Lecture 17: Neural Nets

Mark Hasegawa-Johnson

ECE 417: Multimedia Signal Processing, Fall 2023

1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle $\rightarrow$ Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle $\rightarrow$ Parabola
9. Summary

Outline

1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle $\rightarrow$ Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle $\rightarrow$ Parabola
9. Summary

What is a Neural Network?

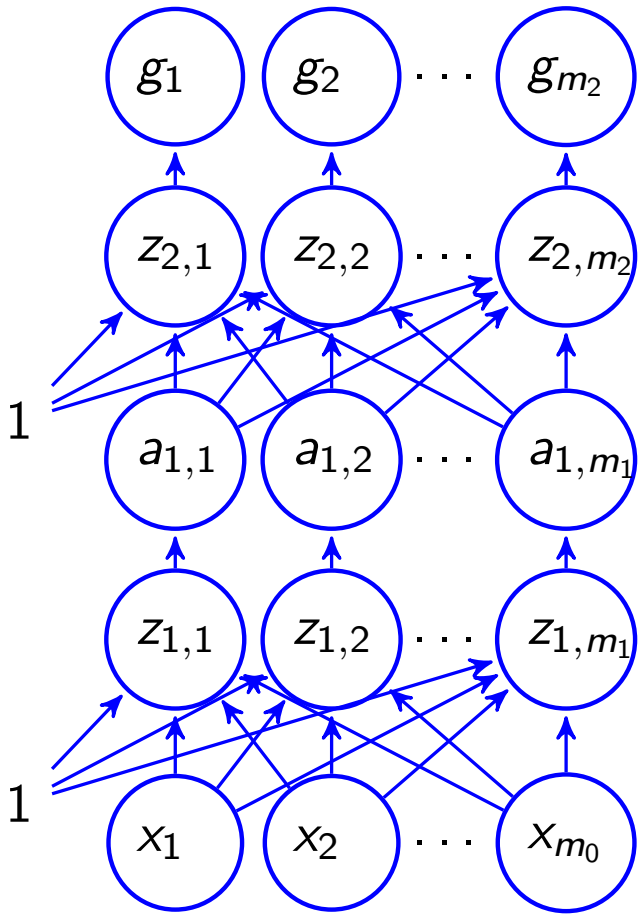
- Computation in biological neural networks is performed by billions of simple cells (neurons), each of which performs one very simple computation.
- Biological neural networks learn by strengthening the connections between some pairs of neurons, and weakening other connections.

What is an Artificial Neural Network?

- Computation in an artificial neural network is performed by millions 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

$$\mathcal{L} = E\left[-\ln \Pr(\mathbf{y}|\mathbf{g}(\mathbf{x}))\right]$$



$$\mathbf{g}(\mathbf{x}) = \mathbf{a}_2 = \mathbf{f}_2(\mathbf{z}_2)$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2$$

$$\mathbf{a}_1 = \mathbf{f}_1(\mathbf{z}_1)$$

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$

$\mathbf{a}_0 = \mathbf{x}$ is the input vector

Why use neural nets?

Barron (1993) showed that a neural net is a **universal approximator:** it can approximate any function arbitrarily well, if the number of hidden nodes is large enough. Assume:

- Linear output nodes: $\mathbf{g}(\mathbf{x}) = \mathbf{a}_2 = \mathbf{z}_2$
- Hidden nodes are n smooth scalar nonlinearities:

$$a_{1,k} = f_1(z_{1,k}), \quad 1 \leq k \leq m_1, \text{ where } \frac{\partial a_{1,k}}{\partial z_{1,k}} \text{ is finite}$$

- Smooth target function: The target vectors $\mathbf{y}$ are all computed by some function $\mathbf{y}(\mathbf{x})$ that is unknown but smooth (its Fourier transform has finite first moment)

$$\text{Then:} \quad \max_{\mathbf{y}(\mathbf{x})} \min_{\mathbf{g}(\mathbf{x})} E\left[\|\mathbf{y}(\mathbf{x}) - \mathbf{g}(\mathbf{x})\|^2\right] \leq \mathcal{O}\left\{\frac{1}{m_1}\right\}$$

Outline

1 Intro

2 Classification Example: Arbitrary Classifier

3 Regression Example: Semicircle → Parabola

4 Scalar Nonlinearities

5 Loss Functions

6 Learning: Gradient Descent

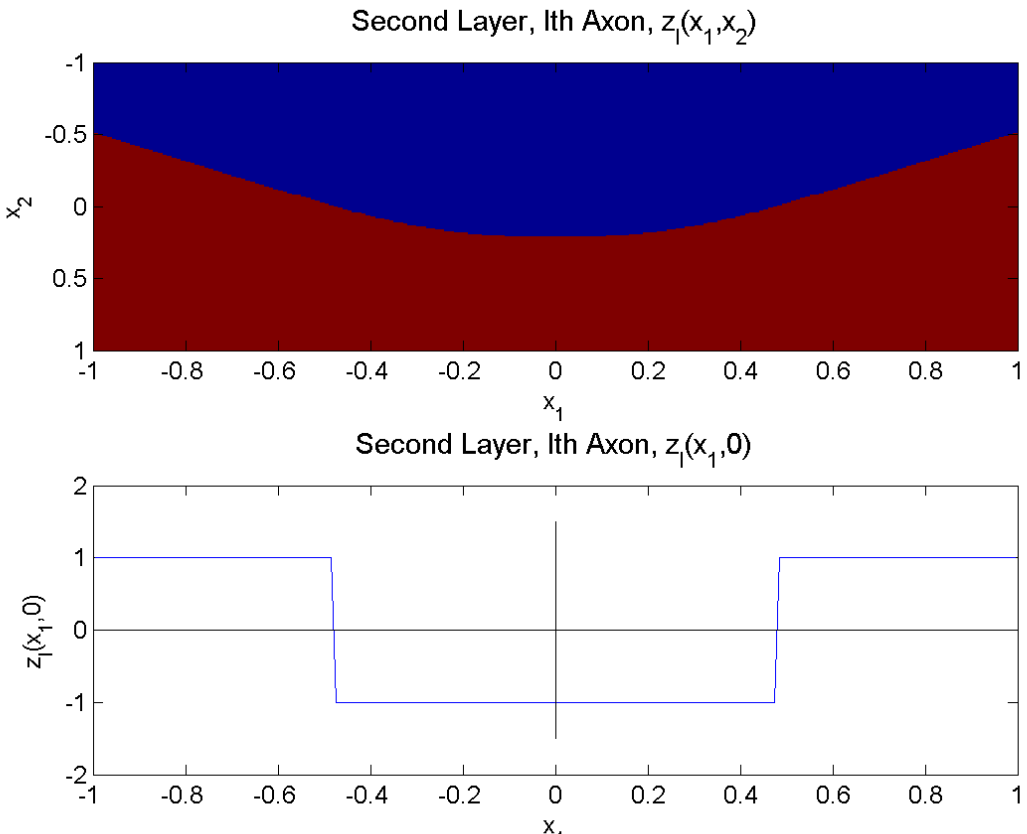
7 Back-Propagation

8 Backprop Example: Semicircle → Parabola

9 Summary

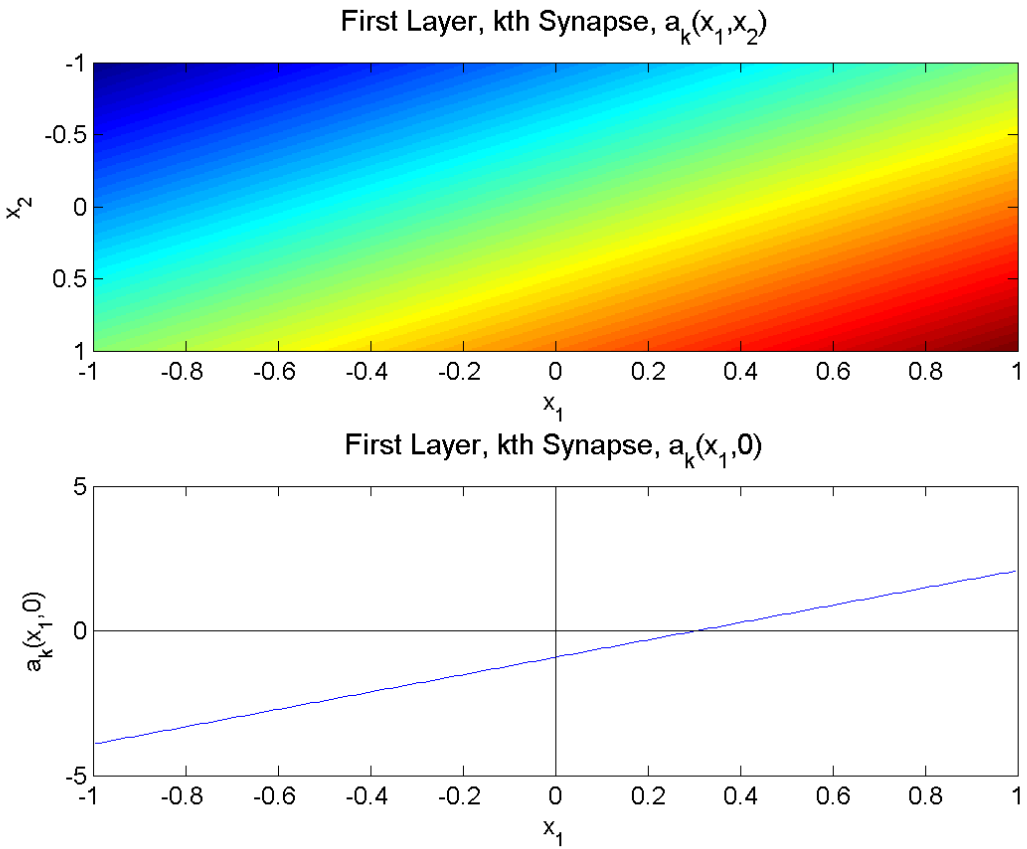
Target: Can we get the neural net to compute this function?

Suppose our goal is to find some weights and biases, $\mathbf{W}_1$, $\mathbf{b}_1$, $\mathbf{W}_2$, and $\mathbf{b}_2$ so that the output activation is a scalar binary classifier that classifies every pixel $\mathbf{x} = [x_1, x_2]$ as being either "red" or "blue," where the ground truth is shown in this picture:



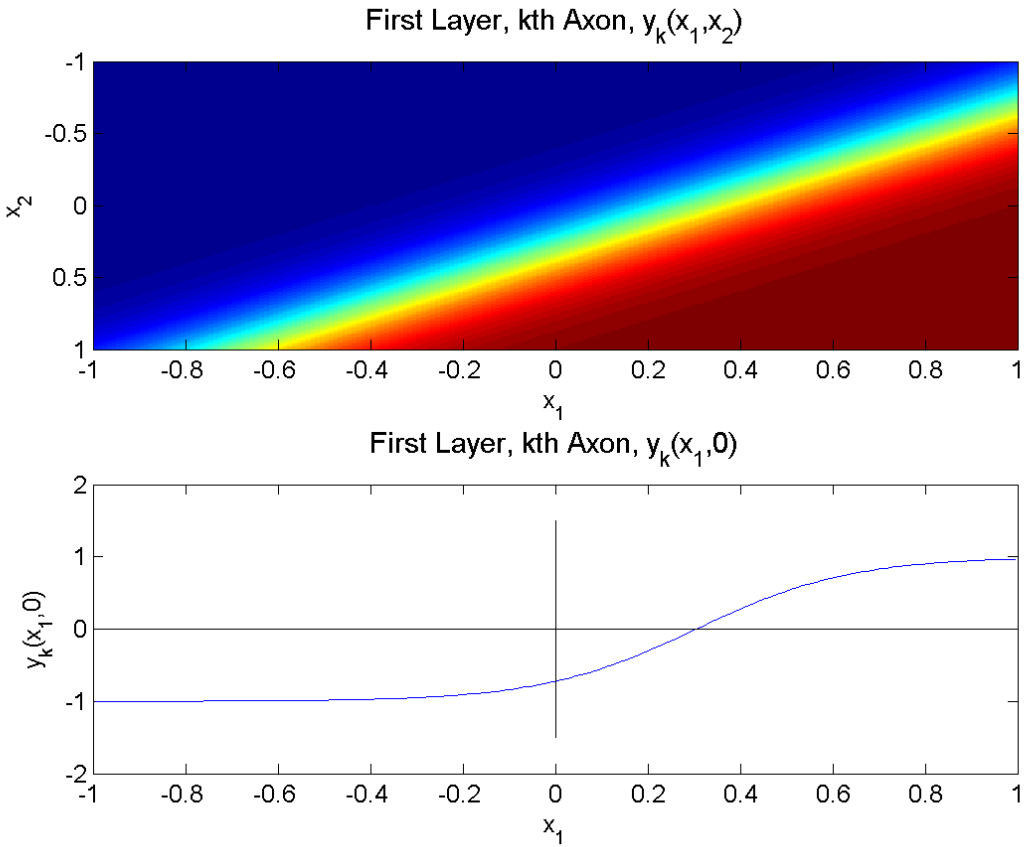
First Layer Excitation: $\mathbf{z}_1 = \mathbf{b}_1 + \mathbf{W}_1\mathbf{x}$

