TRAINING NEURAL NETWORKS, PART I

- ThS. Đoàn Chánh Thống
- ThS. Nguyễn Cường Phát
- ThS. Nguyễn Hữu Lợi
- ThS. Trương Quốc Dũng
- ThS. Nguyễn Thành Hiệp
- ThS. Võ Duy Nguyên
- ThS. Nguyễn Văn Toàn
- ThS. Lê Ngô Thục Vi
- TS. Nguyễn Duy Khánh
- TS. Nguyễn Tấn Trần Minh Khang

Administrative

- Assignment 1 due Thursday (today), 11:59pm on Canvas.
- Bài tập 1 sẽ nộp vào Thứ Năm (hôm nay), 23:59 trên Canvas.
- Assignment 2 out today.
- Bài tập 2 ra hôm nay.
- Project proposal due Tuesday April 25.
- Đề xuất dự án vào thứ ba ngày 25 tháng 4.

Administrative

- Notes on backprop for a linear layer and vector/tensor derivatives linked to Lecture 4 on syllabus.
- Ghi chú về backprop cho lớp tuyến tính và đạo hàm vectơ/tensor được liên kết với Bài giảng 4 trong giáo trình.

Overview

1. One time setup
   + activation functions, preprocessing, weight initialization, regularization, gradient checking
2. Training dynamics
   + Babysitting the learning process,
   + parameter updates, hyperparameter optimization
3. Evaluation
   + model ensembles

PART 01

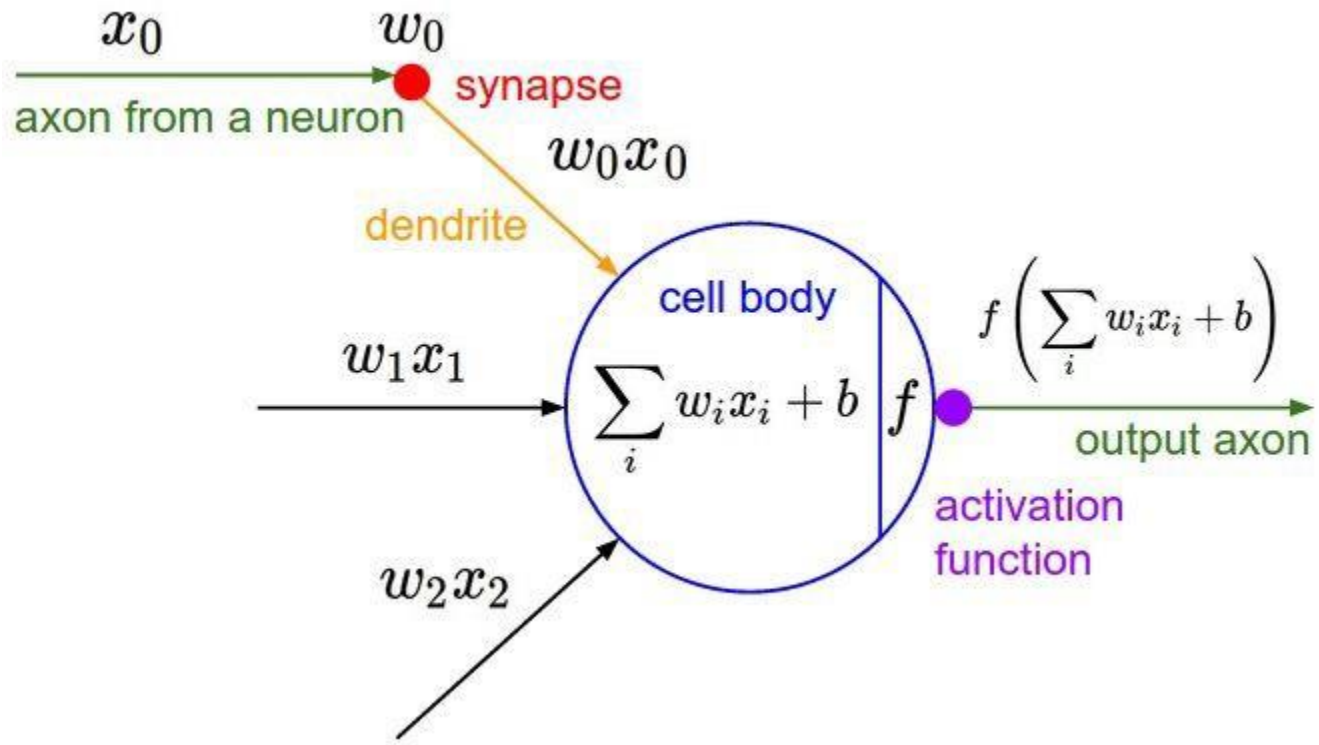
Training Neural Networks, Part 1

- Activation Functions.
- Các hàm kích hoạt.
- Data Preprocessing.
- Tiền xử lý dữ liệu.
- Weight Initialization.
- Khởi tạo trọng số.
- Batch Normalization.
- Chuẩn hóa đầu ra.
- Babysitting the Learning Process.
- Chăm sóc quá trình học tập.
- Hyperparameter Optimization.
- Tối ưu hóa siêu tham số.

Training Neural Networks, Part 1

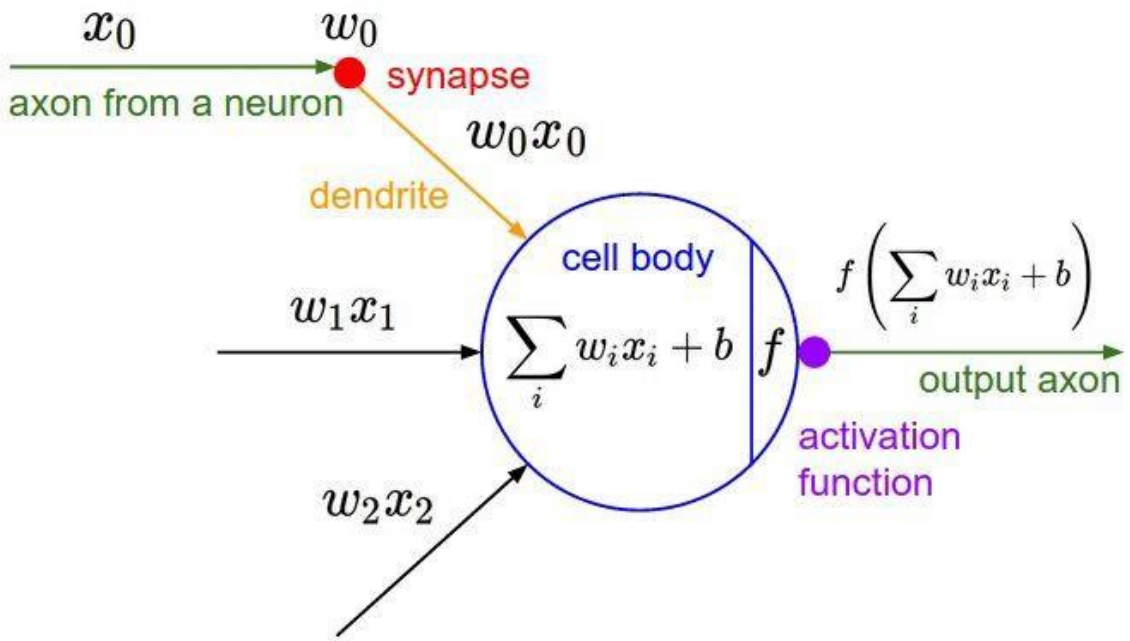
ACTIVATION FUNCTIONS

Activation functions

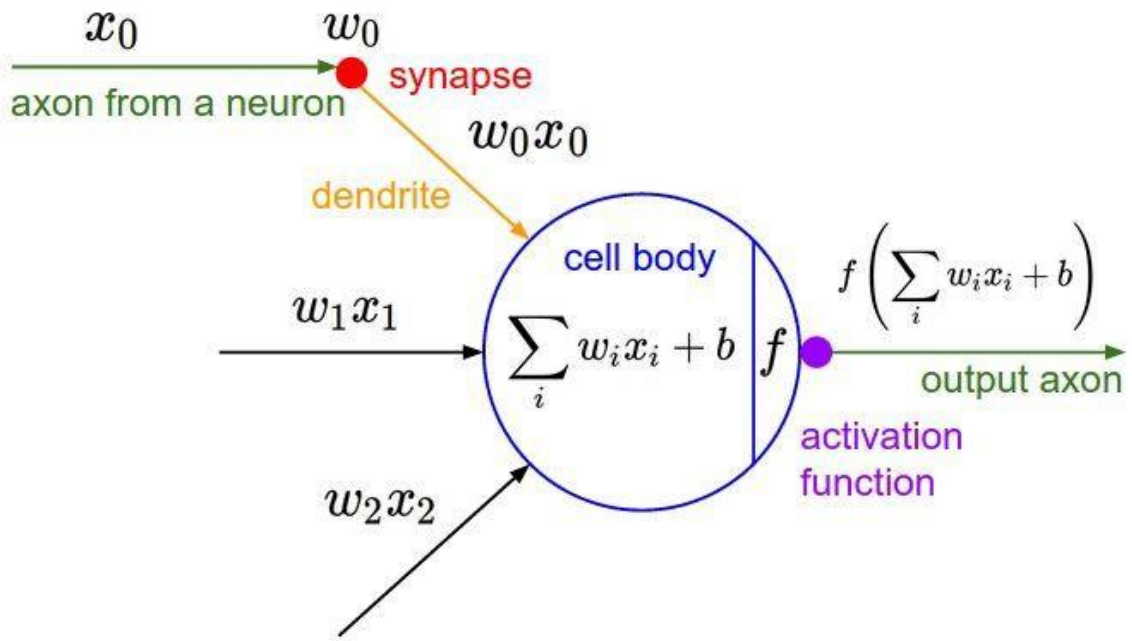


Activation functions

– Nơ ron nhận mấy tín hiệu đầu vào?

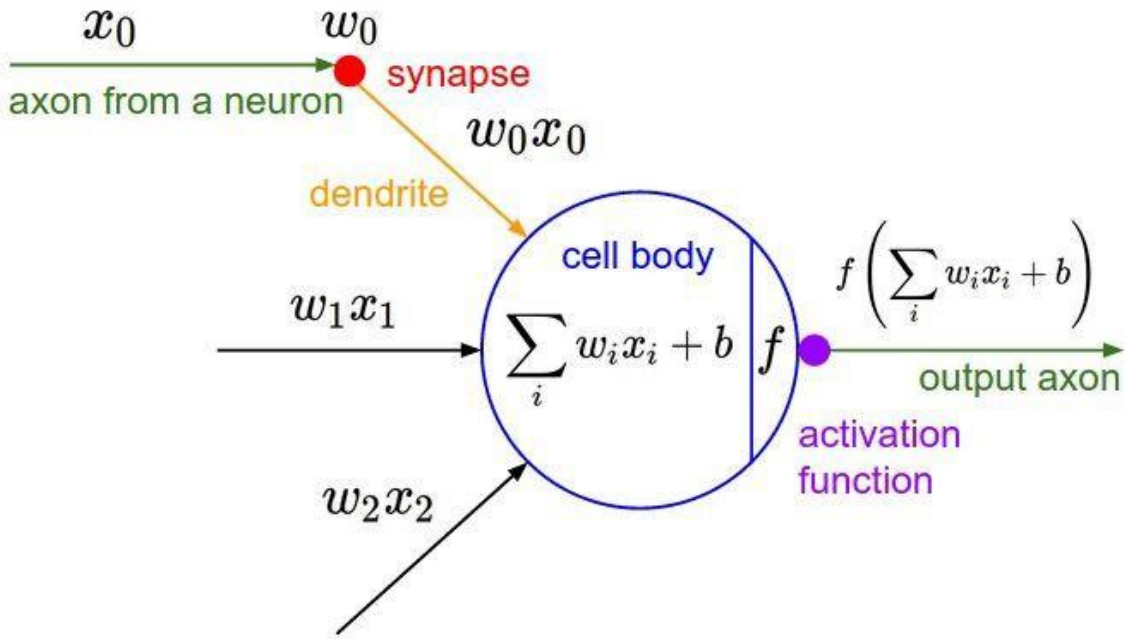


Activation functions



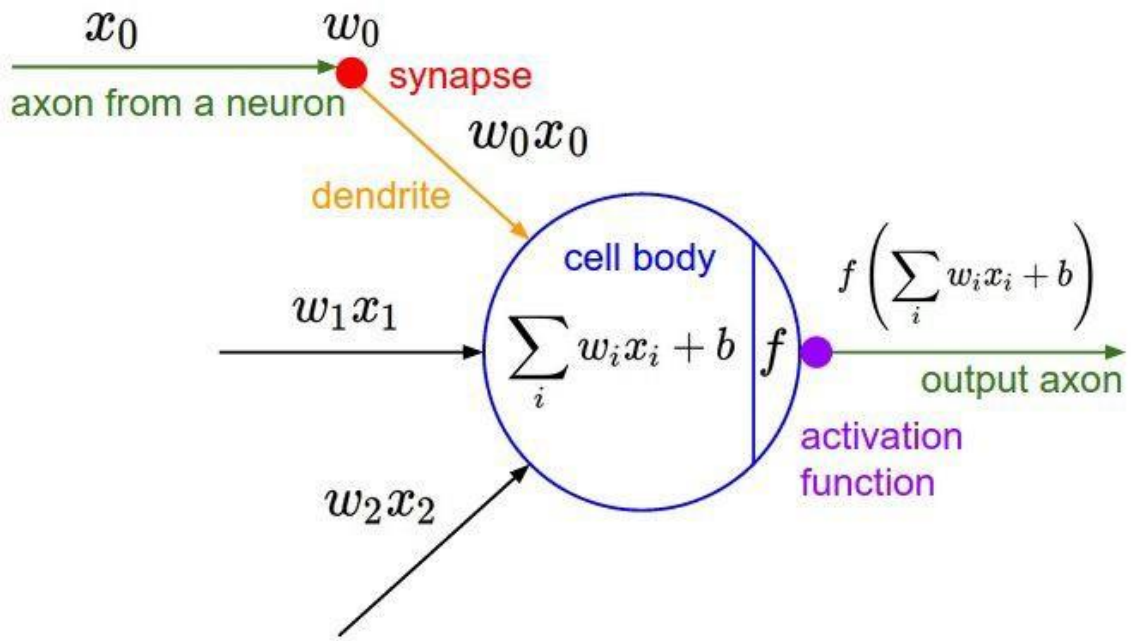
- Nơ ron nhận mấy tín hiệu đầu vào?
- Nơ ron nhận 3 tín hiệu đầu vào.

Activation functions



– Tín hiệu đầu vào thứ nhất là gì?

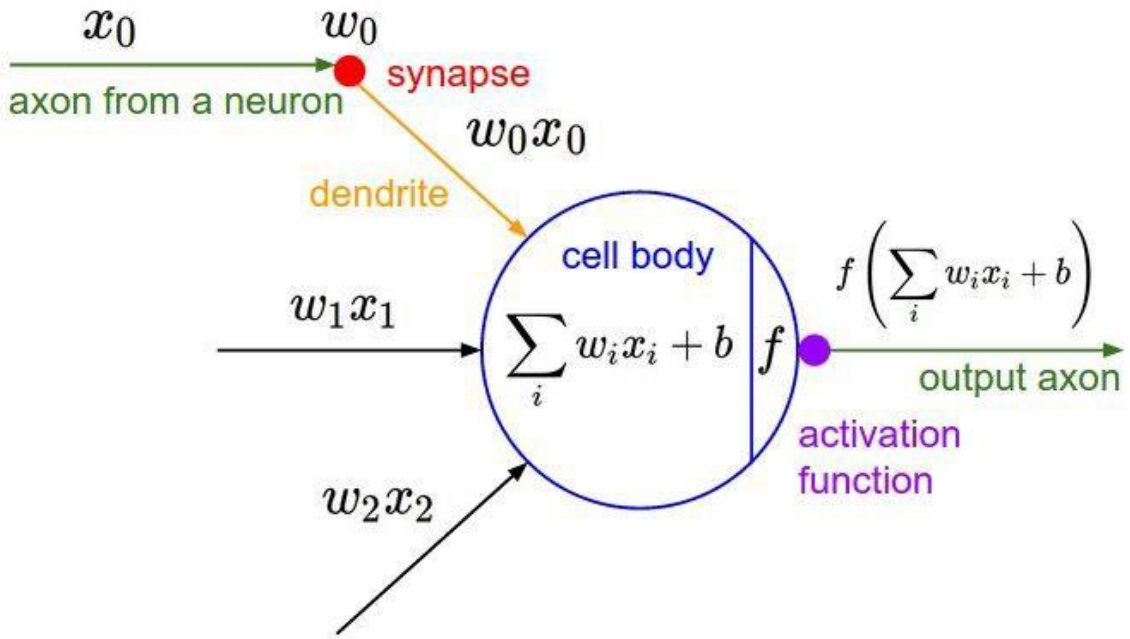
Activation functions



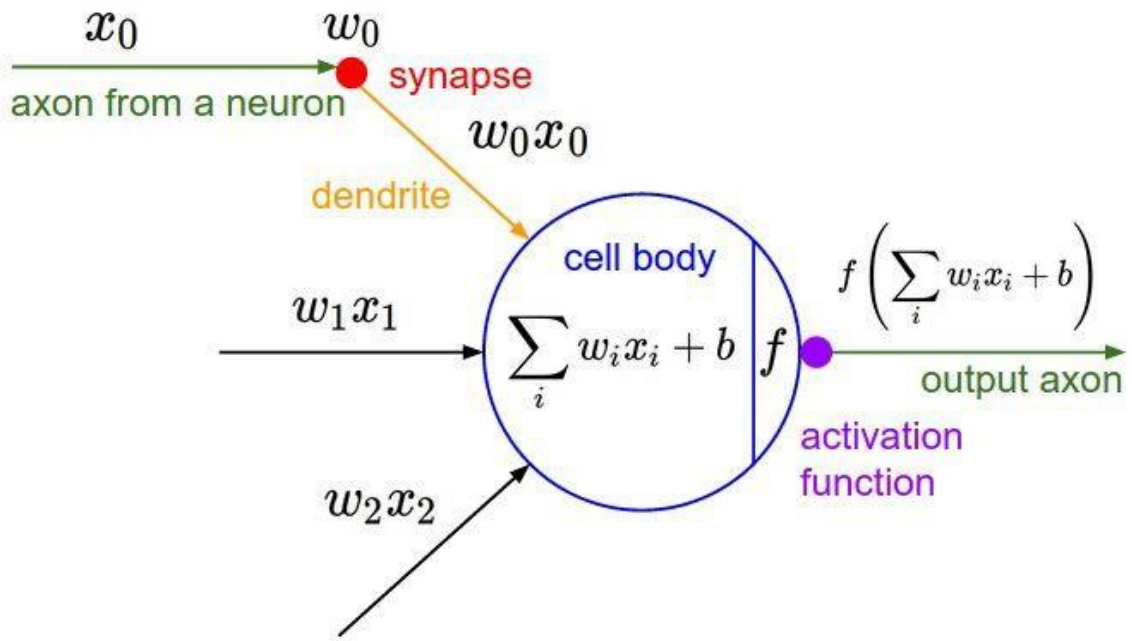
- Tín hiệu đầu vào thứ nhất là gì?
- Tín hiệu đầu vào thứ nhất là x_0.

Activation functions

– Tín hiệu đầu vào thứ hai là gì?



Activation functions

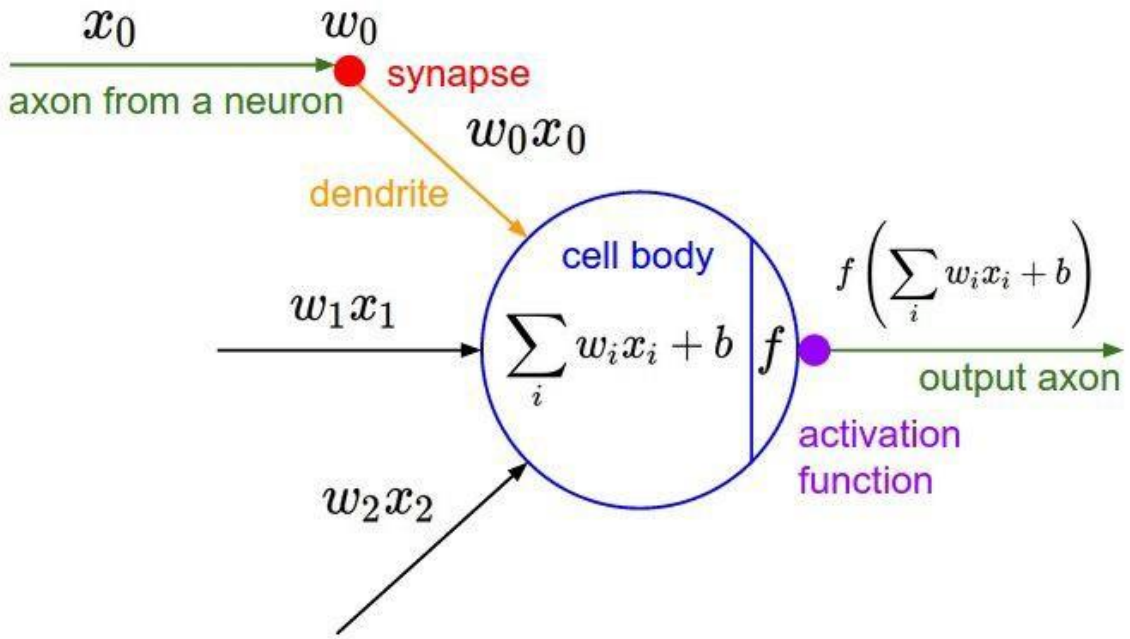


– Tín hiệu đầu vào thứ hai là gì?

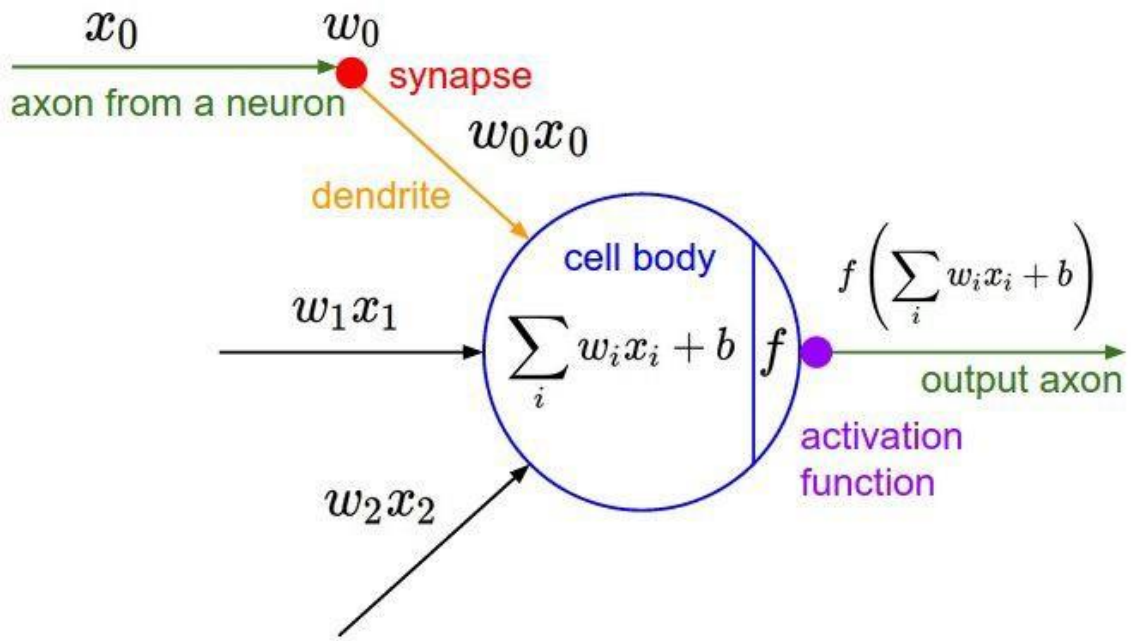
– Tín hiệu đầu vào thứ hai là x_1.

Activation functions

– Tín hiệu đầu vào thứ ba là gì?



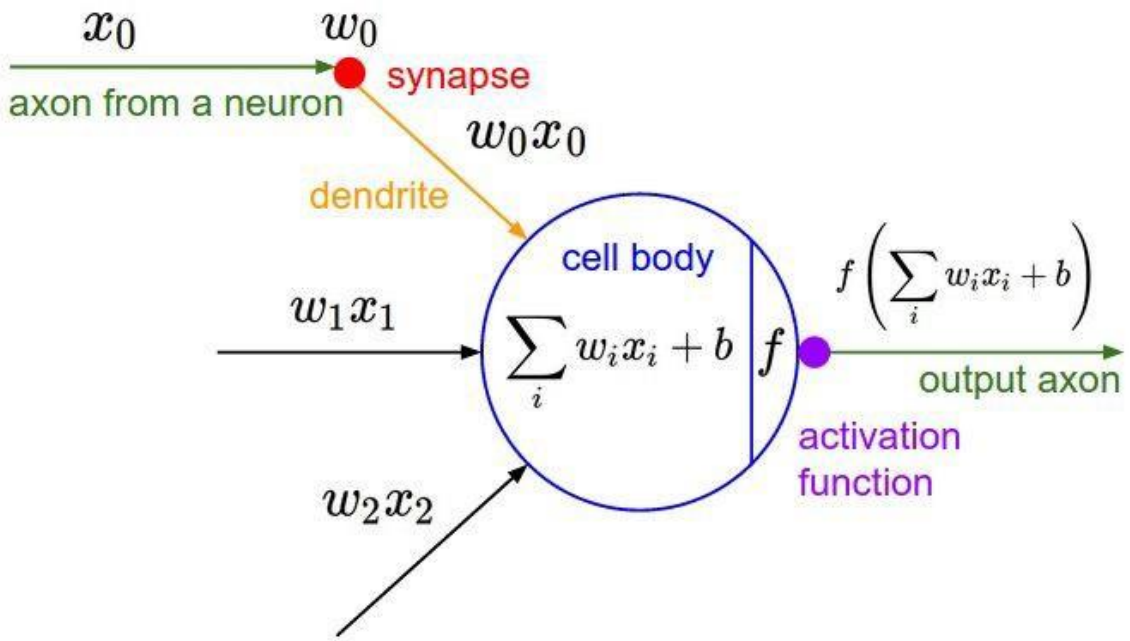
Activation functions



– Tín hiệu đầu vào thứ ba là gì?

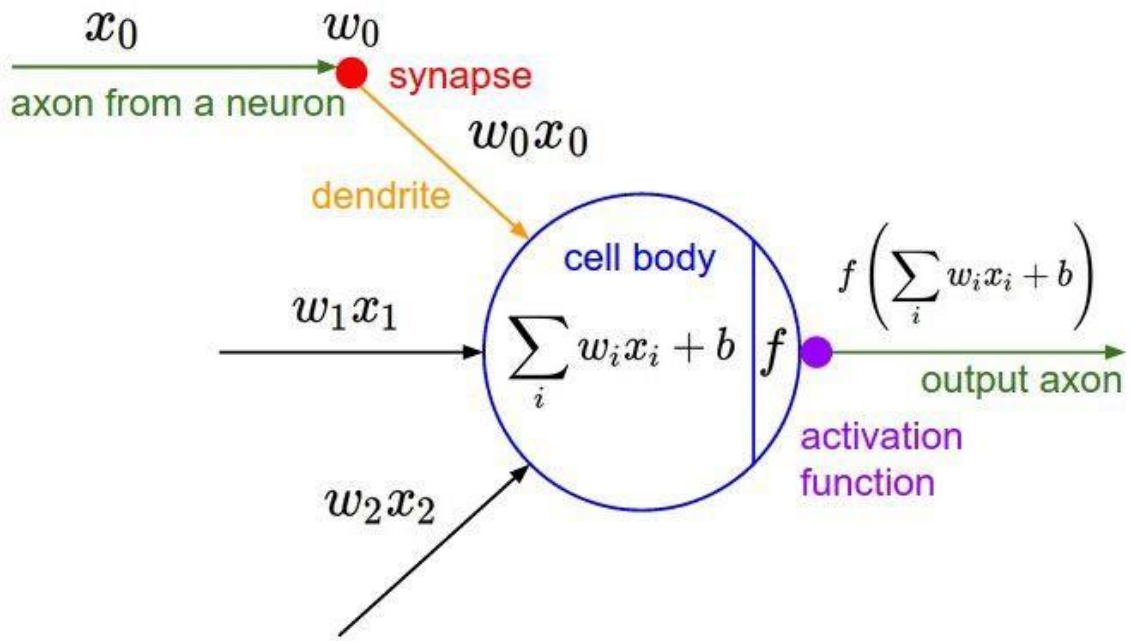
– Tín hiệu đầu vào thứ ba là x_2.

