PHẦN TÓM TẮT

TÊN ĐỀ TÀI (tiếng Việt)

Mục tiêu của đề tài là xây dựng một hệ thống có thể nhận diện kí tự số viết tay.

So sánh cuộc đấu cờ giữa Garry Kasparov và Deep Blue [8] năm 1996 và AlphaGo năm 2016 [6]. Trong vòng 20 năm có tới 2 chương trình dùng để so tài giữa máy móc và con người. Nhưng thực tế Deep Blue được IBM lập trình cho nhiều nước cờ, trong quá trình thi đấu thì nó sẽ chọn ra nước cờ thích hợp; trong khi đó AlphaGo là một trí tuệ nhân tạo (A.I.) được tạo ra và nó cải thiện tính năng nó bằng cách tự tạo ra tình thế rồi giải quyết tình huống đó rồi sẽ học cách đi nước cờ. AlphaGo hơn Deep Blue ở chỗ là nếu Deep Blue cho chơi loại cờ khác như cờ tướng thì nó sẽ không thể nào chơi được, còn AlphaGo thì nó sẽ tự học từ đầu và sẽ chơi được bất cứ thể loại cờ nào.

Đề tài *Nhận diện kí tự dùng trí tuệ nhân tạo* so với phương pháp nhận diện thông thường dùng thị giác máy tính cũng giống như việc so sánh giữa AlphaGo và Deep Blue. Lấy ví dụ giờ ta không muốn nhận diện kí tự nữa mà chúng ta lại muốn nhận diện vật khác ví dụ như phân biệt chó hay mèo thì chúng ta chỉ cần chỉnh lại cấu trúc của hệ thống, thay đổi dữ liệu và cho nó tự học lại là giải quyết được vấn đề; trong khi so với phương pháp thị giác máy tính thông thường thì ta phải tìm lại đặc điểm của con chó ra sao, đặc điểm của con mèo ra sao rồi phân biệt chúng dựa trên đặc điểm đó.

Với lí do trên đề tài này sẽ trình bày phương pháp chung cho việc nhận dạng, đi xây dựng thuật toán và phân tích các thuật toán. Cho nên sẽ lấy ví dụ là nhận dạng kí tự số viết tay.

ABSTRACT

The goal of the project is building a system that can classify numerical hand-written digits.

Let's compare the competition of Garry Kasparov versus Deep Blue [8] and AlphaGo [6]. Just in 20 years, there are 2 programmes have been created in order to compete with people. Actually, Deep Blue was programmed immense of strategies; in action, the program chooses satisfied strategy to do. On the other hand, AlphaGo is an *Artificial Intelligence (A.I.)* has been created so it can create game and play by its own so AlphaGo could learn from these mistakes. At conclusion, AlphaGo is better than Deep Blue because Deep Blue is only able to play chess while AlphaGo can learn and play another games if it has enough data.

Distinguishing the thesis, *Classifying numerical hand-written digits using artificial intelligence*, between ordinary computer vision methods is identical to the difference between AlphaGo and Deep Blue. Let's take an example. If we want to change the job of our program, classify if a image is dog or cat, we need to adjust some parameters in the program, change the data set. While in ordinary computer vision methods, we have to point out the characteristics, features of cat and dog so as to make decision whether this image is cat or dog.

With all of these reasons, the research shows general method for classification and take example on classifying *numerical hand-written digits - MNIST*

Mục Lục

1 Phần mở đầu

Đối với phương pháp xử lí ảnh truyền thống thì chúng ta trước tiên cần phải chuyển đổi ảnh màu thành ảnh xám, sau đó dùng bộ lọc như Sobel, Canny để tách biên và cạnh, từ đó chúng ta sẽ tìm ra những đặc trưng của vật để nhận diện. Lấy ví dụ nhận diện con mèo thì ta sẽ tìm ra tính chất đặc trưng của con mèo rồi sẽ lập trình cho máy tính nhận dạng nếu những điểm ảnh đó phù hợp với những gì đã lập trình thì sẽ đưa ra quyết định tấm ảnh đó là con mèo; nhưng thực tế thì mèo nó sẽ có đủ mọi tư thế khiến chúng ta không thể nào lập trình được hết tất cả tư thế đó được. Như có thể nhìn thấy sự khác nhau giữa Hình 1 và Hình 2:

Hình 1 – Mèo thông thường

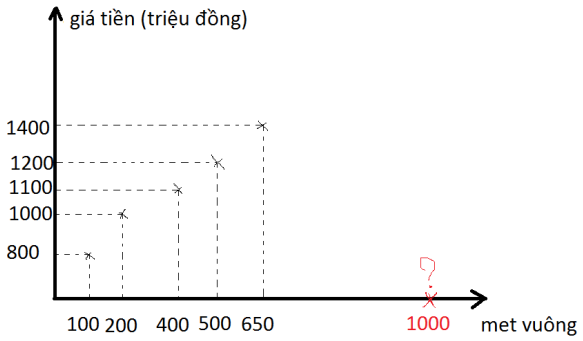
Hình 2 – Mòe tư thế lạ

Nếu loài mèo đổi tư thế thì mỗi lần muốn nhận diện thì phải làm một thuật toán khác. Do đó, đề tài này xin trình bày một phương pháp mới thay thế phương pháp thị giác máy tính cũ đó là dùng *Máy học - Machine Learning* và cụ thể là *Mạng Nơron - Neural Network*, nó có thể học, tự tìm ra đặc điểm của đối tượng mình muốn nhận dạng, nếu có thay đổi gì thì chỉ cần thay đổi cấu trúc của chương trình (ở đây gọi là mẫu) và thay đổi dữ liệu đầu vào nếu muốn nhận dạng đối tượng khác (dữ liệu đầu vào ở đây là những điểm ảnh của một tấm ảnh và có nhiều tấm ảnh như vậy). Và đề tài này tập trung vào phương pháp xây dựng hệ thống; phân tích thuật toán để xây dựng và sẽ được viết dưới dạng mã giả - pseudocode.