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



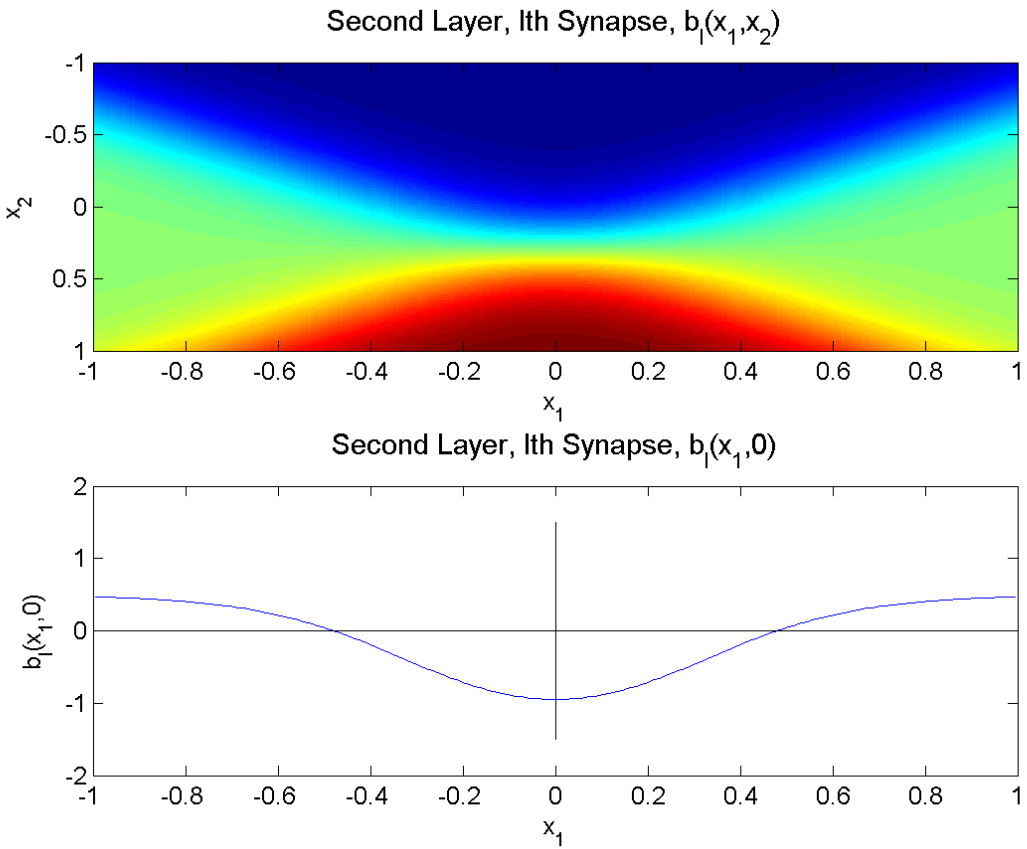
First Layer Activation: $\mathbf{a}_1 = \tanh(\mathbf{z}_1)$

The activation nonlinearity then "squashes" the linear function:



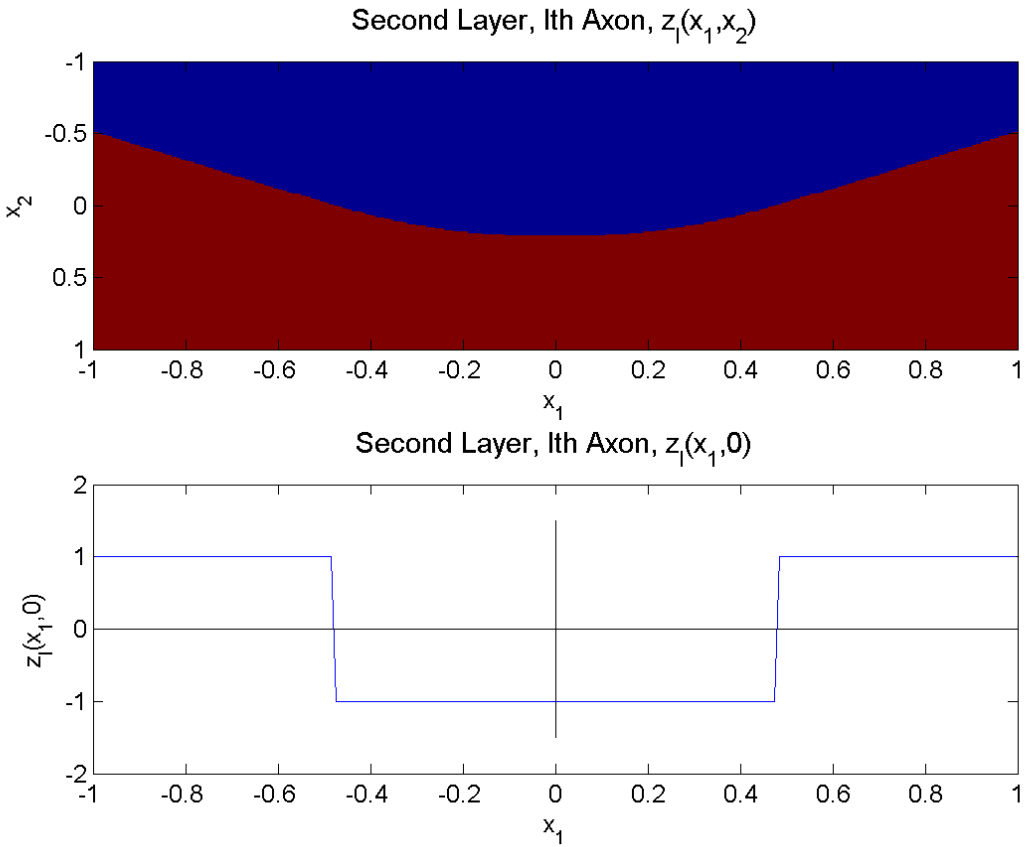
Second Layer Excitation: $z_2 = b_2 + \mathbf{w}_2^T \mathbf{a}_1$

The second layer then computes a linear combination of the first-layer activations:



Second Layer Activation: $a_2 = u(z_2)$

The second layer activation is then just a binary threshold, $a_2 = 1$ if $z_2 > 0$, otherwise $a_2 = 0$:

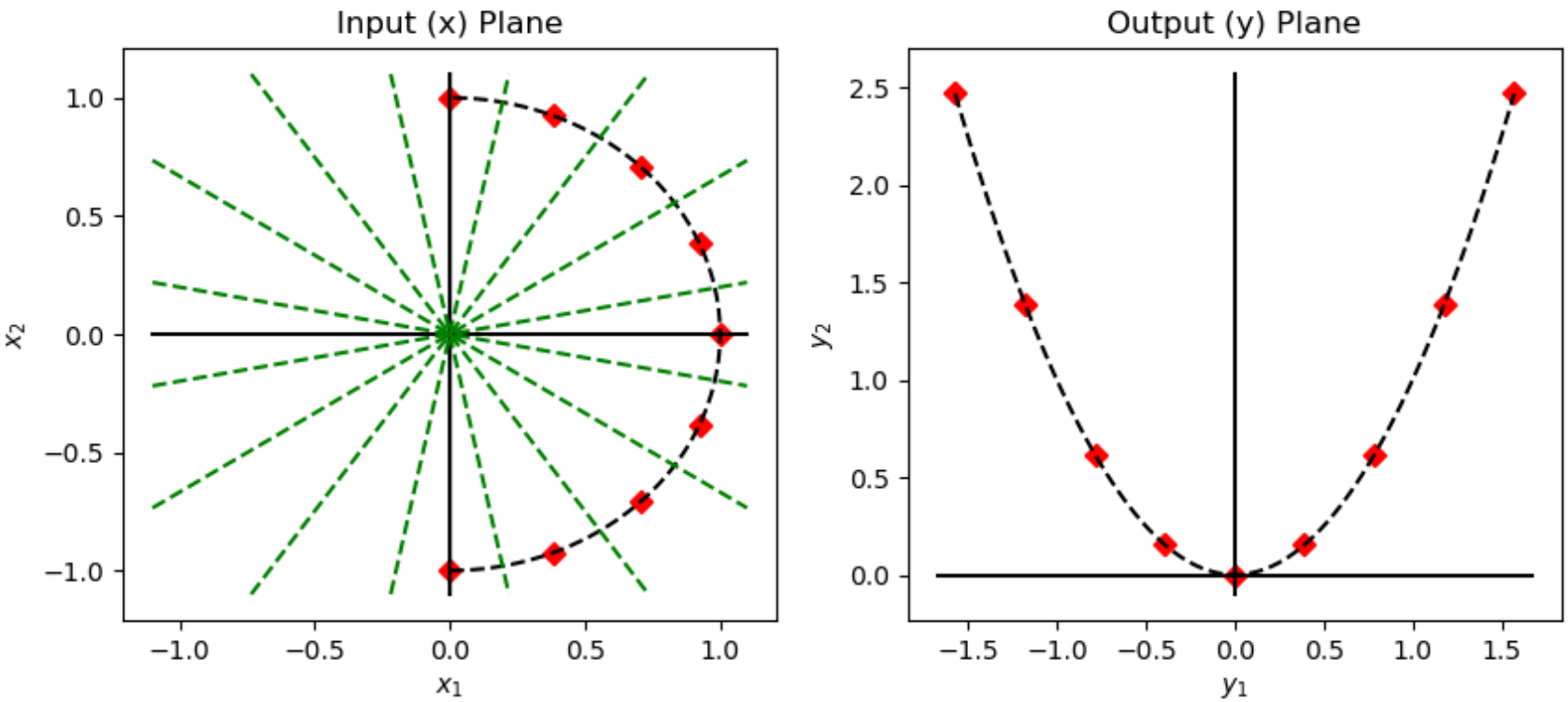


Outline

1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle → Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle → Parabola
9. Summary

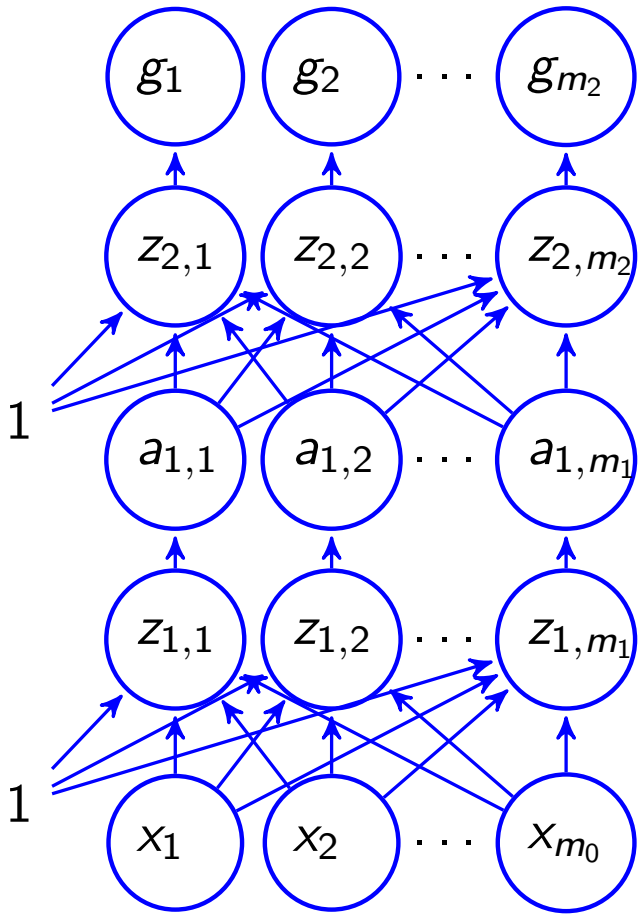
Example #2: Semicircle → Parabola

Can we design a neural net that converts a semicircle ($x_1^2 + x_2^2 = 1$) to a parabola ($y_2 = y_1^2$)?



Two-Layer Feedforward Neural Network

$$\mathcal{L} = E\left[-\ln \Pr(\mathbf{y}|\mathbf{g}(\mathbf{x}))\right]$$



$$\mathbf{g}(\mathbf{x}) = \mathbf{a}_2 = \mathbf{f}_2(\mathbf{z}_2)$$

$$\mathbf{z}_2 = \mathbf{W}_2\mathbf{a}_1 + \mathbf{b}_2$$

$$\mathbf{a}_1 = \mathbf{f}_1(\mathbf{z}_1)$$

$$\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$$

$\mathbf{a}_0 = \mathbf{x}$ is the input vector

Example #2: Semicircle → Parabola

Let's define some notation:

- **Second Layer:** Define $\mathbf{w}_{2,:,j} = \begin{bmatrix} w_{2,1,j} \\ w_{2,n,j} \end{bmatrix}$, the j^{th} column of the $\mathbf{W}_2$ matrix, so that

$$\mathbf{W}_2 = [\mathbf{w}_{2,:,1}, \mathbf{w}_{2,:,2}]$$

- **First Layer Excitation:** Define $\mathbf{w}_{1,k,:}^T = [w_{1,k,1}, \ldots, w_{1,k,m}]$, the k^{th} row of the $\mathbf{W}_1$ matrix, so that

$$\mathbf{W}_1 = \begin{bmatrix} \mathbf{w}_{1,1,:}^T \\ \mathbf{w}_{1,2,:}^T \end{bmatrix}$$

Piece-Wise Constant Approximation

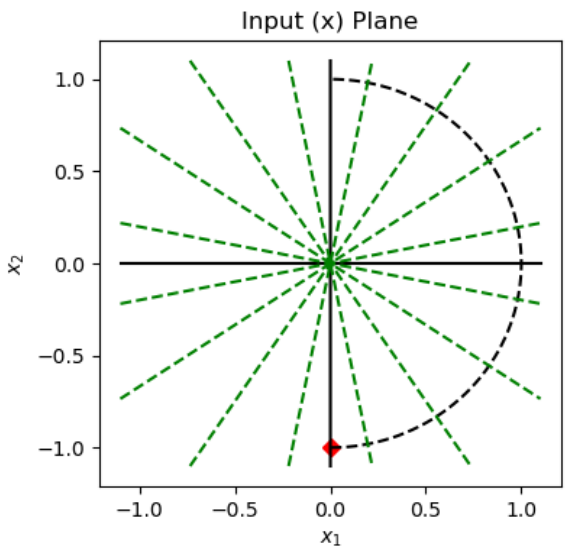
Suppose we want the NN to calculate a piece-wise constant approximation: The second layer of the network approximates $\mathbf{y}$ using a bias term $\mathbf{b}_2$, plus correction vectors $\mathbf{w}_{2,:,j}$, each scaled by its activation $a_{1,j}$:

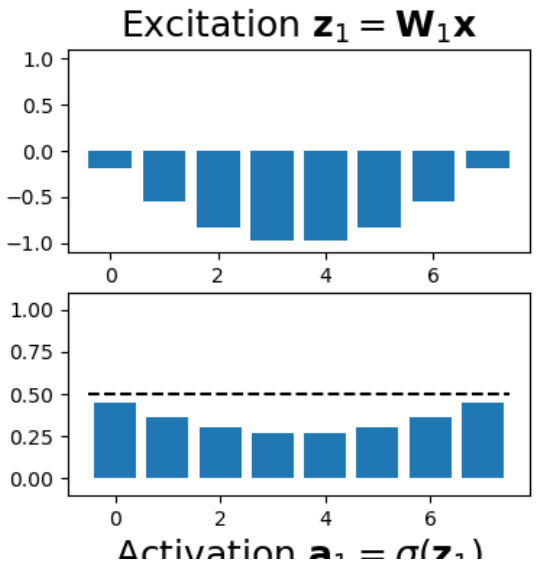
$$\mathbf{g}(\mathbf{x}) = \mathbf{b}_2 + \sum_j \mathbf{w}_{2,:,j} a_{1,j}$$

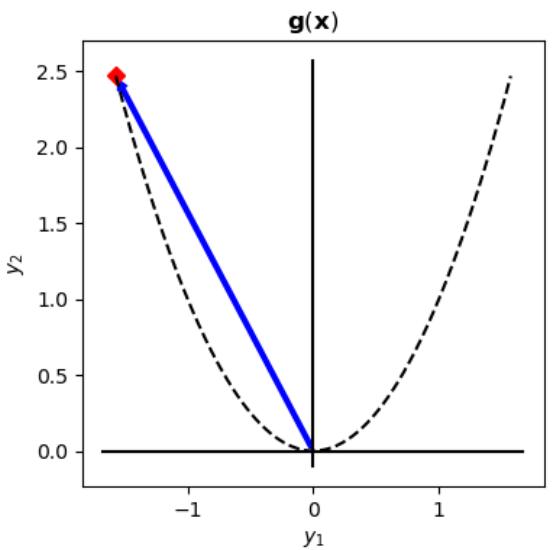
The first layer of the network decides whether or not to "turn on" each of the $a_{1,j}$'s. It does this by comparing $\mathbf{x}$ to a series of linear threshold vectors:

$$a_{1,j} = \sigma\left(\mathbf{w}_{1,k,:}^T \mathbf{x}\right) \approx \begin{cases} 1 & \mathbf{w}_{1,k,:}^T \mathbf{x} > 0 \\ 0 & \mathbf{w}_{1,k,:}^T \mathbf{x} < 0 \end{cases}$$

