

Multilayer Neural Network

Dr Chang Xu

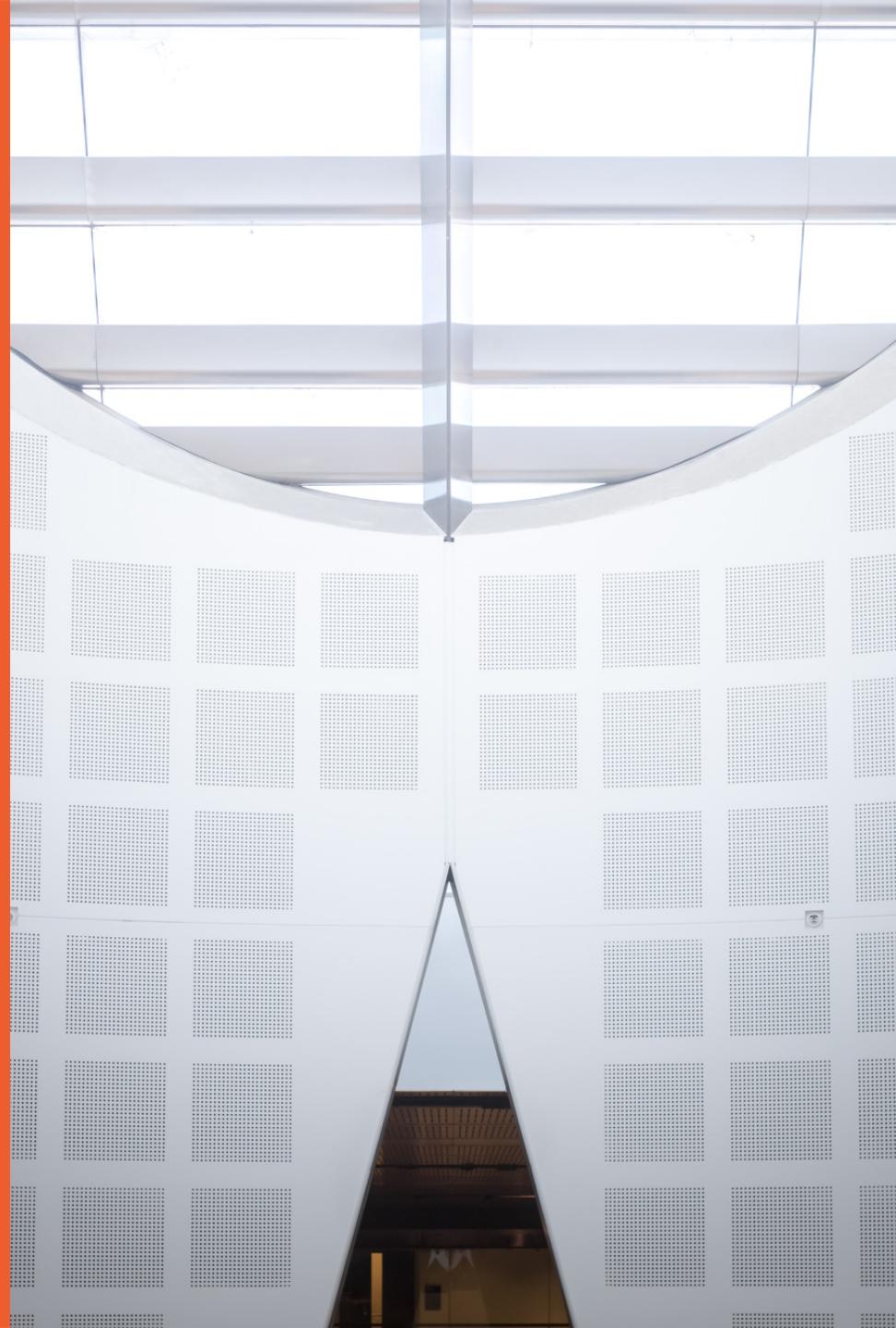
UBTECH Sydney AI Centre

Acknowledgements:

Jue Wang and Xinyuan Chen



THE UNIVERSITY OF
SYDNEY



Quick Review

Outline

- Perceptron
- Multilayer Neural Network
- Backpropagation
- Activation Functions
- Stochastic Gradient Descent

Perceptron: the Prelude of Deep Learning

A neuron cell

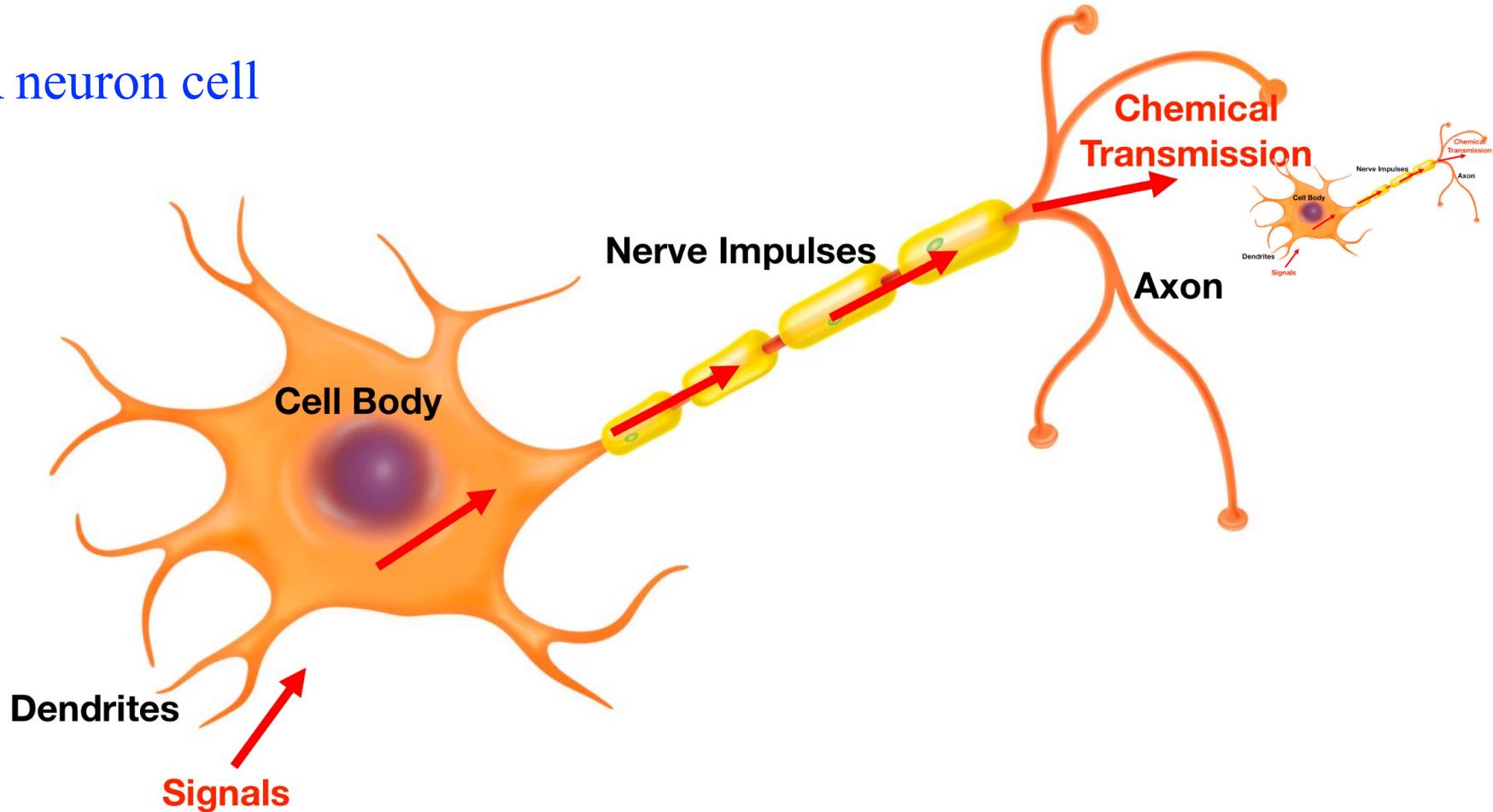
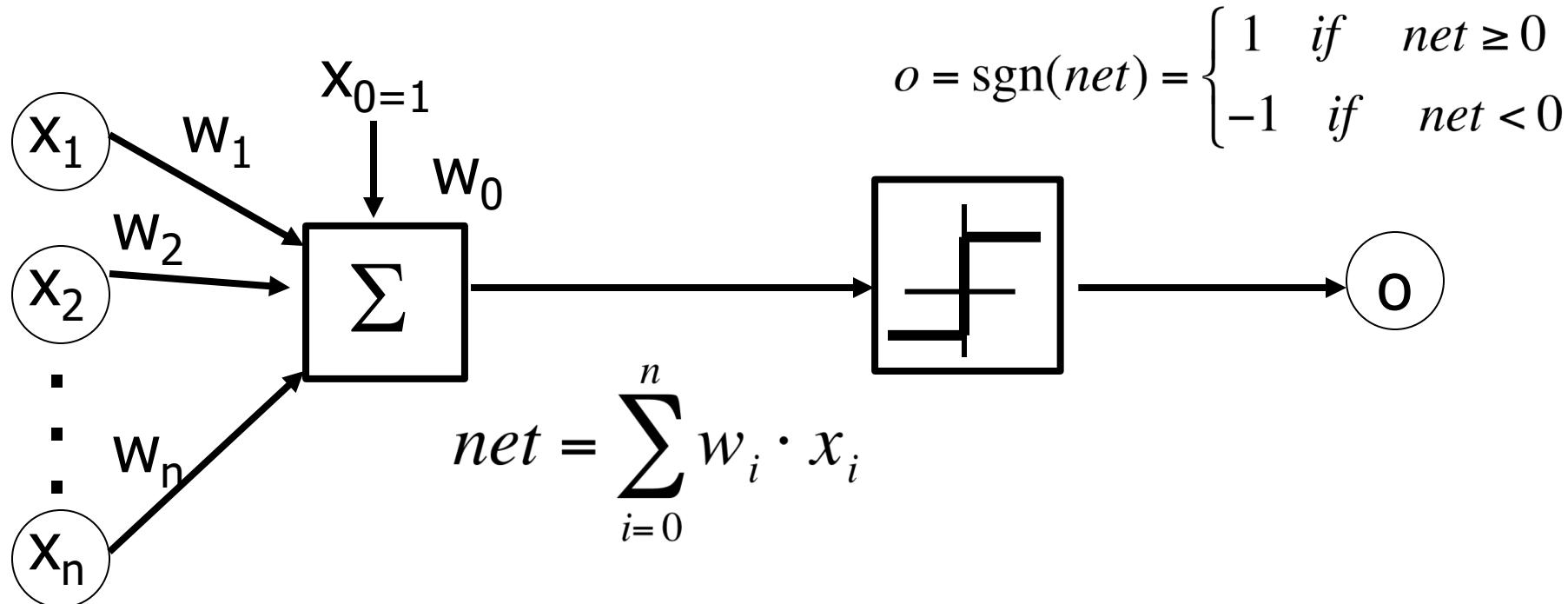


Image credit to: thinglink.com

Perceptron: the Prelude of DL



Perceptron

- The neuron has a real-valued output which is a weighted sum of its inputs.

Neuron's estimate of
the desired output

$$\hat{y} = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

weight vector

input vector

The aim of learning is to minimize the discrepancy between the desired output and the actual output:

$$\min_w \quad \frac{1}{2} \sum_{j=1}^n (y_j - \hat{y}_j)^2$$

Perceptron

- Define the error as the squared residuals summed over all training cases:
- Now differentiate to get error derivatives for weights
- The update rule changes the weights in proportion to their error derivatives summed over all training cases



$$\begin{aligned} L &= \frac{1}{2} \sum_{j=1}^n (y_j - \hat{y}_j)^2 \\ &= \frac{1}{2} \sum_{j=1}^n (y_j - w_i x_{i,j})^2 \end{aligned}$$

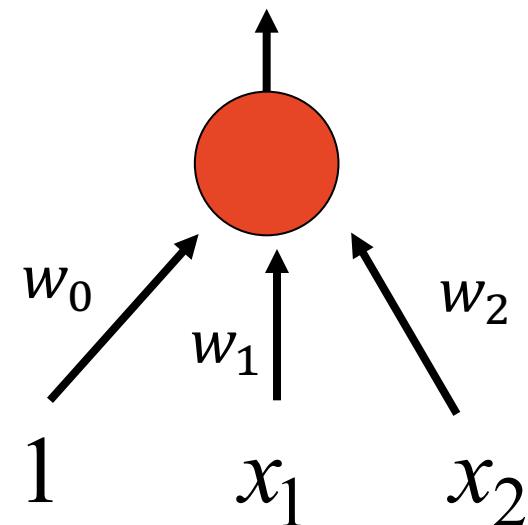
$$\begin{aligned} \frac{\partial L}{\partial w_i} &= \frac{1}{2} \sum_j \frac{\partial L_j}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial w_i} \\ &= - \sum_j (y_j - \hat{y}_j) x_{i,j} \end{aligned}$$

$$w_i \leftarrow w_i - \alpha \frac{\partial L}{\partial w_i}$$

Perceptron

- A linear neuron is a more flexible model if we include a bias.
- We can avoid having to figure out a separate learning rule for the bias by using a trick:
 - A bias is exactly equivalent to a weight on an extra input line that always has an activity of 1.

