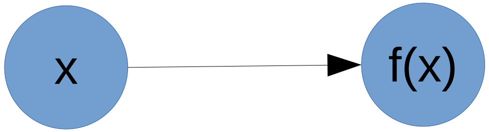
Games AI

Lecture 8.1

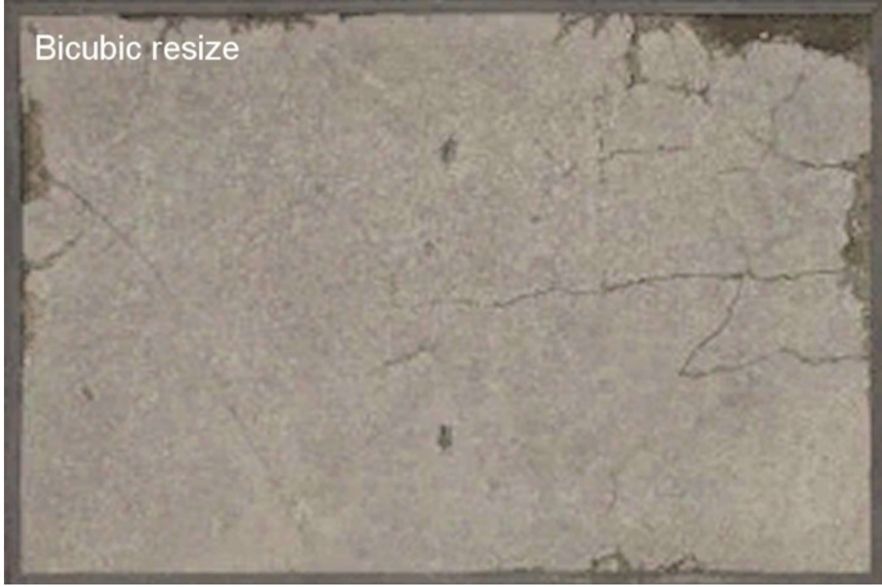
Neural Networks

- Neural Networks
  - Learn **functions** that map an input to an output



- Supreme Commander 2
  - **Input:** 17 ratios between statistics for friendly and enemy units in given radius
    - # units, unit health, damage/sec, …
  - **Outputs**: utility of possible actions
    - Attack weakest, attack closest, attack, generator, attack shield, …

- PCG
  - Input = low-res texture
  - Output = high-res texture



Images: https://www.leadwerks.com/community/blogs/entry/2487-upsampling-textures-with-ai/

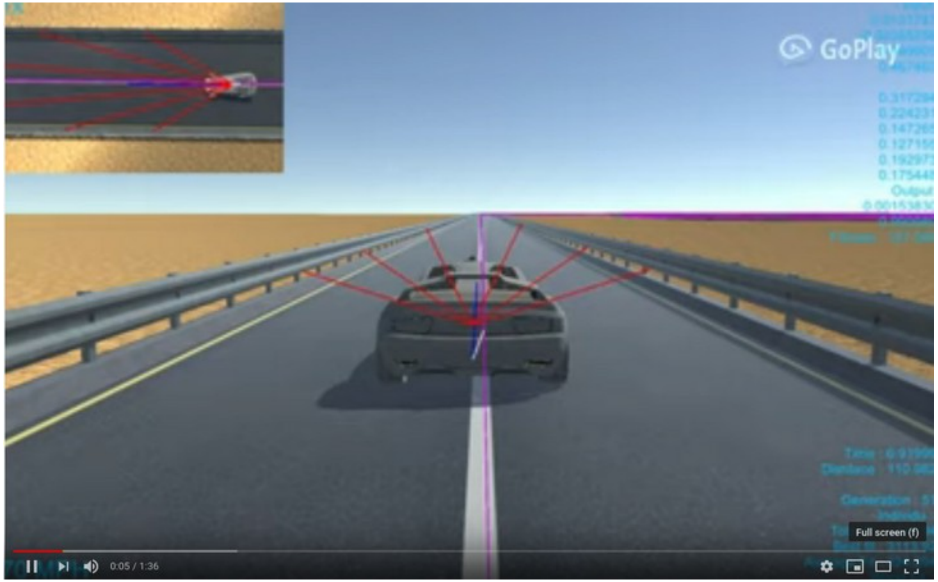
Overview

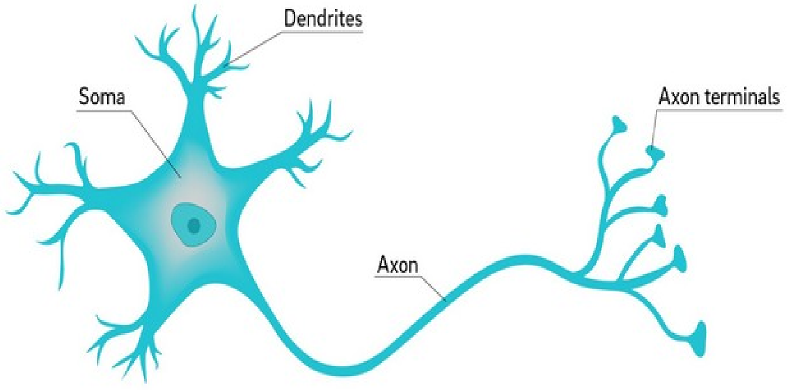
- Game remakes and up-scaling

http://forums.qhimm.com/index.php?topic=18553.msg261290#msg261290

Overview

- Neural-network controlled agents, e.g. using neuroevolution
  - https://youtu.be/_1TOKKgAock





- Neurons
  - Dendrites
  - Cell body
  - Axon
- Take inputs from dendrites
- Combine them to produce output
- Send output along axon

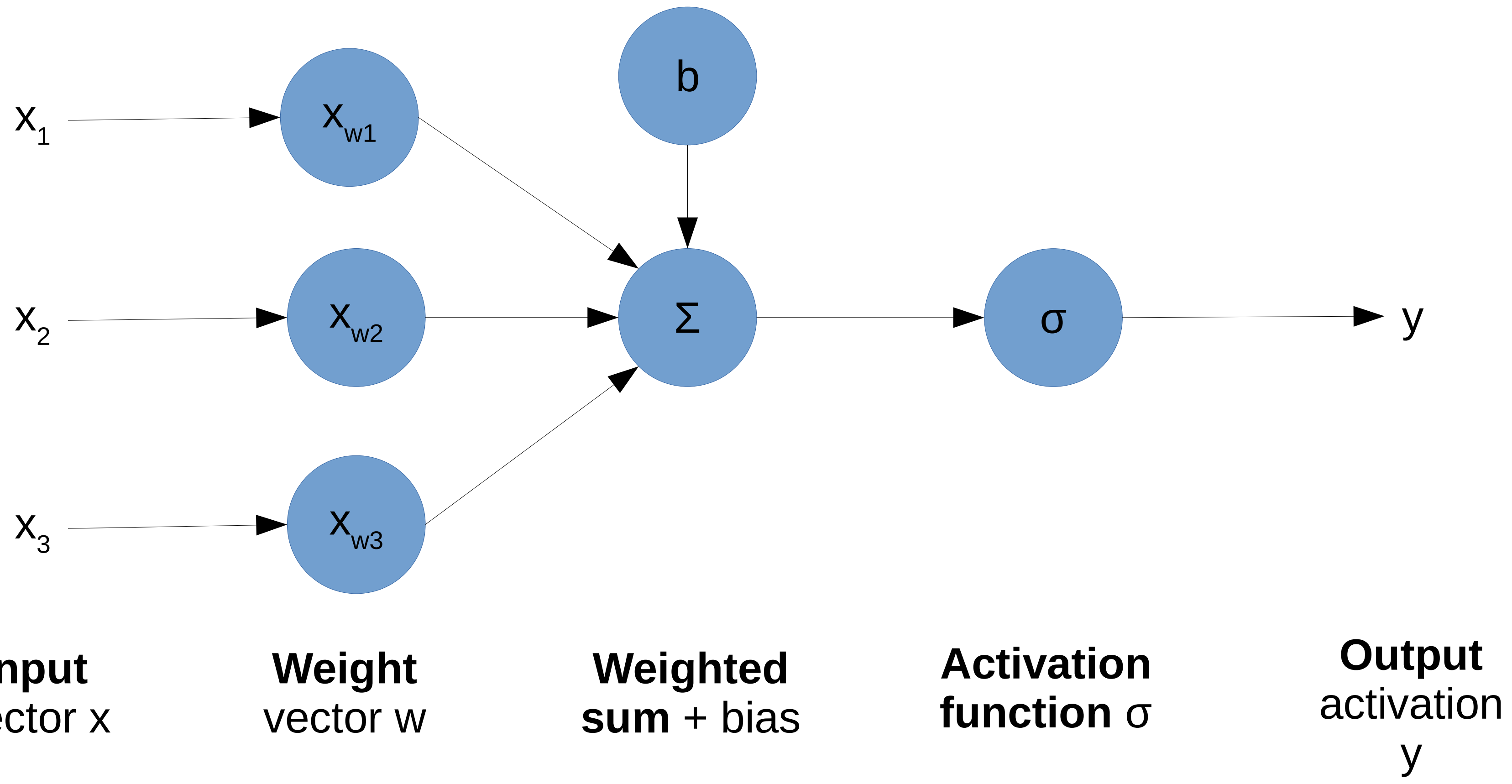
Overview

Jellyfish*	5,600
Brown rat	31,000,000
Orangutan	8,900,000,000
Human	16,340,000,000

