ECE 57000

Neural Networks

Chaoyue Liu

Fall 2024

Motivation

Human brain is a perfect example of a “model” with intelligence

The brain is principally composed of about **10 billion** (biological) neurons (cells), each connected to about 10,000 other neurons.

- The brain is a network made by these neurons and their connections
- Each neuron receives electrochemical inputs from other neurons at the dendrites.
- If the sum of these inputs exceeds a threshold, the neuron fires: outputs a signal along the axon.

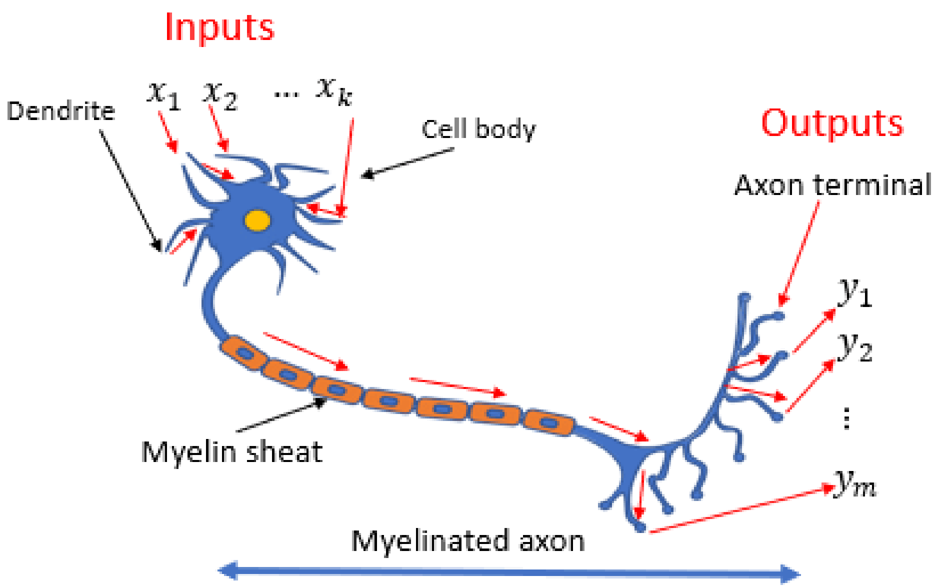
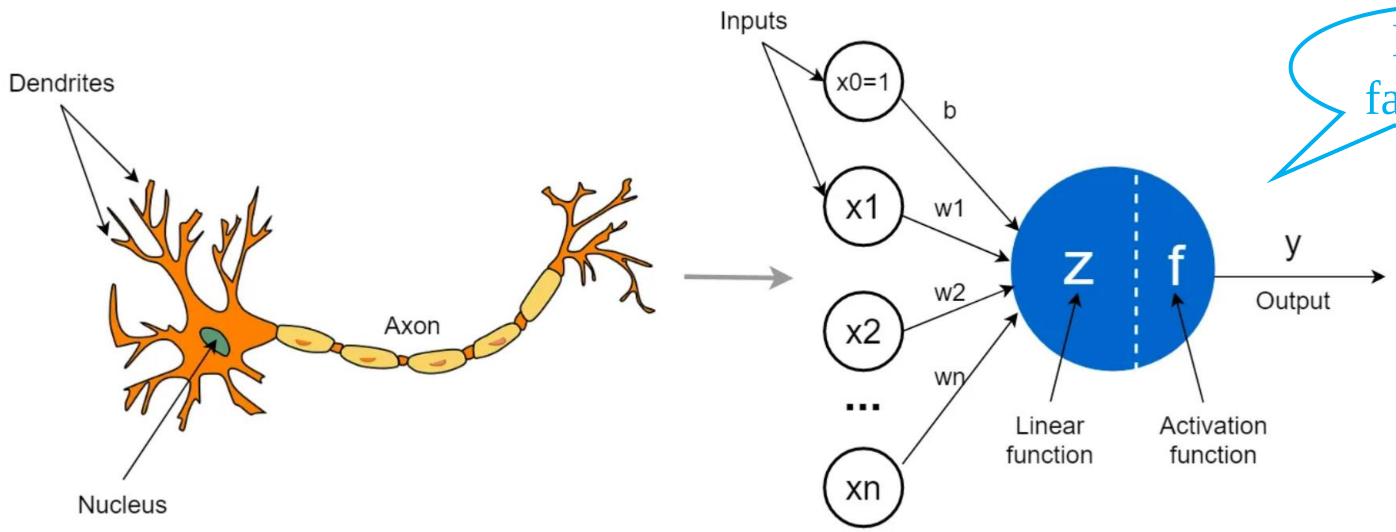


Figure source: Design and Optimization of a Deep Neural Network Architecture for Traffic Light Detection - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Biological-Neuron-Model_fig1_341241129

From biological neuron to artificial neuron

1. The neuron (cell) sums up the inputs
   - can be a weighted sum
2. performs a transformation (activation function)
3. Output a signal for down stream neurons

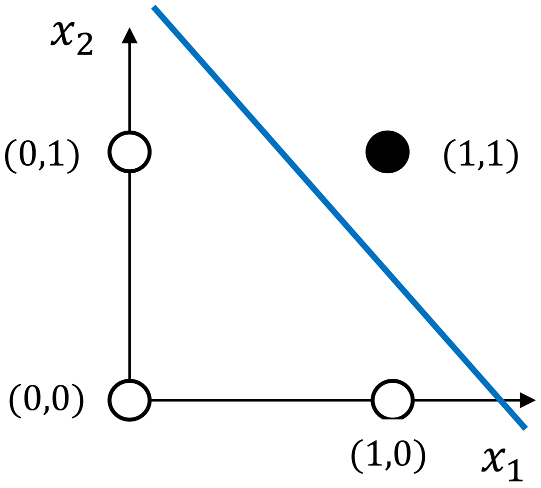


Looks familiar?

