

TRƯỜNG ĐẠI HỌC BÁCH KHOA - ĐẠI HỌC QUỐC GIA TP.HCM
KHOA DIỆN - DIỆN TỬ
BỘ MÔN ĐIỀU KHIỂN VÀ TỰ ĐỘNG HÓA



BÀI TẬP VỀ NHÀ

Trí tuệ nhân tạo trong điều khiển

GVHD: TS. Phạm Việt Cường

Sinh viên: Nhóm 11 - L02

Nguyễn Thành Trung - 1814514

Phan Nguyên Trung - 1814519



Mục lục

1	Đề bài	2
1.1	Bài 1	2
1.2	Bài 2	2
1.3	Bài 3	2
1.4	Bài 4	3
1.5	Bài 5	3
1.6	Bài 6	3
1.7	Bài 7	3
1.8	Bài 8	3
1.9	Bài 9	3
1.10	Bài 10	3
1.11	Bài 11	4
1.12	Bài 12	4
1.13	Bài 13	4
1.14	Bài 14	4
2	Bài làm	5
2.1	Bài 1	5
2.1.1	Thực hiện thuật toán	5
2.1.2	Thử nghiệm với bộ dữ liệu phân tách tuyến tính	6
2.1.3	Thử nghiệm với bộ dữ liệu không phân tách tuyến tính	9
2.1.4	Thực hiện với bộ dữ liệu hoa iris	11
2.2	Bài 2	13
2.2.1	Đạo hàm riêng phần	13
2.2.2	Thực hiện tính toán	14
2.2.3	Kiểm tra kết quả tính toán	16
2.3	Bài 3	18
2.4	Bài 4	19
2.4.1	Gradient Descent cho hàm nhiều biến	19
2.4.2	Sử dụng GD tìm cực tiểu của hàm số 1 biến	19
2.4.3	Sử dụng GD tìm cực tiểu của hàm số $f_1(x, y)$	25
2.4.4	Sử dụng GD tìm cực tiểu của hàm số $f_2(x, y)$	31
2.4.5	Nhận xét	37
2.5	Bài 5	38
2.6	Bài 6	39
2.7	Bài 7	40
2.7.1	Cài đặt các thư viện	40
2.7.2	Đọc ảnh và gán nhãn cho ảnh	40
2.7.3	Tiền xử lý dữ liệu	40



2.7.4	Tải các trọng số của mạng Alexnet	41
2.7.5	Thực hiện kết quả với mạng Alexnet	42
2.7.6	Kết quả thực hiện trên mô hình Inception v3	43
2.7.7	Kết quả thực hiện trên mô hình Resnet18	44
2.7.8	So sánh kết quả	45
2.8	Bài 8	46
2.8.1	Thực hiện với Alexnet	46
2.8.2	Thực hiện với Resnet18	49
2.8.3	Thực hiện với GoogleNet	50
2.8.4	Nhận xét	51
2.9	Bài 9	53
2.9.1	Cài đặt các thư viện	53
2.9.2	Xử lí dữ liệu ảnh	53
2.9.3	Thay đổi mô hình Alexnet	54
2.9.4	Huấn luyện mô hình	55
2.9.5	Dánh giá	59
2.10	Bài 10	60
2.10.1	Sử dụng GA (Genetic Algorithm) để tối ưu hóa hàm mất mát của mạng nơron hồi tiếp (RNN)	60
2.10.2	Nhận xét	64
2.11	Bài 11	65
2.12	Bài 12	66
2.13	Bài 13	67
2.13.1	Sử dụng GA (Genetic Algorithm) để tối ưu hóa hàm mất mát của mạng LSTM	67
2.13.2	Nhận xét	71
2.14	Bài 14	72
2.14.1	Chuẩn hóa dữ liệu	72
2.14.2	Thực hiện bài toán Linear Regression	73
2.14.2.1	Thực hiện bài toán sử dụng Gradient Descent với dữ liệu chưa được chuẩn hóa:	73
2.14.2.2	Thực hiện bài toán sử dụng Gradient Descent với dữ liệu chưa được chuẩn hóa:	74
2.14.2.3	Thực hiện kết quả bằng lời giải giải tích	75



Bảng đánh giá công việc

Họ và tên	Bài tập thực hiện	Tỉ lệ đóng góp
Nguyễn Thanh Trung	HW1, HW2, HW3, HW5, HW7, HW8, HW9, HW14	50%
Phan Nguyên Trung	HW4, HW6, HW10, HW11, HW12, HW13	50%

Bài tập	Sinh viên thực hiện	Kết quả đạt được
HW1	Nguyễn Thanh Trung	Thực hiện được thuật toán Perceptron. Khảo sát thuật toán trên tập dữ liệu hoa iris.
HW2	Nguyễn Thanh Trung	Thiết lập công thức tính đạo hàm trong quá trình lan truyền ngược của mạng ANN. Kiểm tra lại kết quả với một ví dụ thử nghiệm
HW3	Nguyễn Thanh Trung	Giải thích các câu hỏi về SGD, Batch GD và Mini-Batch GD.
HW4	Phan Nguyên Trung	Thực hiện thuật toán Gradient Descent để tìm cực tiểu toàn cục của các hàm không lồi.
HW5	Nguyễn Thanh Trung	Giải thích câu hỏi về việc chia sẻ trọng số và kết nối địa phương dùng trong mạng CNN.
HW6	Phan Nguyên Trung	Tính toán được thông số của mạng Alexnet.
HW7	Nguyễn Thanh Trung	Tính tỉ lệ lỗi top-1 và top-5 của mạng Alexnet và 2 mạng khác với các lớp thuộc tập Imagenet.
HW8	Nguyễn Thanh Trung	Kiểm tra hoạt động của mạng Alexnet và 2 mạng khác với ảnh bị điều chỉnh tỉ lệ kích thước.
HW9	Nguyễn Thanh Trung	Thực hiện transfer-learning với mạng Alexnet.
HW10	Phan Nguyên Trung	Thực hiện thuật toán GA với mạng RNN.
HW11	Phan Nguyên Trung	Giải thích câu hỏi về mã hóa dữ liệu.
HW12	Phan Nguyên Trung	Giải thích câu hỏi về mạng LSTM.
HW13	Phan Nguyên Trung	Thực hiện thuật toán GA với mạng LSTM.
HW14	Nguyễn Thanh Trung	Thực hiện chuẩn hóa dữ liệu về kỳ vọng 0, phương sai 1 Ứng dụng vào giải quyết bài toán Linear Regression.

1 Đề bài

1.1 Bài 1

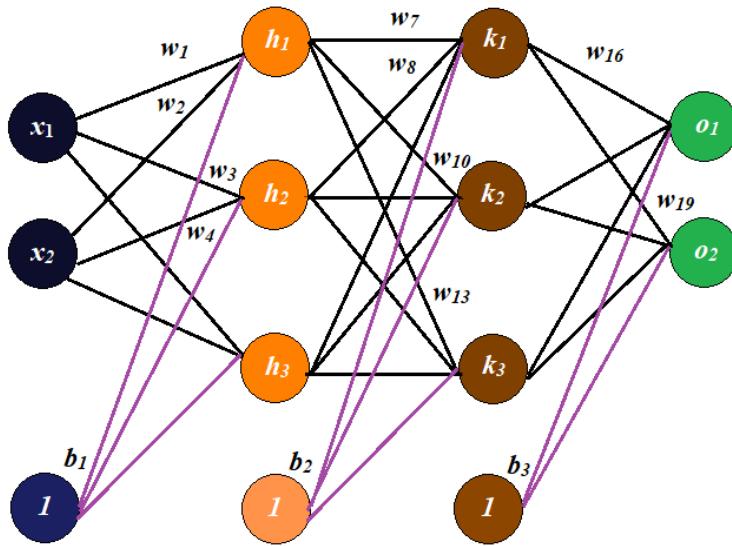
Thực hiện **thuật toán perceptron** bằng ngôn ngữ Python.

1.2 Bài 2

Thực hiện tính đạo hàm riêng phần cho mạng Neural Network có:

- Có 2 giá trị ngõ vào
- Có 2 hidden layer, mỗi hidden layer có 3 node
- Có 2 giá trị ngõ ra

Chọn hàm kích hoạt là hàm sigmoid.



1.3 Bài 3

- Giải thích tại sao đối với batch gradient descent khi cập nhật trọng số ta đem trừ vector gradient thì sẽ đi theo đường dốc nhất, tìm cực trị rất hiệu quả. Tuy nhiên đối với stochastic gradient descent thì lại không phải đường dốc nhất.



- Tại sao stochastic gradient descent có tính chất không ổn định thì có thể tránh được việc rơi vào cực trị địa phương.
- Giải thích tại sao phương pháp mini batch gradient descent lại là phương pháp tốt nhất trong ba phương pháp.

1.4 Bài 4

Tìm cực trị toàn cục của hai hàm không lồi $f(x, y): \mathbb{R}^2 \rightarrow \mathbb{R}$. Sử dụng các phương pháp gradient descent để cập nhật trọng số.

- Khảo sát ít nhất 5 điểm khởi tạo.
- Thay đổi hệ số học..
- Vẽ kết quả thu được.

1.5 Bài 5

Giải thích ý nghĩa của việc sử dụng **kết nối địa phương (local connectivity)** và **trọng số dùng chung (shared weights)** trong mạng nơ-ron tích chập (CNN).

1.6 Bài 6

Tính toán số lượng thông số của mạng **AlexNet**.

1.7 Bài 7

Chọn 100 ảnh thuộc ít nhất 5 lớp trong tập Imagenet. Sau đó sử dụng mạng AlexNet và thêm 2 mạng mới nữa để tính tỉ lệ lỗi top 1 và top 5.

1.8 Bài 8

Sử dụng các ảnh kích thước lớn hơn 224×224 , kiểm tra hoạt động của Alexnet và 2 mạng khác sau khi thay đổi tỉ lệ kích thước ảnh.

1.9 Bài 9

Sử dụng học chuyển tiếp với mạng Alexnet để huấn luyện mô hình gồm 100 tấm ảnh, thuộc ít nhất 5 lớp khác nhau.

1.10 Bài 10

Sử dụng GA (Genetic Algorithm) để tối ưu hóa hàm mất mát của mạng nơ-ron hồi tiếp (RNN) khi huấn luyện model học với từ "hello".



1.11 Bài 11

Tại sao ta lại sử dụng mã hóa one-hot cho trường hợp huấn luyện mạng LSTM.

1.12 Bài 12

Đối với mạng LSTM, những thành phần nào là long term và chõ nào là short term. Tại sao nó có khả năng là long term và short term. Thành phần C_t trong mạng LSTM có ý nghĩa gì?

1.13 Bài 13

Dùng GA (Genetic Algorithm) để tối ưu hóa hàm mất mát của mạng LSTM khi huấn luyện model học với từ "hello".

1.14 Bài 14

Thực hiện chuẩn hóa dữ liệu về zero mean and unit variance.

2 Bài làm

Tất cả source code và hình ảnh liên quan đều được lưu tại [github](#) của nhóm.

2.1 Bài 1

2.1.1 Thực hiện thuật toán

Thực hiện lập trình thuật toán perceptron như sau:

Algorithm: Perceptron Learning Algorithm

```
P ← inputs with label 1;
N ← inputs with label 0;
Initialize w randomly;
while !convergence do
    Pick random x ∈ P ∪ N ;
    if x ∈ P and w.x < 0 then
        | w = w + x ;
    end
    if x ∈ N and w.x ≥ 0 then
        | w = w - x ;
    end
end
//the algorithm converges when all the
inputs are classified correctly
```

Doạn chương trình cho thuật toán như sau:

Đầu vào của hàm là dữ liệu X và nhãn y, đồng thời khởi tạo bộ trọng số ban đầu

```
In [1]: import numpy as np
In [2]: def perceptron(X, y, w_init, it=100):
In [3]: w = [w_init]
In [4]: i = 1
```

Thực hiện lặp liên tục kiểm tra cho từng điểm dữ liệu. Đầu tiên, ta sử dụng hàm $sgn(w, x)$ để tính toán giá trị ngõ ra dự đoán cho từng điểm dữ liệu. Tuy nhiên, do giá trị ngõ ra của hàm $sgn(w, x)$ là $-1, 0$ và 1 nên tiến hành gán lại nhãn dữ liệu chỉ có hai nhãn là 0 và 1 .

```
In [5]: while True:
In [6]:     pred = sgn(w[-1], X).reshape(-1)
In [7]:     pred[pred >= 0] = 1
```

In [8]: `pred[pred < 0] = 0`

Thực hiện tìm vị trí (**id**) của các điểm dữ liệu bị dự đoán sai. Nếu không có điểm dữ liệu nào dự đoán sai hoặc số vòng lặp bằng với số vòng lặp tối đa *it* thì dừng thuật toán. Kết quả trả về gồm có danh sách các trọng số, số điểm bị sai và số lượng vòng lặp khi kết thúc thuật toán. Nếu chưa kết thúc thuật toán, ta sẽ tiến hành cập nhật trọng số ở bước tiếp theo.

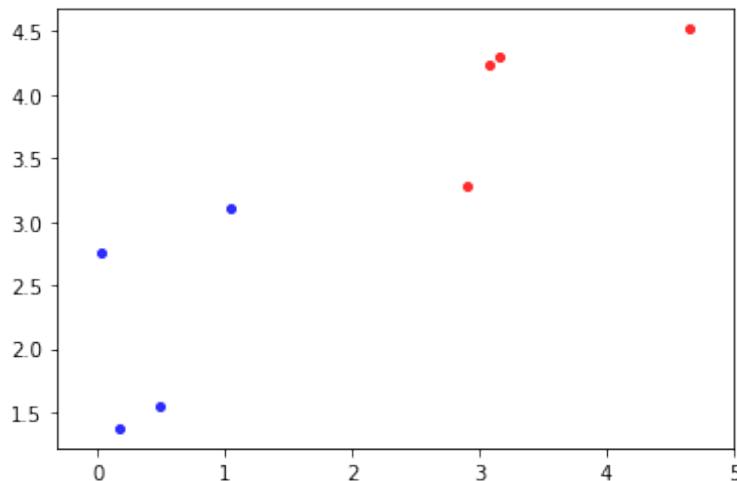
In [9]: `mis_id = np.where(np.equal(pred, y) == False)[0]`
In [10]: `num_mis = mis_id.shape[0]`
In [11]: `if (num_mis == 0) or (i == it):`
In [12]: `return w, mis_id.size, i`

Để tiến hành cập nhật trọng số, ta sẽ tiến hành chọn ngẫu nhiên một điểm trong tập dữ liệu bị sai. Sau đó tiến hành cập nhật trọng số theo dòng lệnh 14.

In [14]: `random_id = np.random.choice(mis_id, 1)[0]`
In [14]: `w_t = w[-1] + (2 * y[random_id] - 1) * X[:, random_id].reshape(-1)`
In [15]: `w.append(w_t)`

2.1.2 Thủ nghiệm với bộ dữ liệu phân tách tuyến tính

Trước tiên, ta sẽ tiến hành tạo giả lập bộ dữ liệu phân tách tuyến tính như hình 2.1.1. Các điểm màu xanh được gán nhãn 0, các điểm màu đỏ được gán nhãn 1.



Hình 2.1.1: Dữ liệu phân tách tuyến tính 1

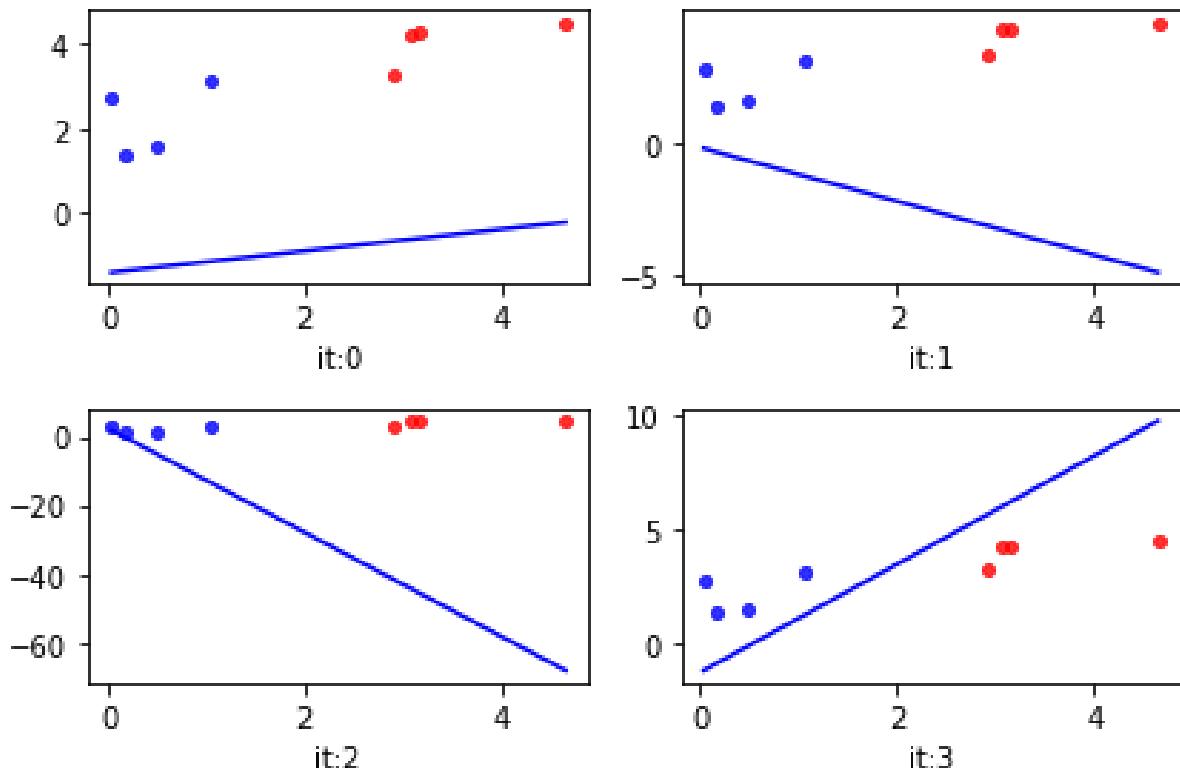
Sau đó thực hiện lệnh để lấy được bộ trọng số:

In [1]: `w, mis_id, it = perceptron(X_bar, y, w_init)`
Out [1]: `w = [-1.4639, 2.7929, -1.1795]`
Out [1]: `mis_point = 0, it = 4`

Kết quả thực hiện việc cập nhật trọng số sau mỗi lần lặp được thể hiện như hình 2.1.2. Ban đầu trong *it* = 0, trọng số khởi tạo không thể phân tách được dữ liệu, tất cả 4 điểm màu xanh đều bị phân tách sai nghĩa là $w \cdot x_{blue} > 0$. Thông qua việc cập nhật trọng số $w_{new} = w_{old} - x_{blue}$ thì giá trị $w_{new} \cdot x_{blue} = w_{old} \cdot x_{blue} - x_{blue} \cdot x_{blue}$ ¹ sẽ nhỏ hơn, nghĩa là bộ trọng số w_{new} sẽ tiến

¹Công thức ở đây bỏ qua sự chính xác về mặt phép toán đối với kích thước của ma trận

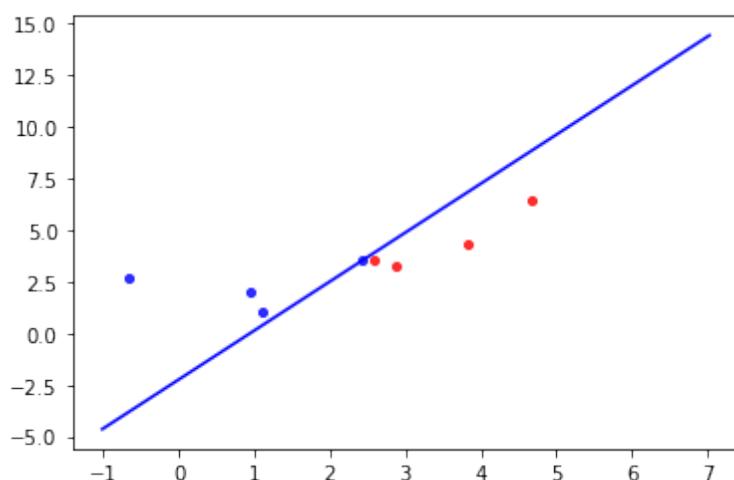
gần hơn đến việc phân tách được dữ liệu. Thực vậy, thông qua vòng lặp $it = 1, it = 2$ ta nhận thấy đường thẳng có xu hướng tiến lại gần vùng phân tách giữa hai loại điểm dữ liệu. Và đến $it = 3$, ta đã tìm được bộ thông số w phân tách hoàn toàn được hai điểm dữ liệu.



Hình 2.1.2: Kết quả thực hiện với bộ dữ liệu phân tách tuyến tính 1

Kết quả sau thu được thì sau 4 vòng lặp, đường thẳng phân tách dữ liệu sẽ có dạng $-1.4639 + 2.7929 * x_1 - 1.1795 * x_2 = 0$ và đường thẳng này phân biệt được hoàn toàn tập dữ liệu.

Tiếp theo, ta thử nghiệm với trường hợp hai điểm dữ liệu ở hai lớp khá khít nhau.



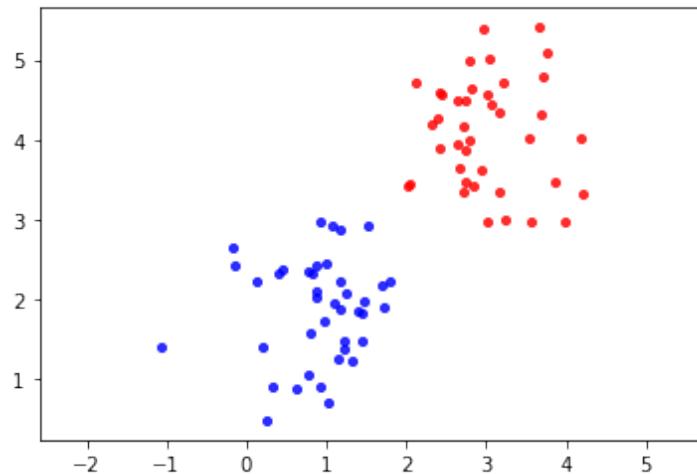
Hình 2.1.3: Kết quả thực hiện với bộ dữ liệu phân tách tuyến tính 2

Sau đó thực hiện lệnh để lấy được bộ trọng số, kết quả thu được như hình 2.1.3.

```
In [1]: w, mis_id, it = perceptron(X_bar, y, w_init)
Out [1]: w = [-22.1209, 23.4257, -9.8692]
Out [1]: mis_point = 0, it = 167
```

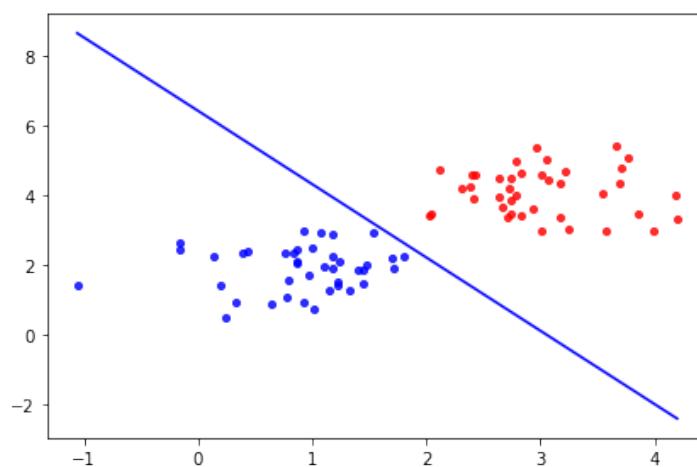
Như vậy, đường thẳng cũng phân biệt được hai điểm dữ liệu. Tuy nhiên lúc này số lượng vòng lặp it lớn hơn lên đến 167 vòng.

Thực hiện đổi với dữ liệu có nhiều điểm dữ liệu hơn (40 điểm cho mỗi lớp) như hình 2.1.4.



Hình 2.1.4: Bộ dữ liệu phân tách tuyến tính 3

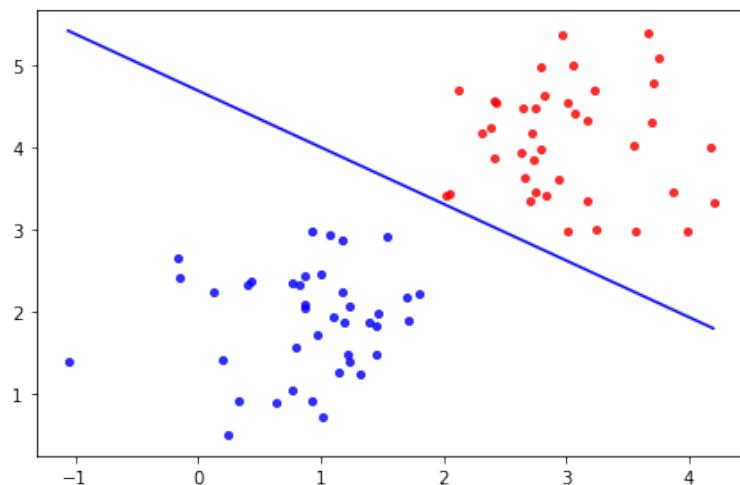
```
In [1]: w, mis_id, it = perceptron(X_bar, y, w_init)
Out [1]: w = [-9.1928, 3.0162, 1.4328]
Out [1]: mis_point = 0, it = 23
```



Hình 2.1.5: Kết quả thực bộ dữ liệu phân tách tuyến tính 3

Thực hiện chạy thêm một kết quả khác

```
In [1]: w, mis_id, it = perceptron(X_bar, y, w_init)
Out [1]: w = [-4.9033, 0.7214, 1.1044]
Out [1]: mis_point = 0, it = 16
```

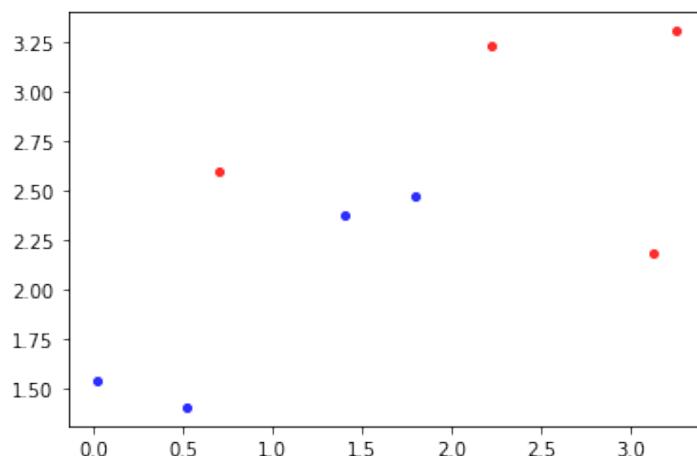


Hình 2.1.6: Kết quả thực bộ dữ liệu phân tách tuyến tính 3

Như vậy, bài toán perceptron có rất nhiều lời giải, đường thẳng (mặt phẳng, siêu phẳng) tìm được chưa phải lời giải tối ưu nhất. Số vòng lặp để ra kết quả phụ thuộc vào bộ trọng số khởi tạo đầu, phụ thuộc vào dữ liệu và việc lấy ngẫu nhiên dữ liệu x để cập nhật trọng số.