Activation functions



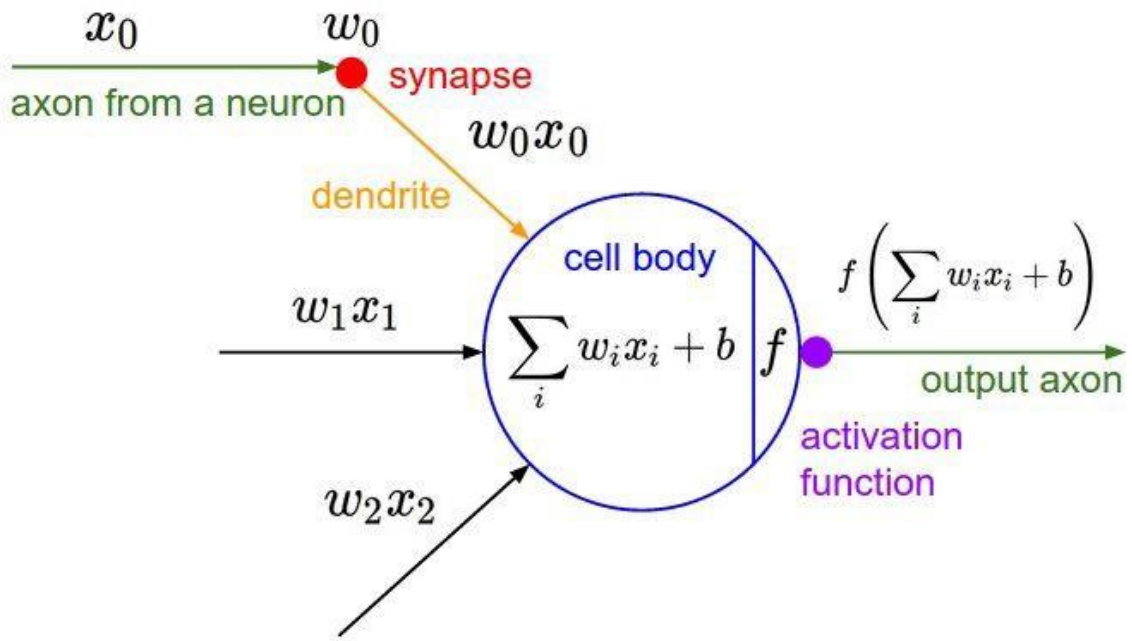
– Các tín hiệu đầu vào có trọng số hay không?

Activation functions



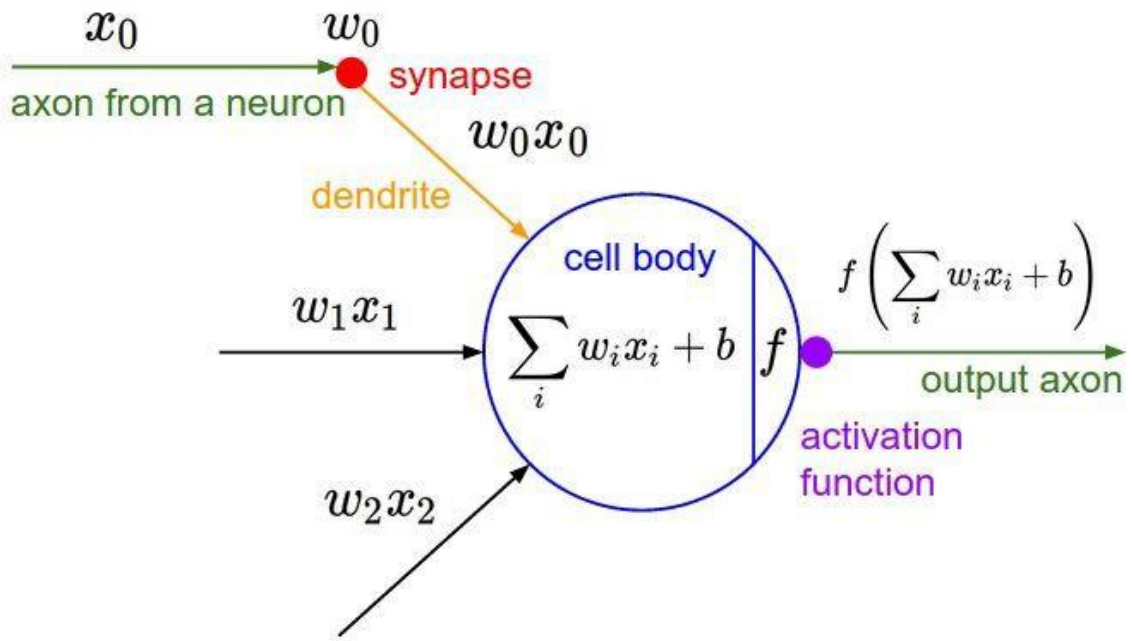
- Các tín hiệu đầu vào có trọng số hay không?
- Các tín hiệu đầu vào có trọng số.

Activation functions



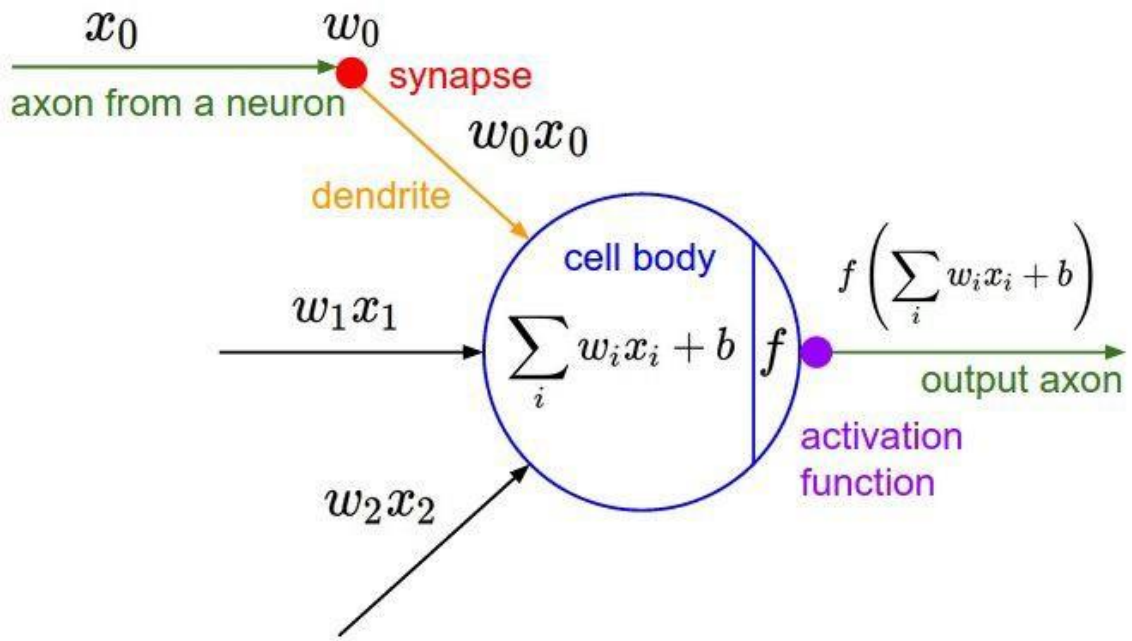
- Trọng số của tín hiệu đầu vào thứ nhất là gì?
- Trọng số của tín hiệu đầu vào thứ nhất là w_0.

Activation functions



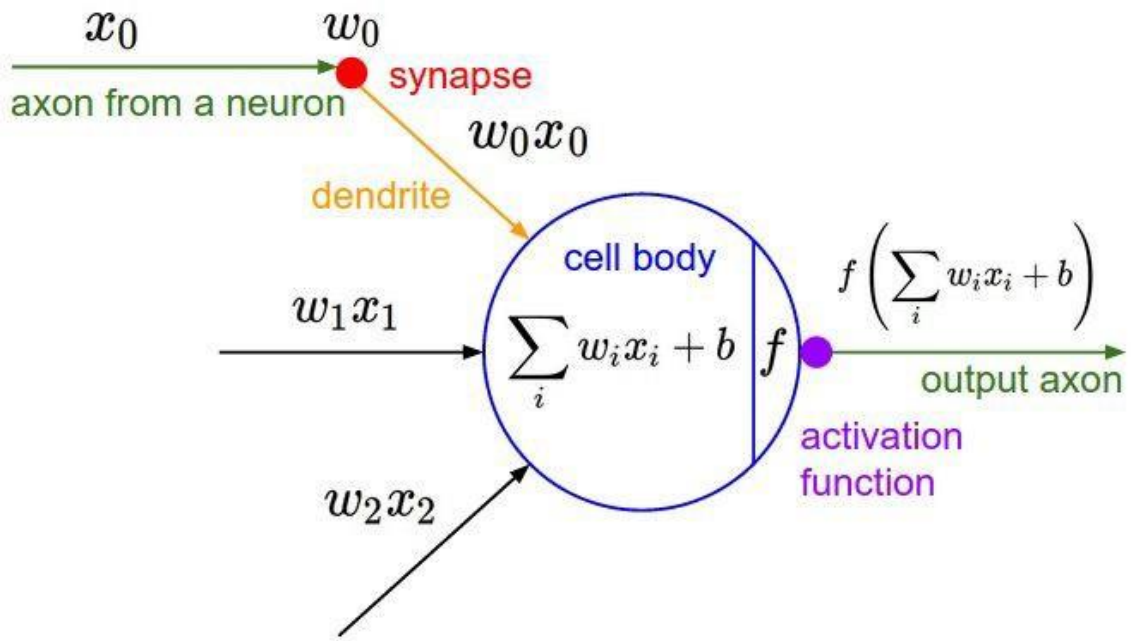
- Trọng số của tín hiệu đầu vào thứ hai là gì?
- Trọng số của tín hiệu đầu vào thứ hai là w_1.

Activation functions



- Trọng số của tín hiệu đầu vào thứ ba là gì?
- Trọng số của tín hiệu đầu vào thứ ba là w_2.

Activation functions

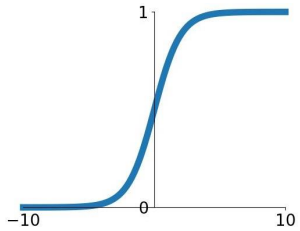


- Hàm activation của nơ ron có tên là gì?
- Hàm activation của nơ ron có tên là f.

Activation function

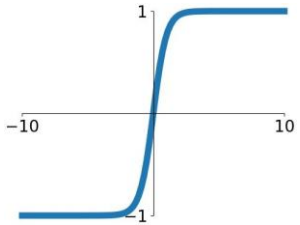
– Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



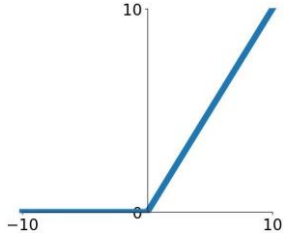
– tanh

$$tanh(x)$$



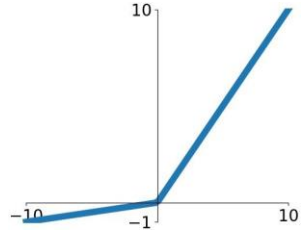
– ReLU

$$max(0, x)$$



– Leaky ReLU

$$max(0.001x, x)$$

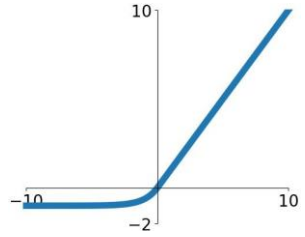


– Maxout

$$max(w_1^T x + b_1, w_2^T x + b_2)$$

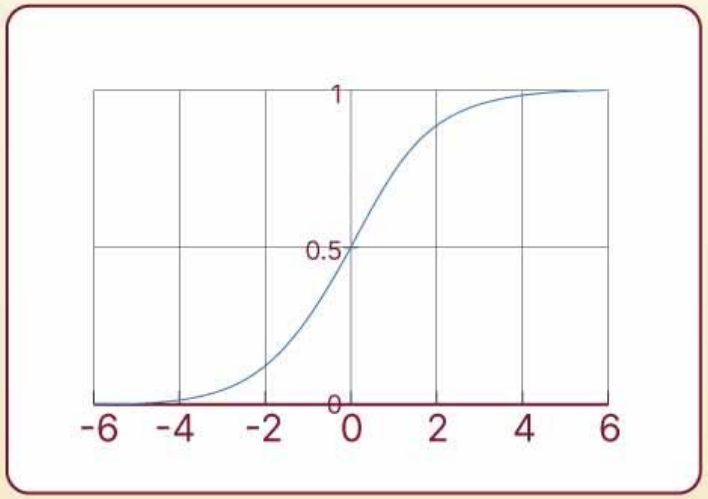
– ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid Activation functions

What is
Sigmoid Function?

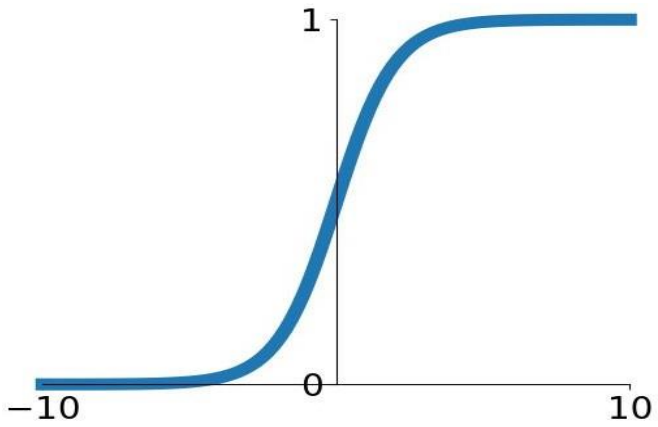


Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



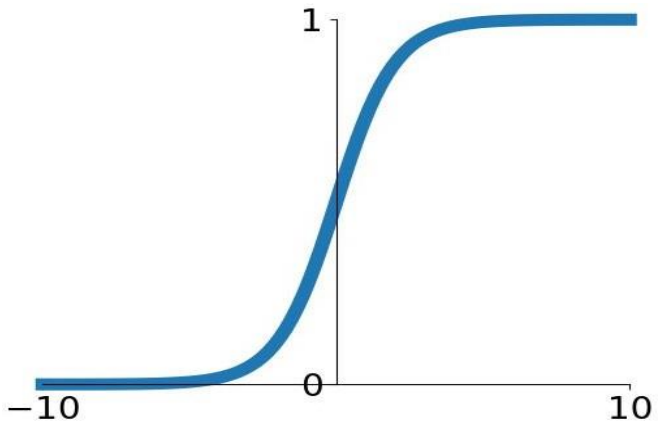
- Hàm Sigmoid là một trong những hàm kích hoạt phổ biến nhất trong mạng nơ-ron, đặc biệt là trong các mạng nơ-ron truyền thống và các mô hình logistic regression.
- Hàm Sigmoid biến đổi giá trị đầu vào thành một giá trị nằm trong khoảng từ 0 đến 1, khiến nó rất hữu ích cho các bài toán phân loại nhị phân.

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



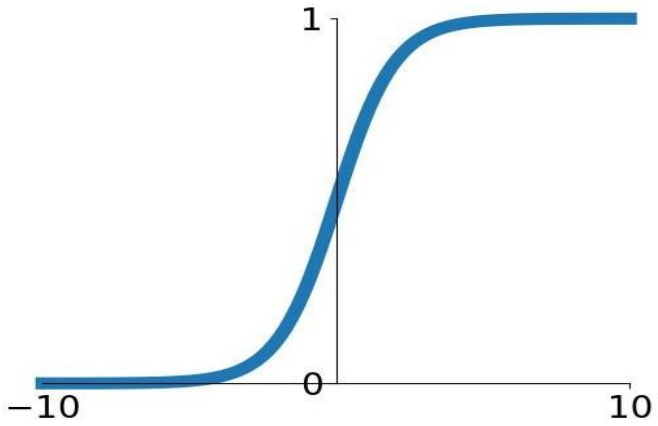
- Khi $x = 0$ giá trị của hàm Sigmoid là bao nhiêu?

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



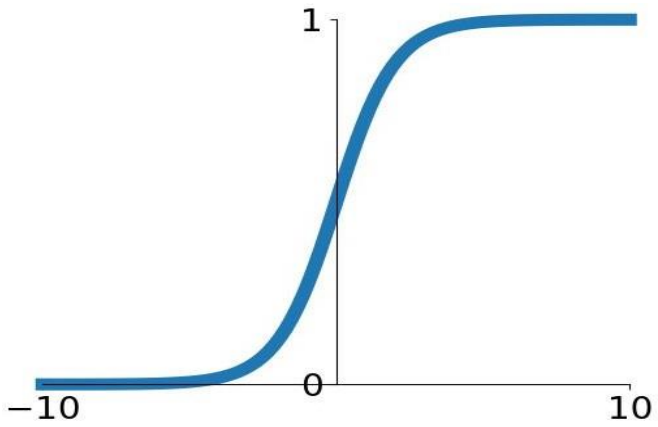
- Khi $x = 0$ giá trị của hàm Sigmoid là bao nhiêu?
- Khi $x = 0$ giá trị của hàm Sigmoid là $\frac{1}{1+e^0} = \frac{1}{1+1} = \frac{1}{2} = 0.5$

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



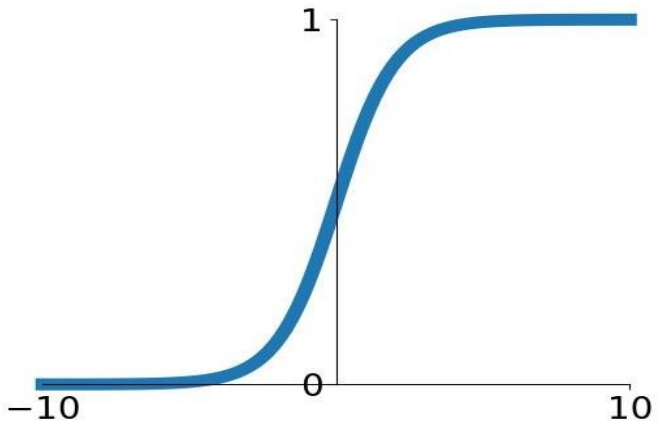
- Khi $x \to +\infty$ giá trị của hàm Sigmoid tiến tới giá trị mấy?

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



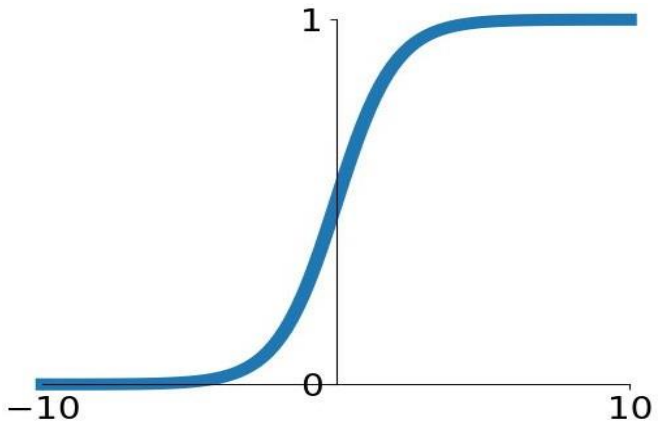
- Khi $x \to +\infty$ giá trị của hàm Sigmoid tiến tới giá trị mấy?
- Khi $x \to +\infty$ giá trị của hàm Sigmoid tiến tới giá trị 1.

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



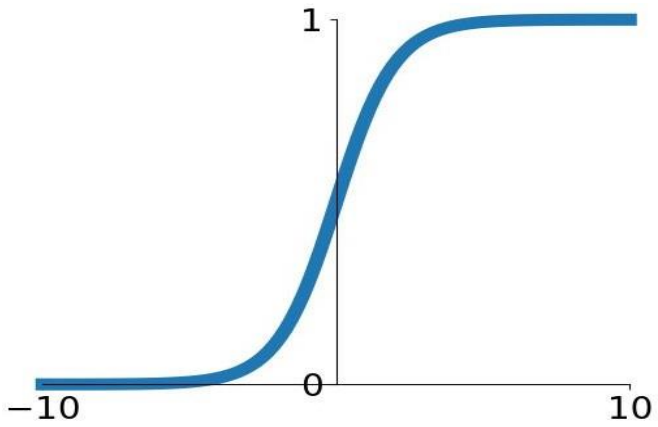
- Khi $x \to -\infty$ giá trị của hàm Sigmoid tiến tới giá trị mấy?

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



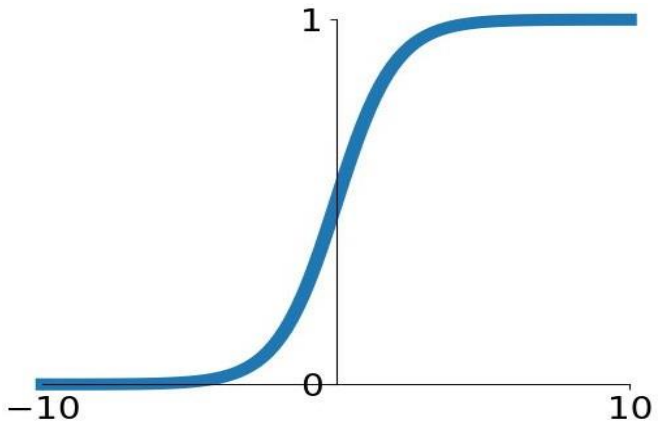
- Khi $x \to -\infty$ giá trị của hàm Sigmoid tiến tới giá trị mấy?
- Khi $x \to -\infty$ giá trị của hàm Sigmoid tiến tới giá trị 0.

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



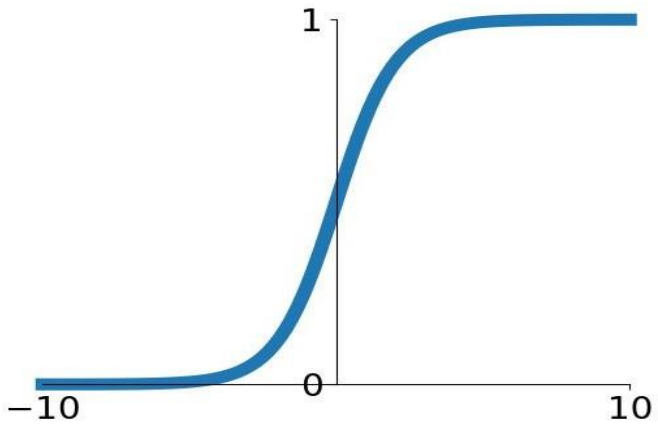
- Hàm Sigmoid có trục đối xứng hay không?
- Hàm Sigmoid không có trục đối xứng.

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



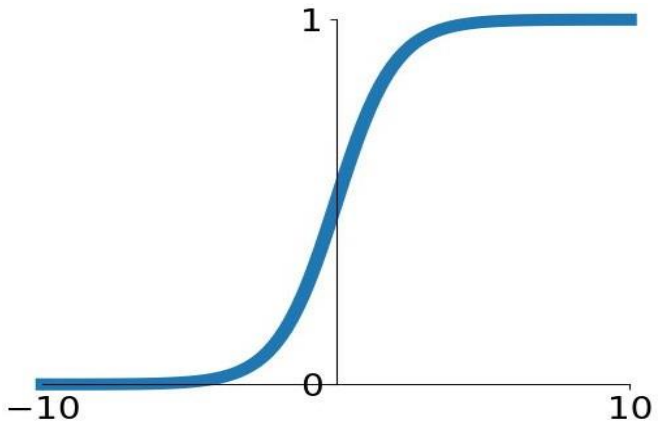
- Hàm Sigmoid có tâm đối xứng hay không?
- Hàm Sigmoid có tâm đối xứng.

Sigmoid Activation functions

- Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



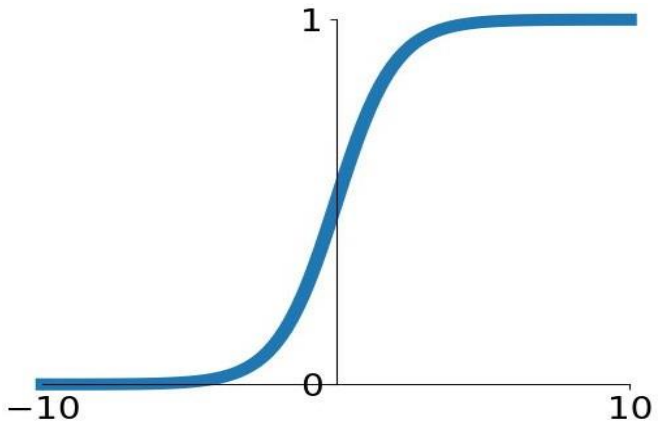
- Tâm đối xứng của hàm Sigmoid ở đâu?
- Tâm đối xứng của hàm Sigmoid là $\left(0, \frac{1}{2}\right)$.

Sigmoid Activation functions

– Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

– The logistic curve



– Đặc điểm của hàm Sigmoid:

+ Giá trị đầu ra: Giá trị của hàm Sigmoid luôn nằm trong khoảng từ 0 đến 1.

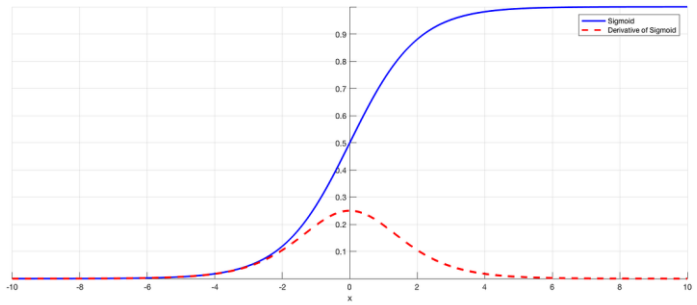
+ Tính liên tục và trơn tru: Hàm Sigmoid là một hàm liên tục và có đạo hàm mọi nơi, giúp cho việc tính toán gradient trong quá trình huấn luyện trở nên dễ dàng.

Sigmoid Activation functions

– Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

– Gradient of sigmoid function curve



– Đặc điểm của hàm Sigmoid:

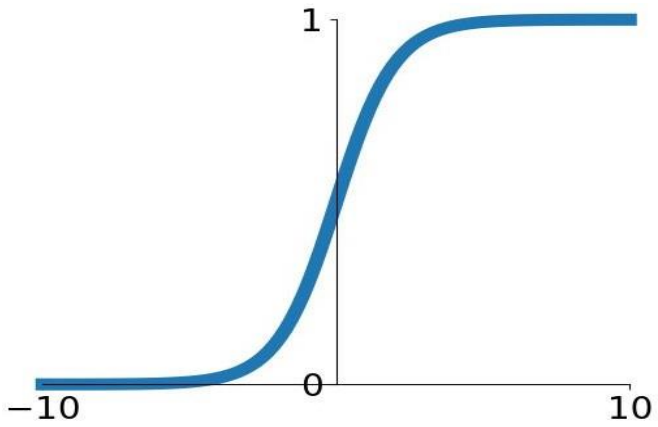
+ Gradient vanishing: Đối với các giá trị đầu vào rất lớn hoặc rất nhỏ, gradient của hàm Sigmoid trở nên rất nhỏ, gây ra vấn đề gradient vanishing trong quá trình huấn luyện mạng nơ-ron sâu.

