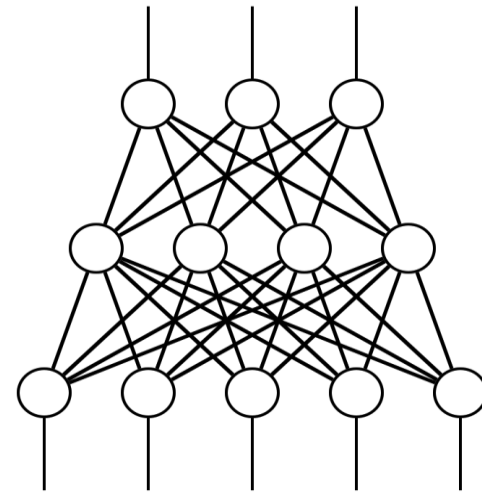


# Multilayer perceptrons – error back-propagation

- Making more powerful networks
  - Using hidden neurons
- Varieties of learning algorithm
- An artificial neuron
- Capabilities of multilayer perceptrons
- Error back-propagation training algorithm
- Comparisons to back-propagation
  - Nearest neighbour
  - Radial basis functions
  - Decision trees

# Making more powerful networks

- Multiple layers of neurons
  - non-linear activation functions.
  - layers of linear units not useful.
- Function should be smooth (although there are minor exceptions)
  - this allows gradient descent.
- In general
  - feed-forward network.
  - no lateral or feedback connections (except if required by a particular application).
- Advantages of hidden neurons
  - extract progressively more complex features from the input.
- Disadvantages of hidden neurons
  - learning is much harder. The learning algorithm must implicitly decide what features should be represented by hidden neurons.



