Lecture 8: Nonlinearities

Mark Hasegawa-Johnson

ECE 417: Multimedia Signal Processing, Fall 2020

Outline

Review: 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.

Review: 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.

Review: 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:

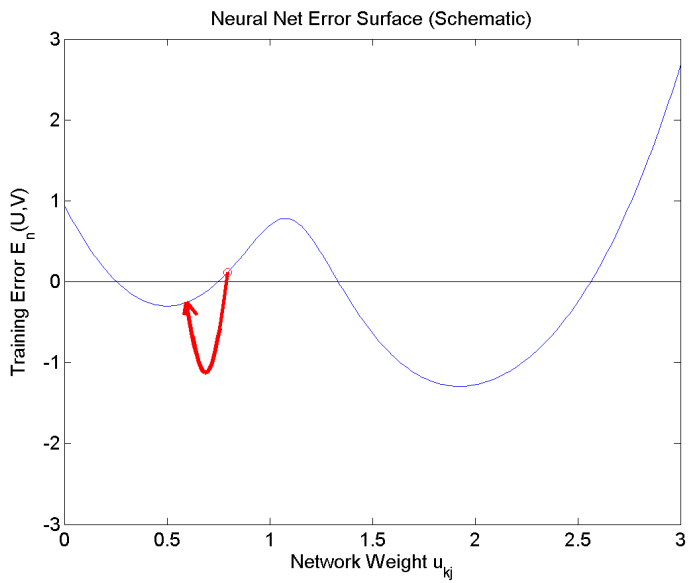
$$h_k = \sigma\left(\bar{w}_k^{(1)}\vec{x}\right) \approx \begin{cases} 1 & \bar{w}_k^{(1)}\vec{x} > 0 \\ 0 & \bar{w}_k^{(1)}\vec{x} < 0 \end{cases}$$

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:

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

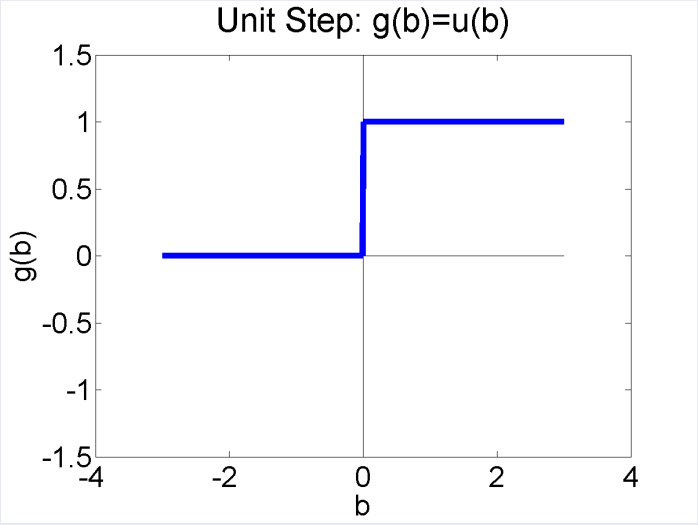
where η is a learning rate (some small constant, e.g., $\eta = 0.02$ or so).



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.

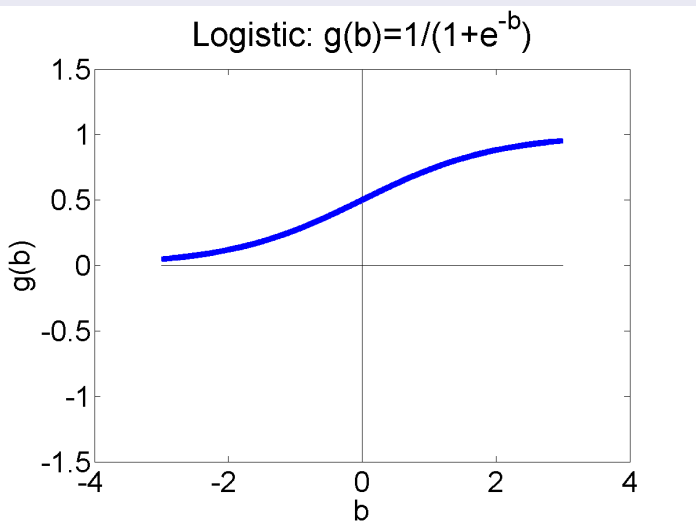
Remember back-prop:

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

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

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

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

Derivative of a sigmoid

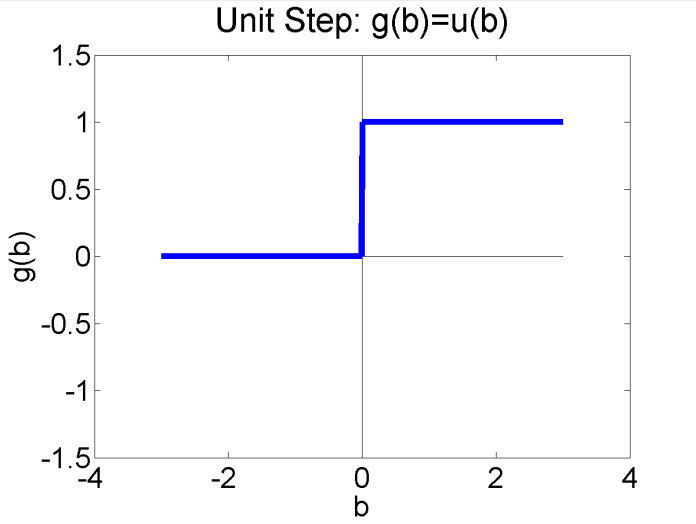
The derivative of a sigmoid is pretty easy to calculate:

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

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:

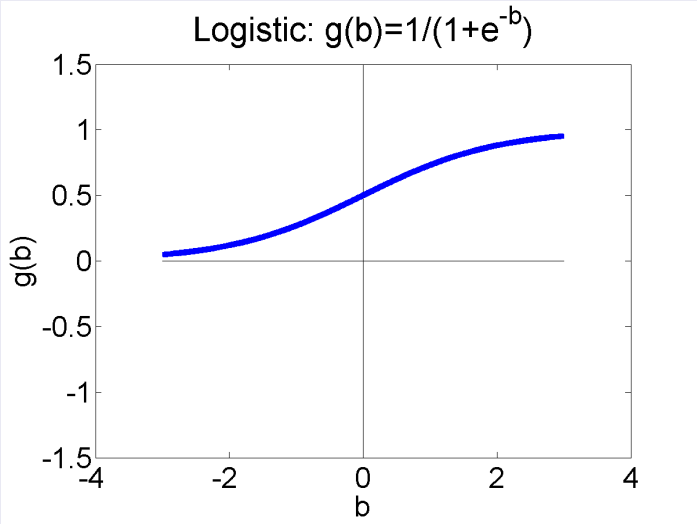
$$\begin{aligned}
\frac{d\sigma}{dx} &= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \left(\frac{1}{1+e^{-x}}\right)\left(\frac{e^{-x}}{1+e^{-x}}\right) \\
&= \left(\frac{1}{1+e^{-x}}\right)\left(1 - \frac{1}{1+e^{-x}}\right) \\
&= \sigma(x)(1-\sigma(x)) \\
&= h(1-h)
\end{aligned}$$

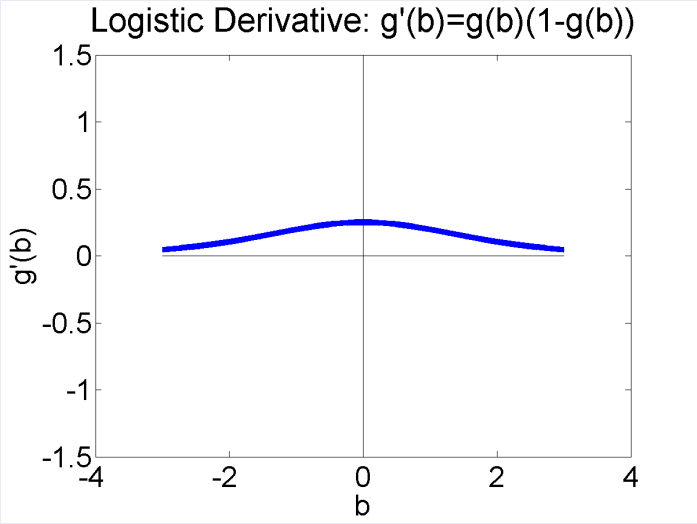
Step function and its derivative



- 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 $h \in \{0, 1\}$:

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

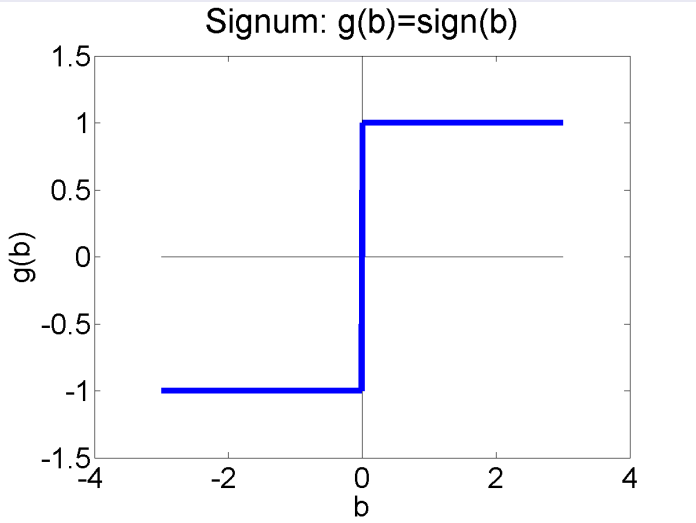
It is usually approximated by the hyperbolic tangent function (tanh), which is just a scaled shifted version of the sigmoid:

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

and which has a scaled version of the sigmoid derivative:

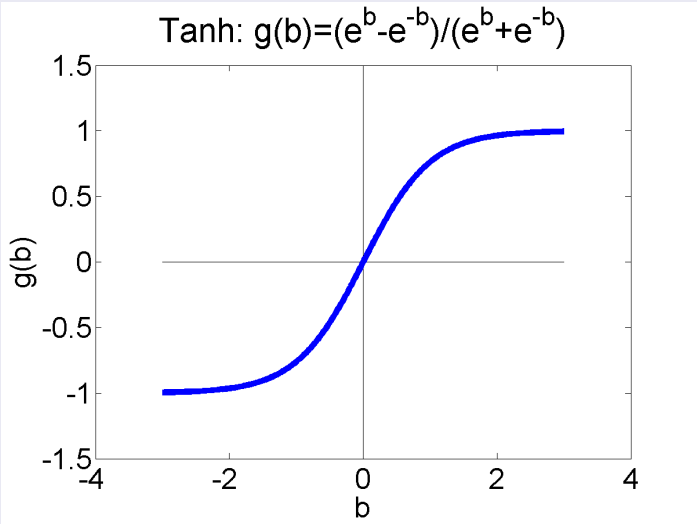
$$\frac{d \tanh(b)}{db} = (1 - \tanh^2(b))$$

