LỜI MỞ ĐẦU

Trong thời đại của thế kỷ 21, cùng với sự nhảy vọt của trình độ khoa học kỹ thuật, con người đã tạo nên và phát triển nhiều công trình khoa học mang tính tầm cỡ, với sự thay thế dần của máy tính cho con người trong các nhiệm vụ mang tính chính xác và tập trung cao.

Để thực hiện được các nhiệm vụ đó, máy tính cần nắm bắt thông tin từ môi trường bên ngoài để xử lý, đó là mục đích chính của thị giác máy tính - Computer vision. Trong đó, nhiệm vụ cơ bản là phân loại hình ảnh theo một nhóm các lớp hữu hạn, hay còn được gọi là nhận dạng hình ảnh.

Máy tính không thể xem hình ảnh như cách mà con người thực hiện, dưới góc nhìn của nó, hình ảnh chỉ đơn thuần là một bó dữ liệu được biểu diễn dưới dạng ma trận số nhiều chiều. Mạng neuron - Neuron Network được sinh ra để giải quyết vấn đề này. Tuy nhiên, vì rào cản tốc độ tính toán cũng như không gian lưu trữ, đến đầu thế kỉ 21, mới được phát triển, ứng dụng rộng rãi và đạt nhiều kết quả ấn tượng.

Mục tiêu của đồ án ***"Tìm hiểu mạng neuron và xây dựng chương trình nhận dạng chó mèo"*** *là tìm hiểu cấu trúc cùng các thành phần của mạng neuron nói chung và mạng neuron chồng chập nói riêng, cũng như các yếu tố ảnh hưởng; xây dựng chương trình bằng ngôn ngữ Python nhằm nhận diện chó và mèo trong tập hợp ảnh cho trước.*

Đồ án được chia làm 5 phần lớn:

- ***Chương I:*** *Giới thiệu về trí tuệ nhân tạo và mạng neuron nhân tạo*
- ***Chương II:*** *Cơ sở lý thuyết mạng neuron nhân tạo*
- ***ChươngIII:*** *Phân tích và thiết kế hệ thống*
- ***Chương IV:*** *Triển khai chương trình và đánh giá kết quả*
- ***Chương V:*** *Đánh giá và rút ra kết luận. Các hướng phát triển*

Trong quá trình thực hiện đề tài em luôn cố gắng tận dụng những kiến thức đã học ở trường cùng với sự tìm tòi, nghiên cứu. Em xin chân thành cảm ơn các thầy, cô giáo bộ môn và đặc biệt là TS. Phạm Minh Tuấn đã tận tình hướng dẫn em hoàn thành đề tài này.

Do thời gian làm đồ án có hạn và trình độ còn nhiều hạn chế nên không thể tránh khỏi những thiếu sót. Em rất mong nhận được sự đóng góp ý kiến của các thầy, cô giáo cũng như các bạn sinh viên để bài đồ án này hoàn thiện hơn nữa.

Một lần nữa em xin chân thành cảm ơn.

.

. Đà Nẵng, tháng 05 năm 2018

Sinh viên thực hiện

Lê Trọng Hiếu

MỤC LỤC

DANH SÁCH HÌNH VẼ

DANH SÁCH BẢNG BIỂU

DANH SÁCH TỪ VIẾT TẮT

(A)NN	(Artificial) Neuron Network
CNN	Convolution Neuron Network
ADAGRAD	Adaptive Subgradient Method
ADAM	Adaptive Moment Estimation
NAG	Nesterov Accelerated Gradient
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent

CHƯƠNG I: GIỚI THIỆU ĐỀ TÀI

I.1. Trí tuệ nhân tạo

I.1.1. Khái niệm

Trí tuệ nhân tạo (Artificial Intelligence) là chuyên nghành hẹp của khoa học máy tính với mục tiêu mô phỏng trí tuệ con người bởi các hệ thống máy tính nhằm thực hiện các tác vụ đặc trưng của trí tuệ con người như: nhận dạng hình ảnh, âm thanh, đưa ra quyết định và dịch,…

I.1.2. Học máy

Học máy (Machine Learning) lại là chuyên ngành hẹp của trí tuệ nhân tạo, là một trong nhiều phương pháp để xây dựng trí tuệ nhân tạo, tập trung vào phát triển các chương trình máy tính có thể truy cập dữ liệu và tiếp thu tri thức từ đó mà không cần sự can thiệp của con người. Arthur Samuel, một trong những người tiên phong trong học máy coi học máy là cung cấp cho máy tính khả năng tự học mà không phải lập trình rõ ràng (explicitly programme) như bằng một dãy các câu lệnh if-then.

I.1.2.1. Quá trình học

I.1.2.1.1. Quy tắc học

Quy tắc học (Learning rule), hay còn gọi là thuật toán huấn luyện (training algorithm) là quá trình chỉnh sửa weight và bias của mạng neuron nhằm thực hiện một tác vụ nào đó. Có thể chia thành ba nhóm: học giám sát (supervised learning), học không giám sát (unsupervised learning) và học tăng cường (reinforcement learning).

I.1.2.1.2. Học giám sát

Người ta đưa vào mạng neuron một tập hợp gọi là tập huấn luyện (training set) mà mỗi phần tử gọi là một ví dụ (example) gồm một input và output mong muốn tương ứng (target):

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, ..., \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

Trong đó Q là số ví dụ của tập huấn luyện.

Với mỗi example, input sẽ được đưa vào mạng neuron, khi đó output của mạng sẽ được so sánh với target. Lúc này, learning rule sẽ được sử dụng nhằm hiệu chỉnh các weight và bias của mạng neuron sao cho output của mạng gần giống với target.

I.1.2.1.3. Học không giám sát

Khác với học giám sát, mỗi ví dụ của tập huấn luyện chỉ gồm input mà không có target. Quy tắc học không giám sát thường được sử dụng trong các tác vụ phân nhóm (clustering operation).

I.1.2.1.4. Học tăng cường

Là thuật toán học có khả năng giải quyết được các bài toán thực tế phức tạp trong đó có sự tương tác giữa hệ thống và môi trường bên ngoài. Với những tình huống môi trường không chỉ đứng yên, cố định mà thay đổi phức tạp thì các phương pháp học truyền thống không thể đáp ứng được.

I.2. Mạng lưới thần kinh nhân tạo

I.2.1. Lịch sử

Những công trình đầu tiên về cơ chế hoạt động của thần kinh được nghiên cứu vào cuối thế kỷ XIX bởi các nhà vật lý, tâm lý học và sinh lý học thần kinh như Hermann von Helmholtz, Ernst Mach and Ivan Pavlov. Những công trình này đưa ra những lý thuyết nền tảng về sự lan truyền xung động thần kinh, cơ chế học, sự tiếp nhận và xử lý hình ảnh cũng như sự hình thành phản xạ có điều kiện và không điều kiện của não người.

Vào những năm 1940, Warren McCulloch và Walter Pitts chỉ ra rằng mạng lưới thần kinh nhân tạo (Artificial Neural Network - ANN) có thể thực hiện được mọi phép tính số học và logic. Và đây được xem như là nền tảng đầu tiên của ANN.

Những ứng dụng đầu tiên của ANN dựa trên sự phát minh ra mạng Perceptron cũng như quy tắc học liên kết (Associated learning rule) của Frank Rosenblatt. Rosenblatt cùng cộng sự đã xây dựng mạng Perceptron và biểu diễn khả năng của nó trong bài toán nhận diện khuôn mẫu (Pattern recognition).

Cùng lúc đó, Bernard Widrow và Ted Hoff giới thiệu một thuật toán học mới và sử dụng nó để huấn luyện mạng lưới thần kinh tuyến tính thích nghi (Adaptive linear neural network) với cấu trúc và khả năng tương tự như Perceptron. Tuy nhiên, cả hai mạng lưới của Rosenblatt và Widrow đều có nhược điểm, chỉ giải được một lớp bài toán, và không thể hiệu chỉnh để huấn luyện mạng lưới phức tạp hơn.

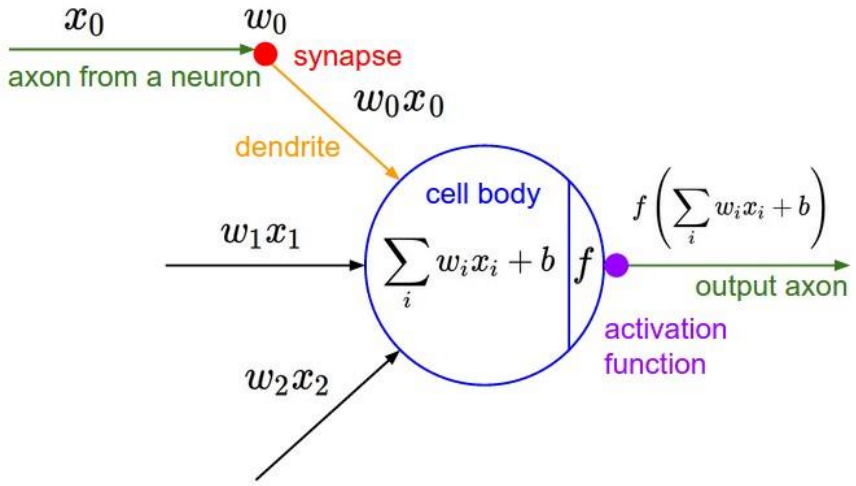
Trải qua thời kỳ gián đoạn bởi vì thiếu ý tưởng mới cũng như sự giới hạn sức mạnh tính toán của máy tính thời kỳ đó, đến những năm 80 của thế kỷ XIX, cả hai rào cản này đều được vượt qua và những nghiên cứu về mạng lưới thần kinh lại được phát triển mạnh mẽ với hai nội dung mới được giới thiệu.

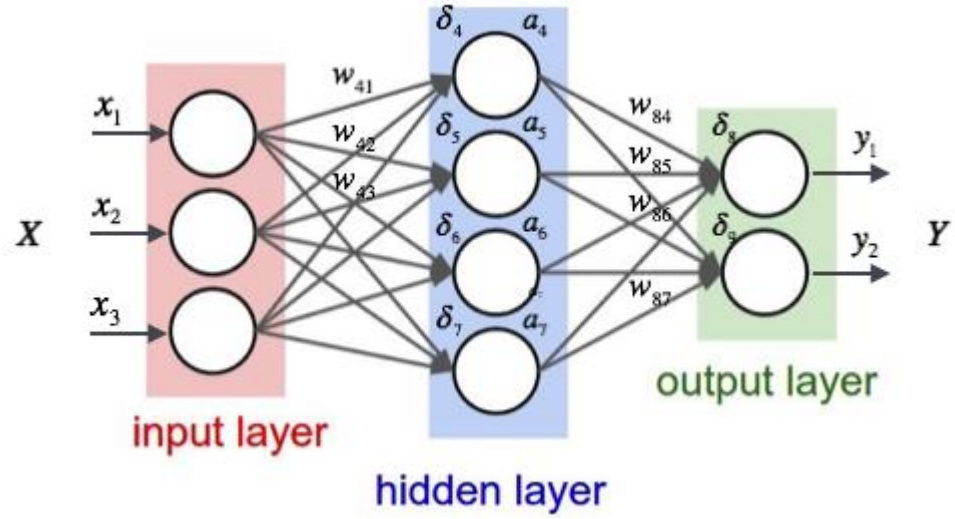
Một là việc sử dụng thống kê để giải thích một lớp các mạng lưới hồi quy (Recurrent network) được phát triển bởi nhà vật lý John Hopfield. Hai là thuật toán truyền ngược (Backpropagation algorithm) dùng để huấn luyện mạng Perceptron nhiều lớp bởi nhiều nhà khoa học độc lập, trong đó tiêu biểu nhất là David Rumelhart và James McClelland.

Kể từ đó đến nay, mạng lưới thần kinh nhân tạo cùng các biến thể của nó đóng một vai trò quan trọng trong trí tuệ nhân tạo nói chung và học máy nói riêng.

I.2.2. Neuron nhân tạo[1]

Mạng neuron được xây dựng từ một hay nhiều thành phần cơ bản, gọi là neuron (Artificial neuron). Chúng nhận một hay nhiều **input**, thay đổi trạng thái nội tại của nó (gồm một bộ các số gọi là **weight** và **bias** kèm một hàm gọi là **activation function**) theo input tương ứng, trả về **output**. Mạng neuron hình thành bằng sự liên kết của các neuron, khi đó output của neuron này là input của neuron khác.





1 Kể từ đây, khi nói đến neuron tức là đang nói đến neuron trong mạng lưới thần kinh nhân tạo.

CHƯƠNG II: CƠ SỞ LÝ THUYẾT

II.1. Không gian vector

II.1.1. Định nghĩa

Không gian vector trên trường *F* là một tập hợp *V* được trang bị hai phép toán:

- Phép cộng $+ : V \times V \to V$, nhận hai phần tử **v** và **w** thuộc *V* và trả về một phần tử **v** + **w** gọi là tổng.
- Phép nhân $\cdot : F \times V \to V$, nhận một phần tử *a* của trường *F* cùng một phần tử **v** thuộc *V* và trả về vector *a***v**.

và thỏa mãn 8 tiên đề sau với **u**, **v**, **w** thuộc *V* và *a*, *b* thuộc *F:*

1. Tính giao hóan đối đối phép cộng trên *V*:
$$\mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$$
2. Tính kết hợp đối với phép cộng trên *V*:
$$\mathbf{u} + \left(\mathbf{v} + \mathbf{w}\right) = \left(\mathbf{u} + \mathbf{v}\right) + \mathbf{w}$$
3. Tồn tại phần tử trung hòa đối với phép cộng trên *V*:
$$\mathbf{0} + \mathbf{v} = \mathbf{v}$$
4. Tồn tại phần tử đối:
$$\$\left(-\mathbf{v}\right) \hat{\imath}\ V : \left(-\mathbf{v}\right) + \mathbf{v} = \mathbf{0}$$
5. Tính kết hợp giữa phép nhân trên *V* và phép nhân trên *F*:
$$a\left(b\mathbf{v}\right) = \left(ab\right)\mathbf{v}$$
6. Tính tương thích giữa phần tử đơn vị 1 của *F* với *V*:
$$1\mathbf{v} = \mathbf{v}$$
7. Tính phân phối giữa phép cộng trên *F* và phép nhân trên *V*:
$$\left(a + b\right)\mathbf{v} = a\mathbf{v} + b\mathbf{v}$$
8. Tính phân phối giữa phép cộng và phép nhân trên V:
$$a\left(\mathbf{v} + \mathbf{u}\right) = a\mathbf{v} + a\mathbf{u}$$

Các phần tử của V và F thường được gọi là vector và scalar tương ứng.

II.1.2. Ví dụ

II.1.2.1. Không gian vector $\mathbb{R}^n$

Với số nguyên dương n bất kỳ, tập hợp gồm các bộ n phần tử thuộc *F* tạo nên không gian vector n chiều trên *F*, kí hiệu là F^n, mỗi phần tử thuộc F^n được viết dưới dạng:

$$x = \left(x_1, x_2, ..., x_n\right), x_i \hat{\imath}\ F$$

Với phép cộng và phép nhân được định nghĩa như sau:

$$x + y = \left(x_1 + y_1, x_2 + y_2, ..., x_n + y_n\right)$$

$$ax = \left(ax_1, ax_2, ..., ax_n\right)$$

Các phần tử trung hòa và phần tử đối là:

$$0 = \left(0, 0, ..., 0\right)$$

$$-x = \left(-x_1, -x_2, ..., -x_n\right)$$

Khi *F* là trường số thực $\mathbb{R}$, ta có không gian vector $\mathbb{R}^n$, đây cũng là đối tượng được khảo sát chính của đồ án.

Ngoài ra, vector **x** thuộc $\mathbb{R}^n$ thường được viết dưới dạng ma trận kích thước 1 x n hoặc n x 1, và được gọi là ma trận hàng, ma trận cột tương ứng.

II.1.2.2. Ma trận

Một ma trận trên trường *F* là một bảng số gồm một hay nhiều hàng và cột, mà các phần tử của nó thuộc *F*. Ma trận gồm m *hàng* và n *cột* gọi là ma trận m x n với m, n là các *chiều* của nó. Ví dụ:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} = \left(a_{i,j}\right), a_{i,j} \in F$$

Ma trận gồm một hàng (cột) lần lượt gọi là *vector hàng* (*cột*). Ma trận có số hàng và cột bằng nhau gọi là *ma trận vuông*.