output neurons

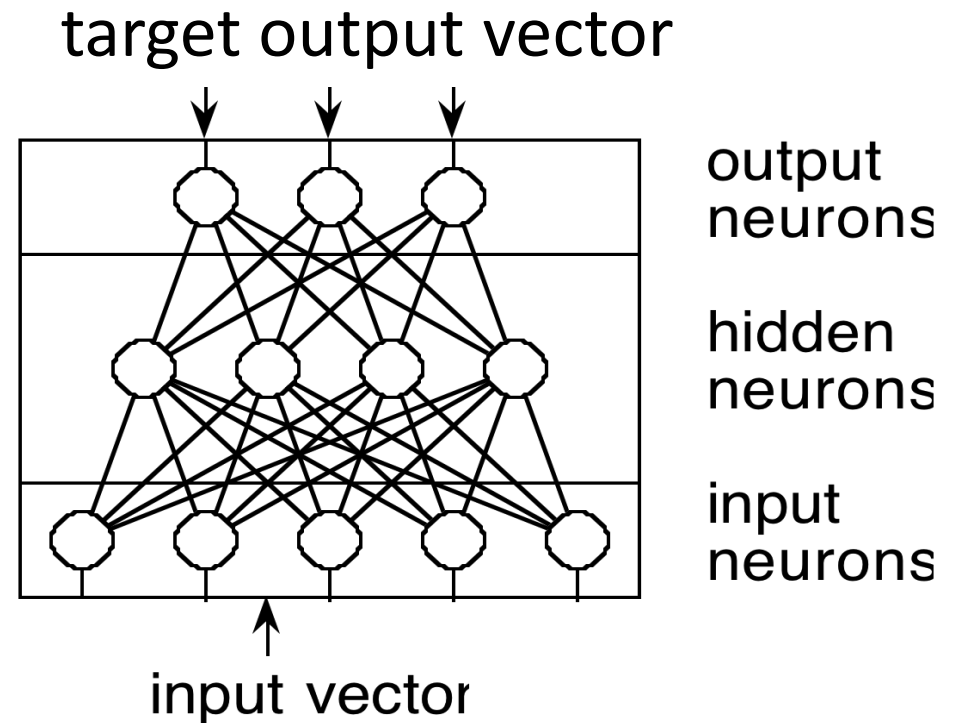
hidden neurons

input neurons

# Varieties of learning algorithm

## Supervised learning

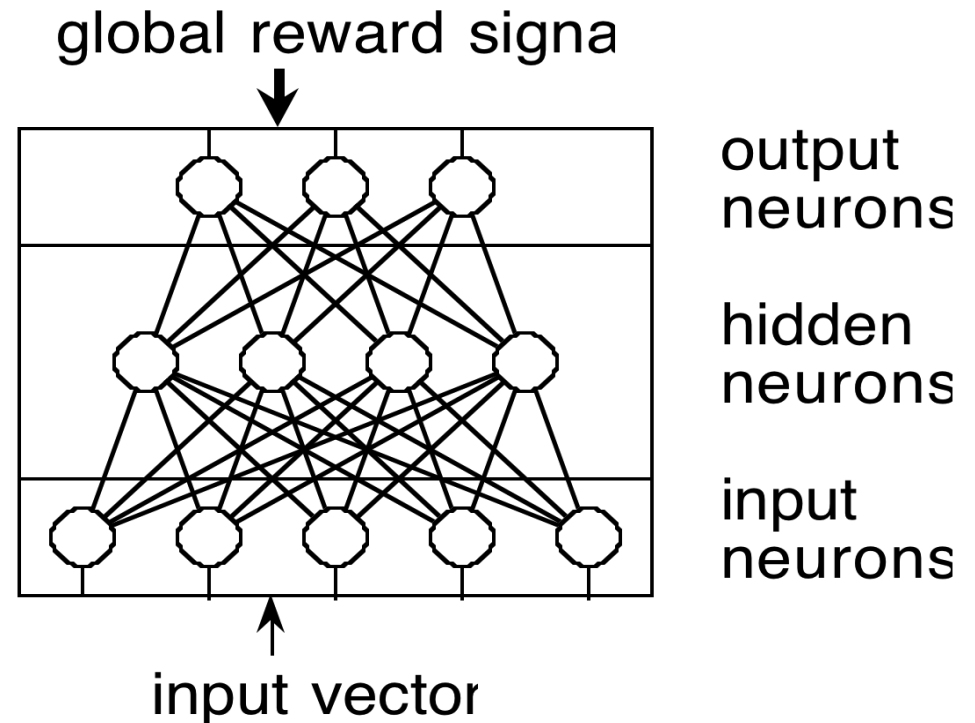
- Requires a 'teacher' who defines the target output vector for each input vector.



# Varieties of learning algorithm 2

## Reinforcement learning

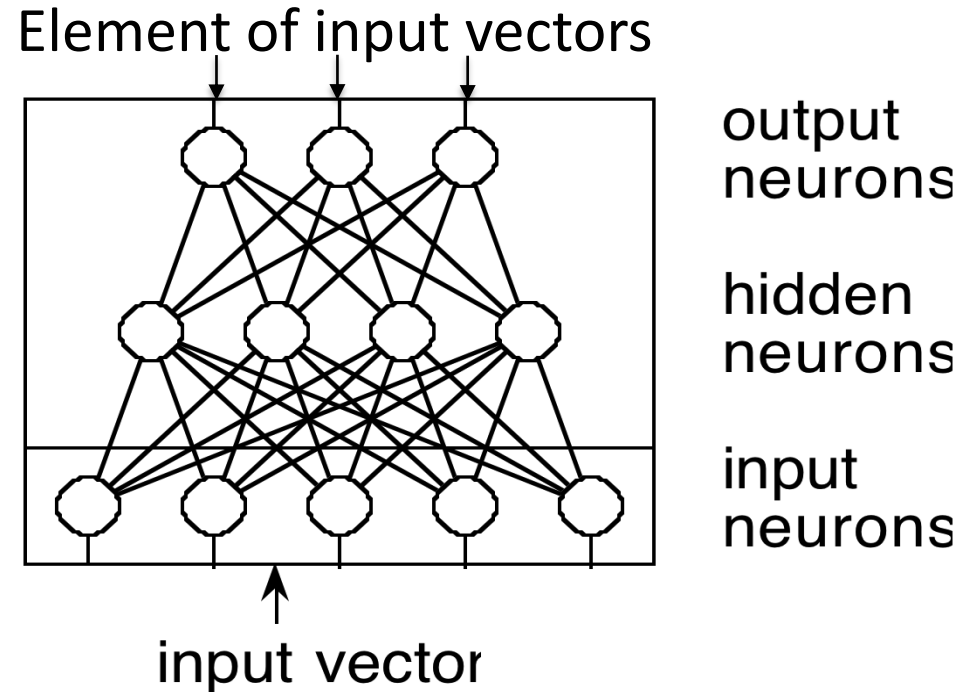
- Optimises sparse rewards
- Is very slow because many input presentations are required so it can assign credit correctly.



# Varieties of learning algorithm 3

## Unsupervised learning

- The processing units model the higher-order statistical structure of the set of input vectors.
- The target output is some element of the set of input vectors, eg the next frame in a video.



