Lecture 17: Neural Nets

Mark Hasegawa-Johnson

ECE 417: Multimedia Signal Processing, Fall 2021

Outline

What is a Neural Network?

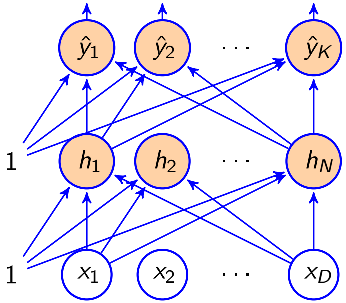
- Computation in biological neural networks is performed by trillions of simple cells (neurons), each of which performs one very simple computation.
- Biological neural networks learn by strengthening the connections between some pairs of neurons, and weakening other connections.

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.

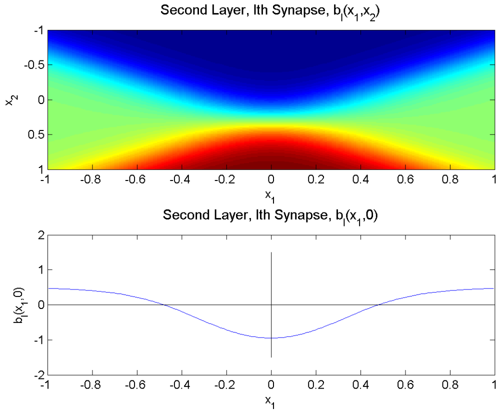
Then (Barron, 1993) showed that...

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

Outline

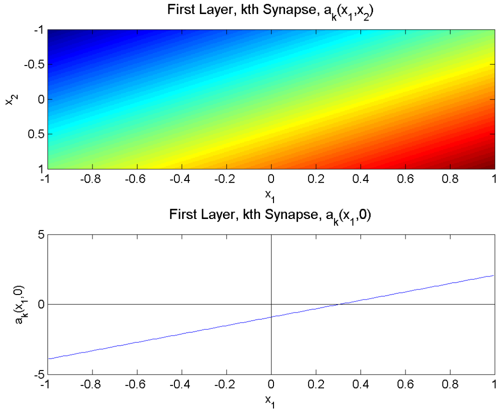
Target: Can we get the neural net to compute this function?

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:



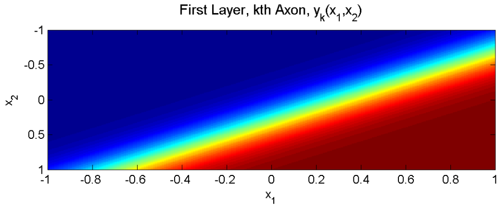
Excitation, First Layer: $e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{2} w_{kj}^{(1)} x_j$

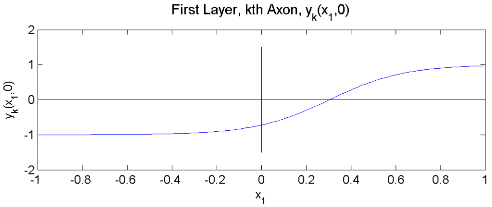
The first layer of the neural net just computes a linear function of $\vec{x}$. Here's an example:



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

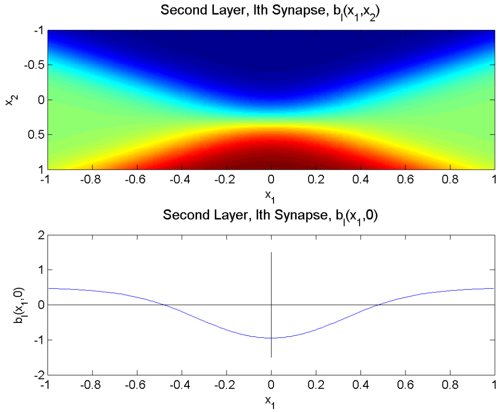
The activation nonlinearity then “squashes” the linear function:





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

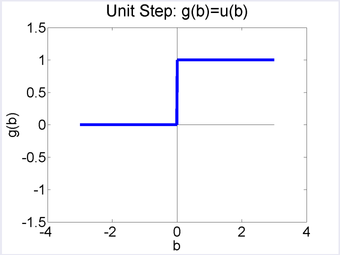
The second layer then computes a linear combination of the first-layer activations, which is sufficient to match our desired function:



Outline

The Basic Binary Nonlinearity: Unit Step (a.k.a. Heaviside function)

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



Pros and Cons of the Unit Step

- **Pro:** it gives exactly piece-wise constant approximation of any desired $\vec{y}$.
- **Con:** if $h_k = u(e_k)$, then you can't use back-propagation to train the neural network.

