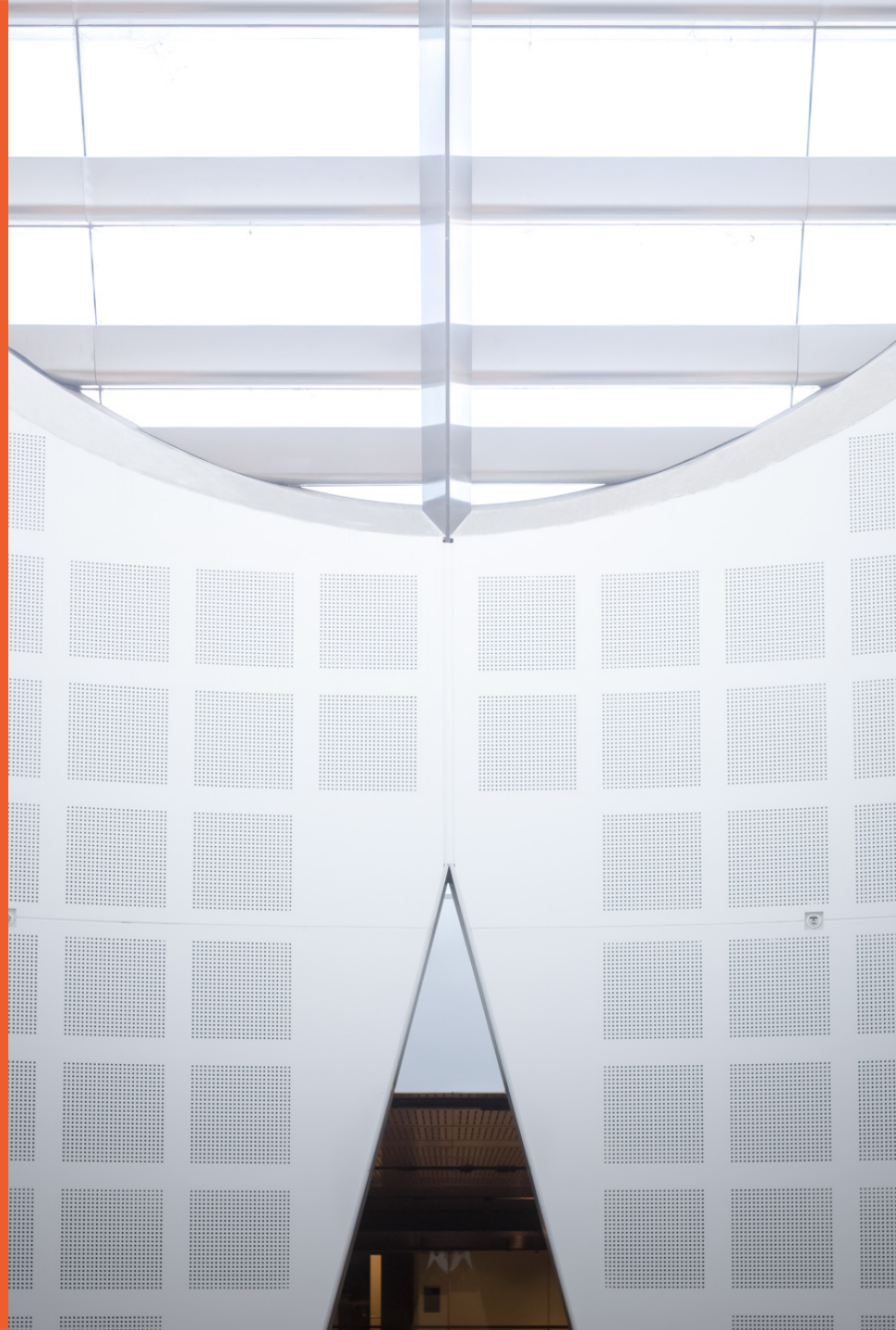
Multilayer Neural Network

Dr Chang Xu

School of Computer Science



THE UNIVERSITY OF SYDNEY

Perceptron: the Prelude of Deep Learning

A neuron cell

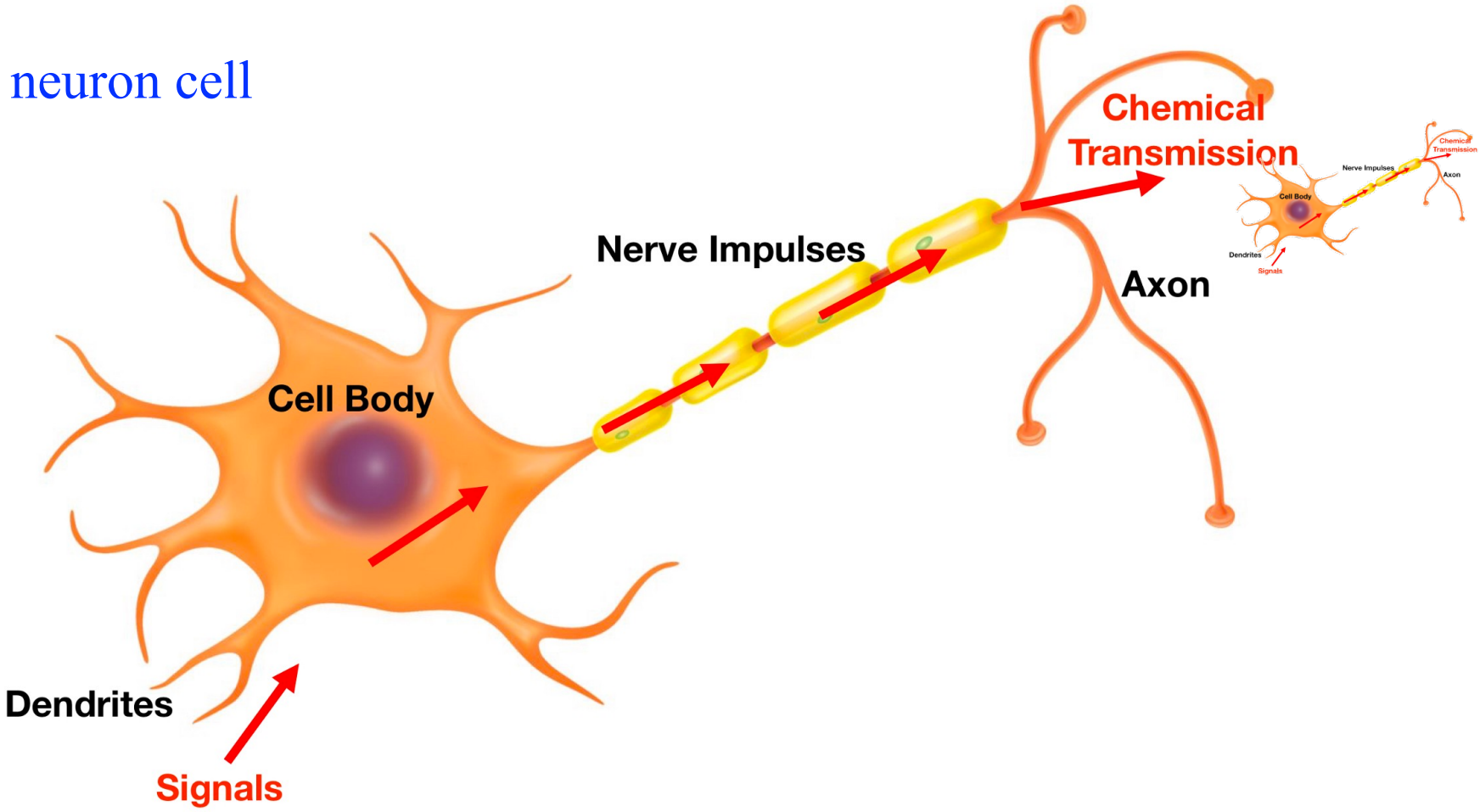
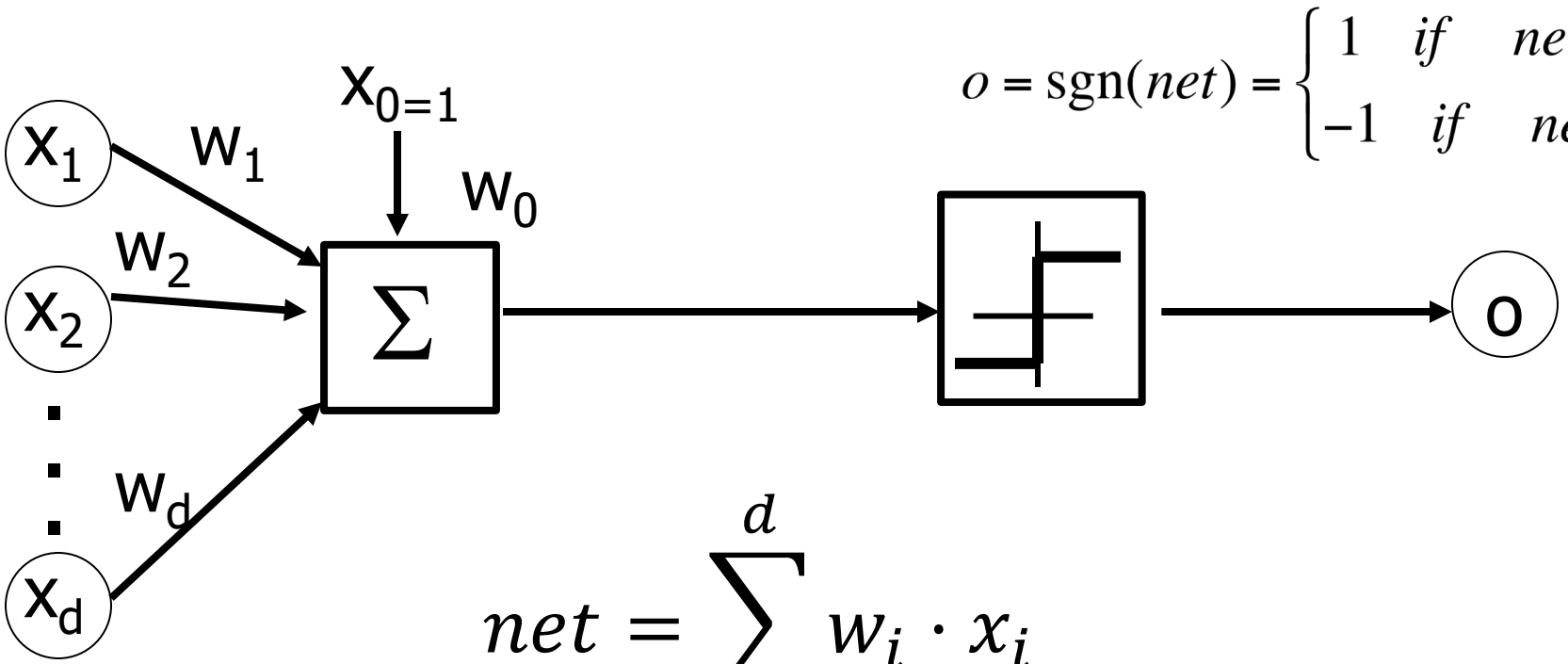


Image credit to: thinglink.com

Perceptron: the Prelude of DL

[Rosenblatt, et al. 1958]



Perceptron

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

Neuron's estimate of the desired output

weight vector

input vector

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

The aim of learning is to minimize the discrepancy between the desired outputs and the actual outputs of n examples:

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

Perceptron

- Define the error as the squared residuals summed over all training cases:

$$L = \frac{1}{2}\sum_{j=1}^{n}\left(y_j - \hat{y}_j\right)^2$$

$$= \frac{1}{2}\sum_{j=1}^{n}\left(y_j - w^T x_j\right)^2$$

- Now differentiate to get error derivatives for weight

$$\frac{\partial L}{\partial w} = \frac{1}{2}\sum_{j}\frac{\partial L_j}{\partial \hat{y}_j}\frac{\partial \hat{y}_j}{\partial w}$$

$$= -\sum_{j}\left(y_j - \hat{y}_j\right)x_j$$

- The update rule changes the weight in proportion to its error derivative summed over all training cases

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

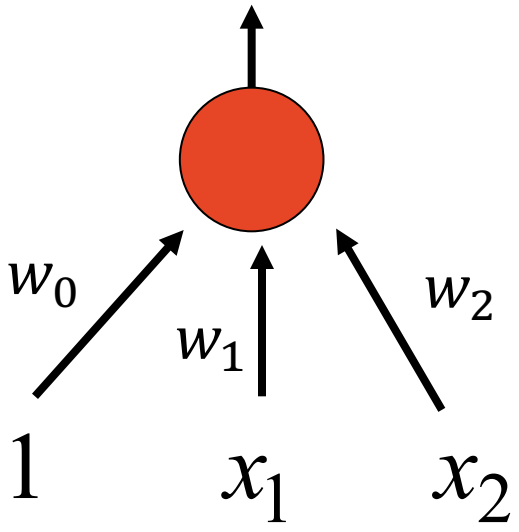
Perceptron

- A linear neuron is a more flexible model if we include a bias.

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

$$= w^T x$$

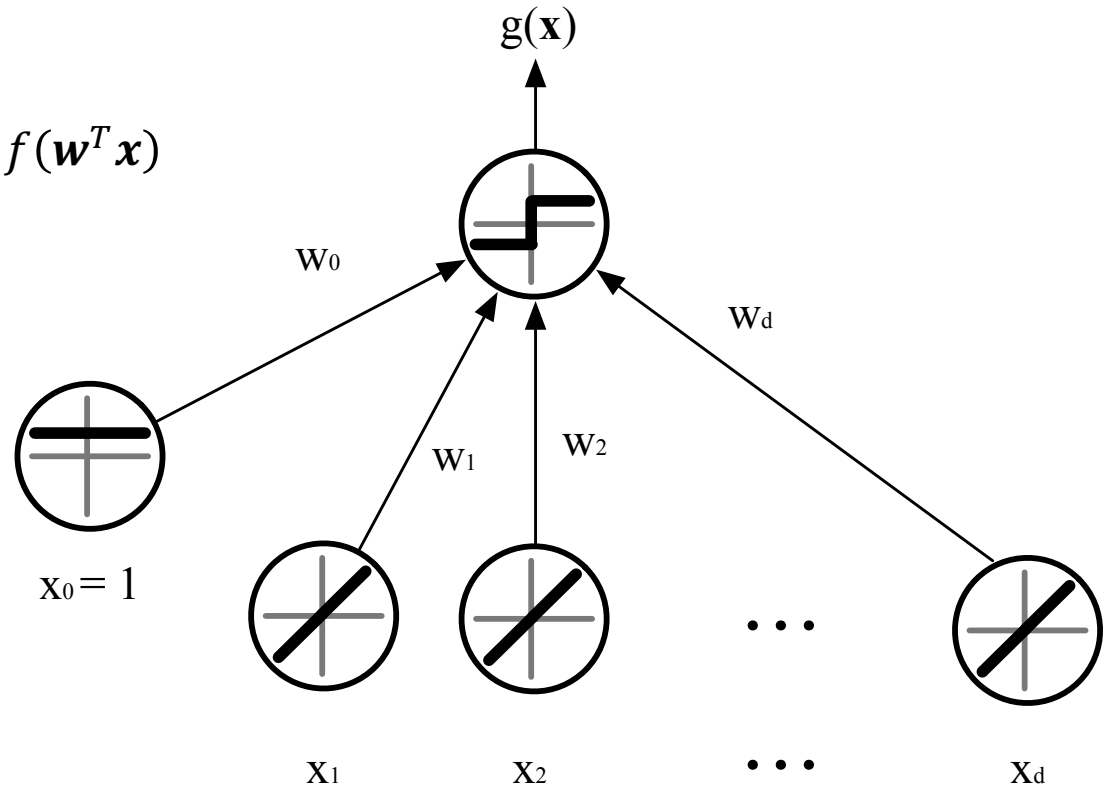
- We can avoid having to figure out a separate learning rule for the bias by using a trick:
  - A bias is exactly equivalent to a weight on an extra input line that always has an activity of 1.



