

Deep Learning

Cho ứng dụng nhận dạng ảnh trong PyThon

Hoàng Hữu Việt, Ph.D

Ngày 29 tháng 9 năm 2018

© 2018 Dự thảo

Tất cả các quyền được bảo vệ. Không có phần nào của tài liệu này được sao chép dưới mọi hình thức mà không có sự cho phép bằng văn bản của tác giả.

ISBN XXX-XX-XXXX-XXX-X
D/XXXX/XXXX/XX

Mục lục

Mục lục	i
Danh sách hình vẽ	v
1 Tổng quan về học sâu	1
1.1 Tổng quan về Trí tuệ nhân tạo, Học máy và Học sâu	1
1.2 Phân loại học máy	1
1.2.1 Học có giám sát	1
1.2.2 Học không có giám sát	2
1.2.3 Học tăng cường	2
1.3 Chu trình thiết kế hệ thống nhận dạng mẫu	3
1.3.1 Chu trình thiết kế	3
1.3.2 Tiền xử lý dữ liệu	3
1.3.3 Trích chọn đặc trưng	4
1.3.4 Lựa chọn mô hình	5
2 Mạng neuron truyền thông	7
2.1 Mô hình neuron	7
2.1.1 Neuron một đầu vào	7
2.1.2 Hàm truyền	7

2.1.3	Neuron nhiều đầu vào	8
2.2	Kiến trúc mạng neuron	9
2.2.1	Mạng neuron một tầng	9
2.2.2	Mạng neuron đa tầng	10
3	Lập trình mạng neuron với Keras	13
3.1	Giới thiệu về Keras	13
3.2	Biểu diễn dữ liệu cho mạng neuron	14
3.2.1	Số vô hướng	14
3.2.2	Vectors	15
3.2.3	Ma trận hai chiều	15
3.2.4	Ma trận ba chiều	15
3.2.5	Biểu diễn dữ liệu cho ứng dụng nhận dạng	16
3.3	Lập trình mạng neuron với Keras	17
3.3.1	Một số ví dụ	18
3.3.2	Thiết lập các tham số của mạng	25
3.4	Đánh giá mô hình học máy	26
3.5	Overfitting và underfitting	32
3.5.1	Giảm kích thước mạng	33
3.5.2	Adding weight regularization	35
3.5.3	Adding dropout	38
4	Mạng nhân chập cho nhận dạng ảnh	43
4.1	Toán tử nhân chập	43
4.2	Toán tử Pooling	43
4.3	Kiến trúc mạng nhân chập	43
4.4	Ứng dụng mạng nhân chập cho bài toán phân lớp hình ảnh	44

4.4.1	Tầng nhân chập	46
4.4.2	Tầng max-pooling	51
4.5	Lập trình mạng nhận chập cho ứng dụng nhận dạng ảnh .	52
4.5.1	Nhận dạng các chữ số viết tay	53
4.5.2	Nhận dạng mặt người	56
4.5.3	Nhận dạng ảnh màu	59
Tài liệu tham khảo		65
Chỉ mục		65

Danh sách hình vẽ

2.1	Neuron một đầu vào	8
2.2	Neuron nhiều đầu vào	9
2.3	Mạng neuron một tầng	10
2.4	Mạng neuron 3 tầng	11
2.5	Mạng neuron 3 tầng biểu diễn dạng ma trận	12
3.1	Số người tìm kiếm Keras trên Google	14
3.2	Biểu diễn dữ liệu thời gian và hình ảnh	17
3.3	100 số chữ số viết tay đầu tiên trong cơ sở dữ liệu huấn luyện của MNIST	22
3.4	Mô hình huấn luyện của mạng neuron	25
3.5	Tạo tập dữ liệu đánh giá đơn giản	27
3.6	Validation-loss	29
3.7	Validation-loss	29
3.8	3-fold cross-validation	30
3.9	4-fold cross-validation	32
3.10	Ảnh hưởng của hàm mất mát dựa trên kích thước mạng cho cơ sở dữ liệu MNIST	36
3.11	Ảnh hưởng của hàm mất mát dựa trên L2 cho cơ sở dữ liệu MNIST	38

3.12 Dropout cho đầu ra của một tầng mạng với tỷ lệ 50%	39
3.13 Ảnh hưởng của hàm mất mát dựa trên adding dropout cho cơ sở dữ liệu MNIST	41
4.1 Kiến trúc mạng nhân chập	44
4.2 Mô hình mạng nhân chập	45
4.3 Mô hình mạng nhân chập	46
4.4 Ảnh được chia thành các mảng cục bộ dựa trên các cạnh .	47
4.5 Phân cấp không gian các mảng	48
4.6 Nguyên lý làm việc của tầng nhân chập	49
4.7 Hiệu ứng biên	50
4.8 Mở rộng biên	50
4.9 convolution-stride	51
4.10 Ví dụ về max-pooling	51
4.11 100 ảnh đầu tiên trong cơ sở dữ liệu MNIST	54
4.12 Thử nghiệm với kiến trúc mạng 1	55
4.13 Thử nghiệm với kiến trúc mạng 2	55
4.14 Thử nghiệm với kiến trúc mạng 3	56
4.15 100 ảnh trong cơ sở dữ liệu IFD	57
4.16 Thử nghiệm với kiến trúc mạng 1	58
4.17 Thử nghiệm với kiến trúc mạng 2	58
4.18 Thử nghiệm với kiến trúc mạng 3	59
4.19 100 ảnh đầu tiên trong cơ sở dữ liệu CIFAR10	60
4.20 Thử nghiệm với kiến trúc mạng 1	61
4.21 Thử nghiệm với kiến trúc mạng 2	61
4.22 Thử nghiệm với kiến trúc mạng 3	62
4.23 Thử nghiệm với kiến trúc mạng 4	63

Chương 1

Tổng quan về học sâu

1.1 Tổng quan về Trí tuệ nhân tạo, Học máy và Học sâu

1.2 Phân loại học máy

1.2.1 Học có giám sát

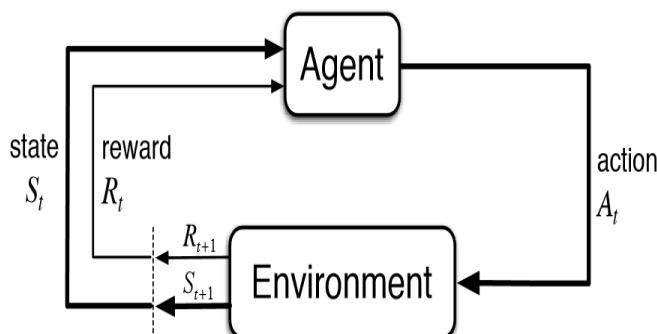
- **Dữ liệu (data):**
 - $D = \{d_1, d_2, \dots, d_n\}$ là một tập n mẫu (examples).
 - Mỗi mẫu $d_i = (x_i, y_i)$, $i = 1, 2, \dots, n$ trong đó x_i là một vector vào và y_i là đầu ra mong muốn (được đưa bởi người giám sát - teacher)
- **Mục tiêu (objective):** học một hàm $f : X \rightarrow Y$ sao cho $y_i \approx f(x_i)$ với $i = 1, 2, \dots, n$.
- **Hai loại bài toán:**
 - Regression: (X rời rạc hoặc liên tục) và (Y liên tục).
 - Classification: (X rời rạc hoặc liên tục) và (Y rời rạc)
- **Ví dụ bài toán phân lớp (regression):**

1.2.2 Học không có giám sát

- **Dữ liệu (data):**
 - $D = \{d_1, d_2, \dots, d_n\}$ là một tập n mẫu (samples).
 - Mỗi mẫu $d_i = (x_i), i = 1, 2, \dots, n$
 - x_i là một vector vào, không có giá trị là đầu ra mong muốn (không biết $y_i, i = 1, 2, \dots, n$)
- **Mục tiêu (objective):** học mô hình quan hệ giữ các mẫu.
- **Hai loại bài toán:**
 - Clustering: nhóm các mẫu có tính chất tương tự với nhau.
 - Density estimation: Tìm mật độ xác suất phân bố của các mẫu.
- Ví dụ: ...

1.2.3 Học tăng cường

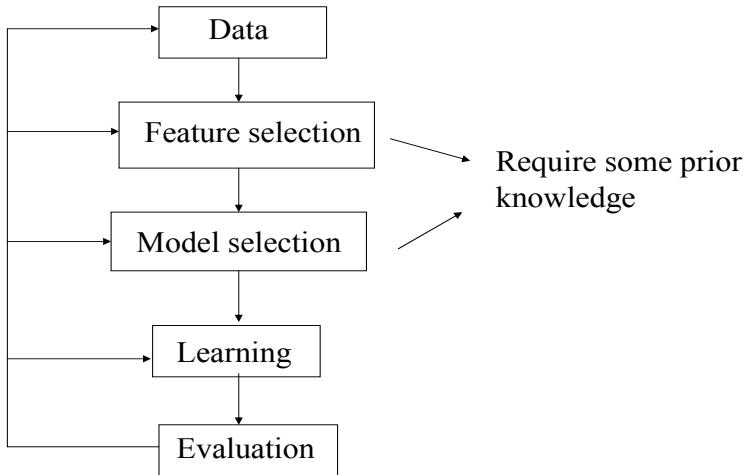
- Tác nhân (agent) học thông qua tương tác với môi trường: quan sát, thực hiện hành động và nhận được điểm thưởng hoặc điểm phạt.
- Tác nhân học để nhận được điểm thưởng nhiều nhất.



1.3 Chu trình thiết kế hệ thống nhận dạng mẫu

1.3.1 Chu trình thiết kế

Chu trình thiết kế (design cycle)



1.3.2 Tiền xử lý dữ liệu

Ngoài việc đánh giá mô hình, những câu hỏi quan trọng cần phải trả lời trước khi xây dựng mô hình là: làm thế nào để chuẩn bị dữ liệu huấn luyện trước khi đưa vào huấn luyện mạng? những kỹ thuật tiền xử lý dữ liệu và những đặc trưng nào của dữ liệu sẽ được dùng cho bài toán nhận dạng?

Mục đích của tiền xử lý dữ liệu là làm cho dữ liệu thô “đã hiểu” hơn với mạng neuron. Tiền xử lý dữ liệu bao gồm vector hóa dữ liệu, chuẩn hóa, xử lý các giá trị bị thiếu và trích chọn đặc trưng.

Vector hóa dữ liệu

Tất cả các dữ liệu vào (gồm dữ liệu và phân lớp) trong mạng neuron đều phải là các vector kiểu tensor. Bất kể dữ liệu nào cần xử lý — âm thanh, hình ảnh, văn bản — đầu tiên phải chuyển thành tensors hay gọi là vector hóa dữ liệu.

Chuẩn hóa dữ liệu

Nói chung, sẽ không an toàn nếu dữ liệu huấn luyện của mạng neuron có giá trị tương đối lớn, ví dụ một số giá trị có giá trị trong khoảng 0-1, một số giá trị khác trong khoảng 100-200. Điều này dẫn đến trọng số mạng được cập nhật những giá trị lớn và do đó và ngăn chặn sự hội tụ của mạng. Do đó, trước khi huấn luyện mạng, dữ liệu thường phải được chuẩn hóa để có độ lệch chuẩn là 1 và trung bình là 0.

Xử lý các giá trị đang thiếu

Nếu một giá trị dữ liệu bị thiếu trong tập mẫu huấn luyện, thì giá trị dữ liệu nên được gán bằng 0 với điều kiện giá trị 0 không thực sự có ý nghĩa trong dữ liệu. Khi đó mạng sẽ học từ tập mẫu dữ liệu huấn luyện với giá trị 0 có nghĩa là dữ liệu bị thiếu và sẽ bỏ qua giá trị này.

1.3.3 Trích chọn đặc trưng

Trích chọn đặc trưng là quá trình sử dụng kiến thức về dữ liệu và các thuật toán học máy để tạo ra các thuật toán làm việc tốt hơn bằng cách biến đổi dữ liệu trước khi đưa vào mạng neuron. Trong nhiều trường hợp, không có lý do để mong đợi các mô hình học máy có thể học tốt với dữ liệu tùy ý. Dữ liệu cần phải được biểu diễn sao cho mô hình học tốt hơn. Trước khi học sâu xuất hiện, trích chọn đặc trưng là bước quan trọng bởi vì các thuật toán học máy truyền thống không đủ mạnh để học các đặc trưng từ dữ liệu.

Mô hình học sâu đã loại bỏ quá trình trích chọn đặc trưng bởi vì mạng neuron có khả năng tự trích chọn đặc trưng từ dữ liệu thô. Vậy liệu chúng ta không còn lo lắng về kỹ thuật trích chọn đặc trưng khi dùng mạng học sâu? không, vì hai lý do:

- Các đặc trưng tốt cho phép chúng ta giải bài toán tốt hơn trong khi dùng ít dữ liệu hơn.
- Các đặc trưng tốt cho phép giải bài toán ít dữ liệu hơn. Nếu chúng ta chỉ có một ít mẫu dữ liệu, đặc trưng dữ liệu trở nên quan trọng hơn.

1.3.4 Lựa chọn mô hình

<trang 97> Trong ví dụ nhận dạng chữ số viết tay trong cơ sở dữ liệu MNIST, chúng ta chia dữ liệu thành ..

Trong học máy, mục đích là đạt được các mô hình tổng quát, tức là các mô hình mà thực hiện tốt nhất trên các dữ liệu chưa biết. Tiếp theo,

...

Chương 2

Mạng neuron truyền thẳng

2.1 Mô hình neuron

2.1.1 Neuron một đầu vào

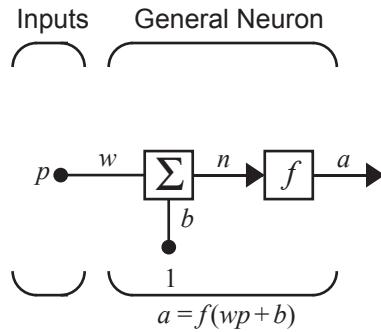
Hình 2.1 mô tả neuron có một đầu vào, trong đó:

- p là một số vô hướng.
- w là một số vô hướng, gọi là trọng số (weight).
- b là một số vô hướng, gọi là độ lệch (bias).
- $n = wp + b$, gọi là đầu vào của mạng.
- f là hàm truyền của mạng.
- $a = f(n) = f(wp + b)$ là đầu ra của mạng.

Ví dụ: Nếu $p = 2, w = 3, b = -1.5$ thì $n = 4.5$ và $a = f(4.5)$. Như vậy đầu ra của mạng sẽ phụ thuộc vào hàm truyền f .

2.1.2 Hàm truyền

Hàm truyền của mạng có thể là một hàm tuyến tính hoặc một hàm phi tuyến của n . Một số hàm truyền thường được sử dụng trong mạng neuron được chỉ ra như bảng 2.1.



Hình 2.1: Neuron một đầu vào

2.1.3 Neuron nhiều đầu vào

Một neuron có \$R\$ đầu vào được mô tả như hình 2.2, trong đó

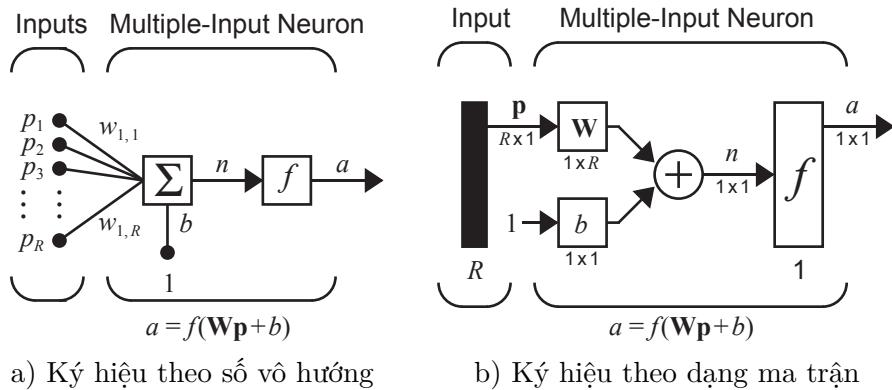
- \$p_1, p_2, \dots, p_R\$ là các số vô hướng.
- \$w_{1,1}, w_{1,2}, \dots, w_{1,R}\$ là các số vô hướng, gọi là trọng số (weight).
- \$b\$ là một số vô hướng, gọi là độ lệch (bias).

Bảng 2.1: Hàm truyền

Hàm	Vào/Ra	Tên hàm trong Keras
Exponential Linear Unit	$a = \begin{cases} n & \text{nếu } n > 0 \\ \alpha(e^n - 1) & \text{nếu } n < 0 \end{cases}$	elu
Scaled Exponential Linear Unit	$a = \alpha \begin{cases} n & \text{nếu } n > 0 \\ \alpha(e^n - 1) & \text{nếu } n < 0 \end{cases}$	selu
softplus	$a = \log_n(1 + e^n)$	softplus
Rectified Linear Unit	$a = \max(0, n)$	relu
Tanh	$a = \frac{2}{1+e^{-2n}} - 1$	tanh
sigmoid	$a = \frac{1}{1+e^{-n}}$	sigmoid
Linear	$a = n$	linear

- $n = w_{1,1}p_1 + w_{1,2}p_2 + \cdots + w_{1,R}p_R + b$, gọi là đầu vào của mạng.
- f là hàm truyền của mạng.