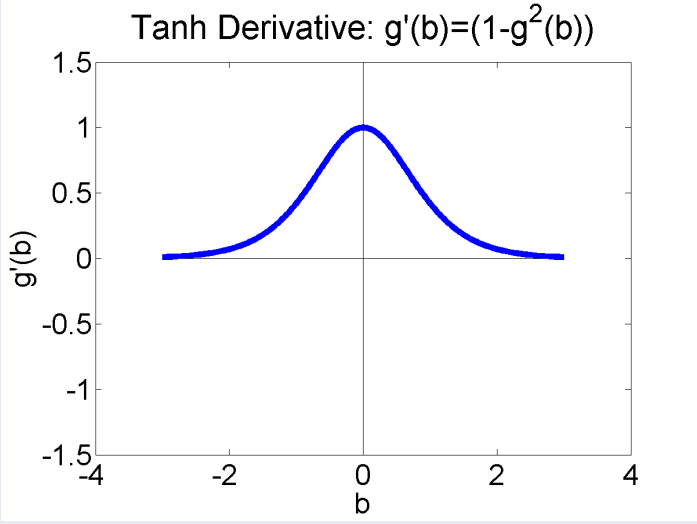
Signum function and its derivative



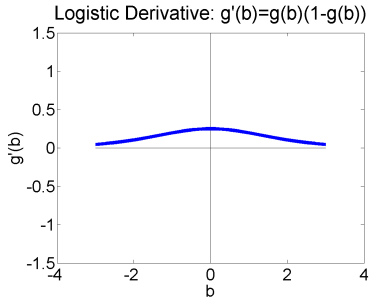
- The derivative of the signum function is the Dirac delta, which is not very useful in backprop.

Tanh function and its derivative





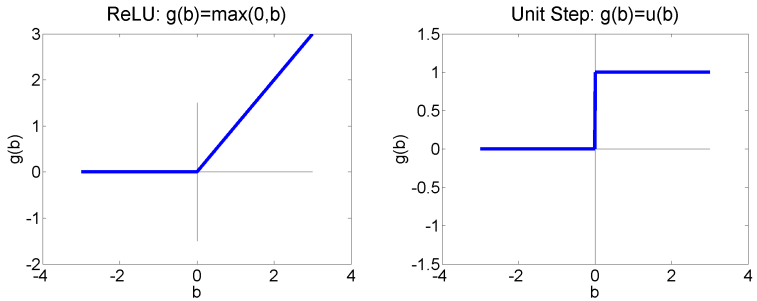
A suprising problem with the sigmoid: Vanishing gradients



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

- 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 $w \to w - \eta \nabla_w \mathcal{L}$ makes w larger. After a few iterations, w gets very large. At that point, $\sigma'(wx) \to 0$, 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

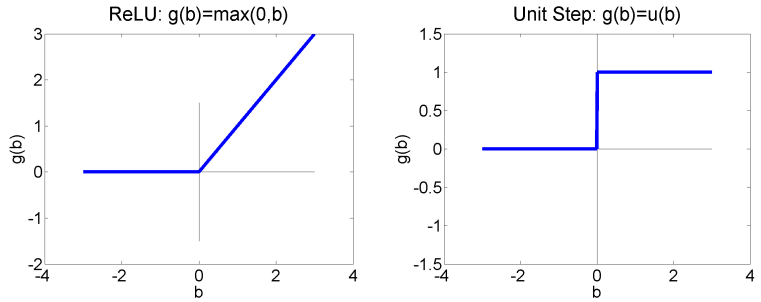


The most ubiquitous solution to the vanishing gradient problem is to use a ReLU (rectified linear unit) instead of a sigmoid. The ReLU is given by

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

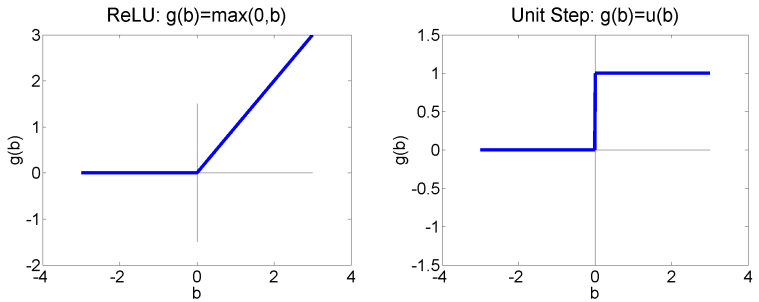
and its derivative is the unit step. Notice that the unit step is equally large ($u(wx) = 1$) for any positive value ($wx > 0$), so no matter how large w gets, back-propagation continues to work.