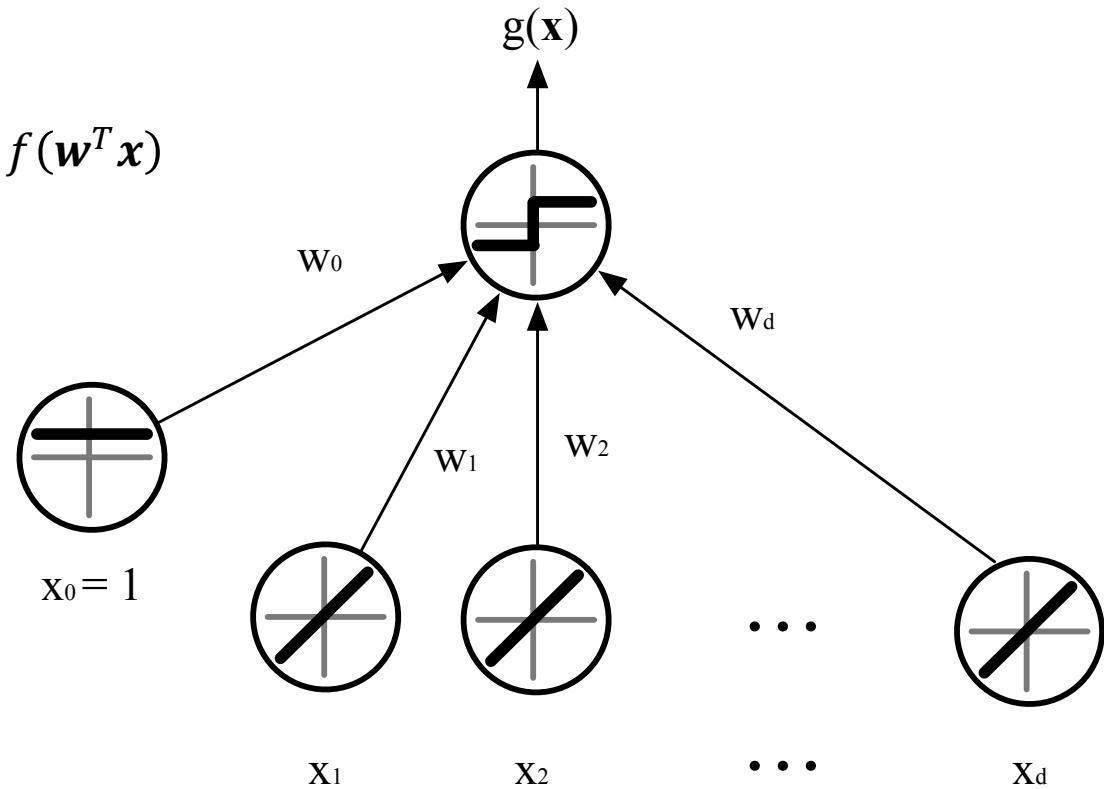
$$\hat{y} = w_0 + \sum_{i=1} w_i$$



Perceptron

$$g(\mathbf{x}) = f\left(\sum_{i=1}^d x_i w_i + w_0\right) = f(\mathbf{w}^T \mathbf{x})$$

$$f(s) = \begin{cases} 1, & \text{if } s \geq 0 \\ -1, & \text{if } s < 0 \end{cases}$$



Linear classifier!

Perceptron

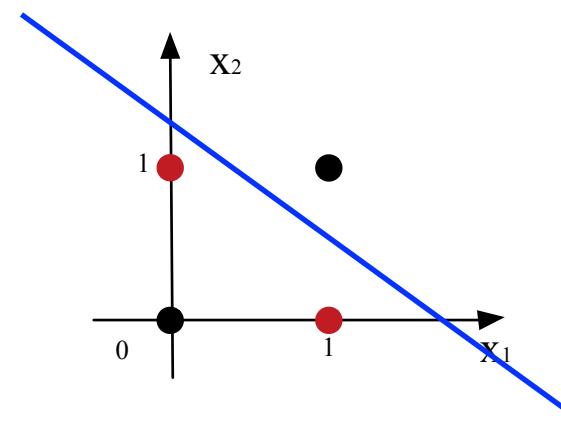
Marvin Minsky & Seymour Papert (1969). *Perceptrons*, MIT Press, Cambridge, MA.

- Before long researchers had begun to discover the Perceptron's limitations.
- Unless input categories were “linearly separable”, a perceptron could not learn to discriminate between them.
- Unfortunately, it appeared that many important categories were not linearly separable.
- E.g., those inputs to an XOR gate that give an output of 1 (namely 10 & 01) are not linearly separable from those that do not (00 & 11).

Perceptron: Limitations

- Solving XOR (exclusive-or) with Perceptron?

| x_1 | x_2 | y |
|-------|-------|-----|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |



- Linear classifier cannot identify non-linear patterns.

$$w_1x_1 + w_2x_2 + b$$

This failure caused the majority of researchers to walk away.

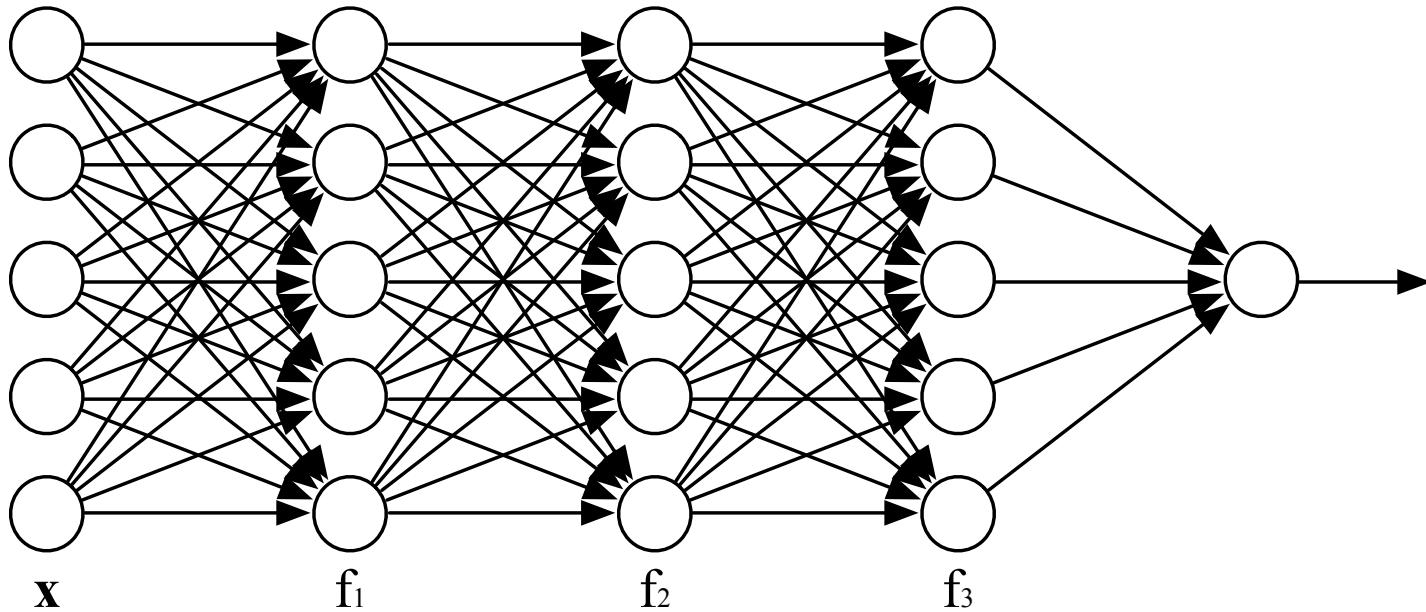
Breakthrough: Multi-Layer Perceptron

- In 1986, Geoffrey Hinton, David Rumelhart, and Ronald Williams published a paper “*Learning representations by back-propagating errors*”, which introduced
 - **Backpropagation**, a procedure to *repeatedly adjust the weights* so as to minimize the difference between actual output and desired output
 - **Hidden Layers**, which are *neuron nodes stacked in between inputs and outputs*, allowing neural networks to learn more complicated features (such as XOR logic)

Outline

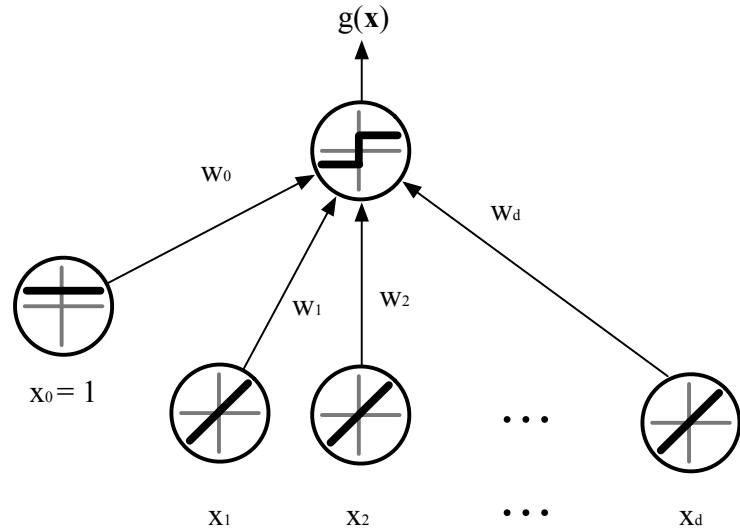
- Perceptron
- **Multilayer Neural Network**
- Backpropagation
- Activation Functions
- Stochastic Gradient Descent

Multilayer Neural Networks

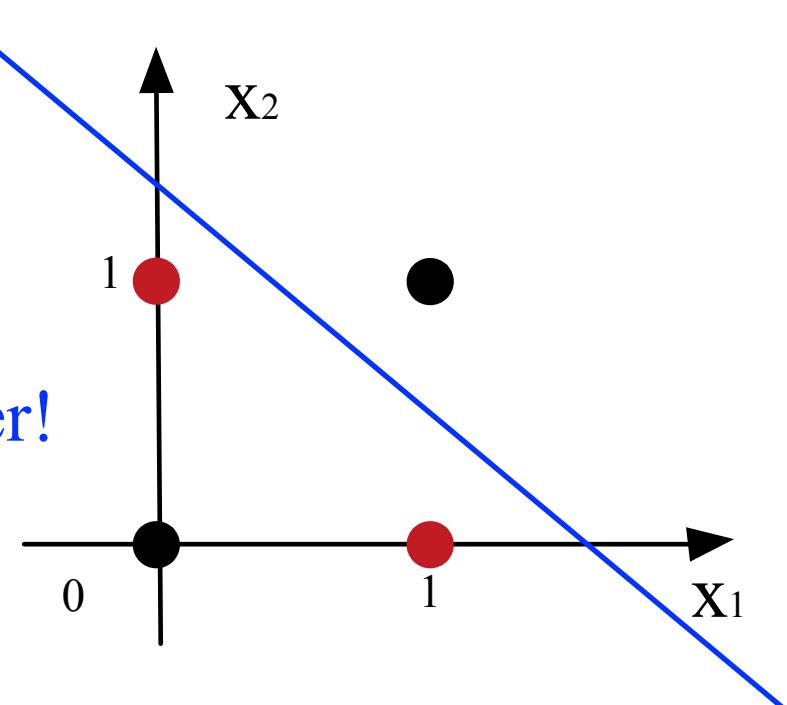


- Function composition can be described by a directed acyclic graph (hence feedforward networks)
- x is the first layer, f_1 is the second layer and so on.
- Depth is the maximum i in the function composition chain
- Final layer is called the output layer

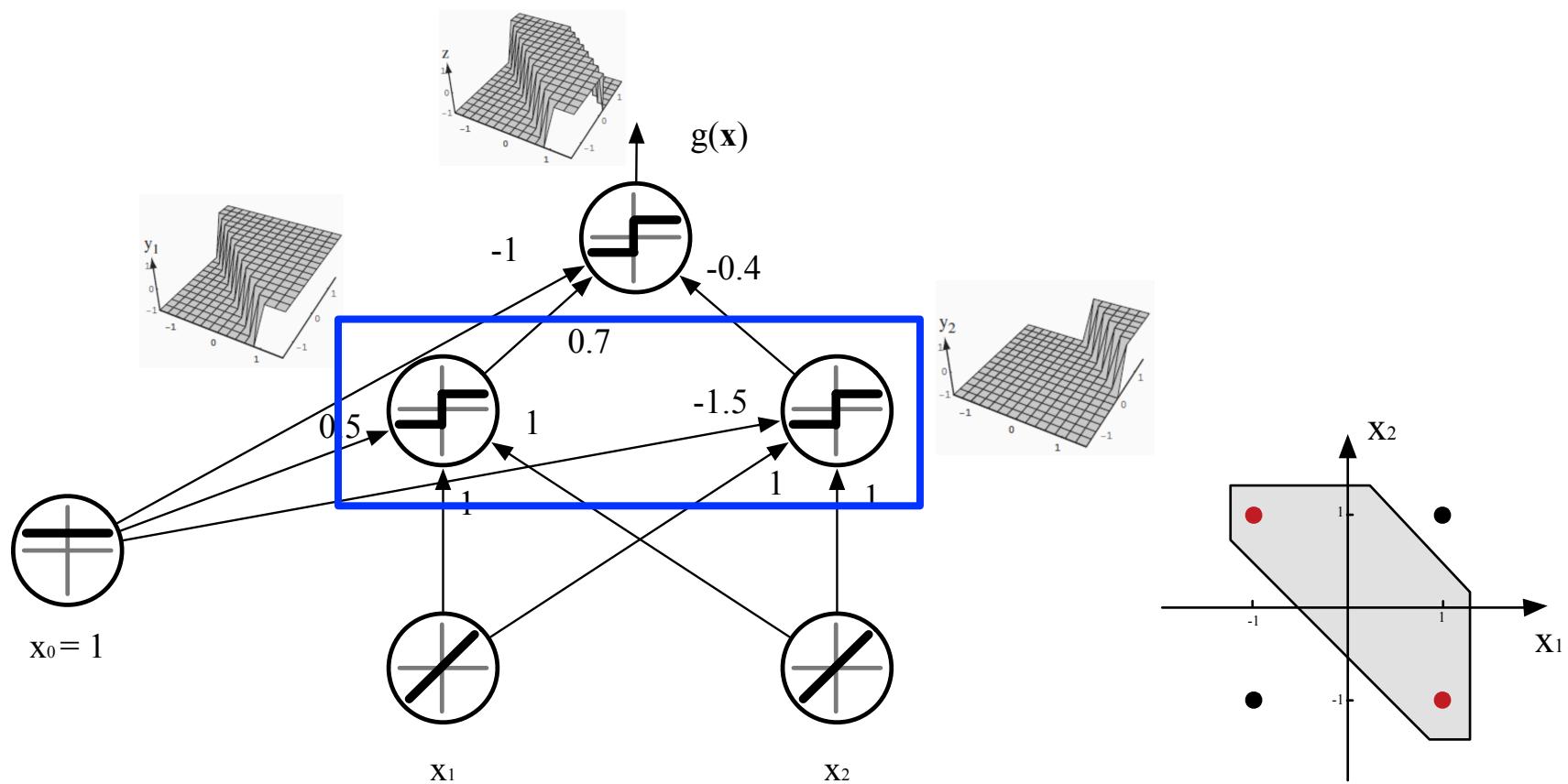
Two-layer Neural Network



Linear classifier!