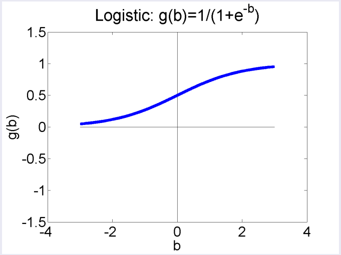
Remember back-prop:

$$\frac{d\mathcal{L}}{dw_{kj}} = \sum_k \left(\frac{d\mathcal{L}}{dh_k}\right)\left(\frac{\partial h_k}{\partial e_k}\right)\left(\frac{\partial e_k}{\partial w_{kj}}\right)$$

but $du(x)/dx$ is a Dirac delta function — zero everywhere, except where it's infinite.

The Differentiable Approximation: Logistic Sigmoid

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



Why to use the logistic function

$$\sigma(b) = \begin{cases} 1 & b \rightarrow \infty \\ 0 & b \rightarrow -\infty \\ \text{in between} & \text{in between} \end{cases}$$

and $\sigma(b)$ is smoothly differentiable, so back-prop works.

Derivative of a sigmoid

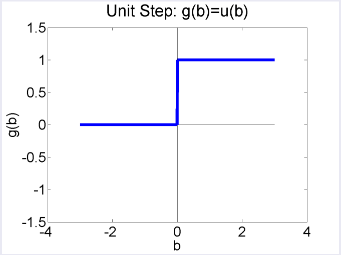
The derivative of a sigmoid is pretty easy to calculate:

$$\sigma(x) = \frac{1}{1+e^{-x}}, \quad \frac{d\sigma}{dx} = \frac{e^{-x}}{(1+e^{-x})^2}$$

An interesting fact that's extremely useful, in computing back-prop, is that if $h = \sigma(x)$, then we can write the derivative in terms of h, without any need to store x:

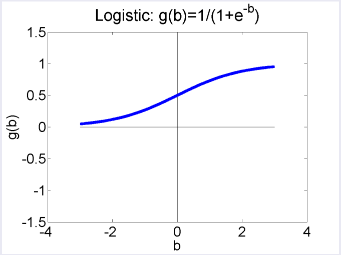
$$\begin{aligned}
\frac{d\sigma}{dx} &= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{e^{-x}}{1+e^{-x}}\right) \\
&= \left(\frac{1}{1+e^{-x}}\right)\left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x)(1-\sigma(x)) \\
&= h(1-h)
\end{aligned}$$

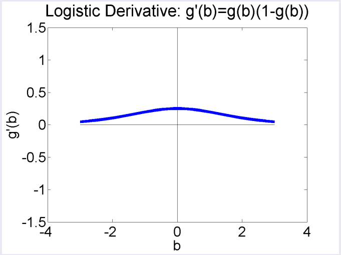
Step function and its derivative



- The derivative of the step function is the Dirac delta, which is not very useful in backprop.

Logistic function and its derivative





Signum and Tanh

The signum function is a signed binary nonlinearity. It is used if, for some reason, you want your output to be $h \in \{-1, 1\}$, instead of $h \in \{0, 1\}$:

$$\text{sign}(b) = \begin{cases} -1 & b < 0 \\ 1 & b > 0 \end{cases}$$

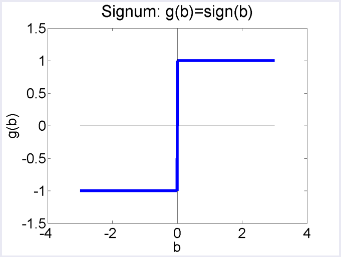
It is usually approximated by the hyperbolic tangent function (tanh), which is just a scaled shifted version of the sigmoid:

$$\tanh(b) = \frac{e^b - e^{-b}}{e^b + e^{-b}} = \frac{1 - e^{-2b}}{1 + e^{-2b}} = 2\sigma(2b) - 1$$

and which has a scaled version of the sigmoid derivative:

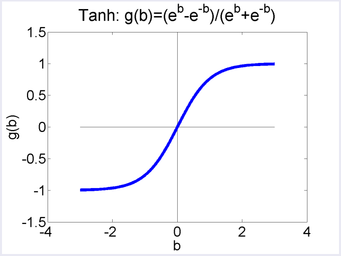
$$\frac{d\tanh(b)}{db} = \left(1 - \tanh^2(b)\right)$$

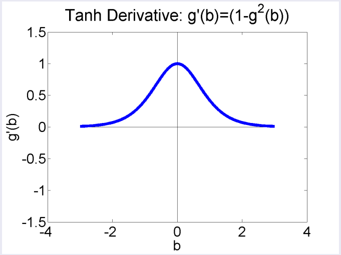
Signum function and its derivative



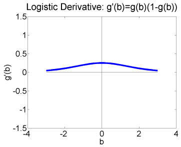
- The derivative of the signum function is the Dirac delta, which is not very useful in backprop.

Tanh function and its derivative





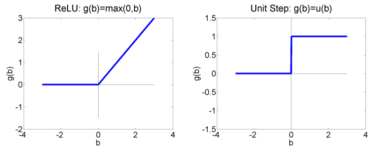
A suprising problem with the sigmoid: Vanishing gradients



The sigmoid has a surprising problem: for large values of w, $\sigma'(wx) \rightarrow 0$.

- When we begin training, we start with small values of w. $\sigma'(wx)$ is reasonably large, and training proceeds.
- If w and $\nabla_w \mathcal{L}$ are vectors in opposite directions, then $w \leftarrow w - \eta \nabla_w \mathcal{L}$ makes w larger. After a few iterations, w gets very large. At that point, $\sigma'(wx) \rightarrow 0$, and training effectively stops.
- After that point, even if the neural net sees new training data that don't match what it has already learned, it can no longer change. We say that it has suffered from the "vanishing gradient problem."

A solution to the vanishing gradient problem: ReLU

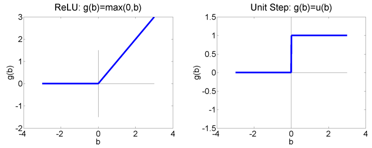


The most ubiquitous solution to the vanishing gradient problem is to use a ReLU (rectified linear unit) instead of a sigmoid. The ReLU is given by

$$\text{ReLU}(b) = \begin{cases} b & b \geq 0 \\ 0 & b \leq 0, \end{cases}$$

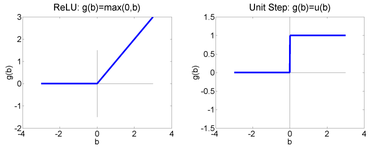
and its derivative is the unit step. Notice that the unit step is equally large ($u(wx) = 1$) for any positive value ($wx > 0$), so no matter how large w gets, back-propagation continues to work.

A solution to the vanishing gradient problem: ReLU



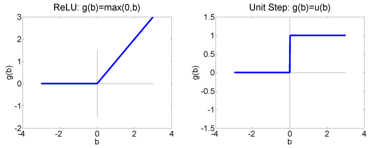
- **Pro:** The ReLU derivative is equally large ($\frac{d\text{ReLU}(wx)}{d(wx)} = 1$) for any positive value ($wx > 0$), so no matter how large w gets, back-propagation continues to work.
- **Con:** If the ReLU is used as a hidden unit ($h_j = \text{ReLU}(e_j)$), then your output is no longer a piece-wise constant approximation of $\vec{y}$. It is now piece-wise linear.
- On the other hand, maybe piece-wise linear is better than piece-wise constant, so. . .