Perceptron

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

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



Linear classifier!

Perceptron "AI winter"

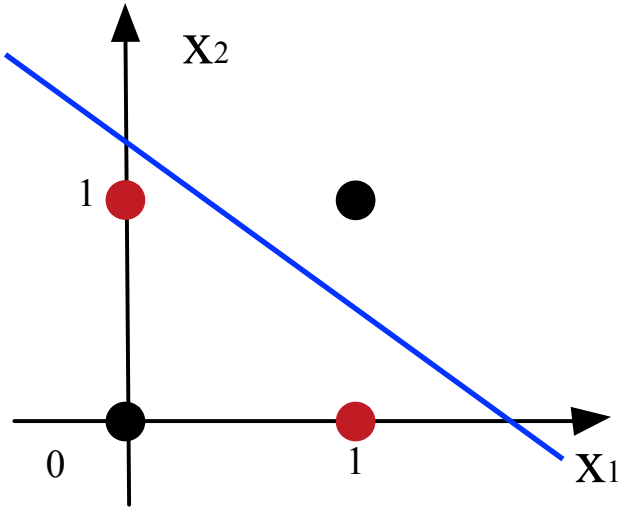
Marvin Minsky & Seymour Papert (1969). *Perceptrons*, MIT Press, Cambridge, MA.

- 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.

$$w_1 x_1 + w_2 x_2 + b$$

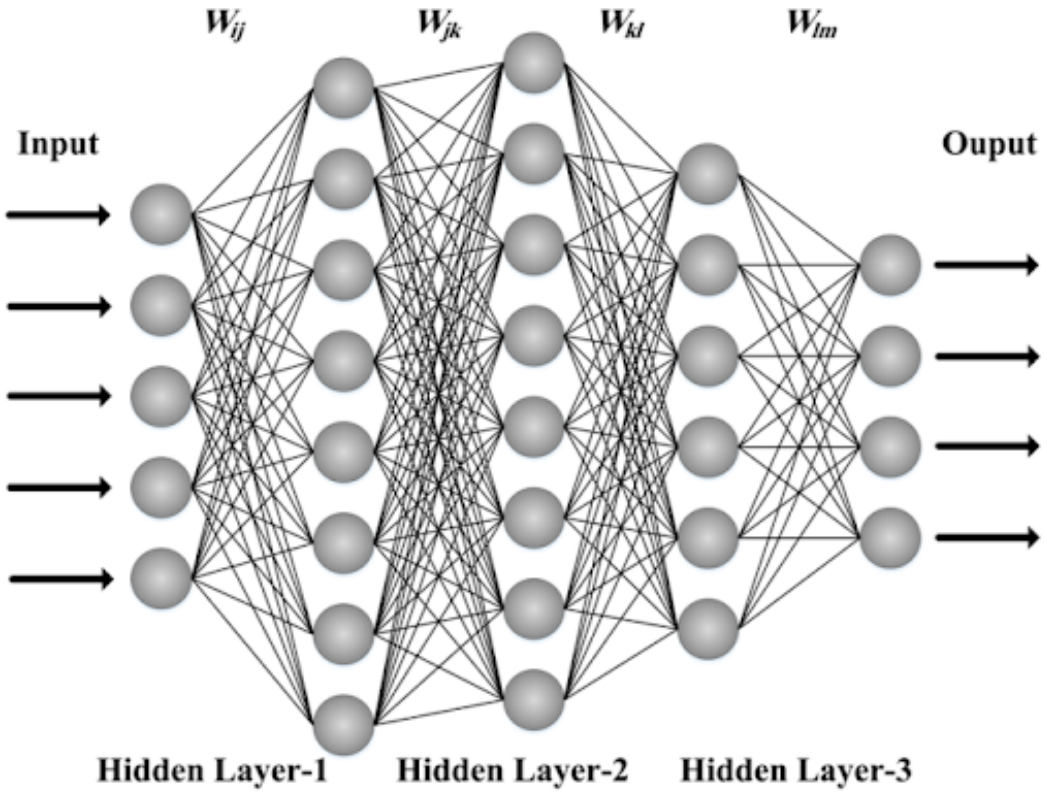
This failure caused the majority of researchers to walk away.

Breakthrough: Multi-Layer Perceptron

- In1986, 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)

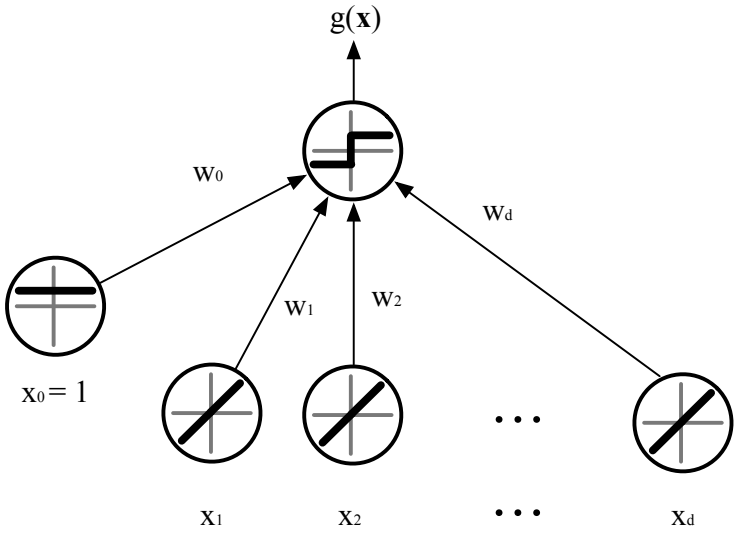
Multilayer Neural Network

Multilayer Neural Networks

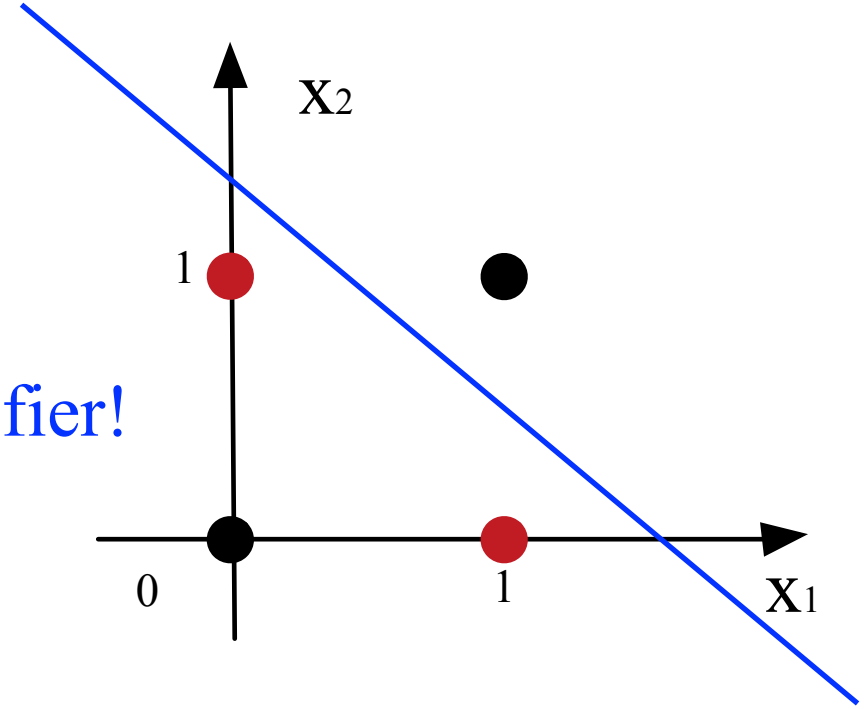


- Function composition can be described by a directed acyclic graph (hence feedforward networks)
- Depth is the maximum i in the function composition chain
- Final layer is called the output layer

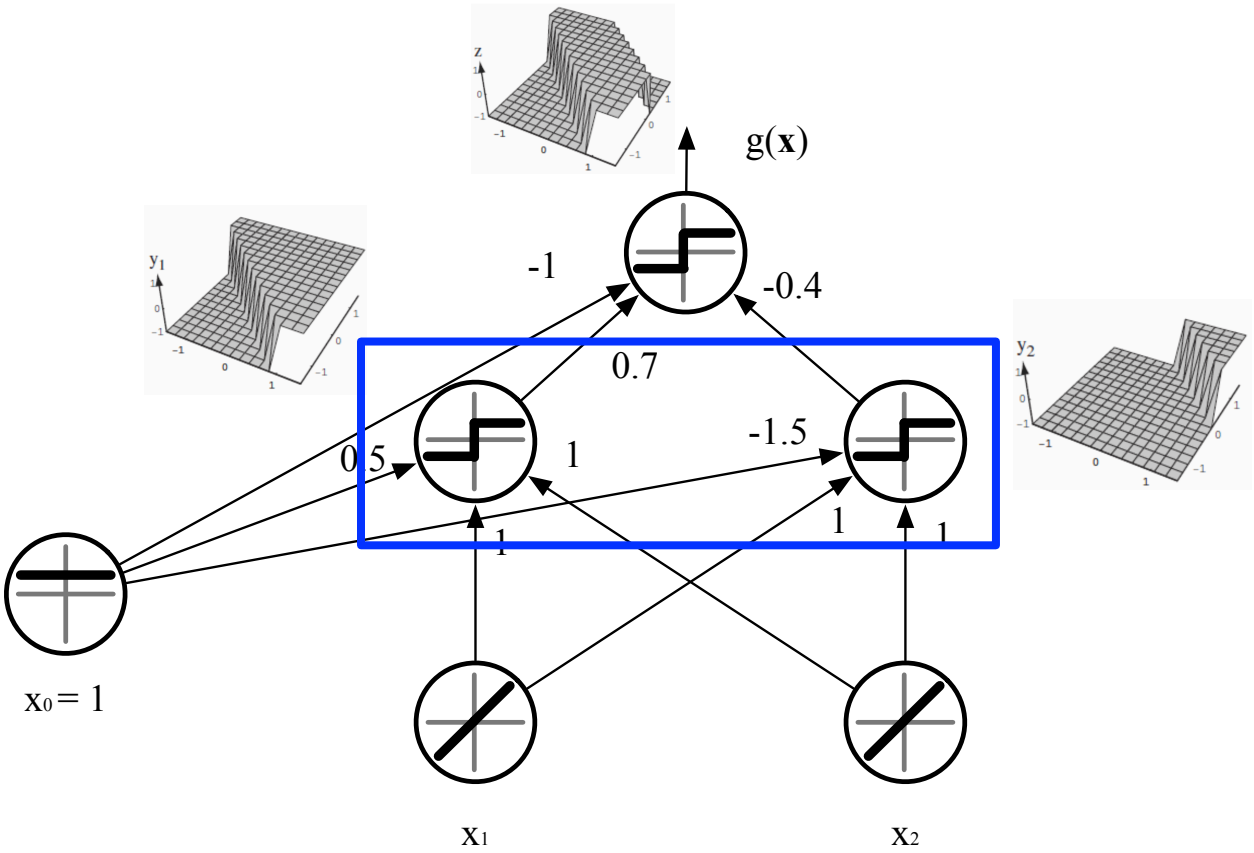
Two-layer Neural Network

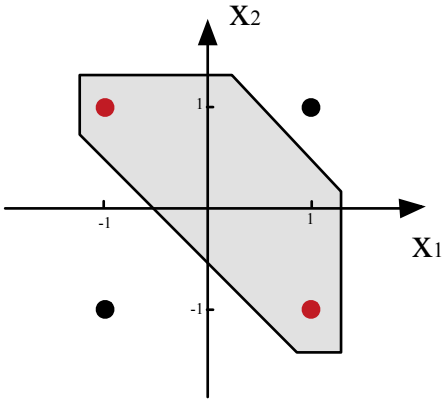


Linear classifier!



