

# Dense and Shallow Neural Networks

<https://shala2020.github.io/>

# Building a Neural Network

- The basic building block of neural networks is the perceptron!
- Perceptrons work well with linear decision boundaries

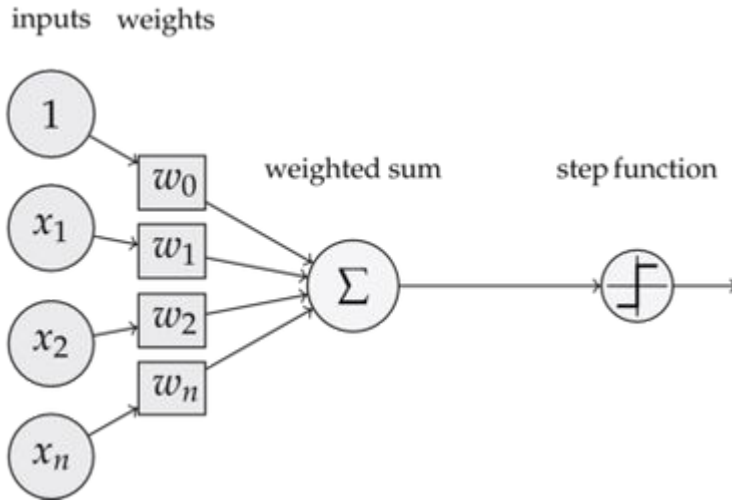


Image source: [blog.knoldus.com](http://blog.knoldus.com)

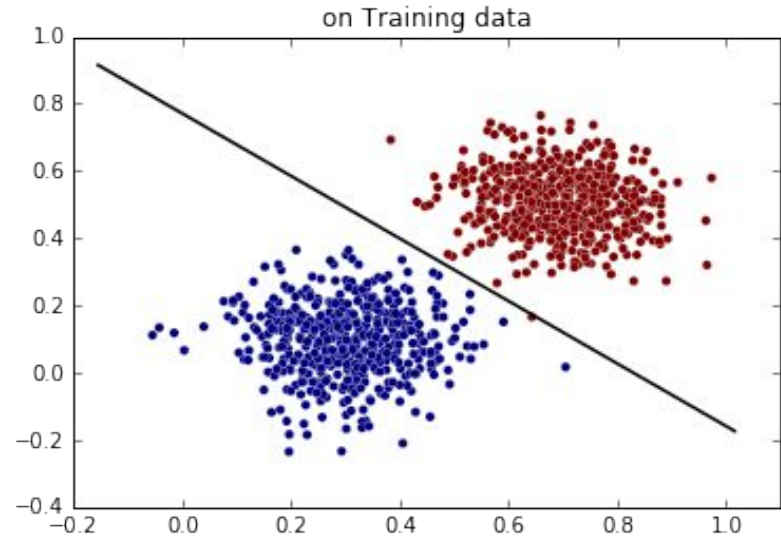
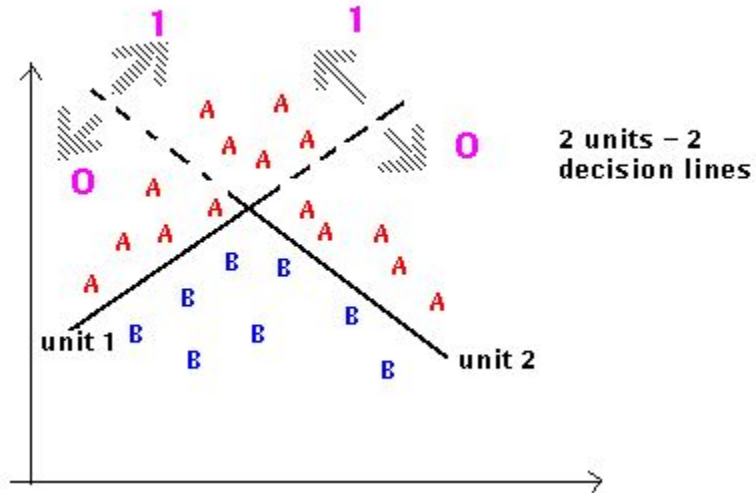


Image source: [nasirml.wordpress.com](http://nasirml.wordpress.com)

# Non- linear decision boundaries

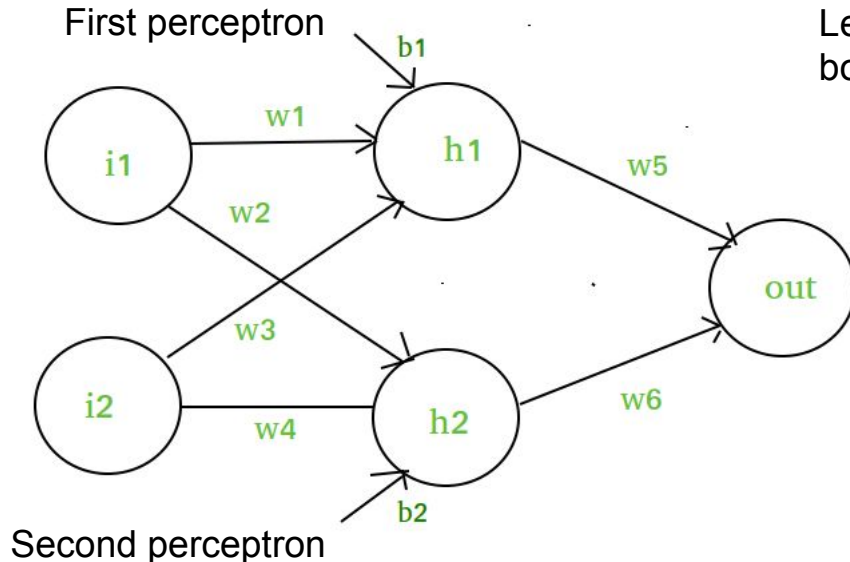
A perceptron can not learn non linear decision boundaries



