COMP 551 – Applied Machine Learning
Lecture 14: Neural Networks

Instructor: Ryan Lowe *(ryan.lowe@mail.mcgill.ca)*

Slides mostly by: Joelle Pineau

Class web page: *www.cs.mcgill.ca/~hvanho2/comp551*

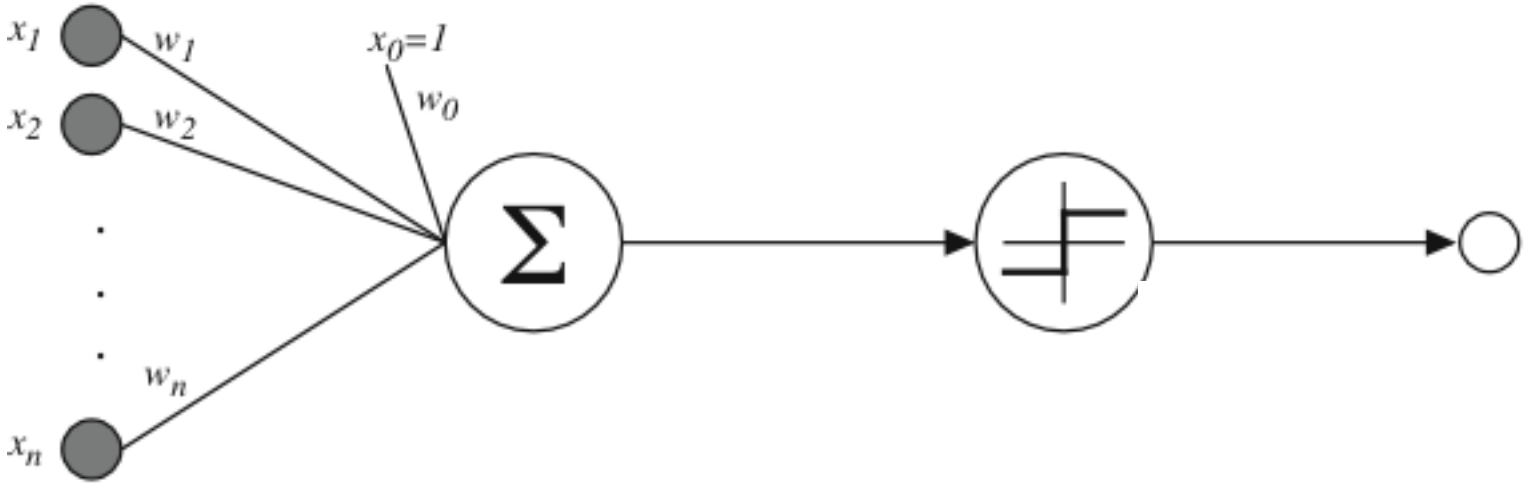
Annnouncements

- **Assignment 3 deadline postponed**
- New deadline: Monday, Feb 26, noon EST

- Questions about assignment 1 grading? See grading TAs during office hours

- My office hours (for now): Monday, 12pm-1pm, MC 232

Recall: the perceptron

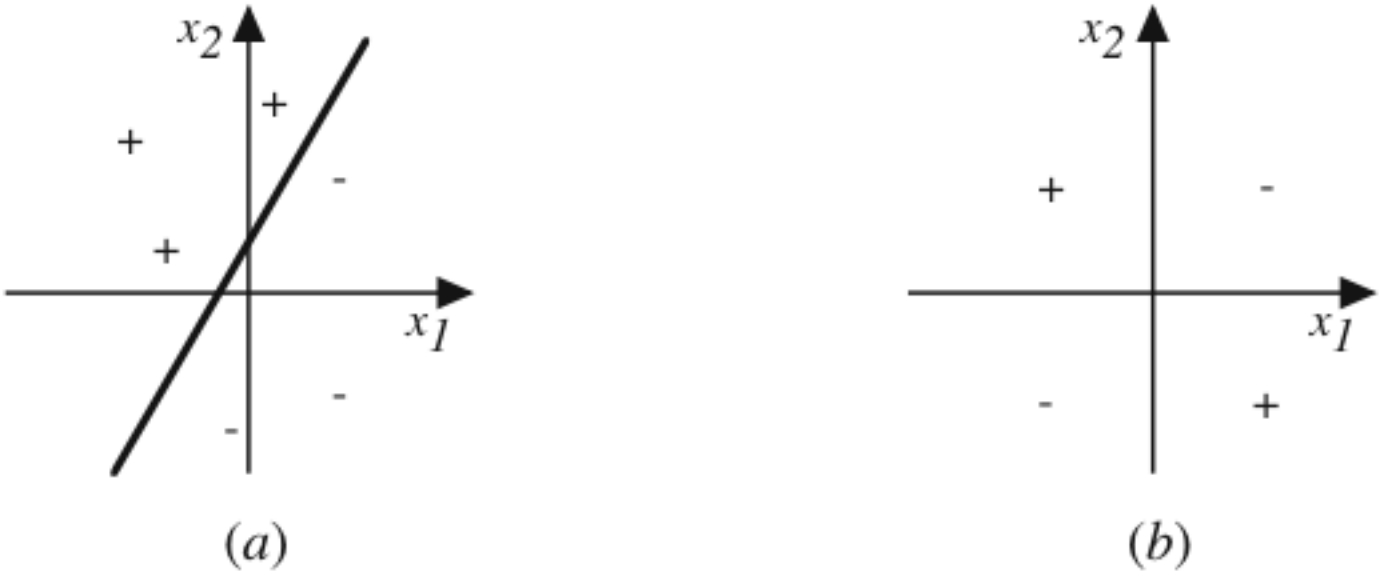


- We can take a linear combination and threshold it:

$$h_{\mathbf{w}}(\mathbf{x}) = \text{sgn}(\mathbf{x} \cdot \mathbf{w}) = \begin{cases} +1 & \text{if } \mathbf{x} \cdot \mathbf{w} > 0 \\ -1 & \text{otherwise} \end{cases}$$

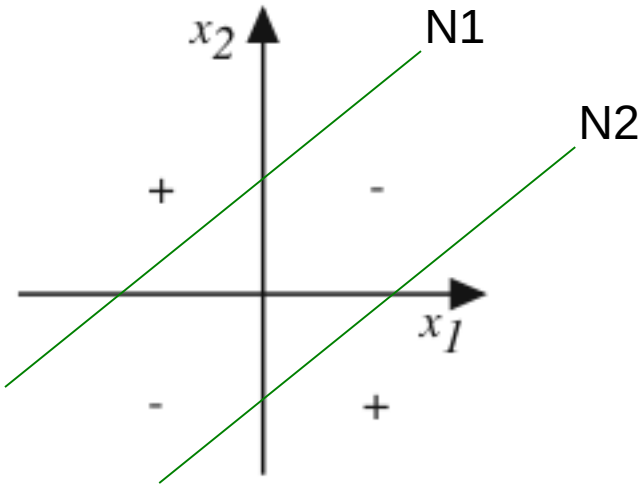
- The output is taken as the predicted class.

Decision surface of a perceptron

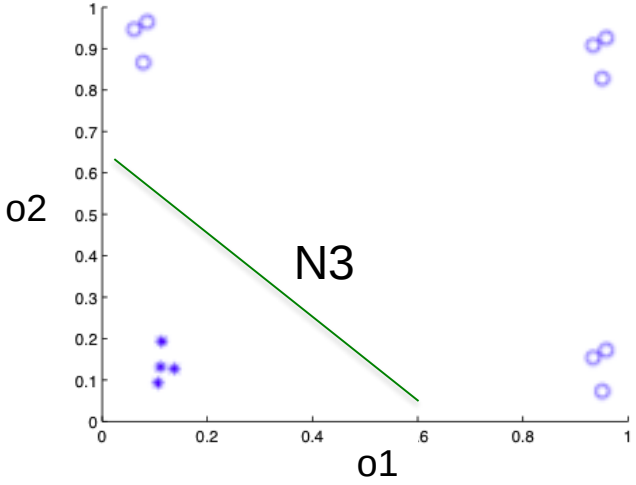


- Can represent many functions.
- To represent non-linearly separable functions (e.g. XOR), we could use a network of perceptron-like elements.
- If we connect perceptrons into networks, the error surface for the network is not differentiable (because of the hard threshold).

