BỘ GIÁO DỤC VÀ ĐÀO TẠO

**TRƯỜNG ĐẠI HỌC THĂNG LONG**



**BÀI TẬP LỚN**

**HỌC MÁY**

**LỚP HỌC PHẦN: 243\_MA239\_04**

**GIẢNG VIÊN: NGUYỄN QUANG HUY**

**SINH VIÊN THỰC HIỆN:NGUYỄN ANH ĐỨC A48274**

**HÀ NỘI – 2025**

**MỤC LỤC**

[Chương 1. Tìm hiểu chung về mạng nơ-ron nhân tạo 2](#_Toc29059720)

[1.1. Định nghĩa 3](#_Toc1583993644)

[1.2. Cấu trúc 3](#_Toc542995083)

[1.2.1 Lớp đầu vào 3](#_Toc1003194010)

[1.2.2 Lớp ẩn 3](#_Toc126382997)

[1.2.3 Lớp đầu ra 3](#_Toc1777147316)

[1.3. Khái niệm cơ bản 4](#_Toc993199945)

[1.3.1 Trọng số (weight) 4](#_Toc2061483088)

[1.3.2 Hệ số điều chỉnh (bias) 4](#_Toc275490406)

[1.3.3 Forward pass 4](#_Toc1424726975)

[1.3.4 Backward pass 5](#_Toc968767747)

[1.4. Dataset 5](#_Toc178961330)

[Chương 2. Xây dựng mạng nơ-ron nhân tạo 6](#_Toc2055506704)

[2.1. Tổng quan 7](#_Toc1286473067)

[2.2. Lớp dense (Dense layer) 8](#_Toc457923654)

[2.3. Hàm kích hoạt (Activation function) 9](#_Toc1601135859)

[2.3.1 ReLu 10](#_Toc904755957)

[2.3.2 Softmax 11](#_Toc1715490389)

[2.4. Hàm mất mát (Loss function) 13](#_Toc2107442256)

[2.5. Optimizers: Combined backpropagation on softmax activation and cross entropy loss 16](#_Toc98085649)

[2.6. ADAM optimizer 17](#_Toc1391606106)

[2.6.1 Momentum 18](#_Toc2044680007)

[2.6.2 RMSProp 19](#_Toc1195319828)

[2.7. Train mô hình 21](#_Toc965426962)

[2.8. Testing & validation 24](#_Toc742176504)

[2.8.1 Hướng giải quyết 24](#_Toc632430170)

[2.8.2 Đề xuất các giải pháp nâng cao 24](#_Toc1114611993)

[2.8.3 Phương pháp 24](#_Toc442156640)

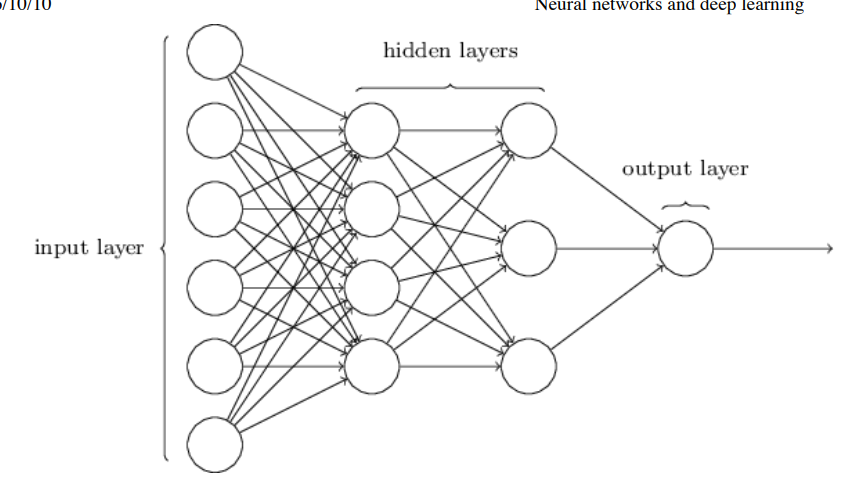
# Tìm hiểu chung về mạng nơ-ron nhân tạo

## Định nghĩa

Mạng nơ-ron là một phương thức trong lĩnh vực trí tuệ nhân tạo (AI), được sử dụng để dạy máy tính xử lý dữ liệu theo cách mô phỏng bộ não con người. Đây là một loại quy trình máy học, được gọi là học sâu, sử dụng các nút hoặc nơ-ron liên kết với nhau trong một cấu trúc phân lớp tương tự như bộ não con người. Phương thức này tạo ra một hệ thống thích ứng được máy tính sử dụng để học hỏi từ sai lầm của chúng và liên tục cải thiện.

## Cấu trúc

Một mạng nơ-ron cơ bản bao gồm các nơ-ron nhân tạo liên kết theo 3 lớp:



### Lớp đầu vào

Thông tin từ bên ngoài đi vào mạng nơ-ron nhân tạo qua lớp đầu vào. Các nút đầu vào xử lý dữ liệu, phân tích hoặc phân loại và sau đó chuyển dữ liệu sang lớp tiếp theo.

### Lớp ẩn

Dữ liệu đi vào lớp ẩn đến từ lớp đầu vào hoặc các lớp ẩn khác. Mạng nơ-ron nhân tạo có thể có một số lượng lớn lớp ẩn. Mỗi lớp ẩn phân tích dữ liệu đầu ra từ lớp trước, xử lý dữ liệu đó sâu hơn và rồi chuyển dữ liệu sang lớp tiếp theo.

### Lớp đầu ra

Lớp đầu ra cho ra kết quả cuối cùng của tất cả dữ liệu được xử lý bởi mạng nơ-ron nhân tạo. Lớp này có thể có một hoặc nhiều nút. Ví dụ: giả sử chúng ta gặp phải một vấn đề phân loại nhị phân (có/không), lớp đầu ra sẽ có một nút đầu ra, nút này sẽ cho kết quả 1 hoặc 0. Tuy nhiên, nếu chúng ta gặp phải vấn đề phân loại nhiều lớp, lớp đầu ra sẽ có thể bao gồm nhiều hơn một nút đầu ra.

## Khái niệm cơ bản

### Trọng số (weight)

Thông tin đi vào các nơ-ron đều được nhân trọng số. Ví dụ, nếu một nơ-ron có hai đầu vào, thì mỗi đầu vào sẽ có một trọng số riêng biệt được gán cho chúng. Chúng ta khởi tạo các trọng số trong mạng một cách ngẫu nhiên và các trọng số này được cập nhật trong quá trình huấn luyện mô hình. Mạng nơ-ron nhân tạo sau khi được huấn luyện sẽ gán trọng số cao hơn cho đầu vào mà nó coi là quan trọng. Nếu trọng số bằng 0, điều đó có nghĩa là đặc trưng tương ứng không phải là thông tin quan trọng.

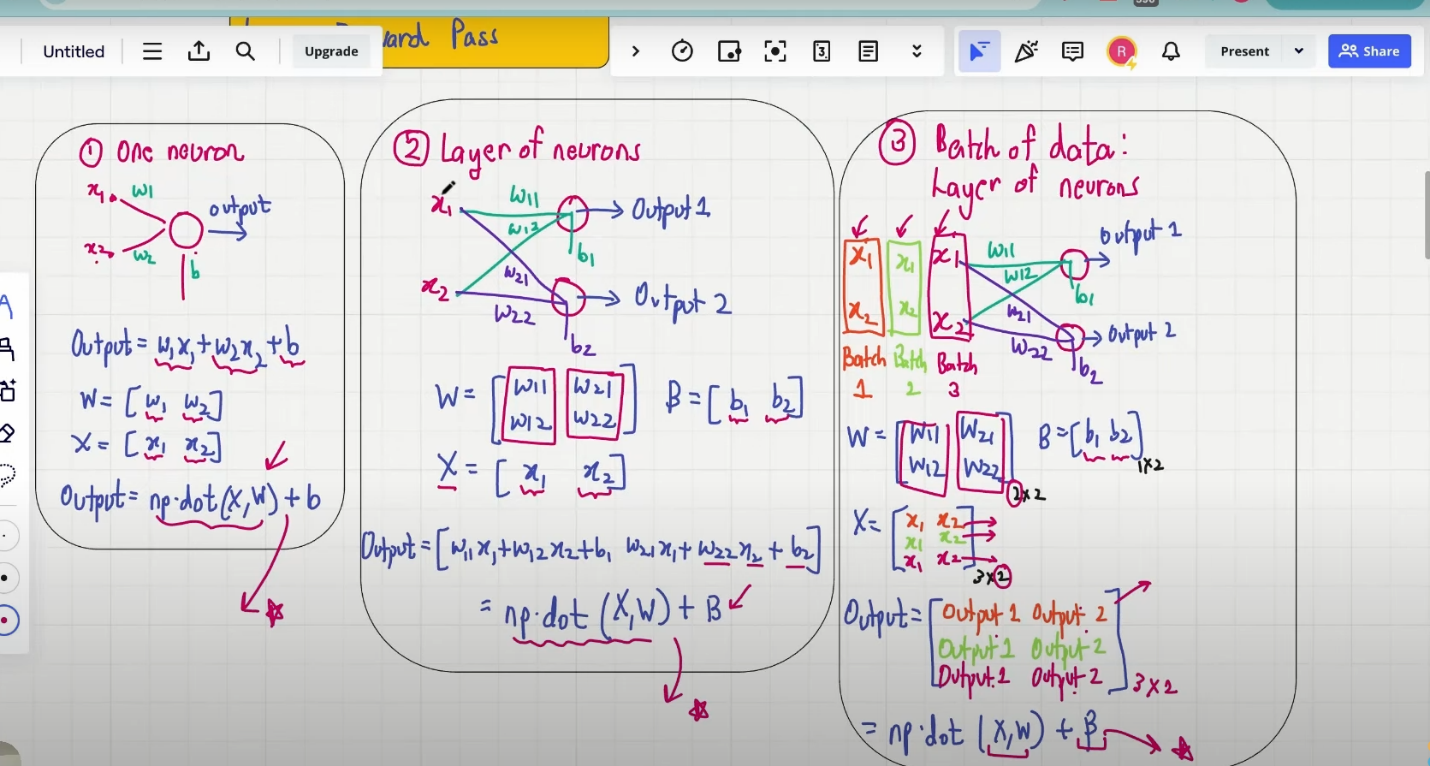
=> xác định mức độ ảnh hưởng của mỗi đầu vào đến từng neuron.