Nếu viết theo dạng ma trận, đầu vào của mạng là $n = Wp + b$, trong đó $p = [p_1, p_2, \dots, p_R]^T$ và $W = [w_{1,1}, w_{1,2}, \dots, w_{1,R}]$ với ký hiệu T là véc tơ chuyển vị. Đầu ra của mạng là $a = f(Wp + b)$.



Hình 2.2: Neuron nhiều đầu vào

2.2 Kiến trúc mạng neuron

2.2.1 Mạng neuron một tầng

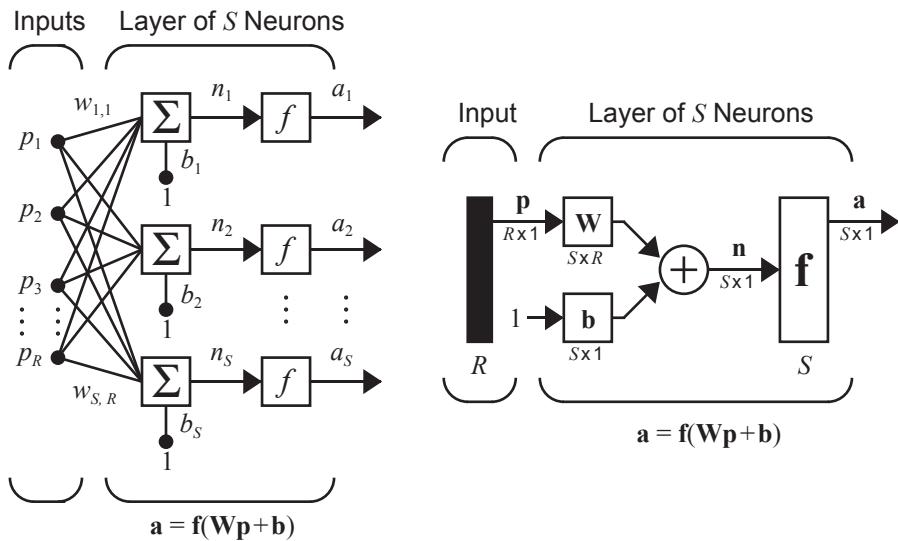
Một mạng neuron một tầng có s neuron được mô tả như hình ?. Mỗi phần tử của véc tơ $p = [p_1, p_2, \dots, p_R]^T$ được nối với mỗi neuron thông qua các đầu vào của ma trận trọng số $W = [w_{i,j}], i = 1, 2, \dots, s$ và $j = 1, 2, \dots, R$. Chú ý rằng thường $R \neq s$ và hàm truyền của các neuron có thể khác nhau.

Ví dụ: Cho các tham số của mạng như sau:

$$W = \begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \end{bmatrix} \text{ và } b = \begin{bmatrix} -2 \\ 1 \end{bmatrix}.$$

Hàm truyền của mạng là

$$f(n) = \begin{cases} 1 & \text{nếu } n \leq 0, \\ -1 & \text{nếu } n < 0. \end{cases}$$



a) Ký hiệu theo số vô hướng

b) Ký hiệu theo dạng ma trận

Hình 2.3: Mạng neuron một tầng

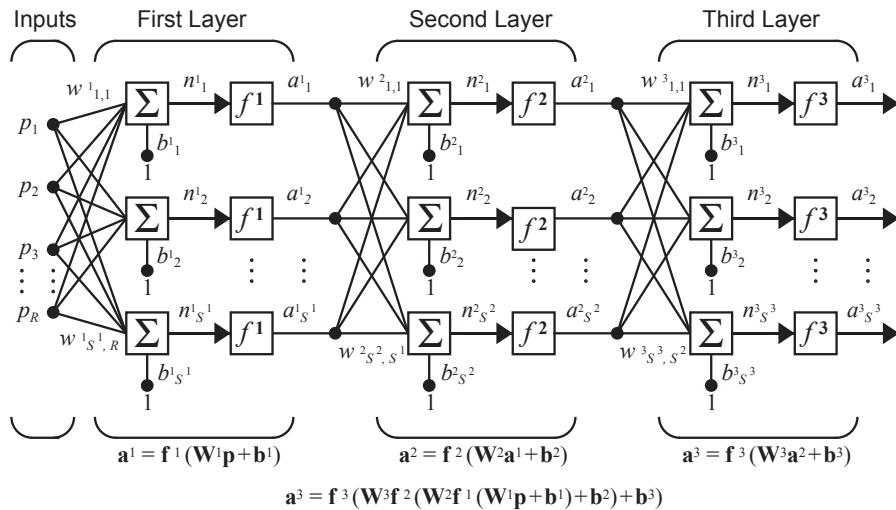
Hãy vẽ mang và tính đầu ra của mang với các mẫu đầu vào như sau:

$$p_1 = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \text{ và } p_2 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

2.2.2 Mang neuron đa tầng

Một mạng neuron đa tầng là một mạng neuron gồm các mạng neuron một tầng ghép nối tiếp nhau. Mỗi tầng có ma trận trọng số W , vector độ lệch b , vector đầu vào mạng n và vector đầu ra mạng a . Chúng ta sẽ sử dụng chỉ số trên cho ma trận trọng số, vector độ lệch, vector đầu vào và vector đầu ra để phân biệt ký hiệu giữa các tầng. Ký hiệu W^i là ma trận trọng số, b^i là vector độ lệch, n^i là vector đầu vào và a^i là vector đầu ra của tầng i . Ví dụ ký hiệu W^1 là ma trận trọng số, b^1 là vector độ lệch, n^1 là vector đầu vào và a^1 là vector đầu ra của tầng 1.

Hình 2.4 chỉ ra một mạng neuron 3 tầng, trong đó đầu ra của tầng 1 là đầu vào của tầng 2 và đầu ra của tầng 2 là đầu vào của tầng 3. Hình 2.5 chỉ ra mang biểu diễn theo dạng ma trận.

**Hình 2.4:** Mạng neuron 3 tầng

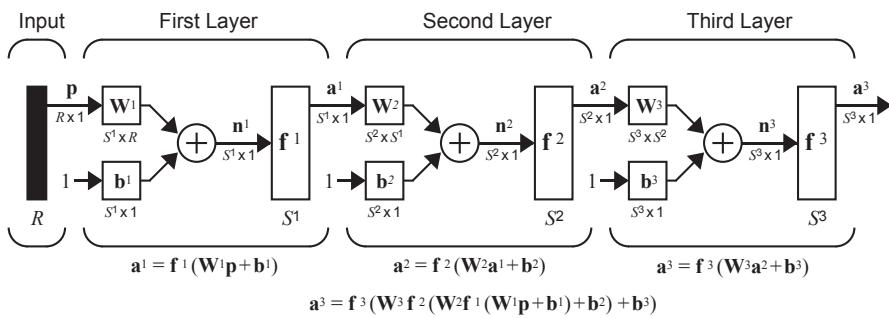
Ví dụ: Cho các tham số của mạng như sau:

$$W^1 = \begin{bmatrix} 1 & 1 & 2 & 2 \\ -2 & -2 & 1 & 1 \end{bmatrix}; b^1 = \begin{bmatrix} -2 \\ 1 \end{bmatrix} \text{ và } f^1(n) = \begin{cases} 1 \text{ nếu } n \leq 0, \\ -1 \text{ nếu } n < 0. \end{cases}$$

$$W^2 = \begin{bmatrix} 1 & 1 \end{bmatrix}; b^2 = 2 \text{ và } f^2(n) = \begin{cases} 1 \text{ nếu } n \leq 0, \\ 0 \text{ nếu } n < 0. \end{cases}$$

Hãy vẽ mạng và tính đầu ra của mạng với các mẫu đầu vào như sau:

$$p_1 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \text{ và } p_2 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix}.$$



Hình 2.5: Mạng neuron 3 tầng biểu diễn dạng ma trận

Chương 3

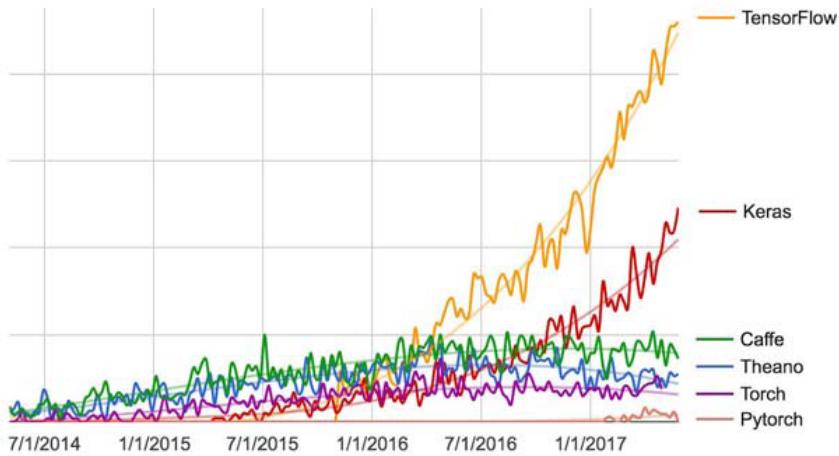
Lập trình mạng neuron với Keras

3.1 Giới thiệu về Keras

Keras là một thư viện cho Python mà cung cấp một cách thuận tiện để định nghĩa và huấn luyện hầu hết các mô hình học sâu (deep learning model). Ban đầu Keras được phát triển cho những người nghiên cứu với mục đích thiết lập môi trường thử nghiệm nhanh nhất. Keras có các đặc trưng chính sau:

- Cho phép chạy cùng mã lệnh trên các bộ vi xử lý CPU và GPU.
- Cung cấp các giao diện API thân thiện để phát triển nhanh các mô hình học sâu.
- Hỗ trợ mạng nhân chập (cho nhận dạng ảnh), mạng hồi quy (cho xử lý chuỗi) và sự kết hợp giữa mạng nhân chập và mạng hồi quy.
- Phù hợp với mọi mô hình mạng học sâu.

Keras được cung cấp miễn phí bởi MIT, điều này có nghĩa rằng Keras có thể được dùng miễn phí trong các dự án thương mại. Keras tương thích với các phiên bản của Python từ 2.7 đến 3.6 (giữa năm 2017). Cho đến nay, Keras có hơn 200,000 người dùng, từ các nhà nghiên cứu, các kỹ sư ở các công ty khởi nghiệp cũng như ở các công ty lớn đến các sinh viên ở các trường đại học. Keras được sử dụng ở Google, Netflix,



Hình 3.1: Số người tìm kiếm Keras trên Google

Uber, CERN, Yelp, Square, và ở hàng trăm công ty khởi nghiệp và được áp dụng để giải quyết nhiều bài toán. Hình 3.1 cho biết số người tìm kiếm Keras trên Google.

3.2 Biểu diễn dữ liệu cho mạng neuron

Hiện nay tất cả các hệ thống học máy sử dụng **tensors** như là những cấu trúc dữ liệu cơ bản. Bản chất của **tensors** là các mảng dữ liệu số.

3.2.1 Số vô hướng

Một **tensor** chỉ chứa một giá trị số gọi là **số vô hướng (scalar)**, hay còn gọi là **scalar tensor** hoặc **0-dimensional tensor (0D tensor)**. Trong Numpy, một số thực là một số vô hướng (**scalar tensor**) và cũng có thể được xem như là một mảng chỉ có một phần tử. Ví dụ nếu khai báo

```
import numpy as np
x = np.array(12)
```

thì `x` là một số vô hướng và cũng được xem như là một mảng chỉ có một phần tử, tức là số chiều của mảng bằng 0.

3.2.2 Vectors

Một **vector** là một mảng các số và được gọi là 1D **tensor**. Ví dụ nếu khai báo

```
import numpy as np  
x = np.array([12, 3, 6, 14])
```

thì `x` là một vector có 5 phần tử và `x` được gọi là một vector 5 chiều (**dimension**).

3.2.3 Ma trận hai chiều

Một ma trận hai chiều là một mảng của các **vector** và được gọi là 2D **tensors**. Một ma trận thường có 2 trục (**axes**) và thường được xem là các dòng (**rows**) và các cột (**columns**). Ví dụ nếu khai báo

```
import numpy as np  
x = np.array([[5, 78, 2, 34, 0],  
              [6, 79, 3, 35, 1],  
              [7, 80, 4, 36, 2]])
```

thì `x` là một ma trận 2 chiều có kích thước 3×4 .

3.2.4 Ma trận ba chiều

Một ma trận 3 chiều là 3 ma trận hai chiều được ghép với nhau và được gọi là 3D **tensors**. Ví dụ nếu khai báo

```
import numpy as np  
x = np.array([[[5, 78, 2, 34, 0],  
               [6, 79, 3, 35, 1],  
               [7, 80, 4, 36, 2]],  
              [[[5, 78, 2, 34, 0],  
                [6, 79, 3, 35, 1],  
                [7, 80, 4, 36, 2]],  
              [[[5, 78, 2, 34, 0],  
                [6, 79, 3, 35, 1],  
                [7, 80, 4, 36, 2]]]])
```

thì `x` là một ma trận 3 chiều có kích thước $3 \times 3 \times 5$.

3.2.5 Biểu diễn dữ liệu cho ứng dụng nhân dạng

Dữ liệu vector

Dữ liệu của một vector được biểu diễn bởi một 1D **tensor** và do đó một lô (batch) dữ liệu vector được biểu diễn bởi một 2D **tensors** hay là một ma trận hai chiều theo dạng (**samples**, **features**). Điều này nghĩa là mỗi dòng biểu diễn đặc các trường (**features**) của các mẫu (**samples**). Ví dụ bài toán phân lớp dữ liệu với các mẫu huấn luyện như sau:

$$\left\{ p_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}, \left\{ p_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, t_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\},$$

$$\left\{ p_3 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, t_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}, \left\{ p_4 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, t_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\},$$

$$\left\{ p_5 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, t_5 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}, \left\{ p_6 = \begin{bmatrix} -2 \\ 1 \end{bmatrix}, t_6 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\},$$

$$\left\{ p_7 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, t_7 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}, \left\{ p_8 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}, t_8 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

Khi đó, kiểu dữ liệu được biểu diễn như sau:

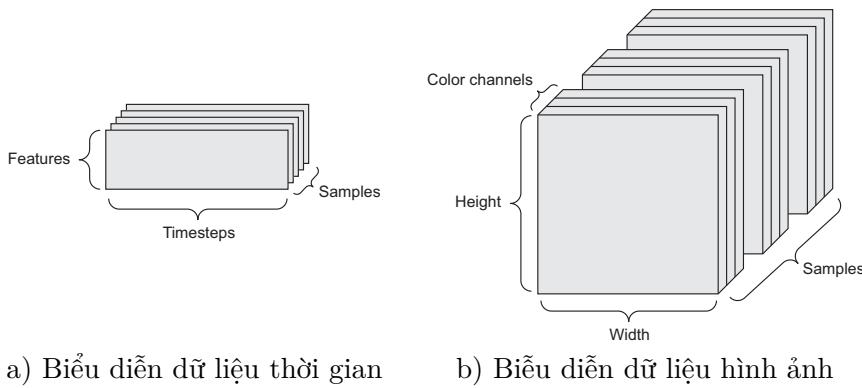
```
import numpy as np
#the traing data
train_data=np.array([[ 1, 1],[ 1, 2],[ 2,-1],[ 2, 0],
                     [-1, 2],[-2, 1],[-1,-1],[-2,-2]])
#the training label
train_label=np.array([[0,0],[0,0],[0,1],[0,1],
                      [1,0],[1,0],[1,1],[1,1]])
```

Dữ liệu chuỗi thời gian

Dữ liệu chuỗi thời gian (**timeseries data**) được biểu diễn bởi một ma trận hai chiều hay một 2D **tensors** theo dạng (**timesteps**, **features**) và do đó một lô (batch) dữ liệu chuỗi thời gian được biểu diễn bởi một 3D **tensors** theo dạng (**samples**, **timesteps**, **features**) như minh họa ở hình 3.2 (a).

Dữ liệu ảnh

Dữ liệu của một ảnh (image data) được biểu diễn bởi ma trận ba chiều hay một 3D tensors của (height, width, color_depth). Ví dụ một ảnh đa mức xám có kích thước 256×256 được biểu diễn bởi 1 ma trận ($256, 256, 1$). Một lô (batch) của các ảnh được biểu diễn bởi một 4D tensors theo dạng (samples, height, width, color_depth) như minh họa hình 3.2 (b). Ví dụ một lô 128 ảnh màu với mỗi ảnh có kích thước 256×256 sẽ được biểu diễn theo dạng ($128, 256, 256, 3$)



Hình 3.2: Biểu diễn dữ liệu thời gian và hình ảnh

Dữ liệu Video

Dữ liệu của một video được biểu diễn bởi một 4D tensors theo dạng (frames, height, width, color_depth). Một lô (batch) của các video được biểu diễn bởi một 5D tensors theo dạng (samples, frames, height, width, color_depth). Ví dụ 4 videos với mỗi video có 240 khung (frame) và mỗi khung có kích thước 144×256 được lưu trữ trong một 5D tensors theo dạng ($4, 240, 144, 256, 3$).

