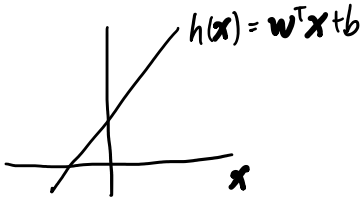
# 11-Nonlinear Models (Neural Networks)

**Linear models**

In previous topics, we mainly dealt with linear models

- Regression $h(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x} + b$



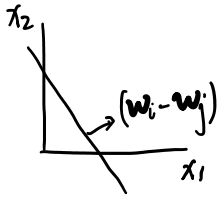

- Classification:

$$h(\boldsymbol{x}) = \operatorname{argmax} \boldsymbol{w}_i^\top \boldsymbol{x} + b_i$$

Category $i$ is referred than Category $j$

$$\boldsymbol{w}_i^\top \boldsymbol{x} + b_i \geq \boldsymbol{w}_j^\top \boldsymbol{x} + b_j \quad \text{i.e., } (\boldsymbol{w}_i - \boldsymbol{w}_j)^\top \boldsymbol{x} + (b_i - b_j) \geq 0$$

Binary classification with logistic regression is a special case of the above.



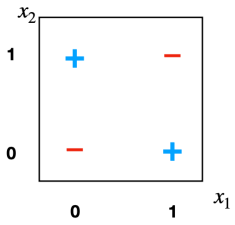

$(\boldsymbol{w}_i - \boldsymbol{w}_j)^\top \boldsymbol{x} + (b_i - b_j)$

Decision boundary between Categories i and j.

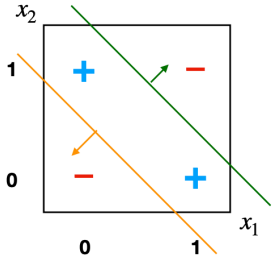
**Nonlinear models**

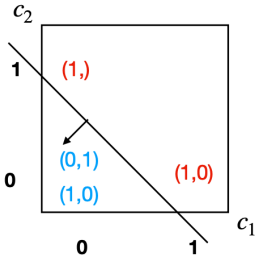
- Non-linear features $\boldsymbol{\phi}(\boldsymbol{x})$
  - E.g., Gaussian discriminant analysis with the different covariance matrices, in which case we have quadratic features of $\boldsymbol{x}$.
- Non-linear kernel $k(\boldsymbol{x}_i, \boldsymbol{x}_j)$
  - A kernel is an inner-product of two data samples that are transformed in a certain vector space. The vector space could be very high-dimensional (e.g., with infinite dimensions). A linear classification in such a high-dimensional space could be non-linear in the original low dimensional space.
- Learnable non-linear mapping
  - We can probably stack a few layers of learnable non-linear functions (e.g., logistic functions) to learn the non-linear feature $\boldsymbol{\phi}(\boldsymbol{x})$ or a non-linear kernel that is appropriate to the task at hand.

## Motivation: XOR, a nonlinear classification problem

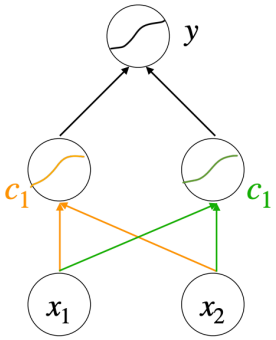



- The problem cannot be solved by logistic regression
- What if we stack multiple logistic regression classifiers?




| Point: | (0,0) | (0,1) | (1,0) | (1,1) |
|---|---|---|---|---|
| Classifier 1: | 1 | 0 | 0 | 0 |
| Classifier 2: | 0 | 0 | 0 | 1 |




- The XOR problem is solvable by three linear classifiers
  - One built upon the other two
  - But this is **programming** [HW]
    - Some machinery that allows you to specify certain things
    - Programming means you specify these things (usually heuristically by human intelligence) that are can be input to the machinery
    - The machinery accomplishes a certain task according to your input (program).
  - Programming is very tedious and only feasible simple tasks
  - We want to **learn** the weights.

- Can we learn the weights?
  - Yes, still by gradient descent.

- Can we compute the gradient?
  - Yes, it's still a differentiable function

$$y = \frac{1}{1 + e^{-(w_0 + w_1 c_1 + w_2 c_2)}}$$

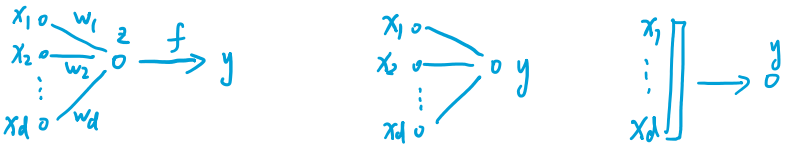
$$= \frac{1}{1+e^{-(w_0+w_1\left\{\frac{1}{1+e^{-(w_{1,0}+w_{1,1}x_1+w_{1,2}x_2)}}\right\}+w_2\left\{\frac{1}{1+e^{-(w_{2,0}+w_{2,1}x_1+w_{2,2}x_2)}}\right\}}}$$

- Again, brute-force computation of gradient is very tedious.
- We need a systematic way of
  - Defining a deep architecture, and
  - Computing its gradient.