2.1.3 Thử nghiệm với bộ dữ liệu không phân tách tuyến tính

Giống như trên, ta tiến hành tạo giả lập bộ dữ liệu không phân tách tuyến tính như hình 2.1.7. Các điểm màu xanh được gán nhãn 0, các điểm màu đỏ được gán nhãn 1.



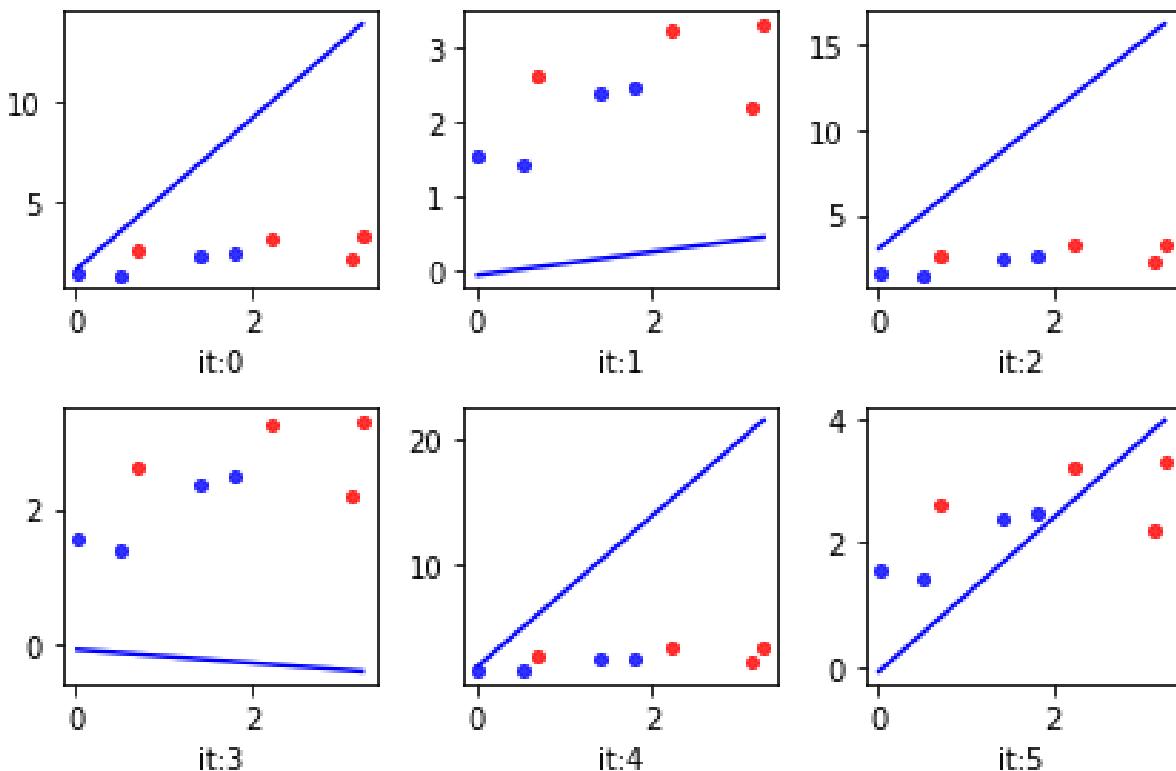
Hình 2.1.7: Bộ dữ liệu không phân tách tuyến tính

Để dễ theo dõi kết quả, ta sẽ giới hạn số lần lặp tối đa là 6. Kết quả thu được như sau:

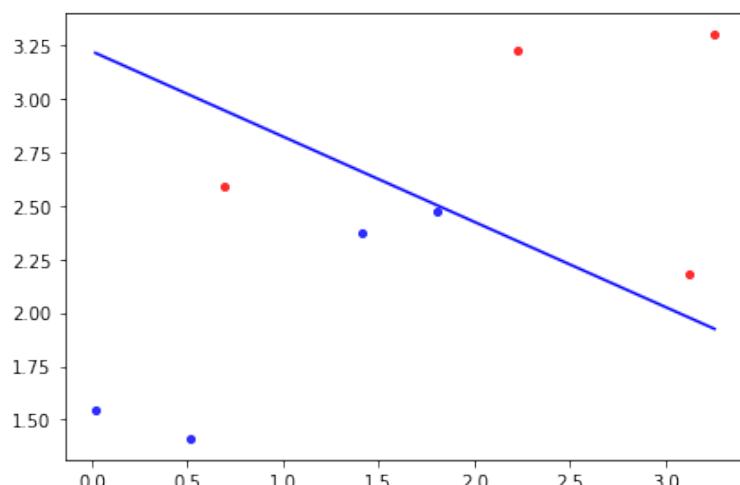
```
In [1]: w, mis_id, it = perceptron(X_bar, y, w_init, it=6)
Out [1]: w = [-0.1823, 2.3468, -1.8808]
Out [1]: mis_point = 2, it = 6
```

Ta sẽ tăng số vòng lặp lên 10000, kết quả thu được như sau:

```
In [1]: w, mis_id, it = perceptron(X_bar, y, w_init, it=6)
Out [1]: w = [-47.5722, 5.8776, 14.7710]
Out [1]: mis_point = 1, it = 10000
```



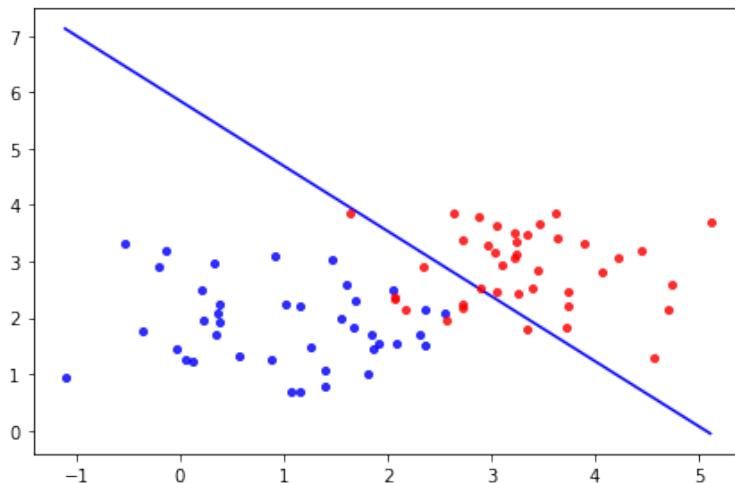
Hình 2.1.8: Kết quả thực hiện trên bộ dữ liệu không phân tách tuyến tính với 6 vòng lặp



Hình 2.1.9: Kết quả thực hiện trên bộ dữ liệu không phân tách tuyến tính với 1000 vòng lặp

Thực hiện đối với dữ liệu có nhiều điểm dữ liệu hơn (40 điểm cho mỗi lớp) thu được kết quả như hình 2.1.10.

```
In [1]: w, mis_id, it = perceptron(X_bar, y, w_init, it=6)
Out [1]: w = [-55.1951, 10.8940, 9.4406]
Out [1]: mis_point = 9, it = 1000
```



Hình 2.1.10: Kết quả thực hiện trên bộ dữ liệu không phân tách tuyến tính

Như vậy, đối với bộ dữ liệu không phân tách tuyến tính thuật toán perceptron không thể phân tách được.

2.1.4 Thực hiện với bộ dữ liệu hoa iris

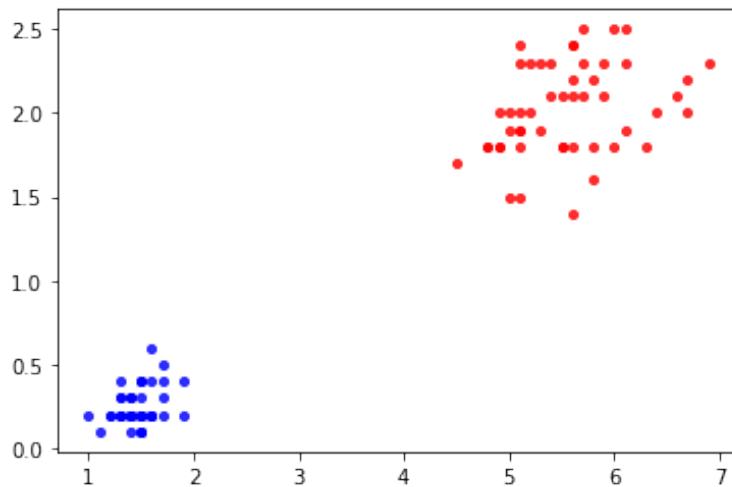
Bộ dữ liệu này gồm 3 loài hoa iris là setosa, versicolor, virginica. Mỗi loại có 50 mẫu với 4 đặc trưng là chiều dài đài hoa, chiều rộng đài hoa, chiều dài cánh hoa, chiều rộng cánh hoa. Tập dữ liệu này có thể được tìm thấy [tại đây](#).

Ở bài toán này, ta chỉ phân biệt hai loại là **setosa** và **virginica** vì các thông số khá sai lệch nhau nên có thể có khả năng phân biệt tuyến tính.

SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5	3.6	1.4	0.2	Iris-setosa
6.3	3.3	6	2.5	Iris-virginica
5.8	2.7	5.1	1.9	Iris-virginica
7.1	3	5.9	2.1	Iris-virginica
6.3	2.9	5.6	1.8	Iris-virginica
6.5	3	5.8	2.2	Iris-virginica



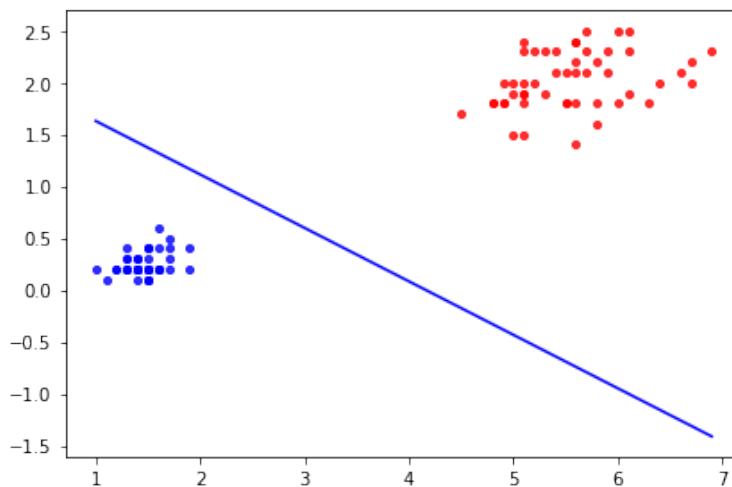
Trước tiên, ta xét hai đặc trưng là chiều dài cánh hoa và chiều rộng cánh hoa. Hai đặc trưng này khi biểu diễn ra không gian 2D phân biệt tuyến tính. Hoa **sentosa** được gán nhãn dữ liệu 0 (màu xanh), hoa **virginica** được gán nhãn dữ liệu 1 (màu đỏ).



Hình 2.1.11: Đặc trưng chiều dài cánh hoa và chiều rộng cánh hoa.

Kết quả thu được như sau:

```
In [1]: w, mis_id, it = perceptron(X_bar, y, w_init)
Out [1]: w = [-4.0391, 0.9709, 1.8829]
Out [1]: mis_point = 0, it = 6
```



Hình 2.1.12: Kết quả thực hiện.

Kết quả thực hiện trên cả 4 đặc trưng:

```
In [1]: w, mis_id, it = perceptron(X_bar, y, w_init)
Out [1]: w = [-0.1481, -1.6783, -5.4281, 6.3407, 4.1070]
Out [1]: mis_point = 0, it = 6
```

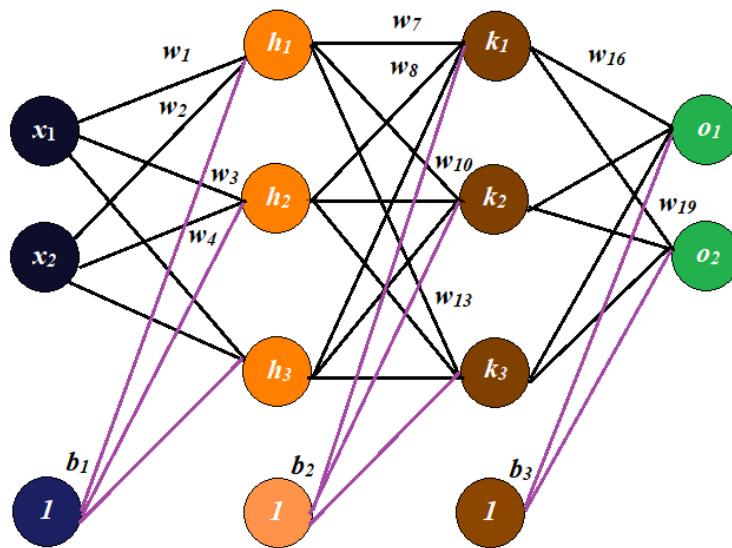
Như vậy, thuật toán perceptron có thể phân loại được 2 loài hoa **sentosa** và **virginica** thông qua hai đặc trưng về chiều dài cánh hoa và chiều rộng cánh hoa hoặc cả bốn đặc trưng.

2.2 Bài 2

Thực hiện tính đạo hàm riêng phần cho mạng Neural Network có:

- Có 2 giá trị ngõ vào
- Có 2 hidden layer, mỗi hidden layer có 3 node
- Có 2 giá trị ngõ ra

Chọn hàm kích hoạt là hàm sigmoid $f(x) = \frac{1}{1+e^{-x}}$.



2.2.1 Đạo hàm riêng phần

Quá trình truyền thuận:

$$net_{h1} = z_{h1} = w_1x_1 + w_2x_2 + b_1 \quad (2.1)$$

$$h_1 = f(z_{h1}) \quad (2.2)$$

$$net_{k1} = z_{k1} = w_7h_1 + w_8h_2 + w_9h_3 + b_2 \quad (2.3)$$

$$k_1 = f(z_{k1}) \quad (2.4)$$

$$net_{o1} = z_{o1} = w_{16}k_1 + w_{17}k_2 + w_{18}k_3 + b_3 \quad (2.5)$$

$$o_1 = f(z_{o1}) \quad (2.6)$$

Hàm mất mát:

$$E = \frac{1}{2} \times [(y_{o1} - o_1)^2 + (y_{o2} - o_2)^2] \quad (2.7)$$



Quá trình lan truyền ngược:

Tại node o_1 :

$$\frac{\partial E}{\partial z_{o1}} = \frac{\partial E}{\partial o_1} \frac{\partial o_1}{\partial z_{o1}} = -(y_{o1} - o_1) f(z_{o1}) [1 - f(z_{o1})] \quad (2.8)$$

Từ công thức (2.8), ta có:

$$\frac{\partial E}{\partial w_{16}} = \frac{\partial E}{\partial z_{o1}} \frac{\partial z_{o1}}{\partial w_{16}} = -(y_{o1} - o_1) f(z_{o1}) [1 - f(z_{o1})] k_1$$

Tại node k_1 :

$$\frac{\partial E}{\partial z_{k1}} = \left(\frac{\partial E}{\partial z_{o1}} \frac{\partial z_{o1}}{\partial k_1} + \frac{\partial E}{\partial z_{o2}} \frac{\partial z_{o2}}{\partial k_1} \right) \frac{\partial k_1}{\partial z_{k1}} = \left(\frac{\partial E}{\partial z_{o1}} w_{16} + \frac{\partial E}{\partial z_{o2}} w_{19} \right) f(z_{k1}) [1 - f(z_{k1})] \quad (2.9)$$

Từ công thức (2.9), ta có:

$$\frac{\partial E}{\partial w_7} = \frac{\partial E}{\partial z_{k1}} \frac{\partial z_{k1}}{\partial w_7} = \left(\frac{\partial E}{\partial z_{o1}} w_{16} + \frac{\partial E}{\partial z_{o2}} w_{19} \right) f(z_{k1}) [1 - f(z_{k1})] \times h_1$$

Tại node h_1 :

$$\begin{aligned} \frac{\partial E}{\partial z_{h1}} &= \left(\frac{\partial E}{\partial z_{k1}} \frac{\partial z_{k1}}{\partial h_1} + \frac{\partial E}{\partial z_{k2}} \frac{\partial z_{k2}}{\partial h_1} + \frac{\partial E}{\partial z_{k3}} \frac{\partial z_{k3}}{\partial h_1} \right) \frac{\partial h_1}{\partial z_{h1}} \\ &= \left(\frac{\partial E}{\partial z_{k1}} w_7 + \frac{\partial E}{\partial z_{k2}} w_{10} + \frac{\partial E}{\partial z_{k3}} w_{13} \right) f(z_{h1}) [1 - f(z_{h1})] \end{aligned} \quad (2.10)$$

Từ công thức (2.10), ta có:

$$\frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial z_{h1}} \frac{\partial z_{h1}}{\partial w_1} = \left(\frac{\partial E}{\partial z_{k1}} w_7 + \frac{\partial E}{\partial z_{k2}} w_{10} + \frac{\partial E}{\partial z_{k3}} w_{13} \right) f(z_{h1}) [1 - f(z_{h1})] \times x_1 \quad (2.11)$$

2.2.2 Thực hiện tính toán

Giả sử ta có dữ liệu $x = [0.1, 0.2]$ và nhãn $y = [1.0, 0.0]$. Các trọng số được khởi tạo là:

- $[w_1, w_2, w_3, w_4, w_5, w_6] = [0.1, 0.3, 0.4, 0.2, 0.15, 0.6]$
- $[w_7, w_8, w_9, w_{10}, w_{11}, w_{12}, w_{13}, w_{14}, w_{15}] = [0.4, 0.25, 0.15, 0.4, 0.2, 0.3, 0.1, 0.25, 0.15]$
- $[w_{16}, w_{17}, w_{18}, w_{19}, w_{20}, w_{21}] = [0.55, 0.1, 0.25, 0.45, 0.7, 0.4]$
- $[b_1, b_2, b_3] = [0.6, 0.6, 0.6]$



Trước tiên, ta khai báo các thông số đầu vào:

```
In [1]: x = np.array([0.1, 0.2])
In [2]: W1 = np.array([0.1, 0.3, 0.4, 0.2, 0.15, 0.6]).reshape(3,2)
In [3]: W2 = np.array([0.4, 0.25, 0.15, 0.4, 0.2, 0.3, 0.1, 0.25, 0.15])
In [4]: W2 = W2.reshape(3,3)
In [5]: W3 = np.array([0.55, 0.1, 0.25, 0.45, 0.7, 0.4]).reshape(2,3)
In [6]: b = np.array([0.6, 0.6, 0.6])
```

Giá trị tại lớp **node h**:

```
In [1]: z_h = W1.dot(x) + b[0]
In [2]: h = sigmoid(z_h)
Out [1]: z_h = [0.67, 0.68, 0.735]
Out [2]: h = [0.66150316, 0.6637387, 0.67590153]
```

Giá trị tại lớp **node k**:

```
In [1]: z_k = W2.dot(h) + b[1]
In [2]: k = sigmoid(z_k)
Out [1]: z_k = [1.13192117, 1.20011946, 0.93347022]
Out [2]: k = [0.75619327, 0.76854603, 0.71777879]
```

Giá trị tại lớp **node o**:

```
In [1]: z_o = W3.dot(k) + b[2]
In [2]: o = sigmoid(z_o)
Out [1]: z_o = [1.2722056, 1.76538071]
Out [2]: o = [0.78112008, 0.85388228]
```

Giá trị hàm mất mát theo phương trình (2.7):

$$E = \frac{1}{2} \times [(1.0 - 0.78112008)^2 + (0.0 - 0.85388228)^2] = 0.38851168132019515$$

Tiếp theo, ta thay đổi w_1 một lượng nhỏ 0.001 ta sẽ có $w_1 = 0.099$. Tương tự ta cũng có kết quả như sau;

```
Out [1]: z_h = [0.6699, 0.68, 0.735]
Out [2]: h = [0.66148077, 0.6637387, 0.67590153]
Out [3]: z_k = [1.13191221, 1.2001105, 0.93346798]
Out [4]: k = [0.75619162, 0.76854444, 0.71777833]
Out [5]: z_o = [1.27220442, 1.76537867]
Out [6]: o = [0.78111987, 0.85388202]
```

$$E' = \frac{1}{2} \times [(1.0 - 0.78111987)^2 + (0.0 - 0.85388202)^2] = 0.3885115081980104$$

Ta có thể tính sấp xỉ đạo hàm:

$$\frac{\partial E}{\partial w_1} = \frac{E - E'}{\Delta w_1} = \frac{(0.38851168132019515 - 0.3885115081980104)}{0.001} = 0.000173122 \quad (2.12)$$

Tương tự, chúng ta cũng khảo sát cho w_{16} và w_7 . Tuy nhiên, ta sẽ tăng w_{16} và w_7 lên 0.001 thay vì giảm đi 0.001.

$$\frac{\partial E}{\partial w_7} = \frac{E - E'}{\Delta w_7} = \frac{(0.38851168132019515 - 0.38851501745815437)}{-0.001} = 0.003336138 \quad (2.13)$$

$$\frac{\partial E}{\partial w_{16}} = \frac{E - E'}{\Delta w_{16}} = \frac{(0.38851168132019515 - 0.38848339725198033)}{-0.001} = -0.028284068 \quad (2.14)$$

2.2.3 Kiểm tra kết quả tính toán

Trước tiên, ta xây dựng hàm tính đạo hàm hàm sigmoid như sau:

```
In [1]: def grad_sigmoid(x):  
In [2]:     return sigmoid(x)*(1-sigmoid(x))
```

Đạo hàm tại lớp node o:

```
In [1]: dE_dzo1 = -(y[0]-o[0])*grad_sigmoid(z_o[0])  
In [2]: dE_dzo2 = -(y[1]-o[1])*grad_sigmoid(z_o[1])  
Out [1]: dE_dzo1 = -0.03742222949881498  
Out [2]: dE_dzo2 = 0.10653661566082473
```

Đạo hàm tại lớp node k:

```
In [1]: dE_dzk1 = (dE_dzo1*W3[0][0] + dE_dzo2*W3[1][0])*grad_sigmoid(z_k[0])  
In [2]: dE_dzk2 = (dE_dzo1*W3[0][1] + dE_dzo2*W3[1][1])*grad_sigmoid(z_k[1])  
In [3]: dE_dzk3 = (dE_dzo1*W3[0][2] + dE_dzo2*W3[1][2])*grad_sigmoid(z_k[2])  
Out [1]: dE_dzk1 = 0.005044088516268637  
Out [2]: dE_dzk2 = 0.012600061060812528  
Out [3]: dE_dzk3 = 0.006737373454073853
```

Đạo hàm tại lớp node h:

```
In [1]: dE_dzh1 = dE_dzk1*W2[0][0] + dE_dzk2*W2[1][0] + dE_dzk3*W2[2][0]  
In [2]: dE_dzh1 = dE_dzh1*grad_sigmoid(z_h[0])  
In [3]: dE_dzh2 = dE_dzk1*W2[0][1] + dE_dzk2*W2[1][1] + dE_dzk3*W2[2][1]  
In [4]: dE_dzh2 = dE_dzh2*grad_sigmoid(z_h[1])  
In [5]: dE_dzh3 = dE_dzk1*W2[0][2] + dE_dzk2*W2[1][2] + dE_dzk3*W2[2][2]  
In [6]: dE_dzh3 = dE_dzh3*grad_sigmoid(z_h[2])  
Out [2]: dE_dzh1 = 0.001731189170690768  
Out [4]: dE_dzh2 = 0.0012198156767701336  
Out [6]: dE_dzh3 = 0.001215170397017581
```

Theo công thức (2.11), ta tính được đạo hàm như sau;

Ta có thể tính sấp xỉ đạo hàm:

$$\frac{\partial E}{\partial w_7} = \frac{\partial E}{\partial z_{h1}} \frac{\partial z_{h1}}{\partial w_1} = \frac{\partial E}{\partial z_{h1}} x_1 = 0.001731189170690768 * 0.1 = 0.000173119 \quad (2.15)$$

Tương tự cho w_{16} và w_7 .



$$\frac{\partial E}{\partial w_7} = \frac{\partial E}{\partial z_{k1}} \frac{\partial z_{k1}}{\partial w_7} = \frac{\partial E}{\partial z_{k1}} h_1 = 0.005044088516268637 * 0.66150316 = 0.003336680 \quad (2.16)$$

$$\frac{\partial E}{\partial w_{16}} = \frac{\partial E}{\partial z_{o1}} \frac{\partial z_{o1}}{\partial w_{16}} = \frac{\partial E}{\partial z_{o1}} k_1 = -0.03742222949881498 * 0.75619327 = -0.0282984381 \quad (2.17)$$

Như vậy, kết quả thực hiện tính toán đạo hàm khi thay đổi khoảng nhỏ giá trị $w_1 = 0.000173122$ (2.12) giống với kết quả khi ta tính toán đạo hàm thông qua lan truyền ngược $w_1 = 0.000173119$ (2.15) với sai số chỉ 0.00173%.

$$\Delta E_{w_1} = \frac{|0.000173122 - 0.000173119|}{0.000173119} \approx 0.00173\%$$

Tương tự, kết quả đạo hàm của hàm mất mát E theo biến w_7 và w_{16} bằng hai cách khác nhau cũng cho kết quả xấp xỉ nhau với sai số rất bé 0.01487% cho biến w_7 và 0.05078% cho biến w_{16} .

$$\Delta E_{w_7} = \frac{|0.003336138 - 0.003336680|}{0.003336680} \approx 0.01487\%$$

$$\Delta E_{w_{16}} = \frac{| -0.028284068 - (-0.0282984381)|}{|-0.0282984381|} \approx 0.05078\%$$

2.3 Bài 3

- Đối với batch gradient descent khi cập nhật trọng số ta đem trừ vector gradient thì sẽ đi theo đường dốc nhất bởi vì mỗi lần cập nhật trọng số thì vector gradient được lấy trung bình trên toàn bộ tập dữ liệu, cho nên nó mang những đặc điểm chung nhất của dữ liệu, mỗi dữ liệu đều đóng góp sức ảnh hưởng của mình vào việc cập nhật trọng số. Khi ta đem trừ vector gradient trung bình này thì hàm chi phí (cost function) sẽ có xu hướng giảm đều. Còn đối với stochastic gradient descent thì lại không phải đường dốc nhất, vì stochastic gradient descent ta đang xét đạo hàm tại một điểm nên không thể đại diện được cho toàn bộ tập dữ liệu cho nên nó chỉ hướng đến độ giảm khi xét tại 1 điểm dữ liệu. Thêm vào đó sẽ có những điểm nhiễu khiến cho đi theo chiều tăng, sai hướng dẫn đến làm cho hàm chi phí khiến cho nó không đi theo độ dốc nhất mà đi theo hình zigzag.
- Phương pháp Stochastic gradient descent có thể tránh được các điểm cực tiểu địa phương bởi vì do nó đi theo hình zigzag nên nó đi được nhiều hướng hơn và có thể tránh được hướng độ dốc nhất tiến đến cực tiểu địa phương, chi tiết xem thêm tại [1]. Có thể rằng, khi trong tình huống rơi vào cực trị địa phương đại đa số đạo hàm tại các điểm trong tập dữ liệu sẽ nhỏ lại, làm cho giá trị trung bình các đạo hàm sẽ nhỏ dẫn đến việc mắc kẹt tại cực trị địa phương. Nhưng đối với stochastic gradient descent, có thể giá trị đạo hàm tại điểm đang khảo sát không nhỏ nên khi cập nhật trọng số sẽ giúp đi khỏi vùng lân cận cực trị địa phương nên có thể tránh được. Tuy nhiên, đây là một sự ngẫu nhiên và không có gì bảo đảm nó sẽ thoát khỏi cực trị địa phương được, nó phụ thuộc vào điểm dữ liệu và giá trị hệ số học (learning rate) hiện tại. Mặt khác, có thể stochastic gradient descent có thể làm cho thuật toán hội tụ lâu hơn khi quá nhiều nhiễu, khiến cho không thể tiến tới cực tiểu toàn cục của bài toán.
- Phương pháp mini batch gradient descent lại là phương pháp tốt nhất trong ba phương pháp bởi nó là phương pháp kết hợp của batch gradient descent và stochastic gradient descent nên mang các ưu điểm của hai phương pháp trên. Batch gradient descent mang lại thông tin tốt về độ dốc tuy nhiên không tránh được cực trị địa phương còn stochastic gradient descent dựa trên một điểm dữ liệu ngẫu nhiên, có thể là nhiều nên sẽ đi theo hướng không phải độ dốc nhất, nhưng nó có thể giúp thoát khỏi các điểm cực trị địa phương ở các hàm non-convex tuy nhiên lại cần phải lặp lại nhiều lần vì tính nhiễu và không ổn định của nó. Mini-batch gradient descent giống như một sự thỏa hiệp cân bằng giữa ưu điểm và nhược điểm của hai phương pháp batch gradient descent và stochastic gradient descent cho nên nó được xem là một phương án tốt hơn cho bài toán tối ưu so với hai phương pháp trên.