3.3 Lập trình mạng neuron với Keras

Các bước cơ bản để lập trình mạng Perceptron với Keras bao gồm:

1. Khai báo các thư viện.

2. Định nghĩa dữ liệu huấn luyện, bao gồm các mẫu dữ liệu vào và đầu ra tương ứng.
3. Tạo mạng neuron.
4. Cấu hình mạng neuron cho quá trình huấn luyện.
5. Huấn luyện mạng.
6. Nhận dạng các mẫu dữ liệu.

3.3.1 Một số ví dụ

Ví dụ 3.1. Phân lớp dữ liệu với các mẫu dữ liệu huấn luyện như sau:

$$\begin{aligned} & \left\{ p_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}, \left\{ p_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, t_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}, \\ & \left\{ p_3 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, t_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}, \left\{ p_4 = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, t_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}, \\ & \left\{ p_5 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, t_5 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}, \left\{ p_6 = \begin{bmatrix} -2 \\ 1 \end{bmatrix}, t_6 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}, \\ & \left\{ p_7 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, t_7 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}, \left\{ p_8 = \begin{bmatrix} -2 \\ -2 \end{bmatrix}, t_8 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}. \end{aligned}$$

Chú ý: Ký hiệu $p_i (i = 1, 2, \dots, 8)$ là dữ liệu đầu vào và t_i là lớp dữ liệu đầu ra tương ứng với đầu vào p_i .

Khai báo các thư viện

Đầu tiên chúng ta cần phải khai báo các thư viện Numpy và Keras. Thư viện Numpy dùng để định nghĩa các mẫu dữ liệu huấn luyện, thư viện Keras dùng để định nghĩa mạng mạng neuron.

```
import warnings
warnings.filterwarnings('ignore')

import numpy as np
from keras import models
from keras import layers
```

Định nghĩa dữ liệu huấn luyện

Dữ liệu huấn luyện bao gồm dữ liệu vào và dữ liệu ra và được lưu trữ trong 2D **tensors**. Dữ liệu vào là một ma trận kích thước 8×2 trong đó mỗi dòng của ma trận là một mẫu dữ liệu vào. Dữ liệu ra cũng là một ma trận kích thước 8×2 trong đó mỗi dòng của ma trận là đầu ra tương ứng với mẫu dữ liệu vào. Dữ liệu huấn luyện được tạo như sau:

```
#the traing data
train_data = np.array([[1,1],[1,2],[2,-1],[2,0],
                      [-1,2],[-2,1],[-1,-1],[-2,-2]],
                     "float32")

#the training targets
train_targets = np.array([[0,0],[0,0],[0,1],[0,1],
                          [1,0],[1,0],[1,1],[1,1]],
                         "float32")
```

Tạo mạng neuron

Tạo mạng neuron bao gồm tạo mô hình mạng và tạo các tầng mạng. Trong ví dụ này, mô hình mạng neuron được tạo là một mạng neuron truyền thẳng (**sequential**). Vì tập dữ liệu huấn luyện là phân biệt tuyến tính do đó chỉ cần sử dụng một mạng Perceptron. Ngoài ra, bài toán phân 4 lớp dữ liệu do đó tầng được định nghĩa chỉ cần 2 neuron để nhận dạng các mẫu và giả sử hàm truyền được sử dụng là hàm tuyến tính. Mã lệnh tạo ra mạng neuron như sau:

```
model = models.Sequential()
model.add(layers.Dense(2, activation='linear', input_dim=2))
```

Cấu hình mạng neuron

Cấu hình mạng neuron bao gồm thiết lập phương pháp tối ưu và hàm mất mát cho mạng. Giả sử rằng phương pháp tối ưu được chọn là 'sgd' (stochastic gradient descent) và hàm mất mát được chọn là lỗi trung bình bình phương (mean squared error), khi đó mạng được cấu hình như sau:

```
model.compile(optimizer = 'sgd', loss = 'mse'))
```

Huấn luyện mạng neuron

Huấn luyện mạng là quá trình tìm tham số của mạng để tối thiểu hàm mất mát. Giả sử rằng mạng được huấn luyện với số vòng lặp là 500 và kích thước của mỗi gói dữ liệu huấn luyện là 2, khi đó mạng được huấn luyện như sau:

```
model.fit(train_data,train_targets,epochs=500,batch_size=2)
```

Nhận dạng mẫu dữ liệu

Nhận dạng mẫu dữ liệu là đưa vào mạng một tập mẫu và tính đầu ra của tập mẫu qua mạng đã huấn luyện. Ví dụ mã lệnh để nhận dạng lại tập dữ liệu huấn luyện như sau:

```
print(model.predict(train_data).round())
```

Chú ý rằng sau khi mạng đã được huấn luyện, các tham số của mạng bao gồm cả trọng số và độ lệch được hiển thị như sau:

```
print(model.get_weights())
```

Toàn bộ mã lệnh của ví dụ 3.1 được mô tả như sau:

Mã lệnh 3.1: Mã lệnh của ví dụ 3.1

```
1 import warnings
2 warnings.filterwarnings('ignore')
3
4 import numpy as np
5 from keras import models
6 from keras import layers
7
8 #the traing data
9 train_data = np.array([[1,1],[1,2],[2,-1],[2,0],
10                      [-1,2],[-2,1],[-1,-1],[-2,-2]],
11                      "float32")
12 #the training targets
13 train_label= np.array([[0,0],[0,0],[0,1],[0,1],
14                      [1,0],[1,0],[1,1],[1,1]],
15                      "float32")
16 print(train_data);
17 print(train_targets);
18
19 model = models.Sequential()
20 #create a network layer
21 model.add(layers.Dense(2,activation='linear',
22                      input_dim = 2))
23 #configures the network for training
24 model.compile(optimizer = 'sgd',loss = 'mse')
25 #train the network
26 model.fit(train_data,train_targets,epochs=500,batch_size=
27                      2)
28 #generates output predictions for the input samples
29 print(model.predict(train_data).round())
```

```
28 #get weights and bias
29 print(model.get_weights())
```

Ví dụ 3.2. Phân lớp dữ liệu với các mẫu dữ liệu huấn luyện như sau (bài toán XOR):

$$\left\{ p_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, t_1 = 0 \right\}, \left\{ p_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, t_2 = 1 \right\},$$

$$\left\{ p_3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, t_3 = 1 \right\}, \left\{ p_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_4 = 1 \right\}.$$

Điểm khác nhau cơ bản giữa ví dụ này và ví dụ 3.1 là dữ liệu huấn luyện không phân biệt tuyến tính, do đó cần phải dữ dụng một mạng 2 tầng. Về cơ bản, mã lệnh lập trình giống với mã lệnh trong ví dụ 3.1, nhưng dòng 17 tạo thêm một tầng mạng để có được mạng 2 tầng. Ngoài ra, dòng 25 hiển thị tham số của tầng 1 (gồm ma trận trọng số và độ lệch) và dòng 26 hiển thị tham số của tầng 2 của mạng. Chú ý rằng nếu định nghĩa lớp đầu ra cho tập dữ liệu huấn luyện là

$$t_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, t_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_4 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Khi đó dòng 17 cần định nghĩa mạng có 2 neuron thay vì 1 neuron.

Mã lệnh 3.2: Mã lệnh của ví dụ 3.2

```
1 import warnings
2 warnings.filterwarnings('ignore')
3
4 import numpy as np
5 from keras import models
6 from keras import layers
7
8 ##create the training data
9 train_data = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
10 #create the training targets
11 train_targets = np.array([[0],[1],[1],[0]], "float32")
12 #create network model
13 model = models.Sequential()
14 #create the first layer of network
15 model.add(layers.Dense(4, activation='tanh', input_dim=2))
16 #create the second layer of network
```

```

17 model.add(layers.Dense(1, activation = 'sigmoid'))
18 #configure the network for training
19 model.compile(optimizer='sgd', loss='binary_crossentropy',
20 metrics = ['accuracy'])
21 #train the network
22 model.fit(train_data, train_targets, epochs=1000, batch_size
23 =1)
24 #generates output predictions for the input samples
25 print(model.predict(train_data).round())
26 #get weights and bias of the first layer
27 print(model.layers[0].get_weights())
28 #get weights and bias of the second layer
29 print(model.layers[1].get_weights())

```

Ví dụ 3.3. Nhận dạng 10 chữ số viết tay trong cơ sở dữ liệu MNIST [<http://yann.lecun.com/exdb/mnist/>]. Tập dữ liệu huấn luyện bao gồm 60,000 ảnh chữ số viết tay và tập nhận dạng bao gồm 10,000 ảnh chữ số viết tay, mỗi chữ số là một ảnh đa mức xám kích thước 28×28 .

Để đọc cơ sở dữ liệu ảnh vào các mảng dữ liệu bằng Keras, chúng ta sử dụng các lệnh sau:

Mã lệnh 3.3: Đọc cơ sở dữ liệu MNIST



Hình 3.3: 100 số chữ số viết tay đầu tiên trong cơ sở dữ liệu huấn luyện của MNIST

```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels)=
mnist.load_data()
```

Ảnh sau đó được mã hóa thành các mảng kiểu Numpy và lớp dữ liệu tương ứng của các ảnh là mảng các số từ 0 đến 9. Tập lệnh sau sẽ hiển thị kích thước của các mảng `train_images`, `train_labels`, `test_images` và `test_labels`.

Mã lệnh 3.4: Hiển thị kích thước dữ liệu huấn luyện và dữ liệu thử nghiệm

```
print(train_images.shape) #(60000, 28, 28)
print(train_labels) #array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
print(test_images.shape) #(10000, 28, 28)
print(test_labels) #array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

Hình 3.3 chỉ ra 100 số chữ số viết tay trong tập huấn luyện được thực hiện bằng các lệnh sau:

Mã lệnh 3.5: Vẽ hình ảnh các chữ số viết tay

```
import matplotlib.pyplot as plt
fig = plt.figure()
cols = 10
rows = 10
for i in range(1, rows*cols+1):
    digit = train_images[i-1]
    fig.add_subplot(rows, cols, i)
    plt.axis('off')
    plt.imshow(digit, cmap = plt.cm.binary)
plt.show()
```

Mã lệnh lập trình được mô tả như sau. Dòng 9-12 tạo ra một mạng neuron 2 tầng, trong đó tầng 1 có 256 neuron và hàm truyền là hàm '`relu`', tầng 2 có 10 neuron dùng để phân 10 lớp dữ liệu và hàm truyền là hàm '`softmax`'. Dòng 14 chuyển ma trận dữ liệu có kích thước (60000, 28, 28) thành ma trận dữ liệu có kích thước (60000, 28 * 28), tức là mỗi dòng của ma trận `train_images` là dữ liệu của một ảnh chữ số. Dòng 15 chuyển kiểu dữ liệu của ma trận `train_images` từ kiểu số nguyên (`uint8`) thành kiểu số thực (`float`). Tương tự với dòng 16-17 cho ma trận `test_images`. Dòng 19 mở thư viện để sử dụng hàm mã hóa lớp dữ liệu đầu ra. Dòng 20-21 mà hóa lớp dữ liệu đầu ra cho tập dữ liệu huấn luyện và tập dữ liệu nhận dạng. Dòng 23 huấn luyện mạng với 5 vòng lặp với kích thước

gói dữ liệu huấn luyện là 128. Dòng 24 đánh giá tỷ lệ nhận dạng chính xác của mạng và dòng 25 hiển thị tỷ lệ chính xác của mạng.

Kết quả chương chạy chương trình cho độ chính xác `test_acc = 97.75%`.

Mã lệnh 3.6: Mã lệnh của ví dụ 3.3

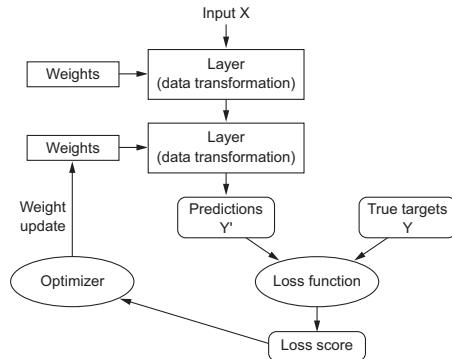
```

1 import warnings
2 warnings.filterwarnings('ignore')
3
4 #import API functions
5 from keras import models
6 from keras import layers
7 from keras.datasets import mnist
8 (train_images,train_labels),(test_images,test_labels) =
    mnist.load_data()
9
10 #normalize the data samples
11 train_images = train_images.reshape((60000,28*28))
12 train_images = train_images.astype('float32')/255
13 test_images = test_images.reshape((10000,28*28))
14 test_images = test_images.astype('float32')/255
15
16 #encode the targets
17 from keras.utils import to_categorical
18 train_labels = to_categorical(train_labels)
19 test_labels = to_categorical(test_labels)
20
21 #create a network model
22 model = models.Sequential()
23 model.add(layers.Dense(256,activation='relu',input_shape
    =(28*28,)))
24 model.add(layers.Dense(10,activation='softmax'))
25 model.compile(optimizer='rmsprop',loss=
    'categorical_crossentropy',metrics=['accuracy'])
26
27 #train the network
28 history = model.fit(train_images,train_labels,epochs = 5,
    batch_size=128)
29
30 #evaluate the trained netwotk
31 test_loss, test_acc = model.evaluate(test_images,
    test_labels)
32 print('test_acc:', test_acc)

```

3.3.2 Thiết lập các tham số của mạng

Mô hình huấn luyện mạng neuron được mô tả như hình 3.4.



Hình 3.4: Mô hình huấn luyện của mạng neuron

Tầng mạng (layers)

Cấu trúc dữ liệu cơ bản của mạng neuron trong Keras là tầng mạng (layer). Một tầng mạng là một mô đun xử lý dữ liệu mà đầu vào là các tensors và đầu ra cũng là các tensors. Tầng mạng có thể được xem như các khối LEGO của học sâu. Xây dựng các mô hình học sâu trong Keras chính là ghép các tầng mạng với nhau.

Mô hình (models)

<Viết chi tiết hơn về các tham số> Mô hình mạng chính là kiến trúc của mạng được định nghĩa....

Hàm mất mát (loss)

Sau khi kiến trúc mạng đã được định nghĩa, cần phải chọn kiểu hàm mất mát (loss function) để cấu hình mạng. Hàm mất mát chính là hàm mục tiêu (objective function) mà sẽ được tối ưu trong khi huấn luyện. Một số hàm mất mát thường dùng gồm:

<Xem tài liệu tham khảo>

Hàm tối ưu (optimizer)

Tiếp theo, cần phải chọn kiểu hàm tối ưu (**Optimizer**) để cấu hình mạng. Hàm tối ưu xác định cách thức mạng sẽ được cập nhật dựa trên hàm mất mát. Một số hàm tối ưu gồm:

3.4 Đánh giá mô hình học máy

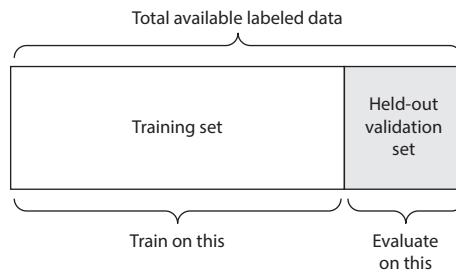
Trong học máy, mục đích là cần phải đạt được một mô hình tổng quát, tức là mạng thực hiện tốt cho cả những dữ liệu chưa biết trước. Đánh giá mô hình thường yêu cầu nhiều tính toán, đặc biệt là với dữ liệu lớn. Tuy nhiên, với bài toán có dữ liệu nhỏ, việc đánh giá mô hình để đạt được một tốt là điều cần thiết. Phần này sẽ đưa ra kỹ thuật để đánh giá mô hình mạng nhằm đạt được mô hình tổng quát.

Đánh giá một mô hình luôn phải chia tách dữ liệu thành ba bộ: huấn luyện (training), đánh giá (validation) và thử nghiệm (test). Mạng được huấn luyện trên bộ dữ liệu huấn luyện và được đánh giá trên bộ dữ liệu đánh giá. Khi có được mô hình đúng đắn, mạng sẽ được sử dụng trên bộ dữ liệu thử nghiệm. Câu hỏi đặt ra là tại sao chúng ta không dùng 2 tập dữ liệu: tập dữ liệu huấn luyện và tập dữ liệu thử nghiệm? chúng ta huấn luyện mạng trên tập dữ liệu huấn luyện và đánh giá trên tập dữ liệu thử nghiệm sẽ đơn giản hơn! Lý do là việc phát triển một mô hình mạng luôn liên quan đến việc điều chỉnh cấu hình của nó: ví dụ như chọn số vòng lặp huấn luyện, chọn số tầng, kích thước của các tầng. Chúng ta điều chỉnh cấu hình bằng cách sử dụng một tín hiệu phản hồi sự thực hiện của mô hình trên dữ liệu đánh giá.

Tạo tập dữ liệu đánh giá đơn giản

Phương pháp này chia dữ liệu đã có thành 2 tập là tập dữ liệu huấn luyện và tập dữ liệu đánh giá như được mô tả ở hình 3.5.

Xét ví dụ bài toán nhận dạng chữ số viết tay trong cơ sở dữ liệu MNIST. Chúng ta sẽ dùng 10000 mẫu dữ liệu đầu tiên trong tập dữ liệu `train_images` làm tập dữ liệu đánh giá và 50000 mẫu dữ liệu còn lại trong tập dữ liệu `train_images` làm tập dữ liệu huấn luyện. Tiếp theo, mạng được huấn luyện trên bộ dữ liệu huấn luyện và được đánh giá trên bộ dữ liệu đánh giá.



Hình 3.5: Tạo tập dữ liệu đánh giá đơn giản

Mã lệnh 3.7: Đánh giá mô hình bằng phương pháp simple hold-out validation