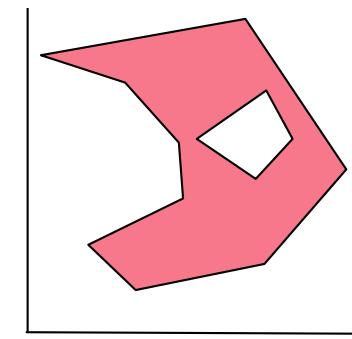
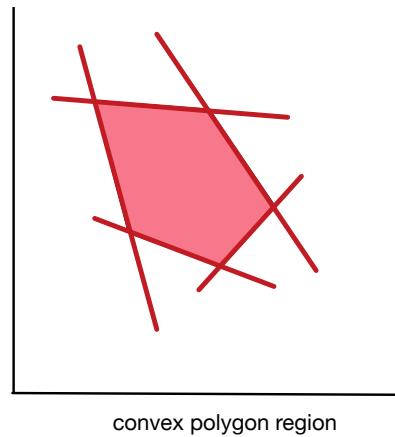
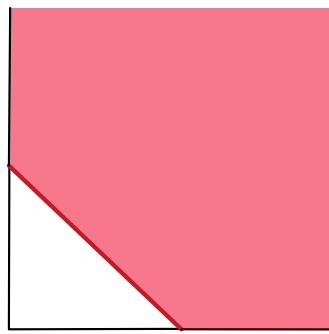
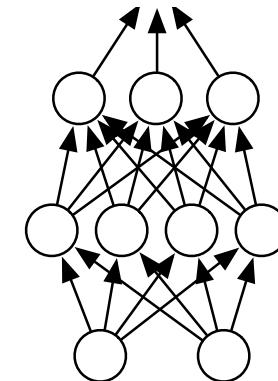
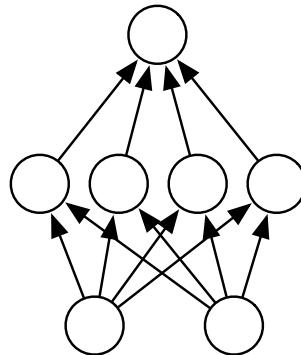
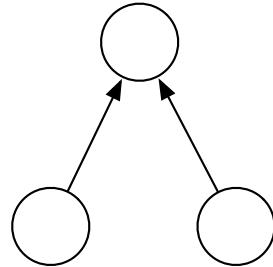


Three-layer Neural Network (with a hidden layer)

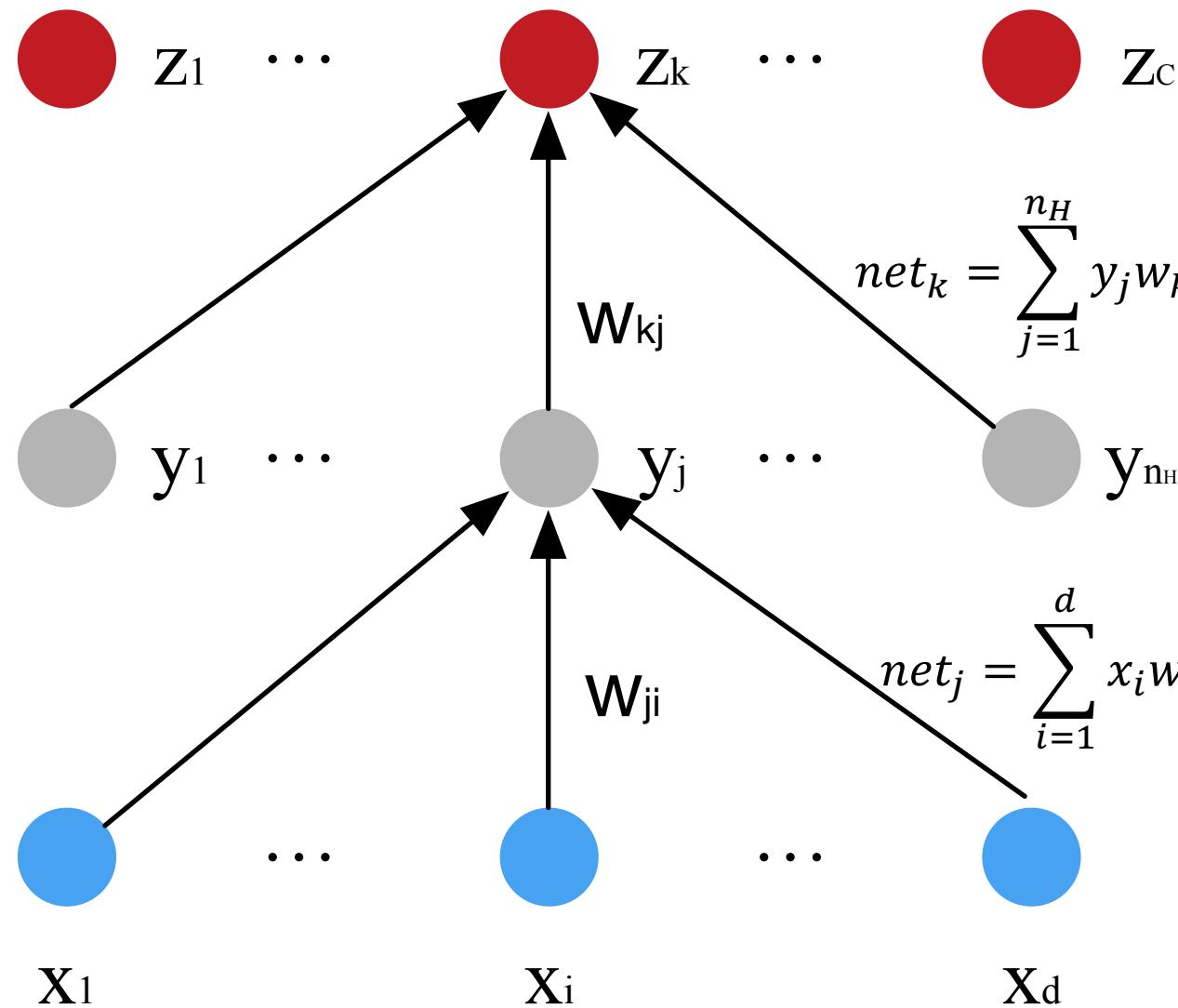


Three-layer Neural Network (with a hidden layer)

- A 2D-input example



Feedforward



$$z_k = f(\text{net}_k)$$

$$= f\left(\sum_{j=1}^{n_H} w_{kj} f(\text{net}_j) + w_{k0}\right)$$

$$Z_C$$

$$\text{net}_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \mathbf{w}_k^T \mathbf{y}$$

$$\text{net}_j = \sum_{i=1}^d x_i w_{ji} + w_{j0} = \sum_{i=0}^d x_i w_{ji} = \mathbf{w}_j^T \mathbf{x}$$

Universal Approximation Theorem

Let $f(\cdot)$ be a **nonconstant, bounded, and monotonically-increasing continuous** function. Let I_m denote the m -dimensional unit hypercube $[0,1]^m$. The space of continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N , real constants $v_i, b_i \in \mathbb{R}^m$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \dots, N$, such that we may define:

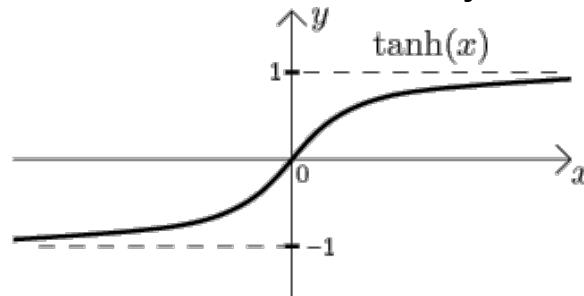
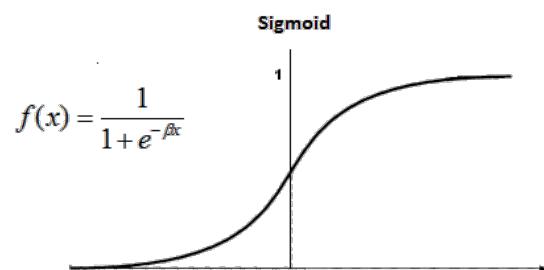
$$\hat{F}(x) = \sum_i^N v_i f(w_i^T x + b_i)$$

as an approximate realization of the function F where F is independent of f ; that is,

$$|\hat{F}(x) - F(x)| < \varepsilon$$

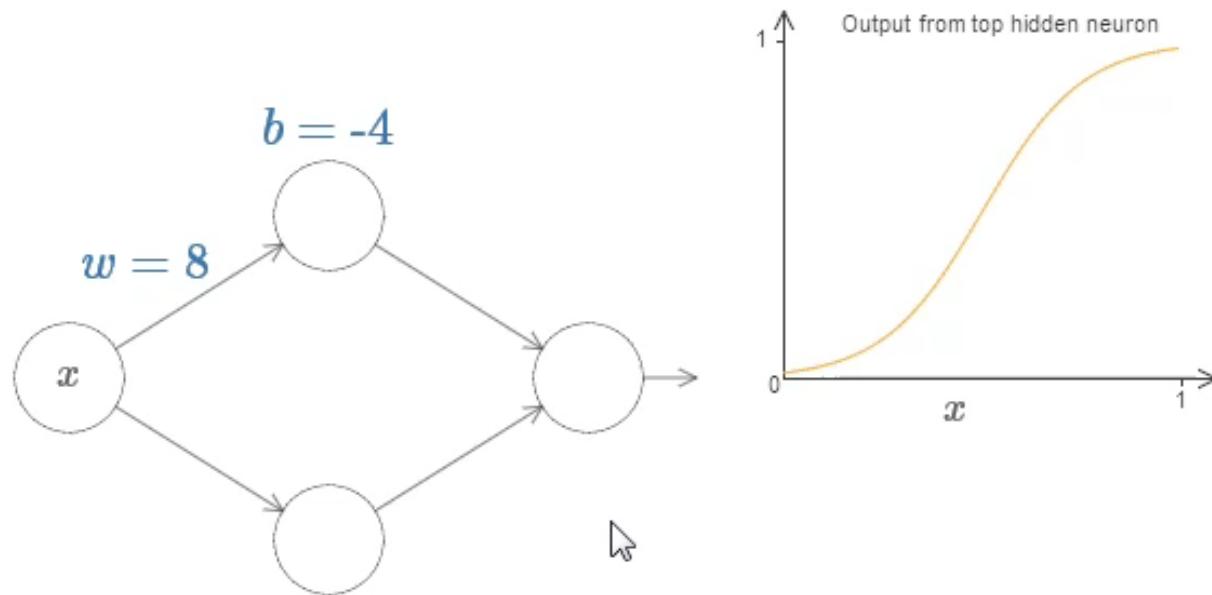
for all $x \in I_m$. In other words, functions of the form $\hat{F}(x)$ are dense in $C(I_m)$.

Universality theorem (Hecht-Nielsen 1989): “Neural networks with a single hidden layer can be used to approximate any continuous function to any desired precision.”



Expressive power of a three-layer neural network

- A visual explanation on one input and one output functions



Video credit to: <https://neuralnetworksanddeeplearning.com>

- The weight changes the shape of the graph.
- The larger the weight, the steeper the graph.
- The bias moves the graph, but doesn't change its shape.

Expressive power of a three-layer neural network

- A visual explanation on one input and one output functions

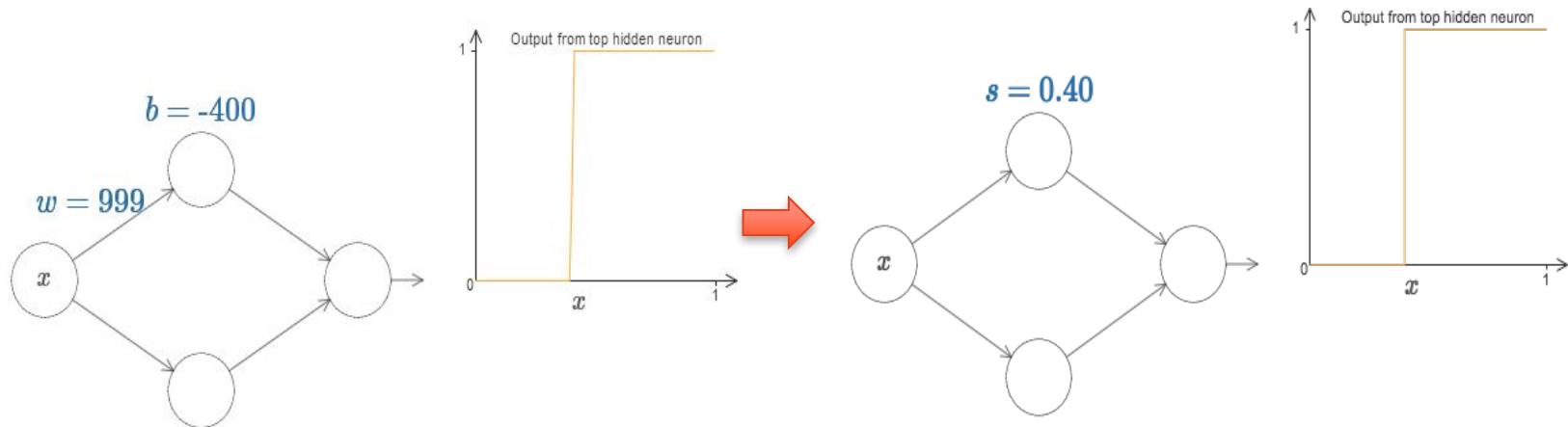


Image credit to: <https://neuralnetworksanddeeplearning.com>

- Increase the weight so that the output is a very good approximation of a step function.
- The step is at position $s = -b/w$.
- We simplify the problem by describing hidden neurons using just a single parameter s .

