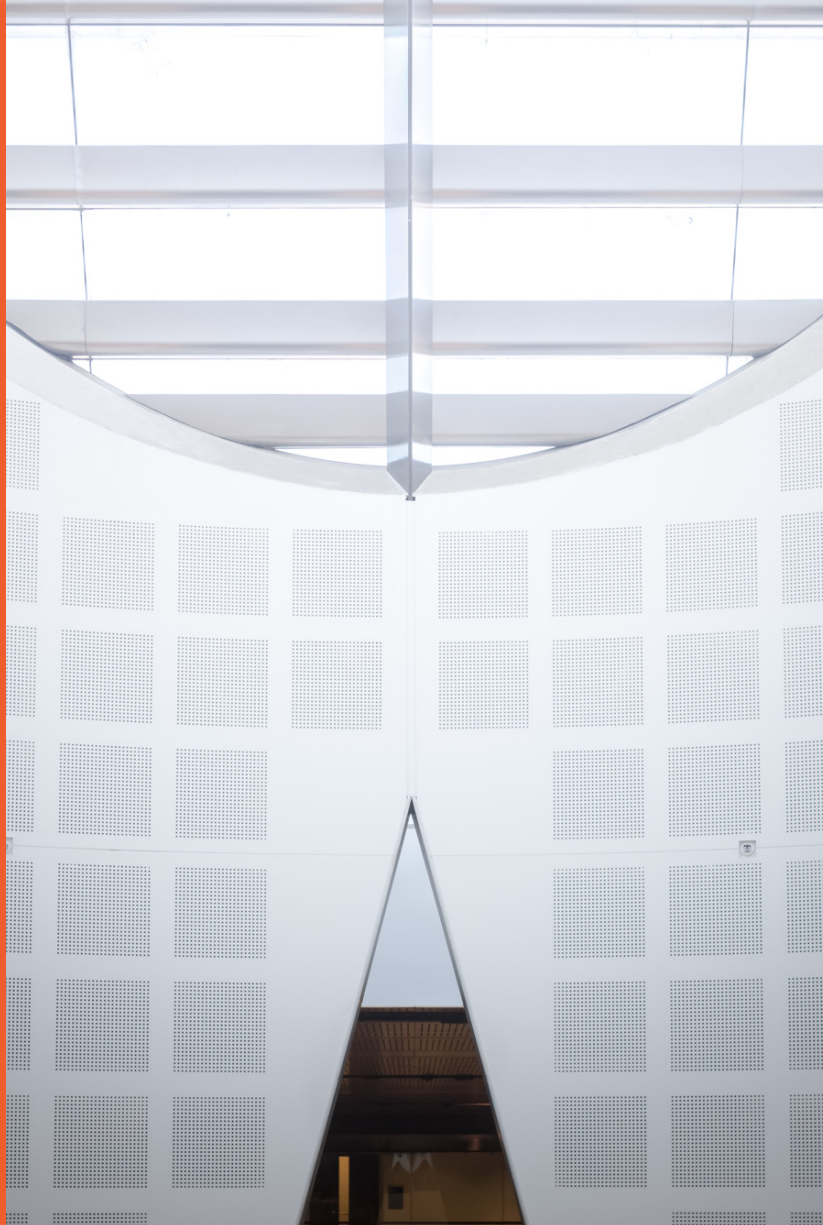
Multilayer Neural Network

Dr Chang Xu

School of Computer Science

THE UNIVERSITY OF SYDNEY



Perceptron: the Prelude of Deep Learning

A neuron cell

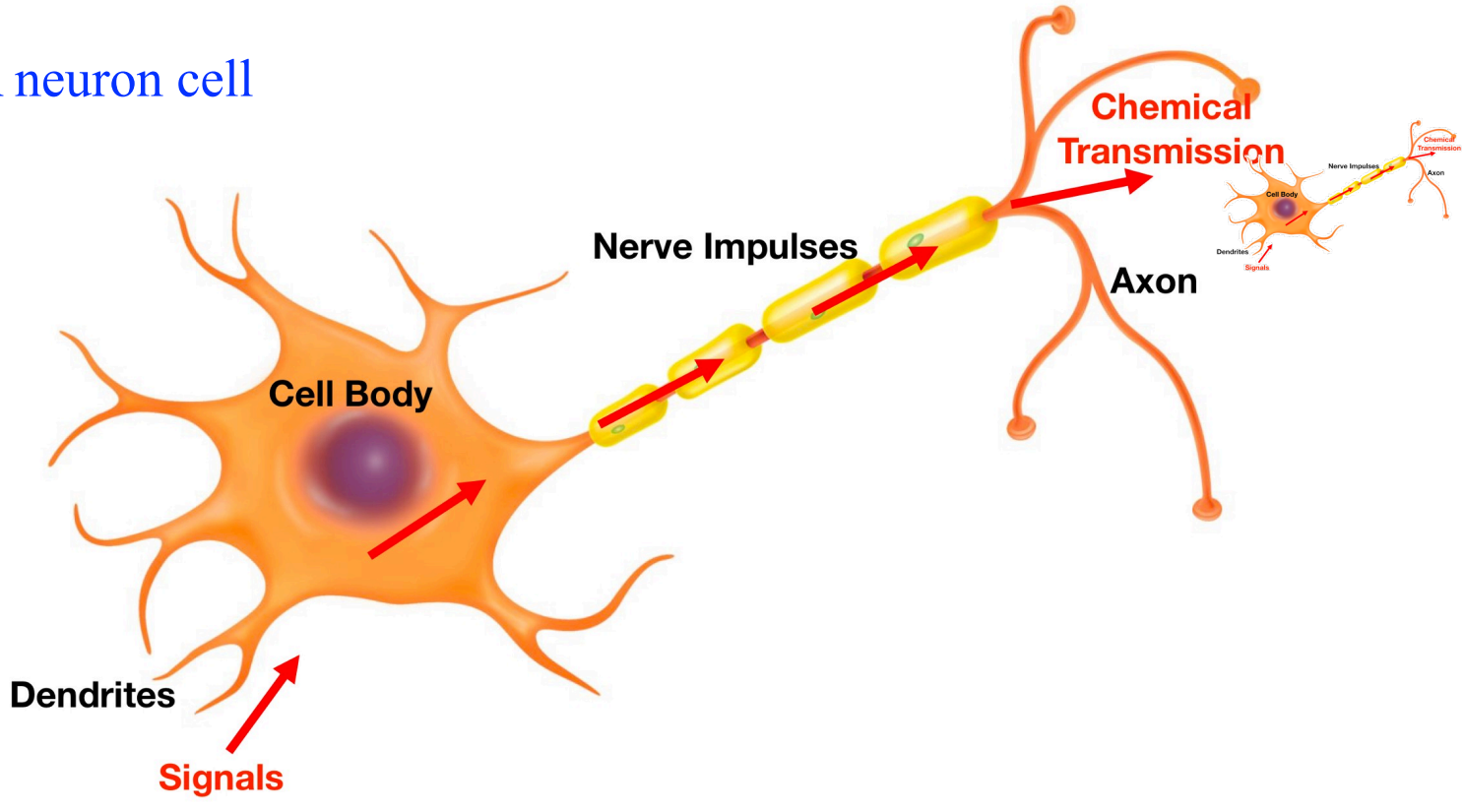
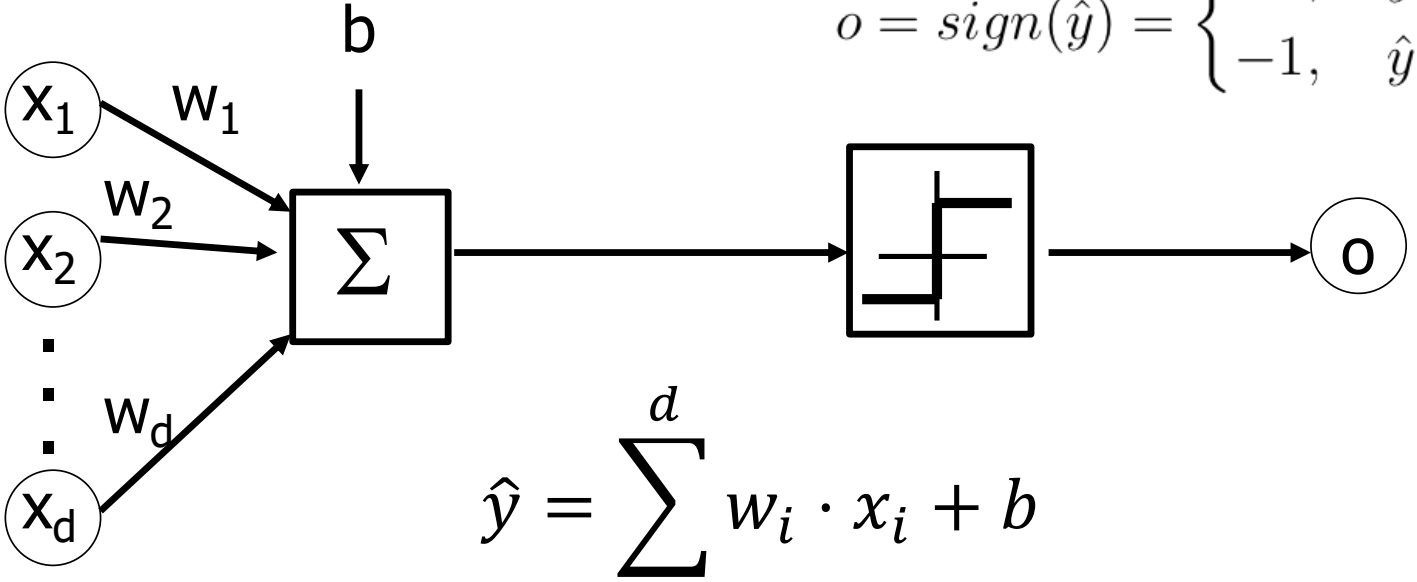


Image credit to: thinglink.com

Perceptron: the Prelude of DL

[Rosenblatt, et al. 1958]



$$o = sign(\hat{y}) = \begin{cases} +1, & \hat{y} \geq 0 \\ -1, & \hat{y} < 0 \end{cases}$$

$$\hat{y} = \sum_{i=1}^{d} w_i \cdot x_i + b$$

Perceptron

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

weight vector

input vector

Neuron's estimate of the desired output

$$\hat{y} = \sum_{i=1}^{d} w_i x_i + b = w^T x + b$$

bias

Recall the decision rule $o = sign(\hat{y}) = \begin{cases} +1, & \hat{y} \geq 0 \\ -1, & \hat{y} < 0 \end{cases}$

If an example can be correctly classified, we have

sign of ground truth sign of prediction

$$y(w^T x + b) > 0$$

correct classification

otherwise

$$y(w^T x + b) < 0$$

Perceptron

If an example can be correctly classified, we have

$$y(w^T x + b) > 0$$

otherwise

$$y(w^T x + b) < 0$$

The objective function of Perceptron can be written as

$$\min_{w,b} L(w, b) = -\sum_{x_i \in \mathcal{M}} y_i(w^T x_i + b)$$

where $\mathcal{M}$ stands for the set of the misclassified examples.