Example: A network representing XOR

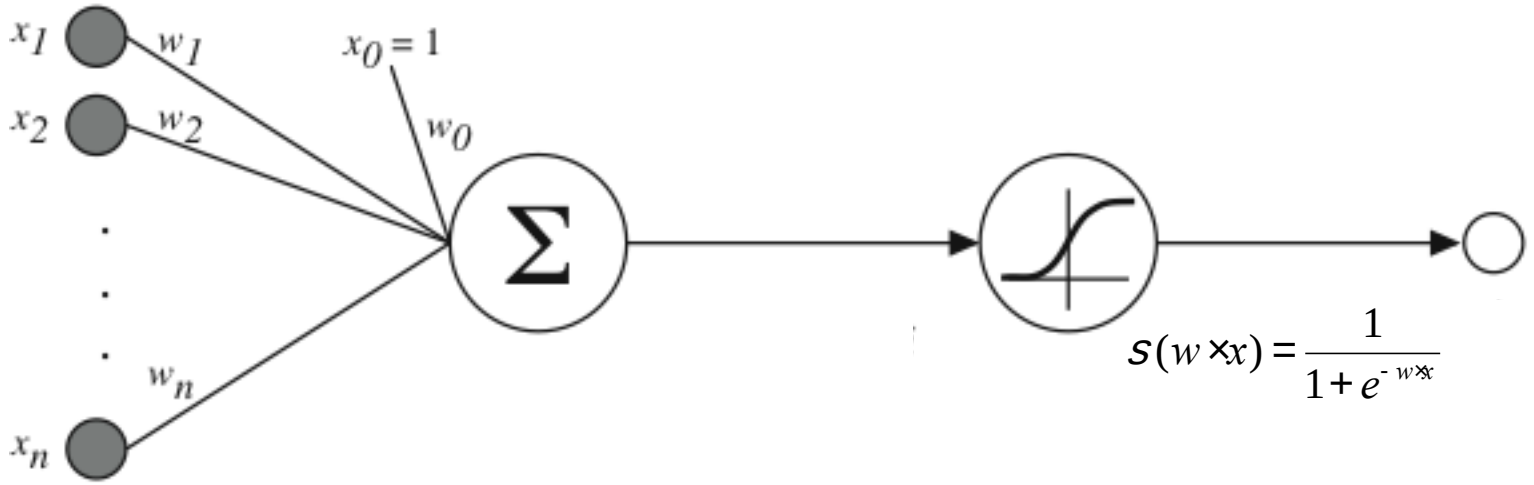


Decision boundary for two neurons in the first hidden layer



Decision boundary for output neuron

Recall the sigmoid function



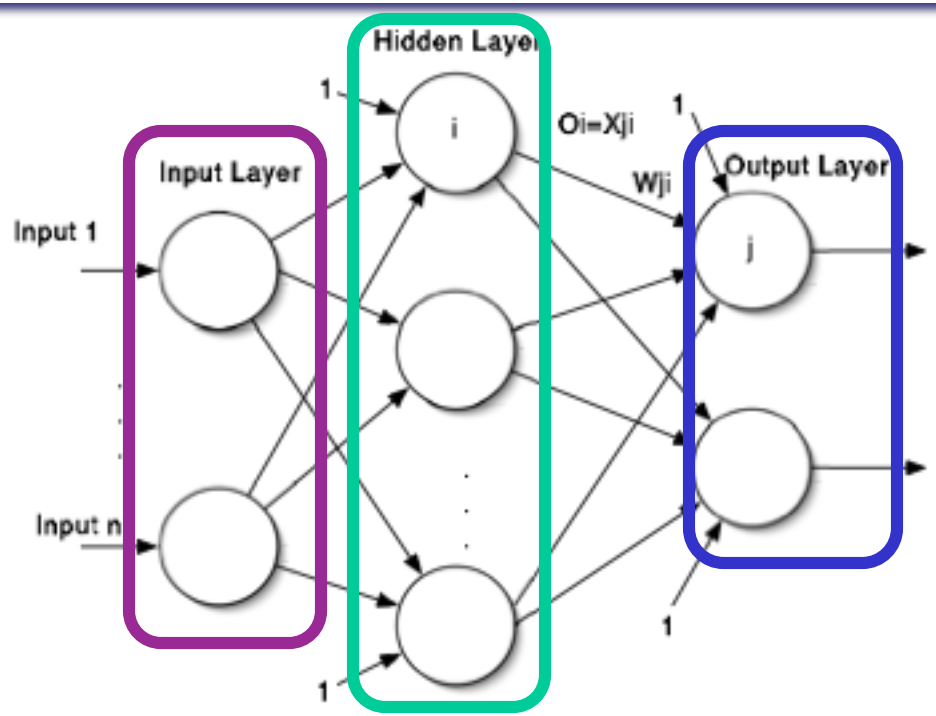
Sigmoid provide "soft threshold", whereas perceptron provides "hard threshold"

- σ is the sigmoid function: $s(z) = \frac{1}{1 + e^{-z}}$
- It has the following nice property: $\frac{ds(z)}{dz} = s(z)(1 - s(z))$

We can derive a **gradient descent rule** to train:

- One sigmoid unit; Multi-layer networks of sigmoid units.

Feed forward neural networks



- A collection of neurons with non-linear activation functions, arranged in layers.
- Layer 0 is the **input layer**, its units just copy the input.
- Last layer (layer K) is the **output layer**, its units provide the output.
- Layers *1, .., K-1* are **hidden layers**, cannot be detected outside of network.

Why this name?

- In **feed-forward networks** the output of units in layer *k* become input to the units in layers *k+1, k+2, …, K*.
- No cross-connection between units in the same layer.
- No backward (“recurrent”) connections from layers downstream.
- Typically, units in layer *k* provide input to units in layer *k+1* only.
- In **fully-connected networks**, all units in layer *k* provide input to all units in layer *k+1*.