A solution to the vanishing gradient problem: ReLU



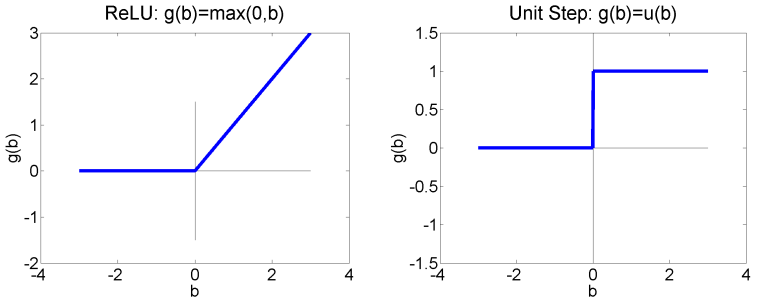
- **Pro:** The ReLU derivative is equally large ($\frac{d\text{ReLU}(wx)}{d(wx)} = 1$) for any positive value ($wx > 0$), so no matter how large w gets, back-propagation continues to work.
- **Con:** If the ReLU is used as a hidden unit ($h_j = \text{ReLU}(e_j)$), then your output is no longer a piece-wise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

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

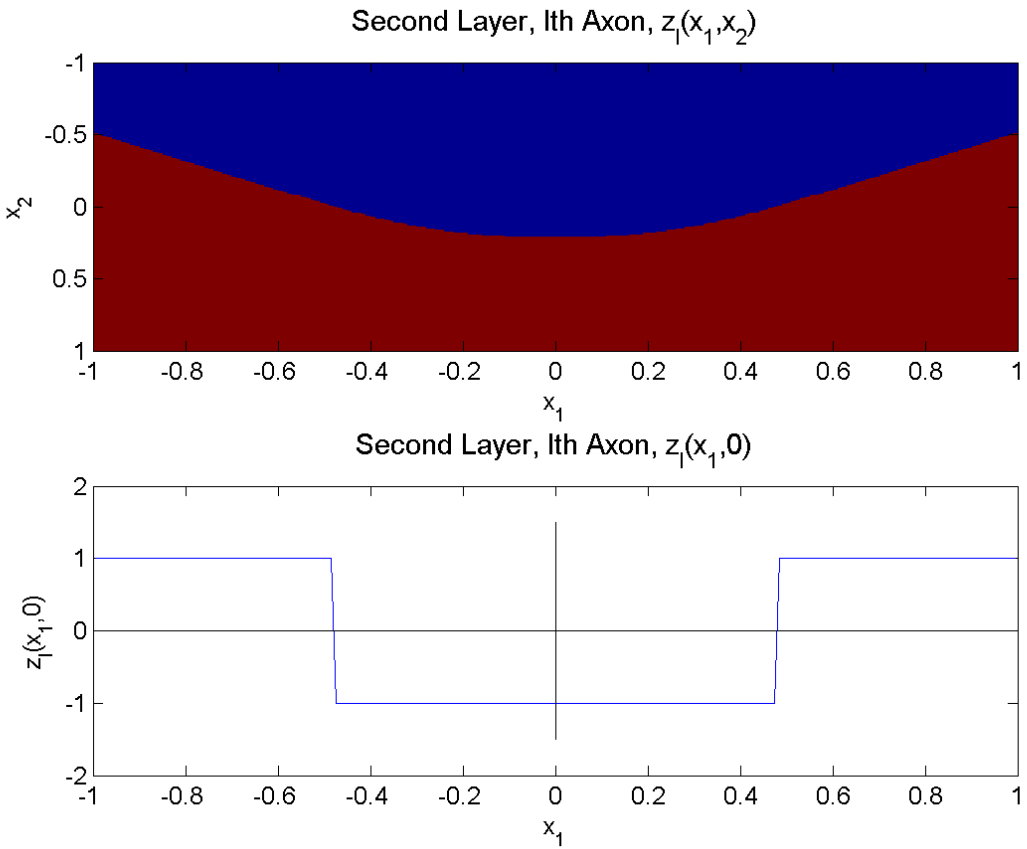
- **Parametric ReLU (PReLU:)** Pro: gradient constant, ouput PWL. The slope of the negative part (a) is a trainable parameter, so can adapt to your dataset. Con: you have to train it.

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

Outline

A classifier target funtion

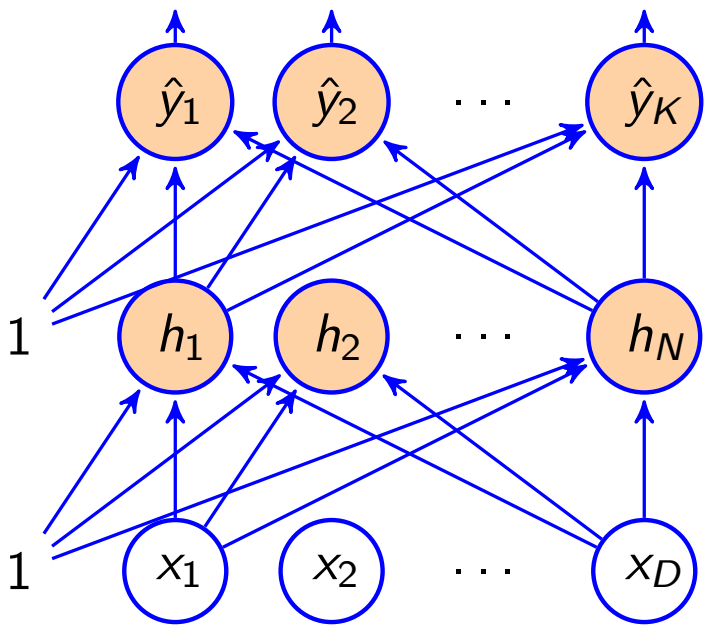
A "classifier" is a neural network with discrete ouputs. For example, suppose you need to color a 2D picture. The goal is to output $\hat{y}(\vec{x}) = 1$ if $\vec{x}$ should be red, and $\hat{y} = -1$ if $\vec{x}$ should be blue:



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 $g(x)$:

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



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

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

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

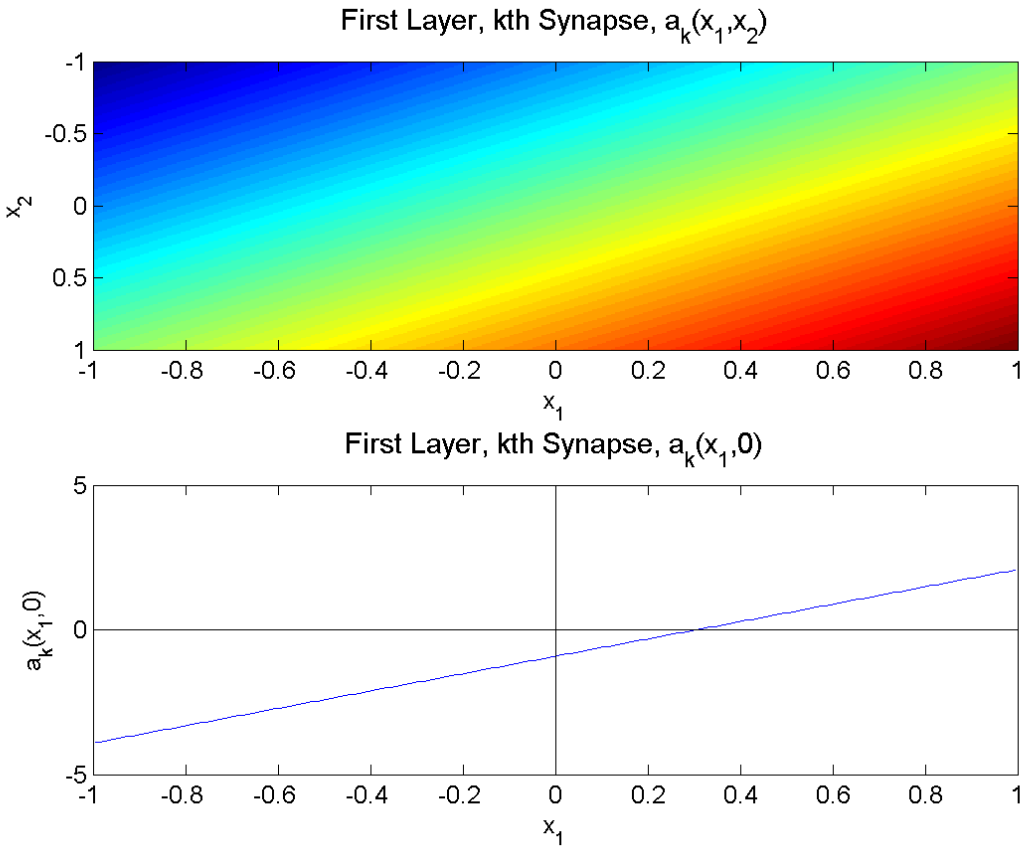
$$e_k^{(1)} = b_k^{(1)} + \sum_{j=1}^{D} w_{kj}^{(1)} x_j$$

$\vec{x}$ is the input vector

Nonlinearities for classifier neural networks

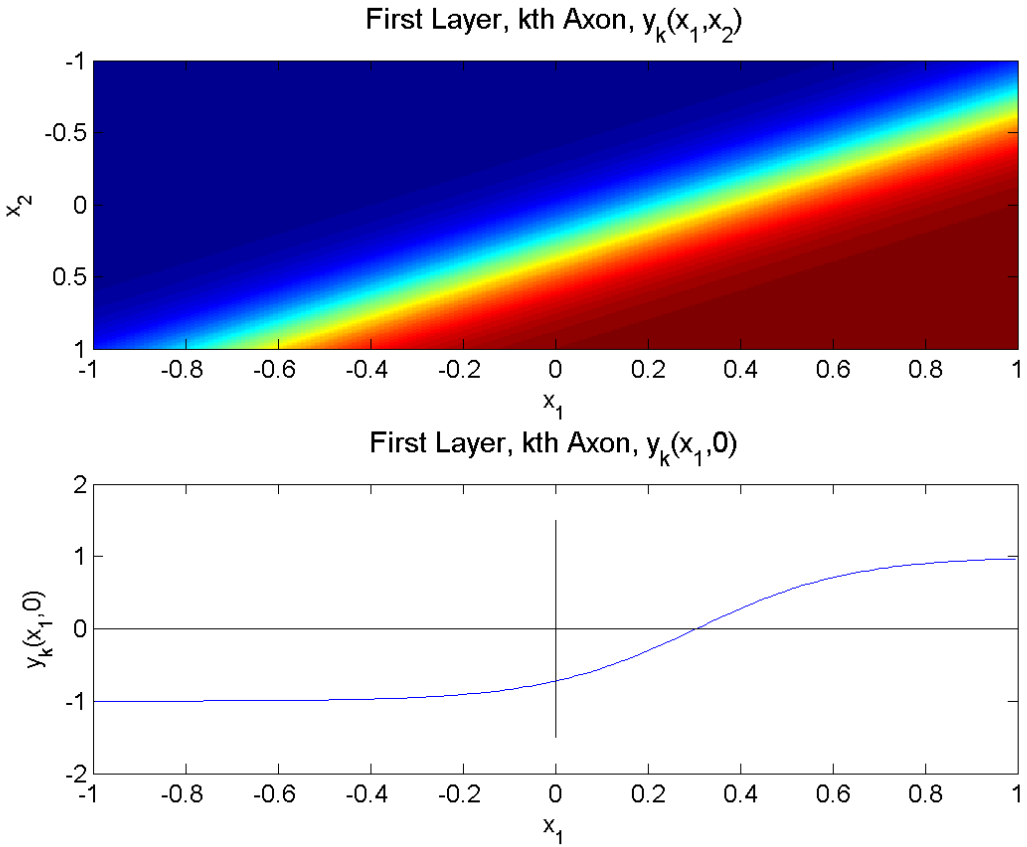
- **During testing:** the output is passed through a hard nonlinearity, e.g., a unit step or a signum.
- **During training:** the output is passed through the corresponding soft nonlinearity, e.g., sigmoid or tanh.