2.4 Bài 4

2.4.1 Gradient Descent cho hàm nhiều biến

Bài toán đặt ra là đi tìm điểm cực tiểu toàn cục của hàm $f(\mathbf{x})$ với \mathbf{x} là một vector. Đạo hàm của hàm số tại một điểm \mathbf{x} được ký hiệu $\nabla_{\mathbf{x}} f(\mathbf{x})$. Thuật toán Gradient Descent sau t vòng lặp là:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_t)$$

Trong đó η là một số dương được gọi là learning rate (tốc độ học). Dấu trừ trong công thức thể hiện việc đi ngược với đạo hàm.

Doan chương trình thực hiện thuật toán Gradient Descent cho hàm 1 biến như sau:

```
def GD(eta, x_0, gradfunction):
    x = x_0
    for it in range(100):
        x_new = x[-1] - eta * gradfunction(x[-1])

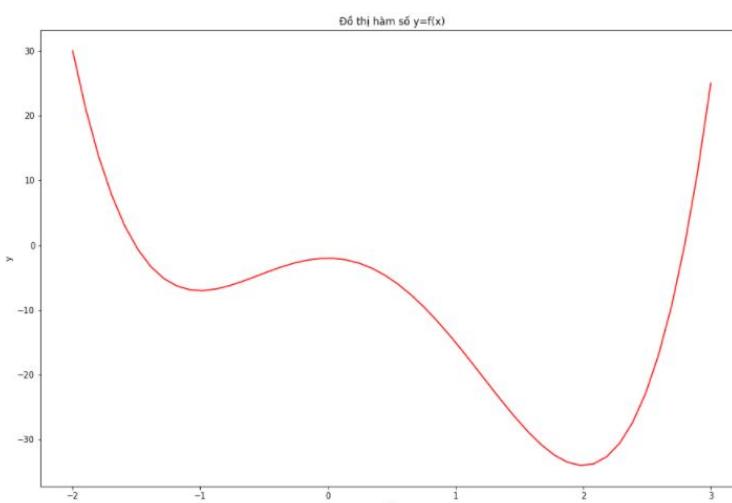
        if abs(gradfunction(x_new)) < 1e-3:
            break

    x = np.vstack((x, x_new))
    return x, it
```

Hàm GD có các tham số đầu vào: η là learning rate (hệ số học), x_0 là giá trị khởi tạo, **gradfunction** là đạo hàm của hàm số ta đang khảo sát. Thuật toán dừng lại khi đạo hàm có độ lớn đủ nhỏ hay giá trị cập nhật mới \mathbf{x}_{t+1} gần như không thay đổi so với giá trị cũ \mathbf{x}_t . Đầu ra là tập hợp các giá trị \mathbf{x} sau mỗi lần cập nhật và **it** số lần cập nhật.

2.4.2 Sử dụng GD tìm cực tiểu của hàm số 1 biến

Xét hàm số $f(x) = 3x^4 - 4x^3 - 12x^2 - 2$

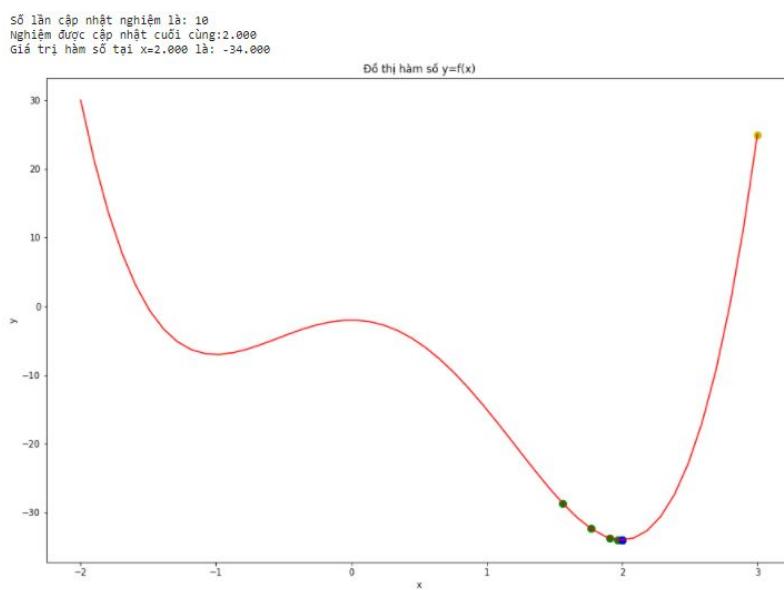


Hình 2.4.1: Đồ thị hàm số $f(x) = 3x^4 - 4x^3 - 12x^2 - 2$

Hàm số $f(x)$ có 3 điểm cực trị gồm 2 cực tiểu và 1 cực đại. Giá trị nhỏ nhất của hàm số là -34 đạt được khi $x = 2$, ngoài ra hàm số còn có 1 cực tiểu địa phương tại $x = -1$. Đạo hàm của hàm số là $f'(x) = 12x^3 - 12x^2 - 24x$.

Ta sẽ tiến hành thử nghiệm thuật toán GD với $\eta = 0.01$ và khảo sát nhiều giá trị khởi tạo khác nhau. Để tiện so sánh, ta quy ước màu vàng là giá khởi tạo đầu tiên, màu xanh dương là kết quả thu được.

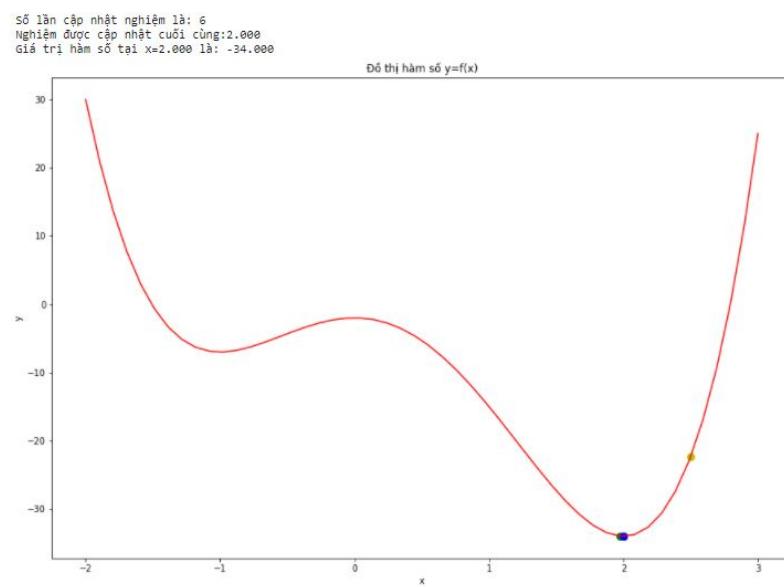
- Giá trị khởi tạo $x_0 = 3$.



Hình 2.4.2: Giá trị khởi tạo $x_0 = 3$

Với điểm khởi tạo $x_0 = 3$, ta thấy thuật toán hội tụ khá nhanh, chỉ sau 10 lần cập nhật, kết quả chính xác tại vị trí cực tiểu toàn cục (global minimum) $x = 2$ của hàm số.

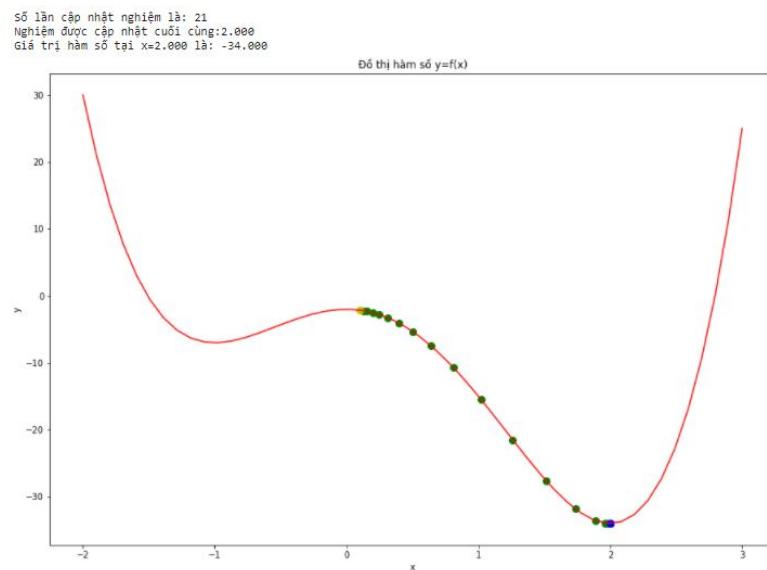
- Giá trị khởi tạo $x_0 = 2.5$



Hình 2.4.3: Giá trị khởi tạo $x_0 = 2.5$

Với điểm khởi tạo $x_0 = 2.5$, thuật toán cũng hội tụ nhanh, chỉ sau 6 lần cập nhật cho kết quả chính xác là vị trí cực tiểu toàn cục $x = 2$ của hàm số.

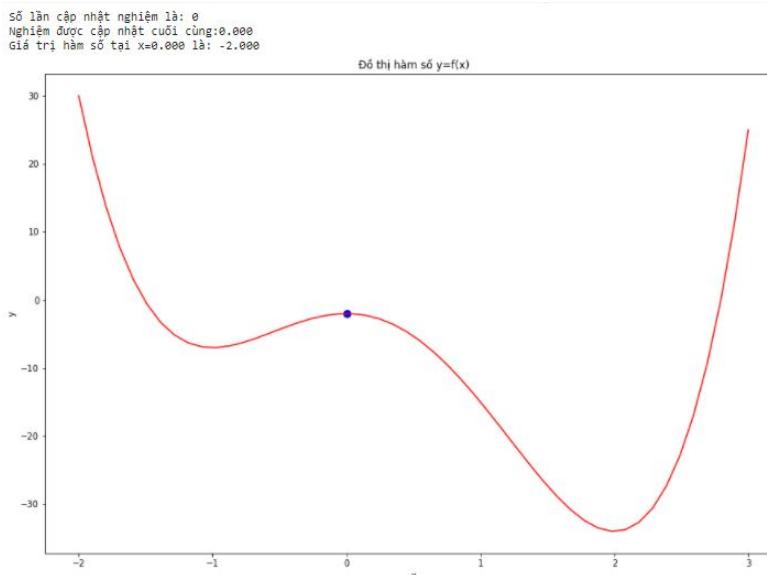
- Giá trị khởi tạo $x_0 = 0.1$



Hình 2.4.4: Giá trị khởi tạo $x_0 = 0.1$

Với điểm khởi tạo $x_0 = 0.1$, ta thấy thuật toán hội tụ sau 21 lần cập nhật giá trị. Kết quả thu được vẫn chính xác tại vị trí cực tiểu toàn cục $x = 2$ của hàm số.

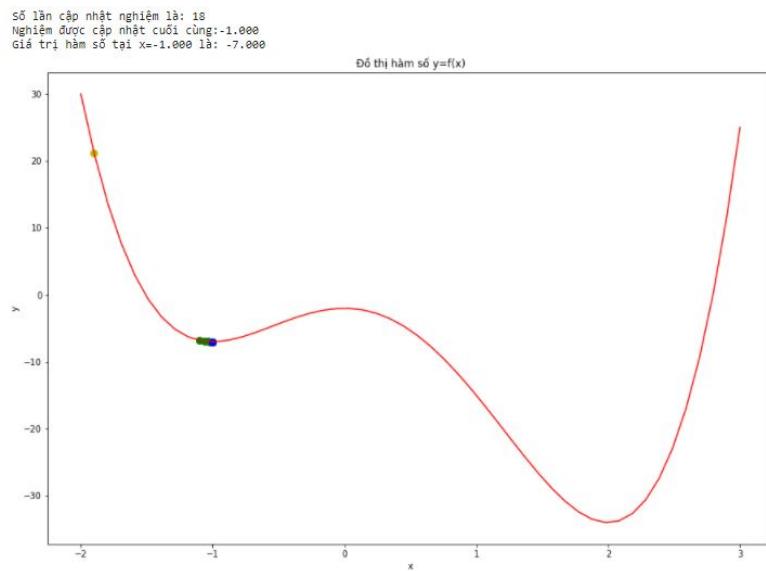
- Giá trị khởi tạo $x_0 = 0$



Hình 2.4.5: Giá trị khởi tạo $x_0 = 0$

Với điểm khởi tạo $x_0 = 0$, thuật toán không cập nhật giá trị và cho kết quả là $x_0 = 0$. Tuy nhiên đây là kết quả sai vì vị trí này không phải là global minimum. Sẽ dễ xảy ra điều này vì giá trị khởi tạo ban đầu bằng với giá trị cực đại của hàm số. Nói cách khác đạo hàm tại điểm khởi tạo bằng 0 nên thuật toán Gradient Descent dừng ngay lập tức.

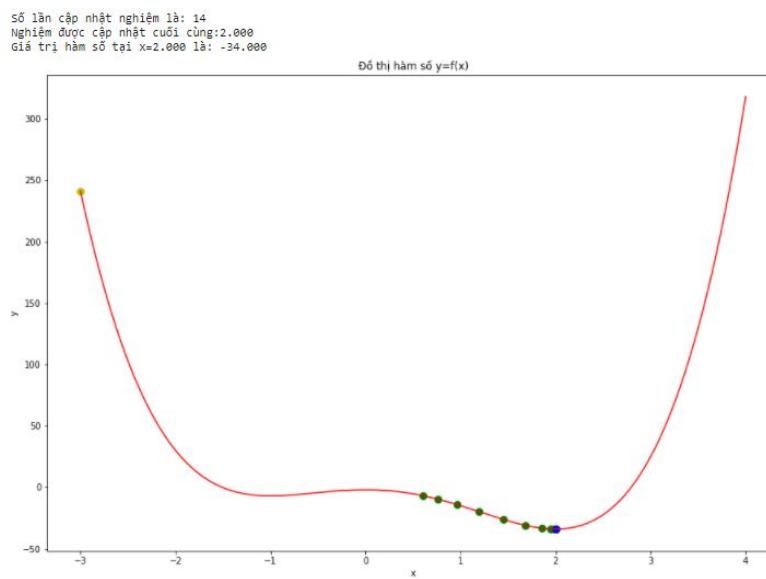
- Giá trị khởi tạo $x_0 = -1.9$



Hình 2.4.6: Giá trị khởi tạo $x_0 = -1.9$

Với điểm khởi tạo $x_0 = -1.9$, thuật toán dừng lại sau 18 lần cập nhật. Tuy nhiên điểm cuối cùng của thuật toán lại là điểm cực tiểu địa phương (local minimum) $x = -1$.

- Giá trị khởi tạo $x_0 = -3$

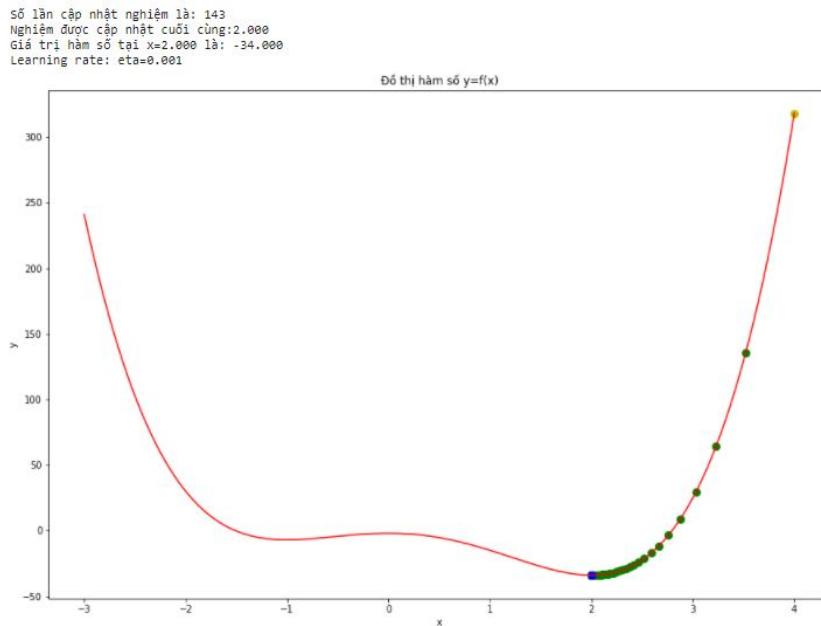


Hình 2.4.7: Giá trị khởi tạo $x_0 = -3$

Với điểm khởi tạo $x_0 = -3$, thuật toán dừng lại khá nhanh sau 14 lần cập nhật, kết quả chính xác là vị trí cực tiểu toàn cục (global minimum) của hàm số.

Khảo sát thuật toán với nhiều learning rate η khác nhau. Để cho việc khảo sát được công bằng, chúng ta sẽ chọn giá trị khởi tạo là $x_0 = -4$ với mọi learning rate khác nhau. Thuật toán sẽ được khảo sát với các giá trị learning rate là 0.001, 0.003, 0.007, 0.01, 0.015.

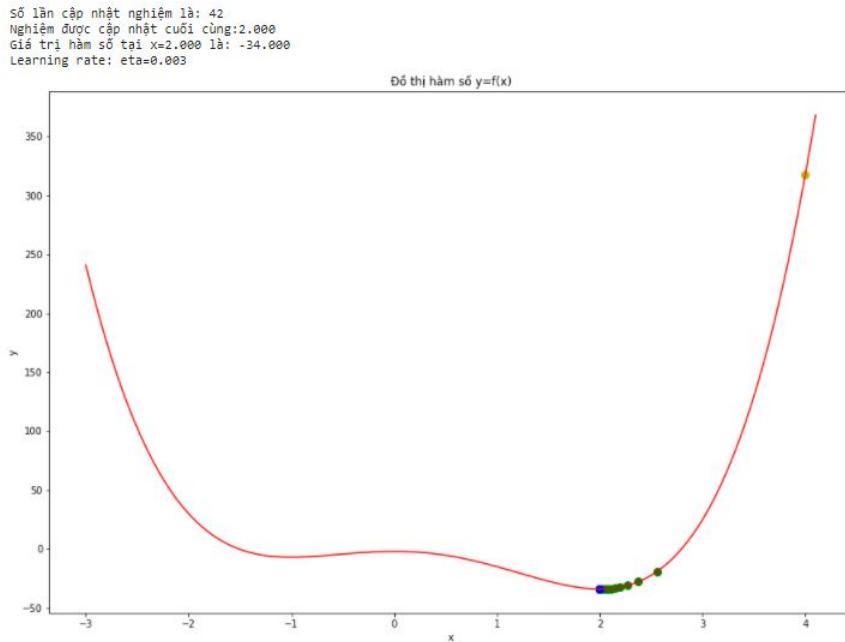
- Với $\eta = 0.001$



Hình 2.4.8: Learning rate $\eta = 0.001$

Với $\eta = 0.001$, thời gian thực hiện thuật toán tuy khá lâu (sau 142 lần cập nhật) nhưng kết quả vẫn chính xác là vị trí global minimum của hàm số.

- Với $\eta = 0.003$

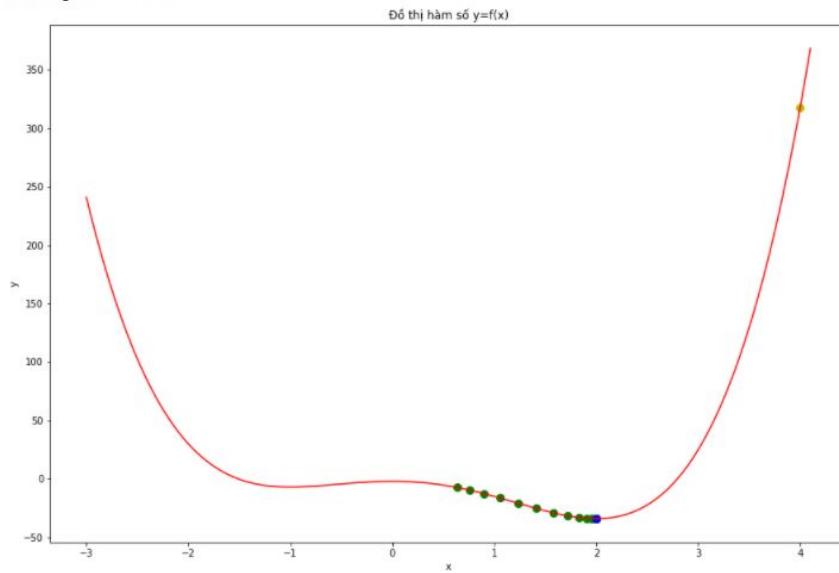


Hình 2.4.9: Learning rate $\eta = 0.003$

Với $\eta = 0.003$, thời gian thực hiện thuật toán đã nhanh hơn đôi chút (sau 42 lần cập nhật). Kết quả vẫn chính xác là vị trí global minimum của hàm số.

- Với $\eta = 0.007$

Số lần cập nhật nghiệm là: 22
Nghiệm được cập nhật cuối cùng: 2.000
Giá trị hàm số tại $x=2.000$ là: -34.000
Learning rate: eta=0.007

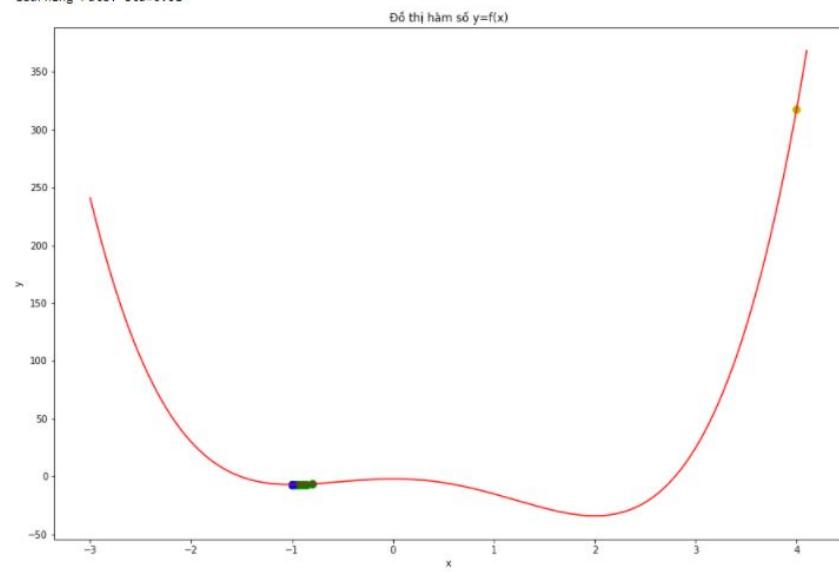


Hình 2.4.10: Learning rate $\eta = 0.007$

Với $\eta = 0.007$, thời gian thực hiện thuật toán khá nhanh (sau 22 lần cập nhật). Kết quả điểm thu được vẫn chính xác là vị trí global minimum của hàm số.

- Với $\eta = 0.01$

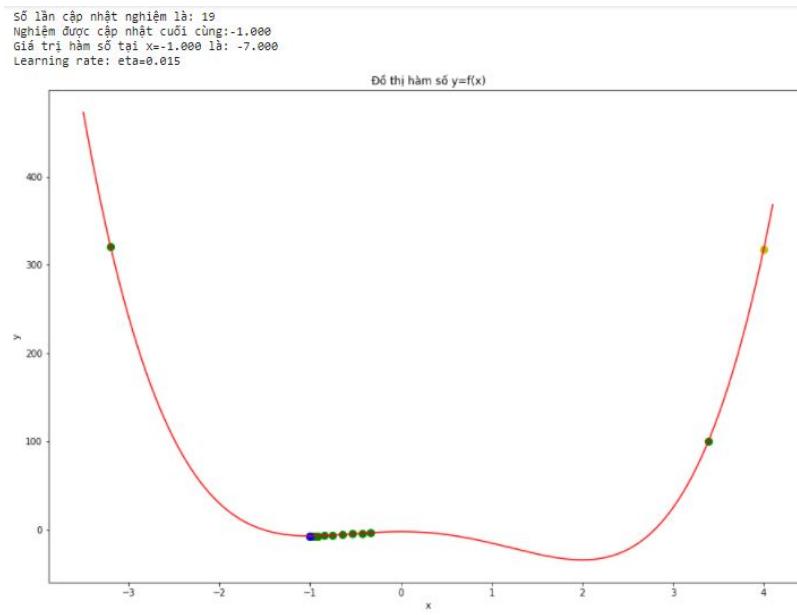
Số lần cập nhật nghiệm là: 21
Nghiệm được cập nhật cuối cùng: -1.000
Giá trị hàm số tại $x=-1.000$ là: -7.000
Learning rate: eta=0.01



Hình 2.4.11: Learning rate $\eta = 0.01$

Với $\eta = 0.01$, thời gian thực hiện thuật toán khá nhanh (sau 21 lần cập nhật). Tuy nhiên kết quả không chính xác nữa. vị trí điểm kết quả bấy giờ là local minimum của hàm số.