### Hệ số điều chỉnh (bias)

Được thêm vào kết quả của phép nhân trọng số với đầu vào. Kết quả của phép nhân luôn là một hàm tuyến tính đi qua gốc tọa độ, sử dụng bias giúp mạng nơ ron có thể dịch chuyển hàm tuyến tính này một cách linh hoạt. Bias cũng là một tham số được học trong quá trình huấn luyện mạng nơ-ron.

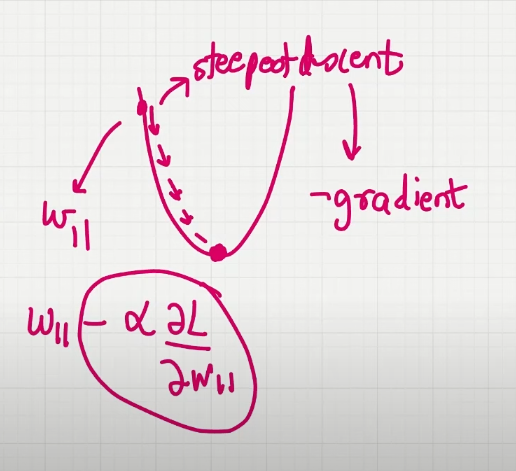
Cách khởi tạo được đề xuất là chọn cả trọng số và bias bằng cách sử dụng các biến ngẫu nhiên Gaussian độc lập, được chuẩn hóa để có trung bình là 0 và độ lệch chuẩn là 1 (cũng là cách mà trong bài này sử dụng (xác suất thống kê)).

### Forward pass



Forward pass là giai đoạn tính toán đầu ra (output) của một mạng neural network, dựa trên dữ liệu đầu vào (input) hiện tại. Đây là bước đầu tiên trong quá trình huấn luyện hoặc dự đoán, tức là mạng nơ-ron trong đó đầu ra từ một lớp được sử dụng làm đầu vào cho lớp kế tiếp. Những mạng như vậy được gọi là mạng nơ-ron lan truyền xuôi (feedforward neural networks). Điều này có nghĩa là chúng không có vòng lặp (loops)

### Backward pass



Khi mà thực hiện forward pass xong cần phải cho máy học tức thay đổi giá trị tốt hơn cho cách lần lặp tiếp theo để giảm thiểu loss, tức là phải tìm điểm cực tiểu của loss function. Khi bắt đầu sẽ ở điểm ngẫu nhiên trong loss function, ta cần phải tiếp tục di chuyển theo hướng để đạt được giá trị cực tiểu. Điểm xuất phát này gọi là step size descent còn đường đi xuống dốc được xác định bởi giá trị âm của gradient hay là đạo hàm riêng của loss function đối với tham số.

VD: bắt đầu với lần tiếp theo sẽ là .

Có thể thực hiện giảm độ dốc cho mò cực tiểu bằng các pp như gradient desenct,...

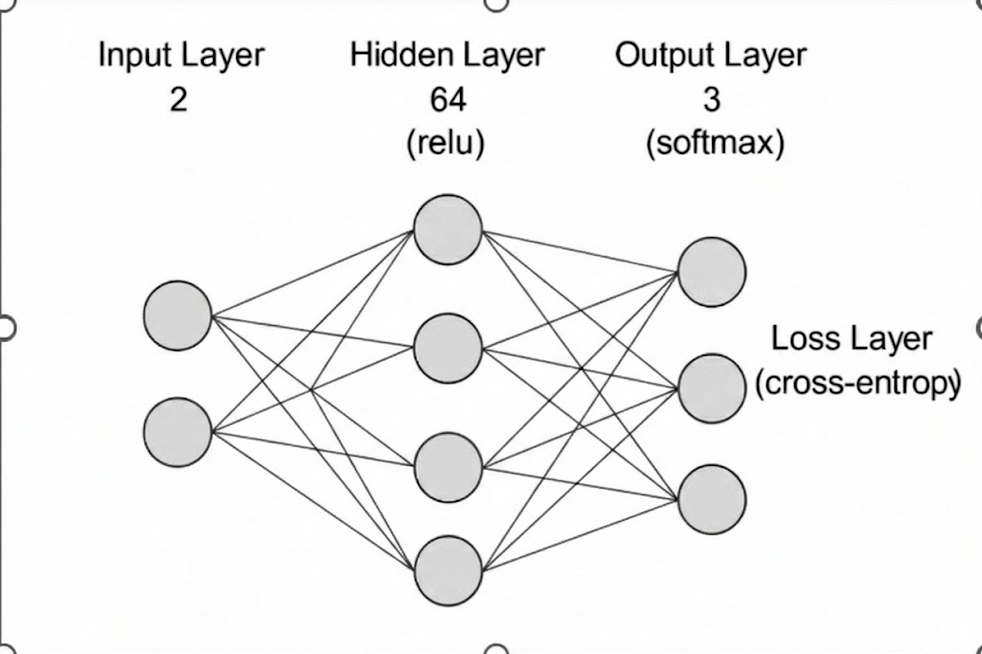
Backward pass là 1 method để tìm đạo hàm riêng của loss đối với các weight, bias, input layer

## Dataset

Spiral Dataset (Dữ liệu hình xoắn ốc) là một tập dữ liệu tổng hợp thường được sử dụng trong lĩnh vực học máy, đặc biệt để đánh giá khả năng phân loại phi tuyến của các mô hình. Dữ liệu gồm các điểm phân bố theo dạng xoắn ốc và được chia thành nhiều lớp, mỗi lớp là một nhánh xoắn khác nhau. Với cấu trúc phức tạp và đan xen, spiral dataset là một ví dụ điển hình giúp kiểm tra độ mạnh của các mô hình học sâu (deep learning) so với các mô hình tuyến tính truyền thống. Trong bài này, dữ liệu được tạo với 3 lớp và 100 điểm mẫu, sau đó trực quan hóa bằng biểu đồ phân tán màu.

# Xây dựng mạng nơ-ron nhân tạo

## Tổng quan



Quá trình xây dựng mạng nơ-ron nhân tạo gồm các bước sau:

* Xây dựng lớp dense (forward + backward);
* Xây dựng hàm kích hoạt (Activation function);
* ReLu: neuron move through layer (forward + backward)
* Softmax: output last neuron
* Xây dựng hàm mất mát: phân loại thành 3 lớp, kết hợp Softmax activation và and Cross-Entropy loss;
* Xây dựng optimizer: ADAM (lr rate, rate decay, momentum, RMSProp, sgd);
* Test & validation với test data
* K-fold cross validation
* L1/L2 Regularization (Refactor layer & loss )
* Dropout layers

**Dataset:** model: X, y = spiral\_data(samples=100, classes=3)

Forward pass

Input (2)

→ Dense Layer (foward) (2 -> 64)

→ ReLU (forward)

→ Dense Layer (foward) (64 -> 3)

→ Softmax

→ Loss Function (Categorical Cross-Entropy)

Backward Pass

Loss Gradient

→ Softmax + Cross-Entropy Backward

→ Dense Layer (backward) (64 -> 3)

→ ReLU (backward)

→ Dense Layer (backward) (64 -> 2)

→ Adam Optimizer (update weights + biases) (Momentum + RMSProp)

→ Update Parameters (dense 1 and dense 2)

( Trong code python có comment do vậy bài báo cáo thiên về lý thuyết và công thức toán học)

## Lớp dense (Dense layer)

Mỗi lớp dense đại diện cho 1 lớp trong mạng nơ-ron, mỗi nơ-ron tầng này đều phải liên kết với các nơ-ron lớp khác (qua forward).

class Layer\_Dense:

    # Layer initialization

    def \_\_init\_\_(self, n\_inputs, n\_neurons):

        # Initialize weights and biases

        self.weights = 0.01 \* np.random.randn(n\_inputs, n\_neurons)

        self.biases = np.zeros((1, n\_neurons))

Ở đây, weight được tạo ra từ một ma trận ngẫu nhiên từ phân phối chuẩn kích thước tương ứng với số đầu vào và số neuron, nhân với 0.01 để làm cho các giá trị này nhỏ, tránh overfitting, bisas là 1 vector toàn số 0 với chiều dài bằng neuron, giá trị này sẽ được học và cập nhật dần trong quá trình huấn luyện.

    # Forward pass

    def forward(self, inputs):

        # Remember input values

        self.inputs = inputs

        # Calculate output values from input ones, weights and biases

        self.output = np.dot(inputs, self.weights) + self.biases

Tương ứng công thức toán học:

Truyền từ input layer sang hidden layer hoặc layer 1 sang layer 2

Thực hiện nhân ma trận để có thể chuyển tiếp

    # Backward pass

    def backward(self, dvalues):

        # Gradients on parameters

        self.dweights = np.dot(self.inputs.T, dvalues)

        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)

        # Gradient on values

        self.dinputs = np.dot(dvalues, self.weights.T)

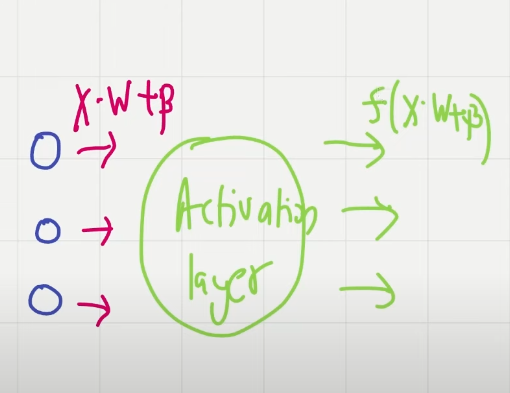
Tương ứng công thức toán học:

* L: hàm mất mát (loss function)
* X: đầu vào của layer (ở đây là self.inputs)
* W: ma trận trọng số (self.weights)
* Z: forward

Tính đạo hàm riêng để tìm gradient của hàm mất mát đối với các tham số (trọng số và bias), từ đó cập nhật các tham số để cải thiện độ chính xác (layer & layer).