*Whole nervous system

- Each neuron is connected to potentially thousands of others.
  - 10^{10} neurons in the human brain
  - 10^{13} neuron connections
- Each neuron is slow (millisecond+)
- Massive parallel processing

Overview



x_1
x_2
x_3
x_{w1}
x_{w2}
x_{w3}
b
Σ
σ
y
Input vector x
Weight vector w
Weighted sum + bias
Activation function σ
Output activation y

b
2.2

x
0.1
0.5
0.2
0.7

w
5
2
-2
1

4

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$

0.98

Input vector x

Weight vector w

(∑= 1.8)

Weighted sum + bias

Activation function σ

Output activation y

1) Calculate the **weighted sum of inputs** ($x_{1..n}$) with weights ($w_{1...n}$)

- $\sum_j w_j x_j$

2) Add the **bias** b

- $\sum_j w_j x_j + b$

3) Pass it into our **activation function** σ

$$\sigma(z) \equiv \frac{1}{1+e^{-z}}$$

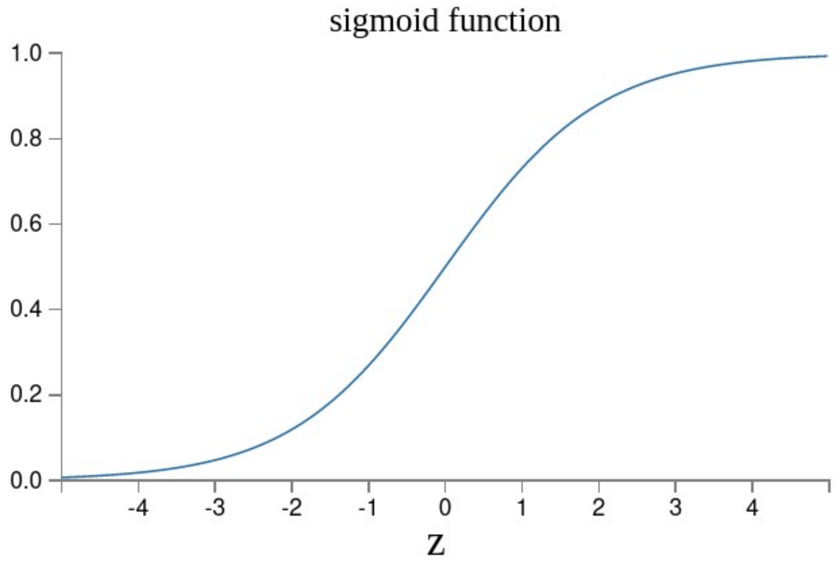
- $\sigma(\sum_j w_j x_j + b)$

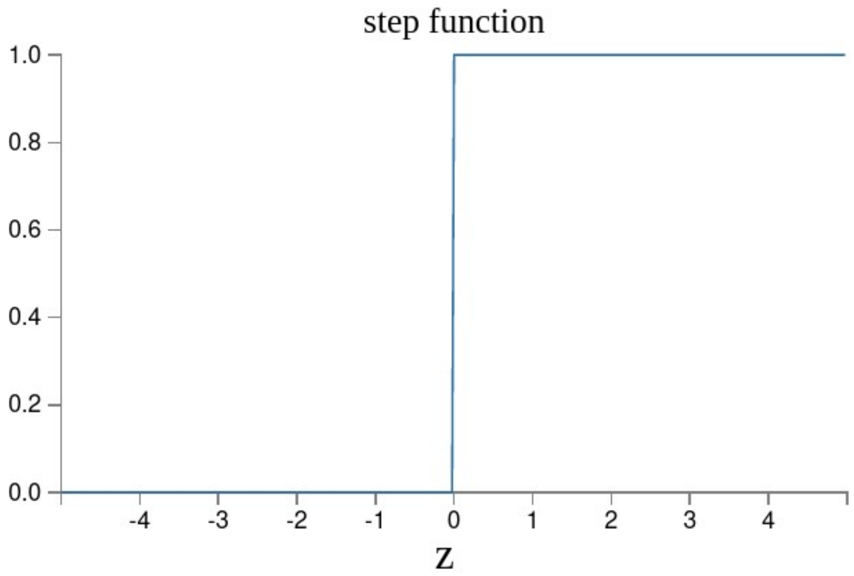
4) And put it all together, and the **activation of a neuron** is:

- $$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}$$

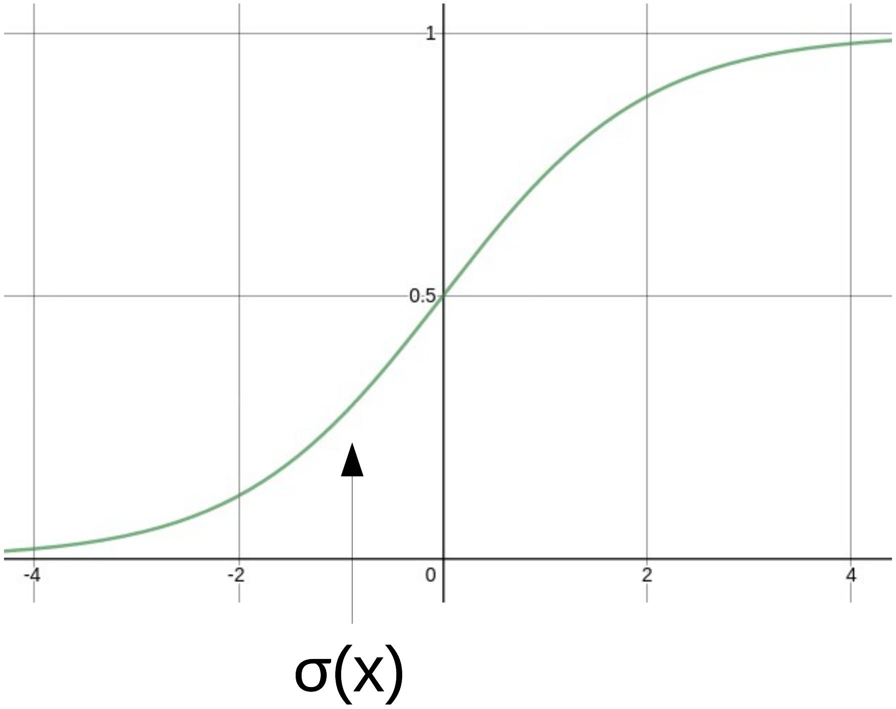
We can also write $\sum_j w_j x_j$ as a **dot product:** w·x

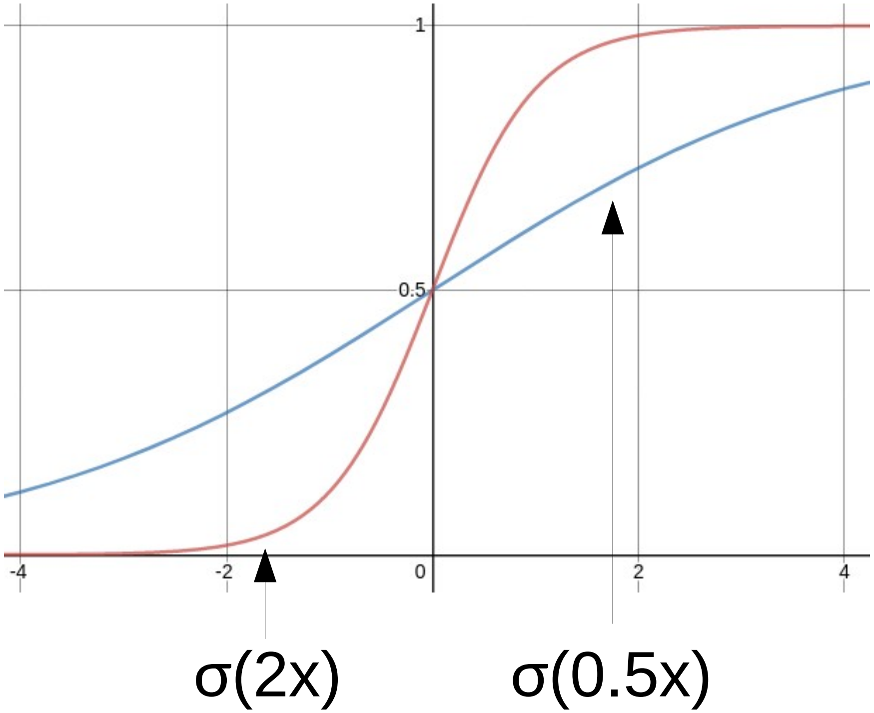
- σ is a **non-linear** function e.g. **logisitic sigmoid** or **step function**
  - Most of the time you will use sigmoid functions