Ma trận kí hiệu bằng các chữ cái in hoa in đậm như **M**, **N**. Phần tử ở hàng i, cột j của ma trận **A** được kí hiệu là $\mathbf{A}_{i,j}$ hoặc **A**[i,j]. Ta có các phép toán cơ bản trên ma trận sau:

- Phép cộng hai ma trận cùng kích thước:

$$\left(\mathbf{A} + \mathbf{B}\right)_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$$

- Phép nhân ma trận với một phần tử thuộc trường *G:*

$$\left(c\mathbf{A}\right)_{i,j} = \left(ca_{i,j}\right)$$

- Phép nhân hai ma trận kích thước m x n và n x p:

$$\left(\mathbf{AB}\right)_{i,j} = \sum_{k=1}^{n} \mathbf{A}_{i,k}\mathbf{B}_{k,j}$$

- Phép chuyển vị:

$$\left(\mathbf{A}^T\right)_{i,j} = \mathbf{A}_{j,i}$$

Với các phép cộng và phép nhân hiển nhiên thì tập các ma trận kích thước m x n trên trường **F** tạo thành không gian vector $\mathbf{F}^{mxn}$ trên **F**.

Nhằm tính toán trên dữ liệu hình ảnh, ta mở rộng khái niệm ma trận 2 chiều ở trên thành ma trận m chiều với kích thước n_1 x n_2 x … x n_m và đồng nhất với kiểu dữ liệu array của thư viện Numpy trong Python, với phép cộng (nhân) hai ma trận **A**, **B** cùng kích thước tạo thành ma trận **C** thứ ba có các phần tử là tổng (tích) của hai phần tử tương ứng của hai ma trận **A** và **B.**

II.2. Lý thuyết tối ưu

II.2.1. Khái niệm và định nghĩa

Trong không gian vector $\mathbb{R}^n$, cho $D \subseteq \mathbb{R}^n$ là một tập khác rỗng và hàm số thực $f: D \longrightarrow \mathbb{R}$. Bài toán tối ưu có dạng

$$\min\left\{ f\left(x\right) : x \in D \right\} \qquad (1.1)$$

là bài toán tìm vector $\mathbf{x}^*$ thuộc *D* sao cho $f(\mathbf{x}^*) \leq f(\mathbf{x})$ với mọi **x** thuộc *D*.

Hàm *f* gọi là *hàm mục tiêu* hay *hàm chi phí*, tập *D* gọi là *tập ràng buộc*. Vector $\mathbf{x}^*$ thuộc D sao cho $f(\mathbf{x}^*) \leq f(\mathbf{x})$ với mọi **x** thuộc *D* gọi là *nghiệm tối ưu* của bài toán và $f(\mathbf{x}^*)$ là giá trị cực tiểu của *f* trên *D*.

Trường hợp $D = \mathbb{R}^n$ ta có bài toán tối ưu không ràng buộc. Trái lại, nếu *D* được cho bởi:

$$D = \left\{ x \in \mathbb{R}^n : g_i\left(x\right) \leq 0, i = \overline{1,m}; h_j\left(x\right) = 0, j = \overline{1,p} \right\}$$

ta có bài toán tối ưu có ràng buộc với $g_i(x), h_j(x): \mathbb{R}^n \longrightarrow \mathbb{R}$ là các hàm ràng buộc.

Điểm $\mathbf{x}^*$ gọi là *cực tiểu địa phương* của f trên *D* nếu tồn tại số thực δ dương sao cho $F(\mathbf{x}^*) \leq F(\mathbf{x})$ với mọi **x** thuộc D thỏa mãn $\|\mathbf{x} - \mathbf{x}^*\| < \delta$.

Điểm $\mathbf{x}^*$ gọi là *cực tiểu địa phương chặt* của f trên D nếu $F(\mathbf{x}^*) < F(\mathbf{x})$ với mọi **x** thuộc D, $\mathbf{x} \neq \mathbf{x}^*$ và $\|\mathbf{x} - \mathbf{x}^*\| < \delta$.

Điểm $\mathbf{x}^*$ gọi là *cực tiểu toàn cục* của F trên D nếu $F(\mathbf{x}^*) \leq F(\mathbf{x})$ với mọi **x** thuộc D.

Điểm $\mathbf{x}^*$ gọi là *cực tiểu toàn cục chặt* của F trên D nếu $F(\mathbf{x}^*) < F(\mathbf{x})$ với mọi **x** thuộc D và $\mathbf{x} \neq \mathbf{x}^*$.

II.2.2. Điều kiện tối ưu

Xét bài toán tối ưu không ràng buộc:

$$\min\left\{ f\left(x\right) : x \hat{I} \square^{n} \right\} \tag{1.2}$$

với f: $\mathbb{R}^n \rightarrow \mathbb{R}$ là một hàm phi tuyến cho trước.

Định lý 1.1: *Nếu* $\mathbf{x}^*$ *thuộc* $\mathbb{R}^n$ *là một điểm cực tiểu địa phương của một hàm* $f(\mathbf{x})$ *khả vi trên* $\mathbb{R}^n$ *thì* $\nabla f(\mathbf{x}^*) = 0$, *và nếu* $f(\mathbf{x})$ *khả vi hai lần thì* $\nabla^2 f(\mathbf{x}^*) \geq 0$ *(nửa xác định dương*[2]*), với* $\nabla f(\mathbf{x})$ *và* $\nabla^2 f(\mathbf{x})$ *lần lượt là gradient và ma trận Hessian của* $f(\mathbf{x})$.

Nếu $\mathbf{x}^*$ *thuộc* $\mathbb{R}^n$ *là một điểm mà tại đó* $f(\mathbf{x})$ *khả vi hai lần và* $\nabla f(\mathbf{x}^*) = 0$, $\nabla^2 f(\mathbf{x}^*) > 0$ *thì* $\mathbf{x}^*$ *là một điểm cực tiểu địa phương chặt.*

II.2.3. Phương pháp hướng dốc nhất

Phương pháp hướng dốc nhất (Steepest descent) khởi tạo một giá trị ban đầu $\mathbf{x}_0$, sau từng bước ta sẽ cập nhật lại giá trị bằng phương trình:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - a_k \mathbf{g}_k \Leftrightarrow \Delta \mathbf{x}_k = - a_k \mathbf{g}_k \tag{1.3}$$

với α_k dương là *tốc độ học* tại bước k, $\mathbf{g}_k$ là gradient tại $\mathbf{x}_k$:

$$\mathbf{g}_k = \nabla f\left(\mathbf{x}\right)^T \Big|_{\mathbf{x}=\mathbf{x}_k} = \left[\begin{array}{cccc} \frac{\partial}{\partial x_1} f\left(\mathbf{x}\right) & \frac{\partial}{\partial x_2} f\left(\mathbf{x}\right) & \cdots & \frac{\partial}{\partial x_n} f\left(\mathbf{x}\right) \end{array} \right]^T \Bigg|_{\mathbf{x}=\mathbf{x}_k} \tag{1.4}$$

Khi đó, sau mỗi bước ta sẽ có:

$$f\left(\mathbf{x}_{k+1}\right) < f\left(\mathbf{x}_k\right)$$

Thật vậy, xét khai triển Taylor bậc nhất của f tại $\mathbf{x}_k$:

$$f\left(\mathbf{x}\right) @ f\left(\mathbf{x}_k\right) + \nabla f\left(\mathbf{x}\right)^T \Big|_{\mathbf{x}=\mathbf{x}_k} \left(\mathbf{x} - \mathbf{x}_k\right) = f\left(\mathbf{x}_k\right) + \mathbf{g}_k^T \left(\mathbf{x} - \mathbf{x}_k\right) \tag{1.5}$$

Thay $\mathbf{x}$ bằng $\mathbf{x}_{k+1}$, ta có:

$$f\left(\mathbf{x}_{k+1}\right) @ f\left(\mathbf{x}_k\right) + \mathbf{g}_k^T \left(\mathbf{x}_k - a_k \mathbf{g}_k - \mathbf{x}_k\right) = f\left(\mathbf{x}_k\right) - a_k \mathbf{g}_k^T \mathbf{g}_k < f\left(\mathbf{x}_k\right) \tag{1.6}$$

Phương pháp hướng dốc nhất kinh điển cài đặt hệ số học α là hằng số ở mọi bước và thường có giá trị 0,1; 0,01;… Sau đây ta sẽ xét các biến thể của phương pháp này.

---

2 Ma trận thực đối xứng $\mathbf{M}$ gọi là xác định dương (nửa xác định dương) nếu $\mathbf{z}^T\mathbf{M}\mathbf{z}$ dương (không âm) với mọi vector $\mathbf{z}$ thực. Tương tự ta có ma trận xác định âm và nửa xác định âm.

II.2.4. Các biến thể của phương pháp hướng dốc nhất

II.2.4.1. Momentum

Là phương pháp mở rộng đơn giản nhất của SGD, ý tưởng là thêm một đại lượng gọi là momentum đóng vai trò như khối lượng làm vật thể có xu hướng tiến về điểm cực tiểu, càng gần điểm cực tiểu tốc độ càng tăng:

$$\Delta \mathbf{x}_k = \gamma \Delta \mathbf{x}_{k-1} - \alpha_k \mathbf{g}_k \tag{1.7}$$

với γ là momentum, thường có giá trị $0{,}8 < \gamma < 1$.

II.2.4.2. ADAGRAD

Phương pháp ADAGRAD (Adaptive Subgradient method) không sử dụng hệ số học là hằng số như phương pháp kinh điển, thay vào đó được hiệu chỉnh như sau:

$$\Delta \mathbf{x}_k = -\frac{\alpha}{\sqrt{\sum_{t=0}^{k} \mathbf{g}_t^2 + \varepsilon}} \mathbf{g}_k \tag{1.8}$$

Với mẫu số là L_2-norm của tất cả các gradient các bước trước và ε rất nhỏ, khoảng 10^{-8} nhằm tránh chia cho 0.

ADAGRAD có ưu điểm là thay đổi tốc độ học tùy theo độ lớn của gradient, điều này có lợi khi huấn luyện mạng neuron nhiều lớp nơi độ lớn của gradient tại mỗi lớp là khác nhau. Tuy nhiên, phương pháp này có nhược điểm là rất nhạy với điều kiện ban đầu, nếu gradient ban đầu lớn dẫn đến tốc độ học chậm về sau, hơn nữa, tốc độ học sẽ tiến nhanh về 0 và dừng hoàn toàn quá trình học.

II.2.4.3. ADADELTA

Phương pháp ADADELTA giữ lại hai ưu điểm của phương pháp momentum và ADAGRAD. Ý tưởng chính là thay vì cộng dồn bình phương gradient của tất cả các bước trước như ADAGRAD, ADADELTA chỉ cộng dồn bình phương một số cố định *w* gradient trước đó, đảm bảo rằng quá trình học vẫn còn ý nghĩa sau nhiều bước cập nhật. Tuy nhiên, lưu giữ *w* gradient trước cũng không hiệu quả, ADADELTA ước lượng trung bình của tổng bình phương gradient bằng phương trình:

$$E\left[\mathbf{g}^2\right]_k = \gamma E\left[\mathbf{g}^2\right]_{k-1} + \left(1 - \gamma\right)\mathbf{g}_k^2 \tag{1.9}$$

Từ đó ta có quy tắc cập nhật:

$$\Delta \mathbf{x}_k = -\frac{\alpha}{\sqrt{E\left[\mathbf{g}^2\right]_k + \varepsilon}} \mathbf{g}_k \tag{1.10}$$

II.2.4.4. ADAM

Quy tắc cập nhật của phương pháp ADAM (Adaptive moment estimation) được cho bởi:

$$x_k = x_{k-1} - \frac{\alpha}{\hat{v}_k + \varepsilon}\hat{m}_k \tag{1.11}$$

$$\hat{m}_k = \frac{m_k}{1-\beta_1^t}, m_k = \beta_1 m_{k-1} + (1-\beta_1) g_t \tag{1.12}$$

$$\hat{v}_k = \frac{v_k}{1-\beta_2^t}, v_k = \beta_2 v_{k-1} + (1-\beta_2) g_t^2 \tag{1.13}$$

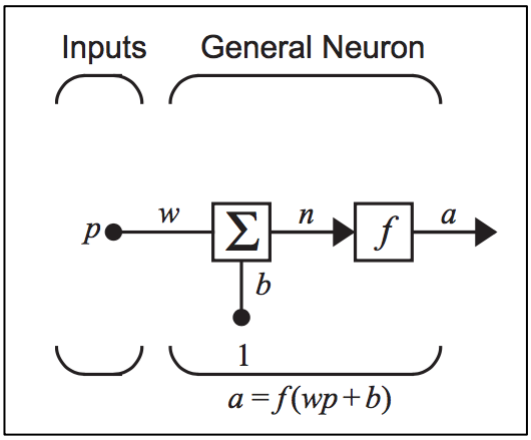
Với các hằng số thường là: $\beta_1 = 0{,}9$; $\beta_2 = 0{,}999$; $\varepsilon = 10^{-8}$; $\alpha = 0{,}001$.

Ngoài ra, trong đồ án này còn sử dụng thêm hai phương pháp khác để so sánh hiệu quả học là RMSprop và Nesterov accelerated gradient - NAG.

II.3. Mạng lưới thần kinh nhân tạo và thuật toán truyền ngược

II.3.1. Single-input neuron và Multiple-input neuron

II.3.1.1. Single-input neuron



HìnhII.1: Cấu trúc single-input neuron

Hình 1.1 mô tả cấu trúc tổng quát của single-input neuron[3] trong đó neuron nhận một input **p**, được kết hợp với một weight **w** bằng một toán tử 2 ngôi bất kỳ nào đó (thường là phép tính nhân, tổng quát hơn là phép tính nhân hai ma trận) rồi cộng thêm một bias **b**, cho ta **net input n**:

$$n = wp + b \tag{1.14}$$

3 Thành phần của neuron được người viết giữ nguyên tiếng Anh.

Đến lượt nó, được đưa vào activation function f, kết quả là output **a** của neuron:

$$a = f(n) = f(wp + b) \tag{1.15}$$

Weight và bias là các tham số có thể hiệu chỉnh được của neuron thông qua quy tắc học nào đó. Activation function có thể tuyến tính hoặc phi tuyến, được chọn khi thiết kế neuron tùy theo mục đích sử dụng. Một số activation function thông dụng như:

- Linear:

$$a = n \tag{1.16}$$

- ReLU (Rectified linear unit):

$$a = \max(n, 0) \tag{1.17}$$

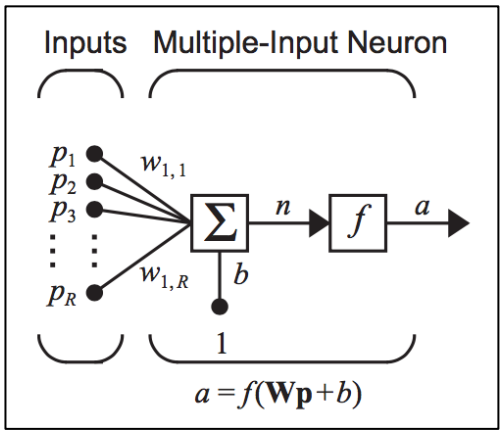
- Sigmoid:

$$a = \frac{1}{1 + e^{-n}} \tag{1.18}$$

- Hyperbolic Tangent Sigmoid:

$$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}} \tag{1.19}$$

II.3.1.2. Multiple-input neuron



Hình II.2: Cấu trúc multiple-input neuron

Cấu trúc tương tự như single-input nhưng thay vì chỉ nhận một input, multiple-input neuron nhận nhiều input $p_1, p_2, \ldots, p_R$ với các weight tương ứng là $w_1, w_2, \ldots, w_R$. Khi đó, net input và output được tính như sau:

$$n = w_1 p_1 + w_2 p_2 + \ldots + w_R p_R + b = \sum_{i=1}^{R} w_i p_i + b \tag{1.20}$$

$$a = f\left(w_1 p_1 + w_2 p_2 + ... + w_R p_R + b\right) = f\left(\sum_{i=1}^{R} w_i p_i + b\right) \quad (1.21)$$

Đặt **p**, **W** lần lượt là ma trận hàng của input và ma trận cột của weight:

$$\mathbf{p} = \begin{bmatrix} p_1 & p_2 & \cdots & p_R \end{bmatrix}$$
$$\mathbf{W} = \begin{bmatrix} w_1 & w_2 & \cdots & w_R \end{bmatrix}^T \quad (1.22)$$

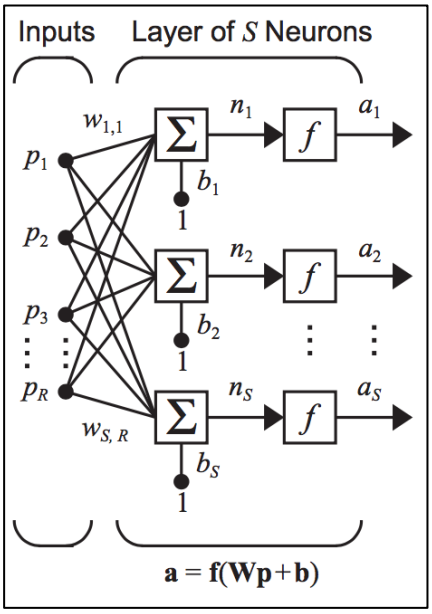
Ta có thể viết lại (1.20), (1.21) thành:

$$n = \mathbf{W}\mathbf{p} + b \quad (1.23)$$

$$a = f\left(\mathbf{W}\mathbf{p} + b\right) \quad (1.24)$$

II.3.2. Cấu trúc mạng neuron kinh điển

Thành phần cơ bản của mạng neuron kinh điển là các lớp (layer). Mỗi layer gồm **S** multiple-input neuron, thường có cùng activation function và cùng nhận input **p** gồm **R** thành phần.