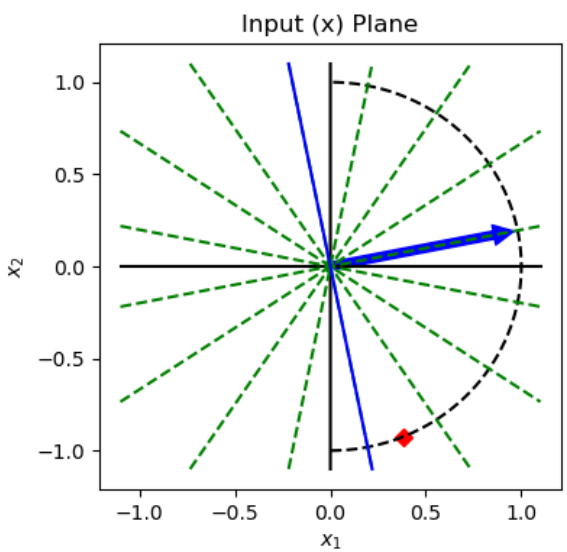
Example #2: Semicircle → Parabola

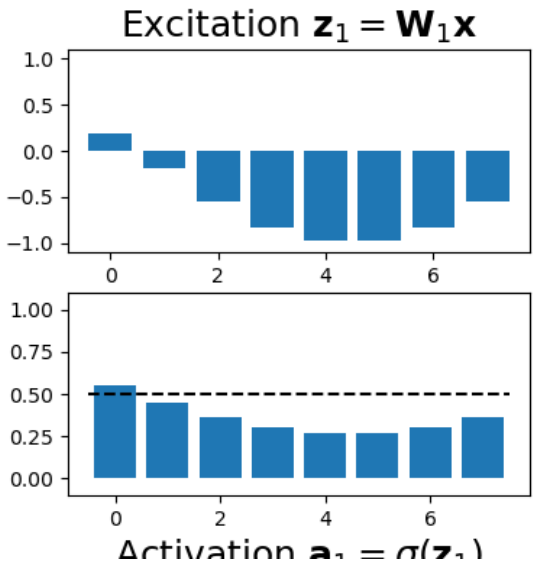


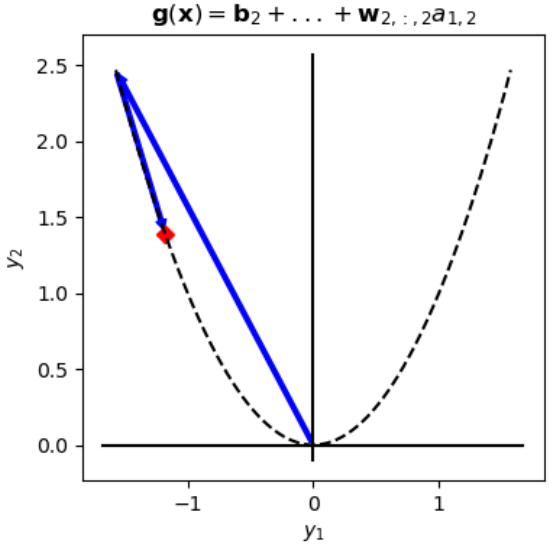




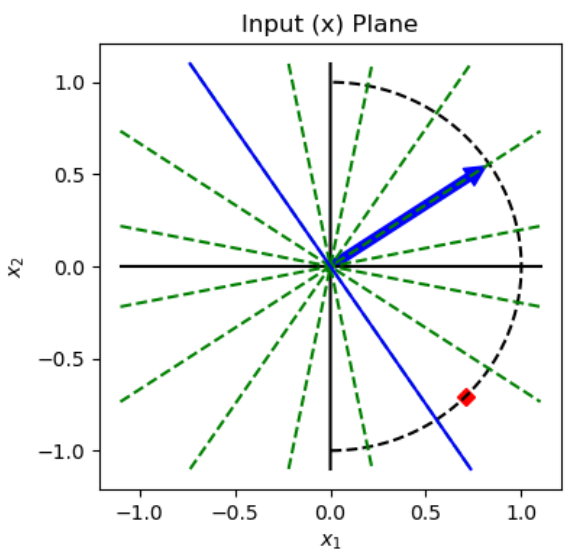
Example #2: Semicircle → Parabola

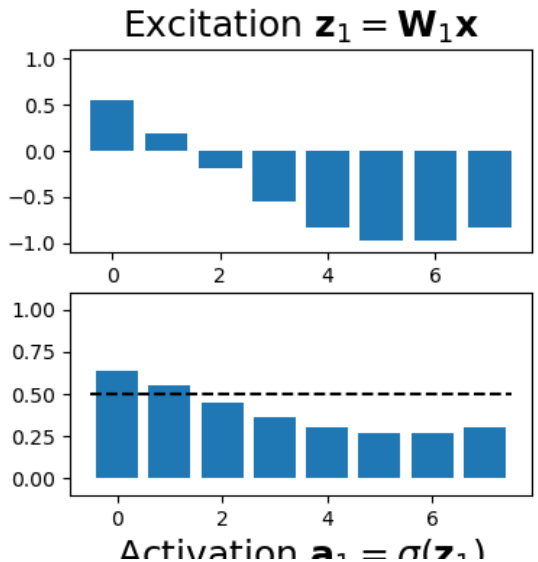


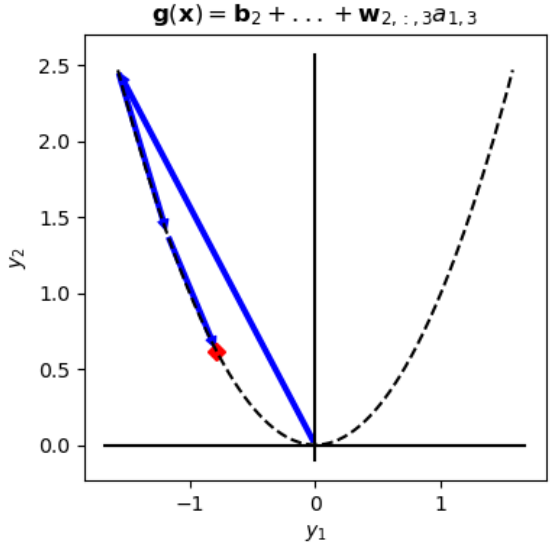




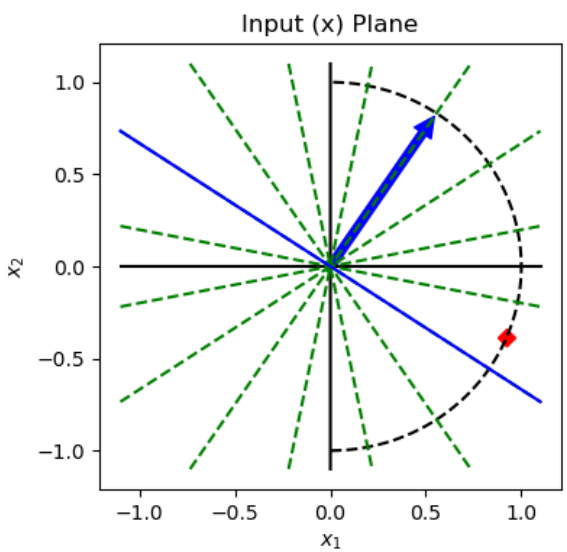
Example #2: Semicircle → Parabola

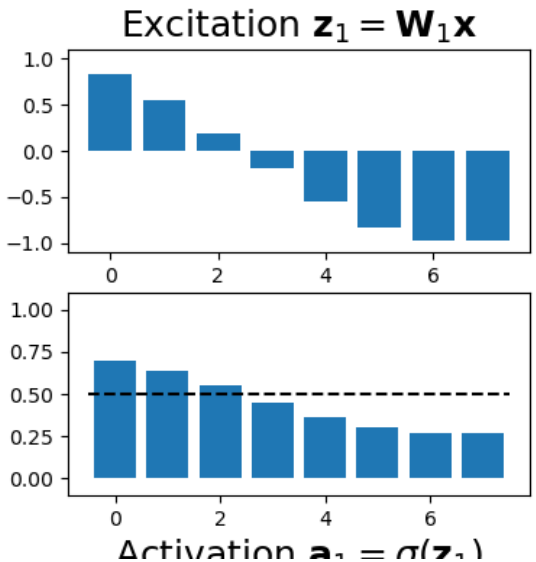


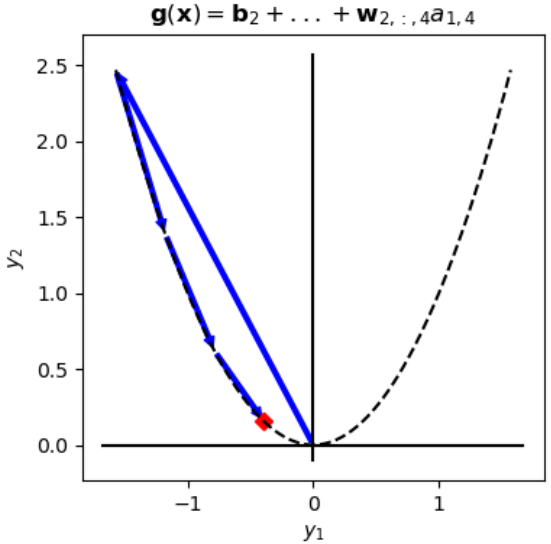




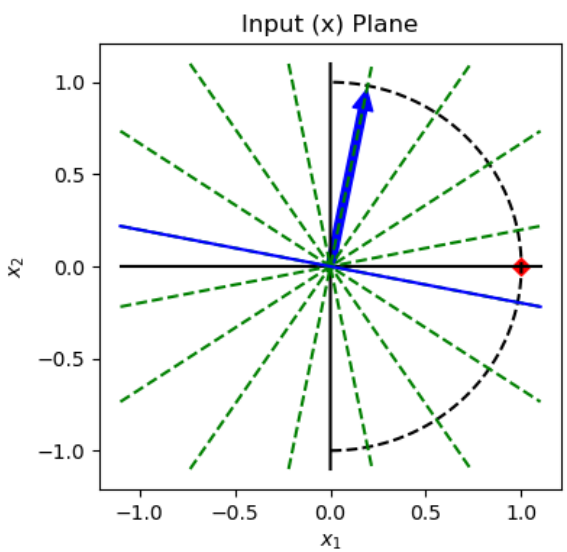
Example #2: Semicircle → Parabola

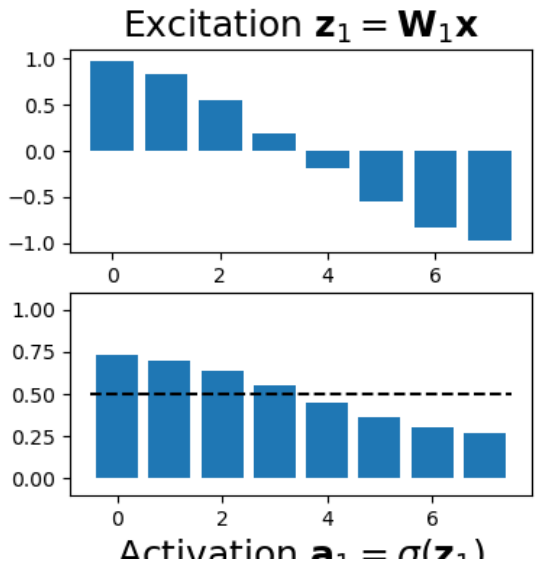


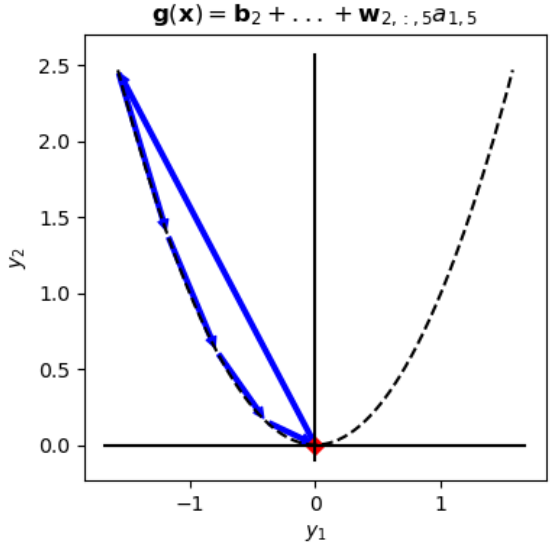




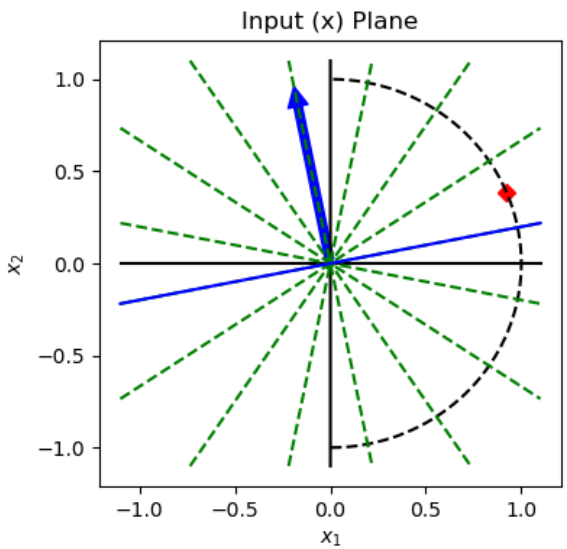
Example #2: Semicircle → Parabola

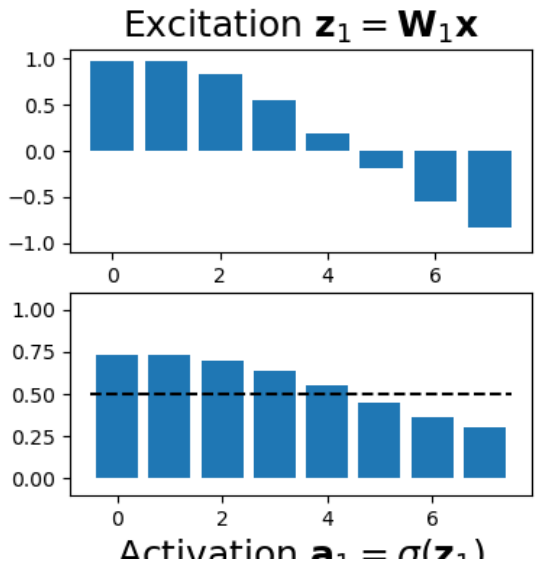


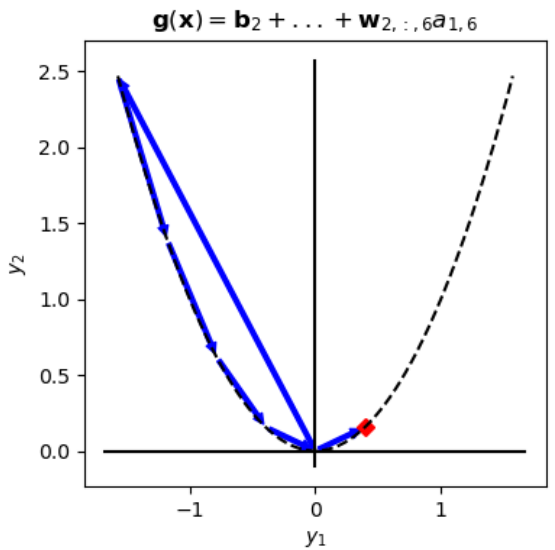




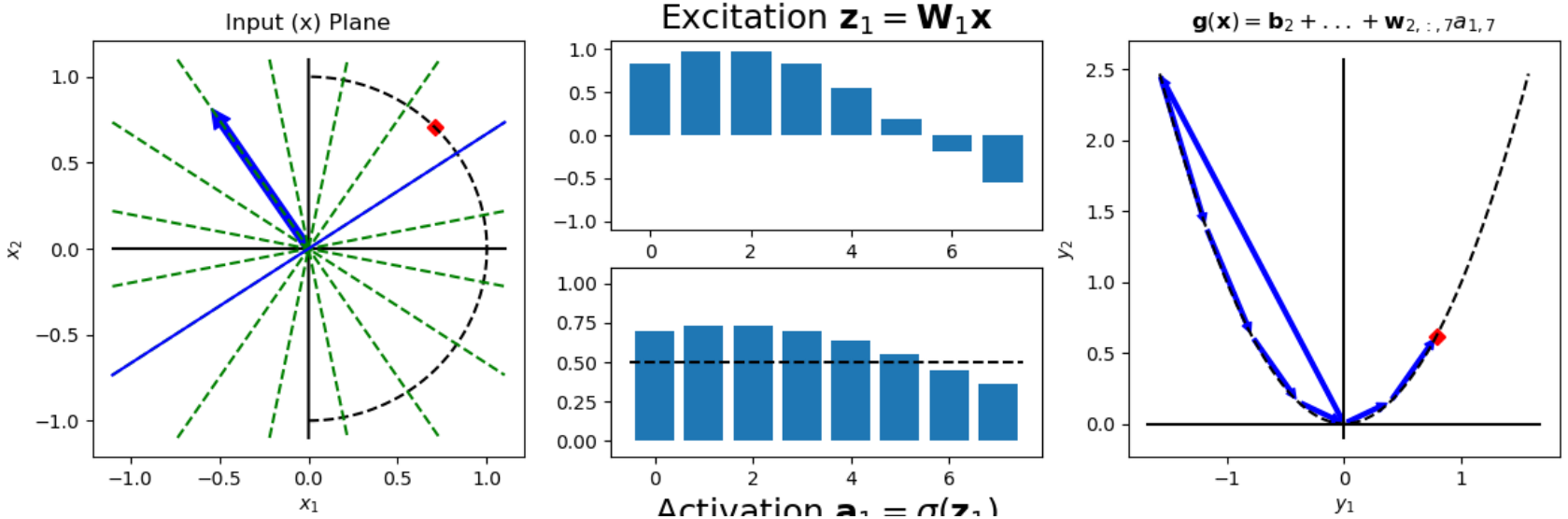
Example #2: Semicircle → Parabola



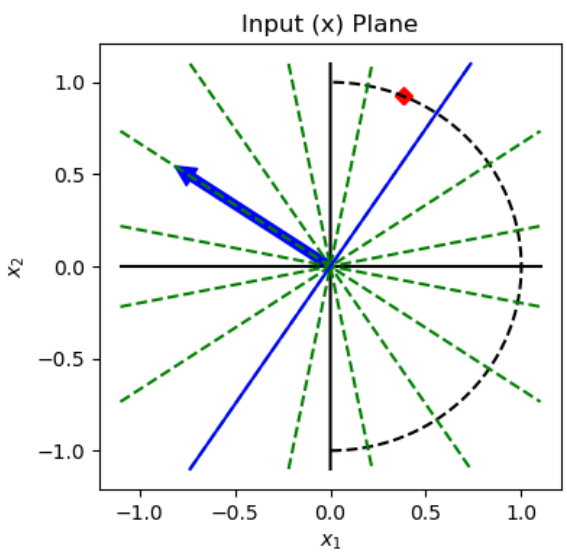


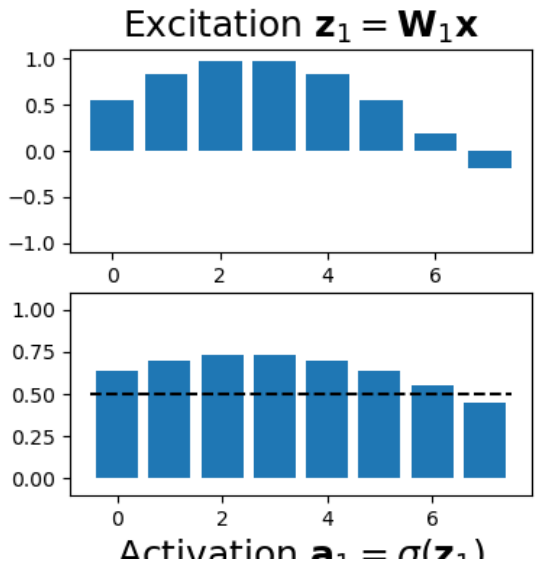


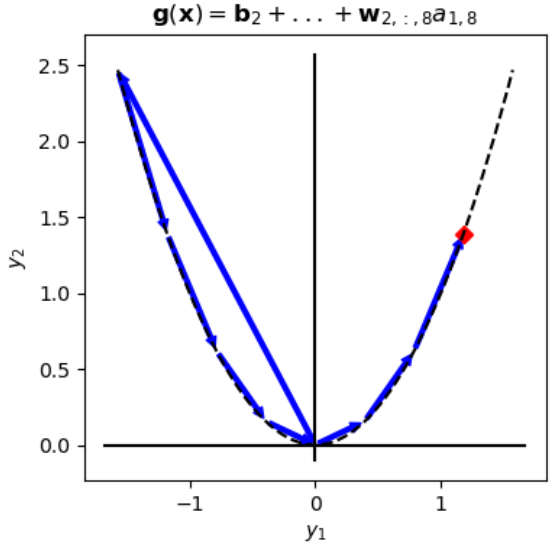
Example #2: Semicircle → Parabola



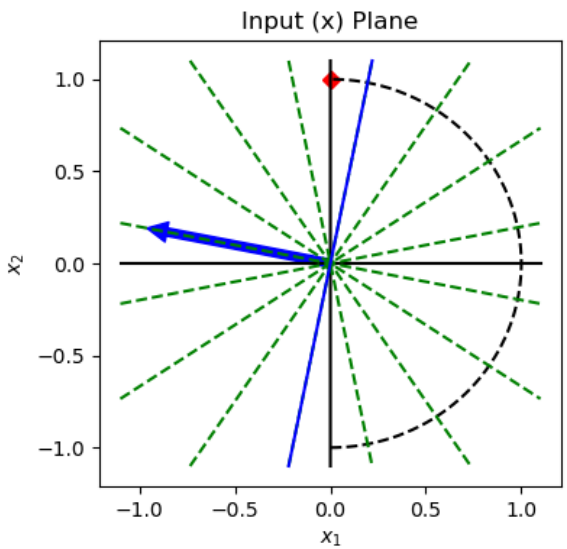
Example #2: Semicircle → Parabola

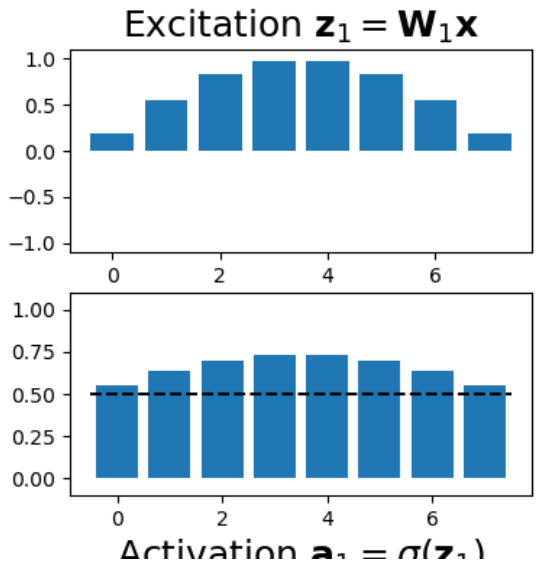


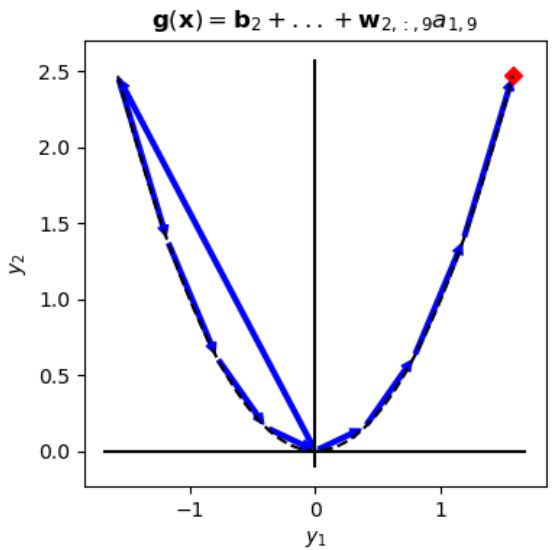




Example #2: Semicircle → Parabola







Outline

1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle $\rightarrow$ Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle $\rightarrow$ Parabola
9. Summary

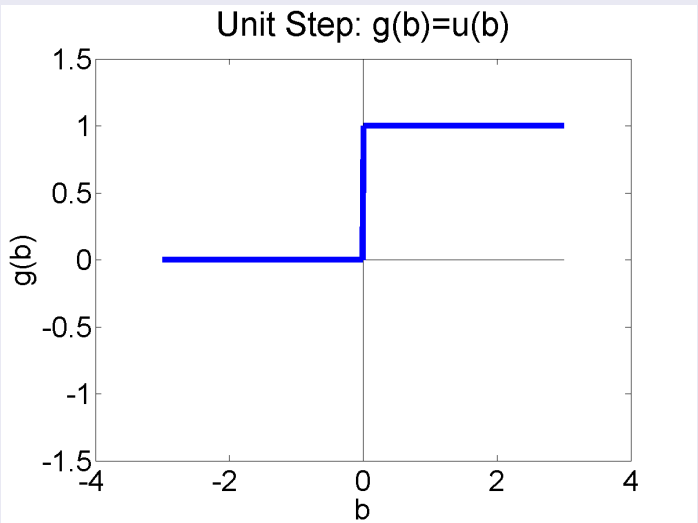
Scalar Nonlinearities

- A layer might have a vector nonlinearity, $\mathbf{a}_l = \mathbf{f}_l(\mathbf{z}_l)$, in which case every element of $\mathbf{a}_l$ depends on every element of $\mathbf{z}_l$. These are used rarely, because they're rarely necessary, and computation is hard. The only one of these we'll ever use is the softmax.
- Most NN nonlinearities are scalar, $\mathbf{a}_l = f_l(\mathbf{z}_l)$, which means that

$$\begin{bmatrix} a_{l,1} \\ \vdots \\ a_{l,m_l} \end{bmatrix} = \begin{bmatrix} f_l(z_{l,1}) \\ \vdots \\ f_l(z_{l,m_l}) \end{bmatrix}$$

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

$$u(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$



Pros and Cons of the Unit Step

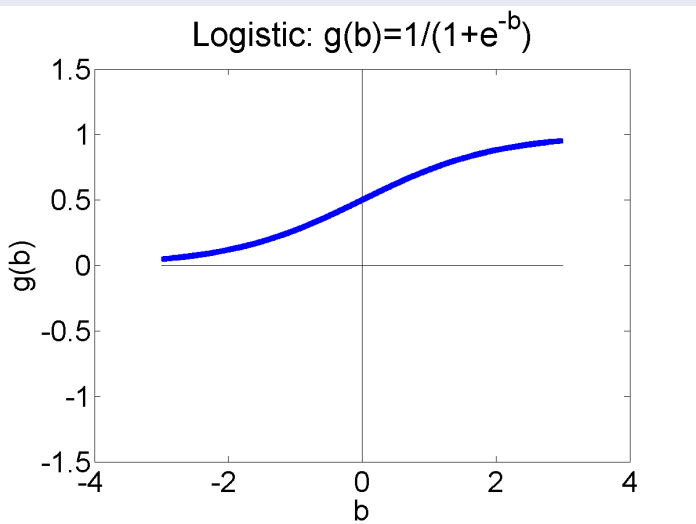
- **Pro:** It gives exactly piece-wise constant approximation of any desired y.
- **Con:** It's not differentiable, so we won't be able to use gradient descent to learn the network weights.