```

1 import warnings
2 warnings.filterwarnings('ignore')
3
4 from keras import models
5 from keras import layers
6 from keras.datasets import mnist
7 (train_images, train_labels), (test_images, test_labels) =
8     mnist.load_data()
9
10 #normalize the data samples
11 train_images = train_images.reshape((60000, 28*28))
12 train_images = train_images.astype('float32')/255
13 test_images = test_images.reshape((10000, 28*28))
14 test_images = test_images.astype('float32')/255
15
16 #encode the targets
17 from keras.utils import to_categorical
18 train_labels = to_categorical(train_labels)
19 test_labels = to_categorical(test_labels)
20
21 #create samples for simple hold-out validation
22 val_train_images = train_images[:10000]
23 val_train_labels = train_labels[:10000]
24 partial_train_images = train_images[10000:]
25 partial_train_labels = train_labels[10000:]
26
27 #create a network model
28 model = models.Sequential()
29 model.add(layers.Dense(256, activation='relu', input_shape
30     =(28*28,)))
31 model.add(layers.Dense(10, activation='softmax'))
32 model.compile(optimizer='rmsprop', loss='
33     categorical_crossentropy', metrics=['accuracy'])

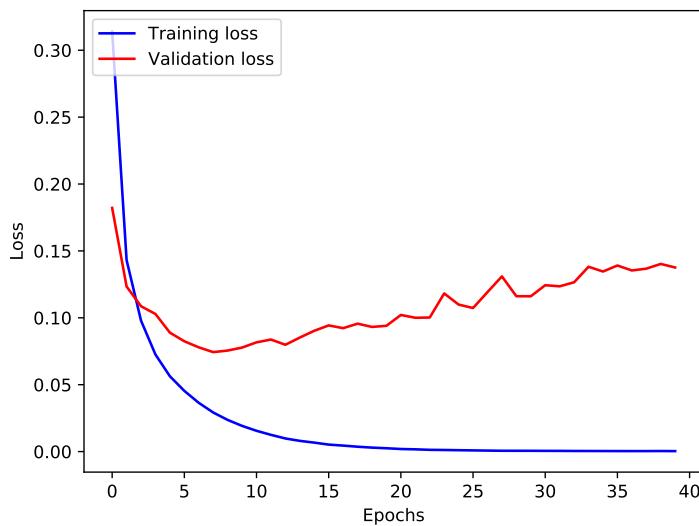
```

```

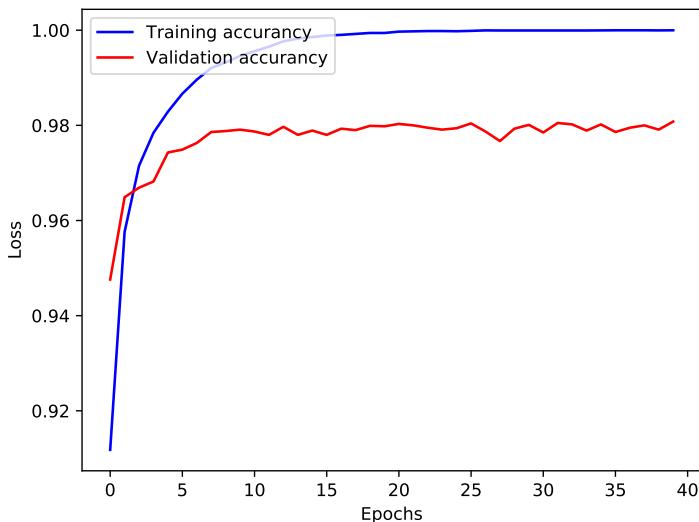
32 #train the network
33 num_epochs = 40
34 history = model.fit(partial_train_images,
35                      partial_train_labels, epochs = num_epochs, batch_size
36                      =128, validation_data=(val_train_images,
37                      val_train_labels))
38
39 #evaluate the trained netwotk
40 test_loss, test_acc = model.evaluate(test_images,
41                                     test_labels)
42 print('test_acc:', test_acc)
43
44 #plot training loss and validation loss
45 import matplotlib.pyplot as plt
46 plt.figure(1)
47 plt.plot(history.history['loss'],'b',label = 'Training
48           loss')
49 plt.plot(history.history['val_loss'],'r',label = '
50           Validation loss')
51 plt.ylabel('Loss')
52 plt.xlabel('Epochs')
53 plt.legend(loc = 'upper left')
54
55 #plot training loss and validation loss
56 plt.figure(2)
57 plt.plot(history.history['acc'],'b',label = 'Training
58           accuracy')
59 plt.plot(history.history['val_acc'],'r',label = '
60           Validation accuracy')
61 plt.ylabel('Loss')
62 plt.xlabel('Epochs')
63 plt.legend(loc = 'upper left')
64 plt.show()

```

Mã lệnh ?? cho tỷ lệ nhận dạng bộ dữ liệu thử nghiệm là 97.98%. Mặc dù số lượng vòng lặp đã tăng lên đến 40, nhưng tỷ lệ nhận dạng bộ dữ liệu thử nghiệm chỉ tăng từ 97.97% lên 97.98%. Rõ ràng dữ độ nhận dạng chính xác tăng không đáng kể. Hình 3.6 chỉ ra validation-loss với 40 epochs. Dễ thấy rằng tại vòng lặp thứ 7, mạng có sự mất mát bé nhất, nghĩa là chỉ cần huấn luyện đến 7 vòng lặp là đủ mà không cần huấn luyện nhiều hơn 7 vòng lặp. Cụ thể, nếu huấn luyện đến vòng lặp thứ 7, tỷ lệ nhận dạng bộ dữ liệu thử nghiệm là 97.33%.



Hình 3.6: Validation-loss

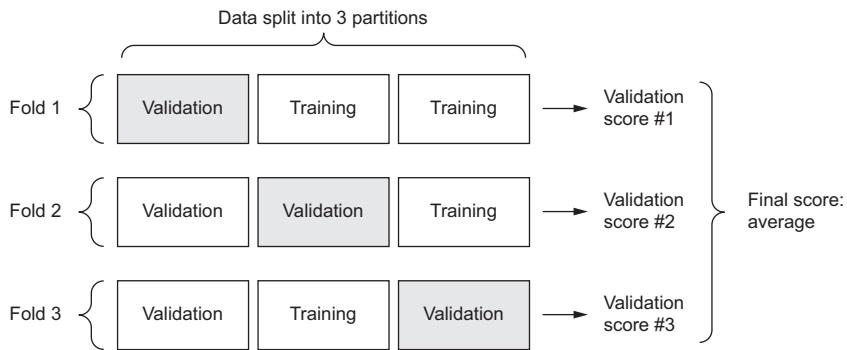


Hình 3.7: Validation-loss

K-fold validation

Đối với những bài toán với dữ liệu nhỏ, phương pháp đánh giá mô hình bằng kiểm tra chéo (cross validation) rất hiệu quả. Ý tưởng chính là chia tập dữ liệu huấn luyện thành k phần (thường k = 4 hoặc k = 5). Tiếp

theo, huấn luyện mạng trên k-1 phần dữ liệu và đánh giá mạng trên phần dữ liệu còn lại mô tả ở như hình 3.8. Điểm (score) đánh giá mô hình là điểm trung bình của k phần đánh giá.



Hình 3.8: 3-fold cross-validation

Xét ví dụ bài toán nhận dạng chữ số viết tay trong cơ sở dữ liệu MNIST. Giả sử $k = 4$, mã lệnh được thực hiện như sau:

Mã lệnh 3.8: Đánh giá mô hình bằng phương pháp K-fold validation

```

1 import warnings
2 warnings.filterwarnings('ignore')
3
4 from keras import models
5 from keras import layers
6 from keras.datasets import mnist
7 (train_images, train_labels), (test_images, test_labels) =
8     mnist.load_data()
9
10 #normalize the data samples
11 train_images = train_images.reshape((60000, 28*28))
12 train_images = train_images.astype('float32')/255
13 test_images = test_images.reshape((10000, 28*28))
14 test_images = test_images.astype('float32')/255
15
16 #encode the targets
17 from keras.utils import to_categorical
18 train_labels = to_categorical(train_labels)
19 test_labels = to_categorical(test_labels)
20
21 #validate a network model using K-fold validation
22 import numpy as np
23 k = 4
  
```

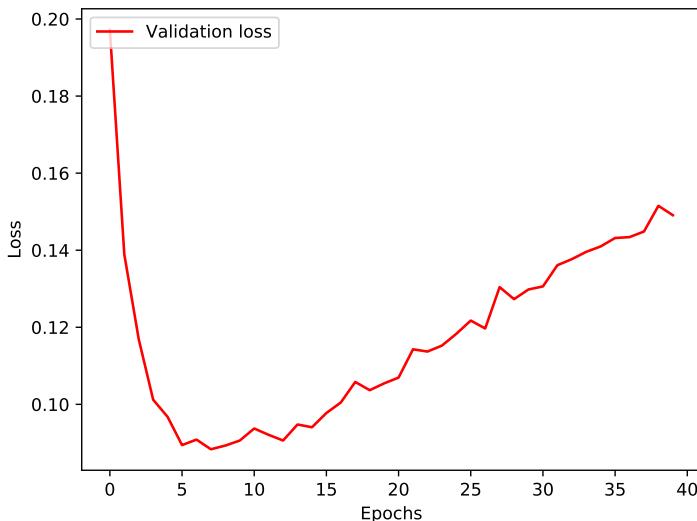
```
23 num_epochs = 40
24 num_val_samples = len(train_images) // k
25 all_loss_histories = []
26 for i in range(k):
27     print('processing fold #', i)
28
29     #create validation samples
30     val_train_images = train_images[i * num_val_samples:
31                                     (i + 1) * num_val_samples]
32     val_train_labels = train_labels[i * num_val_samples:
33                                     (i + 1) * num_val_samples]
34     #create traing samples
35     partial_train_images = np.concatenate([train_images[:i *
36                                         num_val_samples], train_images[(i + 1) *
37                                         num_val_samples:]], axis=0)
38     partial_train_labels = np.concatenate([train_labels[:i *
39                                         num_val_samples], train_labels[(i + 1) *
40                                         num_val_samples:]], axis=0)
41
42     #create a network model
43     model = models.Sequential()
44     model.add(layers.Dense(256, activation='relu',
45                           input_shape=(28 * 28,)))
46     model.add(layers.Dense(10, activation='softmax'))
47     model.compile(optimizer='rmsprop', loss=
48                     'categorical_crossentropy', metrics=['accuracy'])
49
50     #train the network
51     history = model.fit(partial_train_images,
52                          partial_train_labels, epochs = num_epochs,
53                          batch_size=128, verbose=0, validation_data=(
54                             val_train_images, val_train_labels))
55
56     #save the validation loss
57     all_loss_histories.append(history.history['val_loss'])
58
59 #average the validation loss for epoches
60 avg_loss_history = np.mean(all_loss_histories, axis = 0)
61
62 #evaluate the trained netwotk
63 test_loss, test_acc = model.evaluate(test_images,
64                                       test_labels)
65 print('test_acc:', test_acc)
66
67 #plot validation loss
68 import matplotlib.pyplot as plt
69 plt.figure(1)
70 plt.plot(avg_loss_history,'r',label = 'Validation loss')
71 plt.ylabel('Loss')
```

```

60 | plt.xlabel('Epochs')
61 | plt.legend(loc = 'upper left')
62 | plt.show()

```

Hình 3.6 chỉ ra validation-loss với 40 epochs. Với 40 vòng lặp, tỷ lệ nhận dạng bộ dữ liệu thử nghiệm là 97.99%. Tuy nhiên, dễ thấy rằng tại vòng lặp thứ 5 hoặc thứ 7, mạng có sự mất mát bé nhất, nghĩa là chỉ cần huấn luyện đến 7 vòng lặp là đủ mà không cần huấn luyện nhiều hơn 7 vòng lặp.



Hình 3.9: 4-fold cross-validation

3.5 Overfitting và underfitting

Trong ví dụ nhận dạng các chữ số viết tay, bằng việc kiểm tra mô hình, dễ thấy rằng chỉ sau một số vòng lặp (epoch) mô hình sẽ đạt được tối thiểu hàm mất mát (loss function) và sau đó tăng dần. Điều này có nghĩa rằng mô hình đã overfit với dữ liệu huấn luyện. Overfit luôn xảy ra trong mọi bài toán học máy. Học cách giải quyết với overfitting là điều cần thiết để làm chủ lĩnh vực học máy.

Vấn đề cơ bản trong học máy là cần sự cân bằng giữa tối ưu và tổng quát hóa mô hình. Tối ưu hóa mô hình đề cập tới quá trình điều chỉnh một mô hình để nhận được sự khả năng thực hiện tốt nhất có thể trên

dữ liệu huấn luyện, ngược lại, tổng quát hóa mô hình đề cập tới vấn đề làm thế nào để một mô hình đã được huấn luyện thực hiện tốt nhất trên dữ liệu mà trước đó chưa được huấn luyện (dữ liệu dự đoán). Mục đích của học máy là tìm được một mô hình mà tổng quát hóa tốt dựa trên quá trình học dữ liệu huấn luyện.

Khi bắt đầu quá trình huấn luyện mô hình, tối ưu hóa và tổng quát hóa là liên quan với nhau: sự mất mát dữ liệu huấn luyện (training data) càng thấp, sự mất mát trên dữ liệu kiểm tra (validation data) càng thấp và ta nói rằng đây là quá trình underfit. Ví dụ trong hình 3.6, underfit xảy ra khi số vòng lặp bé hơn 7. Sau một số vòng lặp mà mô hình thực hiện trên dữ liệu huấn luyện, sự tổng quát hóa mô hình không còn tiếp diễn và mô hình bắt đầu overfit. Tức là mô hình bắt đầu học các mẫu dữ liệu "đã quen" trong tập dữ liệu huấn luyện mà không "hiểu" các dữ liệu mới".

Để tránh một mô hình học nhưng hiểu nhầm (misleading) các mẫu dữ liệu trong tập huấn luyện, cách tốt nhất là cần phải có nhiều mẫu dữ liệu huấn luyện. Một mô hình được huấn luyện trên nhiều mẫu dữ liệu huấn luyện hơn sẽ tổng quát hóa tốt hơn. Nhưng khi không có nhiều mẫu dữ liệu huấn luyện, giải pháp tốt nhất là điều chỉnh lượng thông tin mà mô hình được phép lưu trữ hoặc thêm các ràng buộc về thông tin được phép lưu trữ. Nếu mô hình chỉ có thể ghi nhớ một số lượng nhỏ các mẫu tin, quá trình tối ưu hóa buộc nó phải tập trung vào các mẫu nổi bật nhất mà có cơ hội tốt hơn để tổng quát hóa. Giải quyết vấn đề overfitting theo cách này được gọi là regularization. Phần này giới thiệu một số kỹ thuật thường được áp dụng trong thực tế cải thiện mô hình phân lớp.

3.5.1 Giảm kích thước mạng

Cách đơn giản nhất để ngăn chặn overfitting là giảm kích thước của mô hình, tức là giảm số các tham số có thể học được trong mô hình hay số tầng và số neuron trên mỗi tầng. Trong mạng neuron, số tham số có thể học được trong một mô hình thường được gọi là dung lượng (capacity) của mô hình. Về cảm nhận, một mô hình với nhiều tham số hơn sẽ có nhiều dung lượng nhớ hơn và vì vậy có thể dễ dàng học một ánh xạ hoàn hảo giữa các mẫu huấn luyện và đầu ra của nó, tức là một ánh xạ mà không có khả năng tổng quát quá tốt. Ví dụ, một mô hình với 500,000 tham số nhị phân có thể dễ dàng học phân lớp mọi chữ số trong tập

huấn luyện MNIST, tức là chỉ cần 10 tham số nhị phân để phân lớp một chữ số trong 50,000 chữ số. Nhưng mô hình này sẽ không có khả năng phân lớp các chữ số mới. Chú ý rằng các mô hình học sâu luôn có xu hướng tốt để fitting với dữ liệu huấn luyện nhưng luôn thách thức với vấn đề tổng quát hóa.

Mặt khác, nếu một mạng có bộ nhớ hạn chế, nó sẽ không dễ dàng học ánh xạ giữa các mẫu huấn luyện và đầu ra của nó. Vì vậy, khi thiết kế mạng cần có một thỏa hiệp giữa việc chọn nhiều dung lượng và ít dung lượng. Tuy nhiên, trong thực tế chúng ta không có các tiêu chí để xác định đúng số lượng tầng mạng cũng như số lượng neuron cho mỗi tầng. Do đó, chúng ta cần phải đánh giá các kiến trúc mạng dựa trên thập kiểm tra (validation set) để tìm ra kích thước chính xác của mô hình cho mỗi bài toán. Quy trình chung để tìm kích thước mô hình thích hợp là bắt đầu với một số ít các tầng với số neuron ít trên mỗi tầng và sau đó số tầng cũng như tăng kích thước của các tầng cho đến khi đạt hàm mất mát bé nhất có thể.

Ví dụ xét bài toán phân lớp các chữ số trong cơ sở dữ liệu MNIST. Mã lệnh để đánh giá các mô hình với số neuron của tầng một lần lượt được chọn là 16, 64, 256, 512, 1024 và 2048 như sau.

Mã lệnh 3.9: Ảnh hưởng của hàm mất mát dựa trên kích thước mạng