Expressive power of a three-layer neural network

- A visual explanation on one input and one output functions

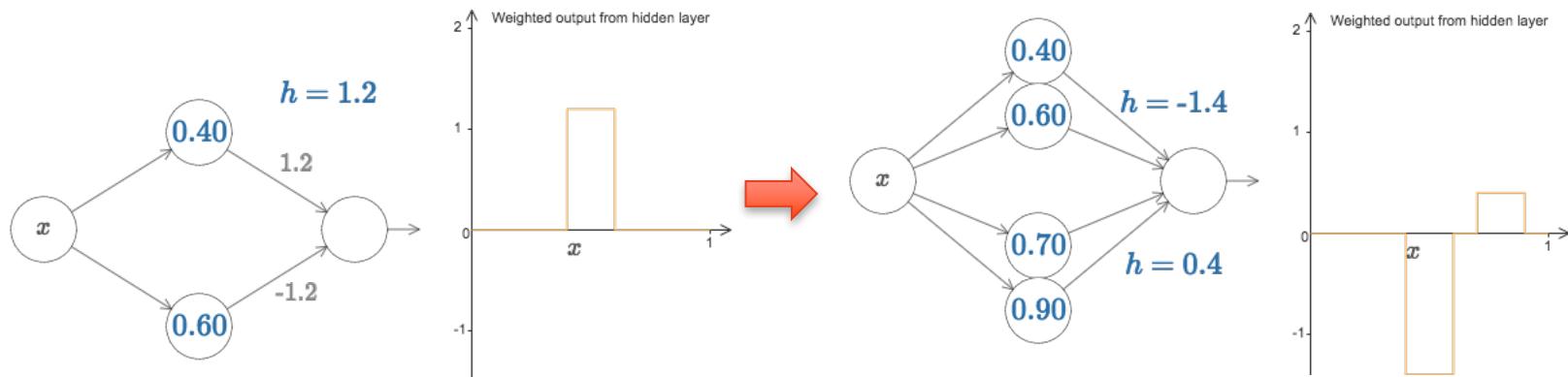


Image credit to: <https://neuralnetworksanddeeplearning.com>

- *Left:* Setting the weights of output to 1.2 and -1.2 , we get a ‘bump’ function, which starts at 0.4, ends at 0.6 and has height 1.2.
- *Right:* By adding another pair of hidden neurons, we can get more bumps in the same network.

Expressive power of a three-layer neural network

- A visual explanation on one input and one output functions

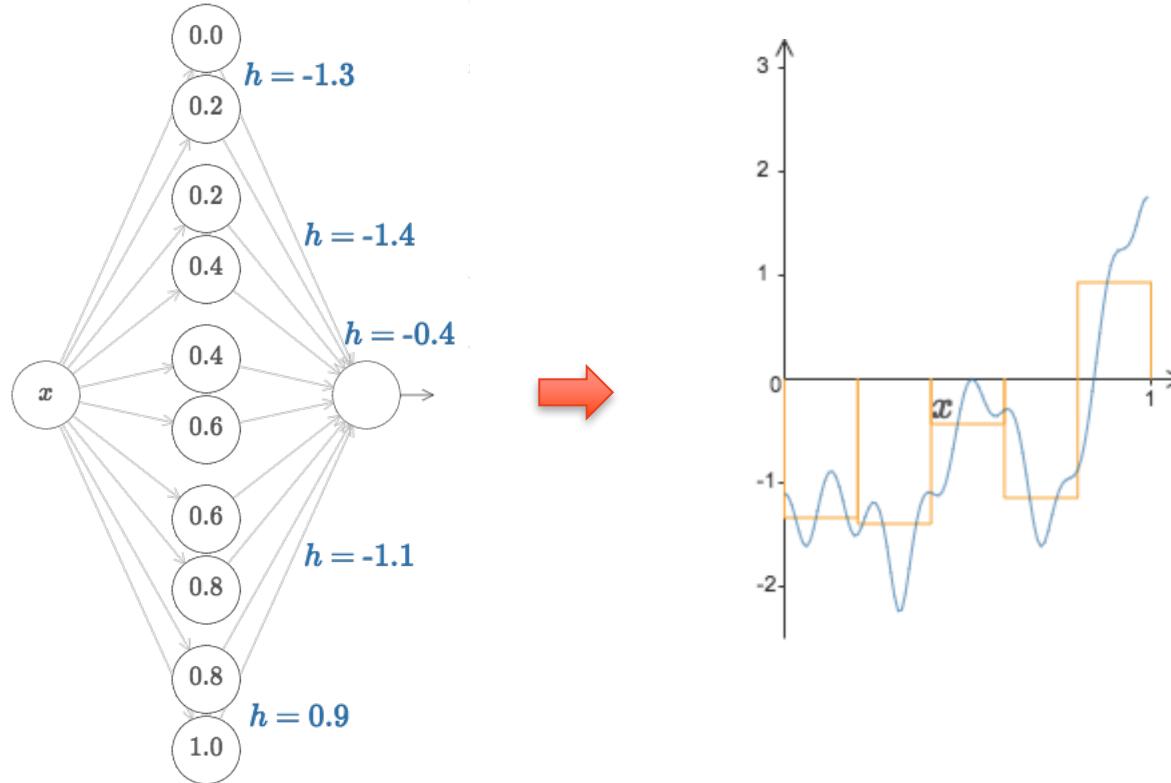
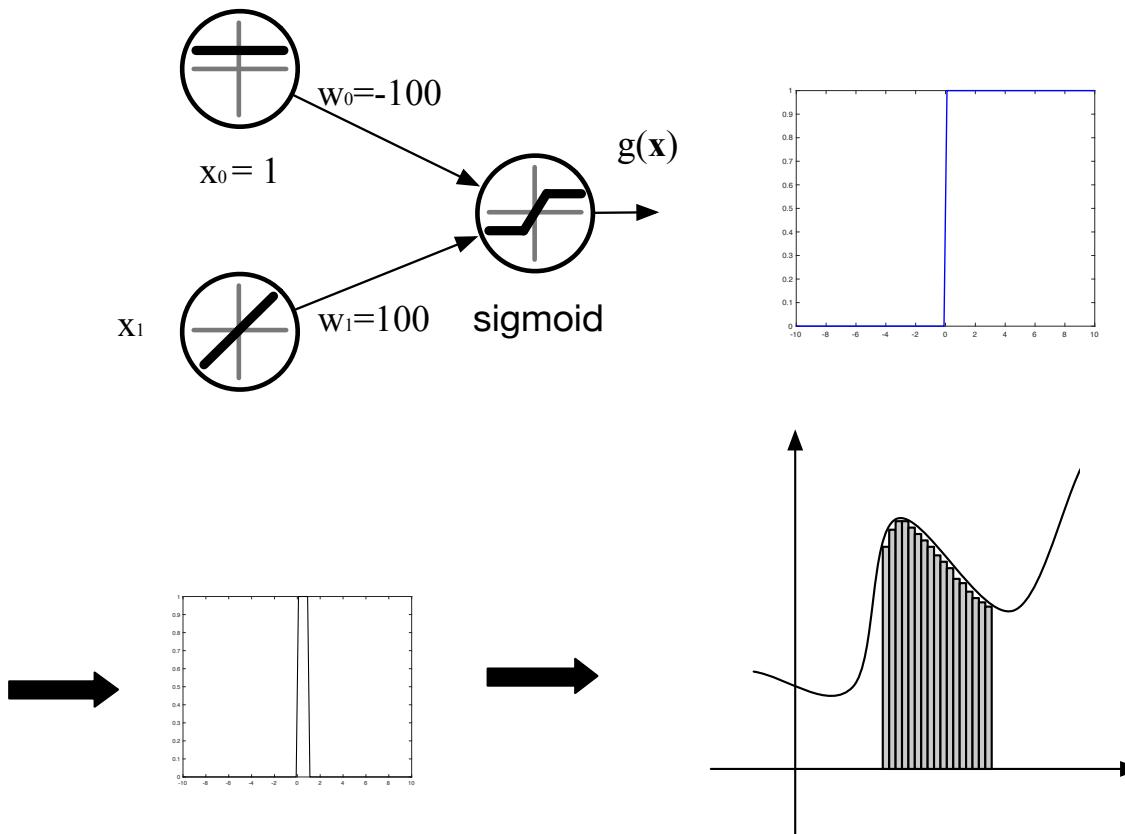


Image credit to: <https://neuralnetworksanddeeplearning.com>

- By increasing the number of hidden neurons, we can make the approximation much better.

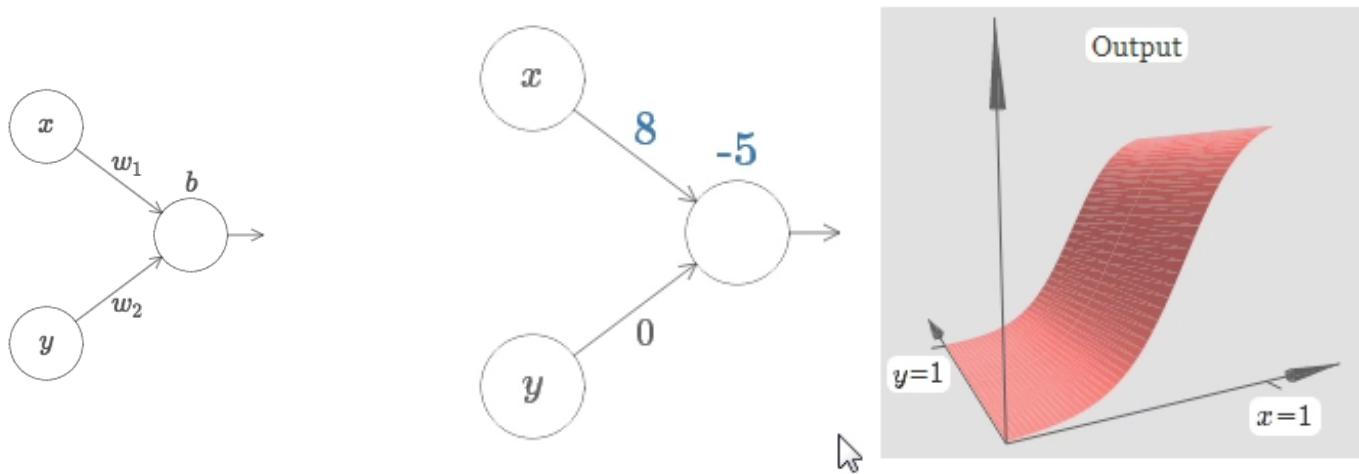
Expressive power of a three-layer neural network

- Overall procedure



Expressive power of a three-layer neural network

- How about multiple input functions? The two input case:
Fix $w_2 = 0$, change w_1 and b :

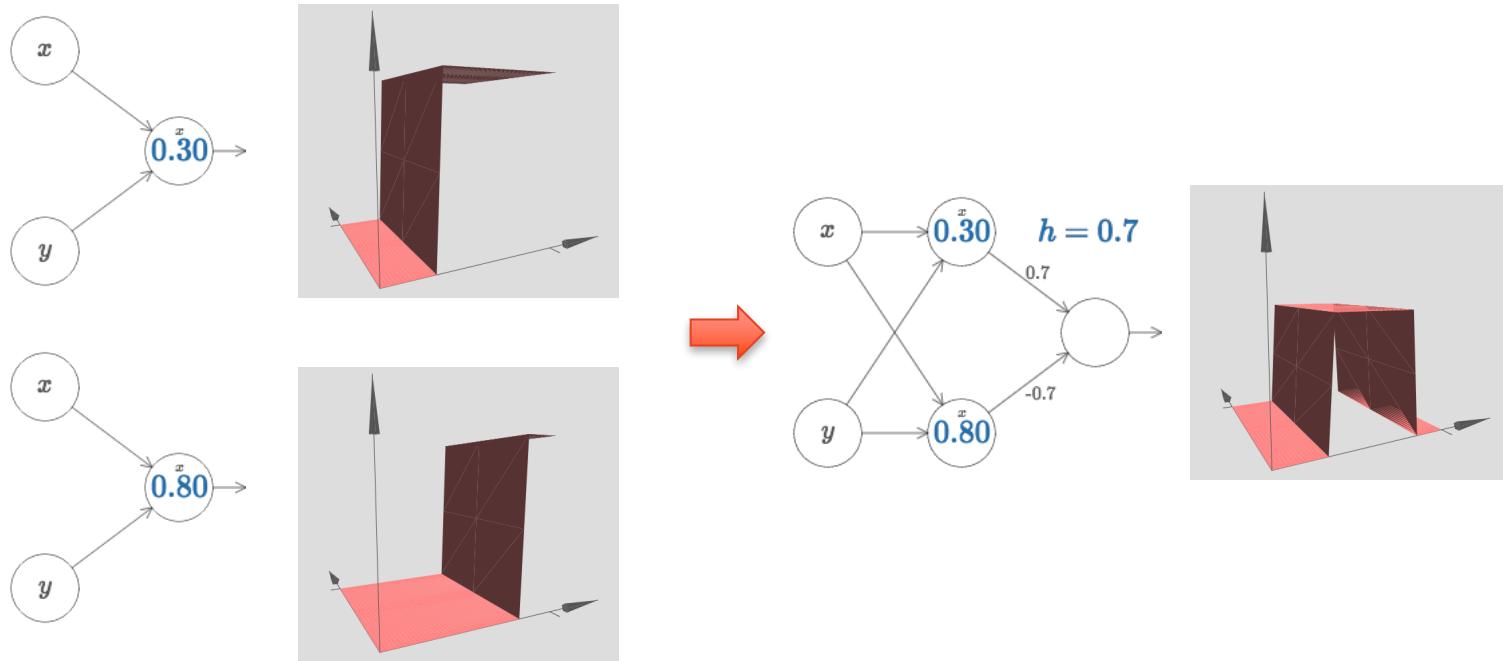


Video credit to: <https://neuralnetworksanddeeplearning.com>

- Similar to one input case, the weight changes the shape and the bias changes the position.
- The step point: $s_x = -b/w_1$