## Hàm kích hoạt (Activation function)

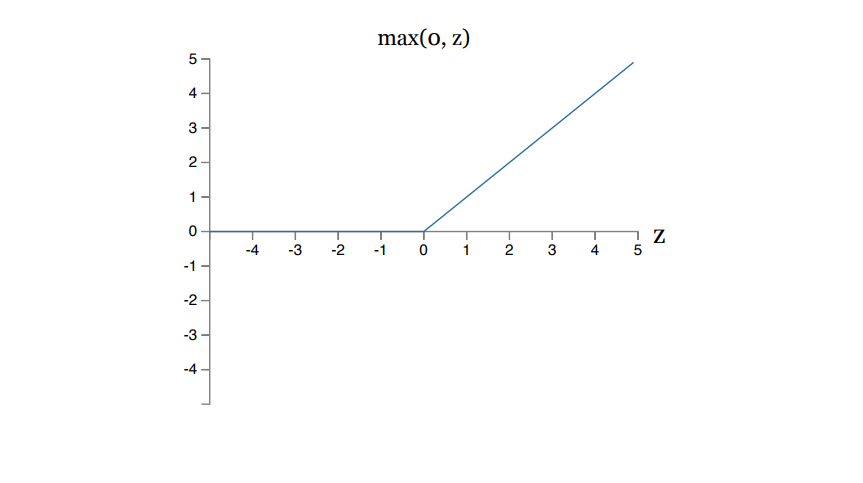
Activation function là hàm số được áp dụng lên đầu ra của mỗi neuron, giúp quyết định xem neuron đó có nên được kích hoạt hay không, giúp mô hình học.



Khi mà thực hiện nhân ma trận như forward như trên dense layer thì ta chỉ thu lại về 1 hàm tuyến tính (tức đồ thị bậc 1 ) (hay còn là hàm đi qua gốc tọa độ) do vậy mạng neuron cần học thay vì đưa trực tiếp hàm tuyến tính trả về từ dense layer vào layer tiếp theo, ta cần truyền qua activation functiontức vị trí của nằm ở giữa forward và layer kế tiếp. đi qua activation functionnhả về hàm phi tuyến tính. Ở đây sử dụng 2 thuật toán Gradient Descent là ReLu và Softmax.

### ReLu

**ReLu** là thuật toán giúp tìm cực tiểu sau mỗi lần chạy

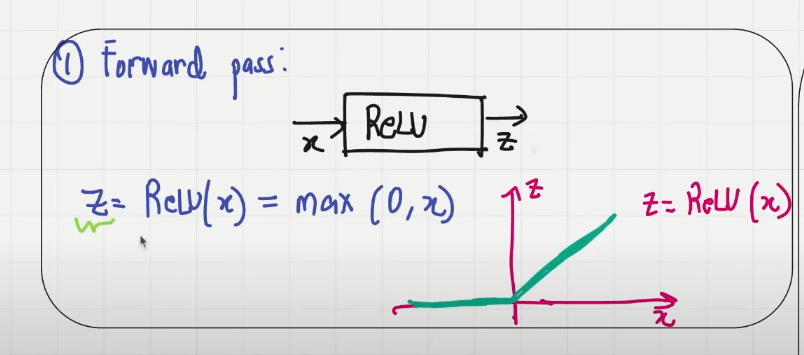


Tạo tính phi tuyến giúp mô hình học, giúp giảm hiện tượng vanishing gradient và tăng tốc độ hội tụ khi dùng Gradient Descent (Sẽ liên tục so sánh với 0 để đảm bảo cho giá trị luôn > 0).

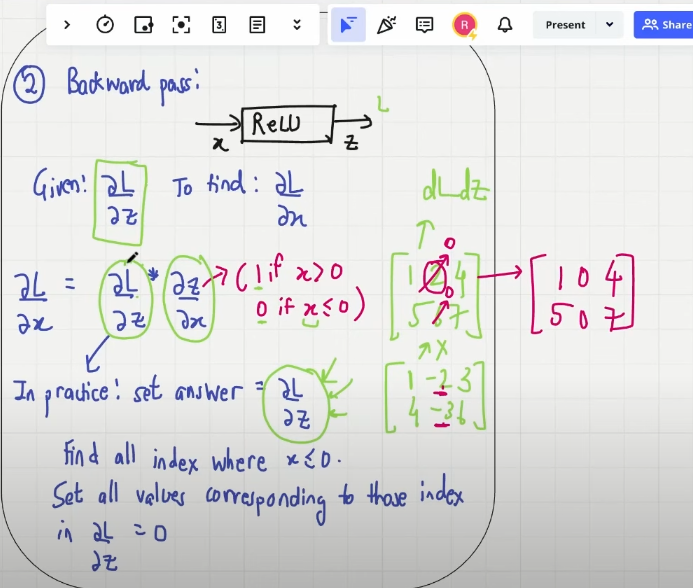
Tưởng tượng, với ReLu này sẽ hiện ra một hình parabol bề lõm quay xuống dưới với mỗi lần for chạy qua thì đều càng gần tới cực tiểu

=> Được sử dụng trong các lớp ẩn (hidden layers).

* ReLu Forward pass:



* ReLu Backward pass:



# ReLU activation

class Activation\_ReLU:

    # Forward pass

    def forward(self, inputs):

        # Remember input values

        self.inputs = inputs

        # Calculate output values from inputs

        self.output = np.maximum(0, inputs)

    # Backward pass

    def backward(self, dvalues):

        # Since we need to modify the original variable,

        # let’s make a copy of values first

        self.dinputs = dvalues.copy()

        # Zero gradient where input values were negative

        self.dinputs[self.inputs <= 0] = 0

* Công thức toán học forward:
* Công thức toán học backward:

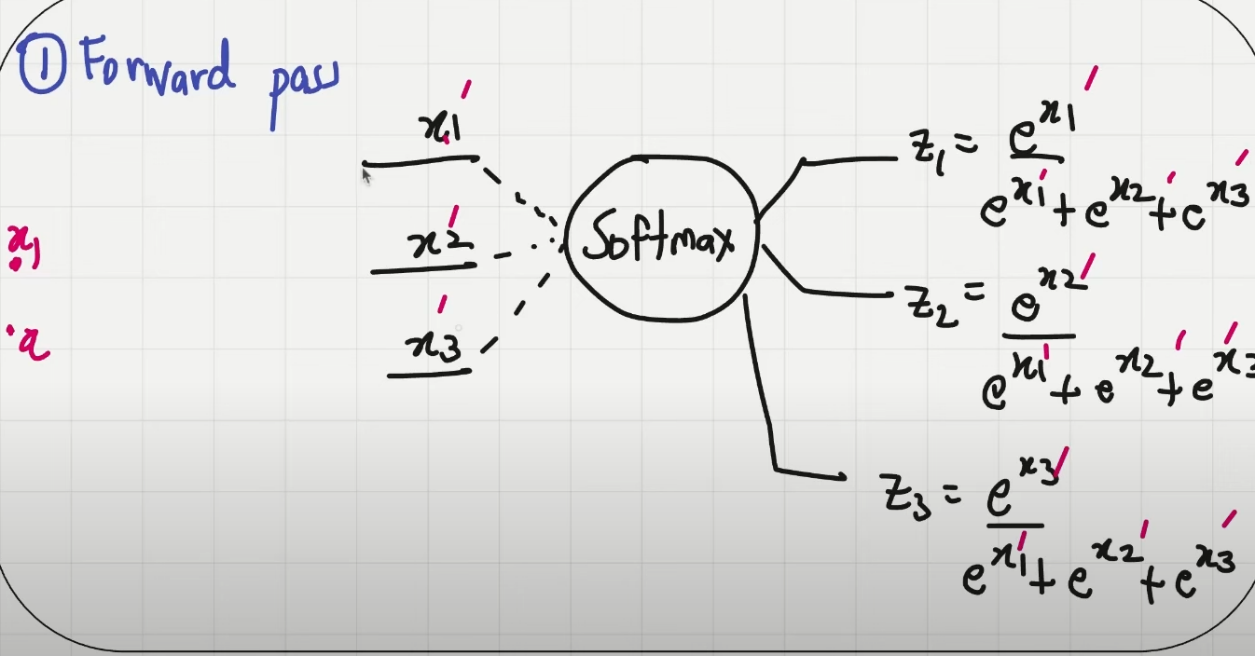
1 if x > 0

=0 if x < 0

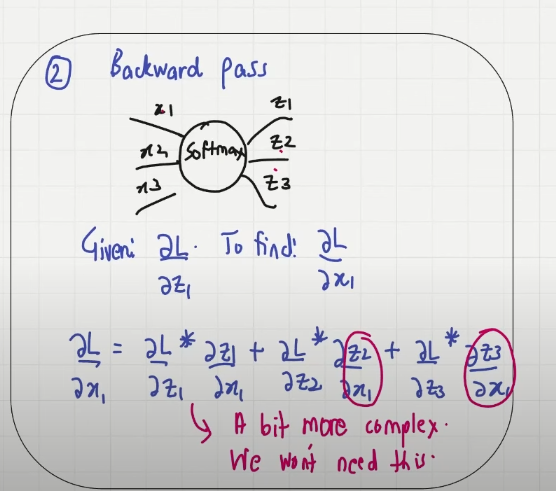
### Softmax

**Softmax** là một activation function thường được sử dụng ở layer cuối cùng của bài classification, với mục đích trả ra đầu ra là giá trị đã được phân loại. Biến đầu ra thành xác suất (tức nghĩa là tổng các giá trị = 1).

* Softmax forward pass:



* Softmax backward pass:



# Softmax activation

class Activation\_Softmax:

 # Forward pass

 def forward(self, inputs):

 # Get unnormalized probabilities

  exp\_values = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))

 # Normalize them for each sample

  probabilities = exp\_values / np.sum(exp\_values, axis=1,keepdims=True)

  self.output = probabilities

* zi​: input;
* n: tổng layer đầu ra;
* : giúp đảm bảo đầu ra luôn dương;
* : là tổng toàn bộ logit được mũ hóa, giúp chuẩn hóa thành xác suất.