- Với $\eta = 0.015$

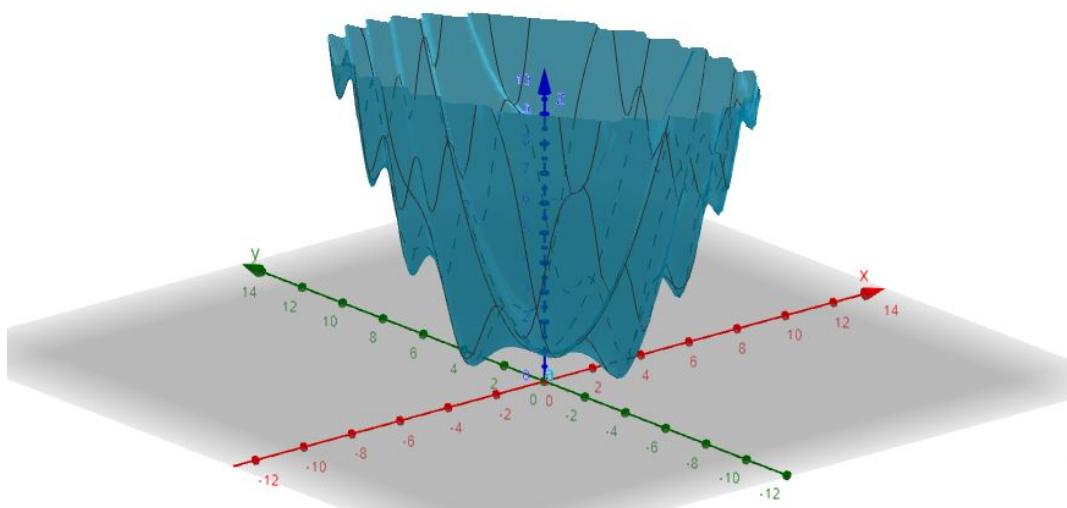


Hình 2.4.12: Learning rate $\eta = 0.015$

Với $\eta = 0.015$, thời gian thực hiện thuật toán khá nhanh (sau 19 lần cập nhật). Tuy nhiên kết quả tìm được không chính xác nữa. vị trí điểm kết quả là local minimum của hàm số.

2.4.3 Sử dụng GD tìm cực tiểu của hàm số $f_1(x, y)$

Xét hàm số $f_1(x, y) = \cos(xy) + \frac{1}{2} x^2 + \frac{1}{4} y^2 + \frac{1}{4} xy$. Đồ thị hàm số có dạng như sau:



Hình 2.4.13: Đồ thị hàm số $f(x, y) = \cos(xy) + \frac{1}{2} x^2 + \frac{1}{4} y^2 + \frac{1}{4} xy$

Hàm số $f_1(x, y)$ là một hàm không lồi và có rất nhiều điểm cực trị. Hàm số có giá trị nhỏ nhất vào khoảng 0.33005 tại hai điểm cực trị có giá trị tọa độ xấp xỉ $(1.36, -1.94)$ và $(-1.36, 1.94)$. Đạo hàm riêng phần theo biến x và y lần lượt là:

$$f'_x(x, y) = -y \sin(xy) + \frac{1}{4} y + x.$$

$$f'_y(x, y) = -x \sin(xy) + \frac{1}{4} x + \frac{1}{2} y.$$

Trong phần này ta cũng áp dụng thuật toán GD nhưng có sửa đổi một phần nhỏ. Thuật toán GD của chúng ta sẽ dừng lại khi giá trị hàm số $f(x, y)$ ở hai giá trị (x, y) liền kề chêch lệch ít. Vì khi đạo hàm của hàm số càng nhỏ thì giá trị hàm số biến thiên càng ít.

Ta sẽ tiến hành thử nghiệm thuật toán GD với $\eta = 0.1$ và khảo sát nhiều giá trị khởi tạo khác nhau. Ta quy ước dòng đầu của kết quả là số lần cập nhật giá trị, dòng tiếp theo là giá trị khởi tạo ban đầu. Sau đó là 4 giá trị kết quả được cập nhật cuối cùng (việc này tiện lợi cho việc so sánh sự thay đổi của kết quả sau những lần thực hiện Gradient Descent). Và kết thúc là giá trị của hàm số tại giá trị kết quả tối ưu mà ta tìm được.

- Giá trị khởi tạo $(x, y) = (-5, 5)$.

```
# X=np.random.randn(1,2)*10
X= np.array([[-5,5]])
x,it,eta=GD2(0.1,X,gradf3,f3)
print(it,"\\n",x[0],"\\n ",x[-4:,:],"\\n",f3(x[-1]))
```

20
[-5. 5.]
[[-2.44863855 3.63974345]
 [-2.47320203 3.6390136]
 [-2.4668224 3.62081463]
 [-2.48198205 3.61816399]]
3.2050405644174247

Hình 2.4.14: Giá trị khởi tạo $(x, y) = (-5, 5)$

Với điểm khởi tạo $(x, y) = (-5, 5)$, ta thấy thuật toán hoàn thành khá nhanh, chỉ sau 20 lần cập nhật. Hàm số $f(x, y)$ đạt giá trị 3.205 tại kết quả cuối là $(x, y) = (-2.4819, 3.618)$. Tuy nhiên điểm kết quả cuối cùng này lại là điểm cực tiểu địa phương (local minimum).

- Giá trị khởi tạo $(x, y) = (-12, 8)$ Với điểm khởi tạo $(x, y) = (-12, 8)$, ta thấy thuật toán

```
# X=np.random.randn(1,2)*10
X= np.array([[-12,8]])
x,it,eta=GD2(0.1,X,gradf3,f3)
print(it,"\n",x[0],"\n \n",x[-4:,:]," \n",f3(x[-1]))
```

56
[-12. 8.]

```
[[ -2.46532817  3.6505658 ]  
[ -2.46055838  3.63130678]  
[ -2.4760992   3.62699825]  
[ -2.47495412  3.61390631]]  
3.2049635088366166
```

Hình 2.4.15: Giá trị khởi tạo $(x, y) = (-12, 8)$

hoàn thành sau 56 lần cập nhật. Hàm số $f(x, y)$ đạt giá trị 3.205 tại kết quả cuối là $(x, y) = (-2.4819, 3.618)$. Giống ở với kết quả ở điểm khởi tạo ở trường hợp phía trên, điểm kết quả cuối cùng này lại là điểm cực tiểu địa phương (local minimum).

- Giá trị khởi tạo $(x, y) = (-1, 1)$

```
# X=np.random.randn(1,2)*10
X= np.array([[-1,1]])
x,it,eta=GD2(0.1,X,gradf3,f3)
print(it,"\n",x[0],"\n \n",x[-4:,:]," \n",f3(x[-1]))
```

18
[-1. 1.]

```
[[ -1.42305617  1.8435312 ]  
[ -1.41814318  1.85741058]  
[ -1.41303615  1.86891681]  
[ -1.40817837  1.87863388]]  
0.3330104776559163
```

Hình 2.4.16: Giá trị khởi tạo $(x, y) = (-1, 1)$

Với điểm khởi tạo $(x, y) = (-1, 1)$, ta thấy thuật toán hội tụ sau 18 lần cập nhật giá trị. Kết quả cuối cùng là $(x, y) = (-1.408, 1.87)$ và giá trị hàm số tại kết quả này bằng 0.333. Kết quả này xấp xỉ giá trị cực trị toàn cục của hàm số $f(x, y)$.

- Giá trị khởi tạo $(x, y) = (-1200, -2000)$

```
# X=np.random.randn(1,2)*10
X= np.array([[-1200,-2000]])
x,it,eta=GD2(0.1,X,gradf3,f3)
print(it,"\\n",x[0],"\\n \\n",x[-4:,:],"\\n",f3(x[-1]))
```

```
168
[-1200. -2000.]

[[ 2.50932063 -3.56740944]
 [ 2.51009156 -3.56608719]
 [ 2.51087262 -3.56501252]
 [ 2.51147211 -3.56402729]]
3.2017494523161774
```

Hình 2.4.17: Giá trị khởi tạo $(x, y) = (-1200, -2000)$

Với điểm khởi tạo $(x, y) = (-1200, -2000)$, ta thấy thuật toán hoàn thành sau 56 lần cập nhật. Hàm số $f(x, y)$ đạt giá trị 3.201 tại kết quả cuối là $(x, y) = (2.511, -3.564)$. Tuy nhiên đây không phải là điểm cực trị toàn cục, đây là cực trị địa phương của hàm số $f(x, y)$.

- Giá trị khởi tạo $(x, y) = (-3.424, -0.776)$

```
X=np.random.randn(1,2)*10
# X= np.array([[-1200,-2000]])
x,it,eta=GD2(0.1,X,gradf3,f3)
print(it,"\\n",x[0],"\\n \\n",x[-4:,:],"\\n",f3(x[-1]))
```

```
44
[-3.42426704 -0.77679003]

[[-1.1461311 -1.60881464]
 [-1.14621602 -1.61008561]
 [-1.14631367 -1.61124968]
 [-1.14641912 -1.612316   ]]
1.495084461468478
```

Hình 2.4.18: Giá trị khởi tạo $(x, y) = (-3.424, -0.776)$

Với điểm khởi tạo $(x, y) = (-3.424, -0.776)$, ta thấy thuật toán hoàn thành sau 44 lần cập nhật. Hàm số $f(x, y)$ đạt giá trị 1.491 tại kết quả cuối là $(x, y) = (-1.146, -1.612)$, đây là cực trị địa phương của hàm số $f(x, y)$, không phải cực trị toàn cục ta mong muốn.

Thực hiện khảo sát thuật toán với nhiều learning rate η khác nhau. Để cho việc khảo sát được công bằng, chúng ta sẽ chọn giá trị khởi tạo là $(x, y) = (10, 20)$ với mọi learning rate khác nhau. Thuật toán sẽ được khảo sát với các giá trị learning rate là 0.01, 0.03, 0.07, 0.1, 0.3, 0.7.

- Với $\eta = 0.01$ Với $\eta = 0.01$, thời gian thực hiện thuật toán rất lâu, sau 393 lần cập nhật.

```
# X=np.random.randn(1,2)*10
X= np.array([[10,20]])
x,it,eta=GD2(0.01,X,gradf3,f3)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f3(x[-1])))
```

```
learning rate= 0.01
Số lần cập nhật nghiệm: 393
Nghiệm khởi tạo: [10. 20.]
4 nghiệm cập nhật cuối cùng:
 [[11.39467464 16.1548428 ]
 [11.39486328 16.15457296]
 [11.39504941 16.15430671]
 [11.39523307 16.15404401]]
Giá trị của hàm số tại nghiệm cuối [11.39523307 16.15404401] là : 175.89210096440397
```

Hình 2.4.19: Khảo sát với $\eta = 0.01$

Tuy nhiên kết quả đạt được là điểm cực trị địa phương.

- Với $\eta = 0.03$

```
# X=np.random.randn(1,2)*10
X= np.array([[10,20]])
x,it,eta=GD2(0.03,X,gradf3,f3)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f3(x[-1])))
```

```
learning rate= 0.03
Số lần cập nhật nghiệm: 244
Nghiệm khởi tạo: [10. 20.]
4 nghiệm cập nhật cuối cùng:
 [[-1.36569012  1.95609913]
 [-1.36597564  1.95556151]
 [-1.36625044  1.95504439]
 [-1.36651493  1.95454701]]
Giá trị của hàm số tại nghiệm cuối [-1.36651493  1.95454701] là : 0.3297532182789408
```

Hình 2.4.20: Khảo sát với $\eta = 0.03$

Với $\eta = 0.03$, thời gian thực hiện thuật toán khá lâu, sau 244 lần cập nhật. Kết quả thu được xấp xỉ với cực trị toàn cục của hàm số $f(x, y)$ với giá trị nhỏ nhất gần bằng 0.33.

- VỚI $\eta = 0.07$

```
# X=np.random.randn(1,2)*10
X= np.array([[10,20]])
x,it,eta=GD2(0.07,X,gradf3,f3)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f3(x[-1])))
```

```
learning rate= 0.07
Số lần cập nhật nghiệm: 93
Nghiệm khởi tạo: [10. 20.]
4 nghiệm cập nhật cuối cùng:
 [[-2.50895896  3.5678688 ]
 [-2.50957723  3.56693081]
 [-2.51013879  3.56607935]
 [-2.5106488   3.56530643]]
Giá trị của hàm số tại nghiệm cuối [-2.5106488   3.56530643] là : 3.20176619510711
```

Hình 2.4.21: Khảo sát với $\eta = 0.07$

VỚI $\eta = 0.07$, thời gian thực hiện thuật toán khá nhanh, sau 93 lần cập nhật. Khá bất ngờ là kết quả cuối cùng không chính xác vì khi tăng learning rate ở các trường hợp tiếp theo cho kết quả gần với cực trị toàn cục.

- VỚI $\eta = 0.1$

```
# X=np.random.randn(1,2)*10
X= np.array([[10,20]])
x,it,eta=GD2(0.1,X,gradf3,f3)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f3(x[-1])))
```

```
learning rate= 0.1
Số lần cập nhật nghiệm: 74
Nghiệm khởi tạo: [10. 20.]
4 nghiệm cập nhật cuối cùng:
 [[-1.36781351  1.95210716]
 [-1.36849956  1.95082324]
 [-1.36909919  1.94970276]
 [-1.36962317  1.94872497]]
Giá trị của hàm số tại nghiệm cuối [-1.36962317  1.94872497] là : 0.3296616427712912
```

Hình 2.4.22: Khảo sát với $\eta = 0.1$

VỚI $\eta = 0.1$, thời gian thực hiện thuật toán khá nhanh, sau 74 lần cập nhật. Kết quả cuối cùng chính xác, rất gần với vị trí global minimum của hàm số.

- Với $\eta = 0.3$

```
# X=np.random.randn(1,2)*10
X= np.array([[10,20]])
x,it,eta=GD2(0.3,X,gradf3,f3)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f3(x[-1])))
```

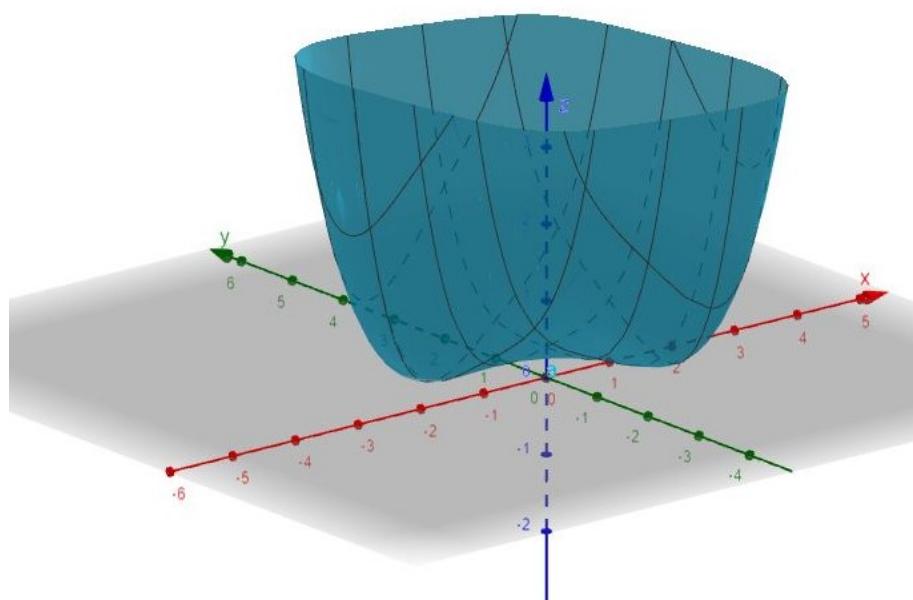
```
learning rate= 0.3
Số lần cập nhật nghiệm: 22
Nghiệm khởi tạo: [10. 20.]
4 nghiệm cập nhật cuối cùng:
 [[-1.36382529  1.94971329]
 [-1.37231611  1.94939455]
 [-1.36980703  1.94504003]
 [-1.37277909  1.94478576]]
Giá trị của hàm số tại nghiệm cuối [-1.37277909  1.94478576] là : 0.32963145665182303
```

Hình 2.4.23: Khảo sát với $\eta = 0.3$

Với $\eta = 0.3$, thời gian thực hiện thuật toán rất nhanh sau 22 lần cập nhật. Kết quả cuối thu được chính xác, rất gần với vị trí global minimum của hàm số.

2.4.4 Sử dụng GD tìm cực tiểu của hàm số $f_2(x, y)$

Xét hàm số $f_2(x, y) = \frac{1}{10}y^4 + \frac{1}{5}x^4 + \frac{3}{5}xy + \frac{1}{5}x + \frac{3}{5}$. Đồ thị hàm số có dạng như sau:



Hình 2.4.24: Đồ thị hàm số $f(x, y) = \frac{1}{10}y^4 + \frac{1}{5}x^4 + \frac{3}{5}xy + \frac{1}{5}x + \frac{3}{5}$

Hàm số $f_2(x, y)$ là một hàm không lồi và có rất nhiều điểm cực trị. Hàm số có giá trị nhỏ nhất vào khoảng 0.083 tại điểm $(-1.03, 1.15)$. Đạo hàm riêng phần theo biến x và y lần lượt là:

$$f'_x(x, y) = \frac{4}{5}x^3 + \frac{3}{5}y + \frac{1}{5}.$$

$$f'_y(x, y) = \frac{2}{5}y^3 + \frac{3}{5}x.$$

- Giá trị khởi tạo $(x, y) = (-0.385, -1.432)$.

```
X=np.random.randn(1,2)
# X= np.array([[0.5,0.5]])
x,it,eta=GD2(0.03,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))

learning rate= 0.03
Số lần cập nhật nghiệm: 78
Nghiệm khởi tạo: [ 0.3857579 -1.43220067]
4 nghiệm cập nhật cuối cùng:
 [[ 0.80515688 -1.06186439]
 [ 0.80574327 -1.06198947]
 [ 0.80630453 -1.06212003]
 [ 0.80684188 -1.0622554 ]]
Giá trị của hàm số tại nghiệm cuối [ 0.80684188 -1.0622554 ] là : 0.45920929737846733
```

Hình 2.4.25: Giá trị khởi tạo $(x, y) = (-0.385, -1.432)$)

Với điểm khởi tạo $(x, y) = (-0.385, -1.432)$, ta thấy thuật toán hoàn thành sau 78 lần cập nhật. Hàm số $f(x, y)$ đạt giá trị 0.459 tại kết quả cuối là $(x, y) = (-0.8065, -1.062)$. Kết quả thu được là điểm cực trị địa phương.

- Giá trị khởi tạo $(x, y) = (-1.568, -0.193)$

```
X=np.random.randn(1,2)
# X= np.array([[0.5,0.5]])
x,it,eta=GD2(0.03,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))

learning rate= 0.03
Số lần cập nhật nghiệm: 155
Nghiệm khởi tạo: [-1.56881967 -0.19379842]
4 nghiệm cập nhật cuối cùng:
 [[-1.031547  1.14483286]
 [-1.0318102 1.14539511]
 [-1.03206335 1.14593555]
 [-1.03230682 1.14645502]]
Giá trị của hàm số tại nghiệm cuối [-1.03230682  1.14645502] là : 0.08332174854380003
```

Hình 2.4.26: Giá trị khởi tạo $(x, y) = (-1.568, -0.193)$

Với điểm khởi tạo $(x, y) = (-1.568, -0.193)$, ta thấy thuật toán hoàn thành sau 155 lần cập nhật. Hàm số $f(x, y)$ đạt giá trị 0.083 tại kết quả cuối là $(x, y) = (-1.032, 1.146)$. Kết quả thu được xấp xỉ với điểm cực trị toàn cục của hàm số $f(x, y)$.

- Giá trị khởi tạo $(x, y) = (0.816, 0.459)$

```
X=np.random.randn(1,2)
# X= np.array([[0.5,0.5]])
x,it,eta=GD2(0.03,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))

learning rate= 0.03
Số lần cập nhật nghiệm: 288
Nghiệm khởi tạo: [ 0.81696684  0.45947053]
4 nghiệm cập nhật cuối cùng:
 [[-1.03161304  1.1450318 ]
 [-1.03187476  1.14558585]
 [-1.03212639  1.14611845]
 [-1.03236831  1.1466304 ]]
Giá trị của hàm số tại nghiệm cuối [-1.03236831  1.1466304 ] là : 0.08331837231749599
```

Hình 2.4.27: Giá trị khởi tạo $(x, y) = (0.816, 0.459)$

Với điểm khởi tạo $(x, y) = (0.816, 0.459)$, ta thấy thuật toán hội tụ sau 288 lần cập nhật giá trị. Hàm số $f(x, y)$ đạt giá trị 0.083 tại kết quả cuối là $(x, y) = (-1.032, 1.146)$ xấp xỉ với điểm cực trị toàn cục của hàm số $f(x, y)$.

- Giá trị khởi tạo $(x, y) = (0.526, -0.778)$

```
X=np.random.randn(1,2)
# X= np.array([[0.5,0.5]])
x,it,eta=GD2(0.03,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))

learning rate= 0.03
Số lần cập nhật nghiệm: 135
Nghiệm khởi tạo: [ 0.52671659 -0.77887889]
4 nghiệm cập nhật cuối cùng:
 [[ 0.80678897 -1.05456385]
 [ 0.80716762 -1.05501263]
 [ 0.8075366 -1.05545024]
 [ 0.80789614 -1.05587696]]
Giá trị của hàm số tại nghiệm cuối [ 0.80789614 -1.05587696] là : 0.45925321881302955
```

Hình 2.4.28: Giá trị khởi tạo $(x, y) = (0.526, -0.778)$

Với điểm khởi tạo $(x, y) = (0.526, -0.778)$, ta thấy thuật toán hoàn thành sau 135 lần cập nhật. Hàm số $f(x, y)$ đạt giá trị 0.459 tại kết quả cuối là $(x, y) = (-0.807, -1.055)$, điểm này là cực trị địa phương của hàm số.

- Giá trị khởi tạo $(x, y) = (0.031, 1.086)$

```
X=np.random.randn(1,2)
# X= np.array([[0.5,0.5]])
x,it,eta=GD2(0.03,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))
```

```
learning rate= 0.03
Số lần cập nhật nghiệm: 124
Nghiệm khởi tạo: [0.03130122 1.08654545]
4 nghiệm cập nhật cuối cùng:
 [[-1.03146397  1.14518945]
 [-1.03173994  1.14573338]
 [-1.03200457  1.14625658]
 [-1.03225832  1.1467598 ]]
Giá trị của hàm số tại nghiệm cuối [-1.03225832  1.1467598 ] là : 0.08331713974767183
```

Hình 2.4.29: Giá trị khởi tạo $(x, y) = (0.031, 1.086)$

Với điểm khởi tạo $(x, y) = (0.031, 1.086)$), ta thấy thuật toán hoàn thành sau 124 lần cập nhật. Hàm số $f(x, y)$ đạt giá trị 0.083 tại kết quả cuối là $(x, y) = (-1.032, 1.146)$, kết quả này xấp xỉ cực trị toàn cục của hàm số $f(x, y)$.

Thực hiện khảo sát thuật toán với nhiều learning rate η khác nhau. Để cho việc khảo sát được công bằng, chúng ta sẽ chọn giá trị khởi tạo là $(x, y) = (-4, -4)$ với các giá trị learning rate khác nhau. Thuật toán sẽ được khảo sát với các giá trị learning rate là 0.001, 0.003, 0.01, 0.03, 0.1.

- Với $\eta = 0.001$

```
# X=np.random.randn(1,2)
X= np.array([[-4,-4]])
x,it,eta=GD2(0.001,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))
```

```
learning rate= 0.001
Số lần cập nhật nghiệm: 8646
Nghiệm khởi tạo: [-4. -4.]
4 nghiệm cập nhật cuối cùng:
 [[-1.00364896  1.08520836]
 [-1.0036913   1.08529934]
 [-1.00373359  1.08539021]
 [-1.00377583  1.08548099]]
Giá trị của hàm số tại nghiệm cuối [-1.00377583  1.08548099] là : 0.08736667980304116
```

Hình 2.4.30: Khảo sát với $\eta = 0.001$

Với $\eta = 0.001$, thời gian thực hiện thuật toán rất lâu, sau 8646 lần cập nhật. Tuy nhiên kết quả cuối cùng rất gần với vị trí global minimum của hàm số.

- VỚI $\eta = 0.003$

```
# X=np.random.randn(1,2)
X= np.array([[-4,-4]])
x,it,eta=GD2(0.003,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))

learning rate= 0.003
Số lần cập nhật nghiệm: 3041
Nghiệm khởi tạo: [-4. -4.]
4 nghiệm cập nhật cuối cùng:
 [[-1.01879508  1.11769993]
 [-1.01886906  1.11785821]
 [-1.01894277  1.11801592]
 [-1.01901621  1.11817305]]
Giá trị của hàm số tại nghiệm cuối [-1.01901621  1.11817305] là : 0.08451506718250519
```

Hình 2.4.31: Khảo sát với $\eta = 0.003$

Với $\eta = 0.003$, thời gian thực hiện thuật toán vẫn khá lâu nhưng đã cải thiện hơn so với giá trị $\eta = 0.001$, sau 3041 lần cập nhật. Kết quả kết quả đã chính xác hơn, càng tiến gần hơn với vị trí global minimum của hàm số.

- VỚI $\eta = 0.01$.

```
# X=np.random.randn(1,2)
X= np.array([[-4,-4]])
x,it,eta=GD2(0.01,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))

learning rate= 0.01
Số lần cập nhật nghiệm: 964
Nghiệm khởi tạo: [-4. -4.]
4 nghiệm cập nhật cuối cùng:
 [[-1.02744169  1.13615719]
 [-1.02758179  1.13645539]
 [-1.02772014  1.13674982]
 [-1.02785674  1.1370405 ]]
Giá trị của hàm số tại nghiệm cuối [-1.02785674  1.1370405 ] là : 0.08358279241547902
```

Hình 2.4.32: Khảo sát với $\eta = 0.01$

Với $\eta = 0.01$, thời gian thực hiện thuật toán vẫn khá lâu nhưng đã cải thiện hơn so với 2

lần trước, sau 961 lần cập nhật. Kết quả kết quả tiến gần hơn với vị trí global minimum của hàm số so với hai trường hợp trên.