Feed-forward neural networks

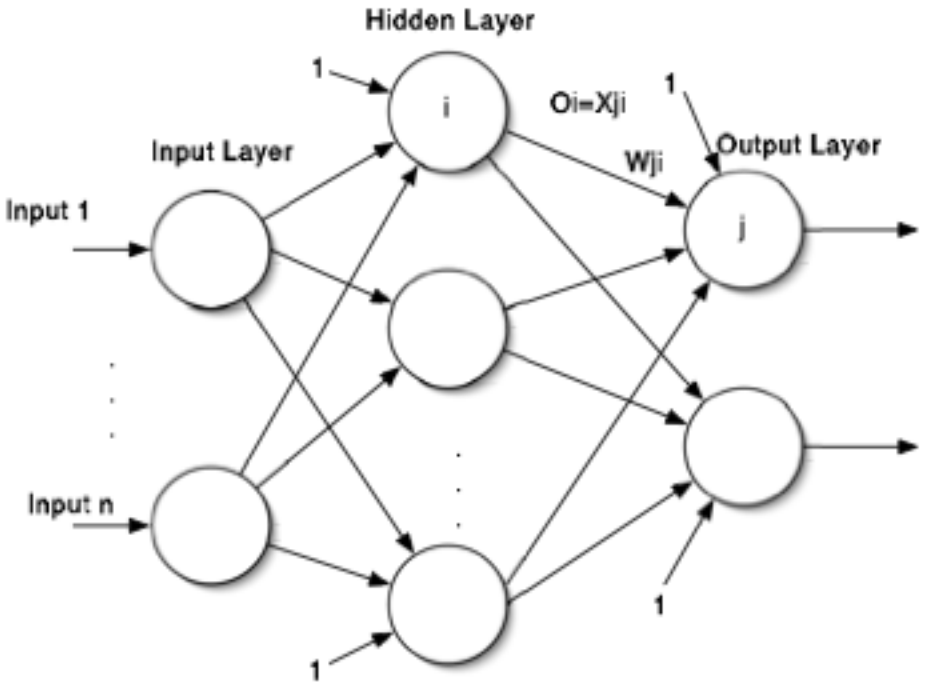
Notation:

- w_{ji} denotes weight on connection from unit i to unit j.
- By convention, $x_{j0} = 1, \ \forall j$
  - Also called bias, **b**
- Output of unit j, denoted o_j is computed using a sigmoid:

$$o_j = \sigma(\boldsymbol{w_j} \cdot \boldsymbol{x_j})$$

where $\boldsymbol{w_j}$ is vector of weights entering unit j

$\boldsymbol{x_j}$ is vector of inputs to unit j



- By definition, $x_{ji} = o_i$.

Given an input, how do we compute the output? How do we train the weights?

Computing the output of the network

- Suppose we want network to make prediction about instance <***x***,*y*=?>.

Run a **forward pass** through the network.

For layer $k = 1 \ldots K$

1. Compute the output of all neurons in layer *k:*

$$o_j = \sigma(\mathbf{w_j} \cdot \mathbf{x_j}), \forall j \in \text{Layer } k$$

2. Copy this output as the input to the next layer:

$$x_{j,i} = o_i, \forall i \in \text{Layer } k, \forall j \in \text{Layer } k+1$$

The output of the last layer is the predicted output *y*.

Learning in feed-forward neural networks

- Assume the network structure (units + connections) is given.
- The learning problem is finding a good set of weights to minimize the error at the output of the network.
- Approach: **gradient descent**, because the form of the hypothesis formed by the network, $h_{\mathbf{w}}$ is:
  - **Differentiable**! Because of the choice of sigmoid units.
  - **Very complex**! Hence direct computation of the optimal weights is not possible.

Gradient-descent preliminaries for NN

- Assume we have a fully connected network:
  - *N* input units (indexed *1, …, N*)
  - *H* hidden units in a single layer (indexed *N+1, …, N+H*)
  - one output unit (indexed *N+H+1*)
- Suppose you want to compute the weight update after seeing instance *<**x**, y>*.
- Let o_i, *i = 1, …, H+N+1* be the outputs of all units in the network for the given input ***x***.
- For regression: the sum-squared error function is:

$$J(\mathbf{w}) = \frac{1}{2}(y - h_{\mathbf{w}}(\mathbf{x}))^2 = \frac{1}{2}(y - o_{N+H+1})^2$$

Gradient-descent update for **output** node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering o_{N+H+1}:
  - Use the chain rule: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

Note: *j* here is any node in the hidden layer

$\partial J(w)/\partial \sigma = -(y - o_{N+H+1})$

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

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -(y - o_{N+H+1}) o_{N+H+1} (1 - o_{N+H+1}) x_{N+H+1,j}$$