Figure source: https://towardsdatascience.com/the-concept-of-artificial-neurons-perceptrons-in-neural-networks-fab22249cbfc

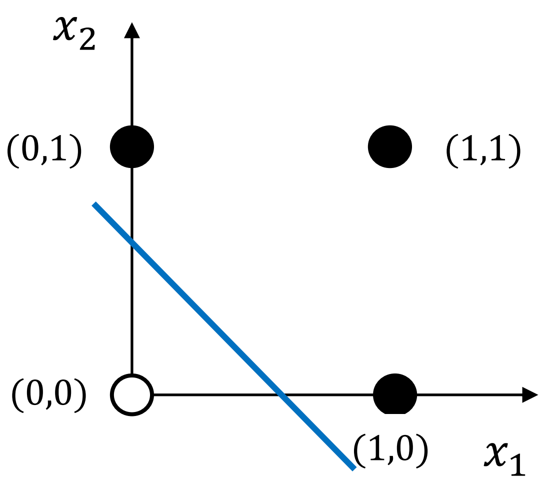
What can a single neuron do?

$y = x_1 \text{ AND } x_2$



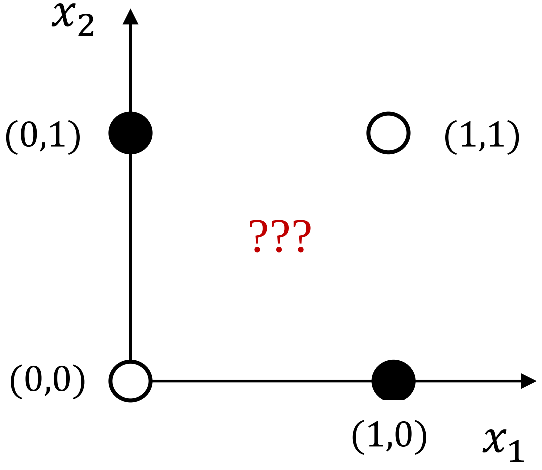
For example:
$f(x_1, x_2) = \sigma(w_1 x_1 + w_2 x_2 + b)$
$w_1 = w_2 = 1, b = -1.5$
$\sigma(z)$ is $\mathbb{I}_{\{z \geq 0\}}$

$y = x_1 \text{ OR } x_2$

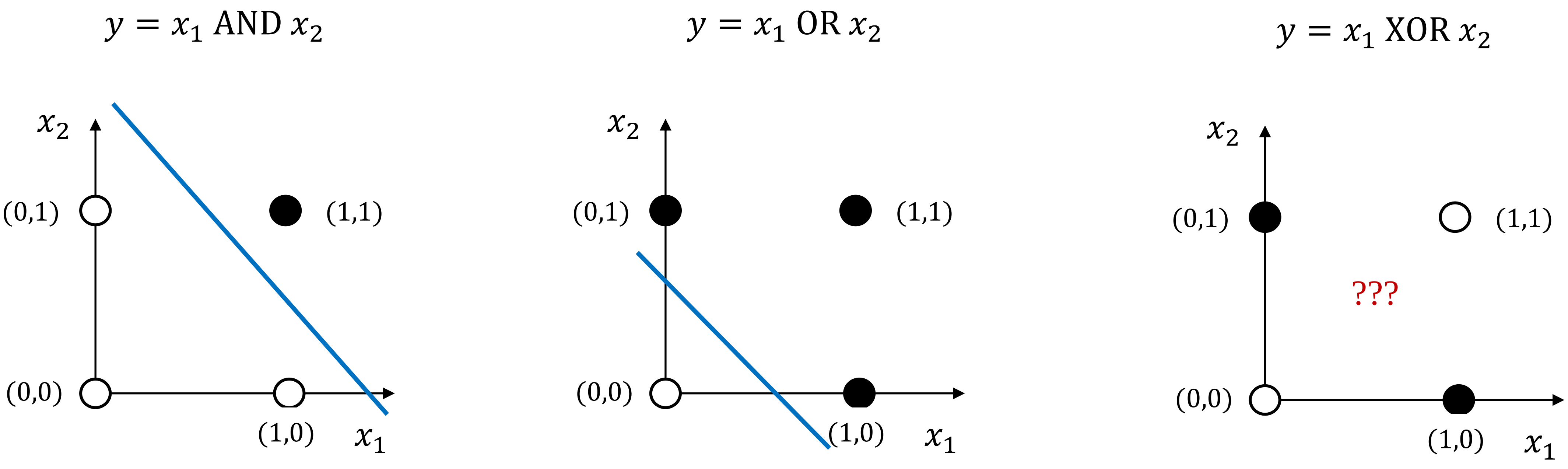


For example:
$f(x_1, x_2) = \sigma(w_1 x_1 + w_2 x_2 + b)$
$w_1 = w_2 = 1, b = -0.5$
σ is $\mathbb{I}_{\{z \geq 0\}}$