- Với $\eta = 0.03$

```
# X=np.random.randn(1,2)
X= np.array([[-4,-4]])
x,it,eta=GD2(0.03,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))
```

```
learning rate= 0.03
Số lần cập nhật nghiệm: 346
Nghiệm khởi tạo: [-4. -4.]
4 nghiệm cập nhật cuối cùng:
 [[-1.03168182  1.14516756]
 [-1.03194072  1.14571644]
 [-1.03218964  1.14624405]
 [-1.03242898  1.1467512 ]]
Giá trị của hàm số tại nghiệm cuối [-1.03242898  1.1467512 ] là : 0.08331593236602294
```

Hình 2.4.33: Learning rate $\eta = 0.03$

Với $\eta = 0.03$, thời gian thực hiện thuật toán khá nhanh, sau 346 lần cập nhật. Kết quả kết quả đã chính xác hơn, càng tiến gần hơn với vị trí global minimum của hàm số.

- Với $\eta = 0.1$

```
# X=np.random.randn(1,2)
X= np.array([[-4,-4]])
x,it,eta=GD2(0.1,X,gradf1,f1)
print("learning rate= {}".format(eta))
print("Số lần cập nhật nghiệm: {}".format(it))
print("Nghiệm khởi tạo: {}".format(x[0]))
print("4 nghiệm cập nhật cuối cùng: \n {}".format(x[-4:,:]))
print("Giá trị của hàm số tại nghiệm cuối {} là : {}".format(x[-1],f1(x[-1])))
```



```
learning rate= 0.1
Số lần cập nhật nghiệm: 35
Nghiệm khởi tạo: [-4. -4.]
4 nghiệm cập nhật cuối cùng:
 [[ 0.83009488 -1.08270499]
 [ 0.82929853 -1.08174264]
 [ 0.82857601 -1.08086777]
 [ 0.82792015 -1.08007229]]
Giá trị của hàm số tại nghiệm cuối [ 0.82792015 -1.08007229] là : 0.45911001519925626
```

Hình 2.4.34: Khảo sát với $\eta = 0.1$

Với $\eta = 0.1$, thời gian thực hiện thuật toán rất nhanh (sau 35 lần cập nhật). Tuy nhiên kết quả thu được lại không chính xác, đã rơi vào local minimum.

2.4.5 Nhận xét

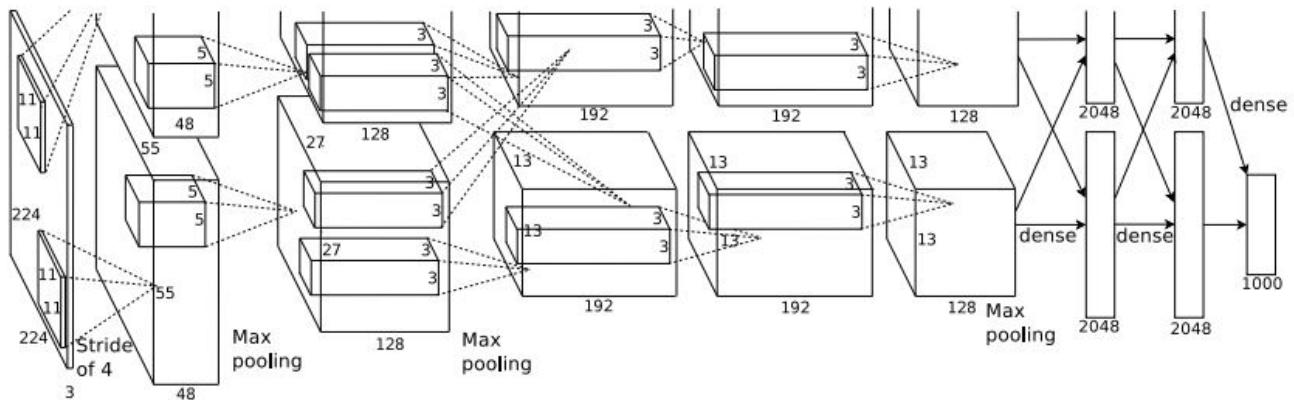
- Việc chọn giá trị khởi tạo đầu tiên cũng có thể ảnh hưởng đến kết quả khi giải bài toán tối ưu bằng cách sử dụng gradient descent.
- Tốc độ học (learning rate) cũng là một yếu tố ảnh hưởng đến quá trình giải bài toán tối ưu khi sử dụng thuật toán gradient descent. Nếu learning rate quá nhỏ thì quá trình thực hiện sẽ diễn ra lâu. Nếu quá lớn thì kết quả cuối cùng của bài toán có thể sẽ không chính xác. Do đó cần lựa chọn thông số phù hợp.

2.5 Bài 5

Sử dụng **kết nối địa phương (local connectivity)** trong mạng CNN giúp ta trích xuất được những đặc trưng tại một vùng ảnh của tấm ảnh đầu vào nhờ vào việc sử dụng bộ lọc (kernel). Bởi các đặc trưng của một đối tượng trong bức ảnh không phải được thể hiện ở một điểm ảnh (pixel) mà là một vùng ảnh (gồm nhiều pixel lân cận nhau). Nếu ta chỉ xét rời rạc từng điểm ảnh mà không xét mối liên kết giữa các điểm ảnh trong một vùng ảnh thì sẽ khó tìm ra được đặc trưng của đối tượng.

Sử dụng **trọng số dùng chung (shared weights)** trong mạng CNN để kernel không chỉ có khả năng kết nối địa phương tại một vị trí cố định mà tại tất cả các vị trí của một tấm ảnh vì thực tế ta không biết vị trí chính xác đặc trưng của đối tượng nằm ở vị trí nào trên tấm ảnh.

2.6 Bài 6



Hình 2.6.1: Kiến trúc mạng Alexnet.

Số lượng thông số của một tầng tích chập được tính như sau:

$$\text{parameters} = (\text{kernel}_{\text{size}} \times \text{kernel}_{\text{size}} \times \text{depth} + 1) \times \text{kernel}_{\text{numbers}}$$

Số lượng thông số của một tầng trong fully connected được tính như sau:

$$\text{parameters} = (\text{input}_{\text{numbers}} + 1) \times \text{output}_{\text{numbers}}$$

Kiến trúc mạng Alexnet được thể hiện như hình 2.6.1. Tầng tích chập đầu tiên được thực hiện trên ảnh đầu vào kích thước $224 \times 224 \times 3$ với 96 kernels kích thước $11 \times 11 \times 3$ với stride bằng 4. Tầng tích chập thứ hai gồm 256 kernels với kích thước $5 \times 5 \times 48$ sau đó qua lớp max-pooling. Tầng tích chập thứ ba có 384 kernels kích thước $3 \times 3 \times 256$. Tầng tích chập thứ tư có 384 kernels với kích thước $3 \times 3 \times 192$ và tầng tích chập thứ năm có 256 kernels kích thước $3 \times 3 \times 192$. Sau đó là các lớp fully connected và đầu ra cuối cùng là một vector softmax kích thước 1000 [2].

Số lượng thông số qua từng tầng là:

Tầng	Công thức xác định	Số lượng thông số
Tầng tích chập 1	$(11 \times 11 \times 3 + 1) \times 48 \times 2$	34,944
Tầng tích chập 2	$(5 \times 5 \times 48 + 1) \times 128 \times 2$	307,456
Tầng tích chập 3	$(3 \times 3 \times 128 + 1) \times 192 \times 2$	885,120
Tầng tích chập 4	$(3 \times 3 \times 192 + 1) \times 192 \times 2$	663,936
Tầng tích chập 5	$(3 \times 3 \times 192 + 1) \times 128 \times 2$	442,624
Tầng FC-1	$(6 \times 6 \times 256 + 1) \times 2048 \times 2$	37,752,832
Tầng FC-2	$(2048 \times 2 + 1) \times 2048 \times 2$	16,781,312
Tầng FC-3	$(2048 \times 2 + 1) \times 1000$	4,097,000
Tổng số thông số		60,965,224

Vậy tổng số lượng thông số của mạng Alexnet tính được là 60,965,224.

2.7 Bài 7

Phần code được viết bằng ngôn ngữ **Python** sử dụng thư viện **Pytorch**. Hướng dẫn chi tiết về lập trình Alexnet với Pytorch được giới thiệu ở [Alexnet with Pytorch](#).

Tập dữ liệu gồm 100 ảnh được trích từ tập validation trong bộ dữ liệu Imagenet. Mỗi lớp sẽ gồm 4 ảnh được đánh số từ 1 đến 4, tổng cộng có 25 lớp. Ví dụ lớp *corn* sẽ gồm 4 ảnh **corn1.jpeg**, **corn2.jpeg**, **corn3.jpeg**, **corn4.jpeg**.

Môi trường nhóm em sử dụng là GoogleColab. Ba mô hình mạng nhóm thử nghiệm là: **Alexnet**, **Resnet-18** và **Inception v3**.

2.7.1 Cài đặt các thư viện

```
import torch
from torchvision import transforms
from torchvision import models
from PIL import Image
import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Phần này cài đặt các thư viện của Pytorch và các thư viện hỗ trợ khác như numpy, os, matplotlib, ...

2.7.2 Đọc ảnh và gán nhãn cho ảnh

```
# Load image
path = '/content/drive/MyDrive/EE3063/HW7/image/'
file_name = os.listdir(path)
images = [Image.open(path + name) for name in file_name]
print(len(images))

[Out]: 100
```

Ta thực hiện đọc từng bức ảnh, tên của mỗi bức ảnh chứa đựng thông tin nhãn của lớp mà bức ảnh đó thuộc về. Chúng ta sẽ gán nhãn cho từng ảnh thông qua xử lý tên của mỗi ảnh.

```
#Get labels
labels = [name.split(name[-6])[0] for name in file_name]
```

2.7.3 Tiền xử lý dữ liệu

Ta sẽ tạo một `transforms` thực hiện điều chỉnh kích thước ảnh về 256×256 , sau đó cắt ảnh ở vị trí trung tâm về kích thước 224×224 . Dựa dữ liệu về dạng tensor và thực hiện chuẩn hóa trên từng kênh màu.



```
# Load image
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
```

Ta sẽ thực hiện transforms trên từng tấm ảnh, kết quả cuối cùng thu được một tensor kích thước $100 \times 3 \times 224 \times 224$.

```
transform_imgs = torch.empty(0, 3, 224, 224)
for img in images:
    tf_img = preprocess(img)
    tensor_img = torch.unsqueeze(tf_img, 0)
    transform_imgs = torch.cat((transform_imgs, tensor_img), dim = 0)

print(transform_imgs.shape)

[Out]: torch.Size([100, 3, 224, 224])
```

2.7.4 Tải các trọng số của mạng Alexnet

Thực hiện tải mô hình Alexnet đã được huấn luyện với tập Imagenet. Thư viện Pytorch.vision.model đã hỗ trợ các mô hình khác nhau².

```
alexnet = models.alexnet(pretrained=True)
print(alexnet)

[Out]: AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1,
      ceil_mode=False))
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
```

²Chi tiết xem tại [TORCHVISION.MODELS](#)

```
(1): Linear(in_features=9216, out_features=4096, bias=True)
(2): ReLU(inplace=True)
(3): Dropout(p=0.5, inplace=False)
(4): Linear(in_features=4096, out_features=4096, bias=True)
(5): ReLU(inplace=True)
(6): Linear(in_features=4096, out_features=1000, bias=True))
```

2.7.5 Thực hiện kết quả với mạng Alexnet

```
model = alexnet
model.eval()
predictions = model(transform_imgs)
output = torch.nn.functional.softmax(predictions, dim=-1)
```

Model sử dụng là mô hình Alexnet, kết quả sau khi thu được đưa hàm softmax với 1000 ngõ ra ứng với 1000 lớp của tập dữ liệu Imagenet.

Sau khi có kết quả ngõ ra, chúng ta sẽ tìm 5 kết quả dự đoán cao nhất cho mỗi tấm ảnh để tính top-5 error rate và top-1 error rate.

```
print('Top 1 error rate: %.2f' %(100*(1-true/top1_id.shape[0])))
print('Top 5 error rate: %.2f' %(100*(1-true/top5_id.shape[0])))
[Out 1]: Top 1 error rate: 41.00
[Out 2]: Top 5 error rate: 12.00
```

Kết quả thu được trên 100 tấm ảnh thử nghiệm, tỉ lệ lỗi top 1 là **41%** và tỉ lệ lỗi top 5 là **12%**. So sánh với bài báo gốc về mạng Alexnet, Tỉ lệ top 1 là **37.5%** và tỉ lệ lỗi top 5 là **17%**. Tỉ lệ lỗi top 5 trong tập thử nghiệm này tốt hơn so với bài báo thấp hơn **5%**, tuy nhiên tỉ lệ lỗi top 1 lại cao hơn **3.5%**. Sở dĩ có sự khác nhau này là do phụ thuộc vào tấm ảnh cũng như số lượng ảnh chúng ta lựa chọn để thử nghiệm.

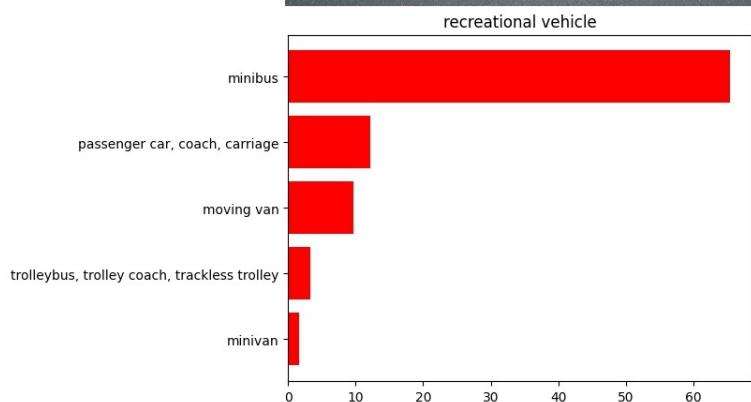
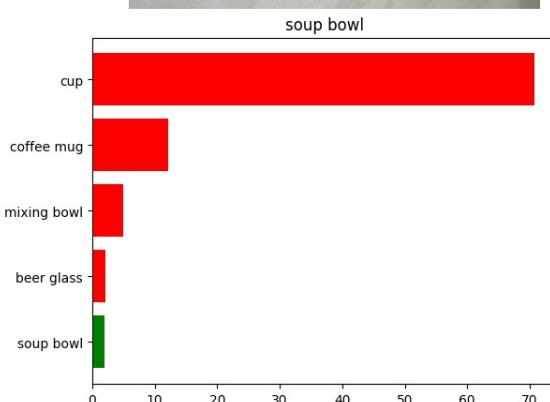
Sau đây là một số kết quả thu được:

- Một số kết quả mô hình hoạt động rất tốt, phân loại đúng ở ngay lớp đầu hoặc lớp thứ hai. Thậm chí, bức hình thứ nhất con hù xám lớn có kết quả xác suất phân loại top-1 là 100%.



Hình 2.7.1: Kết quả tốt thu được khi sử dụng mô hình Alexnet

- Đây là một số kết quả hoạt động không tốt. Đối với ảnh thứ nhất, bát súp bị nhầm lẫn thành tách trà, điều này có thể xảy ra khi hình dạng cái bát cũng rất giống hình dạng một cái tách. Tới kết quả ở top-5 thì cũng có dự đoán được bát canh, tuy nhiên xác suất rất thấp chỉ khoảng 2%. Đối với ảnh thứ hai thì sai hoàn toàn đối với top-5, đây là một chiếc xe dã ngoại tuy nhiên mô hình nhầm lẫn thành xe buýt mini với top-1 có xác suất khoảng 60%, top-2 là nhầm lẫn thành xe khách với xác suất khoảng 12%. Thành thật mà nói, ngay cả em cũng bị nhầm lẫn khi nhìn bức hình và cũng đã kiểm tra lại nhãn dữ liệu khi nhận kết quả này. Vì trông nó rất giống xe buýt mini hay xe khách.



Hình 2.7.2: Kết quả không tốt thu được khi sử dụng mô hình Alexnet

2.7.6 Kết quả thực hiện trên mô hình Inception v3

Đầu tiên, chúng ta cũng thực hiện tải thông số của mạng Inception v3.

```
inception = models.inception_v3(pretrained=True)
```

Sau đó sử dụng thông số của mô hình Inception v3 để dự đoán kết quả cho từng tấm ảnh.

```
model = inception
model.eval()
predictions = model(transform_imgs)
output = torch.nn.functional.softmax(predictions, dim=-1)
```

Kết quả tính toán top-5 error rate và top-1 error rate là

```
print('Top 1 error rate: %.2f' %(100*(1-true/top1_id.shape[0])))
print('Top 5 error rate: %.2f' %(100*(1-true/top5_id.shape[0])))
[Out 1]: Top 1 error rate: 33.00
```

[Out 2]: Top 5 error rate: 11.00



yawl

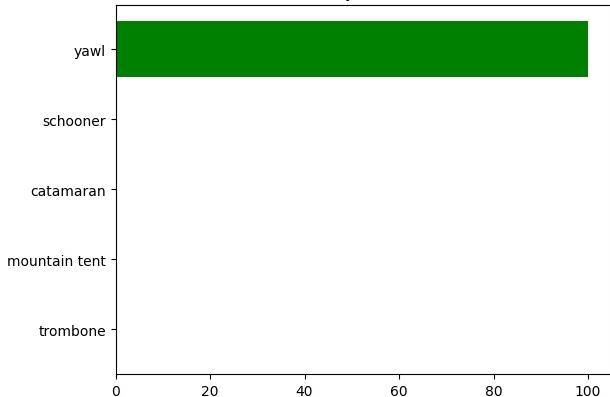
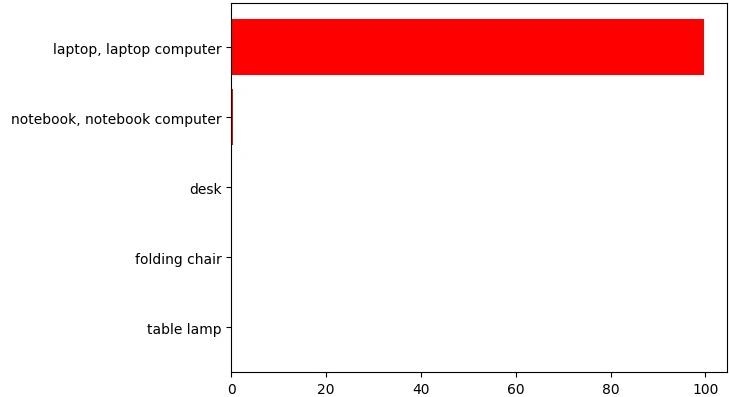


table lamp



Hình 2.7.3: Kết quả thu được khi sử dụng mô hình Inception v3

Kết quả ở tấm hình đầu tiên cho kết quả rất tốt khi phân lớp đúng vào lớp thuyền buồm với xác suất rất cao 99%. Đối với tấm ảnh thứ hai, mô hình phân vào lớp máy tính xách tay với xác suất rất cao 98% trong khi nó là đèn để bàn.

2.7.7 Kết quả thực hiện trên mô hình Resnet18

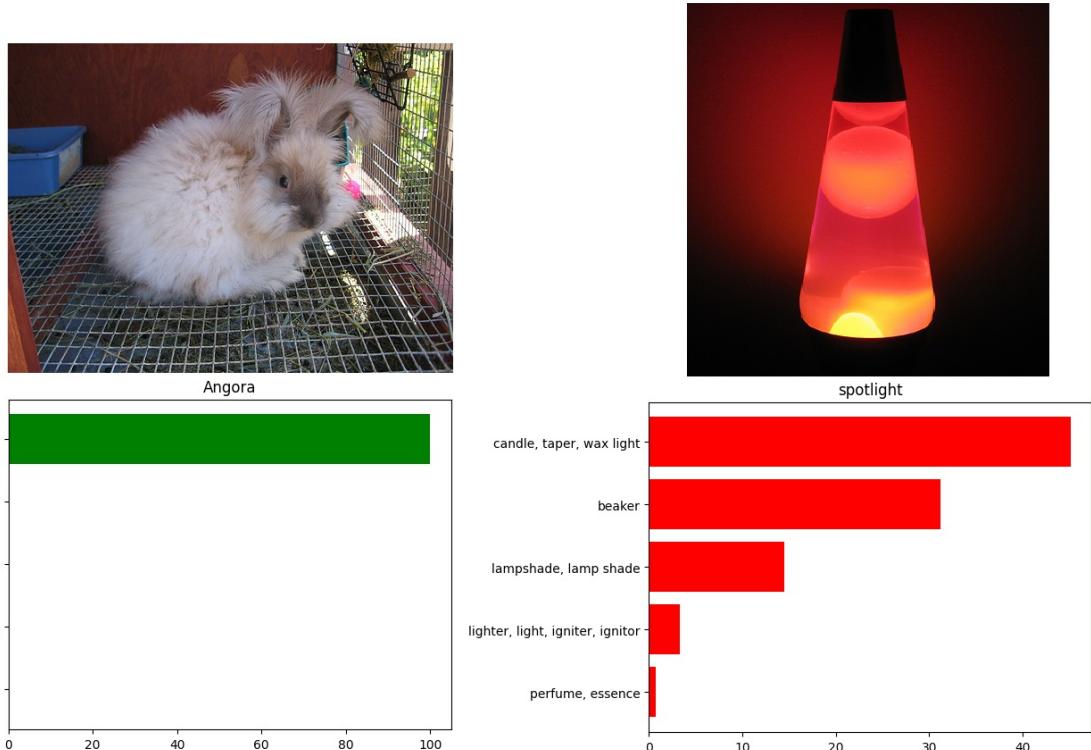
Các bước thực hiện cũng giống như trên, chỉ thay đổi tên mô hình sử dụng.

```
resnet18 = models.resnet18(pretrained=True)

model = resnet18
model.eval()
predictions = model(transform_imgs)
output = torch.nn.functional.softmax(predictions, dim=-1)
```

Kết quả tính toán top-5 error rate và top-1 error rate là

```
print('Top 1 error rate: %.2f' %(100*(1-true/top1_id.shape[0])))
print('Top 5 error rate: %.2f' %(100*(1-true/top5_id.shape[0])))
[Out 1]: Top 1 error rate: 26.00
[Out 2]: Top 5 error rate: 4.00
```



Hình 2.7.4: Kết quả thu được khi sử dụng mô hình Resnet18

Kết quả ở tấm hình đầu tiên cho kết quả rất tốt khi thỏ angora đã được phân lớp đúng với xác suất 100%. Đối với tấm ảnh thứ hai là đèn spotlight, mô hình đã phân lớp sai vào lớp nến với xác suất khoảng 48%.

2.7.8 So sánh kết quả

Đây là bảng so sánh tỉ lệ lỗi top-1 và top-5 đối với 3 mô hình sử dụng:

Mô hình	Top-1 error rate	Top-5 error rate
Alexnet	41%	12%
Inception v3	33%	11%
Resnet18	26%	4%

Hai mô hình Inception v3 và Resnet18 đều cho kết quả cao hơn so với mô hình mạng Alexnet, kết quả khi sử dụng Resnet18 cũng tốt hơn so với khi sử dụng Inception v3. Cụ thể, đối với mô hình Inception v3 tỉ lệ lỗi top-1 đã giảm 8% và tỉ lệ lỗi top-5 đã giảm 1% so với Alexnet.

Mô hình Resnet18 cho kết quả tốt hơn, khi tỉ lệ lỗi top-1 giảm 15% hay xấp xỉ 1.58 lần so với Alexnet, kết quả cho tỉ lệ lỗi top-5 cũng giảm 8% là 3 lần so với Alexnet. Như vậy, mô hình Resnet18 hoạt động tốt hơn so với mô hình Alexnet cũng như Inception v3.

2.8 Bài 8

Về phần lập trình, code trong bài tập 8 phần lớn tương tự như trong bài tập 7, chỉ có một chút thay đổi ở phần xử lý dữ liệu.

Về dữ liệu, nhóm sử dụng tám ảnh con hù xám lớn có độ chính xác dự đoán top-1 trong bài 7 khi dùng Alexnet là 100% (hình 2.7.1). Sau được thực hiện điều chỉnh tỉ lệ kích thước ảnh khác nhau. Cụ thể như tỉ lệ W:H = 10:10, 10:9, 10:8, 10:7, 10:6, 10:5, 10:4, 10:3, 10:2, 10:1, 9:10, 8:10, 7:10, 6:10, 5:10, 4:10, 3:10, 2:10, 1:10. Sau đó thực hiện thêm padding cho các tấm ảnh, điều chỉnh kích thước về 224×224 , tổng cộng có 19 tấm ảnh.

```
def expand2square(pil_img, background_color):
    width, height = pil_img.size
    if width == height:
        return pil_img
    elif width > height:
        result = Image.new(pil_img.mode, (width, width), background_color)
        result.paste(pil_img, (0, (width - height) // 2))
        return result
    else:
        result = Image.new(pil_img.mode, (height, height), background_color)
        result.paste(pil_img, ((height - width) // 2, 0))
        return result
```



Hình 2.8.1: Dữ liệu sau khi padding

2.8.1 Thực hiện với Alexnet

Ta cũng thực hiện tải trọng số của mạng Alexnet đã được huấn luyện sẵn.