Hình II.3: Cấu trúc layer trong mạng neuron

Lúc này, output của layer là một vector:

$$\mathbf{n} = \mathbf{W}\mathbf{p} + \mathbf{b} \quad (1.25)$$

$$\mathbf{a} = f\left(\mathbf{n}\right) \quad (1.26)$$

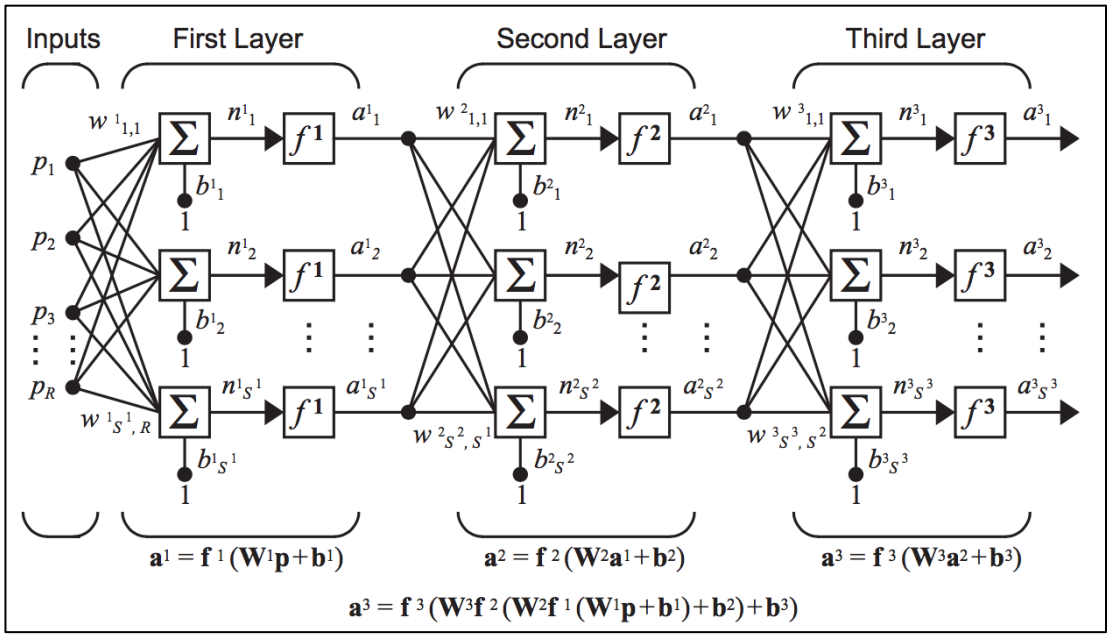
Trong đó, activation function được tính trên từng phần tử của ma trận **n**.

Với **W** là ma trận weight của layer với $\mathbf{w_{i,j}}$ là weight thứ j của neuron thứ i tức là weight liên kết từ thành phần thứ j của input đến neuron thứ i, **b** là ma trận bias của layer với $\mathbf{b_i}$ là bias của neuron thứ i:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & \ddots & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_S \end{bmatrix} \quad (1.27)$$

Mạng neuron cơ bản bao gồm nhiều layer xếp liên tiếp nhau. Mỗi layer có ma trận weight, vector bias của riêng nó. Input của layer này là output của layer kế trước nó, trừ layer đầu tiên nhận input của mạng neuron làm input của nó.

Từ đây, để phân biệt giữa các layer, ta đánh số các layer cũng như các thành phần của nó bằng các chỉ số trên. Ví dụ, $\mathbf{W^2}$ là ma trận weight của layer thứ 2.



Hình II.4: Ví dụ về mạng neuron gồm 3 layer

Layer có output là output của mạng neuron gọi là *output layer*. Các layer khác gọi là *hidden layer*. Mạng neuron ở hình trên có layer thứ ba là output layer, hai layer đầu là hidden layer.

Trên đây là cấu trúc kinh điển và đơn giản nhất của mạng neuron với các neuron được liên kết với toàn bộ các neuron trước nó, do đó còn có tên gọi là fully-connected Neural Network và các layer của nó là fully-connected layer hay FC layer. Phần sau ta sẽ xét mạng neuron mà ở đó mỗi neuron chỉ liên kết với một số nhất định các neuron khác.

II.3.3. Thuật toán truyền ngược

Thuật toán truyền ngược (Backpropagation algorithm) thường được kết hợp với thuật toán hướng dốc nhất nhằm hiệu chỉnh weight cùng bias theo chiều ngược với chiều tính toán của mạng neuron.

Ta xét mạng neuron nhiều lớp kinh điển gồm **M** layer, output của layer này là input của layer tiếp theo:

$$\begin{gathered}\mathbf{a}^0 = \mathbf{p} \\ \mathbf{a}^{m+1} = \mathbf{f}^{m+1}\left(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}\right), m = 0,1,..., M - 1\end{gathered} \tag{1.28}$$

Với **p** là input. Như vậy, ta được output của mạng neuron cũng chính là output của layer cuối cùng $\mathbf{a}^M$.

Với mỗi bộ input và target:

$$\left\{\mathbf{p}_i, \mathbf{t}_i\right\}, i = 1,..., Q \tag{1.29}$$

thuật toán sẽ hiệu chỉnh weight và bias của mạng neuron để cực tiểu hóa hàm chi phí4:

$$F\left(\mathbf{x}\right) = E\left[\mathbf{e}^T\mathbf{e}\right] = E\left[\left(\mathbf{t} - \mathbf{a}\right)^T\left(\mathbf{t} - \mathbf{a}\right)\right] \tag{1.30}$$

với **x** là vector của weight và bias của mạng neuron. Ta sẽ xấp xỉ hàm này bằng sai số đối với từng bộ input và target:

$$\widehat{F}\left(\mathbf{x}\right) = \mathbf{e}^T\left(i\right)\mathbf{e}\left(i\right) = \left(\mathbf{t}\left(i\right) - \mathbf{a}\left(i\right)\right)^T\left(\mathbf{t}\left(i\right) - \mathbf{a}\left(i\right)\right) \tag{1.31}$$

Trước khi xét đến quá trình truyền ngược, ta sẽ đi tính ma trận Jacobian sau:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_2^{m+1}}{\partial n_{S^m}^m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix} \tag{1.32}$$

Với n_j^i, S^i lần lượt là net input của neuron thứ j layer i và số neuron của layer i.

---

4 Ở đây ta chọn hàm chi phí là trung bình bình phương độ lệch cho đơn giản.

Xét phần tử ở hàng i cột j của ma trận:

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial\left(\sum_l^{S^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1}\right)}{\partial n_j^m} = w_{i,j}^{m+1}\frac{\partial a_j^m}{\partial n_j^m}$$
$$= w_{i,j}^{m+1}\frac{\partial f^m\left(n_j^m\right)}{\partial n_j^m} = w_{i,j}^{m+1}\dot{f}^m\left(n_j^m\right) \quad (1.33)$$

Từ đó, ma trận Jacobian ở trên có thể được viết thành:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1}\dot{\mathbf{F}}^m\left(\mathbf{n}^m\right) \quad (1.34)$$

Với:

$$\dot{\mathbf{F}}^m\left(\mathbf{n}^m\right) = \begin{bmatrix} \dot{f}^m\left(n_1^m\right) & 0 & \cdots & 0 \\ 0 & \dot{f}^m\left(n_2^m\right) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \dot{f}^m\left(n_{S^m}^m\right) \end{bmatrix} \quad (1.35)$$

Khi đó, nếu đặt độ nhạy của layer **m** là:

$$\mathbf{s}^m = \frac{\partial \widehat{F}}{\partial \mathbf{n}^m} \quad (1.36)$$

Ta có được hệ thức truy hồi của $s^{\mathbf{m}}$**:**

$$\mathbf{s}^m = \frac{\partial \widehat{F}}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial \widehat{F}}{\partial \mathbf{n}^{m+1}} = \dot{F}^m\left(\mathbf{n}^m\right)\left(\mathbf{W}^{m+1}\right)^T \frac{\partial \widehat{F}}{\partial \mathbf{n}^{m+1}} = \dot{F}^m\left(\mathbf{n}^m\right)\left(\mathbf{W}^{m+1}\right)^T \mathbf{s}^{m+1}$$
$$\Leftrightarrow \mathbf{s}^m = \dot{\mathbf{F}}^m\left(\mathbf{n}^m\right)\left(\mathbf{W}^{m+1}\right)^T \mathbf{s}^{m+1} \quad (1.37)$$

Với phần tử đầu $s^{\mathbf{M}}$**:**

$$s_i^M = \frac{\partial F}{\partial n_i^M} = \frac{\partial\left(\mathbf{t}-\mathbf{a}\right)^T\left(\mathbf{t}-\mathbf{a}\right)}{\partial n_i^M} = \frac{\partial\sum_{j=1}^{S^M}\left(t_j - a_j\right)^2}{\partial n_i^M} = \frac{\partial\sum_{j=1}^{S^M}\left(t_j - a_j\right)^2}{\partial n_i^M}$$

hay: $$s_i^M = -2\left(t_i - a_i\right)\frac{\partial a_i}{\partial n_i^M} = -2\left(t_i - a_i\right)\frac{\partial f^M\left(n_i^M\right)}{\partial n_i^M} = -2\left(t_i - a_i\right)\dot{f}^M\left(n_i^M\right)$$

suy ra: $$\mathbf{s}^M = -2\dot{\mathbf{F}}^M\left(\mathbf{n}^M\right)\left(\mathbf{t}-\mathbf{a}\right) \tag{1.38}$$

Quy trở lại với thuật toán truyền ngược, áp dụng thuật toán hướng dốc nhất, ta có được quy tắc cập nhật như sau:

$$w_{i,j}^m\left(k+1\right) = w_{i,j}^m\left(k\right) - \alpha\frac{\partial\widehat{F}}{\partial w_{i,j}^m} \tag{1.39}$$

$$b_i^m\left(k+1\right) = b_i^m\left(k\right) - \alpha\frac{\partial\widehat{F}}{\partial b_i^m} \tag{1.40}$$

với $w_{i,j}^m(k), b_i^m(k), \frac{\partial\hat{F}}{\partial x}$ lần lượt là weight liên kết giữa neuron thứ **i** và input thứ **j** của layer **m**; bias neuron thứ **i** của layer **m**; đạo hàm riêng phần của **F** theo biến x.

Nhờ các $s^{\mathbf{m}}$ tính được ở trên, ta có:

$$\frac{\partial\widehat{F}}{\partial w_{i,j}^m} = \frac{\partial\widehat{F}}{\partial n_i^m}\frac{\partial n_i^m}{\partial w_{i,j}^m} = s_i^m\frac{\partial\left(\sum_{k=1}^{s^{m-1}} w_{i,k}^m a_k^{m-1} + b_i^m\right)}{\partial w_{i,j}^m} = s_i^m a_j^{m-1} \tag{1.41}$$

$$\frac{\partial\widehat{F}}{\partial b_i^m} = \frac{\partial\widehat{F}}{\partial n_i^m}\frac{\partial n_i^m}{\partial b_i^m} = s_i^m\frac{\partial\left(\sum_{k=1}^{s^{m-1}} w_{i,k}^m a_k^{m-1} + b_i^m\right)}{\partial b_i^m} = s_i^m \tag{1.42}$$

Vậy ta có biểu diễn dưới dạng ma trận quy tắc cập nhật weight và bias của mạng neuron:

$$\mathbf{W}^m\left(k+1\right) = \mathbf{W}^m\left(k\right) - a\mathbf{s}^m\left(\mathbf{a}^{m-1}\right)^T \tag{1.43}$$

$$\mathbf{b}^m\left(k+1\right) = \mathbf{b}^m\left(k\right) - a\mathbf{s}^m \tag{1.44}$$

Tóm lại, thuật toán truyền ngược của thể chia thành 3 bước:

1. Đưa input vào mạng neuron, tính toán output của các layer.
2. Tính toán độ nhạy của từng layer theo phương trình (1.37) và (1.38)
3. Cập nhật weight và bias theo phương trình (1.43) và (1.44)

Bước tính toán đạo hàm riêng phần của hàm chi phí theo từng tham số (weight và bias) là điểm mấu chốt của thuật toán truyền ngược. Ở phần sau, khi phát triển ra mạng neuron chồng chập ta cũng sẽ tiếp tục tính toán đại lượng này.

II.3.4. Phương thức huấn luyện

II.3.4.1. Huấn luyện ngẫu nhiên

Huấn luyện ngẫu nhiên (Stochastic gradient descent - SGD) là thuật toán được cài đặt như trên, weight và bias đều được cập nhật khi mà với mỗi example của training set được đưa vào mạng neuron. Phương thức này có có lợi khi training set chưa có đầy đủ, các phần tử của training set được đưa vào mạng theo thời gian thực, do đó nó còn có tên online gradient descent.

Điều này có thể gây nhiễu, làm ta đi xa khỏi điểm cực tiểu khi gặp những example khác biệt lớn so với phần còn lại. Vì vậy, người ta đưa ra phương thức huấn luyện mới, huấn luyện theo lô.

II.3.4.2. Huấn luyện theo lô

Với phương thức huấn luyện này, mỗi phần tử thuộc tập hợp P gồm Q example được đưa vào mạng neuron, ta tính toán được gradient của weight và bias, cộng dồn chúng lại và cập nhật một lần duy nhất. Điều này có thể khiến cho việc hội tụ về điểm cực tiểu chậm lại cũng như mất nhiều thời gian hơn, nhưng bù lại ta có được sự ổn định. Ta có code giả như sau:

```
for each epoch in numEpoch
    shuffle training set
    for each batch in numBatch
        for each example in batch
            calculate gradient
            accumulate gradient
    update weight, bias
```

Bảng II.1: Phương thức huấn luyện theo lô

Nếu P là training set ta có phương thức huấn luyện theo lô đầy đủ (Batch gradient descent). Ngược lại, nếu P có nhiều hơn một example ta có phương thức mini-bactch gradient descent. Đây là cũng là phương thức được sử dụng trong đồ án này.

II.4. Mạng neuron chồng chập

Mạng neuron chồng chập (Convolutional Neuron Network - CNN) hiện nay là một trong những phương pháp thông dụng nhất trong nhận diện hình ảnh (Image recognition), liên quan mật thiết với thị giác máy tính (Computer Vision) khởi nguồn từ những năm 1980 khi Kunihiko Fukushima đề xuất một cấu trúc mạng neuron mới lấy cảm hứng từ cơ quan thị giác của mèo và khỉ.

Những thuật toán ở thời điểm đó có thể nhận diện một số cấu trúc của hình ảnh tuy nhiên khi chúng được biến đổi như bị dịch chuyển ra cạnh của ảnh, bị lật ngược hoặc bị che khuất thì độ chính xác giảm đi nhiều. Cấu trúc được đề xuất bởi Fukushima có thể tổng quát hóa hình ảnh cũng như loại bỏ những vật thể thừa, tránh được tình trạng over-fitting của mạng neuron truyền thống khi tính toán rất tốt những ảnh mà nó đã học nhưng thể hiện rất tệ đối với những ảnh chúng chưa từng gặp bằng cách giảm thiểu số lượng tham số của mạng neuron.

