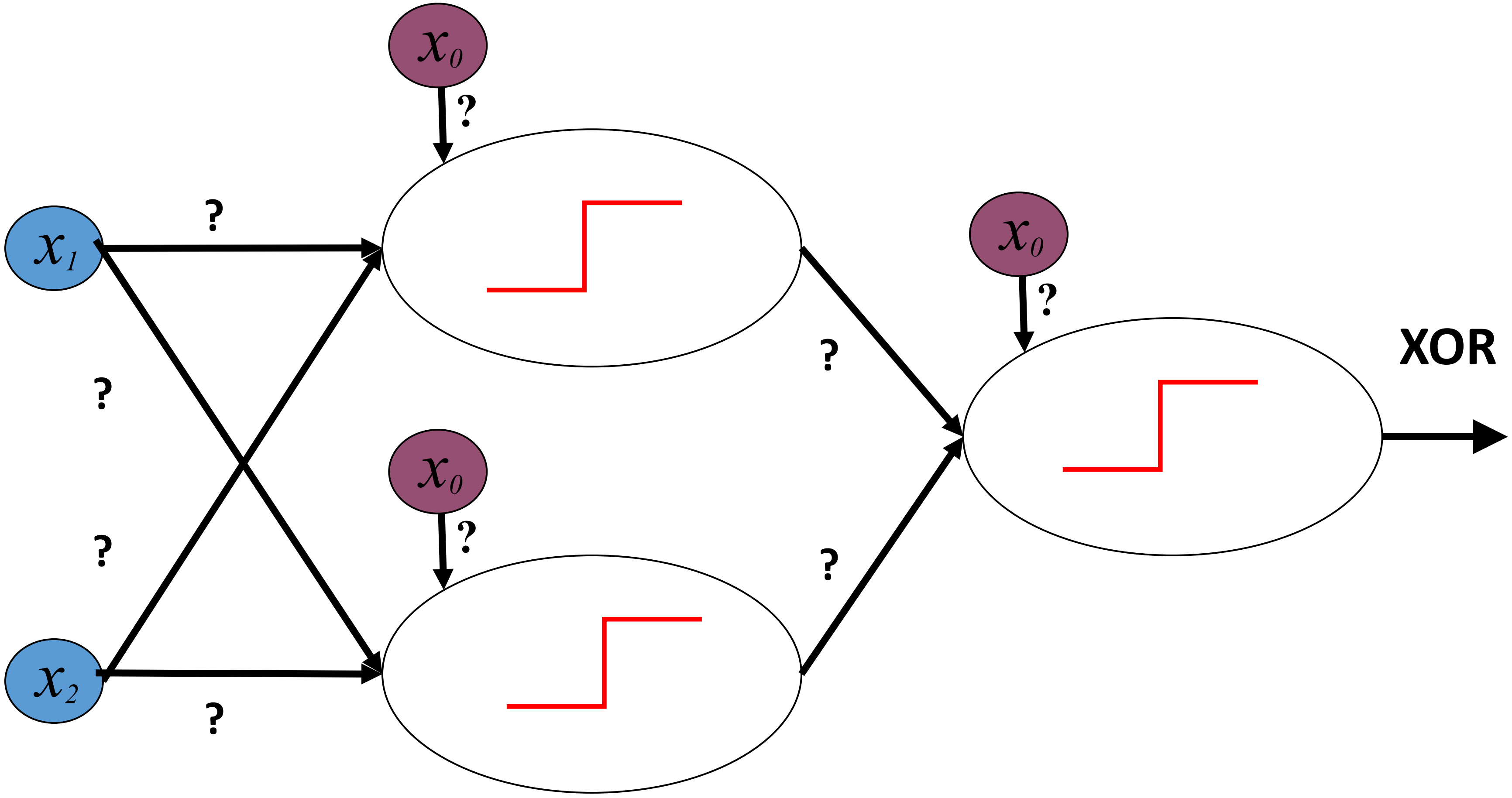
Multilayer Percetprons

Bryan Pardo

Deep Learning

Northwestern University

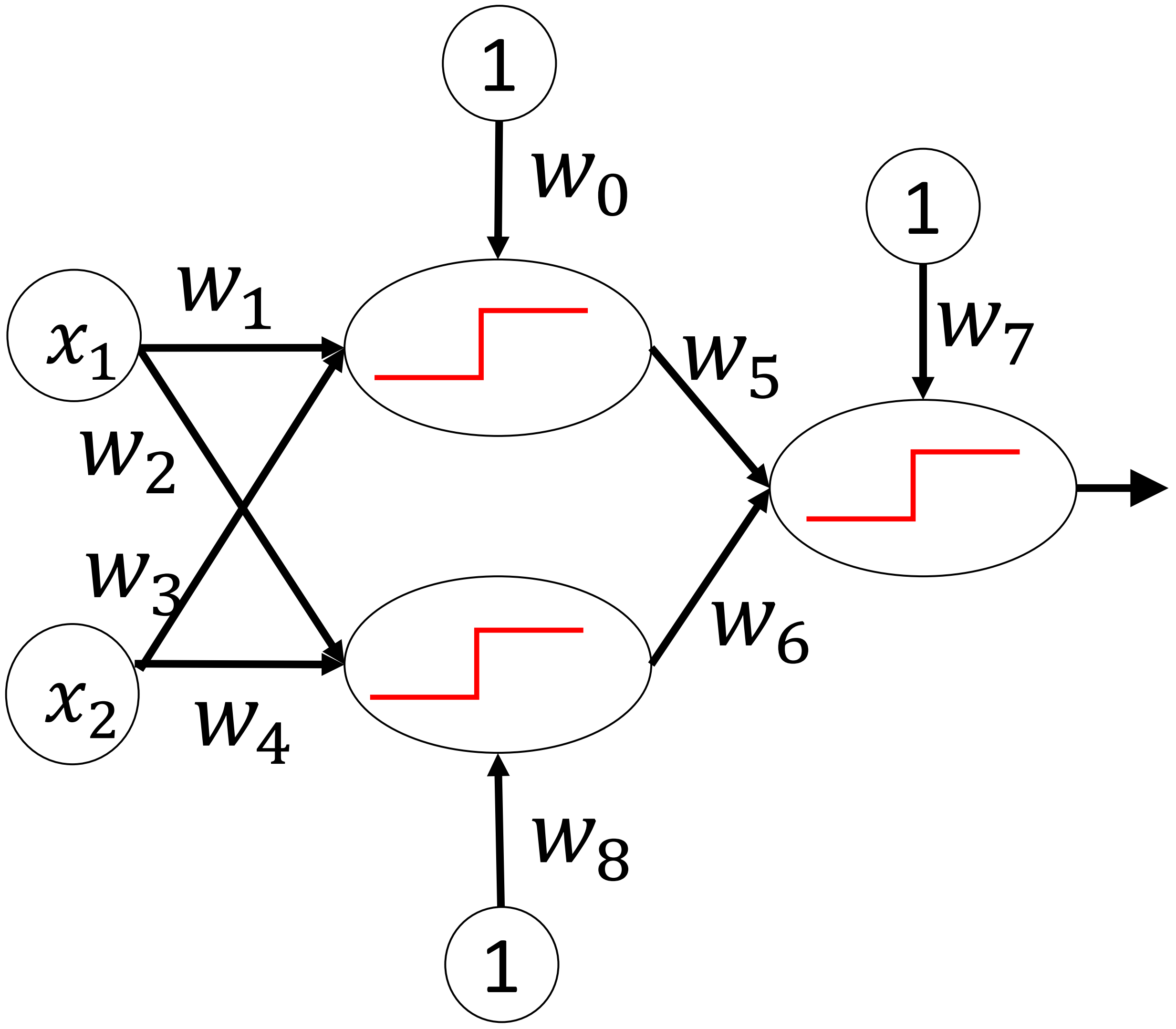
Combining perceptrons can make any Boolean function