# An artificial neuron

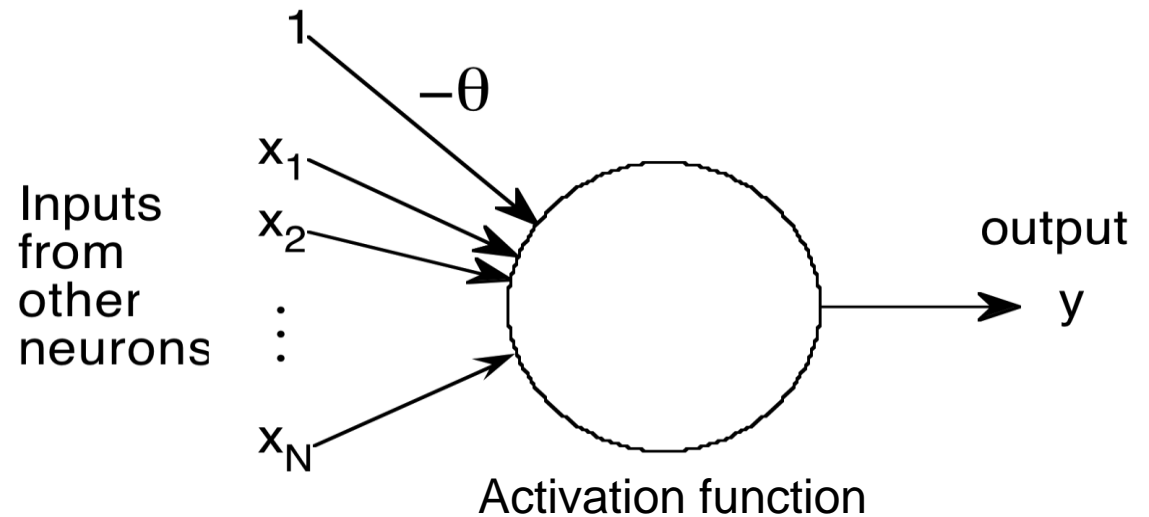
Total input of  
neuron  $j$ :

$$\text{net\_input}_j = \sum_{i=0}^N w_i x_i$$

Output of neuron  $j$ :

$\text{activation\_function}(z)$

where  $z = \text{net\_input}$



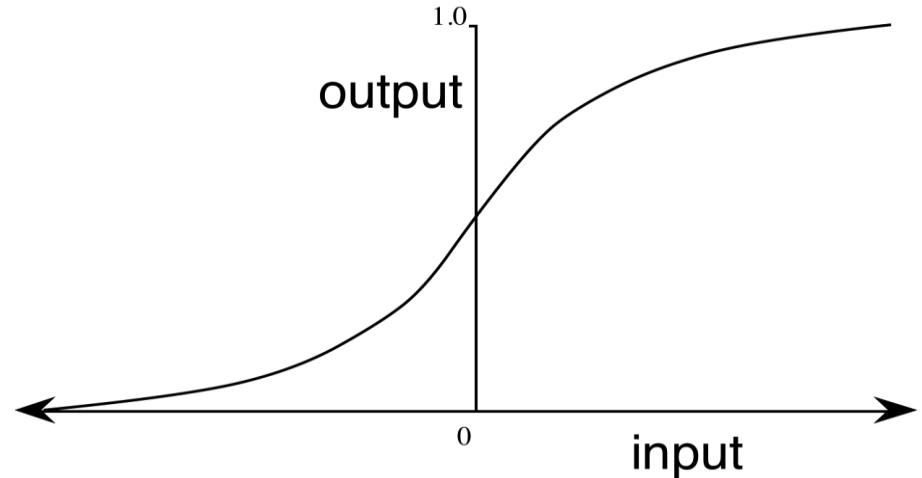
# Sigmoid activation function

One of the simplest activation functions. A smoothed version of the 0 threshold.

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

Historically used in hidden neurons.

Used in output neuron/s to predict two classes 0 or 1 or multiple non-mutually exclusive classes.



