

# DEEP LEARNING

## LECTURE 1 – INTRODUCTION

# TEACHING TEAM

## Instructors:

### Kailash Awati

- > Office: Rm 37, Level 3, Building 5D (Quay St)
- > Email: [kailash.awati@uts.edu.au](mailto:kailash.awati@uts.edu.au)
- > Phone: 9514 4708

### Alex Scriven

- > Email: [Alexander.Scriven@uts.edu.au](mailto:Alexander.Scriven@uts.edu.au)

# OVERVIEW OF SUBJECT

*This subject gives students a holistic understanding of deep learning, based on readings of research papers, coding exercises and practical assignments.*

Three pillars:

- > (Some) mathematics
- > (skimming of) academic papers
- > (coding) coding

What's not included:

- > Deep analysis of any particular architecture
- > Too much emphasis on deployment

# OVERVIEW OF SUBJECT

The SLOs:

- Form an intuitive and mathematical understanding of Vanilla Neural Networks
- Articulate the strengths, weaknesses and use cases of a variety of Deep Learning neural network architectures.
- Implement and optimise a variety of Deep Learning neural network architectures
- Communicate the selection, training & analysis of deep learning models for business use cases in a context-appropriate manner
- Synthesise and interpret relevant academic papers as well as other sources of knowledge relating to deep learning to build a model for current progress and potential future directions, commercial applications and ethical implications
- Understand and utilise the latest work towards visualising and uncovering insights from traditionally ‘black-box’ models.

# BLOCK SESSIONS OUTLINE

Block Session	Details
1 – 4 <sup>th</sup> April	<ul style="list-style-type: none"><li>• Course introduction</li><li>• Introduction to neural networks, building a vanilla neural network from scratch</li></ul>
2 – 11 <sup>th</sup> April	<ul style="list-style-type: none"><li>• Software and hardware</li><li>• Training and optimization in depth</li></ul>
3 – 2 <sup>nd</sup> May	<ul style="list-style-type: none"><li>• Computer vision: types of tasks, CNNs in depth, classic to advanced architectures.</li></ul>
4 – 9 <sup>th</sup> May	<ul style="list-style-type: none"><li>• Sequence models: RNN, LSTM, Transformers &amp; Attention</li><li>• Guest lecture &amp; Lab: NLP in action</li></ul>
5 – 16 <sup>th</sup> May	<ul style="list-style-type: none"><li>• Generative Adversarial Networks</li><li>• Deep Reinforcement Learning (time permitting)</li></ul>
6 – 30 <sup>th</sup> May	<ul style="list-style-type: none"><li>• Model deployment. Interactive lecture-lab building and deploying a simple webapp for your model.</li></ul>

# ASSIGNMENTS

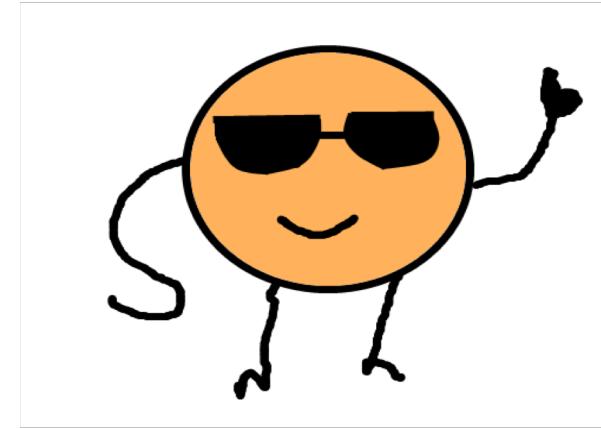
AT	Details	Release	Due	Days
1A	<ul style="list-style-type: none"><li>• Build and optimize a vanilla neural network from scratch (no frameworks)</li></ul>	02/04/2019	21/04/2019	17
1B	<ul style="list-style-type: none"><li>• Build and optimize an image classifier for a business use case</li></ul>	02/05/2019	26/05/2019	24
2A	<ul style="list-style-type: none"><li>• Literature review of key historic and modern academic papers on a chosen deep learning architecture.</li></ul>	16/05/2019	02/06/2019	17
2B	<ul style="list-style-type: none"><li>• Propose application of research undertaken in 2A to an industry problem</li></ul>	16/05/2019	02/06/2019	17
3	<ul style="list-style-type: none"><li>• Build and optimize a sequence-based model for a business use case</li></ul>	30/05/2019	20/06/2019	21

AT3 is groupwork. You may propose a more interesting project (\*conditions apply!)

# OVERVIEW OF SUBJECT

## Important notes:

- Nobody (including me) knows everything about every architecture. Just like 'classic' ML, we have areas we know better than others.
- Diving deep into a particular branch you may very well end up more well versed than me.
- *Facilitator* more than teacher
- This course will focus on breadth rather than depth

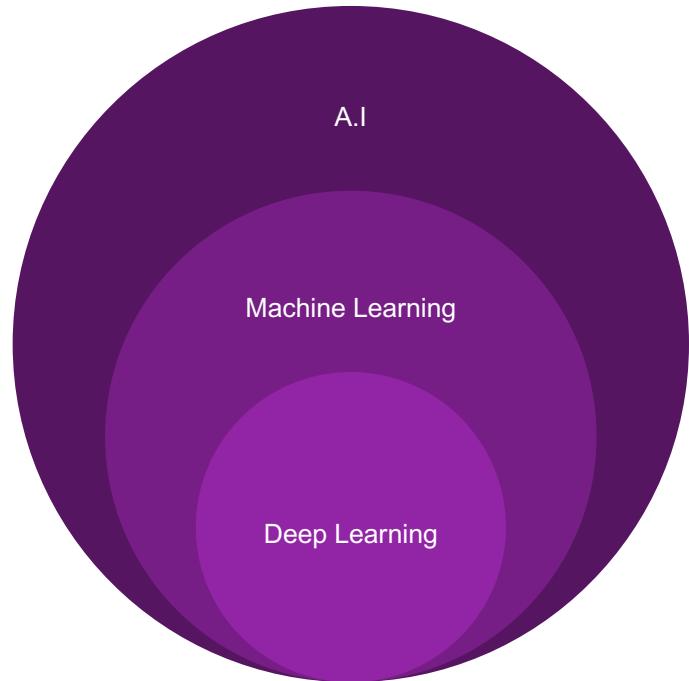


# A note on 'A.I'

This graph appears often, though different people will have different views.

Be aware that **most often** these days when people say 'A.I' they either mean:

- A. Software, that is distinctly none of the above (like rule-based systems)
- B. A neural net architecture (CNN, RNN, DRL, GAN)



# 'A.I' is 'locally' smart

*All you're seeing now — all these feats of AI like self-driving cars, interpreting medical images, beating the world champion at Go and so on — these are very narrow intelligences, and **they're really trained for a particular purpose.** They're situations where we can collect a lot of data...*

-Yann Lecun (Creator of CNN, Facebook Chief AI)

Key Take away: 'Generalised A.I' is a very different concept. Most A.I models break if taken slightly outside of what they are trained for

# An A.I Example

*Google Duplex (Voice assistant booking stunt)*

## **'Enter Duplex'**

*Google Duplex's conversations sound natural thanks to advances in understanding, interacting, timing, and speaking.*

*At the core of Duplex is a recurrent neural network (RNN) designed to cope with these challenges, built using TensorFlow Extended (TFX). To obtain its high precision, we trained Duplex's RNN on a corpus of anonymized phone conversation data.*

<https://ai.googleblog.com/2018/05/duplex-ai-system-for-natural-conversation.html>

# In other words.....

Geoffrey Hinton and Demis Hassabis: AGI is nowhere close to being a reality

KYLE WIGGERS @KYLE\_L\_WIGGERS DECEMBER 17, 2018 12:30 PM



Above: Geoffrey Hinton (left) and Demis Hassabis (right)

Image Credit: Google

Ad closed by Google  
Report this ad  
Why this ad? ⓘ

MOST READ

*Hassabis cautions that they by no means suggest AGI is around the corner — far from it. Unlike the AI systems of today, he says, people draw on intrinsic knowledge about the world to perform prediction and planning....*

*We don't have systems that can transfer in an efficient way knowledge they have from one domain to the next. I think you need things like concepts or extractions to do that*

[https://venturebeat.com/2018/12/17/  
geoffrey-hinton-and-demis-hassabis-  
agi-is-nowhere-close-to-being-a-  
reality/](https://venturebeat.com/2018/12/17/geoffrey-hinton-and-demis-hassabisagi-is-nowhere-close-to-being-a-reality/)

To be perfectly blunt.....

*“All the impressive  
achievements of deep  
learning amount to just  
curve fitting”*

*– Judea Pearl*



# Visually speaking...

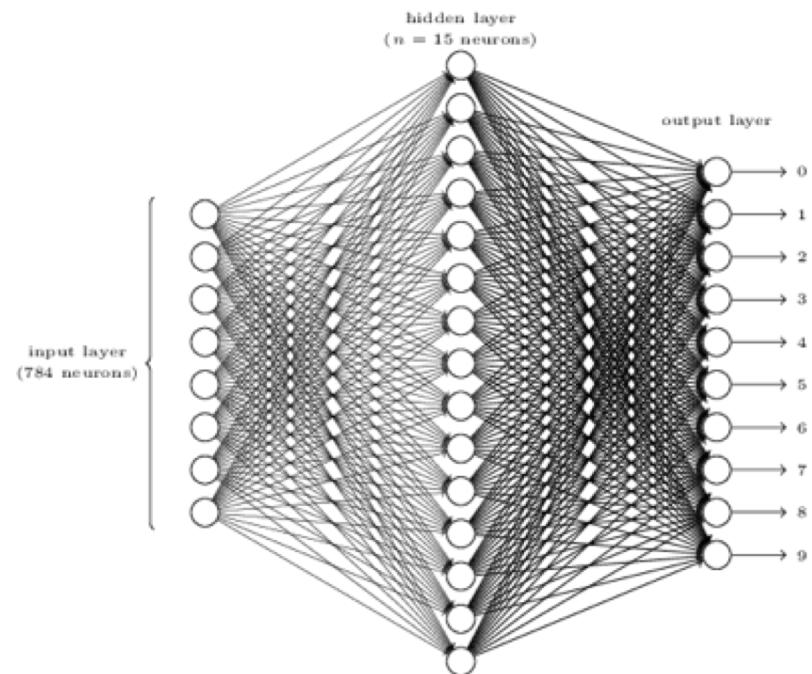


(An introduction to)

# Neural Networks

# Neural Networks

1. **WHAT** are Neural Networks
2. **WHY** are Neural Networks
3. **HOW** do we 'Neural Networks'



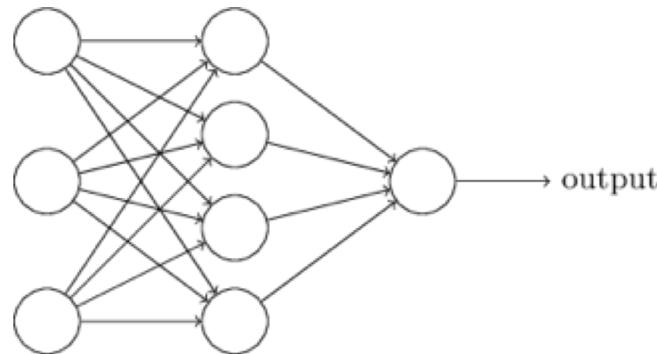
# [WHAT] Neural Networks

## neural network

*noun*

plural noun: **neural networks**

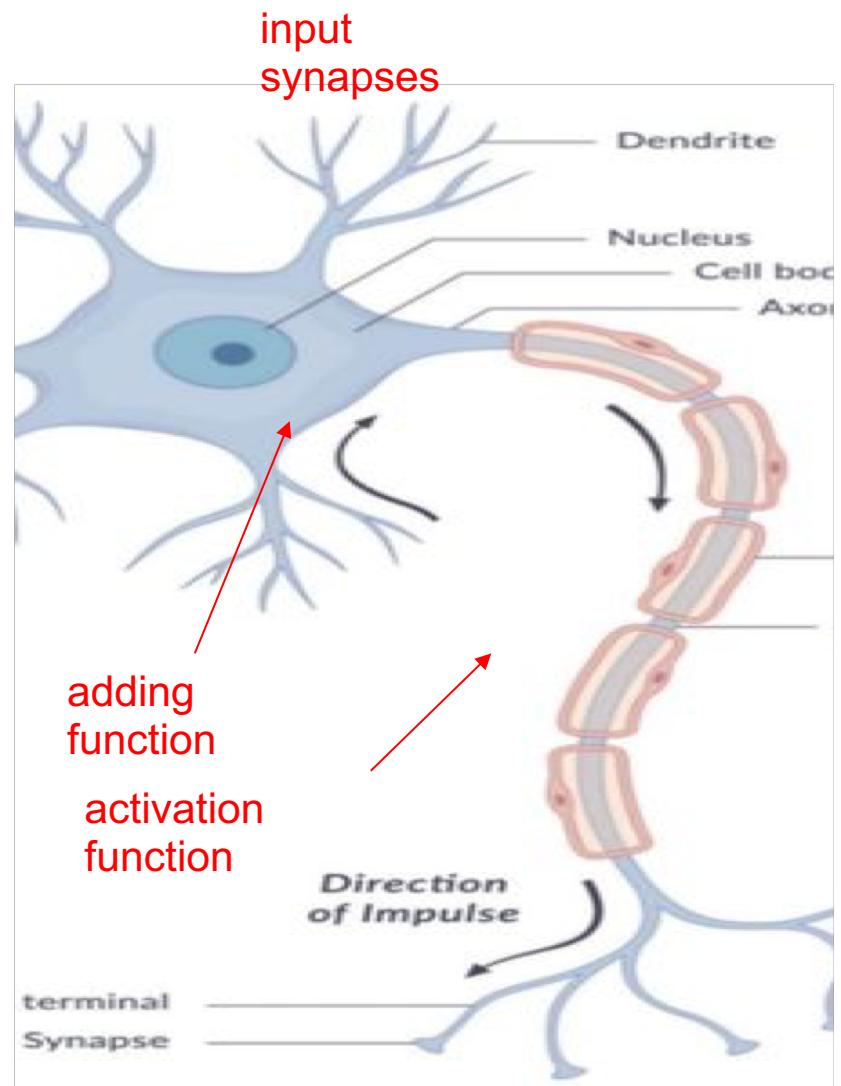
a computer system modelled on the human brain and nervous system.