A solution to the vanishing gradient problem: the ReLU



- **Pro:** The ReLU derivative is equally large ($\frac{d\text{ReLU}(wx)}{d(wx)} = 1$) for any positive value ($wx > 0$), so no matter how large w gets, back-propagation continues to work.
- **Pro:** If the ReLU is used as a hidden unit ($h_j = \text{ReLU}(e_j)$), then your output is no longer a piece-wise constant approximation of $\vec{y}$. It is now piece-wise linear.
- **Con:** ??

The dying ReLU problem



- **Pro:** The ReLU derivative is equally large ($\frac{d\text{ReLU}(wx)}{d(wx)} = 1$) for any positive value ($wx > 0$), so no matter how large w gets, back-propagation continues to work.
- **Pro:** If the ReLU is used as a hidden unit ($h_j = \text{ReLU}(e_j)$), then your output is no longer a piece-wise constant approximation of $\vec{y}$. It is now piece-wise linear.
- **Con:** If $wx + b < 0$, then ($\frac{d\text{ReLU}(wx)}{d(wx)} = 0$), and learning stops. In the worst case, if b becomes very negative, then all of the hidden nodes are turned off—the network computes nothing, and no learning can take place! This is called the "Dying ReLU problem."

Solutions to the Dying ReLU problem

- **Softplus:** Pro: always positive. Con: gradient$\rightarrow 0$ as $x \rightarrow -\infty$.

$$f(x) = \ln\left(1 + e^x\right)$$

- **Leaky ReLU:** Pro: gradient constant, output piece-wise linear. Con: negative part might fail to match your dataset.

$$f(x) = \begin{cases} x & x \geq 0 \\ 0.01x & x \leq 0 \end{cases}$$

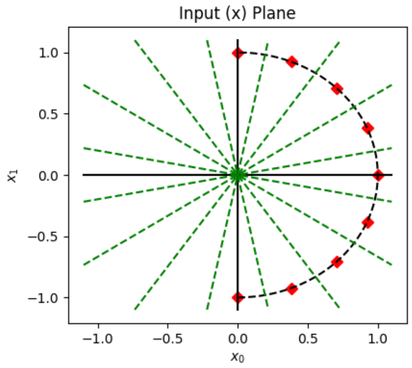
- **Parametric ReLU (PReLU:)** Pro: gradient constant, ouput PWL. The slope of the negative part (a) is a trainable parameter, so can adapt to your dataset. Con: you have to train it.

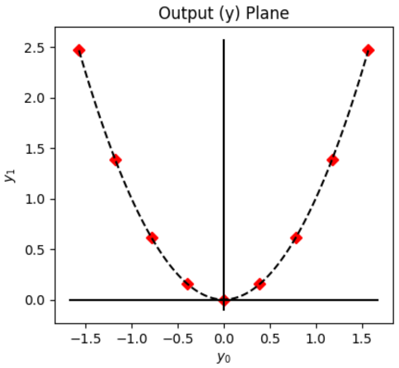
$$f(x) = \begin{cases} x & x \geq 0 \\ ax & x \leq 0 \end{cases}$$

Outline

Example #2: Semicircle → Parabola

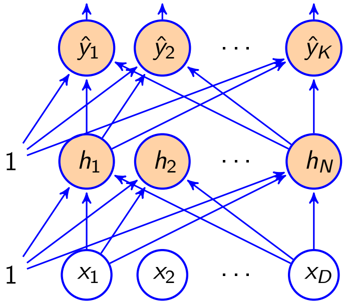
Can we design a neural net that converts a semicircle ($x_0^2 + x_1^2 = 1$) to a parabola ($y_1 = y_0^2$)?