Đề tài này sẽ được viết trên ngôn ngữ lập trình Python 3.5 và sử dụng thư viện TensorFlow và sẽ lấy ví dụ là nhận diện kí tự số viết tay. Lí do là vì ngôn ngữ Python 3.5 là ngôn ngữ miễn phí được cộng đồng hỗ trợ rộng rãi và là ngôn ngữ cực mạnh trong việc phân tích dữ liệu; TensorFlow thì sẽ tối ưu hóa việc tính toán bằng cách chia việc tính toán trên nhiều nhân của CPU và tập MNIST là tập dữ liệu cực lớn bao gồm 42000 tấm ảnh kí tự số viết tay dùng để huấn luyện mạng Nơron và đã được chuyển hóa thành bảng tính excel với từng dòng tương ứng với tất cả điểm ảnh của ảnh đó và đã biết được nội dung tấm ảnh đó là số mấy. Và tập để kiểm tra độ chính xác gồm 20000 tấm ảnh cũng tương tự như tập huấn luyện nhưng không biết tấm ảnh đó là số mấy và nhiệm vụ đoán số là của mẫu.

2 Tổng quan về mạng Nơron

Trước khi tìm hiểu sâu về mạng Nơron thì chúng ta sẽ tiếp cận bài toán tìm quy luật phân bố giá nhà $P(Y|X)$ với điều kiện là chúng ta biết quy luật phân bố diện tích nhà $P(X = x_i)$ và giá nhà dựa trên diện tích nhà $P(Y|x = x_i)$ với Y là biến ngẫu nhiên biểu thị giá nhà và X là biến ngẫu nhiên biểu thị diện tích nhà. Bây giờ chúng ta biết một số diện tích ngôi nhà và giá bán của nó, bài toán đặt ra bây giờ là định giá một ngôi nhà khác với diện tích không nằm trong tập dữ liệu ban đầu Hình 3.



Hình 3 – Sự phục thuộc giữa giá nhà và diện tích nhà

Như hình trên cho ta biết nếu ngôi nhà có diện tích $100m^2$ thì sẽ có giá là 800 triệu đồng, $200m^2$ thì có giá là 1000 triệu đồng, $400m^2$ có giá 1100 triệu đồng, $500m^2$ thì có giá 1200 triệu đồng, $650m^2$ thì có giá 1400 triệu đồng. Bây giờ ta định giá ngôi nhà có diện tích $1000m^2$

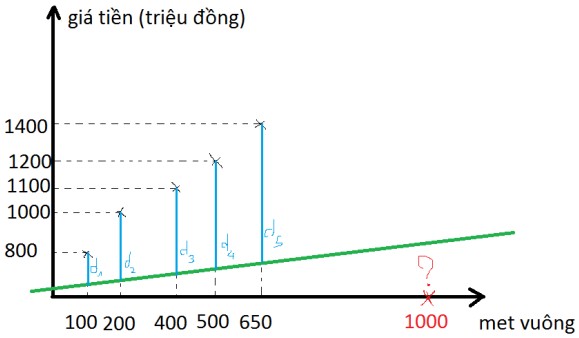
Như ta biết trong thực tế thì giá tiền một ngôi nhà phụ thuộc tuyến tính với diện tích một ngôi nhà nên ta có thể xem giá ngôi nhà $\hat{y} = \omega_1 x + \omega_0$. Nếu bây giờ ta có thể tìm được ω_1 và ω_0 thì ta sẽ tìm được quy luật phụ thuộc của giá nhà (y) theo diện tích nhà (x) hay nói cách khác là $P(Y|X = x_i)$.

Mục đích của đường thẳng $\hat{y} = \omega_1 x + \omega_0$ là đi qua tập dữ liệu làm sao cho nó sai số ít nhất, tức là tổng khoảng cách từ mỗi điểm trong tập dữ liệu đến đường thẳng đó chiếu theo trục giá tiền. Mục tiêu của bài toán này là tìm ω_1 và ω_0 sao cho tổng bình phương sai số $\sum_{i=1}^{5} d_i^2$ đạt giá trị nhỏ nhất (Hình 4) hay nói cách khác là $\sum_{i=1}^{n}(y - \hat{y})^2$). Đây là tổng sai số bình phương *(Mean Square Error - MSE)*. Do đây là hàm sai số nên ta sẽ gọi nó là hàm thất thoát hay gọi làm hàm ***loss***. Nhiệm vụ là biến đổi ω_0 và ω_1 sao cho hàm loss nhỏ nhất tức là sai số ít nhất dẫn đến phương trình sau đây:

$$\text{argmin}_{\omega_0,\omega_1} \sum_{i=1}^{n}(y - \hat{y})^2 \tag{1}$$

Có thể viết lại phương trình (1) thành:

$$\text{argmin}_{\omega_0,\omega_1} \sum_{i=1}^{n} (y - \omega_1 x - \omega_0)^2 \quad (1')$$



Hình 4 – Sai số của tiên đoán

Để giải được phương trình (1') thì ta làm các bước sau đây:

Algorithm 1 Cập nhật giá trị ω_0 và ω_1

1: Chọn giá trị α
2: Khởi tạo giá trị cho ω_0 và ω_1
3: Thay tất cả cặp giá trị (x,y) vào hàm loss
4: Tính đạo hàm từng phần của hàm loss theo từng biến ω_0 và ω_1: $\frac{\partial loss}{\partial \omega_0}$ và $\frac{\partial loss}{\partial \omega_1}$
5: *Vòng lặp*:
$\omega_0 := \omega_0 - \alpha \frac{\partial loss}{\partial \omega_0}$
$\omega_1 := \omega_1 - \alpha \frac{\partial loss}{\partial \omega_1}$

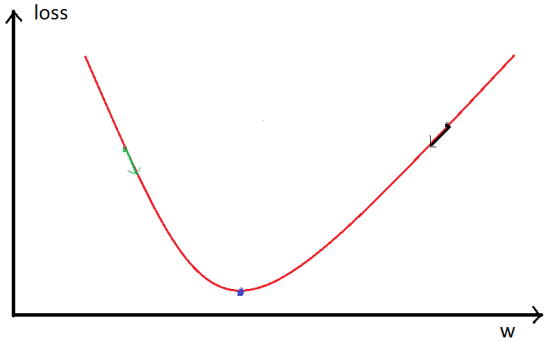
Thuật toán 1 có tên gọi là *Gradient Descent* với α gọi là hệ số học (*learning rate*). Nếu giá trị α hợp lí (thường rất nhỏ và gần bằng 0) thì thuật toán này bảo đảm sau một số vòng lặp thì hàm loss sẽ hội tụ về vị trí xấp xỉ với vị trí cực tiểu toàn cục (*global minimum*) (Hình 5). Nếu chúng ta chọn α quá lớn thì dẫn đến độ hội tụ không ổn định và nếu số lần lặp quá lớn thì sẽ dẫn đến không thể hội tụ được (Hình 7) và ngược lại nếu α quá nhỏ thì dẫn đến sẽ cần nhiều vòng lặp hơn để hội tụ (Hình 6). Trong thực tế thì giá nhà phụ thuộc vào diện tích, số lầu,... Diện tích, số lầu gọi là đặc trưng của dữ liệu và thay vì biểu thị bằng một biến x như trên thì ta sẽ dùng một vector có số chiều tương ứng với số đặc trưng của dữ liệu và sẽ có vector ω tương ứng:

$$X = \begin{bmatrix} 1 & x_1 & x_2 & \dots & x_n \end{bmatrix} \qquad \omega = \begin{bmatrix} \omega_0 & \omega_1 & \omega_2 & \dots & \omega_n \end{bmatrix}$$