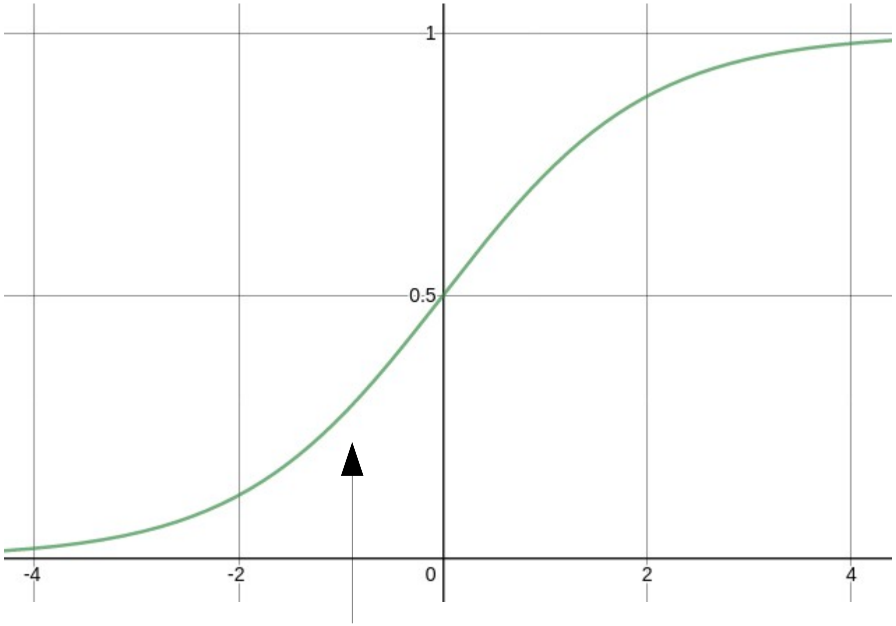


- **Weights** effectively change the steepness of the sigmoid

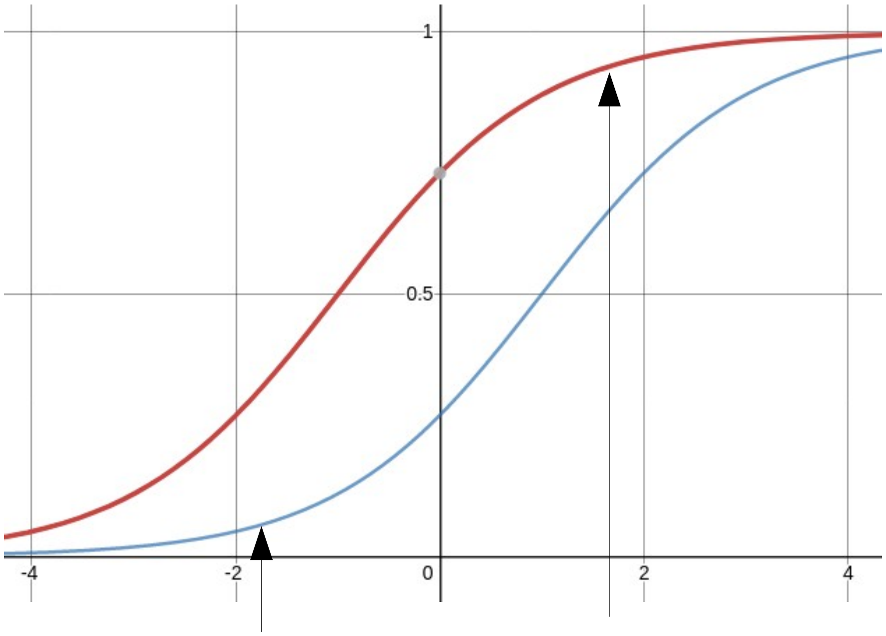




- **Bias** is a constant that effectively shifts the activation function left/right

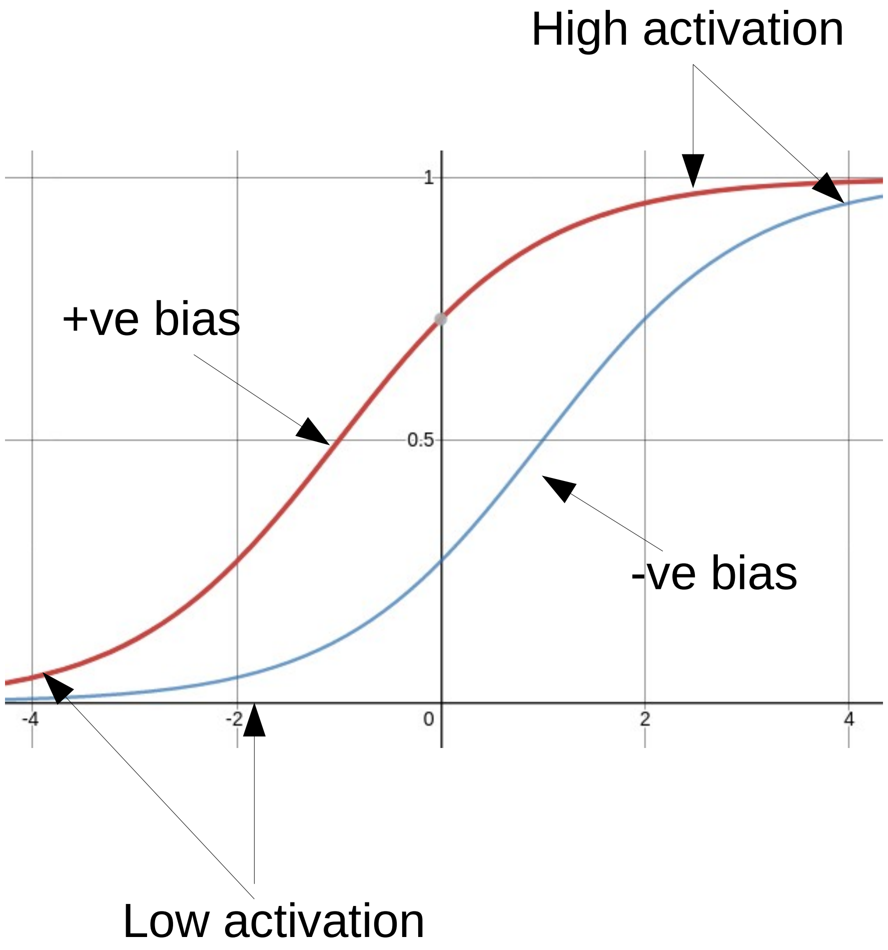


σ(x)

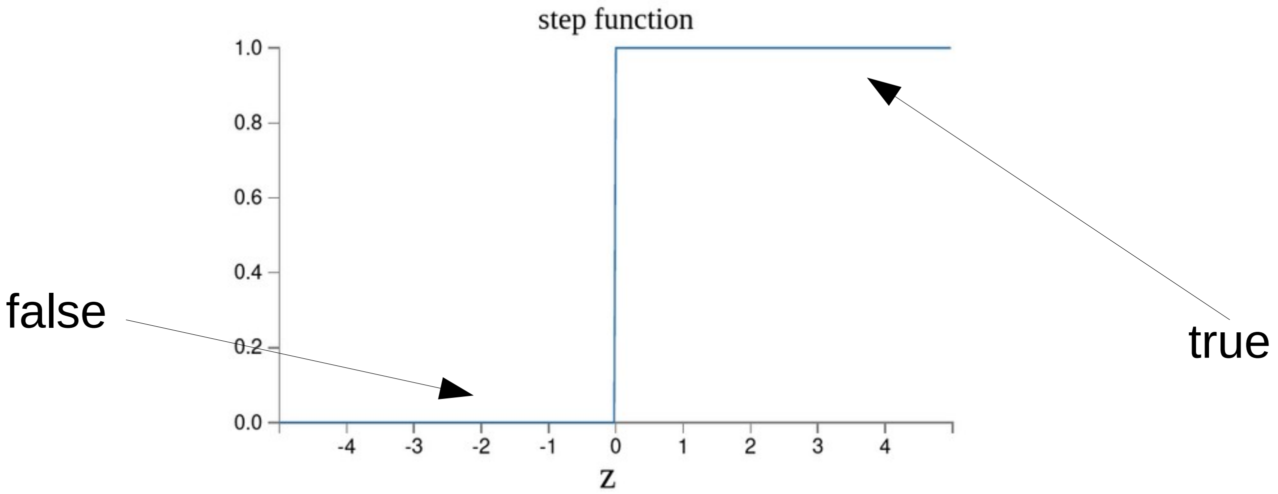


σ(x-1) σ(x+1)

