# Lecture 7: Neural Nets

Mark Hasegawa-Johnson

ECE 417: Multimedia Signal Processing, Fall 2020

## Outline

## What is a Neural Network?

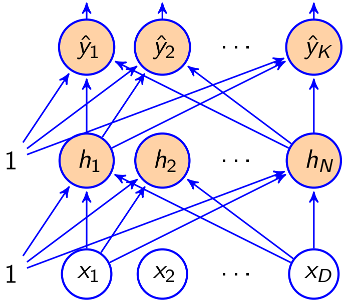
- Computation in biological neural networks is performed by trillions of simple cells (neurons), each of which performs one very simple computation.
- Biological neural networks learn by strengthening the connections between some pairs of neurons, and weakening other connections.

## What is an Artificial Neural Network?

- Computation in an artificial neural network is performed by thousands of simple cells (nodes), each of which performs one very simple computation.
- Artificial neural networks learn by strengthening the connections between some pairs of nodes, and weakening other connections.

## Two-Layer Feedforward Neural Network

$$\hat{y} = h(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = e_k^{(2)}$$

$$e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(e_k^{(1)})$$

$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

## Neural Network = Universal Approximator

Assume...

- Linear Output Nodes: $\hat{y}_k = e_k^{(2)}$
- Smoothly Nonlinear Hidden Nodes: $\frac{d\sigma}{de}$ finite
- Smooth Target Function: $\hat{y} = h(\vec{x}, W, b)$ approximates $\vec{y} = h^*(\vec{x}) \in \mathcal{H}$, where $\mathcal{H}$ is some class of sufficiently smooth functions of $\vec{x}$ (functions whose Fourier transform has a first moment less than some finite number $C$)
- There are $N$ hidden nodes, $\hat{y}_k$, $1 \le k \le N$
- The input vectors are distributed with some probability density function, $p(\vec{x})$, over which we can compute expected values.

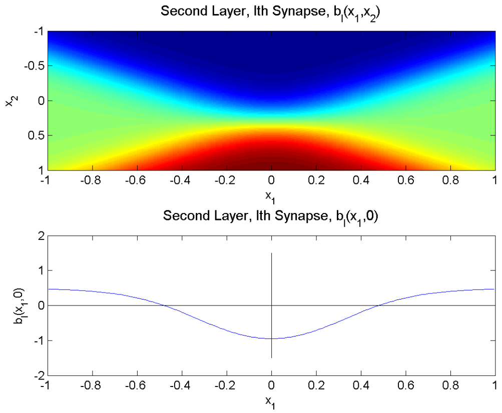
Then (Barron, 1993) showed that...

$$\max_{h^*(\vec{x}) \in \mathcal{H}} \min_{W,b} E\left[ \left| h(\vec{x}, W, b) - h^*(\vec{x}) \right|^2 \right] \le \mathcal{O}\left\{ \frac{1}{N} \right\}$$

## Outline

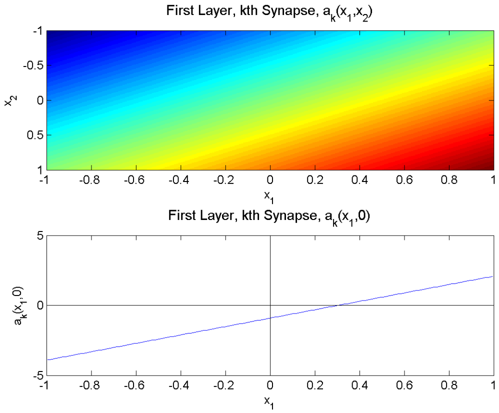
## Target: Can we get the neural net to compute this function?