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

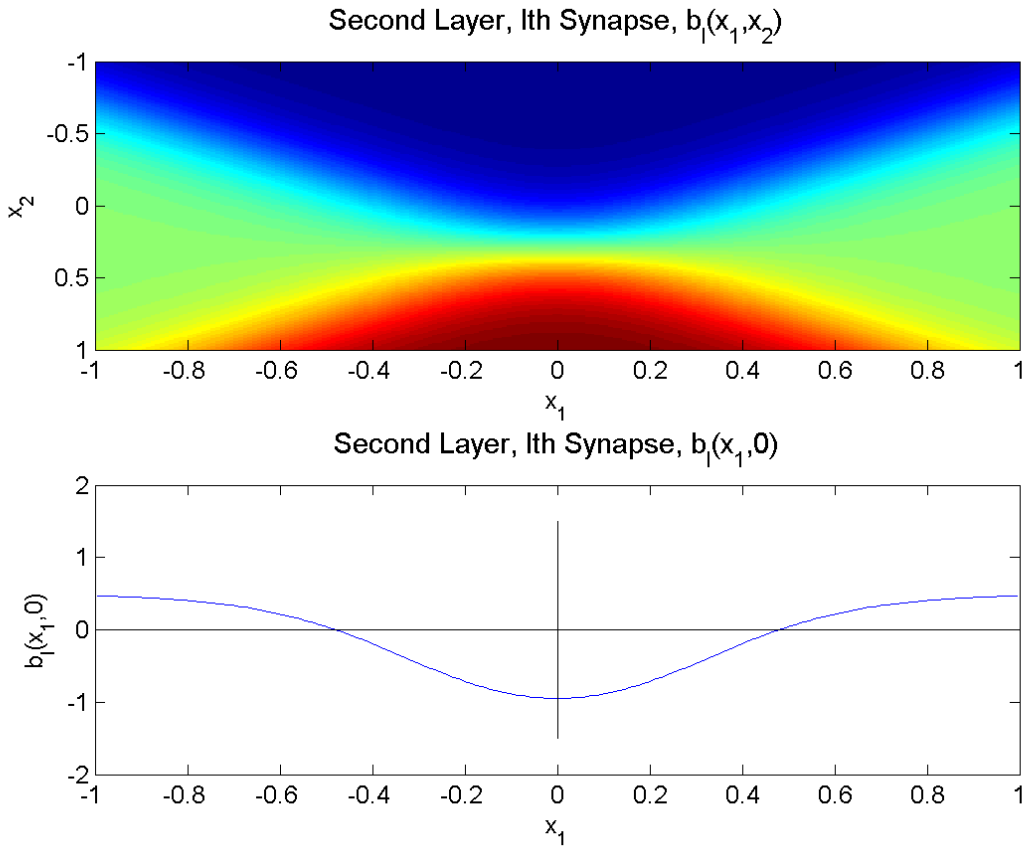


Activation, First Layer: $h_k = \tanh(e_k^{(1)})$

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

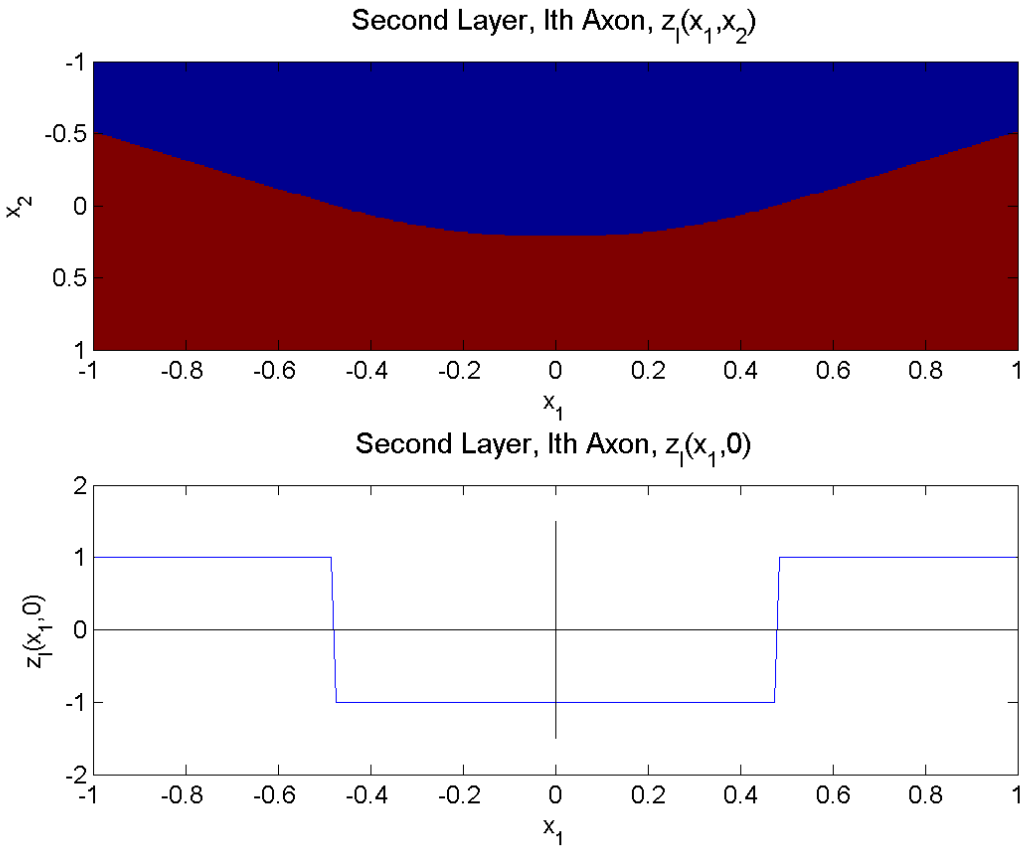


Excitation, Second Layer: $e_k^{(2)} = b_k^{(2)} + \sum_{j=1}^{2} w_{kj}^{(2)} h_j$



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

Review: MSE

Until now, we've assumed that the loss function is MSE:

$$\mathcal{L} = \frac{1}{2n} \sum_{i=1}^{n} \|\vec{y}_i - \hat{y}(\vec{x}_i)\|^2$$

- MSE makes sense if $\vec{y}$ and $\hat{y}$ are both real-valued vectors, and we want to compute $\hat{y}_{MMSE}(\vec{x}) = E[\vec{y}|\vec{x}]$. But what if $\hat{y}$ and $\vec{y}$ are discrete-valued (i.e., classifiers?)
- Surprise: MSE works surprisingly well, even with discrete $\vec{y}$!
- But a different metric, binary cross-entropy (BCE) works slightly better.

MSE with a binary target vector

- Suppose y is just a scalar binary classifier label, $y \in \{0, 1\}$ (for example: "is it a dog or a cat?")
- Suppose that the input vector, $\vec{x}$, is not quite enough information to tell us what y should be. Instead, $\vec{x}$ only tells us the probability of $y = 1$:

$$y = \begin{cases} 1 & \text{with probability } p_{Y|\vec{X}}(1|\vec{x}) \\ 0 & \text{with probability } p_{Y|\vec{X}}(0|\vec{x}) \end{cases}$$

- In the limit as $n \to \infty$, assuming that the gradient descent finds the global optimum, the MMSE solution gives us:

$$\begin{aligned} \hat{y}(\vec{x}) &\to_{n\to\infty} E[y|\vec{x}] \\ &= \left(1 \times p_{Y|\vec{X}}(1|\vec{x})\right) + \left(0 \times p_{Y|\vec{X}}(0|\vec{x})\right) \\ &= p_{Y|\vec{X}}(1|\vec{x}) \end{aligned}$$

Pros and Cons of MMSE for Binary Classifiers

- **Pro:** In the limit as $n \to \infty$, the global optimum is $\hat{y}(\vec{x}) \to p_{Y|\vec{X}}(1|\vec{x})$.
- **Con:** The sigmoid nonlinearity is hard to train using MMSE. Remember the vanishing gradient problem: $\sigma'(wx) \to 0$ as $w \to \infty$, so after a few epochs of training, the neural net just stops learning.
- **Solution:** Can we devise a different loss function (not MMSE) that will give us the same solution ($\hat{y}(\vec{x}) \to p_{Y|\vec{X}}(1|\vec{x})$), but without suffering from the vanishing gradient problem?

Binary Cross Entropy

Suppose we treat the neural net output as a noisy estimator, $\hat{p}_{Y|\vec{X}}(y|\vec{x})$, of the unknown true pmf $p_{Y|\vec{X}}(y|\vec{x})$:

$$\hat{y}_i = \hat{p}_{Y|\vec{X}}(1|\vec{x}),$$

so that

$$\hat{p}_{Y|\vec{X}}(y_i|\vec{x}_i) = \begin{cases} \hat{y}_i & y_i = 1 \\ 1 - \hat{y}_i & y_i = 0 \end{cases}$$

The binary cross-entropy loss is the negative log probability of the training data, assuming i.i.d. training examples:

$$\begin{aligned} \mathcal{L}_{BCE} &= -\frac{1}{n} \sum_{i=1}^{n} \ln \hat{p}_{Y|\vec{X}}(y_i|\vec{x}_i) \\ &= -\frac{1}{n} \sum_{i=1}^{n} y_i \left(\ln \hat{y}_i\right) + (1 - y_i) \left(\ln(1 - \hat{y}_i)\right) \end{aligned}$$

The Derivative of BCE

BCE is useful because it has the same solution as MSE, without allowing the sigmoid to suffer from vanishing gradients. Suppose $\hat{y}_i = \sigma(wh_i)$.