Expressive power of a three-layer neural network

- How about multiple input functions? The two input case:



Video credit to: <https://neuralnetworksanddeeplearning.com>

Expressive power of a three-layer neural network

- How about multiple input functions? The two input case:

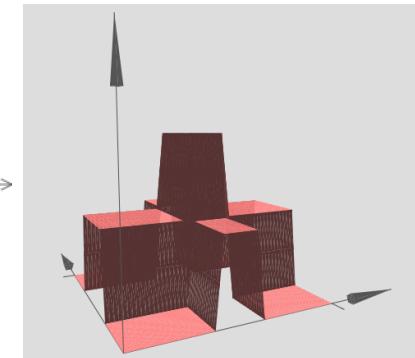
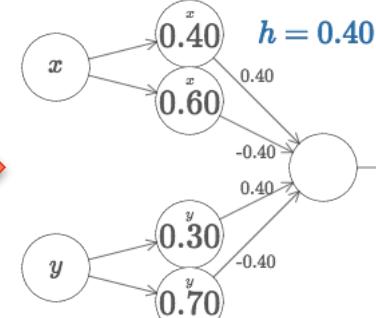
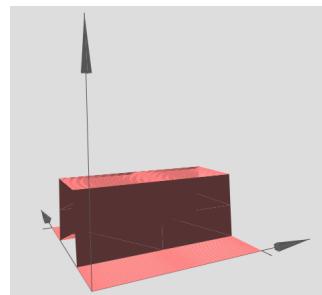
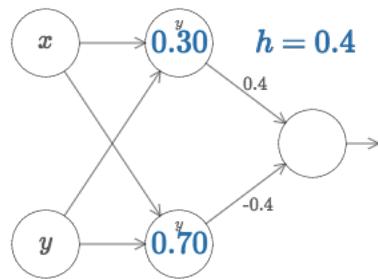
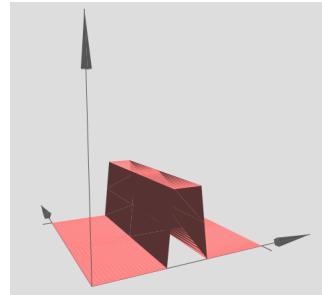
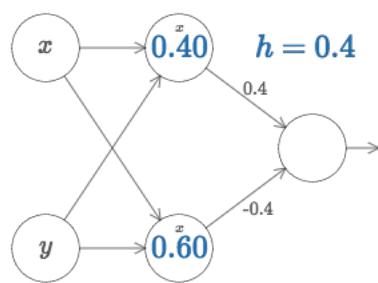


Image credit to: <https://neuralnetworksanddeeplearning.com>

Expressive power of a three-layer neural network

- How about multiple input functions? The two input case:

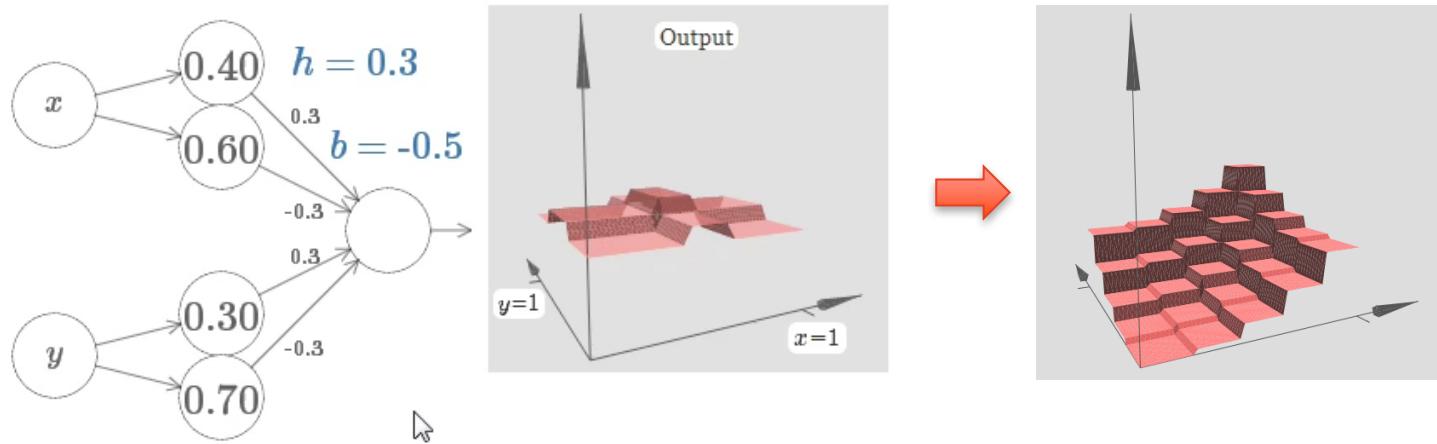


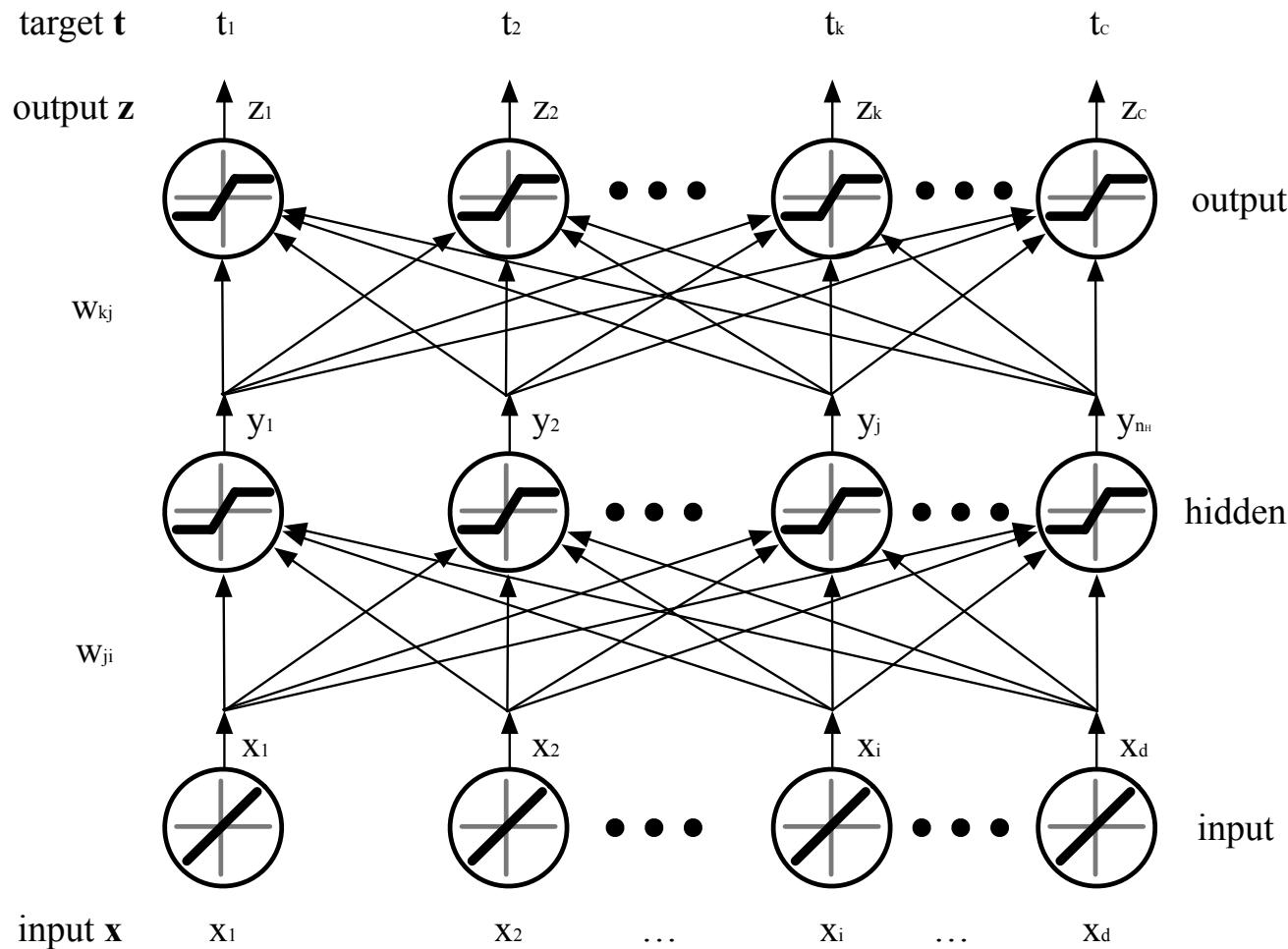
Image credit to: <https://neuralnetworksanddeeplearning.com>

- By adding **bias** to the output neuron and adjusting the value of h and b , we can construct a ‘tower’ function.
- The same idea can be used to compute as many towers as we like. We can also make them as thin as we like, and whatever height we like. As a result, we can ensure that the weighted output from the second hidden layer approximates any desired function of two variables.

Outline

- Perceptron
- Multilayer Neural Network
- **Backpropagation**
- Activation Functions
- Stochastic Gradient Descent

A three-layer neural network for illustration



$$z_k = f(\text{net}_k)$$

$$\text{net}_k = \sum_{j=1}^{n_h} y_j w_{kj} + w_{k0}$$

$$y_j = f(\text{net}_j)$$

$$\text{net}_j = \sum_{i=1}^d x_i w_{ji} + w_{j0}$$

Training error

- Euclidean distance.

$$J(t, z) = \frac{1}{2} \sum_{k=1}^C (t_k - z_k)^2 = \frac{1}{2} \|\mathbf{t} - \mathbf{z}\|^2$$

- If both $\{t_k\}$ and $\{z_k\}$ are probability distributions, we can use cross entropy.

$$J(t, z) = - \sum_{k=1}^C t_k \log z_k$$

- Cross entropy is asymmetric. In some cases we may use this symmetric form.

$$J(t, z) = - \sum_{k=1}^C (t_k \log z_k + z_k \log t_k)$$

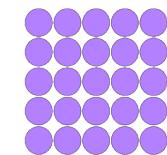
Cross Entropy Loss

- Given two probability distributions t and z ,

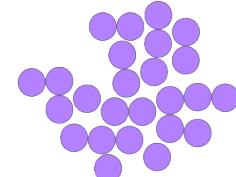
$$\begin{aligned} \text{CrossEntropy}(t, z) &= - \sum_i t_i \log z_i \\ &= - \sum_i t_i \log t_i + \sum_i t_i \log t_i - \sum_i t_i \log z_i \\ &= - \sum_i t_i \log t_i + \sum_i t_i \log \frac{t_i}{z_i} \\ &= \text{Entropy}(t) + D_{KL}(t|z) \end{aligned}$$

Entropy

is a measure of degree of disorder or randomness.



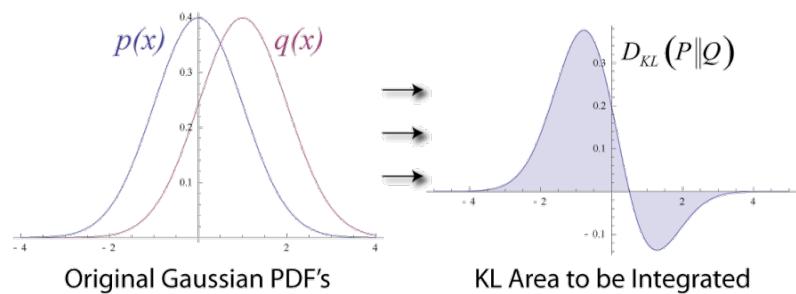
Low Entropy



High Entropy

KL divergence

is a measure of the information lost when Z is used to approximate T



Gradient descent

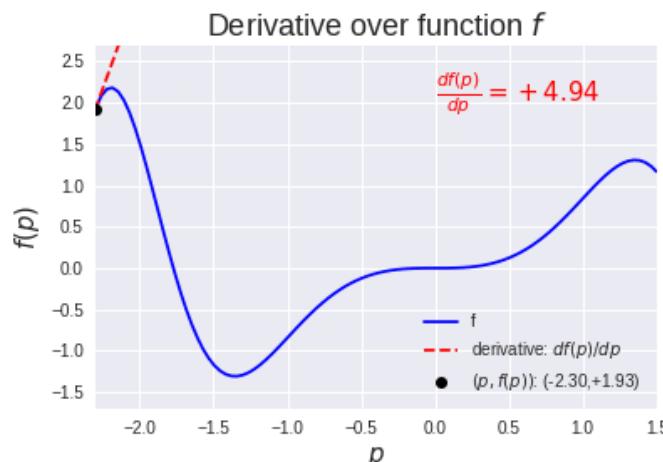
- Weights are initialized with random values, and then are updated in a direction reducing the error.

$$\Delta \mathbf{w} = -\eta \frac{\partial J}{\partial \mathbf{w}}$$

where η is the learning rate.