But a combination of 2 perceptrons can!

# Perceptrons working together

For the previous example, let's combine the 2 perceptrons together. Each perceptron will tell us which side the input lies on, and we will use these 2 values to classify the input to classify the input

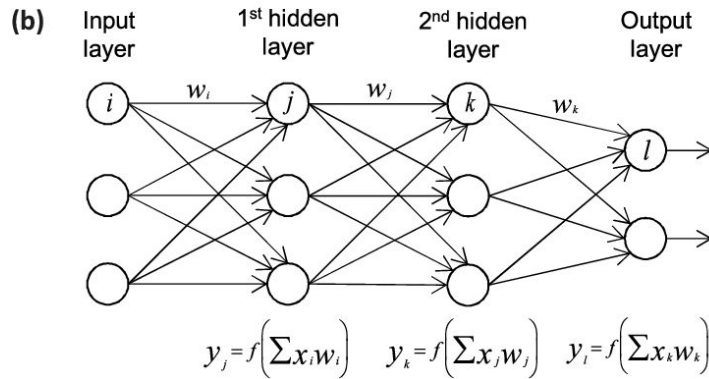
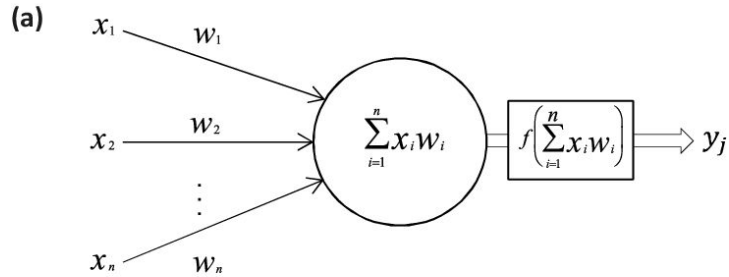


Let  $h=1$  signify the input is 'above' the perceptron boundary as marked in the previous slide

For  $w_5=w_6=1$  and a step activation function with threshold 0.5, output will be 1 for all points of class A and 0 for all points of class B

Thus using a layer of two perceptrons we were able to achieve a non linear decision boundary !

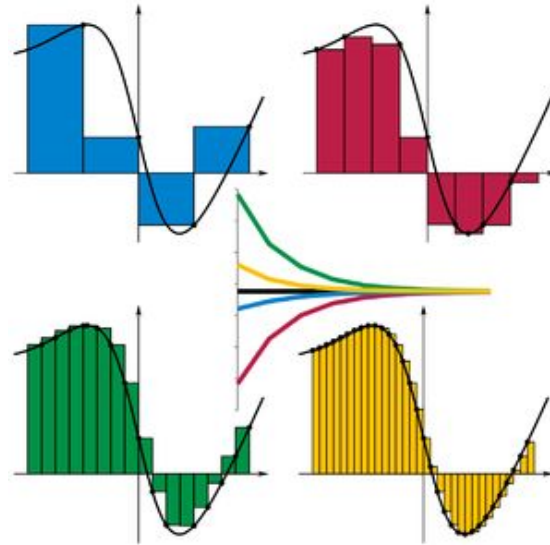
# Neural Networks



At every node, we take a linear combination of inputs and pass it through a non-linear activation function

# Approximating a real valued function

Similar to how we combined 2 perceptrons, we can divide the real line into a large number of intervals using perceptrons and approximate the function in each interval. As the number of hidden units increase (blue to yellow), the approximation gets better



# Universal Approximation Theorem

- A neural network with a single hidden layer can approximate a function to any degree given that the single hidden layer can have arbitrary number of units