# MODEL OF A NEURON

The neuron is an information processing unit fundamental to a neural network. Neurons have three characteristics:

1. A set of input synapses (*connecting links*), each with a weight
2. An *adding function* to sum the input signals
3. An *activation function* to limit the output range of the neuron



# Limitations of ‘Brain’ Model

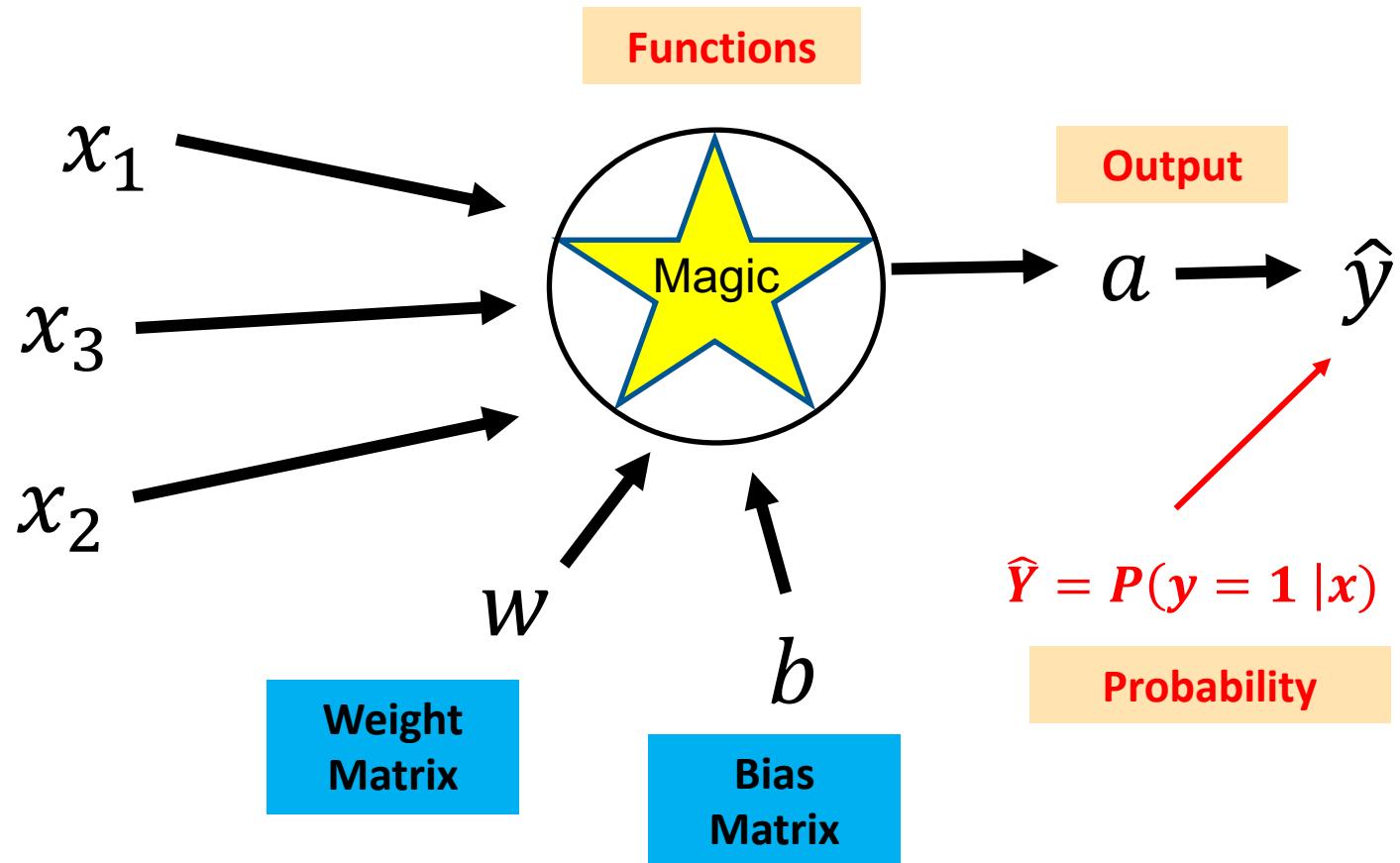
The ‘Brain’ model is more of a historical name that has stuck around (probably because it sounds cooler than ‘multi-layer perceptron’) but has little *real* relation to our understanding of the brain today\*.

In real life, ‘Neurons’:

1. Have many **different types** that are quite distinct from each other
2. The computations at the dendrites can be **very complex** functions
3. The simple weight-matrix approach is a vast simplification of the non-linear, **dynamic system** of interactions of our synapses.

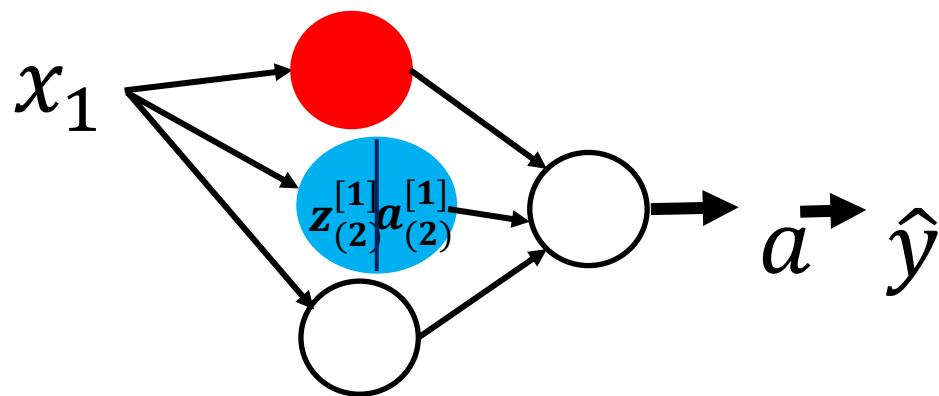
\*In computer-vision, the techniques hold closer semblance to real human cognition.

# Let's start with a neuron



# Some notation

- Let us use superscript with square brackets to denote which layer we are referring to
- Let us use subscript with round brackets to denote which neuron (top to bottom) in a layer we are referring to
- So  $z_{(2)}^{[1]}$  is the ‘adding’ function of the second neuron in the 1<sup>st</sup> layer.



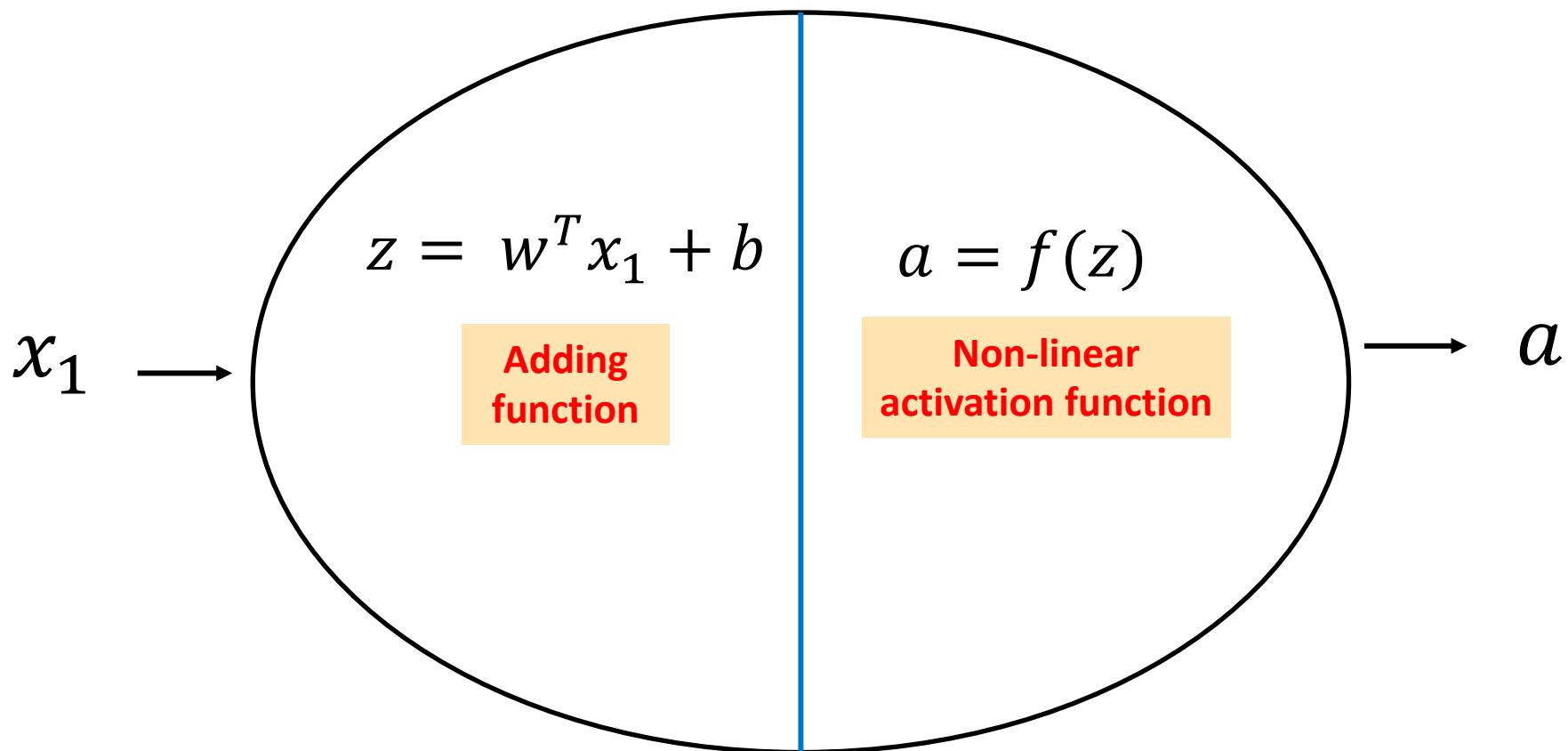
## 'W' AND 'B'

- '**w**' is our matrix of weights that we will apply to each of our input data points.

(This becomes our hyperparameter to 'tune')

- '**b**' is our bias term.
  - Similar to intercept in linear regression and serve much the same function (allows us to shift the function above the origin)
  - Has a value of 1 and is not connected to other nodes.
  - Without it, we are assuming our activation is through the origin
    - $Y = mx + \text{b}$

# Inside a Neuron



**Why transpose W?**

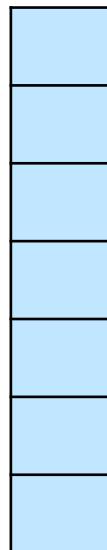
# Why transpose W?

- Recall: ' $w$ ' is our matrix of weights that we will apply to each of our input data points.

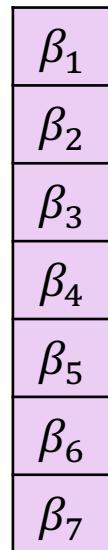
$w$  will therefore have a weight per variable

- We have  $w \in \mathbb{R}^{n_x}$  where  $n_x$  is the number of features of  $x$

Consider 1  
training example  
of 10 features



$x_1$



$w_1$

Can I multiply  
these together  
currently?

No!

