tại lớp gộp được sắp xếp lại từ ngõ vào lớp kết nối đầy đủ. Sử dụng phép toán Kronecker cho phép tính lan truyền ngược các giá trị lỗi từ lớp gộp về lớp tích chập.

$$fm = \begin{bmatrix} \mathbf{94} & 18 & \mathbf{80} & 49 & 11 & 55 \\ 86 & 78 & 41 & 41 & \mathbf{57} & 31 \\ 5 & \mathbf{99} & 56 & \mathbf{79} & \mathbf{95} & 21 \\ 14 & 9 & 45 & 68 & 20 & 60 \\ 55 & 73 & 65 & 19 & \mathbf{44} & 9 \\ \mathbf{75} & 28 & \mathbf{88} & 38 & 32 & 3 \end{bmatrix}$$

$$mp = \begin{bmatrix} 94 & 80 & 57 \\ 99 & 79 & 95 \\ 75 & 88 & 44 \end{bmatrix} \tag{4.2}$$

$$dmp = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{bmatrix}$$

Sử dụng toán tử Kronecker, nhân Kronecker ma trận dmp, mp với ma trận cửa sổ 2x2 để nhân đôi kích thước ma trận dmp, mp như sau:

```
dp = np.kron(dmp,np.ones((2,2)))
print(dp)
fm1 = np.kron(mp,np.ones((2,2)))
print(dp)
```

```
dp =
[[0.1 0.1 0.2 0.2 0.3 0.3]
 [0.1 0.1 0.2 0.2 0.3 0.3]
 [0.4 0.4 0.5 0.5 0.6 0.6]
 [0.4 0.4 0.5 0.5 0.6 0.6]
 [0.7 0.7 0.8 0.8 0.9 0.9]
 [0.7 0.7 0.8 0.8 0.9 0.9]]
fm1=
[[94. 94. 80. 80. 57. 57.]
 [94. 94. 80. 80. 57. 57.]
 [99. 99. 79. 79. 95. 95.]
 [99. 99. 79. 79. 95. 95.]
 [75. 75. 88. 88. 44. 44.]
 [75. 75. 88. 88. 44. 44.]]
```

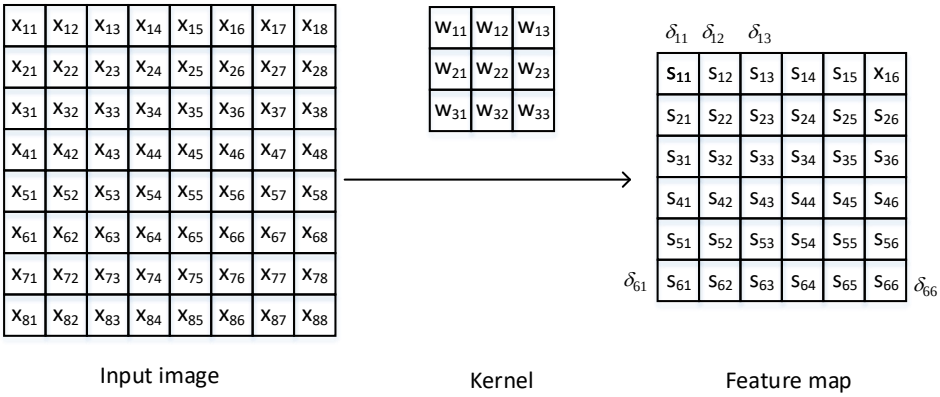
Các giá trị sai số tại vị trí các giá trị lớn nhất trong cửa sổ được lan truyền từ lớp gộp, trong khi đó sai số tại các giá trị khác sẽ được thiết lập về 0 vì chỉ có các giá trị có giá trị lớn nhất ảnh hưởng đến sai số của lớp ngõ ra. So sánh với các giá trị trong fm để xoá các sai số tại các giá trị không phải lớn nhất trong cửa sổ về 0.

```
dconv=np.multiply(np.equal(fm,fm1)*1,dp)
print(dconv)
```

```
[[0.1 0.  0.2 0.  0.  0. ]
 [0.  0.  0.  0.  0.3 0. ]
 [0.  0.4 0.  0.5 0.6 0. ]
 [0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.9 0. ]
 [0.7 0.  0.8 0.  0.  0. ]]
```

Sai số tại ngõ ra các lớp gộp được lan truyền ngược lên các Feature map như đã trình bày ở trên. Đối với mỗi Feature map được tạo ra, khi

thực hiện phép tương quan chéo từ ảnh ngõ vào với một bộ lọc sẽ tồn tại một ma trận các giá trị sai số được lan truyền ngược từ lớp gộp như mô tả trong hình 4.9.



Hình 4.9: Biểu diễn sai số trong lớp gộp

Đối với mỗi Feature map chúng ta có:

$$\frac{\partial C}{\partial s_{11}} = \delta_{11} \rightarrow \frac{\partial C}{\partial w_{11}} = \delta_{11}.x_{11} \tag{4.3}$$

Khi bộ lọc K trượt sang phải 1 đơn vị, x_{12} được đặt trùng với w_{11}, chúng ta có:

$$\frac{\partial C}{\partial s_{12}} = \delta_{12} \rightarrow \frac{\partial C}{\partial w_{11}} = \delta_{12}.x_{12} \tag{4.4}$$

Tương tự khi bộ lọc trược đến vị trí cuối cùng chúng ta có:

$$\frac{\partial C}{\partial s_{lk}} = \delta_{lk} \rightarrow \frac{\partial C}{\partial w_{11}} = \delta_{lk}.x_{lk} \tag{4.5}$$

Trong đó giả sử kích thước của Feature map la $l \times k$, kích thước của ảnh ngõ vào là $m \times n$, chúng ta có:

$$\frac{\partial C}{\partial w_{11}} = \begin{bmatrix} x_{11} & \cdots & x_{1k} \\ \vdots & \ddots & \vdots \\ x_{l1} & \cdots & x_{lk} \end{bmatrix} \circ \begin{bmatrix} \delta_{11} & \cdots & \delta_{1k} \\ \vdots & \ddots & \vdots \\ \delta_{l1} & \cdots & \delta_{lk} \end{bmatrix} \tag{4.6}$$

Trong đó ° là tích vô hướng của 2 ma trận; l, k là kích thước của Feature map có thể được tính từ m, n như sau:

$$\begin{aligned} l &= \frac{m - K + 2P}{S} + 1 \\ k &= \frac{n - K + 2P}{S} + 1 \end{aligned} \tag{4.7}$$

Với m, n là kích thước ảnh đầu vào, K là kích thước bộ lọc, P là kích thước biên được gán giá trị 0 thêm vào ảnh đầu vào và S là bước trượt.

Tại vị trí đầu tiên khi thực hiện tích chập bộ lọc với ảnh chúng ta cũng thu được:

$$\frac{\partial C}{\partial s_{11}} = \delta_{11} \rightarrow \frac{\partial C}{\partial w_{12}} = \delta_{11}.x_{12} \tag{4.8}$$

Khi bộ lọc K trượt sang phải 1 đơn vị, x_{12} được đặt trùng với w_{11}, chúng ta có:

$$\frac{\partial C}{\partial s_{12}} = \delta_{12} \rightarrow \frac{\partial C}{\partial w_{12}} = \delta_{12}.x_{13} \tag{4.9}$$

Tương tự khi bộ lọc trược đến vị trí cuối cùng chúng ta có:

$$\frac{\partial C}{\partial s_{lk}} = \delta_{lk} \rightarrow \frac{\partial C}{\partial w_{lk}} = \delta_{lk}.x_{lk+1} \tag{4.10}$$

$$\frac{\partial C}{\partial w_{12}} = \begin{bmatrix} x_{11} & \cdots & x_{1k+1} \\ \vdots & \ddots & \vdots \\ x_{l1} & \cdots & x_{lk+1} \end{bmatrix} \circ \begin{bmatrix} \delta_{11} & \cdots & \delta_{1k} \\ \vdots & \ddots & \vdots \\ \delta_{l1} & \cdots & \delta_{lk} \end{bmatrix} \tag{4.11}$$

Một cách tổng quát chúng ta có:

$$\begin{bmatrix} \frac{\partial C}{\partial w_{11}} & \frac{\partial C}{\partial w_{12}} & \frac{\partial C}{\partial w_{13}} \\ \frac{\partial C}{\partial w_{21}} & \frac{\partial C}{\partial w_{22}} & \frac{\partial C}{\partial w_{23}} \\ \frac{\partial C}{\partial w_{31}} & \frac{\partial C}{\partial w_{32}} & \frac{\partial C}{\partial w_{33}} \end{bmatrix} = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix} \times \begin{bmatrix} \delta_{11} & \cdots & \delta_{1k} \\ \vdots & \ddots & \vdots \\ \delta_{l1} & \cdots & \delta_{lk} \end{bmatrix} \tag{4.12}$$

Trong đó phép toán "×" là phép tương quan chéo (cross-correlation).

Quá trình cập nhật trọng số cho các bộ lọc được thực hiện tương tự như các trọng số mạng kết nối đầy đủ.

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial C}{\partial w_{ij}} \tag{4.13}$$

Chương trình huấn luyện mạng nơ-ron tích chập với lớp ngõ vào kích thước 28×28 cho các ngõ vào là các ảnh của ký tự viết tay. Mô hình bao gồm 1 lớp tích chập với 16 bộ lọc kích thước 5×5, hàm kích hoạt sử dụng đơn vị chỉnh lưu tuyến tính (ReLU), lớp gộp sử dụng Max pooling, lớp kết nối đủ bao gồm 512 nơ-ron ở lớp ẩn và 10 nơ-ron ở lớp ngõ ra sử dụng hàm Softmax. Trong ví dụ này các ảnh được tải về và được lưu

dưới dạng các ma trận có kích thước 60,000×28×28. Ngoài thư viện numpy, chúng ta sử dụng thư viện *scipy* và *ckimage* để hỗ trợ hàm Max pooling và phép tương quan chéo.

```
import datetime as dt
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import matplotlib.pyplot as plt
import datetime as dt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=x_train/255.0;
x_test=x_test/255.0;
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
print(x_train.shape)
im_size = (28,28)
kernels_size = (16,5,5)
ConvolutionSize=(kernels_size[0],(im_size[0]-
kernels_size[1]+1),(im_size[0]-kernels_size[2]+1))
PoolingSize = (kernels_size[0],np.int((im_size[0]-
kernels_size[1]+1)/2),np.int((im_size[0]-
kernels_size[2]+1)/2))
flatten_size = np.int(((im_size[0]-
kernels_size[1]+1)/2)*((im_size[0]-
kernels_size[1]+1)/2)*kernels_size[0])
kernels = np.random.uniform(-0.5,0.5,kernels_size)
# define network parameters
learningRate = 0.1
BatchSize= 128
Epoch=10
Momentum=0.95
NumOfTrainSample=60000
NumOfTestSample=1000
```

```
NumInput=flatten_size
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
0.5,0.5,(NumHidden,NumInput)))
bh= np.random.uniform(0,0.5,(1,NumHidden))
del_Wh= np.zeros((NumHidden,NumInput))
del_bh= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-0.5,0.5,(NumOutput,NumHidden))
bo= np.random.uniform(0,0.5,(1,NumOutput))
del_Wo= np.zeros((NumOutput,NumHidden))
del_bo= np.zeros((1,NumOutput))
from scipy import signal
import skimage.measure       # install sckit-
image: pip3 install scikit-image
def ReLU(x):
    return np.maximum(x,0)
def sigmoid(x):
    return 1./(1.+np.exp(-x))
def softmax(x):
    return np.divide(np.matrix(np.exp(x)),np.mat(np.sum(
np.exp(x),axis=1)))
def AccTest(outN,labels):    # calculate the matching sc
ore
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,kernels,Wh,bh,Wo,bo):
    x,_,_=ConvolutionLayer(samples,kernels)
    OutH1=sigmoid(np.dot(x,Wh.T)+bh)
    OutN=softmax(np.dot(OutH1,Wo.T)+bo)
    return OutN
def ConvolutionLayer(x,kernels):
```

```
    Conv =np.empty([x.shape[0],kernels.shape[0],24,24])
    Pool =np.empty([x.shape[0],kernels.shape[0],12,12])
    PLFlatten =np.empty([x.shape[0],12*12*kernels.shape[
0]])
    for j in range (x.shape[0]):
        for i in range(kernels.shape[0]):
            Conv[j,i,:,:] = signal.correlate2d(x[j,:,:],
 kernels[i,:,:], mode ='valid', boundary='symm')
            Fmap=ReLU(Conv[j,i,:,:])
            Pool[j,i,:,:]=skimage.measure.block_reduce(F
map, (2,2), np.max)
        PLFlatten[j,:]= np.ndarray.flatten(Pool[j])


    return  PLFlatten, Pool,Conv


# Train the network with
back propagation, stoshastic Gradient descent
SampleIdx=np.arange(NumOfTrainSample)
Cost_Entropy=np.zeros(Epoch)
Acc=np.zeros(Epoch)
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSize)
))
IdxCost=0;
t_start=t1=dt.datetime.now()
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]
        #print(Batch_sample)
        sample=(x_train[Batch_sample,:,:])
        targ=np.matrix(y_train [Batch_sample,:])
        # Feedforward propagation,
```

