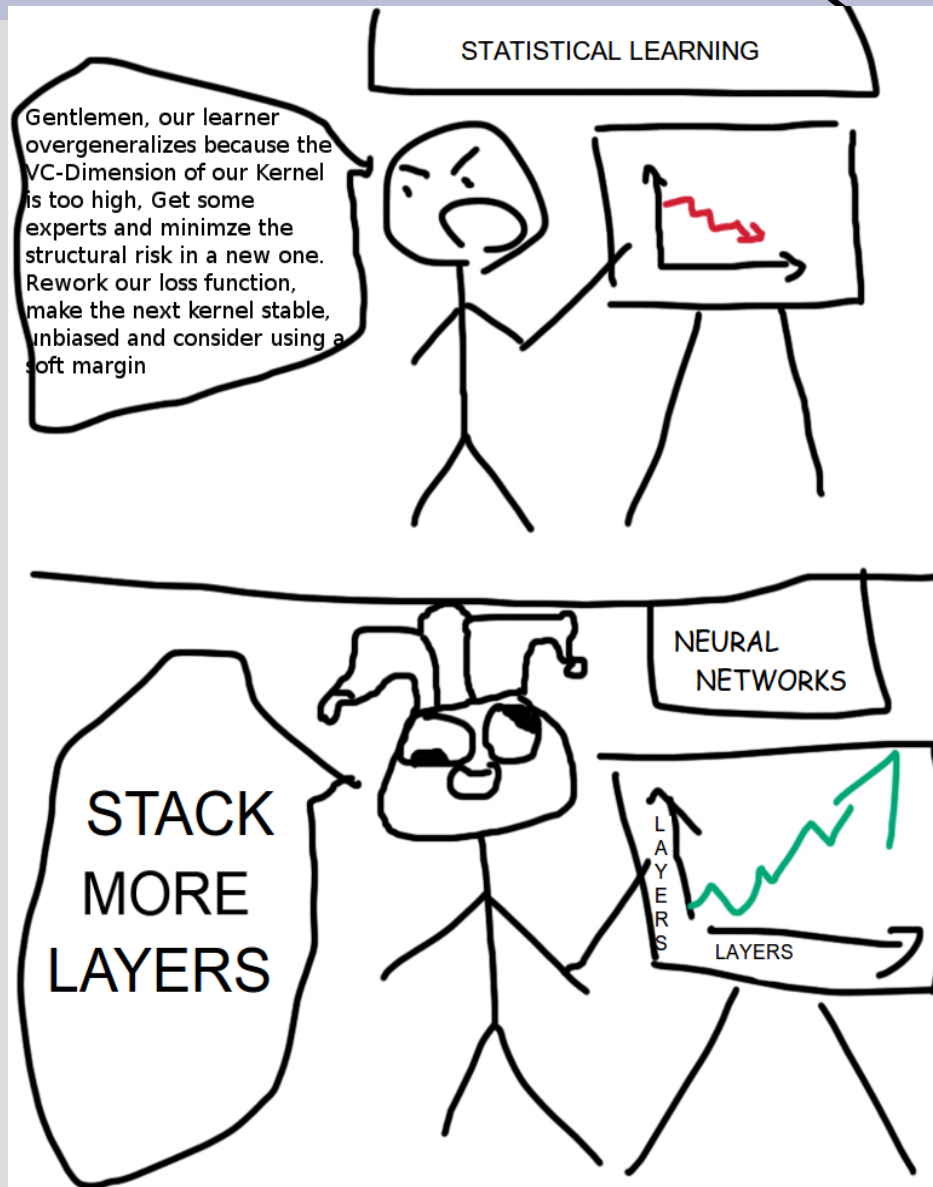


# Neural networks (Ch. 18.7)



# Review: Gradient Descent

Our iterative approach for gradient descent:

$$h_w(x) = w_0 + w_1 \cdot a_j + w_2 \cdot b_j + w_3 \cdot c_j + \dots = w \cdot x_j = w^T x_j$$

$$w_i \leftarrow w_i - \alpha \cdot \frac{\partial}{\partial w} \text{Loss}(h_w)$$