Và đây là công thức của ma trận hóa $\hat{y} = \omega^T x$

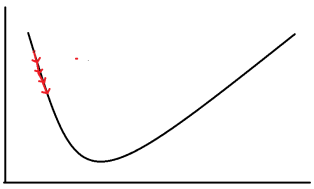
Việc ma trận hóa là cực kì quan trọng trong lĩnh vực *Máy học* vì ma trận hóa giúp cho việc tính toán trở thành tính toán song song trong khi nếu dùng vòng lặp thì nó sẽ tính từng công việc nên sẽ không tận dụng hết tài nguyên

của máy. Lấy ví dụ CPU của máy tính có nhiều nhân thì việc lập trình song song sẽ sử dụng hết tất cả các nhân đó trong một lệnh, trong khi dùng vòng lặp thì chỉ sử dụng đúng một nhân dẫn đến tốn thời gian.

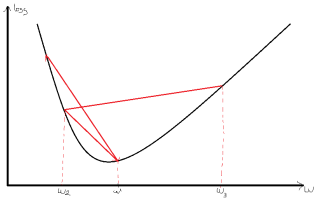


Hình 5 – Giải thích thuật toán Gradient Descent

Điểm màu lam là cực tiểu toàn cục. Nếu giá trị khởi tạo ω là tại điểm màu lục thì theo như hình là *gradient* của hàm loss sẽ có giá trị nhỏ hơn 0 ($\frac{\partial loss}{\partial \omega} < 0$) thì giá trị $\omega = \omega$ cộng thêm một đại lượng dương thì giá trị ω sẽ tiến đến gần giá trị cực tiểu toàn cục (tiến theo chiều dương trục w). Nếu giá trị ω đã nằm tại cực tiểu toàn cục thì $\frac{\partial loss}{\partial \omega} = 0$ dẫn đến ω sẽ không thay đổi. Trường hợp cuối cùng là ω nằm tại điểm màu đen thì *gradient* của hàm loss sẽ có giá trị lớn hơn 0 ($\frac{\partial loss}{\partial \omega} > 0$) nên giá trị ω mới sẽ bằng giá trị ω cũ cộng với đại lượng *âm* nên dẫn đến giá trị ω sẽ tiến theo chiều âm của trục w, điều đó có nghĩa là ω sẽ tiến đến cực tiểu toàn cục. Trường hợp không gian *n chiều* cũng tương tự.



Hình 6 – giá trị α nhỏ



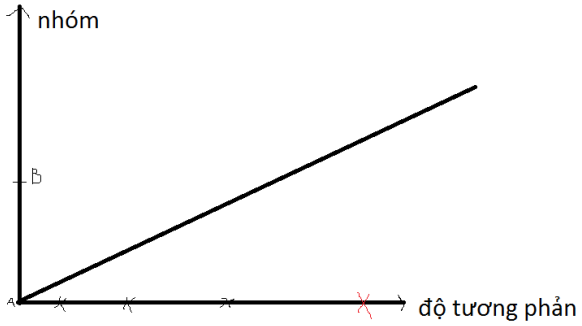
Hình 7 – giá trị α lớn

Phương pháp trên gọi là *Linear Regression* [1]. Phương pháp này dùng cho dữ liệu dạng tuyến tính nhưng nếu dùng để phân loại ảnh thì không thể được. Để huấn luyện hệ thống phân biệt được ảnh thì chúng ta cần phải dùng phương pháp khác đó là *mạng Nơron*

2.1 Giới thiệu về mạng Nơron

Lấy một ví dụ đơn giản. Chúng ta có một điểm ảnh và chúng ta muốn phân loại nó thuộc nhóm A hay B. Nếu giá trị của x cứ tăng liên tục đến một mức

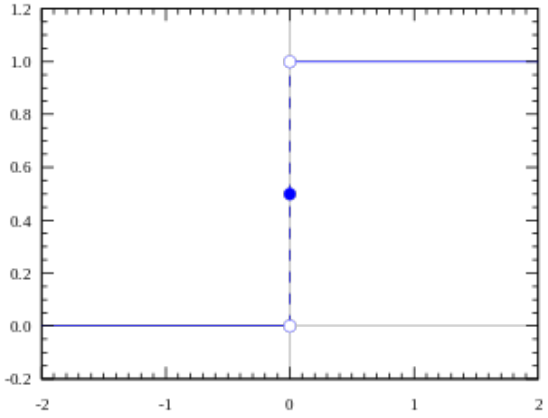
nào đó sẽ vượt khỏi tầm là nhóm A hay B (Hình 8). Để giải quyết vấn đề đó thì chúng ta cần đặt một ngưỡng *threshold* sao cho nếu giá trị $\hat{y} > threshold$ thì chúng ta xem đó là giá trị A và ngược lại, phương pháp đó gọi là phương pháp *Perceptron*.



Hình 8 – Phân biệt là nhóm A hay B

2.1.1 Perceptron

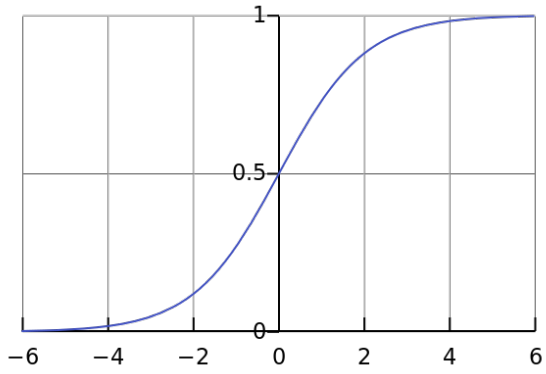
Perceptron có dạng đồ thị Hình 9.