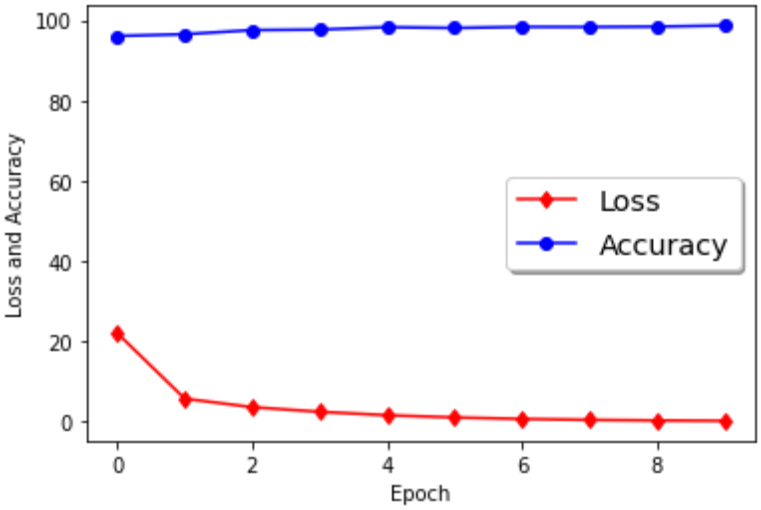
```
        x,pool,conv=ConvolutionLayer(sample,kernels)
        ah=sigmoid(np.dot(x,Wh.T)+bh)
        ao =softmax(np.dot(ah,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=-
np.sum(np.multiply(targ,np.log10(ao)))
        IdxCost+=1;
        do=(targ-ao)
        dWo=np.matrix(np.dot(do.T,ah)/BatchSize)
        dbo=np.mean(do,0)
        #back propagate error
        temp=np.dot(do,Wo)
        temp1=np.multiply(ah,(1-ah))
        dh=np.multiply(temp,temp1)
        dwh1=np.dot(dh.T,x)/BatchSize
        dbh1=np.mean(dh,0)
        # Update weight
        WoUpdate=learningRate*dWo  + Momentum*del_Wo
        boUpdate= learningRate*dbo + Momentum*del_bo
        del_Wo=WoUpdate;
        del_bo=boUpdate;
        WhUpdate=learningRate*dwh1 + Momentum*del_Wh
        bhUpdate=learningRate*dbh1 + Momentum*del_bh
        del_Wh=WhUpdate;
        del_bh=bhUpdate;
        # update
        Wo=Wo+ WoUpdate
        #print(WoUpdate)
        bo+=boUpdate
        Wh=Wh+ WhUpdate
        bh+=bhUpdate
        #-----------------------------------------
        #update for convolution layer,
        #back propagate error to the convolution layer
        dflatten=np.dot(dh,Wh) # assume the the derivati
```

```
ve of ReLU is 1 (possitive case)
        dkernels = np.zeros(kernels.shape)
        for m in range (BatchSize):
            #reshape the flattened data
            dflattened=dflatten[m,:]
            dpooling=np.array(dflattened).reshape(kernel
s.shape[0],12,12)
            #back propagate error to convolution layer
            mpool = np.kron(pool[m],np.ones((2,2)))
            dp = np.kron(dpooling,np.ones((2,2)))
            dconv=np.multiply(np.equal(conv[m],mpool)*1,
dp)
            for j in range(kernels.shape[0]):
                dkernels[j]=dkernels[j]+ signal.correlat
e2d(sample[m], dconv[j], mode ='valid', boundary='symm')
        kernels = kernels + learningRate*dkernels/BatchS
ize
    Cost_Entropy[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
    print("Cross-Entropy %f" % Cost_Entropy[ep])
    #test the model
    RealOutN=feedforward(x_test,kernels,Wh,bh,Wo,bo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=t1=dt.datetime.now()
print("Total time : ")
print(t_end-t_start)
plt.plot(Cost_Entropy,"dr-",label='Loss')
plt.plot(Acc*100,"ob-",label='Accuracy')
plt.ylabel('Loss and Accuracy')
plt.xlabel('Epoch')
legend = plt.legend(loc='center right', shadow=True, fon
```

```
tsize='x-large')
plt.show()
```

```
Cross-Entropy 22.977073
Accuracy: 0.956500
0:05:27.294705
Training epoch 1
Cross-Entropy 6.464964
Accuracy: 0.968600
0:05:26.583689
Training epoch 2
Cross-Entropy 4.449640
Accuracy: 0.977000
0:05:28.562441
Training epoch 3
Cross-Entropy 2.909645
Accuracy: 0.981200
0:05:29.482621
Training epoch 4
Cross-Entropy 2.243753
Accuracy: 0.974600
0:05:29.451650
Training epoch 5
Cross-Entropy 1.524023
Accuracy: 0.983300
0:05:29.035859
Training epoch 6
Cross-Entropy 0.959882
Accuracy: 0.983400
0:05:29.881670
Training epoch 7
Cross-Entropy 0.730163
Accuracy: 0.985400
0:05:30.250497
Training epoch 8
Cross-Entropy 0.460981
Accuracy: 0.983500
```

```
0:05:33.949041
Training epoch 9
Cross-Entropy 0.296893
Accuracy: 0.986600
Total time :
1:00:50.478084
```



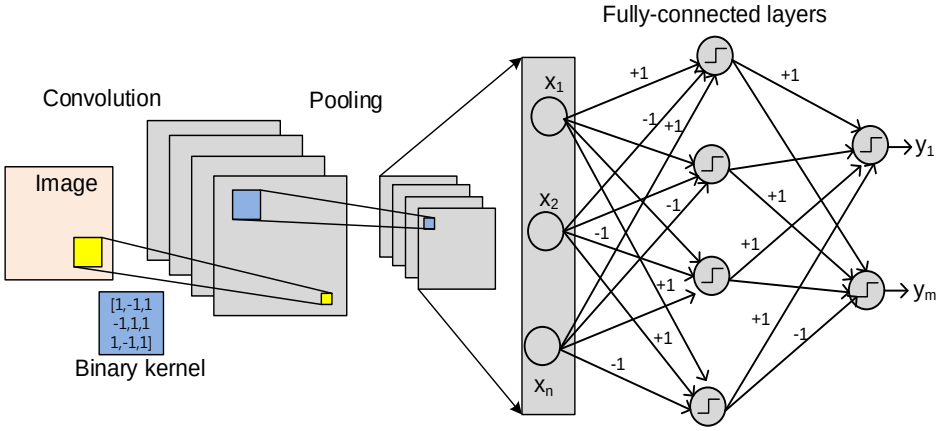
Mô hình mạng nơ-ron tích chập cho độ chính xác cao hơn mạng nơ-ron với 1 lớp ngõ vào, 1 lớp ẩn và 1 lớp ngõ ra như các ví dụ trước. Tuy nhiên huấn luyện các mạng nơ-ron tích chập mất nhiều thời gian hơn. Trong các ứng dụng học sâu, các giải thuật được lập trình và thực thi trên các kiến trúc xử lý song song GPU (Graphic Processing Unit) cho phép rút ngắn thời gian huấn luyện cũng như thời gian thực thi của các mạng tích chập.

Chương 5
LƯỢNG TỬ HÓA CÁC THÔNG SỐ MÔ HÌNH MẠNG NƠ-RON

5.1 MẠNG NƠ-RON VỚI TẬP TRỌNG SỐ NHỊ PHÂN

Các mạng học sâu (Deep Neural network) hiện nay khá hiệu quả trong các ứng dụng thông minh nhân tạo. Tuy nhiên, các mạng nơ-ron học sâu sử dụng nhiều thông số và số lượng các phép tính toán lớn đòi hỏi các máy tính thực thi các mạng nơ-ron học sâu phải có cấu hình mạnh và bộ nhớ lớn. Ứng dụng các mạng nơ-ron nhiều lớp trong các thiết bị đầu cuối của hệ thống IoT hoặc các thiết bị di động có phần bị hạn chế do tài nguyên và tốc độ của các thiết bị này không đáp ứng được yêu cầu về bộ nhớ và tốc độ xử lý của các mạng nhiều lớp. Việc tối ưu các mạng nhiều lớp để thực thi trên các thiết bị có cấu hình thấp vẫn luôn được quan tâm nghiên cứu trong thời gian gần đây. Trong các mô hình tối ưu mạng nhiều lớp chúng ta có thể kể đến là các phương pháp tối ưu tập trọng số và quá trình tính toán thông qua quá trình lượng tử hoá tập trọng số. Phương pháp lượng tử hoá tập trọng số là cho phép giảm kích thước bộ nhớ lưu trữ tập trọng số đồng thời đơn giản hoá quá trình tính toán. Mỗi trọng số được biểu diễn thông qua một ô nhớ 32 hoặc 64 bit. Lượng tử hoá trọng số cho phép biểu diễn các trọng số với các giá trị có độ dài bit ít hơn. Ví dụ như một mạng nơ-ron trong đó các giá trị trọng số được biểu diễn bởi các giá trị nhị phân. Mạng nơ-ron với các giá trị trọng số nhị phân có thể được gọi là mạng nơ-ron nhị phân. Mạng nơ-ron nhị phân về cấu trúc tương tự như các mạng nơ-ron thông thường khác. Tuy nhiên, các mạng nơ-ron thông thường sử dụng 32 bit hoặc 64 bit để lưu các giá trị trọng số vì các giá trị trọng số này là các số thực. Trong mạng nơ-ron nhị phân, các trọng số được chuyển về dạng nhị phân (0, 1) hoặc (-1, +1). Ngõ ra của các hàm kích hoạt của nơ-ron cũng được thể hiện dưới dạng các giá trị nhị phân. Bằng cách sử dụng các giá trị nhị phân biểu diễn trọng số, các mạng nơ-ron nhị phân sử dụng bộ nhớ ít hơn để lưu giá trị trọng số. Hơn nữa, các phép toán trên các giá trị nhị phân sử dụng tài nguyên và thời gian thực thi ít hơn so với thực thi trên các giá trị 32 bit hoặc 64 bit. Các mạng nơ-ron nhị phân là một cấu trúc lý tưởng cho việc triển khai các ứng dụng mạng nơ-ron nhiều lớp trên các thiết bị có cấu hình thấp như các hệ thống nhúng (embedded system), các máy tính nhỏ (mini-computer) phục vụ cho các ứng dụng có tính lưu động như

các thiết bị dân dụng, robot, các thiết bị đầu cuối trong các hệ thống IoT. Mô hình mạng nơ-ron nhị phân có thể được triển khai cho các mạng nhiều lớp và cả các mạng nơ-ron tích chập. Một mạng nơ-ron nhị phân được minh họa như hình sau:



Hình 5.1: Mạng nơ-ron tích chập sử dụng tập trọng số nhị phân.