```
alexnet = models.alexnet(pretrained=True)
```

Sau đó thực hiện tính accuracy:

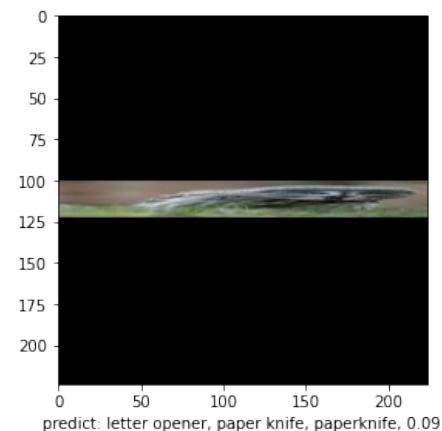
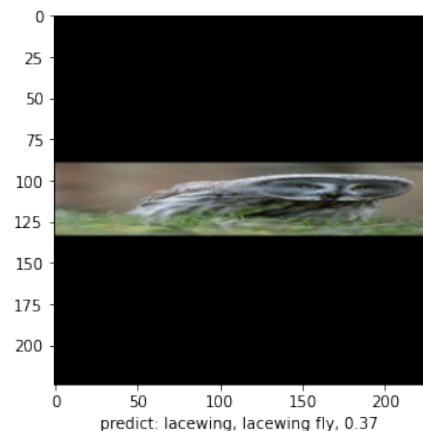
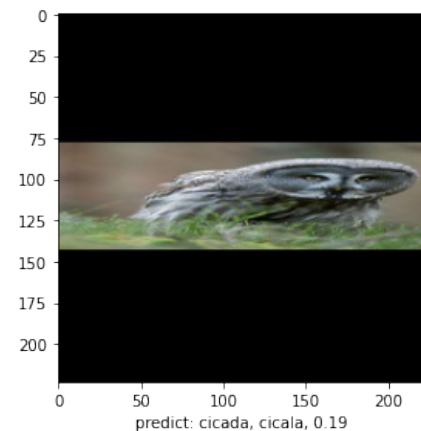
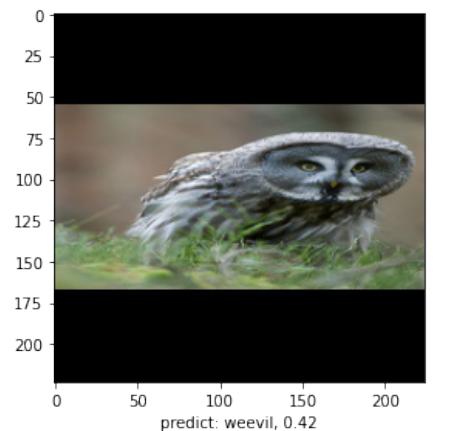
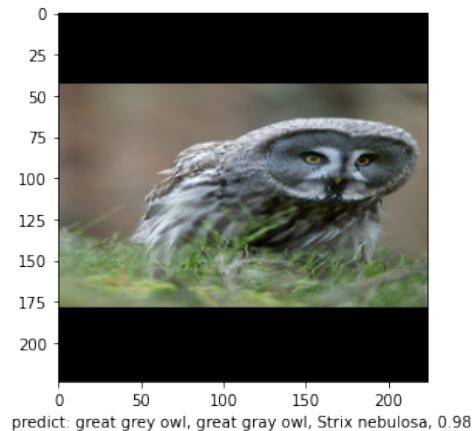
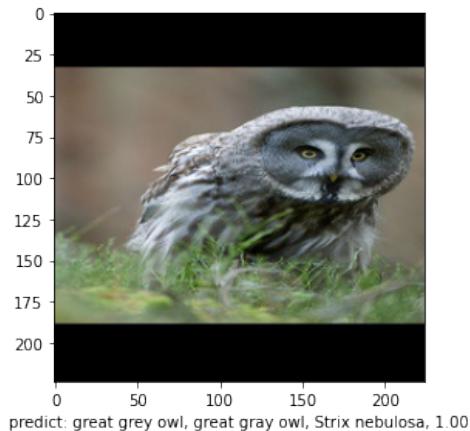
```
top1_prob, top1_id = torch.topk(output, 1)
true = 0

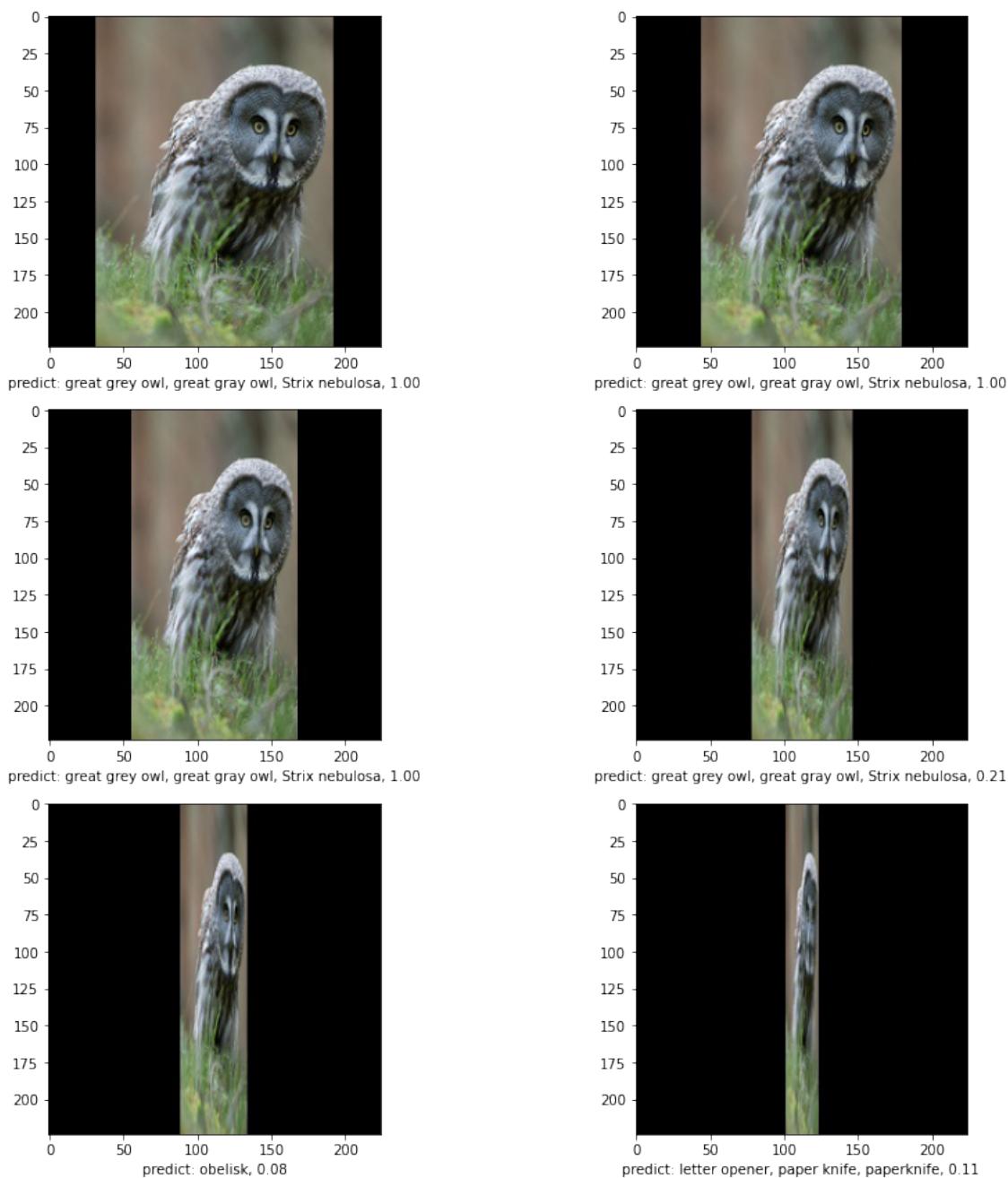
for i in range(top1_id.shape[0]):
    if categories[top1_id[i][0]].startswith(labels[i]):
        true += 1

print('Accuracy: %.2f' %(100*(true/top1_id.shape[0])))
```

[Out]: Accuracy: 63.16

Dây là một số kết quả dự đoán của model, 9 tấm ảnh được lấy có tỉ lệ kích thước lần lượt là 10:7, 10:6, 10:5, 10:3, 10:2, 10:1, 7:10, 6:10, 5:10, 3:10, 2:10 và 1:10.





Hình 2.8.2: Kết quả dự đoán với model Alexnet

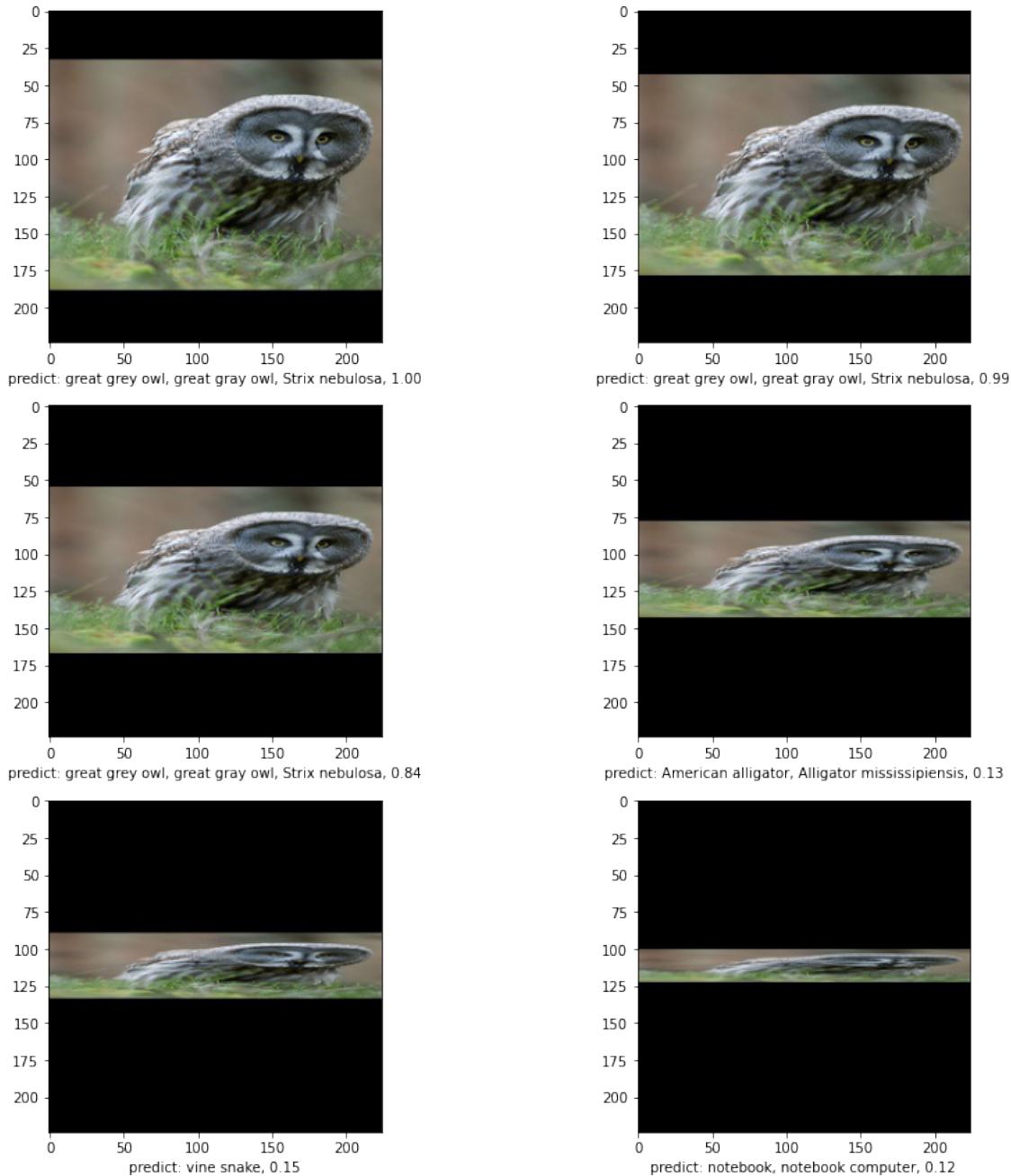
Kết quả thực hiện, độ chính xác của mạng Alexnet cho tập dữ liệu này là 63.16%. Các tấm ảnh có tỉ lệ 10:7, 10:6 dự đoán rất tốt, tuy nhiên qua đến các tấm 10:5, 10:3, 10:2, 10:1 thì đã dự đoán sai lớp. Đặc biệt đối với ảnh tỉ lệ 10:1, thì gần như model không thể đưa ra kết quả do xác suất dự đoán top-1 chỉ có 9% rất thấp. Có lẽ là do con hù đã bị kéo căng ra và model không thể xác định được những đặc trưng vốn có của nó. Tương tự cho 9 tấm còn lại, các tấm có tỉ lệ 7:10, 6:10, 5:10, 3:10 cho kết quả phân lớp đúng, tuy đối với ảnh 3:10 xác suất dự đoán khá thấp. Đối với ảnh 1:10 và 2:10, model cũng cho kết quả xác suất dự đoán thấp chỉ có 11% và 9%.

2.8.2 Thực hiện với Resnet18

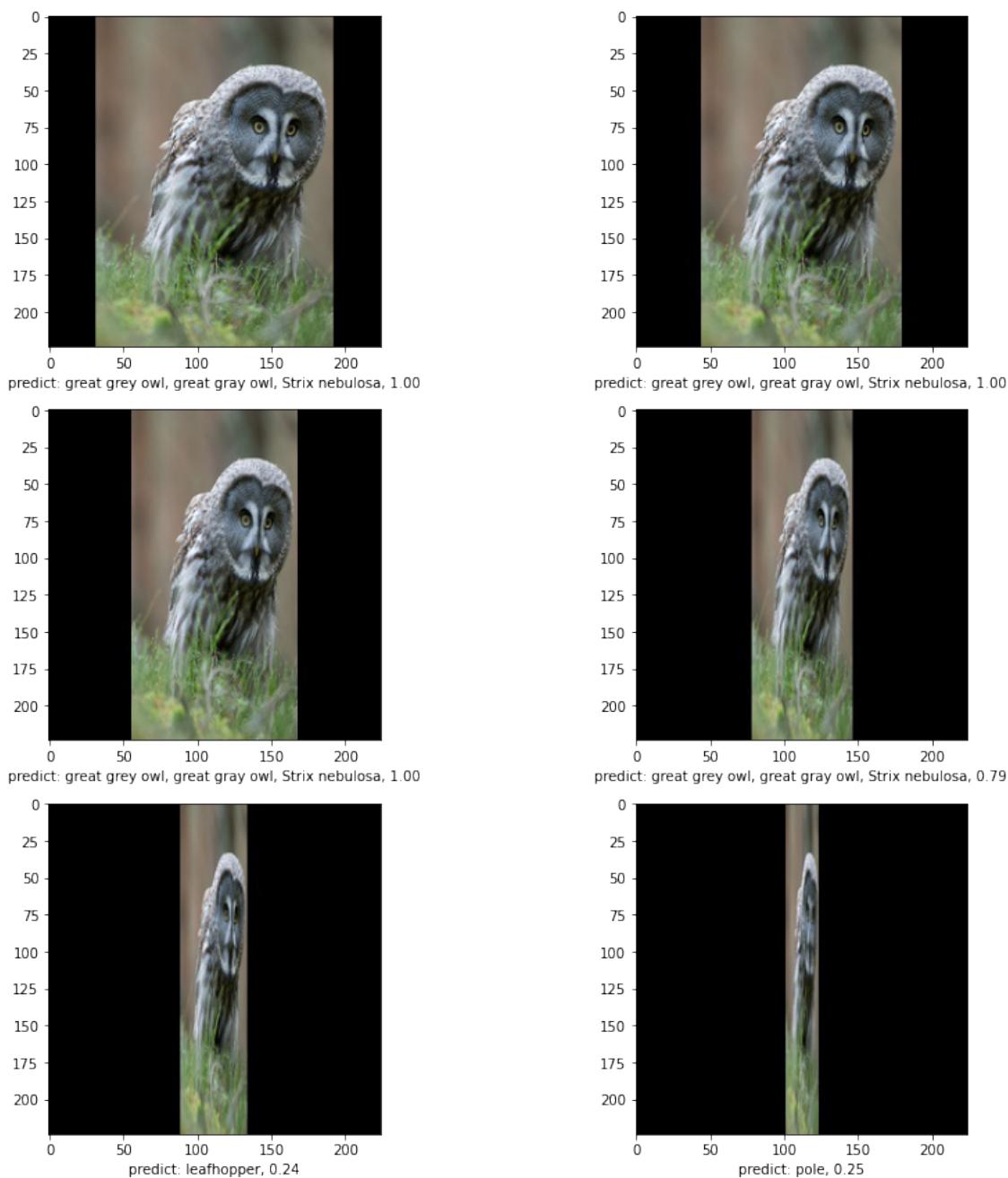
Thực hiện tương tự đối với model resnet18, chúng ta cũng tiến hành tải model và dự đoán kết quả.

```
resnet18 = models.resnet18(pretrained=True)
print('Accuracy: %.2f' %(100*(true/top1_id.shape[0])))
```

[Out]: Accuracy: 73.68



Kết quả cho thấy, độ chính xác của mạng Resnet18 cho tập dữ liệu này là 73.68%. Kết quả phân lớp đúng cho các ảnh có tỉ lệ 10:7, 10:6, 10:5, 7:10, 6:10, 5:10 và 3:10. Phân lớp sai cho các tấm ảnh có tỉ lệ kích thước 10:3, 10:2, 10:1, 2:10 và 1:10. Xác suất dự đoán top-1 cho các tấm ảnh phân lớp sai này cũng rất thấp.



Hình 2.8.3: Kết quả dự đoán với model Resnet18

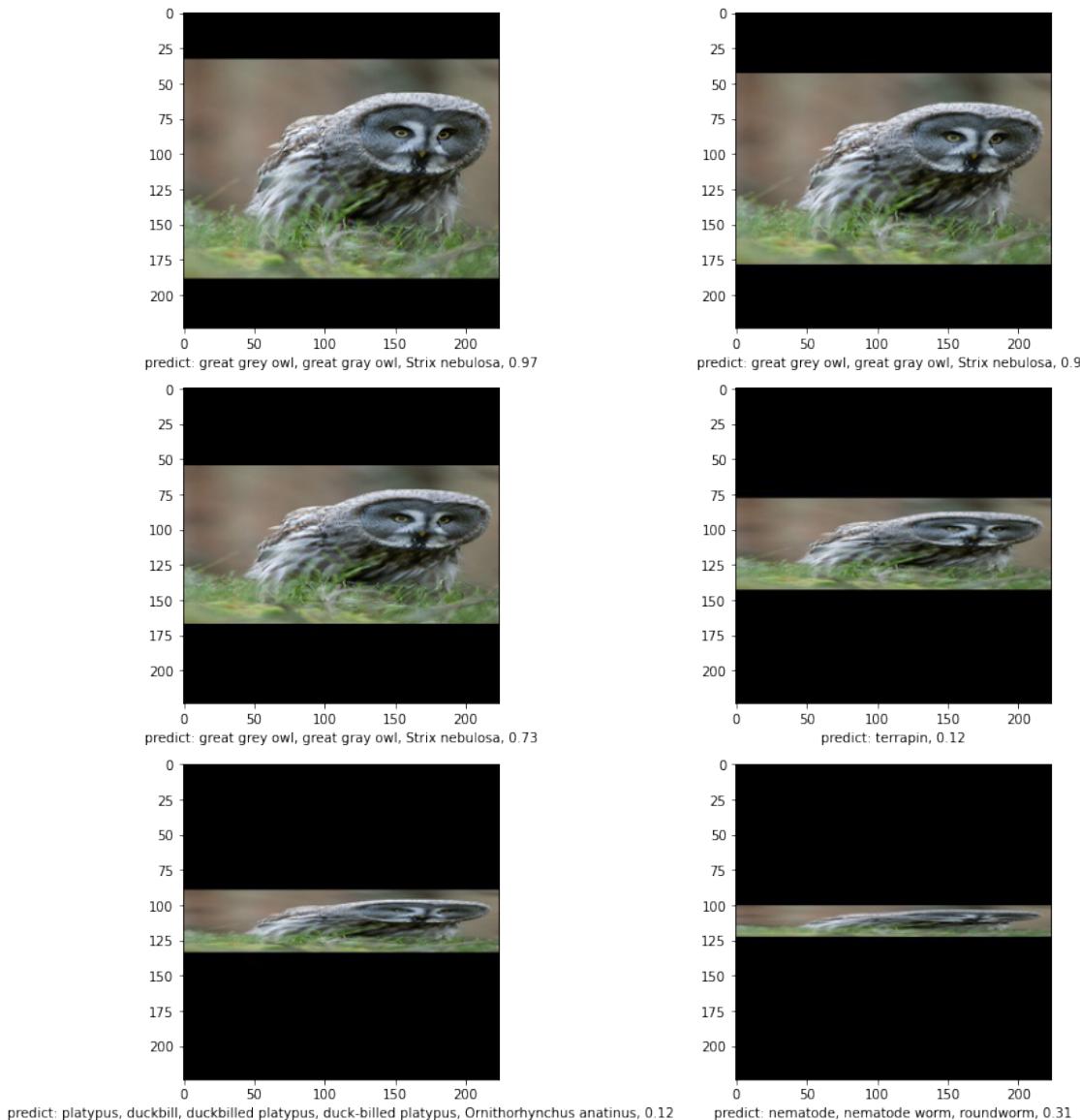
2.8.3 Thực hiện với GoogleNet

Ta tiến hành tải thông số của mạng GoogleNet và tiến hành dự đoán phân lớp, tính toán accuracy cho tập dữ liệu:

```
print('Accuracy: %.2f' %(100*(true/top1_id.shape[0])))
```

[Out]: Accuracy: 73.68

Kết quả thu được accuracy là 63.16%, model GoogleNet cũng hoạt động tốt khi phân loại được các ảnh có tỉ lệ kích thước 10:7, 10:6, 10:5, 7:10, 6:10, 5:10, 3:10 và phân lớp sai đối với tỉ lệ kích thước 10:3, 10:2, 10:1, 2:10 và 1:10.

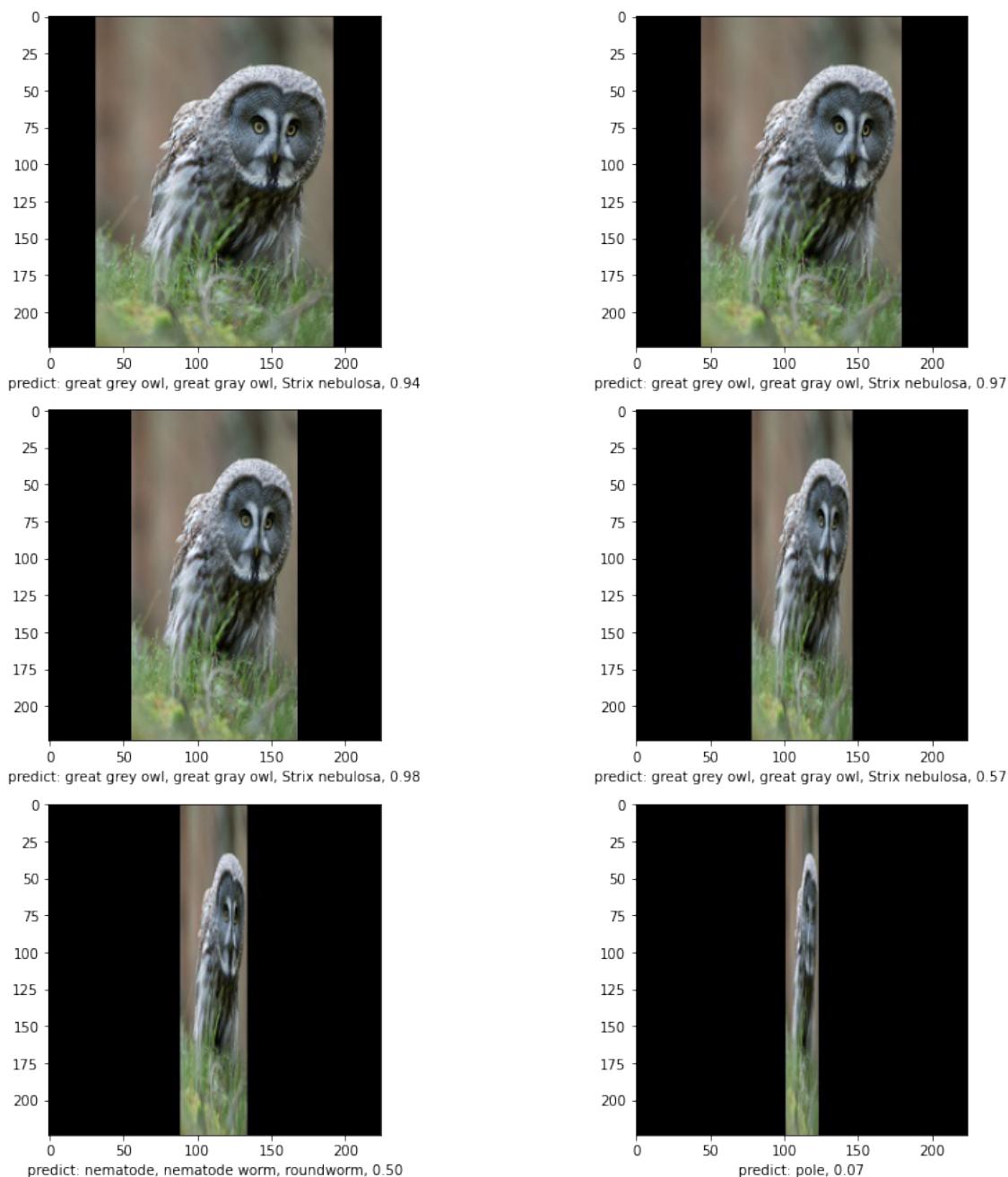


2.8.4 Nhận xét

Sau đây là bảng tổng hợp kết quả accuracy của tập dữ liệu khi thực hiện trên ba model khác nhau.

<i>Model</i>	<i>Accuracy</i>
Alexnet	63.16%
Resnet18	73.68%
GoogleNet	73.68%

Kết quả cho thấy, hai model resnet18 và googlenet có accuracy bằng nhau 73.68% và hoạt động tốt hơn so với alexnet khi có accuracy cao hơn 10.52%. Model Alexnet phân lớp sai đối với các ảnh có tỉ lệ kích thước 10:5, 10:4, 10:3, 10:2, 10:1, 2:10, 1:10. Hai model resnet18 và googlenet đều phân loại sai ở các ảnh có tỉ lệ 10:3, 10:2, 10:1, 2:10 và 1:10. Tuy nhiên, đối với cùng một ảnh thì các model lại đưa ra các kết quả dự đoán sai khác nhau. Ví dụ đối với ảnh tỉ lệ 10:2, mạng alexnet dự đoán là lacewing, mạng resnet18 dự đoán vine snake còn mạng googlenet thì dự đoán là platypus.



Hình 2.8.4: Kết quả dự đoán với model GoogleNet

Như vậy, khi ta thay đổi tỉ lệ kích thước hình ảnh quá nhiều, trực quan là ta kéo căng hay làm dẹt tấm ảnh thì sẽ làm cho các model hoạt động không tốt. Đối với các sự thay đổi tỉ lệ nhỏ như 10%, 20% thì các model còn hoạt động tốt, nhưng nếu ta thay đổi 80%, 90% thì model gần như không thể dự đoán kết quả đúng đắn được. Vì khi ta thay đổi tỉ lệ kích thước ảnh, các đặc trưng về hình dáng, góc cạnh của vật thể thay đổi quá nhiều dẫn đến mô hình sẽ cho ra kết quả không chính xác.

2.9 Bài 9

Phần lập trình được thực hiện bằng python với thư viện **Pytorch**. Chi tiết về hướng dẫn lập trình học chuyển tiếp với Pytorch được giới thiệu ở [Transfer Learning Tutorial with Pytorch for beginner](#).

Dữ liệu trong bài này do nhóm **tự chụp khoảng 100 tấm**, tìm kiếm **trên internet** khoảng **90 tấm**. Tập dữ liệu gồm có **193 tấm ảnh** thuộc **10 lớp** là hammer, bolts, wrench, screw, plastic screw anchors, screwdriver, pliers, soldering iron, hex nuts và paper knives liên quan đến các dụng cụ sửa chữa trong nhà. Trong đó, tập huấn luyện (training set) có 113 ảnh, tập kiểm thử (validation set) có 40 ảnh và tập kiểm tra (test set) có 40 ảnh với tỉ lệ chia xấp xỉ **60-20-20**.

2.9.1 Cài đặt các thư viện

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import transforms, models, datasets

from PIL import Image
import os
import sys
import time
import copy

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Tiếp theo là kiểm tra phần cứng sử dụng là cpu hay gpu thông qua lệnh:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

2.9.2 Xử lí dữ liệu ảnh

Bước đầu tiên, chúng ta sẽ tạo transforms cho ảnh. Đối với ảnh thuộc tập huấn luyện ngoài việc chỉnh kích thước ảnh và cắt ảnh về kích thước 224×224 ta còn thực hiện lật ảnh theo chiều ngang, lật theo chiều dọc và thay đổi độ bão hòa của ảnh. Đối với ảnh thuộc tập kiểm thử ta chỉ điều chỉnh kích thước ảnh.