The derivative of the unit step is called the Dirac delta function, $\frac{\partial u}{\partial z} = \delta(z)$, where $\delta(z)$ is defined to be the function such that $\delta(0) = \infty$, $\delta(z) = 0$ for any $z \neq 0$, and

$$\int_{-\epsilon}^{\epsilon} \delta(z)dz = 1 \ \forall \epsilon \in \Re_+$$

The Differentiable Approximation: Logistic Sigmoid

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



Why to use the logistic function

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

and $\sigma(z)$ is smoothly differentiable, so we can use gradient descent for training.

Derivative of a sigmoid

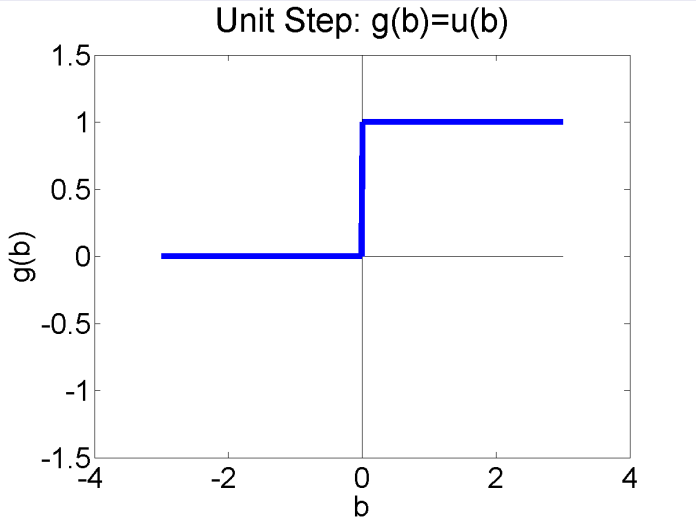
The derivative of a sigmoid is pretty easy to calculate:

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

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

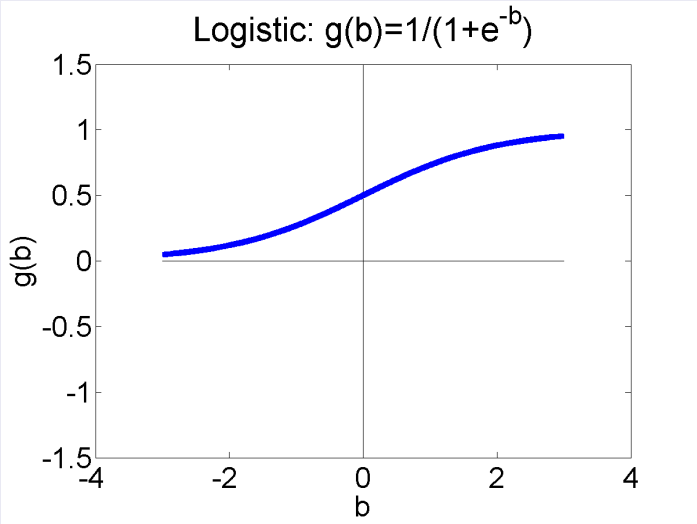
$$\begin{aligned}
\frac{d\sigma}{dz} &= \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \left(\frac{1}{1+e^{-z}}\right)\left(\frac{e^{-z}}{1+e^{-z}}\right) \\
&= \left(\frac{1}{1+e^{-z}}\right)\left(1-\frac{1}{1+e^{-z}}\right) \\
&= \sigma(z)(1-\sigma(z))
\end{aligned}$$

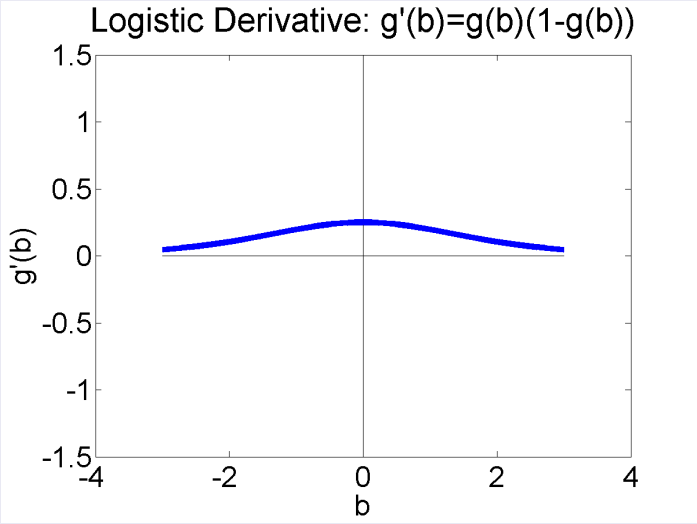
Step function and its derivative



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

Logistic function and its derivative





Signum and Tanh

The signum function is like the unit step, but two-sided. It is used if, for some reason, you want your output to be $y \in \{-1, 1\}$, instead of $y \in \{0, 1\}$:

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

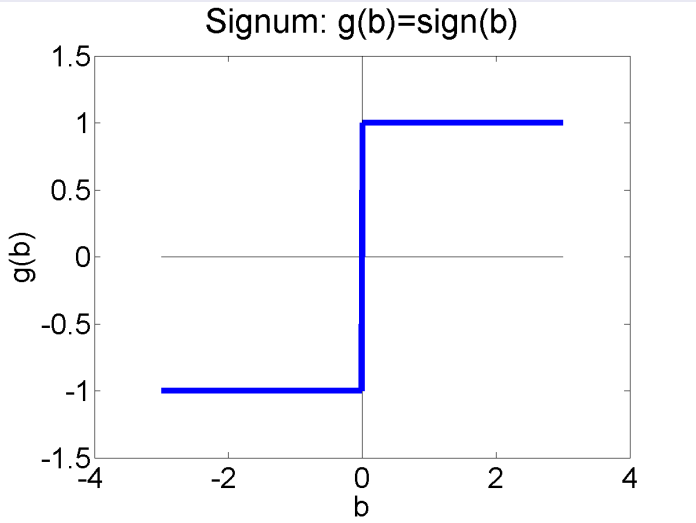
It is usually approximated by the hyperbolic tangent function (tanh), which is just a scaled shifted version of the sigmoid:

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

and which has a scaled version of the sigmoid derivative:

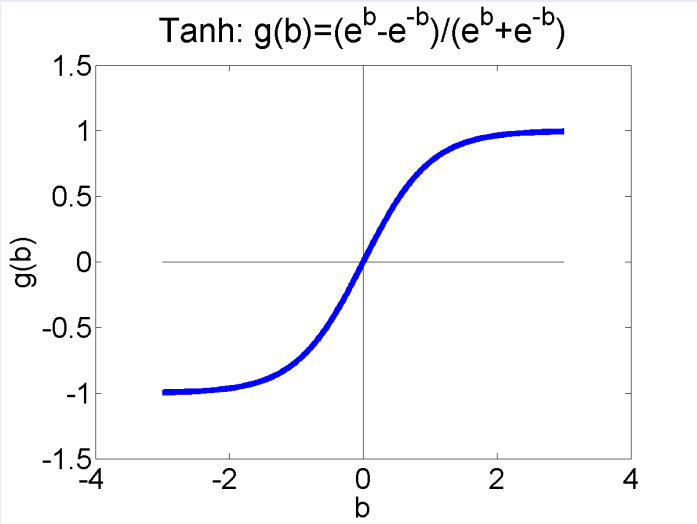
$$\frac{d \tanh(z)}{dz} = (1 - \tanh^2(z))$$

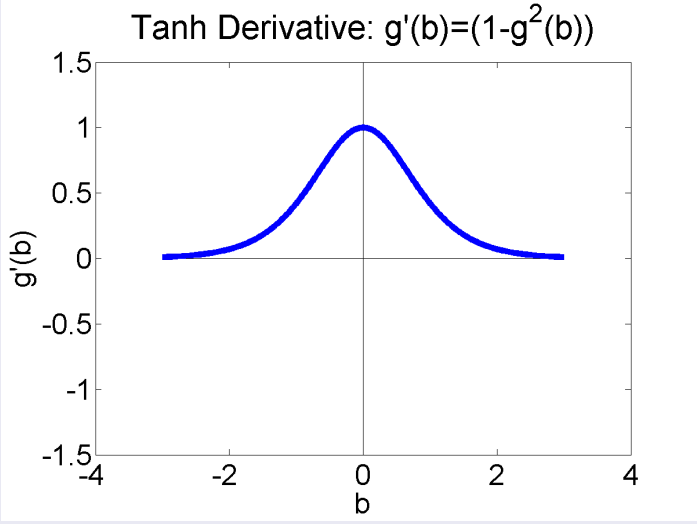
Signum function and its derivative

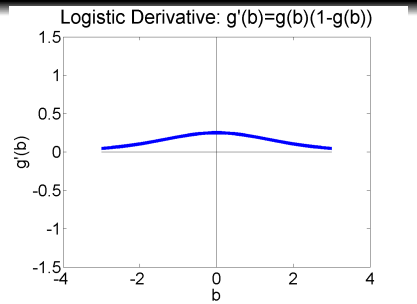


- The derivative of the signum function is $2\delta(z)$, which is not very useful in gradient descent.

Tanh function and its derivative



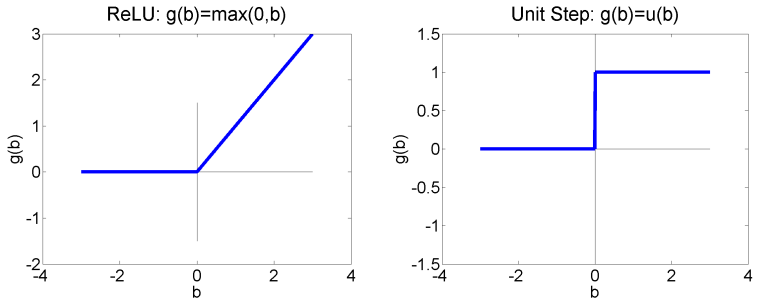




The sigmoid has a surprising problem: for large values of w, $\frac{\partial \sigma(wx)}{\partial w} \rightarrow 0$.

- When we begin training, we start with small values of w. $\frac{\partial \sigma(wx)}{\partial w}$ is reasonably large, so gradient descent works well for training.
- If we have lots of training tokens with similar target values, then the gradient ($\frac{\partial \mathcal{L}}{\partial w}$) is similar for all of them. Gradient descent adds up many of these values as $w \leftarrow w - \frac{\partial \mathcal{L}}{\partial w}$. After a few such examples, w gets very large. At that point, $\frac{\partial \sigma(wx)}{\partial w} \rightarrow 0$, 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

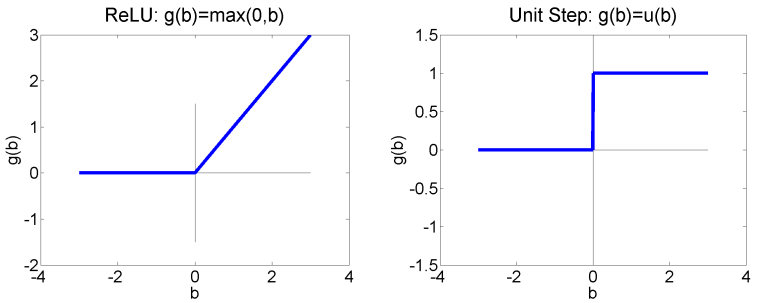


The most ubiquitous solution to the vanishing gradient problem is to use a ReLU (rectified linear unit) instead of a sigmoid. The ReLU is given by

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

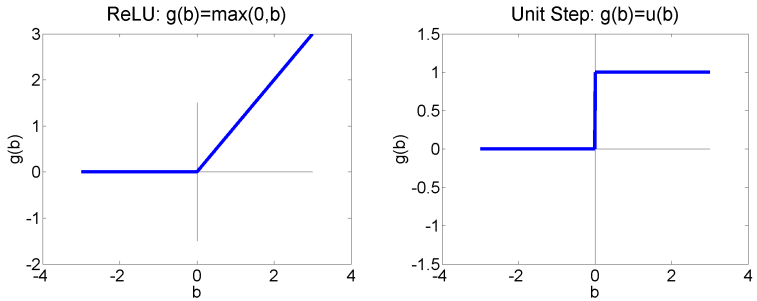
and its derivative is the unit step. Notice that the unit step is equally large ($u(wx) = 1$) for any positive value ($wx > 0$), so no matter how large w gets, back-propagation continues to work.