Sigmoid Activation functions

– Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

– The logistic curve



– Lợi ích:

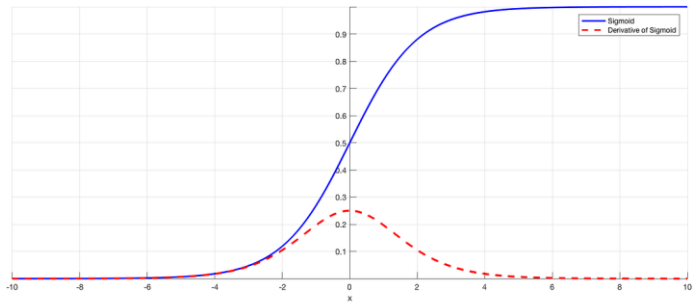
+ Giá trị đầu ra nằm trong khoảng (0,1): Điều này làm cho hàm Sigmoid trở thành lựa chọn phù hợp cho các bài toán phân loại nhị phân, nơi đầu ra cần biểu thị xác suất.

Sigmoid Activation functions

– Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

– Gradient of sigmoid function curve



– Lợi ích:

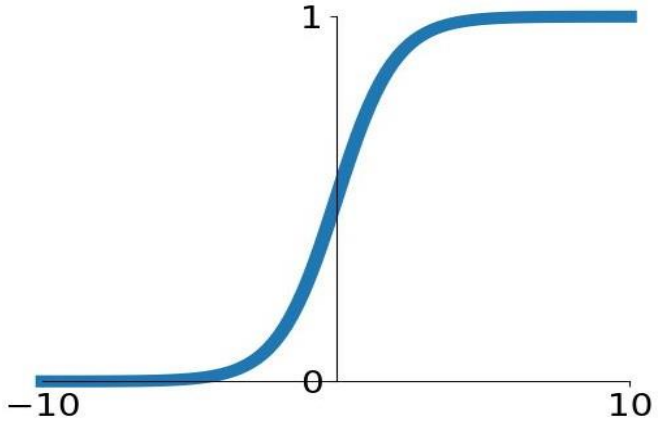
+ Đạo hàm đơn giản: Đạo hàm của hàm Sigmoid có thể được biểu diễn đơn giản bằng chính giá trị của nó, $\sigma'(x) = \sigma(x)\big(1 - \sigma(x)\big)$ giúp quá trình tính toán gradient hiệu quả hơn.

Sigmoid Activation functions

– Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

– The logistic curve



– Nhược điểm:

+ Gradient vanishing: Khi giá trị đầu vào rất lớn hoặc rất nhỏ, gradient của hàm Sigmoid trở nên rất nhỏ, làm chậm quá trình học của mạng.

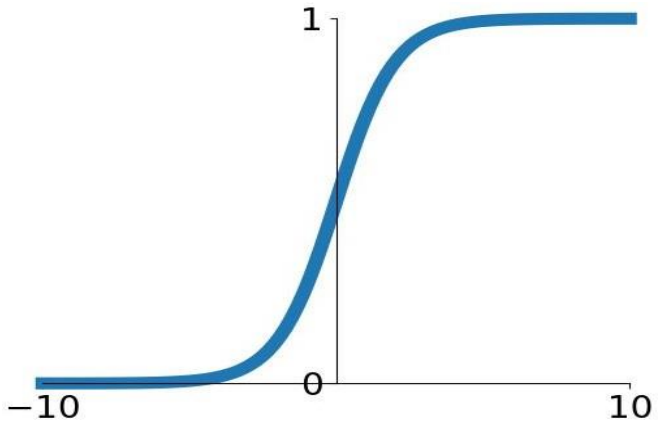
+ Không zero-centered: Giá trị đầu ra của hàm Sigmoid luôn dương, gây ra độ lệch trong quá trình cập nhật trọng số.

Sigmoid Activation functions

– Sigmoid

$$Sigmoid(x) = \frac{1}{1 + e^{-x}}$$

– The logistic curve



– Ứng dụng:

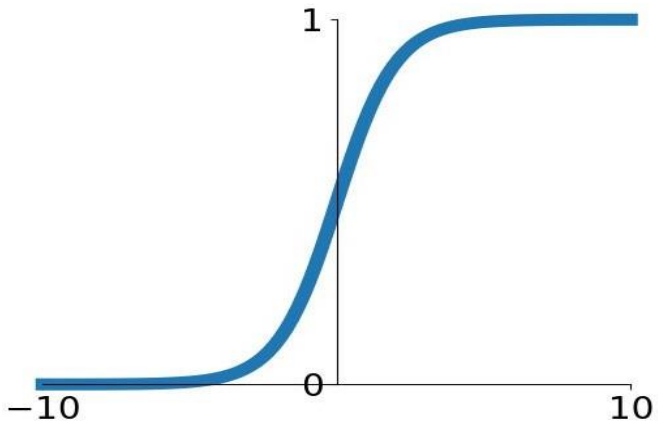
+ Hàm Sigmoid thường được sử dụng trong các lớp đầu ra của các mạng nơ-ron cho các bài toán phân loại nhị phân và trong các mô hình logistic regression.

Sigmoid Activation functions

- Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



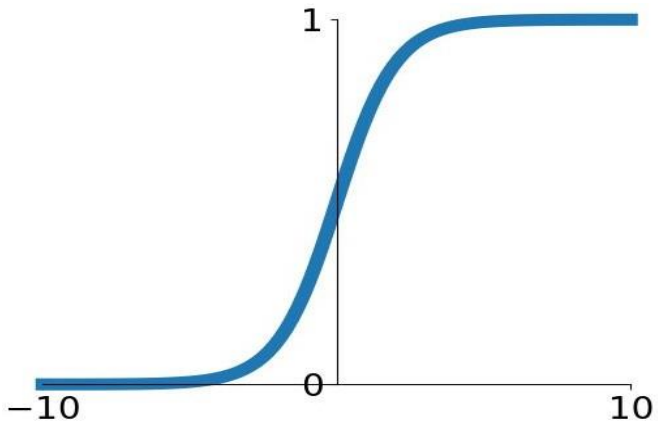
- Squashes numbers to range $[0, 1]$.
- Historically popular since they have nice interpretation as a saturating "firing rate" of a neuron.

Sigmoid Activation functions

- Sigmoid

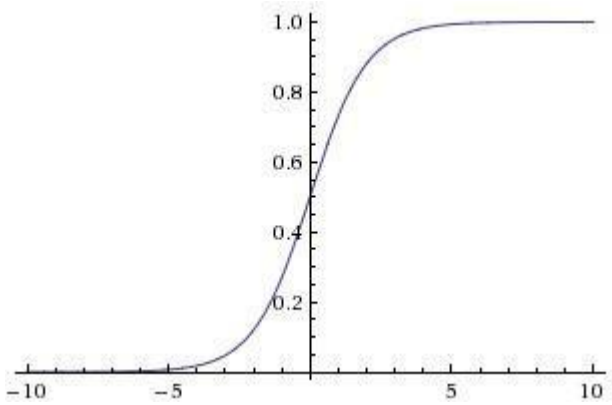
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve

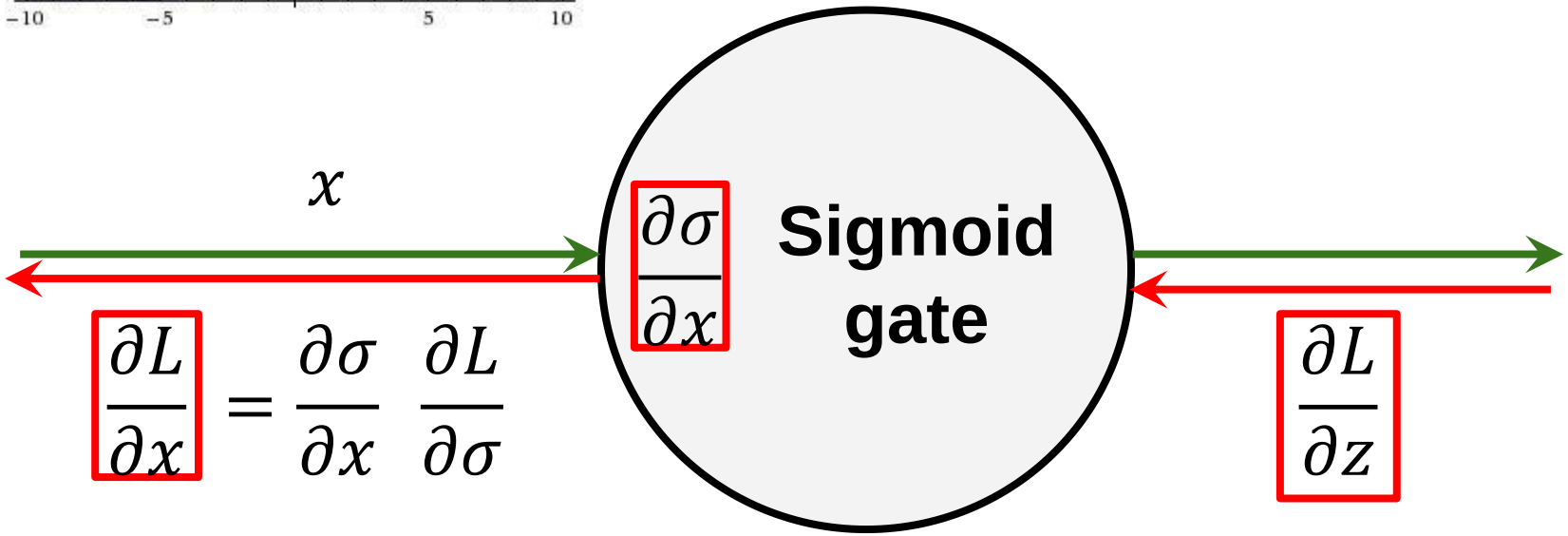


- 3 problem:
  + Saturated neurons “kill” the gradients.

Sigmoid Activation functions



- What happens when $x = -10$?
- What happens when $x = 0$?
- What happens when $x = 10$?

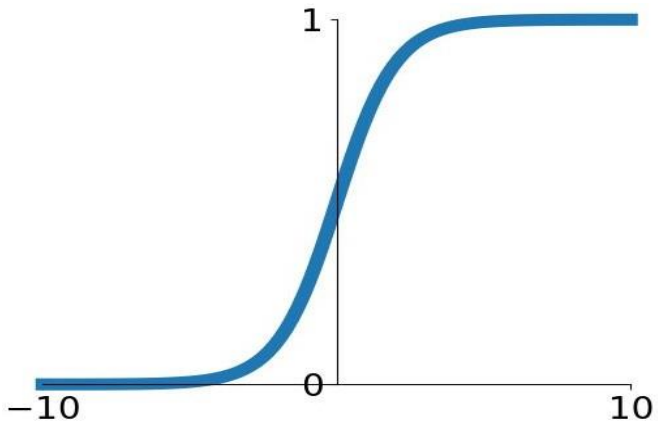


Sigmoid Activation functions

- Sigmoid

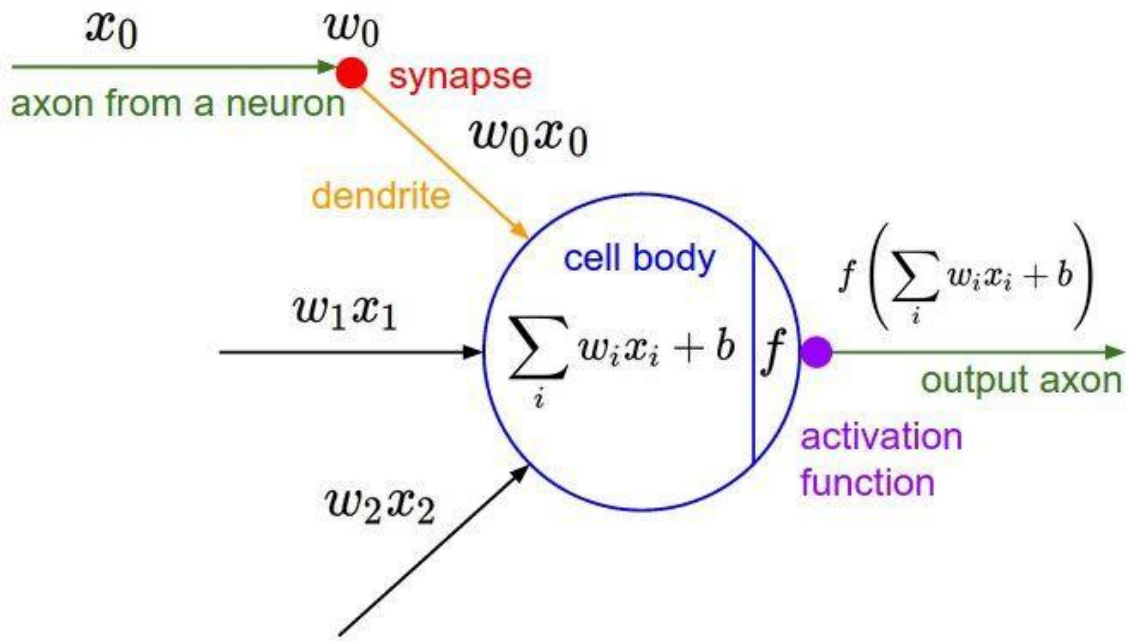
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



- 3 problem:
  + Saturated neurons “kill” the gradients.
  + Sigmoid outputs are not zero-centered.

Sigmoid Activation functions



- Consider what happens when the input to a neuron (x) is always positive:

$$f\left(\sum_i w_i x_i + b\right)$$

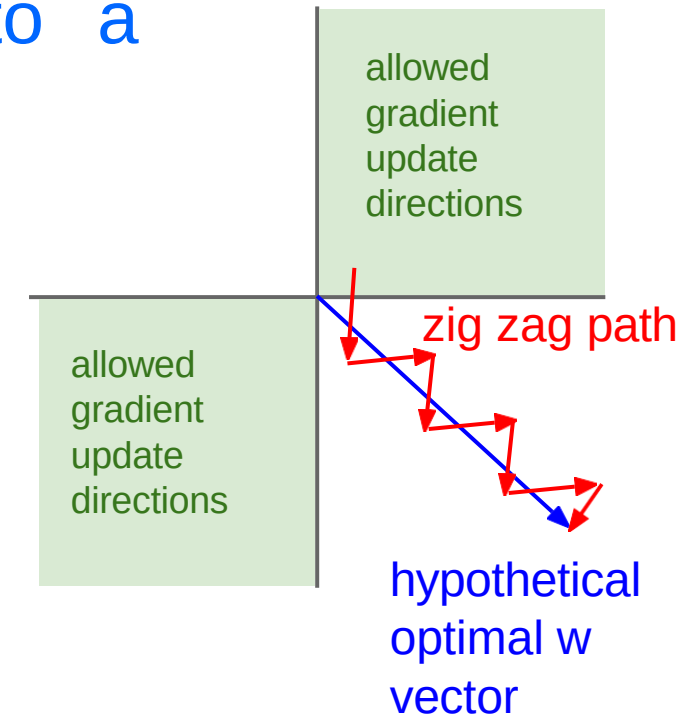
- What can we say about the gradients on **w**?

Sigmoid Activation functions

- Consider what happens when the input to a neuron (x) is always positive:

$$f\left(\sum_i w_i x_i + b\right)$$

- What can we say about the gradients on **w**?
- Always all positive or all negative :(
- (this is also why you want zero-mean data!)

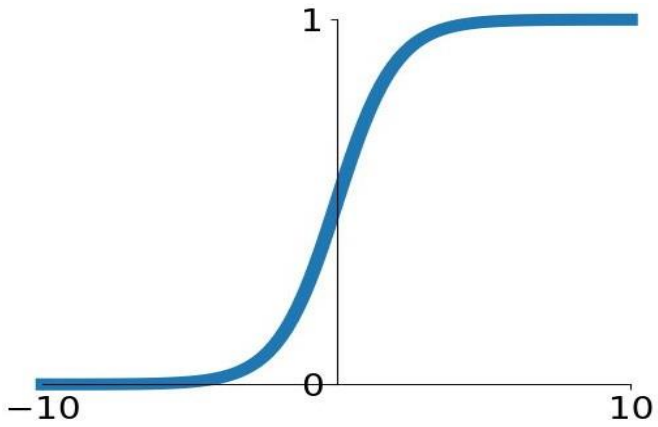


Sigmoid Activation functions

- Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- The logistic curve



- 3 problem:
  + Saturated neurons “kill” the gradients.
  + Sigmoid outputs are not zero-centered.
  + exp() is a bit compute expensive.

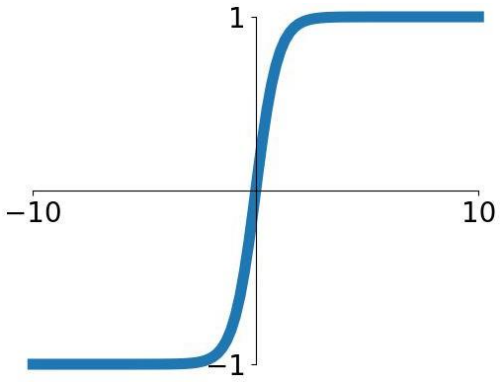
tanh Activation functions

tanh Activation functions

- Hyperbolic tangent function – tanh function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



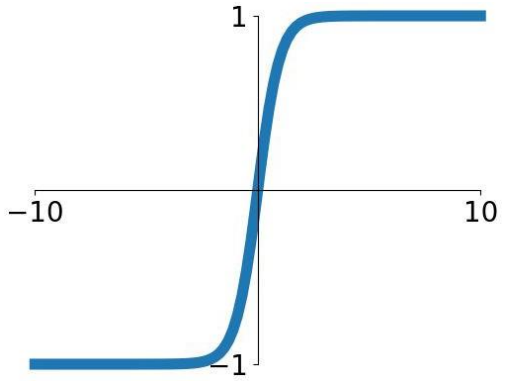
- Hàm Tanh (hyperbolic tangent function) là một hàm kích hoạt thường được sử dụng trong các mạng nơ-ron.
- Hàm Tanh (hyperbolic tangent function) chuyển đổi các giá trị đầu vào thành một giá trị nằm giữa −1 và 1.

tanh Activation functions

- Hyperbolic tangent function – tanh function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



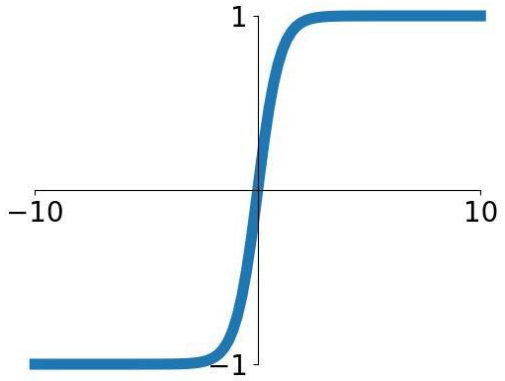
- Đặc điểm của hàm tanh:
  + Giá trị đầu ra: Giá trị của hàm Tanh nằm trong khoảng từ -1 đến 1.
  + Đối xứng gốc (zero-centered): Điều này có nghĩa là giá trị trung bình của đầu ra gần bằng 0, giúp quá trình huấn luyện của mạng nơ-ron dễ dàng hơn so với hàm Sigmoid, vì nó giúp tránh tình trạng gradient vanishing.

tanh Activation functions

- Hyperbolic tangent function – tanh function:

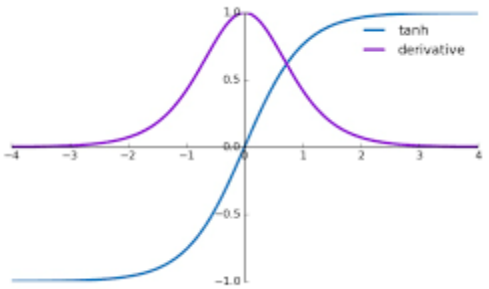
$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



- Đặc điểm của hàm tanh:
  + Gradient: Gradient của hàm Tanh lớn nhất ở gần 0 và giảm dần khi đầu vào trở nên lớn hơn hoặc nhỏ hơn.

- Gradient of tanh function:

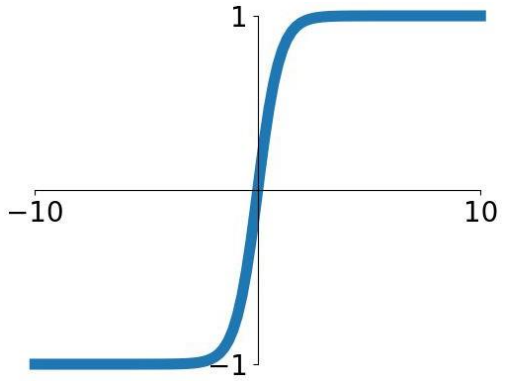


tanh Activation functions

- Hyperbolic tangent function – tanh function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



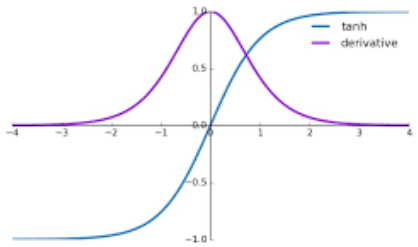
- Lợi ích:
  + Zero-centered: Đầu ra của hàm Tanh có giá trị trung bình gần bằng 0, giúp giảm thiểu sự lệch lạc trong gradient và tăng tốc độ huấn luyện.
  + Gradient lớn hơn so với hàm Sigmoid, hàm Tanh có gradient lớn hơn, giúp tránh tình trạng gradient vanishing trong các lớp sâu.

tanh Activation functions

- Hyperbolic tangent function – tanh function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- Gradient of tanh function:



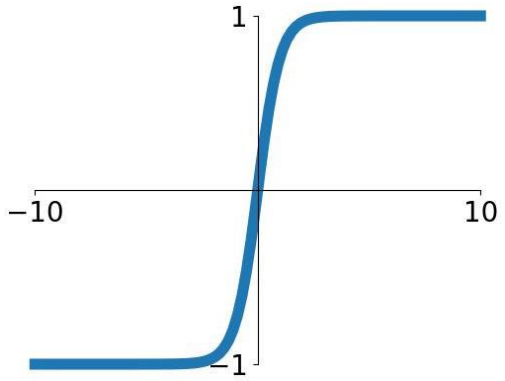
- Nhược điểm:
  + Gradient Vanishing: Khi đầu vào của hàm Tanh nằm ở các giá trị rất lớn hoặc rất nhỏ, gradient của nó sẽ trở nên rất nhỏ. Điều này làm cho các lớp phía dưới của mạng (gần với đầu vào) học chậm hơn nhiều so với các lớp phía trên, dẫn đến hiệu suất kém của toàn bộ mạng nơ-ron.

tanh Activation functions

- Hyperbolic tangent function – tanh function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



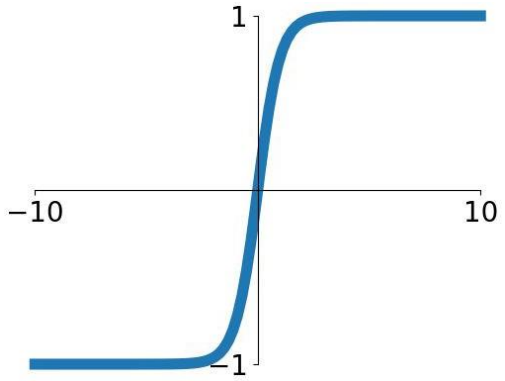
- Nhược điểm:
  + Độ phức tạp tính toán cao hơn ReLU: Hàm Tanh tính toán dựa trên hàm mũ, do đó độ phức tạp tính toán của nó cao hơn so với các hàm kích hoạt đơn giản như ReLU.

tanh Activation functions

- Hyperbolic tangent function – tanh function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



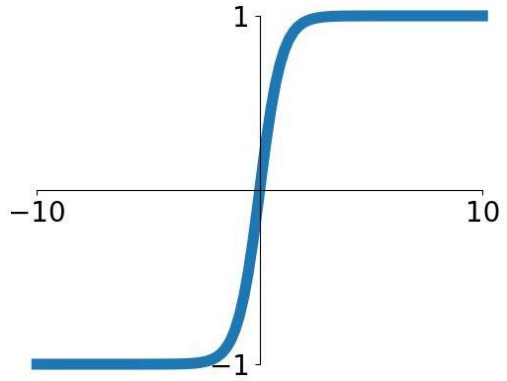
- Nhược điểm:
  + Không phù hợp với tất cả các loại dữ liệu: Mặc dù hàm Tanh hoạt động tốt cho các bài toán mà dữ liệu được phân phối quanh zero, nhưng nó có thể không phù hợp cho tất cả các loại dữ liệu. Ví dụ, dữ liệu có giá trị trung bình không quanh zero có thể không được xử lý tốt bởi hàm Tanh.

tanh Activation functions

- Hyperbolic tangent function – tanh function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



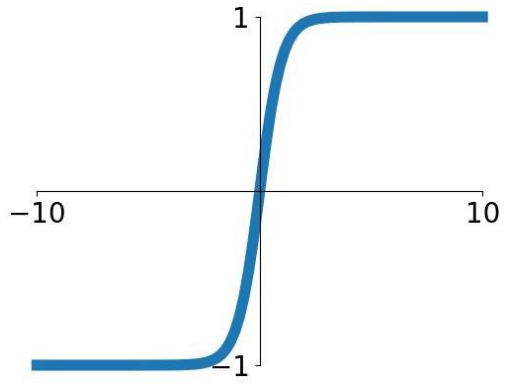
- Nhược điểm:
  + Lạm dụng hàm Tanh có thể dẫn đến overfitting: Nếu không được sử dụng đúng cách, việc lạm dụng hàm Tanh có thể dẫn đến overfitting, đặc biệt khi không có đủ regularization hoặc dữ liệu huấn luyện.

tanh Activation functions

- Hyperbolic tangent function – tanh function:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



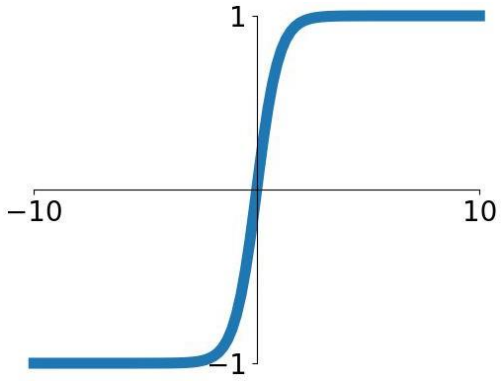
- Ứng dụng:
  + Hàm Tanh được sử dụng rộng rãi trong các lớp ẩn của mạng nơ-ron, đặc biệt trong các mô hình RNN (Recurrent Neural Networks).

tanh Activation functions

- tanh function (hyperbolic tangent function)

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

- The tanh curve



- Squashes numbers to range $[-1,1]$
- zero centered (nice)
- still kills gradients when saturated :(.