Perceptron

$$\min_{w,b} L(w,b) = -\sum_{x_i \in \mathcal{M}} y_i (w^T x_i + b)$$

Gradient descent to solve the optimization problem.

$$\nabla_w L(w,b) = -\sum_{x_i \in \mathcal{M}} y_i x_i$$

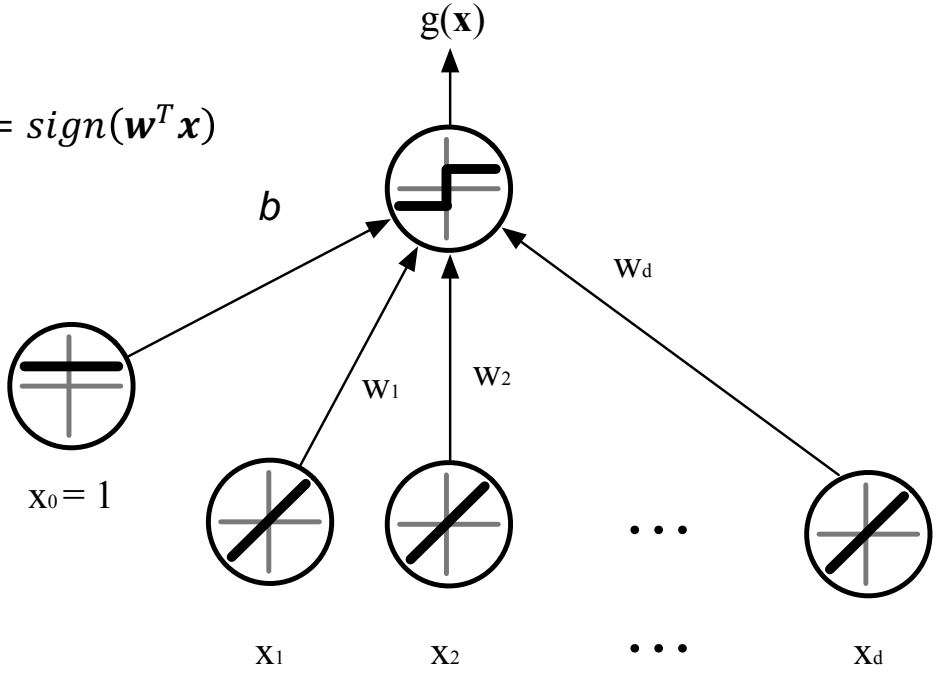
$$\nabla_b L(w,b) = -\sum_{x_i \in \mathcal{M}} y_i$$

Perceptron

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

linear

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



Linear classifier!

Perceptron "AI winter"

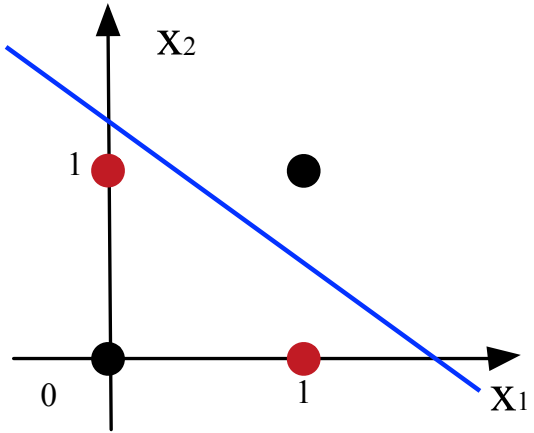
Marvin Minsky & Seymour Papert (1969). *Perceptrons*, MIT Press, Cambridge, MA.

- 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.

$$w_1 x_1 + w_2 x_2 + b$$

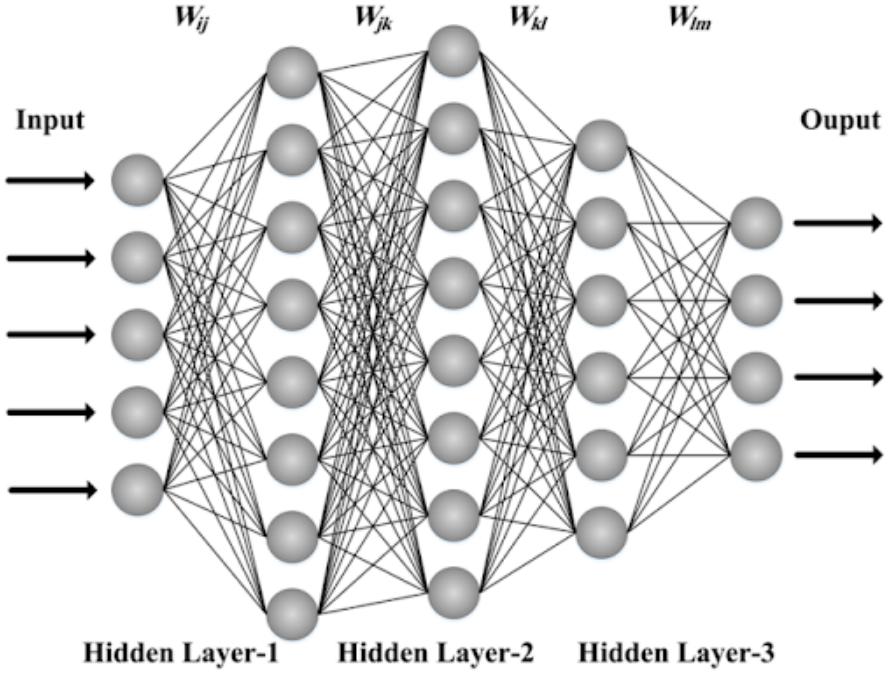
This failure caused the majority of researchers to walk away.

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

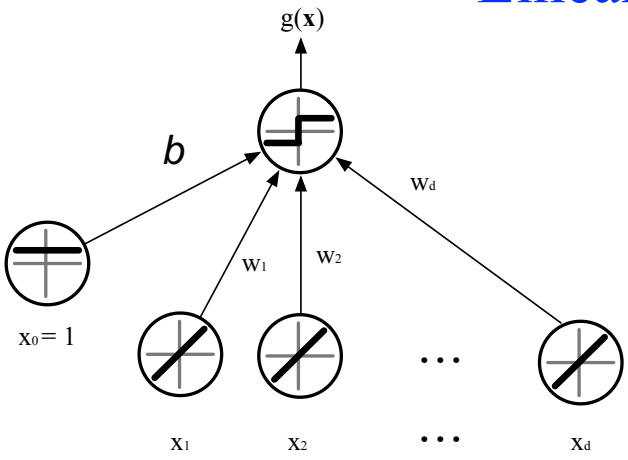
Multilayer Neural Networks



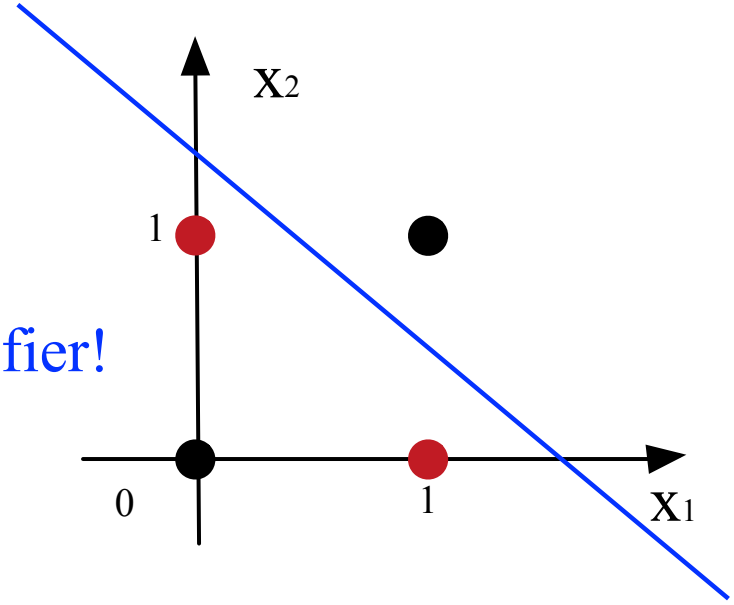
- Function composition can be described by a directed acyclic graph (hence feedforward networks)
- Depth is the maximum i in the function composition chain
- Final layer is called the output layer

stands for "probability of nonlinearity in the neural network"

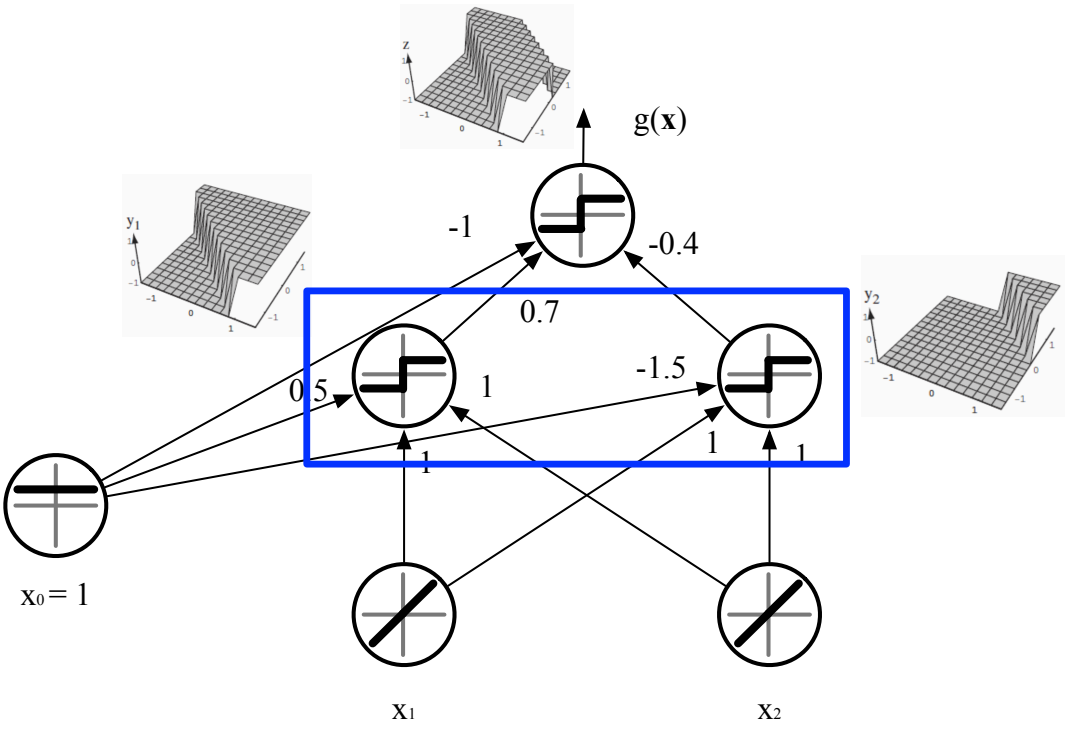
Two-layer Neural Network

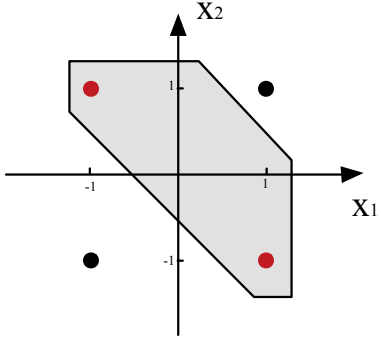


Linear classifier!