…if you can set the weights & connections right

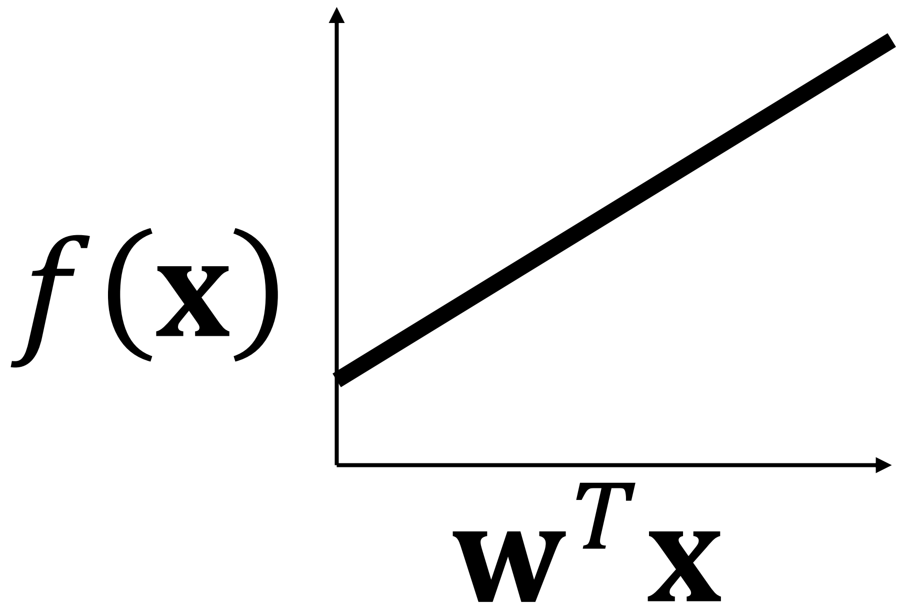
A problem with step functions: assignment of error

- Stymies multi-layer weight learning
- Limits us to a single layer of units
- Thus, only linear functions
- You can hand-wire XOR perceptrons, but the sytem can't learn XOR with perceptrons



Linear Units & Delta Rule

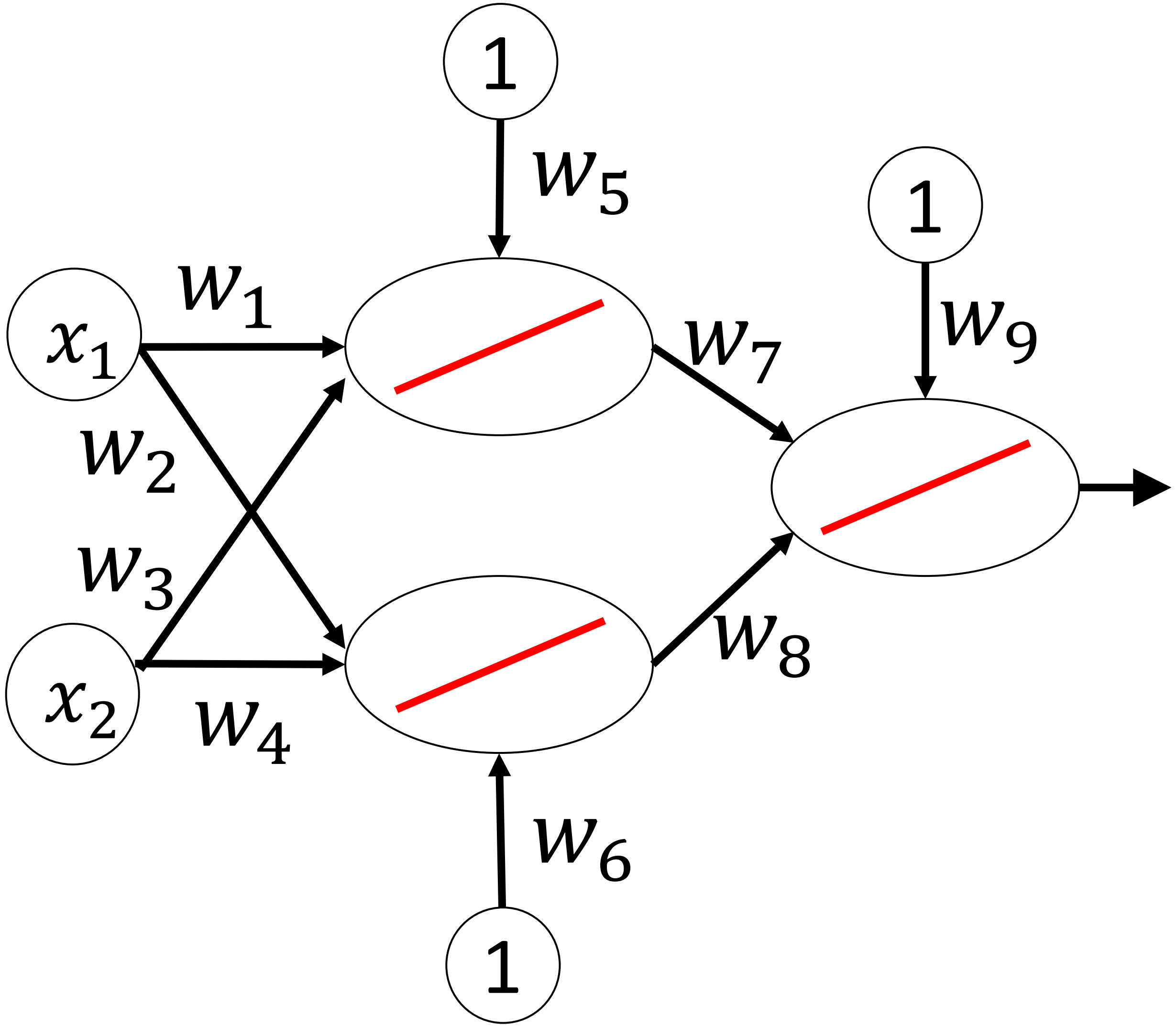
Solution: Remove the step function



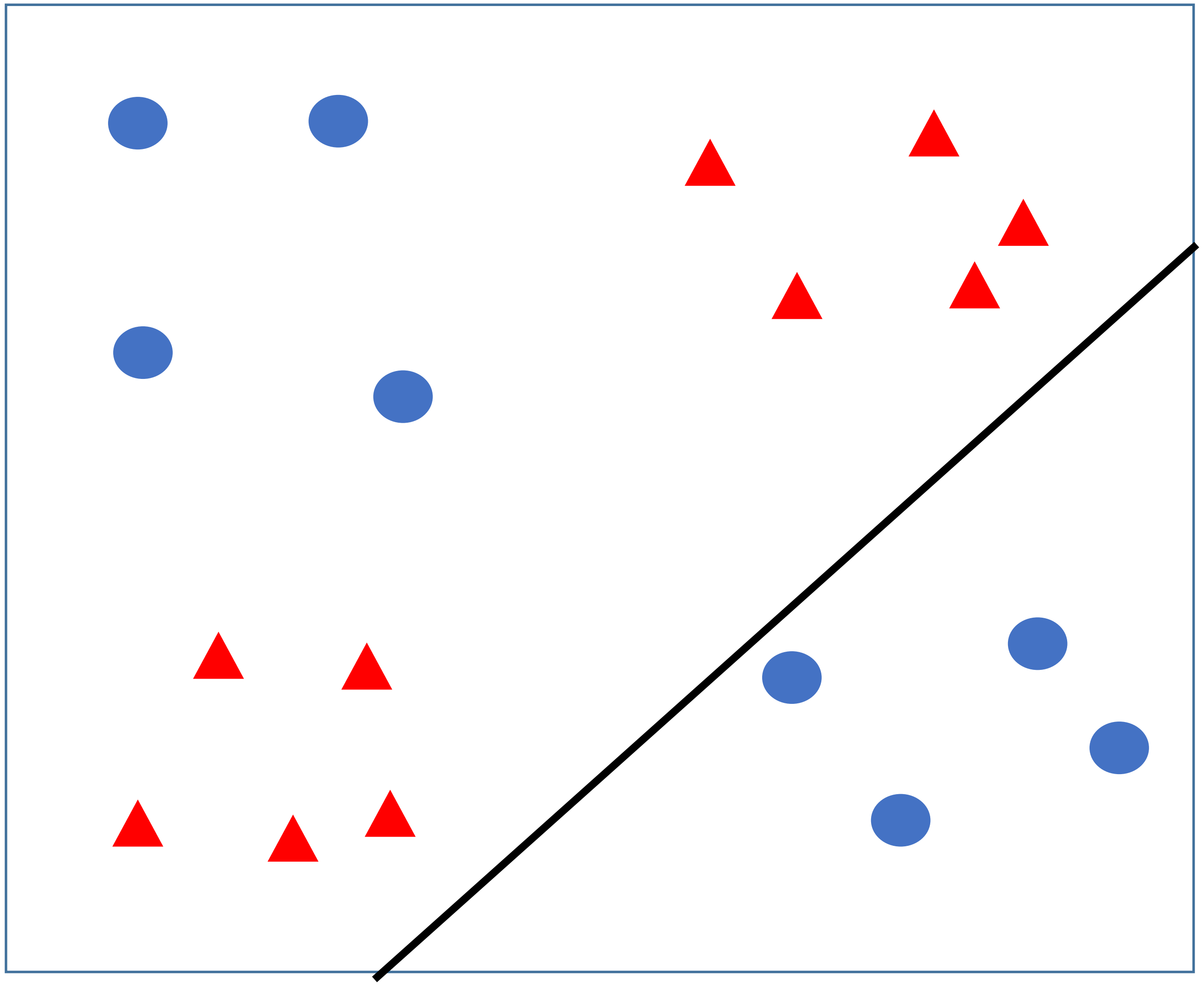
$$f(\mathbf{x}) = \sum_{i=0}^{n} w_i x_i = \mathbf{w}^T\mathbf{x}$$