A solution to the vanishing gradient problem: ReLU



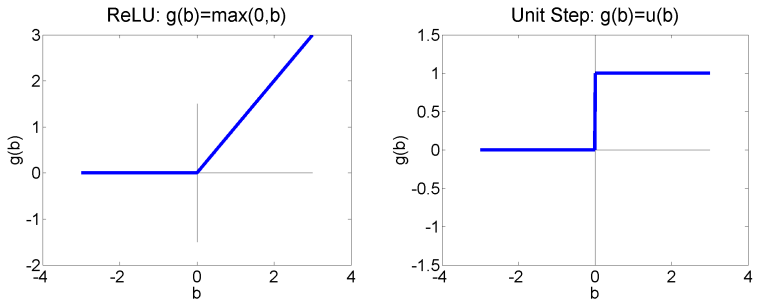
- **Pro:** The ReLU derivative is equally large ($\frac{d\text{ReLU}(wx)}{d(wx)} = 1$) for any positive value ($wx > 0$), so no matter how large w gets, back-propagation continues to work.
- **Con:** If the ReLU is used as a hidden unit ($a_j = \text{ReLU}(z_j)$), then your output is no longer a piece-wise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

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

- **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

1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle $\rightarrow$ Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle $\rightarrow$ Parabola
9. Summary

How to train a neural network

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

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

2. Randomly **initialize** $\mathbf{W}_1$, $\mathbf{b}_1$, $\mathbf{W}_2$, and $\mathbf{b}_2$.
3. Perform **forward propagation**: find out what the neural net computes as $\mathbf{g}(\mathbf{x}_i)$ for each $\mathbf{x}_i$.
4. Define a **loss function** that measures how badly $\mathbf{g}(\mathbf{x}_i)$ differs from $\mathbf{y}_i$.
5. Perform **back propagation** to find the derivatives of the loss w.r.t. $\mathbf{a}_1$, $\mathbf{z}_1$, $\mathbf{a}_2$, and $\mathbf{z}_2$.
6. Calculate the **loss gradients** $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_1}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_1}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2}$, and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}_2}$.
7. Perform **gradient descent** to improve $\mathbf{W}_1$, $\mathbf{b}_1$, $\mathbf{W}_2$, and $\mathbf{b}_2$.
8. Repeat steps 3-6 until convergence.

Loss Functions in General: Probability

- We want $\mathbf{g}(\mathbf{x})$ to give us as much information as possible about $\mathbf{y}$. We formalize that by interpreting $\mathbf{g}(\mathbf{x})$ as some kind of probability model, and by maximizing $\Pr(\mathbf{y}|\mathbf{g}(\mathbf{x}))$.
- If we assume the training examples are independent, then it makes sense to multiply their probabilities.

$$\Pr(\mathcal{D}) = \prod_{i=1}^{n} \Pr(\mathbf{y}_i|\mathbf{x}_i)$$

Loss Functions in General: Negative Log Probability

- Adding log probabilities is much easier than multiplying probabilities.

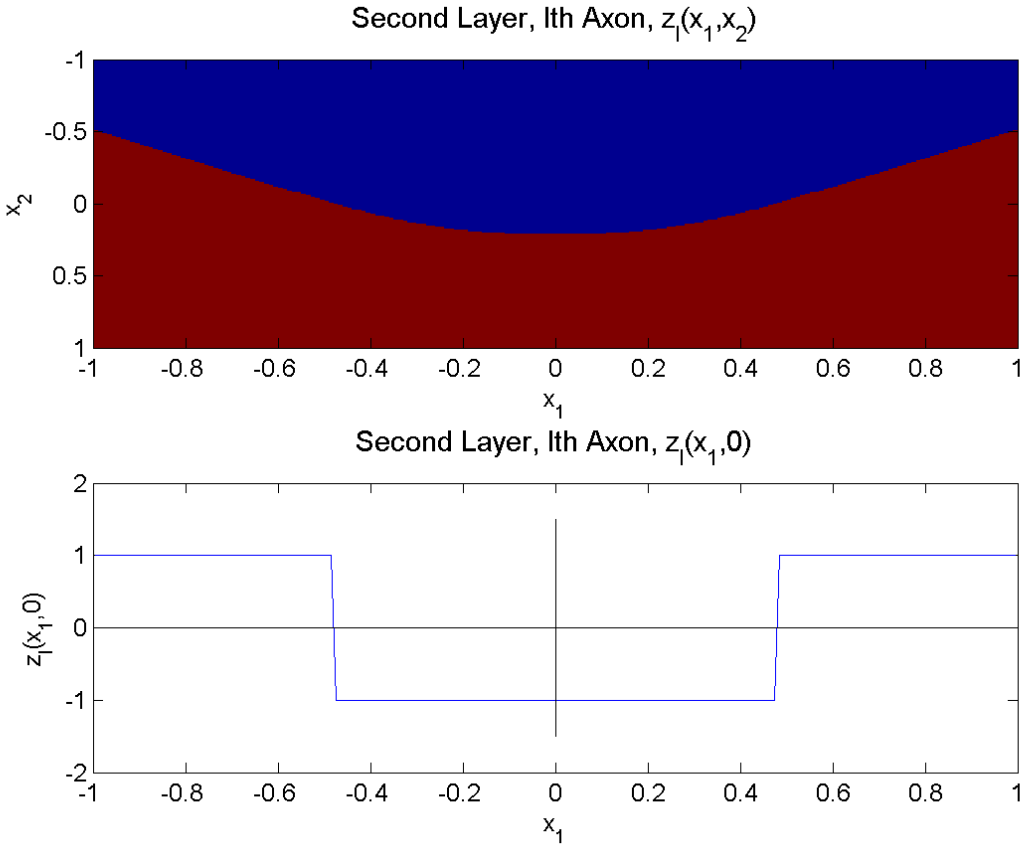
$$\Pr(\mathcal{D}) = \prod_{i=1}^{n} \Pr(\mathbf{y}_i|\mathbf{x}_i)$$

$$\ln \Pr(\mathcal{D}) = \sum_{i=1}^{n} \ln \Pr(\mathbf{y}_i|\mathbf{x}_i)$$

- "Minimizing the loss" is the same thing as "maximizing the log probability" if we set the "loss" ($\mathcal{L}$) equal to the average negative log probability. We will often use $E[\cdot]$ to mean "average value:"

$$\mathcal{L} = E[-\ln \Pr(\mathbf{y}|\mathbf{x})] = -\frac{1}{n} \sum_{i=1}^{n} \Pr(\mathbf{y}_i|\mathbf{x}_i)$$

Loss Function: Binary Classifier



For a binary classifier, the target output is binary, $y \in \{0, 1\}$. Suppose we interpret this as the observed value of a Bernoulli random variable, Y.

Loss Function: Binary Classifier

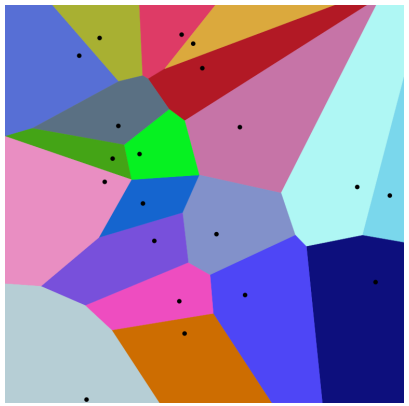
Suppose we use a sigmoid output nonlinearity, so that $0 < g(\mathbf{x}) < 1$, then we can interpret $g(\mathbf{x})$ as the probability that Y should be one:

$$g(\mathbf{x}) = \Pr(Y = 1|\mathbf{x})$$

Under this interpretation, the loss function is

$$\begin{aligned}\mathcal{L} &= E\left[-\ln \Pr(Y = y|\mathbf{x})\right] \\ &= -E\left[y \ln \Pr(Y = 1|\mathbf{x}|) + (1 - y) \ln \Pr(Y = 0|\mathbf{x})\right] \\ &= -E\left[y \ln g(\mathbf{x}) + (1 - y) \ln(1 - g(\mathbf{x}))\right]\end{aligned}$$

Loss Function: Multinomial Classifier

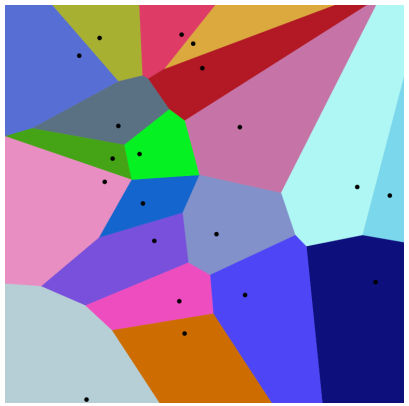


https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg

The best way to formulate an m-class classifier is by having an m-element **one-hot** output vector, $\mathbf{y} \in \{0,1\}^m$, where

$$y_k = \begin{cases} 1 & k \text{ is the correct class} \\ 0 & \text{otherwise} \end{cases}$$

Loss Function: Multinomial Classifier



Now we want the neural net output, $\mathbf{g}(\mathbf{x}) = [g_1(\mathbf{x}), g_2(\mathbf{x}), \ldots]^T$, to be interpretable as a set of probabilities, $g_k(\mathbf{x}) = \Pr(Y_k = 1|\mathbf{x})$. In order to be interpreted as probabilities, the outputs need to satisfy: $0 \leq g_k(\mathbf{x}) \leq 1$ and $\sum_k g_k(\mathbf{x}) = 1$. An output nonlinearity that satisfies these conditions is the **softmax**, defined as

$$g_k(\mathbf{x}) = \underset{k}{\text{softmax}}(\mathbf{z}_2) = \frac{e^{z_{2,k}}}{\sum_{k'} e^{z_{2,k'}}}$$

Loss Function: Multinomial Classifier

We can interpret $g_k(\mathbf{x}) = \Pr(Y_k = 1|\mathbf{x})$ if

$$Y_k = \begin{cases} 1 & k \text{ is the correct class} \\ 0 & \text{otherwise} \end{cases}, \qquad g_k(\mathbf{x}) = \frac{e^{z_{2,k}}}{\sum_{k'} e^{z_{2,k'}}}$$

With these definitions, the loss function is the **cross-entropy loss**:

$$\begin{aligned} \mathcal{L} &= E\left[-\ln \Pr(\mathbf{y}|\mathbf{x})\right] \\ &= -E\left[\sum_k y_k \ln g_k(\mathbf{x})\right] \\ &= -\frac{1}{n}\sum_{i=1}^{n} \ln g_{\text{correct class}}(\mathbf{x}_i) \end{aligned}$$

Loss Function: Regression

For a nonlinear regression neural net, the target function $\mathbf{y}$ is an arbitrary real vector. In this case, a useful probability model assumes that

$$\mathbf{y} = \mathbf{g}(\mathbf{x}) + \mathbf{v},$$

where $\mathbf{v}$ is zero-mean, unit variance Gaussian noise:

$$\Pr(\mathbf{y}|\mathbf{g}(\mathbf{x})) = e^{-\frac{1}{2}\|\mathbf{y}-\mathbf{g}(\mathbf{x})\|^2}$$

The loss function is then just the mean squared error:

$$\begin{aligned}\mathcal{L} &= E\left[-\ln \Pr(\mathbf{y}|\mathbf{x})\right] \\ &= E\left[\frac{1}{2}\|\mathbf{y}-\mathbf{g}(\mathbf{x})\|^2\right] \\ &= -\frac{1}{2n}\sum_{i=1}^{n}\|\mathbf{y}_i-\mathbf{g}(\mathbf{x}_i)\|^2\end{aligned}$$

Loss Function: Regression

Minimum Mean Squared Error (MMSE)⇒ $\mathbf{g}(\mathbf{x})$ = conditional expected value of $\mathbf{y}$

$$\mathbf{g}^*(\mathbf{x}) = \text{argmin}\, \mathcal{L} = \text{argmin}\, \frac{1}{2} E\left[\|\mathbf{y} - \mathbf{g}(\mathbf{x})\|^2\right]$$

which is minimized by

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

Outline

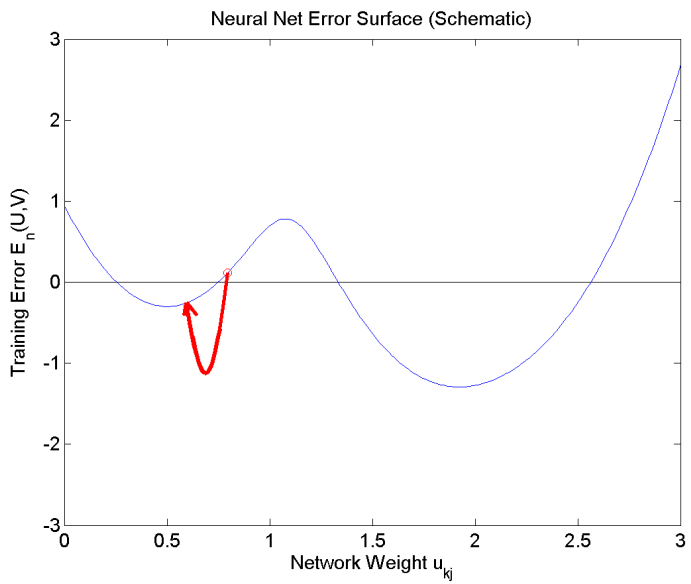
1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle $\rightarrow$ Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle $\rightarrow$ Parabola
9. Summary

Gradient Descent: How do we improve **W** and **b**?

Given some initial neural net parameter we want to find a better value of the same parameter. We do that using gradient descent:

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}} \right)^T,$$

where η is a learning rate (some small constant, e.g., $\eta = 0.001$ or so).