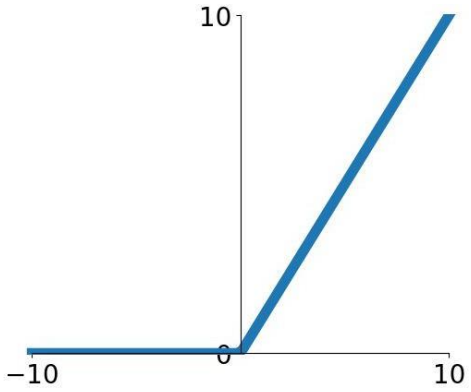
ReLU Activation functions

ReLU Activation functions

- $ReLU$ function (Rectified Linear Unit)

$$ReLU(x) = max(0, x)$$

- The $ReLU$ curve



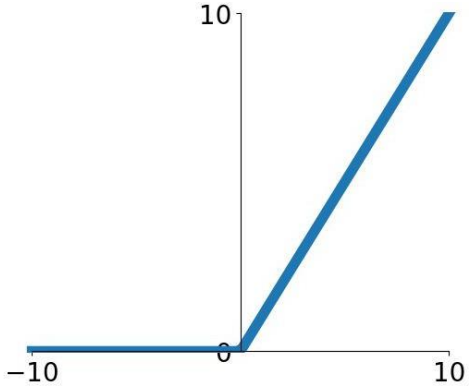
- Hàm $ReLU$ (Rectified Linear Unit) là một trong những hàm kích hoạt phổ biến nhất được sử dụng trong mạng nơ-ron sâu.
- Hàm $ReLU$ chuyển đổi các giá trị đầu vào thành giá trị đầu ra bằng cách giữ nguyên các giá trị dương và thay thế các giá trị âm bằng giá trị 0.

ReLU Activation functions

- $ReLU$ function (Rectified Linear Unit)

$$ReLU(x) = max(0, x)$$

- The $ReLU$ curve



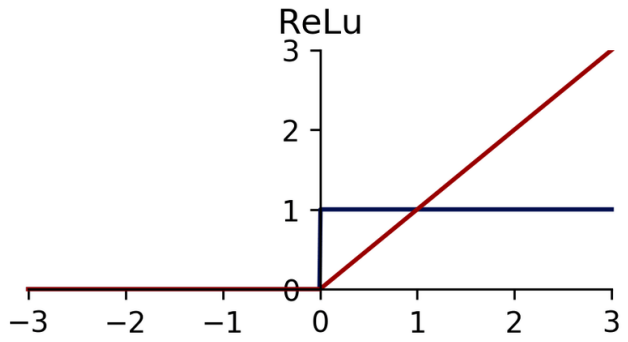
- Đặc điểm của hàm $ReLU$:
  - + Giá trị đầu ra: Giá trị của hàm $ReLU$ là 0 nếu đầu vào nhỏ hơn hoặc bằng 0, và bằng chính giá trị đầu vào nếu lớn hơn 0.
  - + Tính đơn giản: Hàm $ReLU$ rất đơn giản và dễ tính toán, giúp tăng tốc độ huấn luyện của mạng nơ-ron.

ReLU Activation functions

- $ReLU$ function (Rectified Linear Unit)

$$ReLU(x) = max(0, x)$$

- Gradient of $ReLU$ function:



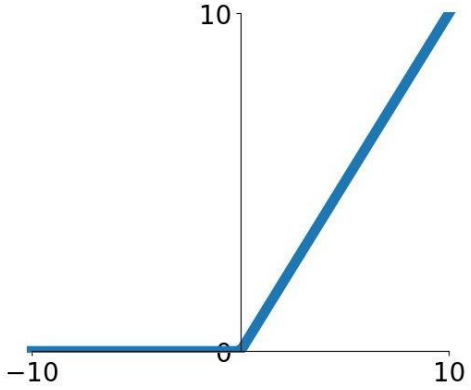
- Đặc điểm của hàm ReLU:
  + Giải quyết vấn đề gradient vanishing: So với các hàm kích hoạt khác như Sigmoid và Tanh, hàm $ReLU$ giúp giải quyết vấn đề gradient vanishing, bởi vì gradient của $ReLU$ là 1 đối với các giá trị đầu vào dương.

ReLU Activation functions

- $ReLU$ function (Rectified Linear Unit)

  $ReLU(x) = max(0, x)$

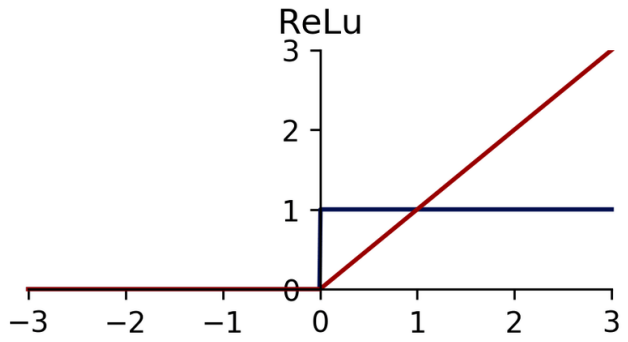
- The $ReLU$ curve



- Lợi ích:
  + Tăng tốc độ hội tụ: Hàm $ReLU$ giúp tăng tốc độ hội tụ của các thuật toán tối ưu hóa, làm cho quá trình huấn luyện nhanh hơn.
  + Đơn giản và hiệu quả: Hàm $ReLU$ rất đơn giản và hiệu quả trong việc tính toán, giúp giảm thời gian tính toán và tài nguyên cần thiết.

ReLU Activation functions

- $ReLU$ function (Rectified Linear Unit)
  $ReLU(x) = max(0, x)$
- Gradient of $ReLU$ function:



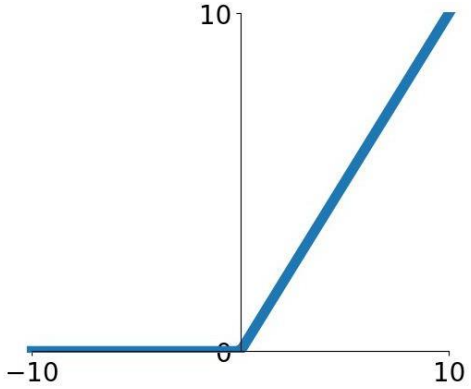
- Nhược điểm:
  + Dead neurons: Một số nơ-ron có thể "chết" trong quá trình huấn luyện nếu đầu vào của chúng luôn âm, dẫn đến gradient bằng 0 và không cập nhật được. Vấn đề này có thể được khắc phục bằng các biến thể của $ReLU$ như $Leaky\ ReLU$ hoặc $Parametric\ ReLU$.

ReLU Activation functions

- $ReLU$ function (Rectified Linear Unit)

  $ReLU(x) = max(0, x)$

- The $ReLU$ curve



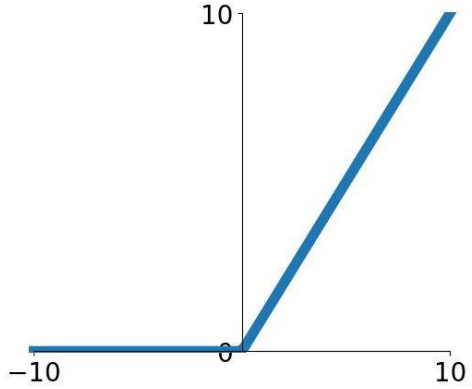
- Ứng dụng:
  + Hàm $ReLU$ được sử dụng rộng rãi trong các lớp ẩn của mạng nơ-ron, đặc biệt là trong các mô hình CNN (Convolutional Neural Networks).

ReLU Activation functions

- $ReLU$ function (Rectified Linear Unit)

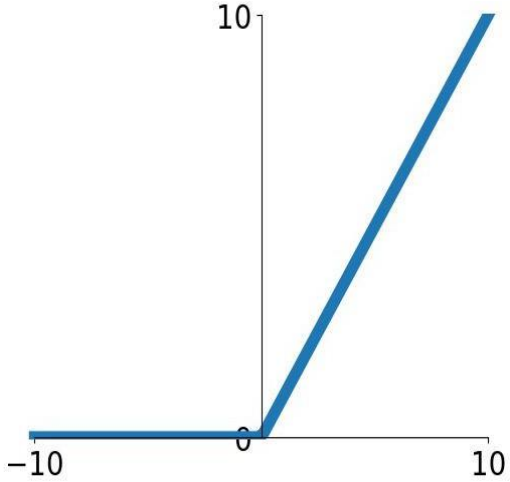
$$ReLU(x) = max(0, x)$$

- The $ReLU$ curve

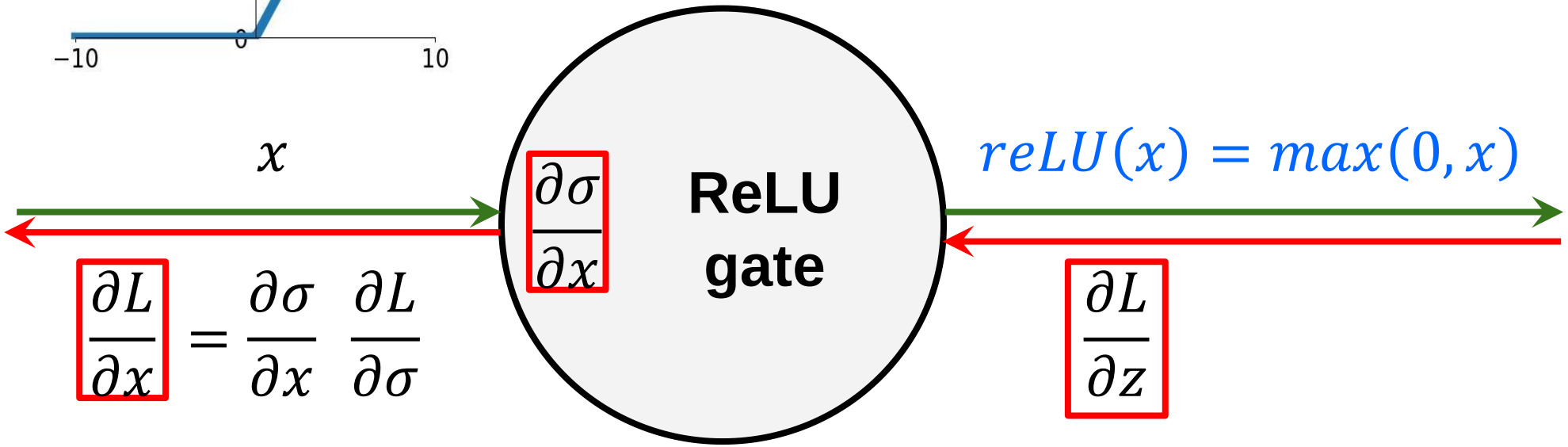


- Does not saturate (in +region).
- Very computationally efficient.
- Converges much faster than sigmoid/tanh in practice (e.g. 6x).
- Actually more biologically plausible than sigmoid.

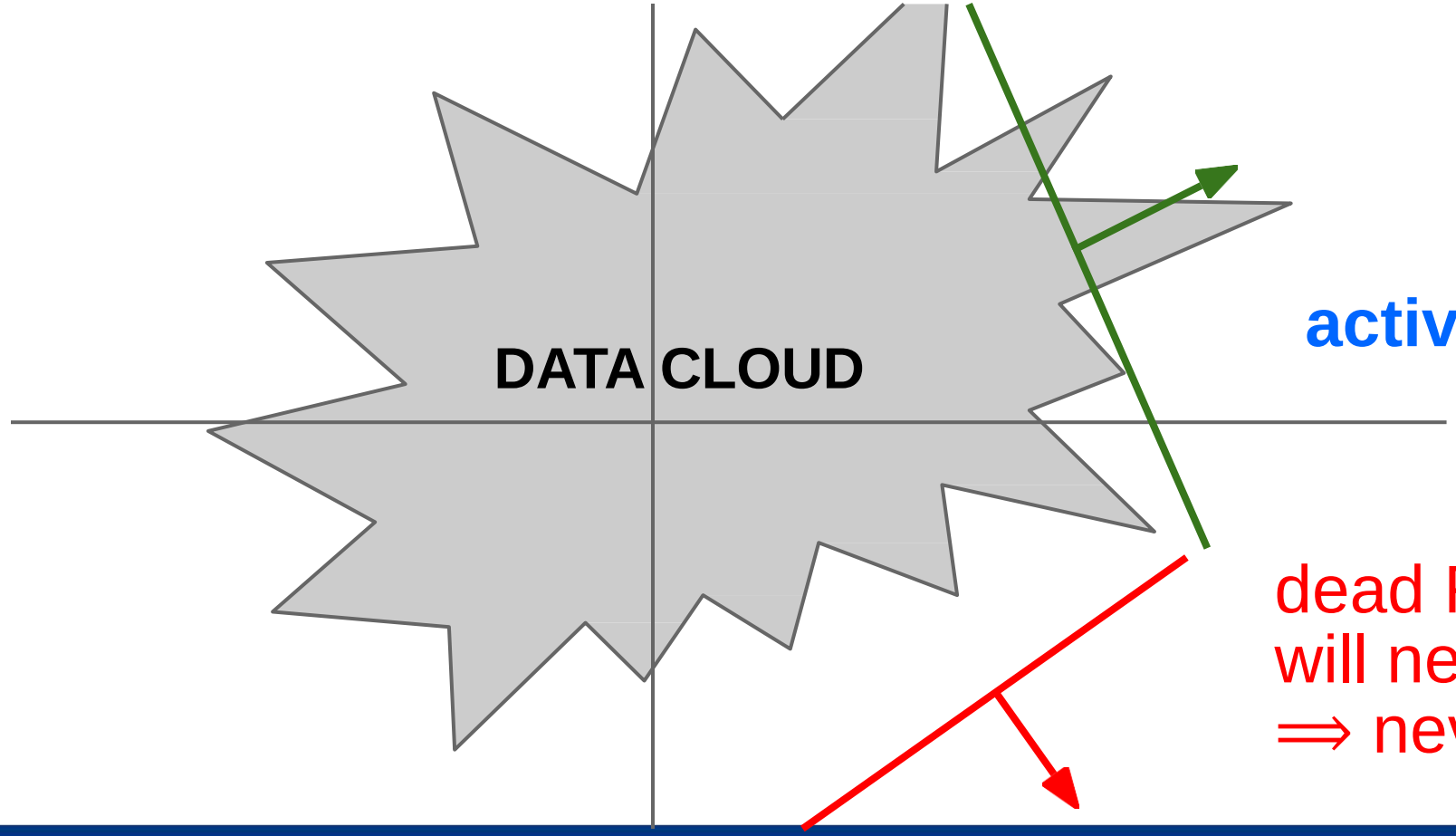
ReLU Activation functions



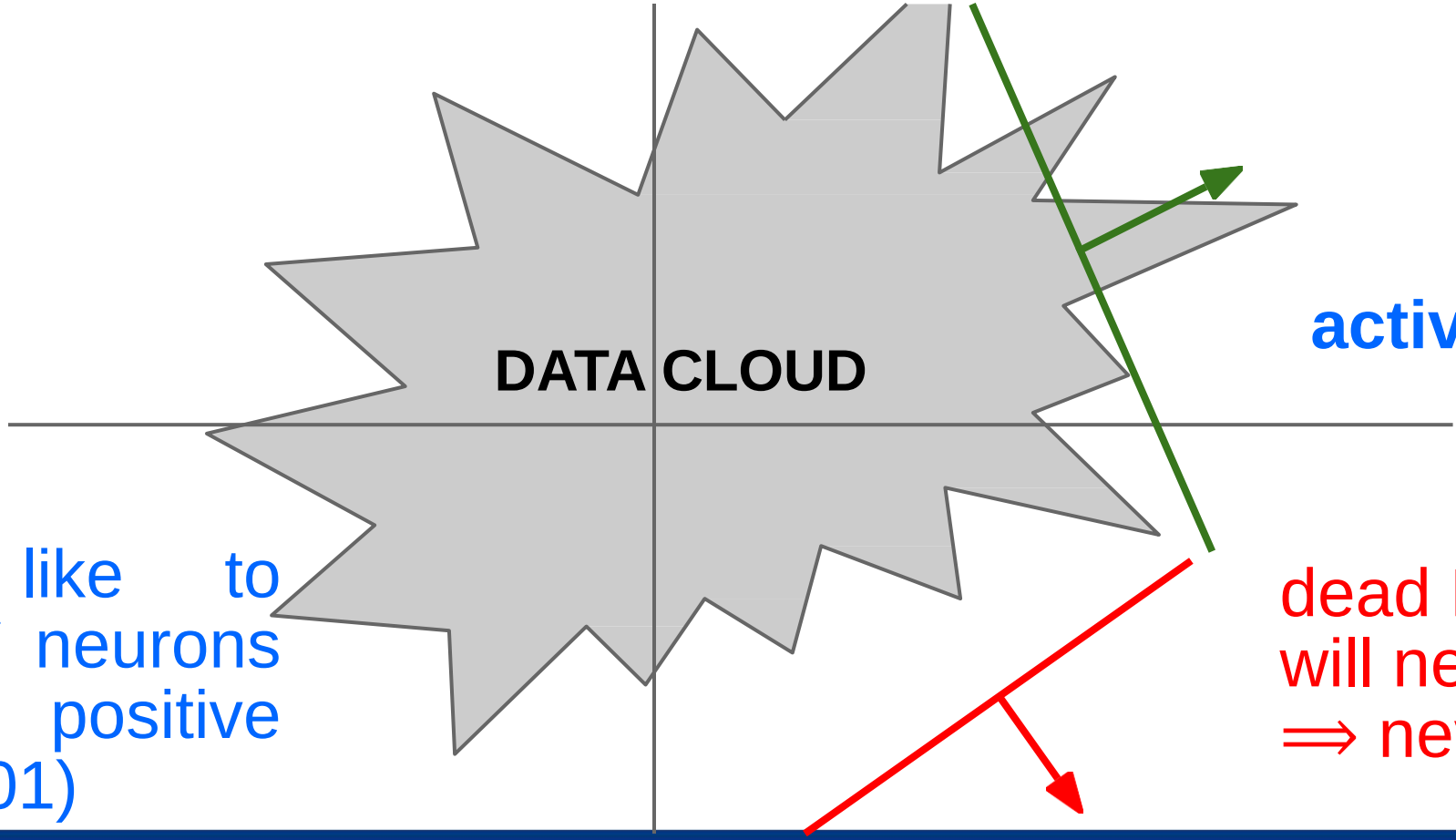
- What happens when $x = -10$?
- What happens when $x = 0$?
- What happens when $x = 10$?



ReLU Activation functions



ReLU Activation functions



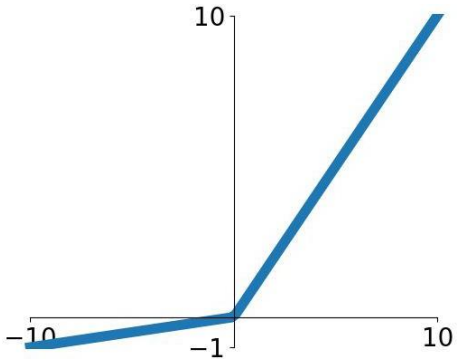
⇒ people like to initialize $ReLU$ neurons with slightly positive biases (e.g. 0.01)

Leaky ReLU Activation functions

- Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

- The Leaky $ReLU$ curve



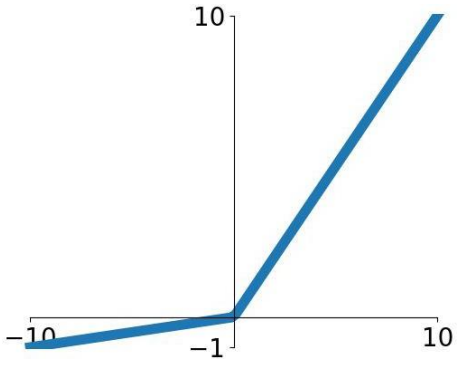
- Leaky $ReLU$ là một biến thể của hàm $ReLU$ nhằm khắc phục nhược điểm của $ReLU$, đó là "dead neurons" (nơ-ron chết).
- Trong Leaky $ReLU$, thay vì đưa các giá trị âm về giá trị 0 như trong $ReLU$, các giá trị âm được cho một độ dốc nhỏ, giúp nơ-ron tiếp tục học.

Leaky ReLU Activation functions

– Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

– The Leaky $ReLU$ curve



– Đặc điểm của Leaky $ReLU$:

+ Giải quyết vấn đề "dead neurons": Bằng cách cho phép gradient nhỏ chạy qua các giá trị âm, Leaky $ReLU$ giúp tránh tình trạng nơ-ron chết.

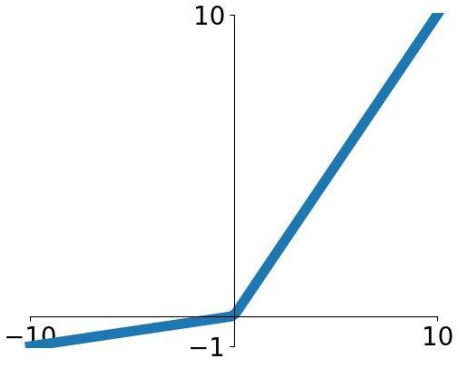
+ Giữ lại các ưu điểm của $ReLU$: Leaky $ReLU$ vẫn giữ lại tính đơn giản và hiệu quả của $ReLU$, bao gồm tính không tuyến tính và khả năng tăng tốc độ hội tụ của mô hình.

Leaky ReLU Activation functions

- Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

- The Leaky $ReLU$ curve



- Lợi ích:

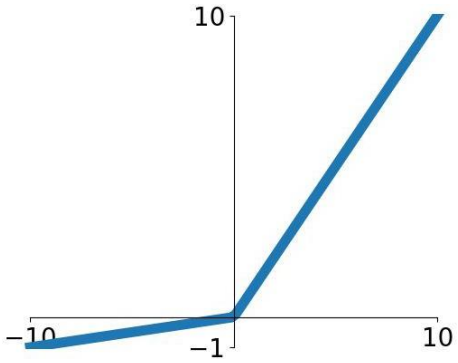
+ Giải quyết vấn đề "dead neurons": Trong $ReLU$, các giá trị âm của đầu vào bị đưa về 0, khiến các nơ-ron có thể "chết" nếu chúng không bao giờ kích hoạt (đầu vào luôn âm). $Leaky\ ReLU$ khắc phục vấn đề này bằng cách cho phép một độ dốc nhỏ (thường là 0.01) cho các giá trị âm, giúp các nơ-ron này tiếp tục học.

Leaky ReLU Activation functions

– Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

– The Leaky $ReLU$ curve



– Lợi ích:

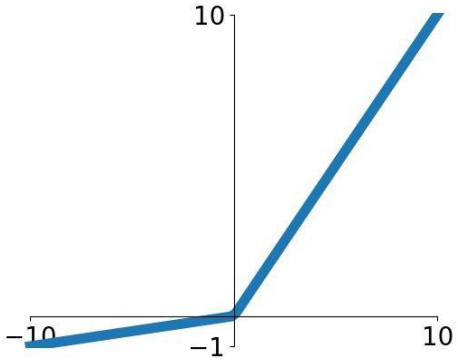
+ Giữ lại các ưu điểm của $ReLU$: $Leaky\ ReLU$ giữ lại các ưu điểm của $ReLU$ như tính không tuyến tính, đơn giản và dễ tính toán, và khả năng tăng tốc độ hội tụ của quá trình huấn luyện.

Leaky ReLU Activation functions

- Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

- The Leaky $ReLU$ curve



- Lợi ích:

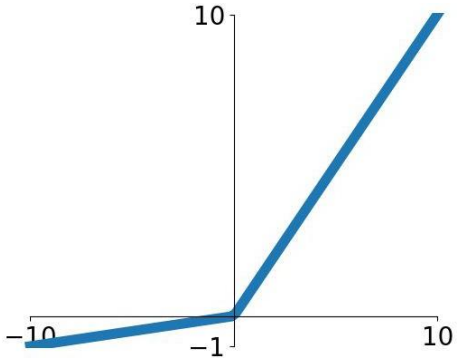
+ Tránh vấn đề gradient vanishing: Leaky $ReLU$ tránh vấn đề gradient vanishing vì độ dốc không bao giờ bằng 0, giúp mạng nơ-ron học nhanh hơn và hiệu quả hơn so với các hàm kích hoạt khác như Sigmoid hay Tanh.

Leaky ReLU Activation functions

- Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

- The Leaky $ReLU$ curve



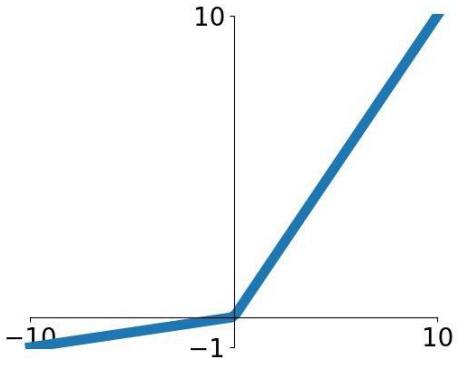
- Lợi ích:
  + Dễ dàng triển khai: Leaky $ReLU$ dễ dàng triển khai trong các thư viện học máy phổ biến như TensorFlow và PyTorch, và không yêu cầu thay đổi nhiều trong kiến trúc mạng hiện có.

Leaky ReLU Activation functions

- Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

- The Leaky $ReLU$ curve



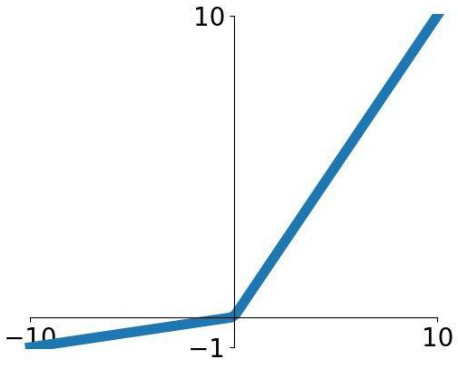
- Nhược điểm:
  + Tham số α cố định: Trong nhiều trường hợp, giá trị của α (độ dốc của phần âm) được cố định và không thay đổi trong quá trình huấn luyện. Việc này có thể không tối ưu cho mọi bài toán và mọi loại dữ liệu. Một biến thể khác như Parametric ReLU (PReLU) cho phép giá trị của α được học, giúp mô hình linh hoạt hơn.

Leaky ReLU Activation functions

– Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

– The Leaky $ReLU$ curve



– Nhược điểm:

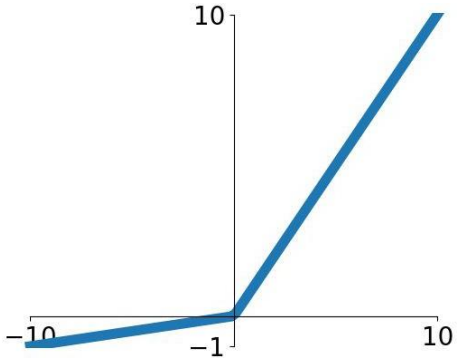
+ Không giải quyết được mọi vấn đề gradient vanishing: Mặc dù Leaky $ReLU$ giảm thiểu vấn đề gradient vanishing, nó không loại bỏ hoàn toàn vấn đề này trong các mạng nơ-ron cực kỳ sâu. Trong những trường hợp này, các kỹ thuật khác như Batch Normalization có thể cần thiết.

Leaky ReLU Activation functions

- Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

- The Leaky $ReLU$ curve



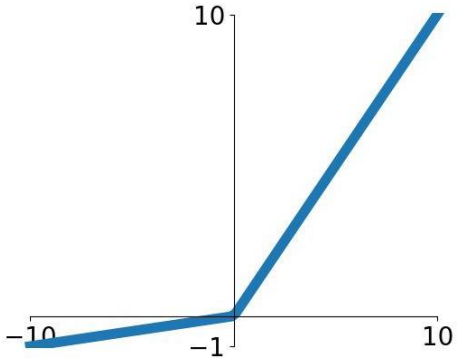
- Nhược điểm:
  + Có thể dẫn đến độ dốc không mong muốn: Với các giá trị âm lớn, độ dốc của Leaky $ReLU$ vẫn sẽ là α, có thể dẫn đến độ dốc không mong muốn và ảnh hưởng đến hiệu suất huấn luyện.