We must have form

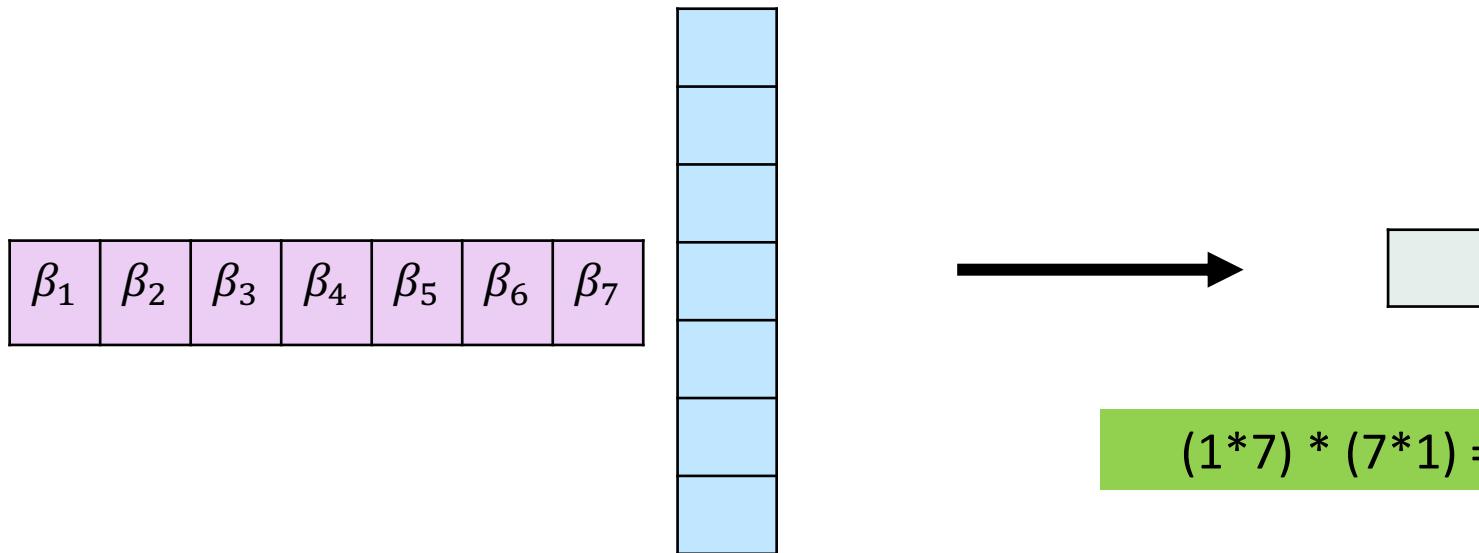
$$\mathbb{R}^{n_x * m} * \mathbb{R}^{m * a}$$

$$\text{Or } (m * n) * (n * p) = m * p$$

Where  $m$  = rows,  $n$  = columns

# Why transpose W?

- To allow us to undertake matrix calculus we must transpose something



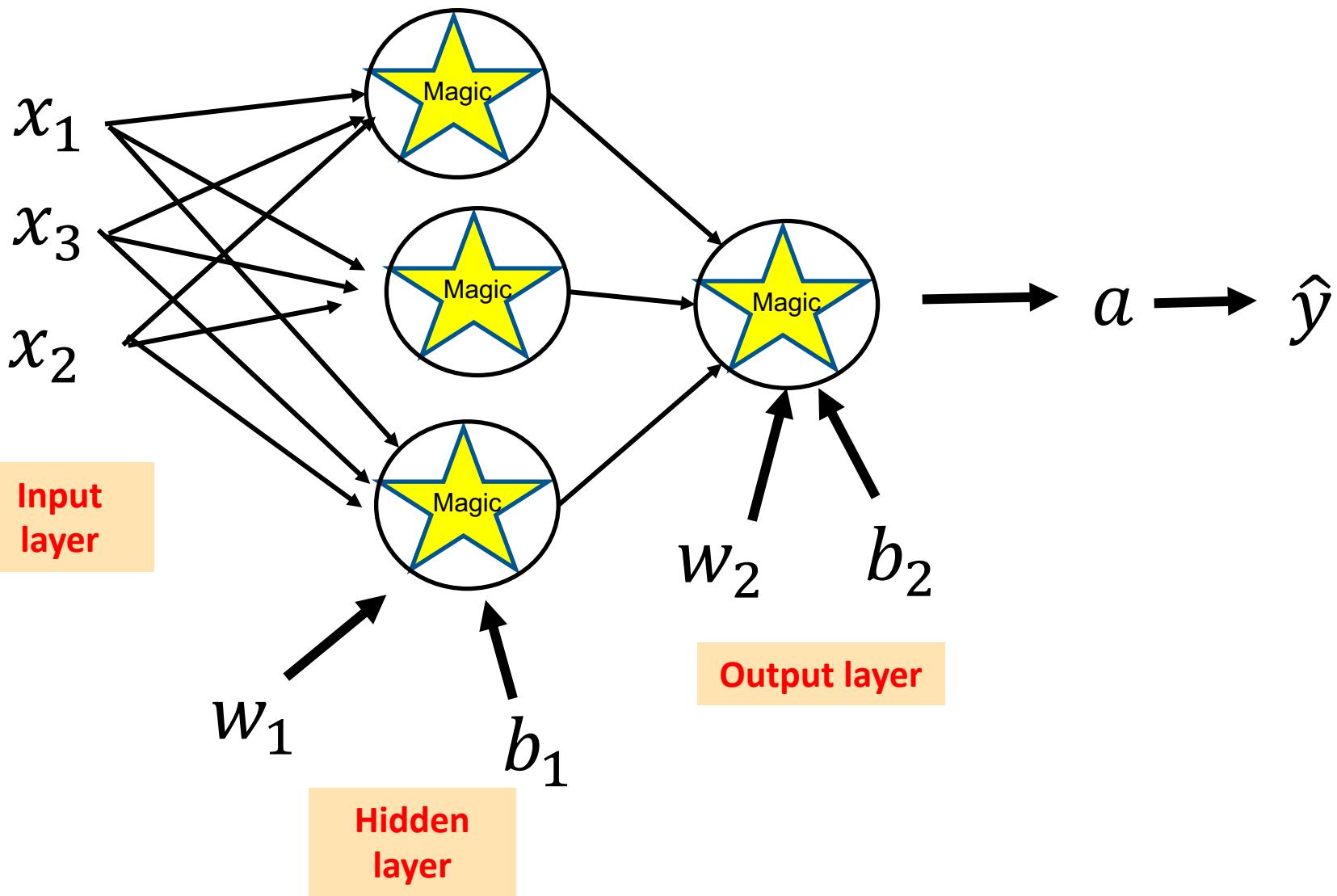
Can I multiply  
these together  
currently?

$$(1 \times 7) * (7 \times 1) = (1 \times 1)$$

Yes!

Columns(x) = rows(w)

# Neuron to ‘network’



# RELATION TO LINEAR MODELS

- A single neuron is effectively a generalised linear model, i.e. a linear regression model inside a function  $f$
- If the activation function ( $f$ ) is the logistic function, then our single neuron model is the same as logistic regression
- Neural networks go much further than linear models, due to the *network* of neurons

# ACTIVATION FUNCTIONS

- What is the activation for?
  - These are to scale our output and add non-linear properties
- Why non-linear?
  - Consider if we only undertook ‘adding’ functions at each node.
  - Then we are only making linear transformations. Linear-by-linear-by-linear.
  - Hence this is no more powerful than one linear node!

# Let's prove that!

- Consider a NN with no activation function, we just do the multiplication step:

- $\mathbf{a}^l = z^l = \mathbf{w}_l^T \mathbf{x}_l + b_l$
- $a^{l+1} = z^{l+1} = \mathbf{w}_{l+1}^T \mathbf{a}^l + b_l$

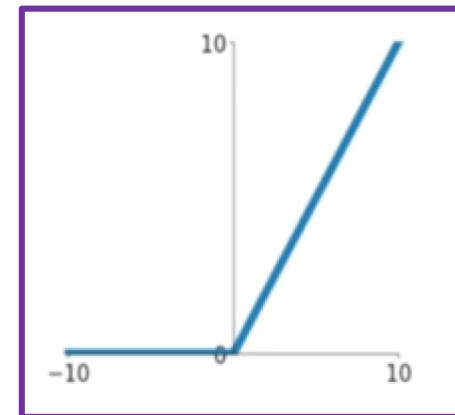
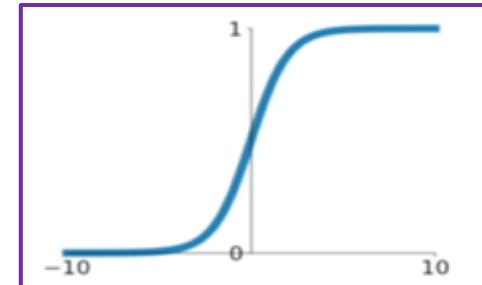
Substitute Line 1 into Line 2

- $a^{l+1} = z^{l+1} = \mathbf{w}_{l+1}^T * (\mathbf{w}_l^T \mathbf{x}_l + b_l) + b_l$

We are just making linear transformations, no  
'curve' to our function!

# Some common activation functions

- Sigmoid
  - $\sigma(x) = \frac{1}{1+e^{-x}}$
  - Historically popular, not anymore (except for last layer)
- ReLU (Rectified linear unit) (Krizhevsky et al. 2012)
  - $\max(0, x)$
  - Safe, very popular these days. Fixes a number of problems with sigmoid to do with gradient flow.

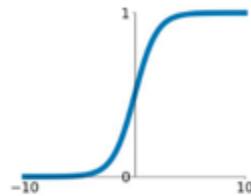


# Other Activation Functions

## Activation Functions

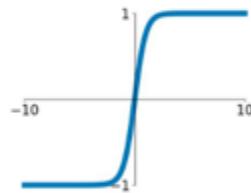
### Sigmoid

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



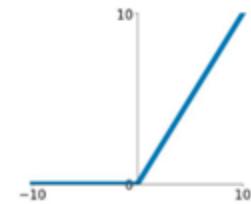
### tanh

$$\tanh(x)$$



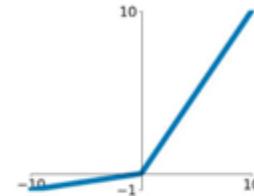
### ReLU

$$\max(0, x)$$



### Leaky ReLU

$$\max(0.1x, x)$$

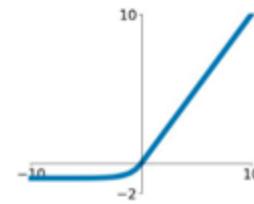


### Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

### ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# From instances to vectors

- To ease the maths in our networks, we like to operate in vectors and matrixes.
- This means that rather than a ‘for loop’ we can simply undertake efficient matrix calculations.
- So let us take all our weights and stack them:

$$\circ \quad W^{[1]T} = \begin{bmatrix} - & w_1^{[1]T} & - \\ - & w_2^{[1]T} & - \\ - & w_3^{[1]T} & - \end{bmatrix}$$

- And the same for our Z, A and X (now using capital letters to denote a matrix).

$$X = \begin{bmatrix} | & | & | \\ x_1 & x_2 & x_3 \\ | & | & | \end{bmatrix}$$

$X \in \mathbb{R}^{n_x * m}$   
(Features high,  
samples long)

# From instances to vectors

- Given X is of the form:

$$X = \begin{bmatrix} | & | & | \\ x_1 & x_2 & x_3 \\ | & | & | \end{bmatrix} \quad X \in \mathbb{R}^{n_x * m}$$

(Features high, samples long)

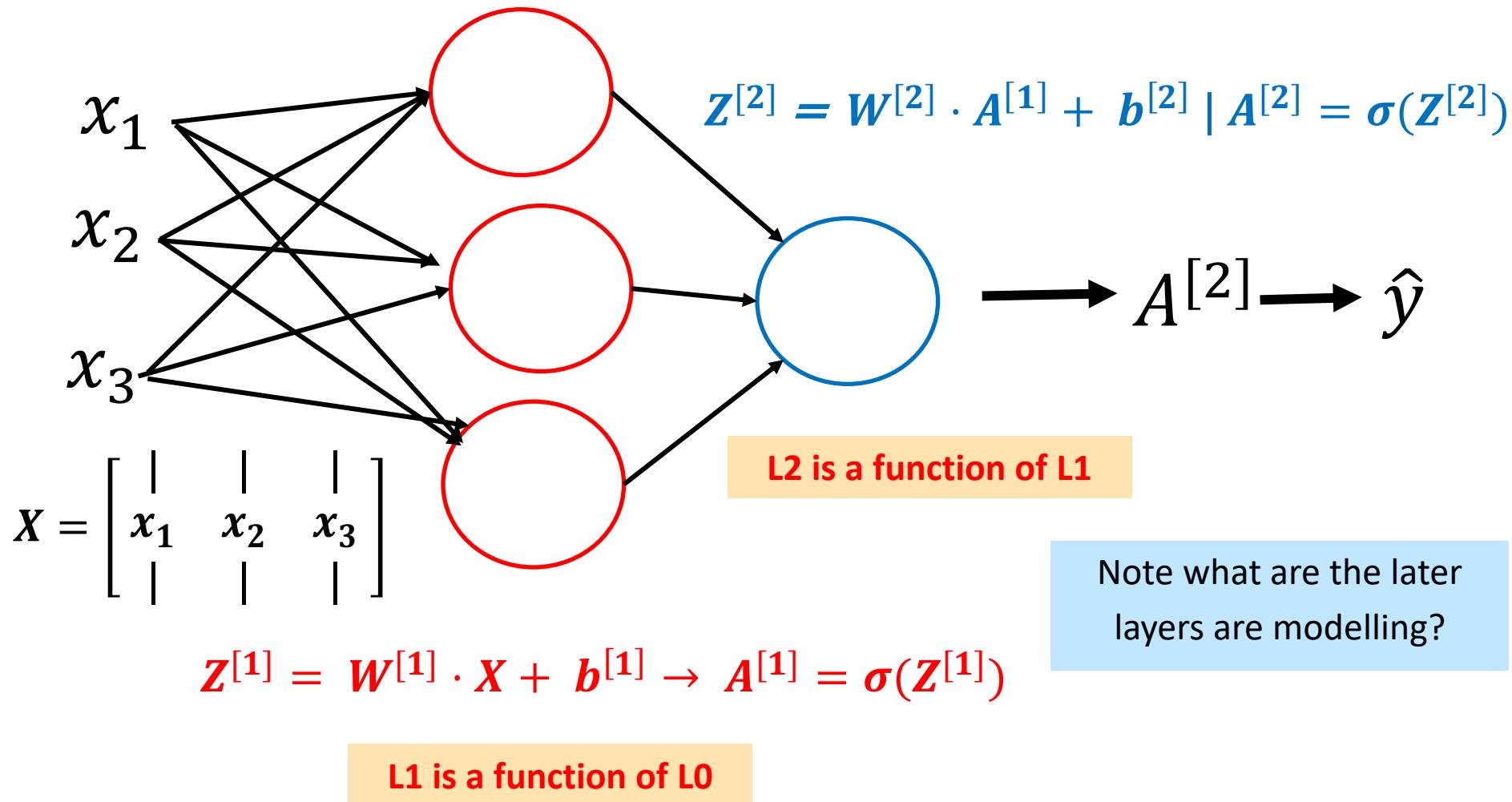
- What is Y's dimension?