```
import numpy as np
import time
import matplotlib.pyplot as plt

layer_sizes = np.array([16,64,256,512,1024,2048])
plt_styles = [ '-k^', '-kv', '-r<', '-b>', '-g+', '-mx' ]
num_epochs = 40
for i in range(len(layer_sizes)):
    layer_size = layer_sizes[i]
    print('Running the model with the first layer size',
          layer_size)
    start_time = time.time()
    model = models.Sequential()
    model.add(layers.Dense(layer_size, activation='relu',
                           input_shape=(28*28,)))
    model.add(layers.Dense(10, activation='softmax'))
    model.compile(optimizer='rmsprop', loss='
        categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(partial_train_images,
                        partial_train_labels, epochs = num_epochs,
                        batch_size=128, verbose = 0, validation_data=(val_train_images, val_train_labels))
```

```

print("Elapsed time is %s seconds" % (time.time() -
    start_time))

#evaluate the trained network
test_loss, test_acc = model.evaluate(test_images,
    test_labels)
print('test_acc :', test_acc)
plt.plot(history.history['val_loss'], plt_styles[i],
    label = 'The first layer size: ' + np.str(
    layer_size))

plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(loc = 'upper right')
plt.show()

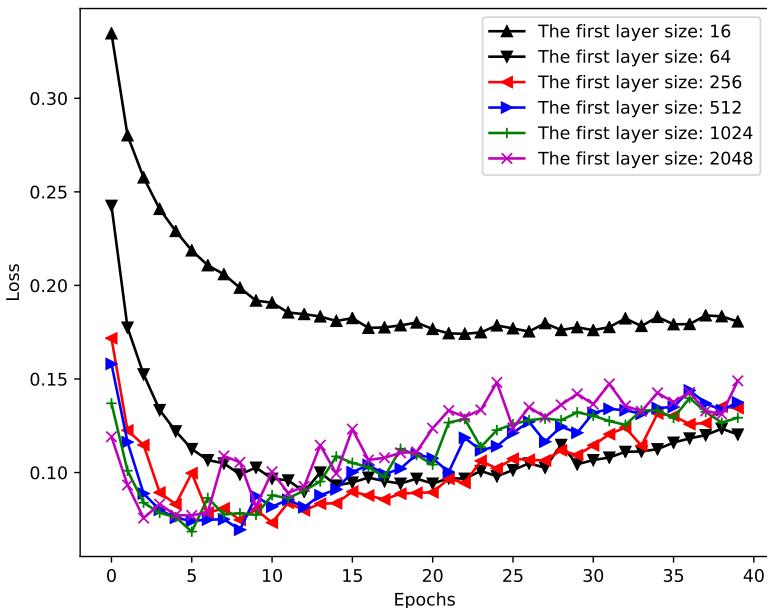
```

Hình 3.10 chỉ ra hàm loss với 40 vòng lặp. Để thấy rằng kích thước Sau ? vòng lặp, hàm loss của các mô hình ????. Tỷ lệ nhận nhận dạng bộ dữ liệu kiểm tra (test) cho các mô hình với tầng một có số neuron 16, 64, 256, 512, 1024 và 2048 tương ứng là 95.39%, 97.48%, 98.10%, 98.19%, 98.32% và 98.01%. Thời gian để huấn luyện các mô hình với tầng một có số neuron 16, 64, 256, 512, 1024 và 2048 tương ứng là 69.54, 98.06, 214.61, 470.63, 942.06 và 1983.41 giây. Để thấy rằng với kích thước tầng 1 là 256 neuron, mạng cũng có thể đạt được tỷ lệ nhận dạng xấp xỉ với mạng có kích thước tầng 1 lớn hơn 216 trong một thời gian huấn luyện bé hơn nhiều.

3.5.2 Adding weight regularization

Một phương pháp phổ biến để giảm thiểu overfitting là thiết lập các ràng buộc trọng của mạng để các trọng số chỉ lấy các giá trị nhỏ. Phương pháp này được gọi là *weight regularization* và được thực hiện bằng cách cộng thêm hàm mất mát của mạng một giá trị được kết hợp với các trọng số có giá trị lớn. Giá trị cộng thêm gồm 2 dạng:

- “L1 regularization”: Giá trị được cộng tỷ lệ với giá trị tuyệt đối của các hệ số trọng số.
- “L2 regularization”: Giá trị được cộng tỷ lệ với bình phương của giá trị của các hệ số trọng số.



Hình 3.10: Ảnh hưởng của hàm mất mát dựa trên kích thước mạng cho cơ sở dữ liệu MNIST

Trong Keras, “L1 regularization” hoặc “L2 regularization” hoặc đồng thời “L1 regularization” và “L2 regularization” được bổ sung như sau cho một tầng mạng như sau:

```
from keras import regularizers
kernel_regularizer = regularizers.l1(0.001)
kernel_regularizer = regularizers.l2(0.001)
kernel_regularizer = regularizers.l1_l2(l1=0.001, l2=0.001)
```

Xét ví dụ bài toán nhận dạng các chữ số trong cơ sở dữ liệu MNIST, mã lệnh để để đánh giá 2 mô hình mạng gồm mô hình không sử dụng “L2 regularization” và mô hình sử dụng “L2 regularization” như sau.

Mã lệnh 3.10: Ảnh hưởng của hàm mất mát dựa trên “L2 regularization”

```
from keras import regularizers
num_epochs = 40
#create a network model without adding L2 weight
#regularization
model = models.Sequential()
```

```

model.add(layers.Dense(256, activation='relu', input_shape
                      =(28*28,)))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop', loss='
                      categorical_crossentropy', metrics=['accuracy'])
history1 = model.fit(partial_train_images,
                      partial_train_labels, epochs = num_epochs, batch_size
                      =128, verbose = 0,validation_data=(val_train_images,
                      val_train_labels))
#evaluate the trained network
test_loss, test_acc = model.evaluate(test_images,
                                      test_labels)
print('test_acc:', test_acc)

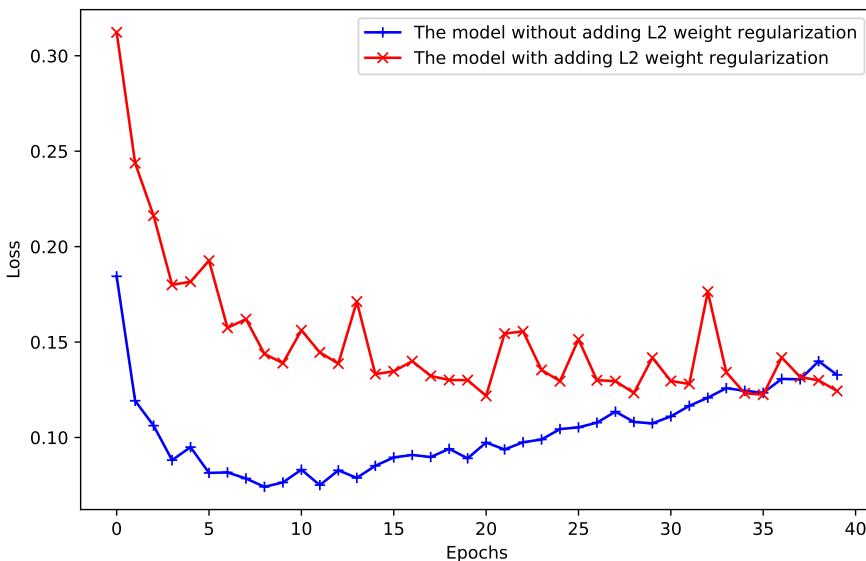
#create a network model with adding L2 weight
#regularization
model = models.Sequential()
model.add(layers.Dense(256, kernel_regularizer=
                      regularizers.l2(0.001), activation='relu',input_shape
                      =(28*28,)))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop', loss='
                      categorical_crossentropy', metrics=['accuracy'])
history2 = model.fit(partial_train_images,
                      partial_train_labels, epochs = num_epochs, batch_size
                      =128, verbose = 0,validation_data=(val_train_images,
                      val_train_labels))
#evaluate the trained network
test_loss, test_acc = model.evaluate(test_images,
                                      test_labels)
print('test_acc:', test_acc)

#plot validation loss
import matplotlib.pyplot as plt
plt.plot(history1.history['val_loss'],'-b+',label = 'The
model without adding L2 weight regularization')
plt.plot(history2.history['val_loss'],'-rx',label = 'The
model with adding L2 weight regularization')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(loc = 'upper right')
plt.show()

```

Tỷ lệ nhận dạng chính xác trên tập dữ liệu thử nghiệm của mô hình không sử dụng “L2 regularization” và mô hình sử dụng “L2 regularization” tương ứng là 97.89% và 97.77%. Rõ ràng hai mô hình này cho tỷ lệ nhận dạng chính xác xấp xỉ nhau. Hình 3.11 chỉ ra hàm mất mát của 2 mô hình. Dễ thấy rằng mô hình không sử

dụng “L2 regularization” bị ”overfitting” chỉ sau một số ít vòng lặp huấn luyện nhưng mô hình sử dụng “L2 regularization” đã tránh được ”overfitting”. Chú ý rằng hàm mất mát của mô hình sử dụng “L2 regularization” lớn hơn hàm mất mát của mô hình không sử dụng “L2 regularization” không có nghĩa là mô hình sử dụng “L2 regularization” có tỷ lệ nhận dạng chính xác thấp hơn mô hình không sử dụng “L2 regularization” mà là vì tất cả các giá trị của ma trận trọng số của tầng 1 được cộng thêm $0.001 * \langle\text{giá trị trọng số}\rangle$ vào tổng hàm mất mát của mạng.



Hình 3.11: Ảnh hưởng của hàm mất mát dựa trên L2 cho cơ sở dữ liệu MNIST

3.5.3 Adding dropout

Dropout là một trong những kỹ thuật hiệu quả nhất và thường xuyên được sử dụng cho mạng neuron để giảm overfitting. Dropout, được áp dụng cho một tầng mạng trong khi huấn luyện, là gán ngẫu nhiên một số phần tử của các đầu ra của tầng mạng bằng 0. Giả sử có một tầng mạng mà trong khi huấn luyện có đầu ra là vector $[0.2, 0.5, 1.3, 0.8, 1.1]$ ứng với một mẫu đầu vào nào đó. Khi áp dụng kỹ thuật dropout, vector này sẽ có một số phần tử ngẫu nhiên được gán bằng 0, chẳng hạn $[0, 0.5, 1.3, 0, 1.1]$. Tỷ lệ số phần tử ngẫu nhiên được gán bằng 0 thường được thiết lập là từ 0.2 đến 0.5. Ở thời gian kiểm tra mạng (test), không có

phần tử nào của đầu ra của tầng mạng bị dropout mà thay vào đó là các giá trị của đầu ra của tầng mạng được nhân với một giá trị tương đương với tỷ lệ dropout.

Một ví dụ về kỹ thuật dropout được minh họa trong hình 3.12 trong đó tỷ lệ dropout của đầu ra của tầng mạng được thiết lập là 50%, tức là có 8 phần tử ngẫu nhiên được gán bằng 0.

0.3	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.2	1.9	0.3	1.2
0.7	0.5	1.0	0.0

0.0	0.2	1.5	0.0
0.6	0.1	0.0	0.3
0.0	1.9	0.3	0.0
0.7	0.0	0.0	0.0

Hình 3.12: Dropout cho đầu ra của một tầng mạng với tỷ lệ 50%

Trong Keras, chúng ta có thể dropout đầu ra của một tầng mạng bằng cách thêm một tầng Dropout sau tầng đó. Ví dụ dropout một tầng mạng với tỷ lệ 50% là:

```
model.add(layers.Dropout(0.5))
```

Xét bài toán nhận dạng các chữ số viết tay trong cơ sở dữ liệu MNIST. Mã lệnh so sánh hàm mất mát của mô hình mạng không sử dụng tầng Dropout với mô hình sử dụng tầng Dropout sau tầng 1 như sau:

Mã lệnh 3.11: Ảnh hưởng của hàm mất mát dựa trên adding dropout

```
num_epochs = 40
#create a network model without adding dropout
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_shape
                      =(28*28,)))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop', loss='
                     categorical_crossentropy', metrics=['accuracy'])
history1 = model.fit(partial_train_images,
                      partial_train_labels, epochs = num_epochs, batch_size
                      =128, verbose = 0, validation_data=(val_train_images,
                      val_train_labels))
#evaluate the trained network
```

```

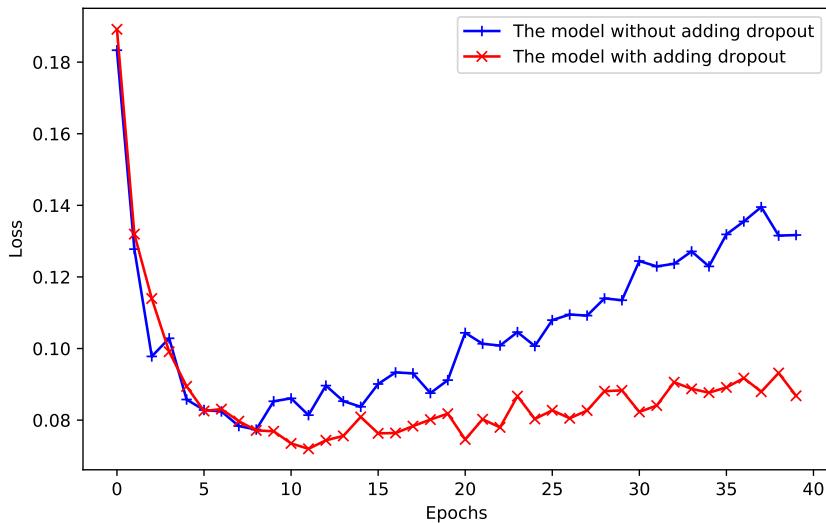
test_loss, test_acc = model.evaluate(test_images,
    test_labels)
print('test_acc:', test_acc)

#create a network model with adding dropout
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_shape
    =(28*28,)))
model.add(layers.Dropout(0.35))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop', loss=
    'categorical_crossentropy', metrics=['accuracy'])
history2 = model.fit(partial_train_images,
    partial_train_labels, epochs = num_epochs, batch_size
    =128, verbose = 0, validation_data=(val_train_images,
    val_train_labels))
#evaluate the trained netwotk
test_loss, test_acc = model.evaluate(test_images,
    test_labels)
print('test_acc:', test_acc)

#plot validation loss
import matplotlib.pyplot as plt
plt.plot(history1.history['val_loss'], '-b+', label = 'The
    model without adding dropout')
plt.plot(history2.history['val_loss'], '-rx', label = 'The
    model with adding dropout')
plt.ylabel('Loss')
plt.xlabel('Epochs')
plt.legend(loc = 'upper right')
plt.show()

```

Hình 3.13 chỉ ra kết quả so sánh của hàm mất mát trong 2 mô hình mạng. Để thấy rằng mô hình không sử dụng tầng Dropout đạt được cực tiểu của hàm mất mát sớm hơn so với mô hình sử dụng tầng Dropout. Tuy nhiên mô hình không sử dụng tầng Dropout sẽ overfitting rất nhanh. Trong khi mô hình sử dụng tầng Dropout đạt được cực tiểu của của hàm mất mát bé hơn và overfit chậm hơn và nhỏ hơn so với mô hình không sử dụng Dropout . Tỷ lệ nhận dạng trên tập dữ liệu thử nghiệm của mô hình không sử dụng tầng Dropout là 98.07%, và của mô hình sử dụng tầng Dropout là 98.08%. Như vậy có thể thấy rằng mô hình sử dụng tầng Dropout cho tỷ lệ nhận dạng chính xác xấp xỉ với mô hình không sử dụng tầng Dropout, nhưng mô hình sử dụng tầng Dropout trách được overfit.



Hình 3.13: Ảnh hưởng của hàm mất mát dựa trên adding dropout cho cơ sở dữ liệu MNIST

Bài tập

Bài tập 3.1. Sử dụng phương pháp đánh giá mô hình mạng cho bài toán nhận dạng các chữ số viết tay:

- Nếu dùng 64 neuron cho tầng 1, mạng nên dừng ở vòng lặp thứ bao nhiêu?
- Nếu dùng 512 neuron cho tầng 1, mạng nên dừng ở vòng lặp thứ bao nhiêu?

Bài tập 3.2. Bài toán MNIST-Fashition:

- Nếu dùng 64 neuron cho tầng 1, mạng nên dừng ở vòng lặp thứ bao nhiêu?
- Nếu dùng 512 neuron cho tầng 1, mạng nên dừng ở vòng lặp thứ bao nhiêu?

Bài tập 3.3. Cho 2 thư mục **sign-train** và **sign-test** chứa các ảnh chữ ký của 100 người, trong đó thư mục **sign-train** chứa ảnh dùng để huấn luyện và thư mục **sign-test** dùng chứa các ảnh nhận dạng. Tên tệp trong 2 thư mục chứa một dấu '-' và các ký tự trước dấu '-' biểu diễn lớp dữ liệu của ảnh, các ký tự sau dấu '-' là số thứ tự chữ ký của mỗi người. Hãy thiết kế một mạng neuron để nhận dạng các chữ ký.

Hướng dẫn:

- Đọc dữ liệu huấn luyện và dữ liệu thử nghiệm vào các mảng số liệu:

```

import warnings
warnings.filterwarnings('ignore')

import numpy as np
import glob
from PIL import Image
from keras import models
from keras import layers

train_dir = 'sign-train/'
num_classes = 100
train_images = []
train_labels = []
for i in range(1,num_classes + 1):
    filenames = train_dir + str(i) + '-*.*.png'
    for filename in glob.glob(filenames):
        train_labels.append(i-1)
        train_image = Image.open(filename)
        train_image = np.array(train_image)
        train_images.append(train_image)

train_images = np.array(train_images)

test_dir = 'sign-test/'
test_images = []
test_labels = []
for i in range(1,num_classes + 1):
    filenames = test_dir + str(i) + '-*.*.png'
    for filename in glob.glob(filenames):
        test_labels.append(i-1)
        test_image = Image.open(filename)
        test_image = np.array(test_image)
        test_images.append(test_image)

test_images = np.array(test_images)

```

- Hoán đổi ngẫu nhiên bộ dữ liệu huấn luyện: vì các ảnh của các lớp được đọc tuần tự, cần hoán đổi ngẫu nhiên.

Sau khi có được dữ liệu huấn luyện, dữ liệu kiểm tra, các bước còn lại giống như bài toán nhận dạng chữ số viết tay.

Chương 4

Mạng nhân chập cho nhận dạng ảnh

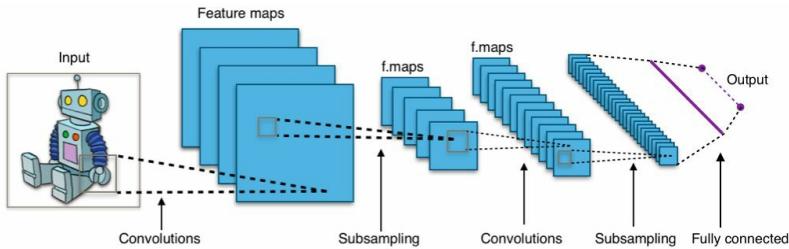
Chương này giới thiệu về mạng neuron nhân chập (convolutional neural networks - CNN), một mạng điển hình cho mô hình học sâu được dùng trong các ứng dụng của lĩnh vực thị giác máy tính (computer vision). Chương này cũng giới thiệu một ứng dụng của mạng nhân chập cho các bài toán phân lớp các hình ảnh và những vấn đề liên quan để tăng hiệu quả của quá trình nhận dạng.

4.1 Toán tử nhân chập

4.2 Toán tử Pooling

4.3 Kiến trúc mạng nhân chập

Một mạng neuron nhân chập là một mạng neuron truyền thăng có nhiều tầng, trong đó tầng nhân chập và tầng pooling được sắp xếp hoán đổi lẫn nhau. Tầng cuối cùng là một tầng kết nối đầy đủ (dense). Kiến trúc mạng neuron nhập chập được mô tả như hình 4.1 [1].



Hình 4.1: Kiến trúc mạng nhân chập

4.4 Ứng dụng mạng nhân chập cho bài toán phân lớp hình ảnh

Một mạng nhận chập cơ bản phân lớp các chữ số viết tay trong cơ sở dữ liệu MNIST được mô tả bằng đoạn mã lệnh sau:

```
from keras import layers
from keras import models
model = models.Sequential()
model.add(layers.Conv2D(32,(3,3),activation='relu',
    input_shape=(28,28,1)))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64,(3,3),activation='relu'))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64,(3,3),activation='relu'))
```

Tầng nhân chập (Conv2D) có tham số vào là một ma trận 3 chiều của ảnh: chiều cao (height), chiều rộng (width) và số kênh (channels). Ví dụ đầu vào của tầng nhân chập thứ nhất là (28, 28, 1) vì mỗi chữ số viết tay trong MNIST có chiều cao 28, chiều rộng 28 và là ảnh đa mức xám (số kênh là 1). Mô hình của mạng đã tạo (dùng lệnh `print(model.summary())`) như hình 4.2. Đầu ra của mỗi tầng Conv2D và MaxPooling2D là một mảng 3 chiều (`height, width, channels`). Chiều cao và chiều rộng giảm khi độ sâu của mạng tăng lên và số kênh chính là tham số đầu tiên của tầng Conv2D (32 hoặc 64).

Bước tiếp theo là bổ sung các tầng Dense vào cuối mạng để chuyển đầu ra của tầng Conv2D thành một số vô hướng cho nhiệm vụ phân lớp. Để thực hiện nhiệm vụ này, đầu tiên chuyển mảng 3 chiều thành một số vô hướng (thêm tầng `Flatten()`) và sau đó bổ sung thêm một số ít các tầng Dense như sau:

```
model.add(layers.Flatten())
```

Layer (type)	Output Shape	Param #
<hr/>		
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
<hr/>		
maxpooling2d_1 (MaxPooling2D)	(None, 13, 13, 32)	0
<hr/>		
conv2d_2 (Conv2D)	(None, 11, 11, 64)	18496
<hr/>		
maxpooling2d_2 (MaxPooling2D)	(None, 5, 5, 64)	0
<hr/>		
conv2d_3 (Conv2D)	(None, 3, 3, 64)	36928
<hr/>		
Total params: 55,744		
Trainable params: 55,744		
Non-trainable params: 0		

Hình 4.2: Mô hình mạng nhân chập

```
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

Chú ý rằng chúng ta đang xét bài toán phân lớp 10 chữ số viết tay, tức là phân 10 lớp do đó tầng cuối cùng phải có đầu ra là 10. Mô hình của mạng cho bài toán phân lớp như hình 4.3. Như vậy, đầu ra của tầng Conv2D cuối cùng là (3,3,64) được chuyển thành các vector kích thước (576,) trước khi đi qua các tầng Dense.

Mã lệnh để huấn luyện mạng và đánh giá độ chính xác của mạng cho dữ liệu thử nghiệm của bài toán phân lớp như sau:

```
from keras.datasets import mnist
from keras.utils import to_categorical
(train_images, train_labels), (test_images, test_labels) =
    mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32')/255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32')/255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop', loss='
    categorical_crossentropy', metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size
    =64)
```

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
maxpooling2d_1 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_2 (Conv2D)	(None, 11, 11, 64)	18496
maxpooling2d_2 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_3 (Conv2D)	(None, 3, 3, 64)	36928
flatten_1 (Flatten)	(None, 576)	0
dense_1 (Dense)	(None, 64)	36928
dense_2 (Dense)	(None, 10)	650

Total params: 93,322
Trainable params: 93,322
Non-trainable params: 0

Hình 4.3: Mô hình mạng nhân chập

```
test_loss, test_acc = model.evaluate(test_images,
test_labels)
```

So với kết quả của mạng mà tất cả các tầng là Dense ở chương trước, tỷ lệ nhận dạng chính xác đã tăng lên 99.08%.

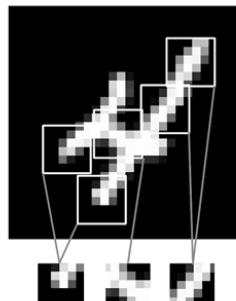
4.4.1 Tầng nhân chập

Sự khác nhau cơ bản giữa tầng Dense (tầng kết nối đầy đủ) và tầng Conv2D (tầng nhân chập) là tầng Dense học các mẫu tổng thể (global patterns) trong khi tầng Conv2D học các mẫu cục bộ (local patterns). Ví dụ các điểm ảnh là các mẫu tổng thể và tập các điểm ảnh trong một cửa sổ 3×3 như mô tả ở hình 4.4 là các mẫu cục bộ. Điều này dẫn đến các đặc tính của mạng nhân chập như sau:

- Có khả năng học các mẫu dịch chuyển: Ví dụ sau khi học một mẫu nào đó ở góc trên bên trái của ảnh, mạng nhân chập có thể nhận ra mẫu đã học có mặt ở bất kỳ vị trí nào khác trong ảnh. Trong đó mạng kết nối đầy đủ phải học lại các mẫu mới nếu mẫu

xuất hiện ở một ví trí mới. Điều này cho phép mạng nhân chập hiệu quả hơn khi nhận dạng ảnh.

- Có khả năng học phân cấp không gian các mẫu (spatial hierarchies of patterns): Tầng 1 của mạng nhân chập sẽ học các mẫu cục bộ nhỏ (ví dụ các cạnh), tầng 2 sẽ học các mẫu lớn hơn được tạo ra từ tầng 1, v.v. như minh họa ở hình 4.5.



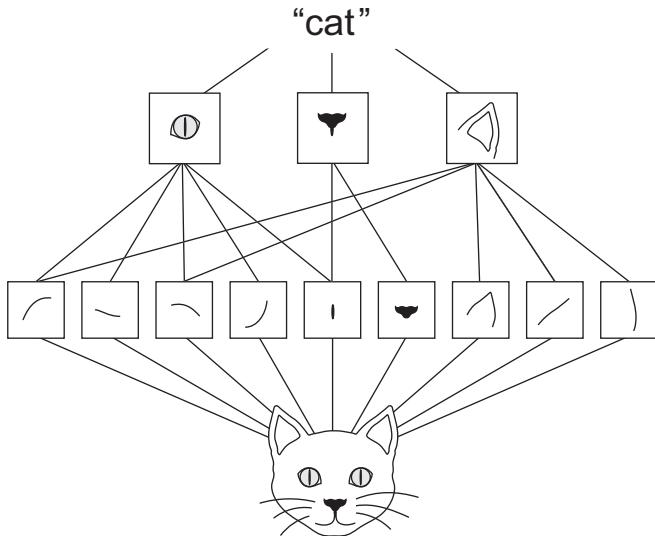
Hình 4.4: Ảnh được chia thành các mẫu cục bộ dựa trên các cạnh

Tầng mạng nhân chập thao tác trên mảng 3 chiều (3D tensors): chiều cao (height), chiều rộng (width) và chiều sâu (depth) (còn gọi là kênh - channels), gọi là các bản đồ đặc trưng (feature maps). Ví dụ với một ảnh màu RGB, chiều sâu là 3 vì ảnh có 3 màu RGB và với ảnh đa mức xám, chiều sâu là 1.

Trong ví dụ nhân dạng chữ số viết tay trong MNIST, đầu vào của tầng Conv2D đầu tiên là một bản đồ đặc trưng kích thước (28, 28, 1) và đầu ra là một bản đồ đặc trưng kích thước (26, 26, 32): tầng Conv2D tính 32 bộ lọc trên đầu vào của mạng. Đầu ra của tầng Conv2D là 32 kênh với mỗi kênh có kích thước 26×26 . Mỗi kênh được gọi là bản đồ đáp ứng (response map) của bộ lọc trên dữ liệu vào, chỉ ra sự đáp ứng của bộ lọc với các mẫu ở các vị trí khác nhau của dữ liệu vào như mô tả ở hình ?.

Tầng nhân chập được định nghĩa với hai tham số:

- Kích thước của bộ lọc được trích từ dữ liệu vào - thông thường là 3×3 hoặc 5×5 .
- Chiều sâu của đầu ra - số các bộ lọc được tính bởi toán tử nhân chập (trong ví dụ trên là 32 và 64).



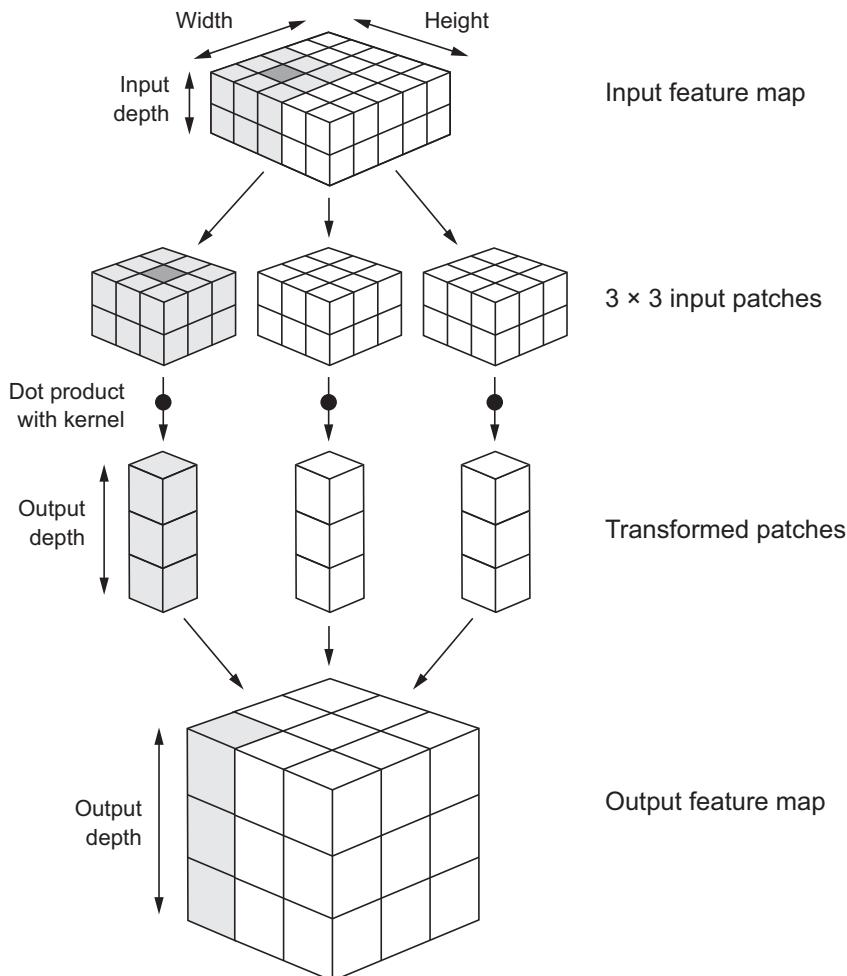
Hình 4.5: Phân cấp không gian các mău

Trong thư viện Keras, những tham số trên là những tham số đầu tiên của tầng Conv2D, cụ thể là `Conv2D(output_depth, (window_height, window_width))`.

Về cơ bản, tầng nhân chập làm việc như sau: dịch chuyển các cửa sổ kích thước 3×3 hoặc 5×5 trên dữ liệu vào, dừng lại vị trí có thể, và trích ra các khối có kích thước (`window_height, window_width, input_depth`) của các đặc trưng. Mỗi khối sau đó được chuyển thành một vector (`output_depth,`) bằng toán tử nhân vô hướng. Tất cả các khối sau đó được ghép lại thành một bản đồ đặc trưng đầu ra với kích thước (`height, width, output_depth`). Mỗi vị trí không gian trong bản đồ đặc trưng đầu ra với vị trí tương ứng trong bản đồ đặc trưng đầu vào. Toàn bộ quá trình được mô tả như hình 4.6. Chú ý rằng chiều rộng và chiều cao của bản đồ đặc trưng đầu ra có thể khác với chiều rộng và chiều cao của bản đồ đặc trưng đầu vào vì hiệu ứng biên và ??

Hiệu ứng biên

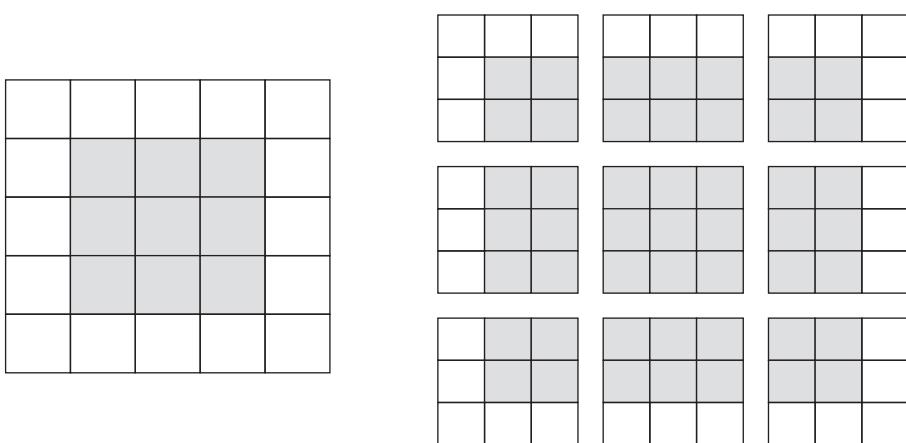
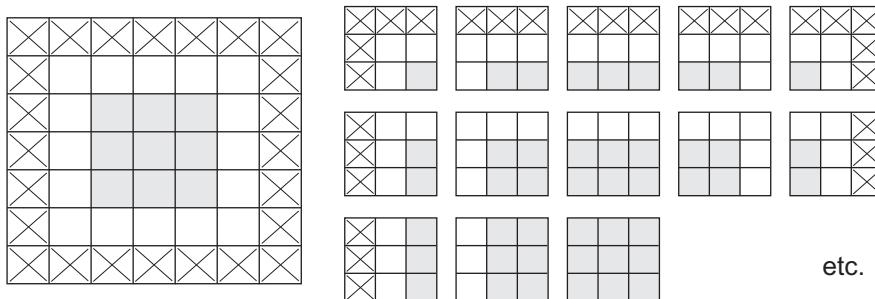
Xét một bản đồ đặc trưng kích thước 5×5 , tức là có 25 ô (25 tiles). Nếu chọn cửa sổ nhân chập kích thước 3×3 , ta sẽ có khối như hình 4.7. Vì vậy đầu ra của bản đồ đặc trưng có kích thước là 3×3 . Điều này giải



Hình 4.6: Nguyên lý làm việc của tầng nhân chập

thích cho ví dụ ở trên: kích thước đầu vào của tầng Conv2D đầu tiên là 28×28 nhưng đầu ra kích thước là 26×26 .