Leaky ReLU Activation functions

- Leaky $ReLU$ function

$$Leaky\ ReLU(x) = max(0.01x, x)$$

- The Leaky $ReLU$ curve



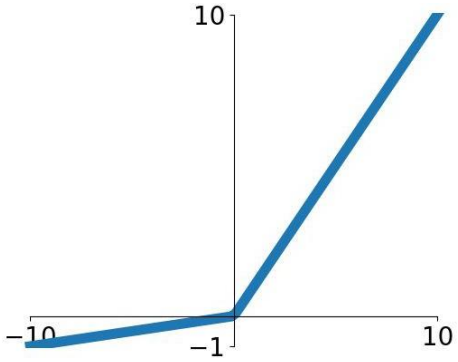
- Ứng dụng:
  + Leaky $ReLU$ thường được sử dụng trong các mạng nơ-ron sâu, đặc biệt là trong các tình huống mà hàm $ReLU$ thông thường gây ra vấn đề nơ-ron chết.

Leaky ReLU Activation functions

- Leaky $ReLU$ function (Rectified Linear Unit)

$$Leaky\ ReLU(x) = max(0.01x, x)$$

- The Leaky $ReLU$ curve



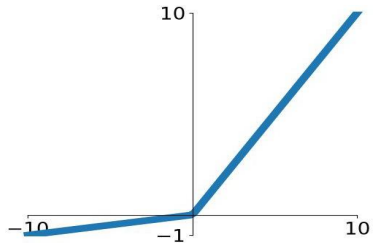
- Does not saturate.
- Computationally efficient.
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x).
- will not “die”.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



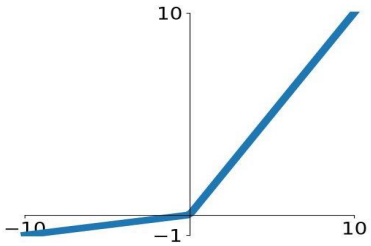
- Các hàm kích hoạt Parametric Rectifier Activation functions, hay còn gọi là $Parametric\ ReLU\ (PReLU)$, là một biến thể của $ReLU$ với độ dốc của phần âm là một tham số có thể học được. Thay vì sử dụng một giá trị cố định cho độ dốc của phần âm như trong $Leaky\ ReLU$, $PReLU$ cho phép mạng nơ-ron học giá trị này trong quá trình huấn luyện.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



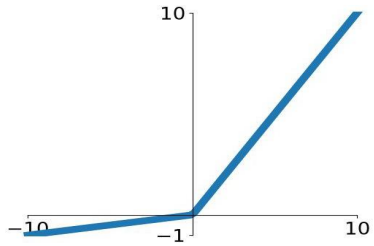
- Đặc điểm của PReLU:
  + Khả năng học tham số α trong quá trình huấn luyện, giúp mô hình tự động điều chỉnh độ dốc của phần âm để tối ưu hiệu suất. Điều này giúp mô hình linh hoạt hơn và có thể tự điều chỉnh để phù hợp với dữ liệu và bài toán cụ thể.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



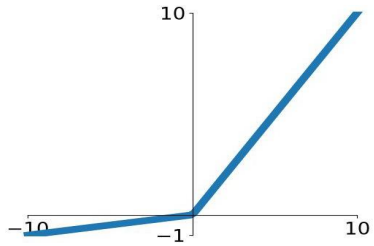
- Đặc điểm của PReLU:
  + Giải quyết vấn đề "dead neurons": Giống như $Leaky\ ReLU$, $PReLU$ cũng giúp tránh tình trạng nơ-ron chết bằng cách cho phép gradient chạy qua các giá trị âm. Điều này giúp các nơ-ron không bị "chết" và tiếp tục học trong suốt quá trình huấn luyện.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



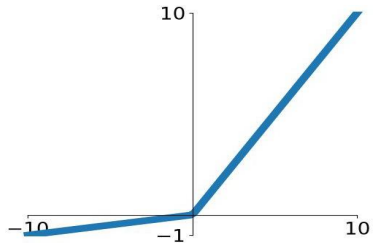
- Đặc điểm của PReLU:
  + Linh hoạt hơn Leaky ReLU: Vì α được học thay vì cố định, $PReLU$ linh hoạt hơn và có thể thích nghi tốt hơn với các đặc điểm của dữ liệu.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



- Lợi ích:

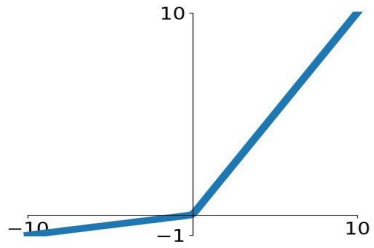
+ Tham số α có thể học được: Thay vì sử dụng một giá trị cố định cho độ dốc của phần âm như trong Leaky ReLU, PReLU cho phép giá trị α được học trong quá trình huấn luyện. Điều này giúp mô hình linh hoạt hơn và có thể tự điều chỉnh để phù hợp với dữ liệu và bài toán cụ thể.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



- Lợi ích:

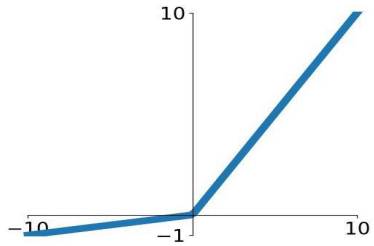
+ Tăng khả năng biểu diễn của mạng nơ-ron: Việc học α giúp $PReLU$ có khả năng biểu diễn tốt hơn các hàm phi tuyến phức tạp, làm tăng khả năng biểu diễn của mạng nơ-ron và cải thiện hiệu suất.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



- Lợi ích:

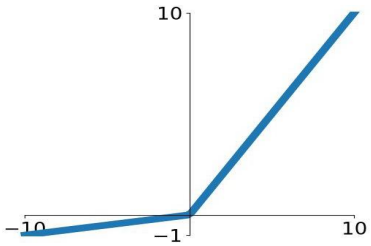
+ Tránh gradient vanishing: $PReLU$ giúp tránh tình trạng gradient vanishing bằng cách đảm bảo rằng gradient không bao giờ bằng 0, ngay cả khi đầu vào là các giá trị âm.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



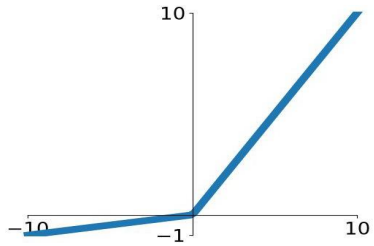
- Nhược điểm:
  + Tăng thêm tham số cần học: Việc cho phép học tham số α làm tăng số lượng tham số cần học của mô hình. Điều này có thể làm tăng độ phức tạp của mô hình và yêu cầu thêm thời gian và tài nguyên để huấn luyện.

Parametric rectifier Activation functions

– Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

– The Parametric rectifier curve



– Nhược điểm:

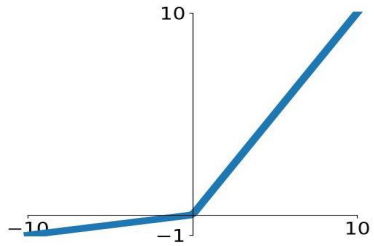
+ Nguy cơ overfitting: Việc có thêm các tham số α có thể làm tăng nguy cơ overfitting, đặc biệt khi dữ liệu huấn luyện không đủ lớn hoặc không được xử lý tốt.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



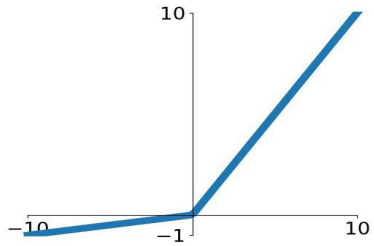
- Nhược điểm:
  + Không phải lúc nào cũng cần thiết: Trong một số trường hợp, việc sử dụng $ReLU$ hoặc $Leaky\ ReLU$ có thể đủ tốt và đơn giản hơn. Sử dụng $PReLU$ có thể là quá mức cần thiết và không mang lại lợi ích rõ ràng so với các hàm kích hoạt khác.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



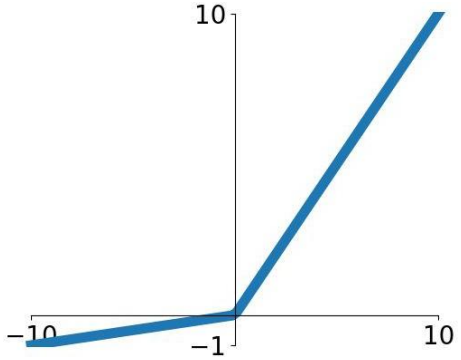
- Ứng dụng:
  + $PReLU$ thường được sử dụng trong các mạng nơ-ron sâu, đặc biệt là trong các mô hình cần sự linh hoạt cao và có thể hưởng lợi từ việc tự điều chỉnh các tham số kích hoạt.

Parametric rectifier Activation functions

- Parametric rectifier Activation functions

$$PReLU(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha x & nếu\ x \leq 0 \end{cases}$$

- The Parametric rectifier curve



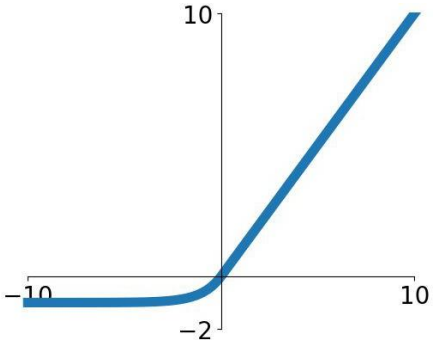
- Does not saturate.
- Computationally efficient.
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x).
- will not “die”.

ELU Activation functions

- Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

- ELU curve



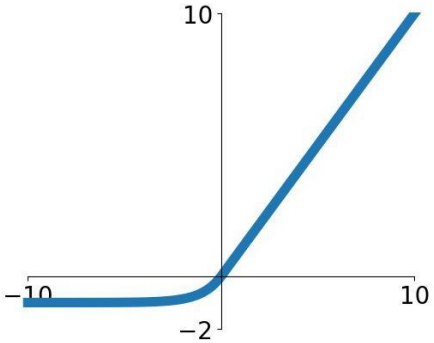
- ELU (Exponential Linear Unit) là một hàm kích hoạt khác được đề xuất nhằm cải thiện hiệu suất của các mạng nơ-ron. ELU giúp khắc phục một số nhược điểm của các hàm kích hoạt khác như $ReLU$ và $Leaky\ ReLU$.

ELU Activation functions

- Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

- ELU curve



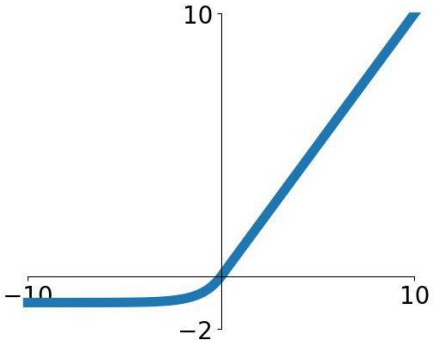
- Đặc điểm của ELU:
  - + Giữ lại giá trị dương: Đối với các giá trị dương, ELU giữ nguyên đầu vào như $ReLU$.
  - + Giá trị âm có độ dốc: Đối với các giá trị âm, ELU áp dụng hàm mũ, tạo ra độ dốc mềm mại hơn so với $Leaky\ ReLU$, giúp tránh hiện tượng "dead neurons".

ELU Activation functions

— Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

— ELU curve



— Đặc điểm của ELU :

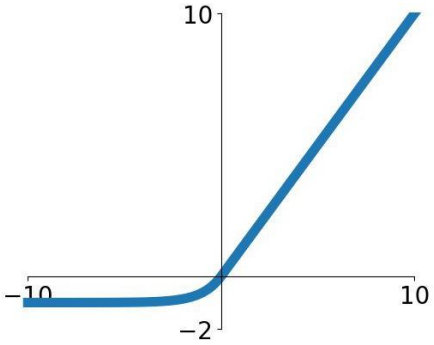
+ Trung bình gần bằng không: Đầu ra của hàm ELU có giá trị trung bình gần bằng 0, giúp giảm độ lệch trong quá trình huấn luyện, tương tự như Tanh nhưng tránh được gradient vanishing.

ELU Activation functions

- Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

- ELU curve



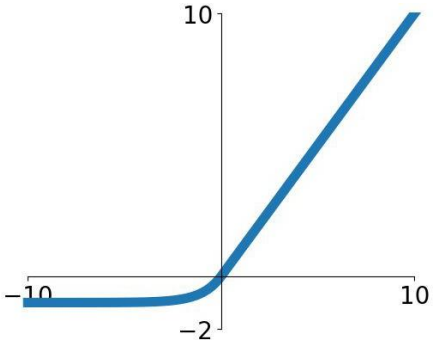
- Lợi ích:
  + Tránh "dead neurons": ELU tránh tình trạng nơ-ron chết bằng cách sử dụng hàm mũ cho các giá trị âm.
  + Gradient lớn hơn ở các giá trị âm: Điều này giúp mạng học nhanh hơn và giảm khả năng bị mắc kẹt ở các điểm cực tiểu cục bộ.

ELU Activation functions

– Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

– ELU curve



– Lợi ích:

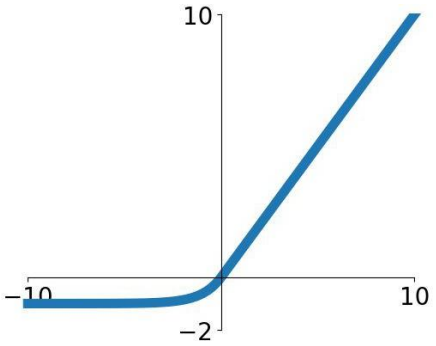
+ Trung bình gần bằng không: Giúp giảm độ lệch của đầu ra, cải thiện sự hội tụ của quá trình huấn luyện.

ELU Activation functions

— Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

— ELU curve



— Nhược điểm:

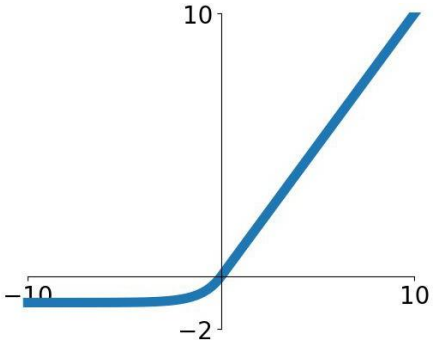
+ Độ phức tạp tính toán cao hơn: Hàm ELU yêu cầu tính toán hàm mũ cho các giá trị đầu vào âm, điều này làm tăng độ phức tạp tính toán so với các hàm kích hoạt đơn giản như $ReLU$ và $Leaky\ ReLU$. Việc này có thể làm chậm quá trình huấn luyện và yêu cầu tài nguyên tính toán nhiều hơn.

ELU Activation functions

- Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

- ELU curve



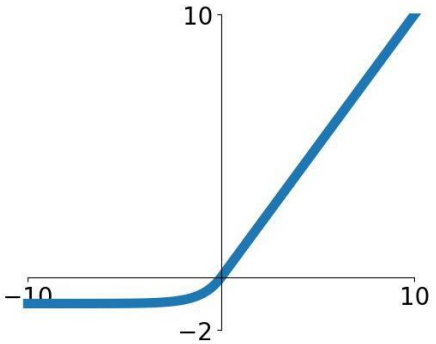
- Nhược điểm:
  + Không loại bỏ hoàn toàn vấn đề gradient vanishing: Mặc dù ELU giúp giảm vấn đề gradient vanishing so với hàm $Sigmoid$ và $Tanh$, ELU vẫn không loại bỏ hoàn toàn vấn đề này. Các phương pháp như Batch Normalization có thể cần thiết để giải quyết vấn đề gradient vanishing một cách triệt để.

ELU Activation functions

— Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

— ELU curve



— Nhược điểm:

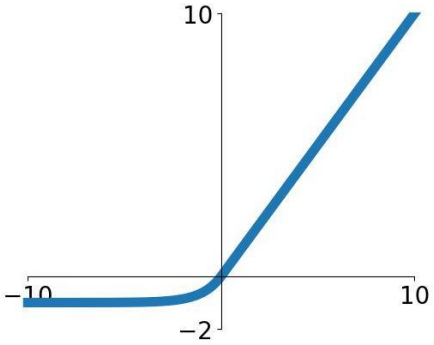
+ Không phải lúc nào cũng hiệu quả hơn $ReLU$: Trong một số trường hợp, $ReLU$ có thể hoạt động tốt hơn ELU, đặc biệt khi yêu cầu tốc độ huấn luyện nhanh. ELU có thể không mang lại lợi ích rõ ràng so với $ReLU$ trong mọi tình huống và có thể gây ra độ phức tạp không cần thiết.

ELU Activation functions

— Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

— ELU curve



— Nhược điểm:

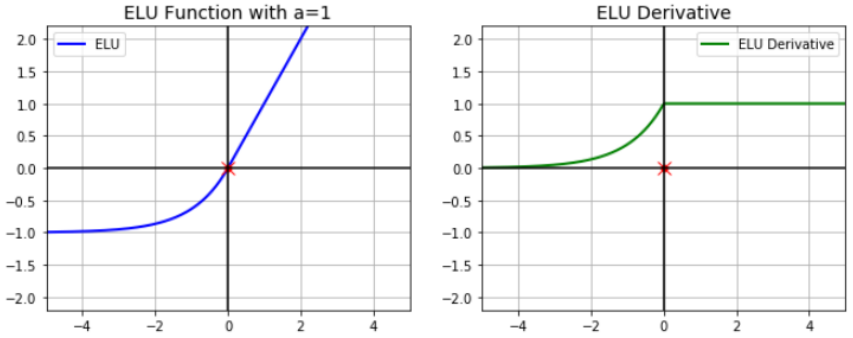
+ Tham số α: ELU có một tham số α xác định độ dốc của phần âm. Việc lựa chọn giá trị α phù hợp có thể không dễ dàng và cần thử nghiệm để tìm ra giá trị tối ưu cho từng bài toán cụ thể.

ELU Activation functions

– Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

– ELU curve



– Nhược điểm:

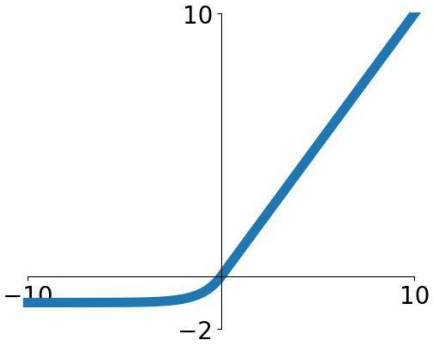
+ Khả năng gây ra exploding gradients: Với các giá trị đầu vào rất lớn, đầu ra của ELU cũng có thể trở nên rất lớn, dẫn đến vấn đề exploding gradients trong một số trường hợp. Mặc dù điều này ít phổ biến hơn so với vấn đề gradient vanishing, nhưng nó vẫn là một nguy cơ cần được xem xét.

ELU Activation functions

– Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

– ELU curve



– Ứng dụng:

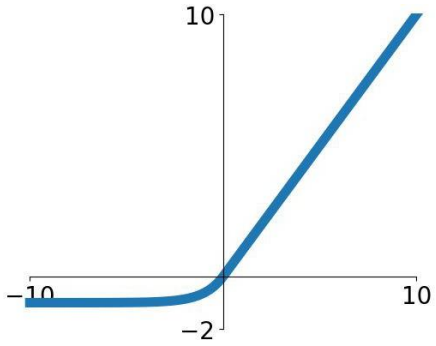
+ ELU thường được sử dụng trong các mạng nơ-ron sâu, đặc biệt là trong các mô hình yêu cầu sự ổn định và hiệu suất cao.

ELU Activation functions

- Exponential Linear Units (ELU)

$$\mathrm{ELU}(x) = \begin{cases} x & nếu\ x > 0 \\ \alpha(e^x - 1) & nếu\ x \leq 0 \end{cases}$$

- ELU curve



- All benefits of $ReLU$
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

- Computation requires exp()

Maxout Activation functions

- Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

- Maxout curve

- Hàm kích hoạt Maxout là một hàm kích hoạt khác được thiết kế để khắc phục một số hạn chế của các hàm kích hoạt truyền thống như ReLU, Sigmoid, và Tanh. Maxout không phải là một hàm kích hoạt theo nghĩa thông thường, mà là một hàm tuyến tính từng đoạn. Hàm Maxout được giới thiệu trong bài báo "Maxout Networks" của Ian Goodfellow et al. năm 2013.

Maxout Activation functions

- Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

- Maxout curve

- Đặc điểm của Maxout:
  + Tuyến tính từng đoạn: Maxout có tính chất tuyến tính từng đoạn, cho phép mô hình hóa các quan hệ phi tuyến phức tạp.
  + Tránh gradient vanishing: Vì Maxout chọn giá trị lớn nhất từ các đơn vị tuyến tính, nó không gặp phải vấn đề gradient vanishing.

Maxout Activation functions

- Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

- Maxout curve

- Đặc điểm của Maxout:
  + Đa dạng hóa hàm kích hoạt: Maxout có thể biểu diễn bất kỳ hàm kích hoạt nào bằng cách chọn các trọng số và bias phù hợp.

Maxout Activation functions

- Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

- Maxout curve

- Lợi ích:
  + Khả năng biểu diễn mạnh mẽ: Maxout có khả năng biểu diễn các hàm phi tuyến phức tạp hơn so với các hàm kích hoạt truyền thống.
  + Giải quyết vấn đề gradient vanishing: Maxout tránh được vấn đề gradient vanishing, giúp cải thiện quá trình huấn luyện mạng nơ-ron sâu.

Maxout Activation functions

– Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

– Maxout curve

– Nhược điểm:

+ Tăng số lượng tham số: Maxout yêu cầu tăng số lượng tham số đáng kể vì nó cần nhiều đơn vị tuyến tính (linear units) cho mỗi đầu ra. Điều này có thể làm tăng kích thước và độ phức tạp của mô hình, đòi hỏi thêm tài nguyên tính toán và bộ nhớ.

Maxout Activation functions

- Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

- Maxout curve

- Nhược điểm:
  + Overfitting: Với số lượng tham số tăng lên, nguy cơ overfitting cũng tăng theo, đặc biệt khi dữ liệu huấn luyện không đủ lớn hoặc không được xử lý tốt. Điều này đòi hỏi phải áp dụng các kỹ thuật regularization như dropout hoặc weight decay để giảm thiểu overfitting.

Maxout Activation functions

- Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

- Maxout curve

- Nhược điểm:
  + Độ phức tạp tính toán: Việc tính toán đầu ra của Maxout phức tạp hơn so với các hàm kích hoạt khác như ReLU, vì nó yêu cầu tính toán nhiều hàm tuyến tính và sau.

Maxout Activation functions

- Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

- Maxout curve

- Ứng dụng:
  + Maxout thường được sử dụng trong các mô hình mạng nơ-ron sâu yêu cầu khả năng biểu diễn cao và tránh các vấn đề gradient vanishing.

Maxout Activation functions

- Maxout

$$Maxout(x) = \max_{i \in [1,k]} \left(w_i^T x + b_i \right)$$

- Maxout curve

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

- Problem: doubles the number of parameters/neuron :(

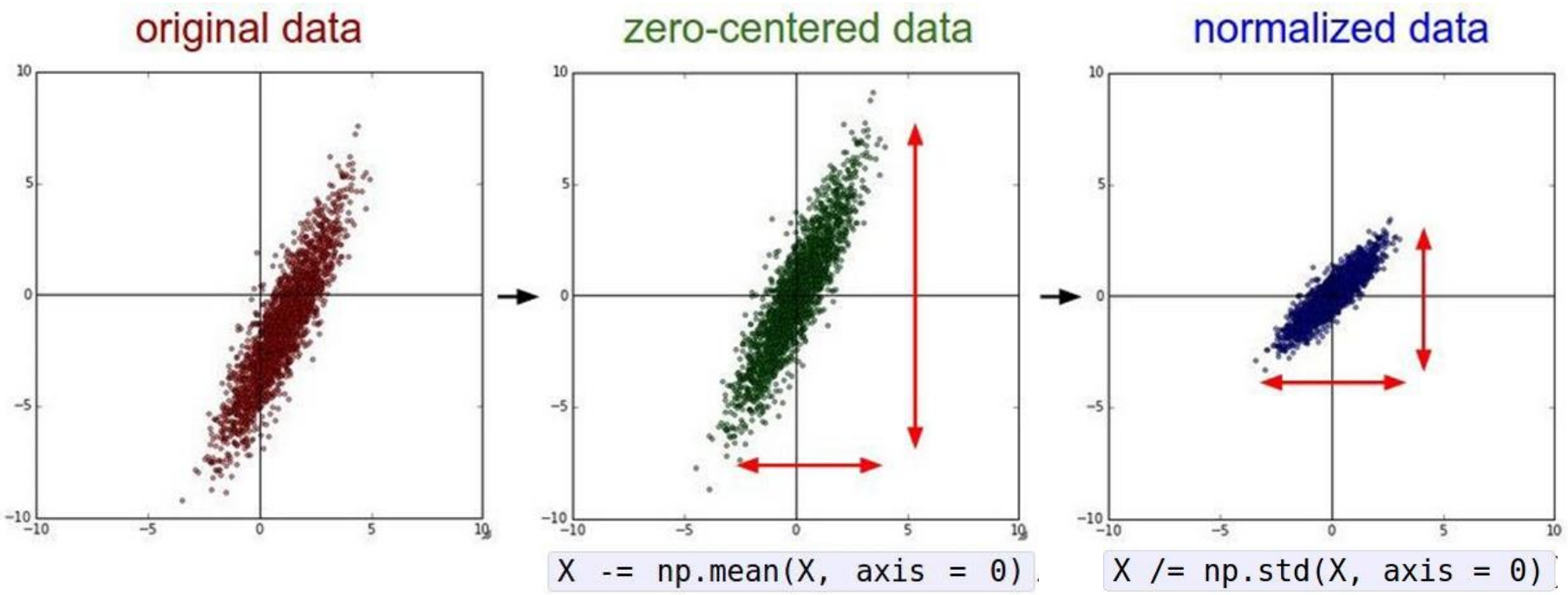
TLDR: In practice:

- Use $ReLU$. Be careful with your learning rates.
- Sử dụng $ReLU$. Hãy cẩn thận với tốc độ huấn luyện.
- Try out $Leaky\ ReLU$ / Maxout / ELU.
- Hãy thử $Leaky\ ReLU$ / Maxout / ELU.
- Try out $\tanh$ but don't expect much.
- Hãy thử $\tanh$ nhưng đừng mong đợi nhiều.
- Don't use sigmoid.
- Đừng sử dụng sigmoid.

Training Neural Networks, Part 1

DATA PREPROCESSING

Step 1: Preprocess the data



– Assume $X[N \times D]$ is data matrix, each example in a row.

Zero-centered data

- Zero-centered data (dữ liệu trung tâm hóa về zero) là một kỹ thuật trong tiền xử lý dữ liệu, trong đó giá trị trung bình của dữ liệu được dịch chuyển về zero.
- Điều này có nghĩa là các giá trị của dữ liệu được điều chỉnh sao cho trung bình của chúng bằng zero.
- Kỹ thuật này thường được sử dụng để cải thiện hiệu suất và tốc độ hội tụ của các thuật toán học máy, đặc biệt là các thuật toán tối ưu hóa gradient.

Zero-centered data

– Lợi ích của Zero-Centered Data:

+ Cải thiện tốc độ hội tụ: Khi dữ liệu được trung tâm hóa về zero, quá trình học của các thuật toán tối ưu hóa gradient trở nên hiệu quả hơn, giúp tăng tốc độ hội tụ.
+ Giảm độ lệch: Zero-centered data giúp giảm độ lệch của dữ liệu, làm cho các bước cập nhật trong quá trình huấn luyện nhất quán hơn.
+ Ổn định các mô hình học sâu: Trung tâm hóa dữ liệu về zero giúp ổn định các mạng nơ-ron sâu bằng cách giảm nguy cơ gradient vanishing và exploding.