## Hàm mất mát (Loss function)

Hàm mất mát (Loss function) đo lường sự khác biệt giữa đầu ra của mô hình (dự đoán) và giá trị thực tế, sử dụng Categorical Cross-Entropy function, rất phù hợp cho các bài toán phân loại đa lớp. Loss này càng nhỏ độ chính xác càng cao, đằng sau khi softmax trả ra đầu ra dùng loss phân loại.

# Common loss class

class Loss:

 # Calculates the data and regularization losses

 # given model output and ground truth values

 def calculate(self, output, y):

  # Calculate sample losses

  sample\_losses = self.forward(output, y)

  # Calculate mean loss

  data\_loss = np.mean(sample\_losses)

  # Return loss

  return data\_loss

Class loss này là lớp cha dùng để xây dựng các loại loss function ở bài toán này sử dụng Loss\_CategoricalCrossentropy. Class loss cung cấp một phương thức chung là calculate() để:

* Gọi hàm forward() từ lớp con để tính loss của từng mẫu;
* Tính trung bình loss toàn batch để trả về một con số duy nhất.

Công thức toán học tương ứng

* Lmean: loss trung bình;
* N: cỡ mẫu;
* li: loss phần tử thứ i.

Kết quả: trả ra trung bình cộng của các loss.

class Loss\_CategoricalCrossentropy(Loss):

    # Forward pass

    def forward(self, y\_pred, y\_true):

        # Number of samples in a batch

        samples = len(y\_pred)

        # Clip data to prevent division by 0

        # Clip both sides to not drag mean towards any value

        y\_pred\_clipped = np.clip(y\_pred, 1e-7, 1 - 1e-7)

        # Probabilities for target values -

        # only if categorical labels

        if len(y\_true.shape) == 1:

            correct\_confidences = y\_pred\_clipped[

                range(samples),

                y\_true

            ]

        # Mask values - only for one-hot encoded labels

        elif len(y\_true.shape) == 2:

            correct\_confidences = np.sum(

                y\_pred\_clipped \* y\_true,

                axis=1

            )

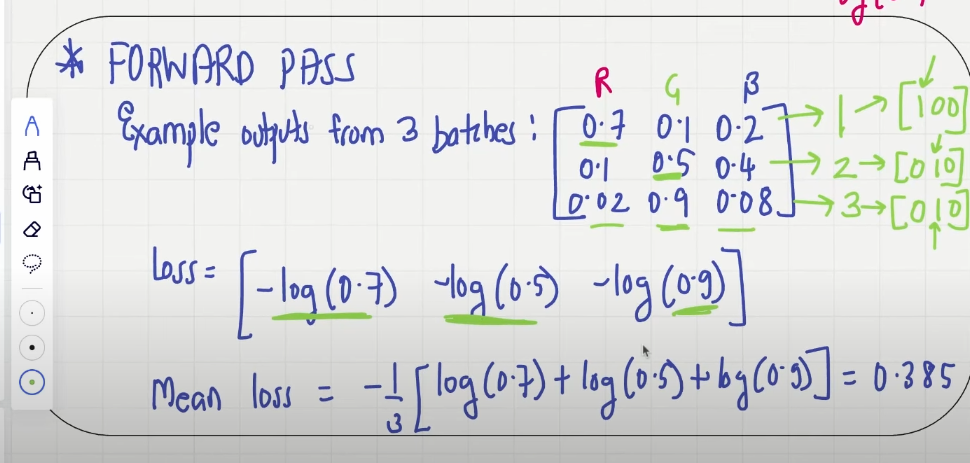
        # Losses

        negative\_log\_likelihoods = -np.log(correct\_confidences)

        return negative\_log\_likelihoods

Categorical Crossentropy Loss – dùng để đo mức độ sai lệch giữa nhãn thật và dự đoán khi bài toán là phân loại nhiều lớp (multi-class classification) (Ở đây confidence là 3 lớp)

***Phương pháp forward:***



Đầu vào là mảng dự đoán, 2 là mảng mục tiêu. Cần đảm bảo không có giá trị rất lớn hoặc nhỏ cho nên sử dụng np.clip với việc đưa ra 1 giá trị tối thiểu (1^e-7) và giá trị tối đa (1 – 1e^-7) để ngăn cản log(0) gây lỗi.

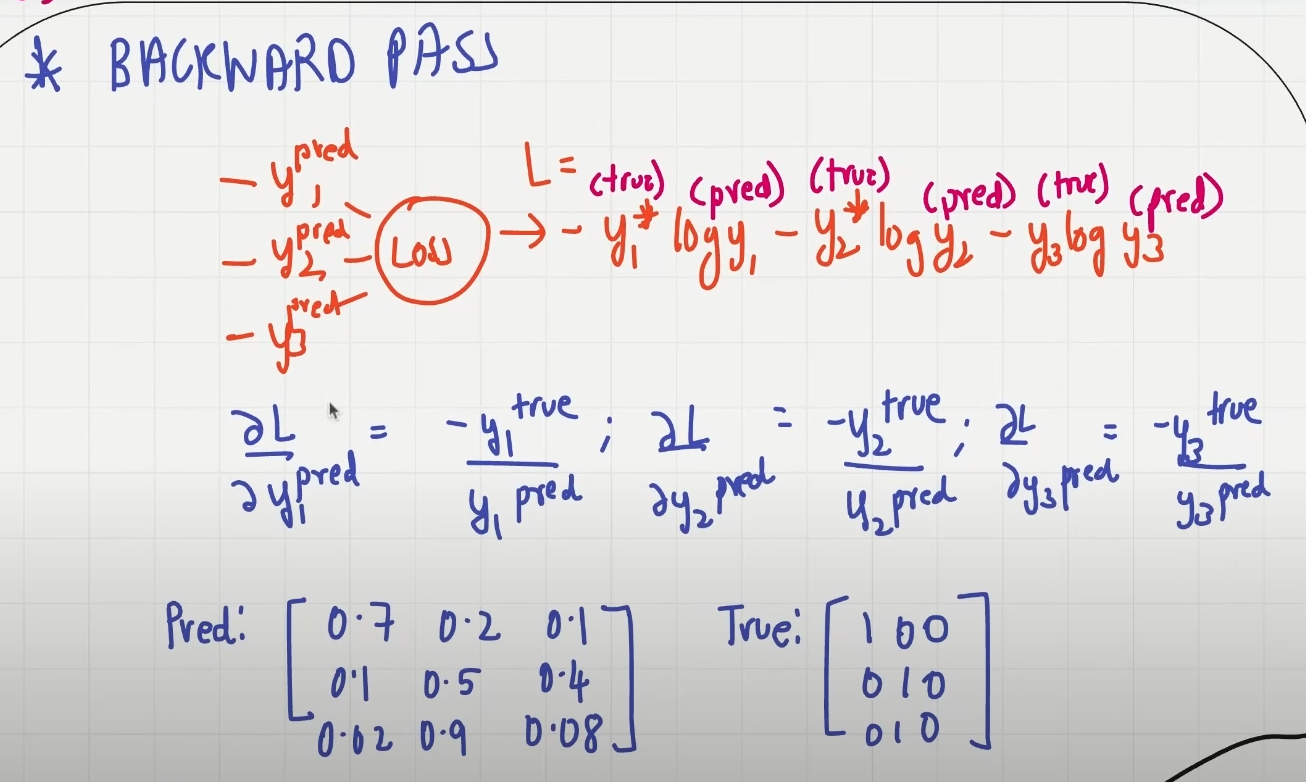
* Phân 2 trường hợp:
* Mảng 1 chiều (chỉ có số): lấy xác suất ứng với nhãn đúng trong dự đoán của mẫu thứ i nhân ma trận hàng với cột trả ra đơn giá trị
* Mảng 2 chiều (one-hot encoded): chỉ có 1 phần tử bằng 1, phần còn lại bằng 0 khác biệt với cái trên là nhân xong cộng lại

Công thức toán tương đương ở phần này

Do ở đây trả ra kết quả là xác suất cho nên khi mà lấy ra log của 1 số nhỏ hơn 1 kết quả đưa ra đều là âm cho nên ta cần đánh giá độ tốt của mô hình bằng cross-entropy loss. công thức Negative Log Likelihood (NLL) – logarit âm của xác suất dự đoán đúng:

Công thức toán học chung cuộc cho class cross entropy loss:

***Phương pháp backward:***



Tính đạo hàm của hàm loss đối với đầu vào của lớp, dùng để cập nhật trọng số trong quá trình huấn luyện mạng neural.

    # Backward pass

    def backward(self, dvalues, y\_true):

        # Number of samples

        samples = len(dvalues)

        # Number of labels in every sample

        # We'll use the first sample to count them

        labels = len(dvalues[0])

        # If labels are sparse, turn them into one-hot vector

        if len(y\_true.shape) == 1:

            y\_true = np.eye(labels)[y\_true]

        # Calculate gradient

        self.dinputs = -y\_true / dvalues

        # Normalize gradient

        self.dinputs = self.dinputs / samples

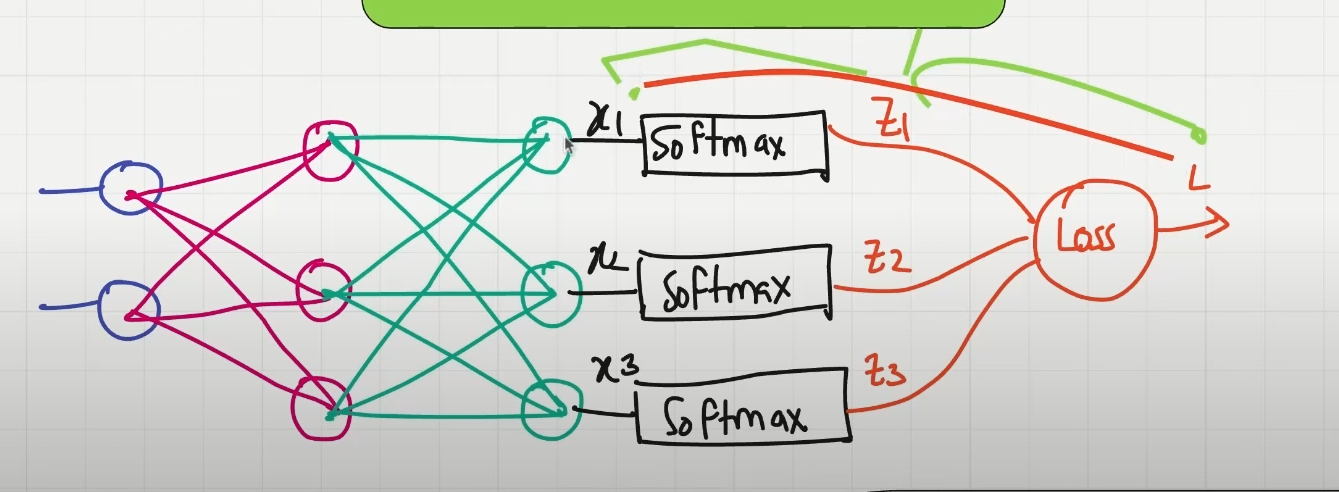
Phần này tính đạo hàm của loss (cross-entropy) đối với đầu vào của softmax,

Chuẩn bị gradient để truyền ngược trong mạng neural,

Giúp mô hình biết cần thay đổi trọng số thế nào để giảm loss ở bước cập nhật tiếp theo.