Hình 9 – Đồ thị của Perceptron

Dựa vào dạng đồ thị này ta không thể tính được đạo hàm, điều đó dẫn đến việc là không thể học được. Để có thể học được và có thể làm cho giá trị $\hat{y}$ nằm trong khoảng [0, 1] thì ta cần tìm một hàm khác là phiên bản làm mượt của hàm bước nhảy trên. Hàm đó có tên là hàm *sigmoid* có công thức và dạng đồ thị Hình 10 :

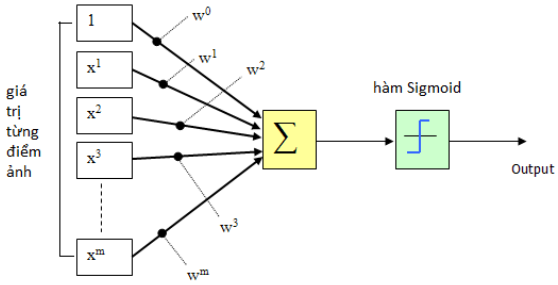
$$\hat{y} = \frac{1}{1-e^{-\omega^T x}} \tag{2}$$



Hình 10 – Đồ thị của hàm Sigmoid

Đồ thị này thể hiện độ chắc chắn về sự quyết định của mẫu bằng cách tính $P(y = A|x)$ hoặc $P(y = B|x)$

Perceptron có dạng cấu trúc như sau (Hình 11):



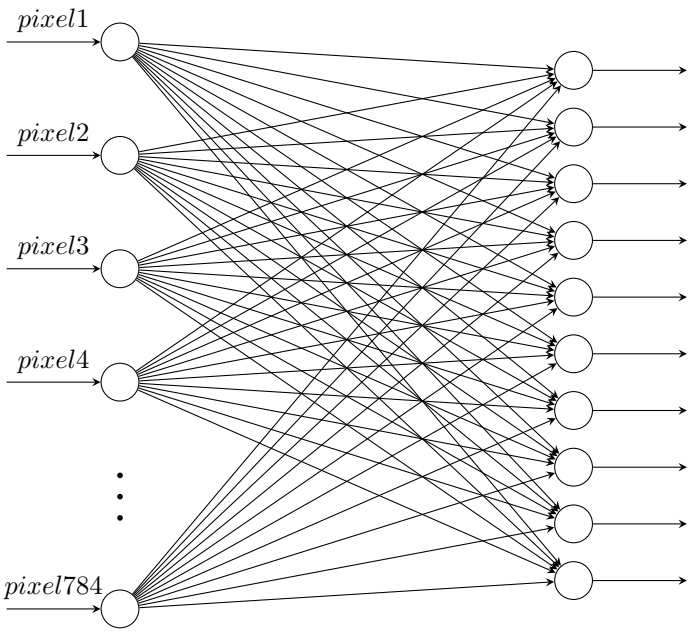
Hình 11 – Cấu trúc Perceptron

Algorithm 2 Thuật toán Perceptron

1: Chọn ngưỡng để xem ảnh đó thuộc nhóm nào: threshold
2: Đưa giá trị độ tương phản của từng điểm ảnh trong một tấm ảnh vào input
3: Tính $\hat{y} = \frac{1}{1-e^{-\omega^T x}}$
4: So sánh $\hat{y}$ và threshold và rút ra kết luận

Tuy *Perceptron* giải quyết được vấn đề về biến đổi giá trị output thuộc khoảng [0,1] và có thể tính đạo hàm tại mọi điểm nhưng nó chưa có khái niệm về tự động tách ra các đặc trưng như đã nêu ra ở phần giới thiệu. Để làm rõ hơn về việc học của mạng Nơron và tách đặc trưng thì chúng ta sẽ giải quyết ở chương sau-*Cấu trúc học nhiều tầng - Deep Learning*

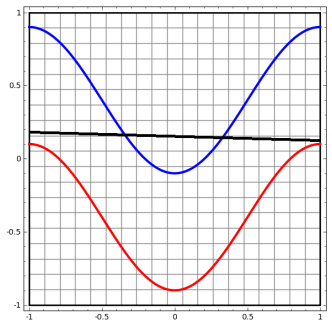
Perceptron để phân biệt kí tự có cấu trúc như sau:



Cấu trúc trên chỉ có 2 lớp, lớp thứ nhất là lớp đầu vào, mỗi vòng tròn tương ứng với 1 nốt, ta đưa giá trị từng điểm ảnh vào từng nốt tương ứng. Lớp tiếp theo là lớp đầu ra, đầu ra có dạng *one-hot-encoding* tức là nếu đầu ra là số 1 thì hệ thống trên sẽ cho ra vector [0,1,0,0,0,0,0,0,0,0] (đây là trường hợp chắc chắn 100%-thường có trong giá trị nhãn là y, nếu $\hat{y}$ thì sẽ có dạng vector [0.04,0.8,0.1,0.02,0.01,0.1,0.1,0.1,0,0]). *Perceptron* chuyển trực tiếp từ giá trị điểm ảnh từ ảnh gốc trực tiếp ra đầu ra, không có biến đổi tính chất của ảnh (như tách biên, cạnh được nêu ra ở phần giới thiệu)

Thực chất *Perceptron* sẽ vẽ đường thẳng để phân biệt dữ liệu (Hình 12). Nhưng do tính hạn hẹp của hệ thống (*small capacity*) nên *Perceptron* khó có thể phân biệt chính xác được. Muốn phân biệt được chính xác thì chúng ta cần phải vẽ đường cong, tức là biến đổi về mặt hình học nhưng vẫn giữ nguyên*tính chất topo* của đường thẳng vẽ ra bởi *Perceptron*. Điều đó có nghĩa là lớp đầu vào phải bị biến đổi, tức là phải đi qua một hoặc nhiều lớp để biến đổi mà giữ nguyên *tính topo*, những lớp đó được gọi là *Hidden Layers* [5]. Sau khi dữ liệu đầu vào đi qua một lớp *Hidden Layer* thì dữ liệu sẽ bị biến đổi và bây giờ lớp *Hidden Layer* đó sẽ đảm nhiệm vai trò như lớp dữ liệu đầu vào và sẽ tiếp tục truyền dữ liệu vào lớp tiếp theo là lớp đầu ra. Trong lớp *Hidden Layer* thì mỗi nốt sẽ là một hàm $\sigma()$ để có thể bẽ cong đường thẳng đó. Biểu diễn toán học như sau:

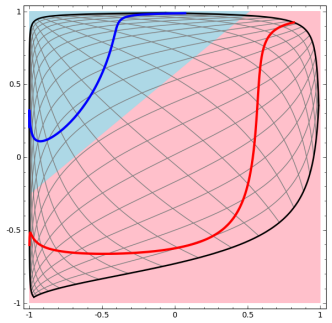
$$h^{(1)} = \sigma(\omega_1^T x + b^{(1)})$$
$$\hat{y} = \sigma(\omega_2^T h^1 + b^{(2)})$$



Hình 12 – Nhiệm vụ của Perceptron

Để phân biệt đường màu xanh và màu đỏ thì Perceptron chỉ có thể vẽ được đường thẳng màu đen và những điểm nào nằm bên trên đường thẳng đó sẽ được xem là thuộc tập màu xanh và ngược lại cho nên sẽ có một phần dữ liệu bị phân biệt sai

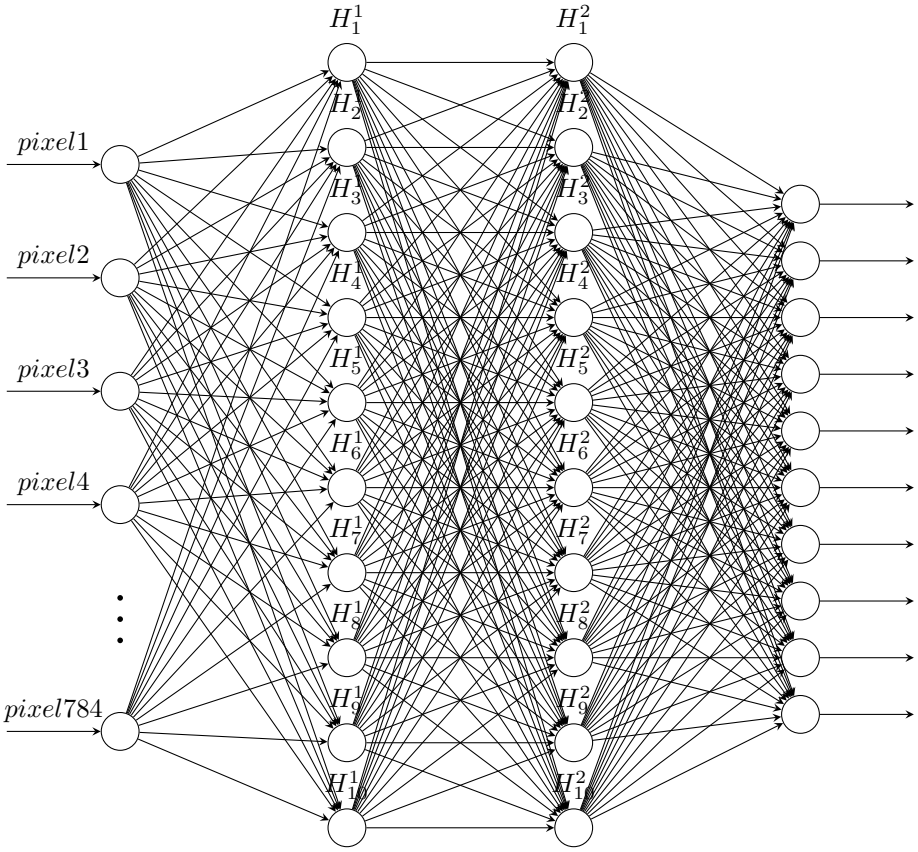
Sau khi qua lớp *Hidden Layer* thì dữ liệu đầu vào sẽ có dạng: Hình 13. Từ đây dữ liệu từ lớp *Hidden Layer* sẽ được đưa vào lớp tiếp theo là đầu ra, lúc này xem như *Perceptron* 2 lớp và nó sẽ vẽ đường thẳng để phân biệt như hình 13.



Hình 13 – Dạng hình học của lớp hidden layer

Lớp *Hidden Layer* có nhiệm vụ là tách đặc trưng của dữ liệu (trường hợp trong đề tài này là ảnh xám). Làm thế nào để chọn số nốt trên một *Hidden Layer* và số *Hidden Layers* trên một mạng Nơron sẽ được giải quyết trong chương sau.

2.1.2 Mạng Nơron truyền thẳng - Feed forward Neural Network



Cấu trúc trên được gọi là một mạng Nơron truyền thẳng. Do cấu trúc này có 2 lớp *Hidden Layers* nên được gọi là *Cấu trúc học nhiều tầng - Deep Learning.* Ở *Cấu trúc học nhiều tầng* thì mỗi lớp *Hidden Layer* có nhiệm vụ tách đăng trưng của lớp trước đó, ví dụ như lớp H^1 có nhiệm vụ là tách biên của lớp trước, sau đó sẽ lấy dữ liệu mới (lúc này chỉ toàn là biên) để đưa vào lớp tiếp theo H^2, lớp này có thể làm nhiệm vụ là tìm ra góc, đường viền của ảnh, tiếp theo thì những thông tin từ lớp này sẽ được đưa vào lớp tiếp theo là lớp đầu ra, ở đây sẽ phân biệt tấm ảnh ban đầu thuộc lớp nào dựa vào những thông tin được tổng hợp lại [3].

Giả sử bây giờ chúng ta có giá trị tất cả trọng số nối các lớp lại thì trước khi chúng ta có thể kiểm tra xem tấm ảnh đưa vào là số mấy thì chúng ta cần định nghĩa hàm *softmax*:

Hàm *Softmax* nhận giá trị đầu vào là một vector x có n phần tử và trả về một vector có độ dài bằng với vector đầu vào và giá trị mỗi phần tử là:

$$x_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Đây là thuật toán để kiểm tra xem giá trị một tấm ảnh là số mấy:

Algorithm 3 Truyền thẳng (x)

1: Tính giá trị vector $z^{(1)} = \omega_1^T x + b^{(1)}$
2: Tính giá trị vector $h^{(1)} = \sigma(z^{(1)}$
3: Tính giá trị vector $z^{(2)} = \omega_2^T h^{(1)} + b^{(2)}$
4: Tính giá trị vector $h^{(2)} = \sigma(z^{(2)})$
5: Tính giá trị vector $z^{(3)} = \omega_3^T h^{(2)} + b^{(3)}$
6: Tính giá trị vector $o^{(3)} = \sigma(z^{(3)})$
7: Tính giá trị vector $res = Softmax(o^{(3)})$
8: Trả về vị trí có giá trị lớn nhất của vector res