# Capabilities of multilayer perceptrons

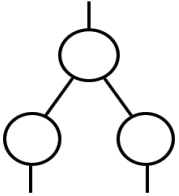
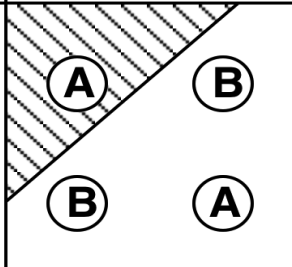
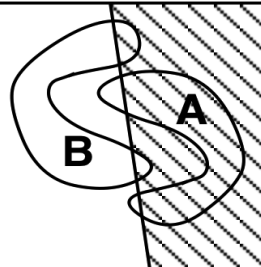
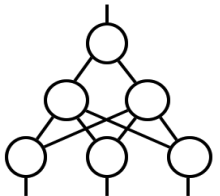
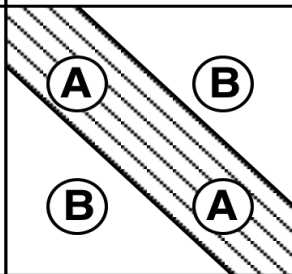
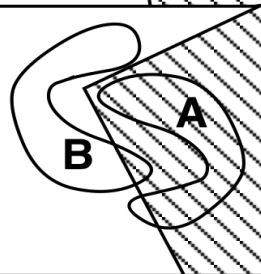
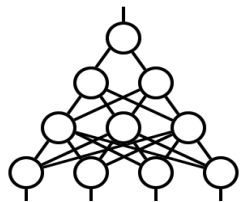
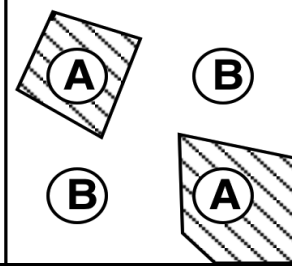
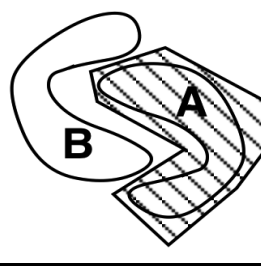
- 1 perceptron
  - decision hyperplane
  - half-plane decision region
- 2 layers of perceptrons
  - 1st layer neurons – half-plane decision regions
  - 2nd layer neurons – **and** decision half-planes to form convex decision region
- 3 layers of perceptrons
  - 1st layer neurons – half-plane decision regions
  - 2nd layer neurons – convex decision regions
  - 3rd layer neurons – **or** convex regions to form concave decision region



# Capabilities of multilayer perceptrons

## 2

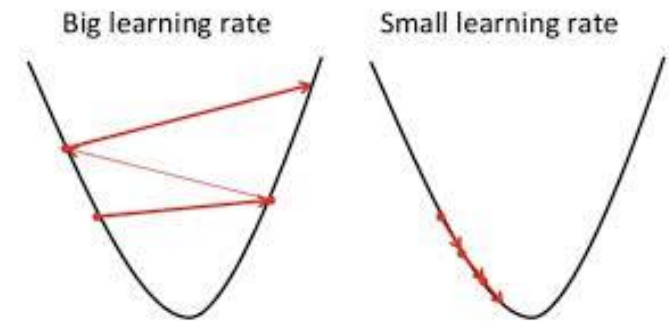
- Shading shows decision regions for class A.
- Note: conventionally input neurons are drawn same as hidden and output neurons. Input neurons are not processing neurons.

Structure	XOR	Meshed	Decision regions
single layer 			half-plane bounded by hyperplane
two layers 			convex open or closed regions
three layers 			convex or concave - arbitrary complexity

# Error back-propagation training alg 1

Why don't we solve for the best weights?

- The analytic solution relies on it being linear and having a squared error measure.
- Iterative methods are usually less efficient but they are much easier to generalise.



## Forward pass

1. Initialise weights (including bias weights) to small random values.
2. Present input  $X$  & target output  $T$

$$\begin{aligned} &(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_M) \\ &(t_1, t_2, \dots, t_M) \end{aligned}$$

# Error back-prop training alg 2

3. Calculate output  $Y$  ( $y_1, y_2, \dots, y_M$ )

1<sup>st</sup> hidden 
$$h_j^I = f(\sum_{i=0}^{N1} w_{ij} x_i), 0 \leq j \leq N1$$

2<sup>nd</sup> hidden 
$$h_k^{II} = f(\sum_{j=0}^{N1} w_{jk}^I h_j^I), 0 \leq k \leq N2$$

Output 
$$y_l = f(\sum_{k=0}^{N2} w_{kl}^{II} h_k^{II}), 0 \leq l \leq M$$

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

# Error back-prop training alg 3

## Chain rule refresher

$$y = f(g(x))$$

$$u = g(x)$$

$$y = f(u)$$

$$\frac{dy}{dx} = \frac{du}{dx} \times \frac{dy}{du}$$

## Derivative of Sigmoid function

$$y = \frac{1}{1+e^{-x}}$$

$$\frac{dy}{dx} = -\frac{1}{(1+e^{-x})^2} (-e^{-x})$$

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

$$= \frac{1}{1+e^{-x}} \left(1 - \frac{1}{1+e^{-x}}\right) = y(1-y)$$

# Error back-prop training alg 3

## Backward pass

4. Calculate the cost e.g. squared error loss:

$$L = \frac{1}{2} \sum_{l=0}^M (t_l - y_l)^2$$

and it's gradient w.r.t. the output  $y_l$

$$\begin{aligned} \frac{dL}{dy_l} &= -(t_l - y_l) \\ &= (y_l - t_l) \end{aligned}$$