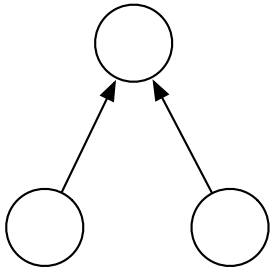
Three-layer Neural Network (with a hidden layer)

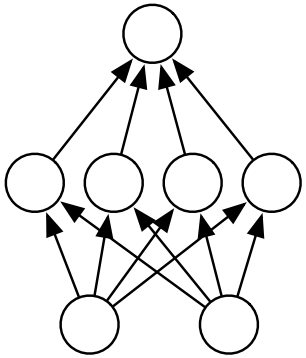


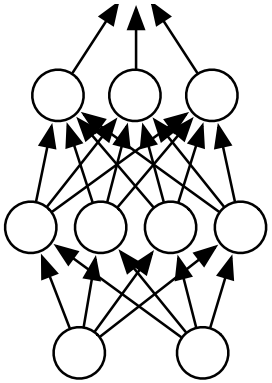


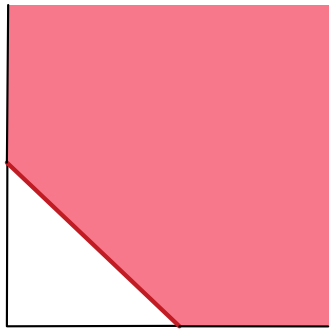
Three-layer Neural Network (with a hidden layer)

- A 2D-input example

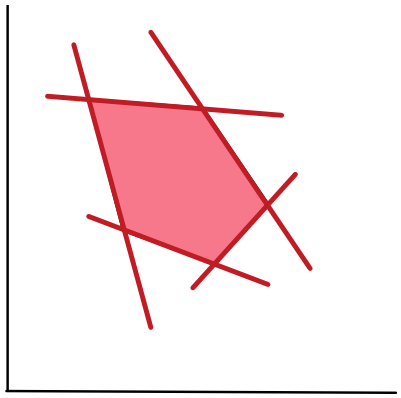




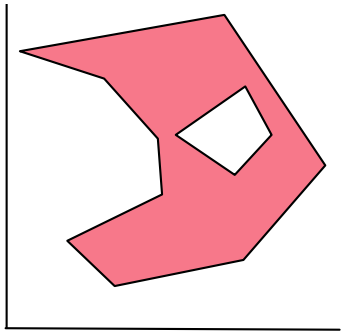




separating hyperplane



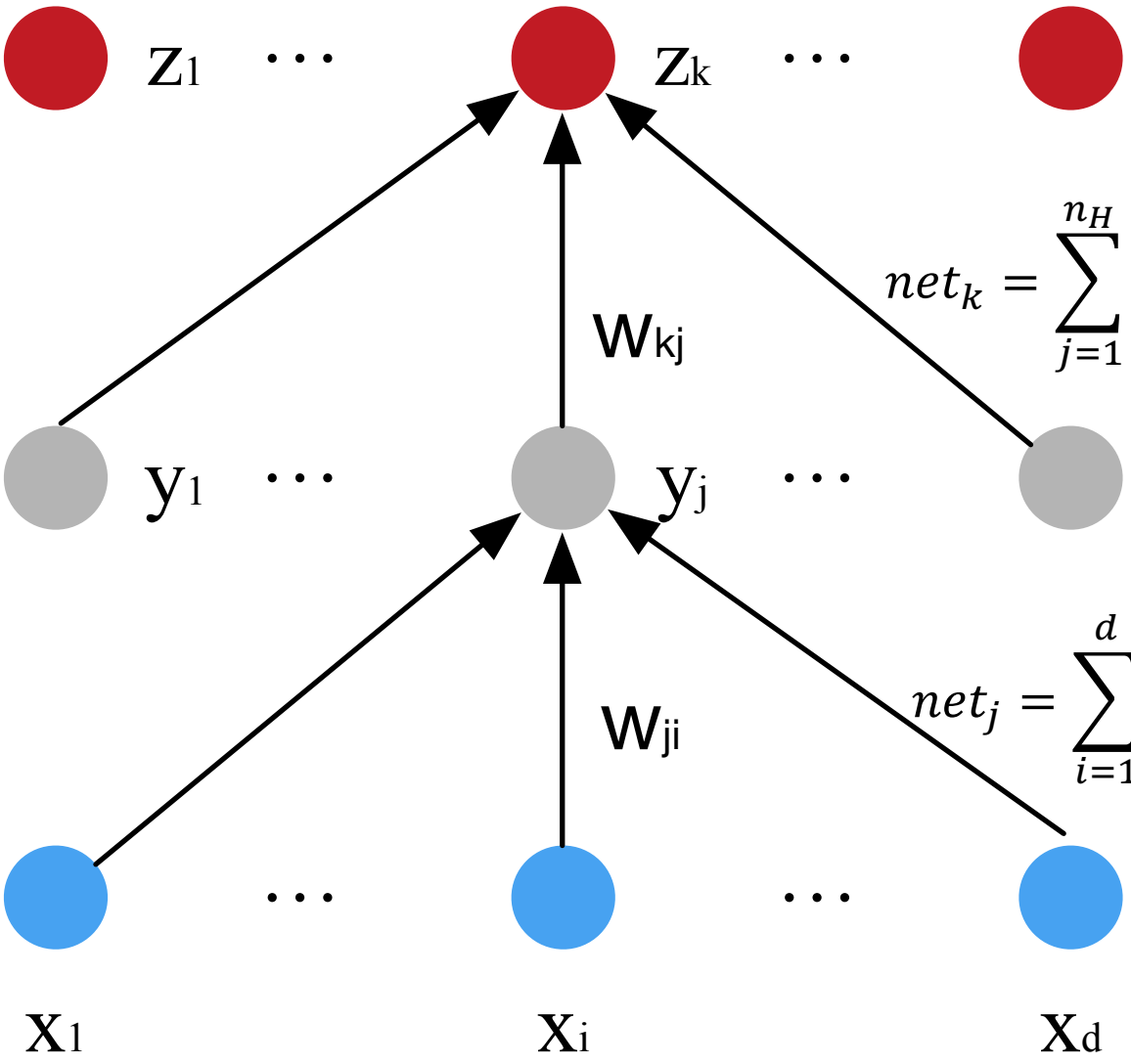
convex polygon region



composition of polygons

Feedforward

$$z_k = f(net_k)$$
$$= f\left(\sum_{j=1}^{n_H} w_{kj} f(net_j) + w_{k0}\right)$$



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

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

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 exists an integer N, real constants $v_i, b_i \in \mathbb{R}^m$ and real vectors $w_i \in \mathbb{R}^m$, where $i = 1, \cdots, N_i$ such that we may define:

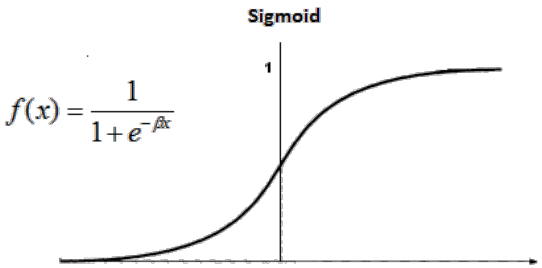
$$\hat{F}(x) = \sum_{i}^{N} v_i f(w_i^T x + b_i)$$

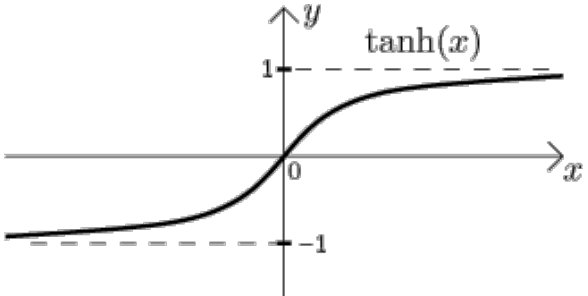
as an approximate realization of the function F where F is independent of f; that is,

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

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

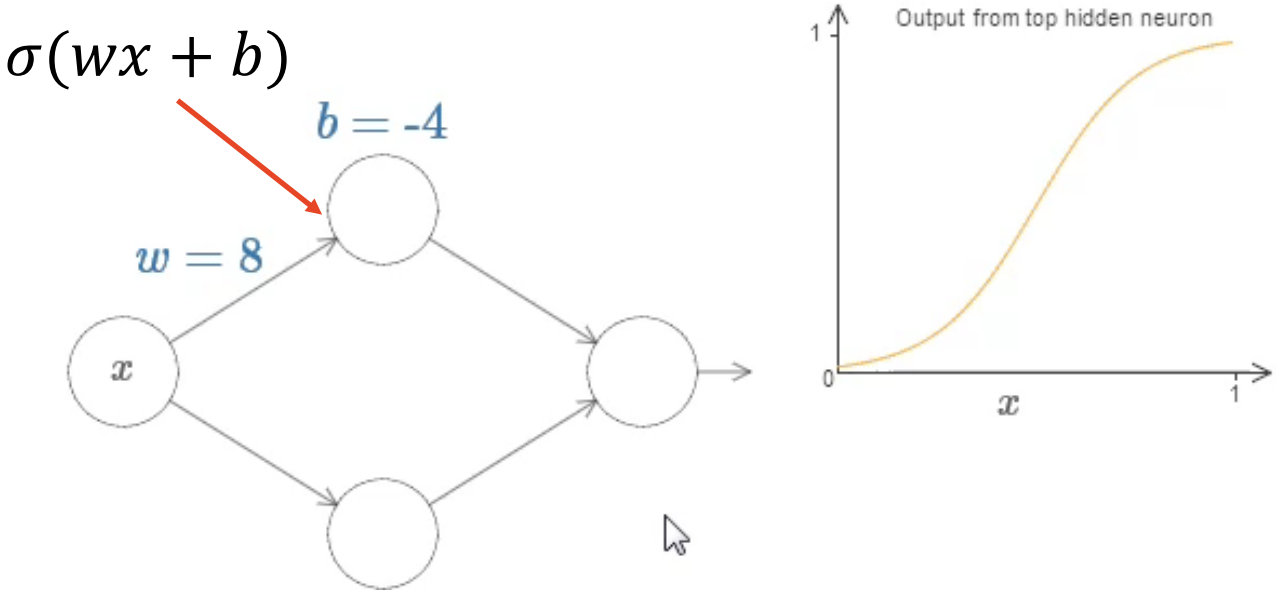
Universality theorem (Hecht-Nielsen 1989): "Neural networks with a single hidden layer can be used to approximate any continuous function to any desired precision."





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 curve.
- The larger the weight, the steeper the curve.
- 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

