COMP 551 – Applied Machine Learning
Lecture 14: Neural Networks

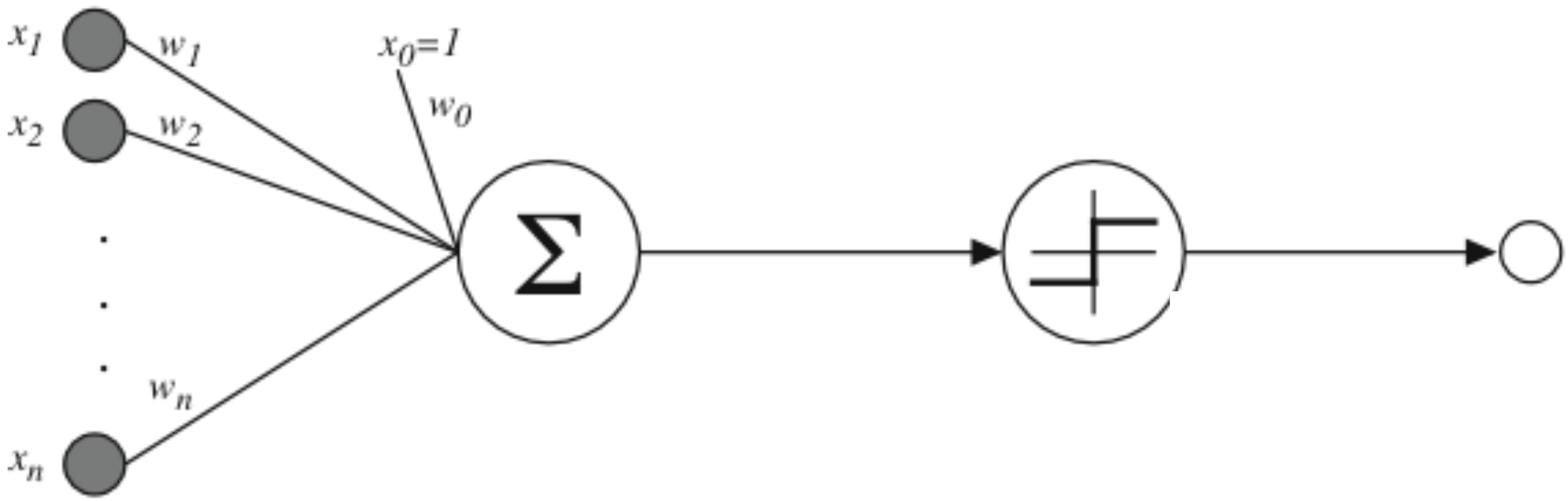
Instructor: Joelle Pineau (*jpineau@cs.mcgill.ca)*

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

Kaggle: Project 2

#	△pub	Team Name	Kernel	Team Members	Score	Entries	Last
1	—	💫💫 2045 💫💫			0.81705	36	2d
2	▲1	Juicebox3.0 (Stay hydrated)			0.81082	25	1d
3	▼1	ZSV			0.81067	23	1d
4	—	Nothing but Nets			0.80600	24	4d
5	▲2	Jale Li			0.80577	10	1d
6	▼1	DMT			0.80414	29	1d
7	▲1	NAS			0.80290	24	1d
8	▼2	AJGARS			0.80271	25	1d
9	—	WIL			0.80071	28	2d
10	▲2	Cereal Killer			0.79736	22	1d
11	▼1	team-02-unmerged			0.79712	2	1d
12	▼1	team-02			0.79712	15	1d
13	—	LazyLearner			0.79319	26	1d
14	▲2	LR			0.79234	24	2d
15	▲5	Epte			0.79176	18	3d