Công thức toán học:

## Optimizers: Kết hợp backpropagation với softmax activation và cross entropy loss



Nếu mà thực hiện tách riêng các hàm như trên thì khi mà tìm đạo hàm riêng của loss là rất khó bởi vì phụ thuộc vào kết quả từ softmax (như công thức kể trên của softmax). Một ví dụ dễ hiểu là loss là a1 thì phải tìm kết quả của cả softmax z1, z2,… zn, dẫn đến tính toán chậm.

Gộp Softmax activation + Categorical Crossentropy loss lại với nhau để tối ưu hiệu suất và đơn giản hóa tính đạo hàm trong quá trình học (backpropagation).

# Softmax classifier - combined Softmax activation

# and cross-entropy loss for faster backward step

class Activation\_Softmax\_Loss\_CategoricalCrossentropy:

    # Creates activation and loss function objects

    def \_\_init\_\_(self):

        self.activation = Activation\_Softmax()

        self.loss = Loss\_CategoricalCrossentropy()

    # Forward pass

    def forward(self, inputs, y\_true):

        # Output layer's activation function

        self.activation.forward(inputs)

        # Set the output

        self.output = self.activation.output

        # Calculate and return loss value

        return self.loss.calculate(self.output, y\_true)

    # Backward pass

    def backward(self, dvalues, y\_true):

        # Number of samples

        samples = len(dvalues)

        # If labels are one-hot encoded,

        # turn them into discrete values

        if len(y\_true.shape) == 2:

            y\_true = np.argmax(y\_true, axis=1)

        # Copy so we can safely modify

        self.dinputs = dvalues.copy()

        # Calculate gradient

        self.dinputs[range(samples), y\_true] -= 1

        # Normalize gradient

        self.dinputs = self.dinputs / samples

Công thức toán học:

Ảnh có chứa văn bản, Phông chữ, chữ viết tay, tài liệu

Mô tả được tạo tự động

## ADAM optimizer

Adam (Adaptive Moment Estimation) là một thuật toán tối ưu phổ biến trong huấn luyện mạng neural, kết hợp giữa Momentum và RMSProp.

### Momentum

**Momentum (động lượng)** – được sử dụng để tăng tốc độ giảm dần độ dốc, sự hội tụ và cũng làm giảm số giao động trong phương pháp **gradient desenct.** Phương pháp Momentum điều chỉnh gradient descent theo hai cách, giúp nó gần giống hơn với chuyển động vật lý của quả bóng:

* Nó đưa vào khái niệm "vận tốc" (velocity) cho các tham số mà ta đang cố tối ưu;
* Gradient tác động lên vận tốc, chứ không tác động trực tiếp lên "vị trí" (tức là giá trị tham số), giống như cách các lực vật lý thay đổi vận tốc trước, rồi mới ảnh hưởng đến vị trí sau;
* Phương pháp momentum cũng thêm một thành phần giống như ma sát (friction), giúp giảm dần vận tốc theo thời gian.

Nếu mà không dùng momentum thì bouding của for & back sẽ nhảy loạn liên tục rồi sau đó mới đến được cực tiểu

Khi thêm momentum, bounding sẽ tạo ra 1 đường thẳng trơn tru hướng đến cực tiểu, nếu thêm động lượng hướng hến gradient desenct, giúp giảm dao động và sự hội tụ giảm dần độ dốc trở nên nhanh hơn.

Nếu mà không có momentum:

Ảnh có chứa văn bản, chữ viết tay, Phông chữ, thư pháp

Mô tả được tạo tự động

Nếu có momentum:

Ảnh có chứa Phông chữ, chữ viết tay, hàng

Mô tả được tạo tự động

**Cách hoạt động:**

* Pasts matters;
* Giá trị quá khứ ảnh hưởng đến quá trình cập nhật trọng số hiện tại.

**Behind the scene:**

Ảnh có chứa thiết kế

Mô tả được tạo tự động với mức tin cậy trung bình

Mỗi lần cập nhật (bounding) là một bước cộng vector. Ta có thể hình dung hai vector — một từ bên trái, một từ bên phải — tạo thành hai cạnh vuông góc: một hướng ngang, một hướng xuống phía cực tiểu.

Khi cộng hai vector trái – phải, nếu chúng có độ lớn bằng nhau và hướng ngược nhau (theo định nghĩa cộng vector có hướng (khi chúng có độ lớn bằng nhau và hướng ngược nhau)), thì chúng sẽ triệt tiêu nhau. Kết quả còn lại là vector hướng xuống — chính là hướng đi dần về cực tiểu.

**Công thức toán học:**

Tương tự với biases

Sau mỗi lần chạy cần điều chỉnh leraning rate theo công thức của **rate decay**:

    def pre\_update\_params(self):

        if self.decay:

            self.current\_learning\_rate = self.learning\_rate \* (1. / (1. + self.decay \* self.iterations))

giúp cho khi mà học tới 1 mức độ nhất định không làm thay đổi loss funct thì sẽ giũ nguyên để cho lr không nhảy lung tung đặc trưng cho sự hội tụ ổn định

Công thức toán học của **rate decay:**

* t: iteration number;
* alpha: learning rate;
* decay: lear rate decay (loanh quanh tầm 10^-3).

### RMSProp

**RMSProp – Root Mean Square Propagation**: giúp cải thiện tốc độ hội tụ bằng cách điều chỉnh learning rate cho từng tham số dựa trên lịch sử gradient để mà tìm kích thước bước thích ứng hoặc hiệu quả.

Công thức:

Giải thích đại lượng:

* rho: (~0.9) cache memory decay rate hay "mức độ ghi nhớ quá khứ" của các gradient cũ (rô theo tiếng hy lạp).

Với việc đưa ra nhiều trọng số cho past cache, phương pháp này ưu tiên trí nhớ về gradient cũ nhiều hơn gradient hiện tại. Thêm weight sum của past and present (khá giống với momentum, khác biệt so với người tiền nhiệm là thuật toán adagrad là thêm gradient^2). Cũng do tốc độ chậm nên đây là bài toán đánh đổi tốc độ lấy sự ổn định tránh bị vanishing, nó cũng là lý do RMSProp được ưa chuộng.

Công thức chuyển đổi trạng thái weight hay bisases:

* parmGradient: đạo hàm của hàm mất mát;
* epsilon = 1e-7.

Như vậy, có thể thấy Momentum cho tốc độ và RMSprop cho sự ổn định, nên trong thực thế, cả hai phương pháp này được kết hợp với nhau.

import numpy as np

# Adam optimizer

class Optimizer\_Adam:

    # Initialize optimizer - set settings

    def \_\_init\_\_(self, learning\_rate=0.001, decay=0., epsilon=1e-7, beta\_1=0.9, beta\_2=0.999):

        self.learning\_rate = learning\_rate

        self.current\_learning\_rate = learning\_rate

        self.decay = decay

        self.iterations = 0

        self.epsilon = epsilon

        self.beta\_1 = beta\_1

        self.beta\_2 = beta\_2

    # Call once before any parameter updates

    def pre\_update\_params(self):

        if self.decay:

            self.current\_learning\_rate = self.learning\_rate \* (1. / (1. + self.decay \* self.iterations))

    # Update parameters

    def update\_params(self, layer):

        # If layer does not contain cache arrays, create them filled with zeros

        if not hasattr(layer, 'weight\_cache'):

            layer.weight\_momentums = np.zeros\_like(layer.weights)

            layer.weight\_cache = np.zeros\_like(layer.weights)

            layer.bias\_momentums = np.zeros\_like(layer.biases)

            layer.bias\_cache = np.zeros\_like(layer.biases)

        # Update momentum with current gradients

        layer.weight\_momentums = self.beta\_1 \* layer.weight\_momentums + (1 - self.beta\_1) \* layer.dweights

        layer.bias\_momentums = self.beta\_1 \* layer.bias\_momentums + (1 - self.beta\_1) \* layer.dbiases

        # Get corrected momentum

        # self.iteration is 0 at first pass and we need to start with 1 here

        weight\_momentums\_corrected = layer.weight\_momentums / (1 - self.beta\_1 \*\* (self.iterations + 1))

        bias\_momentums\_corrected = layer.bias\_momentums / (1 - self.beta\_1 \*\* (self.iterations + 1))

        # Update cache with squared current gradients

        layer.weight\_cache = self.beta\_2 \* layer.weight\_cache + (1 - self.beta\_2) \* layer.dweights\*\*2

        layer.bias\_cache = self.beta\_2 \* layer.bias\_cache + (1 - self.beta\_2) \* layer.dbiases\*\*2

        # Get corrected cache

        weight\_cache\_corrected = layer.weight\_cache / (1 - self.beta\_2 \*\* (self.iterations + 1))

        bias\_cache\_corrected = layer.bias\_cache / (1 - self.beta\_2 \*\* (self.iterations + 1))

        # Vanilla SGD parameter update + normalization with square rooted cache

        layer.weights += -self.current\_learning\_rate \* weight\_momentums\_corrected / (np.sqrt(weight\_cache\_corrected) + self.epsilon)

        layer.biases += -self.current\_learning\_rate \* bias\_momentums\_corrected / (np.sqrt(bias\_cache\_corrected) + self.epsilon)

    # Call once after any parameter updates

    def post\_update\_params(self):

        self.iterations += 1

**ADAM** sẽ kết hợp đạo hàm riêng từ RMSProp vớimomentum

Công thức toán học:

- beta\_1 là hệ số cho momentum;

* beta\_2 là hệ số cho cache;
* alpha: lr đã tính theo rate decat.