Loss Gradient

Suppose we use MSE loss:

$$\mathcal{L} = E\left[\|\mathbf{y} - \mathbf{g}(\mathbf{x})\|^2\right] = E\left[\|\mathbf{y} - \mathbf{a}_2(\mathbf{x})\|^2\right],$$

where the expectation means "average over the training data." Since $\mathcal{L}$ is a scalar, it should be possible to compute $\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l}$, but how? For example, can we use the chain rule? Let's try:

$$\mathbf{a}_2 = f_2(\mathbf{z}_2), \quad \mathbf{z}_2 = \mathbf{b}_2 + \mathbf{W}_2\mathbf{a}_1$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_2} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2}\right) \times \left(\frac{\partial \mathbf{a}_2}{\partial \mathbf{z}_2}\right) \times \ldots \mathbf{?}$$

We'd like to put $\partial \mathbf{z}_2 / \mathbf{W}_2$ here, but the derivative of a vector w.r.t. a matrix is not well-defined. We need write it out in smaller pieces, to make sure we get it right.

Loss Gradient using the Chain Rule

If we write out $\mathbf{z}_l = \mathbf{b}_l + \mathbf{W}_l \mathbf{a}_{l-1}$ in terms of its scalar components, it is:

$$z_{l,k} = b_{l,k} + \sum_j w_{l,k,j} a_{l-1,j}$$

The chain rule gives us

$$\frac{\partial \mathcal{L}}{\partial w_{l,k,j}} = E\left[\left(\frac{\partial \mathcal{L}}{\partial z_{l,k}}\right)\left(\frac{\partial z_{l,k}}{\partial w_{l,k,j}}\right)\right] = E\left[\left(\frac{\partial \mathcal{L}}{\partial z_{l,k}}\right) a_{l-1,j}\right],$$

where, again, the expectation means "average over the training samples." Now, remember that the gradient matrix is the matrix whose $(j,k)^{\text{th}}$ element is $\frac{\partial \mathcal{L}}{\partial w_{l,k,j}}$, thus:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = E\left[\mathbf{a}_{l-1}\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}\right],$$

- $\mathbf{a}_{l-1} = [a_{l-1,1}, a_{l-1,2}, \ldots, a_{l-1,m_{l-1}}]^T$ is a column vector,
- $\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} = [\frac{\partial \mathcal{L}}{\partial z_{l,1}}, \ldots, \frac{\partial \mathcal{L}}{\partial z_{l,m_l}}]$ is a row vector

How to train a neural network

1. Find a **training dataset**.
2. Randomly **initialize** $\mathbf{W}_l$ and $\mathbf{b}_l$.
3. Perform **forward propagation**: $\mathbf{z}_l = \mathbf{b}_l + \mathbf{W}_l \mathbf{a}_{l-1}$.
4. Define a **loss function**: $\mathcal{L} = E\left[-\ln \Pr(\mathbf{y}|\mathbf{x})\right]$ in general, of which one example is the MSE loss $\mathcal{L} = E\left[\frac{1}{2}\|\mathbf{y} - \mathbf{g}(\mathbf{x})\|^2\right]$.
5. Perform **back propagation**.
6. Calculate the **loss gradients**,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = E\left[\mathbf{a}_{l-1}\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}\right], \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}_l} = E\left[\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}\right]$$

7. Perform **gradient descent**:

$$\mathbf{W}_l \leftarrow \mathbf{W}_l - \eta\left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l}\right)^T$$

$$\mathbf{b}_l \leftarrow \mathbf{b}_l - \eta\left(\frac{\partial \mathcal{L}}{\partial \mathbf{b}_l}\right)^T$$

8. Repeat steps 3-6 until convergence.

Outline

1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle $\rightarrow$ Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle $\rightarrow$ Parabola
9. Summary

Back-Propagation

Now we know that

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = E\left[\mathbf{a}_{l-1}\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}\right]$$

How do we find $\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}$? Answer: use the chain rule! Since $\mathbf{a}_l$ and $\mathbf{z}_l$ are vectors, their Jacobian matrices are well defined, so

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}_l}\right) \times \left(\frac{\partial \mathbf{a}_l}{\partial \mathbf{z}_l}\right)$$

and

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_l} = \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}_{l+1}}\right) \times \left(\frac{\partial \mathbf{z}_{l+1}}{\partial \mathbf{a}_l}\right)$$

Derivative of the MSE Loss

The MSE loss is

$$\begin{aligned}\mathcal{L} &= E\left[\frac{1}{2}\|\mathbf{y}-\mathbf{g}(\mathbf{x})\|^2\right] \\ &= E\left[\frac{1}{2}\|\mathbf{y}-\mathbf{a}_2\|^2\right] \\ &= \frac{1}{2n}\sum_{i=1}^{n}\left[\|\mathbf{y}_i-\mathbf{a}_2(\mathbf{x}_i)\|^2\right]\end{aligned}$$

So its derivative is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}_2} = -\frac{1}{n}\left(\mathbf{y}-\mathbf{a}_2\right)^T$$

Back-Prop Through a Scalar Nonlinearity

All of the nonlinearities we know **except softmax** are scalar nonlinearities, and can be written as

$$\begin{bmatrix} a_{l,1} \\ \vdots \\ a_{l,m_l} \end{bmatrix} = \begin{bmatrix} f_l(z_{l,1}) \\ \vdots \\ f_l(z_{l,m_l}) \end{bmatrix}$$

The Jacobian of this transformation is a diagonal matrix whose diagonal elements are the derivatives of $f_l(\cdot)$ with respect to each of the elements of $\mathbf{z}_l$. Let's call this matrix $f_l'(\mathbf{z}_l)$:

$$\frac{\partial \mathbf{a}_l}{\partial \mathbf{z}_l} = f_l'(\mathbf{z}_l) \equiv \begin{bmatrix} \frac{\partial f_l(z_{l,1})}{\partial z_{l,1}} & 0 & \cdots & 0 \\ 0 & \frac{\partial f_l(z_{l,2})}{\partial z_{l,2}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{\partial f_l(z_{l,m_l})}{\partial z_{l,m_l}} \end{bmatrix}$$

Back-Prop Through a Linear Layer

A linear layer is

$$\mathbf{z}_l = \mathbf{b}_l + \mathbf{W}_l \mathbf{a}_{l-1}$$

Its Jacobian is

$$\frac{\partial \mathbf{z}_l}{\partial \mathbf{a}_{l-1}} = \mathbf{W}_l$$

Back-Prop in a Two-Layer Neural Net

Putting it all together, for an MSE loss,

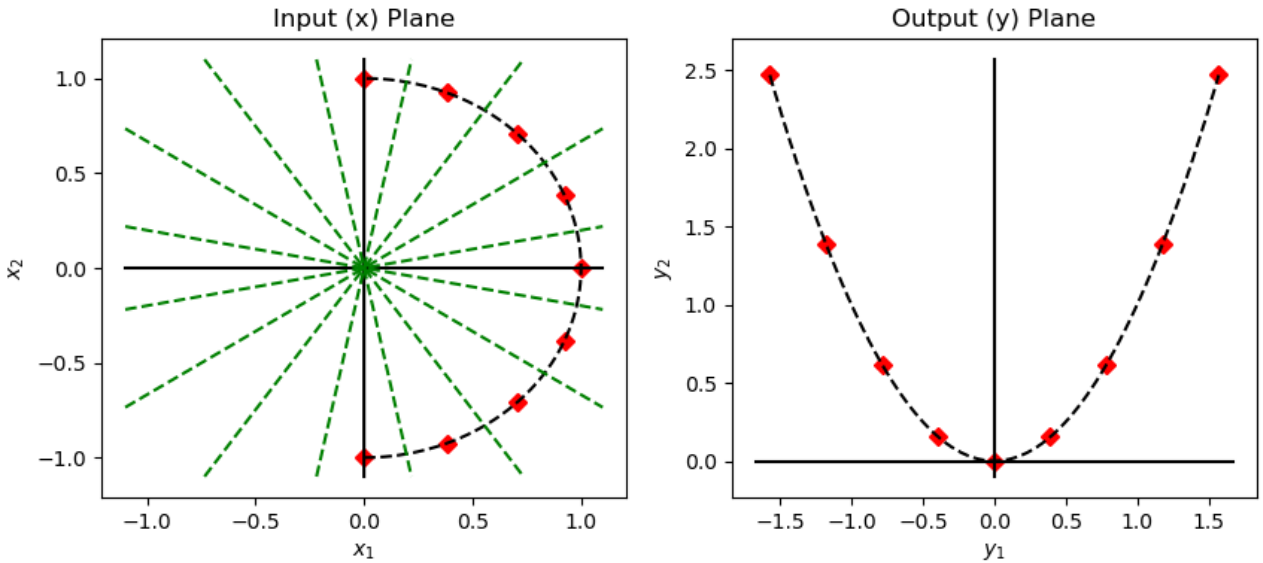
$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_2} = -\frac{1}{n}(\mathbf{y} - \mathbf{a}_2)^T f_2'(\mathbf{z}_2)$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}_1} = -\frac{1}{n}(\mathbf{y} - \mathbf{a}_2)^T f_2'(\mathbf{z}_2)\mathbf{W}_2 f_1'(\mathbf{z}_1)$$

Outline

1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle $\rightarrow$ Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle $\rightarrow$ Parabola
9. Summary

Backprop Example: Semicircle $\rightarrow$ Parabola

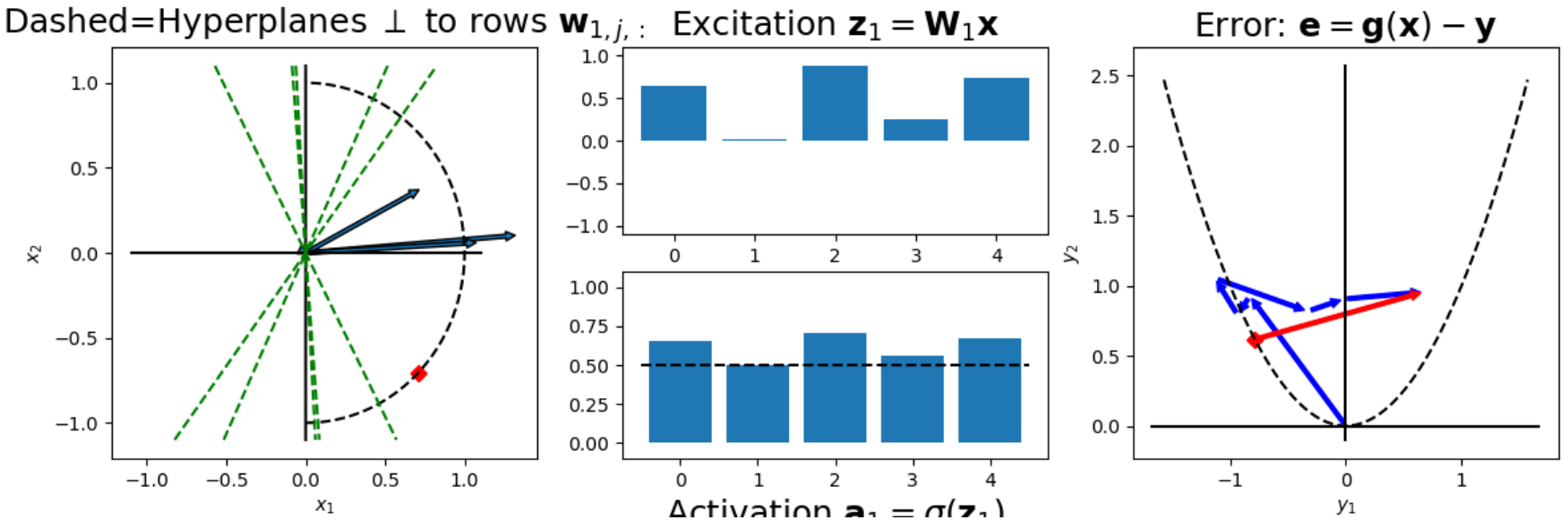


Remember, we are going to try to approximate this using:

$$\mathbf{g}(\mathbf{x}) = \mathbf{b}_2 + \mathbf{W}_2\sigma\left(\mathbf{W}_1\mathbf{x}\right)$$

Randomly Initialized Weights

Here's what we get if we randomly initialize $\mathbf{W}_1$, $\mathbf{b}_2$, and $\mathbf{W}_2$. The red vector on the right is the estimation error for this training token, $\mathbf{e} = \mathbf{g}(\mathbf{x}) - \mathbf{y}$. It's huge!