```
# Transform data
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
```



```
        transforms.ColorJitter(saturation=0.5),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229,
            0.224, 0.225])
    ],
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229,
            0.224, 0.225])
    ])
}
```

Tiếp theo, ta tiến hành đọc ảnh và chuyển về dạng tensor trong pytorch với kích thước batch là 8.

```
path = '/content/drive/MyDrive/EE3063/HW9/image'
batch_size = 8
num_classes = 10

data = {x: datasets.ImageFolder(os.path.join(path, x), data_transforms[x])
        for x in ['train', 'val']}

# Load data
train_data = DataLoader(data['train'], batch_size, shuffle=True)
val_data = DataLoader(data['val'], batch_size, shuffle=True)

train_size = len(data['train'])
val_size = len(data['val'])
print(train_size)
print(val_size)

[Out 1]: 113
[Out 2]: 40
```

2.9.3 Thay đổi mô hình Alexnet

Ta thực hiện tải mô hình Alexnet và điều chỉnh các tầng ở phía cuối để huấn luyện lại.

```
alexnet = models.alexnet(pretrained=True)
model = alexnet

for para in model.parameters():
    para.requires_grad = False
model.classifier[6] = nn.Linear(in_features=4096, out_features=10, bias=True)
model.classifier.add_module("7", nn.LogSoftmax(dim = 1))
print(model)
model.to(device)
```

[Out] :

```
AlexNet(  
    (features): Sequential(  
        (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))  
        (1): ReLU(inplace=True)  
        (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)  
        (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))  
        .....  
    (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))  
    (classifier): Sequential(  
        (0): Dropout(p=0.5, inplace=False)  
        (1): Linear(in_features=9216, out_features=4096, bias=True)  
        (2): ReLU(inplace=True)  
        (3): Dropout(p=0.5, inplace=False)  
        (4): Linear(in_features=4096, out_features=4096, bias=True)  
        (5): ReLU(inplace=True)  
        (6): Linear(in_features=4096, out_features=10, bias=True)  
        (7): LogSoftmax(dim=1)))
```

Sau khi tải xong các thông số mô hình Alexnet, ta thực hiện "đóng băng" các thông số đã được huấn luyện sẵn. Sau đó, thực hiện thay đổi lớp FCN thứ 6 với 10 đầu ra ứng với 10 lớp của bài toán. Rồi thêm một lớp thứ 7 để tính softmax đầu ra.

2.9.4 Huấn luyện mô hình

Hàm train model dùng để huấn luyện mô hình. Ta sẽ tiến hành tính hàm loss và thực hiện các thuật toán tối ưu để tìm các trọng số trong lớp FCN thứ 6 và thứ 7 trình bày ở phần trên. Hàm này thực hiện trả về mô hình có thông số validation accuracy cao nhất trong toàn bộ quá trình training. Đồng thời, list history sẽ lưu trữ giá trị loss và accuracy của tập training và tập validation sau mỗi epoch để ta vẽ hình quan sát.

```
def train_model(model, criterion, optimizer, num_epochs=5):  
    since = time.time()  
    history = []  
  
    best_model_wts = copy.deepcopy(model.state_dict())  
    best_acc = 0.0  
  
    for epoch in range(num_epochs):  
  
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))  
        print('-' * 10)  
  
        train_loss = 0.0  
        train_acc = 0.0  
        val_loss = 0.0  
        val_acc = 0.0  
  
        # Training  
        model.train()  
        for train_x, train_y in train_data:  
            train_x = train_x.to(device)
```

```
train_y = train_y.to(device)
output_train = model(train_x)
loss_train = criterion(output_train, train_y)

# zero the parameter gradients
optimizer.zero_grad()
loss_train.backward()

# gradient descent or adam step
optimizer.step()

train_loss += loss_train.item() * train_x.size(0)
_, train_pred = torch.max(output_train.data, 1)
correct_train = train_pred.eq(train_y.data.view_as(train_pred))
acc_train = torch.mean(correct_train.type(torch.FloatTensor))
train_acc += acc_train.item() * train_x.size(0)

# Validation
with torch.no_grad():
    model.eval()

    for val_x, val_y in val_data:
        val_x = val_x.to(device)
        val_y = val_y.to(device)
        output_val = model(val_x)
        loss_val = criterion(output_val, val_y)

        val_loss += loss_val.item() * val_x.size(0)
        _, val_pred = torch.max(output_val.data, 1)
        correct_val = val_pred.eq(val_y.data.view_as(val_pred))
        acc_val = torch.mean(correct_val.type(torch.FloatTensor))
        val_acc += acc_val.item() * val_x.size(0)

    avg_train_loss = train_loss / train_size
    avg_train_acc = train_acc / train_size
    avg_val_loss = val_loss / val_size
    avg_val_acc = val_acc / val_size

history.append([avg_train_loss, avg_val_loss, avg_train_acc, avg_val_acc])

print('{} Loss: {:.4f} Acc: {:.4f}'.format('train', avg_train_loss,
                                             avg_train_acc))
print('{} Loss: {:.4f} Acc: {:.4f}'.format('val', avg_val_loss,
                                             avg_val_acc))

# deep copy the model
if avg_val_acc >= best_acc:
    best_acc = avg_val_acc
    best_model_wts = copy.deepcopy(model.state_dict())

time_elapsed = time.time() - since
print('Training complete in {:.0f}m {:.0f}s'.format(
```

```
    time_elapsed // 60, time_elapsed % 60))
print('Best val Acc: {:.4f}'.format(best_acc))

# load best model weights
model.load_state_dict(best_model_wts)
return model, history
```

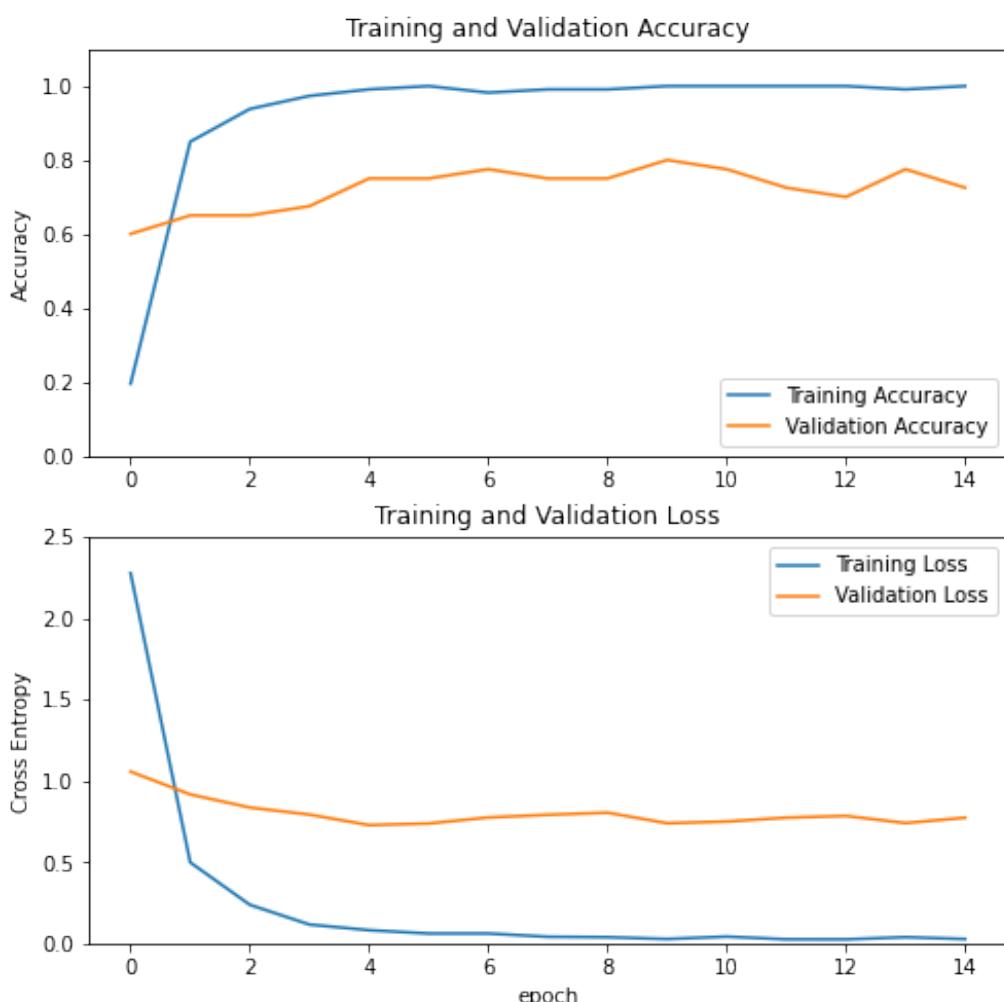
Tiếp theo ta sẽ tiến hành lựa chọn hàm mất mát (loss function) và thuật toán tối ưu (optimizer). Ở bài toán này, nhóm em lựa chọn hàm Loss Negative Log-Likelihood và thuật toán tối ưu là Adam với hệ số học khởi tạo $lr = 0.001$.

```
criterion = nn.NLLLoss()
# Observe that all parameters are being optimized
optimizer = optim.Adam(alexnet.parameters(), lr=1e-3)
```

Tiến hành huấn luyện model bằng lệnh:

```
model, history = train_model(model, criterion, optimizer, num_epochs=15)
```

Sau 15 epochs huấn luyện model, kết quả thu được như sau:



Hình 2.9.1: Loss và Accuracy của quá trình huấn luyện



```
Epoch 0/14
-----
train Loss: 2.2768 Acc: 0.1947
val Loss: 1.0569 Acc: 0.6000
Epoch 1/14
-----
train Loss: 0.4986 Acc: 0.8496
val Loss: 0.9164 Acc: 0.6500
Epoch 2/14
-----
train Loss: 0.2377 Acc: 0.9381
val Loss: 0.8359 Acc: 0.6500
Epoch 3/14
-----
train Loss: 0.1164 Acc: 0.9735
val Loss: 0.7924 Acc: 0.6750
Epoch 4/14
-----
.....
-----
train Loss: 0.0383 Acc: 0.9912
val Loss: 0.8042 Acc: 0.7500
Epoch 9/14
-----
train Loss: 0.0272 Acc: 1.0000
val Loss: 0.7393 Acc: 0.8000
Epoch 10/14
-----
train Loss: 0.0419 Acc: 1.0000
val Loss: 0.7496 Acc: 0.7750
Epoch 11/14
-----
train Loss: 0.0254 Acc: 1.0000
val Loss: 0.7736 Acc: 0.7250
Epoch 12/14
-----
train Loss: 0.0250 Acc: 1.0000
val Loss: 0.7844 Acc: 0.7000
Epoch 13/14
-----
train Loss: 0.0385 Acc: 0.9912
val Loss: 0.7402 Acc: 0.7750
Epoch 14/14
-----
train Loss: 0.0271 Acc: 1.0000
val Loss: 0.7733 Acc: 0.7250

Training complete in 0m 19s
Best val Acc: 0.800000
```

Kết quả cho thấy, accuracy của tập kiểm thử chỉ rơi vào khoảng 80%. Kết quả này không quá cao như mong đợi, tuy nhiên cũng tạm chấp nhận được với tập dữ liệu này.

2.9.5 Đánh giá

Chúng ta tiến hành tính tỉ lệ lỗi top-1 và top-3, kết quả thu được như sau:

```
print('Top-3 error rate: %.2f' %(100*(1-true/top3_id.shape[0])))
print('Top-1 error rate: %.2f' %(100*(1-true/top1_id.shape[0])))
[Out 1]: Top-3 error rate: 10.00
[Out 2]: Top-1 error rate: 32.50
```



Hình 2.9.2: Kết quả thu được sau khi huấn luyện

Đối với hai tấm ảnh đầu tiên, model đã hoạt động tốt khi phân loại đúng vào lớp của tấm ảnh, đối với con tán là 96% và đối với cờ lê là 100%. Tuy nhiên, đối với hai tấm ảnh dao rọc giấy và mỏ hàn thì model đã phân lớp sai khi cho rằng chúng thuộc lớp tua vít (crewdriver). Bởi tua vít giống với hai vật này ở điểm đều sẽ có một lớp nhựa bên dưới và một đầu kim loại ở trên.

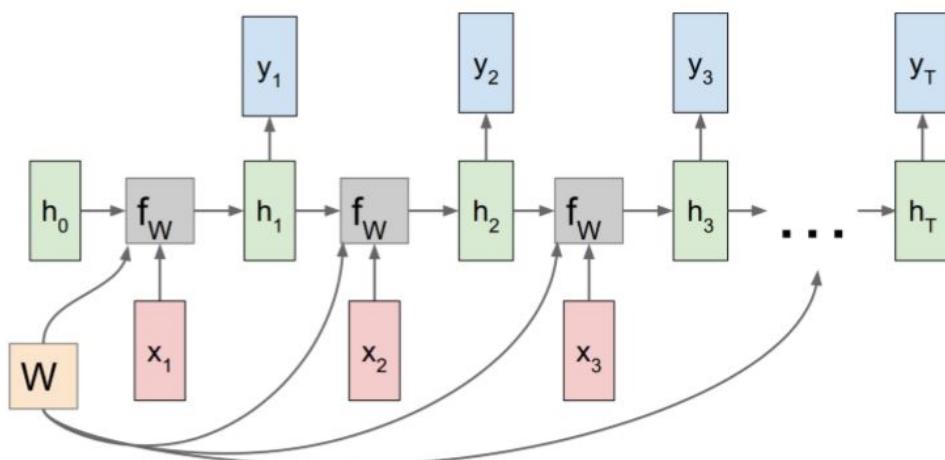
Bằng cách sử dụng transfer learning với mô hình mạng Alexnet, ta đã huấn luyện được một model phân loại làm việc khá tốt với tập dữ liệu nhỏ chỉ khoảng 200 tấm ảnh thuộc 10 lớp, kết quả thu được có tỉ lệ lỗi top-3 là 10% và tỉ lệ lỗi top-1 là 32.5%.

2.10 Bài 10

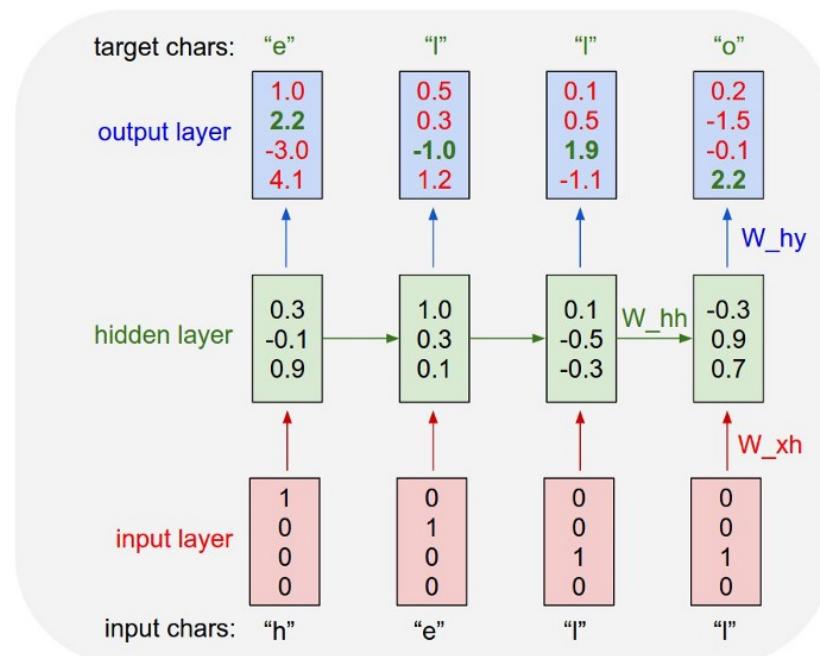
Bài này nhóm em sẽ sử dụng Pygad để thực hiện thuật toán. Pygad là một thư viện Python mã nguồn mở để xây dựng thuật toán di truyền GA (Genetic Algorithm) và tối ưu hóa các thuật toán học máy [3].

2.10.1 Sử dụng GA (Genetic Algorithm) để tối ưu hóa hàm mất mát của mạng nơron hồi tiếp (RNN)

Ta thực hiện tối ưu hàm mất mát của mạng RNN bằng thuật toán GA khi huấn luyện model học với từ "hello". Mô hình học sẽ nhận ngõ vào ban đầu là kí tự "h" và cho các ngõ ra lần lượt là các kí tự "e", "l", "l", "o".



Hình 2.10.1: Mô hình mạng RNN. Nguồn: <https://bitly.com.vn/gxb8uk>



Hình 2.10.2: Model học với từ 'Hello'. Nguồn: <https://bitly.com.vn/h24ug5>



Ta tiến hành mã hóa các kí tự trong chuỗi 'Hello' theo phương pháp one-hot vector. Với kí tự "H":[1 0 0 0], "e" : [0 1 0 0], "l" : [0 0 1 0], "o" : [0 0 0 1]. Sau đó định nghĩa một số hàm cần thiết cho việc tính toán.

```
import pygad
import numpy as np
import scipy
X= np.array([[1, 0, 0, 0]]).T
Y = np.array([[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 0, 1]]).T
def softmax(x):
    return np.exp(x)/sum(np.exp(x))
def crossEntropy(X,Y):
    return -np.sum(np.log(X)*Y)
```

Tiếp theo sử dụng thư viện Pygad chạy thuật toán GA với mạng RNN.

```
def GA_RNN(solution, solution_idx):
    Wxh = solution[0:12].reshape(3, 4)
    Whh = solution[12:21].reshape(3, 3)
    Why = solution[21:33].reshape(4, 3)
    bh = solution[33:36].reshape(3, 1)
    by = solution[36: ].reshape(4, 1)
    hs = np.zeros((4, 1))
    xs = np.zeros((4, 4))
    h_s = np.zeros((3, 1))
    ys = X
    # ps = np.zeros((4, 1))

    for t in range(4):
        xs = ys[:, -1].reshape(4, 1)# encode in 1-of-k representation
        # hidden state
        h_t = np.tanh(np.dot(Wxh, xs) + np.dot(Whh, h_s[:, -1].reshape(3, 1)) + bh)
        h_s = np.hstack((h_s, h_t))
        # unnormalized log probabilities for next chars
        y_t = softmax(np.dot(Why, h_t) + by)
        ys = np.hstack((ys, y_t))

        h_s = h_s[:, 1:]
        ys = ys[:, 1:]
    return 1/np.sum(crossEntropy(ys, Y))

    h_s = h_s[:, 1:]
    ys = ys[:, 1:]
    return 1/np.sum(crossEntropy(ys, Y))
fitness_function = GA_RNN
num_generations = 100
num_parents_mating = 10
sol_per_pop = 20
num_genes = 40
init_range_low = -2
init_range_high = 5
parent_selection_type = "sss"
keep_parents = 1
```

```
crossover_type = "single_point"
mutation_type = "random"
mutation_percent_genes = 20

ga_instance = pygad.GA(
    num_generations=num_generations,
    num_parents_mating=num_parents_mating,
    fitness_func=fitness_function,
    sol_per_pop=sol_per_pop,
    num_genes=num_genes,
    init_range_low=init_range_low,
    init_range_high=init_range_high,
    parent_selection_type=parent_selection_type,
    keep_parents=keep_parents,
    crossover_type=crossover_type,
    mutation_type=mutation_type,
    mutation_percent_genes=mutation_percent_genes)
ga_instance.run()
solution, solution_fitness, solution_idx = ga_instance.best_solution()
```

Kết quả sau khi chạy đoạn code trên được các thông số của mạng RNN.

Áp dụng các thông số vừa tìm được ở trên vào mô hình mạng RNN. Sử dụng hàm cross-entropy để đánh giá kết quả thu được.

```
def RNN(solution, solution_idx, X, Y):
    Wxh = solution[0:12].reshape(3,4)
    Whh = solution[12:21].reshape(3,3)
    Why = solution[21:33].reshape(4,3)
    bh = solution[33:36].reshape(3,1)
    by = solution[36: ].reshape(4,1)
    hs = np.zeros((4,1))
    Loss = 0
    xs = np.zeros((4,4))
    h_s = np.zeros((3,1))
    ys = X
    # ps = np.zeros((4,1))

    for t in range(X.shape[0]):
        xs = ys[:, -1].reshape(4,1) # encode in 1-of-k representation
        # hidden state
        h_t = np.tanh(np.dot(Wxh, xs) + np.dot(Whh, h_s[:, -1].reshape(3,1)) + bh)
        h_s = np.hstack((h_s, h_t))
        # unnormalized log probabilities for next chars
        y_t = softmax(np.dot(Why, h_t) + by)
        ys = np.hstack((ys, y_t))

    h_s = h_s[:, 1:]
    ys = ys[:, 1:]
    return h_s, ys, crossEntropy(ys, Y)

h, y, Loss = RNN(solution, solution_idx, X, Y)
print("Kết quả thu được y = \n {} \n Kết quả sau khi làm tròn 3 chữ số y= \n {}.\n Loss= {} ".format(y, np.round(y, decimals=3), Loss))
```

Wxh:

```
[[ -1.186 -2.404 -0.586  3.923]
 [ 5.033  4.317 -1.629  3.363]
 [-0.857  2.914  3.232 -0.991]]
```

Whh:

```
[[ 4.043  1.406  3.481]
 [-0.478  5.421 -0.356]
 [ 3.681  2.175  0.213]]
```

Why:

```
[[ 0.698 -3.716  3.095]
 [ 9.298  1.457 -7.241]
 [ 6.294  7.404  7.079]
 [ 5.012 -9.129  3.419]]
```

bh:

```
[[ 6.872]
 [-2.439]
 [-0.985]]
```

by:

```
[[ -1.774]
 [ 3.537]
 [ 2.621]
 [ 2.336]]
```

Hình 2.10.3: Thông số của mạng RNN sau khi chạy thuật toán GA.

Kết quả thu được y =

```
[[2.94249421e-13 1.26320540e-11 4.23625623e-09 7.85283918e-07]
 [9.99991341e-01 7.95859427e-08 1.78454521e-06 1.75795812e-07]
 [8.65945428e-06 9.99999920e-01 9.99996193e-01 1.69329008e-05]
 [4.65860851e-12 3.54672541e-10 2.01782596e-06 9.99982106e-01]]
```

Kết quả sau khi làm tròn 3 chữ số y=