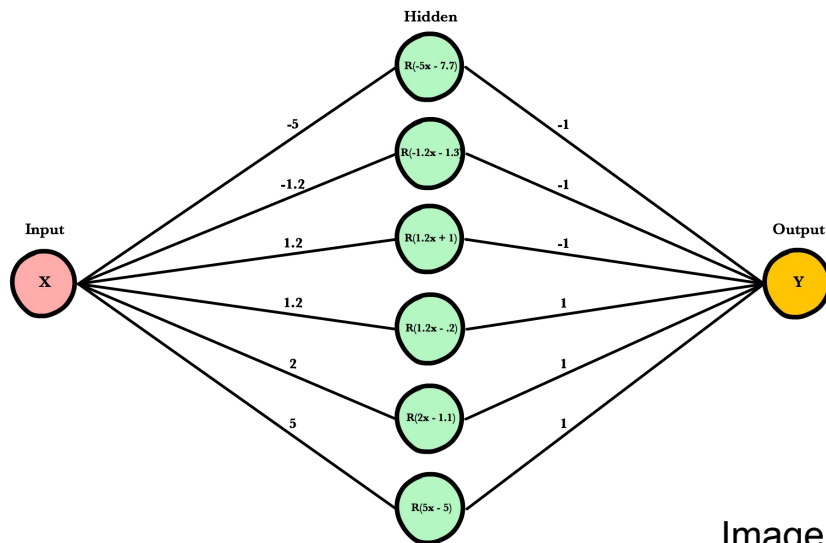
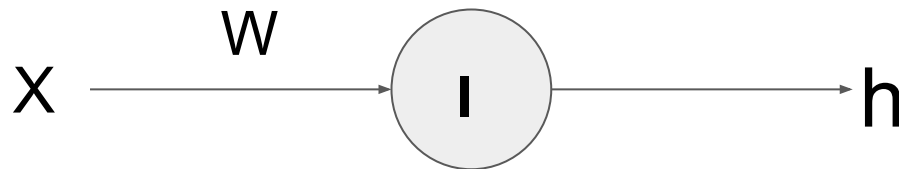


Image source: [towardsdatascience.com](https://towardsdatascience.com)

# Linear Regression as a NN

- Identity Function as an activation



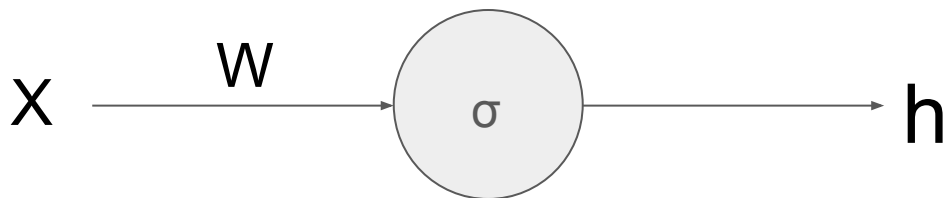
$$h = I(W^T X) = W^T X$$

- Loss Function is MSE



# Logistic Regression as a Sigmoid

- Sigmoid Function as activation
- Loss Function : Cross Entropy (binary cross entropy)




$$h = \sigma(W^T X) = \frac{1}{1 + \exp(-W^T X)}$$

# Multi - Class Classification

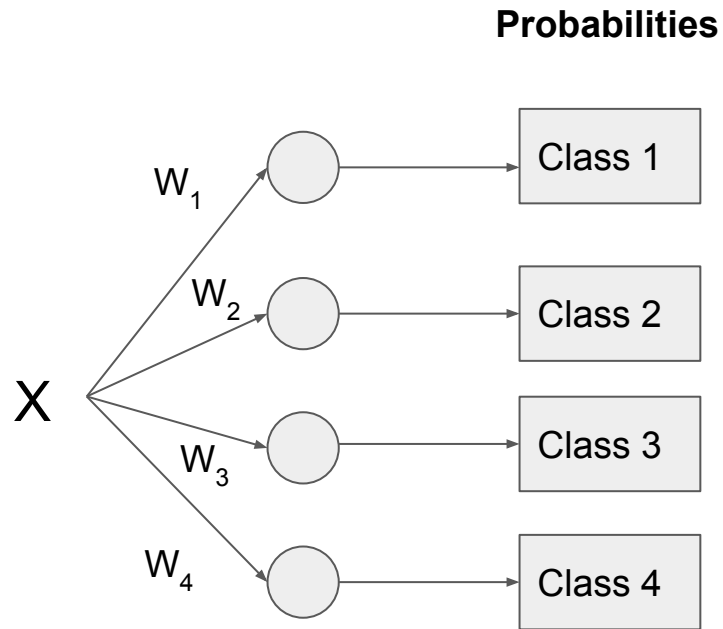
$$P(y=j \mid \theta^{(i)}) = \frac{e^{\theta^{(i)}}}{\sum_{j=0}^k e^{\theta_k^{(i)}}}$$

where  $\theta = w_0x_0 + w_1x_1 + \dots + w_kx_k = \sum_{i=0}^k w_ix_i = w^T x$

Softmax function



$$L_i = \sum_{j=0}^k y_{i,j} * \log h_{i,j}$$



# Why do we need activation functions ?

- The advantage of neural networks lies in the ability to learn complex non-linear relationships
- If there were no activation functions, each layer would be a composition of linear transforms, which is again a linear transform! So the network would be equivalent to a linear model