Recall the perceptron

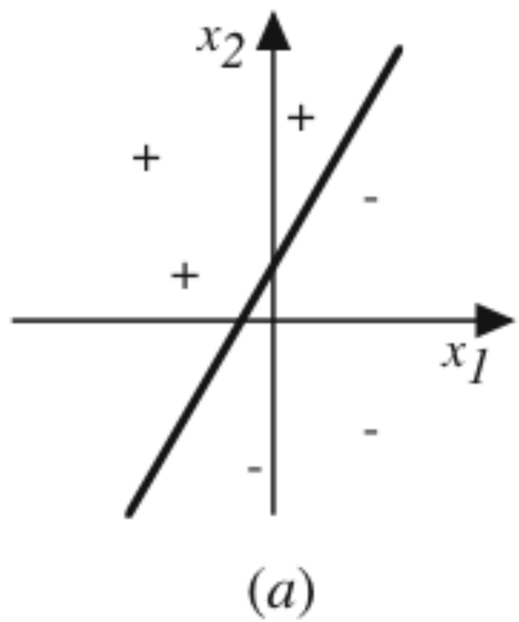


- We can take a linear combination and threshold it:

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

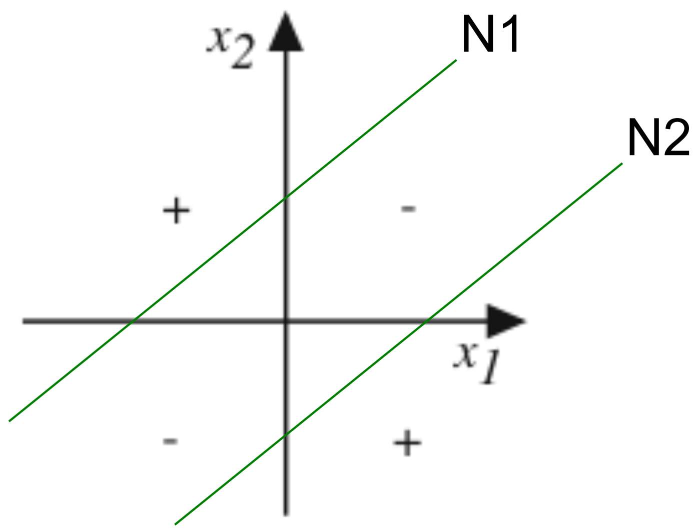
- The output is taken as the predicted class.