```
[[0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 1. 0.]
 [0. 0. 0. 1.]].
```

Loss= 3.0440205226291212e-05

Hình 2.10.4: Kết quả thu được từ mạng RNN

Kết quả thu được là tập hợp của 4 vector cột có giá trị lần lượt là [0 1 0 0] ứng với chữ "e", [0 0 1 0] ứng với chữ "l", [0 0 0 1] ứng với chữ "l" và [0 0 0 1] ứng với chữ "o". Kết quả rất chính xác và độ lớn hầm mốt rất nhỏ (Loss = $3 \cdot 10^{-5}$).



2.10.2 Nhận xét

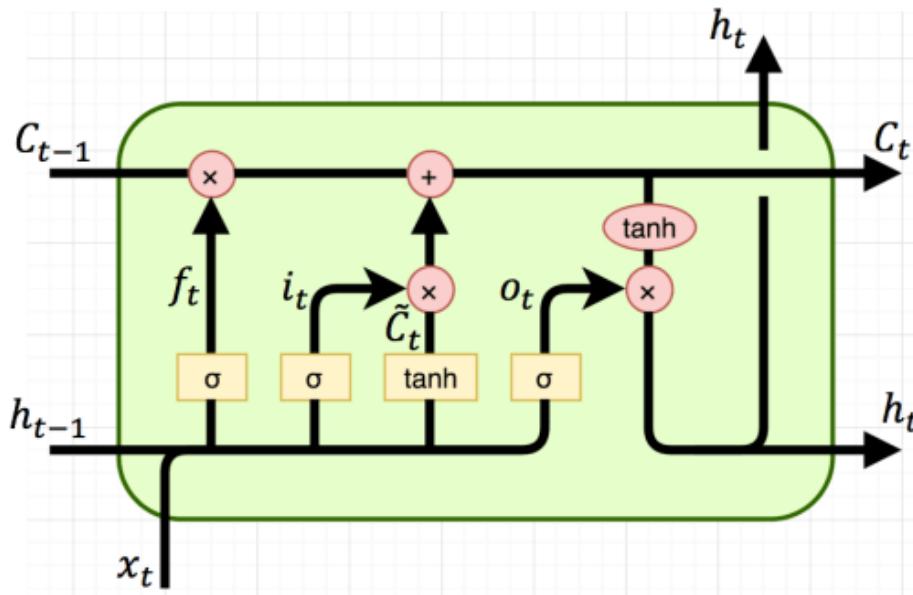
- Thuật toán GA cho ra kết quả rất nhanh và chính xác. Một phần vì mạng RNN của chúng ta cũng khá đơn giản.
- Các thông số của mạng RNN ở trên không phải là nghiệm duy nhất cho bài toán. Thực hiện thuật toán nhiều lần sẽ cho ra kết quả nghiệm khác nhau ở mỗi lần thử và độ chính xác khác nhau.
- Tuy nhiên nhiều khi thuật toán cũng không cho ra nghiệm tốt. Ta có thể cải thiện điều này bằng cách sửa đổi các thông số đầu vào của hàm GA trong thư viện Pygad.

2.11 Bài 11

- Máy tính hiện nay không có khả năng làm việc trực tiếp với các loại chữ cái, ký tự, văn bản mà chỉ thao tác dưới dạng số. Do đó khi thực hiện một bài toán thì điều chúng ta cần làm là mã hóa các dữ liệu thành những con số trước khi đưa vào máy tính xử lý. Trong bài toán trong bài 10, dữ liệu đầu vào là các chữ cái "h", "l", "o", "e". Những chữ cái này là các biến độc lập nên chúng ta cần một cách mã hóa sao cho các chữ cái này không có bất kì mối liên hệ hay tương quan nào với nhau. One-hot encoding là một loại mã phù hợp để thực hiện điều trên. Đây là loại mã dựa trên số loại (class) trong một đặc trưng để tạo ra một vector có số chiều bằng với số class, dữ liệu thuộc lớp nào thì vị trí của lớp đó trong vector sẽ được đánh dấu lên 1. Các vị trí trong vector này có vai trò ngang bằng và độc lập với nhau. Vì vậy không có bất kì sự tương quan nào giữa các dữ liệu khi ta dùng one-hot encoding.
- Nếu ta sử dụng hai cách mã hóa là mã hóa dữ liệu dưới dạng các số thập phân (1, 2, 3, 4, ...) hay mã hóa dưới dạng số nhị phân (00, 01, 10, 11, ...) thì các biến dữ liệu của chúng ta trong bài toán sẽ không độc lập với nhau. Vì như đã đề cập máy tính làm việc với những con số, khi mã hóa dưới dạng số thập phân nó sẽ ngầm hiểu rằng chúng tương quan với nhau. Ví dụ dữ liệu thuộc lớp 1 sẽ tương quan nhiều với dữ liệu ở lớp số 2 hơn là dữ liệu ở lớp số 4. Tương tự với mã hóa nhị phân, dữ liệu ở lớp 00 sẽ tương quan nhiều với dữ liệu ở lớp 01 hơn là dữ liệu ở lớp 11. Do đó hai cách mã hóa trên không phù hợp để áp dụng vào bài toán của chúng ta.

2.12 Bài 12

LSTM (Long short term memory) là một trong những kiến trúc mạng đặc biệt trong các loại mô hình RNN. Nó là một cải tiến của mạng RNN thông thường để vừa có thể thực hiện được chức năng "long term memory" và "short term memory".



Hình 2.12.1: Mô hình mạng LSTM.

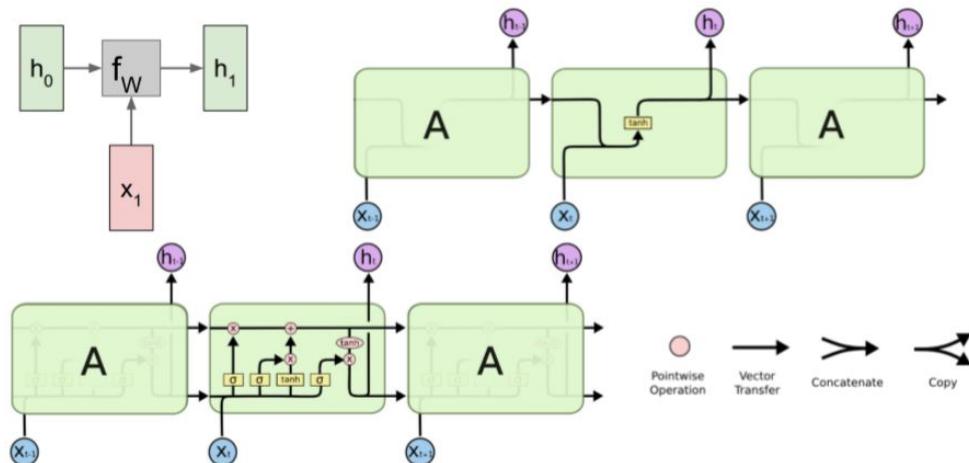
- Chức năng Short term memory: Trong LSTM, phần \tilde{C}_t khá giống với phần h_t trong RNN. Với công thức cập nhật là: $\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$. Ta thấy hàm kích hoạt của \tilde{C}_t là hàm tanh có giá trị trong khoảng $[-1,1]$ và đạo hàm trong khoảng $[0,1]$. Khi cập nhật trọng số, do tính chất của mạng nên đạo hàm của state sẽ theo nguyên tắc chain rule. Giả sử chẳng may đạo hàm ở những tầng xa có giá trị rất nhỏ (gần bằng 0) thì khi tính đạo hàm ở những state gần hơn ta nhân với các giá trị nhỏ này sẽ dẫn đến đạo hàm ở các state hiện tại cũng nhỏ theo. Đây là hiện tượng Vanishing Gradient. Các state càng xa state cuối thì khi xảy ra hiện tượng này hầu như thông số không được cập nhật đồng nghĩa với việc mạng của chúng ta sẽ không học được các thông tin ở những state đó. Vì vậy mặc dù mô hình của chúng ta theo cấu trúc có thể học được những thông tin ở xa, nhưng thực tế do Vanishing Gradient nó chỉ có thể học được những thông tin gần. Đó là lí do tại sao ta nói rằng phần này chính là phần short term memory trong mạng LSTM của chúng ta.
- Chức năng Long term memory: Một trong những cải tiến của mạng LSTM với RNN là nó có khả năng Long term memory. Vì so với RNN, LSTM có thêm luồng thông tin C_t đi từ state này sang state khác. Nó là tổng hợp của các thành phần f_t, i_t, c_t, o_t cho phép các thông tin được lưu trữ tùy thuộc vào mức độ quan trọng của nó.
- LSTM có thể đối phó với vấn đề phụ thuộc lâu dài vì trong cấu trúc mạng LSTM có khối forget gate. Khối này có điều chỉnh lượng thông tin đi qua dựa vào x_t và h_{t-1} . Những thông tin không cần thiết sẽ bị loại bỏ khi qua khối này trước khi đưa vào luồng thông tin C_t . Do đó có thể tránh được các thông tin sau bị phụ thuộc với các thông tin ở rất xa nó.

2.13 Bài 13

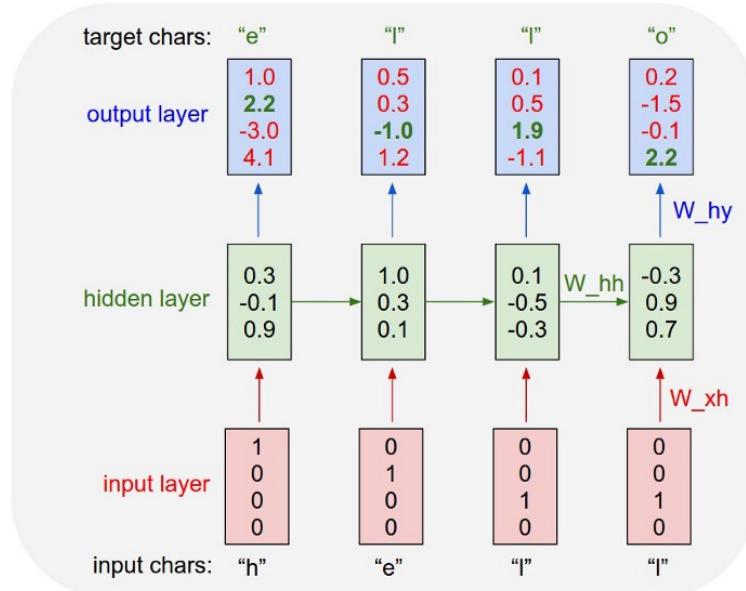
2.13.1 Sử dụng GA (Genetic Algorithm) để tối ưu hóa hàm mất mát của mạng LSTM

Thuật toán áp dụng là GA (đã được thực hiện ở bài 10), tuy nhiên model mạng sẽ là LSTM (Long Short Term Memory) và ta cũng huấn luyện học với từ "Hello". Mô hình học sẽ nhận ngõ vào ban đầu là kí tự "h" và cho ra các output lần lượt là các kí tự "e", "l", "l", "o".

Ta thực hiện tối ưu hóa hàm mất mát của mạng RNN bằng thuật toán GA khi huấn luyện model học với từ "hello". Mô hình học sẽ nhận ngõ vào ban đầu là kí tự "h" và cho các ngõ ra lần lượt là các kí tự "e", "l", "l", "o".



Hình 2.13.1: Mô hình mạng LSTM. Nguồn: <https://bitly.com.vn/1mq10q>



Hình 2.13.2: Model học với từ 'Hello'. Nguồn: <https://bitly.com.vn/h24ug5>

Ta tiến hành mã hóa các kí tự trong chuỗi '**Hello**' theo phương pháp one-hot vector. Với kí tự "H":[1 0 0 0], "e" : [0 1 0 0], "l" : [0 0 1 0], "o" : [0 0 0 1]. Sau đó định nghĩa một số hàm cần thiết cho việc tính toán.

```
import pygad
import numpy as np
import scipy
X= np.array([[1,0,0,0]]).T
Y = np.array([[0,1,0,0],[0,0,1,0],[0,0,1,0],[0,0,0,1]]).T
def softmax(x):
    return np.exp(x)/sum(np.exp(x))
def crossEntropy(X,Y):
    return -np.sum(np.log(X)*Y)
```

Tiếp theo sử dụng thư viện Pygad chạy thuật toán GA với mạng RNN.

```
def LSTM(solution, solution_idx,X,Y):
    Wf = solution[0:21].reshape(3,7)
    Wi = solution[21:42].reshape(3,7)
    Wc = solution[42:63].reshape(3,7)
    Wo = solution[63:84].reshape(3,7)
    bf = solution[84:87].reshape(3,1)
    bi = solution[87:90].reshape(3,1)
    bc = solution[90:93].reshape(3,1)
    bo = solution[93:96].reshape(3,1)
    Why = solution[96:108].reshape(4,3)
    by = solution[108:112].reshape(4,1)

    h_s = np.zeros((3,1))
    y_s = X
    C_s = np.zeros((3,1))

    for t in range(X.shape[0]):
        x_t = y_s[:, -1].reshape(4,1)
        z_t = np.vstack((x_t, h_s[:, -1].reshape(3,1))) ## x_t + h_(t-1)

        f_t = sigmoid(np.dot(Wf, z_t) + bf)
        i_t = sigmoid(np.dot(Wi, z_t) + bi)
        C_t1 = np.tanh(np.dot(Wc, z_t) + bi)
        C_t = f_t * C_s[:, -1].reshape(3,1) + i_t * C_t1
        o_t = sigmoid(np.dot(Wo, z_t) + bo)
        h_t = o_t * np.tanh(C_t)
        y_t = softmax(np.dot(Why, h_t) + by)

        h_s = np.hstack((h_s, h_t))
        y_s = np.hstack((y_s, y_t))
        C_s = np.hstack((C_s, C_t))

    h_s = h_s[:, 1:]
    y_s = y_s[:, 1:]
    return h_s, y_s, crossEntropy(y_s, Y)

fitness_function = LSTM_GA
num_generations = 100
num_parents_mating = 10
sol_per_pop = 20
num_genes = 116
```

```
init_range_low = -2
init_range_high = 5
parent_selection_type = "sss"
keep_parents = 1
crossover_type ="single_point"
mutation_type = "random"
mutation_percent_genes = 20

ga_instance = pygad.GA(
    num_generations=num_generations,
    num_parents_mating=num_parents_mating,
    fitness_func=fitness_function,
    sol_per_pop=sol_per_pop,
    num_genes=num_genes,
    init_range_low=init_range_low,
    init_range_high=init_range_high,
    parent_selection_type=parent_selection_type,
    keep_parents=keep_parents,
    crossover_type=crossover_type,
    mutation_type=mutation_type,
    mutation_percent_genes=mutation_percent_genes)

ga_instance.run()
solution, solution_fitness, solution_idx = ga_instance.best_solution()
```

Kết quả sau khi chạy đoạn code trên được các thông số của mạng LSTM.

Wf:

```
[[ -3.231 -6.098  1.397  1.67   0.536 -5.929 -2.829]
 [ 3.944  3.435 -0.399  1.947 -0.251  5.454  1.956]
 [ 1.074  0.791  3.521  4.603  6.056  1.949  4.359]]
```

Wi:

```
[[ 3.687  0.746  1.894 -1.827 -0.665  0.873  6.061]
 [-0.965  6.552 -1.464  2.026  1.656  2.542  0.807]
 [-1.302  2.947 -1.313  0.12   3.517 -1.76   -2.466]]
```

Wc:

```
[[ -3.32    7.1     3.226   1.212  -3.257  -1.68    3.537]
 [-3.198   6.489  -1.213   5.732   2.398   1.302   0.873]
 [-2.041  -0.708   7.896  -5.708   5.177  -1.351   8.03  ]]
```

Wo:

```
[[ 3.497  2.813  2.886  4.704 -0.717 -2.886  1.585]
 [ 2.838 -0.613  4.181 -1.331  8.143 -2.068  2.395]
 [ 1.488 -2.089  1.068  1.02   -0.46   6.011  3.402]]
```

Why:

```
[[ -3.014   5.098  -1.262]
 [-10.003  -4.149   1.157]
 [ 8.435  -2.521  -0.446]
 [-4.109   9.484   7.392]]
```

Hình 2.13.3: Thông số của mạng LSTM sau khi chạy thuật toán GA.

```
↳ bf:  
  [[-0.36 ]  
   [ 3.879]  
   [ 7.573]]  
bi:  
  [[-0.028]  
   [-0.015]  
   [ 2.714]]  
bc:  
  [[2.868]  
   [1.87 ]  
   [2.517]]  
bo:  
  [[ 3.108]  
   [-3.82 ]  
   [-2.78 ]]  
by:  
  [[-1.322]  
   [ 5.4 ]  
   [ 8.9 ]  
   [ 0.04 ]]
```

Hình 2.13.4: Thông số của mạng LSTM sau khi chạy thuật toán GA.

Áp dụng các thông số vừa tìm được ở trên vào mô hình mạng LSTM. Sử dụng hàm crossentropy để đánh giá kết quả thu được.

```
# LSTM function  
def LSTM(solution, solution_idx,X,Y):  
    Wf = solution[0:21].reshape(3,7)  
    Wi = solution[21:42].reshape(3,7)  
    Wc = solution[42:63].reshape(3,7)  
    Wo = solution[63:84].reshape(3,7)  
    bf = solution[84:87].reshape(3,1)  
    bi = solution[87:90].reshape(3,1)  
    bc = solution[90:93].reshape(3,1)  
    bo = solution[93:96].reshape(3,1)  
    Why = solution[96:108].reshape(4,3)  
    by = solution[108:112].reshape(4,1)  
  
    h_s = np.zeros((3,1))  
    y_s = X  
    C_s = np.zeros((3,1))  
  
    for t in range(X.shape[0]):  
        x_t = y_s[:, -1].reshape(4,1)
```



```
z_t = np.vstack((x_t,h_s[:, -1].reshape(3,1))) ## x_t + h_(t-1)

f_t = sigmoid(np.dot(Wf,z_t)+bf)
i_t = sigmoid(np.dot(Wi,z_t)+bi)
C_t1 = np.tanh(np.dot(Wc,z_t)+bi) # C mu t
C_t = f_t*C_s[:, -1].reshape(3,1) + i_t*C_t1
o_t = sigmoid(np.dot(Wo,z_t)+bo)
h_t = o_t*np.tanh(C_t)
y_t = softmax(np.dot(Why, h_t) + by)

h_s = np.hstack((h_s,h_t))
y_s = np.hstack((y_s,y_t))
C_s = np.hstack((C_s,C_t))

h_s = h_s[:,1:]
y_s = y_s[:,1:]

return h_s,y_s,crossEntropy(y_s,Y)

h,y,Loss = LSTM(solution,solution,X,Y)
print("Kết quả thu được y = \n {} \n Kết quả sau khi làm tròn 3 chu so y= \n {}.\n Loss= {}".format(y,np.round(y,decimals=3),Loss))
```

```
Kết quả thu được y =
[[3.92802140e-05 4.46306774e-09 7.09707238e-08 6.55187401e-06]
 [9.99856985e-01 2.18921955e-05 1.13958137e-06 1.56525284e-05]
 [7.09024364e-05 9.99976796e-01 9.99994536e-01 1.46341881e-05]
 [3.28318884e-05 1.30729093e-06 4.25337319e-06 9.99963161e-01]]
Kết quả sau khi làm tròn 3 chu so y=
[[0. 0. 0. 0.]
 [1. 0. 0. 0.]
 [0. 1. 1. 0.]
 [0. 0. 0. 1.]].
Loss= 0.00020853219432735644
```

Hình 2.13.5: Kết quả thu được từ mạng RNN

Kết quả thu được là tập hợp của 4 vector cột có giá trị lần lượt là [0 1 0 0] ứng với chữ "e", [0 0 1 0] ứng với chữ "l", [0 0 0 1] ứng với chữ "l" và [0 0 0 1] ứng với chữ "o". Kết quả rất chính xác và độ lớn hàm mất mát cũng rất nhỏ (Loss= 0.0002).

2.13.2 Nhận xét

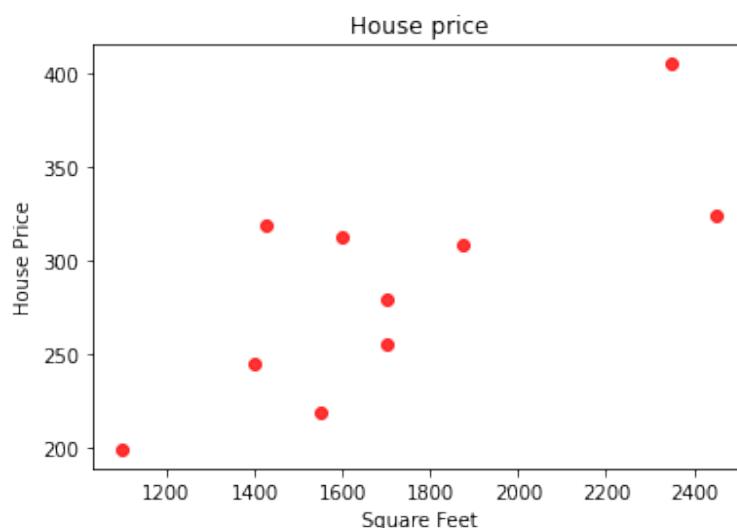
- Thuật toán GA cho ra kết quả rất nhanh và chính xác.
- Các thông số của mạng RNN ở trên không phải là nghiệm duy nhất cho bài toán. Thực hiện thuật toán nhiều lần sẽ cho ra kết quả nghiệm khác nhau ở mỗi lần thử và độ chính xác khác nhau.
- Tuy nhiên nhiều khi thuật toán cũng không cho ra nghiệm tốt. Ta có thể cải thiện điều này bằng cách sửa đổi các thông số đầu vào của hàm GA trong thư viện Pygad.

2.14 Bài 14

Tập dữ liệu cho bài 14 là tập dữ liệu dự đoán giá nhà dựa trên diện tích của căn nhà. Tập dữ liệu gồm có 10 mẫu bao gồm 2 cột, cột House_Price là giá trị về giá ngôi nhà và cột X_Square là cột về diện tích ngôi nhà. Nhóm sẽ thực hiện chuẩn hóa dữ liệu về kỳ vọng 0 và phương sai 1. Sau đó, thực hiện khảo sát tập dữ liệu này với thuật toán Linear Regression.

	House_Price	X_Square
0	245	1400
1	312	1600
2	279	1700
3	308	1875
4	199	1100
5	219	1550
6	405	2350
7	324	2450
8	319	1425
9	255	1700

Hình 2.14.1: Tập dữ liệu cho bài 14



Hình 2.14.2: Đồ thị biểu diễn của dữ liệu

2.14.1 Chuẩn hóa dữ liệu

Cho một tập dữ liệu gồm N phần tử \mathbf{x}_i . Công thức chuẩn hóa dữ liệu về dạng kỳ vọng 0 và phương sai 1 là:

$$\mathbf{x}' = \frac{\mathbf{x} - \mu}{\sigma} \quad (2.18)$$



với

$$\mu = \frac{\sum_{i=1}^N \mathbf{x}_i}{N} \quad (2.19)$$

$$\sigma^2 = \frac{\sum_{i=1}^N (\mathbf{x}_i - \mu)^2}{N} \quad (2.20)$$

Đoạn chương trình thực hiện chuẩn hóa dữ liệu:

```
def mean(x):
    x = x.reshape(-1, 1)
    return x.sum() / x.shape[0]

def variance(x):
    x = x.reshape(-1, 1)
    m = mean(x)
    return np.sqrt(mean((x - m) ** 2))

def normalize(x):
    m = mean(x)
    s = variance(x)
    return (x - m) / s, m, s
```

2.14.2 Thực hiện bài toán Linear Regression

2.14.2.1 Thực hiện bài toán sử dụng Gradient Descent với dữ liệu chưa được chuẩn hóa:

Đoạn chương trình Gradient Descent cho bài toán Linear Regression. Hàm **cost** dùng để tính giá trị hàm mất mát và hàm **grad** để tính giá trị đạo hàm của hàm mất mát theo **w** dùng cho quá trình cập nhật trọng số. Hàm GD dùng để tính kết quả và trả về giá trị trọng số **w**, số lần lặp **it** và giá trị hàm mất mát **cost**. Thuật toán sẽ dừng lại khi giá trị hàm mất mát nhỏ hơn hoặc bằng 0.001 hoặc vượt quá số lần lặp cho phép **loop**.

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2N} \|\bar{\mathbf{X}}\mathbf{w} - \mathbf{y}\|_2^2$$

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial \mathbf{w}} = \frac{1}{N} \bar{\mathbf{X}}^T (\bar{\mathbf{X}}\mathbf{w} - \mathbf{y})$$

```
def grad(w):
    N = X_bar.shape[0]
    return 1/N * X_bar.T.dot(X_bar.dot(w) - y)

def cost(w):
    N = X_bar.shape[0]
    return (1/(2 * N)) * np.linalg.norm(X_bar.dot(w) - y, 2) ** 2
```



```
def GD(w_init, grad, eta, loop):
    w = [w_init]

    for it in range(loop):
        # update weight
        w_new = w[-1] - eta * grad(w[-1])

        if cost(w_new) < 0.001:
            break
        w.append(w_new)
    return w[-1], it, cost(w[-1])
```

Tiếp theo, chúng ta tiến hành thuật toán và thu được kết quả như sau, chúng ta sẽ cài đặt thông số $w = [1, 1]$, hệ số học là 0.01 và số lần lặp tối đa là 2000.

```
# Add one
one = np.ones((X.shape[0], 1))
X_bar = w_new = np.concatenate((one, X), axis=1)

w_init = np.array([1, 1])
eta = 0.001
loop = 2000
(w, it, cost) = GD(w_init, grad, eta, loop)
w_0 = w[0]
w_1 = w[1]
print(w)
print('Loss:', cost)

[Out 1]: [nan nan]
[Out 2]: Loss: nan
```

Kết quả thu được không như mong muốn. Nhóm đã tiến hành thay đổi learning rate để kiểm tra, với các giá trị learning rate bằng 0.0001 và 0.00001 kết quả đều thu được là nan.

Tiếp theo, chúng ta sẽ thử chuẩn hóa dữ liệu rồi sử dụng thuật toán Gradient Descent để thử nghiệm.

2.14.2.2 Thực hiện bài toán sử dụng Gradient Descent với dữ liệu chưa được chuẩn hóa:

Trước tiên ta sẽ thực hiện dữ liệu, sau đó tiếp hành thêm đặc trưng 1 để tạo X_{bar} và sử dụng thuật toán GD như bài trên. Do đã sử dụng chuẩn hóa, nên sau khi thu được kết quả ta phải sử dụng giá trị mean và variace để biến đổi kết quả về giống dạng trước khi chuẩn hóa. Chúng ta cũng tiến hành đặt thông số $w = [1, 1]$, hệ số học là 0.01 và số lần lặp tối đa là 2000.

```
one = np.ones((X.shape[0], 1))
X_nor, mean, var = normalize(X)
X_bar_nor = w_new = np.concatenate((one, X_nor), axis=1)

w_init = np.array([1, 1])
eta = 0.001
loop = 200000
(w, it, cost) = GD(w_init, grad, eta, loop)
```

```
w_0 = w[0] - w[1]*mean/var
w_1 = w[1]/var
w = [w_0, w_1]
print(w)
print('Loss:', cost(w))

[Out 1]: [98.24832962136779, 0.10976773783008971]
[Out 2]: Loss: 683.2782612153995
```

Sau khi sử dụng chuẩn hóa dữ liệu, ta đã thu được kết quả là $\text{House_Price} = 98.2483 + 0.1098 * \text{X_Square}$. Tuy nhiên, giá trị hàm mất mát khá cao 683.2782. Ta sẽ tiến hành vẽ hình và khảo sát thêm kết quả giải bằng lời giải giải tích để kiểm tra thêm. Kết quả sau vẽ hình cũng khá tốt, đường thẳng cũng thể hiện được xu hướng của tập dữ liệu.



Hình 2.14.3: Kết quả thực hiện bằng GD sử dụng chuẩn hóa

2.14.2.3 Thực hiện kết quả bằng lời giải giải tích

Ta sẽ tiến hành thực hiện bài toán bằng lời giải giải tích, kết quả giá trị thông số có công thức

$$\mathbf{w} = (\bar{\mathbf{X}}^T \bar{\mathbf{X}})^{\dagger} \bar{\mathbf{X}}^T \mathbf{y}$$

Doạn chương trình thực hiện như sau:

```
def LinearRegression_equa(X_bar, y):
    A = np.dot(X_bar.T, X_bar)
    B = np.dot(X_bar.T, y)
    w = np.dot(np.linalg.pinv(A), B)
    return w
```

Ta sẽ tiến hành thực hiện trên tập dữ liệu bình thường (không chuẩn hóa) để kiểm tra kết quả.

```
one = np.ones((X.shape[0], 1))
X_bar = w_new = np.concatenate((one, X), axis=1)
w = LinearRegression_equa(X_bar, y)
print(w)
w_0 = w[0]
w_1 = w[1]
print(cost(w))
```

```
[Out 1]: [98.24832962 0.10976774]
[Out 2]: 683.2782612153992
```



Hình 2.14.4: Kết quả thực hiện bằng lời giải giải tích

Kết quả thực hiện bằng GD với chuẩn hóa giống so với kết quả thu được bằng lời giải giải tích với kết quả là **House_Price** = $98.2483 + 0.1098 * X_{Square}$. Tuy nhiên, giá trị của hàm mất mát rất cao khoảng 683.2783. Đây rất có thể là nguyên nhân khiến dữ liệu khi chưa được chuẩn hóa hoạt động không được. Chúng ta tiến hành xuất kết quả hàm mất mát qua từng it, kết quả như sau:

```
10397376723146.646
9.974970578115715e+19
9.569725200165242e+26
9.180943411261384e+33
8.807956357966097e+40
8.450122359830182e+47
8.106825805457387e+54
7.777476093417249e+61
7.461506616184658e+68
7.158373785358108e+75
6.867556096479648e+82
6.588553231847673e+89
6.32088519977902e+96
```



6.06409150883964e+103

...
inf
inf
inf
inf
inf
inf
inf
...
nan
nan
nan
nan
nan
nan
nan
...

Ta có thể thấy, giá trị của hàm măt mát khởi tạo rất cao và tăng qua từng ít, nó làm ảnh hưởng rất lớn đến kết quả bài toán. Nguyên nhân có thể là do dữ không tương đồng về khoảng giá trị của **X** và **y** khi giá trị của **X** lớn gấp nhiều lần so với giá trị của **y**. Khi ta tiến hành chuẩn hóa dữ liệu thì đã làm thu hẹp khoảng chênh lệch này nên đã thu được kết quả tốt.

Tài liệu tham khảo

- [1] Rong Ge et al. “Escaping From Saddle Points - Online Stochastic Gradient for Tensor Decomposition”. In: *CoRR* abs/1503.02101 (2015). arXiv: [1503 . 02101](https://arxiv.org/abs/1503.02101). URL: <http://arxiv.org/abs/1503.02101>.
- [2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.
- [3] Ahmed Fawzy Gad. *PyGAD - Python Genetic Algorithm*. Truy cập 1/11/2021. 2020. URL: <https://pygad.readthedocs.io/en/latest/>.