Gradient-descent update for **output** node

- Derivative with respects to the weights $w_{N+H+1,j}$ entering o_{N+H+1}:
  - Use the chain rule: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

$$\frac{\partial J}{\partial w_{N+H+1,j}} = -(y - o_{N+H+1}) o_{N+H+1}(1 - o_{N+H+1}) x_{N+H+1,j}$$

- Hence, we can write: $$\frac{\partial J}{\partial w_{N+H+1,j}} = -\delta_{N+H+1} x_{N+H+1,j}$$

where:

$$\delta_{N+H+1} = (y - o_{N+H+1}) o_{N+H+1}(1 - o_{N+H+1})$$

Gradient-descent update for **hidden** node

- The derivative wrt the other weights, $w_{l,j}$ where $j = 1, \ldots, N$ and $l = N+1, \ldots, N+H$ can be computed using chain rule:

$$\frac{\partial J}{\partial w_{l,j}} = -(y - o_{N+H+1}) o_{N+H+1}(1 - o_{N+H+1}) \cdot \frac{\partial}{\partial w_{l,j}}(\mathbf{w}_{N+H+1} \cdot \mathbf{x}_{N+H+1})$$

$$= -\delta_{N+H+1} w_{N+H+1,l} \frac{\partial}{\partial w_{l,j}} x_{N+H+1,l}$$

Note: now j is any node in the **input** layer, and l is any node in the hidden layer

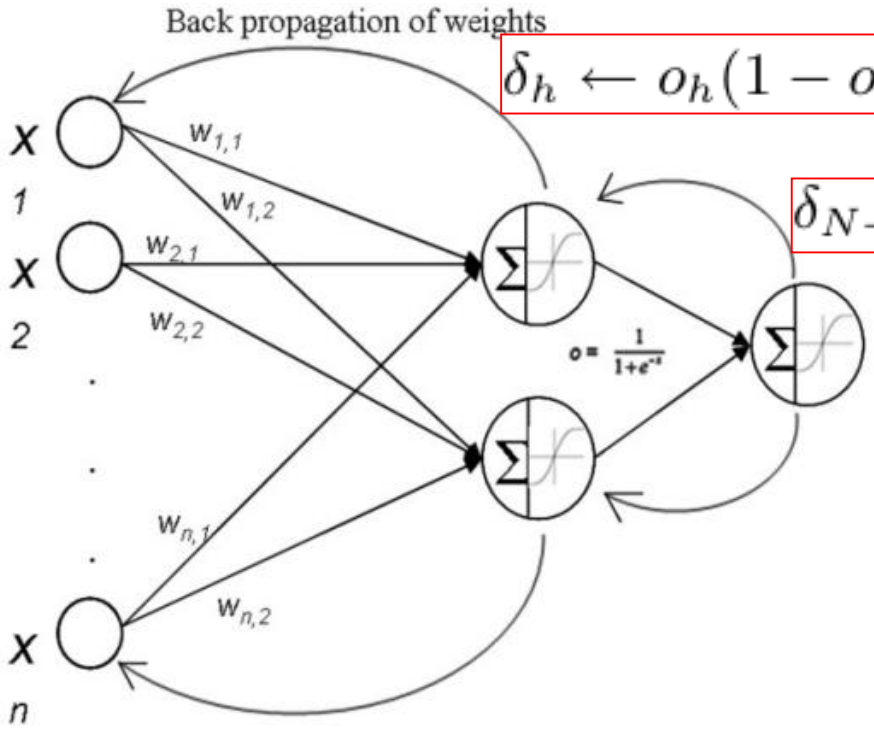
- Recall that $x_{N+H+1,l} = o_l$. Hence we have:

$$\frac{\partial}{\partial w_{l,j}} x_{N+H+1,l} = o_l(1 - o_l) x_{l,j}$$

- Putting these together and using similar notation as before:

$$\frac{\partial J}{\partial w_{l,j}} = -o_l(1 - o_l)\delta_{N+H+1} w_{N+H+1,l} x_{l,j} = -\delta_l x_{l,j}$$

Gradient-descent update for **hidden** node



$$\delta_h \leftarrow o_h(1-o_h)w_{N+H+1,h}\delta_{N+H+1}$$

$$\delta_{N+H+1} \leftarrow o(1-o)(y-o)$$

Note: now *h* is any node in the hidden layer

Image from: http://openi.nlm.nih.gov/detailedresult.php?img=2716495_bcr2257-1&req=4

Stochastic gradient descent (SGD)

- Initialize all weights to small random numbers.