$y = x_1 \text{ XOR } x_2$

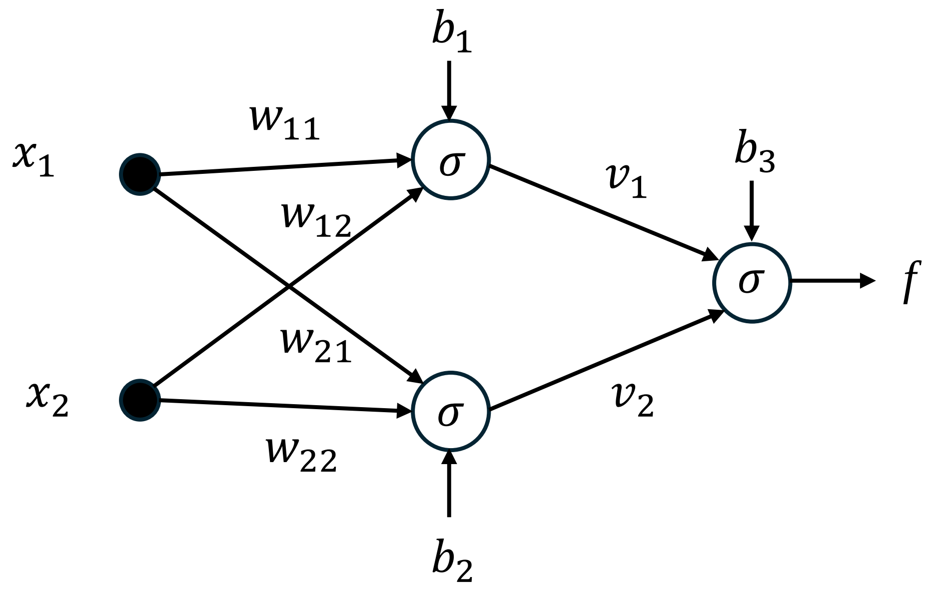


What can a single neuron do?

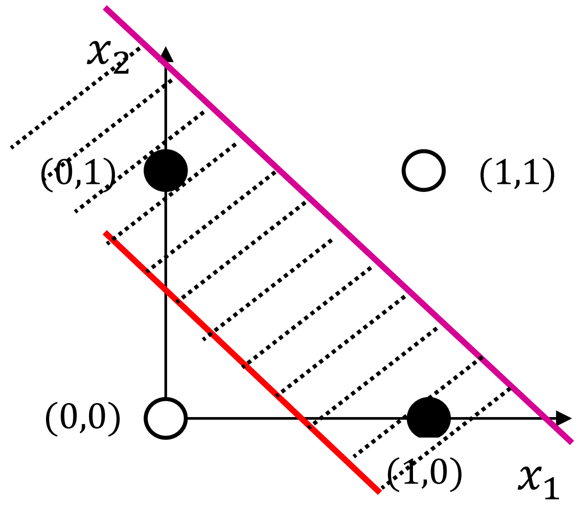


A single neuron is no better than a linear model (i.e., logistic regression)

From a single neuron to a neural network



$y = x_1 \text{ XOR } x_2$



AND: $w_{11} = w_{12} = 1, b_1 = -1.5$

OR: $w_{21} = w_{22} = 1, b_2 = -0.5$

$v_1 = -1, v_2 = 1, b_3 = -0.5$

From a single neuron to a neural network

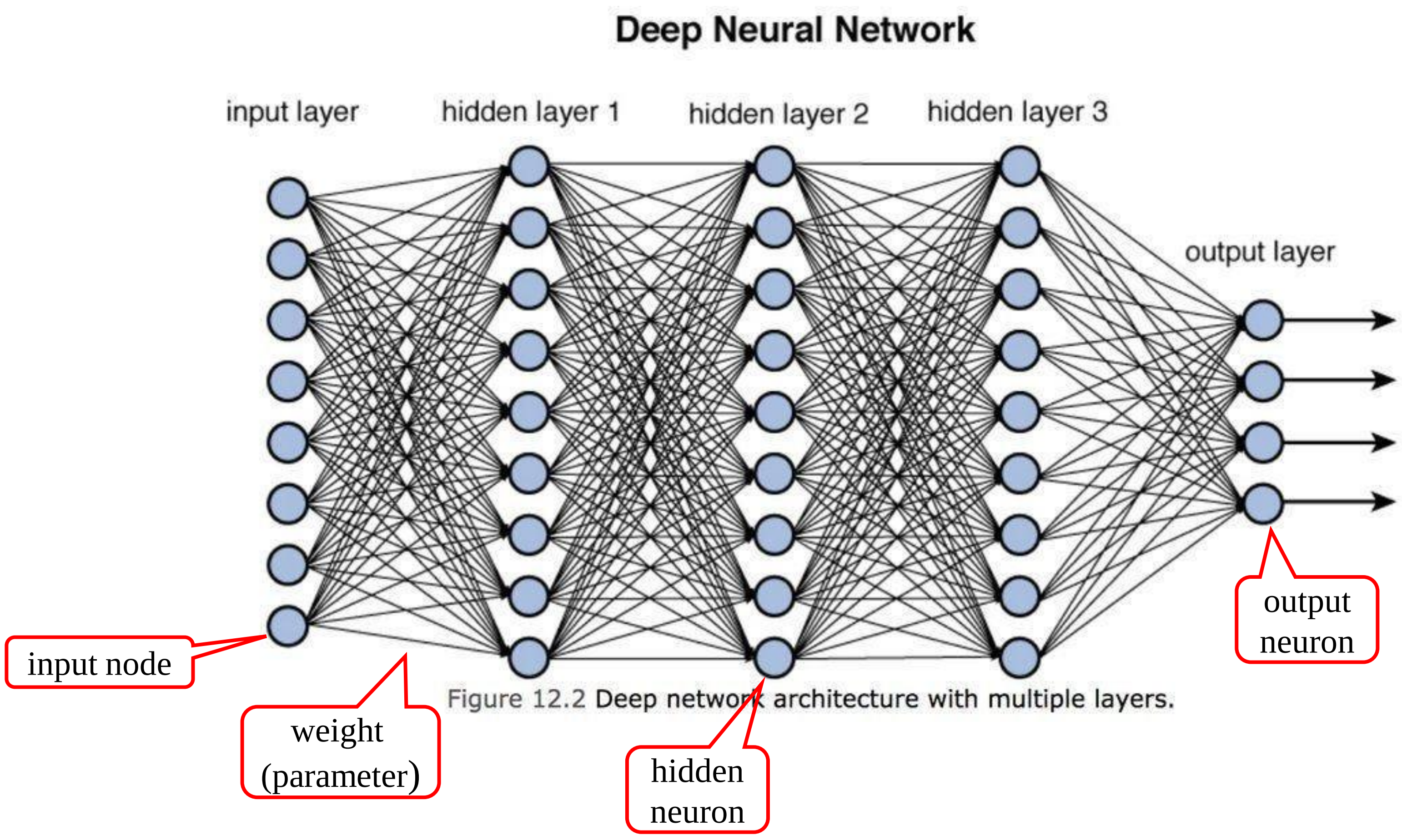


Figure 12.2 Deep network architecture with multiple layers.