Năm 1998, LeCun và cộng sự xây dựng mạng neuron chồng chập LeNet-5 nhằm nhận dạng chữ số viết tay với kích thước ảnh 32x32. Tuy nhiên vì sự hạn chế của phần cứng máy tính lúc đó nên không thể phát triển cho ảnh có độ phân giải cao hơn.

Đến năm 2005, tại Hội nghị quốc tế về Phân tích và Nhận dạng tài liệu ICDAR, Dave Steinkraus và cộng sự đề xuất giải pháp sử dụng GPU cho các giải thuật học máy và đã được áp dụng thành công trên các cơ sở dữ liệu ảnh lớn như MNIST, ImageNet bởi Ciresan và cộng sự vào năm 2012.

II.4.1. Tích chập

Tích chập (Convolution) của hai hàm **f** và **g** liên tục là một biến đổi tích phân, trả về hàm thứ ba được định nghĩa bởi:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(x)\,g(t - x)\,dx \qquad (1.45)$$

Nếu **f** và **g** là hai hàm rời rạc, nhận các biến số nguyên ta có:

$$(f * g)[n] = \sum_{k=-\infty}^{+\infty} f[x]\,g[n - x] \qquad (1.46)$$

Mở rộng ra với hàm hai biến, ta có định nghĩa *tích chập 2 chiều*:

$$(f * g)[m,n] = \sum_{x=-\infty}^{+\infty} \sum_{y=-\infty}^{+\infty} f[x,y]\,g[m - x,n - y] \qquad (1.47)$$

Tích chập hai chiều thường được sử dụng trong xử lý hình ảnh để làm mịn, làm mờ ảnh hoặc nhận diện cạnh. Khi đó, hàm **f** là ảnh cần xử lý cho dưới dạng ma trận vuông **I** kích thước **W x W** và **g** gọi là *kernel* hay *filter* thường có dạng ma trận vuông **K** kích thước *F x F* với **F** là số nguyên lẻ, được quét trên ảnh theo chiều từ trái sang phải, từ trên xuống dưới.

Khi áp dụng tích chập 2 chiều, ta cần thêm hai tham số:

1. **S - stride**: khoảng cách giữa hai tâm kernel trong một lần dịch chuyển.
2. **P - zero-padding**: độ rộng của viền các số 0 ở quanh ảnh, nhằm điều chỉnh kích thước của ảnh khi tính tích chập.

Khi đó, kết quả trả về là một ma trận **R** có kích thước mỗi chiều là:

$$\frac{W - F + 2P}{S} + 1 \qquad (1.48)$$

Với mỗi phần tử của ma trận được tính như sau:

$$\mathbf{R}[i,j] = \sum_{u=-\lfloor F/2 \rfloor}^{\lfloor F/2 \rfloor} \sum_{v=-\lfloor F/2 \rfloor}^{\lfloor F/2 \rfloor} \mathbf{I}[i-u, j-v]\mathbf{K}[u,v] \qquad (1.49)$$

Trong đồ án này, ta sẽ sử dụng các tham số **F** = 5, **S** = 1 và **P** = 0**.**

II.4.2. Các layer trong mạng neuron chồng chập

II.4.2.1. Convolution layer

Khác với layer kinh điển, mỗi neuron của convolution layer chỉ liên kết với một số nhất định các input thay vì tất cả input. Ngoài ra, weight được dùng chung cho các neuron thuộc cùng một *độ sâu* (Parameter sharing), điều này làm giảm đáng kể số lượng weight của mạng neuron, do đó làm giảm thời gian tính toán, không gian bộ nhớ cũng như tránh "học vẹt" (overfitting).

Trong convolution layer[5], các neuron được xếp theo ma trận 3 chiều $\mathbf{W_2}$ x $\mathbf{W_2}$ x $\mathbf{D_2}$[6]***, nhận input là*** một dãy gồm $\mathbf{D_1}$ ma trận vuông kích thước $\mathbf{W_1}$ ***x*** $\mathbf{W_1}$ (gọi là một *lát cắt*), hay ta có thể coi là ma trận 3 chiều kích thước $\mathbf{W_1}$ ***x*** $\mathbf{W_1}$ ***x*** $\mathbf{D_1}$**.** Chiều thứ ba được gọi là độ sâu hay *channel*.

Các neuron trong một lát cắt nhận duy nhất một ma trận 3 chiều kích thước **F** ***x*** **F** ***x*** $\mathbf{D_1}$***, (cùng độ sâu với input) gọi là*** **filter** ***của lát cắt đó, làm weight chung như vậy ta có một ma trận*** *K* ***filter 4 chiều kích thước*** **F x F x** $\mathbf{D_1}$ **x** $\mathbf{D_2}$***. Khi đó, net input của neuron ở hàng*** **i***, cột* **j***, độ sâu* **k** ***là tổng các tích chập 2 chiều giữa input và filter theo lắt cắt tương ứng cùng với bias của neuron đó:***

$$n[i,j,k] = \sum_{d=1}^{D_1} \sum_{u=-\lfloor F/2 \rfloor}^{\lfloor F/2 \rfloor} \sum_{v=\lfloor -F/2 \rfloor}^{\lfloor F/2 \rfloor} \mathbf{I}[i-u, j-v, d]\mathbf{K}[u,v,d,k] + b[k] \qquad (1.50)$$

Tóm lại, convolution layer:

- ***Nhận một input*** *I* ***kích thước*** $\mathbf{W_1}$ ***x*** $\mathbf{W_1}$ ***x*** $\mathbf{D_1}$
- ***Một bộ các tham số:***
  - ***Số filter*** **K**

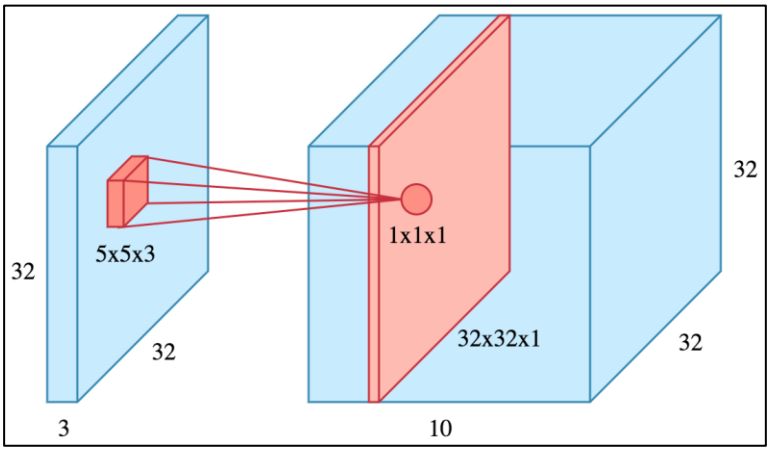
---

5 Gọi tắt là Conv layer

6 Trên thực tế, chiều rộng và chiều dài của khối neuron không nhất phải bằng nhau.

- ***Kích thước filter* F**
- ***Stride* S**
- ***Zero-padding* P**

- ***Trả về output kích thước* W_2 *x* W_2 *x* D_2 *với:***
  - $W_2 = \frac{W_1 - F + 2P}{S} + 1$
  - $D_2 = K$
- Mỗi filter ***có*** $F.F.D_1$ weight, toàn layer có $F.F.D_1.K$ weight, **K** bias
- Ma trận W_2 x W_2 thứ d của output, gọi là *feature map*, là kết quả khi áp dụng filter thứ d lên input.



Hình II.5: Minh họa Convolution layer

II.4.2.2. ReLU layer

Như đã nói ở phần trên, ReLU là activation function được định nghĩa bởi:

$$f(x) = x^+ = \max(x, 0) \quad (1.51)$$

Hàm này được sử dụng lần đầu tiên để huấn luyện mạng neuron vào năm 2011 đến năm 2018 trở thành activation function phổ biến nhất của mạng neuron, với nhiều ưu điểm vượt trội so với các hàm khác như: là hàm bất bão hòa dẫn đến ít khả năng gradient bị triệt tiêu, phản đối xứng và độ phức tạp tính toán thấp.

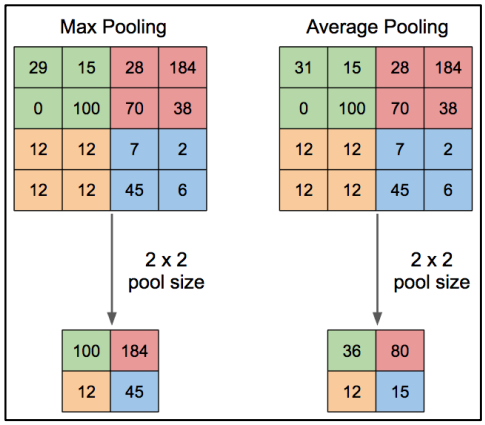
II.4.2.3. Pooling layer

Chức năng của Pooling layer[7] là làm giảm kích thước ảnh, dẫn đến giảm số lượng tham số và do đó kiểm soát được overfitting. Layer này áp dụng một filter (thường là 2x2) lên từng lát cắt của input, quét theo chiều từ trái sang phải, từ trên xuống dưới với stride **S**

7 Gọi tắt là Pool layer

(thường là 2) tương tự như convolution layer. Filter này là một hàm đa biến, mỗi biến là một phần tử trong vùng chiếu của filter. Nếu là hàm trung bình cộng - mean, ta có mean pooling hay average pooling; tương tự ta có max pooling.

$$n_{\text{mean}} = f\left(a,b,c,d\right) = \frac{a+b+c+d}{4} \qquad (1.52)$$
$$n_{\text{max}} = f\left(a,b,c,d\right) = \max\left(a,b,c,d\right)$$



Hình II.6: Ví dụ về max pooling và average pooling lên một lát cắt

II.4.2.4. Flatten layer

Layer này đơn giản, chỉ là chuyển ma trận 3 chiều thành 1 chiều nhằm làm input cho fully-connected layer.

II.4.3. Thuật toán truyền ngược trong CNN

Quá trình truyền ngược trong CNN vẫn cùng nguyên tắc như mạng neuron kinh điển, tức là tính toán đạo hàm riêng phần của hàm chi phí theo biến số cần cập nhật, sau đó áp dụng phương trình sau:

$$\mathbf{x}\left(k+1\right) = \mathbf{x}\left(k\right) - \alpha\frac{\partial\widehat{F}}{\partial\mathbf{x}} \qquad (1.53)$$

Ta sẽ lần lượt khảo sát qua các loại layer của CNN.

II.4.3.1. Convolution layer

Với input **I** kích thước $\mathbf{W_1}$ x $\mathbf{W_1}$ x $\mathbf{D_1}$ và $\mathbf{D_2}$ filter **K** kích thước **F** x **F** x $\mathbf{D_1}$ hay có thể coi như một filter **K** kích thước **F** x **F** x $\mathbf{D_1}$ x $\mathbf{D_2}$ có được ouput **N** kích thước[8] $\mathbf{(W_1 - F + 1)}$ x $\mathbf{(W_1 - F + 1)}$ x $\mathbf{D_1}$ hay $\mathbf{W_2}$ x $\mathbf{W_2}$ x $\mathbf{D_2}$ được tính như sau:

8 Ở đây ta chọn stride 1, zero-padding 0.

$$\mathbf{N}[i,j,k] = \sum_{d=1}^{D_1} \sum_{u=-[F/2]}^{[F/2]} \sum_{v=[-F/2]}^{[F/2]} \mathbf{I}[i-u, j-v, d]\mathbf{K}[u,v,d,k] + b[k], k = \overline{1...D_2} \quad (1.54)$$

Như vậy phần tử **K[u, v, d, k]** liên quan đến tất cả các phần tử ở lát cắt thứ k của output nên đạo hàm riêng phần của **K[u, v, d, k]** cần phải xét tới các phần tử này.

Giả sử ta đã tính được các đại lượng:

$$\frac{\partial \widehat{F}}{\partial \mathbf{N}[i,j,k]} \quad (1.55)$$

thì ta có:

$$\frac{\partial \widehat{F}}{\partial \mathbf{K}[u,v,d,k]} = \sum_{i=1}^{W_2}\sum_{j=1}^{W_2} \frac{\partial \widehat{F}}{\partial \mathbf{N}[i,j,k]} \frac{\partial \mathbf{N}[i,j,k]}{\partial \mathbf{K}[u,v,d,k]} \quad (1.56)$$

$$= \sum_{i=1}^{W_2}\sum_{j=1}^{W_2} \frac{\partial \widehat{F}}{\partial \mathbf{N}[i,j,k]} \mathbf{I}[i-u, j-v, d]$$

Đặt:

$$\frac{\partial \widehat{F}}{\partial \mathbf{N}[i,j,k]} = \dot{\mathbf{N}}_k[i,j] \quad (1.57)$$

và $$\mathbf{I}[i-u, j-v, d] = \mathbf{I}_d[i-u, j-v]$$

Nếu ta xoay ma trận $\mathbf{I_d}$ 180 độ thành ma trận rot $\mathbf{I}_d$, tức là

$$\mathbf{I}_d[i-u, j-v] = \text{rot}\,\mathbf{I}_d[u-i, v-j] \quad (1.58)$$

thì ta có:

$$\frac{\partial \widehat{F}}{\partial \mathbf{K}[u,v,d,k]} = \sum_{i=1}^{W_2}\sum_{j=1}^{W_2} \text{rot}\,\mathbf{I}_d[u-i, v-j]\dot{\mathbf{N}}_k[i,j] \quad (1.59)$$

Như vậy đạo hàm riêng phần của hàm chi phí ứng với ma trận **K** ở chiều thứ 3 và 4 lần lượt là d và k chính là:

$$\frac{\partial \widehat{F}}{\partial \mathbf{K}_{d,k}} = \text{rot}\,\mathbf{I}_d * \dot{\mathbf{N}}_k \quad (1.60)$$

Tương tự, ta tính được:

$$\frac{\partial \widehat{F}}{\partial b[k]} = \sum_{i=1}^{W_2}\sum_{j=1}^{W_2} \frac{\partial \widehat{F}}{\partial \mathbf{N}[i,j,k]} \frac{\partial \mathbf{N}[i,j,k]}{\partial b[k]} = \sum_{i=1}^{W_2}\sum_{j=1}^{W_2} \dot{N}_k[i,j] M_k[i,j] \quad (1.61)$$

với:

$$M_k\left[i,j\right]=\sum_{d=1}^{D_1}\sum_{u=-\left[F/2\right]}^{\left[F/2\right]}\sum_{v=\left[-F/2\right]}^{\left[F/2\right]}\mathbf{I}\left[i-u,j-v,d\right]\mathbf{K}\left[u,v,d,k\right] \quad (1.62)$$

II.4.3.2. ReLU layer

Layer này được tính toán đơn giản hơn:

$$\frac{\partial\widehat{F}}{\partial x}=\frac{\partial\widehat{F}}{\partial\mathrm{ReLU}\left(x\right)}\frac{\partial\mathrm{ReLU}\left(x\right)}{\partial x}=\frac{\partial\widehat{F}}{\partial\mathrm{ReLU}\left(x\right)}\frac{\partial\max\left(x,0\right)}{\partial x}=\frac{\partial\widehat{F}}{\partial\mathrm{ReLU}\left(x\right)}\Lambda\left(x\geq 0\right) \quad (1.63)$$

trong đó toán tử logic Λ(x) trả về 1 nếu mệnh đề x đúng, ngược lại trả về 0.

II.4.3.3. Pooling layer

Với mean pooling ta có:

$$\frac{\partial\widehat{F}}{\partial a}=\frac{\partial\widehat{F}}{\partial f_{\mathrm{mean}}\left(a,b,c,d\right)}\frac{\partial f_{\mathrm{mean}}\left(a,b,c,d\right)}{\partial a}=\frac{\partial\widehat{F}}{\partial f_{\mathrm{mean}}\left(a,b,c,d\right)}\frac{\partial\left(\frac{a+b+c+d}{4}\right)}{\partial a}$$
$$\frac{\partial\widehat{F}}{\partial a}=\frac{1}{4}\frac{\partial\widehat{F}}{\partial f_{\mathrm{mean}}\left(a,b,c,d\right)} \quad (1.64)$$

Tương tự với max pooling:

$$\frac{\partial\widehat{F}}{\partial a}=\frac{\partial\widehat{F}}{\partial f_{\max}\left(a,b,c,d\right)}\frac{\partial f_{\max}\left(a,b,c,d\right)}{\partial a}=\frac{\partial\widehat{F}}{\partial f_{\max}\left(a,b,c,d\right)}\frac{\partial\max\left(a,b,c,d\right)}{\partial a}$$
$$\frac{\partial\widehat{F}}{\partial a}=\frac{\partial\widehat{F}}{\partial f_{\max}\left(a,b,c,d\right)}\Lambda\left(a=\max\left(a,b,c,d\right)\right) \quad (1.65)$$

II.4.3.4. Flatten layer

Đây là layer nằm liền trước FC layer. Thuật toán truyền ngược ở layer này chỉ đơn giản là chuyển sai số từ tensor-1D sang tensor-3D để truyền tiếp đến các conv layer và pooling layer.