Suppose our goal is to find some weights and biases, $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$ so that $\hat{y}(\vec{x})$ is the nonlinear function shown here:



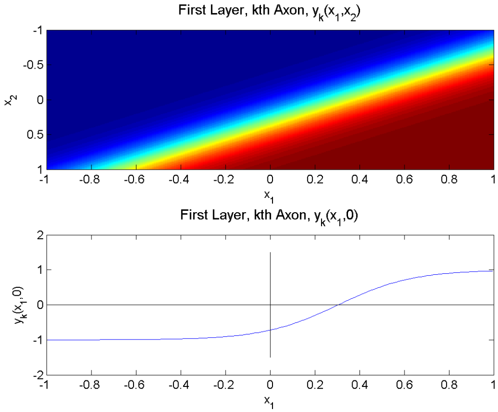
## Excitation, First Layer: $e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{2} w_{kj}^{(1)} x_j$

The first layer of the neural net just computes a linear function of $\vec{x}$. Here's an example:



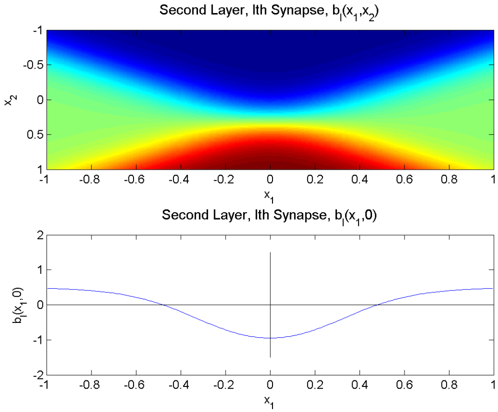
## Activation, First Layer: $h_k = \tanh(e_k^{(1)})$

The activation nonlinearity then "squashes" the linear function:



## Second Layer: $\hat{y}_k = b_k^{(2)} + \sum_{j=1}^{2} w_{kj}^{(2)} h_k$

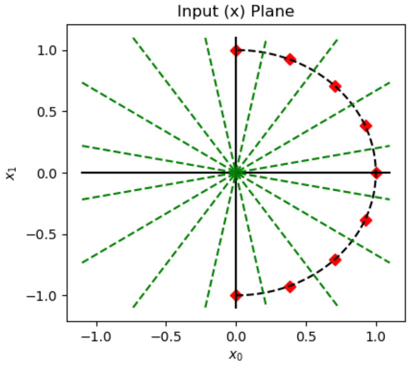
The second layer then computes a linear combination of the first-layer activations, which is sufficient to match our desired function:

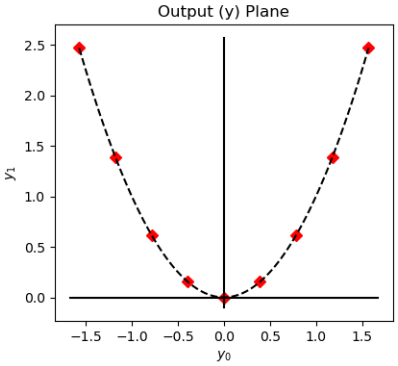


## Outline

## Example #2: Semicircle $\rightarrow$ Parabola

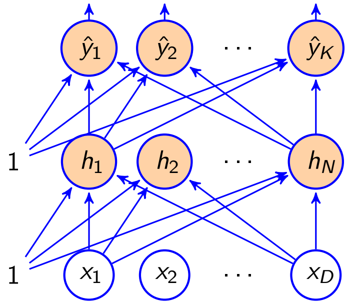
Can we design a neural net that converts a semicircle $(x_0^2 + x_1^2 = 1)$ to a parabola $(y_1 = y_0^2)$?





# Two-Layer Feedforward Neural Network

$$\hat{y} = h(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = e_k^{(2)}$$

$$e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(e_k^{(1)})$$

$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

## Example #2: Semicircle $\rightarrow$ Parabola

Let's define some vector notation:

- **Second Layer:** Define $\vec{w}_j^{(2)} = \begin{bmatrix} w_{0j}^{(2)} \\ w_{1j}^{(2)} \end{bmatrix}$, the $j^{\text{th}}$ column of the $W^{(2)}$ matrix, so that

$$\hat{y} = \vec{b} + \sum_j \vec{w}_j^{(2)} h_j \quad \text{means} \quad \hat{y}_k = b_k + \sum_j w_{kj}^{(2)} h_j \forall k.$$