Three-layer Neural Network (with a hidden layer)

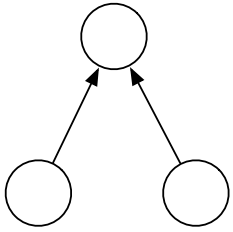


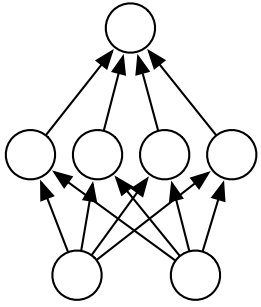


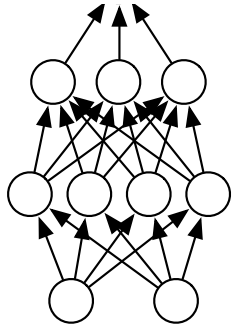
Three-layer Neural Network (with a hidden layer)

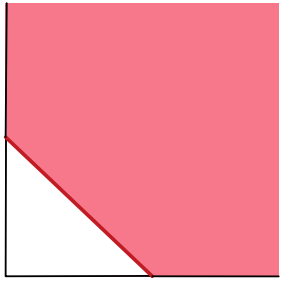
- A 2D-input example

more hidden neurons / layers

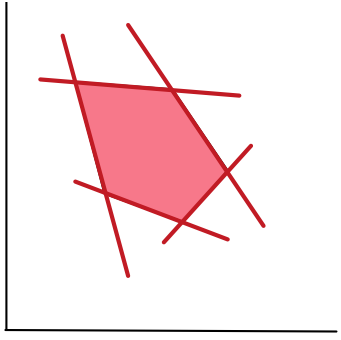




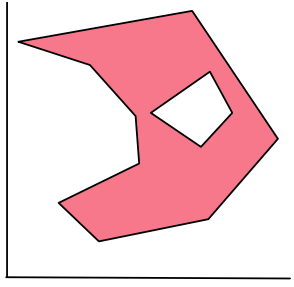




separating hyperplane

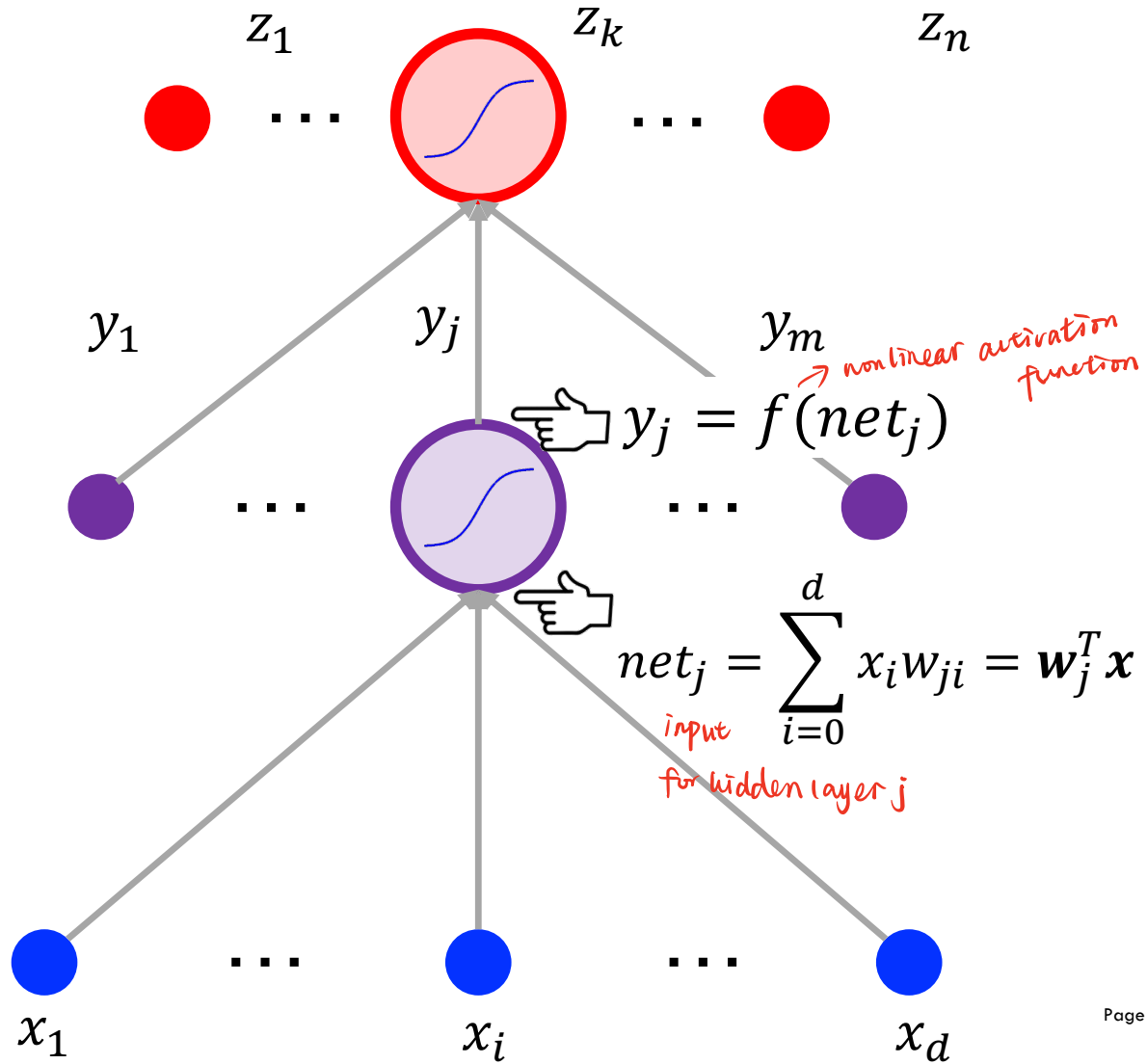


convex polygon region



composition of polygons

Feedforward



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,

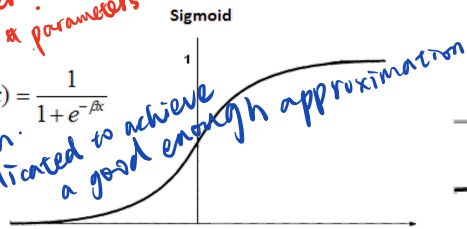
$$\left|\hat{F}(x) - F(x)\right| < \varepsilon$$

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

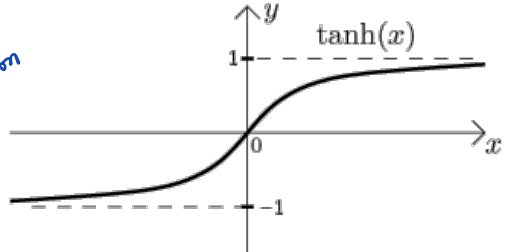
If the dataset is not that large, we need to control the complexity to avoid overfitting

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

Why not just 1 hidden layer? ⇒ multiple layers can reduce complexity & # parameters



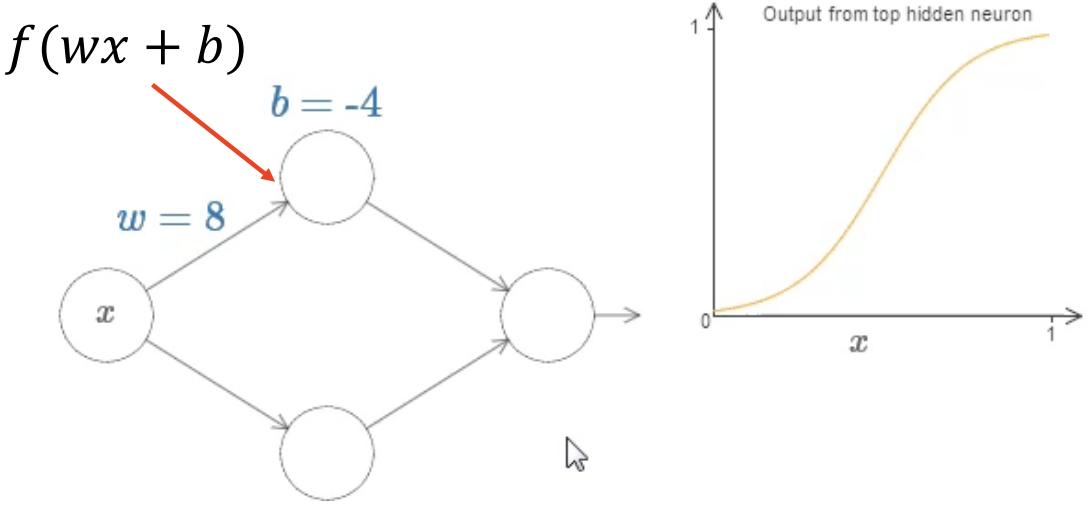
Rigorously: complicated to achieve a good enough approximation