Zero-centered data

- Cách trung tâm hóa dữ liệu về zero: Để trung tâm hóa dữ liệu về zero, ta cần tính toán giá trị trung bình của dữ liệu và sau đó trừ giá trị trung bình này khỏi mỗi điểm dữ liệu. Công thức như sau:

$$\mathbf{x}' = \mathbf{x} - \mu$$

- Trong đó:
  + $\mathbf{x}$ là điểm dữ liệu ban đầu.
  + μ là giá trị trung bình của tập dữ liệu.
  + $\mathbf{x}'$ là điểm dữ liệu sau khi đã trung tâm hóa về zero.

Zero-centered data

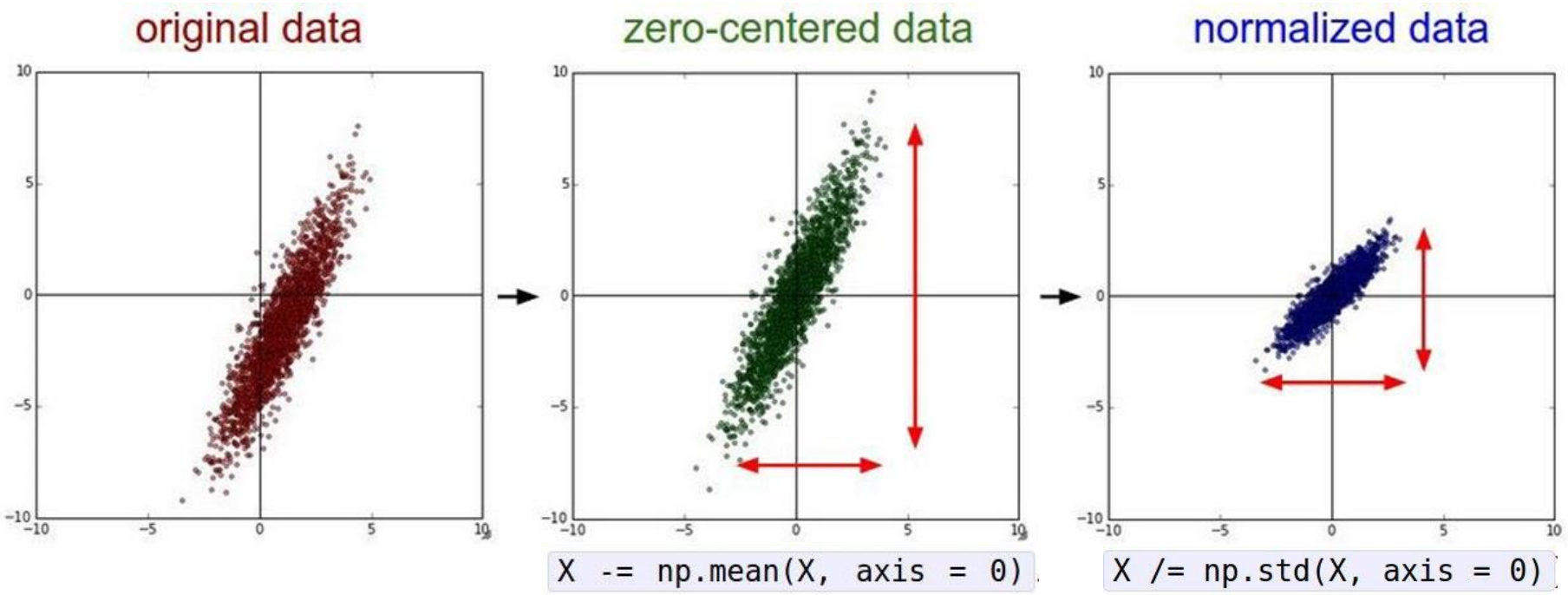
– Ví dụ về trung tâm hóa dữ liệu trong Python:

```
11.import numpy as np

12.# Tạo dữ liệu giả lập
13.data = np.array([1, 2, 3, 4, 5])

14.# Tính giá trị trung bình của dữ liệu
15.mean = np.mean(data)
```

Zero-centered data

```
16.# Trung tâm hóa dữ liệu về zero
17.zero_centered_data = data - mean

18.print("Dữ liệu ban đầu:", data)
19.print("Giá trị trung bình:", mean)
20.print("Dữ liệu trung tâm hóa về zero:",
   zero_centered_data)
```

Step 1: Preprocess the data



– Assume $X[N \times D]$ is data matrix, each example in a row.

Normalized data

- Normalized data (chuẩn hóa dữ liệu) là một kỹ thuật trong tiền xử lý dữ liệu nhằm mục đích điều chỉnh các giá trị của dữ liệu vào một khoảng nhất định hoặc một tỉ lệ xác định, giúp cải thiện hiệu suất của các mô hình học máy và các thuật toán học sâu.
- Quá trình này giúp các đặc trưng (features) của dữ liệu có quy mô và đơn vị đo lường tương đồng, từ đó giúp các thuật toán hoạt động hiệu quả hơn.

Normalized data

– Lợi ích của Normalized Data:

+ Tăng tốc độ hội tụ: Các thuật toán tối ưu hóa gradient, như Gradient Descent, hội tụ nhanh hơn khi dữ liệu được chuẩn hóa.
+ Cải thiện độ chính xác: Dữ liệu chuẩn hóa giúp các mô hình học máy tránh việc một đặc trưng chi phối quá trình học, do đó cải thiện độ chính xác.
+ Giảm ảnh hưởng của outliers: Khi dữ liệu được chuẩn hóa, các giá trị ngoại lệ có thể ít ảnh hưởng hơn đến mô hình.

Normalized data

– Các phương pháp chuẩn hóa dữ liệu:

+ Min-Max Normalization (Chuẩn hóa Min-Max)

+ Z-score Normalization (Chuẩn hóa Z-score)

+ Robust Scaler

Normalized data

– Min-Max Normalization (Chuẩn hóa Min-Max):

+ Chuẩn hóa dữ liệu vào một khoảng [0, 1] hoặc [-1, 1].

+ Công thức:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

+ Trong đó x là giá trị ban đầu, x_{min} và x_{max} lần lượt là giá trị nhỏ nhất và lớn nhất của tập dữ liệu.

Normalized data

– Z-score Normalization (Chuẩn hóa Z-score):

+ Chuẩn hóa dữ liệu sao cho giá trị trung bình của nó là 0 và độ lệch chuẩn là 1.

+ Công thức:

$$x' = \frac{x - \mu}{\sigma}$$

+ Trong đó:

- x là giá trị ban đầu,
- μ là giá trị trung bình và
- σ là độ lệch chuẩn của tập dữ liệu.

Normalized data

– Robust Scaler:

+ Chuẩn hóa dữ liệu bằng cách sử dụng median (trung vị) và interquartile range (IQR).
+ Thích hợp cho các tập dữ liệu có outliers.
+ Công thức:

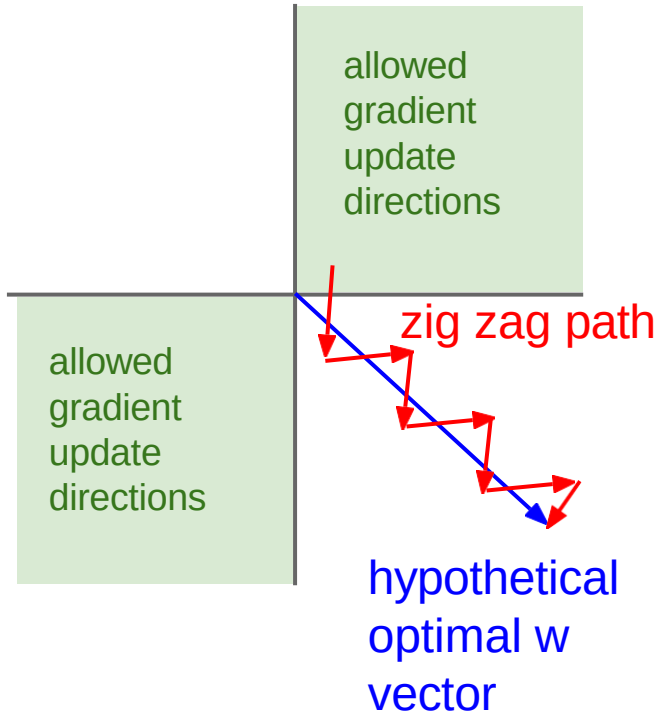
$$x' = \frac{x - median}{IQR}$$

+ Trong đó median là trung vị và IQR là khoảng giữa các phân vị thứ nhất (25%) và thứ ba (75%).

Step 1: Preprocess the data

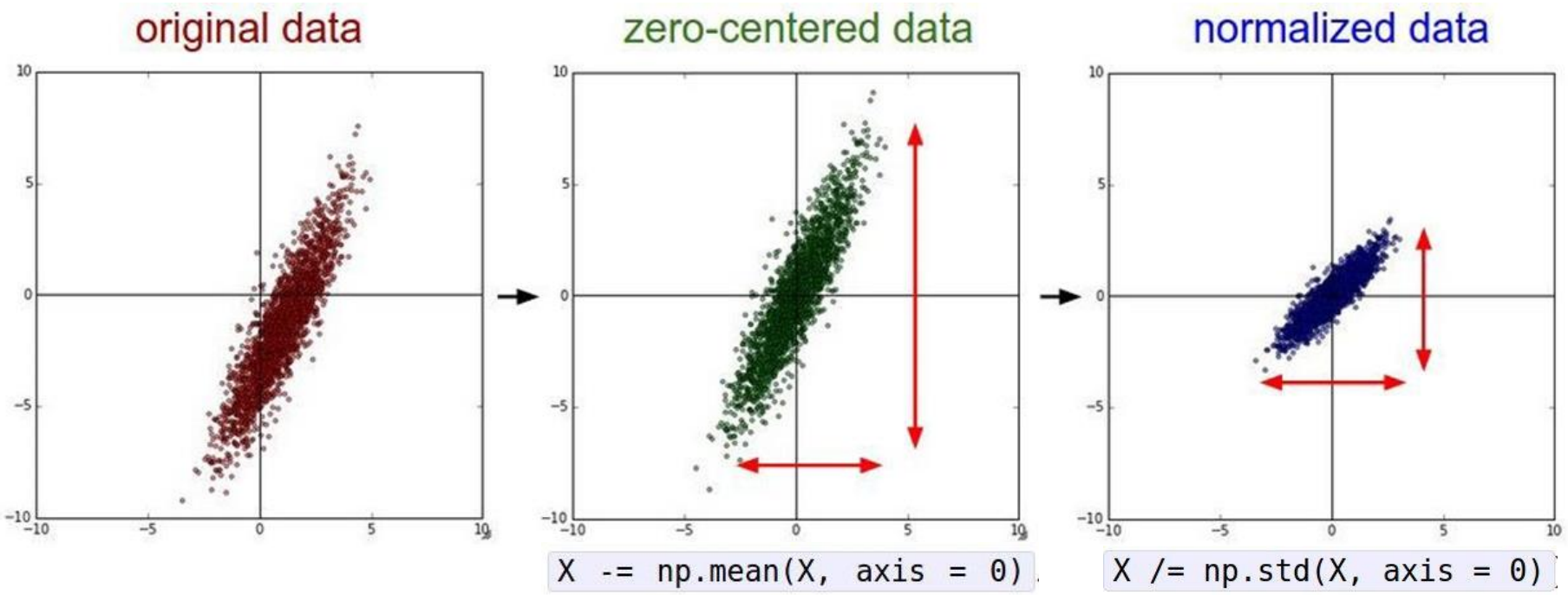
- Remember: Consider what happens when the input to a neuron is always positive...

$$f\left(\sum_i w_i x_i + b\right)$$



- What can we say about the gradients on w?
- Always all positive or all negative :(
- (this is also why you want zero-mean data!)

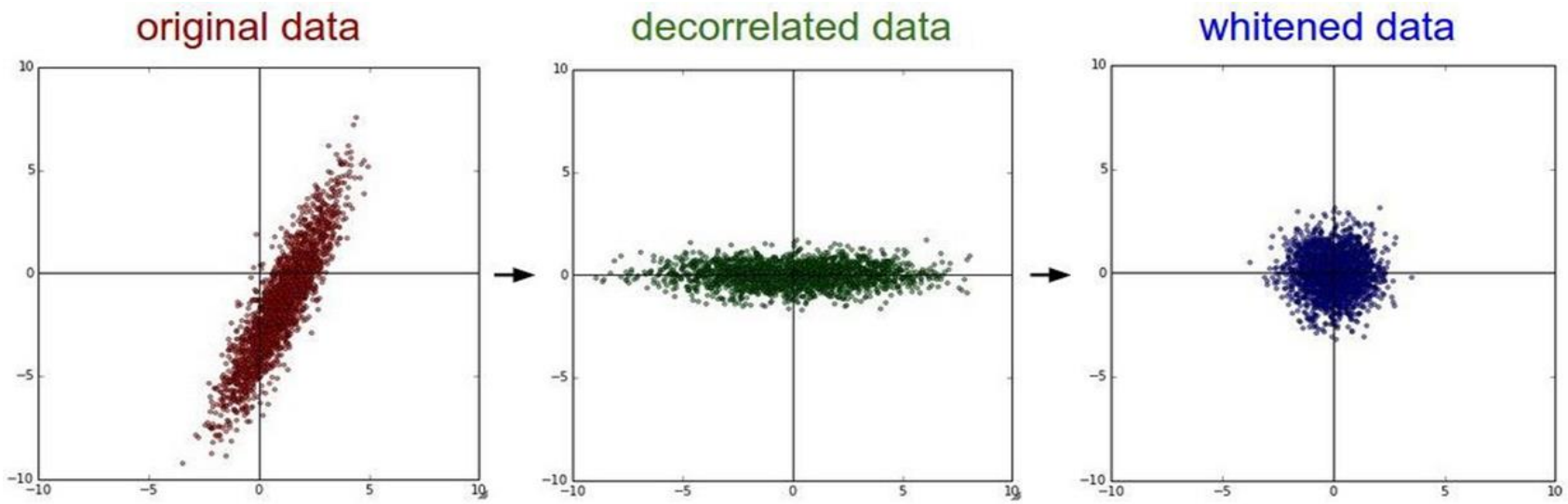
Step 1: Preprocess the data



– Assume $X[N \times D]$ is data matrix, each example in a row.

Step 1: Preprocess the data

– In practice, you may also see PCA and Whitening of the data



data has diagonal covariance matrix

covariance matrix is the identity matrix

Decorrelated data

- Dữ liệu decorrelated (dữ liệu không tương quan) là dữ liệu đã được biến đổi sao cho các đặc trưng (features) của nó không còn tương quan với nhau.
- Tức là, mối quan hệ tuyến tính giữa các đặc trưng đã được loại bỏ. Điều này giúp giảm thiểu sự dư thừa thông tin và làm cho các thuật toán học máy hoạt động hiệu quả hơn.

Decorrelated data

- Lợi ích của dữ liệu không tương quan:
  + Giảm đa cộng tuyến (multicollinearity): Khi các đặc trưng không tương quan, vấn đề đa cộng tuyến giữa các đặc trưng được giảm thiểu, giúp cải thiện độ ổn định của các mô hình học máy.
  + Cải thiện hiệu suất mô hình: Các mô hình học máy thường hoạt động tốt hơn khi các đặc trưng không tương quan, vì chúng có thể học được các mối quan hệ riêng rẽ và độc lập giữa các đặc trưng và mục tiêu.

Decorrelated data

- Lợi ích của dữ liệu không tương quan:
  - + …
  - + Giảm kích thước dữ liệu: Khi loại bỏ sự tương quan giữa các đặc trưng, có thể giảm số lượng đặc trưng cần thiết mà không làm mất đi nhiều thông tin, từ đó giảm kích thước dữ liệu và tăng hiệu quả tính toán.

Decorrelated data

– Phương pháp decorrelation (không tương quan hóa dữ liệu):

+ Principal Component Analysis (PCA):

- PCA là một kỹ thuật phân tích dữ liệu phổ biến được sử dụng để giảm chiều dữ liệu và loại bỏ sự tương quan giữa các đặc trưng.
- PCA biến đổi các đặc trưng ban đầu thành các thành phần chính (principal components) sao cho các thành phần này không tương quan với nhau.

Decorrelated data

– Phương pháp decorrelation (không tương quan hóa dữ liệu):

+ Orthogonalization:

- Orthogonalization là quá trình biến đổi các vector thành các vector trực giao (không tương quan với nhau) thông qua các phép biến đổi toán học như Gram-Schmidt orthogonalization.

Decorrelated data

– Phương pháp decorrelation (không tương quan hóa dữ liệu):

+ Sử dụng các ma trận hiệp phương sai (covariance matrix):

• Bằng cách sử dụng ma trận hiệp phương sai, có thể loại bỏ sự tương quan giữa các đặc trưng thông qua các phép biến đổi tuyến tính.

Decorrelated data

– Lợi ích và hạn chế của PCA:

+ Lợi ích:

- Giảm chiều dữ liệu, làm cho mô hình nhẹ hơn và nhanh hơn.
- Loại bỏ sự tương quan giữa các đặc trưng, cải thiện hiệu suất mô hình.

Decorrelated data

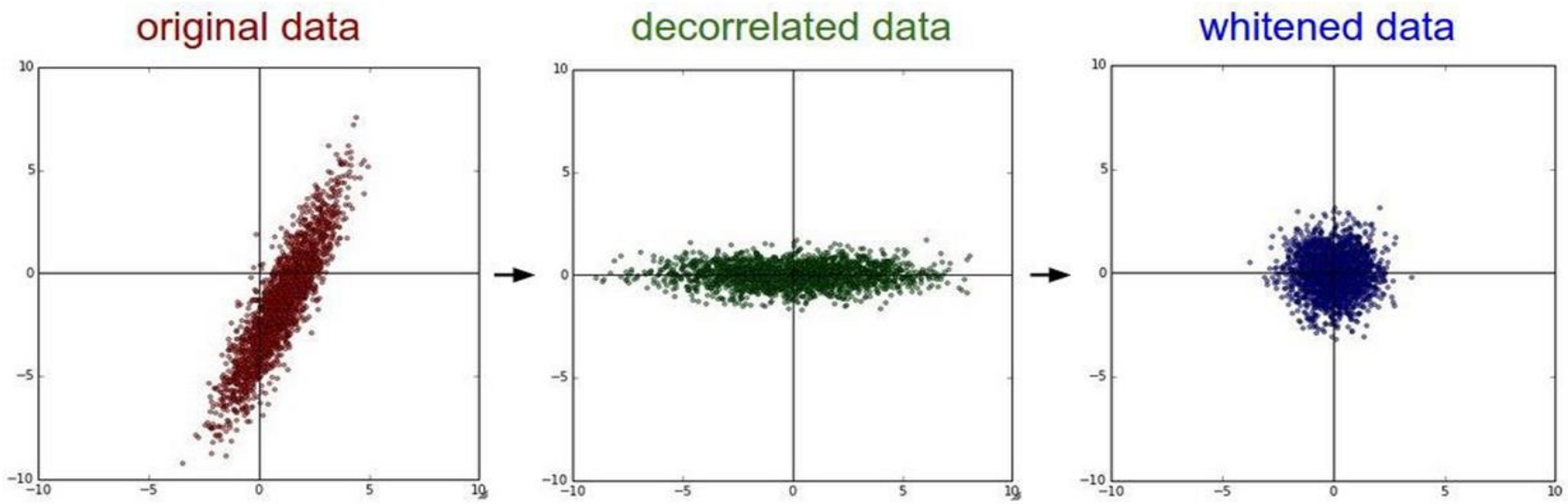
– Lợi ích và hạn chế của PCA:

+ Hạn chế:

- Mất mát thông tin: PCA có thể làm mất một số thông tin quan trọng, đặc biệt nếu chỉ giữ lại một số ít thành phần chính.
- Khó giải thích: Các thành phần chính không có ý nghĩa dễ hiểu như các đặc trưng ban đầu.

Step 1: Preprocess the data

– In practice, you may also see PCA and Whitening of the data



data has diagonal covariance matrix

covariance matrix is the identity matrix

Whitening data

- Whitening data (chuẩn hóa trắng dữ liệu) là một kỹ thuật tiền xử lý dữ liệu trong học máy và thống kê.
- Whitening data (chuẩn hóa trắng dữ liệu) nhằm mục đích làm cho dữ liệu trở nên không tương quan và có phương sai đơn vị (unit variance).
- Điều này có nghĩa là các đặc trưng của dữ liệu sau khi được "whitened" sẽ không còn tương quan với nhau và mỗi đặc trưng sẽ có phương sai bằng 1.

Whitening data

- Lợi ích của Whitening Data:
  - + Loại bỏ sự tương quan: Whitening giúp loại bỏ sự tương quan giữa các đặc trưng, làm cho các đặc trưng trở nên độc lập với nhau.
  - + Đơn vị phương sai: Sau khi whitening, tất cả các đặc trưng sẽ có cùng đơn vị phương sai, giúp cân bằng tầm quan trọng của các đặc trưng trong quá trình huấn luyện.
  - + Cải thiện hiệu suất mô hình: Một số thuật toán học máy hoạt động hiệu quả hơn khi dữ liệu đã được whitening, vì chúng giả định rằng các đặc trưng độc lập và có phương sai đơn vị.

Whitening data

– Phương pháp thực hiện Whitening:

1. Zero-Centering (Trung tâm hóa về zero): Trước hết, dữ liệu được trung tâm hóa bằng cách trừ giá trị trung bình của từng đặc trưng.
2. Tính toán ma trận hiệp phương sai: Ma trận hiệp phương sai của dữ liệu được tính toán.

Whitening data

– Phương pháp thực hiện Whitening:

3. Phân rã ma trận hiệp phương sai: Ma trận hiệp phương sai được phân rã thành các thành phần chính bằng cách sử dụng Phân tích Thành phần Chính (PCA).
4. Chuẩn hóa phương sai: Dữ liệu được biến đổi để có phương sai đơn vị bằng cách chia cho căn bậc hai của các giá trị riêng (eigenvalues).

Whitening data

– Thuật toán Whitening: Giả sử X là dữ liệu gốc, quá trình whitening có thể được biểu diễn như sau:

1. Trung tâm hóa dữ liệu:

$$X_{centered} = X - \mu$$

+ Trong đó: μ là vector giá trị trung bình của X.

2. Tính toán ma trận hiệp phương sai:

$$C = \frac{1}{n} X_{centered}^{T} X_{centered}$$

Whitening data

3. Phân rã ma trận hiệp phương sai:

$$C = EDE^T$$

+ Trong đó:

- E là ma trận vector riêng (eigenvectors).
- D là ma trận đường chéo các giá trị riêng (eigenvalues).

4. Biến đổi whitening:

$$X_{whitened} = ED^{-\frac{1}{2}}E^T X_{centered}$$

Whitening data

- Lợi ích:
  + Loại bỏ sự tương quan giữa các đặc trưng.
  + Đảm bảo các đặc trưng có phương sai đơn vị.
  + Cải thiện hiệu suất của một số thuật toán học máy.

Whitening data

- Hạn chế:
  - + Tốn kém tính toán: Whitening yêu cầu phân rã ma trận hiệp phương sai, điều này có thể tốn kém về mặt tính toán đối với các tập dữ liệu lớn.
  - + Mất thông tin: Trong một số trường hợp, quá trình whitening có thể làm mất một số thông tin quan trọng trong dữ liệu.

In practice for Images: Center Only

- e.g. consider CIFAR-10 example with [32,32,3] images
- Subtract the mean image (e.g. AlexNet)
- (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
- (mean along each channel = 3 numbers)

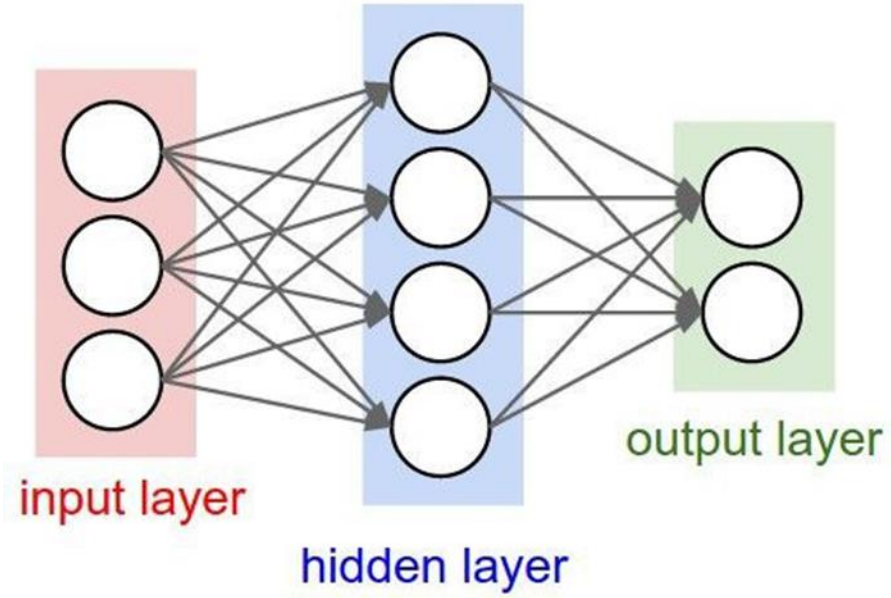
Not common to normalize variance, to do PCA or whitening

Training Neural Networks, Part 1

WEIGHT INITIALIZATION

Weight initialization

– Q: What happens when $W = 0$ init is used?



Weight initialization

- First idea: Small random numbers (gaussian with zero mean and 1e-2 standard deviation).

```
W = 0.01* np.random.randn(D,H)
```

- Works ~okay for small networks, but problems with deeper networks.