- **First Layer Activation Function:**

$$h_k = \sigma\left(e_k^{(1)}\right)$$

- **First Layer Excitation:** Define $\bar{w}_k^{(1)} = [w_{k0}^{(1)}, w_{k1}^{(1)}]$, the $k^{\text{th}}$ row of the $W^{(1)}$ matrix, so that

$$e_k^{(1)} = \bar{w}_k^{(1)} \vec{x} \quad \text{means} \quad e_k^{(1)} = \sum_j w_{kj}^{(1)} x_j \forall k.$$

## Second Layer = Piece-Wise Approximation

The second layer of the network approximates $\hat{y}$ using a bias term $\vec{b}$, plus correction vectors $\vec{w}_j^{(2)}$, each scaled by its activation $h_j$:

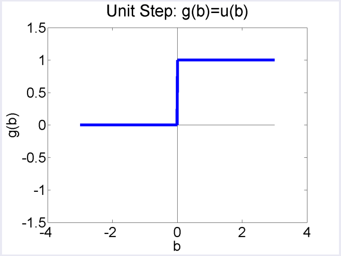
$$\hat{y} = \vec{b}^{(2)} + \sum_j \vec{w}_j^{(2)} h_j$$

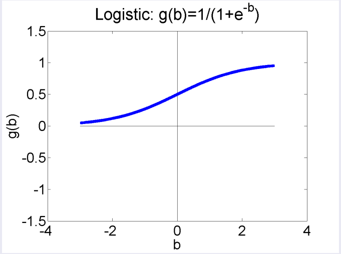
The activation, $h_j$, is a number between 0 and 1. For example, we could use the logistic sigmoid function:

$$h_k = \sigma\left(e_k^{(1)}\right) = \frac{1}{1 + \exp(-e_k^{(1)})} \in (0, 1)$$

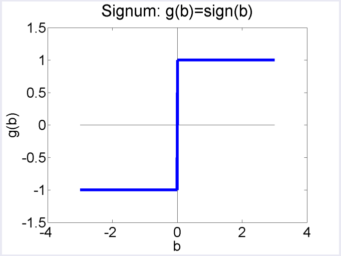
The logistic sigmoid is a differentiable approximation to a unit step function.

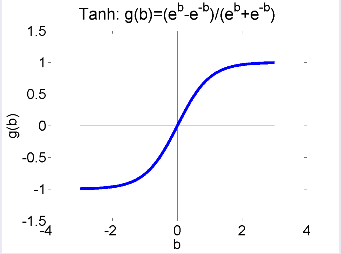
## Step and Logistic nonlinearities





## Signum and Tanh nonlinearities



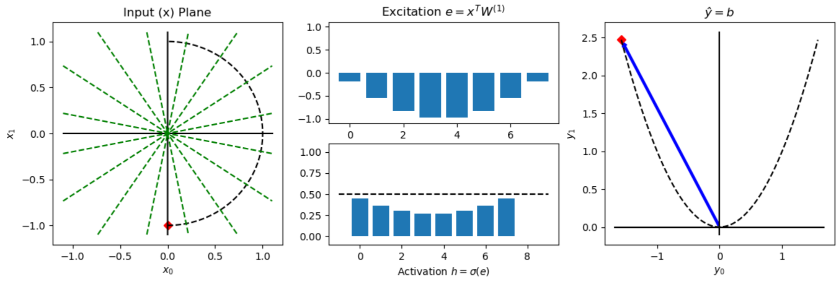


## First Layer = A Series of Decisions

The first layer of the network decides whether or not to “turn on” each of the $h_j$’s. It does this by comparing $\vec{x}$ to a series of linear threshold vectors:

$$h_k = \sigma\left(\bar{w}_k^{(1)}\vec{x}\right) \approx \begin{cases} 1 & \bar{w}_k^{(1)}\vec{x} > 0 \\ 0 & \bar{w}_k^{(1)}\vec{x} < 0 \end{cases}$$

## Example #2: Semicircle $\rightarrow$ Parabola



## Outline

## How to train a neural network

1. Find a **training dataset** that contains $n$ examples showing the desired output, $\vec{y}_i$, that the NN should compute in response to input vector $\vec{x}_i$:

$$\mathcal{D} = \{(\vec{x}_1, \vec{y}_1), \ldots, (\vec{x}_n, \vec{y}_n)\}$$

2. Randomly **initialize** the weights and biases, $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
3. Perform **forward propagation**: find out what the neural net computes as $\hat{y}_i$ for each $\vec{x}_i$.
4. Define a **loss function** that measures how badly $\hat{y}$ differs from $\vec{y}$.
5. Perform **back propagation** to improve $W^{(1)}$, $\vec{b}^{(1)}$, $W^{(2)}$, and $\vec{b}^{(2)}$.
6. Repeat steps 3-5 until convergence.