We can always find a neural network to approximate a given function

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

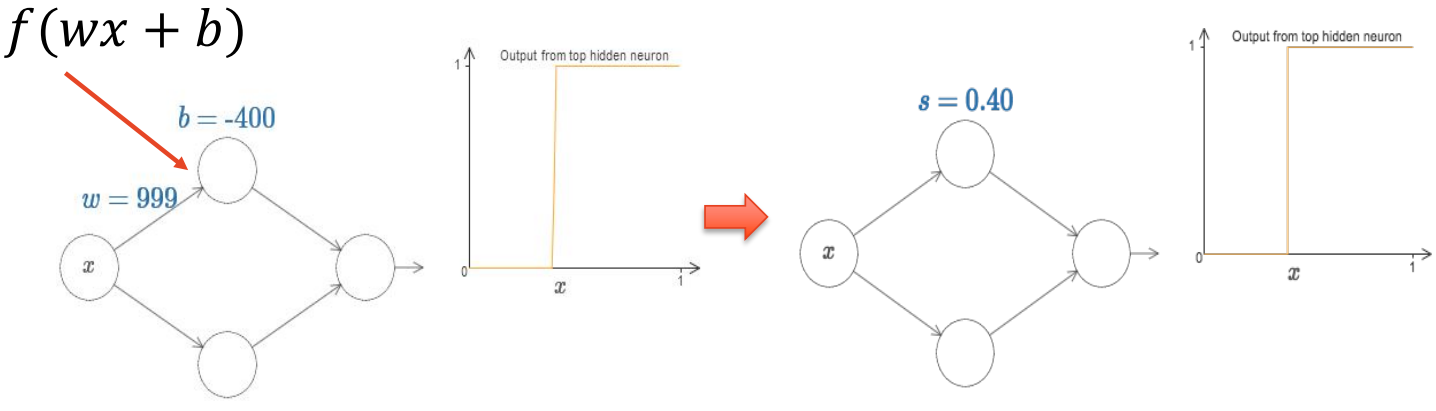


Image credit to: https://neuralnetworksanddeeplearning.com

- 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

add more neurons ⇒ have more bumps

- A visual explanation on one input and one output functions

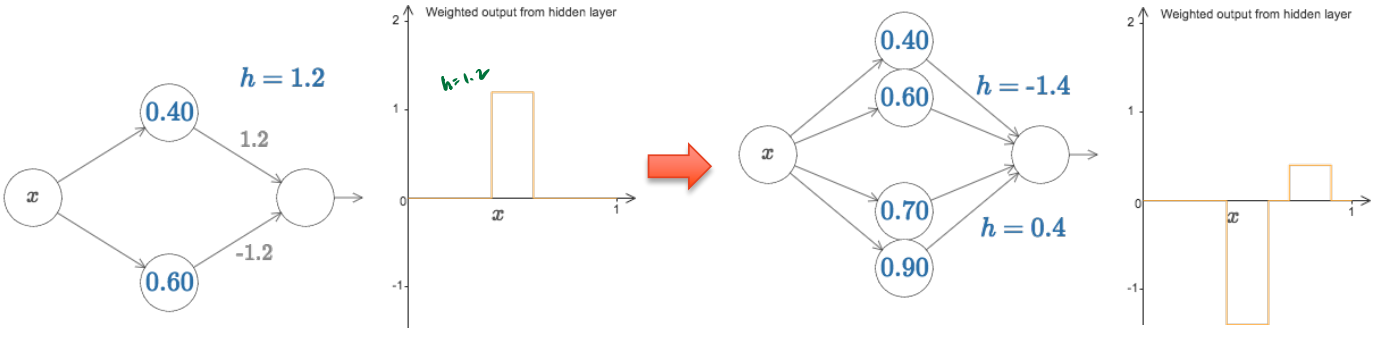


Image credit to: https://neuralnetworksanddeeplearning.com

- *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

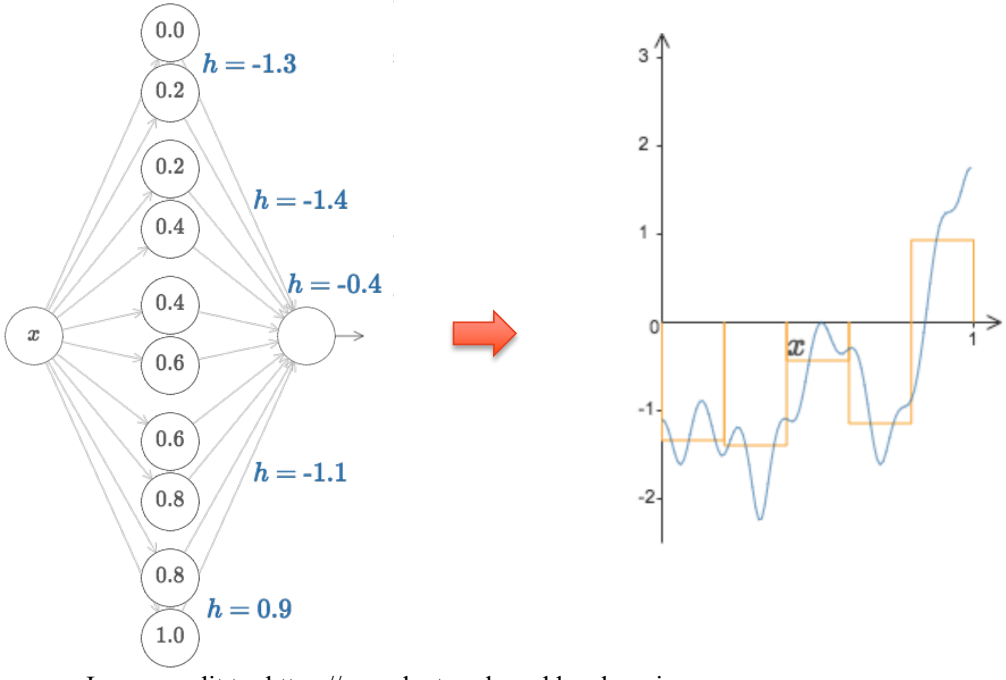


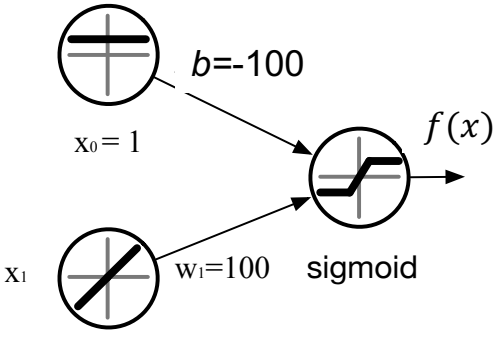
Image credit to: https://neuralnetworksanddeeplearning.com

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

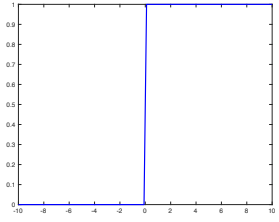
more bumps

Expressive power of a three-layer neural network

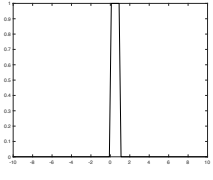
- Overall procedure



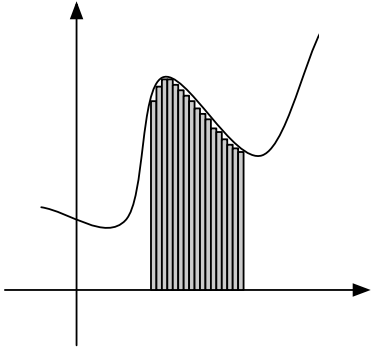
step-like functions



more hidden neurons



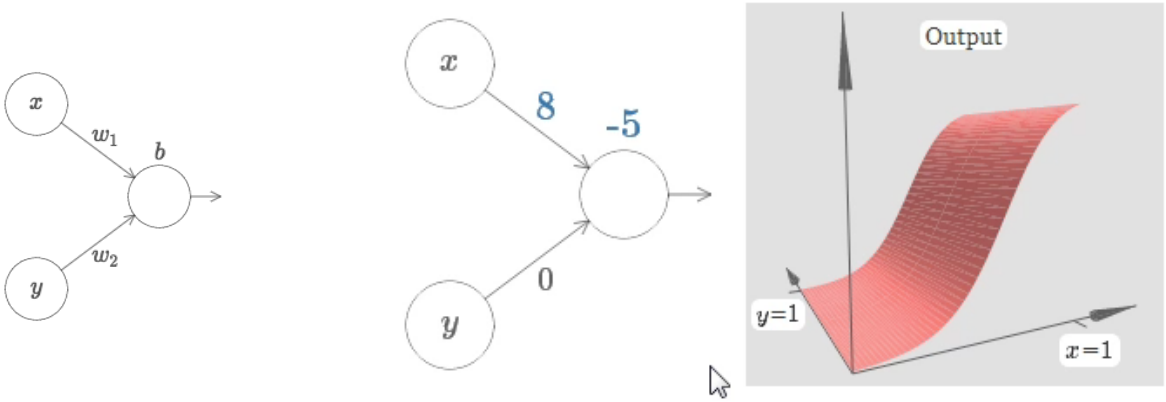
bump functions



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:



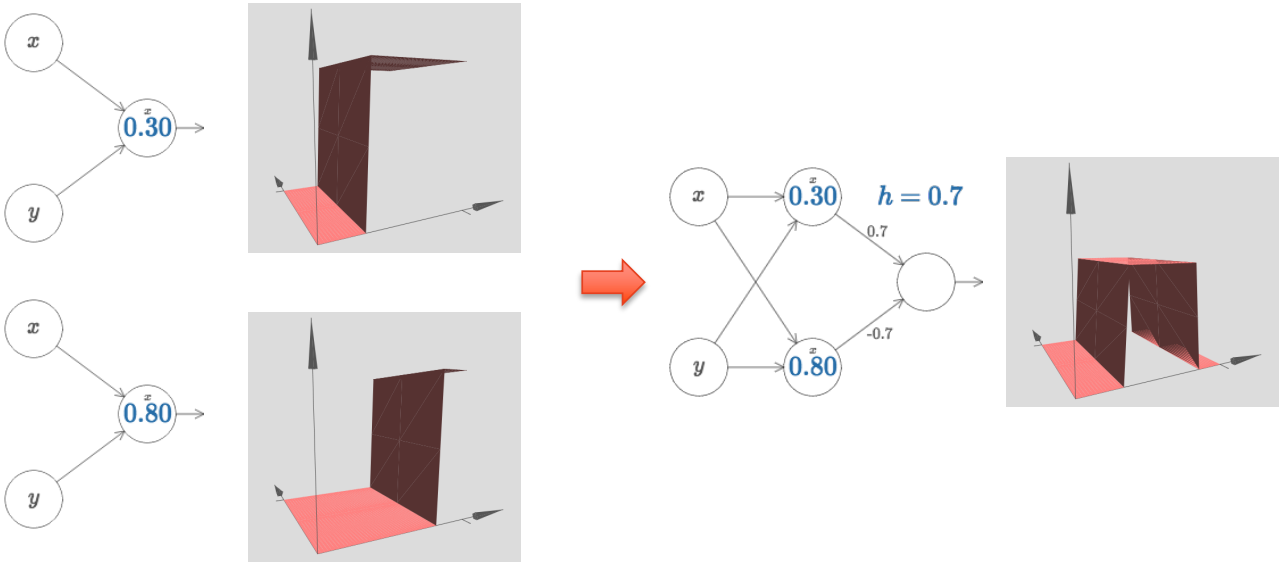
Video credit to: https://neuralnetworksanddeeplearning.com

  - 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

in 3D.

- 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:

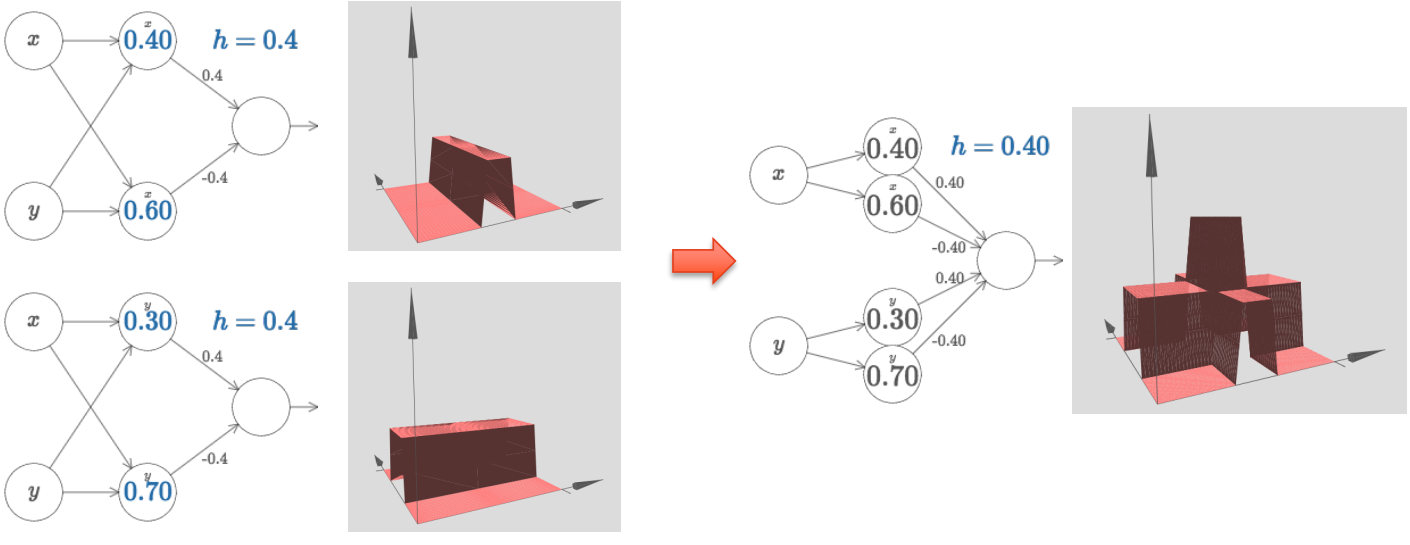


Image credit to: https://neuralnetworksanddeeplearning.com

Expressive power of a three-layer neural network

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

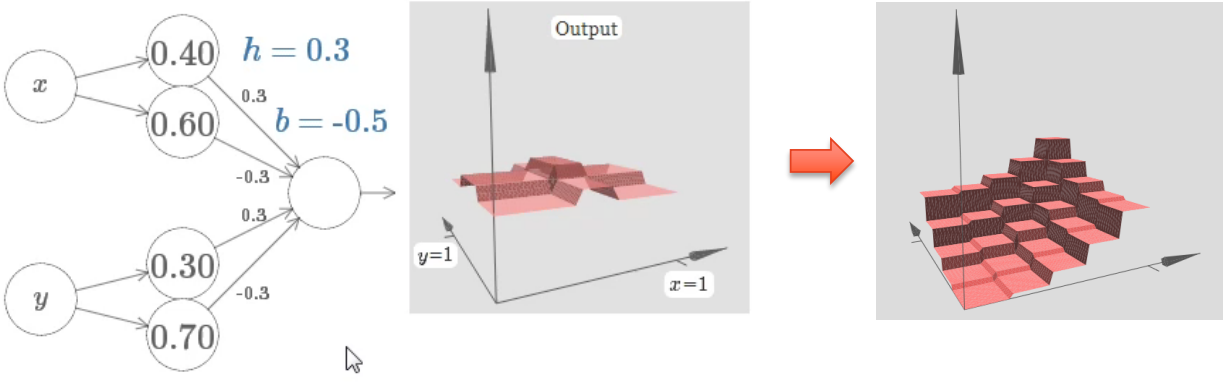


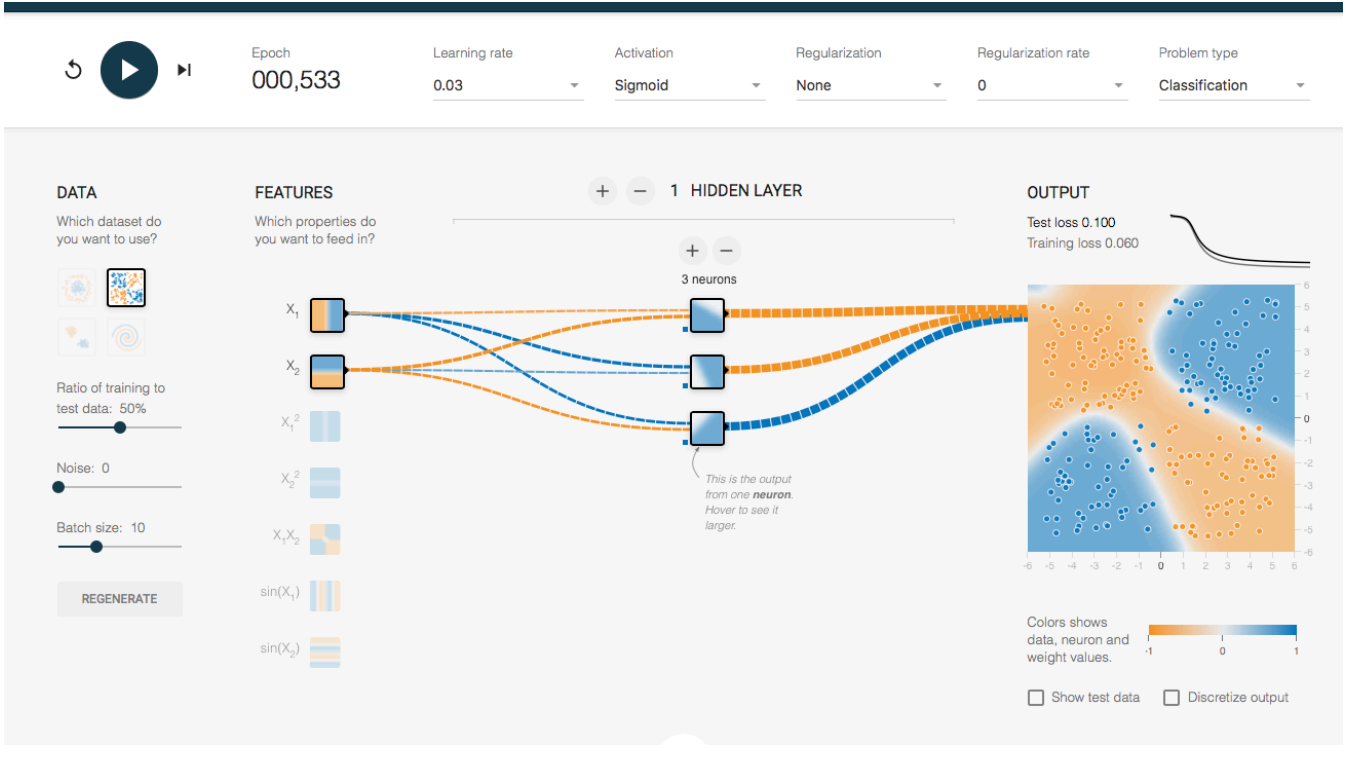
Image credit to: https://neuralnetworksanddeeplearning.com

- 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.

A toy example

http://playground.tensorflow.org/

1. Linear case without an hidden layer
2. Nonlinear case with a hidden layer



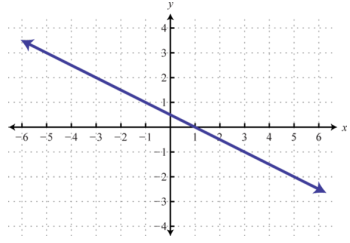
Activation Functions

Activation functions

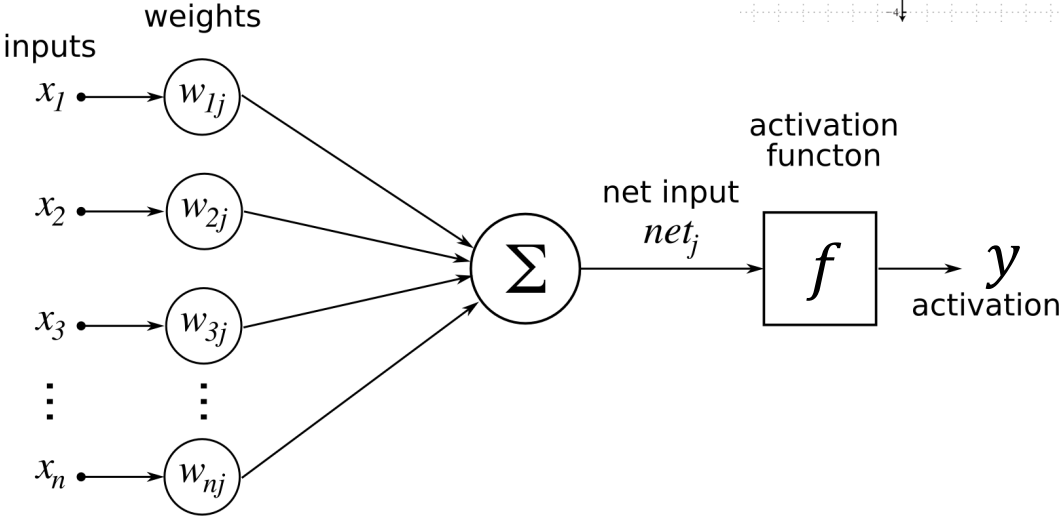
introduce nonlinearity

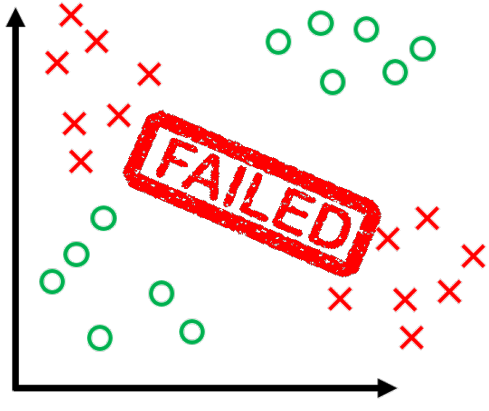
- Must be nonlinear: otherwise it is equivalent to a linear classifier

$w^T x$



Linear function!





Activation functions

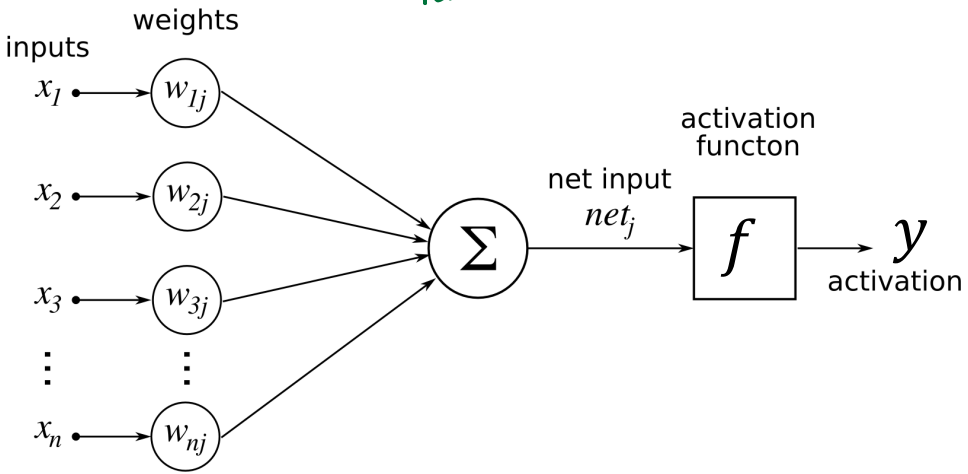
we need to calculate gradient (use gradient descent)

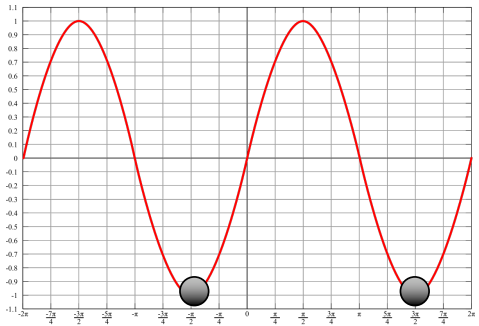
- Continuous and differentiable **almost everywhere**
- Monotonicity: otherwise it introduces additional local extrema in the error surface

激活函数如果单调的时候，可以保证是凸函数.

expect monotonicity for activation function

reduce probability to discover local minimum



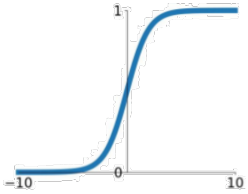


An output value might correspond to multiple input values.

Activation function $f(s)$

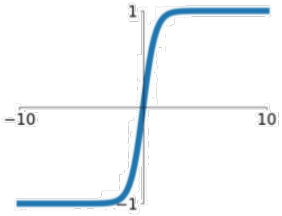
- Popular choice of activation functions (single input)
  - Sigmoid function

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



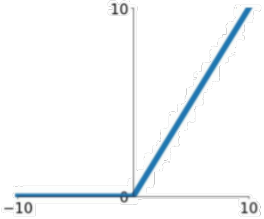
  - Tanh function: shift the center of Sigmoid to the origin

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



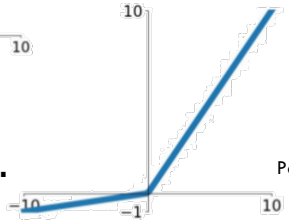
  - Rectified linear unit (ReLU)

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



  - 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

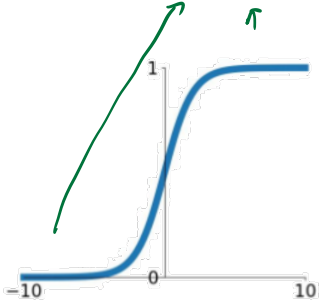
- Saturation.

Saturation

flat at extreme ⇒ no gradient ⇒ no update

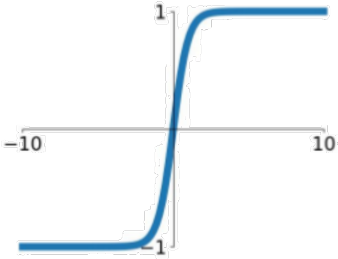
Sigmoid function

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



Tanh function

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

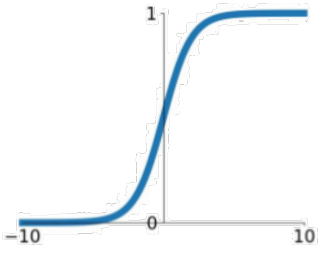


❖ At their extremes, their derivatives are close to 0, which kills gradient and learning process

Not zero centered activation

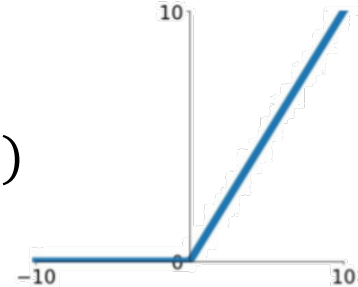
Sigmoid function

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



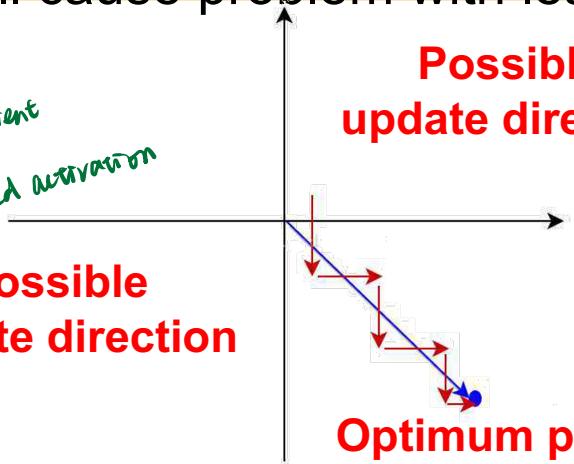
Relu function

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



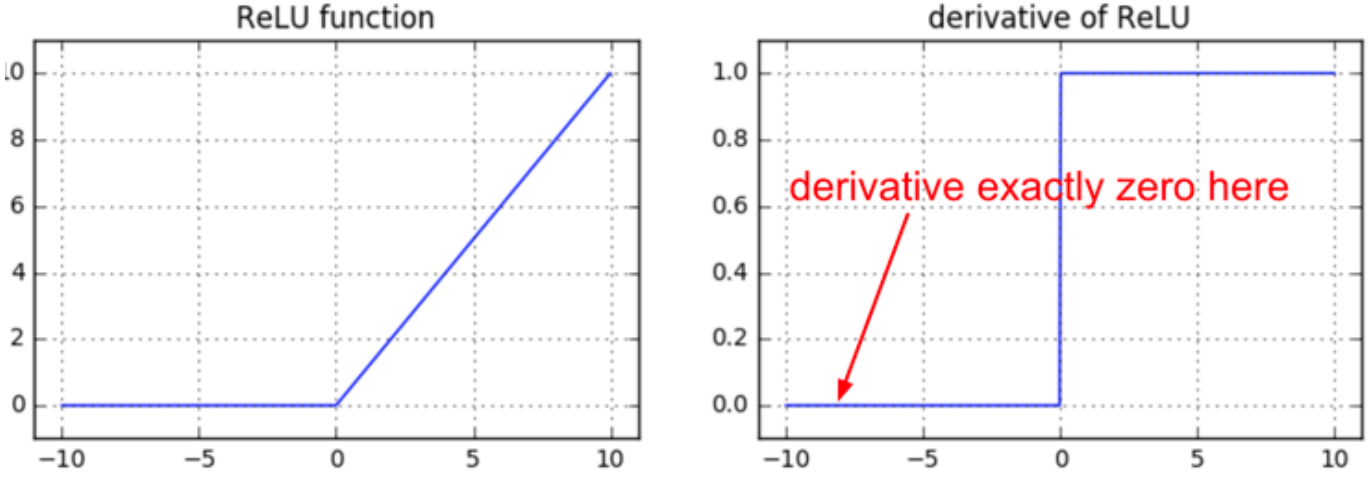
- During training, all gradients will be all positive or all negative that will cause problem with learning.