CHƯƠNG III: PHÂN TÍCH THIẾT KẾ HỆ THỐNG

III.1. Cấu trúc mạng neuron

Cấu trúc thông thường của mạng neuron được cho như sau:

Input -> [[Conv -> ReLU]*N -> Pool?]*M -> [FC -> ReLU]*K -> FC

với dấu "*" thể hiện số lần lặp, "?" thể hiện có hoặc không lớp Pooling. Các hệ số thường là $0 \leq N \leq 3$, $M \geq 0$, $0 \leq K < 3$.

Sau đây là ví dụ mạng neuron gồm 12 layer.

Mỗi ảnh được đưa vào mạng neuron và tính toán dưới kiểu dữ liệu array với shape [252, 252, 3] trong thư viện Numpy. Output cuối cùng của mạng là một array với shape [2,1].

STT	Layer	Input	Output
1	Conv1 - 8 kernel	252 x 252 x 3	248 x 248 x 8
2	Max Pool 1	248 x 248 x 8	124 x 124 x 8
3	Conv2 - 16 kernel	124 x 124 x 8	120 x 120 x 16
4	Max Pool 2	120 x 120 x 16	60 x 60 x 16
5	Conv3 - 24 kernel	60 x 60 x 16	56 x 56 x 24
6	Max Pool 3	56 x 56 x 24	28 x 28 x 24
8	Conv3 - 32 kernel	28 x 28 x 24	24 x 24 x 32
9	Max Pool 4	24 x 24 x 32	12 x 12 x 32
10	Flatten	12 x 12 x 32	4608 x 1
11	FC1 - ReLU	4608 x 1	2048 x 1
12	FC2 - sigmoid	2048 x 1	2 x 1

Bảng III.1: Cấu trúc mạng neuron

Các conv layer được cài đặt với các tham số cố định S = 1, P = 0, F = 5 với số filter tăng dần: 8, 16, 24, 32; theo sau bởi một hàm ReLU và xem như là activation function của layer đó.

Các max pooling layer được cài đặt ngay sau Conv - ReLu layer với vùng chiếu 2x2, stride 2.

Vectorize layer là một layer phụ với mục đích chuyển array shape [28, 28, 32] về array shape [4608,1] để đưa vào FC layer.

III.1.1. Khởi tạo

Các bias được khởi tạo bằng 0. Còn weight khởi tạo theo khuyến cáo của Xavier Glorot hoặc Yann LeCun. Ví dụ như theo phân phối chuẩn với trung bình bằng 0 và phương sai:

$$s = \sqrt{\frac{2}{fan_{in} + fan_{out}}} \quad (1.15)$$

Với fan_{in} và fan_{out} lần lượt là số input và output liên kết với weight đó. Ví dụ như Conv layer đầu tiên có $fan_{in} = 5.5.3 = 75$ và $fan_{out} = 8$.

III.1.2. Huấn luyện

Sử dụng phương pháp huấn luyện mini-batch với kích thước batch thường khoảng 25-50% tập training. Số epoch thường từ 10-30.

III.2. Cấu trúc chương trình

Chương trình bao gồm 5 lớp: Data, Function, Parameter, Layer, Network và một hàm main. Ngoài ra còn có các hàm phụ khác nhằm đơn giản hóa chương trình.

III.2.1. Lớp Data

Thực hiện việc xử lý dữ liệu của chương trình.

III.2.1.1. Thuộc tính

Chứa một thuộc tính duy nhất là kiểu dữ liệu tuple lưu trữ array mã hóa ảnh, với độ dài 2 hoặc 4 trước và sau khi chia cắt.

III.2.1.2. Phương thức

III.2.1.2.1. __init__(path, sizeImage)

Giai đoạn nhập dữ liệu được cài đặt trong hàm __init__ của lớp Data với các đối số path - chứa đường dẫn đến thư mục chứa ảnh, sizeImage - kích thước ảnh chuẩn hóa.

Tại thư mục chứa ảnh, sử dụng hàm os.listdir để khởi tạo mảng Images chứa tất cả tên file của thư mục.

Mỗi ảnh được đưa vào chương trình thông qua các bước sau:

1. Với mỗi phần tử image thuộc Images, sử dụng hàm os.path.join để tạo đường dẫn hệ thống pathImage đến file.
2. Sử dụng hàm cv2.imread để đọc ảnh từ đường dẫn pathImage, và lưu ảnh dưới dạng array của thư viện Numpy, với shape [W, H, 3] với **W**, **H** là kích thước thực của ảnh, mỗi phần tử là kiểu dữ liệu numpy.float32 thuộc đoạn [0,1].
3. Sử dụng hàm skimage.transform.resize với các tham số mặc định để chuẩn hóa ảnh với kích thước mỗi ảnh là sizeImage x sizeImage x 3, rồi chèn vào list trainImages.
4. Tìm trong chuỗi image chuỗi "cat" (hoặc "dog") để gắn nhãn cho ảnh dưới dạng array [0,1] (hoặc [1,0]) với shape [2,1], rồi chèn vào list trainLabels.

III.2.1.2.2. Standardize

Được cài đặt trong hàm standardize của lớp Data, qua đó ảnh được chuẩn hóa sao cho trong tập ảnh, tại mỗi vị trí của ảnh có trung bình là 0 và độ lệch chuẩn là 1, theo phương trình:

$$X = \frac{X - \overline{X}}{s} \quad (1.15)$$

với X là một phần tử của array mã hóa ảnh, $\bar{X}, s$ lần lượt là trung bình cộng và độ lệch chuẩn của tất cả các phần tử trong tập ảnh ở vị trí tương ứng.

III.2.1.2.3. Shuff

Được cài đặt trong hàm shuff của lớp, sử dụng hàm random.shuffle nhận hai mảng trainImages và trainLabels và trả về hai mảng tương ứng đã được xáo trộn.

III.2.1.2.4. Split

Nhằm phục vụ cho quá trình huấn luyện, hai mảng trainImages và trainLabels được chia thành hai phần là train và validate để huấn luyện và kiểm thử với tỉ lệ 4:1 thông qua hàm split của lớp.

III.2.2. Lớp Function

Là lớp tính toán activation function, gồm 3 hàm là: __init__, calculate, deviation. Hàm __init__ đơn giản chỉ là nhận input là một chuỗi và gán cho thuộc tính function.

III.2.2.1. Thuộc tính

Lớp này chỉ có một thuộc tính duy nhất là function lưu tên của function.

III.2.2.2. Phương thức

III.2.2.2.1. __init__

Gắn tên của function cho thuộc tính function.

III.2.2.2.2. Calculate

Nhận input là dữ liệu ảnh và trả về output là dữ liệu sau khi tính toán. Tùy theo activation function mà cách tính khác nhau. Sau đây là một số đoạn code được sử dụng:

Sigmoid	return 1/(1 + np.exp(-input))

ReLU	R = np.copy(input)
	`R[R < 0] = 0`
	`return R`

Vectorize	S, D = input.shape[0], input.shape[2]
	R = np.zeros((S*S, D), dtype=np.float32)
	for i in range(S*S):
	  for k in range(D):
	  R[i, k] = input[i % S, i//S, k]
	return R.reshape(-1, 1)

Bảng III.2: Code minh họa của phương thức calculate của lớp Function

Riêng layer pooling được tính toán phức tạp hơn, thay vì sử dùng nhiều vòng lặp for thì input được chuyển về kiểu dữ liệu ndarray của thư viện Numpy rồi sử dụng hàm numpy.lib.stride_tricks.as_stride để tăng tốc độ tính toán.

III.2.2.2.3. Deviation

Nhận input là 3 đối số N, A, D lần lượt là net input, kết quả sau khi qua hàm activation và sai số được truyền về từ thuật toán truyền ngược. Output của hàm là sai số của net input. Lệnh đầu tiên là R = np.copy(N) nhằm tránh ghi đè lên đối số N.

Sigmoid	return A * (1-A) * D
ReLU	R[R > 0] = 1 R[R <= 0] = 0 return R * D

Vectorize	s, d = A.shape[0], A.shape[-1]
	R = np.zeros((s, s, d), dtype=np.float32)
	for a in range(s):
	  for b in range(s):
	  for c in range(d):
	  R[a, b, c] = D[c*s*s + b*s + a, 0]
	return R

Mean pooling	R[...] = 1/4 return R * D
Max pooling	Z = zoom(A) R[Z != N] = 0 R[Z== N] = 1 return R * D

III.2.3. Lớp Parameter

Là lớp trừu tượng có nhiệm vụ chính là khởi tạo giá trị ban đầu và huấn luyện weight và bias của hai layer convolution và fully-connected thông qua hai hàm chính là initValue và learn, ngoài ra còn có hai hàm phụ là generateDistribution và getFans hỗ trợ cho hàm initValue.

III.2.3.1. Thuộc tính

III.2.3.1.1. Thuộc tính lớp

Lớp có các thuộc tính mặc định cho tất cả các đối tượng của nó là các hằng số cần thiết trong quá trình huấn luyện:

alpha = 0.001, decay = 0.9, epsilon = 10**(-8), beta1 = 0.9, beta2 = 0.99.

III.2.3.1.2. Thuộc tính đối tượng

Gồm có các thuộc tính:

- value: lưu giá trị của weight và bias
- D: lưu sai số được truyền về để huấn luyện
- initializer, optimizer: phương thức khởi tạo và huấn luyện.

Ngoài ra, lớp còn có các thuộc tính phụ tùy theo phương thức huấn luyện.

III.2.3.2. Phương thức

III.2.3.2.1. __init__(typeParameter, shapeParameter, optimizer, initializer):

Lớp được khởi tạo thông qua 4 đối số: typeParameter - nhận một trong hai giá trị "weight" hoặc "bias", shapeParameter - shape array lưu trữ tham số, optimizer - một trong các thuật toán tối ưu được đề cập đến trong mục **1.1.4**, initializer - phương thức khởi tạo weight.

Các bias được khởi tạo giá trị ban đầu đều bằng 0. Mặt khác, các weight được khởi tạo theo phân phối chuẩn hoặc phân phối đều trong khoảng [-s, s] với giá trị trung bình bằng 0. Với phương sai trong phân phối chuẩn hoặc sai số s trong phân phối đều tùy theo khuyến cáo đưa ra bởi các tác giả Yann LeCun, Xavier Glorot. Ví dụ Yann LeCun đề nghị sử dụng phân phối chuẩn với phương sai:

$$\sqrt{\frac{1}{n_{in}}} \tag{1.15}$$

Với n_{in} là số lượng input của filter layer đó chiếu đến.

III.2.3.2.2. initValue và generateDitribution

Là hai phương thức nhằm khởi tạo giá trị của weight và bias.

III.2.3.2.3. learn(g)

Tùy theo optimizer mà giá trị thay đổi của weight và bias khác nhau, xem chi tiết ở phần phụ lục.

III.2.4. Lớp Layer

Là lớp trừu tượng cho các loại layer dùng trong mạng neuron.

III.2.4.1. Thuộc tính

- typeLayer: là các kí tự kí hiệu cho loại Layer: C, P, V, F lần lượt là convolution, pooling, vectorize, fully-connected layer.
- activation: là object của lớp Function cụ thể hóa cho từng layer.
- learnable: là biến bool cho biết layer đó có thể huấn luyện được không, tương đương với convolution layer hoặc fully-connected layer.
- weight, bias: là object của lớp Parameter trong trường hợp learnable = true.

III.2.4.2. Phương thức

III.2.4.2.1. __init__(typeLayer, initializer = None, optimizer = None, shapeParameter = None, activation = None, alpha = None):

Khởi tạo object thông qua các đối số typeLayer, initializer, optimizer, shapeParameter, activation. Trừ typeLayer thì các đối số còn lại mặc định bằng None.

Các đối số initializer, optimizer, shapeParameter và activation được truyền tiếp cho lớp Parameter và lớp Function.

III.2.4.2.2. learn(DW, DB)

Gọi hàm learn của lớp Parameter.

III.2.4.2.3. calculate(input)

Ở đây chỉ đề cập đến convolution layer vì nó chiếm thời gian thực thi nhiều nhất trong các layer. Việc tính toán thực hiện trong một vòng for với mỗi lần lặp tính một channel của output thông qua lời gọi hàm conv3D.

Hàm conv3D thực thi một vòng lặp for với mỗi lần lặp tính tích chập 2 chiều giữa filter và input theo độ sâu tương ứng thông qua lời gọi hàm conv2D, sau đó cộng dồn chúng lại với nhau.

Hàm conv2D thực chất gọi đến hàm scipy.ndimage.convolve để tính toán tích chập 2 chiều, output của hàm này bảo toàn kích thước của input nên được cắt 4 biên để thu được output mong muốn.

III.2.5. Lớp Network

Là lớp quan trọng nhất của chương trình, có nhiệm vụ xây dựng và tính toán mạng neuron.

III.2.5.1. Thuộc tính

Thuộc tính chính là list layers chứa thông tin các layer trong mạng neuron. Ngoài ra còn các thuộc tính phụ cần thiết trong quá trình khởi tạo cũng như tính toán như L - số layer của mạng, optimizer và initializer - với ý nghĩa đã được đề cập đến ở phần trên.

III.2.5.2. Phương thức

III.2.5.2.1. add(typeLayer, shape = None, activation = "sigmoid", shapeInput = None")

Nhiệm vụ khởi tạo layer tương ứng và thêm vào list layers.

III.2.5.2.2. forward(data)

Thực hiện một vòng lặp for nhằm tính toán output của mạng neuron bằng cách gọi các hàm calculate của lớp Layer.

Kết quả trả về là tuple (N, A) chứa net input và output của từng layer trong mạng.

III.2.5.2.3. backward(N, A, input, target)

Nhận input là hai list N, A ở phương thức forward cũng như input và target output nhằm thực hiện thuật toán truyền ngược.

Đây là hàm có khối lượng tính toán và thời gian thực thi lâu nhất của mạng neuron, nhất là khi truyền ngược qua các convolution layer.

Kết quả trả về là tuple (e, DW, DB)lần lượt chứa kết quả của hàm chi phí, list chứa sai số của weight và bias của từng layer.

III.2.5.2.4. update(DW, DB, batchSize)

Gọi các hàm learn của lớp layer sau mỗi batchSize example.

III.2.5.2.5. calculate(maxLoop, batchSize, data)

Là hàm thực thi chính của chương trình với phương thức mini-batch.

CHƯƠNG IV: TRIỂN KHAI VÀ ĐÁNH GIÁ KẾT QUẢ

IV.1. Triển khai

Chương trình được viết dưới ngôn ngữ Python 3.6.4, cài đặt và thực thi trên IDE Visual Studio Code version 1.22.1 trên máy tính DELL Inspiron N4448, Processor Intel Core i5-5200U @2.20GHz (4CPUs). Ngoài ra, chương trình còn sử dụng trình biên dịch Numba, nhằm tăng tốc độ tính toán bằng cách chuyển đổi các tính toán trên Numpy array thành mã máy dưới nền tảng LLVM complier.

Dữ liệu được lấy từ Kaggle theo link *https://www.kaggle.com/c/dogs-vs-cats/data*.

IV.2. Kết quả

Sau đây ta khảo sét sự ảnh hưởng của các initializer, optimizer cũng như một số thành phần khác của mạng neuron nhằm tìm ra cấu trúc tối ưu cho CNN.