## Loss Function: How should $h(\vec{x})$ be "similar to" $h^*(\vec{x})$?

### Minimum Mean Squared Error (MMSE)

$$W^*, b^* = \arg\min \mathcal{L} = \arg\min \frac{1}{2n}\sum_{i=1}^{n} \|\vec{y}_i - \hat{y}(\vec{x}_i)\|^2$$

### MMSE Solution: $\hat{y} \rightarrow E\left[\vec{y}|\vec{x}\right]$

If the training samples $(\vec{x}_i, \vec{y}_i)$ are i.i.d., then

$$\lim_{n\rightarrow\infty} \mathcal{L} = \frac{1}{2}E\left[\|\vec{y} - \hat{y}\|^2\right]$$

which is minimized by

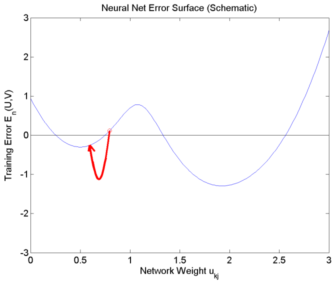
$$\hat{y}_{MMSE}(\vec{x}) = E\left[\vec{y}|\vec{x}\right]$$

## Gradient Descent: How do we improve $W$ and $b$?

Given some initial neural net parameter (called $u_{kj}$ in this figure), we want to find a better value of the same parameter. We do that using gradient descent:

$$u_{kj} \leftarrow u_{kj} - \eta \frac{d\mathcal{L}}{du_{kj}},$$

where $\eta$ is a learning rate (some small constant, e.g., $\eta = 0.02$ or so).



## Gradient Descent = Local Optimization

Given an initial $W$, $b$, find new values of $W$, $b$ with lower error.

$$w_{kj}^{(1)} \leftarrow w_{kj}^{(1)} - \eta \frac{d\mathcal{L}}{dw_{kj}^{(1)}}$$

$$w_{kj}^{(2)} \leftarrow w_{kj}^{(2)} - \eta \frac{d\mathcal{L}}{dw_{kj}^{(2)}}$$

## $\eta$ =Learning Rate

- If $\eta$ too large, gradient descent won't converge. If too small, convergence is slow.
- Second-order methods like L-BFGS and Adam choose an optimal $\eta$ at each step, so they're MUCH faster.

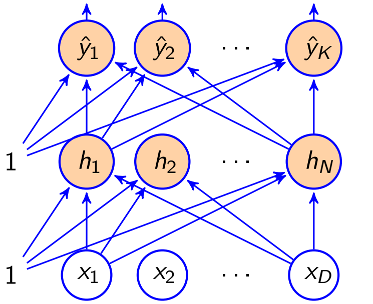
## Computing the Gradient: Notation

- $\vec{x}_i = [x_{1i}, \ldots, x_{Di}]^T$ is the $i^{\text{th}}$ input vector.
- $\vec{y}_i = [y_{1i}, \ldots, y_{Ki}]^T$ is the $i^{\text{th}}$ target vector (desired output).
- $\hat{y}_i = [\hat{y}_{1i}, \ldots, \hat{y}_{Ki}]^T$ is the $i^{\text{th}}$ hypothesis vector (computed output).
- $\vec{e}_i^{(l)} = [e_{1i}^{(l)}, \ldots, e_{Ni}^{(l)}]^T$ is the excitation vector after the $l^{\text{th}}$ layer, in response to the $i^{\text{th}}$ input.
- $\vec{h}_i = [h_{1i}, \ldots, h_{Ni}]^T$ is the hidden nodes activation vector in response to the $i^{\text{th}}$ input. (No superscript necessary if there's only one hidden layer).
- The weight matrix for the $l^{\text{th}}$ layer is