Depth: # of layers
Width: # of neurons per hidden layer

Fully-connected neural networks (a.k.a multi-layer perceptron)

Figure source: https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964

Neural network definition

Inference: given an input, compute the neural network's prediction/output

- A forward pass: from input layer to output layer

Input layer: $\mathbf{h}^{(0)} = \mathbf{x}$

hidden layers:

for l in {1,2,⋯,L-1}:

$\tilde{\mathbf{h}}^{(l)} = \mathrm{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$, namely $\tilde{h}_i^{(l)} = \sum_{j=1}^{m_{l-1}} W_{ij}^{(l)} h_j^{(l-1)} + b_i^{(l)}$

$\mathbf{h}^{(l)} = \sigma(\tilde{\mathbf{h}}^{(l)})$, namely $h_i^{(l)} = \sigma(\tilde{h}_i^{(l)})$

output layer:

$\tilde{f}(\mathbf{x}) = \mathrm{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$, namely $\tilde{f}_i(\mathbf{x}) = \sum_{j=1}^{m_{L-1}} W_{ij}^{(L)} h_j^{(L-1)} + b_i^{(L)}$

or, equivalently: $\tilde{f}(\mathbf{x}) = \mathrm{W}^{(L)}\sigma\left(\cdots\sigma\left(\mathrm{W}^{(2)}\sigma\left(\mathrm{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right)\cdots\right) + \mathbf{b}^{(L)}$

Neural network definition

Input layer: $\mathbf{h}^{(0)} = \mathbf{x}$

hidden layers:

for l in {1,2,⋯,L-1}:

$$\tilde{\mathbf{h}}^{(l)} = \mathrm{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{h}^{(l)} = \sigma(\tilde{\mathbf{h}}^{(l)})$$

output layer:

$$\tilde{f}(\mathbf{x}) = \mathrm{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$$

$\tilde{\mathbf{h}}$: pre-activations
$\mathbf{h}$: post-activations
m_l: width, # of neurons at layer l
$\mathrm{W}^{(l)}$: weight matrices, $m_l \times m_{l-1}$
$\mathbf{b}^{(l)}$: bias, m_l-dimensional

(Trainable) parameters: $\{\mathrm{W}^{(l)}, \mathbf{b}^{(l)}\}_{l=1}^{L+1}$, denoted by $\mathbf{w}$

Activation function $\sigma(\cdot)$

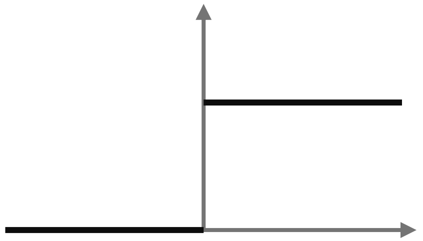
- The activation function is applied element-wise:

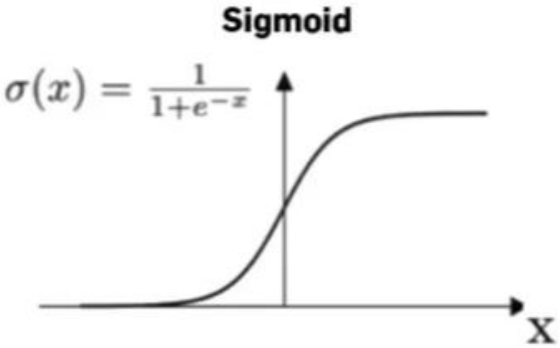
$$h_i^{(l)} = \sigma(\tilde{h}_i^{(l)})$$

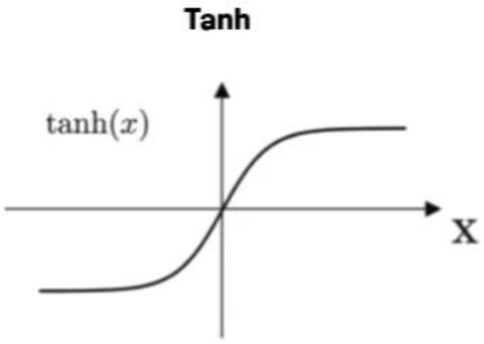
- Activation function is non-linear.

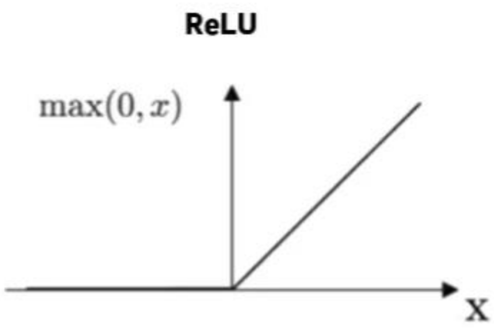
Examples:

Step function









Mathematically smooth

Mostly used in practice

Activation function $\sigma(\cdot)$

Q: why should the activation function be <u>non-linear</u>?