Better & worse than a perceptron

- All changes in input result in changed output
- This gives us a gradient everywhere
- We can learn multiple layers of weights.
- **Combining linear functions only gives you linear functions**
- you can't represent XOR



Many linear units: Only linear decisions



This is XOR.

A multilayer perceptron with linear units CANNOT learn XOR

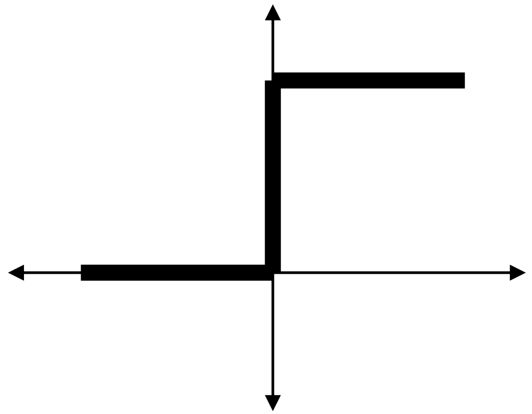
The Sigmoid Unit

Rumelhart, David E., James L. McClelland, and PDP Research Group. Parallel distributed processing. Vol. 1. Cambridge, MA, USA:: MIT press, 1987.

Sigmoid (aka Logistic) function: best of both

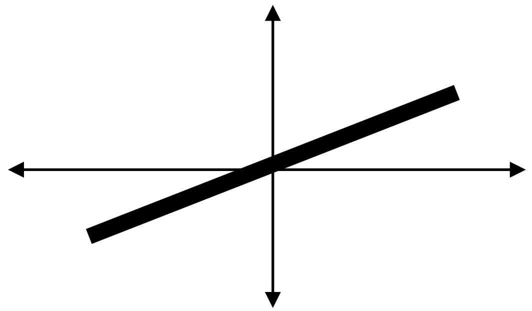
- Perceptron

$$f(x) = \begin{cases} 1\ if\ 0 < \sum_{i=0}^{n} w_i x_i \\ -1\ else \end{cases}$$



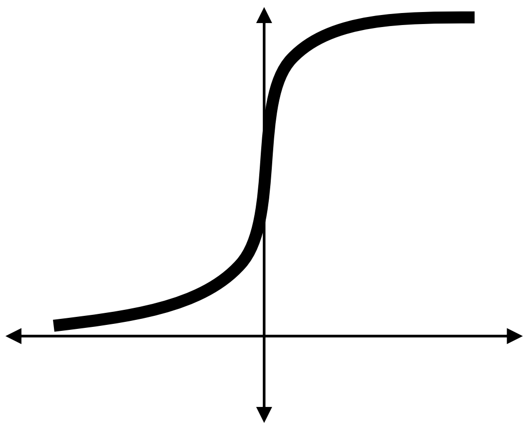
- Linear

$$f(x) = \mathbf{w}^T \mathbf{x} = \sum_{i=0}^{n} w_i x_i$$



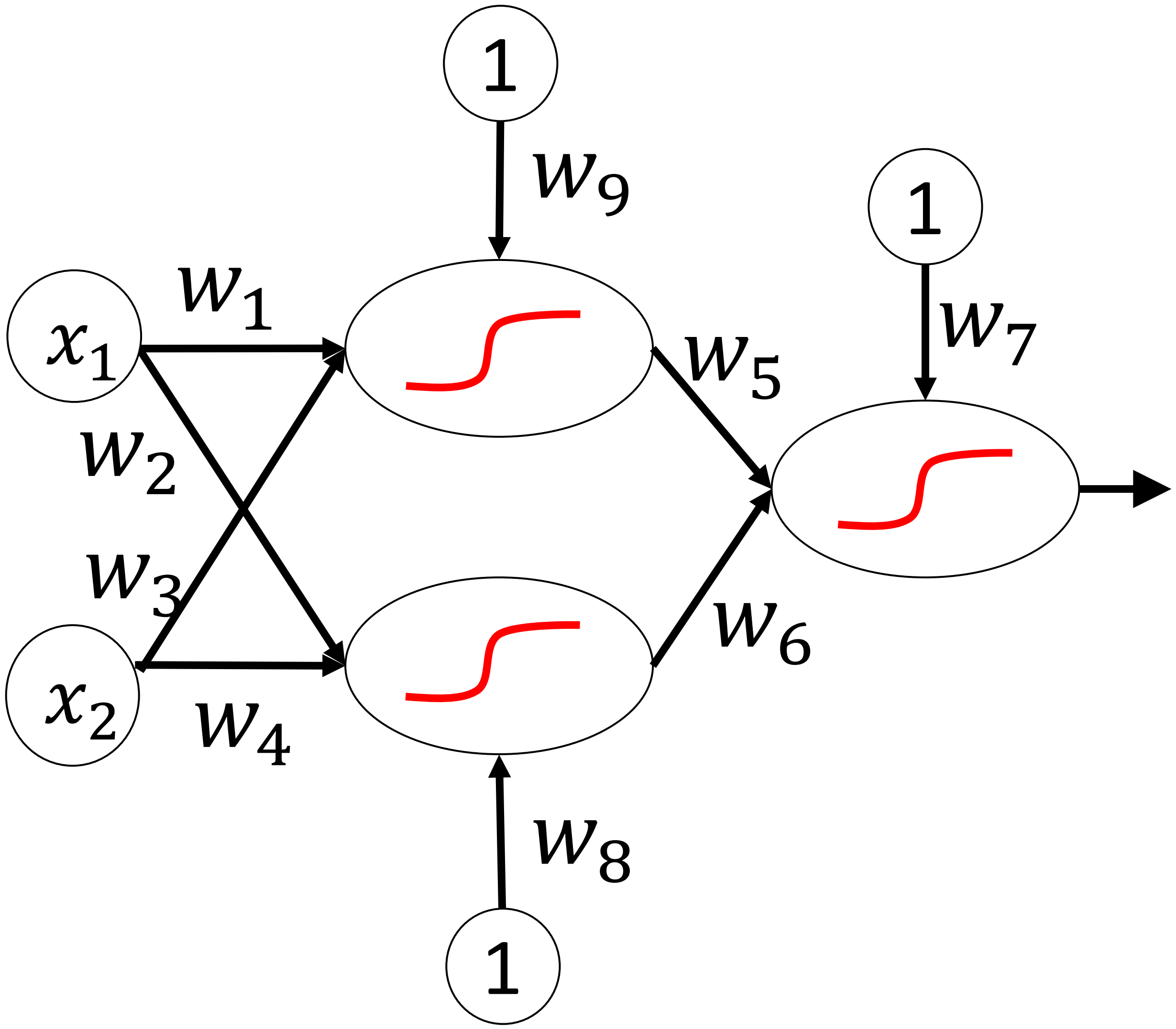
- Sigmoid

$$f(x) = \frac{1}{1 + e^{-(\mathbf{w}^T \mathbf{x})}}$$



A network of sigmoid units

- Small changes in input result in output
- This gives us a gradient everywhere
- We can learn multiple layers of weights.
- Combining layers gives non-linear functions