# Various activation Functions

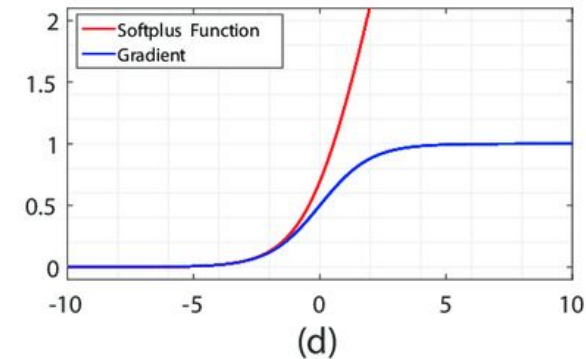
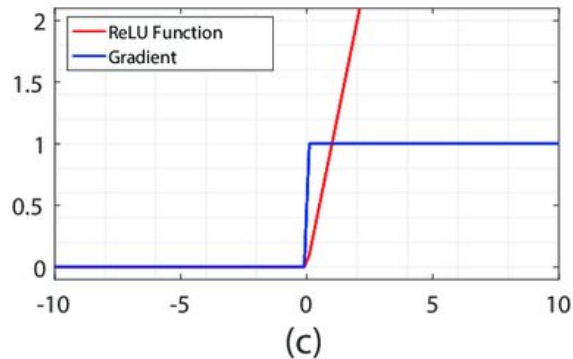
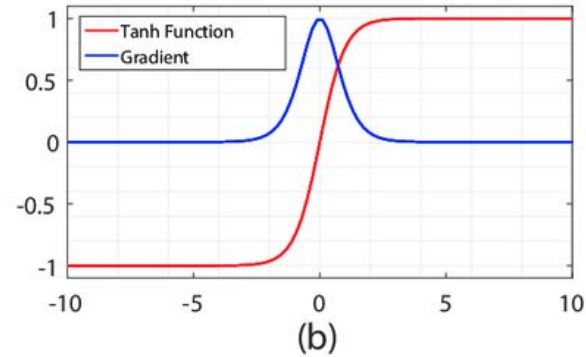
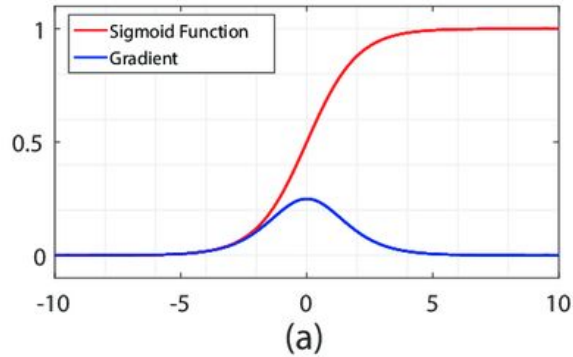
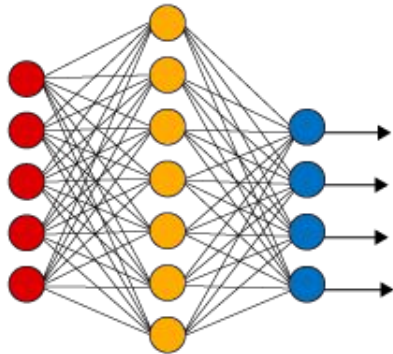


Image source: Ranking to Learn and Learning to Rank: On the Role of Ranking in Pattern Recognition Applications - Scientific Figure on ResearchGate.

# Advantages of multiple hidden layers

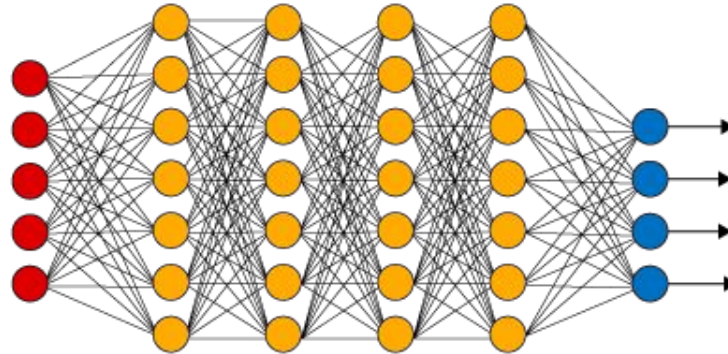
- The output of each layer is a function of the inputs and as the network has more and more hidden layers, their outputs become a composition of functions on the input. This helps deeper layers capture more and more complex features