Suppose the activation $\sigma(\cdot)$ is linear: $\sigma(z) = az$, where a is a scalar. Then

$$\sigma\left(\mathrm{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right) = a\mathrm{W}^{(l)}\mathbf{h}^{(l-1)} + a\mathbf{b}^{(l)}$$

Still a linear function

As a consequence, the whole neural network remains a <u>linear model</u>:

$$f(\mathbf{x}) = \mathrm{W}^{(L)}\sigma\left(\cdots\sigma\left(\mathrm{W}^{(2)}\sigma\left(\mathrm{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right)\cdots\right) + \mathbf{b}^{(L)} = \mathrm{W}\mathbf{x} + \mathbf{b}$$

Composition of multiple linear functions is still a linear function

For example: let $\sigma(z) = z$, bias terms $\mathbf{b}^{(l)} = 0$

$$f(\mathbf{x}) = \underbrace{\mathrm{W}^{(L)}\mathrm{W}^{(L-1)}\ldots\mathrm{W}^{(2)}\mathrm{W}^{(1)}}_{\mathrm{W}}\mathbf{x}$$

Universal Approximation Theorem

We have seen in the XOR problem that neural networks perform better than linear model

Universal Approximation Theorem *:

For any continuous function g on a compact domain $\mathcal{X} \in \mathbb{R}^d$ and any positive number $\epsilon > 0$, there is always a neural network f and weights $\mathbf{w}$, such that

$$|f_{\mathbf{w}}(\mathbf{x}) - g(\mathbf{x})| < \epsilon, \forall \mathbf{x} \in \mathcal{X}$$

- Activation function $\sigma(\cdot)$ is required to be continuous, bounded and non-constant
- It may require the network to have a large width
- Does not require a large depth: one hidden layer is OK.

*: Hornik, K., Stinchcombe, M. and White, H. (1989) 'Multilayer feedforward networks are universal approximators', *Neural Networks*, 2(5), pp. 359–366.

Universal Approximation Theorem

Effect of the depth:

To approximate the same target function g on $\mathcal{X}$ with the same precision ϵ, a shallow (one hidden layer) neural network may require exponentially many more neurons than a deep (multi-layer) neural network

More detailed discussion can be found here:
M. Telgarsky. Representation benefits of deep feedforward networks. arXiv preprint arXiv:1509.08101, 2015

Loss functions

Regression:

- $f(\mathbf{w}, \mathbf{x}_i) = \tilde{f}(\mathbf{w}, \mathbf{x}_i)$
- MSE loss: $\mathcal{L}(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}\left(y_i - f(\mathbf{w}, \mathbf{x}_i)\right)^2$

Binary classification:

- Output: $f(\mathbf{w}, \mathbf{x}_i) = \sigma(\tilde{f}(\mathbf{w}, \mathbf{x}_i))$
- Logistic loss: $\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} y_i \log f(\mathbf{w}, \mathbf{x}_i) + (1 - y_i) \log\left(1 - f(\mathbf{w}, \mathbf{x}_i)\right)$

Multi-class classification:

- ???

One-hot encoding

Multi-class classification:

- for example, four classes: $y \in \{\text{"cat"}, \text{"dog"}, \text{"horse"}, \text{"car"}\}$

$$\text{"cat"} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad \text{"dog"} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \qquad \text{"horse"} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \qquad \text{"car"} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

Requirements for neural network

- Dimension of output $f(\mathbf{w}, \mathbf{x})$: # of classes K (K=4, in the example)
- Value of each output neuron $f_k(\mathbf{w}, \mathbf{x}), k \in \{1,2, \cdots, K\}$, is between 0 and 1
- Sum of output neurons $\sum_{k=1}^{K} f_k(\mathbf{w}, \mathbf{x}) = 1$

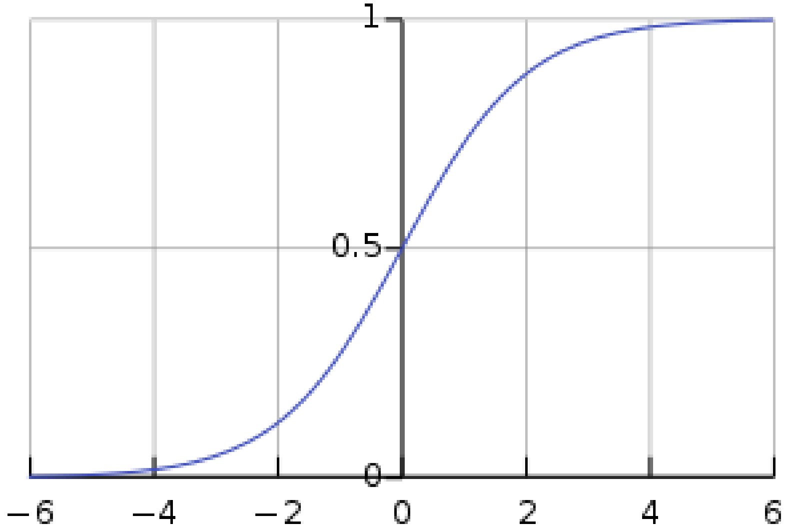
One-hot encoding

Requirements for neural network

- Dimension of output $f(\mathbf{w}, \mathbf{x})$: # of classes K (K=4, in the example)
- Value of each output neuron $f_k(\mathbf{w}, \mathbf{x}), k \in \{1,2,\cdots,K\}$, is between 0 and 1
- Sum of output neurons $\sum_{k=1}^{K} f_k(\mathbf{w}, \mathbf{x}) = 1$

Sigmoid function? $\sigma(a) = \dfrac{1}{1 + e^{-a}}$

No!



Softmax

The *Softmax* function:

- Given a vector $\mathbf{o} = (o_1, o_2, \cdots, o_K)^T$

$$softmax(\mathbf{o}) = \frac{1}{\sum_{k=1}^{K} e^{o_k}} (e^{o_1}, e^{o_2}, \cdots, e^{o_K})^T$$

Each element is interpreted as a **probability**

- A generalization of *sigmoid*
  - Binary classification *K*=2

$$softmax(\mathbf{o}) = \frac{1}{e^{o_1} + e^{o_2}} (e^{o_1}, e^{o_2})^T$$
$$= \left(\frac{1}{1 + e^{-(o_1 - o_2)}}, \frac{e^{-(o_1 - o_2)}}{1 + e^{-(o_1 - o_2)}} \right)^T$$
$$= \left(\sigma(o_1 - o_2), 1 - \sigma(o_1 - o_2) \right)^T$$

Note: binary classification has only one output, instead of two.