Hình 5.1 minh hoạ một mạng nơ-ron tích chập trong đó các trọng số cho các lớp chập và các lớp kết nối đủ đều được biểu diễn dưới dạng các giá trị nhị phân. Phương pháp huấn luyện mạng nơ-ron với giải thuật giảm dần độ dốc của hàm mất mát được thực hiện trên tập trọng số là số thực. Phương pháp cập nhật trọng số gặp khó khăn khi triển khai cho các mạng nơ-ron mà trong đó các tập trọng số là các giá trị nhị phân bởi vì giá trị cập nhật có thể là rất nhỏ. Để huấn luyện mạng nơ-ron nhị phân có thể sử dụng một số phương pháp riêng biệt, tuy nhiên nó cũng dựa trên nguyên lý của Gradient Descent. Một cách đơn giản nhất là có thể dùng phương pháp huấn luyện mạng nơ-ron nhiều lớp với giải thuật Gradient Descent. Tập trọng số thu được 32 bit sau đó được lượng tử hóa sang nhị phân sử dụng số bit ít hơn, cụ thể trong trường hợp này là 1 bit. Việc lượng tử hóa như vậy đương nhiên sẽ làm cho sai số lớn và dẫn đến độ chính xác của mô hình giảm đi đáng kể. Các tập trọng số nhị phân được biểu diễn bởi các giá trị +1 và -1. Các giá trị có dấu -1 được sử dụng để mã hóa cho các giá trị 0. Ngõ ra của nơ-ron là tổ hợp vector của các ngõ vào, cụ thể là tổng của các ngõ vào khi chúng được nhân với một trọng số tương ứng. Việc thực hiện các phép nhân và phép cộng sử dụng khá nhiều tài nguyên của hệ thống. Trong các mạng nơ-ron nhị phân, các phép toán nhân có được thực hiện thông qua các phép toán xử lý bit. Sử dụng các phép toán xử lý bit cho phép chương trình được thực thi nhanh hơn.

Đối với các ngõ vào là các giá trị nhị phân, việc thực thi các phép toán logic hiệu quả hơn so với thực thi các phép nhân số thực 32 bit. Tổng các giá trị ngõ vào nhân với trọng số tương ứng được biểu diễn thông qua số lượng bit có giá trị bằng 1. Phép cộng được thực hiện bằng cách đếm giá trị bit 1 trong thanh ghi kết quả.

5.2 HUẤN LUYỆN MẠNG NƠ-RON NHỊ PHÂN

Việc huấn luyện mạng nơ-ron nhị phân tương đối khó hơn so với mạng nơ-ron thông thường trong đó sử dụng tập trọng số là số thực. Phương pháp đơn giản để huấn luyện mạng nơ-ron nhị phân là huấn luyện mạng nơ-ron với tập trọng số là số thực để đạt giá trị độ chính xác mong muốn và lượng tử hóa tập trọng số sau khi mạng nơ-ron huấn luyện kết thúc. Việc này đương nhiên làm cho độ chính xác giảm đáng kể.

Trong một số phương pháp khác, các giá trị nhị phân sẽ được sử dụng trong quá trình lan truyền ngược với giải thuật Gradient Descent. Việc này sẽ đem lại kết quả tốt hơn so với việc lượng tự hóa các tập trọng số sau khi đã huấn luyện xong. Courbariaux đề xuất sử dụng các giá trị trọng số nhị phân kết hợp với các tập trọng số là số thực trong quá trình huấn luyện với Gradient Descent. Việc tính toán các giá trị Gradient đối với các trọng số nhị phân tương tự như các tập trọng số sử dụng số thực, tuy nhiên việc cập nhật trọng số sử dụng Gradient Descent dường như không khả thi đối với tập trọng số nhị phân vì phương pháp Gradient Descent thông thường tạo ra các giá trị cập nhật trọng số rất nhỏ. Các giá trị này có thể nhỏ hơn đối với các nơ-ron ở các lớp đầu tiên qua quá trình lan truyền ngược sai số qua nhiều lớp ẩn sử dụng các hàm kích hoạt phi tuyến như hàm Sigmoid. Để giải quyết vấn đề cập nhật trọng số nhị phân sử dụng Gradient, Courbariaux đề xuất phương pháp vẫn giữ các giá trị trọng số thực W_R, các tập trọng số này được nhị phân hóa trong mạng để thu được tập trọng số nhị phân, W_B. Tập trọng số thực W_R sau đó được cập nhật thông qua quá trình lan truyền ngược sai số và phương pháp Gradient Descent. Tập trọng số thực W_R chỉ được sử dụng trong quá trình huấn luyện. Trong quá trình hoạt động (Inference), tập trọng số W_R không cần thiết và chỉ sử dụng tập trọng số nhị phân, W_B. Do đó chỉ tập trọng số nhị phân được lưu và sử dụng cho mạng sau khi kết thúc quá trình huấn luyện. Quá trình nhị phân hoá sử dụng hàm dấu như minh hoạ trong biểu thức (5.1). Kết quả các trọng số W_B có giá trị 1 hoặc -1.

$$W_B = sign(W_R) \tag{5.1}$$

Tính các giá trị Gradient cho các hàm dấu là vô nghĩa vì các giá trị Gradient của hàm dấu bằng 0 hoặc không được định nghĩa. Trong huấn

luyện mạng nơ-ron nhị phân, chúng ta ước lượng giá trị Gradient cho các trọng số nhị phân bằng các chuyển toàn bộ Gradient của trọng số thực sang trọng số nhị phân, thực hiện theo công thức sau:

$$\frac{\partial L}{\partial W_R} = \frac{\partial L}{\partial W_B} \tag{5.2}$$

Các hàm kích hoạt cũng được nhị phân hoá. Việc nhị phân hoá các hàm kích hoạt được thực hiện đơn giản bằng các hàm dấu, tương tự như cách thực hiện trên các trọng số. Hàm dấu (sign function) được sử dụng thay cho các hàm phi tuyến khi thực thi hàm kích hoạt trong mô hình mạng nơ-ron nhị phân. Để tối ưu mô hình, các Gradient cần được loại bỏ trong quá trình lan truyền ngược nếu như ngõ vào của hàm kích hoạt có giá trị quá lớn.

Một ví dụ về chương trình huấn luyện mạng nơ-ron với tập trọng số nhị phân và hàm kích hoạt là hàm dấu được viết như sau:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
# reshape the input data
x_train=np.reshape(x_train,(60000,784))
x_test= np.reshape(x_test,(10000,784))
#Convert to binary
x_train[x_train<=128] = 0
x_train[x_train>128] = 1
x_test[x_test<=128]= 0
x_test[x_test>128]= 1
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
y_train[y_train==0]= -1
y_test[y_test==0]= -1
print("----------------------------------")
import datetime as dt
```

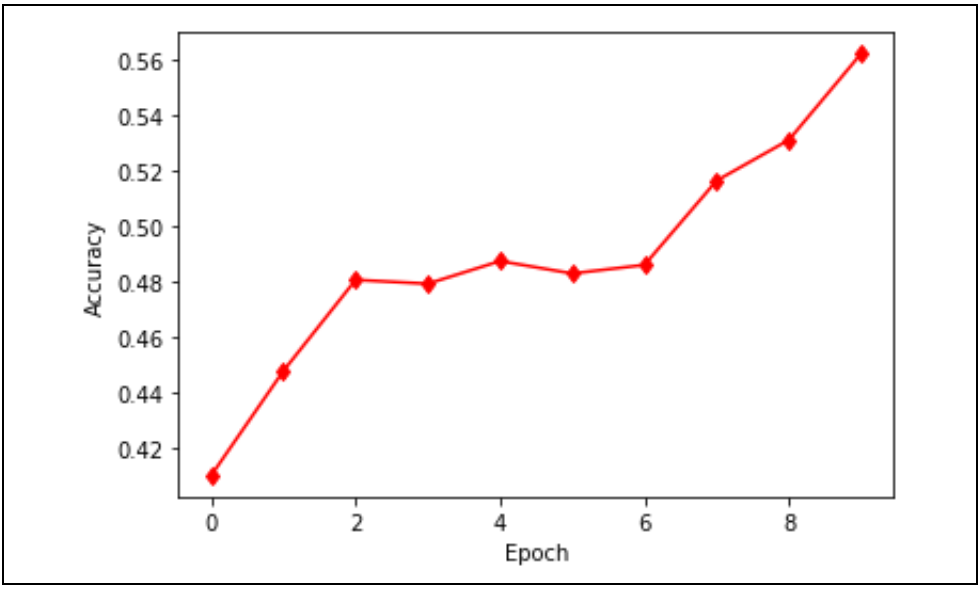
```
def AccTest(outN,labels):    # calculate the matching sc
ore
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,Wh,bh,Wo,bo):
    OutH1=np.sign(np.dot(samples,Wh.T)+bh)
    OutN=np.sign(np.dot(OutH1,Wo.T)+bo)
    return OutN
# define network parameters
learningRate = 0.01
BatchSize= 150
Epoch=10
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
1,1,(NumHidden,NumInput)))
bh= np.random.uniform(-1,1,(1,NumHidden))
dWh= np.zeros((NumHidden,NumInput))
dbh= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-1,1,(NumOutput,NumHidden))
bo= np.random.uniform(-1,1,(1,NumOutput))
dWo= np.zeros((NumOutput,NumHidden))
dbo= np.zeros((1,NumOutput))
# Train the network with back propagation, SGD
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
Acc=np.zeros(Epoch)
MSE=np.zeros(Epoch)
```

```
IdxCost=0;
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSize)
))
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]
        #print(Batch_sample)
        x=np.matrix(x_train[Batch_sample,:])
        y=np.matrix(y_train [Batch_sample,:])
          # Feedforward propagation
        a=np.sign(np.dot(x,Wh.T)+bh)
        o =np.sign(np.dot(a,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=np.mean(np.mean(np.power((y-
o),2),axis=1))
        IdxCost+=1;
        #calculate loss function
        do=(y-o)
        dWo=np.matrix(np.dot(do.T,a)/BatchSize)
        dbo=np.mean(do,0)
        Wo=Wo + learningRate*dWo
        bo=bo + learningRate*dbo
        #back propagate error through sign function
        dh=np.dot(do,Wo)
        #dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        dWh=np.dot(dh.T,x)/BatchSize
        dbh=np.mean(dh,0)
        # Update weight
        Wh=Wh + learningRate*dWh
```

```
        bh=bh + learningRate*dbh
        #binary weights
        WBo = np.sign(Wo)
        bBo = np.sign(bo)
        WBh = np.sign(Wh)
        bBh = np.sign(bh)
    MSE[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
    print("MSE %f" % MSE[ep])
    #test the model
    RealOutN=feedforward(x_test,WBh,bBh,WBo,bBo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=dt.datetime.now() - t_start
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