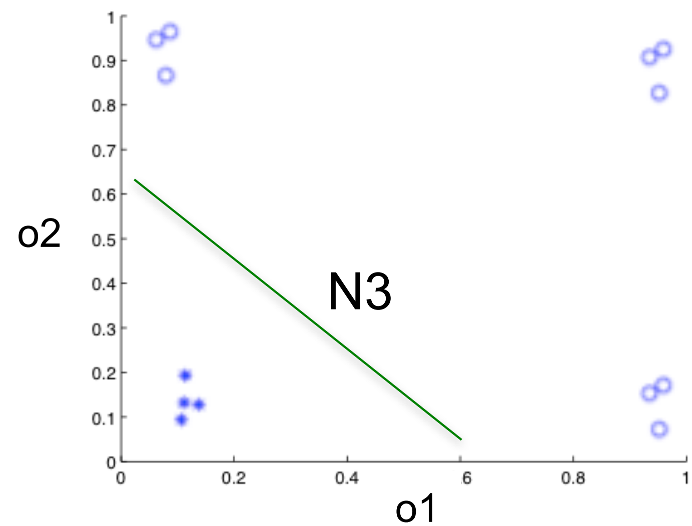
Decision surface of a perceptron



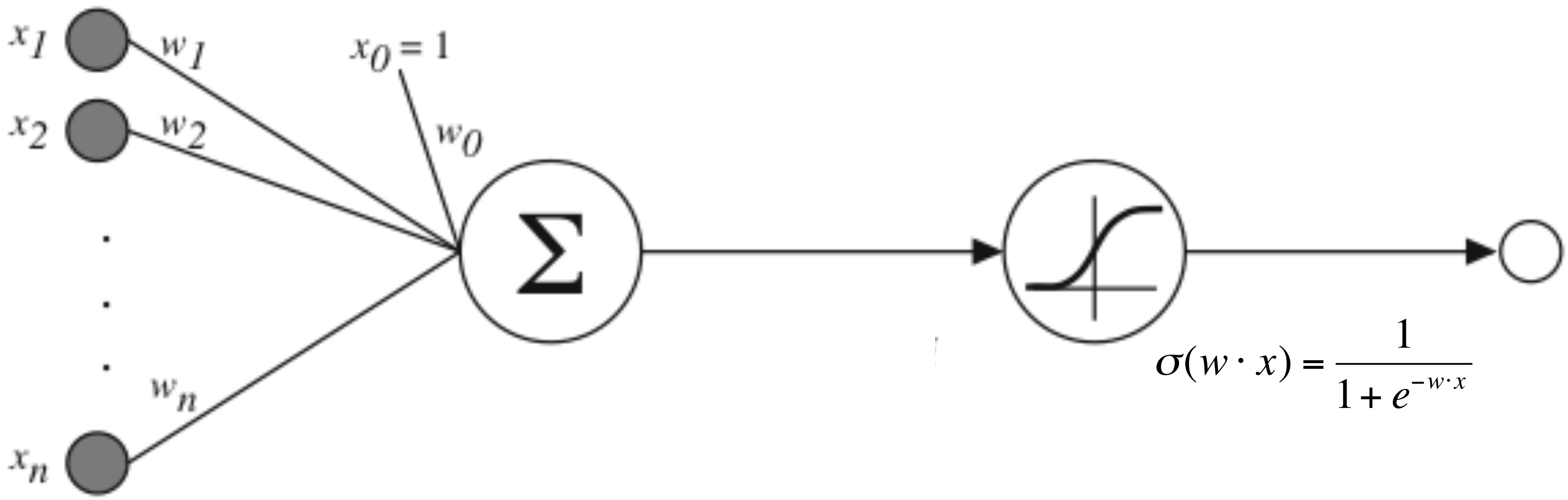
- Can represent many functions.
- To represent non-linearly separate functions (e.g. XOR), we could use a network of perceptron-like elements.
- If we connect perceptrons into networks, the error surface for the network is not differentiable (because of the hard threshold).

Example: A network representing XOR





Recall the sigmoid function



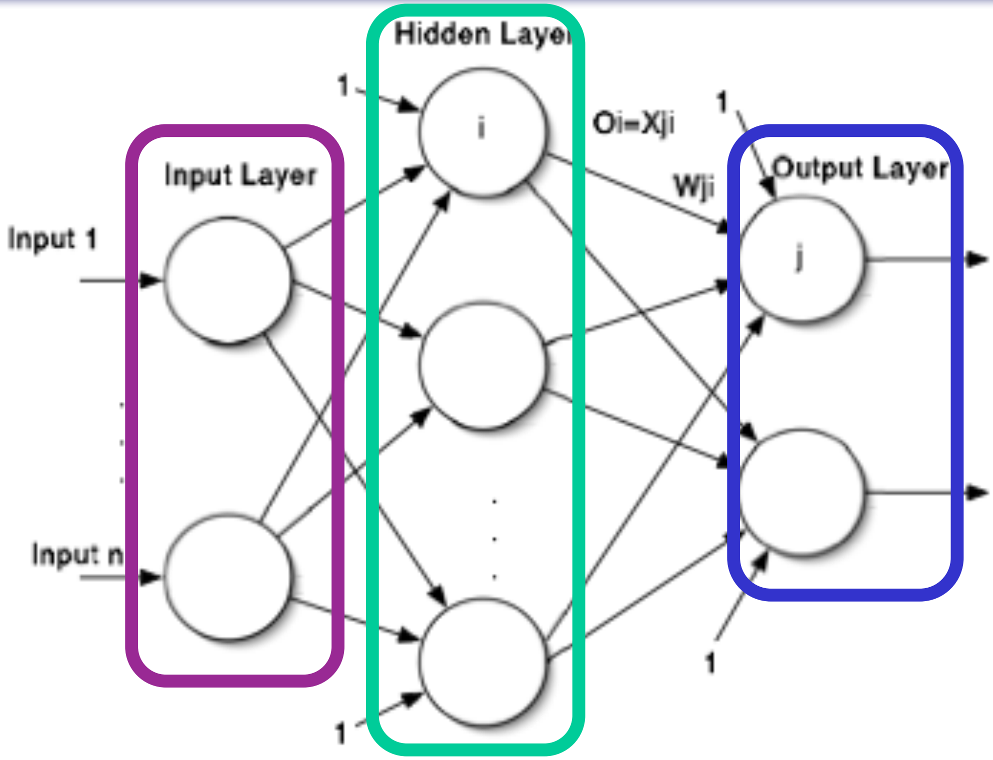
Sigmoid provide “soft threshold”, whereas perceptron provides “hard threshold”