Cross-entropy loss

Maximizing log likelihood:

$$\mathcal{L}_{CE}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{\sum_{k=1}^{K} -y_{i,k} \log f_k(\mathbf{w}; x_i)}_{\ell(\mathbf{w}; x_i, y_i)}$$

- A generalization of *logistic loss*
  - Binary classification *K*=2

$$\mathcal{L}_{CE}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} -y_i \log f(\mathbf{w}; x_i) - (1 - y_i) \log f(\mathbf{w}; x_i)$$

Derivative of cross-entropy loss

Binary classification

$$\tilde{f} \xrightarrow{sigmoid} f \longrightarrow \text{logistic loss } \ell$$

$$\frac{d\ell}{d\tilde{f}} = \frac{d\ell}{df} \cdot \frac{df}{d\tilde{f}} = y - f$$

Derivative of logistic loss

Derivative of *sigmoid*

Multi-class classification

$$\tilde{f}_k \xrightarrow{softmax} f_k \longrightarrow \ell_{CE}$$

$$\frac{d\ell_{CE}}{d\tilde{f}} = y - f$$

A vector of dimension K

Backpropagation

Goal: compute the gradients $\nabla_{\mathbf{w}}\mathcal{L}$, i.e.,

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}}, \frac{\partial \mathcal{L}}{\partial b^{(l)}}, \forall l = 1,2,\cdots,L$$

- Loss functions $\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \ell(\mathbf{w}; \mathbf{x}_i, y_i)$

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial \ell_i}{\partial W^{(l)}}, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial \ell_i}{\partial \mathbf{b}^{(l)}}$$

- Using the *chain rule*,

$$\frac{\partial \ell}{\partial W^{(l)}} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial \tilde{f}} \cdot \frac{\partial \tilde{f}}{\partial \mathbf{h}^{(L)}} \cdot \frac{\partial \mathbf{h}^{(L)}}{\partial \tilde{\mathbf{h}}^{(L)}} \cdot \frac{\partial \tilde{\mathbf{h}}^{(L)}}{\partial \mathbf{h}^{(L-1)}} \cdots \frac{\partial \tilde{\mathbf{h}}^{(l)}}{\partial W^{(l)}}$$

$$\frac{\partial \ell}{\partial \mathbf{b}^{(l)}} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial \tilde{f}} \cdot \frac{\partial \tilde{f}}{\partial \mathbf{h}^{(L)}} \cdot \frac{\partial \mathbf{h}^{(L)}}{\partial \tilde{\mathbf{h}}^{(L)}} \cdot \frac{\partial \tilde{\mathbf{h}}^{(L)}}{\partial \mathbf{h}^{(L-1)}} \cdots \frac{\partial \tilde{\mathbf{h}}^{(l)}}{\partial \mathbf{b}^{(l)}}$$

input layer: $\mathbf{h}^{(0)} = \mathbf{x}$
hidden layers:
  for l in {1,2,⋯,L-1}:
    $\tilde{\mathbf{h}}^{(l)} = \mathrm{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$
    $\mathbf{h}^{(l)} = \sigma(\tilde{\mathbf{h}}^{(l)})$
output layer:
  $\tilde{f}(\mathbf{x}) = \mathrm{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$

Backpropagation

$$\frac{\partial \ell}{\partial W^{(l)}} = \frac{\partial \ell}{\partial f} \cdot \frac{\partial f}{\partial \tilde{f}} \cdot \frac{\partial \tilde{f}}{\partial \mathbf{h}^{(L-1)}} \cdot \frac{\partial \mathbf{h}^{(L-1)}}{\partial \tilde{\mathbf{h}}^{(L-1)}} \cdot \frac{\partial \tilde{\mathbf{h}}^{(L-1)}}{\partial \mathbf{h}^{(L-2)}} \cdot \frac{\partial \mathbf{h}^{(L-2)}}{\partial \tilde{\mathbf{h}}^{(L-2)}} \cdots \frac{\partial \tilde{\mathbf{h}}^{(l+1)}}{\partial \mathbf{h}^{(l)}} \cdot \frac{\partial \mathbf{h}^{(l)}}{\partial \tilde{\mathbf{h}}^{(l)}} \cdot \frac{\partial \tilde{\mathbf{h}}^{(l)}}{\partial \mathrm{W}^{(l)}}$$

$$= \left((f-y)^T \cdot \mathrm{W}^{(L)} \cdot \sigma'\left(\tilde{\mathbf{h}}^{(L-1)}\right) \cdot \mathrm{W}^{(L-1)} \cdot \sigma'\left(\tilde{\mathbf{h}}^{(L-2)}\right) \cdots W^{(l+1)} \sigma'\left(\tilde{\mathbf{h}}^{(l)}\right)\right)^T \cdot \left(\mathbf{h}^{(l-1)}\right)^T$$

$\mathbb{R}^{1\times K}$ $\mathbb{R}^{K\times m_{L-1}}$ $\mathbb{R}^{m_{L-1}\times m_{L-1}}$ $\mathbb{R}^{m_{L-1}\times m_{L-2}}$ $\mathbb{R}^{m_{L-2}\times m_{L-2}}$ $\mathbb{R}^{m_l\times m_l}$ $\mathbb{R}^{1\times m_{l-1}}$

$\sigma'\left(\tilde{\mathbf{h}}^{(l)}\right)$: a diagonal matrix, with entries $\sigma'\left(\tilde{\mathbf{h}}^{(l)}\right)_{ii} = \sigma'(\tilde{h}_i{}^{(l)})$