Sigmoid changes (almost) everything

Easy to differentiate

$\sigma'(\mathbf{w}^T\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x})(1\text{-}\ \sigma(\mathbf{w}^T\mathbf{x}))$

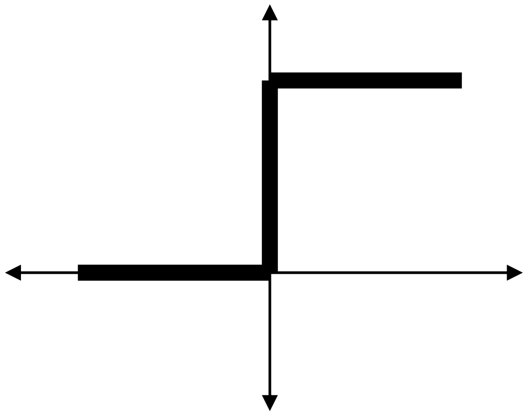
Gradient everywhere

This allows backpropagation of the gradient through multiple layers

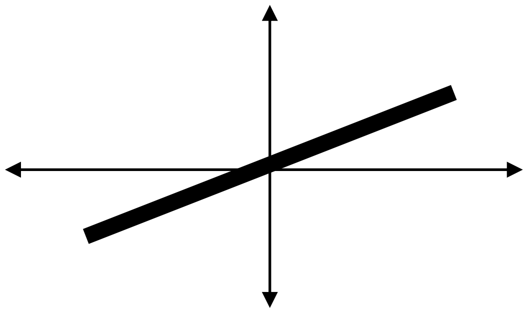
Nonlinearity allows arbitrary nonlinear functions to be built by using multiple layers.

Sigmoid (aka Logistic) function: best of both

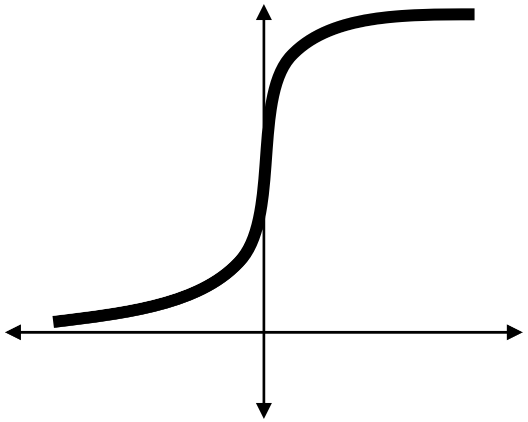
- Perceptron $$f(x) = \begin{cases} 1\ if\ 0 < \sum_{i=0}^{n} w_i x_i \\ -1\ else \end{cases}$$



- Linear $$f(x) = \mathbf{w}^T\mathbf{x} = \sum_{i=0}^{n} w_i x_i$$



- Sigmoid $$f(x) = \sigma(x) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x})}}$$



What’s cool about the sigmoid function

- It looks like a rounded step function, so we can build circuits of arbitrary functions like we can with perceptrons

- It has non-zero slope everywhere and no sharp corners

- The derivative of the function is this: $\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$

- …and it’s easy to plug into the gradient descent algorithm to get the learning rule.

For each dimension *i*, take the partial derivative

$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z} \frac{\partial z}{\partial w_i}$ gives the change of our loss function L with respect to weight w_i

Here, $L = \frac{1}{2}(y - \hat{y})^{\mathbf{2}}$ and $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$ and $z = \mathbf{w}^T\mathbf{x}$

Therefore $\frac{\partial L}{\partial \hat{y}} = (y - \hat{y}) = \big(y - \sigma(z)\big)$

..and $\frac{\partial \hat{y}}{\partial z} = \sigma(z)(1 - \sigma(z))$, as was given to us.

…and $\frac{\partial z}{\partial w_i} = x_i$, since $z = \mathbf{w}^T\mathbf{x} = w_0x_0 \ldots + w_ix_i \ldots + w_dx_d$

Therefore, $\frac{\partial L}{\partial w_i} = \big(y - \sigma(z)\big)\sigma(z)(1 - \sigma(z))x_i$

For each dimension *i*, take the partial derivative

From the previous slide: $\frac{\partial L}{\partial w_i} = \big(y - \sigma(z)\big)\sigma(z)(1 - \sigma(z))x_i$

Let's compose $\sigma(z) = \frac{1}{1+e^{-z}}$ and $z = \mathbf{w}^T\mathbf{x}$ into one function (called $\sigma(\mathbf{x})$), to get the following:

$$\sigma(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$$

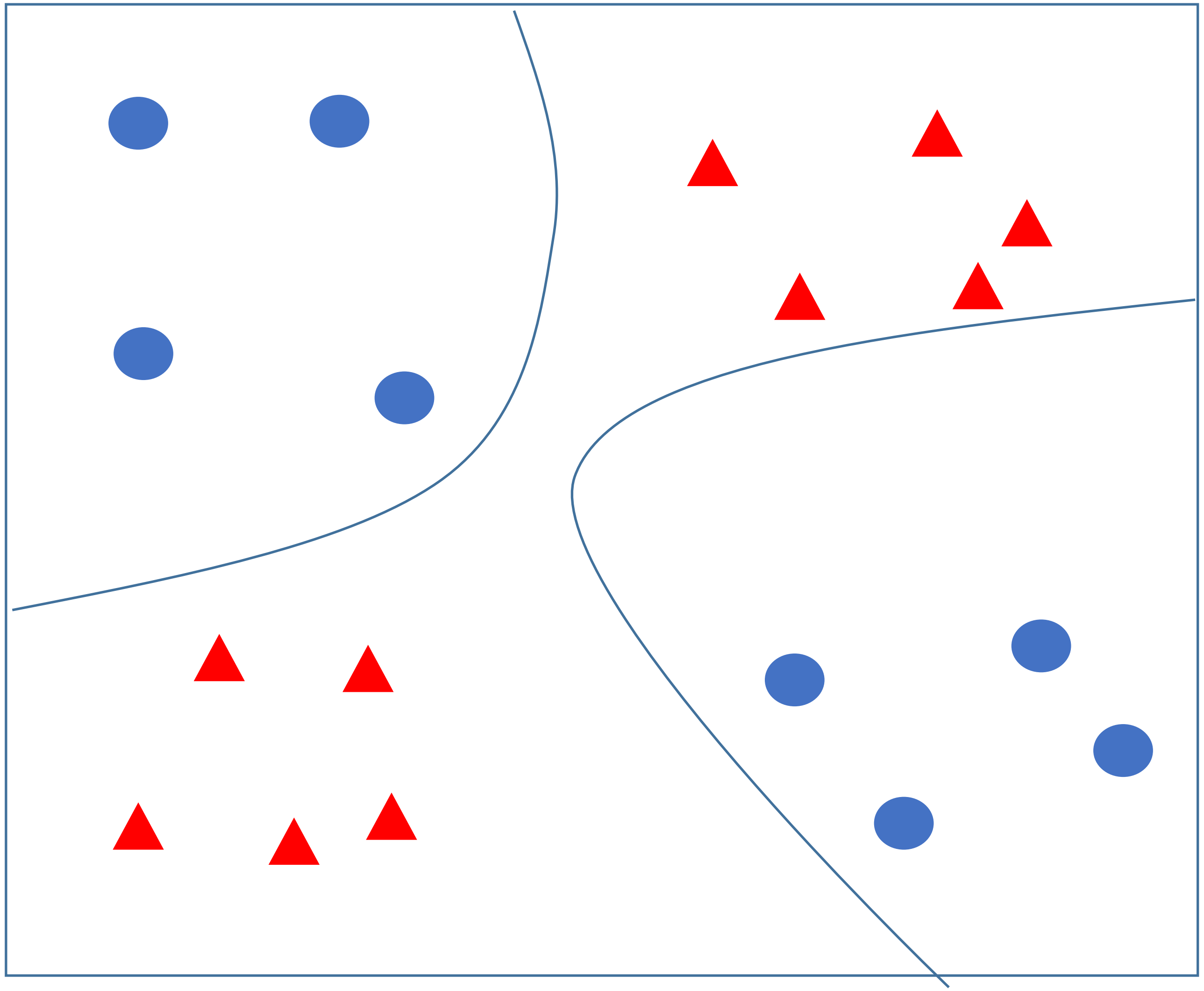
This lets us now write the change in loss as:

$$\frac{\partial L}{\partial w_i} = \big(y - \sigma(\mathbf{x})\big)\sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))x_i$$