For a single variable:

$$w_i \leftarrow w_i + \alpha \cdot (y - h_w(x)) \cdot x_i$$

For multiple variables:

$$w_i \leftarrow w_i + \alpha \cdot \sum_{j=1}^N x_{j,i} (y_j - h_w(x_j))$$

# Biology: brains

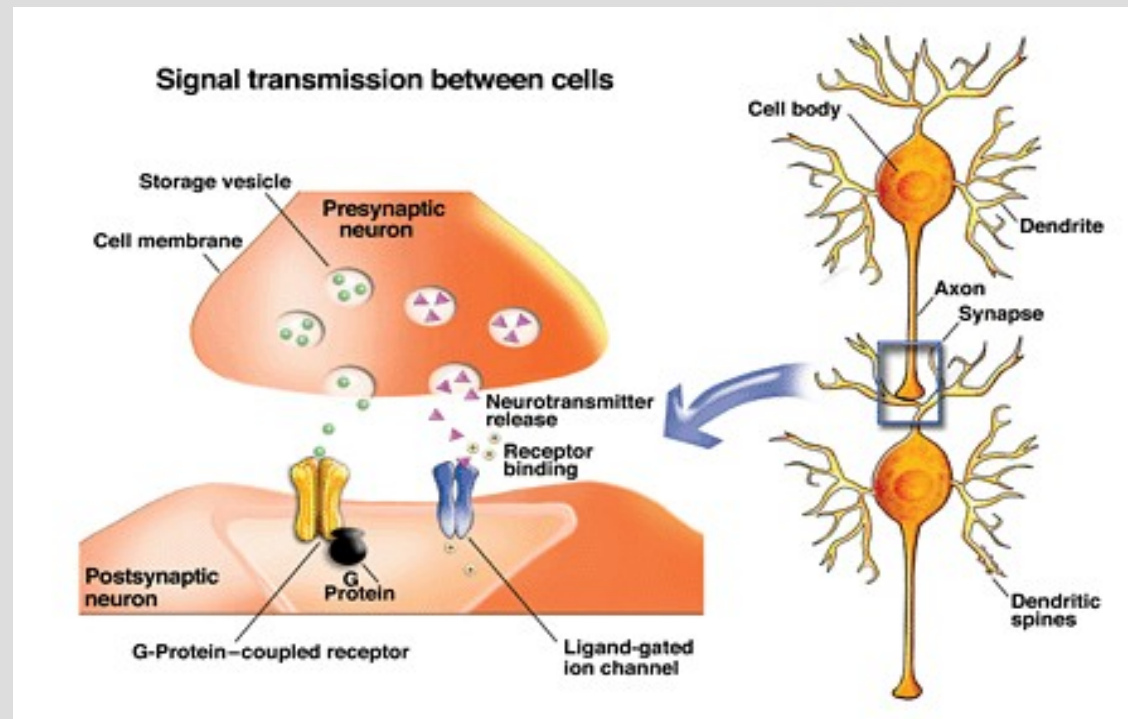
Computer science is fundamentally a creative process: building new & interesting algorithms

As with other creative processes, this involves mixing ideas together from various places

Neural networks get their inspiration from how brains work at a fundamental level (simplification... of course)

# Biology: brains

(Disclaimer: I am **not** a neuroscience-person)  
Brains receive small chemical signals at the “input” side, if there are enough inputs to “activate” it signals an “output”



# Biology: brains

An analogy is sleeping: when you are asleep, minor sounds will not wake you up

However, specific sounds in combination with their volume will wake you up



# Biology: brains

Other sounds might help you go to sleep  
(my majestic voice?)

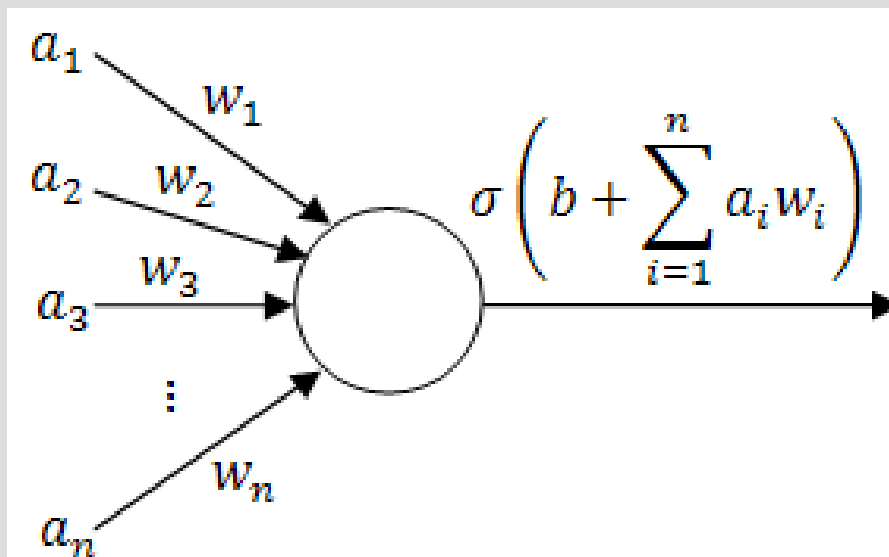
Many babies tend to sleep better with “white noise” and some people like the TV/radio on