Iteratively update

$$\mathbf{w}(t + 1) = \mathbf{w}(t) + \Delta \mathbf{w}(t)$$

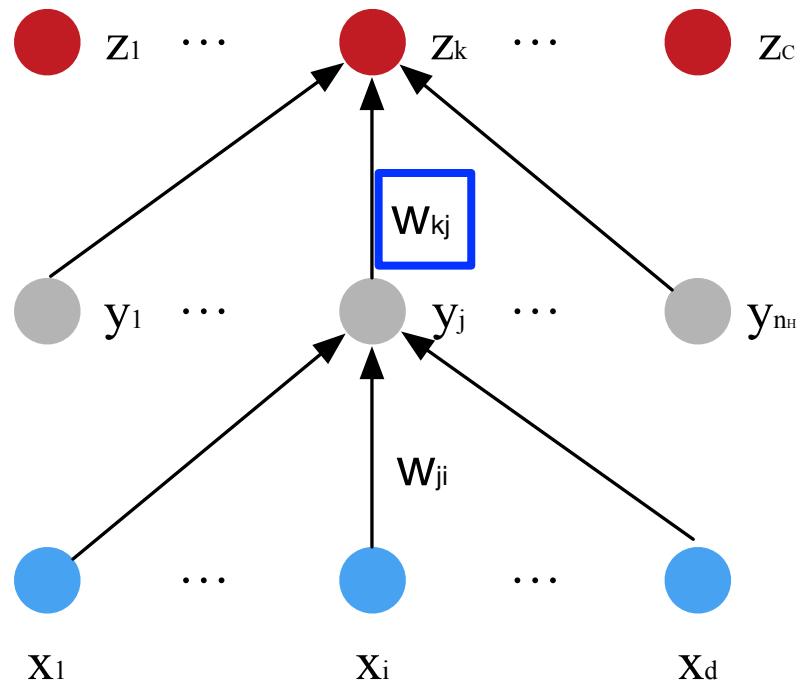


Hidden-to-output weight w_{kj}

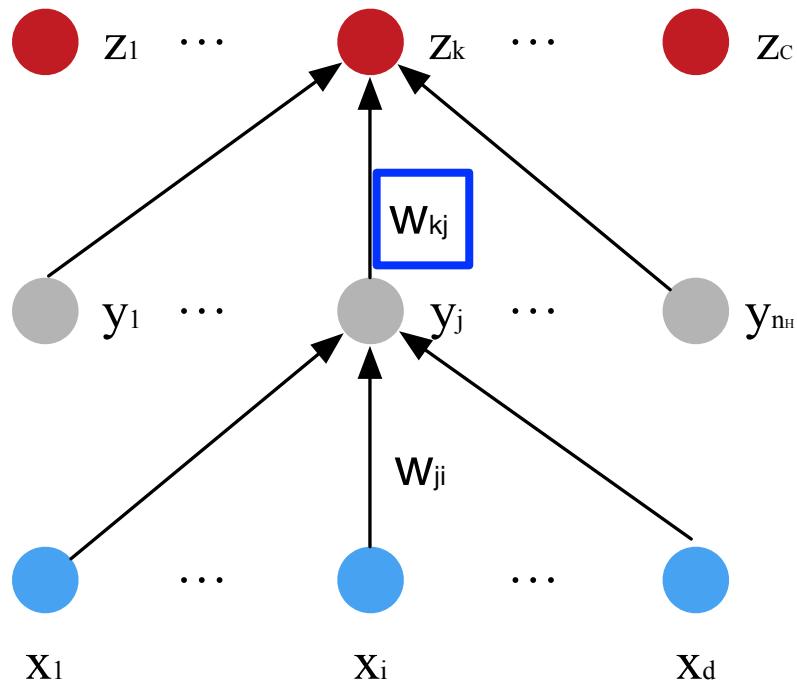
- Chain rule

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} \frac{\partial net_k}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} y_j$$

$$\begin{cases} J(\mathbf{w}) = \frac{1}{2} \|\mathbf{t} - \mathbf{z}\|^2 \\ z_k = f(net_k) \\ net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} \end{cases}$$



Hidden-to-output weight w_{kj}



$$\left\{ \begin{array}{l} J(\mathbf{w}) = \frac{1}{2} \|\mathbf{t} - \mathbf{z}\|^2 \\ z_k = f(\text{net}_k) \\ \text{net}_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} \end{array} \right.$$

- Define the *sensitivity* of unit k

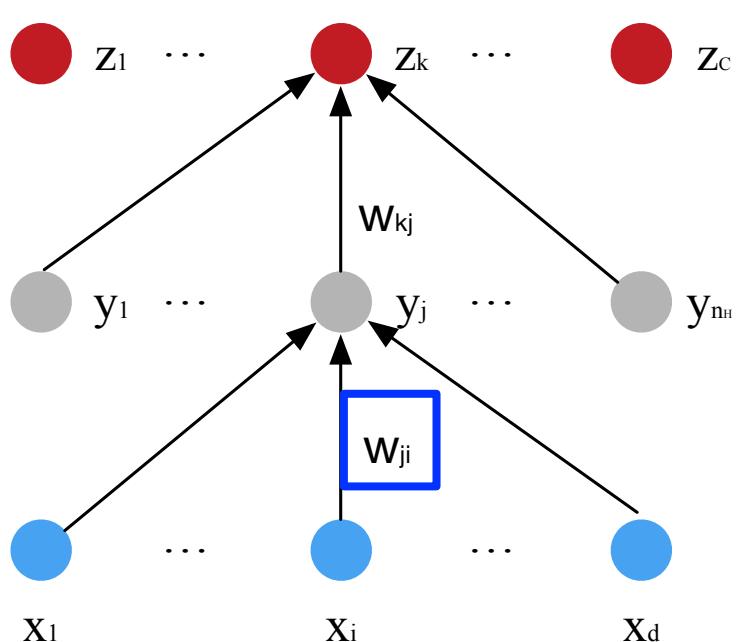
$$\delta_k = -\frac{\partial J}{\partial \text{net}_k} = -\frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial \text{net}_k} = (t_k - z_k) f'(\text{net}_k)$$

- **Update w_{kj}**

$$\Delta w_{kj} = -\eta \frac{\partial J}{\partial \text{net}_k} y_j = \eta \delta_k y_j$$

Input-to-hidden weight w_{ji}

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{w_{ji}}$$



$$\frac{\partial J}{\partial y_j} = \sum_{k=1}^C \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial y_j} = - \sum_{k=1}^C (t_k - z_k) \frac{\partial z_k}{\partial y_j}$$

$$= - \sum_{k=1}^C (t_k - z_k) \frac{\partial z_k}{\partial net_k} \frac{\partial net_k}{\partial y_j}$$

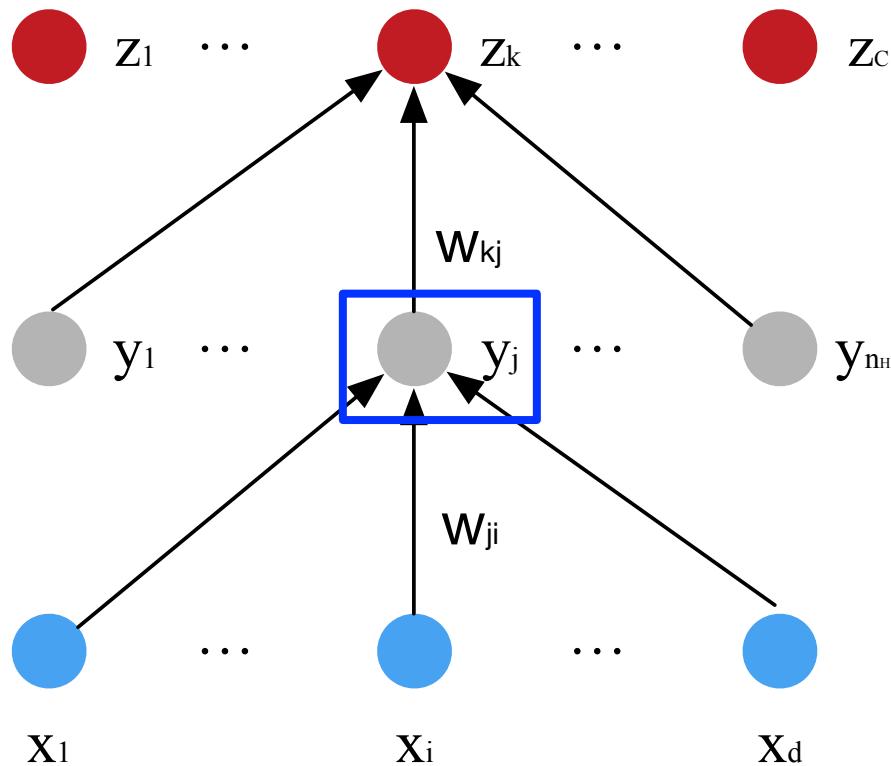
$$= - \sum_{k=1}^C (t_k - z_k) f'(net_k) w_{kj}$$

$$= - \sum_{k=1}^C \delta_k w_{kj}$$

Input-to-hidden weight w_{ji}

- Sensitivity for a hidden unit j

$$\delta_j = -\frac{\partial J}{\partial net_j} = -\frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial net_j} = f'(net_j) \sum_{k=1}^C \delta_k w_{kj}$$



The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units.

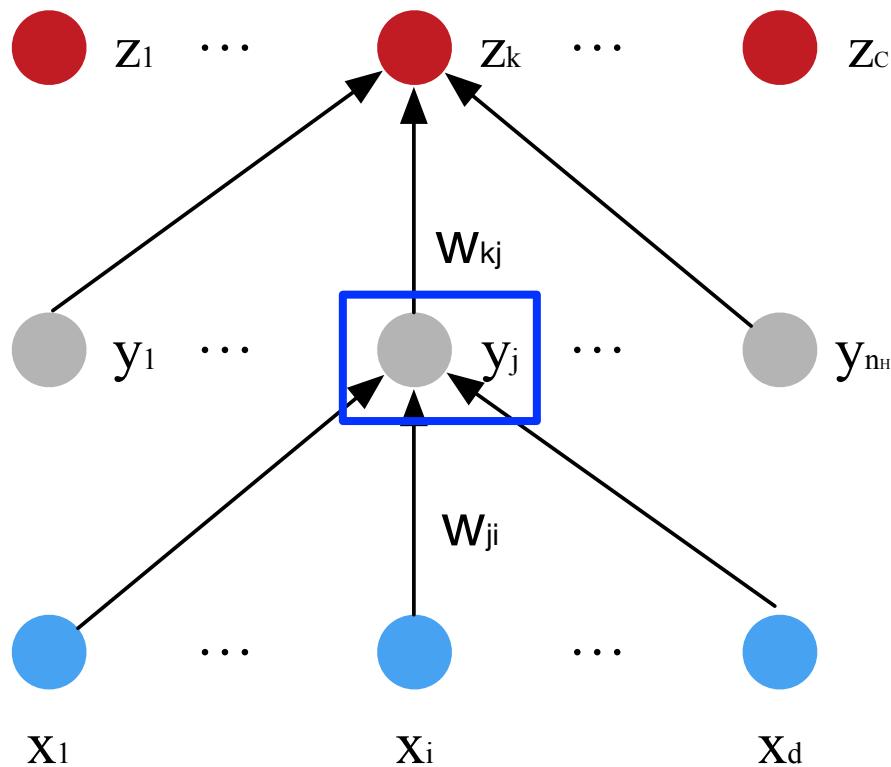
The output unit sensitivities are propagated “back” to the hidden units.

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$

Input-to-hidden weight w_{ji}

- Sensitivity for a hidden unit j

$$\delta_j = -\frac{\partial J}{\partial \text{net}_j} = -\frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial \text{net}_j} = f'(\text{net}_j) \sum_{k=1}^C \delta_k w_{kj}$$



Update w_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial J}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}} = \eta \delta_j x_i$$

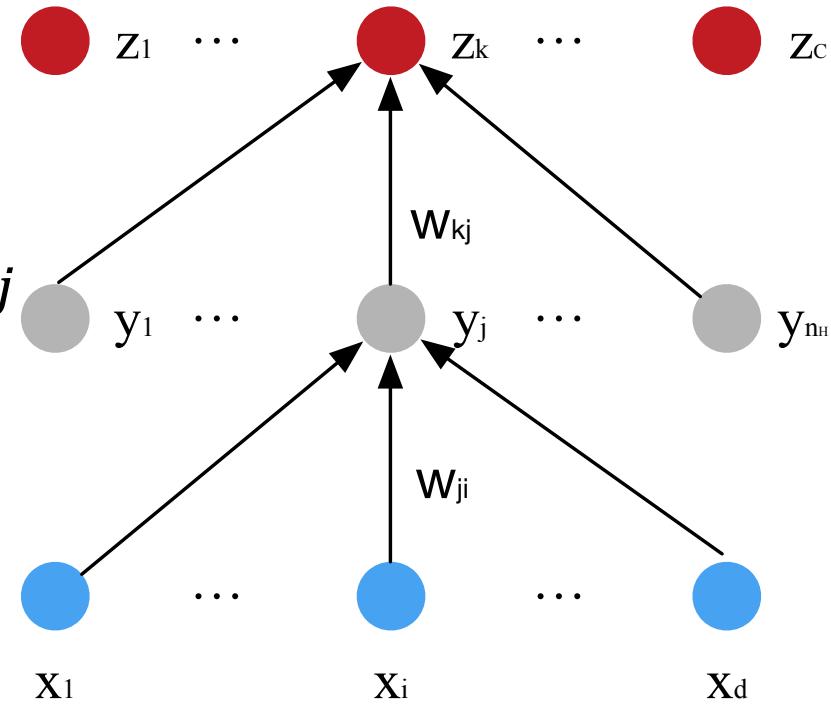
$$\left(\text{net}_j = \sum_{i=1}^d x_i w_{ji} + w_{j0} = \sum_{i=0}^d x_i w_{ji} = \mathbf{w}_j^T \mathbf{x} \right)$$