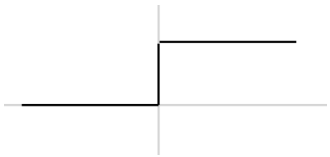
## Artificial Neural Network

- A perceptron [Rosenblatt, 1958]

$$z = \boldsymbol{w}^\top \boldsymbol{x} + b$$
$$y = f(z)$$



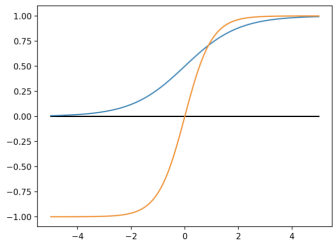
where $f$ is a binary thresholding function



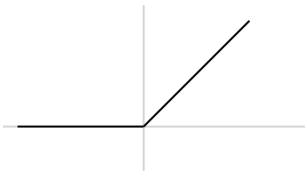
- A perceptron-like neuron, unit, or node

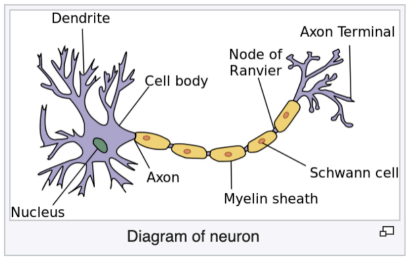
$$z = \boldsymbol{w}^\top \boldsymbol{x} + b$$
$$y = f(z)$$

$f$ is an activation function, e.g., sigmoid, tanh, ReLU




  - Usually we use nonlinear activation
  - Linear activation may be used for regression

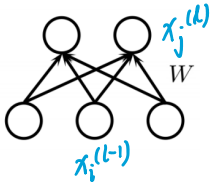


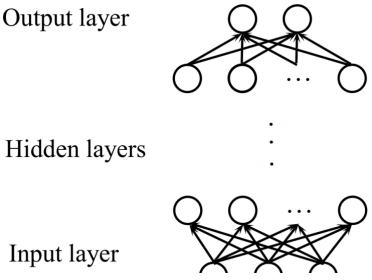



Source: https://en.wikipedia.org/wiki/Neuron

- A multi-layer neural network, or a multi-layer perceptron

A common structure is layer-wise fully connected



For each node $j$ at layer $l$,

$$z_j^{(l)} = \sum_i w_{ji}^{(l)} x_i^{(l-1)} + b_j^{(l)}$$

$$x^{(l)} = f(z_j^{(l)})$$

To simply notations, we omit the layer L, but call the output of the current layer as $y$ and the input of the current layer $x$, which is the output of the lower layer. In the simplified notation,

$z_j = \sum_i w_{ji} x_i + b_j$

$y_j = f(z_j)$

Node-wise notation

$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$

$\mathbf{y} = f(\mathbf{z})$

where $\mathbf{x} \in \mathbb{R}^{N_x}$, $\mathbf{b}, \mathbf{z}, \mathbf{y} \in \mathbb{R}^{N_y}$

$\mathbf{W} \in \mathbb{R}^{N_y \times N_x}$

Vector notation (common in papers)

$\mathbf{Z} = \mathbf{X}\mathbf{W} + \text{repmat}(\mathbf{b})$

$\mathbf{Y} = f(\mathbf{Z})$

where $\mathbf{X} \in \mathbb{R}^{M \times N_x}$

$\mathbf{Z}, \mathbf{Y} \in \mathbb{R}^{M \times N_y}$

$\mathbf{W} \in \mathbb{R}^{N_x \times N_y}$

Batch notation (Common in implementation)

Since we have multiple layers, we need a recursive algorithm that computes the activation of all nodes automatically.

**Forward propagation (FP)**

- Initialization
  $\mathbf{y}^{(0)} = \mathbf{x}$ known
- Recursion
  If $\mathbf{y}^{(l-1)}$ is known, we can compute $\mathbf{y}^{(l)}$
- Termination
  when $\mathbf{y}^{(L)}$ is computed

**Gradient of multi-layer neural networks**

Main idea: if we can compute the gradient for one layer, we may use chain rule to compute the gradient for all layers.

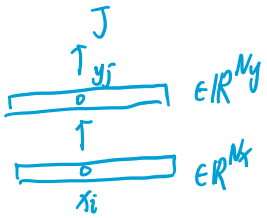
Recursion on what?

Attempt #1 $\frac{\partial J}{\partial \mathbf{W}^{(l)}}$: Bad idea. No direct connection between $\frac{\partial J}{\partial \mathbf{W}^{(l)}}$ and $\frac{\partial J}{\partial \mathbf{W}^{(l-1)}}$

Attempt #2: $\frac{\partial J}{\partial \mathbf{x}^{(l)}}$ or $\frac{\partial J}{\partial \mathbf{z}^{(l)}}$:

Good idea. $\frac{\partial J}{\partial x^{(l-1)}}$ and $\frac{\partial J}{\partial W^{(l)}}$ can be easily computed.