Nếu muốn kích thước của bản đồ đặc trưng đầu ra giống với kích thước bản đồ đặc trưng đầu vào, chúng ta có thể bổ sung thêm (padding) một số dòng và một số cột trên mỗi cạnh của bản đồ đặc trưng đầu vào để trung tâm của cửa sổ nhân chập nằm ở các ô của các cạnh của bản đồ đặc trưng đầu vào. Ví dụ với cửa sổ nhân chập thước 3×3 , chúng ta sẽ thêm 1 cột ở bên trái, 1 cột ở bên phải, 1 dòng ở phía trên và 1 dòng ở phía dưới như hình 4.8. Tuy nhiên với cửa sổ nhân chập thước 5×5 ,

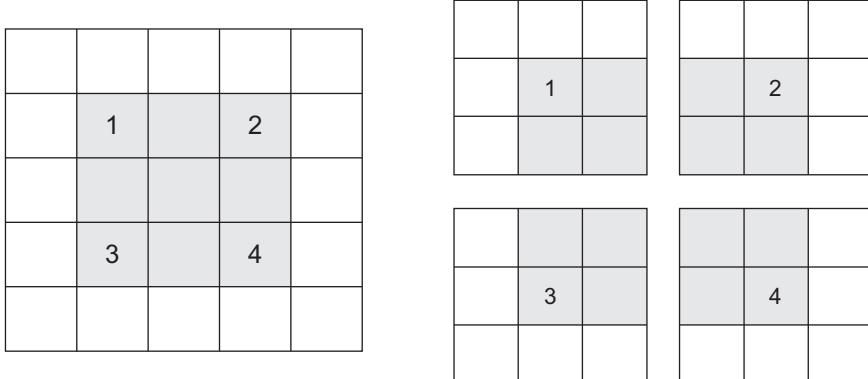
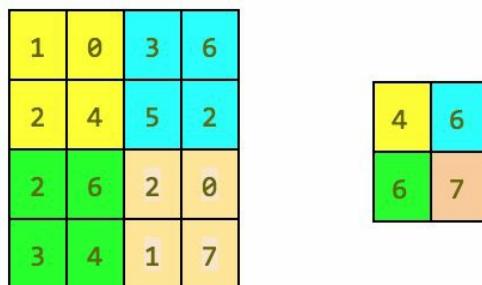
**Hình 4.7:** Hiệu ứng biên**Hình 4.8:** Mở rộng biên

ta sẽ thêm 2 cột ở bên trái, 2 cột ở bên phải, 2 dòng ở phía trên và 2 dòng ở phía dưới.

Chú ý: Trong tầng Conv2D, sự mở rộng cạnh được thiết lập thông qua tham số padding với hai giá trị là "valid" (không mở rộng cạnh) và "same" (có mở rộng cạnh). Giá trị ngầm định của tham số padding là "valid".

UNDERSTANDING CONVOLUTION STRIDES

Một nhân tố khác có thể ảnh hưởng đến kích thước đầu ra là **strides**. Các phép toán nhân chập mô tả ở trên đều giả sử rằng trung tâm của cửa sổ nhân chập luôn được đặt ở mọi ô kề nhau. Khoảng cách giữa hai

**Hình 4.9:** convolution-stride**Hình 4.10:** Ví dụ về max-pooling

cửa sổ nhân chập liên tục là một tham số của nhân chập và được gọi là **stride**, ngầm định là 1. Hình 4.9 chỉ ra một ví dụ minh họa với kích thước cửa sổ nhân chập là 3×3 và **stride = 2** trên một đầu vào kích thước 5×5 .

4.4.2 Tầng max-pooling

Trong ví dụ đã xét ở trên, dễ thấy rằng kích thước của các bản đồ đặc trưng giảm một nửa sau mỗi tầng MaxPooling2D được gọi. Vai trò chính của tầng MaxPooling2D là giảm kích thước bản đồ đặc trưng đầu vào. Tầng MaxPooling2D thường dùng kích thước cửa sổ nhân chập là 2×2 với **stride = 2** để giảm một nửa kích thước đầu vào. Hình 4.10 mô tả một ví dụ với tầng max-pooling.

Bảng 4.1: Kiến trúc mạng 1

Kiểu tầng mạng	Tham số vào	Tham số ra	Số tham số
Conv2D	(3,3,’relu’)	(none, 26, 26, 32)	320
MaxPooling	(2,2)	(none, 13, 13, 32)	0
Dropout	0.35	(none, 13, 13, 32)	0
Flatten	none	(None, 5408)	0
Dense	(64,’relu’)	(none, 64)	346176
Dense	(10,’softmax’)	(none, 10)	650

Bảng 4.2: Kiến trúc mạng 2

Kiểu tầng mạng	Tham số vào	Tham số ra	Số tham số
Conv2D	(3,3,’relu’)	(none, 26, 26, 32)	320
MaxPooling	(2,2)	(none, 13, 13, 32)	0
Dropout	0.35	(none, 13, 13, 32)	0
Conv2D	(3,3,’relu’)	(none, 11, 11, 64)	18496
MaxPooling	(2,2)	(none, 5, 5, 64)	0
Dropout	0.35	(none, 5, 5, 64)	0
Flatten	none	(None, 1600)	0
Dense	(64,’relu’)	(none, 64)	102464
Dense	(10,’softmax’)	(none, 10)	650

4.5 Lập trình mạng nhận dạng cho ứng dụng nhận dạng ảnh

Các mạng nhân chập được dùng cho các thử nghiệm nhận dạng ảnh bao gồm:

Kiến trúc mạng 1: là một mạng 6 tầng với các tham số mạng được mô tả như bảng 4.1.

Kiến trúc mạng 2: là một mạng 9 tầng với các tham số mạng được mô tả như bảng 4.2.

Bảng 4.3: Kiến trúc mạng 3

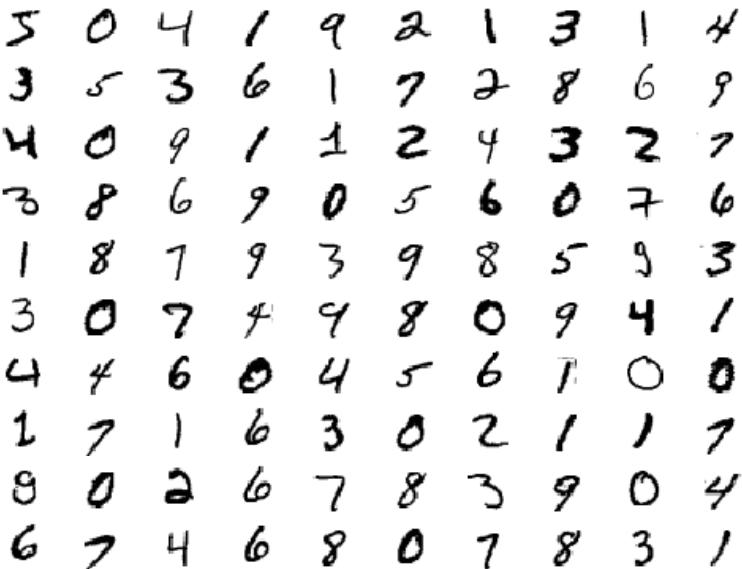
Kiểu tầng mạng	Tham số vào	Tham số ra	Số tham số
Conv2D	(3,3,'relu')	(none, 26, 26, 32)	320
MaxPooling	(2,2)	(none, 13, 13, 32)	0
Dropout	0.35	(none, 13, 13, 32)	0
Conv2D	(3,3,'relu')	(none, 11, 11, 64)	18496
MaxPooling	(2,2)	(none, 5, 5, 64)	0
Dropout	0.35	(none, 5, 5, 64)	0
Conv2D	(3,3,'relu')	(none, 3, 3, 64)	36928
MaxPooling	(2,2)	(none, 1, 1, 64)	0
Dropout	0.35	(none, 1, 1, 64)	0
Flatten	none	(None, 1600)	0
Dense	(64,'relu')	(none, 64)	4160
Dense	(10,'softmax')	(none, 10)	650

Kiến trúc mạng 3: là một mạng 12 tầng với các tham số mạng được mô tả như bảng 4.3.

4.5.1 Nhận dạng các chữ số viết tay

Trong phần này chúng tôi thực hiện các thử nghiệm nhận dạng các chữ số viết tay trong cơ sở dữ liệu MNIST trong thư viện Keras. Cơ sở dữ liệu MNIST bao gồm 60000 ảnh huấn luyện và 10000 ảnh kiểm tra (test). Các ảnh chữ số viết tay trong cơ sở dữ liệu là các ảnh đa mức xám với mỗi ảnh có kích thước 28×28 . Hình 4.11 chỉ ra 100 ảnh chữ số đầu tiên trong cơ sở dữ liệu MNIST.

Để đánh giá mô hình mạng, chúng tôi sử dụng phương pháp đánh giá đơn giản (simple-validation). Cụ thể, chúng tôi lấy 50000 ảnh trong 60000 ảnh huấn luyện làm tập dữ liệu huấn luyện (training set) và 10000 ảnh còn lại làm tập dữ liệu đánh giá (validation set), và giữ nguyên bộ dữ liệu kiểm tra (test). Chúng tôi sử dụng 10 vòng lặp (epoch) để huấn luyện và kích thước mỗi lô (batch) dữ liệu là 128.



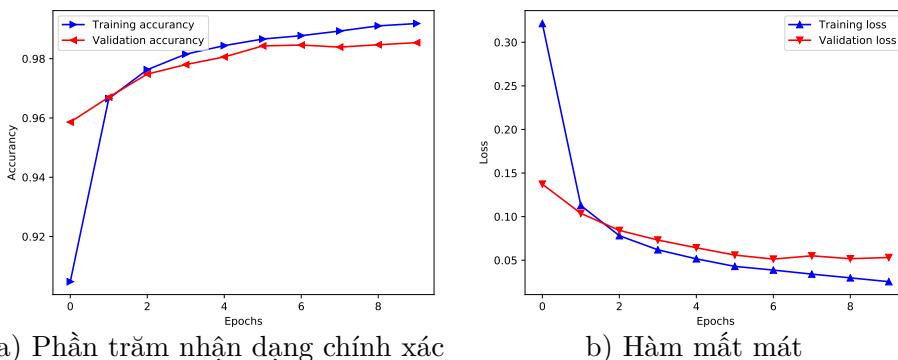
Hình 4.11: 100 ảnh đầu tiên trong cơ sở dữ liệu MNIST

Thử nghiệm với kiến trúc mạng 1

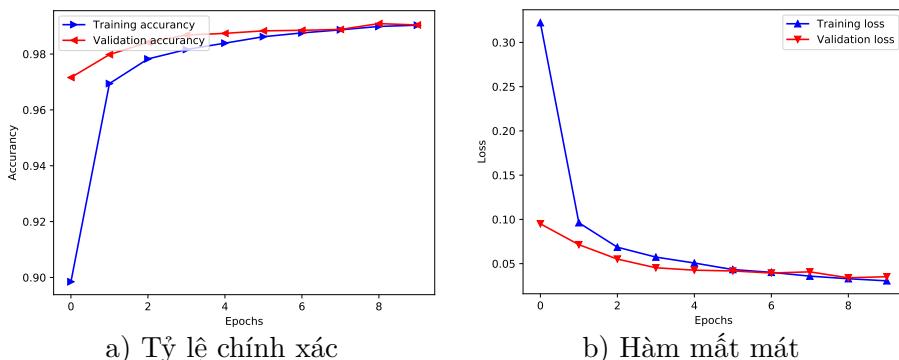
Hình 4.12(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 10 vòng lặp, phần trăm nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá khoảng 99% và 98% tương ứng. Hình 4.12(b) chỉ ra rằng hàm mất mát của dữ liệu kiểm tra đạt cực tiểu tại vòng lặp thứ 6, sau đó hàm có xu hướng không giảm, tức là chúng ta chỉ cần dừng quá trình huấn luyện mạng sau 6 vòng lặp là đủ. Sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 98.56%. So với mạng MLP, rõ ràng bằng việc sử dụng một tầng nhân chập, tỷ lệ nhận dạng đã tăng lên từ 97.97% (xem phần mạng MLP) lên 98.56%.

Thử nghiệm với kiến trúc mạng 2

Hình 4.13 chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 10 vòng lặp, phần trăm nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá đều đạt khoảng 99%. Hình 4.13(b) chỉ ra rằng hàm mất mát của dữ liệu kiểm tra đạt cực tiểu tại vòng lặp thứ 8, sau đó hàm mất mát

**Hình 4.12:** Thử nghiệm với kiến trúc mạng 1

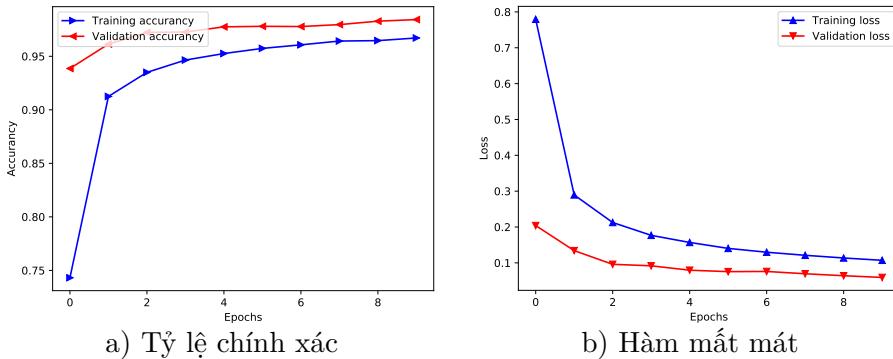
có xu hướng không giảm, tức là chúng ta chỉ cần dừng quá trình huấn luyện sau 8 vòng lặp là đủ. So với hình 4.12(b), dễ thấy rằng giá trị hàm mất mát của kiến trúc mạng này bé hơn so với kiến trúc mạng 1, tức là mạng sẽ cho kết quả nhận dạng chính xác hơn mạng 1. Cụ thể, sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 99,19%.

**Hình 4.13:** Thử nghiệm với kiến trúc mạng 2

Thử nghiệm với kiến trúc mạng 3

Hình 4.14(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 10 vòng lặp, phần trăm nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu khi kiểm tra đều đạt khoảng 97%. Hình ??(b) chỉ ra rằng hàm mất mát của dữ

liệu kiểm tra đạt cực tiểu tại vòng lặp thứ 9. Tuy nhiên đồ thị của hàm mất mát đang có xu hướng giảm, nghĩa là chúng ta cần huấn luyện thêm một số vòng lặp nữa để đạt được giá trị bé nhất của hàm mất mát. So với hình 4.13(b), dễ thấy rằng giá trị hàm mất mát của kiến trúc mạng này lớn hơn so với kiến trúc mạng 2, tức là mạng sẽ cho kết quả nhận dạng chính xác kém hơn mạng 2. Cụ thể, sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 98.67%.



a) Tỷ lệ chính xác

b) Hàm mất mát

Hình 4.14: Thử nghiệm với kiến trúc mạng 3

Như vậy với 3 kiến trúc đã chọn, kiến trúc mạng 2 cho tỷ lệ nhận dạng chính xác cao nhất.

4.5.2 Nhận dạng mặt người

Trong phần này chúng tôi thực hiện các thử nghiệm nhận dạng ảnh mặt người. Cở sở dữ liệu ảnh mặt người bao gồm 663 ảnh đa mức xám của 61 người, trung bình mỗi người có 11 ảnh được chụp ở các góc chụp khác nhau. Mỗi ảnh có kích thước là 64×48 . Chúng tôi chia cơ sở dữ liệu thành 2 tập là tập dữ liệu huấn luyện và tập dữ liệu kiểm tra, trong đó tập dữ liệu huấn luyện có 602 ảnh và tập dữ liệu kiểm tra có 61 ảnh. Hình 4.15 chỉ ra 100 ảnh mặt người đầu tiên trong tập dữ liệu huấn luyện.

Để đánh giá mô hình, chúng tôi sử dụng phương pháp đánh giá đơn giản (simple-validation). Cụ thể, chúng tôi lấy 402 ảnh trong 602 ảnh huấn luyện làm tập dữ liệu huấn luyện (training) và 200 ảnh còn lại làm tập dữ liệu đánh giá (validation) và giữ nguyên tập dữ liệu kiểm tra (test). Vì tập mẫu dữ liệu huấn luyện tương đối ít, do đó chúng tôi tăng



Hình 4.15: 100 ảnh trong cơ sở dữ liệu IFD

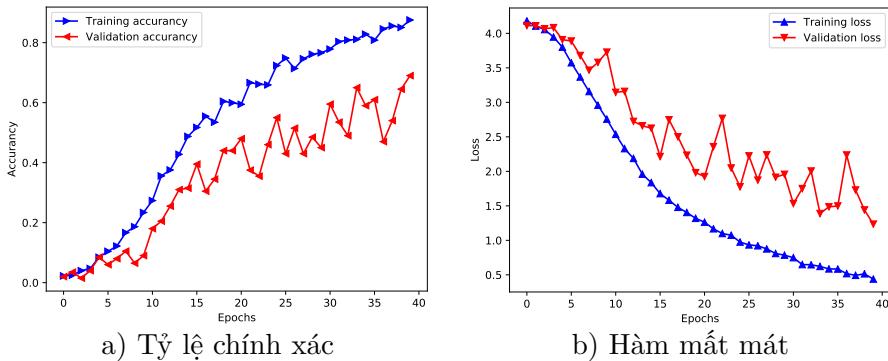
vòng lặp huấn luyện lên 40 vòng lặp (epoch) và chọn kích thước mỗi lô (batch) dữ liệu là 8.