$$\begin{aligned}
\nabla_w \mathcal{L} &= -\frac{1}{n}\left(\sum_{i:y_i=1} \nabla_w \ln \sigma(wh_i) + \sum_{i:y_i=0} \nabla_w \ln(1-\sigma(wh_i))\right) \\
&= -\frac{1}{n}\left(\sum_{i:y_i=1} \frac{\nabla_w \sigma(wh_i)}{\sigma(wh_i)} + \sum_{i:y_i=0} \frac{\nabla_w (1-\sigma(wh_i))}{1-\sigma(wh_i)}\right) \\
&= -\frac{1}{n}\left(\sum_{i:y_i=1} \frac{\hat{y}_i(1-\hat{y}_i)h_i}{\hat{y}_i} + \sum_{i:y_i=0} \frac{-\hat{y}_i(1-\hat{y}_i)h_i}{1-\hat{y}_i}\right) \\
&= -\frac{1}{n}\sum_{i=1}^{n} (y_i - \hat{y}_i)\, h_i
\end{aligned}$$

Why Cross-Entropy is Useful for Machine Learning

Binary cross-entropy is useful for machine learning because:

1. **Just like MSE, it estimates the true class probability:** in the limit as $n \to \infty$, $\nabla_W \mathcal{L} \to E\left[(Y - \hat{Y})H\right]$, which is zero only if
$$\hat{Y} = E\left[Y|\vec{X}\right] = p_{Y|\vec{X}}(1|\vec{x})$$
2. **Unlike MSE, it does not suffer from the vanishing gradient problem of the sigmoid.**

Unlike MSE, BCE does not suffer from the vanishing gradient problem of the sigmoid.

The vanishing gradient problem was caused by $\sigma' = \sigma(1 - \sigma)$, which goes to zero when its input is either plus or minus infinity.

- If $y_i = 1$, then differentiating $\ln \sigma$ cancels the σ term in the numerator, leaving only the $(1 - \sigma)$ term, which is large if and only if the neural net is wrong.
- If $y_i = 0$, then differentiating $\ln(1 - \sigma)$ cancels the $(1 - \sigma)$ term in the numerator, leaving only the σ term, which is large if and only if the neural net is wrong.

So binary cross-entropy ignores training tokens only if the neural net guesses them right. If it guesses wrong, then back-propagation happens.

Outline

Multinomial Classifier

Suppose, instead of just a 2-class classifier, we want the neural network to classify $\vec{x}$ as being one of K different classes. There are many ways to encode this, but one of the best is

$$\vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_K \end{bmatrix}, \quad y_k = \begin{cases} 1 & k = k^* \quad (k \text{ is the correct class}) \\ 0 & \text{otherwise} \end{cases}$$

A vector $\vec{y}$ like this is called a "one-hot vector," because it is a binary vector in which only one of the elements is nonzero ("hot"). This is useful because minimizing the MSE loss gives:

$$\hat{y} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_K \end{bmatrix} = \begin{bmatrix} \hat{p}_{Y_1|\vec{X}}(1|\vec{x}) \\ \hat{p}_{Y_2|\vec{X}}(1|\vec{x}) \\ \vdots \\ \hat{p}_{Y_K|\vec{X}}(1|\vec{x}) \end{bmatrix},$$

One-hot vectors and Cross-entropy loss

The cross-entropy loss, for a training database coded with one-hot vectors, is

$$\mathcal{L}_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ki} \ln \hat{y}_{ki}$$

This is useful because:

1. **Like MSE, Cross-Entropy has an asymptotic global optimum at:** $\hat{y}_k \to p_{Y_k|\vec{X}}(1|\vec{x})$.
2. **Unlike MSE, Cross-Entropy with a softmax nonlinearity suffers no vanishing gradient problem.**

Softmax Nonlinearity

The multinomial cross-entropy loss is only well-defined if $0 < \hat{y}_{ki} < 1$, and it is only well-interpretable if $\sum_k \hat{y}_{ki} = 1$. We can guarantee these two properties by setting

$$\begin{aligned}\hat{y}_k &= \underset{k}{\text{softmax}}\left(W\vec{h}\right) \\ &= \frac{\exp(\bar{w}_k \vec{h})}{\sum_{\ell=1}^{K} \exp(\bar{w}_\ell \vec{h})},\end{aligned}$$

where $\bar{w}_k$ is the k^{th} row of the W matrix.

Sigmoid is a special case of Softmax!

$$\underset{k}{\text{softmax}}\left(W\vec{h}\right) = \frac{\exp(\bar{w}_k\vec{h})}{\sum_{\ell=1}^{K}\exp(\bar{w}_\ell\vec{h})}.$$

Notice that, in the 2-class case, the softmax is just exactly a logistic sigmoid function:

$$\underset{1}{\text{softmax}}(W\vec{h}) = \frac{e^{\bar{w}_1\vec{h}}}{e^{\bar{w}_1\vec{h}} + e^{\bar{w}_2\vec{h}}} = \frac{1}{1 + e^{-(\bar{w}_1-\bar{w}_2)\vec{h}}} = \sigma\left((\bar{w}_1 - \bar{w}_2)\vec{h}\right)$$

so everything that you've already learned about the sigmoid applies equally well here.

Outline

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

- Use MSE to achieve $\hat{y} \rightarrow E\left[\vec{y}|\vec{x}\right]$. That's almost always what you want.
- 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.
- After you're done training, you can make your cell phone app more efficient by throwing away the uncertainty:
  - Replace softmax output nodes with max
  - Replace logistic output nodes with unit-step
  - Replace tanh output nodes with signum