The promise of many layers

- Each layer learns an abstraction of its input representation (we hope)
-
- As we go up the layers, representations become increasingly abstract
- The hope is that the intermediate abstractions facilitate learning functions that require non-local connections in the input space (recognizing rotated & translated digits in images, for example)
- Modern neural networks are up to 100 layers deep

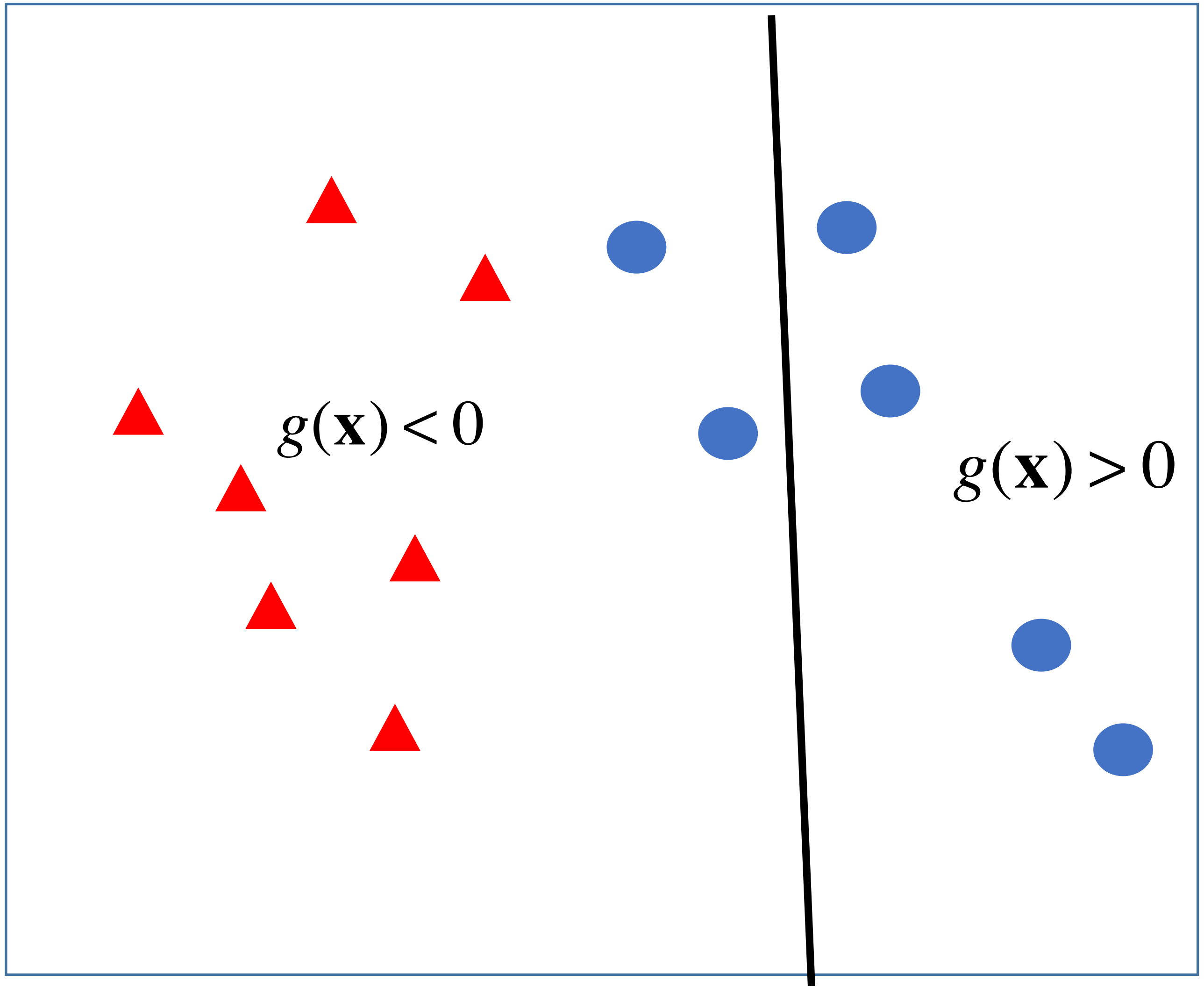
Multilayer Perceptron with sigmoid units



This is XOR.

A multilayer perceptron with sigmoid units CAN learn XOR…or any other arbitrary Boolean function.

Example objective J : sum of squared errors



$$h(x) = f(x) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x})}}$$

$$SSE = \sum_{i}^{n} (y_i - h(\mathbf{x}_i))^2$$

Gradient non-zero everywhere!

Backpropagation of error

Where we left off

- We have the $\sigma(x)$ sigmoid function that we can train with gradient descent, because it's differentiable and has a non-zero gradient everywhere.
- We can plug multiple sigmoids together to form arbitrary Boolean functions, by just interpreting the last output with sign($\sigma(x)$)
- We now need a way to have error from the output sigmoid function to flow to the input, so we can adjust the parameters of every $\sigma(x)$ on the path from the input to the output when we do our gradient descent.

Consider one output node

Let's define a function…

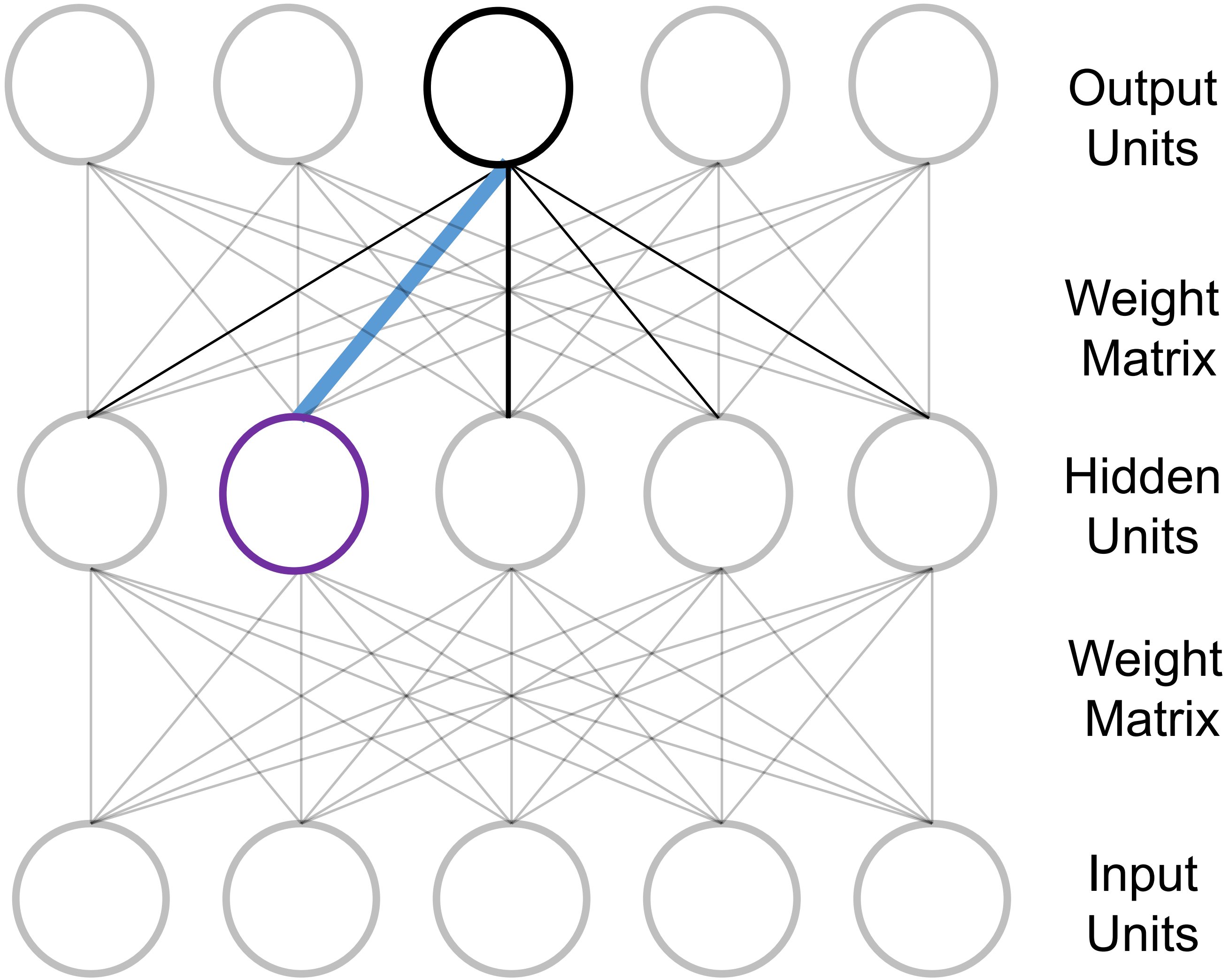
$$\delta = \big(y - \sigma(\mathbf{x})\big)\,\sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))$$