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

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 sigmoid activation, 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 $k+1, k+2, \ldots, K$.
- 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:

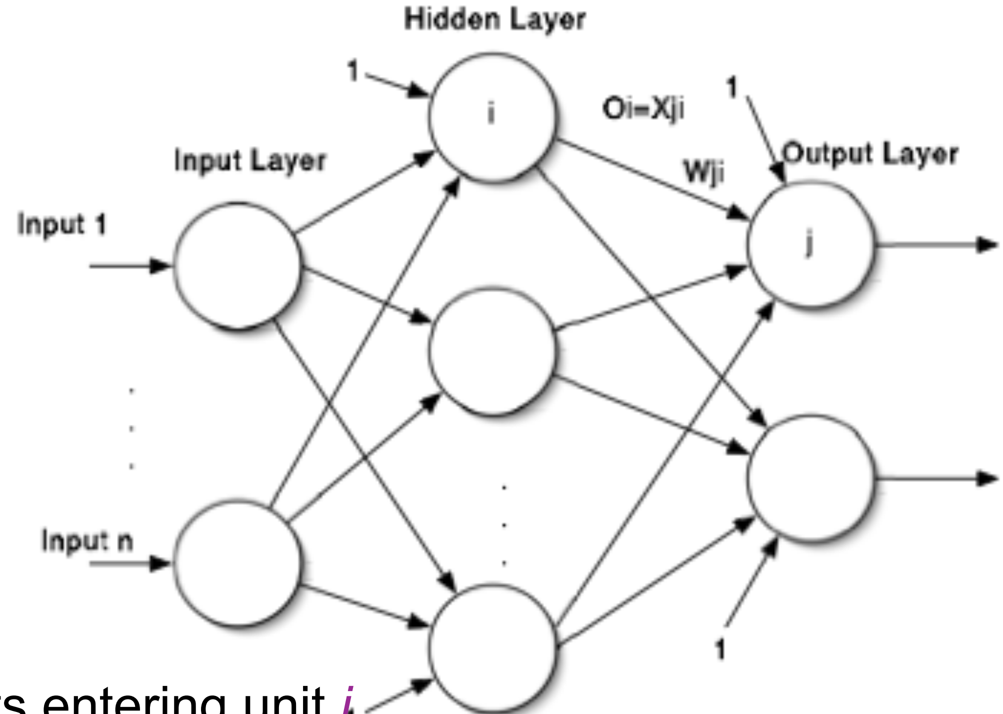
- w_{ji} denotes weight on connection from unit i to unit j.
- By convention, $x_{j0} = 1, \ \forall j$
- Output of unit j, denoted o_j is computed using a sigmoid:

  $$o_j = \sigma(\mathbf{w}_j \cdot \mathbf{x}_j)$$

  where $\mathbf{w}_j$ is vector of weights entering unit j

  $\mathbf{x}_j$ 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 *… 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 $\langle \mathbf{x}, y \rangle$.
- Let o_i, $i = 1, \ldots, H+N+1$ be the outputs of all units in the network for the given input $\mathbf{x}$.
- 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: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

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

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

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

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: $\partial J(w)/\partial w = (\partial J(w)/\partial\sigma) \cdot (\partial\sigma/\partial w)$