Initialization

- Repeat until convergence:
  - Pick a training example.
  - Feed example through network to compute output $o = o_{N+H+1}$

Forward pass

  - For the output unit, compute the correction:

$$\delta_{N+H+1} \leftarrow o(1-o)(y-o)$$

  - For each hidden unit h, compute its share of the correction:

$$\delta_h \leftarrow o_h(1-o_h)w_{N+H+1,h}\delta_{N+H+1}$$

Backpro-pagation

  - Update each network weight:

$$w_{h,i} \leftarrow w_{h,i} + \alpha_{h,i}\delta_h x_{h,i}$$

Gradient descent

Flavours of gradient descent

- **Stochastic gradient descent**: Compute error on a single example at a time (as in previous slide).

- **Batch gradient descent**: Compute error on all examples.
  - Loop through the training data, accumulating weight changes.
  - Update all weights and repeat.

- **Mini-batch gradient descent**: Compute error on small subset.
  - Randomly select a “mini-batch” (i.e. subset of training examples).
  - Calculate error on mini-batch, apply to update weights, and repeat.

Expressiveness of feed-forward NN

A single sigmoid neuron?

- Same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?

Expressiveness of feed-forward NN

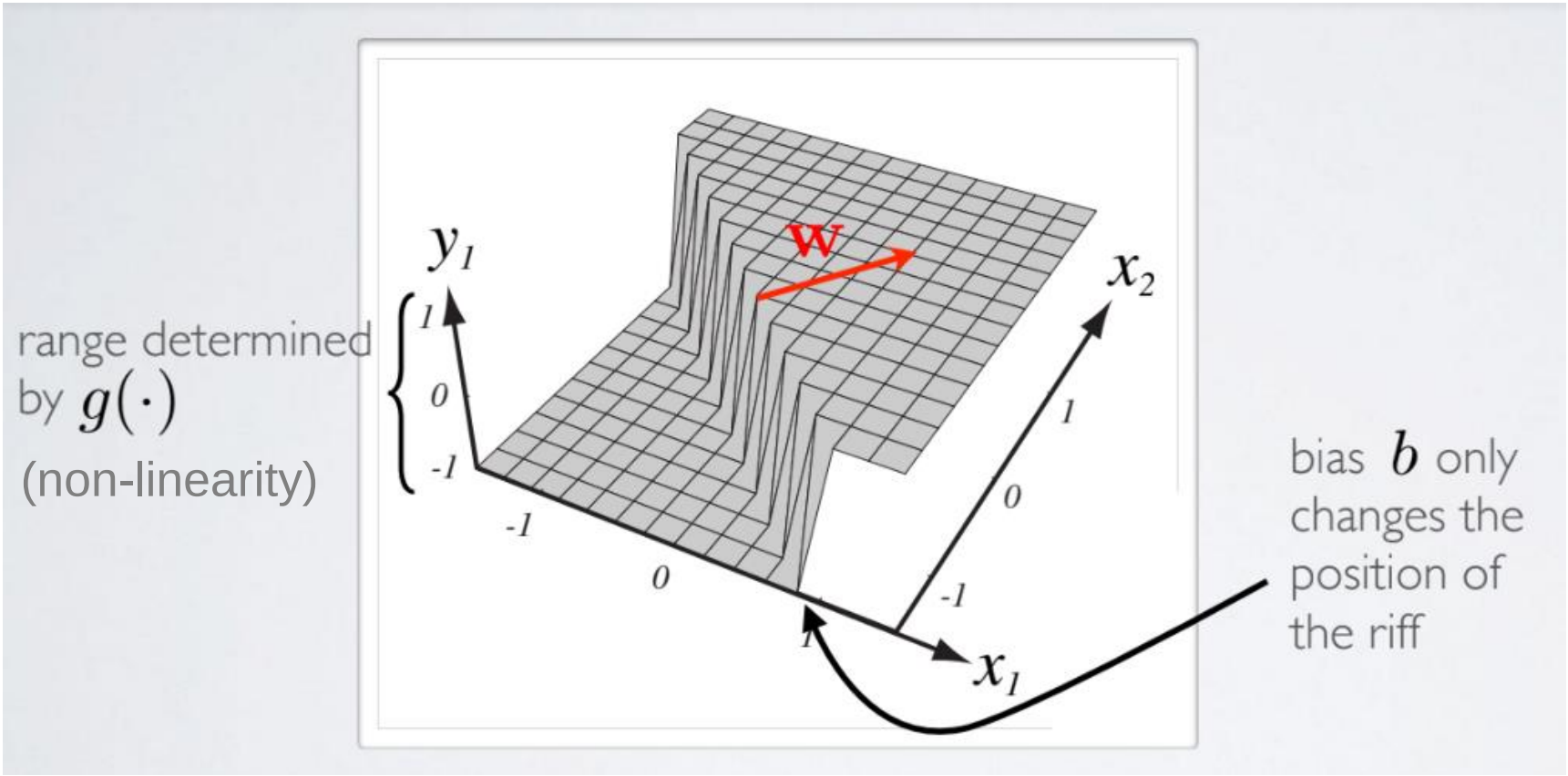


Image from: Hugo Larochelle's & Pascal Vincent's slides

Expressiveness of feed-forward NN

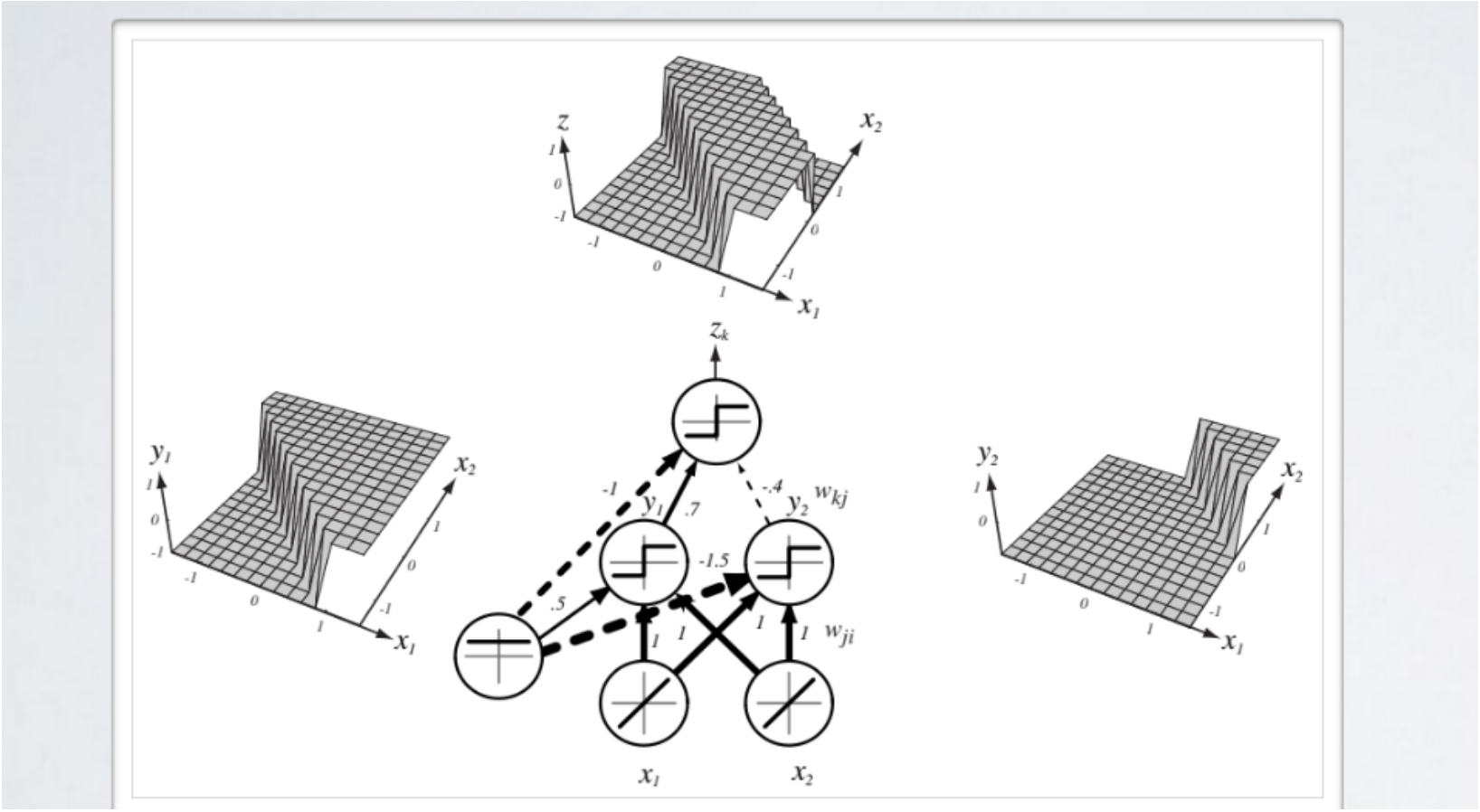


Image from: Hugo Larochelle's & Pascal Vincent's slides

Expressiveness of feed-forward NN

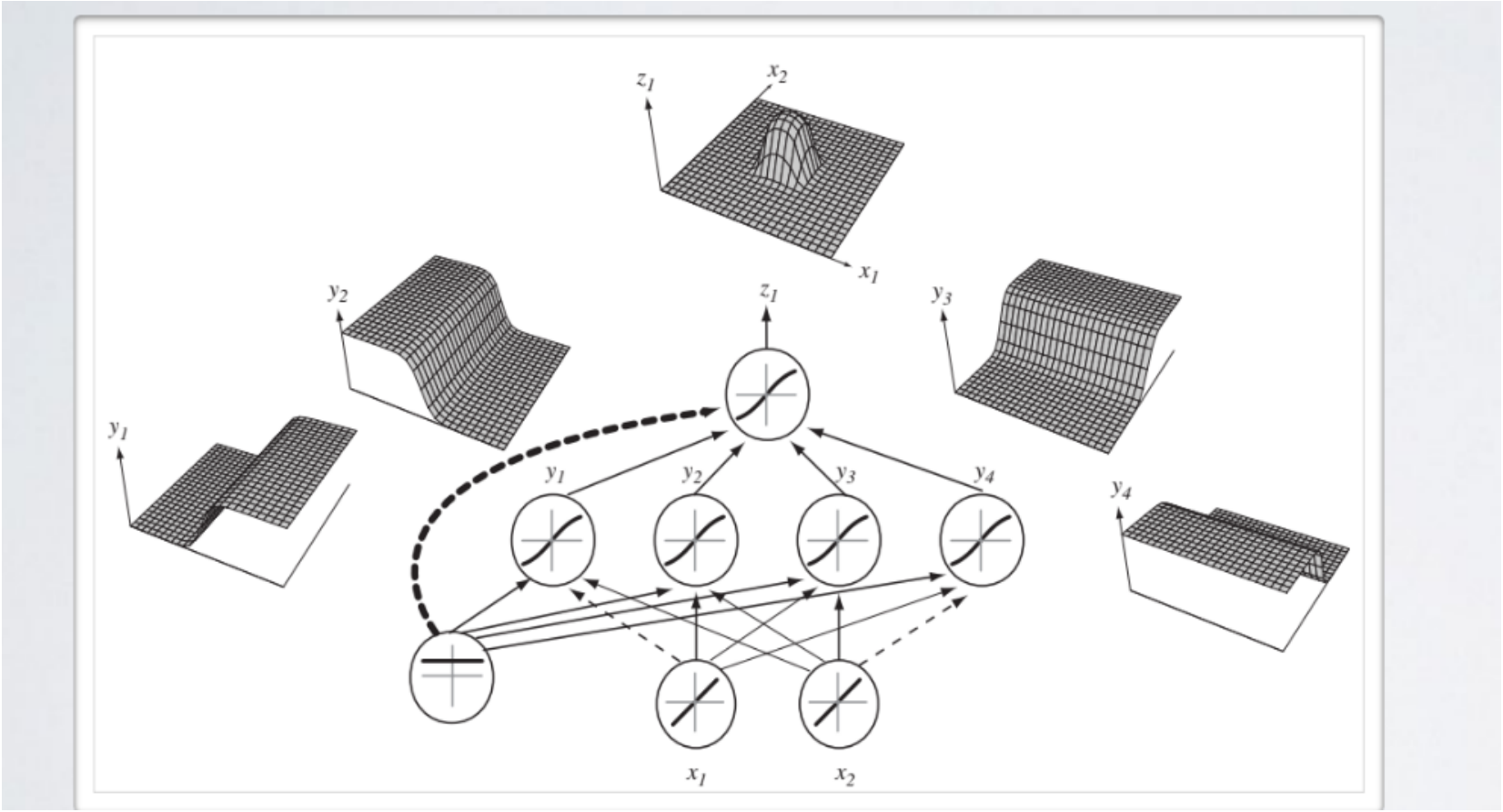


Image from: Hugo Larochelle's & Pascal Vincent's slides

Expressiveness of feed-forward NN

A single sigmoid neuron?

- Same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?

- Universal approximation theorem (Hornik, 1991):
  - *"Every bounded continuous function can be approximated* ***with arbitrary precision*** *by a single-layer neural network*"
- But might require a number of hidden units that is exponential in the number of inputs.
- Also, **this doesn't mean that we can easily learn the parameter values!**

Expressiveness of feed-forward NN

A single sigmoid neuron?

- Same representational power as a perceptron: Boolean AND, OR, NOT, but not XOR.

A neural network with a single hidden layer?

- Universal approximation theorem (Hornik, 1991)
- But might require a number of hidden units that is exponential in the number of inputs.
- Also, **this doesn't mean that we can easily learn the parameter values!**

A neural network with two hidden layers?

- Any function can be approximated to arbitrary accuracy by a network with two hidden layers.

Final notes

- What you should know:
  - Definition / components of neural networks.
  - Training by backpropagation
  - Stochastic gradient descent and its variants

- Additional information about neural networks:

  Video & slides from the Montreal Deep Learning Summer School:

  http://videolectures.net/deeplearning2017_larochelle_neural_networks/

  https://drive.google.com/file/d/0ByUKRdiCDK7-c2s2RjBiSms2UzA/view?usp=drive_web

  https://drive.google.com/file/d/0ByUKRdiCDK7-UXB1R1ZpX082MEk/view?usp=drive_web

  Manifold perspective on neural nets with cool visualizations:

  http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/