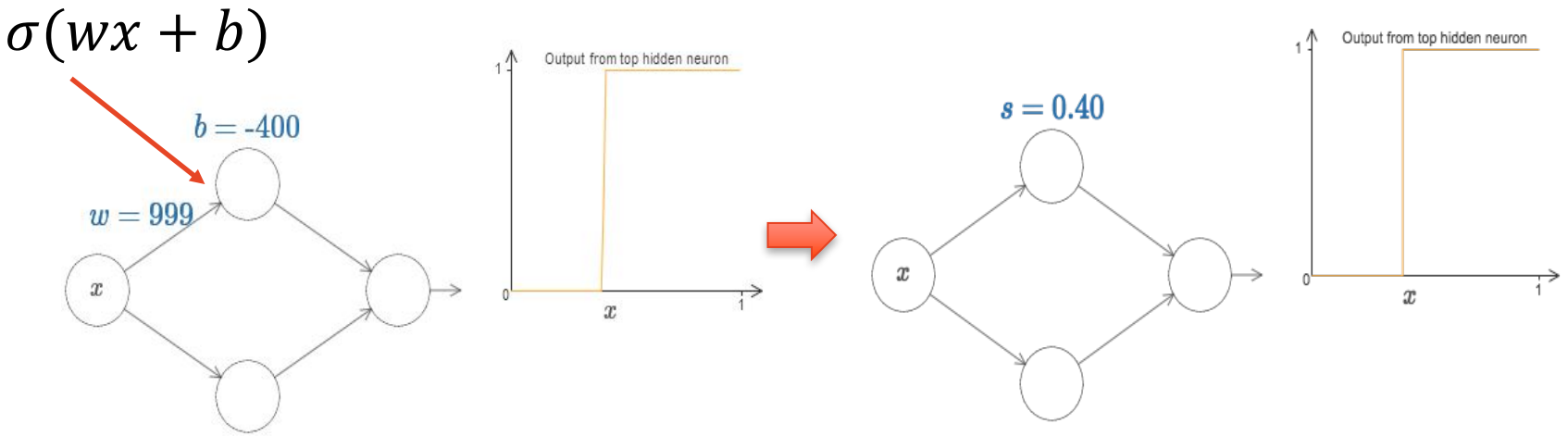


Image credit to: https://neuralnetworksanddeeplearning.com

- 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

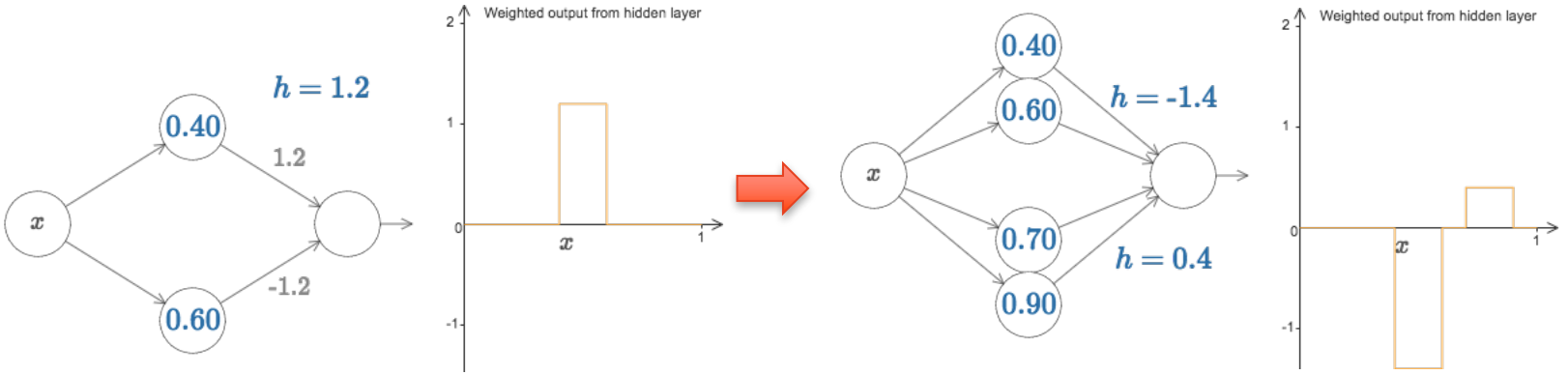


Image credit to: https://neuralnetworksanddeeplearning.com

- *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

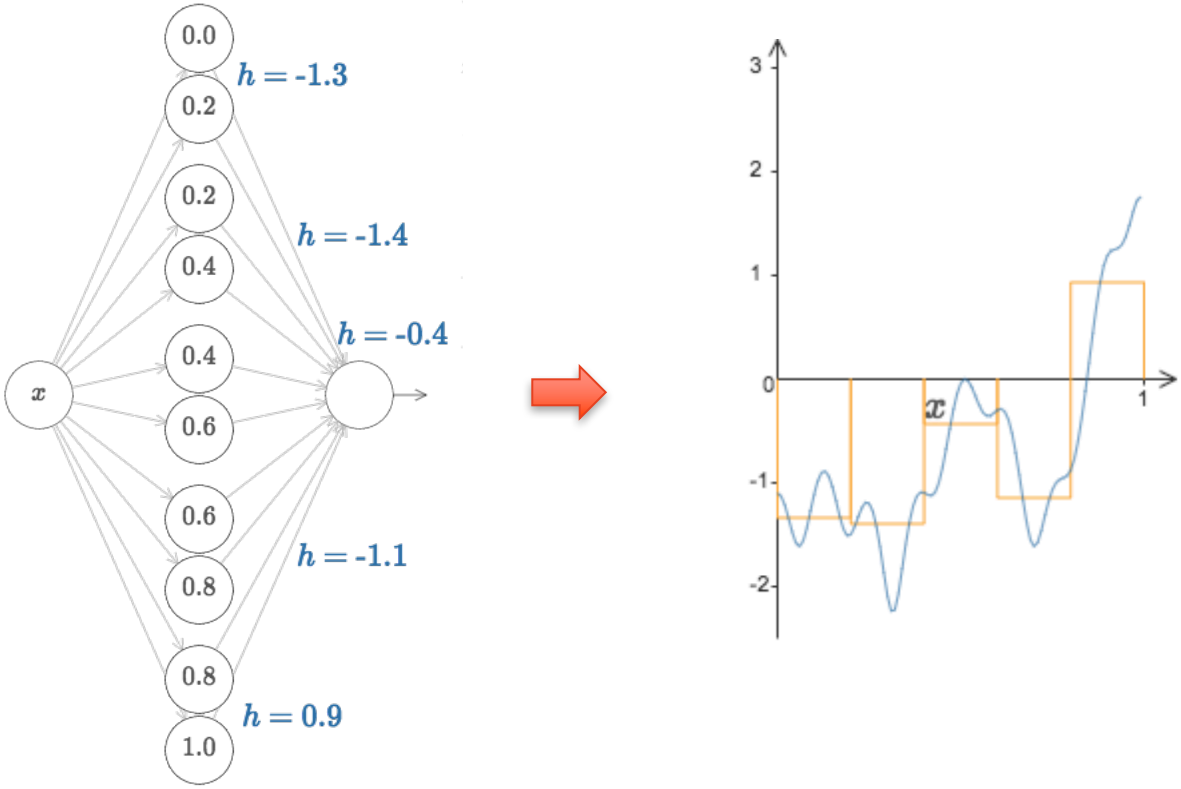
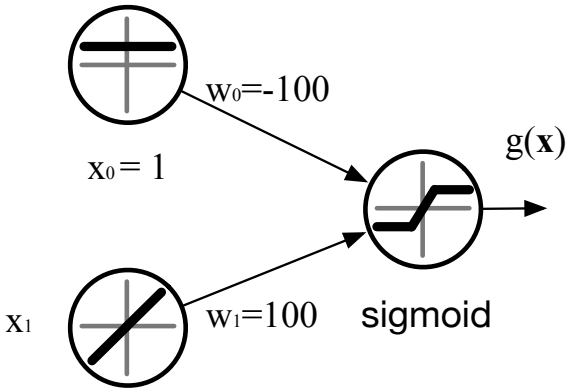


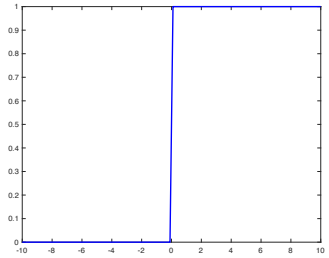
Image credit to: https://neuralnetworksanddeeplearning.com

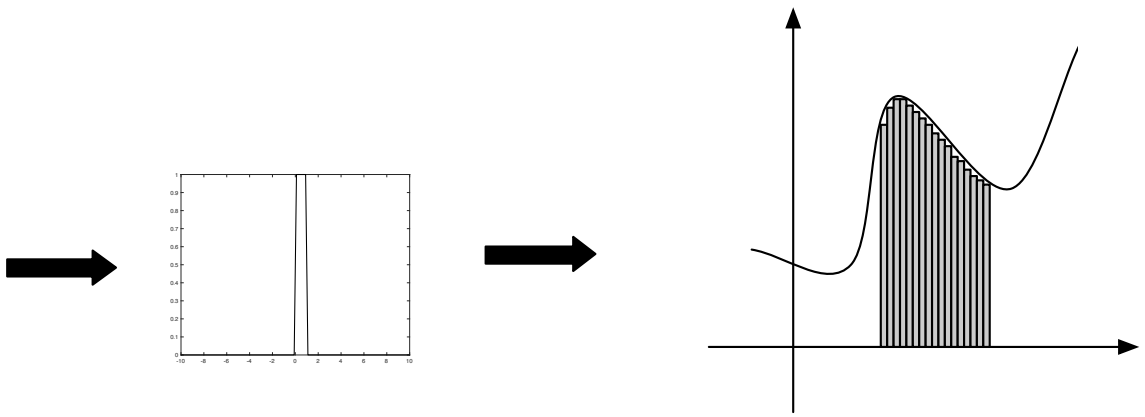
- By increasing the number of hidden neurons, we can make the approximation much better.

Expressive power of a three-layer neural network

- Overall procedure



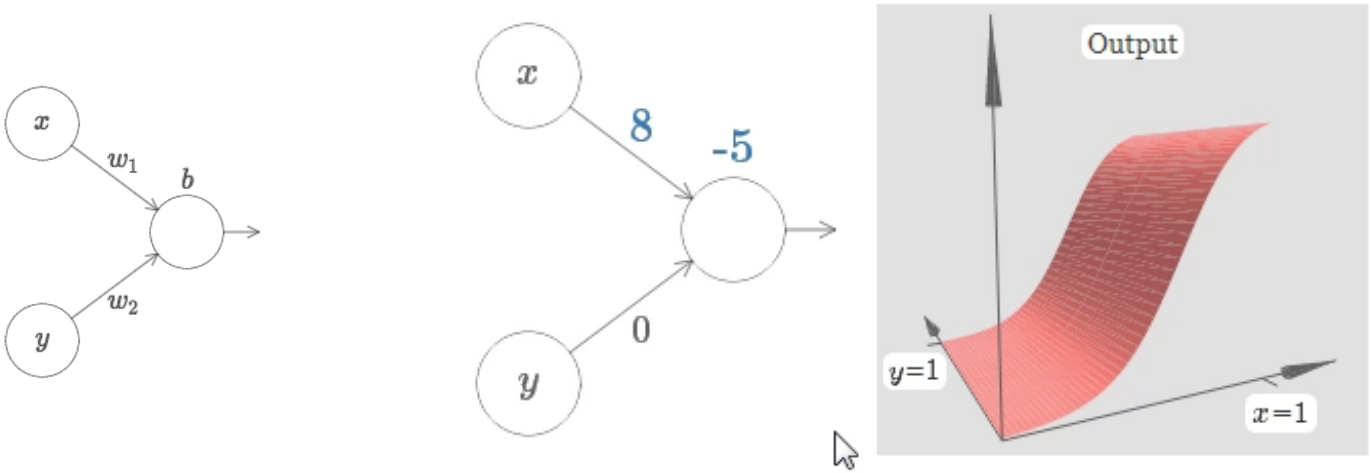




Expressive power of a three-layer neural network

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

  Fix $w_2 = 0$, change w_1 and b:

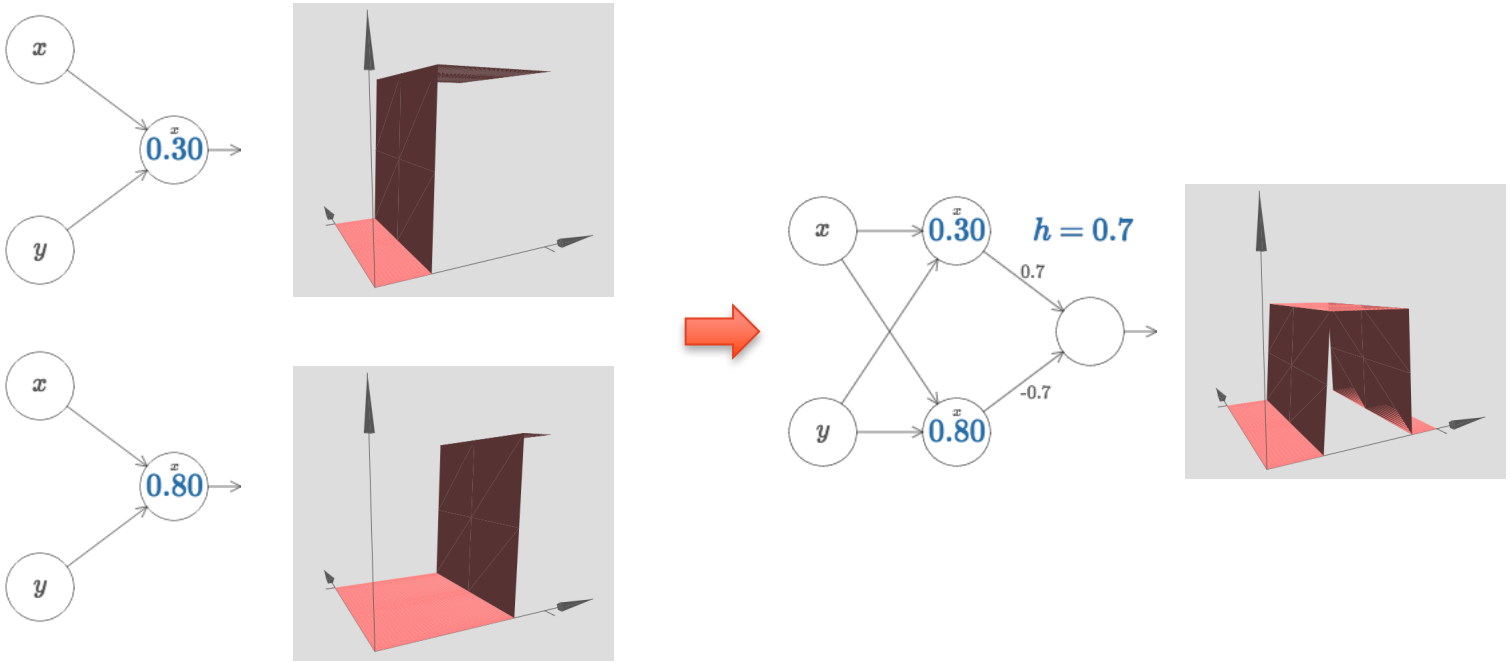


Video credit to: https://neuralnetworksanddeeplearning.com

  - 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:

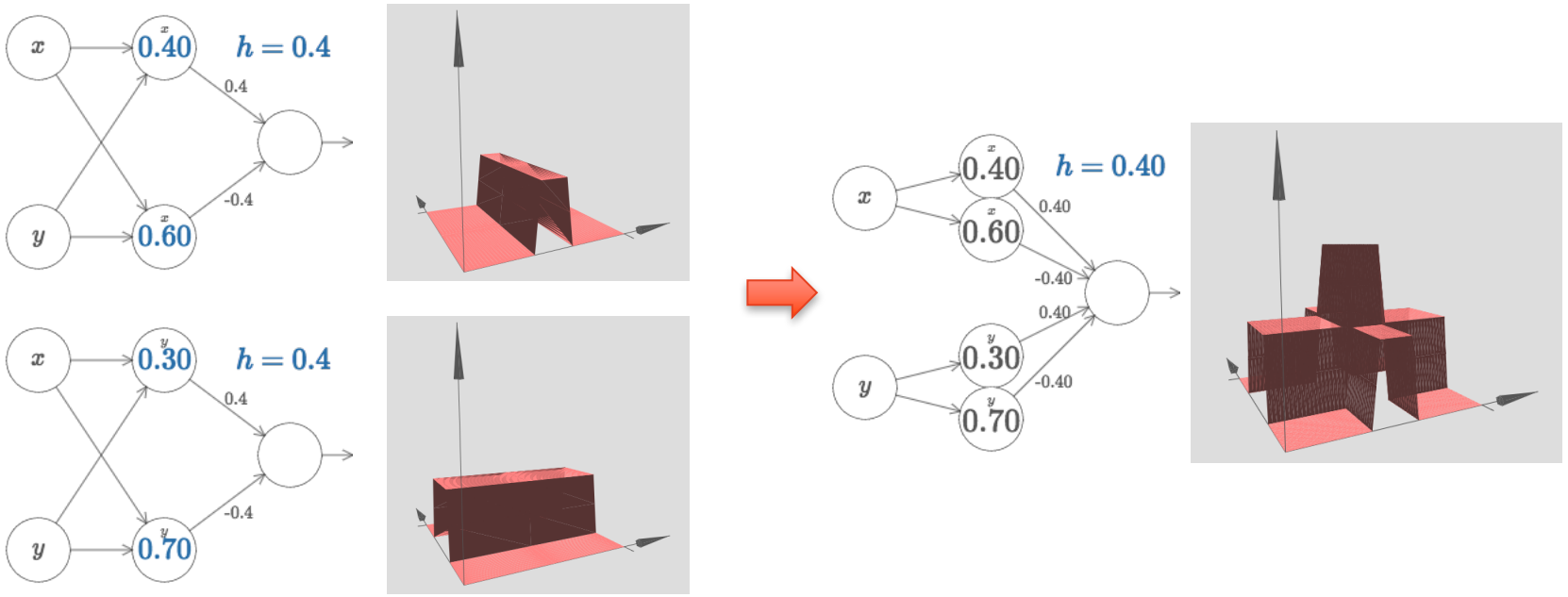


Image credit to: https://neuralnetworksanddeeplearning.com

Expressive power of a three-layer neural network

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

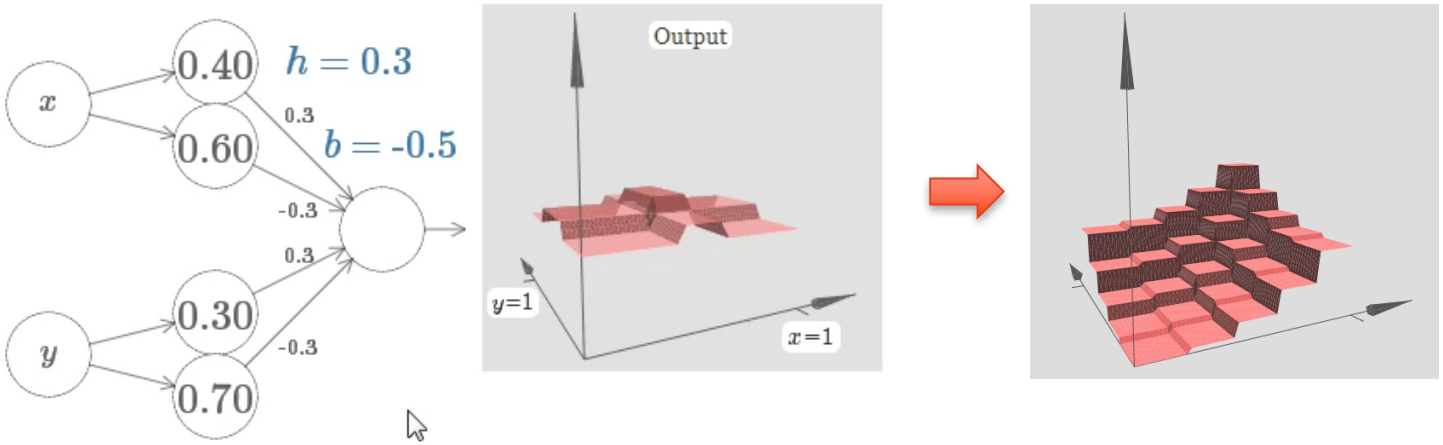


Image credit to: https://neuralnetworksanddeeplearning.com

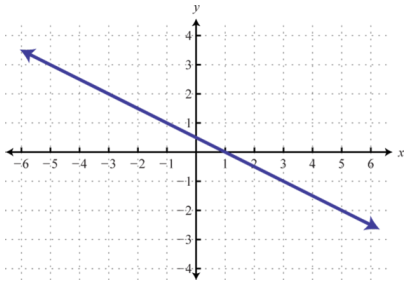
- By adding bias to the output neuron and adjusting the value of h and b, we can construct a 'tower' function.
- The same idea can be used to compute as many towers as we like. We can also make them as thin as we like, and whatever height we like. As a result, we can ensure that the weighted output from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