$$Y = [ y_1 \quad y_2 \quad y_3 ]$$

$$Y \in \mathbb{R}^{1 * m}$$

(Features high, samples long)

# The matrix algebra through our network



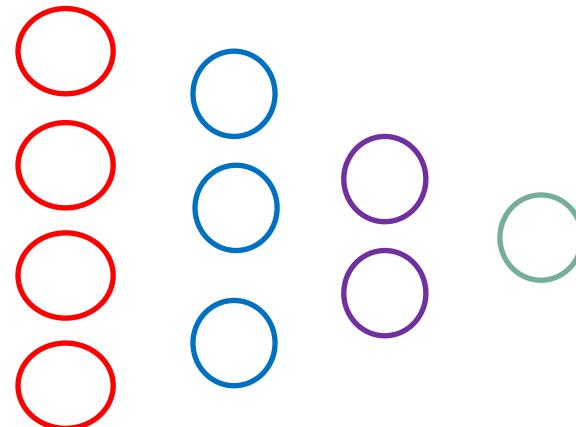
# Generalizing the algebra

- The 'Z' of any given layer, L, can be found with the following formula:
  - $Z^{[L]} = W^{[L]} \cdot A^{[L-1]} + b^{[L]}$
- And the 'A' (activation matrix) of any given layer, L, is found by:
  - $A^{[L]} = g^{[L]}(Z^{[L]})$
  - Where g is the activation function (sigmoid, ReLU, Tanh etc)

# The size of our matrices

- Given the relationships between layers, our different matrices will have defined, relative sizes.

$$X = \begin{bmatrix} | & | & | & | & | \\ x_1 & x_2 & x_3 & x_4 & x_5 \\ | & | & | & | & | \end{bmatrix} \rightarrow$$



- Example above where  $x_n = (1,3)$  : What is the dimensions of each step?
  - $W^{[l]} = (n^{[l]}, n^{[l-1]}) = dW^{[l]}$
  - $Z^{[l]} = A^{[l]} = (n^{[l]}, m) = dZ^{[l]} = dA^{[l]}$
  - $b^{[l]} = (n^{[l]}, 1) = db^{[l]}$

# The size of our matrices





$$A^{[0]} = (3,5)$$

$$W^{[1]} = (4,3)$$

$$Z^{[1]} = (4,5)$$

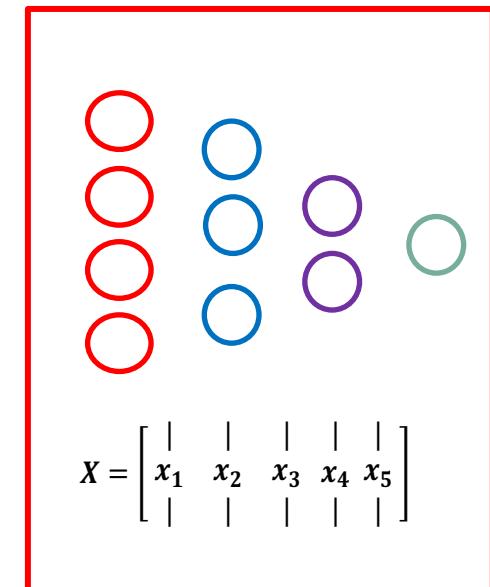
$$b^{[1]} = (4,1)$$

- Let's calculate the sizes:

- $W^{[l]} = (n^{[l]}, n^{[l-1]}) = \mathbf{d}W^{[l]}$

- $Z^{[l]} = A^{[l]} = (n^{[l]}, m) = \mathbf{d}Z^{[l]} = \mathbf{d}A^{[l]}$

- $b^{[l]} = (n^{[l]}, 1) = \mathbf{d}b^{[l]}$



# Our matrices (formulas)


=





+


$$Z^{[1]} = (4,5)$$

$$W^{[1]} = (4,3)$$

$$A^{[0]} = (3,5)$$

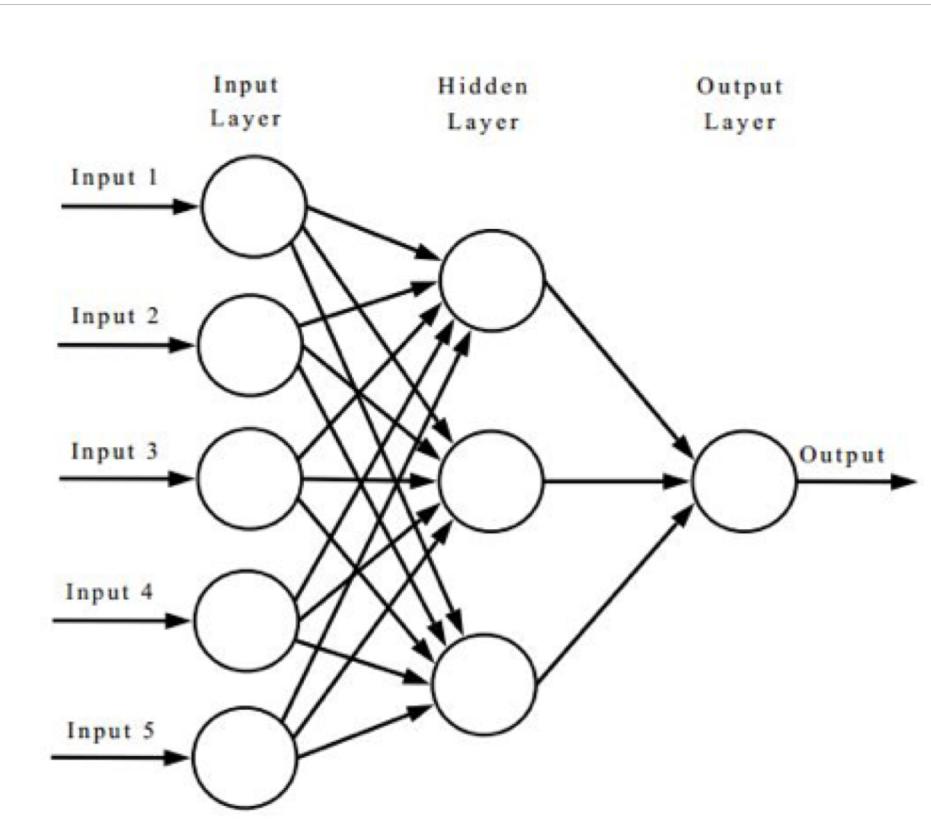
$$b^{[1]} = (4,1)$$

$$Z^{[L]} = W^{[L]} \cdot A^{[L-1]} + b^{[L]}$$

# FEEDFORWARD NEURAL NETWORKS

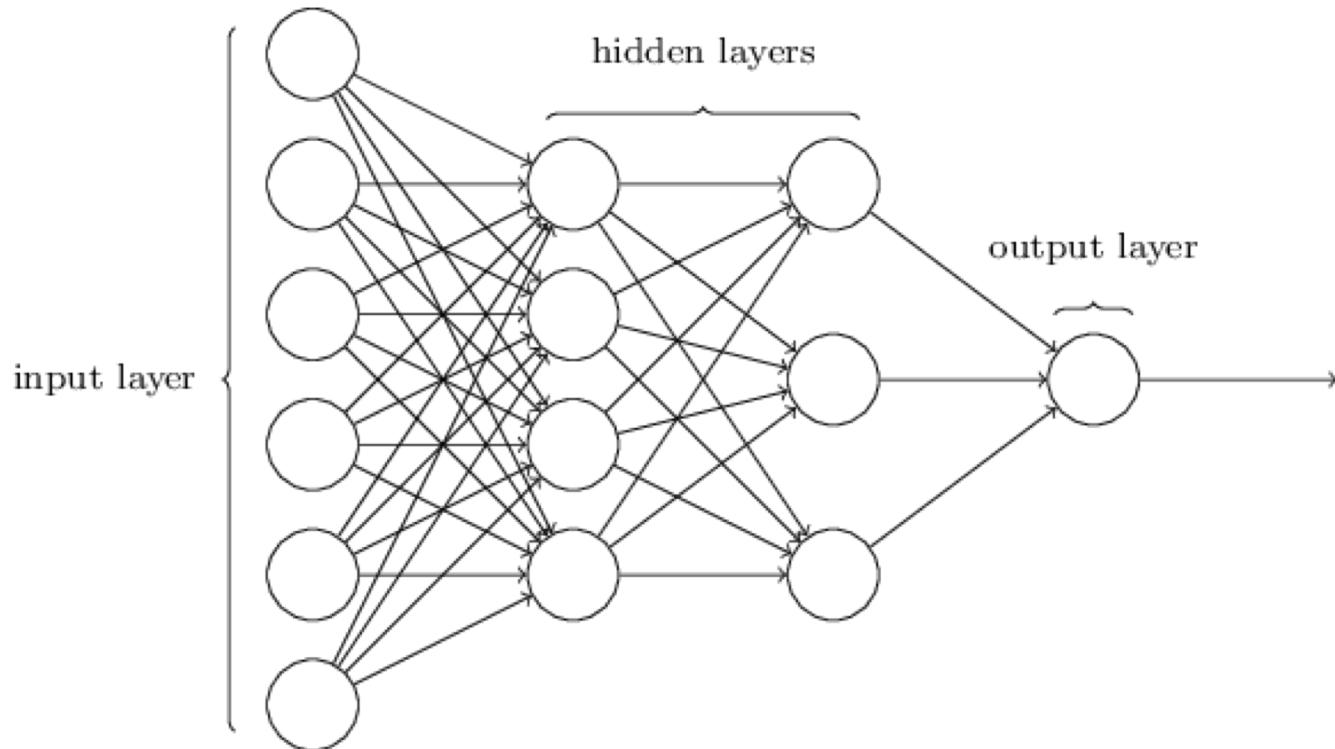
- What we have just seen is called a ‘feedforward neural network’
- It is *feedforward* because information passes through in one direction only, from inputs, through the hidden layer and to the output
- To simplify the matrix algebra we had before:

$$f(x) = f^{(2)}(f^{(1)}(x))$$



# FEEDFORWARD NEURAL NETWORKS

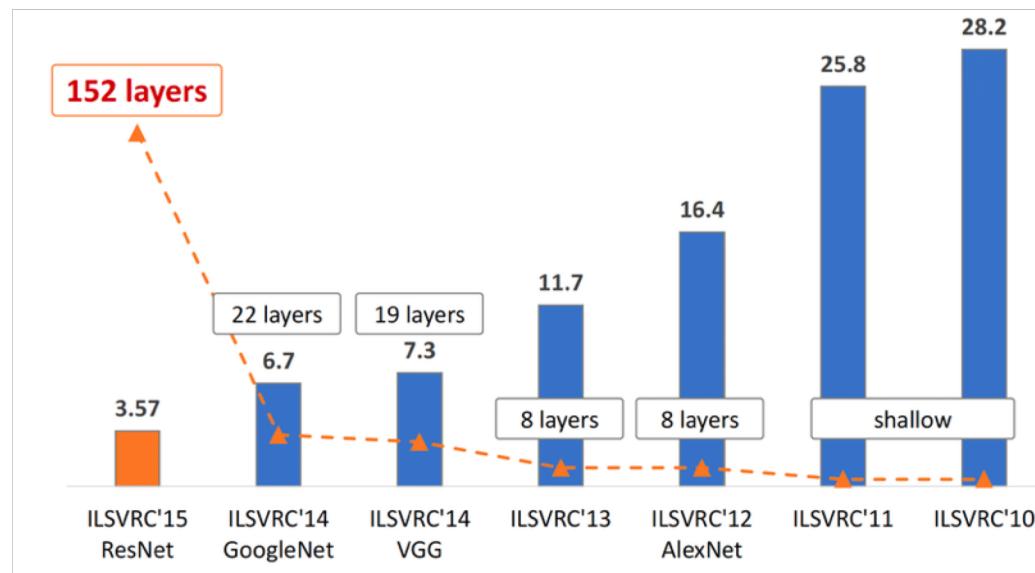
- It is common to have many hidden layers (a deeper neural network)



$$f(x) = f^{(3)} \left( f^{(2)} \left( f^{(1)}(x) \right) \right)$$

# From NN to 'DEEP LEARNING'

- Primarily uses deep neural network architectures
- More layers (deeper) allows for more flexible models
- Requires much more computing power and larger data sets
- Can learn non-linear transformations of data as features, and hierarchies of features