Lets look at some activation statistics

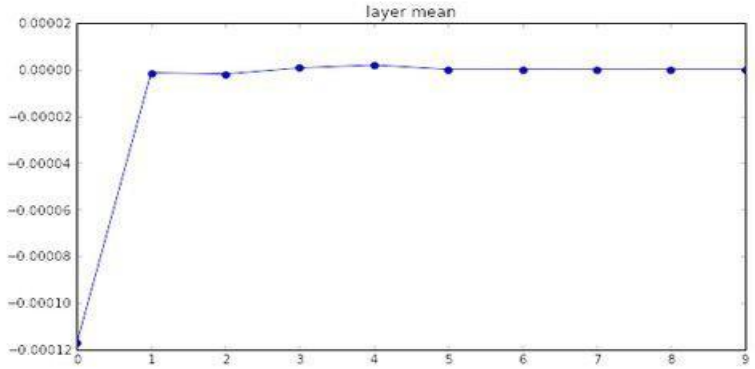
E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

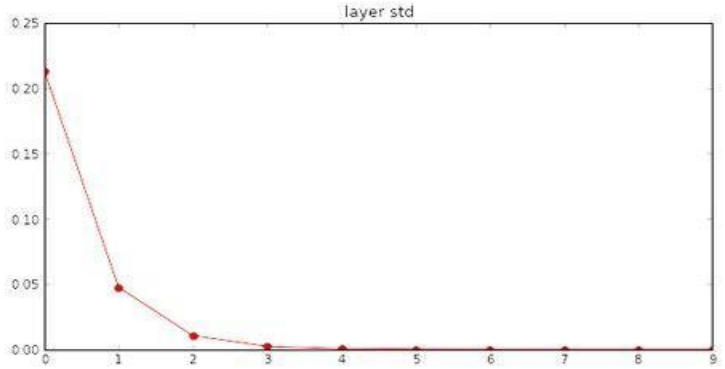
```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

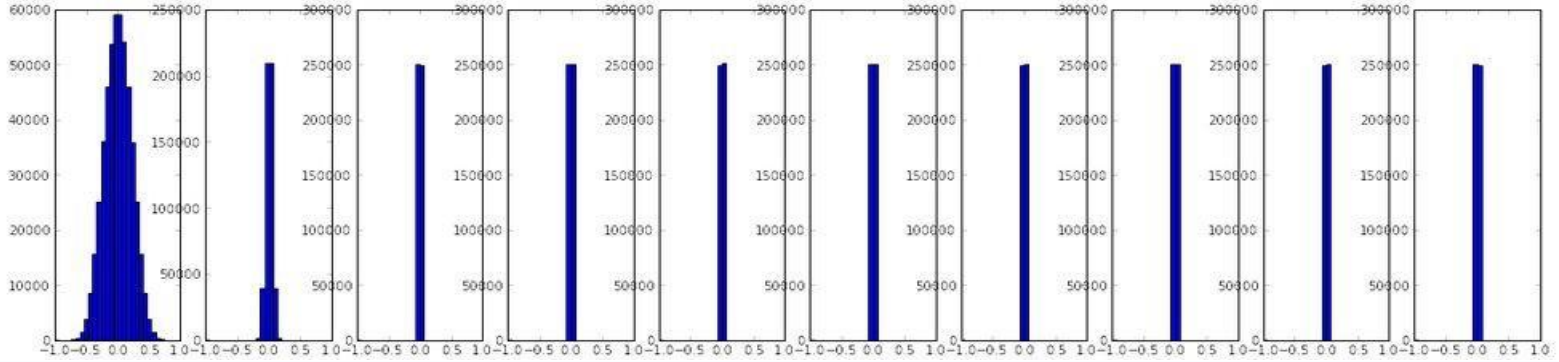
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```







All activations become zero!

Q: think about the backward pass.

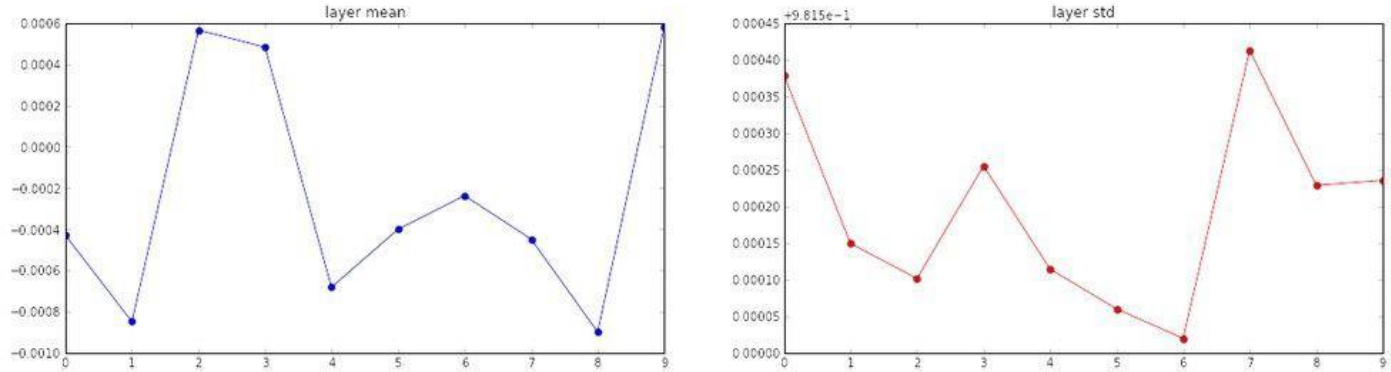
What do the gradients look like?

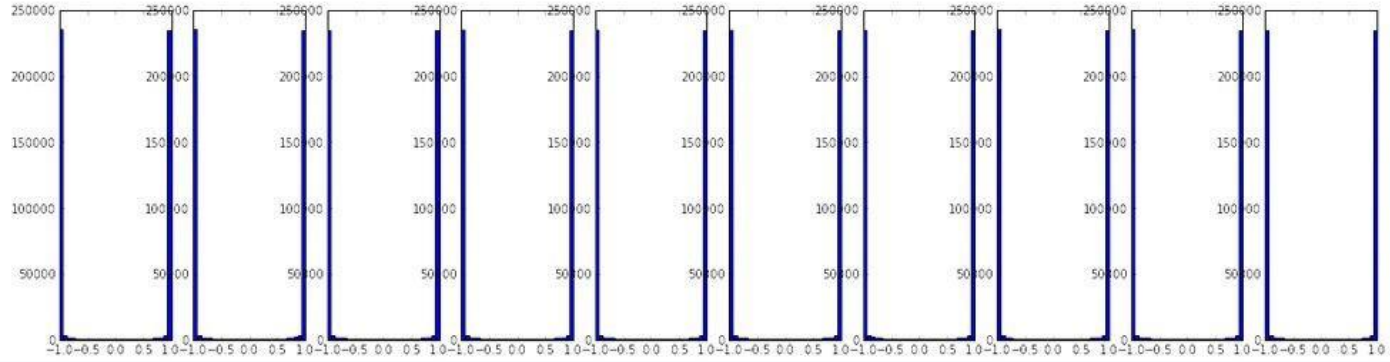
Hint: think about backward pass for a W*X gate.

```
W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization
```

*1.0 instead of *0.01

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```





All activations become zero!

Q: think about the backward pass.

Almost all neurons completely saturated, either -1 and 1.

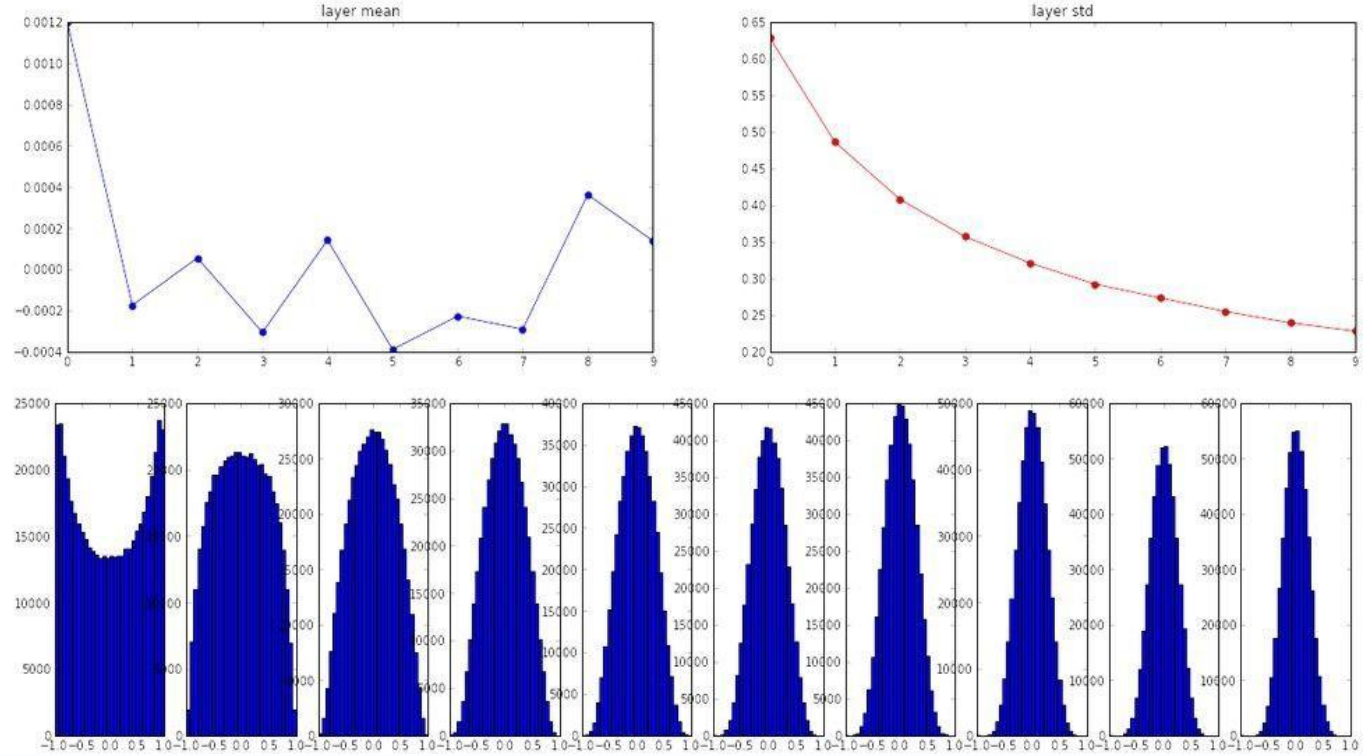
What do the gradients look like?

Gradients will be all zero.

Hint: think about backward pass for a W*X gate.

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

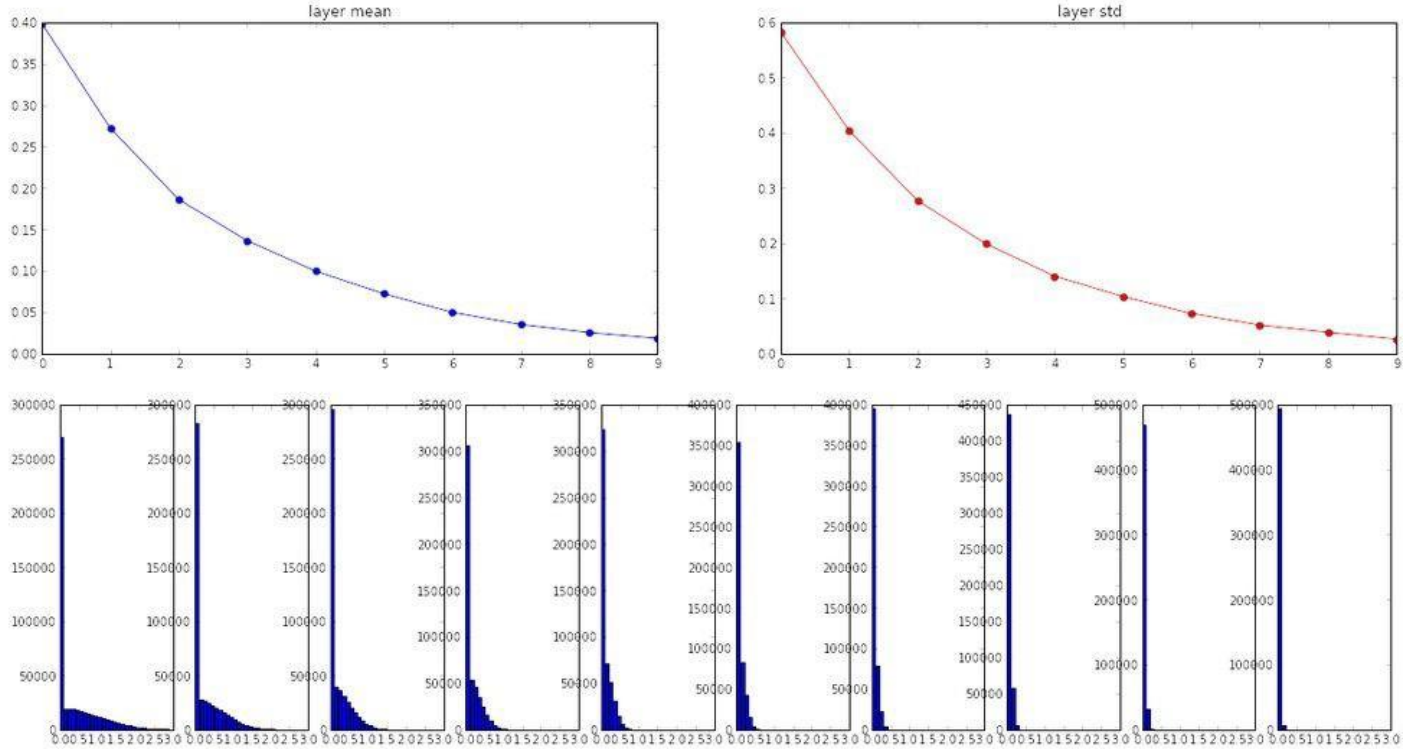


"Xavier initialization" [Glorot et al., 2010]

Reasonable initialization. (Mathematical derivation assumes linear activations)

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

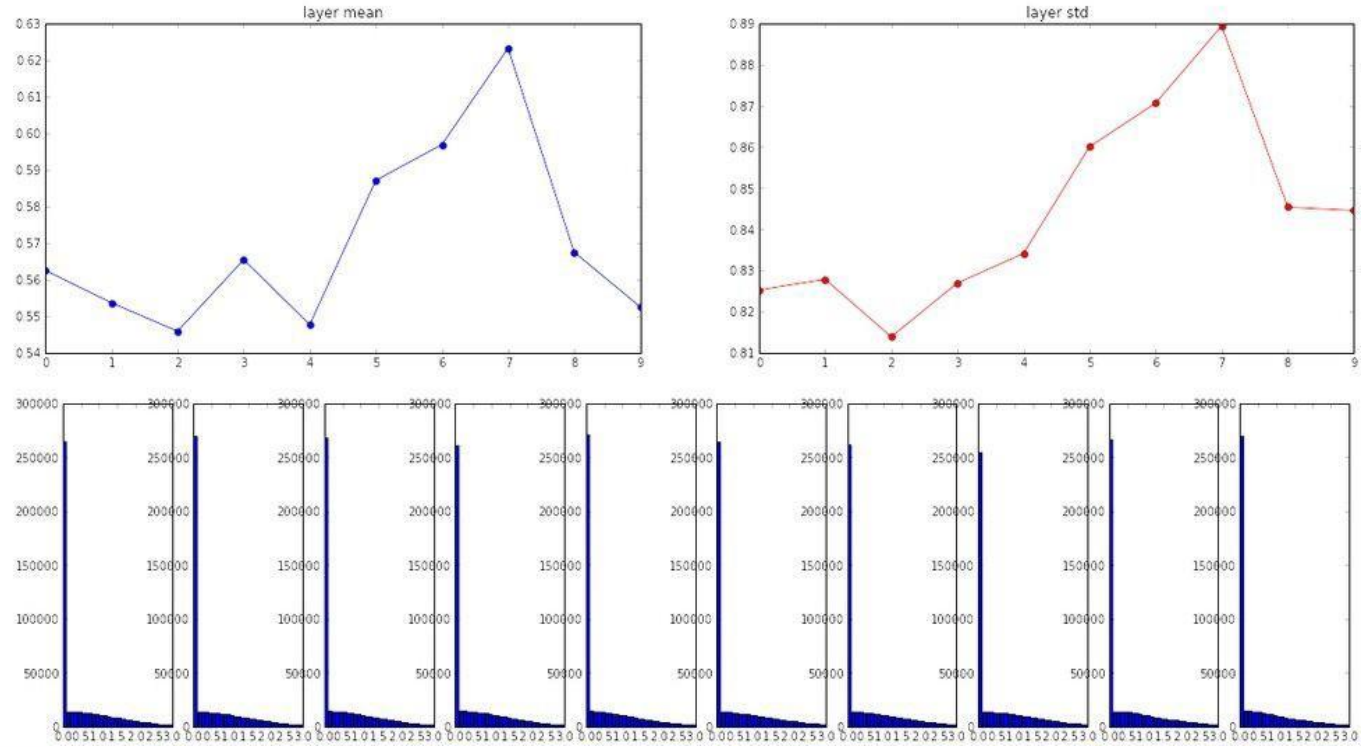
```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```



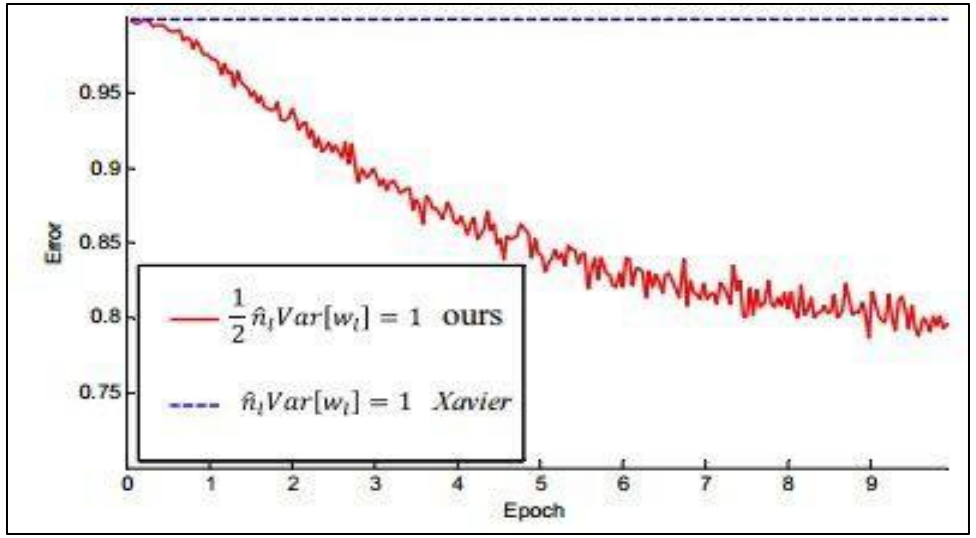
but when using the ReLU nonlinearity it breaks.

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```



He et al., 2015
(note additional /2)



An active area of research…

- Understanding the difficulty of training deep feedforward neural networks by Glorot and Bengio, 2010.
- Exact solutions to the nonlinear dynamics of learning in deep linear neural networks by Saxe et al, 2013.
- Random walk initialization for training very deep feedforward networks by Sussillo and Abbott, 2014.

An active area of research…

- Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification by He et al., 2015.
- Data-dependent Initializations of Convolutional Neural Networks by Krähenbühl et al., 2015.
- All you need is a good init, Mishkin and Matas, 2015.

Training Neural Networks, Part 1

BATCH NORMALIZATION

Batch Normalization

- Batch Normalization là một kỹ thuật trong học máy, đặc biệt là trong mạng nơ-ron sâu, nhằm cải thiện tốc độ huấn luyện và độ ổn định của mô hình bằng cách chuẩn hóa các đầu ra của mỗi lớp trước khi truyền vào lớp kế tiếp.
- Kỹ thuật này được giới thiệu bởi Sergey Ioffe và Christian Szegedy vào năm 2015 trong bài báo "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift".

Batch Normalization

- Định nghĩa: Batch Normalization chuẩn hóa đầu ra của mỗi lớp sao cho giá trị trung bình gần bằng 0 và độ lệch chuẩn gần bằng 1, giúp giảm thiểu sự thay đổi của các tham số trong quá trình huấn luyện.

Batch Normalization

– Cách hoạt động của Batch Normalization:

1. Tính toán giá trị trung bình và độ lệch chuẩn:
   - Trong mỗi batch dữ liệu,
     – Tính giá trị trung bình (μ_B) của các đầu ra (activations).
     – Tính độ lệch chuẩn (σ_B) của các đầu ra (activations).

Batch Normalization

– Cách hoạt động của Batch Normalization (tiếp theo):

2. Chuẩn hóa:

- Sử dụng giá trị trung bình và độ lệch chuẩn để chuẩn hóa các đầu ra (x_i) theo công thức:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

- trong đó ϵ là một giá trị rất nhỏ để tránh chia cho 0.

Batch Normalization

– Cách hoạt động của Batch Normalization (tiếp theo):

3. Tỷ lệ và dịch chuyển:

- Sau khi chuẩn hóa, áp dụng các tham số tỷ lệ (γ) và dịch chuyển (β) để cho phép mô hình học lại các biến đổi cần thiết:

$$y_i = \gamma \hat{x}_i + \beta$$

Batch Normalization

- Lợi ích của Batch Normalization:
  - + Tăng tốc độ huấn luyện: Giảm thiểu vấn đề gradient vanishing và exploding, cho phép sử dụng learning rate lớn hơn.
  - + Ổn định huấn luyện: Giảm độ nhạy cảm của mô hình đối với các thay đổi nhỏ trong tham số, từ đó cải thiện hiệu suất và độ ổn định.
  - + Giảm sự phụ thuộc vào việc khởi tạo trọng số: Batch Normalization làm cho mạng ít nhạy cảm hơn với việc khởi tạo trọng số, giúp quá trình huấn luyện ổn định hơn.

Batch Normalization

– Tổng kết:

+ Batch Normalization là một kỹ thuật mạnh mẽ giúp tăng tốc và ổn định quá trình huấn luyện các mạng nơ-ron sâu bằng cách chuẩn hóa các đầu ra của mỗi lớp trong một batch dữ liệu.
+ Kỹ thuật này đã trở thành một phần quan trọng trong việc thiết kế và huấn luyện các mô hình học sâu hiện đại.
+ Ứng dụng: Batch Normalization được sử dụng rộng rãi trong các mạng nơ-ron sâu hiện đại như: CNN (Convolutional Neural Networks), RNN (Recurrent Neural Networks).

Batch Normalization

– "you want unit gaussian activations? just make them so."

– consider a batch of activations at some layer.

– To make each dimension unit gaussian, apply:

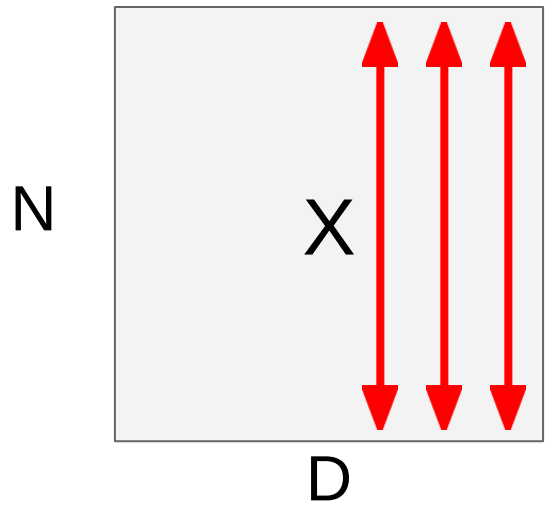
$$\hat{x}^k = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$

this is a vanilla differentiable function...

Batch Normalization

– "you want unit gaussian activations?

– Just make them so."

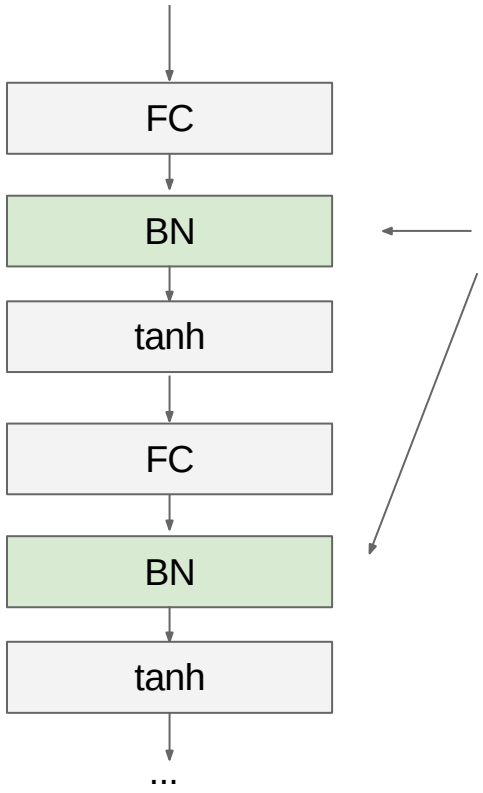


1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\hat{x}^k = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$

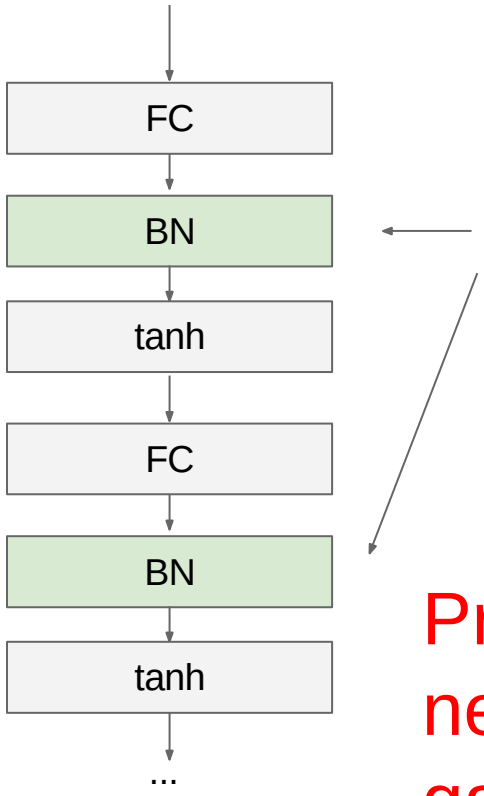
Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^k = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$

Batch Normalization



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\hat{x}^k = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$

Problem: do we necessarily want a unit gaussian input to a tanh layer?

Batch Normalization

- Normalize:

$$\hat{x}^k = \frac{x^{(k)} - E\left[x^{(k)}\right]}{\sqrt{Var\left[x^{(k)}\right]}}$$

- And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\hat{x}^k + \beta^{(k)}$$

- Note, the network can learn:

$$\gamma^{(k)} = \sqrt{Var\left[x^{(k)}\right]}$$

$$\beta^{(k)} = E\left[x^{(k)}\right]$$

- to recover the identity mapping.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network.
- Allows higher learning rates
- Reduces the strong dependence on initialization.
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe.