$$W^{(l)} = \left[\vec{w}_1^{(l)}, \ldots, \vec{w}_j^{(l)}, \ldots\right] = \begin{bmatrix} w_{11}^{(l)} & \cdots & w_{1j}^{(l)} & \cdots \\ \vdots & \ddots & \vdots & \ddots \\ w_{k1}^{(l)} & \cdots & w_{kj}^{(l)} & \cdots \\ \vdots & \ddots & \vdots & \ddots \end{bmatrix}$$

## Two-Layer Feedforward Neural Network

$$\hat{y} = h(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = e_k^{(2)}$$

$$e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(e_k^{(1)})$$

$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

## Computing the Derivative

OK, let's compute the derivative of $\mathcal{L}$ with respect to the $W^{(2)}$ matrix. Remember that $W^{(2)}$ enters the neural net computation as $e_{ki}^{(2)} = \sum_k w_{kj}^{(2)} h_{ji}$. So...

$$\frac{d\mathcal{L}}{dw_{kj}^{(2)}} = \sum_{i=1}^{n} \left(\frac{d\mathcal{L}}{de_{ki}^{(2)}}\right)\left(\frac{\partial e_{ki}^{(2)}}{\partial w_{kj}^{(2)}}\right)$$

$$= \sum_{i=1}^{n} \epsilon_{ki} h_{ki}$$

where the last line only works if we define $\epsilon_{ki}$ in a useful way:

$$\vec{\epsilon}_i = [\epsilon_{1i}, \ldots, \epsilon_{Ki}]^T$$

$$= \nabla_{\vec{e}_i^{(2)}} \mathcal{L} \quad \left(\text{meaning that } \epsilon_{ki} = \frac{\partial \mathcal{L}}{\partial e_{ki}^{(2)}}\right)$$

$$= \frac{1}{n}(\hat{y}_i - \vec{y}_i)$$

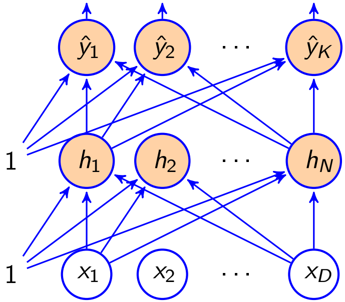
## Digression: Total Derivative vs. Partial Derivative

- The notation $\frac{d\mathcal{L}}{dw_{kj}^{(2)}}$ means "the total derivative of $\mathcal{L}$ with respect to $w_{kj}^{(2)}$." It implies that we have to add up several different ways in which $\mathcal{L}$ depends on $w_{kj}^{(2)}$, for example,

$$\frac{d\mathcal{L}}{dw_{kj}^{(2)}} = \sum_{i=1}^{n} \left(\frac{d\mathcal{L}}{d\hat{y}_{ki}}\right)\left(\frac{\partial \hat{y}_{ki}}{\partial w_{kj}^{(2)}}\right)$$

- The notation $\frac{\partial\mathcal{L}}{\partial\hat{y}_{ki}}$ means "partial derivative." It means "hold other variables constant while calculating this derivative."
- For some variables, the total derivative and partial derivative are the same—it doesn't matter whether we hold other variables constant or not. In fact, $\hat{y}_{ki}$ is one of those, so we could write $\frac{d\mathcal{L}}{d\hat{y}_{ki}} = \frac{\partial\mathcal{L}}{\partial\hat{y}_{ki}}$ for this particular variable.
- On the other hand, the difference starts to matter when we try to compute $\frac{d\mathcal{L}}{dw_{kj}^{(1)}}$.