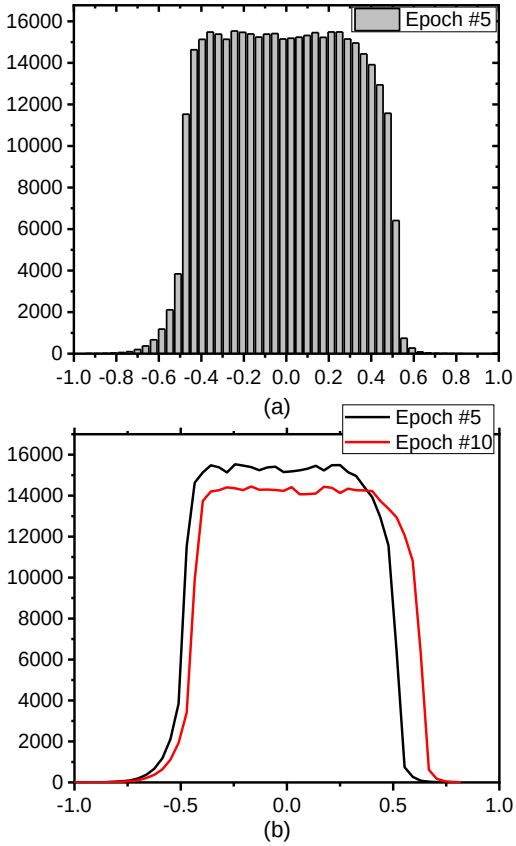
```
Load MNIST Dataset
----------------------------------
(60000, 784)
(60000, 10)
0:00:06.993644
Training epoch 0
MSE 0.339820
Accuracy: 0.409900
0:00:07.034230
Training epoch 1
MSE 0.219947
Accuracy: 0.447700
```

```
0:00:07.013310
Training epoch 2
MSE 0.195687
Accuracy: 0.480600
0:00:07.039093
Training epoch 3
MSE 0.182220
Accuracy: 0.479200
0:00:07.029792
Training epoch 4
MSE 0.173253
Accuracy: 0.487400
0:00:07.055333
Training epoch 5
MSE 0.166607
Accuracy: 0.482900
0:00:07.046096
Training epoch 6
MSE 0.152007
Accuracy: 0.503200
0:00:07.080202
Training epoch 7
MSE 0.147673
Accuracy: 0.516500
0:00:07.067966
Training epoch 8
MSE 0.138073
Accuracy: 0.531200
0:00:07.136679
Training epoch 9
MSE 0.111553
Accuracy: 0.562400
Total time :  0:01:13.746660
```



Trong ví dụ trên, chúng ta huấn luyện mạng với tập trọng số là số thực và cập nhật trọng số với phương pháp Mini-batch Gradient Descent. Tập trọng số thu được cho mạng nơ-ron nhị phân được nhị phân hoá từ tập trọng số thực sử dụng hàm dấu. Độ chính xác của mạng nơ-ron nhị phân thấp hơn so với mạng với tập trọng số sử dụng số thực 32 hoặc 64 bit. Sai số của mạng nơ-ron nhị phân do quá trình nhị phân hoá tập trọng số và việc sử dụng các hàm dấu thay cho các hàm kích hoạt phi tuyến. Quá trình hội tụ của mạng cũng lâu hơn.

Phương pháp cải thiện độ chính xác có thể thực hiện bằng cách sử dụng các ngưỡng động trong quá trình lượng tử đồng thời thực hiện lượng tử hóa trong quá trình huấn luyện. Phương pháp này khá đơn giản, tuy nhiên có thể đạt độ chính xác cao hơn đối với tập dữ liệu ký tự viết tay MNIST. Ý tưởng của phương pháp này là chúng ta huấn luyện mạng nhị phân với phương pháp Gradient Descent truyền thống. Sau mỗi chu kỳ huấn luyện, tập trọng số được cập nhật như cách chúng ta làm với các mạng nơ-ron có tập trọng số thực. Sau đó chúng ta có thể lượng tử hóa các tập trọng số về giá trị nhị phân. Việc xác định ngưỡng trong quá trình lượng tử hóa được dựa vào phân bố của tập trọng số. Sau mỗi lần cập nhật trọng số, chúng ta xác định phân bố của tập trọng số và từ đó lựa chọn giá trị ngưỡng thích hợp cho quá trình huấn luyện.



Hình 5.2: Phân bố của tập trọng số sau lần huấn luyện thứ 5 và thứ 10.

Trong phương pháp nhị phân hoá sử dụng hàm dấu, chúng ta luôn sử dụng giá trị ngưỡng là 0 để phân lớp ngõ ra -1 và +1. Tuy nhiên, việc phân lớp như vậy có khả năng dẫn đến việc phân chia không đều vì phân bố của tập trọng số có thể không đối xứng tại vị trí 0. Phân bố của tập trọng số trong lần huấn luyện thứ 5 và thứ 10 được minh hoạ trong Hình 5.2. Phương pháp cải tiến đơn giản là xác định điểm giữa (mean) của phân bố này để phân lớp -1 và +1 thay cho việc sử dụng hàm dấu. Chương trình được viết lại như sau:

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
#load dataset
print("Load MNIST Dataset")
```

```
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
# reshape the input data
x_train=np.reshape(x_train,(60000,784))
x_test= np.reshape(x_test,(10000,784))
#Convert to binary
x_train[x_train<=128] = 0
x_train[x_train>128] = 1
x_test[x_test<=128]= 0
x_test[x_test>128]= 1
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
y_train[y_train==0]= -1
y_test[y_test==0]= -1
print("----------------------------------")
print(x_train.shape)
print(y_train.shape)
import datetime as dt
def sign(w):
    x=w.copy()
    th=x.mean()
    x[x<th]=-1
    x[x>=th]=1
    return x
def AccTest(outN,labels):    # calculate the matching sc
ore
    OutMaxArg=np.argmax(outN,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy
def feedforward(samples,Wh,bh,Wo,bo):
    OutH1=np.sign(np.dot(samples,Wh.T)+bh)
    OutN=np.sign(np.dot(OutH1,Wo.T)+bo)
    return OutN
# define network parameters
```

```
learningRate = 0.01
BatchSize= 150
Epoch=10
NumOfTrainSample=60000
NumOfTestSample=10000
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=np.matrix(np.random.uniform(-
1,1,(NumHidden,NumInput)))
bh= np.random.uniform(-1,1,(1,NumHidden))
dWh= np.zeros((NumHidden,NumInput))
dbh= np.zeros((1,NumHidden))
#Output layer
Wo=np.random.uniform(-1,1,(NumOutput,NumHidden))
bo= np.random.uniform(-1,1,(1,NumOutput))
dWo= np.zeros((NumOutput,NumHidden))
dbo= np.zeros((1,NumOutput))
# Train the network with back propagation, SGD
SampleIdx=np.arange(NumOfTrainSample)
t_start=t1=dt.datetime.now()
Acc=np.zeros(Epoch)
MSE=np.zeros(Epoch)
IdxCost=0;
Cost=np.zeros(np.int(np.ceil(NumOfTrainSample/BatchSize)
))
for ep in range(Epoch):
    t1=dt.datetime.now()
    # Shuffle the trainning samples
    np.random.shuffle(SampleIdx)
    for i in range(0,NumOfTrainSample-
BatchSize,BatchSize):
        # Mini-batch Gradient descent algorithm
        Batch_sample=SampleIdx[i:i+BatchSize]
```

```
        #print(Batch_sample)
        x=np.matrix(x_train[Batch_sample,:])
        y=np.matrix(y_train [Batch_sample,:])
          # Feedforward propagation
        a=np.sign(np.dot(x,Wh.T)+bh)
        o =np.sign(np.dot(a,Wo.T)+bo)
        #calculate cross-entropy
        Cost[IdxCost]=np.mean(np.mean(np.power((y-
o),2),axis=1))
        IdxCost+=1;
        #calculate loss function
        do=(y-o)
        dWo=np.matrix(np.dot(do.T,a)/BatchSize)
        dbo=np.mean(do,0)
        Wo=Wo + learningRate*dWo
        bo=bo + learningRate*dbo
        #back propagate error through sign function
        dh=np.dot(do,Wo)
        #dh=np.multiply(np.dot(do,Wo),np.multiply(a,(1-
a)))
        dWh=np.dot(dh.T,x)/BatchSize
        dbh=np.mean(dh,0)
        # Update weight
        Wh=Wh + learningRate*dWh
        bh=bh + learningRate*dbh
        #binary weights
        WBo = sign(Wo)
        bBo = sign(bo)
        WBh = sign(Wh)
        bBh = sign(bh)
    MSE[ep]=np.mean(Cost)
    IdxCost=0
    t2=dt.datetime.now()
    print(t2-t1)
    print("Training epoch %i" % ep)
```

```
    print("MSE %f" % MSE[ep])
    #test the model
    RealOutN=feedforward(x_test,WBh,bBh,WBo,bBo)
    Accuracy=AccTest(RealOutN,y_test)
    Acc[ep]=Accuracy
    print("Accuracy: %f" % Accuracy)
t_end=dt.datetime.now() - t_start
print("Total time : ", t_end)
plt.plot(Acc,"dr-")
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.show()
```

Chương 6
PHÁT TRIỂN ỨNG DỤNG MẠNG HỌC SÂU VỚI PYTORCH

6.1 GIỚI THIỆU

Pytorch là thư viện mã nguồn mở được viết cho ngôn ngữ lập trình Python cho các ứng dụng về học sâu và xử lý ngôn ngữ tự nhiên. Pytorch là một framework hỗ trợ việc xây dựng và huấn luyện các mô hình học sâu hiệu quả như các framework khác phổ biến như Tensorflow. Tuy nhiên, ưu điểm của Pytorch là dễ sử dụng và độ linh động cao nên được sử dụng phổ biến trong nghiên cứu lẫn trong ứng dụng. Trong các chương trước, chúng ta tự xây dựng các mô hình và huấn luyện bằng ngôn ngữ Python sử dụng các giải thuật dựa trên phương pháp Gradient Descent. Việc nghiên cứu các lý thuyết và các chương trình như vậy giúp chúng ta hiểu hơn về nguyên lý tối ưu được sử dụng trong các mạng học sâu. Tuy nhiên, khi số lượng các lớp mạng tăng lên thì quá trình tính toán sẽ trở nên phức tạp. Mặt khác, việc thực hiện nhiều phép toán ma trận sẽ mất nhiều thời gian. Xu hướng phát triển các ứng dụng học sâu sử dụng các framework đang trở nên phổ biến. Tensorflow là một framework cho phép xây dựng và huấn luyện các mô hình học sâu được viết bởi Google. Một framework khá phổ biến khác và cũng được sử dụng nhiều nhờ cách sử dụng đơn giản hơn là Keras. Keras cho phép xây dựng các lớp mạng học sâu thông qua mô hình các lớp nối tiếp. Tuy nhiên lớp back-end của Keras có thể là Tensorflow hoặc Caffe. Người dùng Keras gọi các lớp thông qua các tham số mà không quan tâm đến quá trình tối ưu diễn ra như thế nào. Khác với Tensorflow, Pytorch được thiết kế cho phép người lập trình can thiệp vào trong quá trình tối ưu dễ dàng hơn. Mặt khác, Pytoch hỗ trợ các thư viện cho phép thực thi tính toán trên các thiết bị xử lý đồ hoạ (GPU) cho phép chương trình thực thi nhanh hơn. Pytorch thực hiện các phép toán trên các đối tượng được định nghĩa là các tensor. Tensor về bản chất là các đối tượng có thể lưu trữ các dữ liệu nhiều chiều, trong đó phổ biến là các vector, ma trận 2 chiều và ma trận nhiều chiều. Pytorch được thiết kế hướng đến thực hiện các phép toán ma trận trên GPU và cung cấp tính linh hoạt và tốc độ cao nhất cho nghiên cứu và phát triển các ứng dụng học sâu.

Chúng ta dễ dàng sử dụng thư viện numpy tạo ra các vector và ma trận. Trong phần này chúng ta lướt qua một số thao tác liên quan đến ma

trận và vector trên đối tượng tensor của Pytorch để làm cơ sở cho quá trình phát triển các mạng học sâu sử dụng Pytorch.

Tạo tensor trong Pytorch: ví dụ tạo ma trận 2 chiều có kích thước 5×3:

```
import torch
x = torch.empty(5, 3)
print(x)
```

```
tensor([[3.4756e-36, 0.0000e+00, 3.3631e-44],
        [0.0000e+00,        nan, 0.0000e+00],
        [1.1578e+27, 1.1362e+30, 7.1547e+22],
        [4.5828e+30, 1.2121e+04, 7.1846e+22],
        [9.2198e-39, 7.0374e+22, 0.0000e+00]])
```

Khởi tạo ma trận ngẫu nhiên với kích thước 5×3:

```
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.7576, 0.2793, 0.4031],
        [0.7347, 0.0293, 0.7999],
        [0.3971, 0.7544, 0.5695],
        [0.4388, 0.6387, 0.5247],
        [0.6826, 0.3051, 0.4635]])
```

Tạo ma trận với các phần tử kiểu *long* và khởi tạo giá trị 0 cho các phần tử:

```
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

Tạo ma trận với các phần tử cụ thể:

```
x = torch.tensor([5.5, 3])
print(x)
```

```
tensor([5.5000, 3.0000])
```

Tạo một tensor dựa trên một tensor đã tồn tại. Phương pháp này cho phép sử dụng lại thuộc tính của tensor như kiểu dữ liệu:

```
x = x.new_ones(5, 3, dtype=torch.double)
print(x)
x = torch.randn_like(x, dtype=torch.float
print(x)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[-0.2120,  1.0262, -0.5859],
        [ 0.3477, -0.0445, -0.0209],
        [ 0.9384,  0.4931,  1.1257],
        [ 0.6033,  0.1630,  1.1003],
        [ 0.8413, -0.4034, -0.2213]])
```

Có nhiều cú pháp thực hiện các toán tử, trong các ví dụ dưới đây chúng ta sẽ khảo sát một số cú pháp mà Pytorch thực hiện phép cộng:

```
y = torch.rand(5, 3)
print(x + y)
```

```
tensor([[ 0.2609,  1.1194,  0.1897],
        [ 1.1515,  0.2076,  0.1601],
        [ 1.0074,  1.2055,  1.3068],
        [ 1.2575,  0.5953,  1.7687],
        [ 0.8808,  0.0058, -0.0300]])
```

Thực hiện phép cộng thông qua phương thức của torch.

```
print(torch.add(x, y))
```

```
tensor([[ 0.2609,  1.1194,  0.1897],
        [ 1.1515,  0.2076,  0.1601],
        [ 1.0074,  1.2055,  1.3068],
        [ 1.2575,  0.5953,  1.7687],
        [ 0.8808,  0.0058, -0.0300]])
```

Thực hiện phép cộng thông qua phương thức của đối tượng y.

```
# adds x to y
```

```
y.add_(x)
print(y)
```

```
tensor([[ 0.2609,  1.1194,  0.1897],
        [ 1.1515,  0.2076,  0.1601],
        [ 1.0074,  1.2055,  1.3068],
        [ 1.2575,  0.5953,  1.7687],
        [ 0.8808,  0.0058, -0.0300]])
```

6.2 AUTOGRAD – AUTOMATIC DIFFIRENTIATION

Trong các chương trước chúng ta đã khảo sát phương pháp tối ưu hàm mất mát sử dụng phương pháp giảm dần theo độ dốc thông qua việc tính toán các Gradient của hàm mất mát. Phương pháp tối ưu hàm mất mát trong các mạng học sâu hiện đại vẫn dựa trên lý thuyết tối ưu sử dụng phương pháp giảm độ dốc (Gradient descent) và tính toán các Gradient. Pytorch hỗ trợ các hàm *autograd* cho phép tính vi phân tự động cho các phép toán được thực hiện trên các tensor. Autograd cho phép quá trình lan truyền ngược được xây dựng đơn giản hơn. Nếu chúng ta thiết lập thuộc tính *.requires_grad* với thông số True, Pytoch bắt đầu theo dõi tất cả các phép toán trên tensor. Khi kết thúc quá trình tính toán, chúng ta có thể gọi phương thức *.backward()* và thu được tất cả các giá trị Gradient được tính một cách tự động. Gradient của tensor này được tính và lưu trong thuộc tính *.grad*.

Giả sử x là ma trận có kích thước 2×2 với các giá trị đều bằng 1, có thể biểu diễn x dưới dạng ma trận như sau:

$$x = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$
$$y = x + 2 \tag{6.1}$$

Trong phép toán trên, các phần tử của y thu được bằng cách thực hiện phép cộng các phần tử tương ứng của x với giá trị 2.

Tiếp tục thực hiện phép tính trên y để thu được z và ngõ ra out là giá trị trung bình của x, chúng ta có:

$$z = y * y * 3$$
$$out = mean(z) \tag{6.2}$$

Dựa vào các phép toán trên, chúng ta tính được ngõ ra z là ma trận như sau:

$$z = \begin{bmatrix} 27 & 27 \\ 27 & 27 \end{bmatrix}$$
$$Out = (z_{11} + z_{12} + z_{21} + z_{22})/4 \tag{6.3}$$

Sử dụng phương pháp tính Gradient, chúng ta tính được Gradient của hàm ngõ ra Out như sau

$$\frac{\partial Out}{\partial x} = \begin{bmatrix} \frac{\partial Out}{\partial x_{11}} & \frac{\partial Out}{\partial x_{12}} \\ \frac{\partial Out}{\partial x_{21}} & \frac{\partial Out}{\partial x_{22}} \end{bmatrix}$$
$$\frac{\partial Out}{\partial x_{11}} = \frac{\partial Out}{\partial z_{11}} \frac{\partial z_{11}}{\partial y_{11}} \frac{\partial y_{11}}{\partial x_{11}}$$
$$\frac{\partial Out}{\partial x_{11}} = \frac{1}{4} 6y.1 = 4.5 \tag{6.4}$$
$$\frac{\partial Out}{\partial x_{12}} = \frac{1}{4} 6y.1 = 4.5$$
$$\frac{\partial Out}{\partial x_{21}} = \frac{1}{4} 6y.1 = 4.5$$
$$\frac{\partial Out}{\partial x_{22}} = \frac{1}{4} 6y.1 = 4.5$$

Một trong những đặc điểm của Pytorch trong thiết kế các ứng dụng mạng học sâu là tính linh hoạt để cho phép người lập trình tối ưu hàm sai số sử dụng phương pháp Gradient Descent. Đây là một khác biệt của Pytorch so với các thư viện thiết kế học sâu khác như Tensorflow, Keras, trong đó quá trình tối ưu được thực hiện bởi các hàm mà chúng ta chỉ cần đưa vào các tham số. Pytorch gần mô hình toán hơn nên nó phù hợp cho cả giới nghiên cứu và ứng dụng. Pytorch hỗ trợ phương thức tính Gradient tự động cho các đối tượng tensor.

Ví dụ sau sẽ khởi tạo biến x, sau đó thực hiện các phép toán như mô tả ở trên với Pytorch. Các Gradient được tính cho biến x sẽ được lưu lại. Sau quá trình lan truyền thẳng, chúng ta có thể gọi lại phương thức *.backward()* để tính Gradient phục vụ cho các mục đích tối ưu hàm mục tiêu.

```
import torch
x = torch.ones(2, 2, requires_grad=True)
y = x + 2
```

```
z = y * y * 3
out = z.mean()
print(out)
out.backward()
print(x.grad)
```

```
tensor(27., grad_fn=<MeanBackward0>)
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

Bằng cách thiết lập thuộc tính *.requires_grad = True* và sử dụng phương thức *.backward()*, Pytorch tự động tính Gradient của hàm ngõ ra đối với các phần tử trong tensor x. Các kết quả thu được giống với việc tính toán trong (6.4) trước đó.

Bây giờ, chúng ta sẽ sử dụng Pytorch để thiết kế mạng 2 lớp đơn giản cho bài toán nhận dạng ký tự MNIST.Trong ví dụ này, chúng ta thiết kế mạng với 784 ngõ vào tương ứng cho các ảnh 28x28, 512 nơ-ron ở lớp ẩn và 10 nơ-ron ở lớp ngõ ra cho bài toán nhận dạng 10 ký tự viết tay. Chúng ta sẽ sử dụng giải thuật Gradient Descent, trong đó, ứng dụng autograd của Pytorch để cập nhật trọng số. Trong ví dụ này, chúng ta cập nhật trọng số cho từng mẫu dữ liệu. Đương nhiên như đã phân tích trong phần trước, việc cập nhật này sẽ làm chương trình huấn luyện chậm nhiều so với phương pháp sử dụng nhóm mẫu huấn luyện nhỏ (mini-batch). Ví dụ này giúp chúng ta hiểu cách tối ưu một mạng nơ-ron sử dụng autograd hỗ trợ từ Pytorch.

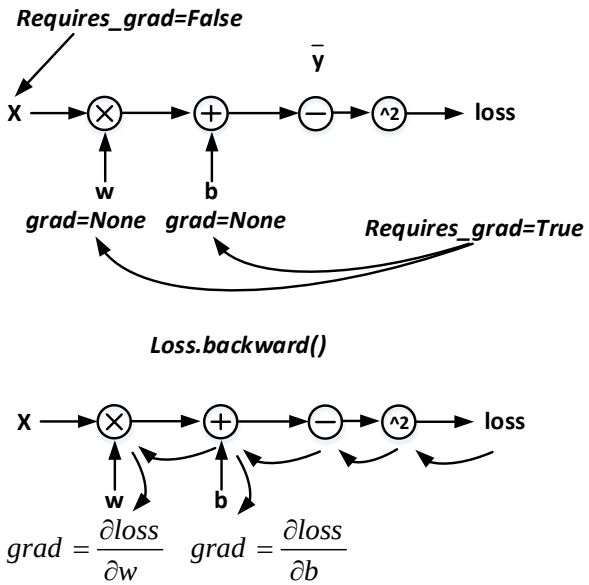
```
import datetime as dt
import torch
import numpy as np
import tensorflow as tf
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
learningRate = 0.1
```

```
BatchSize= 200
Epoch=5
Momentum=0.95
NumOfTrainSample=60000
NumOfTestSample=10000
NumBatch = np.int(NumOfTrainSample/BatchSize)
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=torch.normal(0,0.25,size=(NumHidden,NumInput),requires_g
rad=True)
bh=
torch.normal(0,0.25,size=(1,NumHidden),requires_grad=True)
#del_bh= torch.rand((1,NumHidden),requires_grad=True)
#Output layer
Wo=torch.normal(0,0.25,size=(NumOutput,NumHidden),requires_
grad=True)
bo=
torch.normal(0,0.5,size=(1,NumOutput),requires_grad=True)
def sigmoid(x):
    return 1/(1+torch.exp(-x))
def Forwardpass(samples,Wh,bh,Wo,bo):
    with torch.no_grad():
        zh = torch.matmul(samples,torch.transpose(Wh,0,1))
        ah=sigmoid(zh)
        zo = torch.matmul(ah,torch.transpose(Wo,0,1))
        ao=sigmoid(zo)
    return ao.numpy()
def AccTest(labels,out):    # calculate the matching score
    labels=np.matrix(labels)
    out=np.matrix(out)
    OutMaxArg=np.argmax(out,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy*100
def Cost (y,a):
```

```
    return np.mean(np.power((y-a),2))
x_t=torch.tensor(x_test,dtype=torch.float)
pred=Forwardpass(x_t,Wh,bh,Wo,bo)
Acc=AccTest(y_test,pred)
print(Acc)
for loop in range(5):
    print('---------')
    for ep in range (NumOfTrainSample):
        x=torch.tensor(x_train[ep ,:],dtype=torch.float)
        y=torch.tensor(y_train[ep ,:],dtype=torch.float)
        zh = torch.matmul(x,torch.transpose(Wh,0,1))
        ah=sigmoid(zh)
        zo = torch.matmul(ah,torch.transpose(Wo,0,1))
        ao=sigmoid(zo)
        E = torch.mean((y-ao)**2)
        E.backward()
        with  torch.no_grad():  #  this  will  switch  off
Gradients
            Wh-=learningRate*Wh.grad
            Wo-=learningRate*Wo.grad
            #setting Gradients to be zero
            Wh.grad.zero_()
            Wo.grad.zero_()
        #MSE =MSE+E
        if ep%1000==0:
            print(E)
print('Accuracy: ')
x_t=torch.tensor(x_test,dtype=torch.float)
pred=Forwardpass(x_t,Wh,bh,Wo,bo)
Acc=AccTest(y_test,pred)
print(Acc)
```

Trong chương trình trên chúng ta sử dụng phương thức tính Gradient tự động trên các đối tượng tensor để cập nhật trọng số cho từng mẫu. Quá trình tính toán Gradient tương tự như quá trình chúng ta thực hiện thủ công như đã giới thiệu trong phần trước.

Chúng ta có thể hình dung quá trình tính toán Gradient tự động của Pytorch như sau:



Hình 6.1: Mô tả quá trình tính Gradient của hàm mất mát sử dụng chức năng AutoGrad của Pytorch.

Một nhược điểm của quá trình cập nhật dựa trên từng mẫu huấn luyện riêng lẻ là nó mất khá nhiều thời gian cho quá trình huấn luyện. Phương pháp sử dụng nhóm nhỏ được giới thiệu trong chương trước cho phép quá trình huấn luyện nhanh hơn. Chương trình huấn luyện mạng nơ-ron sử dụng nhóm nhỏ được mô tả bên dưới.