in mini-batch example
some may give positive gradient
others may give negative gradient
⇒ mitigate the not zero centered activation



Dead ReLUs

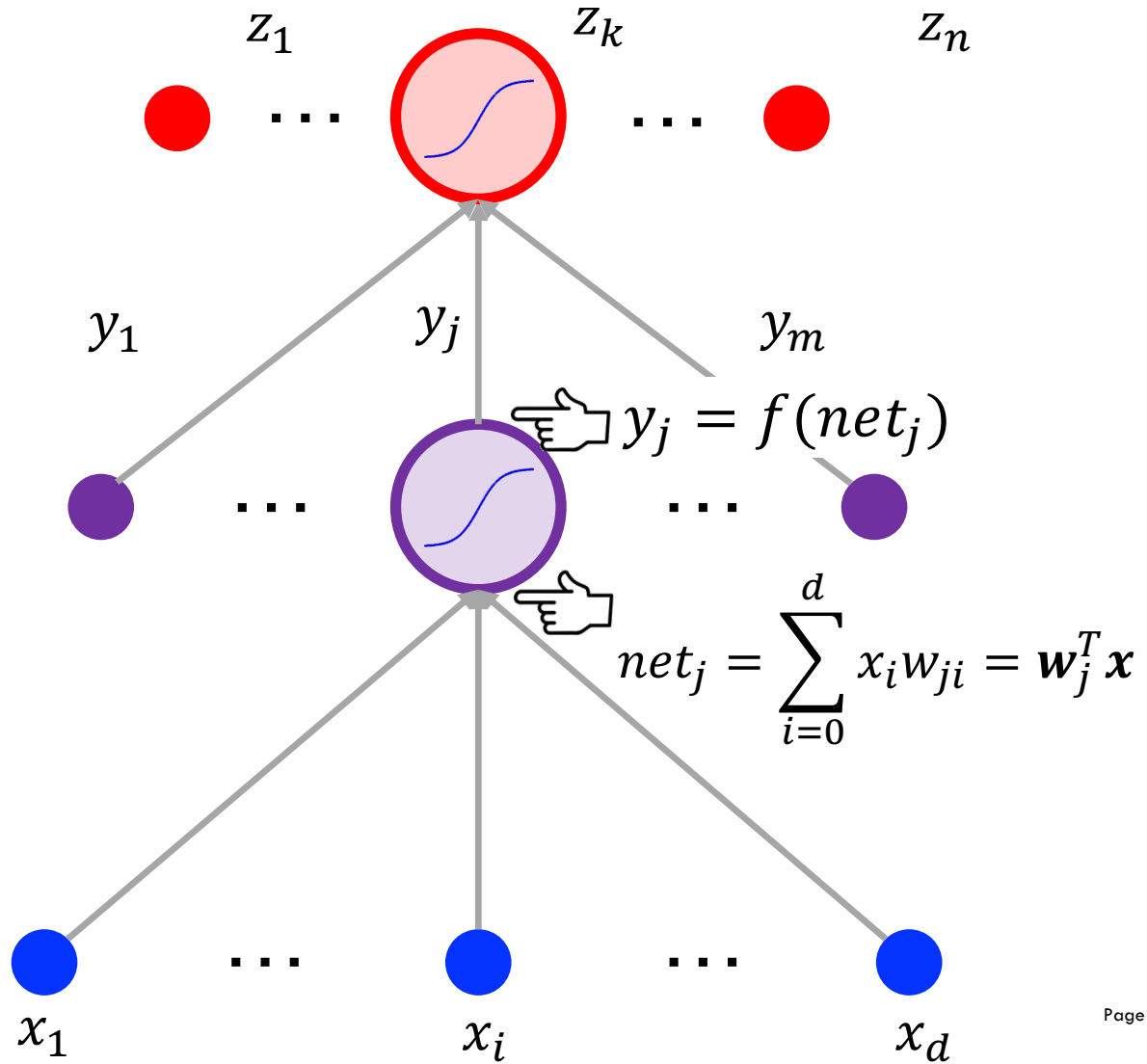
ReLU function $f(s) = \max(0, s)$



- 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

Recall the feedforward calculation



Training error

- Euclidean distance.

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

ground truth / target

output / prediction

- 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 $\boldsymbol{z}$,

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

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

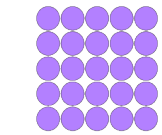
when $t_i = z_i$, $\log 1 = 0$; KL divergence min

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

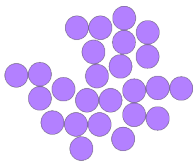
$$= Entropy(\boldsymbol{t}) + D_{KL}(\boldsymbol{t}|\boldsymbol{z})$$

↑ fixed, since t is ground truth

Entropy is a measure of degree of disorder or randomness.

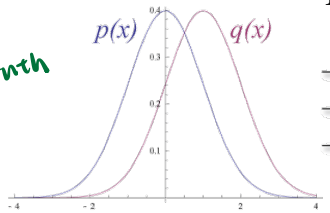


Low Entropy

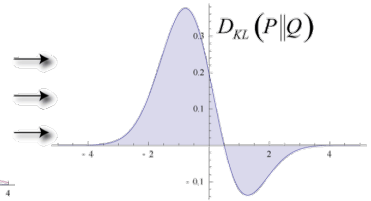


High Entropy

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



Original Gaussian PDF's

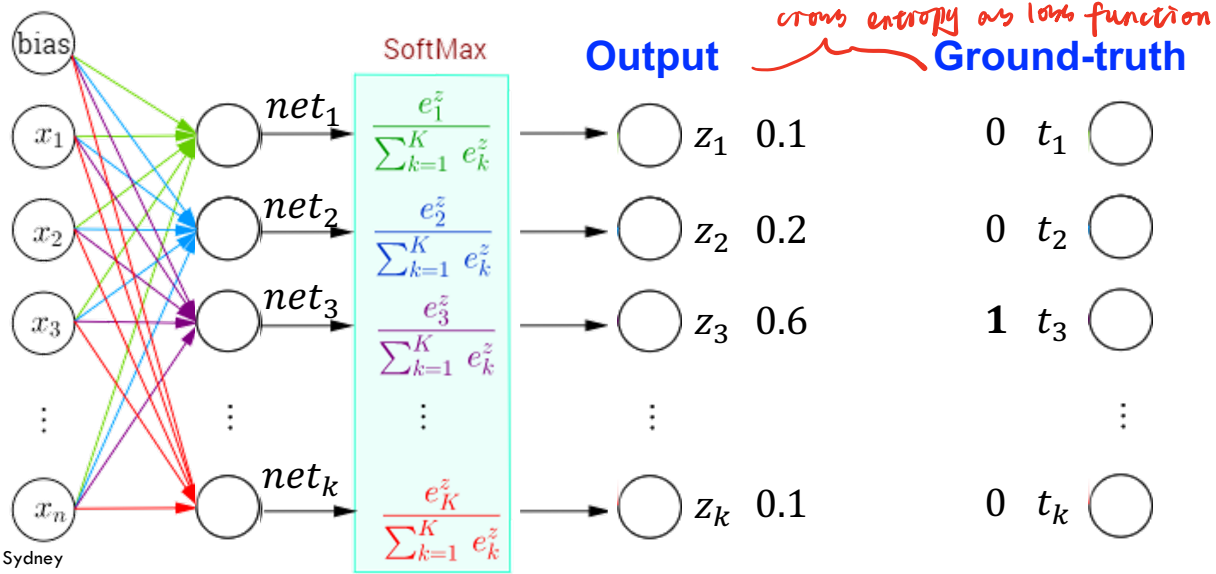


KL Area to be Integrated

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

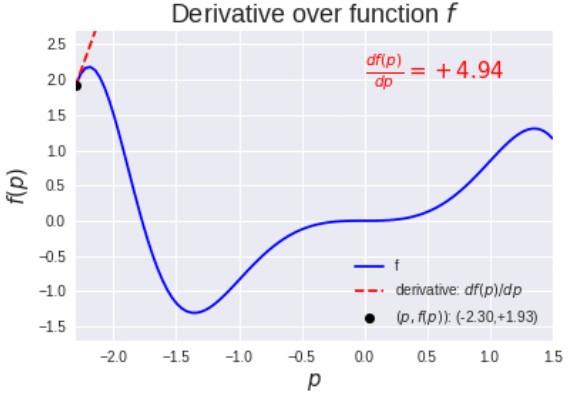
- Weights are initialized with random values, and then are updated in a direction reducing the error.

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

where η is the learning rate.

Iteratively update

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



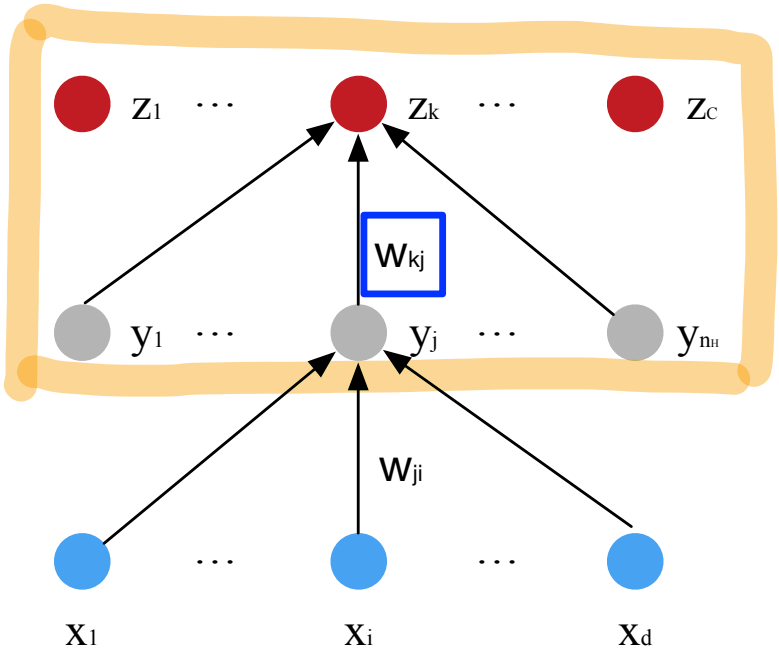
Hidden-to-output weight w_{kj}

- Chain rule

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

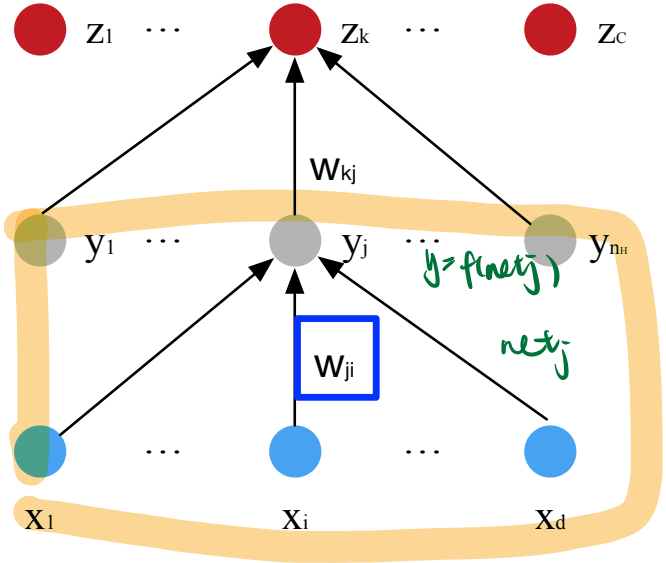
$$\begin{cases} J(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{t} - \boldsymbol{z}\|^2 \\ z_k = f(net_k) \\ net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} \end{cases}$$