Similarly,

$$\frac{\partial \ell}{\partial \mathbf{b}^{(l)}} = \left((f-y)^T \cdot \mathrm{W}^{(L)} \cdot \sigma'\left(\tilde{\mathbf{h}}^{(L-1)}\right) \cdot \mathrm{W}^{(L-1)} \cdot \sigma'\left(\tilde{\mathbf{h}}^{(L-2)}\right) \cdots \sigma'\left(\tilde{\mathbf{h}}^{(l)}\right)\right)^T$$

input layer: $\mathbf{h}^{(0)} = \mathbf{x}$
hidden layers:
for l in {1,2,⋯,L-1}:
$\tilde{\mathbf{h}}^{(l)} = \mathrm{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$
$\mathbf{h}^{(l)} = \sigma(\tilde{\mathbf{h}}^{(l)})$
output layer:
$\tilde{f}(\mathbf{x}) = \mathrm{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$

Backpropagation

Observation: many factors are shared

- between $\frac{\partial \ell}{\partial W^{(l)}}$ and $\frac{\partial \ell}{\partial \mathbf{b}^{(l)}}$
- Between different layers l

$$\frac{\partial \ell}{\partial W^{(l)}} = \Big((f-y)^T \cdot \mathrm{W}^{(L)} \cdot \sigma'\big(\tilde{\mathbf{h}}^{(L-1)}\big) \cdot \mathrm{W}^{(L-1)} \cdot \sigma'\big(\tilde{\mathbf{h}}^{(L-2)}\big) \cdots \mathrm{W}^{(l)} \sigma'\big(\tilde{\mathbf{h}}^{(l)}\big)\Big)^T \cdot \big(\mathbf{h}^{(l-1)}\big)^T$$

$$\frac{\partial \ell}{\partial \mathbf{b}^{(l)}} = \Big(\underbrace{\underbrace{\underbrace{\underbrace{(f-y)^T}_{\delta^{(L)}} \cdot \mathrm{W}^{(L)} \cdot \sigma'\big(\tilde{\mathbf{h}}^{(L-1)}\big)}_{\delta^{(L-1)}} \cdot \mathrm{W}^{(L-1)} \cdot \sigma'\big(\tilde{\mathbf{h}}^{(L-2)}\big)}_{\delta^{(L-2)}} \cdots \mathrm{W}^{(l)} \sigma'\big(\tilde{\mathbf{h}}^{(l)}\big)}_{\delta^{(l)}}\Big)^T$$

Backpropagation

Procedure:

1: forward pass: compute and store all $\tilde{\mathbf{h}}^{(l)}(\mathbf{x}), \mathbf{h}^{(l)}(\mathbf{x}), f(\mathbf{x})$

2: backward pass:

$$\delta^{(L)} = f(\mathbf{x}) - y$$

for l in $\{L-1, \cdots, 2, 1\}$:

$$\delta^{(l)} = \delta^{(l+1)} \cdot \mathrm{W}^{(l+1)} \cdot \sigma'\left(\tilde{\mathbf{h}}^{(l)}(\mathbf{x})\right)$$

//*compute gradients for each component: *//

$$\frac{\partial \ell}{\partial W^{(l)}} = (\delta^{(l)})^T \cdot \left(\mathbf{h}^{(l-1)}\right)^T; \quad \frac{\partial \ell}{\partial \mathbf{b}^{(l)}} = \left(\delta^{(l)}\right)^T$$

input layer: $\mathbf{h}^{(0)} = \mathbf{x}$

hidden layers:

for l in {1,2,⋯,L-1}:

$$\tilde{\mathbf{h}}^{(l)} = \mathrm{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$$

$$\mathbf{h}^{(l)} = \sigma(\tilde{\mathbf{h}}^{(l)})$$

output layer:

$$\tilde{f}(\mathbf{x}) = \mathrm{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$$

Demo of dimension broadcasting in python

Initialization

Q: How to initialize parameters $W^{(l)}, \mathbf{b}^{(l)}$ properly?

Zero initialization: $W^{(l)} = 0, \mathbf{b}^{(l)} = 0$?

No! Zero outputs & zero gradients

input layer: $\mathbf{h}^{(0)} = \mathbf{x}$
hidden layers:
for l in {1,2,⋯,L-1}:
$\tilde{\mathbf{h}}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$
$\mathbf{h}^{(l)} = \sigma(\tilde{\mathbf{h}}^{(l)})$
output layer:
$\tilde{f}(\mathbf{x}) = W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$

output layer: $\delta^{(L)} = f(\mathbf{x}) - y$
hidden layers:
for l in $\{L-1, L-2, \cdots, 2,1\}$:
$\delta^{(l)} = \delta^{(l+1)} \cdot W^{(l+1)} \cdot \sigma'\left(\tilde{\mathbf{h}}^{(l)}(\mathbf{x})\right)$
$\frac{\partial \ell}{\partial W^{(l)}} = (\delta^{(l)})^T \cdot \left(\mathbf{h}^{(l-1)}\right)^T;$
$\frac{\partial \ell}{\partial \mathbf{b}^{(l)}} = \left(\delta^{(l)}\right)^T$

Initialization

Q: How to initialize parameters $W^{(l)}, \mathbf{b}^{(l)}$ <u>properly</u>?

How about $W_{ij}^{(l)} = 1, b_i^{(l)} = 0$ or 1?

No! The same hidden neuron values and gradients at each layer

input layer: $\mathbf{h}^{(0)} = \mathbf{x}$
hidden layers:
for l in {1,2,⋯,L-1}:
$$\tilde{\mathbf{h}}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$$
$$\mathbf{h}^{(l)} = \sigma(\tilde{\mathbf{h}}^{(l)})$$
output layer:
$$\tilde{f}(\mathbf{x}) = W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$$

output layer: $\delta^{(L)} = f(\mathbf{x}) - y$
hidden layers:
for l in $\{L-1, L-2, \cdots, 2,1\}$:
$$\delta^{(l)} = \delta^{(l+1)} \cdot W^{(l+1)} \cdot \sigma'\left(\tilde{\mathbf{h}}^{(l)}(\mathbf{x})\right)$$
$$\frac{\partial \ell}{\partial W^{(l)}} = (\delta^{(l)})^T \cdot \left(\mathbf{h}^{(l-1)}\right)^T;$$
$$\frac{\partial \ell}{\partial \mathbf{b}^{(l)}} = \left(\delta^{(l)}\right)^T$$

Initialization

Q: How to initialize parameters $W^{(l)}, \mathbf{b}^{(l)}$ properly?