Activation Functions

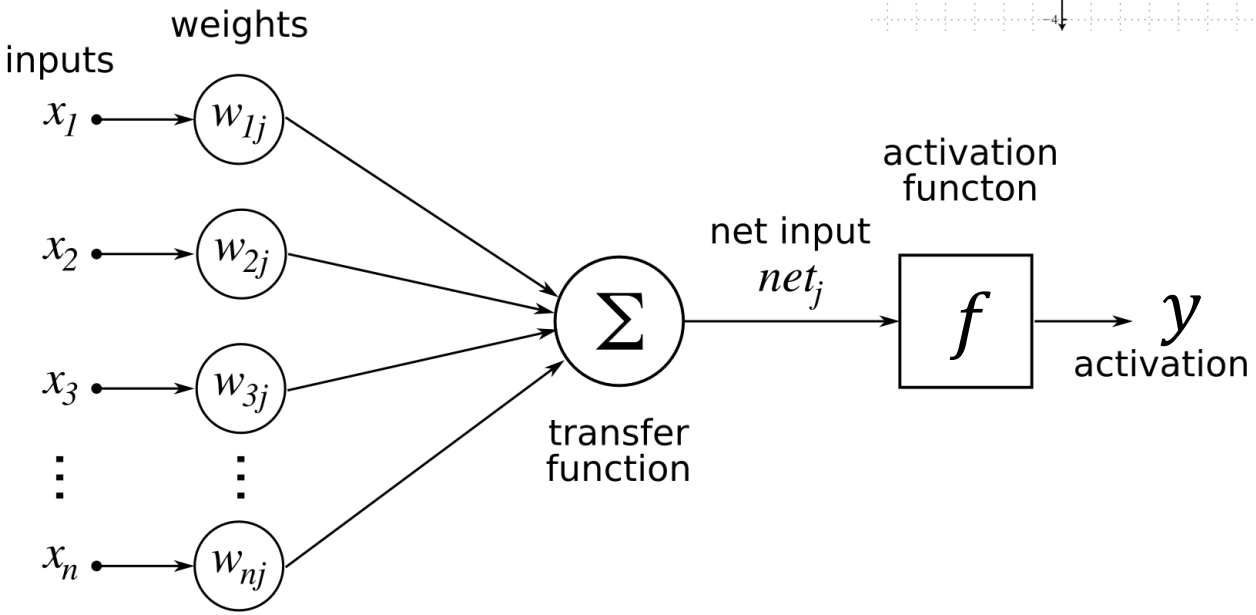
Activation functions

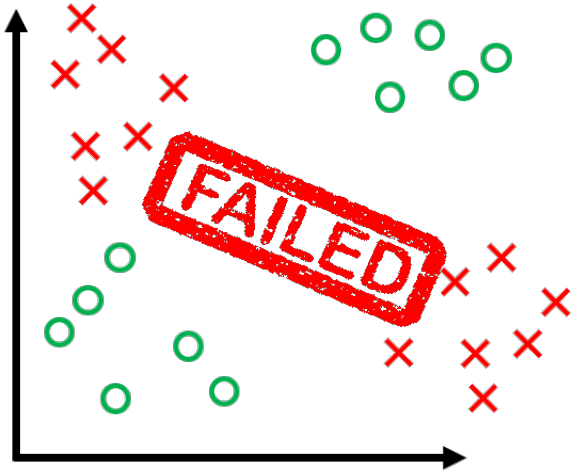
- Must be nonlinear: otherwise it is equivalent to a linear classifier

$w^T x$



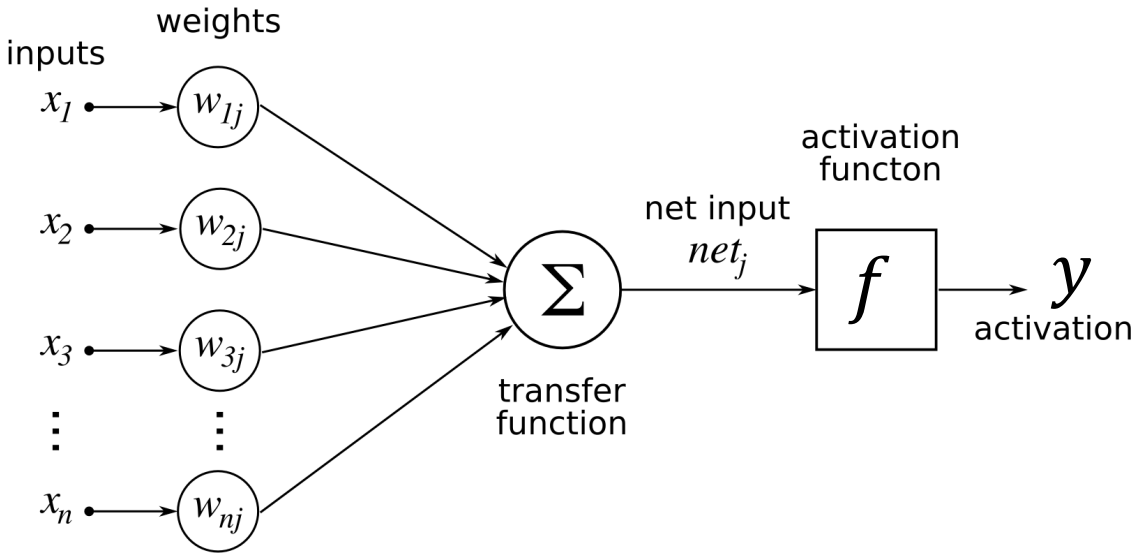
Linear function!

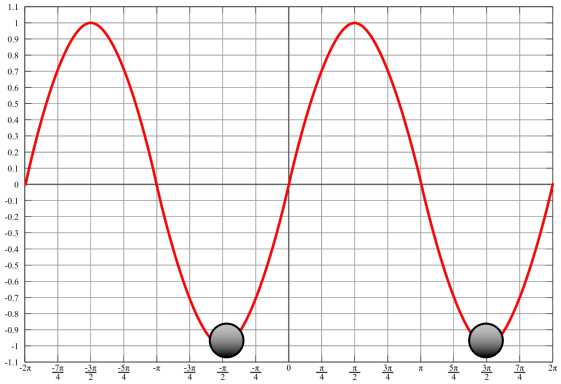




Activation functions

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



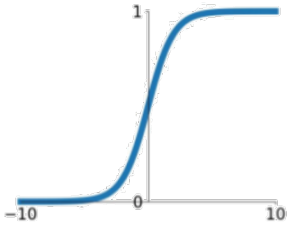


An output value might correspond to multiple input values.

Activation function $f(s)$

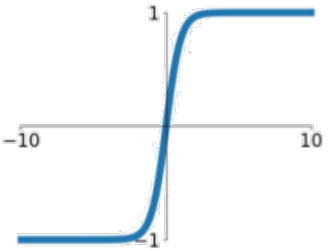
- Popular choice of activation functions (single input)
  - Sigmoid function

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



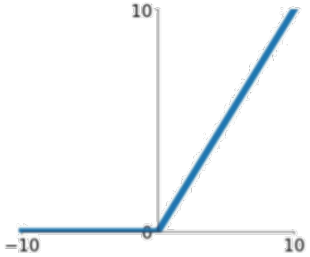
  - Tanh function: shift the center of Sigmoid to the origin

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



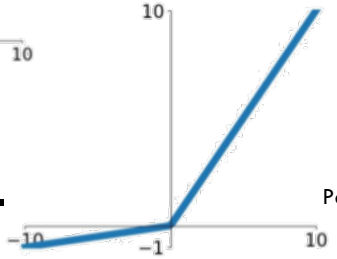
  - Rectified linear unit (ReLU)

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



  - Leaky ReLU

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

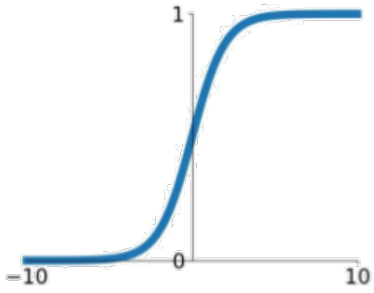


Vanishing gradient problems

- Saturation.

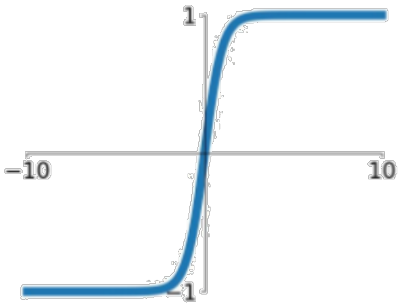
Sigmoid function

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



Tanh function

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

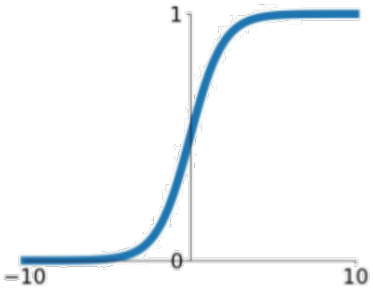


❖ At their extremes, their derivative are close to 0, which kill gradient and learning process

Not zero centered activation

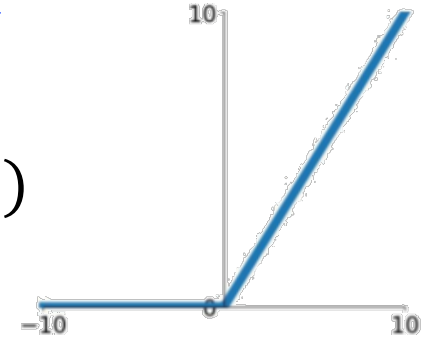
Sigmoid function

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

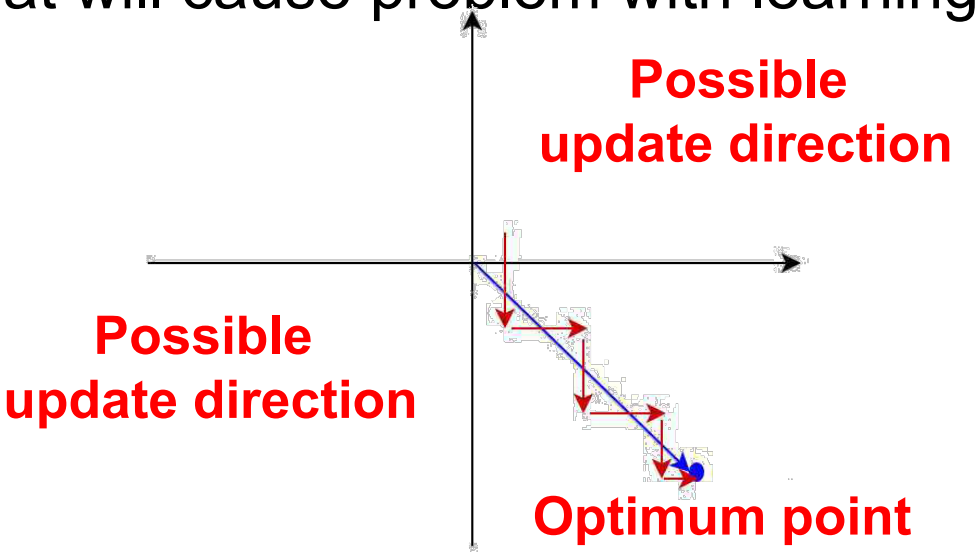


Relu function

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

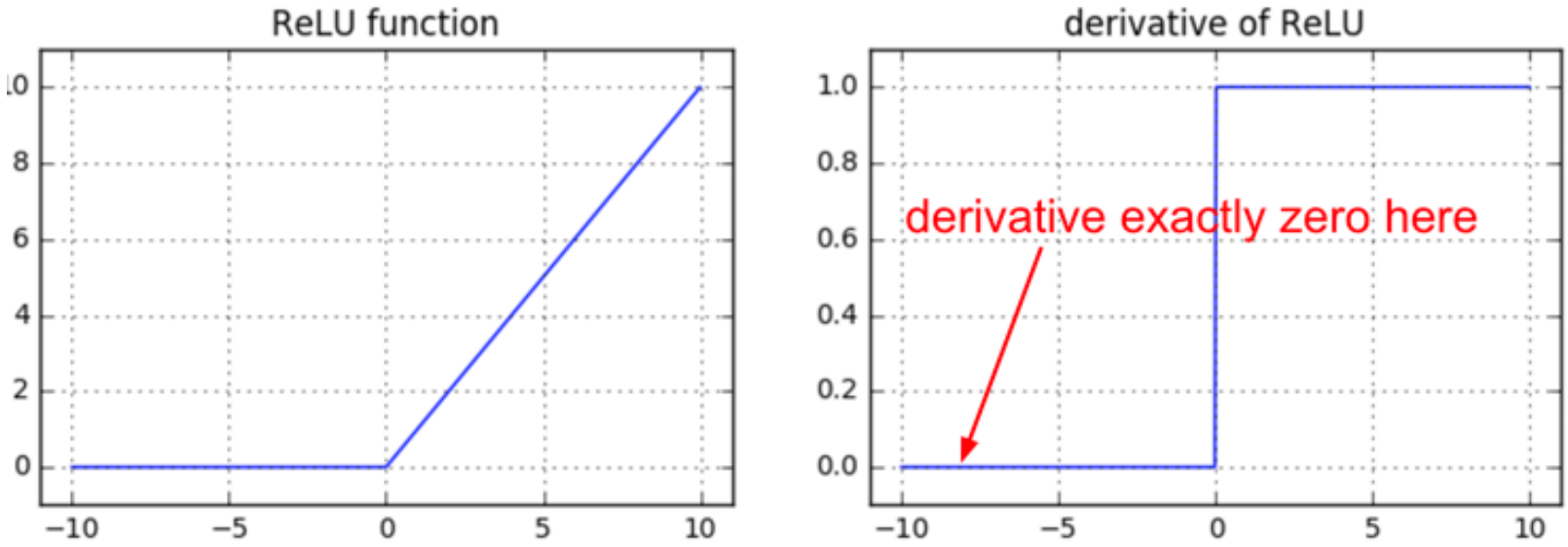


- During training, all gradients will be all positive or all negative that will cause problem with learning.