euclidean



Input-to-hidden weight w_{ji}

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



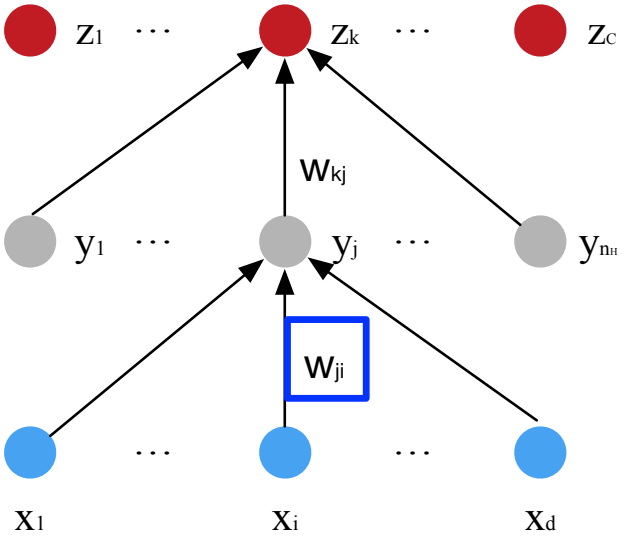
$$\frac{\partial J}{\partial y_j} = \sum_{k=1}^{c}\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial y_j}$$

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

$$y_j = f(net_j)$$

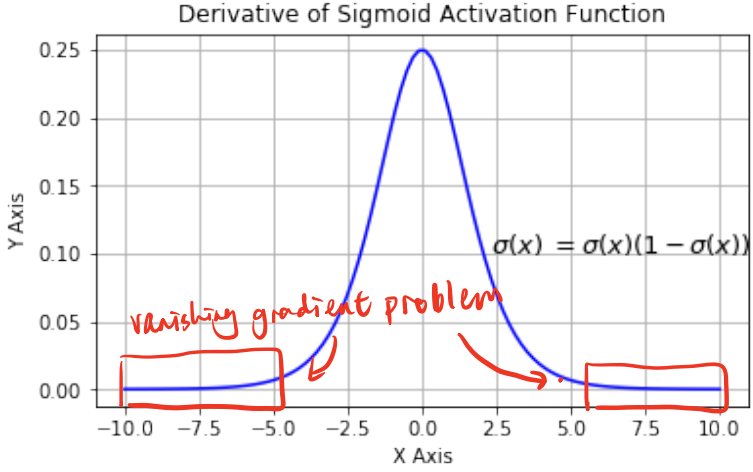
Vanishing gradients problem

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



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

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

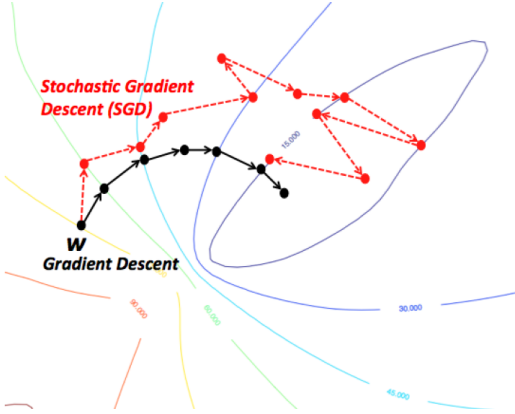


Stochastic Gradient Descent

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

all n element

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

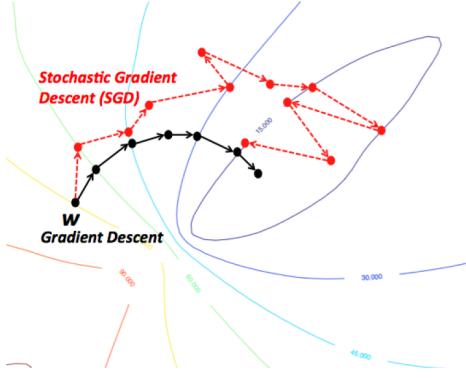
- In SGD, the gradient of $J(\boldsymbol{w})$ is approximated on a single example or a mini-batch of examples

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

in this case, only one example.

One-example based Backpropagation

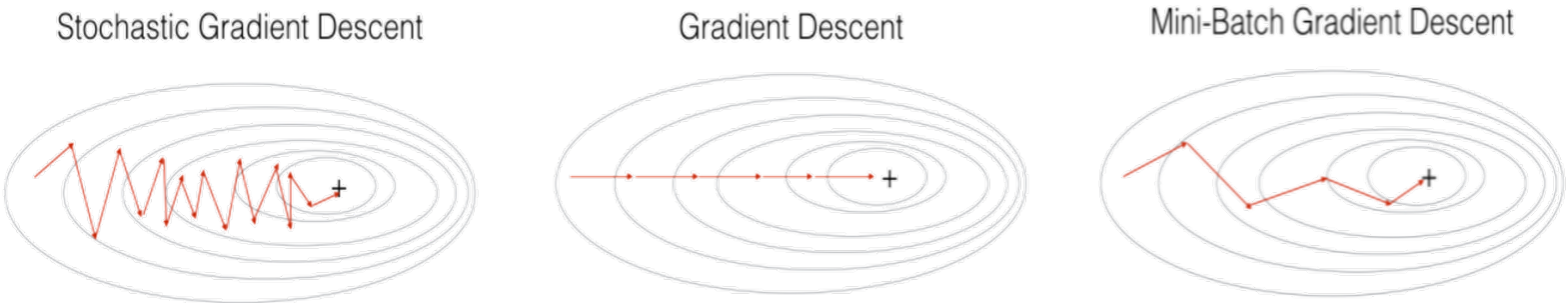
- Algorithm 1 (one-example based BP)
  - 1 **begin initialize** network topology (n_H), $\boldsymbol{w}$, criterion θ, η, $m \leftarrow 0$
  - 2 **do** $m \leftarrow m + 1$
  - 3 $x^m \leftarrow$ randomly chosen pattern
  - 4 $w_{ji} \leftarrow w_{ji} + \eta\delta_j x_i$; $w_{kj} \leftarrow w_{kj} +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

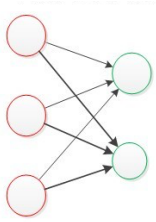
- Algorithm 2 (mini-batch based BP)
  - 1 **begin initialize** network topology (n_H), $\boldsymbol{w}$, criterion θ, η, r $\leftarrow 0$
  - 2 **do** $r \leftarrow r + 1$ (increment epoch)
  - 3 $m \leftarrow 0$; $\Delta w_{ji} \leftarrow 0$; $\Delta w_{kj} \leftarrow 0$;
  - 4 **do** $m \leftarrow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

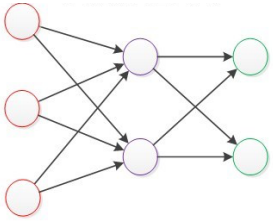
accumulate gradient through n examples.

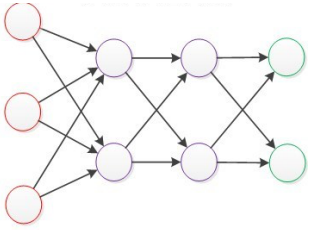
SGD Analysis

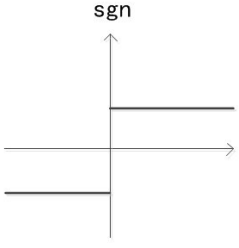
- One-example based SGD
  - Estimation of the gradient is noisy, and the weights may not move precisely down the gradient at each iteration
  - Faster than batch learning, especially when training data has redundancy
  - Noise often results in better solutions
  - The weights fluctuate and it may not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

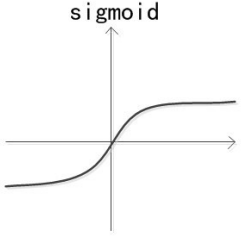
Summarization

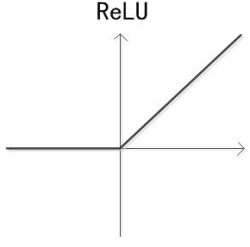


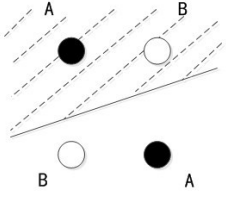


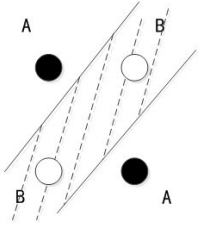


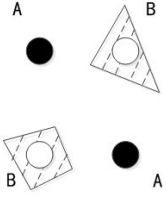


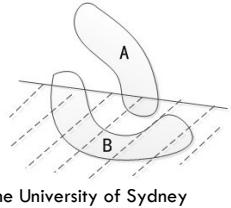


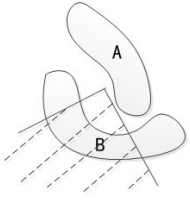


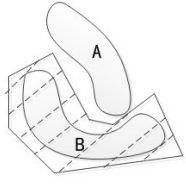












A toy example

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.

before cross-entropy, we need to make sure it's distribution ⇒ pytorch already did it.

Thank you!

1. **Perceptron**

Despite the fantastic name - Deep Learning, the field of neural networks is not new at all. Perceptron can be the foundations for neural networks in 1980s. It was developed to solve a binary classification problem.

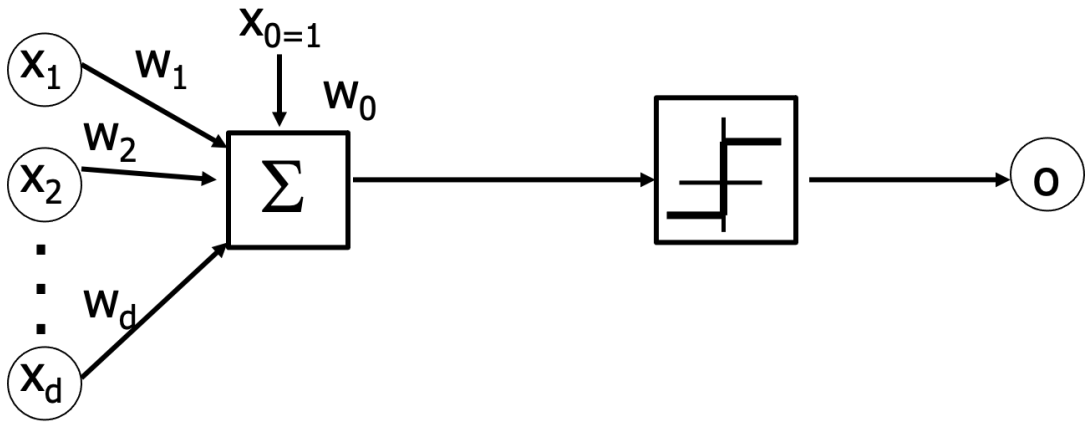


Figure 1. A Perceptron module.