# How big do these networks get?

<https://vimeo.com/154085950>

Alexnet (2012) had  
around 138M  
parameters,

```
INPUT: [224x224x3]    memory: 224*224*3=150K  params: 0      (not counting biases)
CONV3-64: [224x224x64] memory: 224*224*64=3.2M  params: (3*3*3)*64 = 1,728
CONV3-64: [224x224x64] memory: 224*224*64=3.2M  params: (3*3*64)*64 = 36,864
POOL2: [112x112x64]   memory: 112*112*64=800K  params: 0
CONV3-128: [112x112x128] memory: 112*112*128=1.6M  params: (3*3*64)*128 = 73,728
CONV3-128: [112x112x128] memory: 112*112*128=1.6M  params: (3*3*128)*128 = 147,456
POOL2: [56x56x128]    memory: 56*56*128=400K  params: 0
CONV3-256: [56x56x256] memory: 56*56*256=800K  params: (3*3*128)*256 = 294,912
CONV3-256: [56x56x256] memory: 56*56*256=800K  params: (3*3*256)*256 = 589,824
CONV3-256: [56x56x256] memory: 56*56*256=800K  params: (3*3*256)*256 = 589,824
POOL2: [28x28x256]    memory: 28*28*256=200K  params: 0
CONV3-512: [28x28x512] memory: 28*28*512=400K  params: (3*3*256)*512 = 1,179,648
CONV3-512: [28x28x512] memory: 28*28*512=400K  params: (3*3*512)*512 = 2,359,296
CONV3-512: [28x28x512] memory: 28*28*512=400K  params: (3*3*512)*512 = 2,359,296
POOL2: [14x14x512]    memory: 14*14*512=100K  params: 0
CONV3-512: [14x14x512] memory: 14*14*512=100K  params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K  params: (3*3*512)*512 = 2,359,296
CONV3-512: [14x14x512] memory: 14*14*512=100K  params: (3*3*512)*512 = 2,359,296
POOL2: [7x7x512]      memory: 7*7*512=25K  params: 0
FC: [1x1x4096]         memory: 4096  params: 7*7*512*4096 = 102,760,448
FC: [1x1x4096]         memory: 4096  params: 4096*4096 = 16,777,216
FC: [1x1x1000]         memory: 1000  params: 4096*1000 = 4,096,000

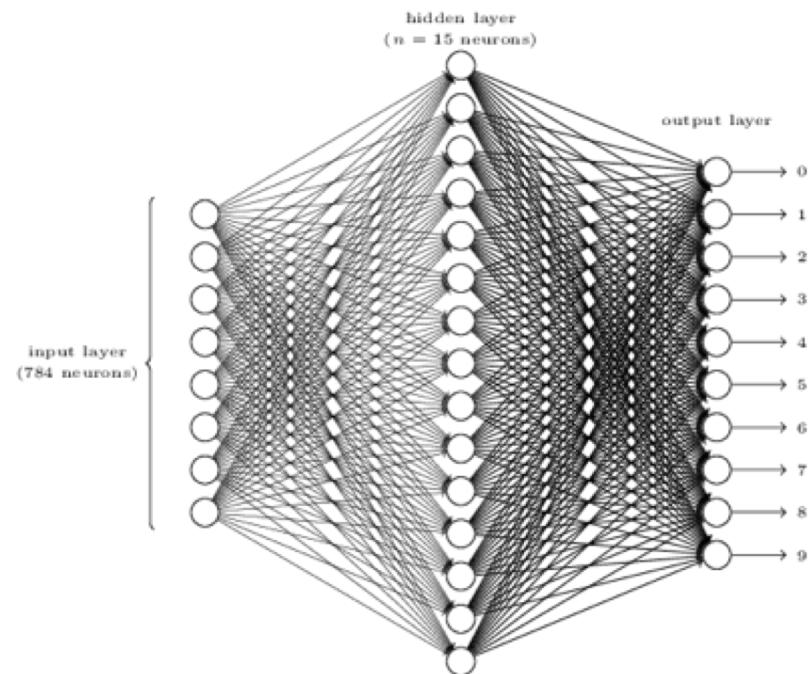
TOTAL memory: 24M * 4 bytes ~= 93MB / image (only forward! ~*2 for bwd)
TOTAL params: 138M parameters
```

# Confused? – Key Takeaways

- Lots of little ‘Neurons’ made up of:
  - A linear function (weight multiplication + bias)
  - A non-linear activation function
- We stack lots of these to be able to map complex data relationships and ‘weird’ data types that conventional models cannot.
  - $f_3 = f_2(f_1(x))$
- To really appreciate how it works and the *beauty* of the different models, make sure you understand the linear algebra occurring

# Neural Networks

1. ~~WHAT~~ are Neural Networks
2. **WHY** are Neural Networks
3. **HOW** do we 'Neural Networks'

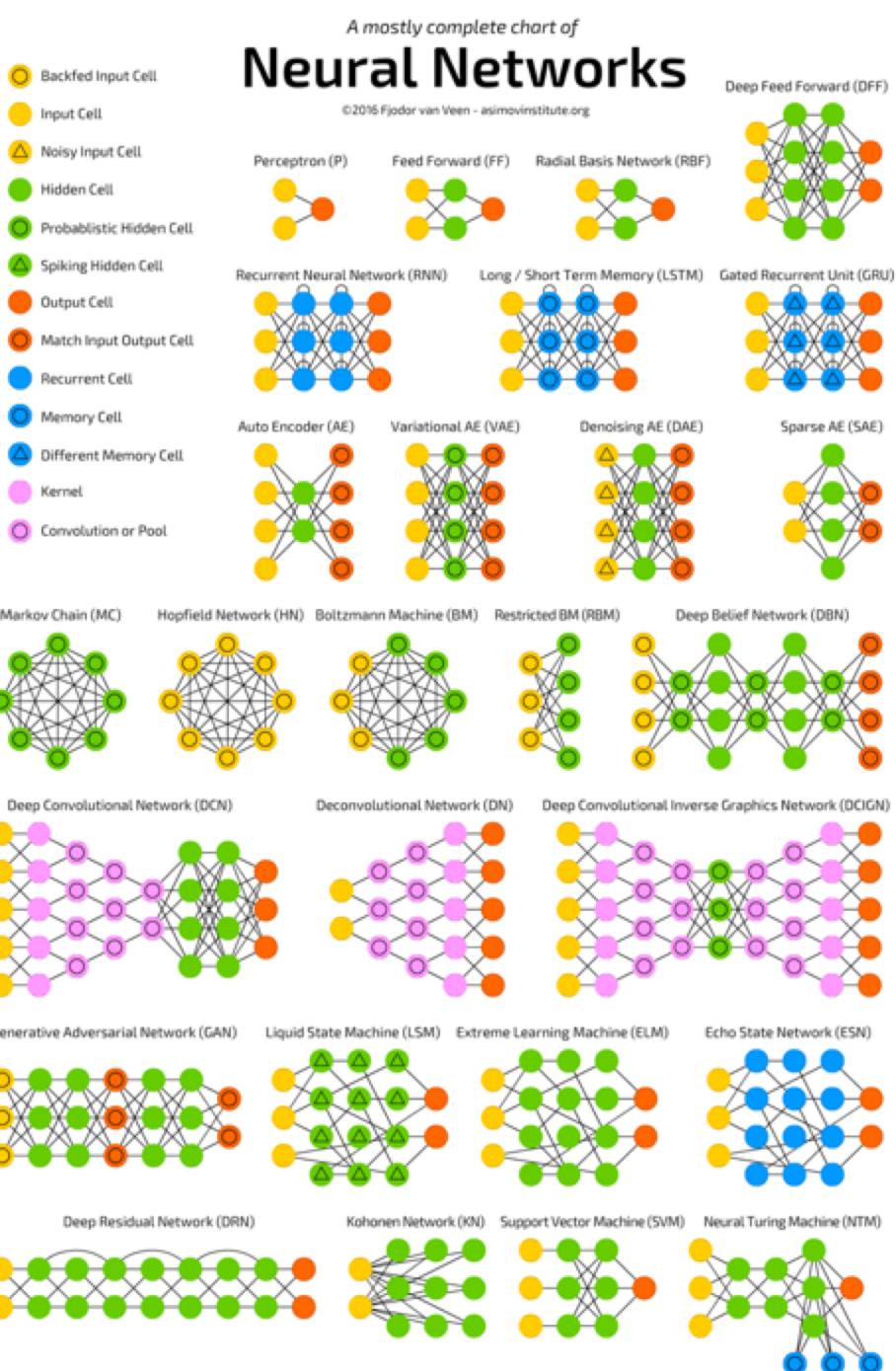


# WHY IT WORKS

- With a sufficiently large (or deep) hidden layer(s), it is possible to represent *any continuous* function of the inputs with arbitrary accuracy
- Think back to the bias-variance trade-off:
  - By using a network of neurons (think *nested GLMs*), we are providing a great deal more flexibility
  - The catch is that we are increasing variance – the model is more likely to overfit to our training data
  - One issue is that there is no clear guidance on selecting the width of each hidden layer (number of neurons) or the depth (number of hidden layers) for a given problem

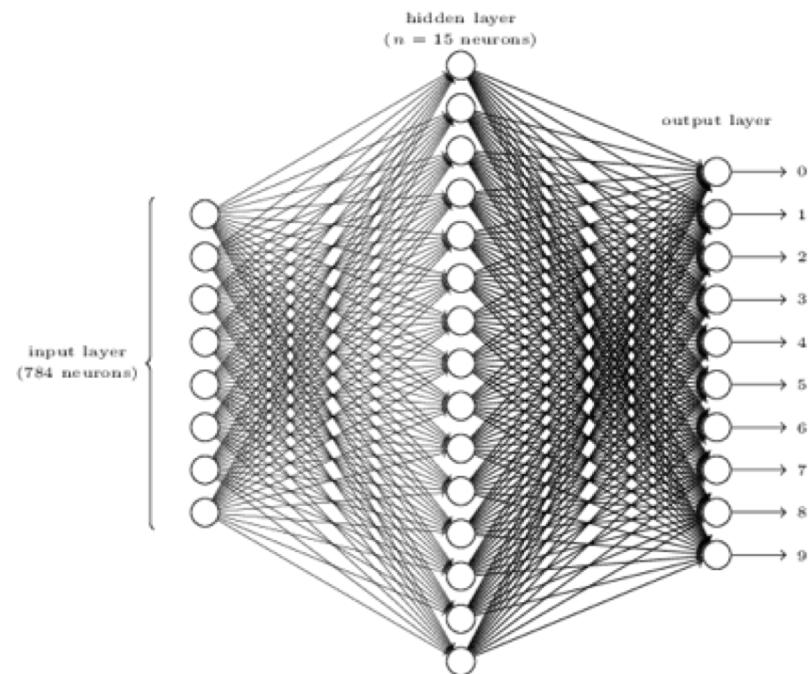
# OTHER TYPES

- **Convolutional neural network**
  - Used in computer vision
- **Recurrent neural network**
  - Sequence Data
- **LSTM**
  - Sequence data (Time series)
- **Auto-encoder**
  - Unsupervised learning
- **Deep Reinforcement Learning**
  - ‘Games’



# Neural Networks

1. ~~WHAT~~ are Neural Networks
2. ~~WHY~~ are Neural Networks
3. **HOW** do we 'Neural Networks'



# How to ‘A.I’ in (6) simple steps

1. **Initialize your network**
  - **Select number of hidden layers, number of hidden units per layer, activation functions**
2. **‘Feed Forward’**
  - **We have covered this essentially!**
3. Calculate a loss
  - How bad did our model do?
4. ‘Back propagate’
  - This will allow us to see how our weights affect our loss
5. Update weights
6. Repeat
7. (Optional – regularize)

# One thing we missed! (Data Preparation)

- We want to **zero-center/ zero-mean** the data.

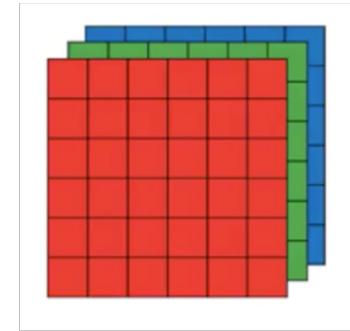
Why?

- So our output is not too sensitive to small changes in weight matrix.
- To assist the gradient optimization path (link to *sigmoid* decline
- We want to **normalize** the data.

Why?