$$\hat{y} = h(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = e_k^{(2)}$$

$$e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(e_k^{(1)})$$

$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

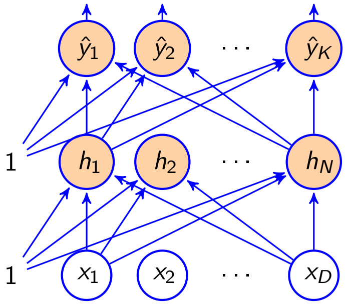
$\vec{x}$ is the input vector

## Back-Propagating to the First Layer

$$\frac{d\mathcal{L}}{dw_{kj}^{(1)}} = \sum_{i=1}^{n} \left(\frac{d\mathcal{L}}{de_{ki}^{(1)}}\right)\left(\frac{\partial e_{ki}^{(1)}}{\partial w_{kj}^{(1)}}\right) = \sum_{i=1}^{n} \delta_{ki} x_{ji}$$

where: $\delta_{ki} = \dfrac{d\mathcal{L}}{de_{ki}^{(1)}}$

$$\hat{y} = h(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = e_k^{(2)}$$

$$e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(e_k^{(1)})$$

$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

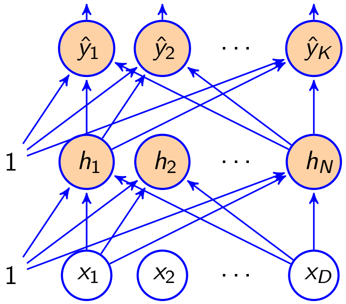
$\vec{x}$ is the input vector

## Back-Propagating to the First Layer

$$\delta_{ki} = \frac{d\mathcal{L}}{de_{ki}^{(1)}}$$

$$= \sum_{\ell=1}^{K} \left(\frac{d\mathcal{L}}{de_{\ell i}^{(2)}}\right) \left(\frac{\partial e_{\ell i}^{(2)}}{\partial h_{ki}}\right) \left(\frac{\partial h_{ki}}{\partial e_{ki}^{(1)}}\right)$$

$$\hat{y} = h(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



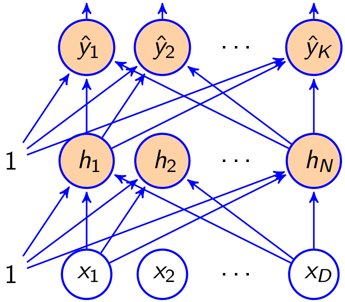
$$\hat{y}_k = e_k^{(2)}$$
$$e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$
$$h_k = \sigma(e_k^{(1)})$$
$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

## Back-Propagating to the First Layer

$$\delta_{ki} = \sum_{\ell=1}^{K} \left(\frac{d\mathcal{L}}{de_{\ell i}^{(2)}}\right) \left(\frac{\partial e_{\ell i}^{(2)}}{\partial h_{ki}}\right) \left(\frac{\partial h_{ki}}{\partial e_{ki}^{(1)}}\right)$$
$$= \sum_{\ell=1}^{K} \epsilon_{\ell i} w_{\ell k}^{(2)} \sigma'(e_{ki}^{(1)})$$

$$\hat{y} = h(\vec{x}, W^{(1)}, \vec{b}^{(1)}, W^{(2)}, \vec{b}^{(2)})$$



$$\hat{y}_k = e_k^{(2)}$$

$$e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{N} w_{kj}^{(2)} h_j$$

$$h_k = \sigma(e_k^{(1)})$$

$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

## Back-Propagating to the First Layer

$$\frac{d\mathcal{L}}{dw_{kj}^{(1)}} = \sum_{i=1}^{n} \left(\frac{d\mathcal{L}}{de_{ki}^{(1)}}\right)\left(\frac{\partial e_{ki}^{(1)}}{\partial w_{kj}^{(1)}}\right) = \sum_{i=1}^{n} \delta_{ki} x_{ji}$$

$$\delta_{ki} = \frac{d\mathcal{L}}{de_{ki}^{(1)}} = \sum_{\ell=1}^{K} \epsilon_{\ell i} w_{\ell k}^{(2)} \sigma'(e_{ki}^{(1)})$$

## The Back-Propagation Algorithm

$$W^{(2)} \leftarrow W^{(2)} - \eta \nabla_{W^{(2)}} \mathcal{L}, \qquad W^{(1)} \leftarrow W^{(1)} - \eta \nabla_{W^{(1)}} \mathcal{L}$$

$$\nabla_{W^{(2)}} \mathcal{L} = \sum_{i=1}^{n} \vec{\epsilon}_i \vec{h}_i^T, \qquad \nabla_{W^{(1)}} \mathcal{L} = \sum_{i=1}^{n} \vec{\delta}_i \vec{x}_i^T$$