- You can think of bias as **how easy it is to activate the neuron**
  - A **big positive bias** means that the “step” between 0 and 1 will happen at really **small** input values
  - A **big negative bias** means that the “step” will happen at really **big** input values

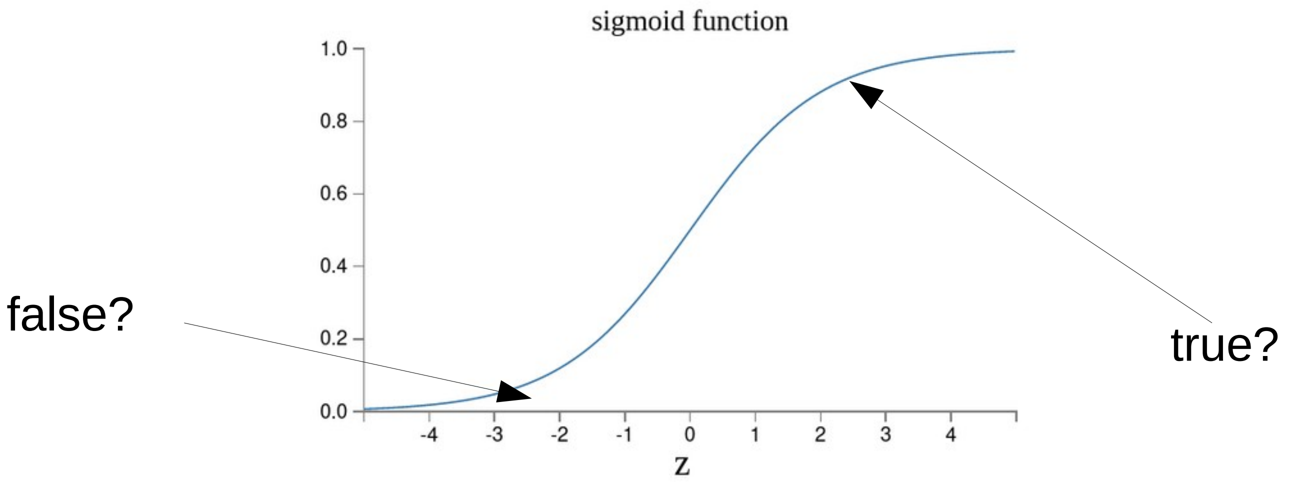


- Each neuron is **making a decision** just like a logic gate
  - And logic gates can be used to build computers
  - If we use the step function…

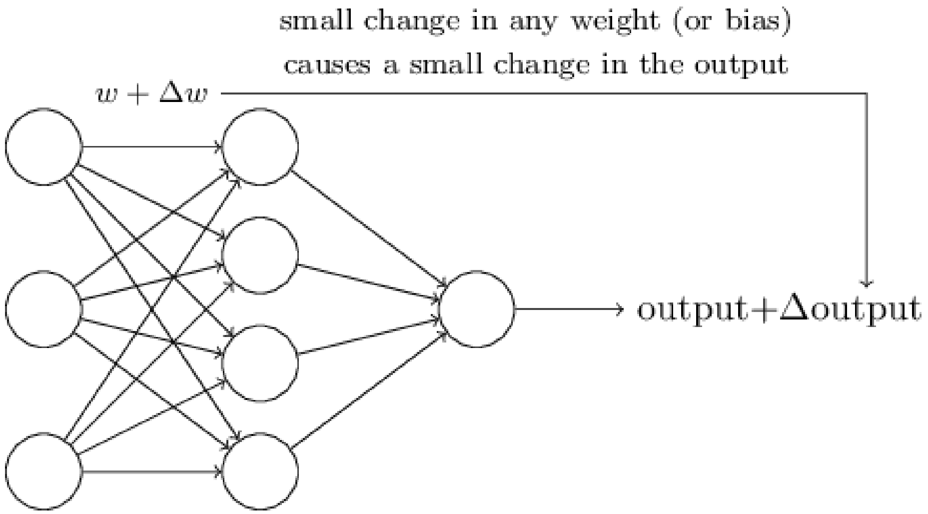


2) Because the sigmoid function has a smooth step, it can be **more or less** true or false

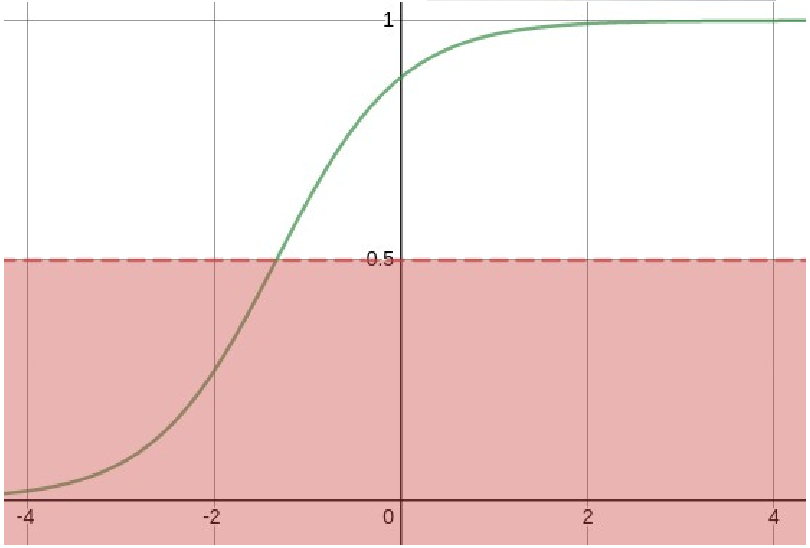
- Which can also be more or less **confident** of being true/false
- Which **allows it to be trained**, so the computer redesigns itself!



- To **train the network**, a small change in the weights must correspond to a small change in the output
  - Because of the sigmoid function, this is the case
  - Why is this not the case for the step function?

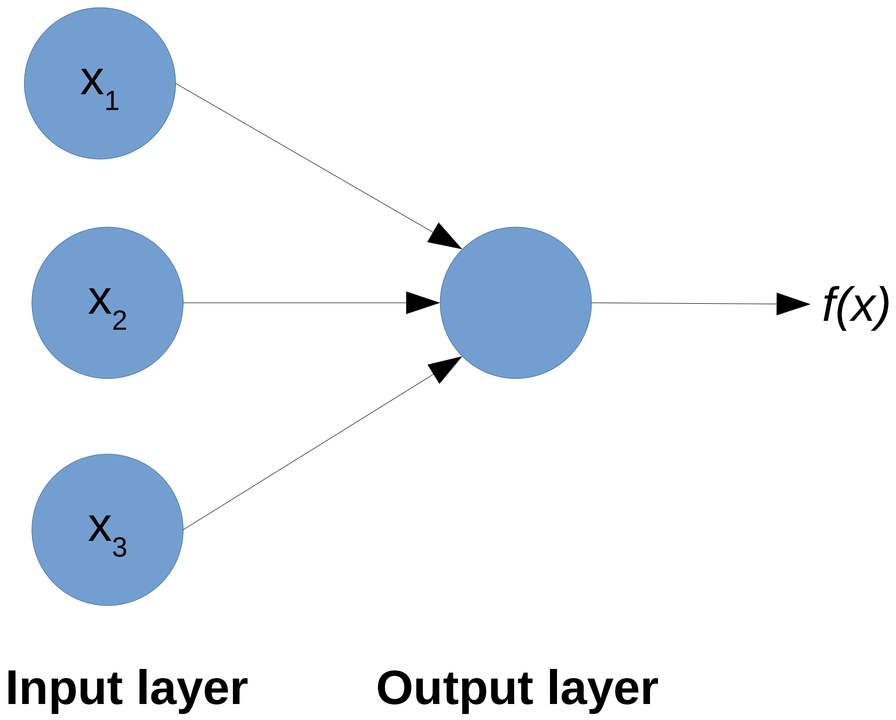


- To **use the network once it is trained**, e.g. to classify inputs into categories, we usually **round** the output to 0 or 1.