How about random initialization?

- Uniform distribution: $\mathcal{U}[-a, a]$
- Gaussian distribution: $\mathcal{N}(\mu, \sigma^2)$
- Each parameter should be initialized i.i.d. (*independent* and *identically distributed*)

Much better!

- Non-zero values at output/hidden layers, non-zero gradient
- Diversity in values/gradients at hidden layers

Kaiming Initialization [1]

For ReLU networks: $W_{ij}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{m_{l-1}}\right)$, i.i.d., $b_i^{(l)} = 0$

Q: What if $W_{ij}^{(l)} \sim \mathcal{N}(0,1)$?

$$\mathbb{E}\left[\tilde{h}_i^{(l)} \mid \tilde{\mathbf{h}}^{(l-1)}\right] = 0$$

$$\begin{aligned}\mathrm{Var}\left[\tilde{h}_i^{(l)}\right] &= \mathbb{E}\left[\left(\sum_{j=1}^{m_{l-1}} W_{ij}^{(l)} \sigma\left(\tilde{h}_j^{(l-1)}\right)\right)^2\right] \\ &= \sum_{j=1}^{m_{l-1}} \mathbb{E}\left[\left(W_{ij}^{(l)}\right)^2\right] \mathbb{E}\left[\sigma^2\left(\tilde{h}_j^{(l-1)}\right)\right] \\ &= m_{l-1} \cdot \frac{1}{2} \cdot \mathrm{Var}\left[\tilde{h}_i^{(l-1)}\right]\end{aligned}$$

input layer: $\mathbf{h}^{(0)} = \mathbf{x}$
hidden layers:
for l in {1,2,⋯,L-1}:
$\tilde{\mathbf{h}}^{(l)} = W^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}$
$\mathbf{h}^{(l)} = \sigma(\tilde{\mathbf{h}}^{(l)})$
output layer:
$\tilde{f}(\mathbf{x}) = W^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)}$

Hence, we need $W_{ij}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{m_{l-1}}\right)$, to make sure $\mathrm{Var}\left[\tilde{h}_i^{(l)}\right] = \mathrm{Var}\left[\tilde{h}_i^{(l-1)}\right]$

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision, pages 1026–1034, 2015.

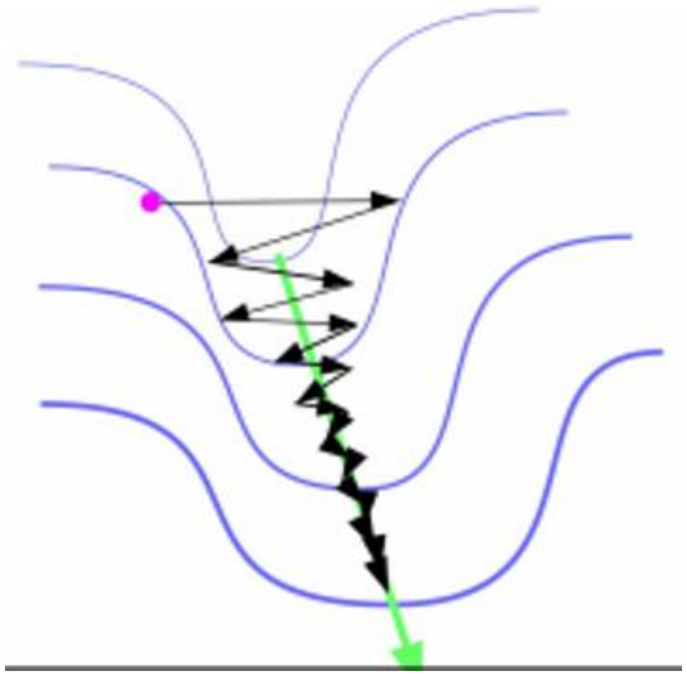
Adaptive learning rate (gradient) algorithms

Motivation:

- Gradients mostly align with “steep eigen-directions”, with small component on “flat eigen-directions”
- Result in slow convergence along “flat eigen-directions”

Idea:

- Have different learning rates in different directions.



Adaptive learning rate (gradient) algorithms

AdaGrad:

$$g_i^{(t+1)} = g_i^{(t)} + \left(\frac{\partial \mathcal{L}(\mathbf{w}^{(t)})}{w_i}\right)^2$$

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{g_i^{(t+1)} + \epsilon}} \cdot \frac{\partial \mathcal{L}(\mathbf{w}^{(t)})}{w_i}$$

RMSProp:

$$g_i^{(t+1)} = \beta \cdot g_i^{(t)} + (1 - \beta) \cdot \left(\frac{\partial \mathcal{L}(\mathbf{w}^{(t)})}{w_i}\right)^2$$

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{g_i^{(t+1)} + \epsilon}} \cdot \frac{\partial \mathcal{L}(\mathbf{w}^{(t)})}{w_i}$$

AdaM:

$$m_i^{(t+1)} = \beta_1 \cdot m_i^{(t)} + (1 - \beta_1) \cdot \frac{\partial \mathcal{L}(\mathbf{w}^{(t)})}{w_i}$$

$$g_i^{(t+1)} = \beta_2 \cdot g_i^{(t)} + (1 - \beta_2) \cdot \left(\frac{\partial \mathcal{L}(\mathbf{w}^{(t)})}{w_i}\right)^2$$

$$\hat{m}_i^{(t+1)} = m_i^{(t+1)} / (1 - \beta_1^{t+1})$$

$$\hat{g}_i^{(t+1)} = g_i^{(t+1)} / (1 - \beta_2^{t+1})$$

$$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta}{\sqrt{\hat{g}_i^{(t+1)} + \epsilon}} \cdot \hat{m}_i^{(t+1)}$$