**Simple Neural Network**



● Input Layer

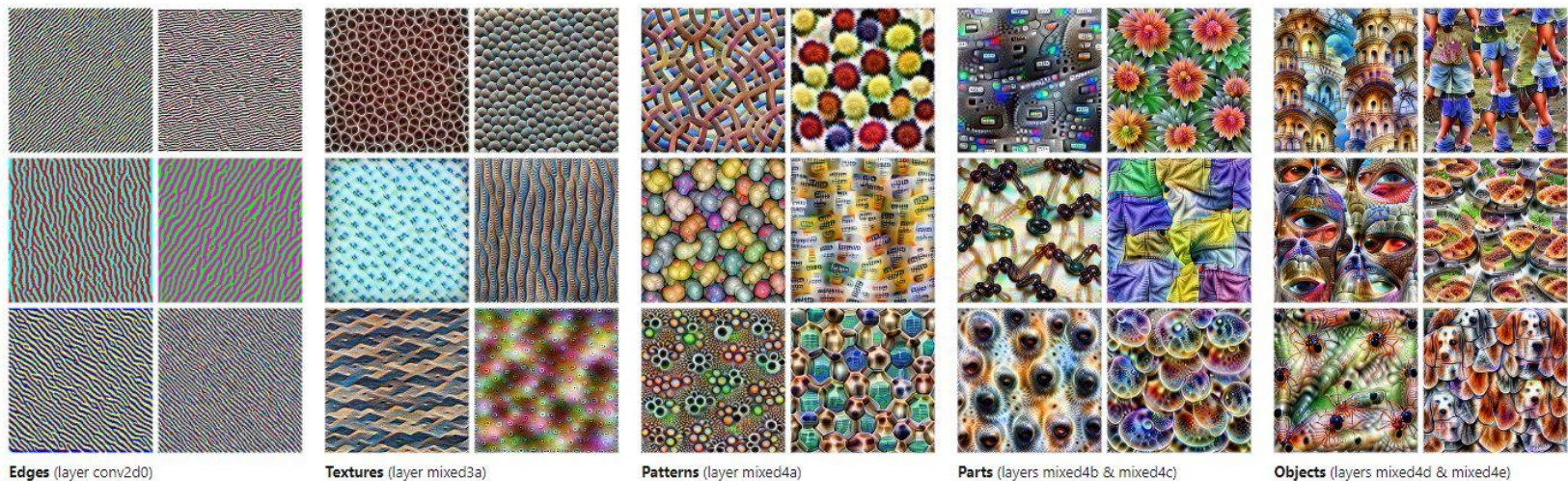
**Deep Learning Neural Network**



● Hidden Layer

● Output Layer

# Multiple hidden layers : a sneak peek

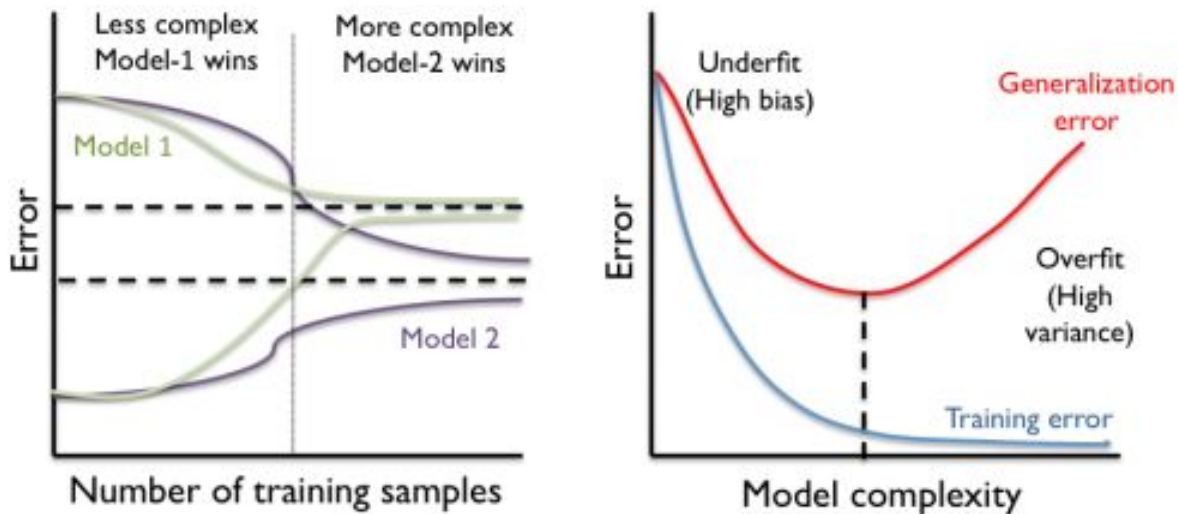


Activation of filters of GoogleNet ; filters of the deeper layers are on the right which are capturing much more complex features

Image source : [distill.pub](http://distill.pub)

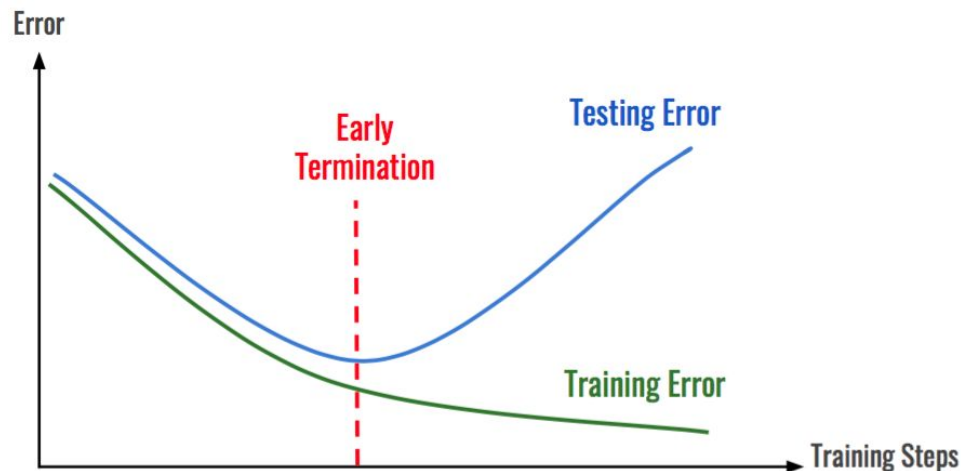
# Overfitting

- Neural networks are arbitrary function approximators and are prone to overfitting - they might easily reach very low train error with high validation error. Complex models with less training data are likely to overfit and limiting the number of neurons helps tackle this



# Overfitting

- Another method to prevent overfitting is early stopping. As the model is trained for more and more epochs, it tends to reduce the training error at the cost of generalisation.