- So our variables can equally contribute  
Different scales will make an *elongated* cost function and hence need a small learning rate step – training takes longer.

Not really needed  
for images – why?



# Data Preparation: Extra

- Look up whitening (Kessy et al., 2016)
  - Similar to PCA but components have unit variance
  - Can first perform PCA, then divide principal components by  $\text{sqrt}(\text{eigenvalues})$
- Additional step could be ZCA ('Mahalanobis whitening')
  - Whitened data has minimal adjustments to the 'space' of the original data.

Generally for images, only zero-mean. For other DNN tasks, consider more pre-processing

# Weight initialization

- Proposition 1: Simply initialize the weights to zero?

**Bad Idea – Why?**

- The weights would all do the same thing, output same values, receive same gradient etc.
- Proposition 2: Add a small non-zero value

**Ok – but has some issues  
(See AT1)**

- Proposition 3: Other, advanced methods

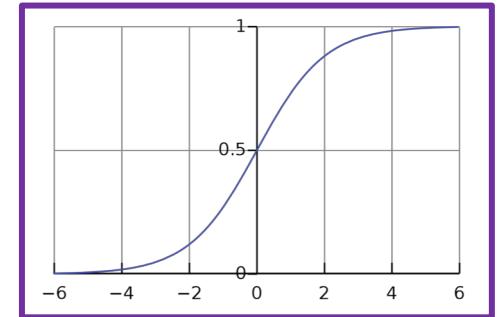
**(See AT1)**

# Creating a loss function

- Let us consider a binary outcome, then we can use a sigmoid loss function

$$L(\hat{Y}, y) = -(y \log \hat{Y} + (1 - y) \log(1 - \hat{Y}))$$

- Consider if  $Y=1$ 
  - $(1 - y) \log(1 - \hat{Y})$  becomes 0
  - We are left with  $-(1) \log(\hat{Y})$
  - What would be  $\hat{Y}$  when we are really sure it IS? (0.999?)
    - $\log(0.990) = \text{Zero!}$
- Consider if  $Y=0$ 
  - $y \log \hat{Y}$  becomes 0
  - We are left with  $\log(1 - \hat{Y})$
  - What would be  $\hat{Y}$  when we are really sure it ISN'T? (0.001?)
    - $\log(0.999) = \text{Zero!}$

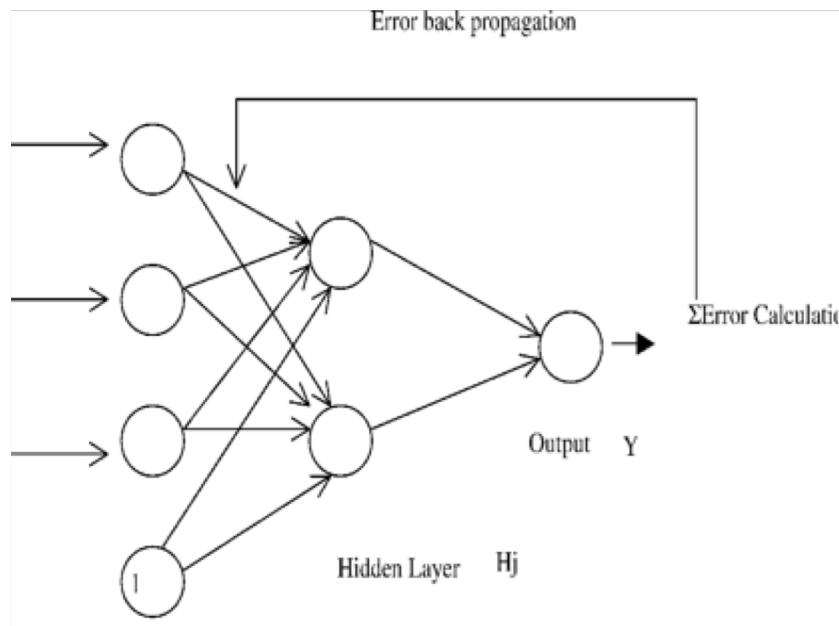


# BACK-PROPAGATION

- Once we have decided on a neural network architecture, the model boils down to a set of weights
- We cannot use optimisation methods from linear models to determine all these weights. Why?
- We use the back-propagation algorithm instead:
  1. Find the error at the output layer (i.e. classification error, MSE)
  2. Use this error to adjust the weight between the last hidden layers and the output layer → we *back-propagate* the error at the output layer to the hidden layer
  3. Repeat 1 and 2 until all hidden layers are updated

# BACK-PROPAGATION

- The idea is that each hidden node is ‘responsible’ for some fraction of the error.
- We divide the error according to the weight of the connection between each hidden node and the output – hidden nodes with a greater reduction in error get more weight



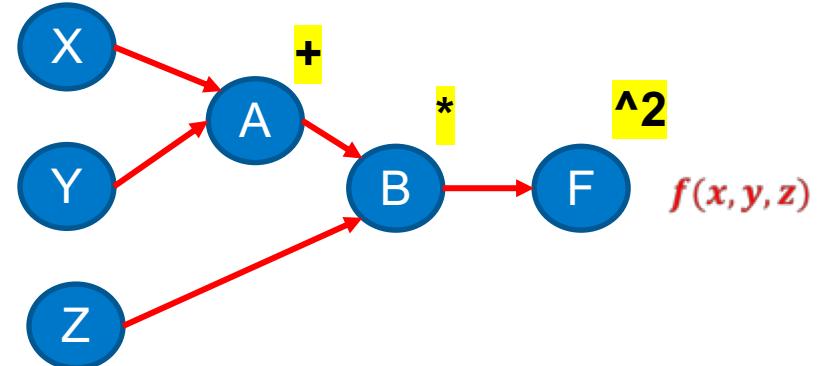
# BACK-PROPAGATION

- Consider a simple computational graph made from the function

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

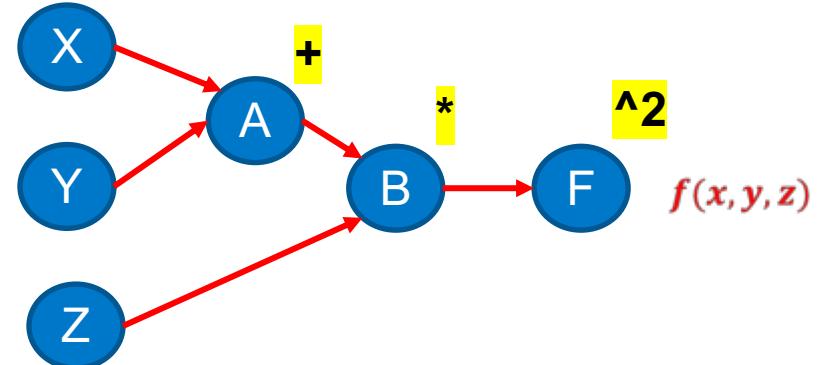
- How does A affect  $f(x, y, z)$ ?

$$\frac{df}{dA} ??$$



# BACK-PROPAGATION

- How does A affect  $f(x, y, z)$ ?  
 $\frac{df}{dA}$  ??
- Hint: What we have are intermediary functions:  
 $A = x + y$



Luckily calculus has a neat trick called the 'chain rule'

$$\frac{df}{dA} = \frac{\partial f}{\partial B} * \frac{\partial B}{\partial A}$$

So what is  $\frac{\partial f}{\partial B}$ ?

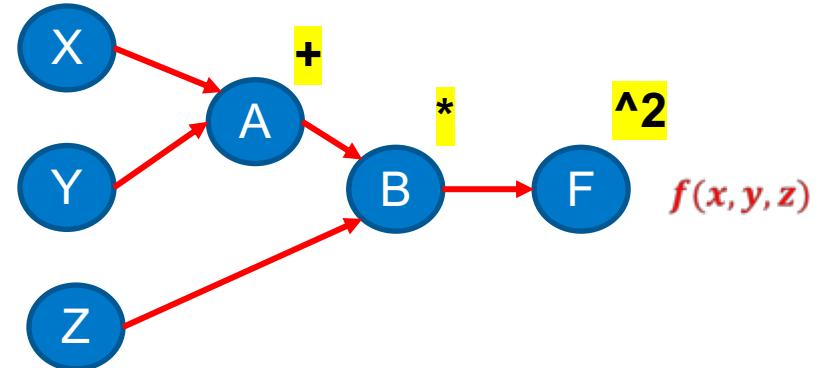
$$\frac{\partial f}{\partial B} = 2B \text{ and } \frac{\partial B}{\partial A} = Z$$

$$\text{so } \frac{df}{dA} = 2BZ$$

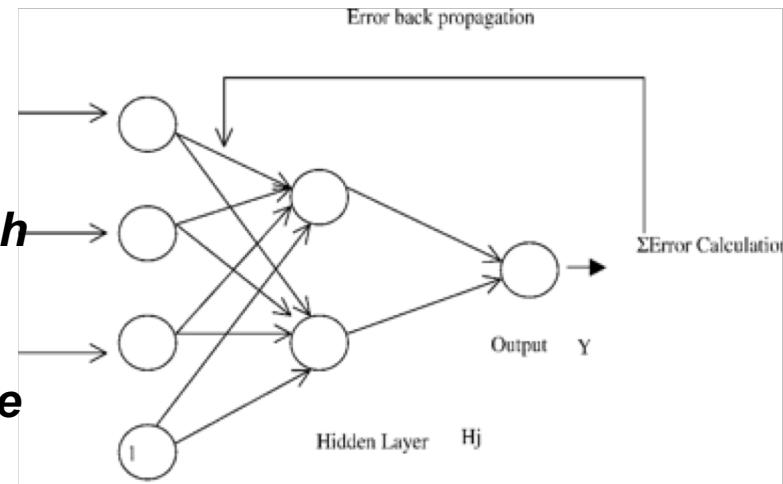
# BACK-PROPAGATION

- How does X affect  $f(x, y, z)$ ?  
 $\frac{df}{dX}$  ??
- What is the chain rule for this?

$$\frac{df}{dx} = \frac{\partial f}{\partial B} * \frac{\partial B}{\partial A} * \frac{\partial A}{\partial x}$$



***The idea is that each hidden node is ‘responsible’ for some fraction of the error***



# BACK-PROPAGATION for Matrices

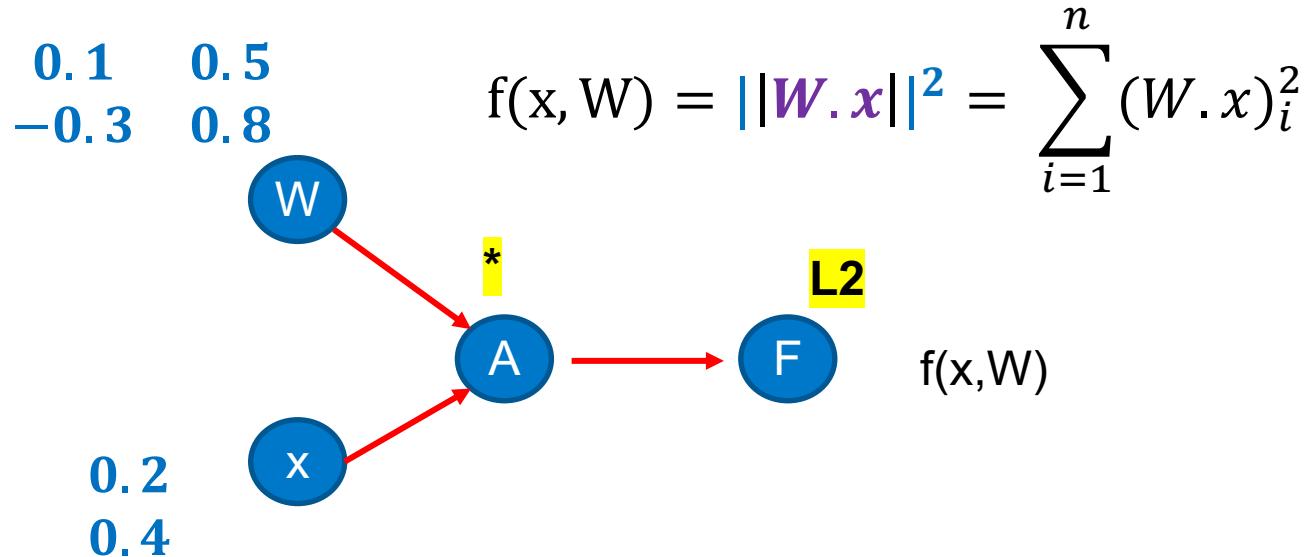
- What if we don't have scalars but have matrices?