```
import datetime as dt
import torch
import numpy as np
import tensorflow as tf
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
```

```
learningRate = 0.02
BatchSize= 200
Epoch=10
Momentum=0.95
NumOfTrainSample=60000
NumOfTestSample=10000
NumBatch = np.int(NumOfTrainSample/BatchSize)
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=torch.normal(0,0.25,size=(NumHidden,NumInput),require
s_grad=True)
bh=
torch.normal(0,0.25,size=(1,NumHidden),requires_grad=Tru
e)
#del_bh= torch.rand((1,NumHidden),requires_grad=True)
#Output layer
Wo=torch.normal(0,0.25,size=(NumOutput,NumHidden),requir
es_grad=True)
bo=
torch.normal(0,0.5,size=(1,NumOutput),requires_grad=True
)
def sigmoid(x):
    return 1/(1+torch.exp(-x))
def Forwardpass(samples,Wh,bh,Wo,bo):
    with torch.no_grad():
        zh=
torch.matmul(samples,torch.transpose(Wh,0,1))
        ah=sigmoid(zh)
        zo = torch.matmul(ah,torch.transpose(Wo,0,1))
        ao=sigmoid(zo)
     return ao.numpy()
def AccTest(labels,out):      # calculate the matching
score
```

```
    labels=np.matrix(labels)
    out=np.matrix(out)
    OutMaxArg=np.argmax(out,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy*100
def Cost (y,a):
    return np.mean(np.power((y-a),2))
x_t=torch.tensor(x_test,dtype=torch.float)
pred=Forwardpass(x_t,Wh,bh,Wo,bo)
Acc=AccTest(y_test,pred)
print(Acc)
for ep in range (Epoch):
    print('epoch:  ',ep,'------------------')
    #shuffle the sample
    SampleIdx=np.array(np.arange(0,NumOfTrainSample))
    np.random.shuffle(SampleIdx)
    samples = torch.empty(BatchSize,1)
    #mini-batch
    MSE=0;
    for batch in range(NumBatch):
        #print(batch)
samples=SampleIdx[batch*BatchSize:(batch+1)*BatchSize]
        #print(samples)
        x_batch=torch.tensor(x_train[samples
,:],dtype=torch.float)
        y_batch=torch.tensor(y_train[samples
,:],dtype=torch.float)
        zh=
torch.matmul(x_batch,torch.transpose(Wh,0,1))
        ah=sigmoid(zh)
        zo = torch.matmul(ah,torch.transpose(Wo,0,1))
        ao=sigmoid(zo)
        E = torch.sum(torch.mean(((y_batch-ao)**2),1))
        #print(E)
```

```
        E.backward()
        with torch.no_grad(): # this will switch off
Gradients
            Wh-=learningRate*Wh.grad
            Wo-=learningRate*Wo.grad
        #    #setting Gradients to be zero
            Wh.grad.zero_()
            Wo.grad.zero_()
        #MSE =MSE+E
        #if ep%1000==0:
         #   print(E)
print('Accuracy: ')
x_t=torch.tensor(x_test,dtype=torch.float)
pred=Forwardpass(x_t,Wh,bh,Wo,bo)
Acc=AccTest(y_test,pred)
print(Acc)
```

Trong các ví dụ trên, chúng ta sử dụng lập trình tối ưu mạng nơ-ron sử dụng phương pháp Gradient. Chỉ có phương thức tính Gradient tự động được sử dụng. Tuy nhiên, Pytorch hỗ trợ khác nhiều module, hàm cho phép thực hiện quá trình tính toán cập nhật trọng số đơn giản hơn. Trong các ví dụ tiếp theo, chúng ta sẽ sử dụng một số module hỗ trợ từ Pytorch để thiết kế và huấn luyện mạng nơ-ron học sâu

6.3 SỬ DỤNG LỚP OPTIMIZATION CỦA PYTORCH.

Pytorch cung cấp lớp optimization (optim) cho phép thực hiện quá trình cập nhật trọng số dựa trên các kết quả tính toán Gradient tự động. Quá trình cập nhật trọng số dựa trên mô hình đi ngược với chiều tăng của hàm số bằng cách xác định Gradient như đã phân tích kỹ ở chương trước. Các giải thuật cập nhật khác nhau sử dụng tốc độ học và momentum linh động khác nhau. Pytorch hỗ trợ các thuật toán cập nhật trọng số như sau:

```
'ASGD',
'Adadelta',
'Adagrad',
'Adam',
'Adamax',
```

```
'LBFGS',
'Optimizer',
'RMSprop',
'Rprop',
'SGD',
'SparseAdam',
```

Sử dụng lớp optim để cập nhật trọng số.

```
import datetime as dt
import torch
import numpy as np
import tensorflow as tf
#load dataset
print("Load MNIST Dataset")
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test)= mnist.load_data()
x_train=np.reshape(x_train,(60000,784))/255.0
x_test= np.reshape(x_test,(10000,784))/255.0
y_train = np.matrix(np.eye(10)[y_train])
y_test = np.matrix(np.eye(10)[y_test])
learningRate = 0.02
BatchSize= 200
Epoch=10
Momentum=0.95
NumOfTrainSample=60000
NumOfTestSample=10000
NumBatch = np.int(NumOfTrainSample/BatchSize)
NumInput=784
NumHidden=512
NumOutput=10; # number of classes
#Hidden layer
Wh=torch.normal(0,0.25,size=(NumHidden,NumInput),requir
es_grad=True)
bh=
torch.normal(0,0.25,size=(1,NumHidden),requires_grad=Tr
```

```
ue)
#Output layer
Wo=torch.normal(0,0.25,size=(NumOutput,NumHidden),requi
res_grad=True)
bo=
torch.normal(0,0.5,size=(1,NumOutput),requires_grad=Tru
e)
def sigmoid(x):
    return 1/(1+torch.exp(-x))
def Forwardpass(samples,Wh,bh,Wo,bo):
    with torch.no_grad():
        zh                                              =
torch.matmul(samples,torch.transpose(Wh,0,1))
        ah=sigmoid(zh)
        zo = torch.matmul(ah,torch.transpose(Wo,0,1))
        ao=sigmoid(zo)
    return ao.numpy()
def AccTest(labels,out):      # calculate the matching
score
    labels=np.matrix(labels)
    out=np.matrix(out)
    OutMaxArg=np.argmax(out,axis=1)
    LabelMaxArg=np.argmax(labels,axis=1)
    Accuracy=np.mean(OutMaxArg==LabelMaxArg)
    return Accuracy*100
def Cost (y,a):
    return np.mean(np.power((y-a),2))
x_t=torch.tensor(x_test,dtype=torch.float)
pred=Forwardpass(x_t,Wh,bh,Wo,bo)
Acc=AccTest(y_test,pred)
print(Acc)
import torch.optim as optim
optimizer = optim.SGD([Wh, Wo],lr=1e-2)
for ep in range (Epoch):
    print('epoch:  ',ep,'------------------')
    #shuffle the samples
```

```
    SampleIdx=np.array(np.arange(0,NumOfTrainSample))
    np.random.shuffle(SampleIdx)
    samples = torch.empty(BatchSize,1)
    #mini-batch
    MSE=0;
    for batch in range(NumBatch):

samples=SampleIdx[batch*BatchSize:(batch+1)*BatchSize]
        x_batch=torch.tensor(x_train[samples
,:],dtype=torch.float)
        y_batch=torch.tensor(y_train[samples
,:],dtype=torch.float)
        zh=
torch.matmul(x_batch,torch.transpose(Wh,0,1))
        ah=sigmoid(zh)
        zo = torch.matmul(ah,torch.transpose(Wo,0,1))
        ao=sigmoid(zo)
        E = torch.sum(torch.mean(((y_batch-ao)**2),1))
        #print(E)
        optimizer.zero_grad()
        E.backward()
        optimizer.step()
print('Accuracy: ')
x_t=torch.tensor(x_test,dtype=torch.float)
pred=Forwardpass(x_t,Wh,bh,Wo,bo)
Acc=AccTest(y_test,pred)
print(Acc)
```

6.4 PYTORCH NN MODULE

Pytorch hỗ trợ các module và các lớp Torch.nn, torch.optim, Dataset, DataLoader cho thiết kế và huấn luyện mạng nơ-ron. Module torch.nn bao gồm các khối cho phép thiết kế các lớp kiến trúc mạng một cách đơn giản tương tự như Keras. Các khối được xây dựng sẵn này được gọi là các module. Các module bao gồm các tham số cài đặt như là các thuộc tính là các kiểu tensor và chúng được tối ưu trong quá trình huấn luyện. Trong ví dụ này, chúng ta sẽ xem xét việc sử dụng các thư viện hỗ trợ để thiết kế một mạng nơ-ron dùng trong ứng dụng nhận dạng ký tự

viết tay. Trong các ví dụ trước chúng ta sử dụng tập ký tự viết tay được lưu trong các tập .csv hoặc tải trực tiếp thông qua thư viện tensorflow. Trong ví dụ này chúng ta sử dụng tập MNIST đã được phân chia thành nhóm các mẫu cho huấn luyện, nhóm các mẫu cho đánh giá và kiểm thử mô hình và được nén trong một tập tin dạng "*mnist.pkl.gz*". Trong ví dụ này, chúng ta tải tập tin *mnist.pkt.gz* là tập tin chứa các mẫu ký tự viết tay được nén sử dụng thư viện Pickle trực tiếp từ trang web http://deeplearning.net. Trong trường hợp chúng ta không mở được trang web deeplearning.net, chúng ta cũng có thể tải tập tin *mnist.pkl.gz* từ các nguồn khác và lưu về thư mục máy tính.

Tập tin *mnist.pkl.gz* chứa tổng cộng 70,000 mẫu ký tự viết tay được chia làm 3 nhóm, bao gồm 50,000 mẫu cho quá trình huấn luyện (training), 10,000 mẫu cho quá trình kiểm thử mô hình trong quá trình huấn luyện (validation) và 10,000 mẫu cho quá trình kiểm thử mô hình sau huấn luyện (testing).

Chương trình tải tập tin *mnist.pkl.gz* từ trang web http://deeplearning.net như sau:

```
from pathlib import Path
import requests
import pickle
import gzip
DATA_PATH = Path("data")
PATH = DATA_PATH / "mnist"
PATH.mkdir(parents=True, exist_ok=True)
URL = "http://deeplearning.net/data/mnist/"
FILENAME = "mnist.pkl.gz"
if not (PATH / FILENAME).exists():
        content = requests.get(URL + FILENAME).content
        (PATH / FILENAME).open("wb").write(content)
with gzip.open((PATH / FILENAME).as_posix(), "rb") as
f:
        ((x_train,   y_train),   (x_valid,   y_valid),
(x_test, y_test)) = pickle.load(f, encoding="latin-1")
```

Trường hợp không tải trực tiếp, chúng ta tải từ các nguồn khác và lưu vào thư mục tương ứng trên ./data/mnist/

```
from pathlib import Path
import pickle
```

```
import gzip
DATA_PATH = Path("data")
PATH = DATA_PATH / "mnist"
PATH.mkdir(parents=True, exist_ok=True)
FILENAME = "mnist.pkl.gz"
with gzip.open((PATH / FILENAME).as_posix(), "rb") as
f:
        ((x_train,   y_train),   (x_valid,   y_valid),
(x_test, y_test)) = pickle.load(f, encoding="latin-1")
```

Sử dụng thư viện pickle để giải nén và thu được 3 nhóm dữ liệu phục vụ cho quá trình huấn luyện và kiểm thử mô hình. Sử dụng matplotlib để kiểm tra đảm bảo dữ liệu đã được tải.