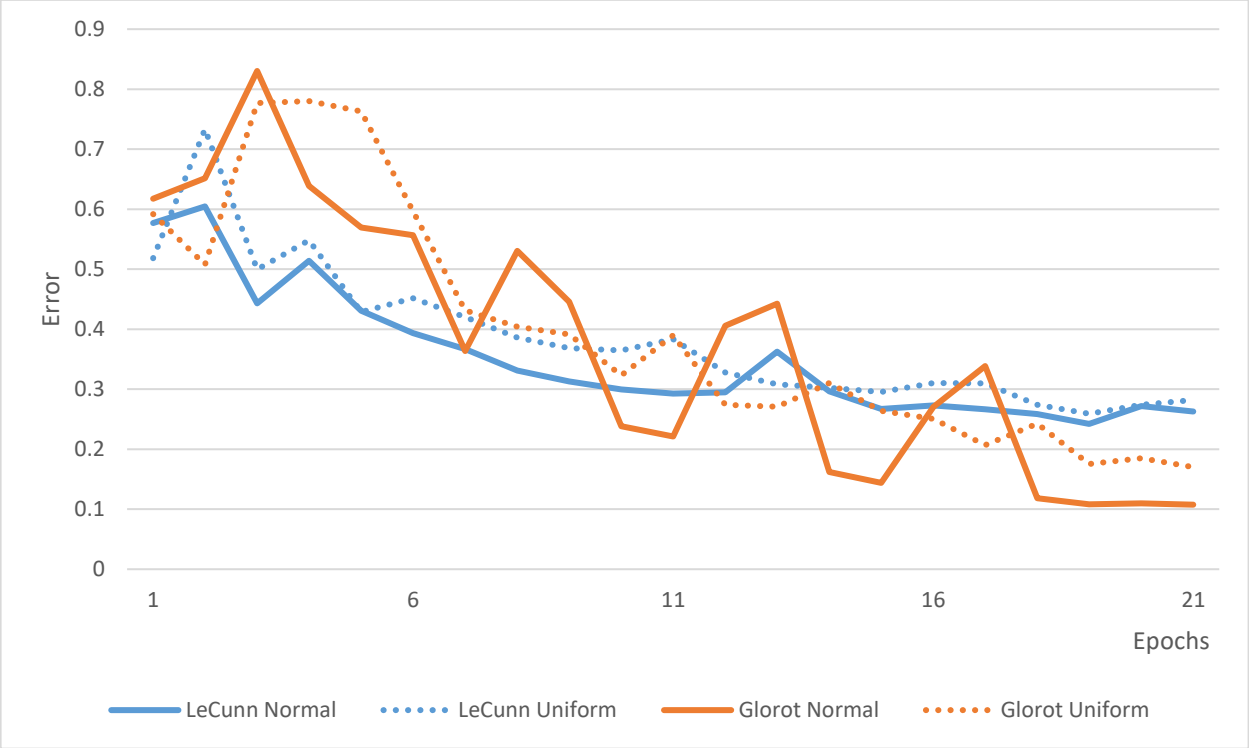
IV.2.1. Ảnh hưởng của initializer

Các initializer được cài đặt và chạy theo các khuyến cáo của LeCunn và Glorot Xavier nhằm lựa chọn initializer tối ưu. Bảng mô tả các initilizer được cho ở bảng sau:

	Phân phối	**Tính chất**
LeCunn 1	Normal	$m = 0, s = \sqrt{\frac{1}{\text{fan}_{\text{in}}}}$
LeCunn 2	Uniform	$\left[-\sqrt{\frac{3}{\text{fan}_{\text{in}}}}, \sqrt{\frac{3}{\text{fan}_{\text{in}}}}\right]$
Glorot 1	Normal	$m = 0, s = \sqrt{\frac{2}{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}}$
Glorot 2	Uniform	$\left[-\sqrt{\frac{6}{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}}, \sqrt{\frac{6}{\text{fan}_{\text{in}} + \text{fan}_{\text{out}}}}\right]$

Bảng IV.1: Tính chất của các initializer

Sai số của từng initializer trong quá trình huấn luyện được trình bày trong đồ thị sau:



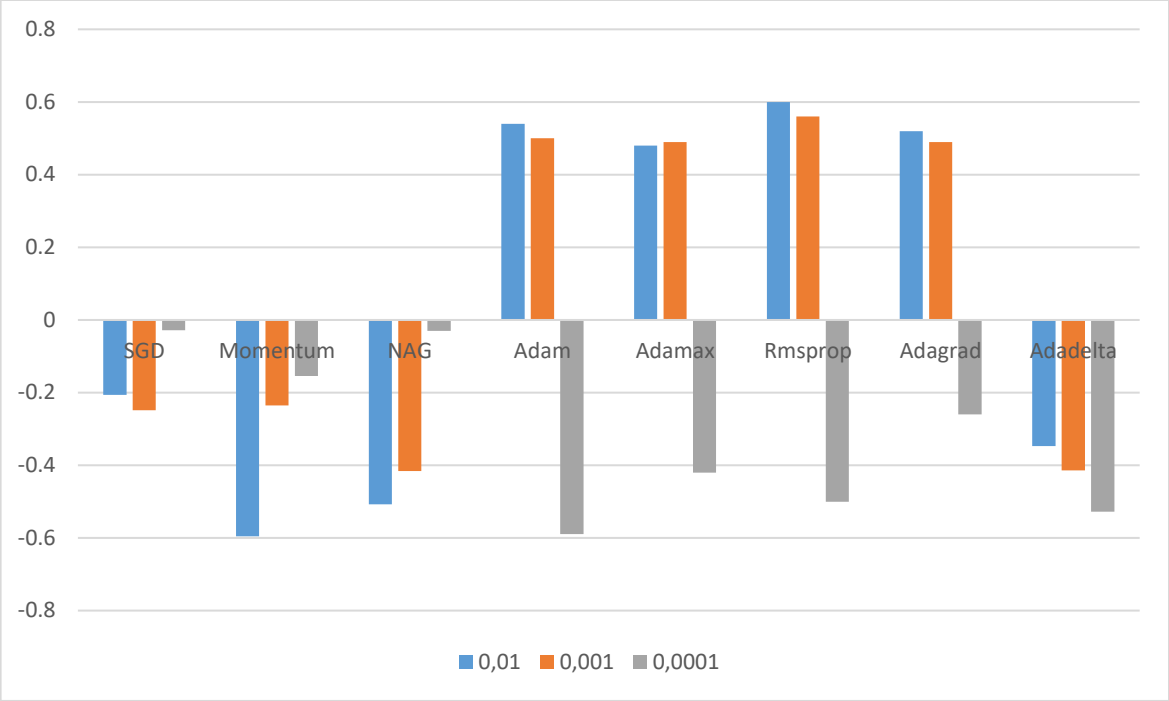
Hình IV.1: Ảnh hưởng của initialier

Nhìn chung, hai initializer của LeCunn có độ giảm ổn định hơn tuy nhiên về hiệu quả học thì initilizer của Glorot cao hơn hẳn với độ giảm trung bình cao hơn 70% (0,466 so với 0,274).

Trong từng cặp initializer thì mức độ cải thiện error của phân phối chuẩn nhìn chung cao hơn phân phối đều. Initializer tối ưu ở đây ta chọn Glorot Uniform về cả hai mặt ổn định và hiệu quả học.

IV.2.2. Ảnh hưởng của optimizer

Vì các optimizer được cài đặt nhạy với hệ số học nên ta cần khảo sát hiệu quả học theo các hệ số học khác nhau ở đây ta chọn ba hệ số học: 0,01; 0,001; 0,0001. Kết quả được cho ở đồ thị sau:



Hình IV.2: Ảnh hưởng của optimizer và hệ số học

Qua đồ thị trên, ta có thể optimizer thành ba nhóm: nhóm 1 gồm SGD, momentum và NAG; nhóm 2 gồm Adam, Adamax, Rmsprop, Adagrad và nhóm 3 chỉ có Adadelta.

Các optimizer thuộc nhóm 1 có hiệu quả học giảm dần theo độ giảm của hệ số học, đây là đặc trưng của các thuật toán optimizer kinh điển.

Các optimizer thuộc nhóm 2 là các optimizer có hệ số học thay đổi qua mỗi lần update và khác nhau cho từng tham số của mạng neuron. Nhóm này rất nhạy với hệ số học, chỉ hiệu quả khi hệ số học khá thấp: 0,0001.

Adadelta được thiết kế để vượt qua được sự ảnh hưởng của hệ số học ở nhóm 2, ngoài ra ta có thể thấy rằng hiệu quả học tăng dần khi ta giảm hệ số học.

IV.2.3. Ảnh hưởng của số layer

IV.2.3.1. Ảnh hưởng của số convolution layer

Ta xây dựng ba mạng neuron A, B và C với cấu hình và kết quả như sau:

	A	**B**	**C**
Cấu trúc			
Số conv layer	1	2	3
Số pool layer	1	2	3
Số FC layer	1	1	1

Tổng số layer	4	6	8
Số lượng tham số	3,2 M	1,1 M	0,1 M
Kết quả			
Thời gian forward	0,01	0,02	0,02
Thời gian backward	0,08	0,04	0,06
Số epoch để Acc training trên 90%	5	6	21
Độ chính xác	**62,5%**	56,25%	43,75%

Bảng IV.2: Ảnh hưởng của số layer convolution

Qua bảng trên ta thấy rằng sự tăng độ sâu của mạng neuron làm giảm đi độ chính xác, có lẽ là do số lượng tham số ít dẫn đến mức độ mô tả thông tin kém đi.

IV.2.3.2. Ảnh hưởng của số FC layer

Tương tự, ta xây dựng ba mạng neuron C, D và E sau đây để so sánh:

	C	**D**	**E**
Cấu trúc			
Số conv layer	1	1	1
Số pool layer	1	1	1
Số FC layer	2	3	4
Tổng số layer	5	6	7
Số lượng tham số	6,5 M	8,7 M	9,2 M
Kết quả			
Thời gian forward	0,01	0,01	0,01
Thời gian backward	0,05	0,07	0,08
Số epoch để Acc training trên 90%	3	4	2
Độ chính xác	**62,5%**	43,75%	62,25%

Bảng IV.3: Ảnh hưởng của số layer fully-connected

Qua bảng trên một lần nữa ta thấy rằng việc tăng độ sâu của mạng neuron không làm gia tăng độ chính xác trong khi thời gian thực thi kém đi.

IV.2.4. Ảnh hưởng của kích thước kernel convolution layer

Ta xây dựng ba mạng neuron với kích thước các kernel lần lượt là 3x3, 5x5 va 7x7 để so sánh:

Kernel size	**3x3**	**5x5**	**7x7**
Số lượng tham số	5,6 M	5,2 M	4,8 M
Số epoch để Acc training trên 90%	4	3	4
Độ chính xác	**62,5%**	57,5%	57,5%

Bảng IV.4:Ảnh hưởng của kích thước kernel

Kernel size, nhìn chung, không ảnh hưởng nhiều đến độ chính xác của bài toán. Để thống nhất, ta chọn kernel size 5x5.

IV.2.5. So sánh với mạng neuron kinh điển

Để so sánh ta xây dựng hai mạng neuron NN và CNN với số layer và kích thước ảnh tương đương nhau, được mô tả trong bảng sau:

	NN	**CNN**
Cấu trúc		
Layer 0	Flatten	Conv - 8x5x5 - ReLU
Layer 1	FC - 2048 - ReLU	Max pooling
Layer 2	FC - 1024 - ReLU	Conv - 16x5x5 - ReLU
Layer 3	FC - 512 - ReLU	Max pooling
Layer 4	FC - 256 - ReLU	Flatten
Layer 5	FC - 128 - ReLU	FC - 2048 - ReLU
Layer 6	FC - 2 - sigmoid	FC - 2 - sigmoid
Số lượng tham số	19,4 M	3,2 M
Kết quả		
Thời gian forward	0,02	0,02
Thời gian backward	0,14	0,05
Số epoch để độ chính xác tập training trên 90%	3	6
Độ chính xác	53,5%	**60%**

Bảng IV.5: So sánh hiệu quả học của hai mạng NN và CNN

Với số layer tương đương nhau, nhưng NN cần số lượng tham số nhiều hơn (19,4 M so với 3,2 M, dù ta đã giảm dần số neuron theo độ sâu), thời gian thực thi lâu hơn (0,16 giây so với 0,07 giây) nhưng độ chính xác thấp hơn hẳn (53,5% - 60%).

IV.2.6. Nghiệm thu kết quả

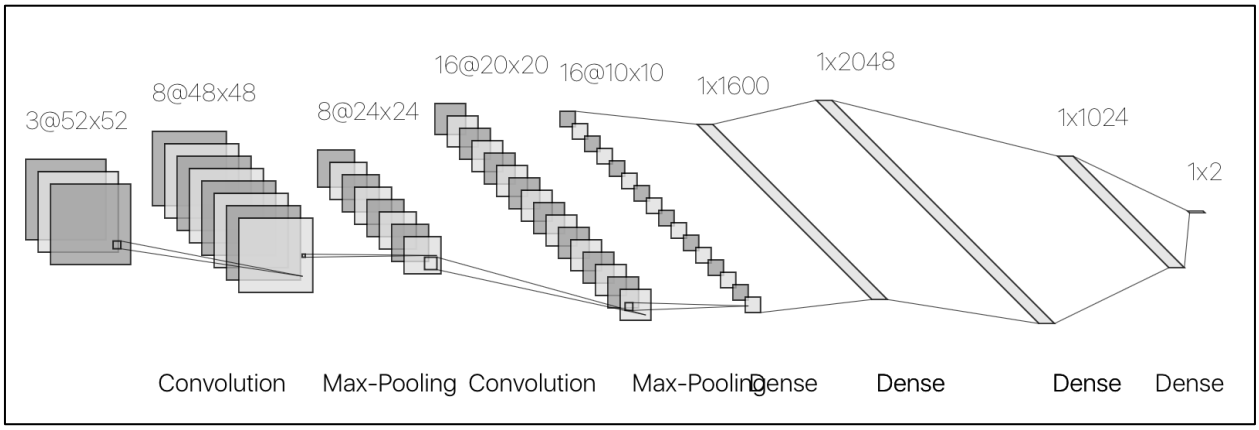
Qua kết quả ở trên, ta xây dựng mạng neuron chồng chập gồm 2 conv layer, theo sau mỗi layer là ReLU layer và max pooling layer, rồi đến flatten layer và 3 FC layer. Optimizer

và initializer lựa chọn là Adadelta và Glorot Uniform, hệ số học 0,01. Kích thước kernel là 5x5. Số lượng neuron các lớp fully-connected là 2048, 1024 và 2.

Dữ liệu ảnh là 15000 ảnh chó và mèo tỉ lệ 1:1, được chia thành hai tập training và validate với số lượng lần lượt là 12000 và 3000. Kích thước ảnh 52x52.

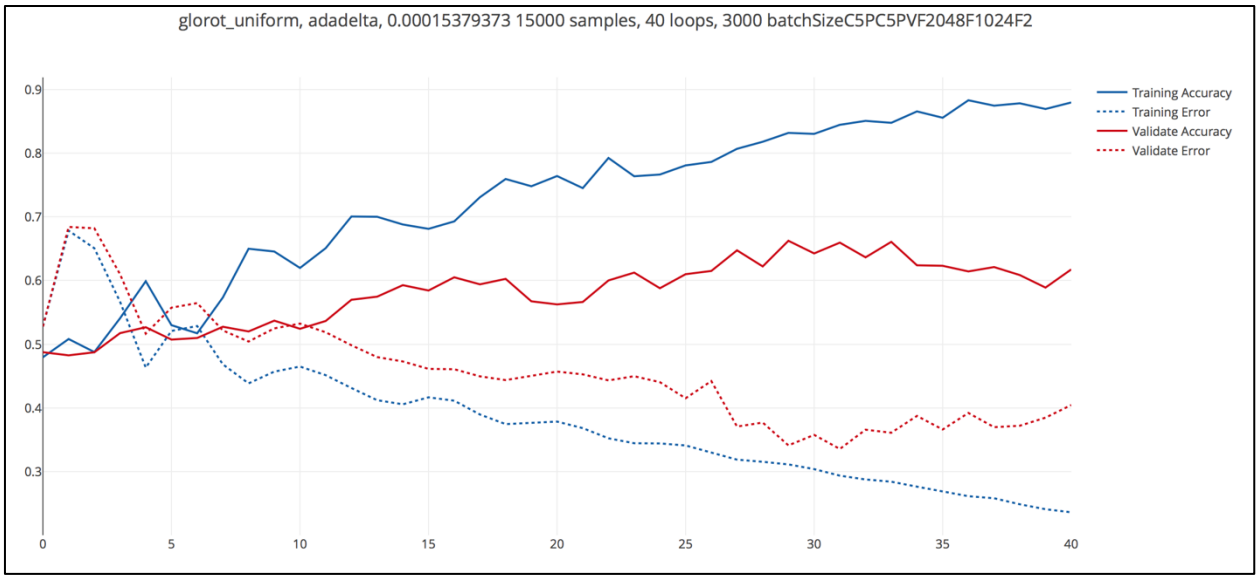
Tổng số lượng tham số của toàn mạng neuron là khoảng 5,4 triệu, tính trung bình mỗi pixel của ảnh được mô tả bởi 663 tham số. Thời gian thực thi chiều forward và backward lần lượt là 0,02 và 0,07 giây mỗi ảnh. Tổng thời gian thực thi là 15 giờ 20 phút.