2.1.3 Thuật toán lan truyền ngược: *Back Propagation*

Như đã trình bày trong thuật toán *Truyền thẳng* thì chúng ta cần phải có một bộ thông số giữa các lớp trong hệ thống. Nhưng chúng ta vẫn chưa đưa ra thuật toán để tìm ra bộ thông số đó. Để tìm được bộ thông số đó thì thuật toán *Lan truyền ngược* sẽ giải quyết được vấn đề. Trước khi đi vào thuật toán *Lan truyền ngược* ta cần đưa ra khái niệm khoảng cách *Kullback-Leibler* và *cross-entropy*

Khoảng cách Kullback-Leibler Khoảng cách Kullback-Leibler [3] dùng để đo sự khác biệt giữa 2 phân bố xác suất P và Q, có công thức:

$$D_{KL}(P||Q) = \sum_i P(i) log \frac{P(i)}{Q(i)}$$

$D_{KL}(P||Q)$ càng nhỏ thì phân bố xác suất P(X) và Q(X) càng giống nhau, nếu $D_{KL}(P||Q) = 0$ thì P(X) = Q(X).

Cross-Entropy Cross-Entropy [7] là một phần quan trọng của lĩnh vực *Lý thuyết thông tin*, dùng để đo lượng thông tin kì vọng giữa 2 phân bố xác suất P và Q, có công thức:

$$H(P, Q) = E_P[-logQ] = H(P) + D_{KL}(P||Q)$$

Thuật toán lan truyền ngược Khi đưa một tấm ảnh đi qua thuật toán *Truyền thẳng* (ở đây lấy ví dụ là hệ thống gồm 4 lớp trong đó có 2 lớp Hidden Layers, 1 lớp đầu vào và 1 lớp đầu ra) thì sẽ cho ra một vector $\hat{y}$. Nếu các trọng số đã được tối ưu hóa thì $\hat{y} \simeq y$ tức là hàm loss tính theo *bình phương sai số*: $loss = (y - \hat{y})^2 \simeq 0$. Nhưng nếu các trọng số chưa được tối ưu hóa thì $loss = (y - \hat{y})^2 \neq 0$, nếu muốn tiên đoán chính xác thì nhiệm vụ của chúng ta là cần tìm bộ thống số ω để làm cho hàm loss tiến về 0.

$loss = (y - \hat{y})^2 = (y - softmax(\omega_3^T \sigma(\omega_2^T \sigma(\omega_1^T x + b^{(1)}) + b^{(2)}) + b^{(3)}))$ (với ω_j là trọng số nối từ lớp j-1 tới lớp j)

Do hàm loss phụ thuộc vào các trọng số ω_i nên nếu ta thay đổi các thông số thì hàm loss sẽ bị thay đổi theo. Do vậy nếu ta dùng phương pháp để tối ưu thông số (như Gradient Descent) thì chắc chắn hàm loss sẽ tiến về 0 với một số vòng lặp và hệ số học nhất định. Thuật toán lan truyền ngược đã được chứng minh và dùng công thức tính đạo hàm theo chuỗi [4, 2]. Thuật toán để cho mạng Nơron học:

Algorithm 4 Huấn luyện mạng Nơron (x)

1: Khởi tạo các giá trị ω theo phân bố Gauss với mean = 0 và standard diviation = 1 và $b^{(l)} = 0$
2: $\hat{y}$ = Truyền thẳng (x)
3: Tính lỗi của lớp đầu ra: $\delta^L = \frac{\partial loss}{\partial h^L} \sigma'(z^L)$
4: Tính lỗi các lớp tiếp theo $\delta^l = ((\omega^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$
5: Đặt $\frac{\partial loss}{\partial b^l} = \delta^l$
6: Đặt $\frac{\partial loss}{\partial \omega^l} = h^{l-1} \delta^l$
7: *Vòng lặp*:
$b^{(l)} := b^{(l)} - \alpha \frac{\partial C}{\partial b^l}$
$\omega^{(l)} := \omega^{(l)} - \alpha \frac{\partial C}{\partial \omega^l}$

Dựa vào tính chất của hàm *sigmoid*, nếu giá trị z mà càng tiến về $+\infty$ hoặc là $-\infty$ thì đạo hàm tại điểm đó của hàm sigmoid sẽ bằng 0. Điều đó có nghĩa là δ^l sẽ bằng 0 dẫn đến giá trị của tham số ω^l và b^l sẽ không được cập nhật nữa (điều này là do giá trị khởi tạo chứ không phải do các giá trị thông số đã đạt tới điểm cực tiểu toàn cục). Để giải quyết được điều đó thì chúng ta dùng hàm *cross-entropy* làm hàm loss [2].

Tại sao Cross-Entropy? Khi đưa giá trị đầu vào là x đi qua hệ thống thì nó sẽ cho ra $\hat{y}$. Ví dụ đưa đầu vào là hình số 1 thì qua hệ thống nó sẽ cho ra một vector có dạng [0.01, 0.8, 0.02, 0.01, 0.02, 0.04, 0.05, 0.05, 0, 0] điều này có nghĩa là hệ thống đưa ra tiên đoán số này là số 0 với mức độ tự tin là 1%, 80% tự tin đây là số 1,... Trong khi đó giá trị vector y là [0, 1, 0, 0, 0, 0 ,0 ,0, 0, 0] điều này có nghĩa là y chắc chắn 100% đây là số 1 (do đây là dữ liệu đã được dán nhãn nên ta biết chắc chắn đây là số 1). Do đó $D_{KL}(y||\hat{y}) \approx 0$ (khoảng cách giữa phân bố xác suất y và $\hat{y}$ trên cùng những sự kiện là gần nhau), nếu $\hat{y}$ đưa tiên đoán là số khác thì $D_{KL}(y||\hat{y})$ sẽ lớn và chúng ta cần tối ưu hóa các thông số bằng cách làm $D_{KL}(y||\hat{y})$ tiến về 0 cho nên ta có $loss = D_{KL}(y||\hat{y})$