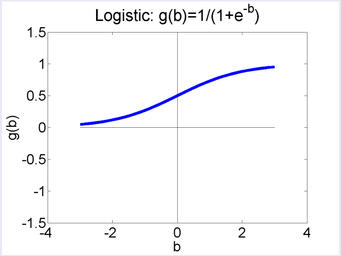
$$\epsilon_{ki} = \frac{1}{n}(\hat{y}_{ki} - y_{ki}), \qquad \delta_{ki} = \sum_{\ell=1}^{K} \epsilon_{\ell i} w_{\ell k}^{(2)} \sigma'(e_{ki}^{(1)})$$

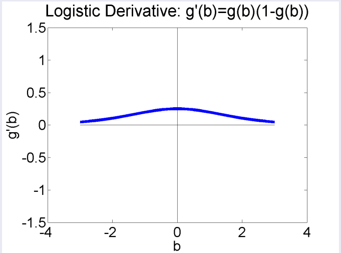
$$\vec{\epsilon}_i = \frac{1}{n}\left(\hat{\vec{y}}_i - \vec{y}_i\right), \qquad \vec{\delta}_i = \sigma'(\vec{e}_i^{(1)}) \odot W^{(2),T} \vec{\epsilon}_i$$

. . . where $\odot$ means element-wise multiplication of two vectors; $\sigma'(\vec{e})$ is the element-wise derivative of $\sigma(\vec{e})$.

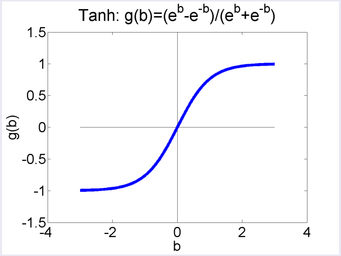
## Derivatives of the Nonlinearities

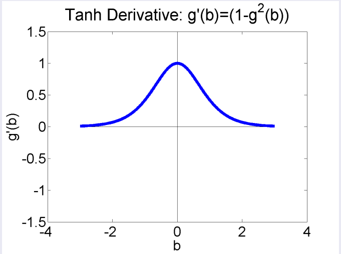
### Logistic





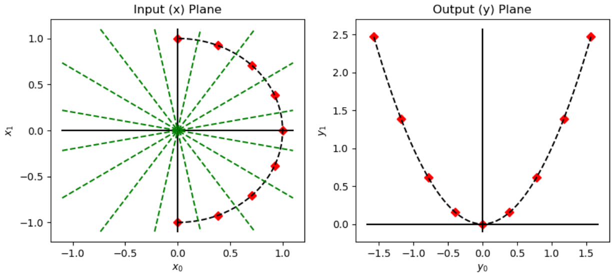
### Tanh





## Outline

## Backprop Example: Semicircle $\rightarrow$ Parabola

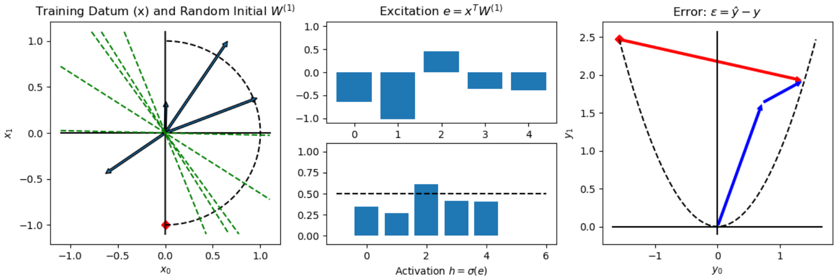


Remember, we are going to try to approximate this using:

$$\hat{y} = \vec{b} + \sum_j \vec{w}_j^{(2)} \sigma \left( \bar{w}_k^{(1)} \vec{x} \right)$$

## Randomly Initialized Weights

Here's what we get if we randomly initialize $\bar{w}_k^{(1)}$, $\vec{b}$, and $\vec{w}_j^{(2)}$. The red vector on the right is the estimation error for this training token, $\vec{\epsilon} = \hat{y} - \vec{y}$. It's huge!