Thử nghiệm với kiến trúc mạng 1

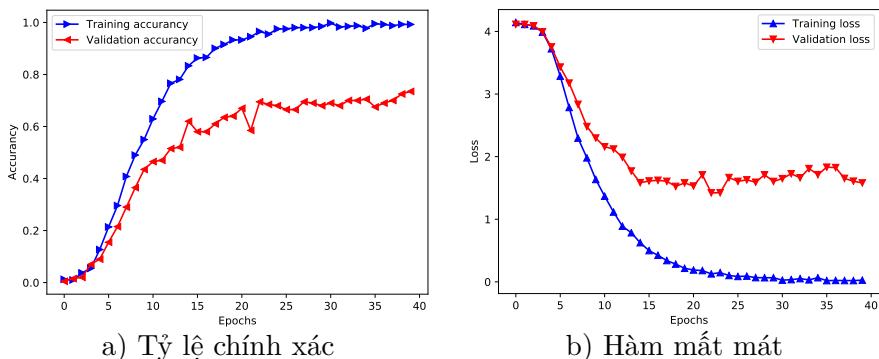
Hình 4.16(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 40 vòng lặp, phần trăm nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá khoảng 83% và 70% tương ứng. Hình 4.16(b) chỉ ra rằng hàm mất mát của tập dữ liệu đánh giá đạt cực tiểu tại vòng lặp thứ 39, tuy nhiên hàm đang có xu hướng giảm do đó cần tăng số vòng lặp để đạt được giá trị hàm mất mát bé hơn. Sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 77.04%. So với bài toán nhận dạng ký tự, kiến trúc mạng đã chọn có tỷ lệ nhận dạng chính xác thấp hơn nhiều.

Thử nghiệm với kiến trúc mạng 2

Hình 4.17(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 40 vòng lặp, phần trăm

**Hình 4.16:** Thử nghiệm với kiến trúc mạng 1

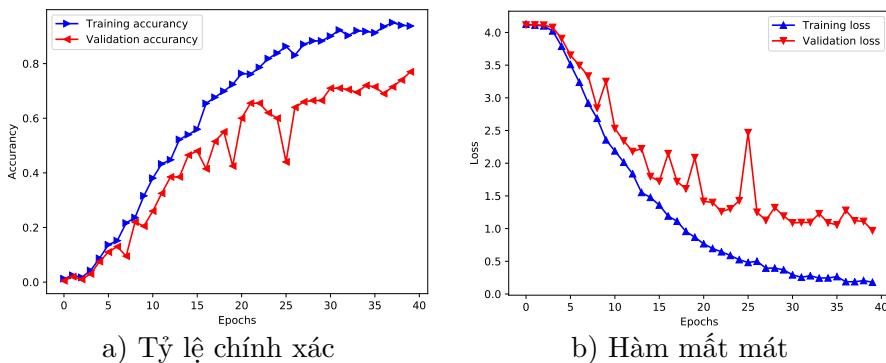
nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá khoảng 97% và 70% tương ứng. Hình 4.17(b) chỉ ra rằng hàm mất mát của dữ liệu đánh giá đạt cực tiểu tại vòng lặp thứ 22, hơn nữa hàm mất mát có xu hướng không giảm do vậy chúng ta chỉ cần chọn 22 vòng lặp là đủ để huấn luyện mạng. Sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 86.88%. So với kiến trúc mạng 1, kiến trúc mạng đã chọn có tỷ lệ nhận dạng chính xác tăng thêm khoảng 11%.

**Hình 4.17:** Thử nghiệm với kiến trúc mạng 2

Thử nghiệm với kiến trúc mạng 3

Hình 4.18(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 40 vòng lặp, phần trăm

nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá khoảng 92% và 67% tương ứng. Hình 4.18(b) chỉ ra rằng hàm mất mát của dữ liệu đánh giá đạt cực tiểu tại vòng lặp thứ 39, hơn nữa hàm mất mát đang có xu hướng giảm do vậy chúng ta tăng thêm số vòng lặp để đạt được giá trị của hàm mất mát bé hơn. Sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 78.68%. So với kiến trúc mạng 2, kiến trúc mạng đã chọn có tỷ lệ nhận dạng chính xác giảm khoảng 10%.



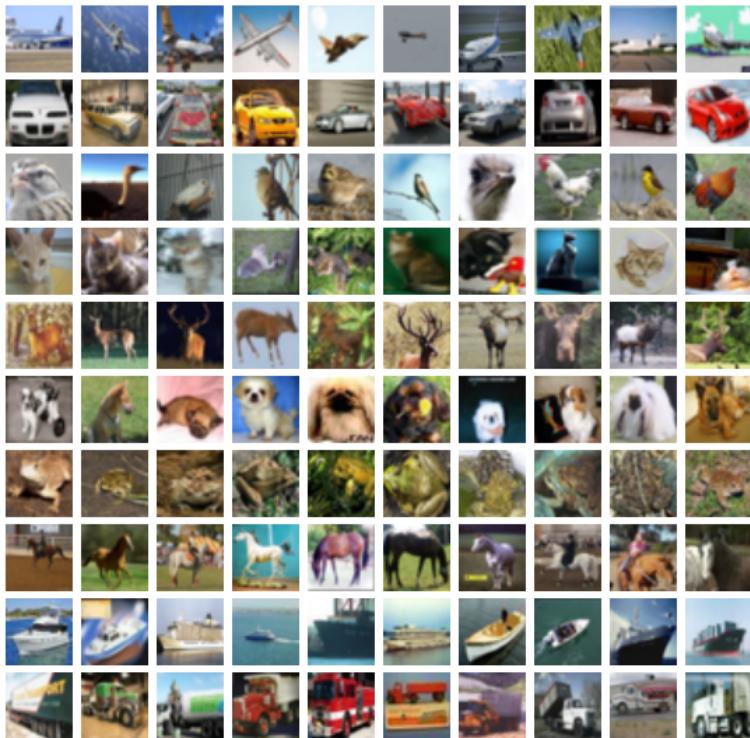
Hình 4.18: Thử nghiệm với kiến trúc mạng 3

Như vậy với 3 kiến trúc mạng đã chọn, kiến trúc mạng 2 cho tỷ lệ nhận dạng cao nhất.

4.5.3 Nhận dạng ảnh màu

Trong phần này chúng tôi thực hiện các thử nghiệm nhận dạng ảnh màu trong cơ sở dữ liệu CIFAR10 trong thư viện Keras. Cơ sở dữ liệu CIFAR10 bao gồm 60000 ảnh màu của 10 lớp dữ liệu, mỗi lớp có 6000 ảnh và mỗi ảnh có kích thước là 32×32 . Cơ sở dữ liệu CIFAR10 được chia thành 2 tập, tập huấn luyện có 50000 ảnh và tập kiểm tra có 10000 ảnh. Các lớp dữ liệu trong cơ sở dữ liệu CIFAR10 bao gồm 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. Hình 4.19 chỉ ra 100 ảnh, mỗi lớp 10 ảnh, trong cơ sở dữ liệu CIFAR10.

Để đánh giá mô hình, chúng tôi sử dụng phương pháp đánh giá đơn giản (simple-validation). Cụ thể, chúng tôi lấy 50000 ảnh trong 60000 ảnh huấn luyện làm tập dữ liệu huấn luyện (training) và 10000 ảnh còn

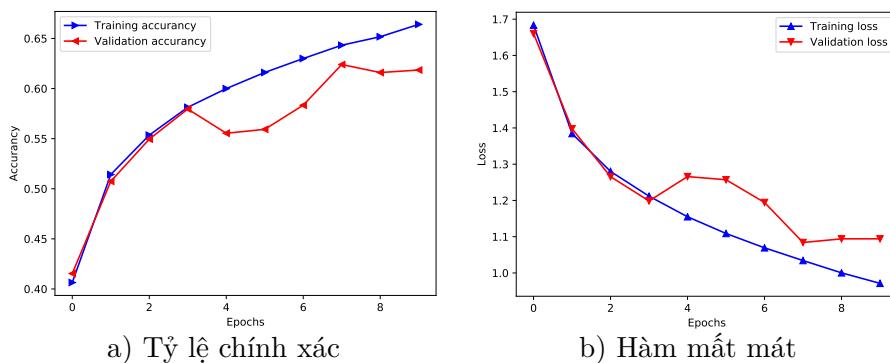


Hình 4.19: 100 ảnh đầu tiên trong cơ sở dữ liệu CIFAR10

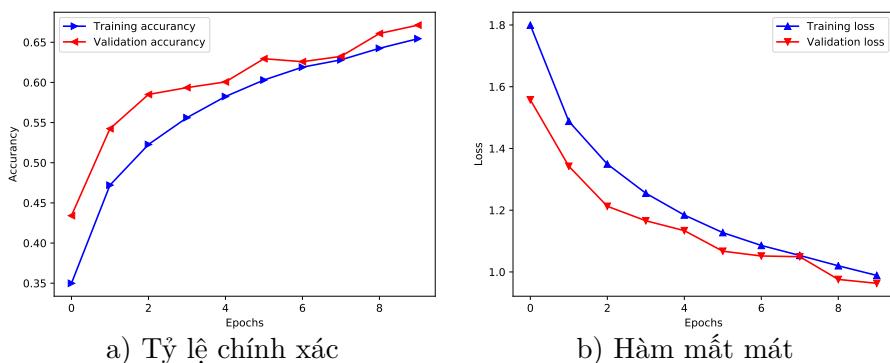
lại làm tập dữ liệu đánh giá (validation), và giữ nguyên bộ dữ liệu kiểm tra (test). Chúng tôi sử dụng 10 vòng lặp (epoch) để huấn luyện và kích thước mỗi lô (batch) dữ liệu là 128.

Thử nghiệm với kiến trúc mạng 1

Hình 4.20(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 10 vòng lặp, phần trăm nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá khoảng 68% và 62% tương ứng. Hình 4.20(b) chỉ ra rằng hàm mất mát của dữ liệu đánh giá đạt cực tiểu tại vòng lặp thứ 7, sau đó hàm có chiều hướng không giảm do đó chỉ cần 7 vòng lặp huấn luyện mang. Sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 61.50%. Rõ ràng với kiến trúc mạng này, tỷ lệ nhận dạng chính xác là rất thấp.

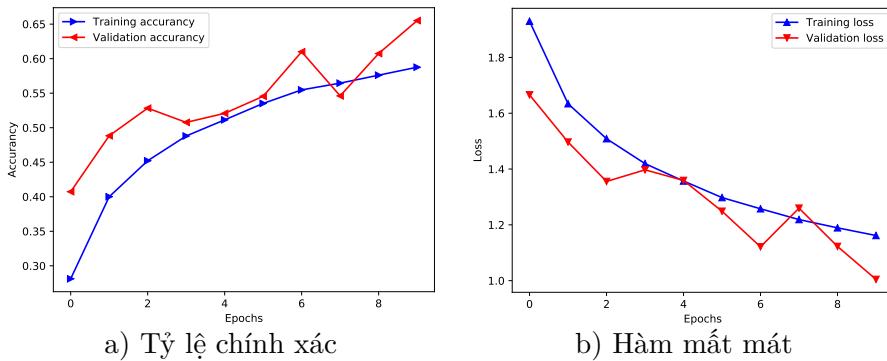
**Hình 4.20:** Thử nghiệm với kiến trúc mạng 1**Thử nghiệm với kiến trúc mạng 2**

Hình 4.21(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 10 vòng lặp, phần trăm nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá khoảng 63% và 66% tương ứng. Hình 4.21(b) chỉ ra rằng hàm mất mát của dữ liệu đánh giá đạt cực tiểu tại vòng lặp thứ 9, tuy nhiên hàm đang có chiều hướng giảm do đó cần tăng thêm các vòng lặp để đạt được giá trị hàm cực tiểu bé hơn. Sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 66.64%. Với kiến trúc mạng này, tỷ lệ nhận dạng chính xác là đã tăng lên so với sử dụng kiến trúc mạng 1.

**Hình 4.21:** Thử nghiệm với kiến trúc mạng 2

Thử nghiệm với kiến trúc mạng 3

Hình 4.22(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 10 vòng lặp, phần trăm nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá khoảng 57% và 66% tương ứng. Hình 4.22(b) chỉ ra rằng hàm mất mát của dữ liệu đánh giá đạt cực tiểu tại vòng lặp thứ 9, tuy nhiên hàm đang có chiều hướng giảm do đó cần tăng thêm các vòng lặp để đạt được giá trị hàm cực tiểu bé hơn. Sau khi huấn luyện xong, kết quả nhận dạng chính xác cho bộ dữ liệu kiểm tra là 64.45%. Với kiến trúc mạng này, tỷ lệ nhận dạng chính xác đã giảm xuống so với sử dụng kiến trúc mạng 2.



Hình 4.22: Thử nghiệm với kiến trúc mạng 3

Thử nghiệm với kiến trúc mạng 4

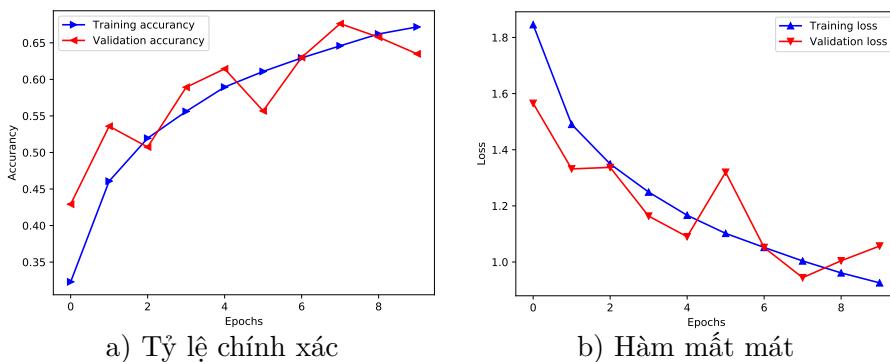
Trong 3 kiến trúc mạng đã sử dụng ở trên, dễ thấy rằng tỷ lệ nhận dạng chính xác cho bộ dữ liệu kiểm tra là khá thấp, do đó chúng tôi tăng số neuron của các tầng trong kiến trúc mạng 3 nhằm đánh giá hiệu quả của mạng. Cụ thể, kiến trúc mạng được định nghĩa như bảng 4.4.

Hình 4.23(a) chỉ ra đồ thị của phần trăm nhận dạng chính xác cho tập dữ liệu huấn luyện và tập dữ liệu đánh giá. Sau 10 vòng lặp, phần trăm nhận dạng chính xác của tập dữ liệu huấn luyện và tập dữ liệu đánh giá khoảng 67% và 63% tương ứng. Hình 4.23(b) chỉ ra rằng hàm mất mát của dữ liệu đánh giá đạt cực tiểu tại vòng lặp thứ 7, sau đó đang có chiều hướng tăng dần do đó chỉ cần dừng ở 7 vòng lặp huấn luyện mạng là đủ. Sau khi huấn luyện xong, kết quả nhận dạng chính

Bảng 4.4: Kiến trúc mạng 4

Kiểu tầng mạng	Tham số vào	Tham số ra	Số tham số
Conv2D	(3,3,’relu’)	(None, 30, 30, 32)	896
MaxPooling	(2,2)	(None, 15, 15, 32)	0
Dropout	0.35	(None, 15, 15, 32)	0
Conv2D	(3,3,’relu’)	(None, 13, 13, 64)	18496
MaxPooling	(2,2)	(None, 6, 6, 64)	0
Dropout	0.35	(None, 6, 6, 64)	0
Conv2D	(3,3,’relu’)	(None, 4, 4, 128)	73856
MaxPooling	(2,2)	(None, 2, 2, 128)	0
Dropout	0.35	(None, 2, 2, 128)	0
Flatten	none	(None, 512)	0
Dense	(64,’relu’)	(none, 512)	262656
Dense	(10,’softmax’)	(none, 10)	5130

xác cho bộ dữ liệu kiểm tra là 63.22%. Với kiến trúc mạng này, tỷ lệ nhận dạng chính xác đã giảm xuống so với sử dụng kiến trúc mạng 3.

**Hình 4.23:** Thử nghiệm với kiến trúc mạng 4

Như vậy, có thể thấy rằng tất cả các kiến trúc mạng đã sử dụng để nhận dạng bộ dữ liệu CIFAR10 cho tỷ lệ nhận dạng tương đối thấp và kiến trúc mạng 2 cho kết quả nhận dạng là tốt nhất.

Tài liệu tham khảo

- [1] GULLI, A., AND PAL, S. *Deep Learning with Keras*. Packt, 2017.