Mà ta có công thức cross-entropy: $H(y, \hat{y}) = E_y[-log\hat{y}] = H(y) + D_{KL}(y||\hat{y})$, H(y) là giá trị kì vọng của lượng thông tin trên phân bố y là không đổi khi ta cập nhật giá trị các thông số, điều đó có nghĩa là khi ta tối ưu $D_{KL}(y||\hat{y})$ thì không khác gì ta tối ưu hàm *Cross-Entropy* $H(y, \hat{y})$. Do đó ta có hàm loss = $H(y, \hat{y})$

Có thể thay thế hàm sigmoid bằng hàm khác? Khi ta dùng hàm loss là hàm *Cross-Entropy* thì có thể bỏ đi phần đạo hàm của hàm sigmoid, nhưng thức tế thì chỉ có thể bỏ đi vai trò của hàm sigmoid từ lớp đầu ra vào lớp kế tiếp, những lớp giữa vẫn tỉ lệ với đạo hàm của hàm sigmoid (do lan truyền ngược dùng quy tắc tính đạo hàm theo chuỗi), do đó ta cần dùng hàm kích hoạt khác làm sao mà bỏ đi tính tuyến tính của đầu vào lẫn không bão hòa, hàm đó có tên gọi là hàm *Rectified Linear Unit - ReLU* và có công thức $f(x) = max(0, x)$ [4]

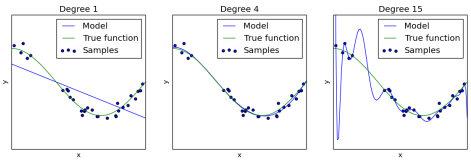
sức chứa của mạng Nơron Nếu trong một lớp *Hidden Layer* mà số nốt quá ít thì sẽ dẫn tới tình trạng *Under fitting* (mạng Nơron chỉ có thể vẽ được *hyper plane* nên sẽ không thể giảm giá trị hàm loss về giá trị tối ưu được), ngược lại, nếu số nốt nhiều quá sẽ dẫn tới tình trang *Over fitting* (Mạng Nơron vẽ đường phân biệt đi qua tất cả mọi điểm, điều đó có nghĩa là hàm loss = 0, nhưng điều đó dẫn tới việc Mạng nơron chỉ học trên tập huấn luyện và không thể khái quát nếu đưa ví dụ mới vào) [2]

giải quyết sức chứa của mạng Nơron

Để chống *Under fitting* thì ta có thể giải quyết bằng cách làm cho mạng nơron phức tạp lên, tức là thêm nhiều nốt trên một *Hidden Layer*

Để chống *Over fitting* thì ta có thể làm các cách sau:

1. Giảm độ phức tạp của mẫu (Giảm số nốt trên một *Hidden Layer*
2. Thêm nhiều dữ liệu
3. Dùng phương pháp L_2 regularization (weights decay) (cộng thêm $\sum_i \omega^2$ vào hàm loss và sau đó cập nhật thông số như cũ. Phương pháp này làm giảm sự ảnh hưởng của trọng số, nếu như = 0 thì nếu quá nhiều nốt trên một lớp thì dẫn đến nếu có sự thay đổi nhỏ của dữ liệu đầu vào thì kết quả sẽ thay đổi rất lớn nên ta phải giảm sự ảnh hưởng của trọng số để cho mạng Nơron chỉ học các tính chất của dữ liệu chứ không học nhiễu để có thể khái quát tốt với dữ liệu không có trong tập huấn luyện) (Hình 14)



Hình 14 – Sự ảnh hưởng của L_2 tới kết quả tiên đoán

2.2 Điểm yếu của mạng Nơron truyền thẳng

Thông thường trong một mạng Nơron thì số *Hidden Layers* càng nhiều thì mạng Nơron đó sẽ tách trích được nhiều đặc trưng của dữ liệu (ở đây là dữ liệu ảnh) điều đó có nghĩa là mạng Nơron sẽ khái quát tốt hơn ở tập dữ liệu mà nó chưa

thấy bao giờ. Trong ví dụ nhận diện kí tự viết tay thì hệ thống gồm 5 lớp, trong đó có 3 lớp *Hidden Layers*, mỗi lớp *Hidden Layers* có 500 *nốt*. Từ lớp thứ nhất nối lớp thứ hai có tổng cộng 784x500 + 500 thông số, từ lớp thứ hai tới lớp thứ ba có tổng cộng 500x500+500 thông số, lớp thứ ba tới lớp thứ tư là 500x500 + 500, lớp thứ tư tới lớp cuối là 500x10 + 10. Tổng cộng là 898510 giá trị. Mà trong *Python* thì sử dụng *double precision* nên giá trị một thông số là cần 24 bytes bộ nhớ (8 bytes để lưu giá trị, 8 bytes để lưu con trỏ trỏ tới kiểu giá trị và 8 bytes để lưu tham chiếu) và xấp xỉ 20.57 MB để lưu giá trị tham số của mạng Nơron đó.

Mạng Nơron truyền thẳng không thể đối phó được với trường hợp kí tự bị dịch chuyển hay kí tự bị teo nhỏ hoặc to ra được [3].

Để giải quyết được vấn đề đó, người ta sử dụng *Mạng Nơron tích chập* [3, 4, 2]

2.3 Mạng Nơron tích chập - Convolutional Neural Networks

Mạng Nơron tích chập cũng giống như *Mạng Nơron truyền thẳng* nhưng trọng số của nó lặp đi lặp lại (*sharing parameter*[1]). *Mạng Nơron tích chập* cực kì mạnh trong lĩnh vực *Thị giác máy tính* và mạng nơron này chuyên xử lí dữ liệu dạng khối (ảnh)

Trong *Mạng Nơron tích chập* có các thành phần chính sau:

1. Lớp Convolutional: dùng để chiết tách đặc trưng của dữ liệu đầu vào (lớp này nằm sau lớp dữ liệu đầu vào) có công dụng chống sự dịch chuyển của ảnh
2. Lớp Max Pooling: Lớp này dùng để chống lại sự teo nhỏ hay phóng to của ảnh
3. Lớp phân loại: Lớp này là mạng nơron truyền thẳng và nó nhận dữ liệu của lớp Max Pooling hoặc lớp Convolutional để tiên đoán dữ liệu đầu vào thuộc lớp nào