Dead ReLUs

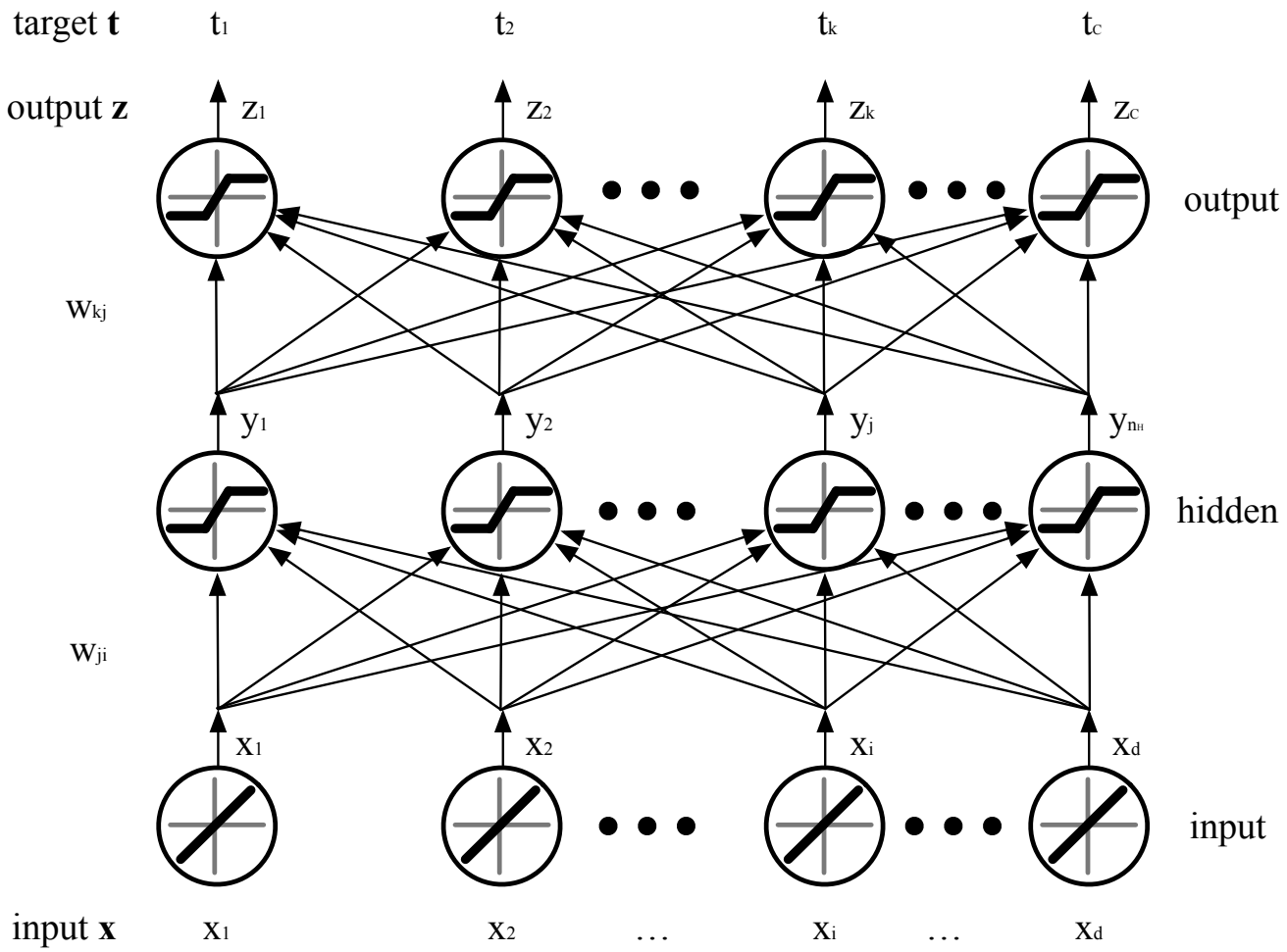
ReLU function $f(s) = \max(0, s)$



- 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.

Backpropagation

A three-layer neural network for illustration



$$z_k = f(net_k)$$

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0}$$

$$y_j = f(net_j)$$

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0}$$

Training error

- Euclidean distance.

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

- 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 $\boldsymbol{t}$ and $\boldsymbol{z}$,

$$CrossEntropy(\boldsymbol{t}, \boldsymbol{z}) = -\sum_i t_i \log z_i$$

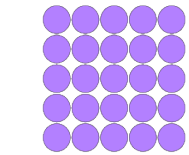
$$= -\sum_i t_i \log t_i + \sum_i t_i \log t_i - \sum_i t_i \log z_i$$

$$= -\sum_i t_i \log t_i + \sum_i t_i \log \frac{t_i}{z_i}$$

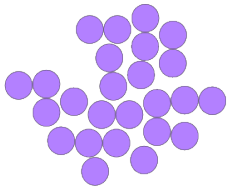
$$= Entropy(\boldsymbol{t}) + D_{KL}(\boldsymbol{t}|\boldsymbol{z})$$

Entropy is a measure of degree of disorder or randomness.

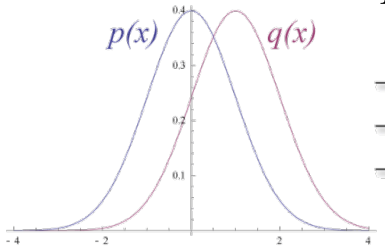
KL (Kullback-Leibler) divergence is a measure of the information lost when Z is used to approximate T



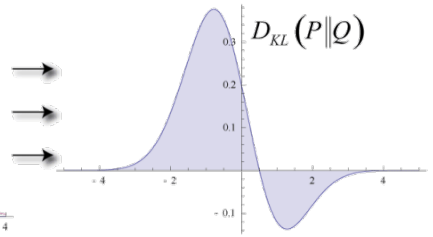
Low Entropy



High Entropy



Original Gaussian PDF's

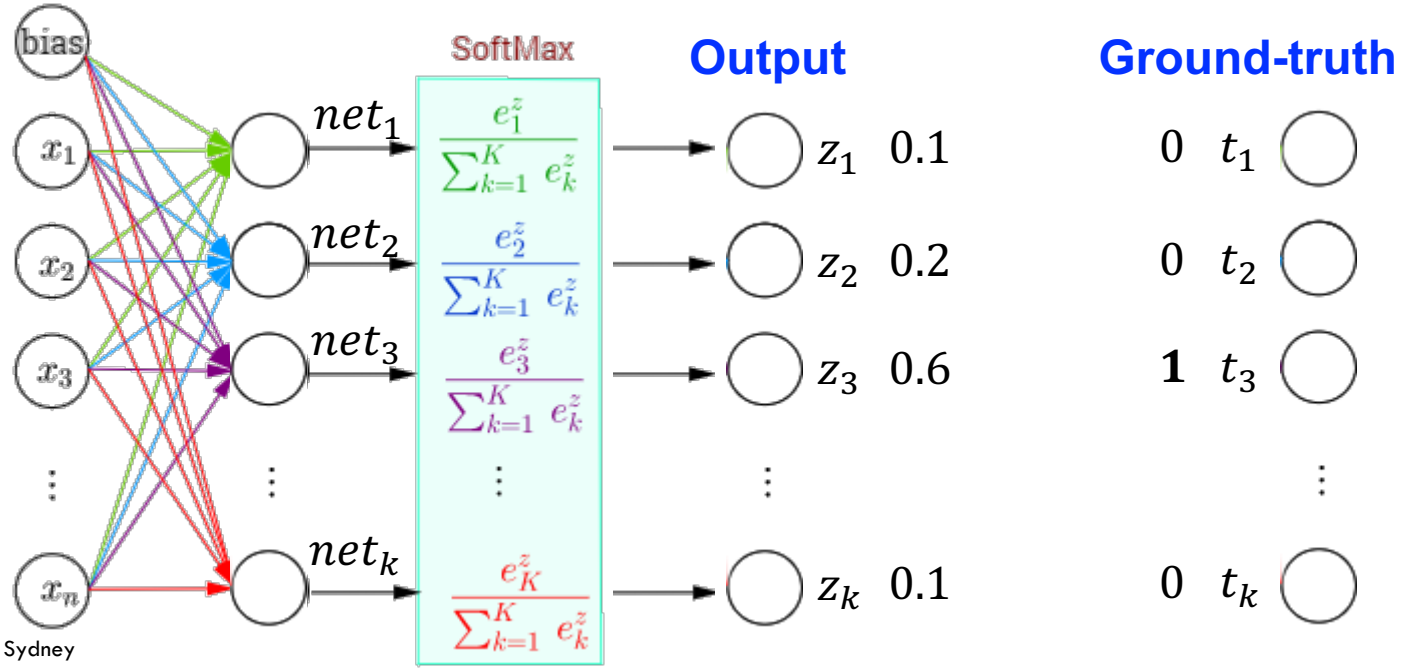


KL Area to be Integrated

Softmax Function

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

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



Gradient descent

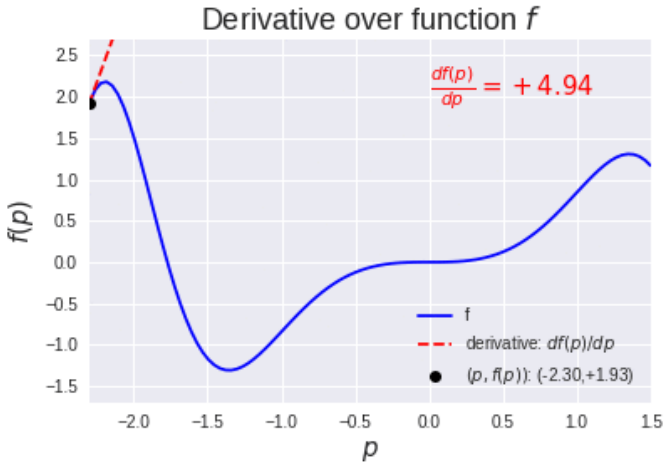
- Weights are initialized with random values, and then are updated in a direction reducing the error.

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

where η is the learning rate.

Iteratively update

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

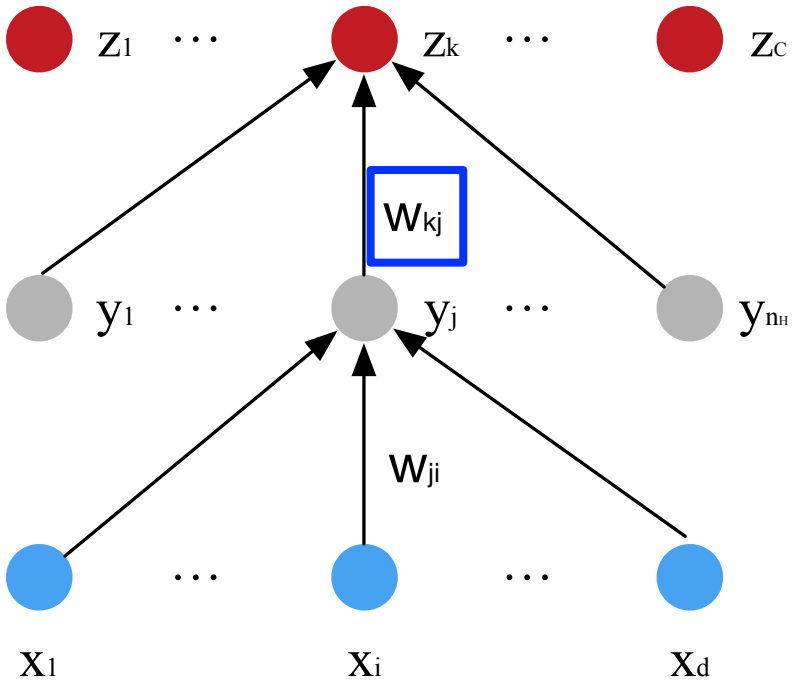


Hidden-to-output weight w_{kj}

- Chain rule

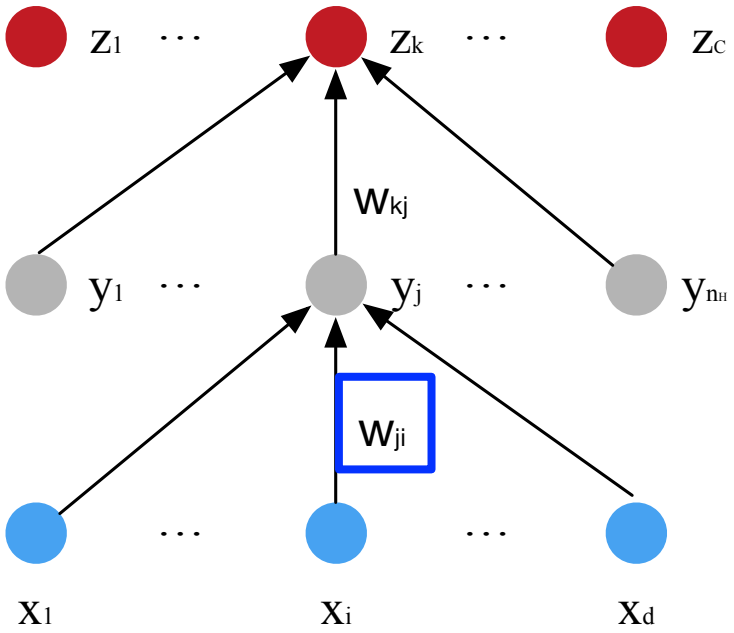
$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} y_j$$