Gradient Updates

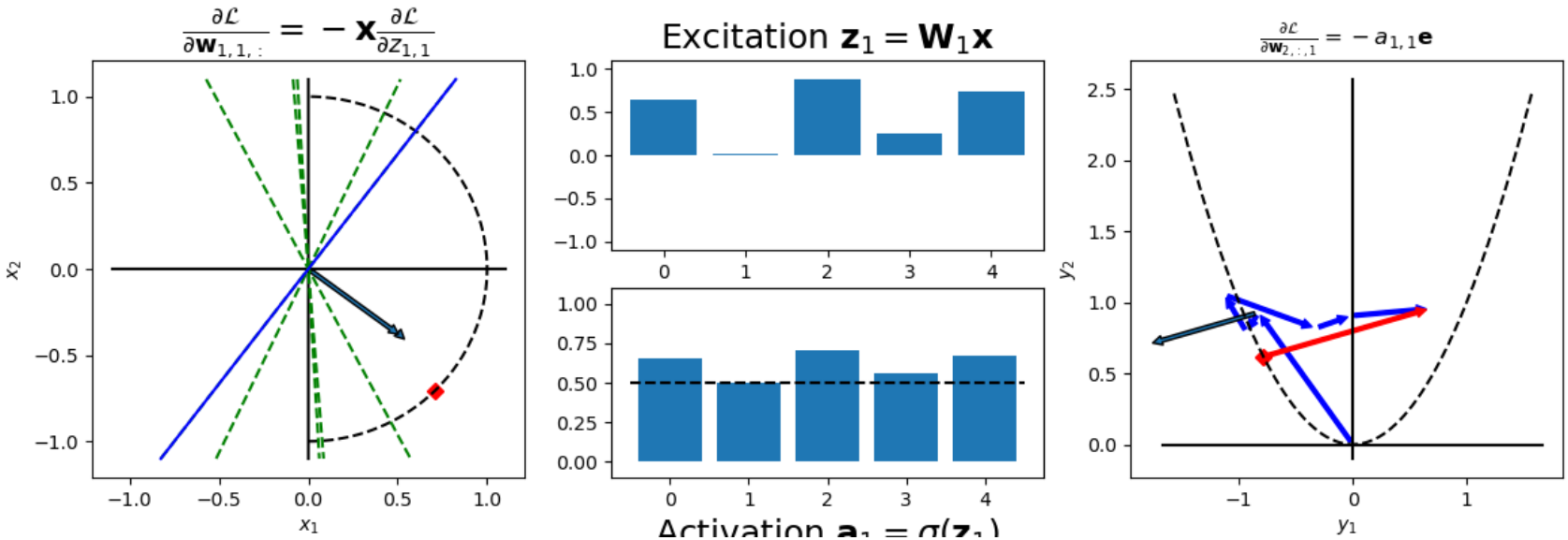
Remember that the output of this network is a series of column vectors, $\mathbf{w}_{2,:,j}$, each of which is multiplied by $a_{1,j}$ and added to the output. Gradient descent updates $\mathbf{w}_{2,:,j}$ by scaling $(\mathbf{y} - \mathbf{g}(\mathbf{x}))$ times the activation $a_{1,j}$, and subtracting:

$$\begin{aligned}\mathbf{w}_{2,:,j} &\leftarrow \mathbf{w}_{2,:,j} - \eta a_{1,j} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{z}_2}\right)^T \\ &= \mathbf{w}_{2,:,j} - \eta a_{1,j}(\mathbf{y} - \mathbf{g}(\mathbf{x}))\end{aligned}$$

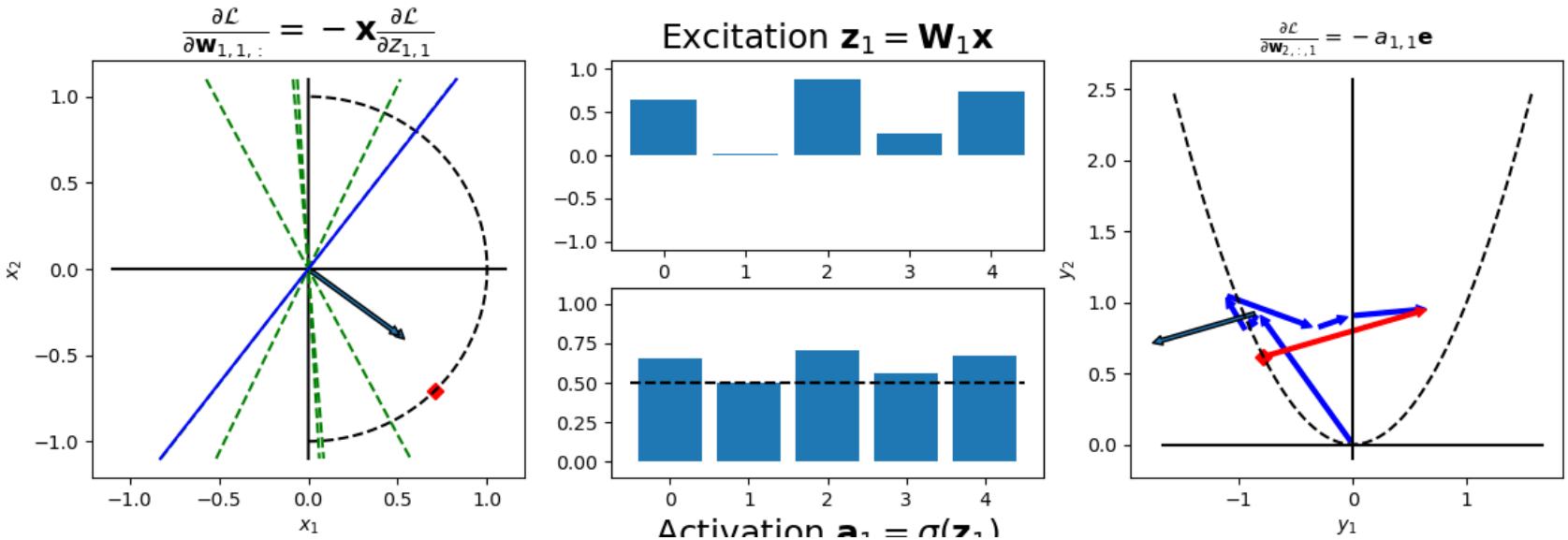
Similarly, remember that the input of this network finds the dot product between $\mathbf{x}$ and $\mathbf{w}_{1,j,:}$, and compares it to a threshold. The corresponding learning rule subtracts $\frac{\partial \mathcal{L}}{\partial z_{1,j}}\mathbf{x}^T$ from each row:

$$\mathbf{w}_{1,j,:} \leftarrow \mathbf{w}_{1,j,:} - \eta \frac{\partial \mathcal{L}}{\partial z_{1,j}}\mathbf{x}^T$$

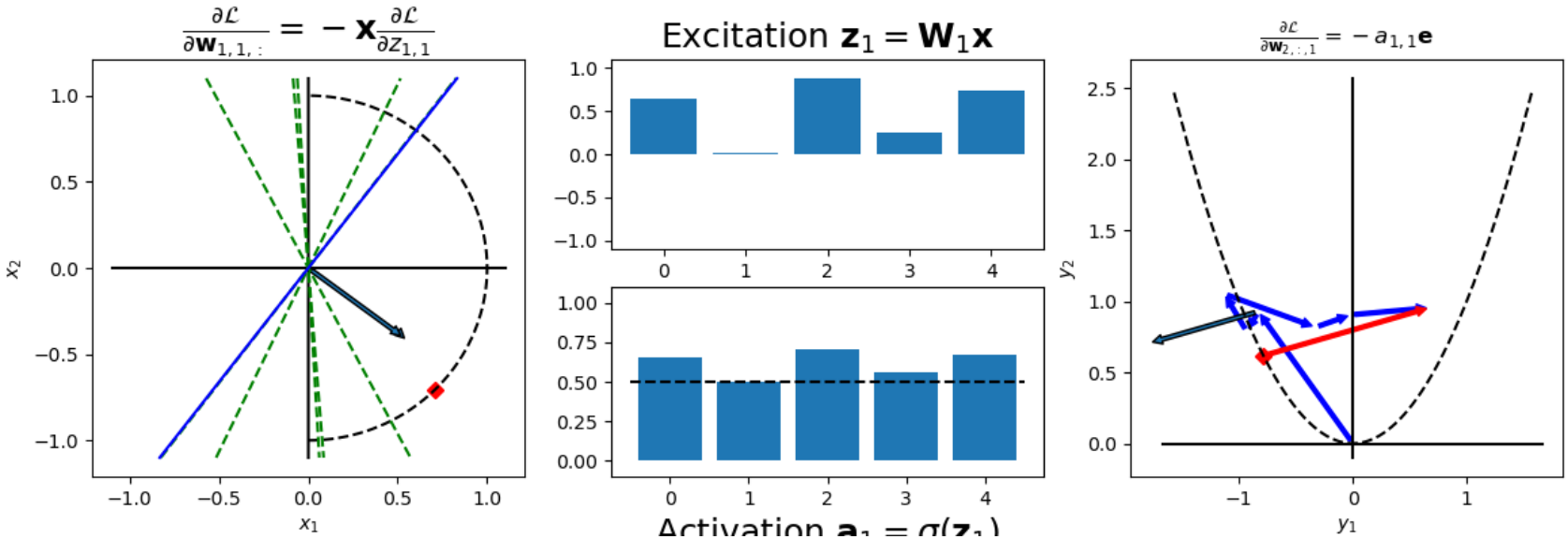
Gradient Updates



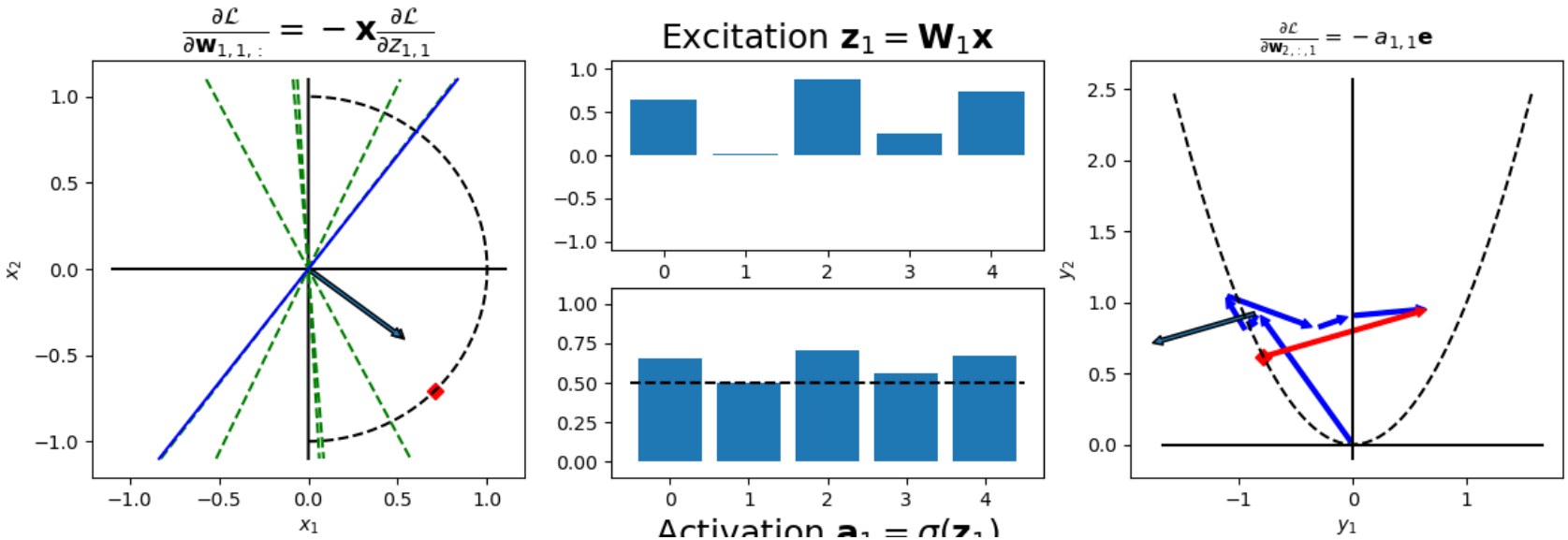
Gradient Updates



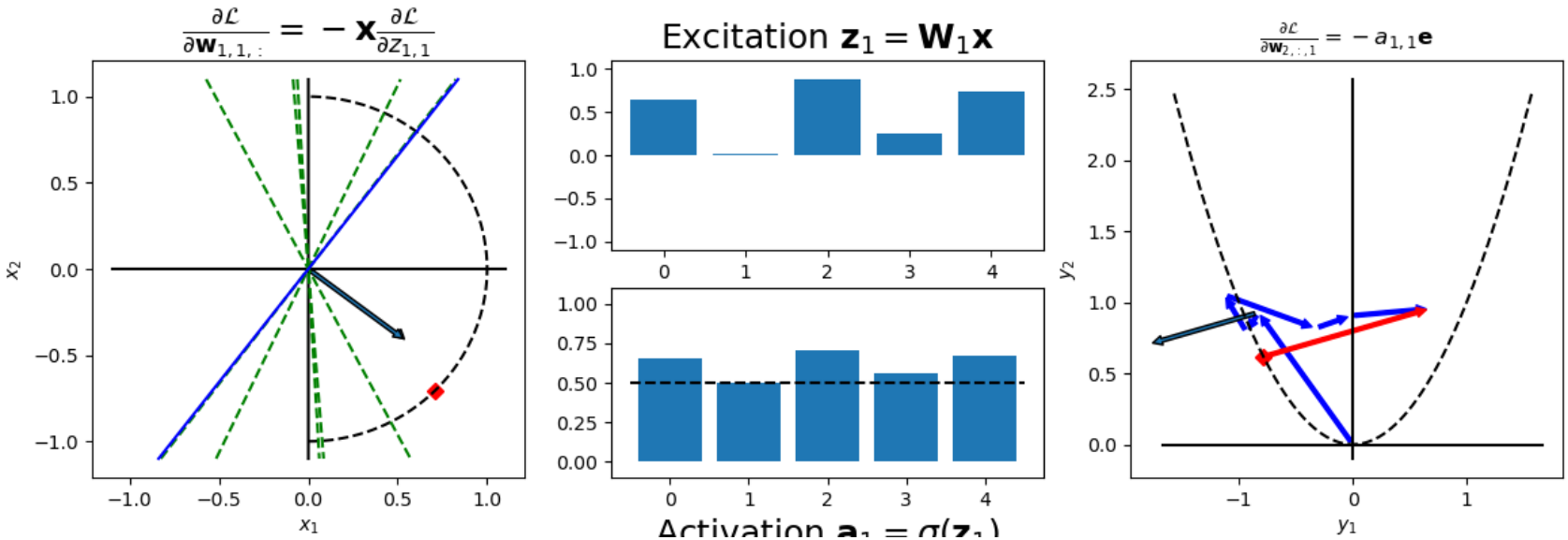
Gradient Updates



Gradient Updates



Gradient Updates



Outline

1. Intro
2. Classification Example: Arbitrary Classifier
3. Regression Example: Semicircle $\rightarrow$ Parabola
4. Scalar Nonlinearities
5. Loss Functions
6. Learning: Gradient Descent
7. Back-Propagation
8. Backprop Example: Semicircle $\rightarrow$ Parabola
9. Summary

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.

Error Metrics Summarized

- 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 hidden layer excitation vectors.
- From the back-propagated gradient vectors, we calculate the weight gradient matrix as

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = E\left[\mathbf{a}_{l-1}\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}\right]$$

- For a **regression** problem, use MSE to achieve $\mathbf{g}(\mathbf{x}) \rightarrow E[\mathbf{y}|\mathbf{x}]$.
- For a **binary classifier** with a sigmoid output, BCE loss gives you the MSE result without the vanishing gradient problem.
- For a **multi-class classifier** with a softmax output, CE loss gives you the MSE result without the vanishing gradient problem.