Everything holds true, except we now get a **jacobian matrix** (first order partial derivatives of the weight matrices with respect to the loss function)

- An example computational graph:
  - **Matrix multiplication** (then)
  - **L2 transformation** (sum of the square of each element) v

$$f(x, W) = \|\mathbf{W} \cdot \mathbf{x}\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$$

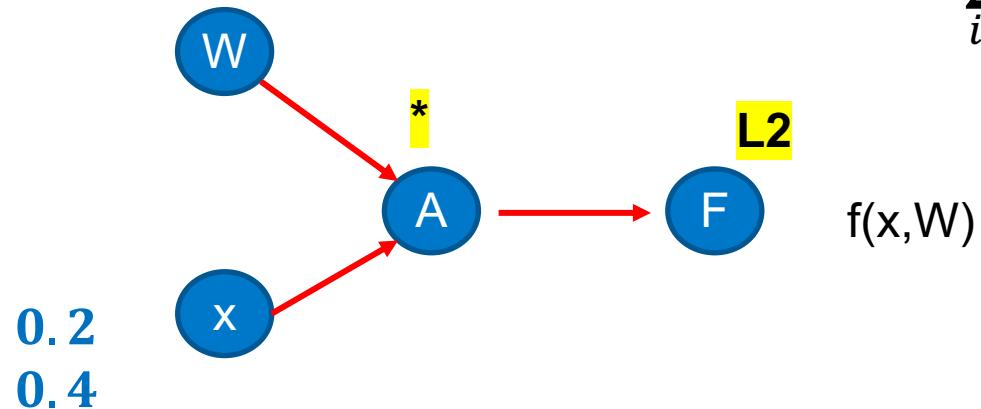
# Jacobian walkthrough



$$\begin{matrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{matrix} \bullet \begin{matrix} 0.2 \\ 0.4 \end{matrix} = \text{????}$$

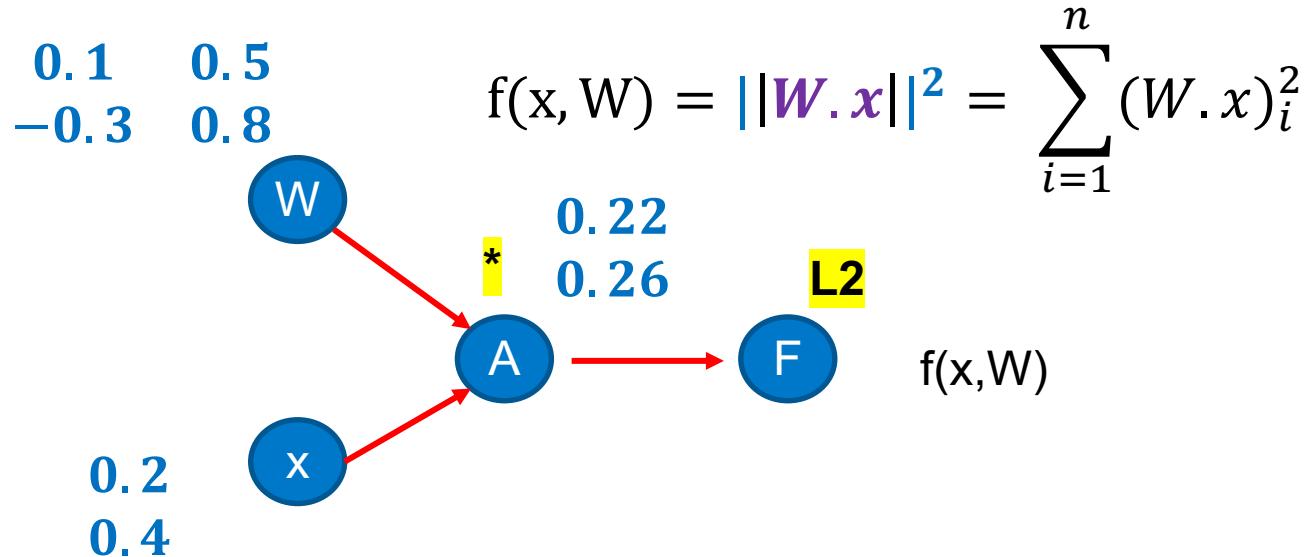
# Jacobian walkthrough

$$\begin{matrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{matrix} \quad f(x, W) = ||W \cdot x||^2 = \sum_{i=1}^n (W \cdot x)_i^2$$



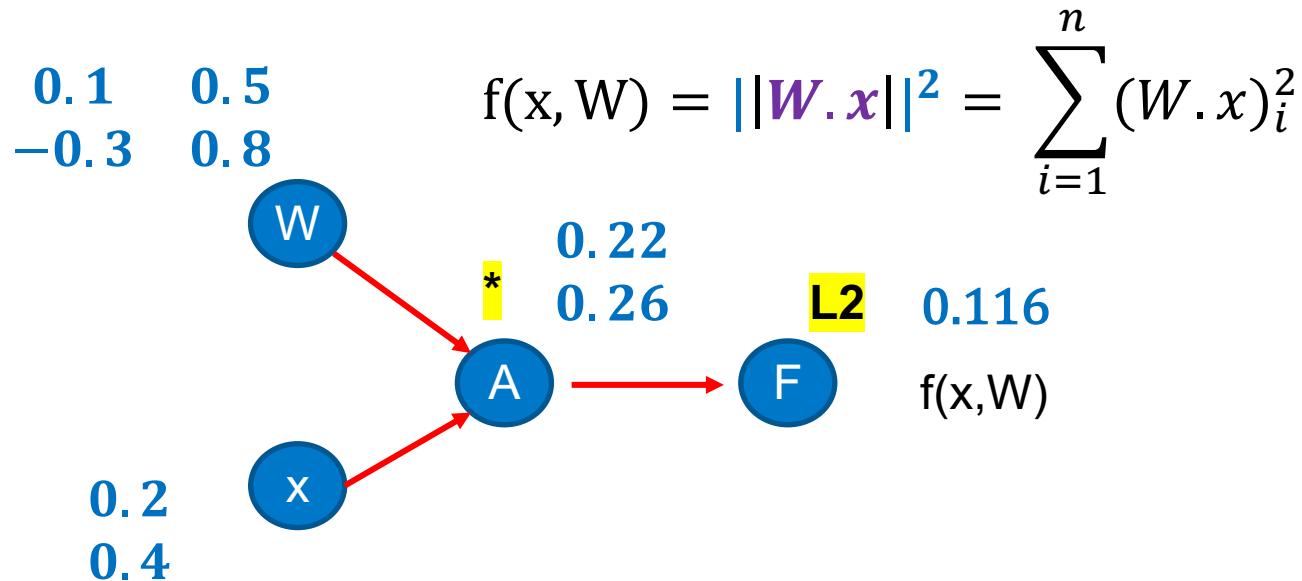
$$\begin{matrix} 0.1 & 0.5 \\ -0.3 & 0.8 \end{matrix} \quad \bullet \quad \begin{matrix} 0.2 \\ 0.4 \end{matrix} = \boxed{(0.1*0.2)+(0.5*0.4) = 0.22} \\ \boxed{(-0.3*0.2)+(0.8*0.4) = 0.26}$$

# Jacobian walkthrough



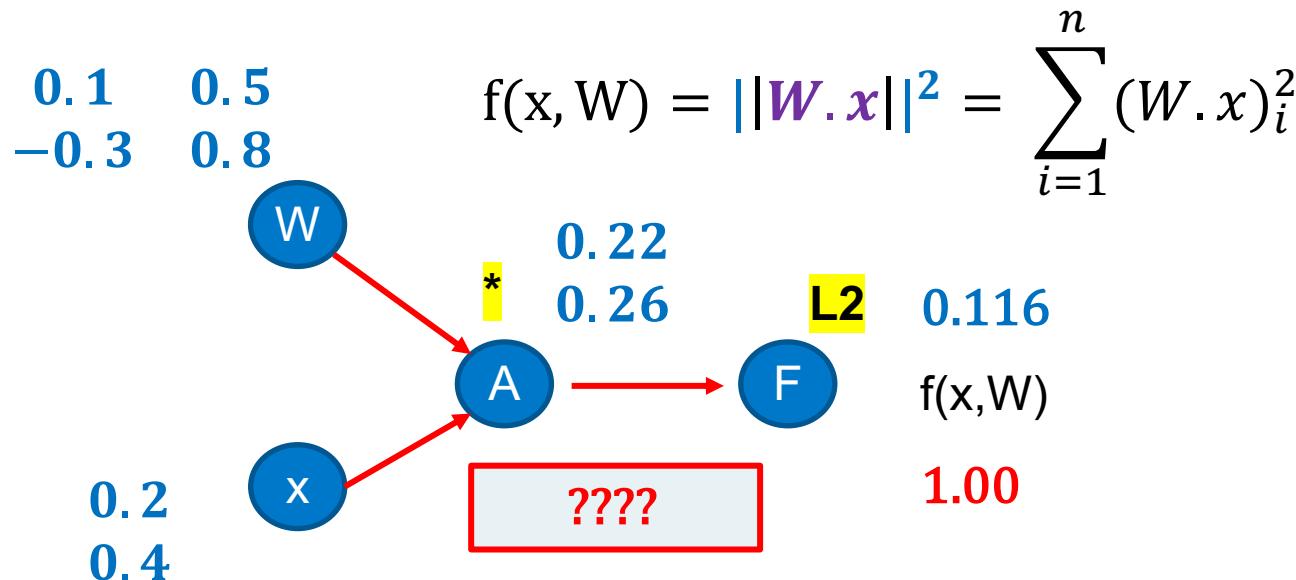
$$\| \begin{pmatrix} 0.22 \\ 0.26 \end{pmatrix} \|^2 = \text{????}$$

# Jacobian walkthrough



$$\left\| \begin{pmatrix} 0.22 \\ 0.26 \end{pmatrix} \right\|^2 = (0.22 * 0.22) + (0.26 * 0.26) = 0.116$$

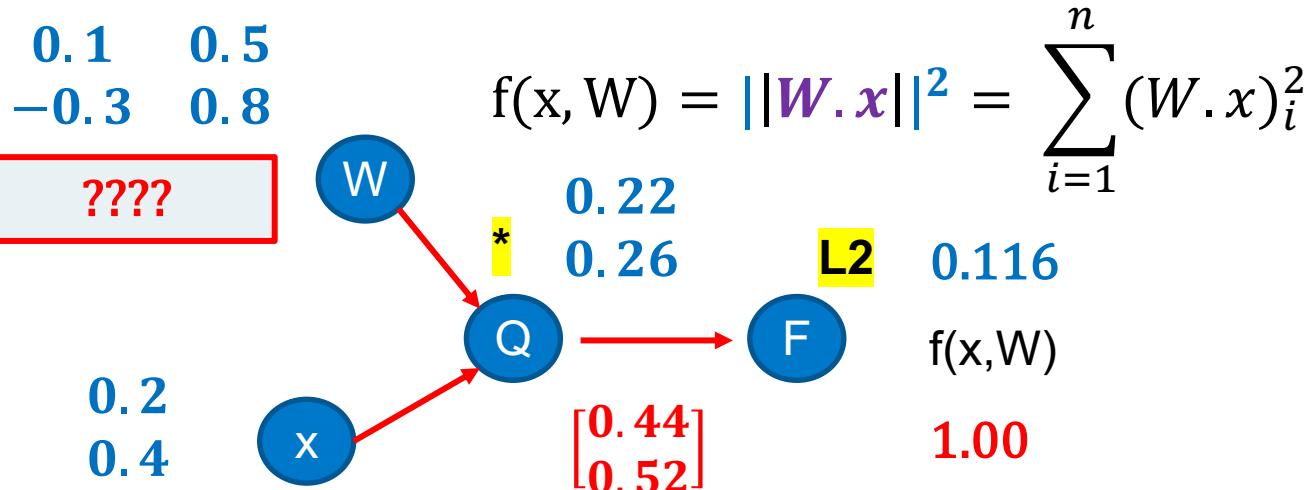
# Jacobian walkthrough (backwards)



- First (1) is derivative of the output with itself.
  - Roll back one layer, what's the function – how do we differentiate it?

$$\frac{df(x)}{dx} \text{ where } f(x) = x^{(2)} \gg f'(x) = 2x = \begin{bmatrix} 2 * 0.22 \\ 2 * 0.26 \end{bmatrix} (1.00) = \begin{bmatrix} 0.44 \\ 0.52 \end{bmatrix}$$

# Jacobian walkthrough (backwards)



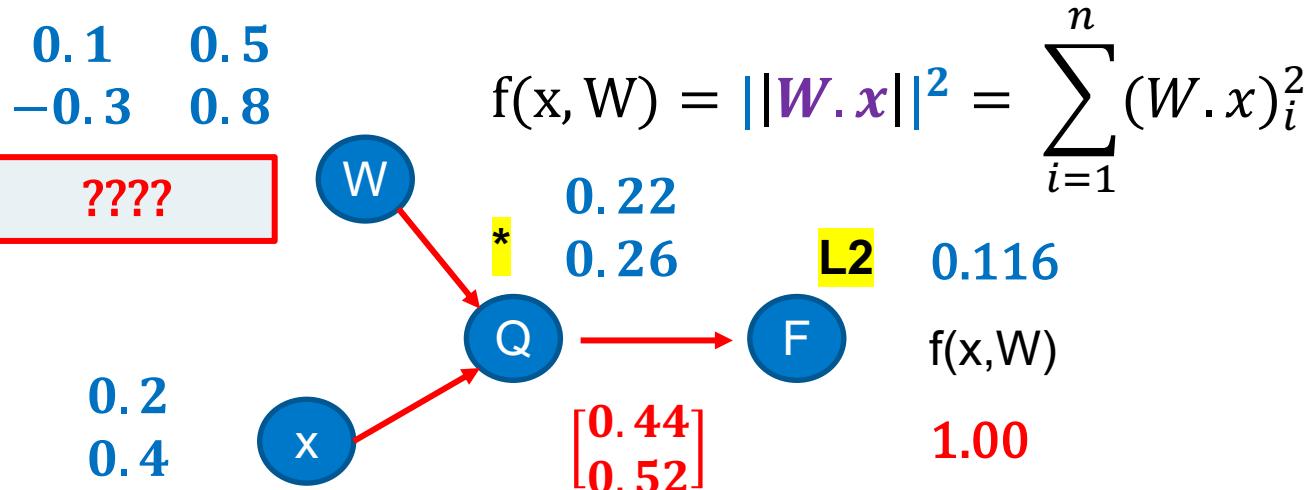
- Roll back another layer, how do we do this?
  - We need to create our **jacobian** to do this.