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



Input-to-hidden weight w_{ji}

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



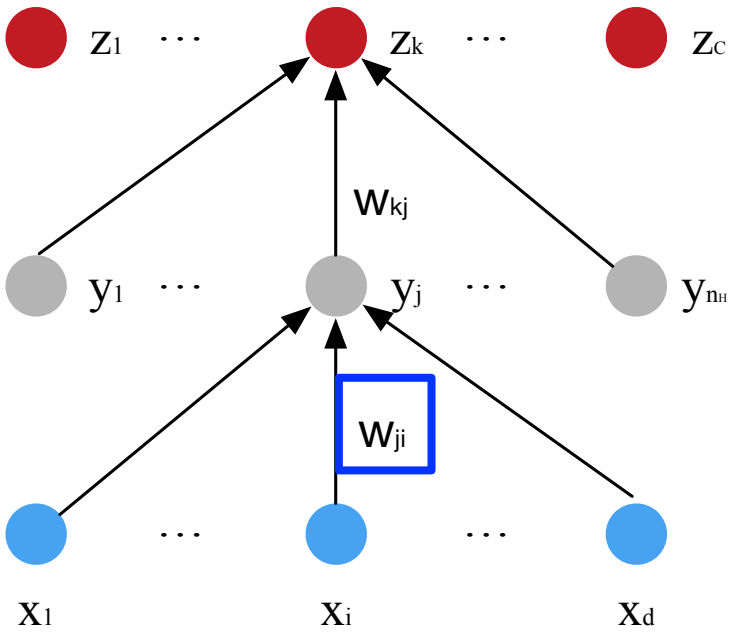
$$\frac{\partial J}{\partial y_j} = \sum_{k=1}^{C} \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial y_j}$$

$$net_j = \sum_{i=1}^{d} x_j w_{ji} + w_{j0}$$

$$y_j = f(net_j)$$

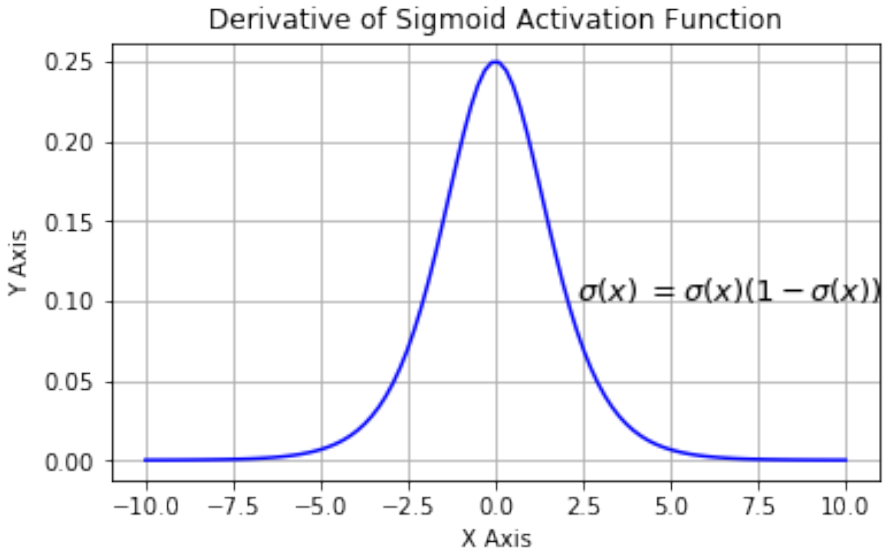
Vanishing gradients problem

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



$$\frac{\partial y_j}{\partial net_j} = \sigma'\left(\sum_{i=1}^{d} x_j w_{ji} + w_{j0}\right)$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x)) = \frac{e^{-x}}{(e^{-x}+1)^2}$$

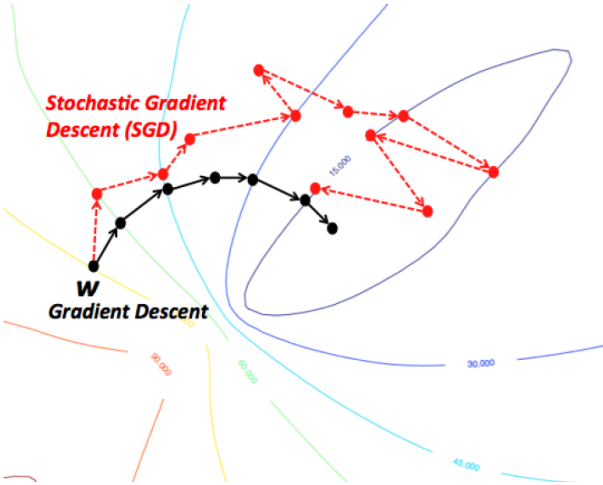


Stochastic Gradient Descent

Stochastic Gradient Descent (SGD)

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

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



- Batch gradient descent

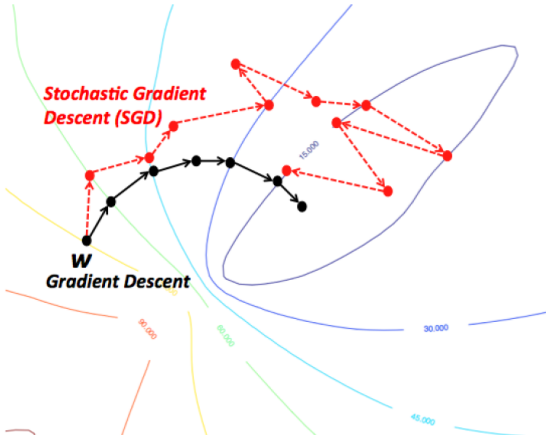
$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \frac{\eta}{n} \sum_{p=1}^{n} \nabla J_p(\boldsymbol{w})$$

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

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

One-example based Backpropagation

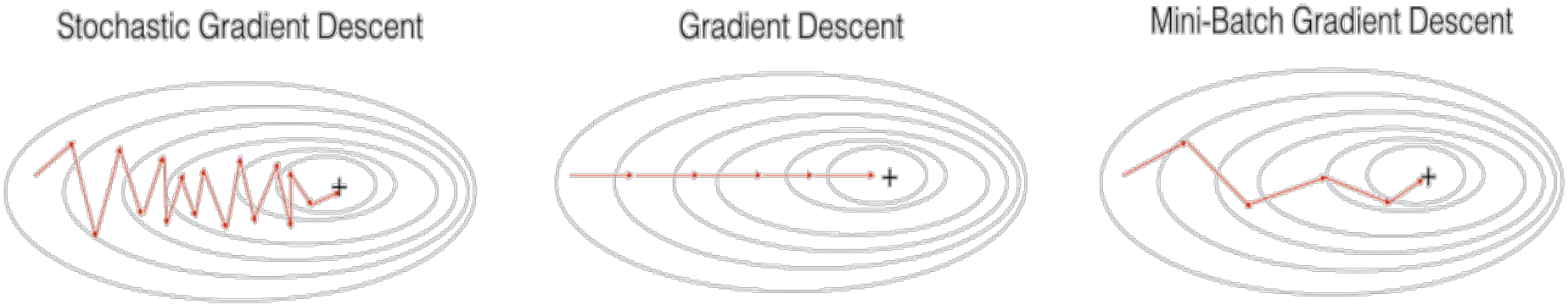
- Algorithm 1 (one-example based BP)
  - 1 **begin initialize** network topology (n_H), $\boldsymbol{w}$, criterion θ, η, $m \leftarrow 0$
  - 2 **do** $m \leftarrow m + 1$
  - 3 $x^m \leftarrow$ randomly chosen pattern
  - 4 $w_{ji} \leftarrow w_{ji} + \eta\delta_j x_i; \quad w_{kj} \leftarrow w_{kj} + \eta\delta_k y_j$
  - 5 **until** $\nabla J(\boldsymbol{w}) < \theta$
  - 6 **return** $\boldsymbol{w}$
  - 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.

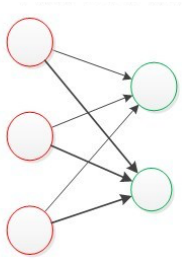


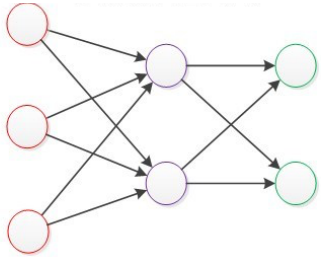
Mini-Batch Backpropagation

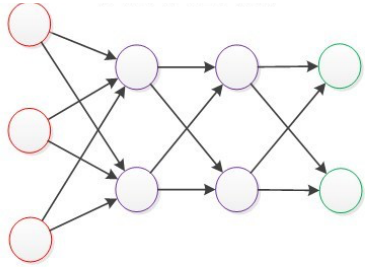
- Algorithm 2 (mini-batch based BP)
  - 1 **begin initialize** network topology (n_H), $\boldsymbol{w}$, criterion θ, η, r $\leftarrow 0$
  - 2 **do** $r \leftarrow r + 1$ (increment epoch)
  - 3 $m \leftarrow 0$; $\Delta w_{ji} \leftarrow 0$; $\Delta w_{kj} \leftarrow 0$;
  - 4 **do** $m \leftarrow m + 1$
  - 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** $\nabla J(\boldsymbol{w}) < \theta$
  - 10 **return** $\boldsymbol{w}$
  - 11 **end**

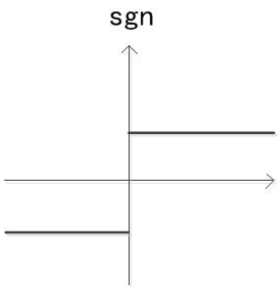
SGD Analysis

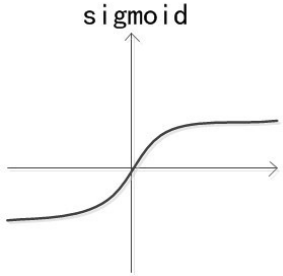
- One-example based SGD
  - Estimation of the gradient is noisy, and the weights may not move precisely down the gradient at each iteration
  - Faster than batch learning, especially when training data has redundancy
  - Noise often results in better solutions
  - The weights fluctuate and it may not 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

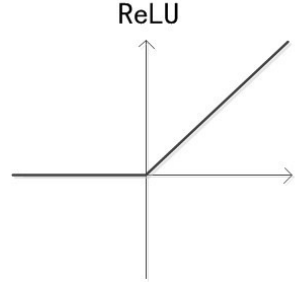


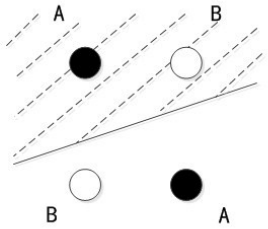


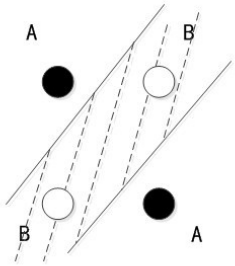


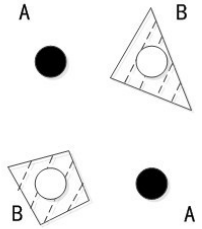

sgn

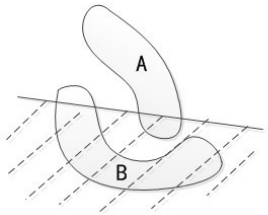

sigmoid

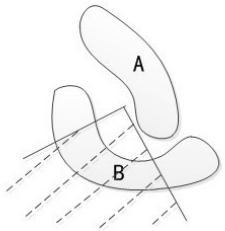

ReLU

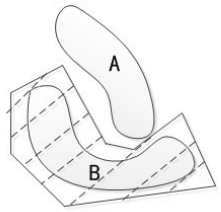

A
B
B
A


A
B
B
A


A
B
B
A

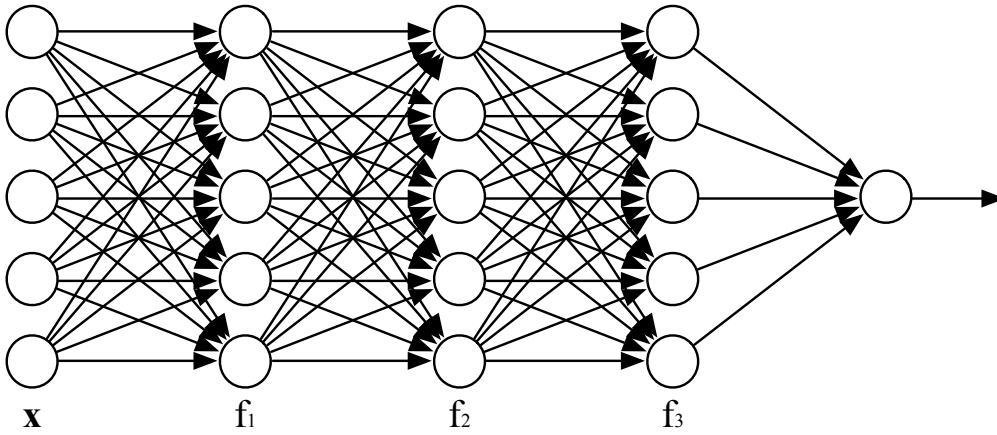

A
B


A
B


A
B

Summarization

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



A toy example

https://github.com/pytorch/examples/blob/master/mnist/main.py

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

https://pytorch.org/docs/stable/nn.html#linear

https://pytorch.org/docs/stable/nn.functional.html#non-linear-activation-functions

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

A toy example

https://github.com/pytorch/examples/blob/master/mnist/main.py

```
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
```

cross_entropy

`torch.nn.functional.cross_entropy(input, target, weight=None, size_average=None, ignore_index=-100, reduce=None, reduction='mean')` [SOURCE]

This criterion combines *log_softmax* and *nll_loss* in a single function.

See `CrossEntropyLoss` for details.

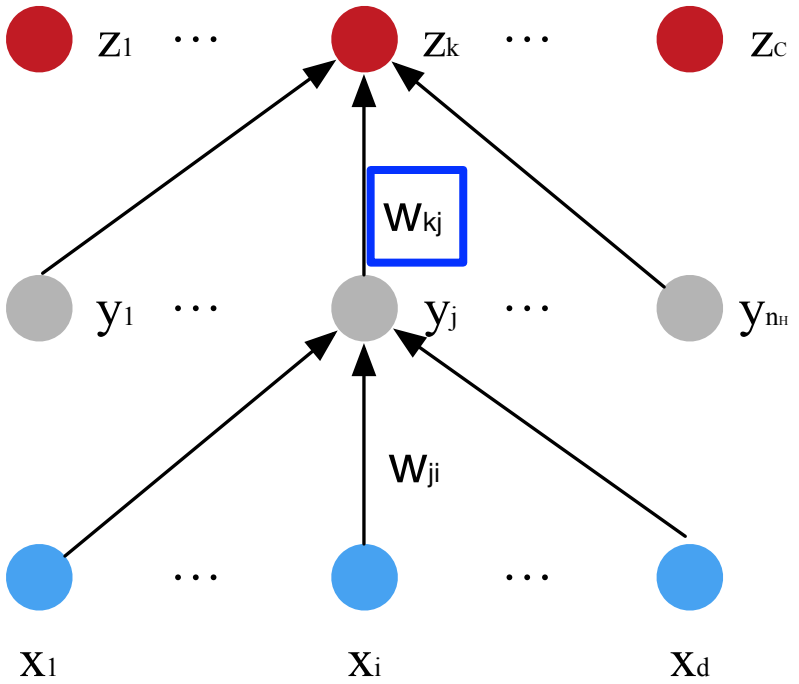
Thank you!