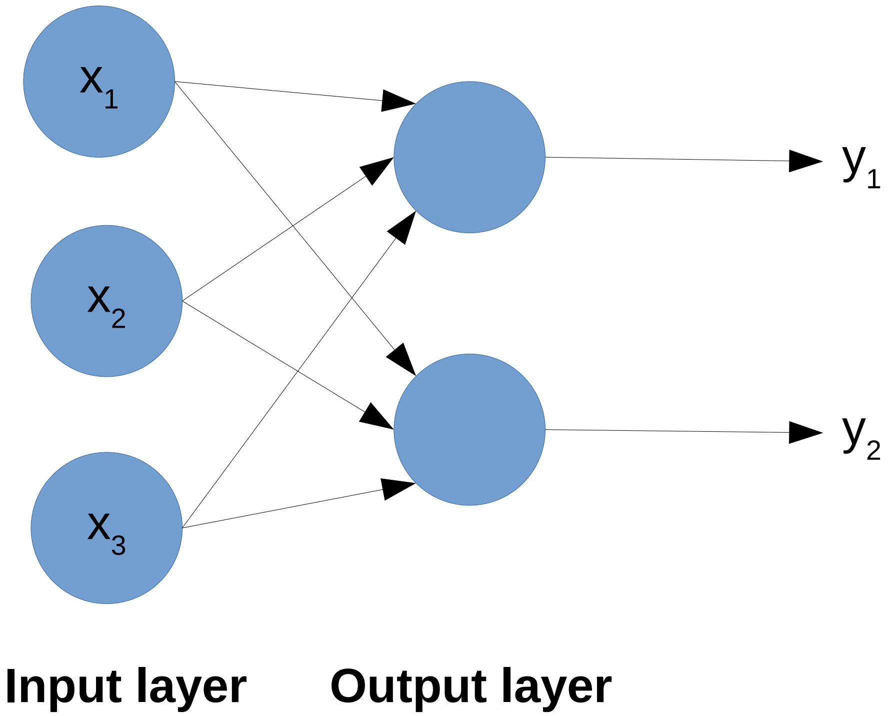
Architecture

- The **Single Layer Perceptron** is the simplest ANN

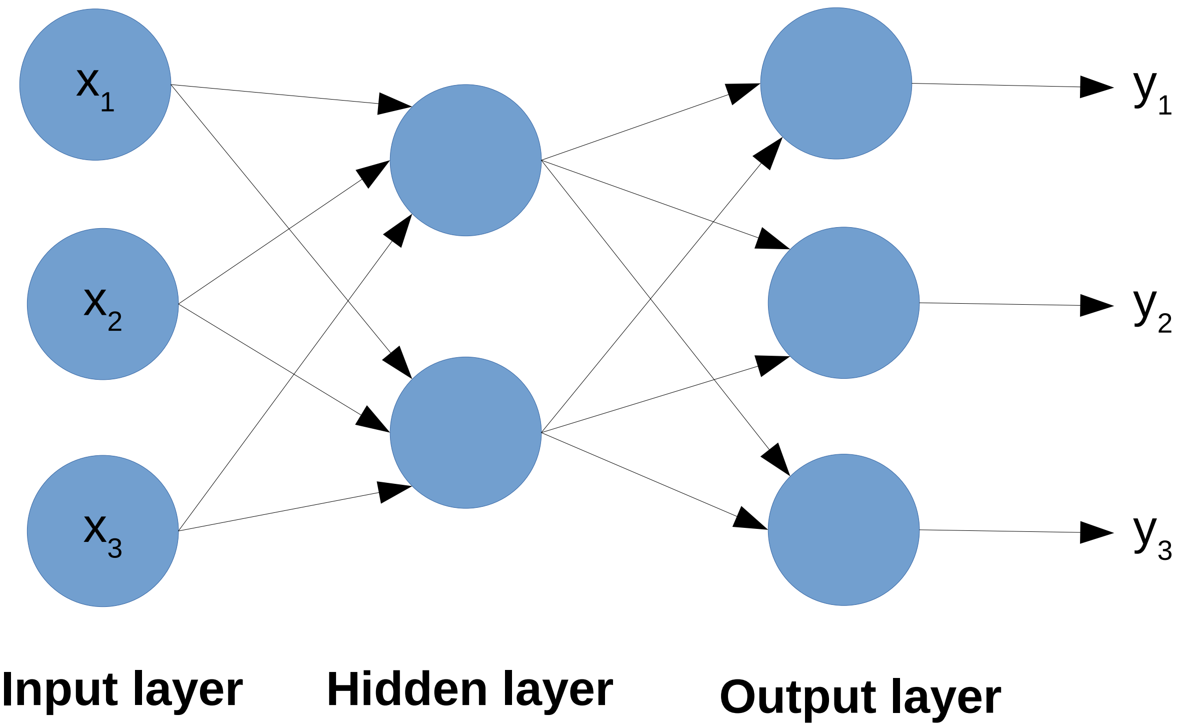


- **Input nodes** are special.
  - They have no inputs from other nodes
  - Their output is whatever value they are inputing into the network is.

- **Nodes only have a single output**, but that output can be used as the input to multiple nodes, hence the multiple arrows in diagrams

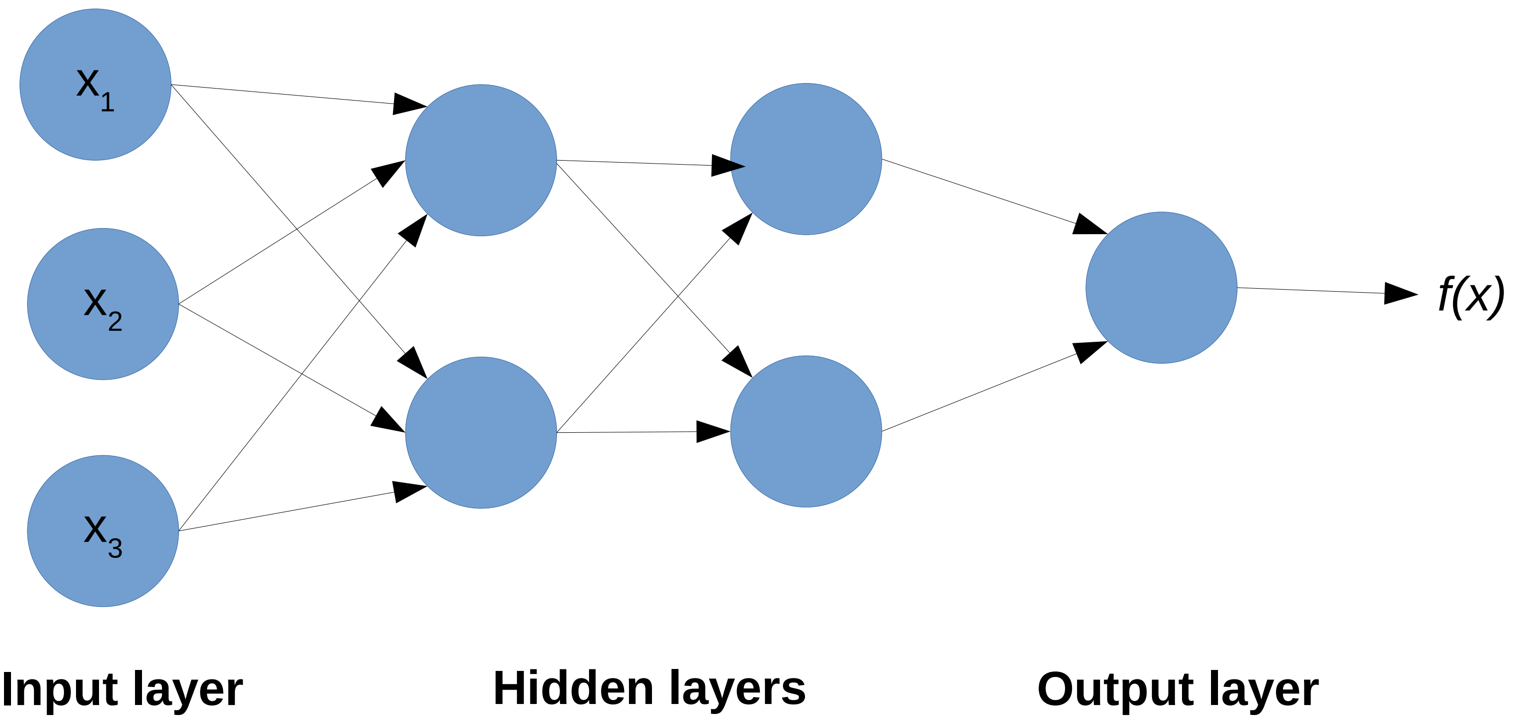


- **Multi-layer Perceptrons** have additional **hidden** layers
  - (a MLP is a fully-connected network)