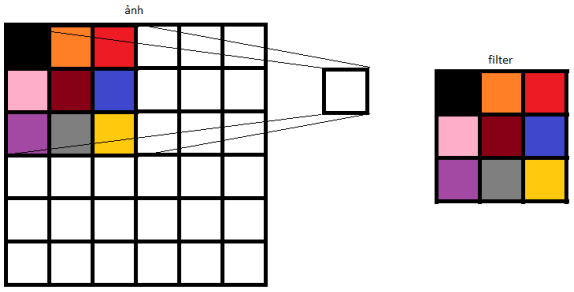
2.3.1 Lớp Convolutional

Lớp này dùng một bộ lọc chạy lướt qua dữ liệu của lớp trước đó và được biểu diễn như sau (Hình 15). Trong đó ta cho bộ lọc chạy khắp toàn bộ tấm ảnh và nốt mới sẽ có công thức sau:

$$C(i,j) = \sum_{a=0}^{k-1} \sum_{b=0}^{k-1} X(i-a, j-b) F(a,b)$$

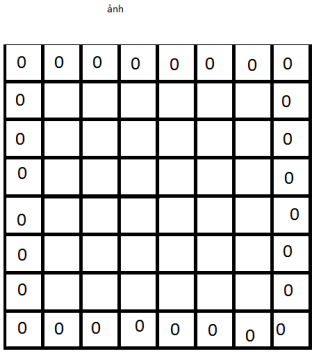
[1]Cho phép tất cả các điểm ảnh chia sẽ trọng số, nói cách khác, những trọng số sẽ học trên các điểm ảnh, dẫn đến là sẽ tìm ra mối quan hệ giữa các điểm ảnh

Trong đó X là dữ liệu ảnh đầu vào (vẫn giữ nguyên kích thước là tensor 3 chiều, chiều dài chiều rộng và chiều sâu = 1). F là kích thước của bộ lọc và là tensor 3 chiều (chiều dài = a, chiều rộng bằng b và chiều sâu = 1)



Hình 15 – Lớp Convolution

Do lớp Convolution dùng tích chập nên sau khi qua một lớp Convolution thì sẽ cho ra một tấm ảnh có kích thước nhỏ hơn kích thước ban đầu, do đó sau khi qua một số lớp nhất định thì ảnh sẽ bị teo lại và không chiết tách được nhiều đặc trưng. Để chống lại việc đó thì người ta thêm dùng phương pháp *padding*, đảm bảo sau một lớp *Convolution* thì đầu ra là ảnh vẫn giữa nguyên kích thước ban đầu (Hình 16).



Hình 16 – Padding

Sau khi bộ lọc lướt qua toàn bộ dữ liệu lớp trước đó và thực hiện tích chập sẽ cho một *Feature map*. Một lớp *Convolution* có thể có nhiều *Feature map* do có nhiều bộ lọc.

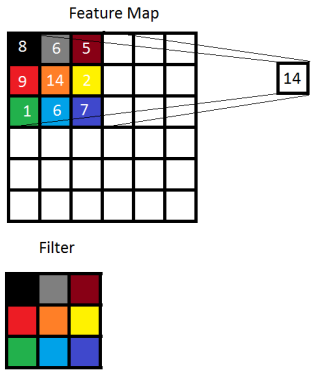
Sau khi tạo ra nhiều *Feature map* thì ta gói tất cả lại thành một tensor 3 chiều với chiều dài là chiều dài của *Feature map*, chiều rộng là chiều rộng của *Feature map* và chiều sâu là số *bộ lọc (Filters)*. Sang lớp *Convolution* kế tiếp thì ta dùng bộ lọc mới là tensor 3 chiều với chiều dài và chiều rộng là tùy chỉnh, còn chiều sâu của bộ lọc là chiều sâu của lớp trước đó (số lượng *Feature maps*)

Sau khi tạo ra *Features Maps* thì các giá trị của từng nốt trong *Features*

Map sẽ đi qua một hàm kích hoạt (sigmoid, tanh, ReLU,...) tạo thành một lớp *Features Maps* mới (do đây là một công đoạn bắt buộc nên xem như 2 lớp này là một)

2.3.2 Lớp Pooling

Lớp *Pooling* được đặt phía sau lớp *Convolution* và sử dụng *Max-Pooling*. Nhiệm vụ của lớp này là bỏ đi những tính chất không quan trọng, nên có thể chống được sự thay đổi về kích thước kí tự. Lớp Pooling làm việc như sau (Hình 17):



Hình 17 – Max-Pooling

Lớp Max-Pooling có công dụng cũng giống như lớp Convolution nhưng nó khác ở chỗ là không có tính toán tích chập gì cả, nó chỉ chọn ra giá trị lớn nhất của Feature map trong phạm vi của bộ lọc. Như hình trên sẽ cho ra giá trị đầu ra là 14

Nếu lớp *Convolution* có các trọng số trong bộ lọc thì ở lớp *Pooling* không có trọng số nào cả, ta chỉ cần xác định kích thước của bộ lọc và bước trượt của nó

2.3.3 Lớp Phân loại

Lớp phân loại này là một mạng Nơron truyền thẳng nhưng thay vì đưa ảnh vào từ đầu thì ở đây, đầu vào của mạng Nơron này là những tính chất quan trọng của ảnh và nhiệm vụ của nó là phân loại ảnh dựa trên những tính chất đó chứ không phải phân loại dựa trên những giá trị của điểm ảnh với các điểm ảnh không liên quan tới nhau.

2.3.4 Thuật toán lan truyền ngược trong CNNs

Trong mạng Nơron này thì thuật toán lan truyền ngược giống với thuật toán lan truyền ngược trong mạng Nơron truyền thẳng, và chỉ có lớp *Convolution* và lớp *Phân loại* là có trọng số nên gradients mới biến đổi, còn lớp Pooling thì không có trọng số gì hết nên gradients truyền thẳng qua và không thay đổi gì.

References

[1] Andrew Ng. *Machine Learning*. 2012. URL: https://www.coursera.org/learn/machine-learning.

[2] Michael A. Nielsen. *Neural Networks and Deep Learning*. http://neuralnetworksanddeeplearning.com/. Determination Press, 2015.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[4] Fei-Fei Li, Justin Johnson, and Andrej Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition*. 2016. URL: http://cs231n.stanford.edu/.

[5] Chirs Olah. *Neural Networks, Manifolds, and Topology*. URL: http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/.

[6] Wikipedia. *AlphaGo*. URL: https://en.wikipedia.org/wiki/AlphaGo.

[7] Wikipedia. *Cross entropy*. URL: https://en.wikipedia.org/wiki/Cross_entropy.

[8] Wikipedia. *Deep Blue versus Garry Kasparov*. URL: https://en.wikipedia.org/wiki/Deep_Blue_versus_Garry_Kasparov.