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 → 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^{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^{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:

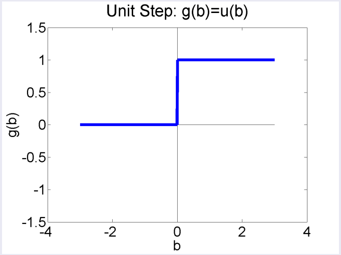
$$\hat{y} = \vec{b}^{(2)} + \sum_j \vec{w}_j^{(2)} h_j$$

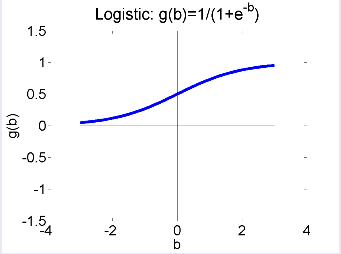
The activation, h_j, is a number between 0 and 1. For example, we could use the logistic sigmoid function:

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

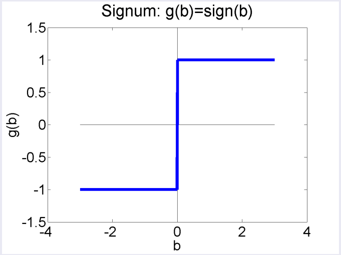
The logistic sigmoid is a differentiable approximation to a unit step function.

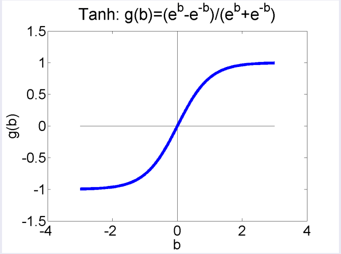
Step and Logistic nonlinearities





Signum and Tanh nonlinearities



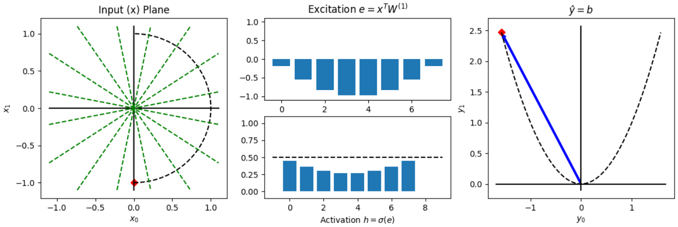


First Layer = A Series of Decisions

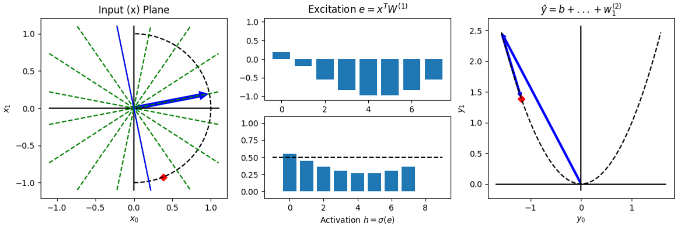
The first layer of the network decides whether or not to "turn on" each of the h_j's. It does this by comparing $\vec{x}$ to a series of linear threshold vectors:

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

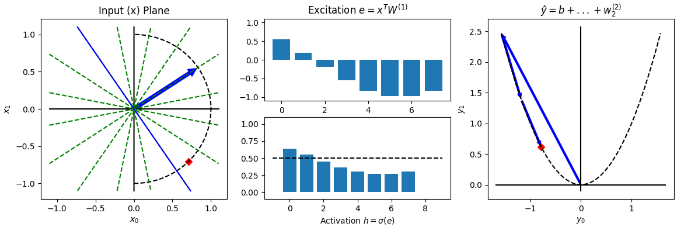
Example #2: Semicircle → Parabola



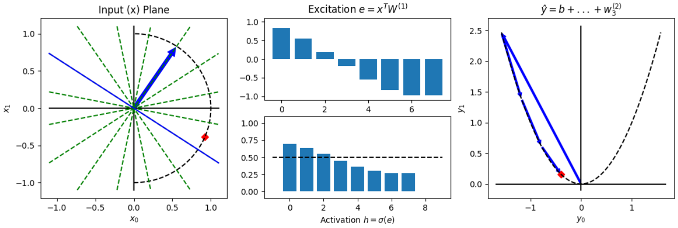
Example #2: Semicircle $\rightarrow$ Parabola