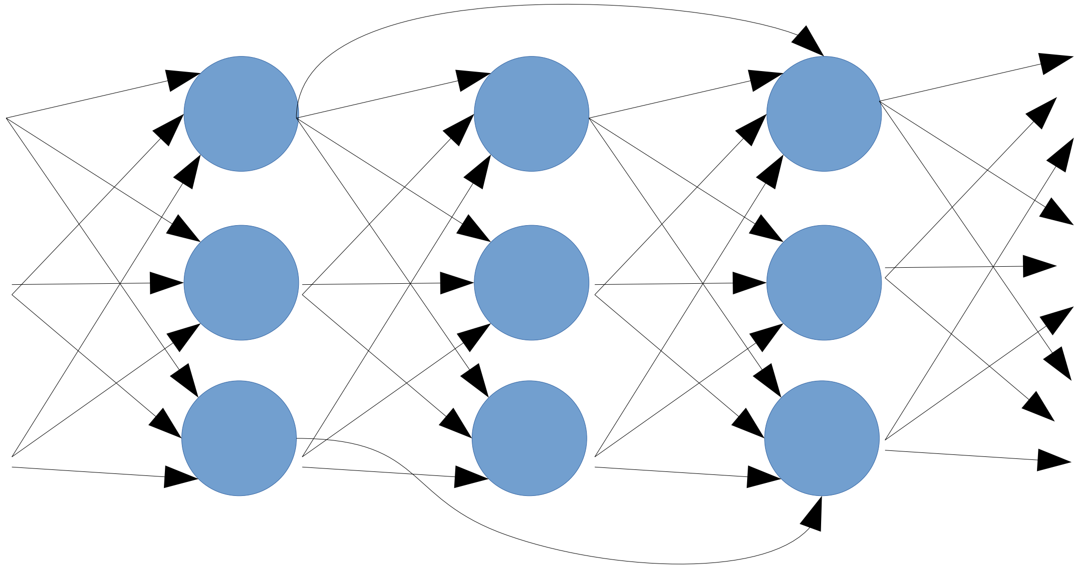
- The **more neurons** you have in a **layer**, the more different properties that layer can process
  - i.e. the more **different ways** that layer is processing the inputs
  - or the more **derived properties** that layer is calculating

- You can have any number of such hidden layers (if you have enough computation power)

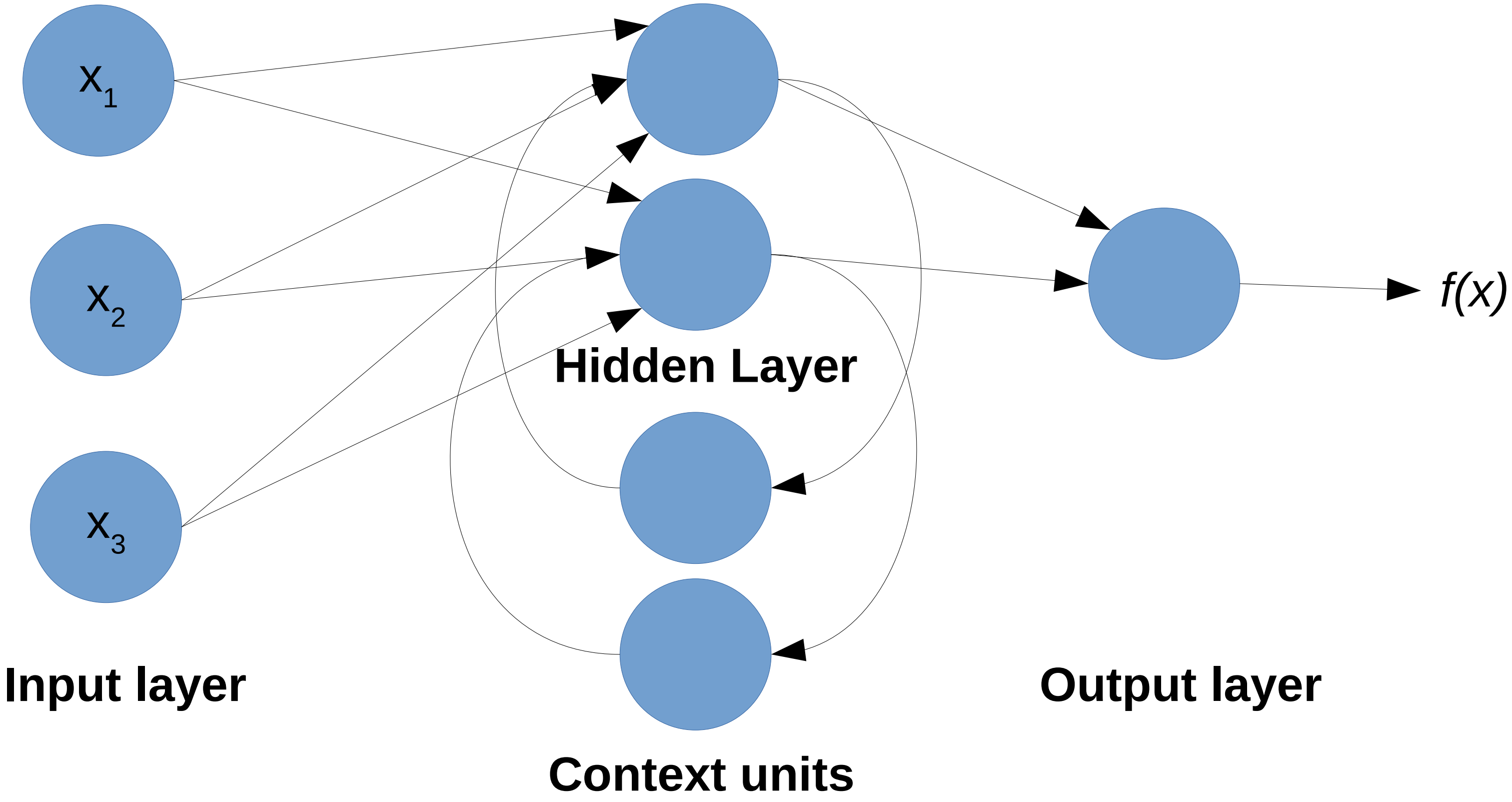


- Each **hidden layer** allows the network to work with higher levels of abstraction
  - The **first** hidden layer derives properties from the input values
  - Then **the second** hidden layer derives properties from these
  - Then **the third** hidden layer…

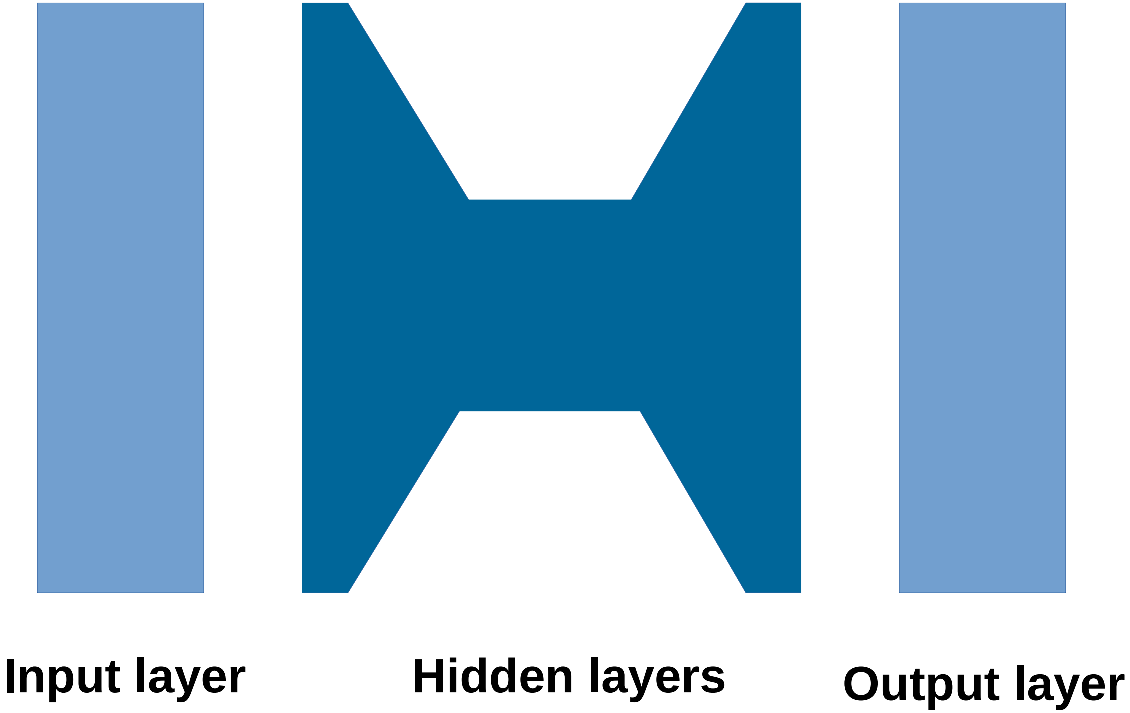
- When you have **deep networks**, you often want connections that skip layers, so properties available at one layer can influence later layers
  - i.e. the data flows better
  - Another impact of this is that networks can be more resiliant when parts of the network are changed or disabled.