Backpropagation summary

- Hidden-to-output weight

$$\Delta w_{kj} = \eta \delta_k y_j$$

$$= \eta(t_k - z_k)f'(net_k)y_j$$



- Input-to-hidden

$$\Delta w_{ji} = \eta \delta_j x_i$$

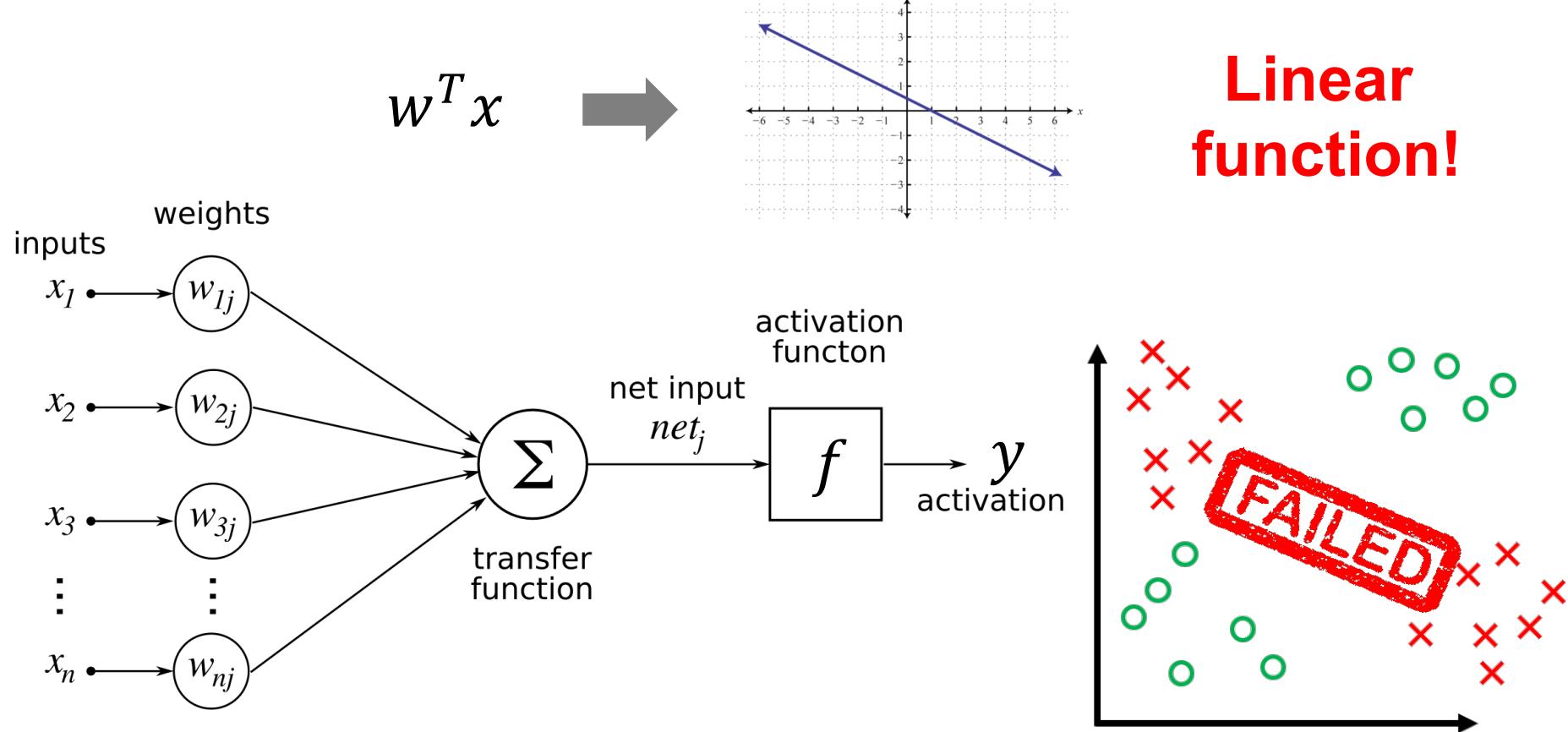
$$= \eta \left[\sum_{k=1}^C \delta_k w_{kj} \right] f'(net_j) x_i$$

Outline

- Perceptron
- Multilayer Neural Network
- Backpropagation
- **Activation Functions**
- Stochastic Gradient Descent

Desirable properties of activation functions

- Must be **nonlinear**: otherwise it is equivalent to a linear classifier



Desirable properties of activation functions

- Bounded outputs:

Desirable property when the output is meant to represent a probability



Probability:

0.7

0.05

0.1

0.1

0.05



APPLES



BANANA



ORANGE



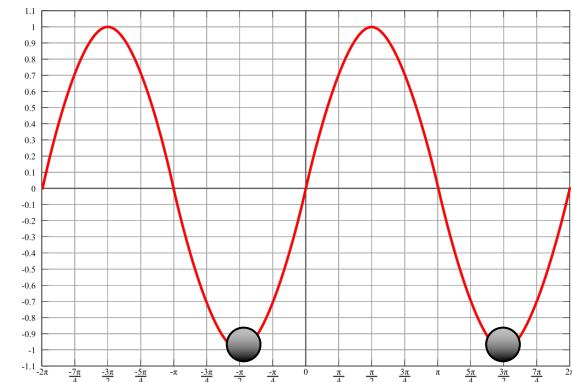
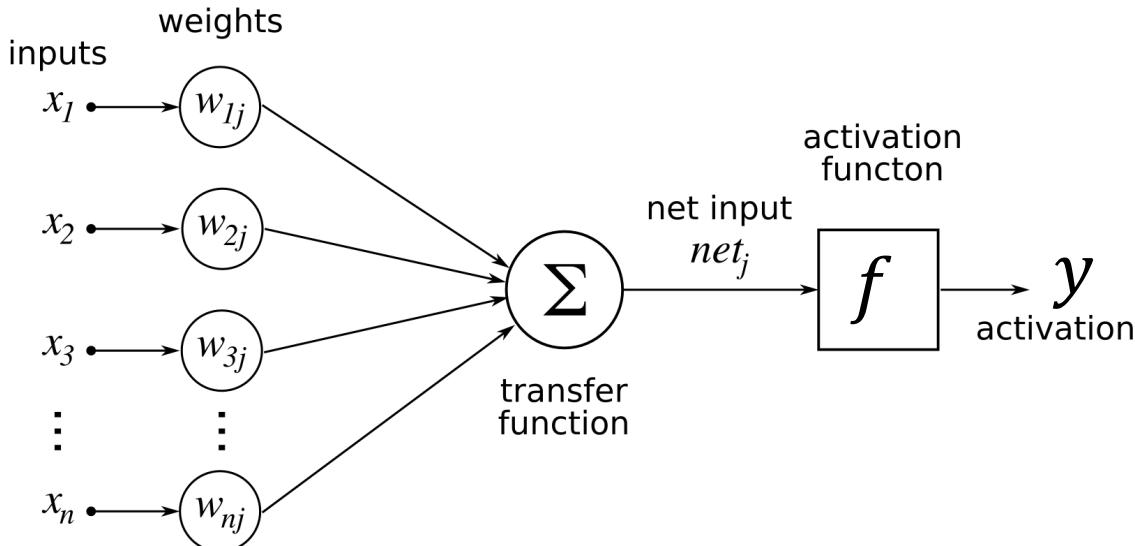
PEACH



PINEAPPLE

Desirable properties of activation functions

- Continuous and differentiable **almost everywhere**
- **Monotonicity**: otherwise it introduces additional local extrema in the error surface

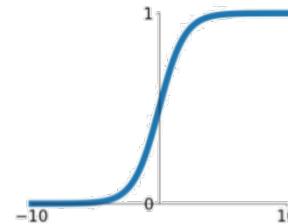


An output value might correspond to multiple input values.

Activation function $f(s)$

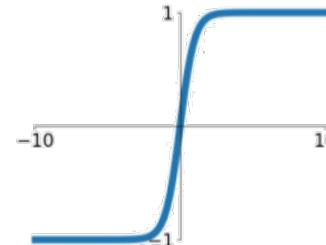
- Popular choice of activation functions (single input)
 - Sigmoid function

$$f(s) = \frac{1}{1 + e^{-s}}$$



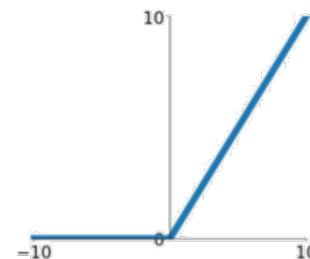
- Tanh function: shift the center of Sigmoid to the origin

$$f(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$$



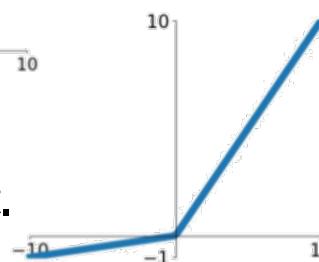
- Rectified linear unit (ReLU)

$$f(s) = \max(0, s)$$



- Leaky ReLU

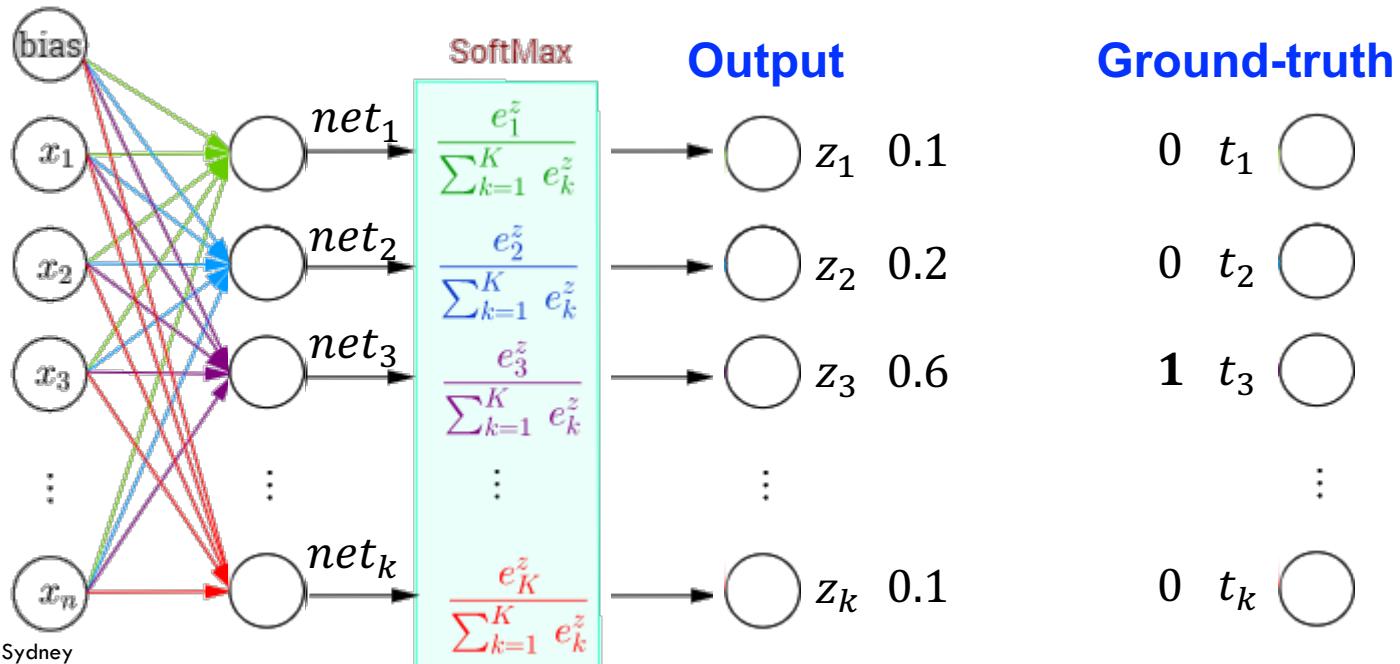
$$f(s) = \begin{cases} s, & \text{if } s \geq 0 \\ \alpha s, & \text{if } s < 0 \end{cases}, \quad \alpha \text{ is a small constant.}$$



Softmax Activation Function

- ❖ In multi-class classification, **softmax activation function** before the output layer is to assign conditional probabilities (given x) to each one of the K classes.

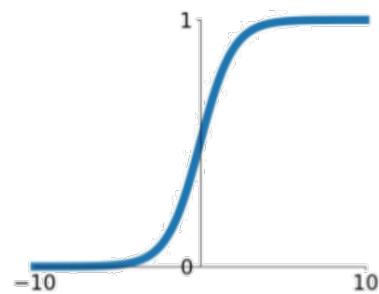
$$\hat{P}(\text{class}_k|x) = z_k = \frac{e^{net_k}}{\sum_{i=1}^K e^{net_i}}$$



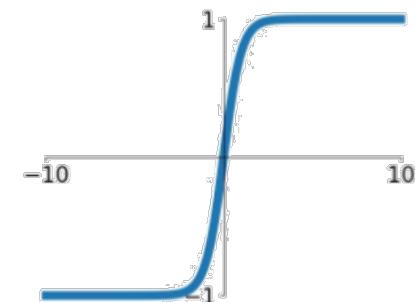
Vanishing gradient problems

- Saturation.

$$f(s) = \frac{1}{1 + e^{-s}}$$



$$f(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$$

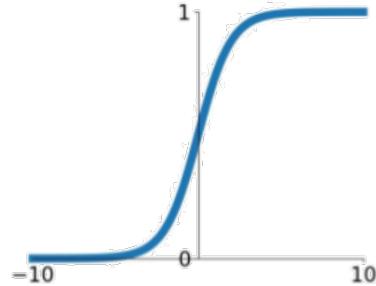


- ❖ At their extremes, their derivative are close to 0, which kill gradient and learning process

Not zero centered activation

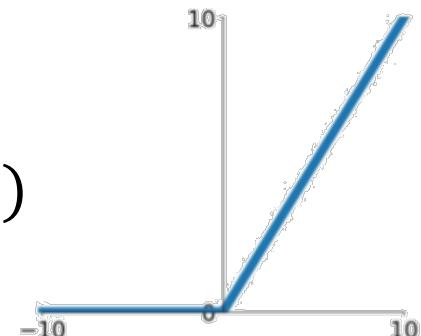
Sigmoid function

$$f(s) = \frac{1}{1 + e^{-s}}$$



Relu function

$$f(s) = \max(0, s)$$



- ❖ During training, all gradients will be all positive or all negative that will cause problem with learning.