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

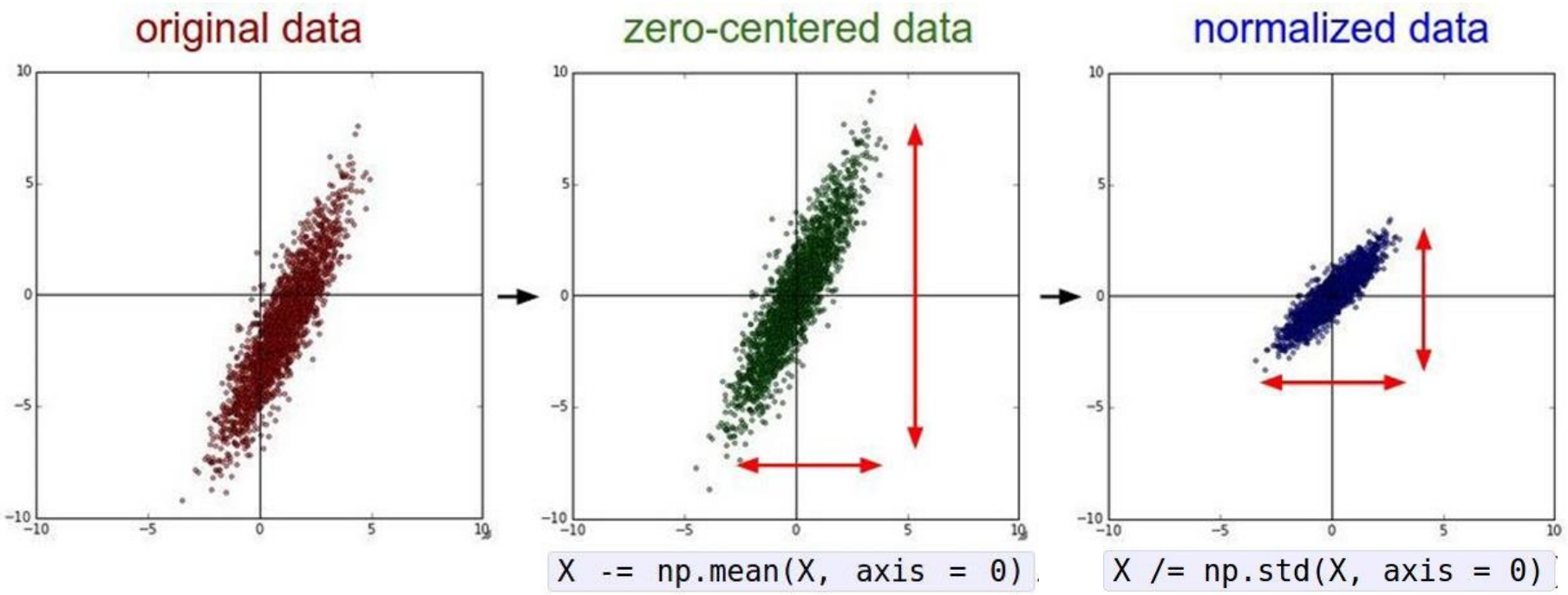
$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Note: at test time BatchNorm layer functions differently:
- The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.
- (e.g. can be estimated during training with running averages)

Training Neural Networks, Part 1

BABYSITTING THE LEARNING PROCESS

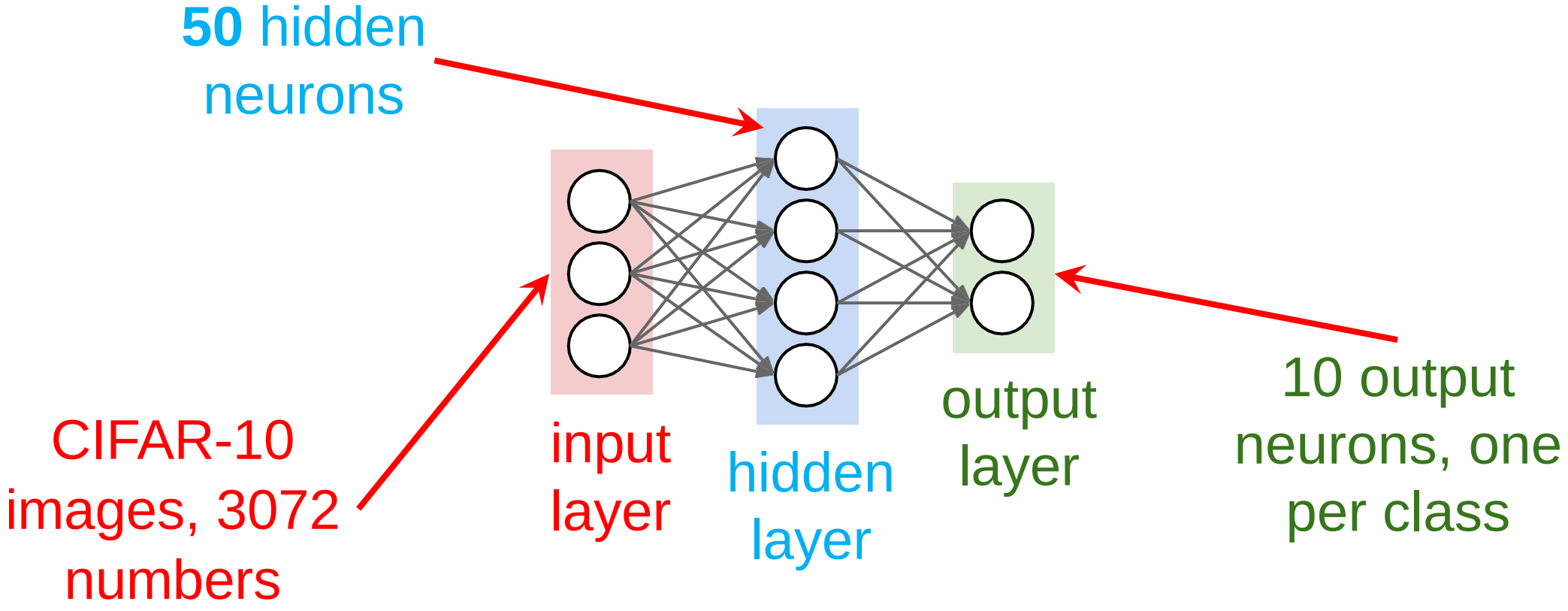
Step 1: Preprocess the data



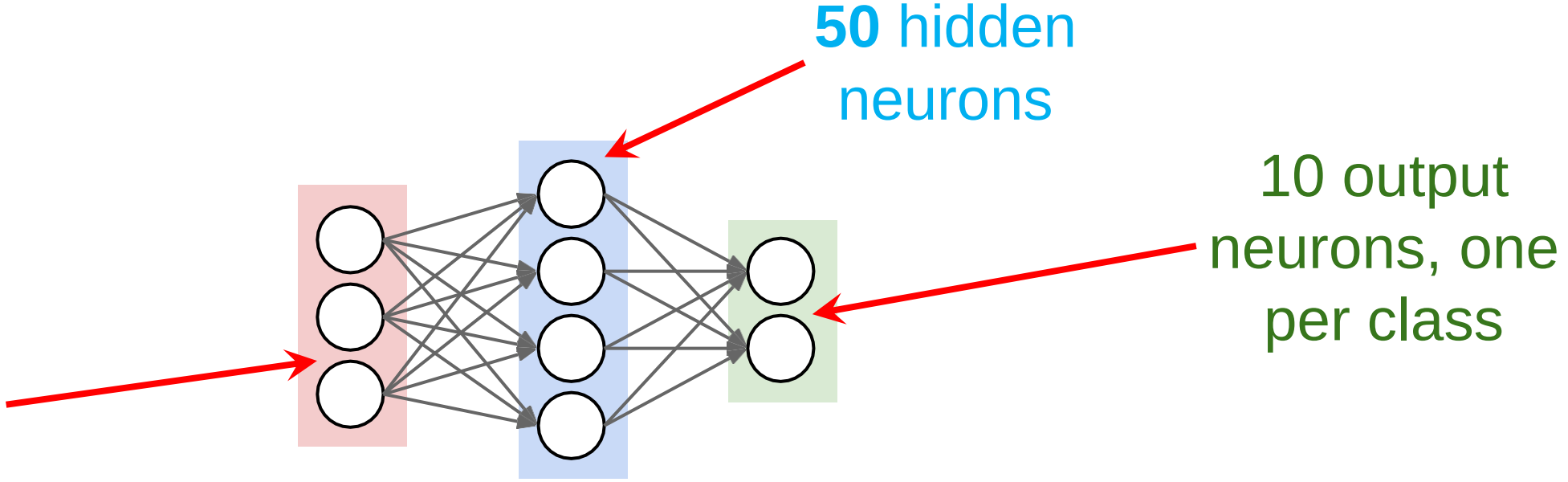
– Assume $X[N \times D]$ is data matrix, each example in a row.

Step 2: Choose the architecture:

– Say we start with one hidden layer of 50 neurons:



Double check – the loss is reasonable:




```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

Double check – the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

disable regularization

```
2.30261216167
```

returns the loss and the gradient for all parameters

loss ~2.3. “correct “ for 10 classes

Double check – the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

```
3.06859716482
```

returns the loss and the gradient for all parameters

loss went up, good. (sanity check)

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

The above code:

1. take the first 20 examples from CIFAR-10
2. turn off regularization (reg = 0.0)
3. use simple vanilla 'sgd'

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Very small loss, train accuracy 1.00, nice!

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing:

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

loss not going down: learning rate too low

Loss barely changing: Learning rate is probably too low

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

loss not going down:
learning rate too low

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

Now let's try learning rate 1e6.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

loss not going down:
learning rate too low
loss exploding:
learning rate too high

cost: NaN almost always means high learning rate...

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

Now let's try learning rate 3e-3.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

loss not going down:
learning rate too low
loss exploding:
learning rate too high

3e-3 is still too high. Cost explodes….

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

3e-3 is still too high. Cost explodes....

⇒ Rough range for learning rate we should be cross-validating is somewhere [1e-3 ... 1e-5]

loss not going down:
learning rate too low
loss exploding:
learning rate too high

3e-3 is still too high. Cost explodes….

⟹ Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]

Training Neural Networks, Part 1

HYPERPARAMETER OPTIMIZATION

Cross-validation strategy

- **coarse** ⟹ **fine** cross-validation in stages
- First stage: only a few epochs to get rough idea of what params work
- Second stage: longer running time, finer search… (repeat as necessary)

- Tip for detecting explosions in the solver:
- If the cost is ever > 3 * original cost, break out early.

Ex: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

nice

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range →

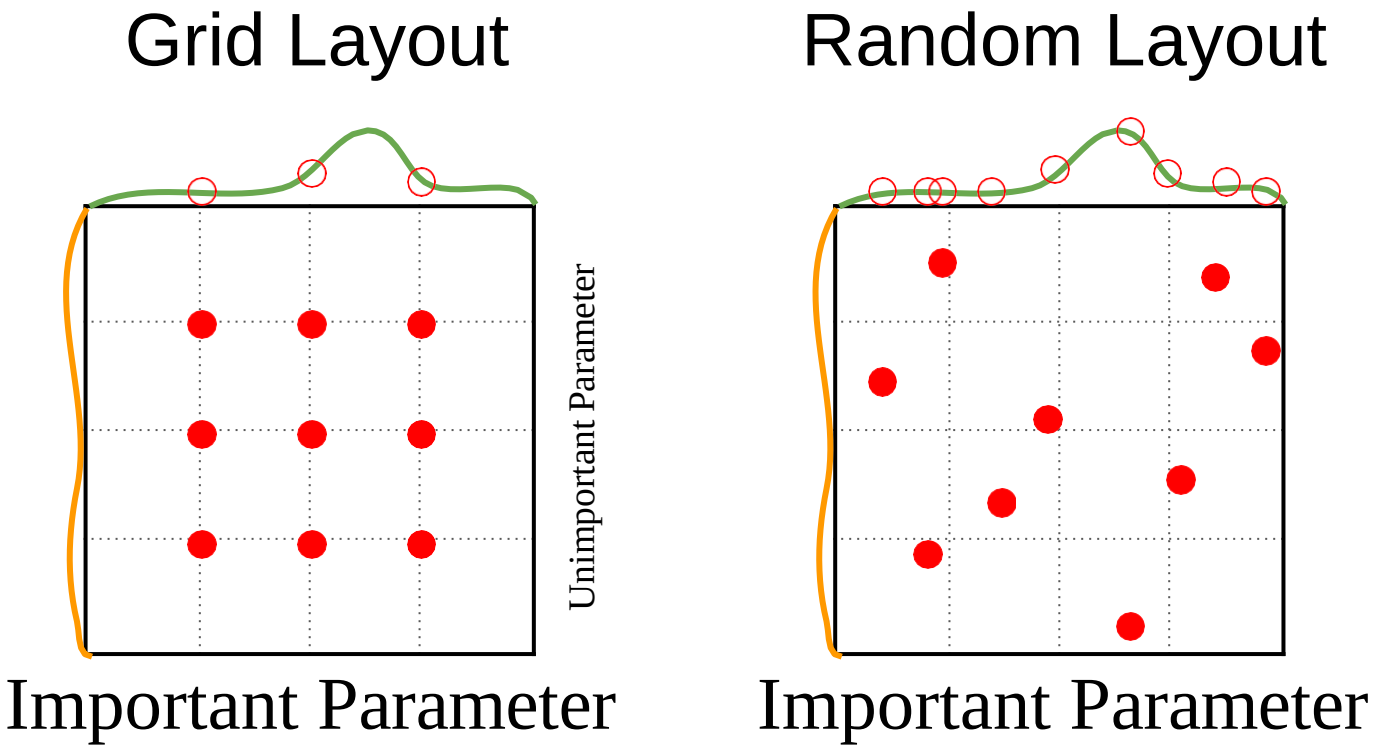
```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

But this best cross-validation result is worrying. Why?

Random Search vs. Grid Search

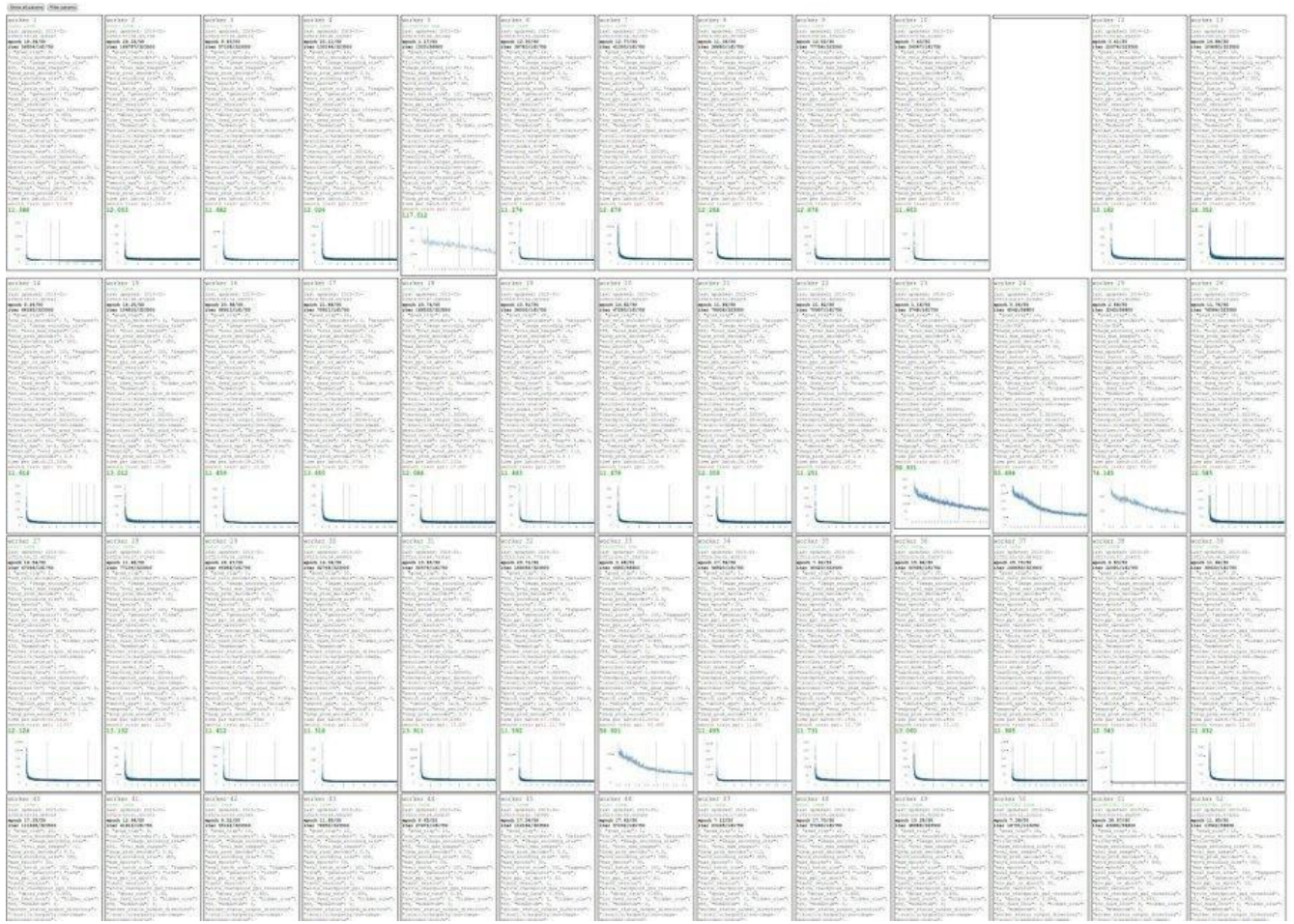


Hyperparameters to play with:

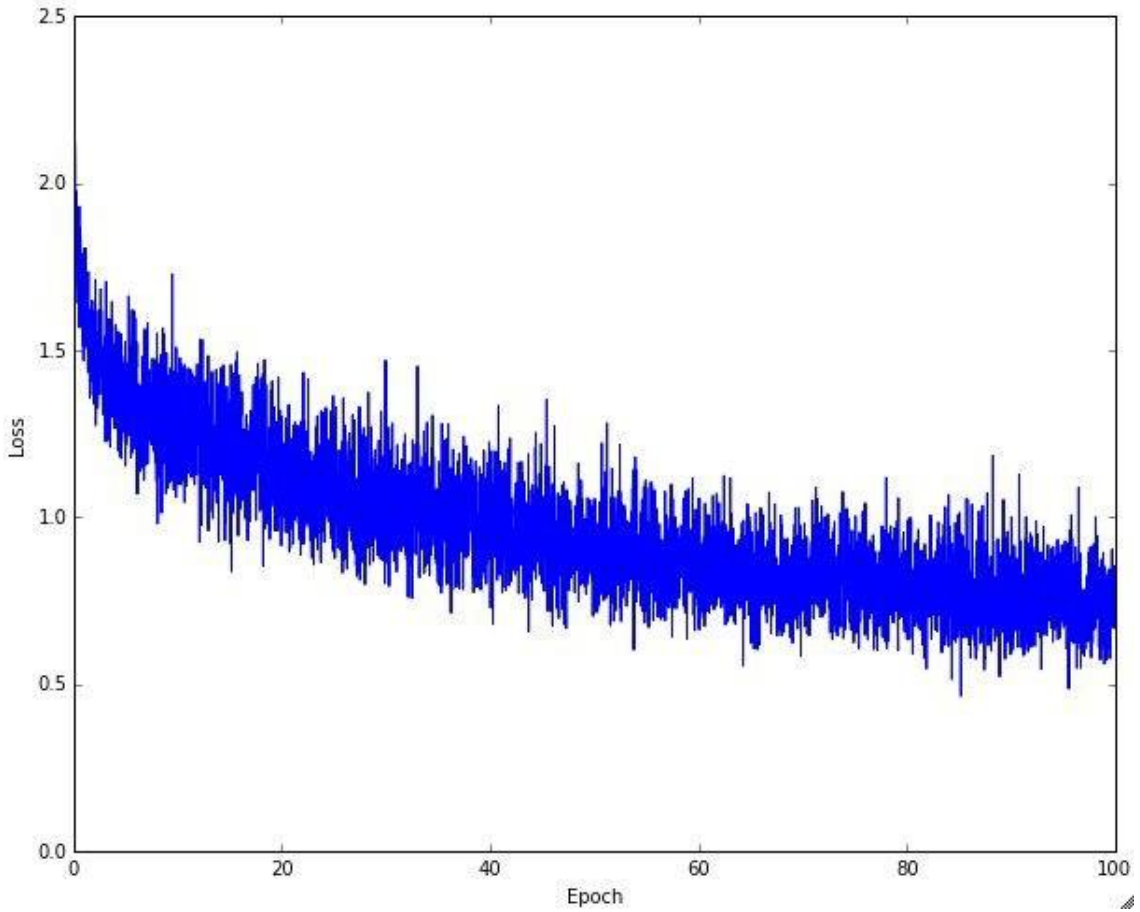
- Network architecture
- Learning rate, its decay schedule, update type
- Regularization (L2/Dropout strength)

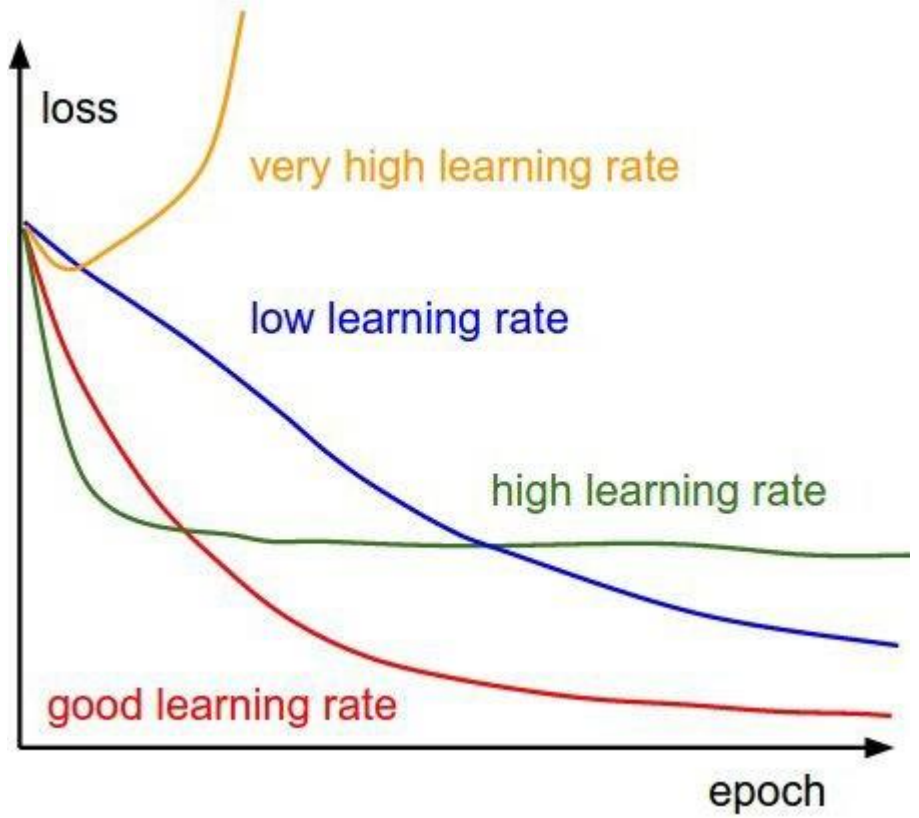
neural networks practitioner
music = loss function

Cross-validation “command center”

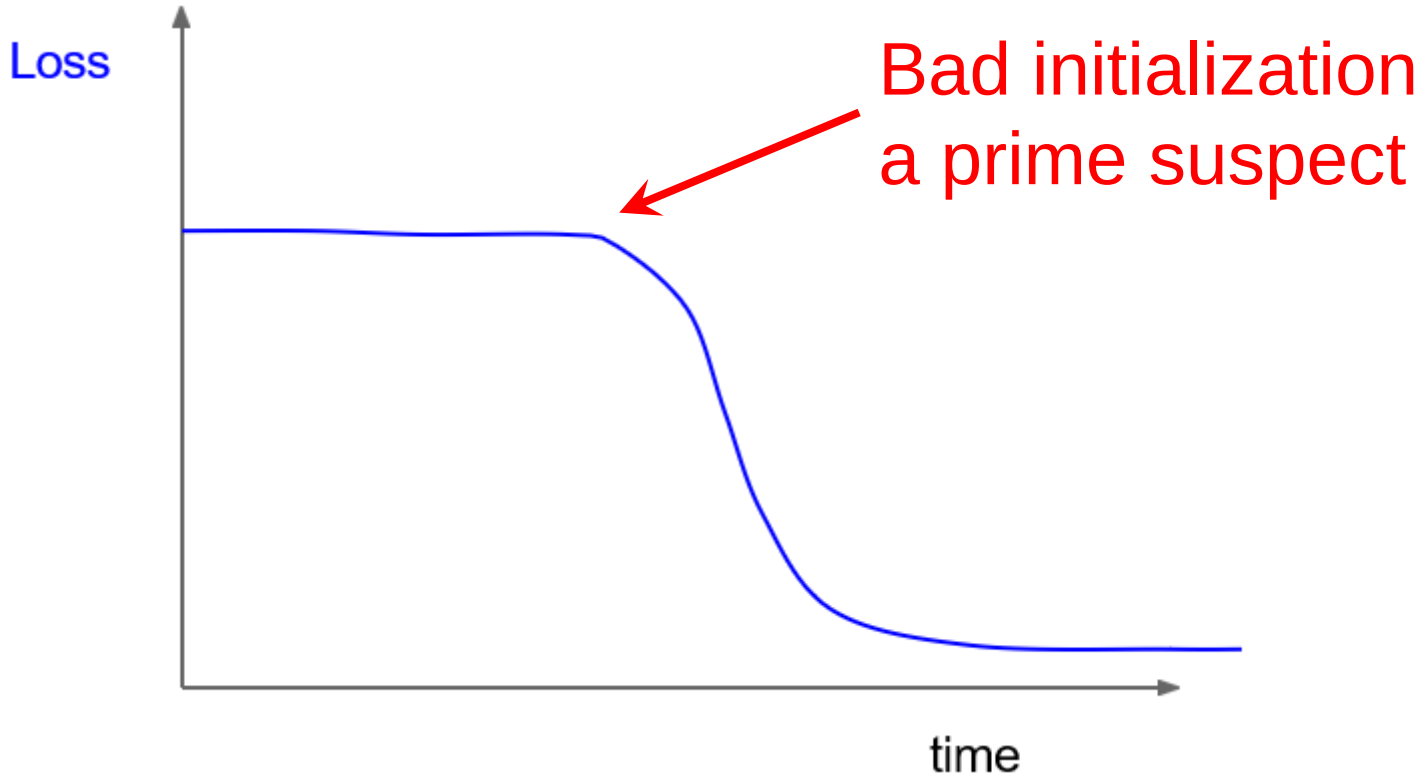


Monitor and visualize the loss curve

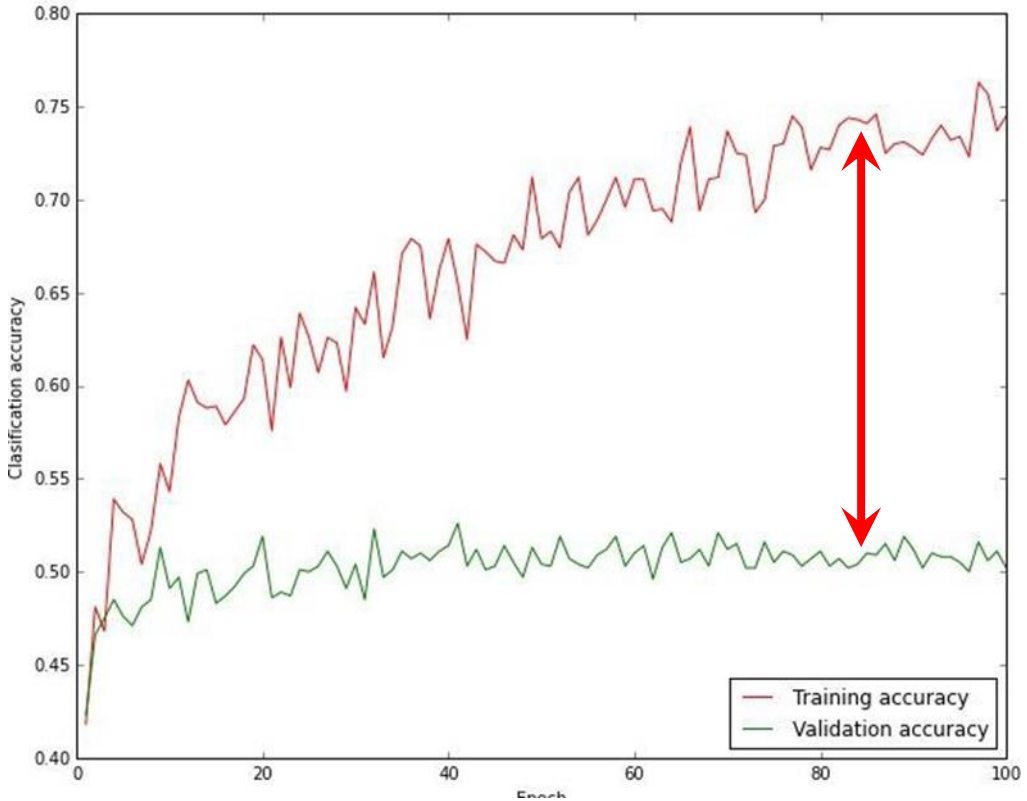




Monitor and visualize the loss curve



Monitor and visualize the accuracy:



big gap = overfitting $\Longrightarrow$ increase regularization strength?

no gap $\Longrightarrow$ increase model capacity?

Track the ratio of weight updates / weight magnitudes

- ratio between the updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay) want this to be somewhere around 0.001 or so

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

Summary

- Activation Functions (use $ReLU$).
- Hàm kích hoạt (khuyến khích $ReLU$).
- Data Preprocessing (images: subtract mean).
- Tiền xử lý dữ liệu.
- Weight Initialization (use Xavier init).
- Khởi tạo trọng số.

Summary

- Batch Normalization (use).
- Chuẩn hóa đầu ra trung gian (khuyến cáo sử dụng).
- Babysitting the Learning process.
- Chăm sóc quá trình học tập.
- Hyperparameter Optimization (random sample hyperparams, in log space when appropriate).
- Tối ưu hóa siêu tham số.

NEXT TIME:

Next: Training Neural Networks, Part 2

- Parameter update schemes.
- Cập nhật tham số.
- Learning rate schedules
- Tốc độ học.
- Gradient checking
- Kiểm tra đạo hàm.
- Regularization (Dropout etc.)
- Chính quy hóa (bỏ học nặng).
- Evaluation (Ensembles etc.)
- Đánh giá.
- Transfer learning / fine-tuning.
- Chuyển giao, tinh chỉnh.

Chúc các bạn học tốt

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN TP.HCM

Nhóm UIT-Together

TS. Nguyễn Tấn Trần Minh Khang