5. Back propagate the gradients using the chain rule of differentiation starting with the output layer and working backwards towards the input layer.

output neuron:  $\frac{dy_l}{dz_l} = y_l(1 - y_l)$  derivative of sigmoid

$$\frac{dL}{dz_l} = y_l(1 - y_l) (y_l - t_l) \quad \text{chain rule}$$

# Error back-prop training alg 4

## Backward pass (continued)

hidden neuron:  $\frac{dL}{dh_k^{II}} = \sum_l \frac{dL}{dz_l} w_{kl}$  chain rule

$$\frac{dL}{dz_k^{II}} = h_k^{II} (1 - h_k^{II}) \sum_l \frac{dL}{dz_l} w_{kl}$$

chain rule and derivative of sigmoid

output weights:  $\frac{dL}{dw_{kl}} = \frac{dL}{dz_l} h_k^{II}$  chain rule

hidden weights:  $\frac{dL}{dw_{jk}} = \frac{dL}{dz_k^I} h_j^I$  chain rule

# Error back-prop training alg 4

## Weights update

5. Update weights

$$w_{kl}(t + 1) = w_{kl}(t) + \eta \left( -\frac{dL}{dw_{kl}} \right)$$

$$w_{jk}(t + 1) = w_{jk}(t) + \eta \left( -\frac{dL}{dw_{jk}} \right)$$

$$0 < \eta \leq 1$$

6. Repeat etc.

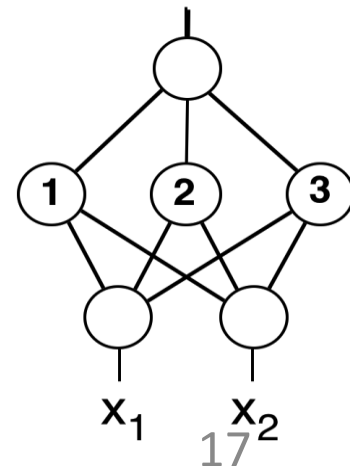
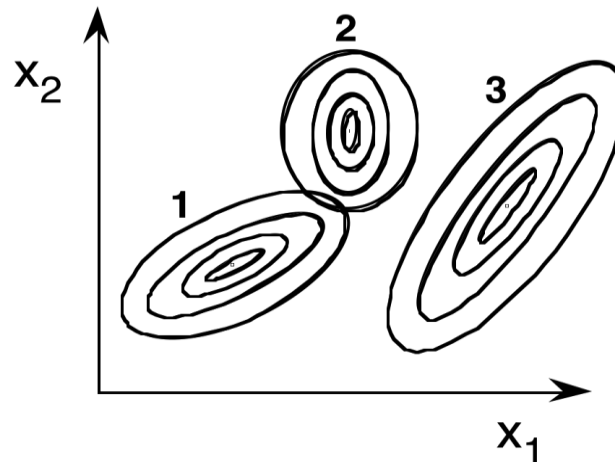
# Nearest neighbour vs back-prop

- Nearest neighbour: exemplar set of numeric vectors and classification, new vector is compared to all exemplars and closest is chosen.
- Few data points – use nearest neighbour. Bp may require lots of training data for good generalisation.
- Noisy data – use back-propagation as it is less sensitive to noise.
- Training time – none for nearest neighbour.
- Classification speed – back-propagation faster, and more efficient.
- Accuracy – both nearest neighbour and back-propagation can be made arbitrarily accurate if enough data is available.



# More alternatives to back-prop

- k-nearest neighbour: find the  $k$  closest exemplars and use majority voting to classify – less sensitive to noise, but more expensive to compute.
- Decision trees (CART, ID3, C4.5) – faster than back-propagation, generalisation often not as good.
- Radial basis functions: this approach is based on sums of gaussian functions. The gaussians are centred on selected training points (or clusters of points). Training time is faster than back-prop, but more neurons may be required, so RBFs are less efficient in practice.