# Overfitting

- weight decay or L2 regularisation : If the weight of some neurons is too large, the model tends to be strongly influenced by the features captured by those neurons leading to bad generalisation. Weight decay is applied by adding an extra term to the loss function

## Regularization: Weight decay

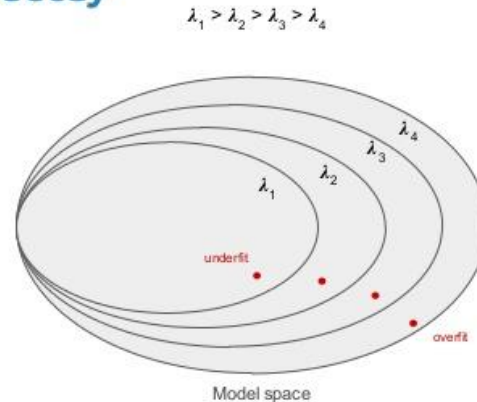
Add a penalty to the loss function for large weights

L2 regularization on weights

$$L = L_{\text{data}} + \frac{\lambda}{2} \|W\|_2^2$$

Differentiating, this translates to decaying the weights with each gradient descent step

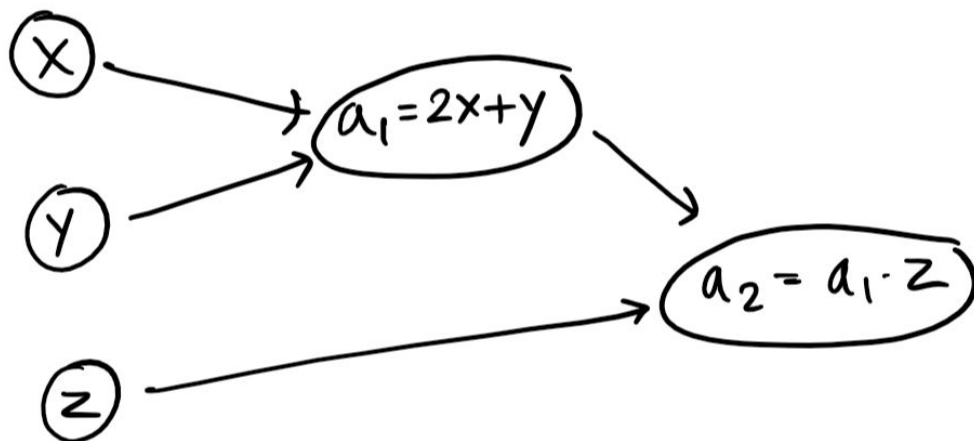
$$w_{t+1} = w_t - \alpha \Delta_w L_{\text{data}} - \lambda w$$



# Computational Graphs

- Computational graphs are used to represent an arbitrary function. It shows us how the inputs combine to get the function output. Consider the function :

$$f(x, y, z) = (2x + y)z$$

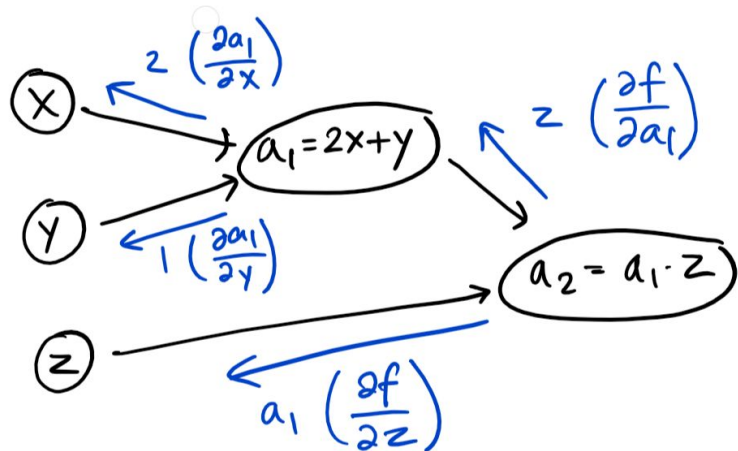


# Backpropagation

- Backpropagation involves the frequent use of chain rule to find gradients.
- The central idea behind backprop and computational graphs is to break down a nasty composition of calculations into small steps and calculating the gradient at each step. The chain rule then helps us merge the gradients of these steps to get the answer