- **Recurrant Neural Networks** have loops so can store state
  - Good for processing temporal streams of data e.g. speech



- **Autoencoders** are used to learn ways to compress data by having small hidden layer



- **Autoencoders** can be used to de-noise images too!

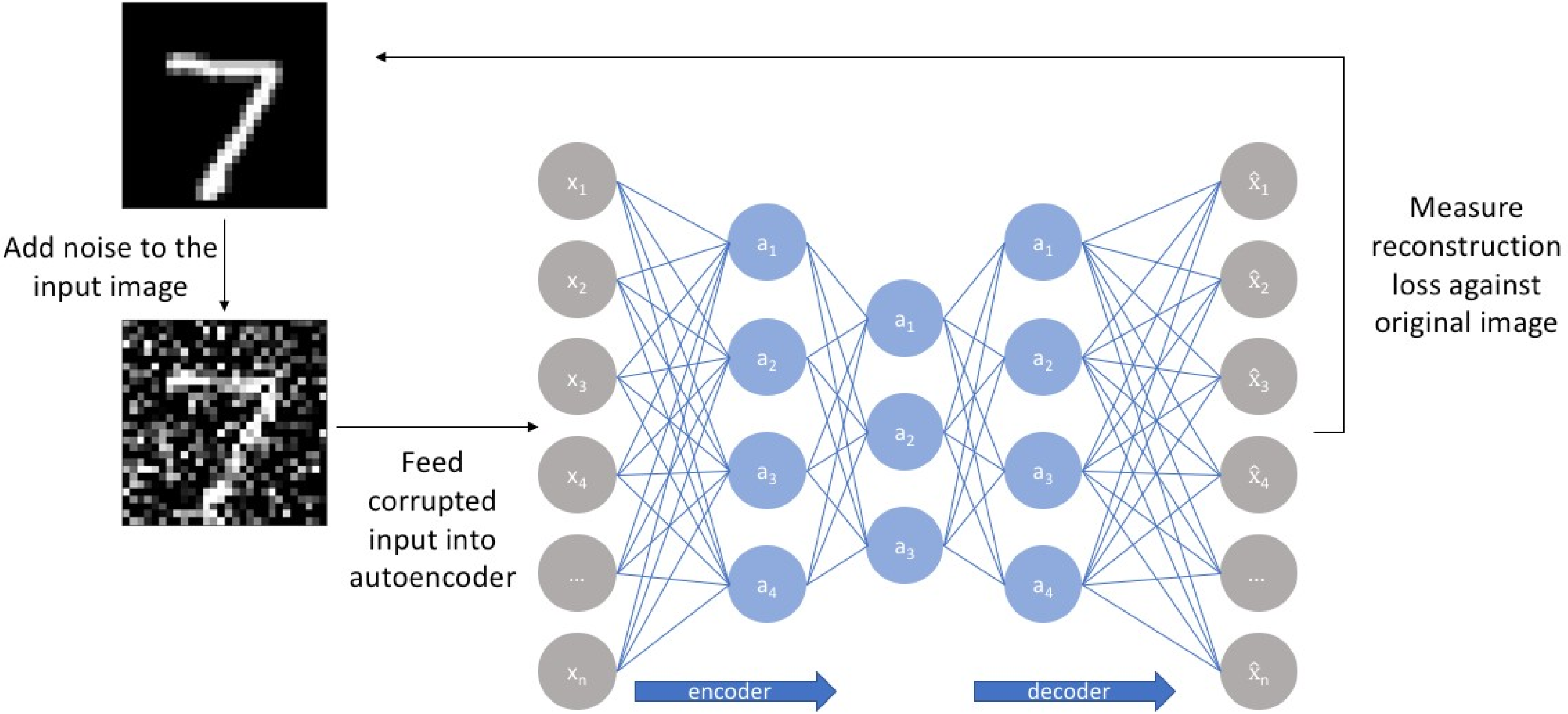
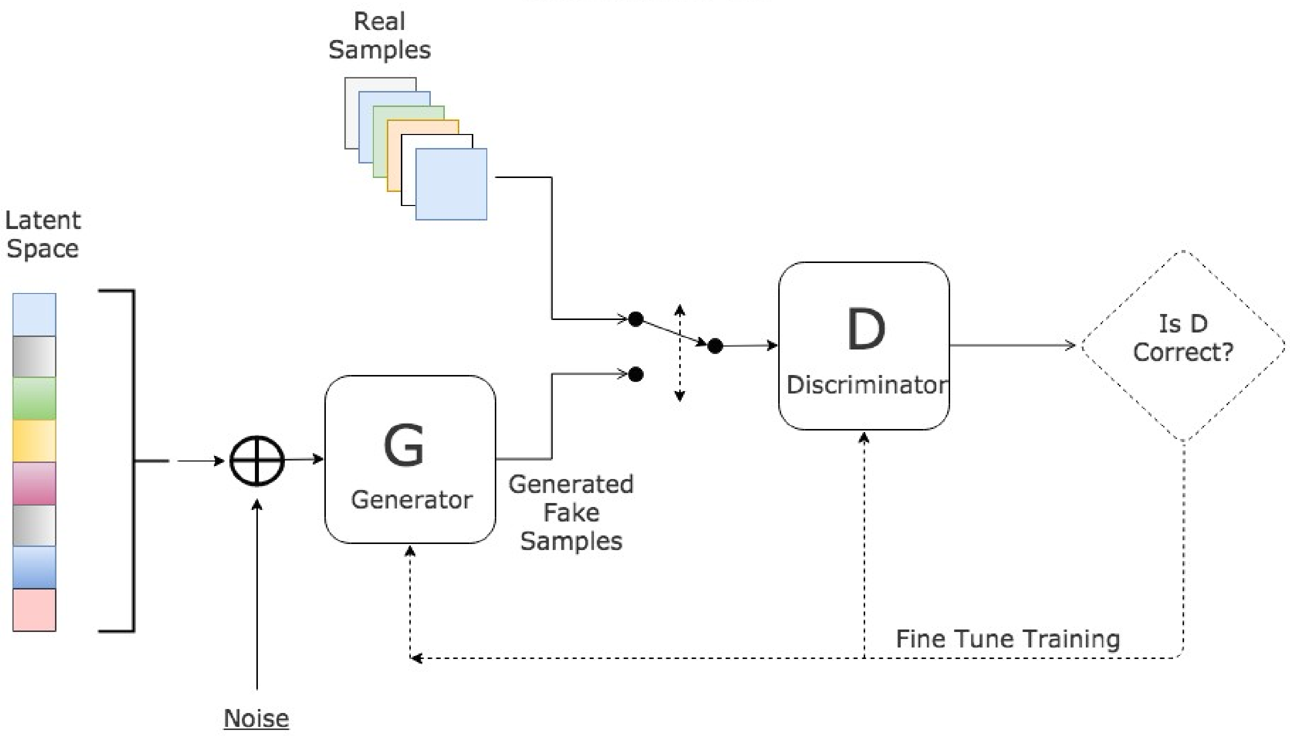


Image: https://www.jeremyjordan.me/autoencoders/

- **Generative Adversarial Networks** are pairs of neural networks that generate content that is very similar to a set of samples




Training

- ANNs are usually* a form of **supervised learning**
  - Your training data is pre-labelled with the correct output
  - e.g.

→ “cat”

*Other than autoencoders, which learn an encoding in an unsupervised manner

- Lets say we’re **categorising animal images** as either: cat, dog, or t-rex
  - We would have 3 output nodes, that give us a **3-dimensional vector**
  - We would define each category as a unit vectors

= [1,0,0]

= [0,1,0]

*

= [0,0,1]

*that’s T-Rex, the band

- At first, it’ll produce results that are pretty much random, e.g.