Now this…

$$\frac{\partial L}{\partial w_i} = \big(y - \sigma(\mathbf{x})\big)\sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))x_i$$

…becomes this: $\frac{\partial L}{\partial w_i} = \delta x_i$

For any output node k we just use this, as before.



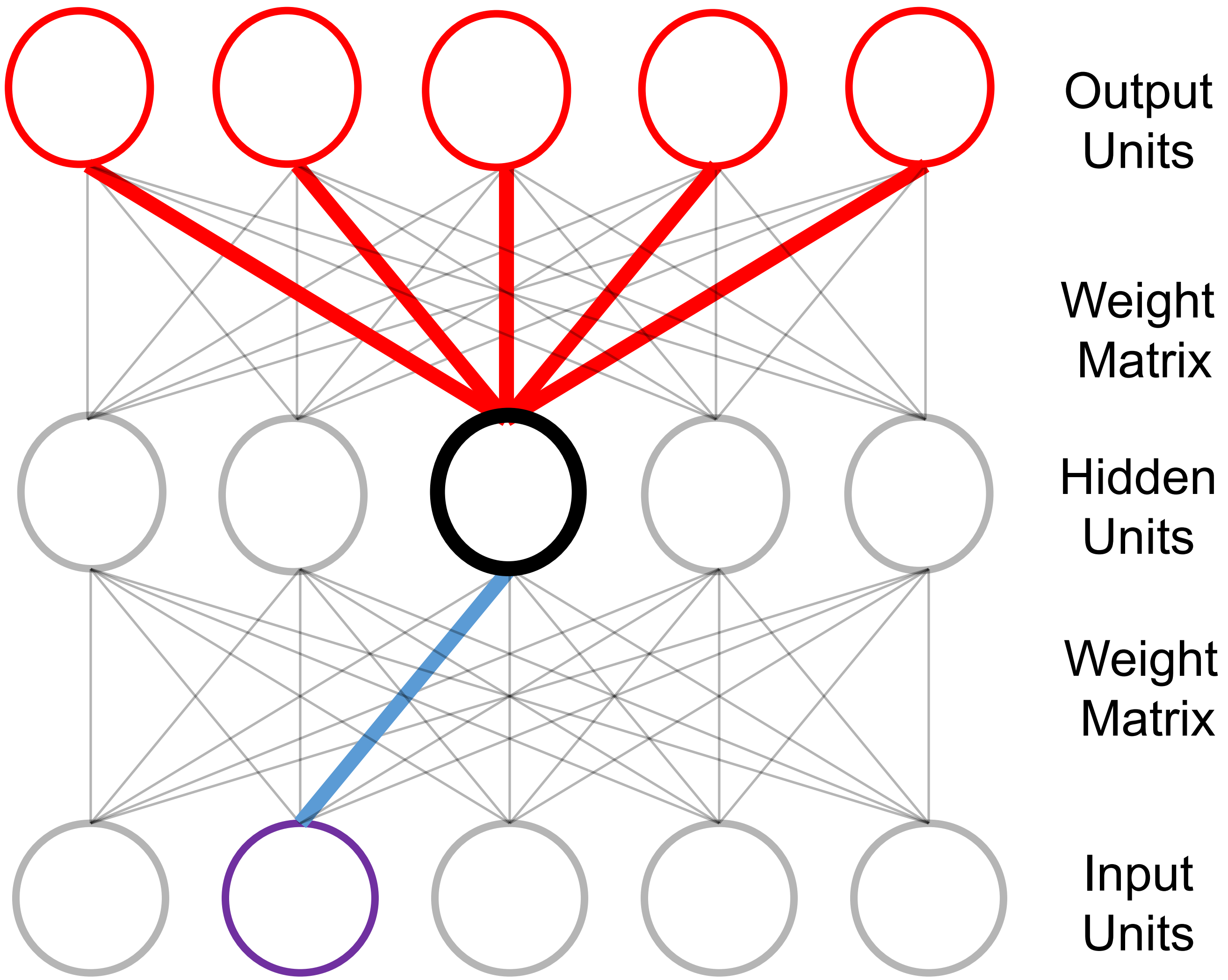
Consider one hidden node

For a hidden node h we need to redefine δ. Instead of comparing the output of the node to a known target output y, we look at its contribution to the output of the k nodes it is connected to at the next layer.

$$\delta = \left(\sum_k w_k \, \delta_k\right) \sigma(\mathbf{x})(1 - \sigma(\mathbf{x}))$$

…and we then do: $\dfrac{\partial L}{\partial w_i} = \delta x_i$

We can do this repeatedly for multiple hidden layers.



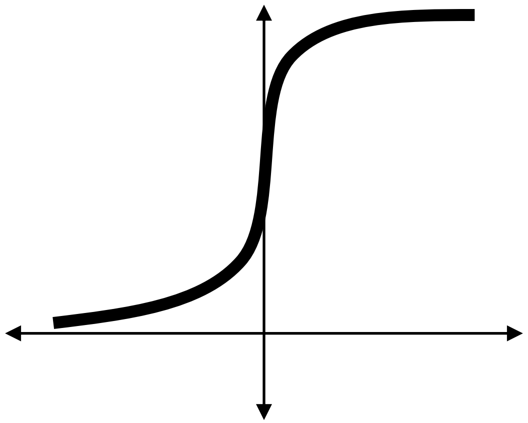
Some stuff I should mention

Sigmoid + SSE are not your only choices

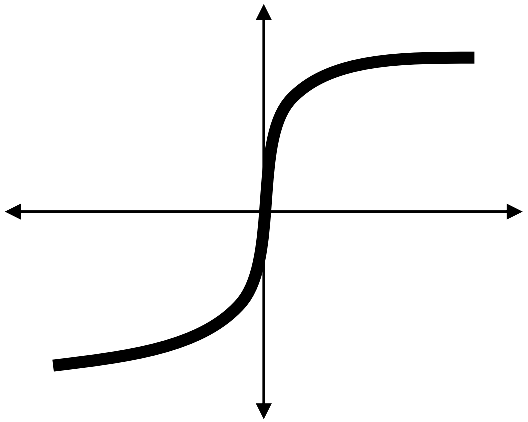
- Pick an activation function
- Pick a loss function
- Make sure they're both differentiable (or sub-differentiable)
- You can now do backpropagation of error

TanH: A shifted sigmoid

- Sigmoid $$f(x) = \frac{1}{1 + e^{-(\mathbf{w}^T\mathbf{x})}}$$

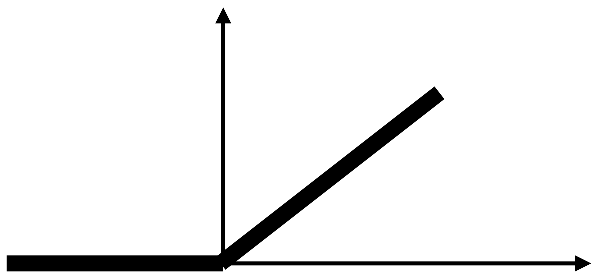


- TanH $$f(x) = \frac{2}{1 + e^{-2(\mathbf{w}^T\mathbf{x})}} - 1$$

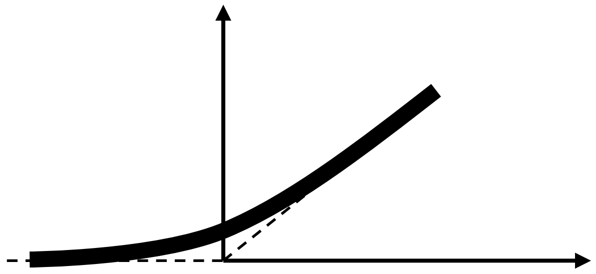


Rectified Linear Unit (ReLU) & Soft Plus :