# Backpropagation

$$f(x, y, z) = (2x + y)z$$



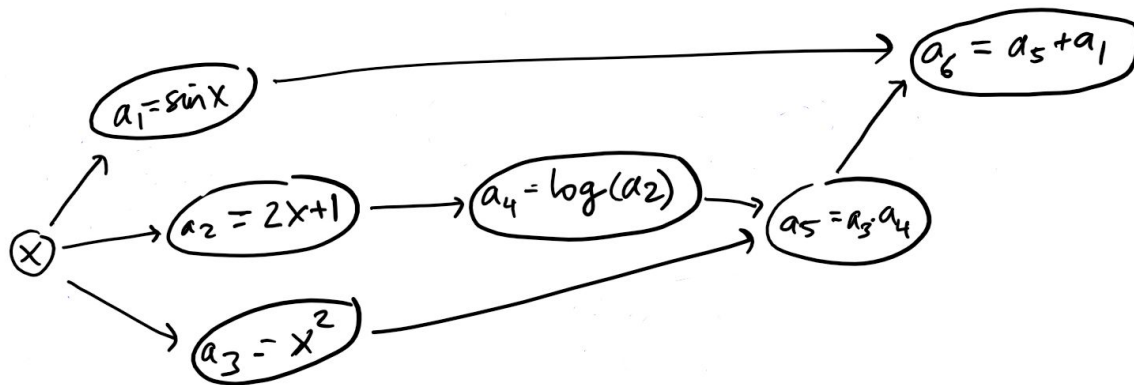
$$\frac{\partial f}{\partial x} = 2 \cdot z \quad ; \quad \frac{\partial f}{\partial y} = 1 \cdot z$$

$$\frac{\partial f}{\partial z} = a_1 = (2x + y)$$

# Computational Graphs

- Consider the function  $f(x) = x^2 \log(2x+1) + \sin(x)$ .

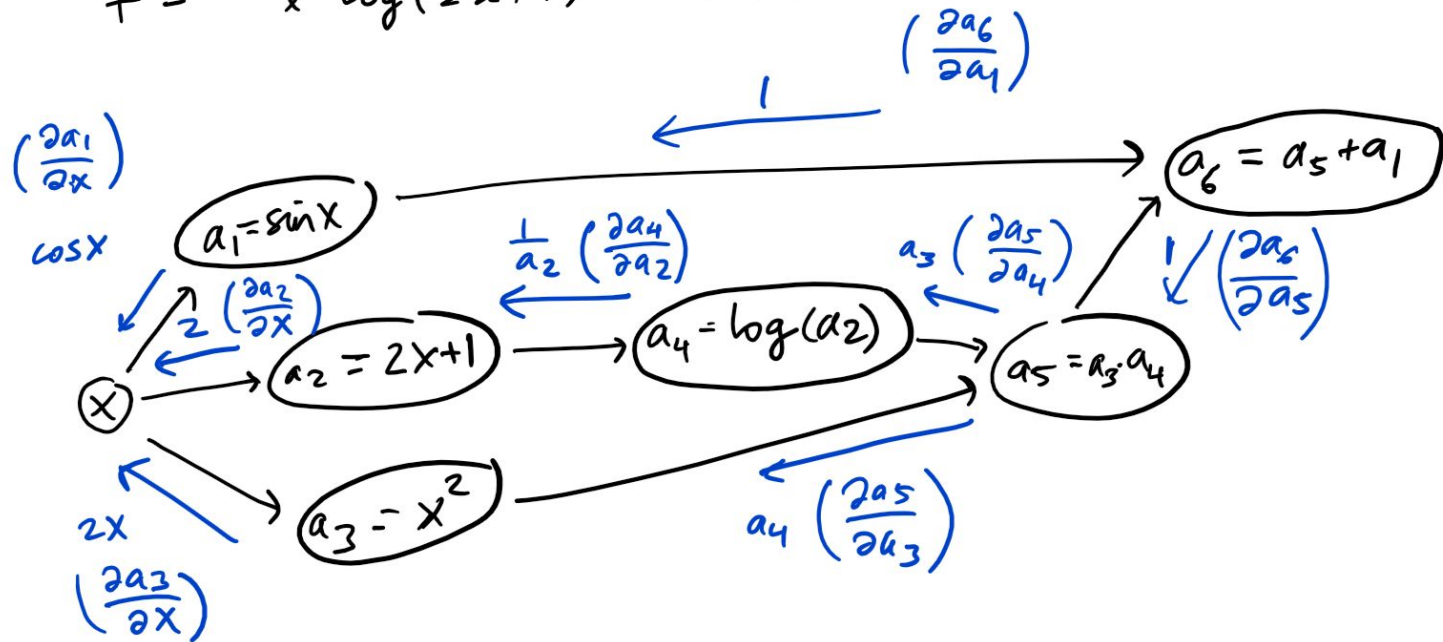
$$f = x^2 \log(2x+1) + \sin(x)$$



$$\frac{\partial f}{\partial x} = 1 \cdot \cos x + 1 \cdot x^2 \cdot \frac{1}{2x+1} \cdot 2 + 1 \cdot \log(2x+1) \cdot 2x$$

# Backpropagating

$$f = x^2 \log(2x+1) + \sin(x)$$



$$\frac{\partial f}{\partial x} = 1 \cdot \cos x + 1 \cdot x^2 \cdot \frac{1}{2x+1} \cdot 2 + 1 \cdot \log(2x+1) \cdot 2x$$

# Backprop in Neural Networks

- This idea of backpropagating on a computational graph is very powerful when dealing with complex functions
- A neural network is also a complex function and we want to find the derivatives of the weights to apply gradient descent