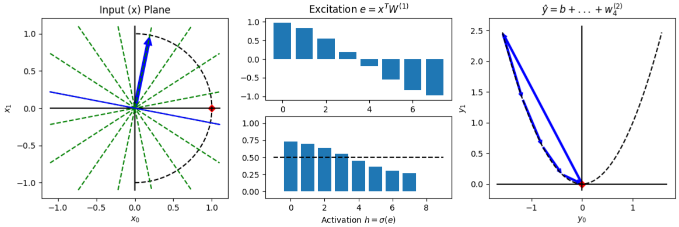
Example #2: Semicircle → Parabola



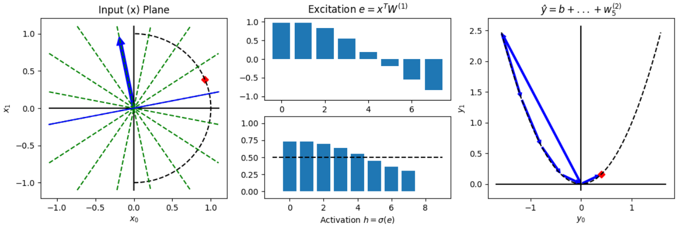
Example #2: Semicircle → Parabola



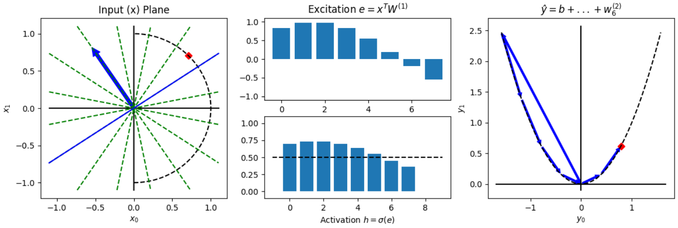
Example #2: Semicircle → Parabola



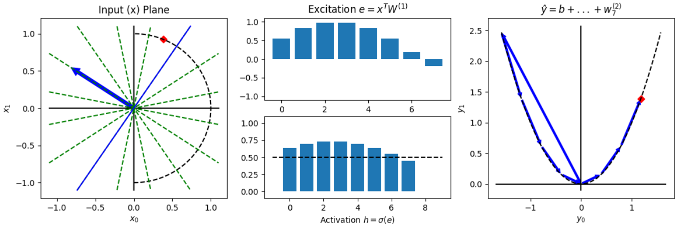
Example #2: Semicircle → Parabola



Example #2: Semicircle $\rightarrow$ Parabola



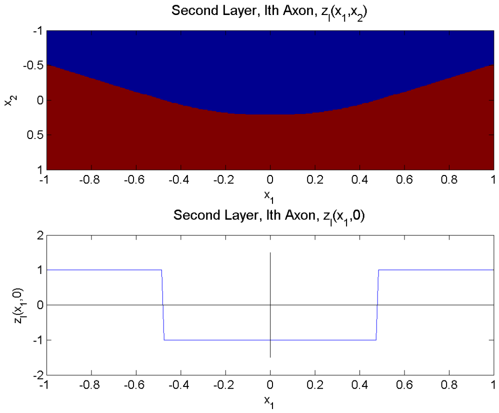
Example #2: Semicircle → Parabola



Outline

A classifier target funtion

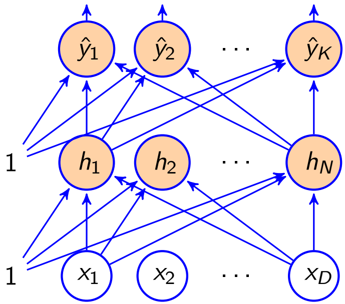
A “classifier” is a neural network with discrete ouputs. For example, suppose you need to color a 2D picture. The goal is to output $\hat{y}(\vec{x}) = 1$ if $\vec{x}$ should be red, and $\hat{y} = -1$ if $\vec{x}$ should be blue:



A classifier neural network

We can discretize the output by simply using an output nonlinearity, e.g., $\hat{y}_k = g(e_k^{(2)})$, for some nonlinearity $g(x)$:

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



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

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

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

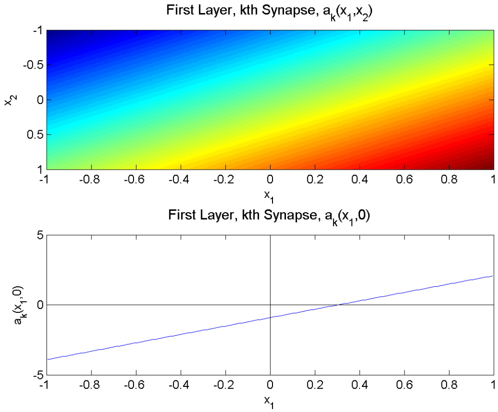
$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

Nonlinearities for classifier neural networks

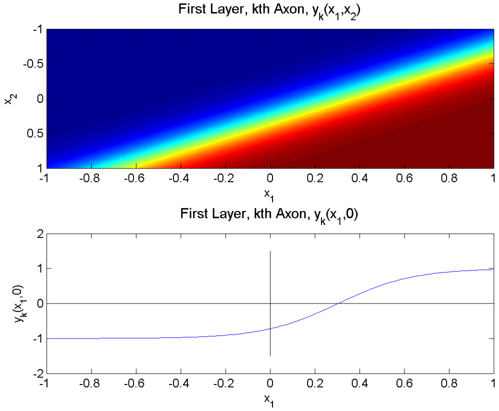
- **During testing:** the output is passed through a hard nonlinearity, e.g., a unit step or a signum.
- **During training:** the output is passed through the corresponding soft nonlinearity, e.g., sigmoid or tanh.

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

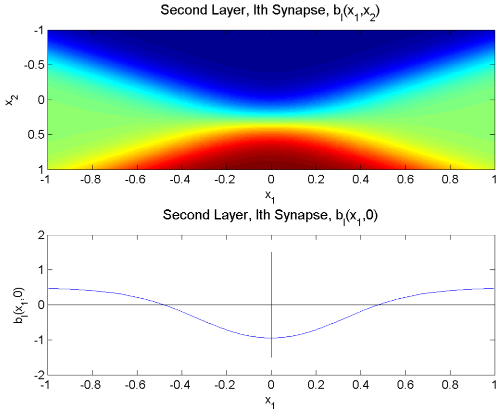


Activation, First Layer: $h_k = \tanh(e_k^{(1)})$

Here, I'm using tanh as the nonlinearity for the hidden layer. But it often works better if we use ReLU or PReLU.

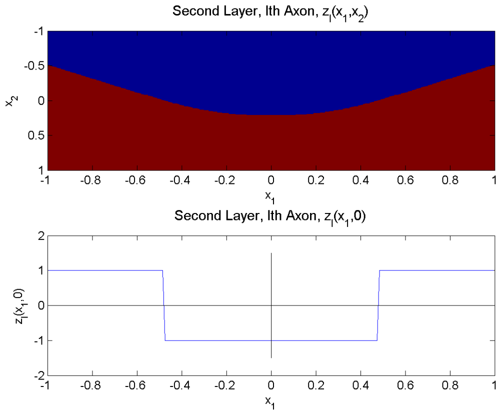


Excitation, Second Layer: $e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{2} w_{kj}^{(2)} h_j$



Activation, Second Layer: $\hat{y}_k = \text{sign}(e_k^{(2)})$

During training, the output layer uses a soft nonlinearity. During testing, though, the soft nonlinearity is replaced with a hard nonlinearity, e.g., signum:



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.

Nonlinearities Summarized

- Unit-step and signum nonlinearities, on the hidden layer, cause the neural net to compute a piece-wise constant approximation of the target function. Unfortunately, they're not differentiable, so they're not trainable.
- Sigmoid and tanh are differentiable approximations of unit-step and signum, respectively. Unfortunately, they suffer from a vanishing gradient problem: as the weight matrix gets larger, the derivatives of sigmoid and tanh go to zero, so error doesn't get back-propagated through the nonlinearity any more.
- ReLU has the nice property that the output is a piece-wise-linear approximation of the target function, instead of piece-wise constant. It also has no vanishing gradient problem. Instead, it has the dying-ReLU problem.
- Softplus, Leaky ReLU, and PReLU are different solutions to the dying-ReLU problem.