## Back-Prop: Layer 2

Remember

$$W^{(2)} \leftarrow W^{(2)} - \eta \nabla_{W^{(2)}} \mathcal{L} = W^{(2)} - \eta \sum_{i=1}^{n} \vec{\epsilon}_i \vec{h}_i^T$$

$$= W^{(2)} - \frac{\eta}{n} \sum_{i=1}^{n} \left( \hat{y}_i - \vec{y}_i \right) \vec{h}_i^T$$

Thinking in terms of the columns of $W^{(2)}$, we have

$$\vec{w}_j^{(2)} \leftarrow \vec{w}_j^{(2)} - \frac{\eta}{n} \sum_{i=1}^{n} \left( \hat{y}_i - \vec{y}_i \right) h_{ji}$$

So, in words, layer-2 backprop means

- Each column, $\vec{w}_j^{(2)}$, gets updated in the direction $\vec{y} - \hat{y}$.
- The update for the $j^{\text{th}}$ column, in response to the $i^{\text{th}}$ training token, is scaled by its activation $h_{ji}$.

## Back-Prop: Layer 1

Remember

$$W^{(1)} \leftarrow W^{(1)} - \eta \nabla_{W^{(1)}} \mathcal{L} = W^{(1)} - \eta \sum_{i=1}^{n} \vec{\delta}_i \vec{x}_i^T$$

$$= W^{(1)} - \eta \sum_{i=1}^{n} \left( \sigma'(\vec{e}_i^{(1)}) \odot W^{(2),T} \vec{\epsilon}_i \right) \vec{x}_i^T$$

Thinking in terms of the rows of $W^{(1)}$, we have

$$\bar{w}_k^{(1)} \leftarrow \bar{w}_k^{(1)} - \eta \sum_{i=1}^{n} \delta_{ki} \vec{x}_i^T$$

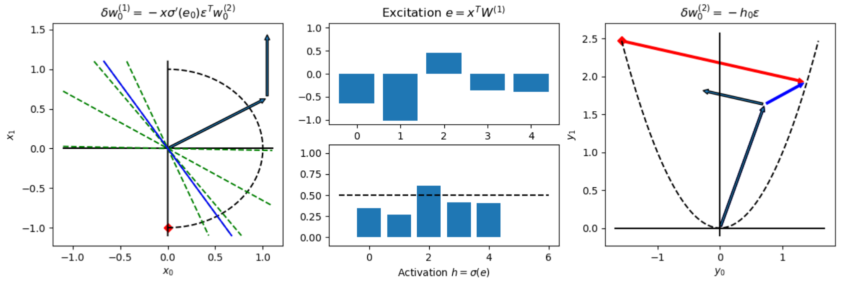
In words, layer-1 backprop means

- Each row, $\bar{w}_k^{(1)}$, gets updated in the direction $-\vec{x}$.
- The update for the $k^{\text{th}}$ row, in response to the $i^{\text{th}}$ training token, is scaled by its back-propagated error term $\delta_{ki}$.

## Back-Prop Example: Semicircle $\rightarrow$ Parabola

For each column $\vec{w}_j^{(2)}$ and the corresponding row $\bar{w}_k^{(1)}$,

$$\vec{w}_j^{(2)} \leftarrow \vec{w}_j^{(2)} - \frac{\eta}{n}\sum_{i=1}^{n}\left(\hat{y}_i - \vec{y}_i\right) h_{ji}, \qquad \bar{w}_k^{(1)} \leftarrow \bar{w}_k^{(1)} - \eta\sum_{i=1}^{n}\delta_{ki}\vec{x}_i^T$$



## Outline

## Summary

- A neural network approximates an arbitrary function using a sort of piece-wise approximation.
- The activation of each piece is determined by a nonlinear activation function applied to the hidden layer.
- Training is done using gradient descent.
- "Back-propagation" is the process of using the chain rule of differentiation in order to find the derivative of the loss with respect to each of the learnable weights and biases of the network.