We consider a local layer




Node-wise representation

FP: $\begin{cases} z_j = \sum_i w_{ji} x_i + b_j \\ y_j = f(z_j) \end{cases}$

Gradient: Suppose $\frac{\partial J}{\partial y_j}$ is known

$$\frac{\partial J}{\partial x_i} = \sum_j \frac{\partial J}{\partial z_j} \cdot \frac{\partial z_j}{\partial x_i}$$

$$= \sum_j \underbrace{\frac{\partial J}{\partial y_j}}_{\text{recursive assumption}} \cdot \underbrace{\frac{\partial y_j}{\partial z_j}}_{f' \text{ (if pointwise)}} \cdot \underbrace{\frac{\partial z_j}{\partial x_i}}_{w_{ji}}$$

Note: $\frac{\partial J}{\partial z_j}$ is directed obtained at the output layer (MLE for GLIM)

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial z_j} \cdot x_i \qquad \frac{\partial J}{\partial b_j} = \frac{\partial J}{\partial z_j}$$

Vector representation

FP: $\begin{cases} \mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b} \\ \mathbf{y} = f(\mathbf{z}) \end{cases}$ $\mathbf{x} \in \mathbb{R}^{N_x}$, $\mathbf{y}, \mathbf{z}, \mathbf{b} \in \mathbb{R}^{N_y}$, $\mathbf{W} \in \mathbb{R}^{N_y \times N_x}$

Assume $\frac{\partial J}{\partial \mathbf{y}} = \begin{pmatrix} \frac{\partial J}{\partial y_1} \\ \vdots \\ \frac{\partial J}{\partial y_{N_y}} \end{pmatrix} \in \mathbb{R}^{N_y}$ known.

$$\underbrace{\frac{\partial J}{\partial \mathbf{z}}}_{\in \mathbb{R}^{N_y}} = \underbrace{\frac{\partial J}{\partial \mathbf{y}}}_{\in \mathbb{R}^{N_y}} .* \underbrace{\left(\frac{\partial y_j}{\partial z_j}\right)_{N_y}}_{\in \mathbb{R}^{N_y}}$$

.*: point-wise multiplication

$$\underbrace{\frac{\partial J}{\partial \mathbf{x}}}_{\in \mathbb{R}^{N_x}} = \underbrace{\mathbf{W}^T}_{\in \mathbb{R}^{N_x \times N_y}} \underbrace{\frac{\partial J}{\partial \mathbf{z}}}_{\in \mathbb{R}^{N_y}}$$

$$\underbrace{\frac{\partial J}{\partial \mathbf{W}}}_{N_y \times N_x} = \underbrace{\frac{\partial J}{\partial \mathbf{z}}}_{N_y} \underbrace{\mathbf{x}^T}_{N_x} \qquad \frac{\partial J}{\partial \mathbf{b}} = \frac{\partial J}{\partial \mathbf{z}}$$

Batch representation = HW

**Backpropagation (BP)**

- Initialization $\frac{\partial J}{\partial y_1}, \ldots, \frac{\partial J}{\partial y_{N_y}}$

- Recursion: If $\frac{\partial J}{\partial \mathbf{y}^{(l)}}$ is known, then $\frac{\partial J}{\partial \mathbf{y}^{(l-1)}}$ can be obtained (or $\frac{\partial J}{\partial \mathbf{z}^{(l)}}$)

  Consequently, we can compute $\frac{\partial J}{\partial \mathbf{W}^{(l)}}$ and $\frac{\partial J}{\partial \mathbf{b}^{(l)}}$
- Termination: when $\frac{\partial J}{\partial \mathbf{W}^{(1)}}$ and $\frac{\partial J}{\partial \mathbf{b}^{(1)}}$ are computed

- A few more thoughs
  - Non-layerwise connection: Topological sort
  - Multiple losses: BP is a linear system
  - Tied weights: Total derivative is the summation

**Auto-differentiation in general**

- Input: <Layers, Edges, Losses>
- Algorithm:
  - Topological sort of all layers
  - Apply losses at respective layers
  - For layer $L$ from last to first:

    For each lower layer $\ell$ of $L$

    $\ell$.gradient += BP from $L$

## Numerical gradient checking

- Definition of partial derivative

$$\frac{\partial}{\partial x_i} f(x_1, \cdots, x_i, \cdots x_n) \stackrel{def}{=} \lim_{\delta \to 0} \frac{f(x_1, \cdots, x_i + \delta, \cdots, x_n) - f(x_1, \cdots, x_i, \cdots x_n)}{\delta}$$

- Numerical gradient checking

$$\frac{\partial}{\partial x_i} f(x_1, \cdots, x_i, \cdots x_n) \approx \frac{f(x_1, \cdots, x_i + \delta, \cdots, x_n) - f(x_1, \cdots, x_i - \delta, \cdots, x_n)}{2\delta}$$