Sử dụng bias correction để đảm bảo kết quả ở các bước đầu tiên không bị lệch do khởi tạo, do đó kết hợp hai giá trị này để điều chỉnh bước nhảy (step size) một cách linh hoạt.

Nếu giải thích theo hiện tượng vật lí thì Momentum giống như 1 quả cầu lao xuống dốc, còn Adam như 1 quả cầu rất nặng có ma sát, vì vậy nó dễ dàng vượt qua local minimum tới global minimum và khi tới global minimum nó không mất nhiều thời gian dao động qua lại quanh đích vì nó có ma sát nên dễ dừng lại hơn.

## Train mô hình

* In ra các bước chạy ngầm:

# Create dataset

X, y = spiral\_data(samples=100, classes=3)

* Đầu tiên tạo dataset với 100 mẫu và 3 lớp rgb

# Create Dense layer with 2 input features and 64 output values

dense1 = Layer\_Dense(2, 64)

* Tạo dense layer với 2 input đầu vào và 64 đầu ra

# Create second Dense layer with 64 input features (as we take output of previous layer here) and 3 output values (output values)

dense2 = Layer\_Dense(64, 3)

* Tạo dense layer từ dense layer 1 với 64 đầu vào do hứng từ dense1 và 3 output ra

# Create ReLU activation (to be used with Dense layer)

activation1 = Activation\_ReLU()

# Create Softmax classifier's combined loss and activation

loss\_activation = Activation\_Softmax\_Loss\_CategoricalCrossentropy()

* Gọi hàm để có thể tạo activation function cũng như loss function do đã code ở trên

# Create optimizer

optimizer = Optimizer\_Adam(learning\_rate=0.02, decay=1e-5)

* Tạo adam optimizers với lr = 0.02 và rate decay = 1e-5

# Train in loop

for epoch in range(10001):

    # Perform a forward pass of our training data through this layer

    dense1.forward(X)

    # Perform a forward pass through activation function

    # takes the output of first dense layer here

    activation1.forward(dense1.output)

    # Perform a forward pass through second Dense layer

    # takes outputs of activation function of first layer as inputs

    dense2.forward(activation1.output)

    # Perform a forward pass through the activation/loss function

    # takes the output of second dense layer here and returns loss

    loss = loss\_activation.forward(dense2.output, y)

    # Calculate accuracy from output of activation2 and targets

    # calculate values along first axis

    predictions = np.argmax(loss\_activation.output, axis=1)

    if len(y.shape) == 2:

        y = np.argmax(y, axis=1)

    accuracy = np.mean(predictions == y)

    if not epoch % 100:

        print(f'epoch: {epoch}, ' +

              f'acc: {accuracy:.3f}, ' +

              f'loss: {loss:.3f}, ' +

              f'lr: {optimizer.current\_learning\_rate}')

    # Backward pass

    loss\_activation.backward(loss\_activation.output, y)

    dense2.backward(loss\_activation.dinputs)

    activation1.backward(dense2.dinputs)

    dense1.backward(activation1.dinputs)

    # Update weights and biases

    optimizer.pre\_update\_params()

    optimizer.update\_params(dense1)

    optimizer.update\_params(dense2)

    optimizer.post\_update\_params()



Khởi tạo vòng lặp với 10000 lần đồng thời thực hiện huấn luyện mô hình sử dụng đạo hàm hàm hợp (chain-rule) tạo các vòng lặp lồng phụ thuộc vào nhau và đồng thời xử lý forward và backward lẫn với update weight & bias sau mỗi lần lặp. Kết quả trả về với độ chính xác 0.780 của các bước chạy ngầm.

# Validate the model

# Create test dataset

X\_test, y\_test = spiral\_data(samples=100, classes=3)

# Perform a forward pass of our testing data through this layer

dense1.forward(X\_test)

# Perform a forward pass through activation function

# takes the output of first dense layer here

activation1.forward(dense1.output)

# Perform a forward pass through second Dense layer

# takes outputs of activation function of first layer as inputs

dense2.forward(activation1.output)

# Perform a forward pass through the activation/loss function

# takes the output of second dense layer here and returns loss

loss = loss\_activation.forward(dense2.output, y\_test)

# Calculate accuracy from output of activation2 and targets

# calculate values along first axis

predictions = np.argmax(loss\_activation.output, axis=1)

if len(y\_test.shape) == 2:

 y\_test = np.argmax(y\_test, axis=1)

accuracy = np.mean(predictions == y\_test)

print(f'validation, acc: {accuracy:.3f}, loss: {loss:.3f}')



Chạy đánh giá: độ chính xác 0.643.

Có thể thấy rằng độ chính xác trên tập train và tập test chênh lệch khoảng 0.1. Điều này cho thấy mô hình có khả năng đã rơi vào tình trạng quá khớp (overfitting). Cụ thể, khi sử dụng thuật toán tối ưu Adam, mô hình đã học quá kỹ các đặc trưng của tập huấn luyện, bao gồm cả các nhiễu (noise) trong dữ liệu, dẫn đến hiệu suất thấp hơn trên tập kiểm tra.

Điều này dẫn đến có khả năng acc thực sự không chính xác là acc = 0.643.

## Testing & validation

### Hướng giải quyết

* Kiểm tra độ phức tạp của bản thân mô hình. Mô hình càng đơn giản thì khả năng khái quát càng tốt. Hiện tại trong code, ta có giảm bớt layer dense (2, 64) có thể dẫn đến độ phức tạp giảm bớt (giảm hyperparameter);
* Giảm bớt epoch ở vòng lặp for vì epoch làm tăng loss;
* Thêm dropout.

### Đề xuất các giải pháp nâng cao

Trong code hiện đang có rất nhiều siêu tham số (hyperparameter) như kiểu fix cứng giá trị như dense layer, learing rate,.. và điều hiện tại cần làm là tuning chúng. Điều này cần 1 tập test data hoàn toàn mới tùy chỉnh thay hyperparameter khác nhau và sau đó tệp này sau được gọi là validation data.

* Testing data: 1 tập dataset chạy nn hoàn toàn mới;
* Validation data: Tối ưu hyperparameter (layer, neuron, activation function, epoch, learing rate,.. ) từ nn.

Việc cần làm hiện tại là chia data làm sao để thành dữ liệu kiểm tra và xác thực

### Phương pháp

1. **K-fold cross validation**

**Case 1:** Khi mà có nhiều data

* Training data: 60%;
* Validation data: 30%;
* Test data: 10%.

Tuy nhiên, trong đại đa số trường hợp thì ko có nhiều data như vậy

**Case 2:** Khi mà có lượng data hạn chế

* Training data: 80%;
* Test data: 20%.

Câu hỏi ở đây vậy dữ liệu validation lấy ở đâu?

**Sử dụng kĩ thuật K-fold cross validation):**

Trong dataset của training data chia nó thành 5 phần A, B, C, D, E

Lần lặp 1: => val error = E1

A: validation

B, C, D, E: training

Lần lặp 2:

B: validation => val error = E2

A, C, D, E: training

..

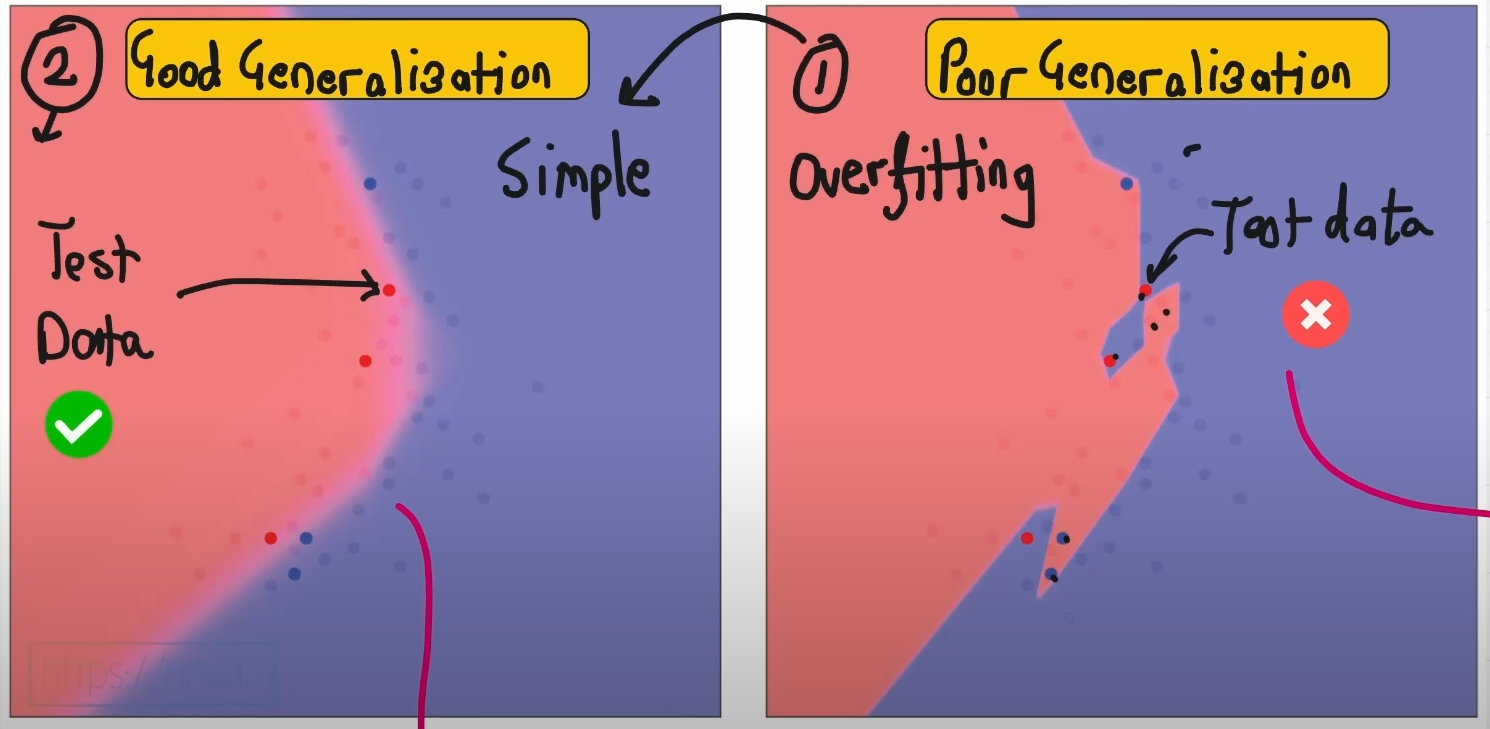
cứ lặp v cho đến hết

Mỗi lần lặp, 1 phần dữ liệu được thành validation, còn lại training

Và sau đó sẽ có công thức toán học:

Tính mean để ra cái giá trị thấp nhất.