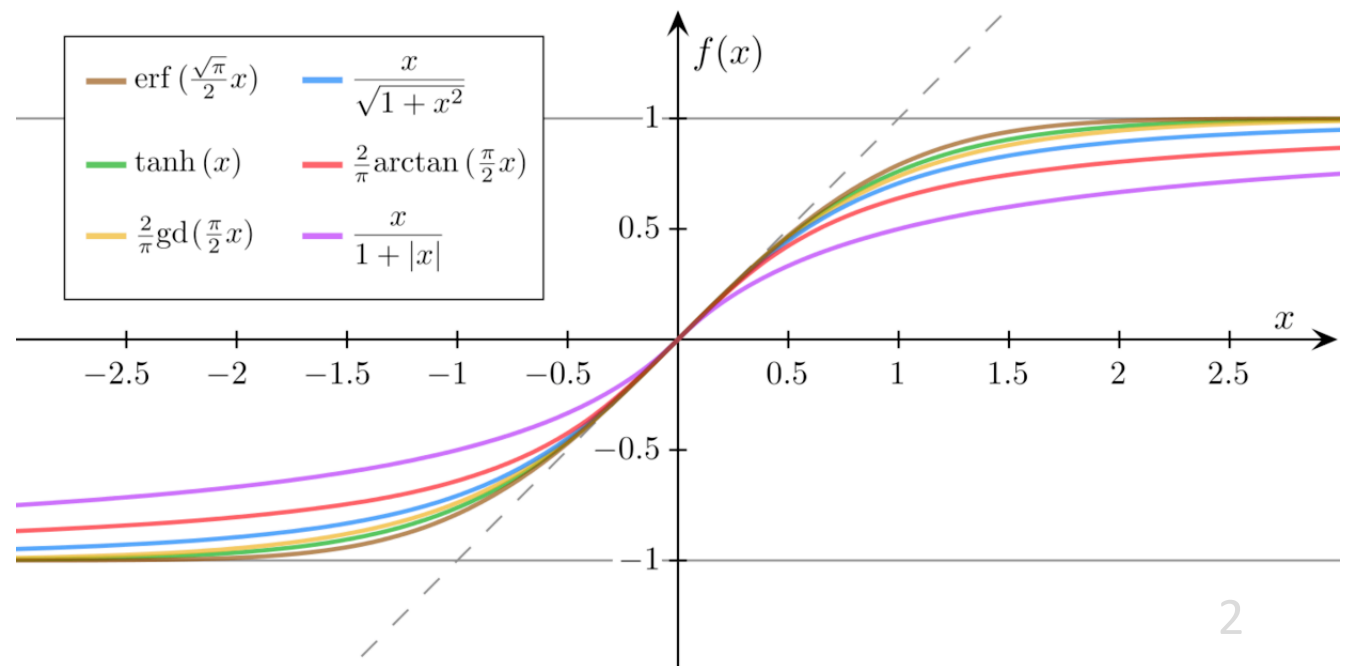
# *ReLU and Softmax Activation Functions*

By Yi Jang <https://github.com/Kulbear/deep-learning-nano-foundation>

- Refresher: The Sigmoid Function
- Rectified Linear Units
- Softmax

# Refresher: The Sigmoid Function

- The sigmoid function has been widely used in machine learning intro materials, especially for the logistic regression and some basic neural network implementations. However, you may need to know that the sigmoid function is not your only choice for the activation function and it does have drawbacks.



- Let's see what a sigmoid function could benefit us.