We consider the input signal to be represented by a d-dimensional feature vector $[x_1, x_2, \cdots, x_d]$. In a Perceptron (see Fig. 1), we plan to learn a linear function parameterized by $\mathbf{w}$ to accomplish the classification task,

$$\hat{y} = \sum_{i=1}^{d} w_i x_i + b, \tag{1}$$

where w_i is the weight on the i-th dimension, b is the bias and $\hat{y}$ is the prediction score of the example x. The binary prediction can then be easily with the help of the *sign* function,

$$o = sign(\hat{y}) = \begin{cases} +1, & \hat{y} \geq 0 \\ -1, & \hat{y} < 0 \end{cases} \tag{2}$$

Perceptron simply uses target values $y = 1 + 1$ for the positive class and $y = -1$ for the negative class. According to the above analysis, we find that if an example can be correctly classified by the Perceptron, we have

$$y(\mathbf{w}^T\mathbf{x} + b) > 0, \tag{3}$$

otherwise

$$y(\mathbf{w}^T\mathbf{x} + b) < 0. \tag{4}$$

Therefore, to maximize the prediction accuracy (i.e., to minimize the cost function for Perceptron), the objective function of Perceptron can be written as,

$$\min_{\mathbf{w},b} \quad L(\mathbf{w}, b) = -\sum_{x_i \in \mathcal{M}} y_i(\mathbf{w}^T\mathbf{x}_i + b), \tag{5}$$

where $\mathcal{M}$ stands for the set of mis-classified examples.
Gradient descent can be applied to optimize the Problem (5). By taking the partial derivative, we can calculate the gradient of the objective function,

$$\nabla_{\mathbf{w}} L(\mathbf{w}, b) = -\sum_{x_i \in \mathcal{M}} y_i \mathbf{x}_i, \tag{6}$$

$$\nabla_b L(\mathbf{w}, b) = -\sum_{x_i \in \mathcal{M}} y_i. \tag{7}$$

Gradient descent methods can then be taken to update the parameters $\mathbf{w}$ and b until the convergence.

2. Multilayer Neural Networks

The Perceptron is only capable of separating data points with a linear classifier, and cannot even handle the simple XOr problem. The solution to this problem is to include an additional layer - known as a hidden layer. Then this kind of feed-forward network is a multilayer perceptron, as shown in Fig. 2.

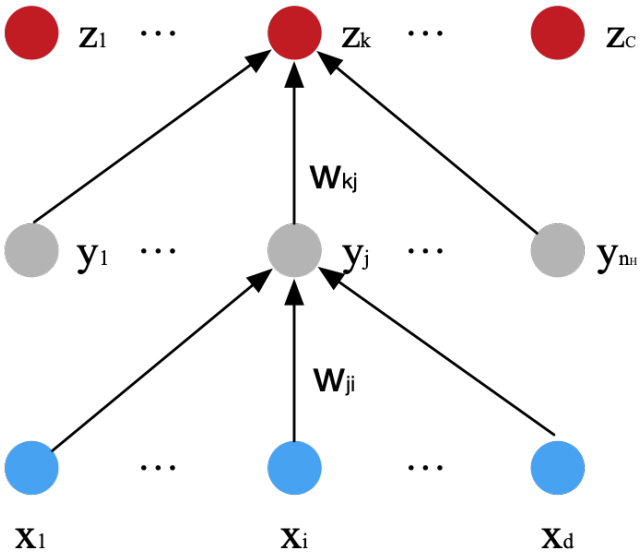


FIGURE 2. A three-layer neural network and the notation used.

Figure 2 shows a simple three-layer neural network, which consists of an input layer, a hidden layer, and an output layer, interconnected by modifiable weights, represented by links between layers. Each hidden unit computes the weighted sum of its inputs to form its scalar net activation, which is denoted simply as *net*. That is, the net activation is the inner product of the inputs with the weights at the hidden unit. Thus, it can be written

$$net_j = \sum_{i=1}^{d} x_i w_{ji} = w_j^T x, \tag{8}$$

where the subscript i indexes units in the input layer; w_{ji} denotes the input-to-hidden layer weights at the hidden unit j. Each hidden unit emits an output that is a nonlinear function of its activation, $f(net)$, that is,

$$y_j = f(net_j). \tag{9}$$

This $f(\cdot)$ is called the activation function or nonlinearity of a unit. Each output unit computes its net activation based on the hidden unit signals as

$$net_k = \sum_{j=0}^{n_H} y_j w_{kj} = w_k^T y, \tag{10}$$

where the subscript k indexes units in the output layer and n_H denotes the number of hidden units. An output unit computes the nonlinear function of its net, emitting

$$z_k = f(net_k) \tag{11}$$

3. **Activation Functions**

Activation functions are functions used in neural networks to decide if a neuron can be fired or not.

3.1. **Sigmoid Function.** The Sigmoid is a non-linear activation function used mostly in feedforward neural networks. It is a bounded differentiable real function, defined for real input values, with positive derivatives everywhere and some degree of smoothness. The Sigmoid function is given by

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{12}$$

However, the Sigmoid activation function suffers major drawbacks:

- Sigmoids saturate and kill gradients. A very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero. Recall that during backpropagation, this (local) gradient will be multiplied to the gradient of this gate's output for the whole objective. Therefore, if the local gradient is very small, it will effectively "kill" the gradient and almost no signal will flow through the neuron to its weights and recursively to its data. Additionally, one must pay extra caution when initializing the weights of sigmoid neurons to prevent saturation. For example, if the initial weights are too large then most neurons would become saturated and the network will barely learn.
- Sigmoid outputs are not zero-centered. This is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centered. This has implications on the dynamics during gradient descent, because if the data coming into a neuron is always positive (e.g. $x > 0$ elementwise in $f = w^T x + b$), then the gradient on the weights w will during backpropagation become either all be positive, or all negative (depending on the gradient of the whole expression f). This could introduce undesirable zig-zagging dynamics in the gradient updates for the weights. However, notice that once these gradients are added up across a batch of data the final update for the weights can have variable signs, somewhat mitigating this issue. Therefore, this is an inconvenience but it has less severe consequences compared to the saturated activation problem above.

3.2. **Hyperbolic Tangent Function (Tanh).** The hyperbolic tangent function known as tanh function, is a smoother zero-centred function whose range lies between -1 to 1, thus the output of the tanh function is given by,

$$f(x) = \frac{e^{-x} - e^{-x}}{e^{-x} + e^{-x}}. \tag{13}$$

The tanh function became the preferred function compared to the sigmoid function in that it gives better training performance for multi-layer neural networks. However, the tanh function could not solve the vanishing gradient problem suffered by the sigmoid functions as well. The main advantage provided by the function is that it produces zero centred output thereby aiding the back-propagation process. The tanh functions have been used mostly in recurrent neural networks for natural language processing.

3.3. **Rectified Linear Unit (ReLU) Function.** The rectified linear unit (ReLU) activation function was proposed by Nair and Hinton 2010, and ever since, has been the most widely used activation function for deep learning applications with state-of-the-art results to date. It offers the better performance and generalization in deep learning compared to the Sigmoid and tanh activation functions. The ReLU activation function performs a threshold operation to each input element where values less than zero are set to zero thus the ReLU is given by

$$f(x) = max(0, x). \tag{14}$$

The main advantage of using the rectified linear units in computation is that, they guarantee faster computation since it does not compute exponentials and divisions, with overall speed of computation enhanced. Another property of the ReLU is that it introduces sparsity in the hidden units as it squishes the values between zero to maximum.
The ReLU has a significant limitation that it is sometimes fragile during training thereby causing some of the gradients to die. This leads to some neurons being dead as well, thereby causing the weight updates not to activate in future data points, thereby hindering learning as dead neurons gives zero activation. To resolve the dead neuron issues, the leaky ReLU was proposed.
The leaky ReLU introduces some small negative slope to the ReLU to sustain and keep the weight updates alive during the entire propagation process. The alpha parameter was introduced as a solution to the ReLUs dead neuron problems such that the gradients will not be zero at any time during training. The LReLU computes the gradient with a very small constant value for the negative gradient α in the range of 0.01 thus the LReLU is computed as

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases} \tag{15}$$

The LReLU has an identical result when compared to the standard ReLU with an exception that it has non-zero gradients over the entire duration thereby suggesting that there no significant result improvement except in sparsity and dispersion when compared to the standard ReLU and tanh function.

4. **Backpropagation**

The backpropagation is one of the simplest and most general methods for supervised training of multilayer neural networks. Networks have two primary modes of operation: feedforward and learning. Feed-forward operation consists of presenting a pattern to the input units and passing the signals through the network in order to yield outputs from the output units. Supervised learning consists of presenting an input pattern and changing the network parameters to bring the actual outputs closer to the desired teaching or target values.
We consider the training error on a pattern to be the sum over output units of the squared difference between the desired output t_k and the actual output z_k:

$$J(w) = \frac{1}{2}\|t - z\|^2 \tag{16}$$

where t and z are the target and the network output vectors and w represents all the weights in the network.
The backpropagation learning rule is based on gradient descent. The weights are initialized with random values, and then they are changed in a direction that will reduce the error:

$$\Delta w = -\eta \frac{\partial J}{\partial w}, \tag{17}$$

where η is the learning rate, and indicates the relative size of the change in weights. Eq. (17) demands that we take a step in weight space that lowers the criterion function. It is clear from Eq. (16) that the criterion function can never be negative; the learning rule guarantees that

learning will stop. This iterative algorithm requires taking a weight vector at the iteration m and updating it as

$$w(m+1) = w(m) + \Delta w. \tag{18}$$

We now turn to the problem of evaluating Eq. (17) for a three-layer net. Consider first the hidden-to-output weights, w_{kj}. Because the error is not explicitly dependent upon w_{kj}, we must use the chain rule for differentiation:

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = -\delta_k \frac{\partial net_k}{\partial w_{kj}}, \tag{19}$$

where the sensitivity of unit k is defined to be

$$\delta_k = -\frac{\partial J}{\partial net_k} \tag{20}$$

and describes how the overall error changes with the unit's net activation and determines the direction of search in weight space for the weights . Assuming that the activation function $f(\cdot)$ is differentiable, we differentiate Eq. (16) and find that for such an output unit, δ_k is simply

$$\delta_k = -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k). \tag{21}$$

Taken together, these results give the weight update or learning rule for the hidden-to-output weights:

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j. \tag{22}$$

The learning rule for the input-to-hidden units is subtle. From Eq. (17), and again using the chain rule, we calculate

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}} \tag{23}$$

The first term can be calculated as

$$\begin{aligned} \frac{\partial J}{\partial y_j} &= \sum_{k=1}^{c} \frac{\partial J}{\partial z_k}\frac{\partial z_k}{\partial y_j} = -\sum_{k=1}^{c} (t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j} \\ &= -\sum_{k=1}^{c} (t_k - z_k) f'(net_k) w_{kj} = -\sum_{k=1}^{c} \delta_k w_{kj}. \end{aligned} \tag{24}$$

For the step above, we had to use the chain rule again. The final sum over output units in Eq. (24) expresses how the hidden unit output y_j, affects the error at each output unit. This will allow us to compute an effective target activation for each hidden unit. We use Eq. (24) to define the sensitivity for a hidden unit as

$$\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj}\delta_k \tag{25}$$

The sensitivity at a hidden unit is simply the sum of the individual sensitivities at the output units weighted by the hidden-to-output weights w_{kj}, all multiplied by $f'(net_j)$. Thus the learning rule for the input-to-hidden weights is

$$\Delta w = \eta x_i \delta_j = \eta f'(net_j) x_i \sum_{k=1}^{c} w_{kj}\delta_k \tag{26}$$

Hence, we conclude the backpropagation algorithm, or more specifically the "backpropagation of errors" algorithm. Backpropagation is just a gradient descent in layered models where application of the chain rule through continuous functions allows the computation of derivatives of the criterion function with respect to all model weights.