1. **L1/L2 Regularization**

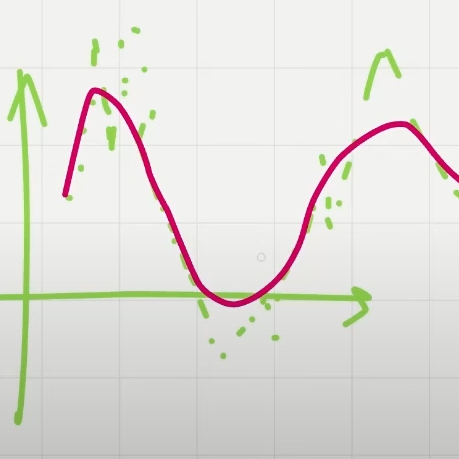


* Mong muốn hiện tại đang là chuyển từ mô hình 1 (overfitting) sang mô hình thứ 2 (regularization);
* Mục tiêu: cho bất kì điểm nào trên model đều có thể phân loại rõ ràng

Với hình 1: Weight, biases lớn, suy ra generalization kém.



Với hình 2: Weight, biases nhỏ, suy ra generalization tốt.



Từ hình này có thể thấy nó đơn giản vì không cố gắng đi qua 1 số điểm nhưng tối ưu về mặt hiệu suất (bài toán đánh đổi) của nó trên dữ liệu mới, suy ra mô hình hoạt động tốt. Do vậy, khi giảm weights & bisases nếu weights & bisases vẫn quá lớn cần áp dụng hình phạt (penalty) vào loss funct.

Hình phạt (penalty) đó được áp dụng thông qua L1/L2 Regularization

Có 2 hình thức regularization + công thức toán học:

=> loss = dataloss + L1

=> loss = dataloss + L2

=> Forward pass



=> Nên nhớ điều cuối cùng thay đổi chi có loss funct

Backward pass:

Derativite L1 & L2

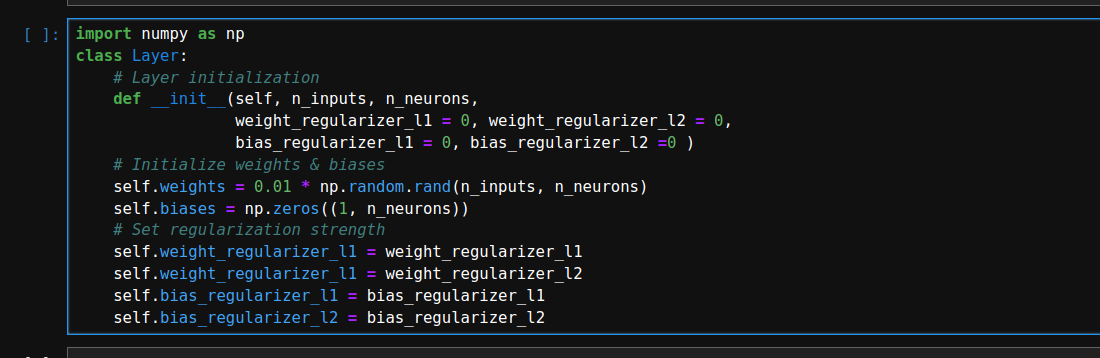
=1 if W\_m > 0

= -1 if W\_m < 0

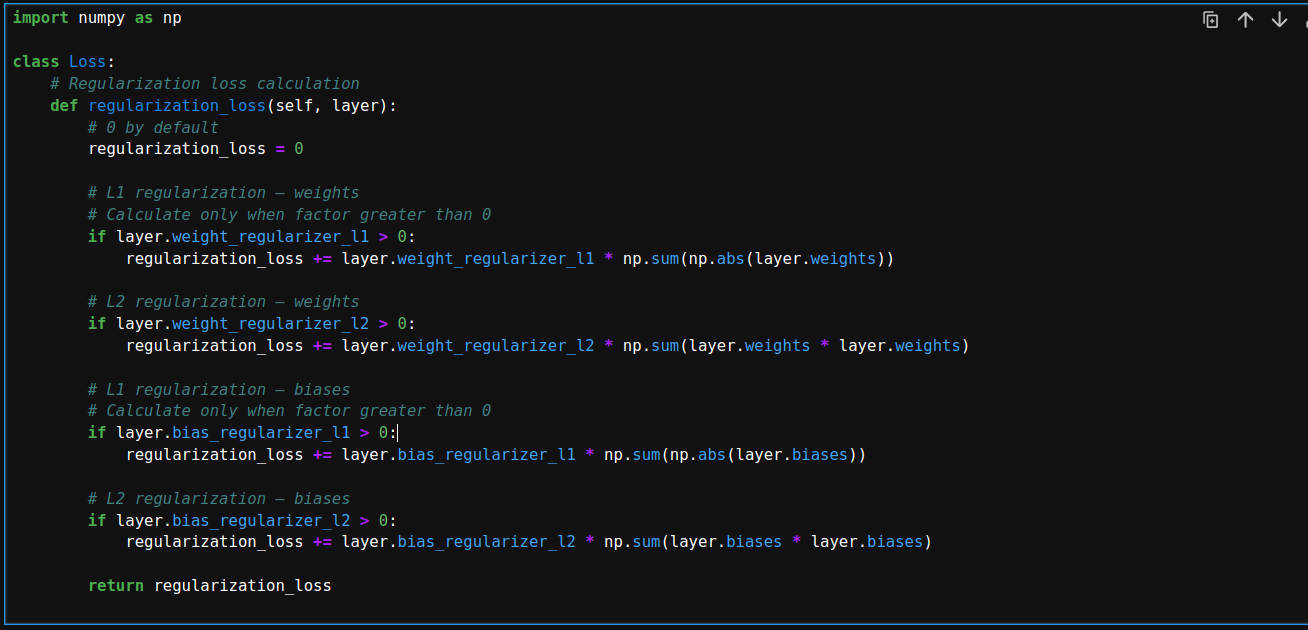
Example: L1 weight = [0.2 0.8 -0.5 ]

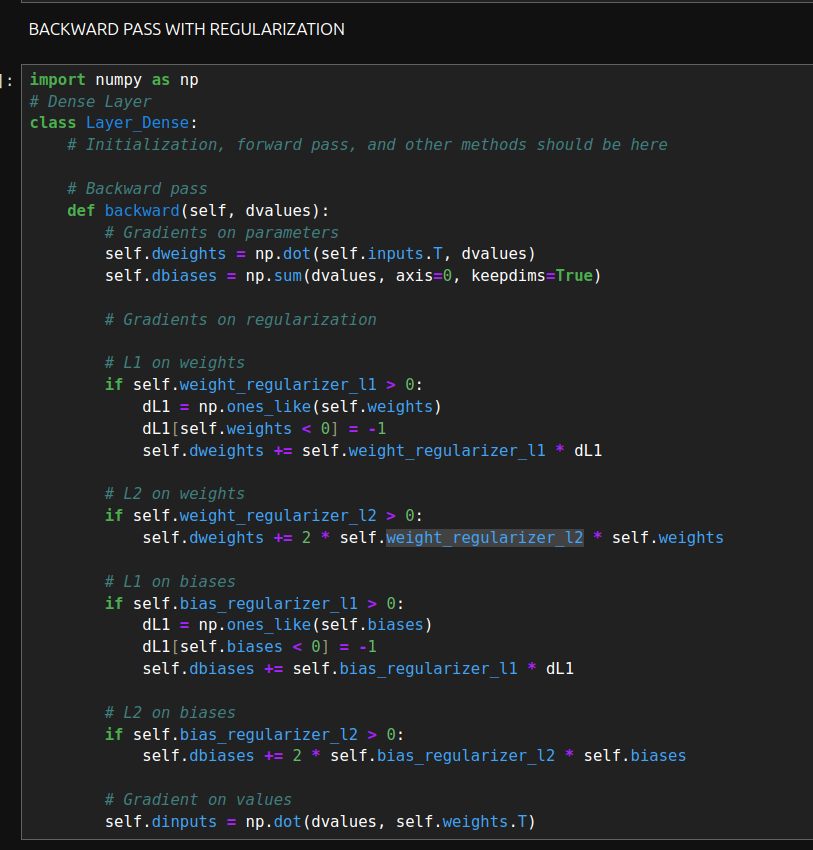
dL1W\_m = [1 1 -1 ] (\* \lambda )

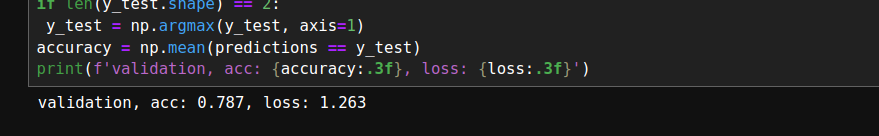
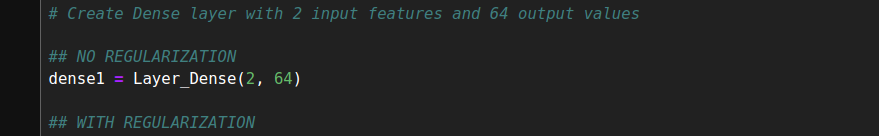
Phải cập nhập lại class layer