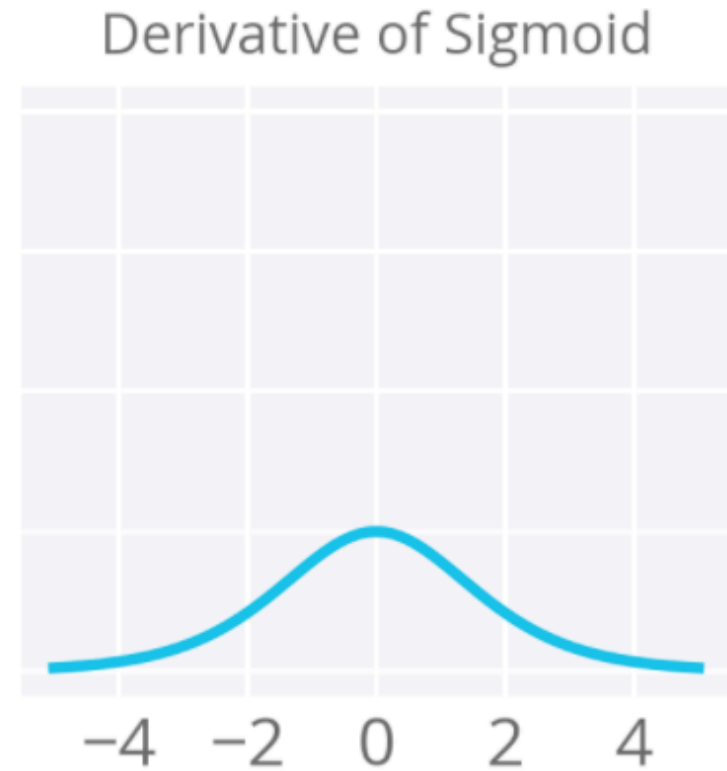
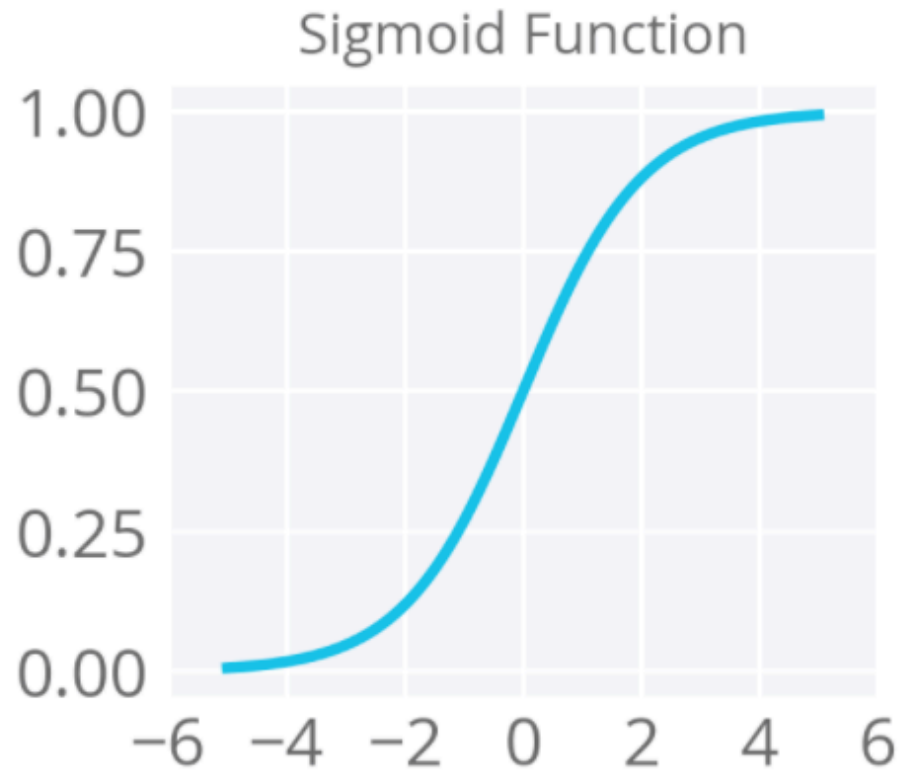
$$f(x) = \text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

- By taking this formula, we can get the derivative of the sigmoid function, note that for shortening the formula, here  $f(x)$  is the sigmoid function.

$$f'(x) = f(x)(1 - f(x))$$

- Despite that, such a result from the derivative is easy to calculate and save times for building models, the sigmoid function actually forces your model "losing" knowledge from the data. Why?

- ... "losing" knowledge from the data. Why? Think about the possible maximum value of the derivative of a sigmoid function.



- For the backpropagation process in a neural network, it means that your errors will be squeezed by (at least) a quarter at each layer. Therefore, deeper your network is, more knowledge from the data will be "lost". Some "big" errors we get from the output layer might not be able to affect the synapses weight of a neuron in a relatively shallow layer much ("shallow" means it's close to the input layer).
- Due to this, sigmoids have fallen out of favor as activations on hidden units.
- Other sigmoid functions [arctangent](#), [hyperbolic tangent](#), [Gudermannian function](#), and [error function](#), [generalised logistic function](#) and [algebraic functions](#)

# Rectified Linear Units

- Instead of sigmoids, most recent deep learning networks use rectified linear units (ReLUs) for the hidden layers. A rectified linear unit has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to the input. ReLUs' machinery is more like a real neuron in your body.  $f(x) = \max(x, 0)$
- [This 'like a real neuron' is not a correct statement in the original!]

- ReLU activations are the simplest non-linear activation function you can use, obviously. When you get the input is positive, the derivative is just 1, so there isn't the squeezing effect you meet on backpropagated errors from the sigmoid function. [Research has shown](#) that ReLUs result in much faster training for large networks. Most frameworks like TensorFlow and TFLearn make it simple to use ReLUs on the the hidden layers, so you won't need to implement them yourself.
- However, such a simple solution is not always a perfect solution. From [Andrej Karpathy's CS231n course](#):
- Unfortunately, ReLU units can be fragile during training and can "die".