Phương thức huấn luyện là mini-batch với kích thước mỗi batch là 3000, số epoch 40.



Hình IV.3:Minh học cấu trúc CNN

Sau đây là kết quả thu được:



Hình IV.4:Kết quả

Sau 40 epoch, độ chính xác và error của tập training là xấp xỉ 90% và 0,2. Độ chính xác trung bình của tập training và validate tăng thêm sau mỗi epoch lần lượt là 0,95% và 0,3%. Độ chính xác tập validate cao nhất ở epoch thứ 30 với tỉ lệ xấp xỉ 65%, các epoch sau đã rơi vào tình trạng overfiting. Tỉ lệ 65% là tương đối thấp so với các nghiên cứu tương đương, có thể giải thích do số lượng tham số còn thấp cũng như cấu trúc của mạng neuron còn khá đơn giản, cần phát triển thêm.

CHƯƠNG V: KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

V.1. Kết luận

Mạng neuron chồng chập là một cấu trúc mới phát triển lên từ mạng neuron kinh điển với một số bước xử lí ban đầu thông qua các filter do đó giảm thời gian thực thi, không gian bộ nhớ cũng như cải thiện độ chính xác cho các bài toán nhận dạng hình ảnh. Qua đồ án này, ta rút ra một số kết luận sau:

1. Initializer ảnh hưởng nhiều đến hiệu quả huấn luyện, ưu tiên chọn phân phối đều theo khuyến cáo của Glorot.
2. Một số optimizer rất nhạy với hệ số học, một số khác có hiệu quả học giảm dần theo chiều giảm của hệ số học. Adadelta có tính ổn định tương đối với sự thay đổi của hệ số học.
3. Độ sâu của neuron có thể ảnh hưởng tiêu cực đến độ chính xác, nhưng cũng không nên xây dựng mạng neuron chỉ với vài layer vì sẽ khó có thể mô tả được dữ liệu.
4. Độ chính xác của CNN nhìn chung cao hơn với NN tương đương, nhưng có thời gian thực thi cũng như không gian bộ nhớ ít hơn hẳn.

V.2. Hướng phát triển

Vì thời gian thực hiện không cho phép nên đồ án còn một số hạn chế. Tuy nhiên, có thể phát triển thêm theo nhiều hướng:

1. Cải tiến convolution layer: bằng các cấu trúc mới như tiled convolution (Quoc V.Le *et at.*), dilated convolution (F. Yu, V. Koltun) hay có thể thay thế activation function của conv layer bằng một multilayer perceptron, còn được gọi là Network in Network được đề xuất bởi Lin *et at.*
2. Cải tiến pooling layer: bằng các hàm khác như L_p - norm, hay kết hợp giữa mean và max trong mixed pooling,…
3. Thay đổi hàm ReLU bằng các biến thể của nó như Leaky ReLU, Parametric ReLU, Randomized ReLU,…
4. Chuẩn hóa để tránh over-fitting: thông qua chuẩn hóa L_p - norm hàm chi phí, dropout.
5. Mở rộng bài toán thành nhận dạng nhiều đối tượng hơn cũng như phát triển các cấu trúc mạng neuron khác nhằm phục vụ cho các tác vụ khác như khoanh vùng đối tượng trong ảnh hay cho xử lí giọng nói,…

TÀI LIỆU THAM KHẢO

[1] K. Andrej and J. Justin, "CS231n: Convolutional Neural Networks for Visual Recognition," 30-May-2018. .

[2] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesus, Neural network design, 2nd edition. Wrocław: Amazon Fulfillment Poland Sp. z o.o.

[3] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," arXiv:1603.07285 [cs, stat], Mar. 2016.

[4] M. D. Zeiler, "ADADELTA: An Adaptive Learning Rate Method," arXiv:1212.5701 [cs], Dec. 2012.

[5] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," arXiv:1412.6980 [cs], Dec. 2014.

[6] J. C. Duchi, E. Hazan, and Y. Singer, Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, vol. 12. 2011.

[7] E. K. P. Chong and S. H. Żak, An introduction to optimization, 4. ed. Hoboken, NJ: Wiley, 2013.

[8] T. Kessler, G. Dorian, and J. H. Mack, "Application of a Rectified Linear Unit (ReLU) Based Artificial Neural Network to Cetane Number Predictions," 2017, p. V001T02A006.

[9] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," arXiv:1502.01852 [cs], Feb. 2015.

[10] L. V. Fausett, Fundamentals of neural networks: architectures, algorithms, and applications. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[11] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," 2009, pp. 248–255.

[13] C. Cao et al., "Look and Think Twice: Capturing Top-Down Visual Attention with Feedback Convolutional Neural Networks," 2015, pp. 2956–2964.

[14] D. Cireşan, U. Meier, and J. Schmidhuber, "Multi-column Deep Neural Networks for Image Classification," arXiv:1202.2745 [cs], Feb. 2012.

[15] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," Biological Cybernetics, vol. 36, no. 4, pp. 193–202, Apr. 1980.

[16] S. S. Haykin and S. S. Haykin, Neural networks and learning machines, 3rd ed. New York: Prentice Hall, 2009.

[17] K.-L. Du and M. N. S. Swamy, Neural Networks and Statistical Learning. London: Springer London, 2014.

[18] C. M. Bishop, Neural networks for pattern recognition. Oxford : New York: Clarendon Press ; Oxford University Press, 1995.

[19] R. Rojas, Neural networks: a systematic introduction. 1996.
[20] G. Orr and K.-R. Müller, Eds., Neural networks: tricks of the trade. Berlin ; New York: Springer, 1998.
[21] N. Qian, "On the momentum term in gradient descent learning algorithms," Neural Networks, vol. 12, no. 1, pp. 145–151, Jan. 1999.
[22] H. Wang and B. Raj, "On the Origin of Deep Learning," arXiv:1702.07800 [cs, stat], Feb. 2017.
[23] W. Forst and D. Hoffmann, Optimization—Theory and Practice. New York, NY: Springer New York, 2010.
[24] J. Gu et al., "Recent Advances in Convolutional Neural Networks," arXiv:1512.07108 [cs], Dec. 2015.
[25] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," J. Physiol. (Lond.), vol. 195, no. 1, pp. 215–243, Mar. 1968.
[26] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," J. Physiol. (Lond.), vol. 160, pp. 106–154, Jan. 1962.
[27] D. Steinkraus, I. Buck, and P. Y. Simard, "Using GPUs for machine learning algorithms," 2005, pp. 1115-1120 Vol. 2.