## The Jacobian chain rule

Let the Jacobian matrix  $\nabla_X Y$  for multidimensional variables  $X = (x_1, \dots, x_m)$  and  $Y = (y_1, \dots, y_n)$  be given as :

$$\nabla_X Y = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_m} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_2}{\partial x_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial x_1} & \frac{\partial y_n}{\partial x_2} & \dots & \frac{\partial y_n}{\partial x_m} \end{bmatrix}$$

1. If  $L = F(Y)$  and  $Y = G(X)$ , then the gradient of  $L$  wrt  $X$  is given by :

$$\nabla_X L = \nabla_Y L \cdot \nabla_X Y$$

For a linear transform  $Y = W \cdot X$ ,

$$\begin{bmatrix} y_1 \\ \vdots \\ \boxed{y_i} \\ \vdots \end{bmatrix} = \begin{bmatrix} W_{11} & \dots & W_{1j} & \dots \\ \vdots & & & \\ W_{i1} & \dots & \boxed{W_{ij}} & \dots \\ \vdots & & & \vdots \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ \boxed{x_j} \\ \vdots \end{bmatrix}$$

$$\nabla_X Y_{ij} = \frac{\partial y_i}{\partial x_j} = W_{ij}$$

$$\nabla_X Y = W$$

---



2. If  $L$  is a scalar,  $X$ ,  $Y$  are vectors and  $W$  is a matrix such that  $L = F(Y)$  and  $Y = W \cdot X$ , then the derivative matrix of  $W$ ,

$$dW_{ij} := \frac{\partial L}{\partial W_{ij}}$$

is given as :

$$dW = \nabla_Y L^T \cdot X^T$$

Think of it from the computational graph perspective :  $W_{ij}$  only affects  $L$  through  $y_i$  since :

$$\begin{bmatrix} y_1 \\ \vdots \\ \boxed{y_i} \\ \vdots \end{bmatrix} = \begin{bmatrix} W_{11} & \cdots & W_{1j} & \cdots \\ \vdots & & & \\ W_{i1} & \cdots & \boxed{W_{ij}} & \cdots \\ \vdots & & & \vdots \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ \vdots \\ \boxed{x_j} \\ \vdots \end{bmatrix}$$

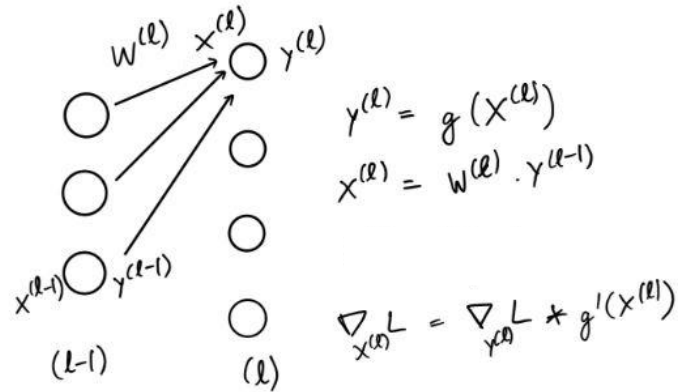
so  $dW_{ij}$  should be given by

$$\frac{\partial L}{\partial W_{ij}} = \frac{\partial L}{\partial y_i} \times x_j$$

And  $\nabla_Y L^T \cdot X^T$  gives

$$\begin{bmatrix} \frac{\partial L}{\partial y_1} \\ \vdots \\ \boxed{\frac{\partial L}{\partial y_i}} \\ \vdots \end{bmatrix} \cdot \begin{bmatrix} x_1 & \cdots & \boxed{x_j} & \cdots \end{bmatrix} = \begin{bmatrix} \frac{\partial L}{\partial W_{11}} & \cdots & \frac{\partial L}{\partial W_{1j}} & \cdots \\ \vdots & & & \\ \frac{\partial L}{\partial W_{i1}} & \cdots & \boxed{\frac{\partial L}{\partial W_{ij}}} & \cdots \\ \vdots & & & \vdots \end{bmatrix} = dW$$

## Backpropagating through the network



Given  $\nabla_{x^{(i)}} L$  if we can find  $dW^{(l)}$  and  $\nabla_{x^{(l-1)}} L$ , then we can recursively do this to find  $dW^{(i)}$  for all  $i$  (all the weights)

$$\begin{aligned} \nabla_{y^{(l-1)}} L &= \nabla_{x^{(l)}} L \cdot W^{(l)} \\ \nabla_{x^{(l-1)}} L &= \nabla_{y^{(l-1)}} L * g'(x^{(l-1)}) \\ \Rightarrow \nabla_{x^{(l-1)}} L &= \nabla_{x^{(l)}} L \cdot W^{(l)} * g'(x^{(l-1)}) \end{aligned}$$

So we have  $\nabla_{x^{(l-1)}} L$ . For  $dW^{(l)}$  we can use the equation derived above :

$$dW = \nabla_y L^T \cdot X^T$$

Note that here since the linear transform is

$$X^{(l)} = W^{(l)} \cdot Y^{(l-1)}$$

we have

$$dW^{(l)} = \nabla_{x^{(l)}} L^T \cdot Y^{(l-1)T}$$