```
from matplotlib import pyplot
import numpy as np
print(x_test.shape)
pyplot.imshow(x_train[0].reshape((28,28)), cmap="gray")
pyplot.show()
print(x_train.shape)
```

Chuyển các dữ liệu tải được sang các tensor.

```
import torch
x_train, y_train, x_valid, y_valid, x_test, y_test =
map(torch.tensor,(x_train,y_train,x_valid,y_valid,x_te
st,y_test))
n, c = x_train.shape
x_train, x_train.shape, y_train.min(), y_train.max()
print(x_train, y_train)
print(x_train.shape)
print(y_train.min(), y_train.max())
```

Pytorch hỗ trợ thiết kế mạng nơ-ron với nhiều mức độ khác nhau. Chúng ta có thể sử dụng chức năng tự động tính Gradient để huấn luyện mạng. Việc thiết kế có thể từ việc thủ công cho đến việc sử dụng các hàm thư viện torch để đơn giản hoá việc thiết kế. Sau khi thiết kế, mạng được huấn luyện với các nhóm mẫu (mini-batch). Hoặc đơn giản hơn, việc thực hiện huấn luyện mạng có thể được thực hiện thông qua các lớp, các module hỗ trợ, trong đó, các nhóm dữ liệu huấn luyện sẽ được chia bởi các lớp hỗ trợ.

Thiết kế và huấn luyện có thể được thực hiện như sau:

```
from pathlib import Path
import pickle
import gzip
DATA_PATH = Path("data")
PATH = DATA_PATH / "mnist"
PATH.mkdir(parents=True, exist_ok=True)
FILENAME = "mnist.pkl.gz"

with gzip.open((PATH / FILENAME).as_posix(), "rb") as f:
        ((x_train,    y_train),    (x_valid,    y_valid),
(x_test, y_test)) = pickle.load(f, encoding="latin-1")
from matplotlib import pyplot
import numpy as np
print(x_test.shape)
pyplot.imshow(x_train[0].reshape((28,28)),cmap="gray")
pyplot.show()
print(x_train.shape)
import torch
x_train,  y_train,  x_valid,  y_valid,  x_test,  y_test  =
map(torch.tensor,(x_train,y_train,x_valid,y_valid,
x_test, y_test))
n, c = x_train.shape
def log_softmax(x):
    return x - x.exp().sum(-1).log().unsqueeze(-1)
def log_sigmoid(x):
    return 1/(1+torch.exp(-x))
def model(xb):
    h=log_sigmoid(xb@Wh + bh).requires_grad_()
    return log_softmax(h @ Wo + bo)
def accuracy(out, yb):
    preds = torch.argmax(out, dim=1)
    return (preds == yb).float().mean()
def nll(input, target):
    return -input[range(target.shape[0]), target].mean()
loss_func = nll
import math
```

```
bs=64 # batch size
#define weights
Wh = torch.randn(784, 512) / math.sqrt(784)
Wh.requires_grad_()
bh = torch.zeros(512, requires_grad=True)
# output layer
Wo = torch.randn(512, 10) / math.sqrt(512)
Wo.requires_grad_()
bo = torch.zeros(10, requires_grad=True)
print('Accuracy: ',accuracy(model(x_test), y_test)*100)
from IPython.core.debugger import set_trace
lr = 0.5  # learning rate
epochs = 5
print('Training process')
for epoch in range(epochs):
    print('Epoch: ',epoch)
    for i in range((n - 1) // bs + 1):
        #         set_trace()
        start_i = i * bs
        end_i = start_i + bs
        xb = x_train[start_i:end_i]
        yb = y_train[start_i:end_i]
        pred = model(xb)
        loss = loss_func(pred, yb)
        loss.backward()
        with torch.no_grad():
            Wh -= Wh.grad * lr
            bh -= bh.grad * lr
            Wo -= Wo.grad * lr
            bo -= bo.grad * lr
            Wh.grad.zero_()
            bh.grad.zero_()
            Wo.grad.zero_()
            bo.grad.zero_()
print('Test      accuracy:      ',accuracy(model(x_test),
y_test)*100)
```

Một số phép toán được thực hiện thông qua các hàm hỗ trợ từ Pytorch. Trong ví dụ trước, chúng ta sử dụng phép nhân ma trận để thực hiện hàm tổ hợp tuyến tính cho ngõ ra mỗi nơ-ron dưới dạng $y = xw^T + b$. Hàm tổ hợp tuyến tính cho mỗi nơ-ron cũng có thể được thực hiện thông qua lớp hỗ trợ *nn.linear*. Mô hình mạng có thể được thiết kế lại bằng cách sử dụng lớp *nn.linear*. Mặt khác, thay vì duyệt hết các mẫu huấn luyện và lấy ra từng nhóm nhỏ để tính giá trị cập nhật, chúng ta sử dụng các lớp hỗ trợ từ Pytorch để lấy các nhóm nhỏ (mini-batch) cho việc tính toán giá trị Gradient tự động để cập nhật trọng số. Chương trình có thể được viết lại như sau:

```
from pathlib import Path
import pickle
import gzip
DATA_PATH = Path("data")
PATH = DATA_PATH / "mnist"
PATH.mkdir(parents=True, exist_ok=True)
FILENAME = "mnist.pkl.gz"
with gzip.open((PATH / FILENAME).as_posix(), "rb") as
f:
        ((x_train,   y_train),   (x_valid,   y_valid),
(x_test, y_test)) = pickle.load(f, encoding="latin-1")
from matplotlib import pyplot
import numpy as np
print(x_test.shape)
pyplot.imshow(x_train[0].reshape((28,28)),
cmap="gray")
pyplot.show()
print(x_train.shape)
import torch
x_train, y_train, x_valid, y_valid, x_test, y_test =
map(
    torch.tensor,(x_train,      y_train,      x_valid,
y_valid,x_test,y_test))
n, c = x_train.shape
lr = 0.5  # learning rate
epochs = 10
from torch import nn
```

```
from torch import optim
def log_sigmoid(x):
    return 1/(1+torch.exp(-x))
class Mnist_Logistic(nn.Module):
    def __init__(self):
        super().__init__()
        self.h = nn.Linear(784, 512)
        self.o = nn.Linear(512, 10)
def forward(self, xb):
        return self.o(log_sigmoid(self.h(xb)))
import torch.nn.functional as F
loss_func = F.cross_entropy
def get_model():
    model = Mnist_Logistic()
    return model, optim.SGD(model.parameters(), lr=lr)
def accuracy(out, yb):
    preds = torch.argmax(out, dim=1)
    return (preds == yb).float().mean()
model, opt = get_model()
print('Accuracy:              ',accuracy(model(x_test),
y_test)*100)
bs =64
from torch.utils.data import DataLoader
from torch.utils.data import TensorDataset
train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=bs)
for epoch in range(epochs):
    print('Epoch:  ',epoch)
    for xb, yb in train_dl:
        pred = model(xb)
        loss = loss_func(pred, yb)

        loss.backward()
        opt.step()
        opt.zero_grad()
```

```
print(loss_func(model(xb), yb))
print('Accuracy:                ',accuracy(model(x_test),
y_test)*100)
```

Mạng nơ-ron chập (CNN) có thể được thiết kế tương tự như trong ví dụ trên bằng cách thêm các lớp tích chập vào mô hình. Chúng ta đã nghiên cứu cấu trúc CNN ở chương trước. Trong ví dụ này chúng ta cũng bổ sung tập dữ liệu cho quá trình validation. Trong quá trình huấn luyện các mạng học sâu, tập dữ liệu thông thường được chia ra 3 nhóm: training set, validation set, và test set. Training set được sử dụng để tính toán Gradient và cập nhật trọng số. Test set được dùng để đánh giá mô hình sau khi huấn luyện. Validation set tương tự như test set, nó được dùng để đánh giá mô hình. Tuy nhiên, validation set được sử dụng trong quá trình huấn luyện. Sau mỗi quá trình cập nhật trọng số, tập validation set được sử dụng để đánh giá mô hình, nhằm mục đích kiểm tra tính quá khớp của mô hình (overfitting), từ đó có thể điều chỉnh các thông số như số lớp, số nơ-ron và các tham số mô hình.

```
from pathlib import Path
import pickle
import gzip
DATA_PATH = Path("data")
PATH = DATA_PATH / "mnist"
PATH.mkdir(parents=True, exist_ok=True)
FILENAME = "mnist.pkl.gz"
with gzip.open((PATH / FILENAME).as_posix(), "rb") as
f:((x_train, y_train), (x_valid, y_valid), (x_test,
y_test)) = pickle.load(f, encoding="latin-1")
from matplotlib import pyplot
import numpy as np
print(x_test.shape)
pyplot.imshow(x_train[0].reshape((28,            28)),
cmap="gray")
pyplot.show()
print(x_train.shape)
import torch
x_train, y_train, x_valid, y_valid, x_test, y_test =
map(torch.tensor,(x_train,y_train,x_valid,y_valid,x_tes
t,y_test))
```

```
n, c = x_train.shape
lr = 0.2  # learning rate
epochs = 10
from torch import nn
from torch import optim
import torch.nn.functional as F
def log_sigmoid(x):
    return 1/(1+torch.exp(-x))
class Mnist_Logistic(nn.Module):
    def __init__(self):
        super().__init__()
        self.h = nn.Linear(784, 512)
        self.o = nn.Linear(512, 10)
    def forward(self, xb):
        return self.o(log_sigmoid(self.h(xb)))
class Mnist_CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1  =  nn.Conv2d(1,  16,  kernel_size=3,
stride=2, padding=1)
        self.conv2  =  nn.Conv2d(16,  16,  kernel_size=3,
stride=2, padding=1)
        self.conv3  =  nn.Conv2d(16,  10,  kernel_size=3,
stride=2, padding=1)
    def forward(self, xb):
        xb = xb.view(-1, 1, 28, 28)
        xb = F.relu(self.conv1(xb))
        xb = F.relu(self.conv2(xb))
        xb = F.relu(self.conv3(xb))
        xb = F.avg_pool2d(xb, 4)
        return xb.view(-1, xb.size(1))
loss_func = F.cross_entropy
def get_model():
    model = Mnist_CNN()
    return model, optim.SGD(model.parameters(), lr=lr)
```

```
def accuracy(out, yb):
    preds = torch.argmax(out, dim=1)
    return (preds == yb).float().mean()
model, opt = get_model()
print('Accuracy: ',accuracy(model(x_test), y_test)*100)
bs =64
from torch.utils.data import DataLoader
from torch.utils.data import TensorDataset
train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=bs, shuffle
= True)
valid_ds = TensorDataset(x_valid, y_valid)
valid_dl = DataLoader(valid_ds, batch_size=bs * 2)
for epoch in range(epochs):
    print('Epoch:  ',epoch)
    model.train()
    for xb, yb in train_dl:
        pred = model(xb)
        loss = loss_func(pred, yb)
        loss.backward()
        opt.step()
        opt.zero_grad()
    model.eval()
    with torch.no_grad():
        valid_loss = sum(loss_func(model(xb), yb) for
xb, yb in valid_dl)
    print(epoch, valid_loss / len(valid_dl))
print('Accuracy: ',accuracy(model(x_test), y_test)*100)
```

6.5 SỬ DỤNG NN.SEQUENTIAL

Pytorch cung cấp lớp nn.sequential cho phép thiết kế mạng nơ-ron theo mô hình các lớp nối tiếp. Điều này tương tự như mô hình Sequential trong thư viện Keras.