- ReLU $f(x) = \max(0, \mathbf{w}^T\mathbf{x})$



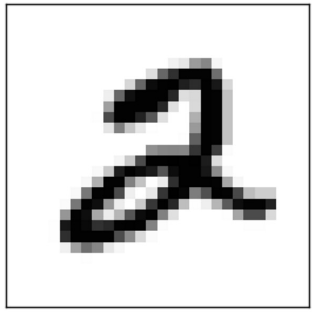
- Soft Plus $f(x) = \ln(1 + e^{\mathbf{w}^T\mathbf{x}})$



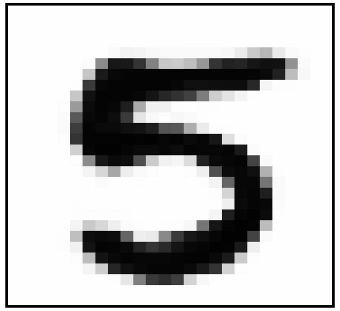
- Both can be combined in layers to make non-linear functions

"One Hot" Encoding

- A vector of values where a single element is 1 and all the rest are 0
- Common way to encode the true label, y, in a multi-class labeling problem
- Can be interpreted as a probability distribution



y = 0 0 1 0 0 0 0 0 0 0



y = 0 0 0 0 0 1 0 0 0 0

Probability distribution

* Discrete random variable X represents some experiment.

* $P(X)$ is the probability distributions over $\{x_1,...,x_n\}$, the set of possible outcomes for X.

* These outcomes are mutually exclusive.

* Their probabilities sum to one : $\sum_{i=1}^{n} P(x_i) = 1$

Soft Max Function

- Turns an N-dimensional vector of real numbers into a probability distribution, even if the numbers are both pos
- For a deep net, a_i is the output of the ith node in the output layer

$$p_i = \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}$$

Why softmax?

Why do I need this? $p_i = \frac{e^{a_i}}{\sum_{j=1}^{N} e^{a_j}}$

Wouldn’t taking the absolute value and averaging do just as well?

$$p_i = \frac{|a_i|}{\sum_{j=1}^{N} |a_j|}$$

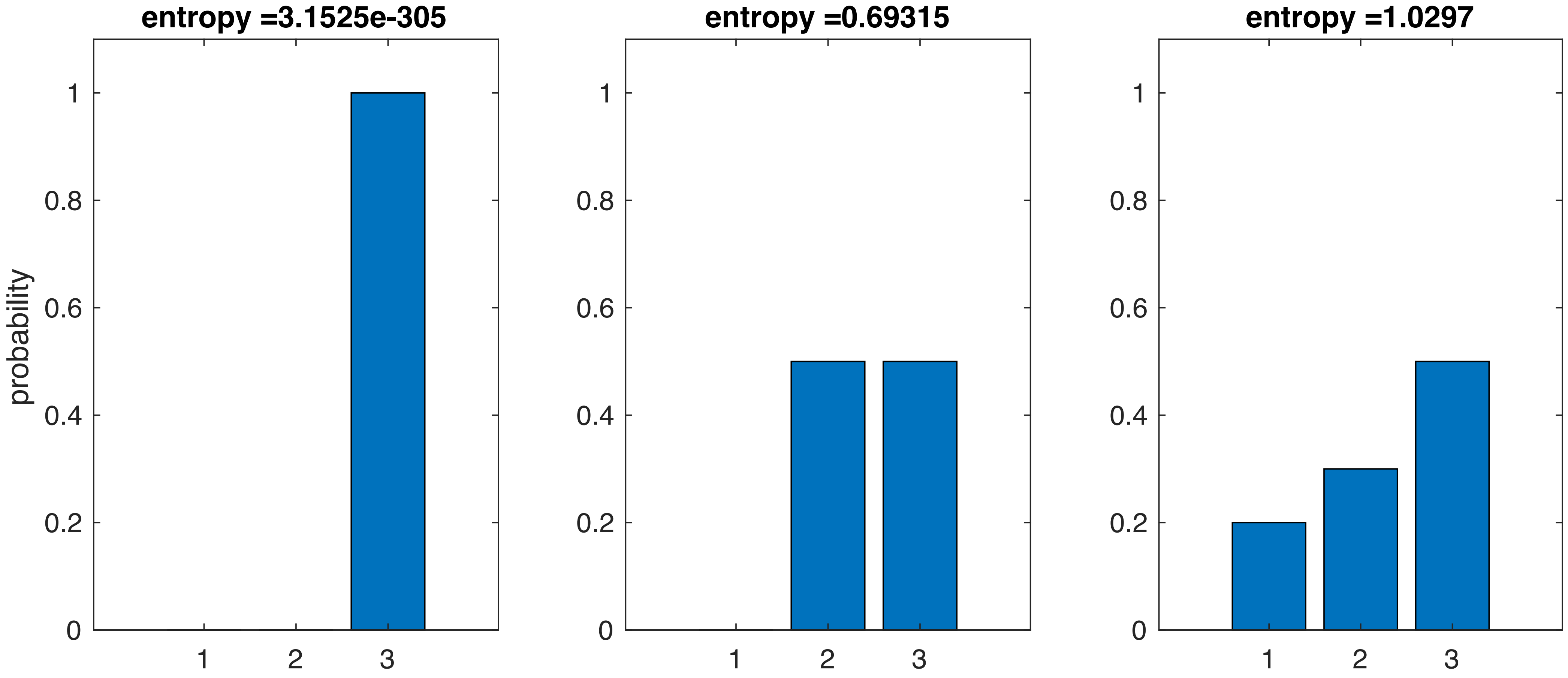
- Softmax is a multivariate extension of the sigmoid (logistic) function
- When combined with cross entropy loss function, the resulting derivative is a very nice one.

Entropy

- Entropy is the measure of the skewedness of a distribution
- The higher the entropy, the harder it is to guess the value a random variable will take when we draw from the distribution.
- Here,