Possible update direction

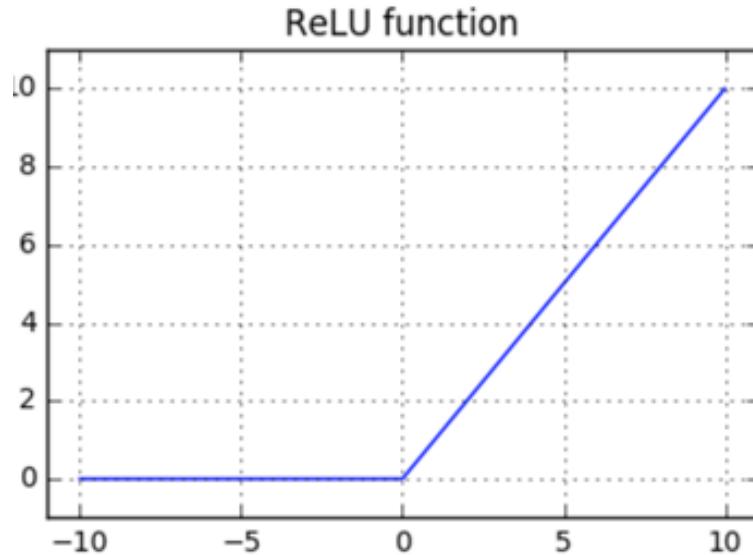
Possible update direction

Optimum point

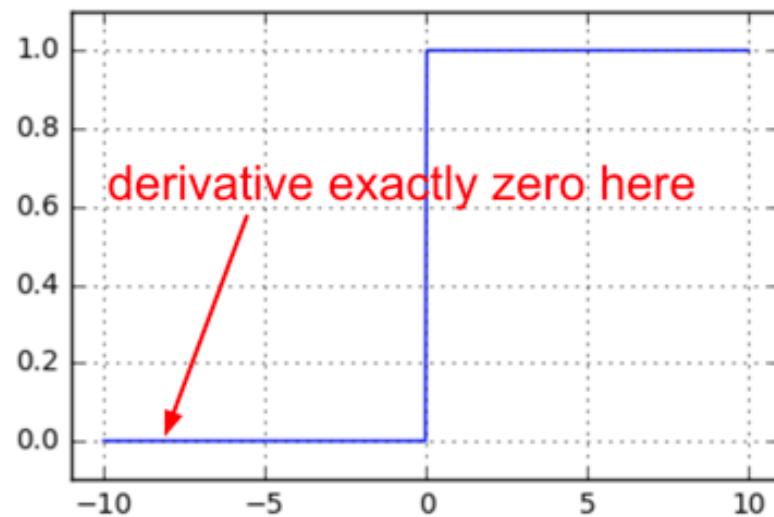
Dead ReLUs

ReLU function

$$f(s) = \max(0, s)$$



derivative of ReLU

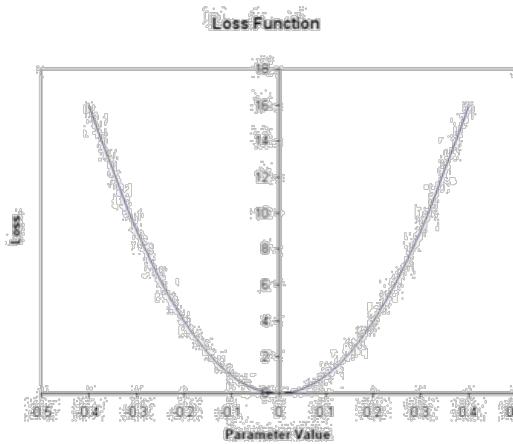


- ❖ For negative numbers, ReLU gives 0, which means some part of neurons will not be activated and be dead.
- ❖ Avoid large learning rate and wrong weight initialization, and try Leaky ReLU, etc.

Example 1

Squared loss function:

$$J(t, z) = \frac{1}{2} \|t - z\|^2$$



$$\frac{\partial J}{\partial z_k} = -(t - z)$$

$$\frac{\partial z_k}{\partial \text{net}_k} = \frac{\partial f(\text{net}_k)}{\partial \text{net}_k}$$

$$= -\frac{-e^{-\text{net}_k}}{(1 + e^{-\text{net}_k})^2}$$

$$= \frac{1 + e^{-\text{net}_k} - 1}{(1 + e^{-\text{net}_k})^2}$$

$$= \frac{1}{1 + e^{-\text{net}_k}} - \frac{1}{(1 + e^{-\text{net}_k})^2}$$

$$= z_k - (z_k)^2$$

- The *sensitivity* of unit k is

$$\delta_k = -\frac{\partial J}{\partial \text{net}_k} = -\frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial \text{net}_k} = (t_k - z_k)z_k(1 - z_k)$$

[Vanishing gradient] When the output z_k approaches to 0 or 1 (*i.e.* saturate), $\frac{\partial z_k}{\partial \text{net}_k}$ gets close to 0 and δ_k is small even if the error $(t_k - z_k)$ is large.

Example 2

Cross entropy loss function:

$$J(t, z) = - \sum_{k=1}^C t_k \log z_k$$

$$\frac{\partial J}{\partial z_k} = \frac{\partial [- \sum_{k=1}^C t_k \log z_k]}{\partial z_k} = - \frac{t_k}{z_k}$$

Softmax activation function:

$$\begin{aligned}\frac{\partial z_k}{\partial \text{net}_k} &= \frac{\partial \left(e^{\text{net}_k} \cdot \frac{1}{\sum_{i=1}^C e^{\text{net}_i}} \right)}{\partial \text{net}_k} \\ &= \frac{\partial (e^{\text{net}_k})}{\partial \text{net}_k} \cdot \frac{1}{\sum_{i=1}^C e^{\text{net}_i}} + \frac{\partial \left(\frac{1}{\sum_{i=1}^C e^{\text{net}_i}} \right)}{\partial \text{net}_k} e^{\text{net}_k} \\ &= \frac{e^{\text{net}_k}}{\sum_{i=1}^C e^{\text{net}_i}} + \frac{e^{\text{net}_k}}{\left(\sum_{i=1}^C e^{\text{net}_i} \right)^2} e^{\text{net}_k} = z_k - z_k^2\end{aligned}$$

- The *sensitivity* of unit k is

$$\delta_k = - \frac{\partial J}{\partial \text{net}_k} = - \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial \text{net}_k} = t_k(1 - z_k)$$

Outline

- Perceptron
- Multilayer Neural Network
- Backpropagation
- Activation Functions
- **Stochastic Gradient Descent**

Stochastic Gradient Descent (SGD)

- Given n training examples, the target function can be expressed as

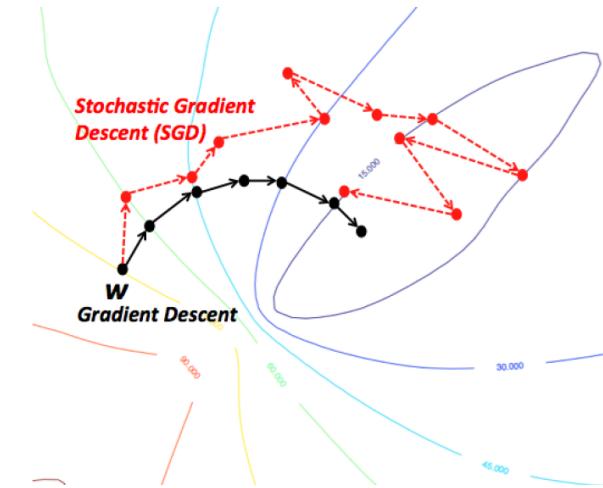
$$J(\mathbf{w}) = \sum_{p=1}^n J_p(\mathbf{w})$$

- Batch gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{n} \sum_{p=1}^n \nabla J_p(\mathbf{w})$$

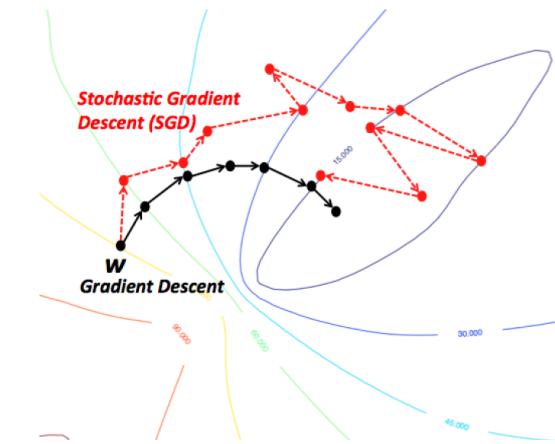
- In SGD, the gradient of $J(\mathbf{w})$ is approximated on a single example or a mini-batch of examples

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla J_p(\mathbf{w})$$



One-example based Backpropagation

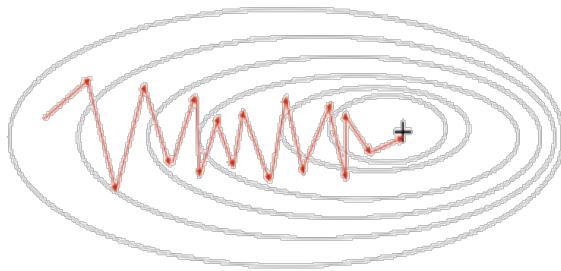
- Algorithm 1 (one-example based BP)
 - 1 begin initialize network topology (n_H), \mathbf{w} , criterion θ , η , $m \leftarrow 0$
 - 2 do $m \leftarrow m + 1$
 - 3 $x^m \leftarrow$ randomly chosen pattern
 - 4 $w_{ji} \leftarrow w_{ji} + \eta \delta_j x_i; \quad w_{kj} \leftarrow w_{kj} + \eta \delta_k y_j$
 - 5 until $\nabla J(\mathbf{w}) < \theta$
 - 6 return \mathbf{w}
 - 7 end
- In one-example based training, a weight update may reduce the error on the single pattern being presented, yet increase the error on the full training set.



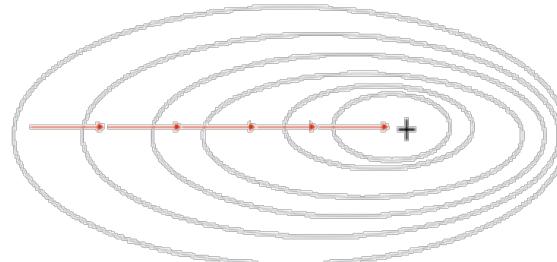
Mini-batch based SGD

- Divide the training set into mini-batches.
- In each epoch, randomly permute mini-batches and take a mini-batch sequentially to approximate the gradient
 - One epoch is usually defined to be one complete run through all of the training data.
- The estimated gradient at each iteration is more reliable.

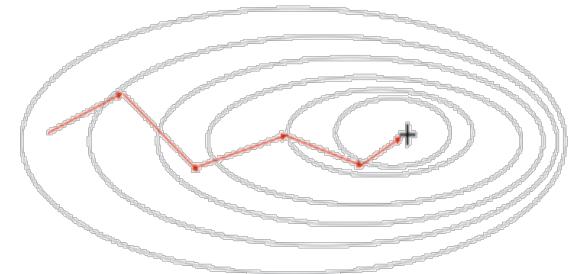
Stochastic Gradient Descent



Gradient Descent



Mini-Batch Gradient Descent



Mini-Batch Backpropagation

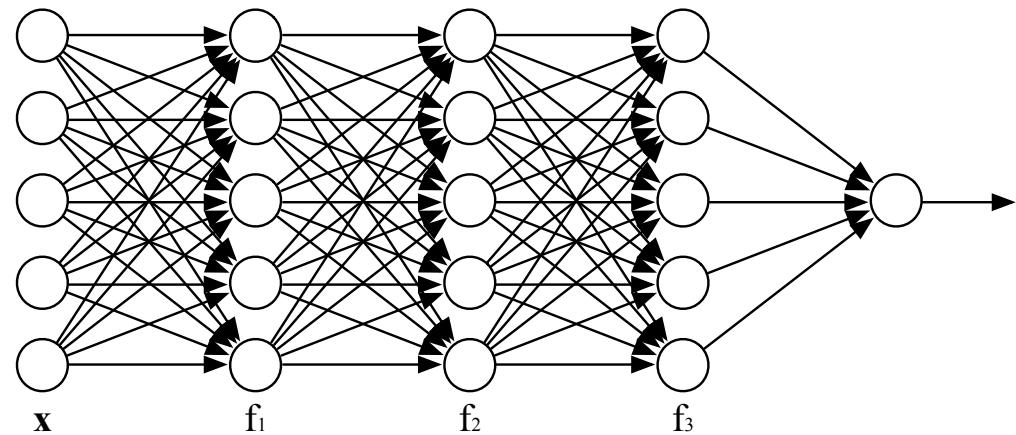
- Algorithm 2 (mini-batch based BP)
 - 1 **begin initialize** network topology (n_H), w , criterion θ , η , $r \leftarrow 0$
 - 2 **do** $r \leftarrow r + 1$ (increment epoch)
 - 3 $m \leftarrow 0$; $\Delta w_{ji} \leftarrow 0$; $\Delta w_{kj} \leftarrow 0$;
 - 4 **do** $m \leftarrow m + 1$
 - 5 $x^m \leftarrow$ randomly chosen pattern
 - 6 $\Delta w_{ji} \leftarrow \Delta w_{ji} + \eta \delta_j x_i$; $\Delta w_{kj} \leftarrow \Delta w_{kj} + \eta \delta_k y_j$
 - 7 **until** $m = n$
 - 8 $w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$; $w_{kj} \leftarrow w_{kj} + \Delta w_{kj}$
 - 9 **until** $\nabla J(w) < \theta$
 - 10 **return** w
 - 11 **end**

SGD Analysis

- One-example based SGD
 - Estimation of the gradient is noisy, and the weights may not move precisely down the gradient at each iteration
 - Faster than batch learning, especially when training data has redundancy
 - Noise often results in better solutions
 - The weights fluctuate and it may not fully converge to a local minimum
- Min-batch based SGD
 - Conditions of convergence are well understood
 - Some acceleration techniques only operate in batch learning
 - Theoretical analysis of the weight dynamics and convergence rates are simpler

Summarization

- Perceptron
- Multilayer Neural Network
- Backpropagation
- Activation Functions
- Stochastic Gradient Descent



Thank you!