```
from pathlib import Path
import pickle
import gzip
```

```
DATA_PATH = Path("data")
PATH = DATA_PATH / "mnist"
PATH.mkdir(parents=True, exist_ok=True)
FILENAME = "mnist.pkl.gz"
with gzip.open((PATH / FILENAME).as_posix(), "rb") as
f:
        ((x_train, y_train), (x_valid, y_valid),
(x_test, y_test)) = pickle.load(f, encoding="latin-1")
from matplotlib import pyplot
import numpy as np
print(x_test.shape)
pyplot.imshow(x_train[0].reshape((28,             28)),
cmap="gray")
pyplot.show()
print(x_train.shape)
import torch
x_train, y_train, x_valid, y_valid, x_test, y_test =
map(torch.tensor,(x_train,y_train,x_valid,y_valid,x_tes
t,y_test))
n, c = x_train.shape
lr = 0.2  # learning rate
epochs = 10
from torch import nn
from torch import optim
import torch.nn.functional as F
class Lambda(nn.Module):
    def __init__(self, func):
        super().__init__()
        self.func = func
    def forward(self, x):
        return self.func(x)
def preprocess(x):
    return x.view(-1, 1, 28, 28)
model = nn.Sequential(
    Lambda(preprocess),
    nn.Conv2d(1,16,kernel_size=3,stride=2, padding=1),
```

```
    nn.ReLU(),
    nn.Conv2d(16,16,kernel_size=3,stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16,10,kernel_size=3,stride=2, padding=1),
    nn.ReLU(),
    nn.AvgPool2d(4),
    Lambda(lambda x: x.view(x.size(0), -1)),
)
loss_func = F.cross_entropy
def accuracy(out, yb):
    preds = torch.argmax(out, dim=1)
    return (preds == yb).float().mean()
opt     =     optim.SGD(model.parameters(),     lr=lr,
momentum=0.9)
print('Accuracy: ',accuracy(model(x_test), y_test)*100)
bs =64
from torch.utils.data import DataLoader
from torch.utils.data import TensorDataset
train_ds = TensorDataset(x_train, y_train)
train_dl    =    DataLoader(train_ds,    batch_size=bs,
shuffle=True)
valid_ds = TensorDataset(x_valid, y_valid)
valid_dl = DataLoader(valid_ds, batch_size=bs * 2)
for epoch in range(epochs):
    print('Epoch:  ',epoch)
    model.train()
    for xb, yb in train_dl:
        pred = model(xb)
        loss = loss_func(pred, yb)
        loss.backward()
        opt.step()
        opt.zero_grad()
    model.eval()
    with torch.no_grad():
        valid_loss  =  sum(loss_func(model(xb),  yb)  for
```

```
xb, yb in valid_dl)
    print(epoch, valid_loss / len(valid_dl))
print('Accuracy:',accuracy(model(x_test),y_test)*100)
```

Bây giờ, chúng ta sẽ kết hợp các lớp trong mô hình sequential của Torch để thiết kế một mạng CNN có cấu trúc gồm 1 lớp Convolution, 1 lớp Max pooling, 2 lớp dense. Trong ví dụ này, chúng ta sử dụng 60,000 mẫu cho việc huấn luyện, thay vì 50,000 mẫu huấn luyện và 10,000 mẫu cho validation như ví dụ trước. Mạng CNN được thiết kế và huấn luyện với chương trình sau đây. Trong chương trình ví dụ, chúng ta cũng khai thác khả năng thực thi trên GPU nếu như máy tính chúng ta có sử dụng card tăng tốc đồ hoạ (Graphic Processing Unit).

```
from pathlib import Path
import pickle
import gzip
import torch
import torch.nn as nn
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torch.utils.data import TensorDataset
DATA_PATH = Path("data")
PATH = DATA_PATH / "mnist"
PATH.mkdir(parents=True, exist_ok=True)
FILENAME = "mnist.pkl.gz"
with gzip.open((PATH / FILENAME).as_posix(), "rb") as
f:
        ((x_train, y_train), (x_valid, y_valid),
(x_test, y_test)) = pickle.load(f, encoding="latin-1")
from matplotlib import pyplot
import numpy as np
print(x_test.shape)
pyplot.imshow(x_train[0].reshape((28, 28)),
cmap="gray")
pyplot.show()
print(x_train.shape)
import torch
```

```
x_train, y_train, x_valid, y_valid, x_test, y_test =
map(
    torch.tensor,     (x_train,     y_train,     x_valid,
y_valid,x_test,y_test))
n, c = x_train.shape
import torch
import torch.nn as nn
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, 32, kernel_size=3, padding=1),
           # nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(32,       32,       kernel_size=3,
padding=1),
            #nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2)
        )
        self.Fc = nn.Sequential(
        nn.Linear(14*14*32, 512),
        nn.Sigmoid(),
        nn.Linear(512, 10),
        nn.Softmax(dim=1)
        )
    def forward(self, x):
        conv1 = self.conv1(x)
        flat  = conv1.view(conv1.size(0), -1)
        out = self.Fc(flat)
        return out
def evaluate(net, loader, device):
```

```
    net.eval()
    num_correct, num_total = 0, 0

    with torch.no_grad():
        for inputs in loader:
            images = inputs[0].to(device)
            labels = inputs[1].to(device)
            outputs = net(images)
            _, preds = torch.max(outputs.detach(), 1)
            num_correct += (preds == labels).sum().item()
            num_total += labels.size(0)
    return num_correct / num_total
def train(max_epochs,batch_size,lr):
    # prepare the MNIST dataset
    train_samples=torch.cat((x_train,x_valid)).view(-1, 1, 28, 28)
    train_labels=torch.cat((y_train, y_valid))
    test_samples=x_test.view(-1, 1, 28, 28)
    train_dataset = TensorDataset(train_samples, train_labels)
    test_dataset = TensorDataset(test_samples, y_test)
    # create the data loader
    train_loader = DataLoader(dataset=train_dataset,
                              batch_size=batch_size,
                              shuffle=True, drop_last=True)
    test_loader = DataLoader(dataset=test_dataset,
                             batch_size=batch_size,
                             shuffle=False)
    # turn on the CUDA if available
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    net = Net().to(device)
    loss_op = nn.CrossEntropyLoss()
    optim   = torch.optim.Adam(net.parameters(), lr=lr)
```

```
    for epoch in range(max_epochs):
        net.train()
        for step, inputs in enumerate(train_loader):
            images = inputs[0].to(device)
            labels = inputs[1].to(device)
            # forward-propagation
            outputs = net(images)
            loss = loss_op(outputs, labels)
            # back-propagation
            optim.zero_grad()
            loss.backward()
            optim.step()
        acc = evaluate(net, test_loader, device)
        print("Epoch  [{}/{}]  loss:  {:.5f}  test  acc:
{:.3f}".format(epoch+1, max_epochs, loss.item(), acc))

    torch.save(net.state_dict(), "mnist-final.pth")
```

TÀI LIỆU THAM KHẢO

1. Tom. M. Mitchell, *Machine Learning*, McGraw-Hill, 1997
2. Simon Haykin, *Neural Networks: A Comprehensive Foundation, Pearson Prentice Hall*, 2005
3. George F Luger, *Artificial intelligence, structure and strategies for complex problem solving*, Peason Education, Inc., 2009
4. Ian Goodfellow, Yoshua Bengio, Aaron Courville, *Deep learning*, The MIT Press, Cambridge, Massachusetts, London, England, 2016
5. Daniel Graupe, *Principles of Artificial Neural Networks*, World Scientific Publishing Co. Pte. Ltd., 2007
6. Jason Brownlee, *Deep Learning for Natural Language Processing: Develop Deep Learning Models for Natural Language in Python*, Machine Learning Mastery, 2017
7. Shai Shale-Shwartz, Shai Ben-David, *Understanding Machine Learning from Theory to Algorithms*, Cambridge University Press, 2014
8. Simon Haykin, *Neural Network and Machine Learning*, Pearson Education, 2009
9. Navin Kuma Manaswi, *Deep learning with Application Using Python: Chabot and Face, Object and Speech Recognition with Tensorflow and Keras*, Springer Science, 2018
10. Francois Chollet, *Deep Learning with Python*, Manning Publication Co., 2018
11. Eli Stevens, Luca Antiga, Deep learning with Pytorch, Manning Publications Co, 2019
12. Palash Goyal, Sumit Panday, Karan Jain, *Deep Learning for Natural Language Procesing: Creating Neural Netork with Python*, Springer Science, 2018
13. Antonio Gulli, Sujit Pal, *Deep Learning with Keras*, Packt Publishing, 2017
14. Willi Richert, Luis Pedro Coelho, *Building Machine Learning Systems with Python*, Packt Publishing Ltd., 2013

15. Santanu Pattanayak, *Pro Deep Learning with Tensorflow: A Mathematical Approach to Advanced Artificial Intelligence in Python*, Apress, 2017
16. Son Ngoc Truong, “A Low-cost Artificial Neural Network Model for Raspberry Pi”, Engineering, Technology & Applied Science Research, vol. 10, no. 2, pp. 5466-5469,Apr. 2020 (ESCI)
17. Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, Journal of Machine Learning Research 15 (2014) 1929-1958
18. Taylor Simons and Dah-Jye Lee, “A Review of Binarized Neural Networks”, Electronics vol. 8, no. 661, 2019
19. Courbariaux, M.; Bengio, Y. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. arXiv 2016, arXiv:1602.02830

Huấn luyện Mạng Nơ-ron học sâu (Sách chuyên khảo)

Trương Ngọc Sơn, Phạm Ngọc Sơn, Lê Minh
Trường Đại học Sư phạm Kỹ thuật Thành phố Hồ Chí Minh

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

Trụ sở:
Phòng 501, Nhà Điều hành ĐHQG-HCM, phường Linh Trung, thành phố Thủ Đức, Thành phố Hồ Chí Minh.
ĐT: 028 62726361
E-mail: vnuhp@vnuhcm.edu.vn

Văn phòng đại diện:
Tòa nhà K-Trường Đại học Khoa học Xã hội & Nhân văn, số 10-12 Đinh Tiên Hoàng, phường Bến Nghé, Quận 1,Thành phố Hồ Chí Minh
ĐT: 028 62726390
Website: www.vnuhcmpress.edu.vn

Chịu trách nhiệm xuất bản và nội dung

TS ĐỖ VĂN BIÊN

Biên tập
LÊ THỊ THU THẢO

Sửa bản in
QUANG NHẬT

Trình bày bìa
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT THÀNH PHỐ HỒ CHÍ MINH

Đối tác liên kết
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT THÀNH PHỐ HỒ CHÍ MINH

Xuất bản lần thứ 1. Số lượng in: 250 cuốn, khổ 16 x 24cm. Số XNĐKXB: 2303-2022/CXBIPH/23-29/ĐHQGTPHCM. QĐXB số: 195/QĐ-NXB cấp ngày 12/7/2022. In tại: Công ty TNHH In và Bao bì Hưng Phú; Địa chỉ: 162A/1, KP1A, phường An Phú, TP Thuận An, Bình Dương. Nộp lưu chiểu: Năm 2022. ISBN: **978-604-73-9195-0.**

Bản quyền tác phẩm đã được bảo hộ bởi Luật Xuất bản và Luật Sở hữu trí tuệ Việt Nam. Nghiêm cấm mọi hình thức xuất bản, sao chụp, phát tán nội dung khi chưa có sự đồng ý của tác giả và Nhà xuất bản.

ĐỂ CÓ SÁCH HAY, CẦN CHUNG TAY BẢO VỆ TÁC QUYỀN!

ISBN:978-604-73-9195-0
NXB ĐHQG-HCM
9 786047 391950

CHÍNH SÁCH CHẤT LƯỢNG

Không ngừng nâng cao chất lượng dạy, học, nghiên cứu khoa học và phục vụ cộng đồng nhằm mang đến cho người học những điều kiện tốt nhất để phát triển toàn diện các năng lực đáp ứng nhu cầu phát triển và hội nhập quốc tế.

ISBN: 978-604-73-9195-0
NXB ĐHQG-HCM
9 786047 391950