$$H(P) = -\sum_{i=1}^{N} P(i)\log(P(i))$$

Some examples

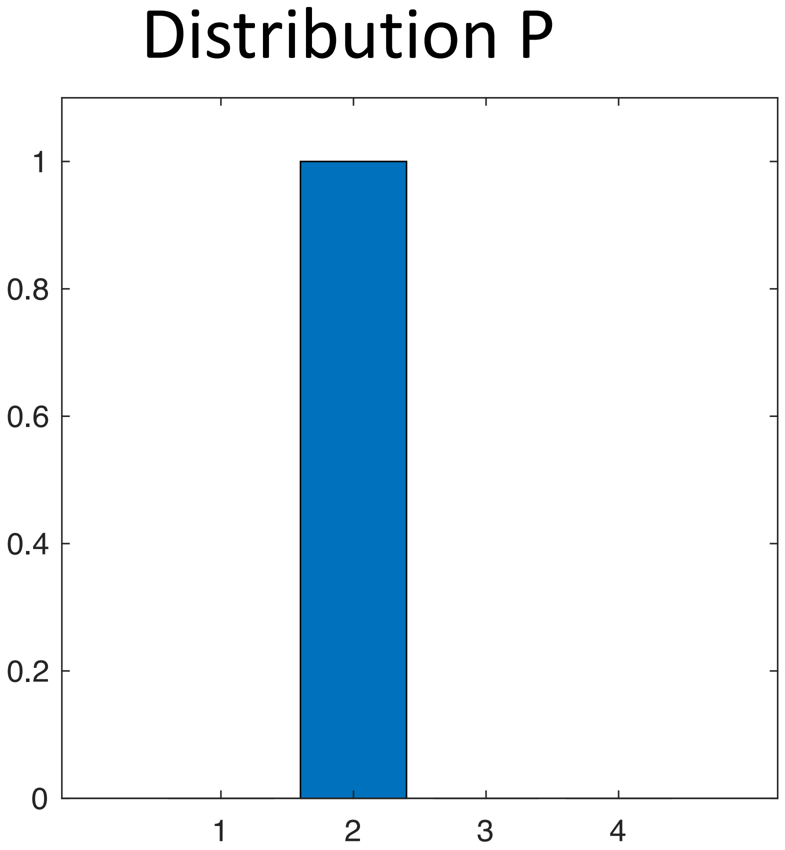


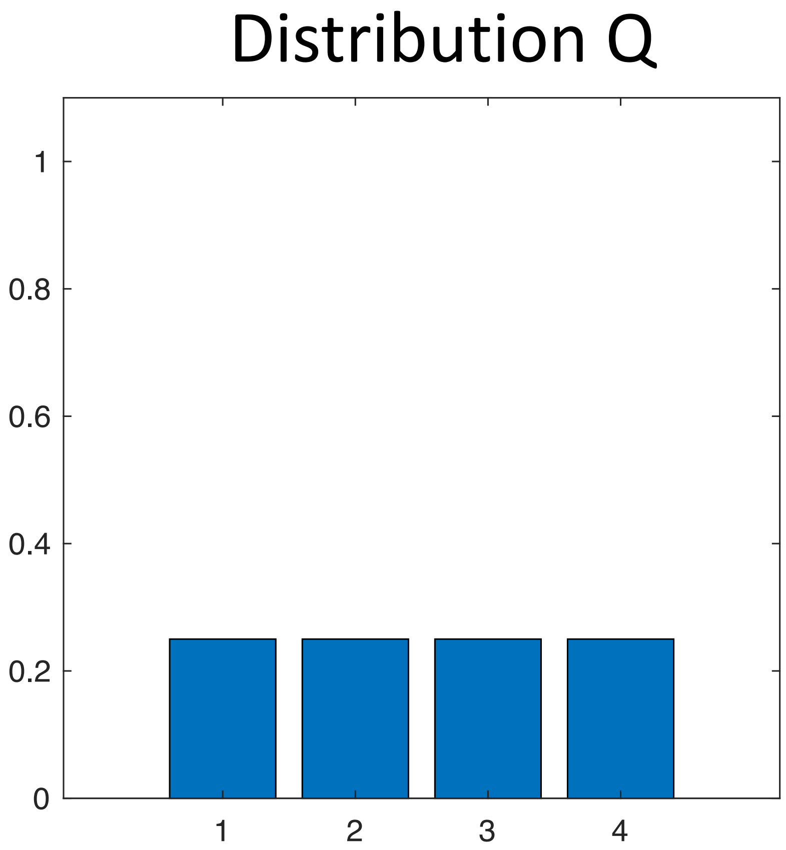
Cross Entropy

- Cross entropy is a measure of the similarity between distributions
- It is *NOT* symmetric.

$$H(P,Q) = -\sum_{i=1}^{N} P(i)\log(Q(i))$$

An example

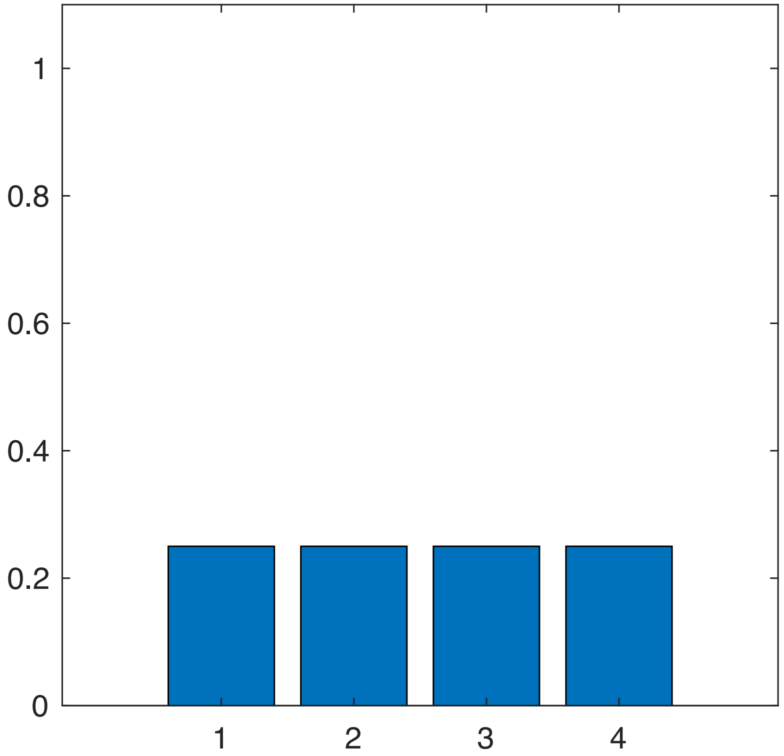




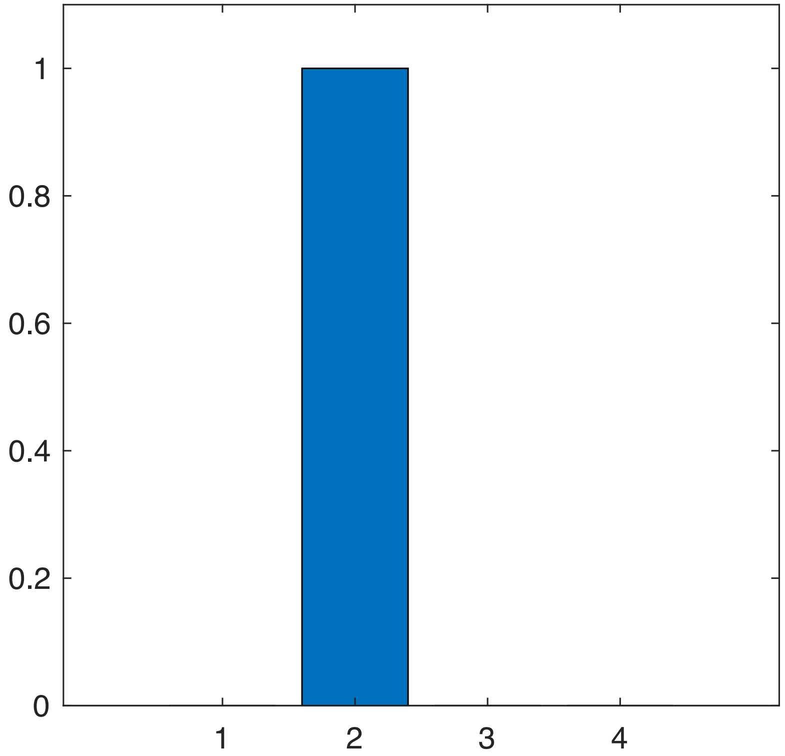
$$H(P,Q) = -\sum_{i=1}^{N} P(i)\log(Q(i)) = 1.39$$

An example

Distribution P



Distribution Q



$$H(P,Q) = -\sum_{i=1}^{N} P(i)\log(Q(i)) = \infty$$

Cross Entropy Loss Function

Given: “true” distribution $y = \{y_1, y_2, \dots y_N\}$ <-often a one-hot encoding

and estimated distribution $\hat{y} = \{\hat{y}_1, \hat{y}_2, \dots \hat{y}_N\}$ <-soft max over the last layer

Define cross entropy loss between 2 distributions as

$$L(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\hat{y}_i)$$

A common approach…

- Define labels with a one-hot vector encoding
- Make the last layer have n nodes for an n-way classification problem
- Apply soft max to the last layer
- Use a cross-entropy loss function
- The resulting derivative of the loss function is wonderfully simple:

$$\frac{\partial L}{\partial a_i} = \hat{y}_i - y_i$$

L is the loss, i is the index to a node, a is the output of the last layer, $\hat{y}$ is the softmax probability distribution over the output layer of the network and y is the one-hot-encoding label.

There are many activation & loss functions

- As a system designer, you need to consider what activation function make sense for your problem
- The right loss function makes the difference between a learnable problem and an unlearnable one
- Different layers may have different activation functions
- Multiple loss functions may be used when teaching the network