$\rightarrow$ [0.4, 0.9, 0.6]

- But if this is in our training dataset, we can compare this output to the known value, in this case [0,0,1] (t-rex)
- This lets us work out the **training error**

- The **training error** can be calculated **per output neuron** i for input *x*
  - *(e_i = error for neuron i, y(x) = target value, a = actual value)*
  - $e_i = y(x)_i - a_i$
- You usually actually use the **Quadratic Error**
  - $(y(x)_i - a_i)^2$
- And if we do this for the **all outputs as a vector,** y(x) and a are vectors (e.g. [0, 0, 1] and [0.4, 0.9, 0.6] it becomes
  - $\| y(x) - a \|^2$
    - *(Where* $\| v \|$ is the length function of the vector, which coverts it into a scalar)

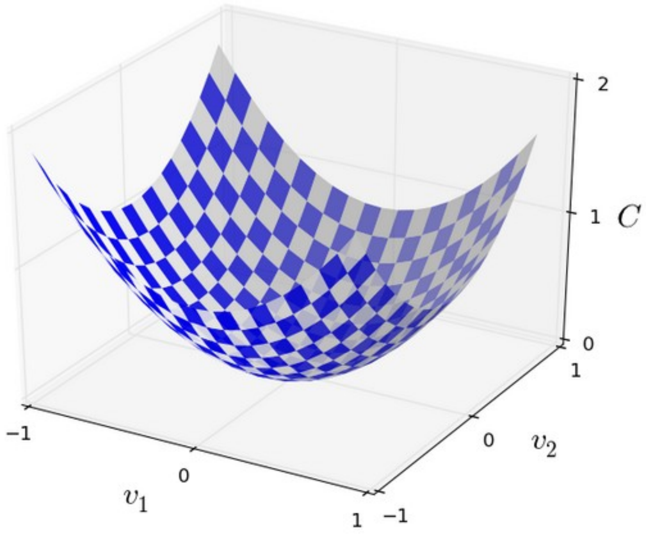
- We could calculate the **mean squared error (MSE)** over our entire test dataset.
  - We could describe a function that gives us this value.
    - Its called a **cost function** (or **error function**)

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

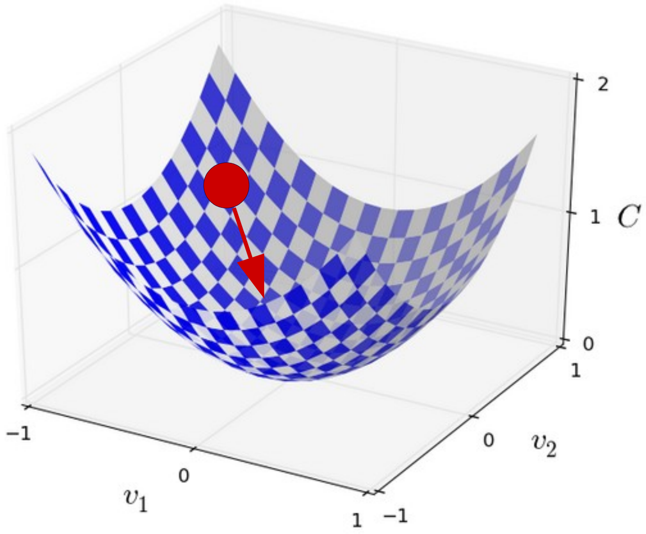
  - This is a function over *w* and *b* which are vectors of the weights and biases in the network
    - Basically for given weights and biases, what is the MSE for our test data?
- Ideally we want C(w,b) ≈ 0 (i.e. basically no error)

(The ½ is just a constant to make things easier in the maths later)

- Cost function / error function
  - Idea behind training: Want to **minimise the cost function**
  - As *w* and *b* might be **very large** vectors, analytic approaches too difficult – (calculus with 1,000,000s of variables!)



- **Imagine** we only had **one weight** and **one bias**
  - C(w,b) would be 2 dimensional and our cost function would by a surface in 3d space
    - (Very much like an **adaptive landscape** in evolutionary algorithms)
  - We want to find the **lowest point** (global minima) on that 2d surface



- **Gradient Descent:**
  1) Start from a random point
     - (i.e. randomly-assigned weights/biases)
  2) Calculate the **gradient** of your cost function at that point
  3) Move downhill until you get to a **local minima**


Backpropagation

- **Backpropagation*** is a way to optimise the weights of a (feed-forward) neural network
- How to actually perform **gradient descent** in an efficient manner
  - Works backwards from the last layer in the network towards the first

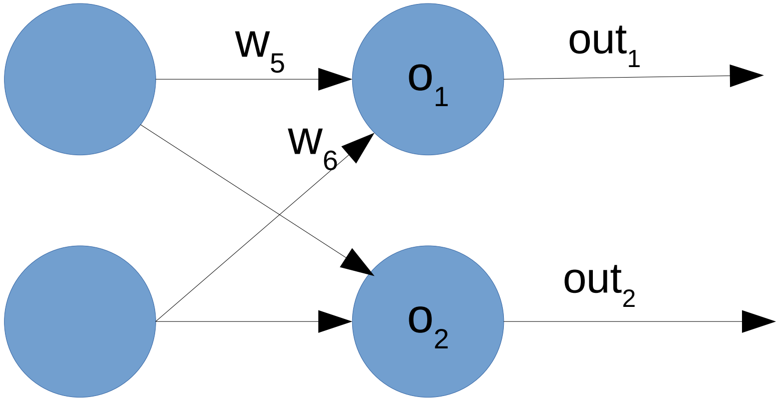
*Comes from: “Backwards propagation of errors”

- We can work out our **total error** for our network **(for a single input)** as we saw before
  - Here we find the squared error of each node and sum over all output nodes*

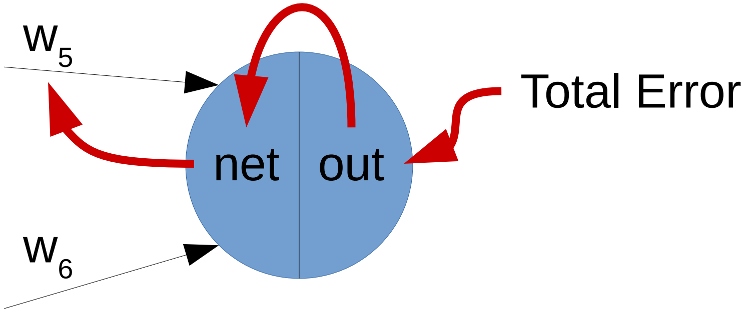
$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

*Again, ½ is just a constant for the maths

- Backpropagation works by calculating **how much each weight contributes** to the **total error**
  - And then **reducing** that weight **proportionally to its contribution**
  - We do this in a sequence of steps using the **chain rule**



- Starting with a weight on an output node:
    1) We work out the how sensitive the **total error** is to **changes in the output of o_1**
    2) Then how sensitive the **output of o_1** is to changes in the **total net input of o_1**
    3) Then how sensitive the **total net input of o_1** is to **changes in w_5**
- With the **chain rule**, this will tells us how much w_5 impacts total error



- We can formally express the chain rule here as the following:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

  - The **sensitivity** of the **total error** to **changes in w_5** is equal to the three other sensitivities we mentioned multiplied together
  - We calculate what these are using **partial derivatives** from **calculus**
    - (Which I'm not going to go through here)

- Once we have found $\frac{\partial E_{total}}{\partial w_5}$ we multiply it my our **learning rate,** η (eta) (a small positive number)
  - We're going to then change our weight by **minus this amount** (because we want to **reduce** our error)
  - The size of our step is proportional to the **sensitivity of the error** to the **weight w_5**
    - So we adjust the important weights faster than the unimportant ones

- We then repeat the process for all the weights on the output layer, storing the new values
- We **then** repeat more or less the same the process for the previous layer
- Finally, we apply the updated weights

- After enough iterations with our training data, the error decreases to a local minima.
  - And our network is ready to use

- There's obviously a lot more to neural networks and how to train them
  - For a well-written introduction, chapter 1 of:
    - http://neuralnetworksanddeeplearning.com
  - An excelent tool for understanding what is going on inside a neural network:
    - https://playground.tensorflow.org/
  - Another helpful tool for understanding, if you can get it to work properly:
    - http://neurovis.mitchcrowe.com/
  - A helpful video on understanding backpropagation
    - https://www.youtube.com/watch?v=6BMwisTZFr4
  - And a worked example of backpropagation

    https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/