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

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

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

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

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: $\partial J(w)/\partial w = (\partial J(w)/\partial \sigma) \cdot (\partial \sigma/\partial w)$

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

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

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:

$$\begin{aligned}\frac{\partial J}{\partial w_{l,j}} &= -(y - o_{N+H+1})o_{N+H+1}(1 - o_{N+H+1}) \\ &\quad \cdot \frac{\partial}{\partial w_{l,j}}(\mathbf{w}_{N+H+1} \cdot \mathbf{x}_{N+H+1}) \\ &= -\delta_{N+H+1} w_{N+H+1,l} \frac{\partial}{\partial w_{l,j}} x_{N+H+1,l}\end{aligned}$$

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

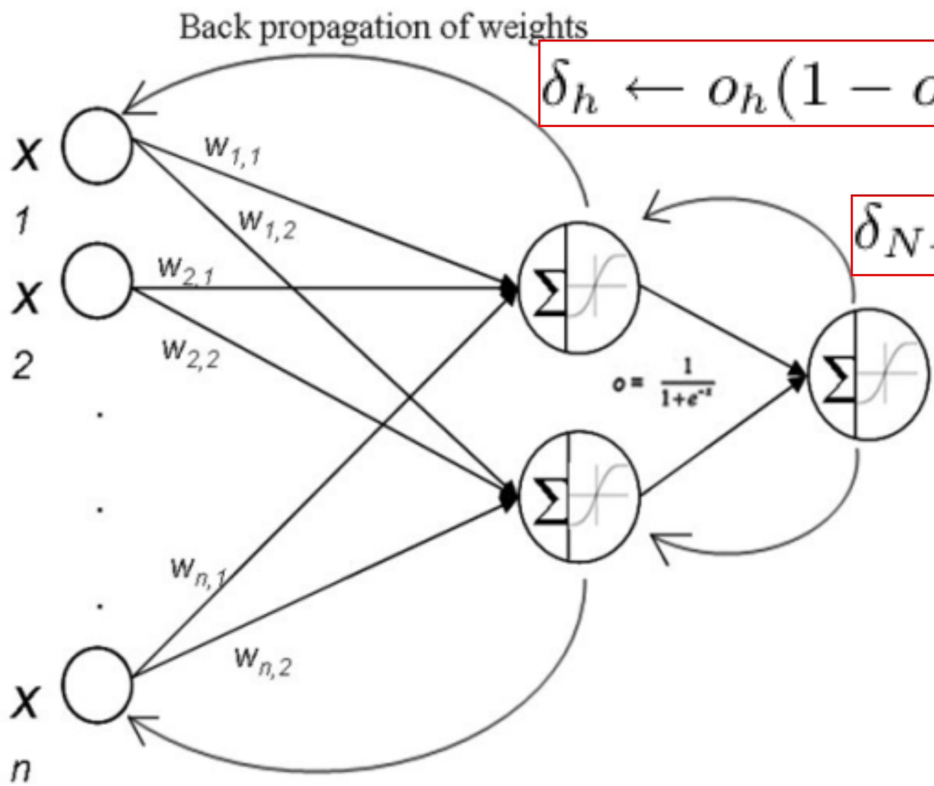
$$\frac{\partial}{\partial w_{l,j}} x_{N+H+1,l} = o_l(1 - o_l)x_{l,j}$$

- 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

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



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

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

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

Stochastic gradient descent

- 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

Organizing the training data

- **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 <u>single</u> sigmoid neuron?

Expressiveness of feed-forward NN

A <u>single</u> sigmoid neuron?

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

Expressiveness of feed-forward NN

A <u>single</u> sigmoid neuron?