# Softmax

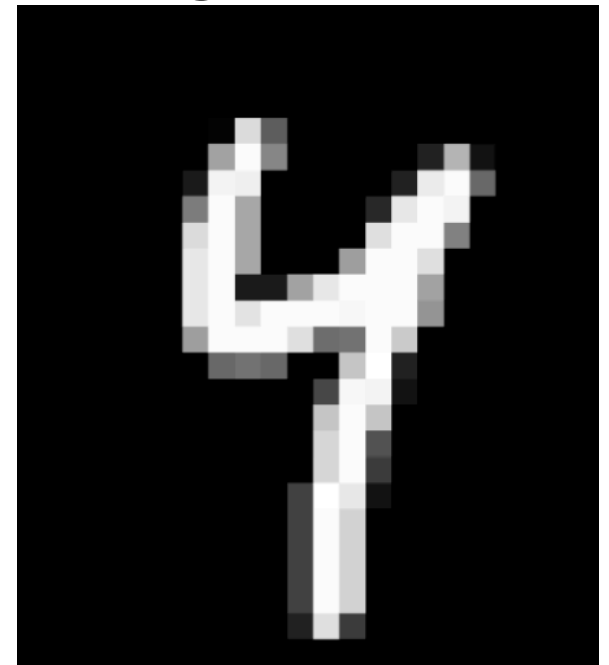
- The sigmoid function can be applied easily, the ReLUs will not vanish the effect during your training process. However, when you want to deal with classification problems, they cannot help much. Simply speaking, the sigmoid function can only handle two classes, which is not what we expect.



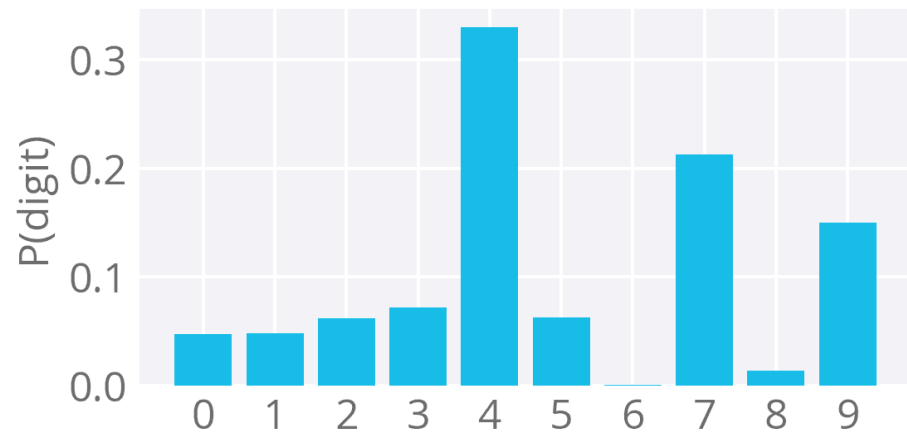
- The output of the softmax function is equivalent to a categorical probability distribution, it tells you the probability that any of the classes are true.
- Mathematically the softmax function is shown below, where  $z$  is a vector of the inputs to the output layer (if you have 10 output units, then there are 10 elements in  $z$ ). And again,  $j$  indexes the output units, so  $j = 1, 2, \dots, K$ .

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$$

- “The material below is taken from a Udacity course.”
- To understand this better, think about training a network to recognize and classify handwritten digits from images. The network would have ten output units, one for each digit 0 to 9. Then if you fed it an image of a number 4 (see below), the output unit corresponding to the digit 4 would be activated.
- Building a network like this requires 10 output units, one for each digit. Each training image is labeled with the true digit and the goal of the network is to predict the correct label.



- So, if the input is an image of the digit 4, the output unit corresponding to 4 would be activated, and so on for the rest of the units.
- For the example image above, the output of the softmax function might look like:
- The image looks the most like the digit 4, so you get a lot of probability there. However, this digit also looks somewhat like a 7 and a little bit like a 9 without the loop completed. So, you get the most probability that it's a 4, but also some probability that it's a 7 or a 9.
- The softmax can be used for any number of classes.



- So, if the input is an image of the digit 4, the output unit corresponding to 4 would be activated, and so on for the rest of the units.
- For the example image above, the output of the softmax function might look like:
- The image looks the most like the digit 4, so you get a lot of probability there. However, this digit also looks somewhat like a 7 and a little bit like a 9 without the loop completed. So, you get the most probability that it's a 4, but also some probability that it's a 7 or a 9.
- The softmax can be used for any number of classes. It's also used for hundreds and thousands of classes, for example in object recognition problems where there are hundreds of different possible objects.