$$\frac{dq}{dw} \text{ where } \begin{bmatrix} q_{11} \\ q_{21} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} * \begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} w_{11}x_{11} + w_{12}x_{21} \\ w_{21}x_{11} + w_{22}x_{21} \end{bmatrix}$$

(We can also use a shortcut)

$$\frac{d(b^T x)}{dx} = b \text{ where our (b=X) and (X=W)}$$

# Jacobian walkthrough (backwards)



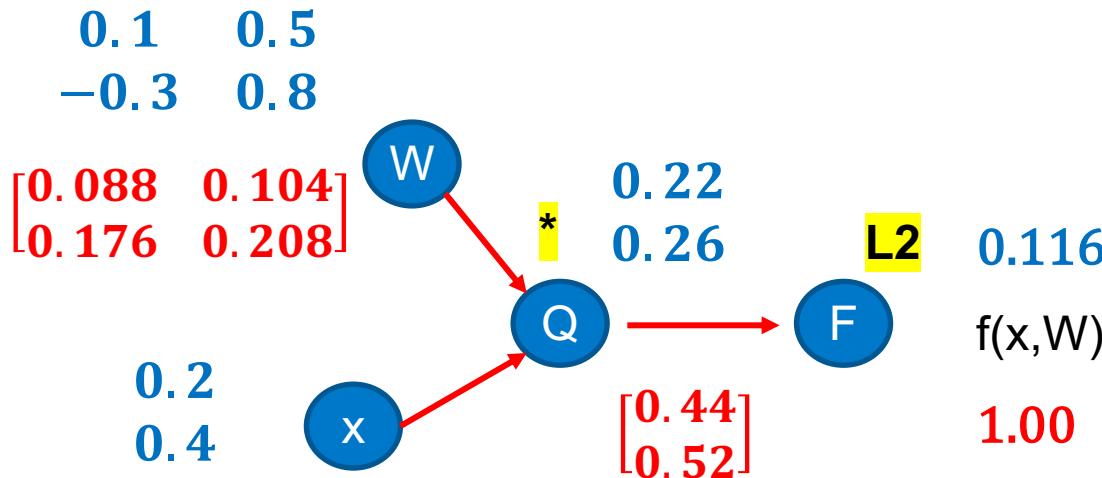
$$\frac{dq}{dw} \text{ where } \begin{bmatrix} q_{11} \\ q_{21} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} * \begin{bmatrix} x_{11} \\ x_{21} \end{bmatrix} = \begin{bmatrix} w_{11}x_{11} + w_{12}x_{21} \\ w_{21}x_{11} + w_{22}x_{21} \end{bmatrix}$$

$$\frac{d(b^T x)}{dx} = b \text{ where our (b=X) and (X=W)}$$

$$[0.2 \quad 0.4] * \begin{bmatrix} 0.44 \\ 0.52 \end{bmatrix} = \begin{bmatrix} 0.088 & 0.104 \\ 0.176 & 0.208 \end{bmatrix}$$

# Jacobian walkthrough (backwards)

$$f(x, W) = \|\mathbf{W} \cdot \mathbf{x}\|^2 = \sum_{i=1}^n (W \cdot x)_i^2$$



Key takeaway:

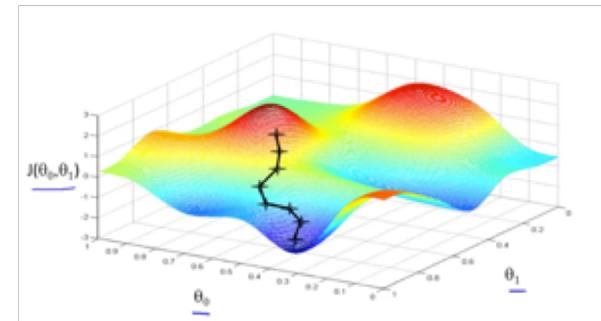
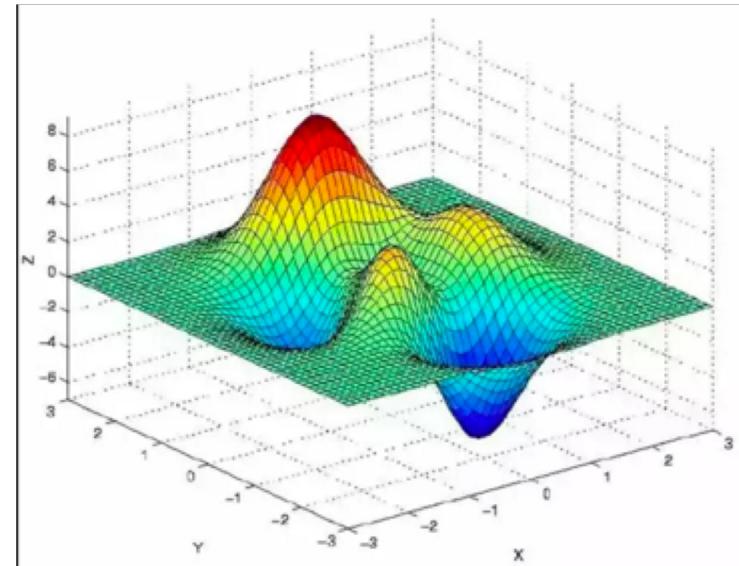
- The fundamental principle of the ‘Chain Rule’ can be applied to matrices in order to determine the influence of weight matrix elements on final output (loss)

# UPDATING THE WEIGHTS - GRADIENT DESCENT

- In multiple dimensions, the **gradient** is the vector of **partial derivatives** along each dimension.
- Think of it like a person standing somewhere in a valley of lots of ups and downs. Your goal is to find the lowest point (and efficiently!)
  - The height is the loss function, X-Y-Z are your parameters

(do we have only 3 parameters? How many do we have?)

1. Start somewhere
2. Take a bunch of little steps around us and see where is the steepest slope (negative derivative)
3. Then take a small step in that direction
4. Repeat



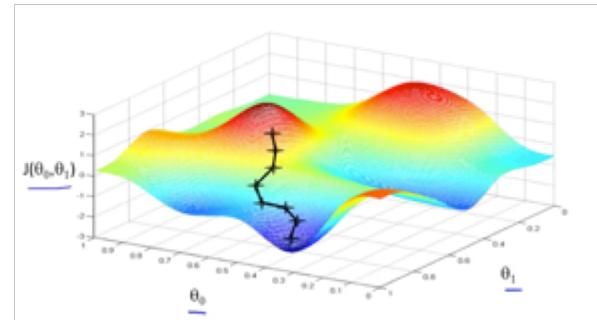
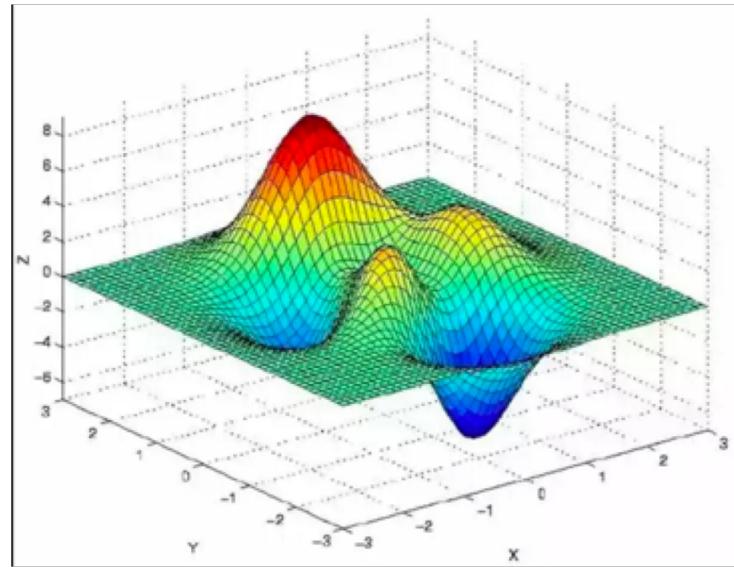
# UPDATING THE WEIGHTS - GRADIENT DESCENT

The small step is termed the learning rate:

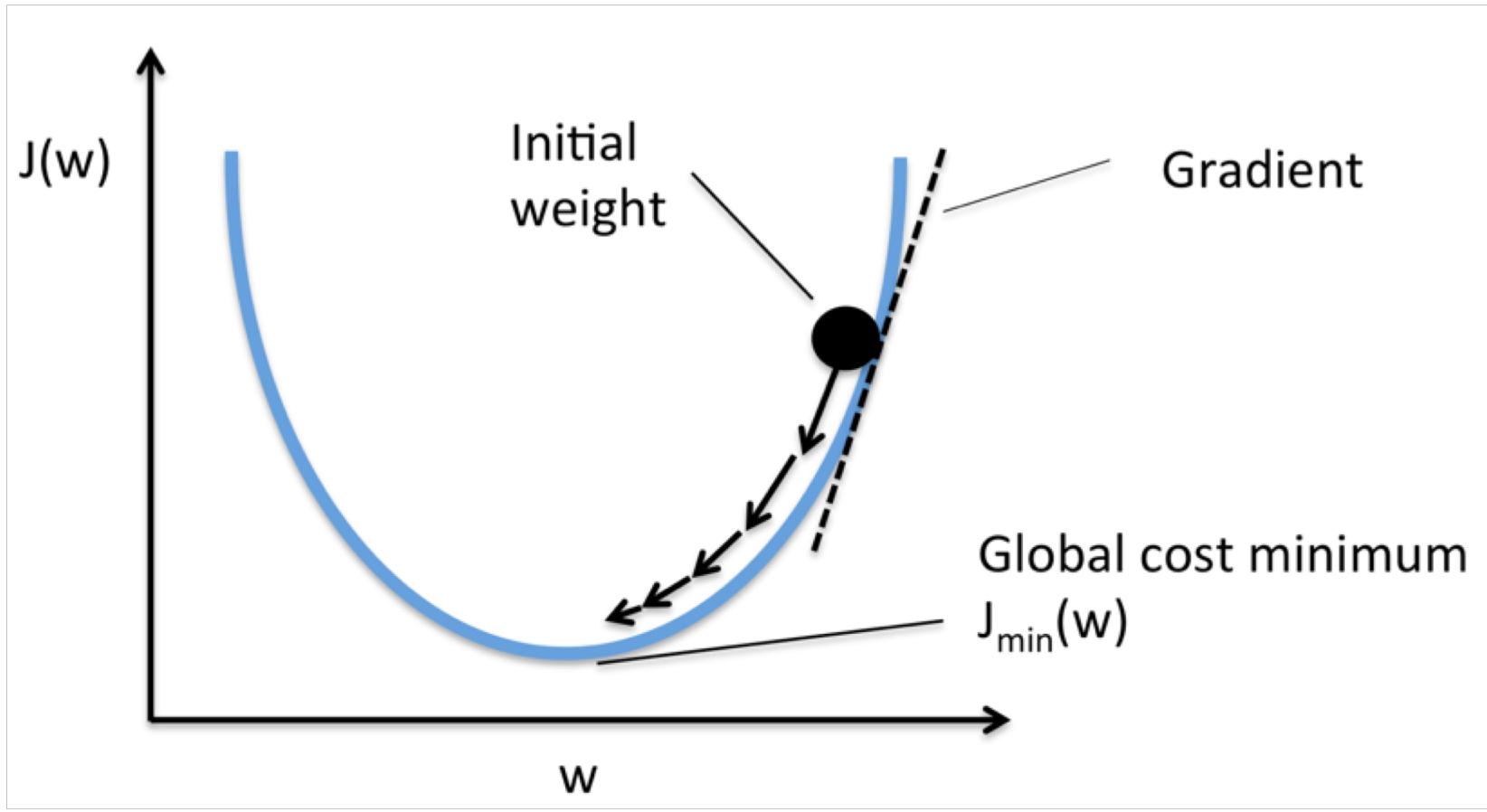
- If  $\alpha$  is too small, gradient descent will take a long time
- If  $\alpha$  is too large, you can fail to converge or overshoot the minimum (diverge)

[https://www.youtube.com/watch?  
v=WnqQrPNYz5Q](https://www.youtube.com/watch?v=WnqQrPNYz5Q)

[https://www.youtube.com/watch?v=  
IHZwWFHWa-w](https://www.youtube.com/watch?v=IHZwWFHWa-w)



# GRADIENT DESCENT



# AT1 BRIEFING + LAB

Walkthrough briefing PDF and notebook