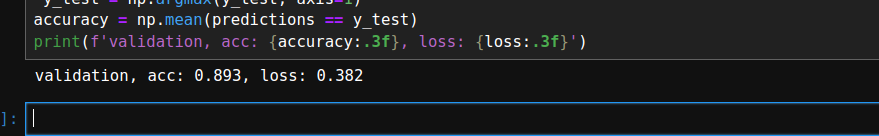
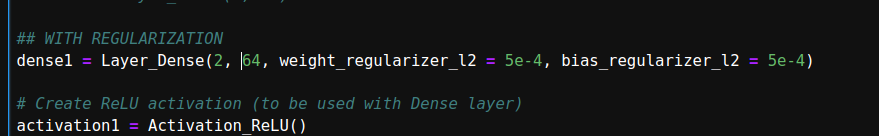


Sau đó, áp dụng công thức toán kể trên để cập nhập weight & bisaes loss funct









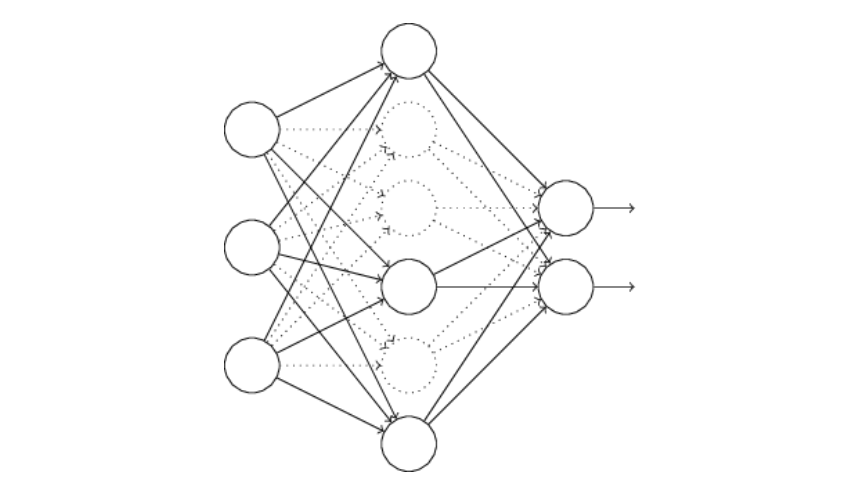
Nếu có regularization => acc: 0.813

Nếu ko có regularization => acc: 0.8

=> Độ chính xác được cải thiện hơn để tránh tình trạng overfitting

=> good generalization

1. **Dropout**



Dropout là kỹ thuật giúp disabled neuron ngẫu nhiên trong quá trình chạy.

Các neuron sẽ có weight & bias ngẫu nhiên liên quan đến chúng.

Ngẫu nhiên chọn 1 số neuron nhất định và đặt đầu ra của nó là 0 để nó không có đóng góp gì trong quá trình truyền tiếp và trong mỗi lần lặp lịa ta chọn các neuron khác nhau và disabled nó

Lý do làm vậy:

Trong quá trình truyền tiếp, 1 số neuron trở nên lazy và chúng phụ thuộc quá nhiều vào các neuron khác

VD: Neuron 2 nhận thấy neuron 1 có loss funct thấp và không cập nhập weigth của nó nhiều lần, do vậy neuron 2 thực hiện trong mỗi lần lặp nó chỉ copy weight từ neuron 1 và nó cũng có thể xảy ra ở nhiều layer khác nhau => **vấn đề đồng thích nghi (co-adaptation)**

Vấn đề này do nó không có generalization perform tốt, đầu ra kia sẽ không cố gắng tạo ra trộng số của riêng nó, nó sẽ cố nhận weghit từ neuron 1 => bad

VD: Neuron 1 sau lần lặp đầu xong thì đến lần 2 sẽ tắt, điều này ép buộc neuron phải tạo ra weight của riêng nó

Tác dụng:

Ngăn chặn vấn đề đồng thích nghi (co-adaptation)

Mô hình luôn sd neuron để ghi nhớ 1 số mẫu or feat nhất định vì nó bị disabled

Forward pass

Sử dụng phân phối nhị thức (binomial) trong xác suất thống kê

np.random.binomial(n, p, size)

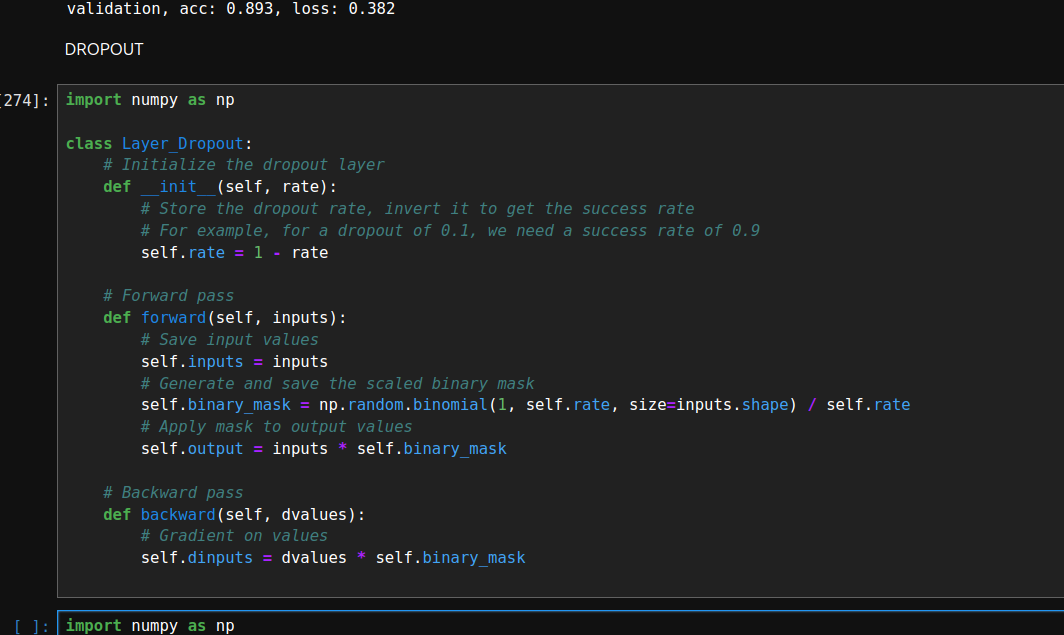
Trả về số lượng thành công trong n lần thử, với xác suất thành công mỗi lần là p.

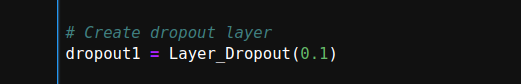
Công thức toán học:

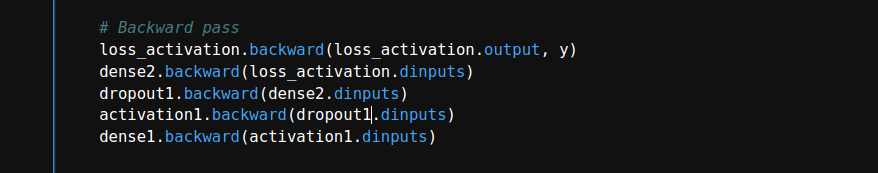
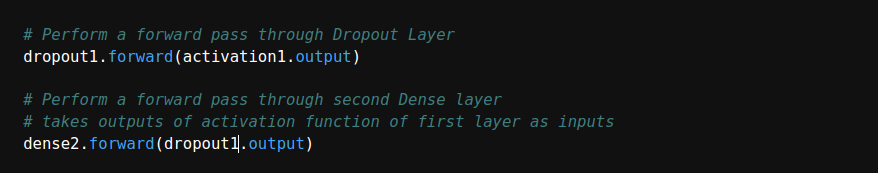
Np.random.binomial(1, 1- dropout, size)/1-dropout

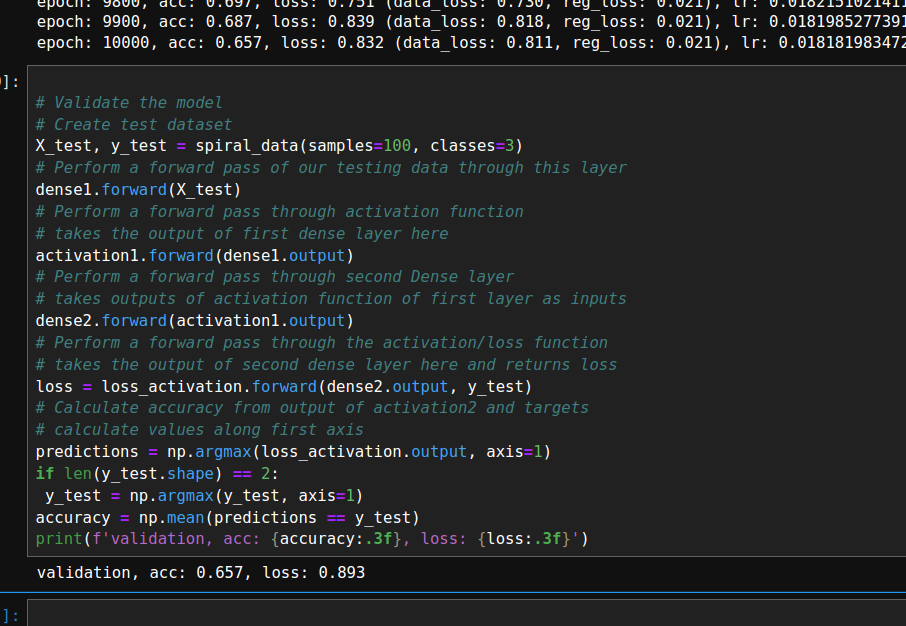
Backward pass

self.dinputs = dvalues \* self.binary\_mask









về độ chính xác này có thể cải thiện nhiều hơn thông qua điều chỉnh hay tối ưu hóa hyperparameter. Tuy nhiên trọng tâm ở đây real acc có thể >= train acc

=> Dropout thực sự giúp good generalization