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

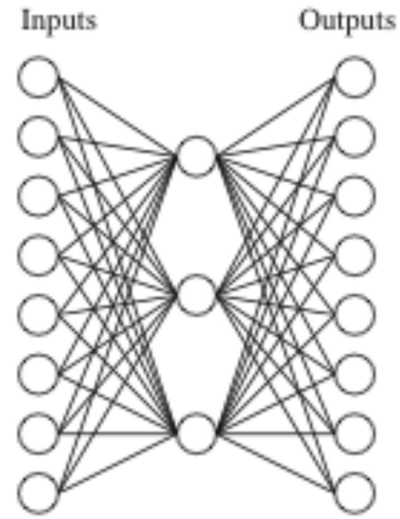
A neural network with a <u>single hidden layer</u>?

Learning the identity function

Input		Output
10000000	$\longrightarrow$	10000000
01000000	$\longrightarrow$	01000000
00100000	$\longrightarrow$	00100000
00010000	$\longrightarrow$	00010000
00001000	$\longrightarrow$	00001000
00000100	$\longrightarrow$	00000100
00000010	$\longrightarrow$	00000010
00000001	$\longrightarrow$	00000001

Learning the identity function

- Neural network structure:



- Learned hidden layer weights:

Input		Hidden Layer				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.15	.99	.99	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.01	.11	.88	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

Expressiveness of feed-forward NN

A <u>single</u> sigmoid neuron?

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

A neural network with a <u>single hidden layer</u>?

- Can represent every boolean function, but might require a number of hidden units that is exponential in the number of inputs.
- Every bounded continuous function can be approximated with arbitrary precision by a boolean function.

Expressiveness of feed-forward NN

A <u>single</u> sigmoid neuron?

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

A neural network with a <u>single hidden layer</u>?

- Can represent every boolean function, but might require a number of hidden units that is exponential in the number of inputs.
- Every bounded continuous function can be approximated with arbitrary precision by a boolean function.

A neural network with <u>two hidden layers</u>?

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?

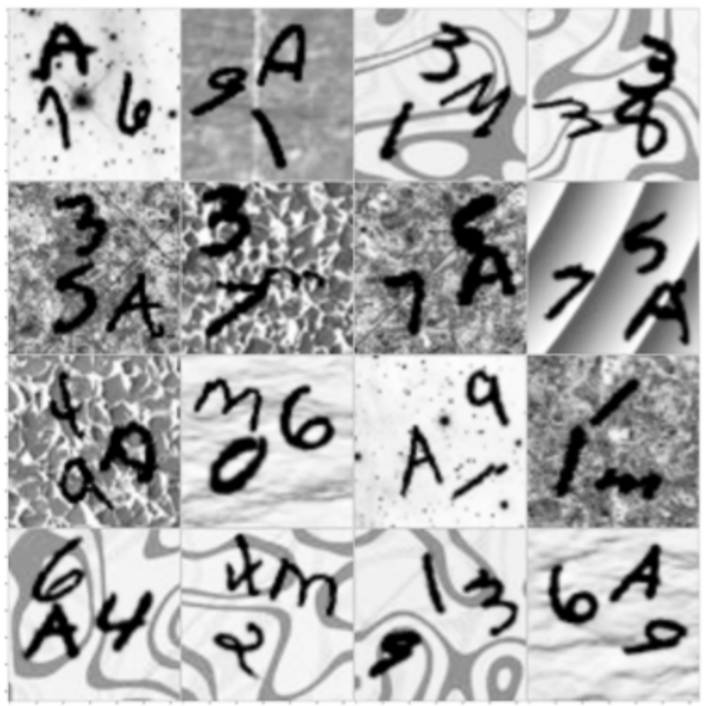
- Can represent every boolean function, but might require a number of hidden units that is exponential in the number of inputs.
- Every bounded continuous function can be approximated with arbitrary precision by a boolean function.

A neural network with two hidden layers?

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

Project 3: Visual add / multiply (due Nov.13)

- For each image, give the result of the equation.
  - If it's an "A", add the 2 digits. If it's an "M", multiply the 2 digits.



Final notes

- What you should know:
  - Definition / components of neural networks.
  - Training by backpropagation.

- 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

- Tutorial 3 is today! TR3120, 6-7pm.

- Project #2 peer reviews will open next Monday on CMT.