Hidden-to-output weight w_{kj}

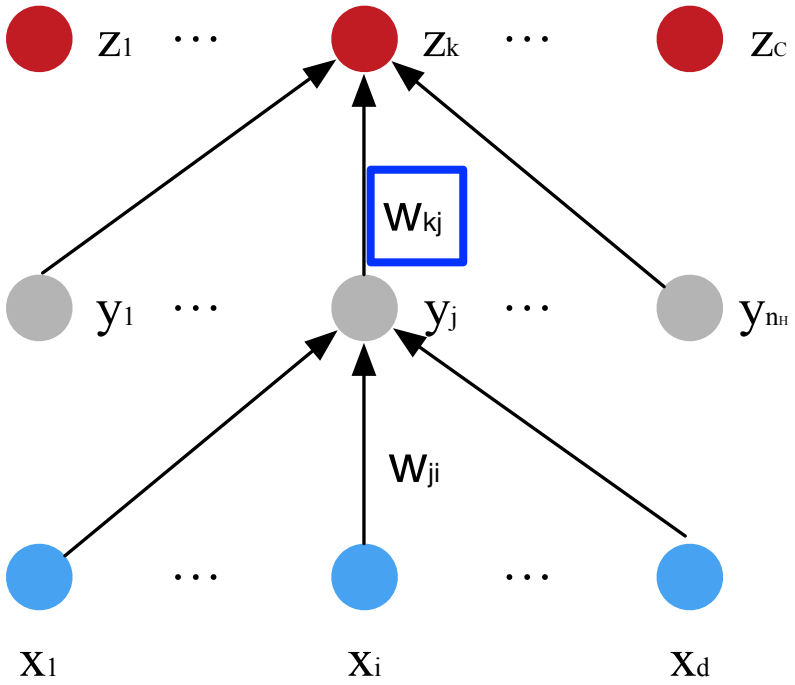
- Chain rule

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

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



Hidden-to-output weight w_{kj}



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

- Define the *sensitivity* of unit k

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

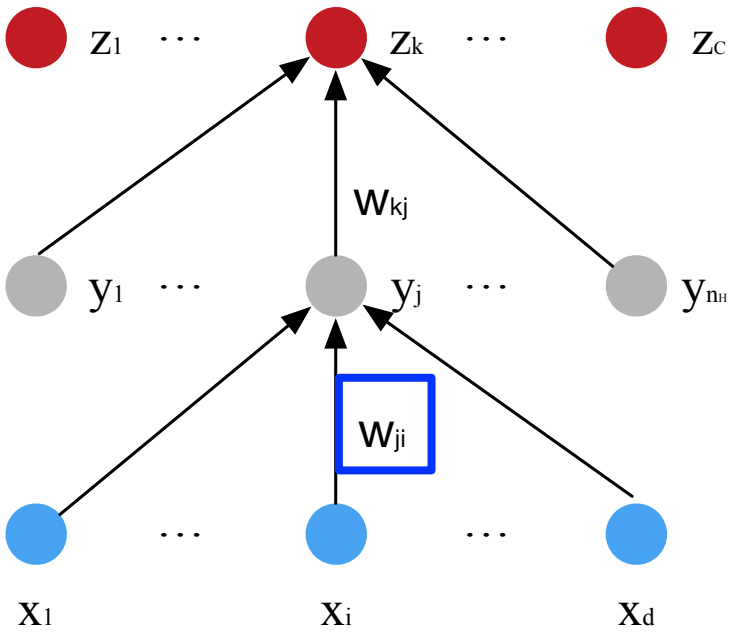
- **Update w_{kj}**

$$\Delta w_{kj} = -\eta \frac{\partial J}{\partial net_k} y_j = \eta \delta_k y_j$$

Input-to-hidden weight w_{ji}

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

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



$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0}$$

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

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

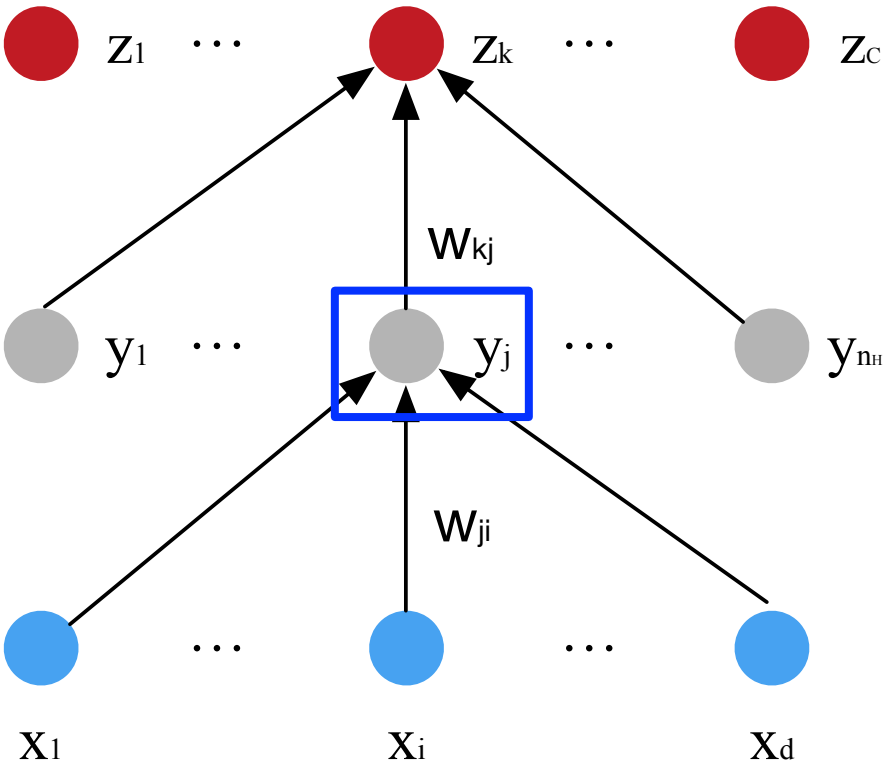
$$= -\sum_{k=1}^{c}(t_k - z_k)f'(net_k)\,w_{kj}$$

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

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.

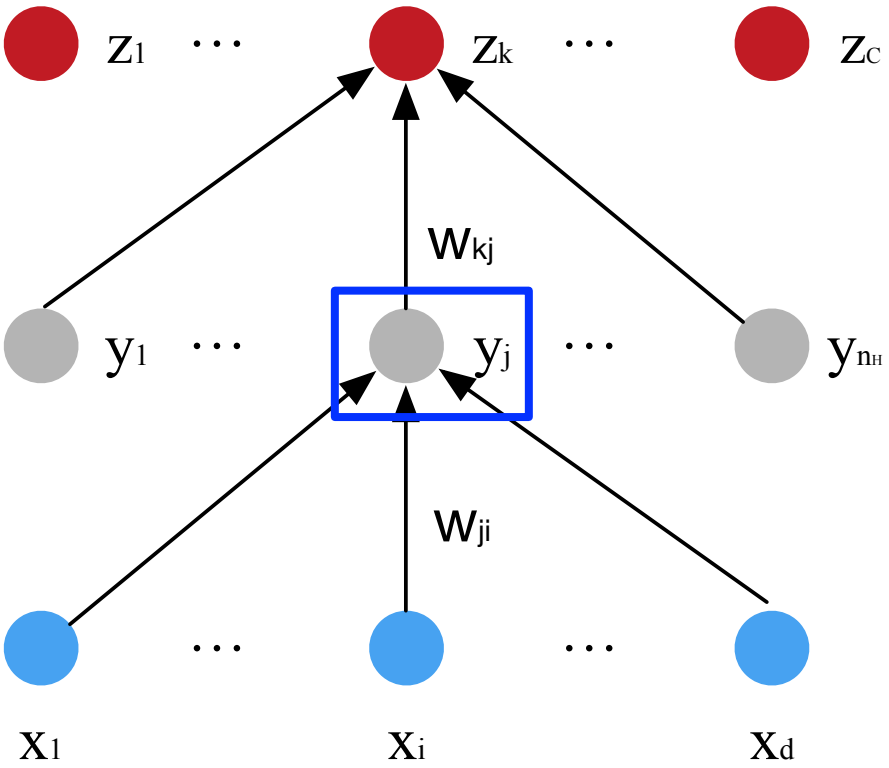
The output unit sensitivities are propagated "back" to the hidden units.

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

Input-to-hidden weight w_{ji}

- Sensitivity for a hidden unit j

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



Update w_{ji}

$$\Delta w_{ji} = -\eta\frac{\partial J}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} = \eta\delta_j x_i$$

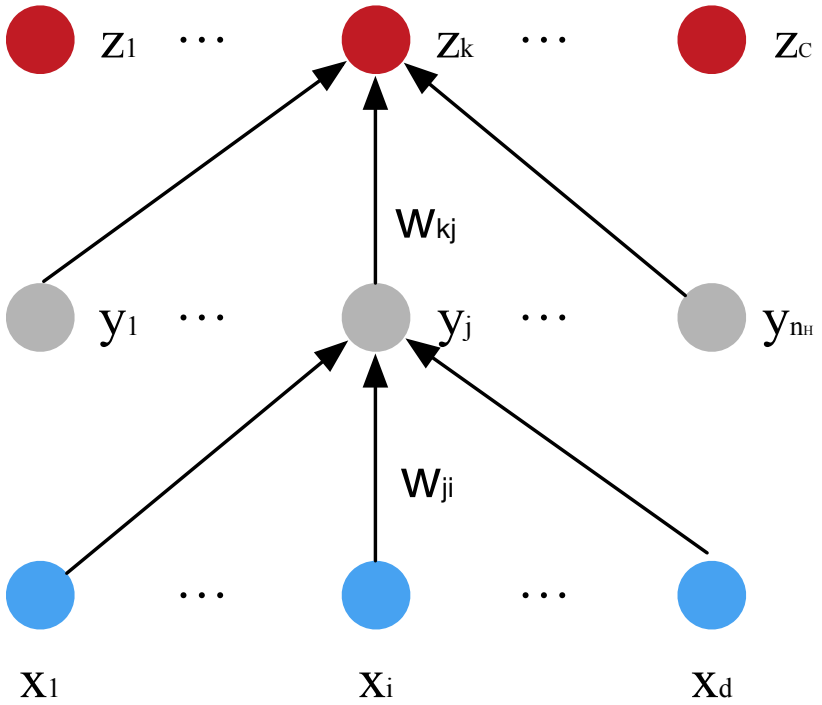
$$\left(net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} = \boldsymbol{w}_j^T\boldsymbol{x}\right)$$

Backpropagation summary

$$\delta_k = -\frac{\partial J}{\partial net_k} \qquad \delta_j = -\frac{\partial J}{\partial net_j}$$

- Hidden-to-output weight

$$\Delta w_{kj} = \eta \delta_k \, y_j$$
$$= \eta (t_k - z_k) f'(net_k) y_j$$



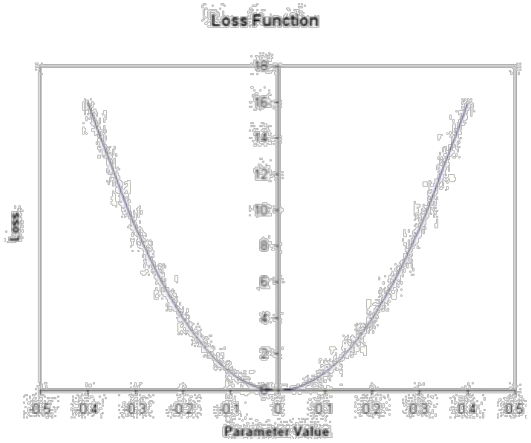
- Input-to-hidden

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

Example 1

Squared loss function:

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



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

$$\frac{\partial z_k}{\partial net_k} = \frac{\partial f(net_k)}{\partial net_k}$$

$$= -\frac{-e^{-net_k}}{(1+e^{-net_k})^2}$$

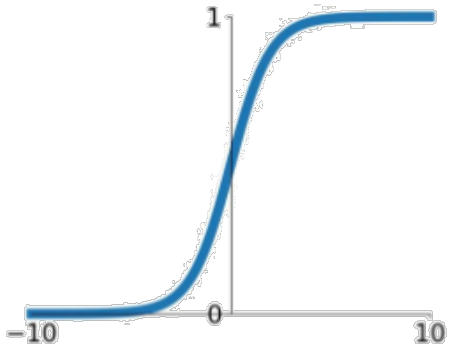
$$= \frac{1+e^{-net_k}-1}{(1+e^{-net_k})^2}$$

$$= \frac{1}{1+e^{-net_k}} - \frac{1}{(1+e^{-net_k})^2}$$

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

Sigmoid activation function:

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



- The *sensitivity* of unit k is

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k)z_k(1-z_k)$$

[Vanishing gradient] When the output z_k approaches to 0 or 1 (*i.e.* saturate), $\frac{\partial z_k}{\partial net_k}$ 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:

$$z_k = \frac{e^{net_k}}{\sum_{i=1}^{C} e^{net_i}}$$

$$\frac{\partial z_k}{\partial net_k} = \frac{\partial\left(e^{net_k} \cdot \frac{1}{\sum_{i=1}^{C} e^{net_i}}\right)}{\partial net_k}$$

$$= \frac{\partial(e^{net_k})}{\partial net_k} \cdot \frac{1}{\sum_{i=1}^{C} e^{net_i}} + \frac{\partial\left(\frac{1}{\sum_{i=1}^{C} e^{net_i}}\right)}{\partial net_k} e^{net_k}$$

$$= \frac{e^{net_k}}{\sum_{i=1}^{C} e^{net_i}} + \frac{e^{net_i}}{\left(\sum_{i=1}^{C} e^{net_i}\right)^2} e^{net_k} = z_k - z_k^2$$

- The *sensitivity* of unit k is

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = t_k(1 - z_k)$$