PHỤ LỤC

```
'''
-------------------------------------------------------------------------
  Module     :     <Simple (Convolutional) Neural Network Classification>
  Author     :     <Le Trong Hieu>
  Date       :     <2018-03-25>
  University :     <Da Nang University of Technology>
-------------------------------------------------------------------------
'''


import matplotlib.pyplot as plt
import numpy as np
import random
import scipy
from numpy.lib.stride_tricks import as_strided
from PIL import Image, ImageOps
from skimage import transform
import os
import cv2
import time
from scipy import ndimage
import sys
import numba as nb
from numba import jit, float32, njit, prange, stencil
from tqdm import tqdm
from plotly import __version__
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
from plotly.graph_objs import Layout, Figure, Scatter
import plotly as py
from scipy.signal import convolve2d
import pickle

#   pylint: disable = E1101

listInit = ["lecun_normal", "lecun_uniform", "glorot_normal", "glorot_uniform"]
listOptimizer = ["sgd","adadelta","momentum","nag","adam","adamax",
"rmsprop","adagrad"]
listAlpha = [0.01,0.001,0.0001]

def printGraph(R):
    N = len(R[0])
    x = np.array(range(N))
    trainA = np.asarray([i[0] for i in R[0]])/ 100
```

```
    trainE = [i[1] for i in R[0]]
    validA = np.asarray([i[0] for i in R[1]])/ 100
    validE = [i[1] for i in R[1]]

    trace0 = Scatter(
        x = x,
        y = trainA,
        line = dict(
            color = ('rgb(22, 96, 167)'),
            width = 2),
        name = 'Training Accuracy'
    )
    trace1 = Scatter(
        x = x,
        y = trainE,
        line = dict(
            color = ('rgb(22, 96, 167)'),
            width = 2,
            dash = 'dot'),
        name = 'Training Error'
    )

    trace2 = Scatter(
        x = x,
        y = validA,
        line = dict(
            color = ('rgb(205, 12, 24)'),
            width = 2),
        name = "Validate Accuracy"
    )

    trace3 = Scatter(
        x = x,
        y = validE,
        line = dict(
            color = ('rgb(205, 12, 24)'),
            width = 2,
            dash = 'dot'),
        name = "Validate Error"
    )

    data = [trace0, trace1, trace2, trace3]
    layout = Layout(
        title = R[2],
```

```
    )
    plot(Figure(data=data, layout = layout), filename= R[2] + ".html")

    print(len(R[3]))
    pickling_on = open(R[2] + ".pickle","wb")
    pickle.dump(R[3], pickling_on)
    pickling_on.close()

def emptyList(lenght):
    return [None for _ in range(lenght)]

def T(x):
    return np.transpose(x)

@jit(float32[:,:](float32[:,:],float32[:,:]), fastmath = True)
def mul(a, b):
    return np.matmul(a, b)

def ZL(x):
    return np.zeros_like(x)

@jit(float32[:,:,:](float32[:,:,:]), fastmath = True)
def zoom(x):
    s, d = x.shape[0], x.shape[2]
    y = np.zeros((s*2, s*2, d), dtype=np.float32)
    for c in range(d):
        y[:, :, c] = ndimage.zoom(x[:, :, c], 2, order=0)
    return y

@jit(float32[:,:](float32[:,:]))
def rotate(x):
    return np.fliplr(np.flipud(x))

@jit(float32[:,:](float32[:,:,:],float32[:,:,:]), fastmath = True)
def conv3D(input, weight):
    S = input.shape[0] - weight.shape[0] + 1
    res = np.zeros((S, S), dtype=np.float32)
    for i in range(input.shape[2]):
        res += convolve2d(input[...,i],weight[...,i],mode='valid')
    return res


class Data:
    def __init__(self, path, sizeImage):
```

```
        Images = os.listdir(path)
        trainImages = []
        trainLabels = []
        l = np.array([[0, 1], [1, 0]], dtype=np.float32).reshape(2, 2, 1)
        for image in tqdm(Images, leave=False):
            pathImage = os.path.join(path, image)
            img = cv2.imread(pathImage)
            trainImages.append(transform.resize(img, (sizeImage, sizeImage, 3)))
            if (image.find("cat")) != -1:
                trainLabels.append(l[1])
            else:
                trainLabels.append(l[0])
        self.data = (np.array(trainImages), np.array(trainLabels))

    def standardize(self):
        x = self.data[0]
        x = (x - np.mean(x, axis=0)) / np.std(x, axis=0)
        self.data = (x, self.data[1])

    @staticmethod
    def shuff(trainData, trainLabels):
        indices = np.array(range(len(trainData))).reshape((len(trainData),))
        random.shuffle(indices)
        trainData = trainData[indices, ...]
        trainLabels = trainLabels[indices, ...]
        return (trainData, trainLabels)

    def split(self, fraction=0.8):
        data = self.data
        index = int(data[0].shape[0]*fraction)
        self.data = (data[0][:index], data[1][:index],
                     data[0][index:], data[1][index:])


class Function:
    def __init__(self, f):
        self.function = f.lower()

    def __str__(self):
        if self.function == None:
            return "None"
        else:
            return "%s" % (self.function)
```

```
    def calculate(self, input):
        if self.function == "sigmoid":
            input[input>20] = 20
            input[input<-20] = -20
            return 1/(1 + np.exp(-input))
        elif self.function == "relu":
            R = np.copy(input)
            R[R < 0] = 0
            return R
        elif self.function == "vectorize":
            S, D = input.shape[0], input.shape[2]
            R = np.zeros((S*S, D), dtype=np.float32)
            for i in range(S*S):
                for k in range(D):
                    R[i, k] = input[i % S, i//S, k]
            return R.reshape(-1, 1)
        else:
            f = np.mean if self.function == "mean" else np.max
            blockShape = (2, 2)
            inputShape = np.array(input.shape)
            blockShape = np.array(blockShape)
            input = np.ascontiguousarray(input)
            newShape = (inputShape[0] // blockShape[0], inputShape[1] //
blockShape[1], inputShape[2]) + tuple(blockShape)
            newStrides = tuple(input.strides[:-1] * blockShape) +
tuple([input.strides[-1], input.strides[0], input.strides[1]])
            output = as_strided(input, shape=newShape, strides=newStrides)
            for _ in range(2):
                output = f(output, axis=-1)
            return np.asarray(output, dtype= np.float32)
    @jit
    def deviation(self, N, A, D):
        R = np.copy(N)
        if self.function == "sigmoid":
            return A * (1-A) * D
        elif self.function == "relu":
            R[R > 0] = 1
            R[R <= 0] = 0
            return R * D
        elif self.function == "vectorize":
            s, d = A.shape[0], A.shape[-1]
            R = np.zeros((s, s, d), dtype=np.float32)
            for a in range(s):
                for b in range(s):
```

```
                    for c in range(d):
                        R[a, b, c] = D[c*s*s + b*s + a, 0]
            return R
        elif self.function == "mean":
            return D /4
        else:
            # max
            Z = zoom(A)
            R[Z != N] = 0
            R[Z== N] = 1
            return R * D


class Parameter:
    alpha = 0.01
    decay = 0.9
    epsilon = 10**(-8)
    beta1 = 0.9
    beta2 = 0.99

    def __init__(self, typeParameter, shapeParameter, optimizer, initializer,
alpha):
        if initializer not in {'lecun_uniform', 'lecun_normal',
                               'glorot_uniform', 'glorot_normal',
                               'he_uniform', 'he_normal'}:
            raise ValueError("Invalid initializer")
        self.alpha = alpha
        self.optimizer = optimizer
        self.initializer = initializer
        self.value = self.initValue(typeParameter, shapeParameter)
        self.D = ZL(self.value)
        if self.optimizer == "adadelta":
            self.Eg2 = ZL(self.value)
            self.ED2 = ZL(self.value)
        elif self.optimizer == "adam" or self.optimizer == "adamax":
            self.m = ZL(self.value)
            self.v = ZL(self.value)
            self.t = 0
        elif self.optimizer == "momentum":
            pass
        elif self.optimizer == "nag" or self.optimizer == "sgd":
            pass
        elif self.optimizer == "adagrad" or self.optimizer == "rmsprop":
            self.cache = ZL(self.value)
```

```
        else:
            raise RuntimeError("Not found optimizer")

    def initValue(self, typeParameter, shapeParameter):
        if typeParameter == "bias":
            if len(shapeParameter) == 4:
                return np.zeros((shapeParameter[3]), dtype = np.float32)
            else:
                return np.zeros((shapeParameter[0], 1), dtype = np.float32)
        else:
            fans = self.getFans(shapeParameter)
        if self.initializer == 'lecun_uniform':
            return self.generateDitribution('uniform', shapeParameter, fans,
'fanIn', 3.)
        elif self.initializer == 'lecun_normal':
            return self.generateDitribution('normal', shapeParameter, fans,
'fanIn', 1.)
        elif self.initializer == 'glorot_uniform':
            return self.generateDitribution('uniform', shapeParameter, fans,
'fanAvg', 3.)
        elif self.initializer == 'glorot_normal':
            return self.generateDitribution('normal', shapeParameter, fans,
'fanAvg', 1.)
        elif self.initializer == 'he_uniform':
            return self.generateDitribution('uniform', shapeParameter, fans,
'fanIn', 6.)
        else:
            # he_normal
            return self.generateDitribution('normal', shapeParameter, fans,
'fanIn', 2.)

    def generateDitribution(self, distribution, shape, fans, mode, scale):
        n = (fans[0]+fans[1])/2 if mode == "fanAvg" else fans[mode != "fanIn"]
        s = np.sqrt(scale / n)
        if distribution == "normal":
            return np.random.normal(loc = 0, scale = s, size = shape)
        else:
            return np.random.uniform(low = -s, high = s, size = shape)

    def getFans(self, shape):
        if len(shape) == 2:
            return (shape[1], shape[0])
        else:
```

```
            return (np.prod(shape[:3]), shape[-1])
    # @jit
    def learn(self, g):
        if self.optimizer == "adadelta":
            self.Eg2 = self.decay * (self.Eg2) + (1-self.decay) * g**2
            self.D = - np.sqrt(self.ED2 + self.epsilon) * g / np.sqrt(self.Eg2 +
self.epsilon)
            self.ED2 = self.decay * self.ED2 + (1-self.decay) * self.D * self.D
        elif self.optimizer == "momentum":
            self.D = self.decay * self.D - self.alpha * g

        elif self.optimizer == "adam":
            self.t += 1
            self.m = self.beta1 * self.m + (1-self.beta1) * g
            self.v = self.beta2 * self.v + (1-self.beta2) * g**2
            mHat = np.divide(self.m , (1 - np.power(self.beta1, self.t)))
            vHat = np.divide(self.v , (1 - np.power(self.beta2, self.t)))
            self.D = - np.divide((self.alpha * mHat) , (np.sqrt(vHat) +
self.epsilon))
        elif self.optimizer == "adamax":
            self.t += 1
            self.m = self.beta1 * self.m + (1-self.beta1) * g
            self.v = np.maximum(self.beta2 * self.v, np.abs(g))
            self.D = - self.alpha * self.m / ((1 - np.power(self.beta1, self.t))
* (self.v+self.epsilon))
        elif self.optimizer == "nag":
            pD = self.D
            self.D = self.decay * self.D - self.alpha * g
            self.D = - self.decay * pD + (1+ self.decay)* self.D
        elif self.optimizer == "adagrad":
            self.cache += np.multiply(g,g)
            self.D = - np.divide(np.multiply(self.alpha, g),
                                 np.sum((np.sqrt(self.cache),
                                         self.epsilon)
                                        )
                                 )
        elif self.optimizer == "rmsprop":
            self.cache = self.decay * self.cache + (1 - self.decay) * g**2
            self.D = - self.alpha * g / (np.sqrt(self.cache) + self.epsilon)
        elif self.optimizer == "sgd":
            self.D = - self.alpha * g
        self.value += self.D
```

```
class Layer:
    def __init__(self, typeLayer, initializer = None, optimizer = None,
shapeParameter = None, activation = None, alpha = None):
        self.typeLayer = typeLayer
        self.activation = Function(activation)
        self.learnable = (typeLayer == "F" or typeLayer == "C")
        if self.learnable:
            self.weight = Parameter("weight", shapeParameter, optimizer,
initializer, alpha)
            self.bias = Parameter("bias", shapeParameter, optimizer,
initializer,alpha)

    def __str__(self):
        s = "type: %s, activation: %s" % (self.typeLayer,
self.activation.__str__())
        if self.learnable:
            s += ", weight shape: %s, bias shape: %s" %(self.weight.value.shape,
self.bias.value.shape)
        return s

    def learn(self, DW, DB):
        self.weight.learn(DW)
        self.bias.learn(DB)

    # @jit(nopython=True)
    def calculate(self, inp):
        if self.typeLayer == "C":
            S, D = inp.shape[0] - self.weight.value.shape[0] + 1,
self.weight.value.shape[3]
            N = np.zeros((S, S, D), dtype = np.float32)
            for q in range(D):
                N[..., q] = conv3D(inp, self.weight.value[..., q]) +
self.bias.value[q]
            return (N, self.activation.calculate(N))
        elif self.typeLayer == "F":
            N = (mul(self.weight.value, inp) + self.bias.value)
            return (N, self.activation.calculate(N))
        else:
            return (inp, self.activation.calculate(inp))


class Network:
    def __init__(self, optimizer, initializer, alpha):
        self.layers = []
```

```
        self.L = 0
        self.optimizer = optimizer.lower()
        self.initializer = initializer.lower()
        self.forwardT = 0.
        self.backwardT = 0.
        self.forwardN = 0
        self.backwardN = 0
        self.alpha = alpha

    def __str__(self):
        s = "Initializer: %s, optimizer: %s, number Parameter: %d
\n" %(self.initializer, self.optimizer, self.numParameter())
        for i in range(self.L):
            s += "Layer: %d, %s \n" %(i, self.layers[i].__str__())
        return s

    def add(self, typeLayer, shape = None, activation = "sigmoid", shapeInput =
None):
        if typeLayer == "C":
            shapeWeight = (shape[0], shape[1],self.layers[-
2].weight.value.shape[3], shape[2]) if shapeInput == None else (shape[0],
shape[1], shapeInput[2], shape[2])
            self.layers.append(Layer(
                                    typeLayer,
                                    optimizer = self.optimizer,
                                    shapeParameter = shapeWeight,
                                    activation = activation,
                                    initializer = self.initializer,
                                    alpha = self.alpha))
        elif typeLayer == "P":
            self.layers.append(Layer(
                                    typeLayer,
                                    activation = activation))
        elif typeLayer == "V":
            self.layers.append(Layer(typeLayer,
                                    activation = "vectorize"))
        else:
            self.layers.append(Layer(
                                    typeLayer,
                                    optimizer = self.optimizer,
                                    shapeParameter = shape,
                                    activation = activation,
                                    initializer = self.initializer,
                                    alpha=self.alpha))
```

```
        self.L += 1

    def trans(self, i):
        return (self.layers[i+1].typeLayer, self.layers[i].typeLayer)

    def accuracy(self, testData, testLabels):
        correctSample = 0
        meanError = 0.
        for indexSample in tqdm(range(len(testData)),leave=False):
            (_, A, _) = self.forward(testData[indexSample, ...])
            e = A[-1] - testLabels[indexSample, ...]
            meanError += mul(T(e), e)[0, 0]
            correctSample += (np.argmax(A[-1]) ==
            np.argmax(testLabels[indexSample, ...]))

        return (correctSample/len(testData) * 100, meanError / len(testData))

    def timeAvg(self):
        print("Avg forward time: %4.2f" %(self.forwardT/ self.forwardN))
        print("Avg backward time: %4.2f" %(self.backwardT/ self.backwardN))
        self.forwardN = 0
        self.forwardT = 0.
        self.backwardN = 0
        self.backwardT = 0.

    def numParameter(self):
        res = 0
        for i in range(self.L):
            if self.layers[i].learnable:
                if self.layers[i].typeLayer == 'C':
                    res += np.prod(self.layers[i].weight.value.shape[:3]) +
self.layers[i].bias.value.shape[0]
                else:
                    res += np.prod(self.layers[i].weight.value.shape) +
self.layers[i].bias.value.shape[0]
        return res

    def forward(self, data):
        t =time.time()
        A = emptyList(self.L)
        N = emptyList(self.L)
        T1 = np.zeros((self.L))
        for i in range(self.L):
            t2 = time.time()
```

```
            if i == 0:
                (N[0], A[0]) = self.layers[0].calculate(data)
            else:
                (N[i], A[i]) = self.layers[i].calculate(A[i-1])
            T1[i] = time.time() - t2
        self.forwardN += 1
        self.forwardT += time.time() - t
        return (N, A, T1)

    def backward(self, N, A, input, target):
        t = time.time()
        DA = emptyList(self.L)
        DAf = emptyList(self.L)
        DW = emptyList(self.L)
        DB = emptyList(self.L)
        T2 = np.zeros(self.L)
        for i in range(self.L):
            if self.layers[i].learnable:
                DW[i] = ZL(self.layers[i].weight.value)
                DB[i] = ZL(self.layers[i].bias.value)

        e = target - A[self.L-1]
        t2 = time.time()
        DA[-1] = -2 * self.layers[-1].activation.deviation(N[-1], A[-1], e)
        DW[-1] = mul(DA[-1], T(A[-2]))
        DB[-1] = DA[-1]
        T2[self.L-1] = time.time() - t2
        for i in range(self.L-2, -1, -1):
            t2 = time.time()
            if self.trans(i) == ('F', 'F'):
                DA[i] = self.layers[i].activation.deviation(N[i], A[i],
mul(T(self.layers[i+1].weight.value), DA[i+1]))
                DW[i] = mul(DA[i], T(A[i-1]))
                DB[i] = DA[i]

            elif self.trans(i) == ('F', 'V'):
                DA[i] = mul(T(self.layers[i+1].weight.value), DA[i+1])

            elif self.trans(i) == ('V', 'P'):
                DA[i] = self.layers[i+1].activation.deviation(N[i], A[i],
DA[i+1])

            elif self.trans(i) == ('P', 'C'):
                inputLayer = input if i == 0 else A[i-1]
```

```
                inputLayer = np.asarray(inputLayer, dtype=np.float32)

                DA[i] = self.layers[i+1].activation.deviation(N[i+1], A[i+1],
zoom(DA[i+1]))
                DAf[i] = self.layers[i].activation.deviation(N[i], A[i], DA[i])

                for p in range(self.layers[i].weight.value.shape[2]):
                    rot = rotate(inputLayer[..., p])
                    for q in range(self.layers[i].weight.value.shape[3]):
                        DW[i][...,p,q] = convolve2d(rot, DAf[i][...,q],
mode='valid')

                for p in range(self.layers[i].weight.value.shape[3]):
                    DB[i][p] += DAf[i][..., p].sum()
            else:
                # from C to P
                DA[i] = ZL(A[i])
                for p in range(A[i].shape[2]):
                    for q in range(A[i+1].shape[2]):
                        DA[i][..., p] += convolve2d(DAf[i+1][...,q],
                        rotate(self.layers[i+1].weight.value[...,p,q]),
mode='full')

            T2[i] = time.time() - t2
        self.backwardN += 1
        self.backwardT += time.time() - t
        return (e, DW, DB,T2)

    def accumulateWeight(self, DW, DB, dw, db):
        for i in range(self.L):
            if self.layers[i].learnable:
                DW[i] += dw[i]
                DB[i] += db[i]
        return (DW, DB)
    @jit(fastmath = True)
    def update(self, DW, DB, batchSize):
        for i in range(self.L):
            if self.layers[i].learnable:
                self.layers[i].learn(DW[i]/batchSize, DB[i]/batchSize)
    @jit
    def initDeviationWeight(self):
        DW = emptyList(self.L)
        DB = emptyList(self.L)
        for i in range(self.L):
```

```
            if self.layers[i].learnable:
                DW[i] = ZL(self.layers[i].weight.value)
                DB[i] = ZL(self.layers[i].bias.value)
        return (DW, DB)
    @jit(fastmath = True)
    def calculate(self, maxLoop, batchSize, data):
        print(self.__str__())
        trainData, trainLabels, testData, testLabels = data.data
        numBatch = len(trainData) // batchSize
        trainLog = []
        validLog = []
        layerLog = []
        trainAcc = self.accuracy(trainData, trainLabels)
        validateAcc = self.accuracy(testData, testLabels)
        print("TRAINING: Accuracy %4.3f %% Error: %4.3f" % trainAcc)
        print("VALIDATE: Accuracy %4.3f %% Error: %4.3f" % validateAcc)
        print("-------------")
        trainLog.append(trainAcc)
        validLog.append(validateAcc)
        for iterator in range(maxLoop):
            print("************")
            print("Loop", iterator+1, "/", maxLoop)
            (trainData, trainLabels) = Data.shuff(trainData, trainLabels)
            for indexBatch in range(numBatch):
                print("Batch", indexBatch+1, "/", numBatch)
                (DW, DB) = self.initDeviationWeight()
                for indexInBatch in tqdm(range(batchSize),leave=False):
                    indexSample = indexInBatch + batchSize * indexBatch
                    (N, A, _) = self.forward(trainData[indexSample, ...])
                    (_, dw, db, _) = self.backward(N, A,
trainData[indexSample, ...],
                                                   trainLabels[indexSample, ...])
                    (DW, DB) = self.accumulateWeight(DW, DB, dw, db)
                self.update(DW, DB, batchSize)
                trainAcc = self.accuracy(trainData, trainLabels)
                validateAcc = self.accuracy(testData, testLabels)
                print("TRAINING: Accuracy %4.3f %% Error: %4.3f" % trainAcc)
                print("VALIDATE: Accuracy %4.3f %% Error: %4.3f" % validateAcc)
                self.timeAvg()
                print("-------------")
                trainLog.append(trainAcc)
                validLog.append(validateAcc)
                layerLog.append(self.layers)
```

```
        s = "%s, %s," %(self.initializer, self.optimizer) + str(self.layers[-
1].weight.alpha) + str(self.numParameter())
        s += " %d samples, %s loops, %d batchSize" %(len(trainData)/0.8, maxLoop,
batchSize)
        for i in range(self.L):
            s+= self.layers[i].typeLayer
            if self.layers[i].learnable:
                s += str(self.layers[i].weight.value.shape[0])
        printGraph((trainLog, validLog, s, layerLog))


def CNN(sizeImage, cnvLayers, spatialSize = 5, depthOutput = 8,
        FCLayers = 2, optimizer = 'adadelta', initializer = 'lecun_normal', alpha
= 0.01, neuron = None):
    net = Network(optimizer = optimizer, initializer = initializer, alpha =
alpha)
    currentSize = [sizeImage, 3]

    net.add('C', shape = (spatialSize, spatialSize, depthOutput), activation =
'relu', shapeInput=(sizeImage, sizeImage,3))
    currentSize = [currentSize[0] - spatialSize + 1, depthOutput]
    net.add('P', activation = 'Max')

    if currentSize[0] % 2 != 0:
        raise ValueError('unvalid size')
    currentSize[0] //= 2

    for _ in range(cnvLayers-1):
        depthOutput *= 2
        net.add('C', shape = (spatialSize, spatialSize, depthOutput),
activation='relu')
        currentSize = [currentSize[0] - spatialSize + 1, depthOutput]
        net.add('P', activation = 'Max')
        if currentSize[0] % 2 !=0:
            raise ValueError('unvalid size')
        currentSize[0] //= 2
        print(currentSize)

    net.add('V')
    vectorSize = int(currentSize[0]**2 * currentSize[1])
    if neuron == None:
        neuron = int(2**int(np.log2(int(currentSize[0]**2 * currentSize[1]))))

    net.add('F', (neuron, vectorSize), 'relu')
```

```
    for _ in range(FCLayers-2):
        net.add('F', (neuron//2, neuron), 'relu')
        neuron //= 2

    net.add('F', (2, neuron), 'sigmoid')
    return net

def NN(sizeImage,FCLayers = 2, optimizer = 'adadelta',
       initializer = 'lecun_normal', alpha = 0.01, neuron = None):
    net = Network(optimizer = optimizer, initializer = initializer, alpha =
alpha)
    net.add('V')
    if neuron == None:
        neuron = int(2**int(np.log2(sizeImage * sizeImage * 3)))
    input = int(sizeImage * sizeImage * 3)
    net.add('F', (neuron, input), 'relu')
    for _ in range(FCLayers-2):
        net.add('F', (neuron//2, neuron), 'relu')
        neuron //= 2

    net.add('F', (2, neuron), 'sigmoid')
    return net

data = Data(path = "C:\\train100", sizeImage = 52)
data.standardize()
data.data = Data.shuff(data.data[0], data.data[1])
data.split(0.8)

# nn = Network(alpha = 0.001, optimizer = "rmsprop", initializer =
"lecun_normal")
# nn.add("V")
# nn.add("F", (512, 7500), activation="relu")
# nn.add("F", (512, 512), activation="relu")
# nn.add("F", (256, 512), activation="relu")
# nn.add("F", (256, 256), activation="relu")
# nn.add("F", (256, 256), activation="relu")
# nn.add("F", (2, 256), activation="sigmoid")
# R = nn.calculate(maxLoop = 5, batchSize = 40, data = data)

# net = CNN(sizeImage = 52,
#         initializer = 'glorot_uniform',
#         optimizer='adadelta', alpha= 0.0001,
#         cnvLayers = 3, spatialSize= 5, depthOutput= 4,
```

```
#         FCLayers = 2, neuron = 2048)
# net.calculate(maxLoop = 10, batchSize = 40, data = data)

net = NN(sizeImage = 52,
        initializer= 'glorot_uniform',
        optimizer='rmsprop', alpha= 0.0001,
        FCLayers = 4, neuron = 1024)
net.calculate(maxLoop = 3, batchSize = 80, data = data)

for opti in listOptimizer:
    for alpha in listAlpha:
        net = CNN(sizeImage = 52, cnvLayers = 2, spatialSize= 5,
            depthOutput= 8, optimizer=opti,alpha= alpha, initializer =
'glorot_uniform')
        net.calculate(maxLoop = 10, batchSize = 32, data = data)
```