# Neural network: basics

Neural networks are connected nodes, which can be arranged into layers (more on this later)

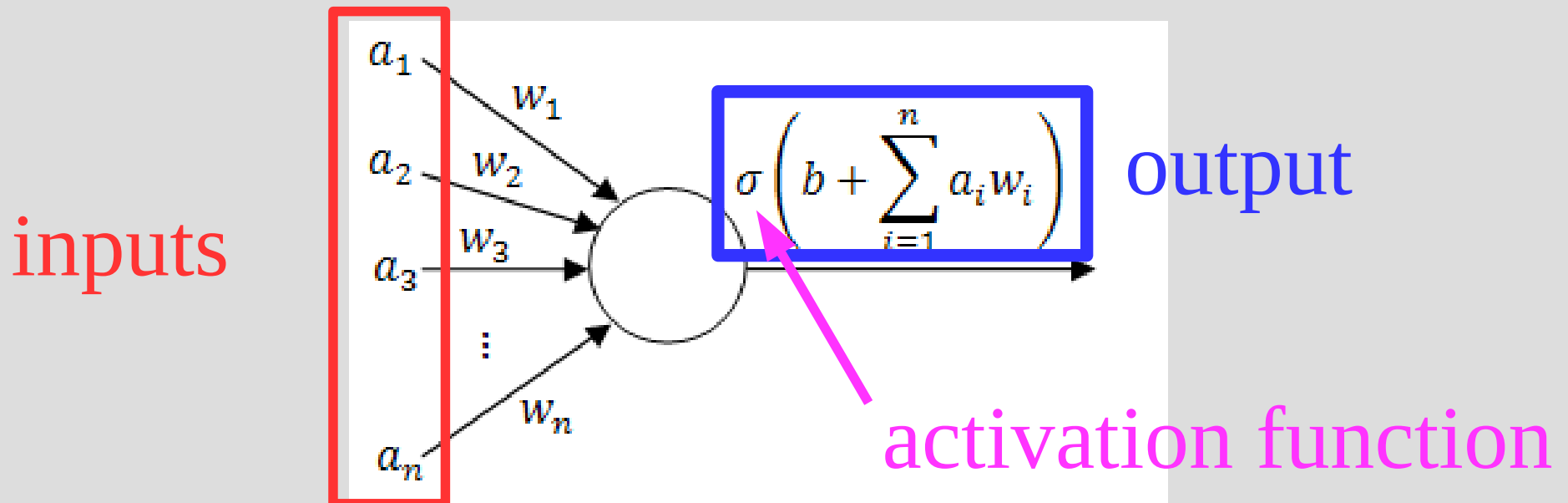
First is an example of a perceptron, the most simple NN; a single node on a single layer



# Neural network: basics

Neural networks are connected nodes, which can be arranged into layers (more on this later)

First is an example of a perceptron, the most simple NN; a single node on a single layer





# Mammals

Let's do an example with mammals...

First the definition of a mammal (wikipedia):

Mammals [posses]:

- (1) a neocortex (a region of the brain),
- (2) hair,
- (3) three middle ear bones,
- (4) and mammary glands

# Mammals

Common mammal misconceptions:

- (1) Warm-blooded
- (2) Does not lay eggs

Let's talk dolphins for one second.

<http://mentalfloss.com/article/19116/if-dolphins-are-mammals-and-all-mammals-have-hair-why-arent-dolphins-hairy>

Dolphins have hair (technically) for the first week after birth, then lose it for the rest of life ... I will count this as “not covered in hair”

# Perceptrons

Consider this example: we want to classify whether or not an animal is mammal via a perceptron (weighted evaluation)

We will evaluate on:

1. Warm blooded? (WB) Weight = 2
2. Lays eggs? (LE) Weight = -2
3. Covered hair? (CH) Weight = 3

If  $(2 \cdot WB + -2 \cdot LE + 3 \cdot CH > 1) \Rightarrow Mammal$

# Perceptrons

Consider the following animals:

Humans {WB=y, LE=n, CH=y}, mam=y

$$2(1) + -2(-1) + 3(1) = 7 > 1 \dots \text{Correct!}$$

Bat {WB=sorta, LE=n, CH=y}, mam=y

$$2(0.5) + -2(-1) + 3(1) = 6 > 1 \dots \text{Correct!}$$

What about these?

Platypus {WB=y, LE=y, CH=y}, mam=y

Dolphin {WB=y, LE=n, CH=n}, mam=y

Fish {WB=n, LE=y, CH=n}, mam=n

Birds {WB=y, LE=y, CH=n}, mam=n

# Neural network: feed-forward

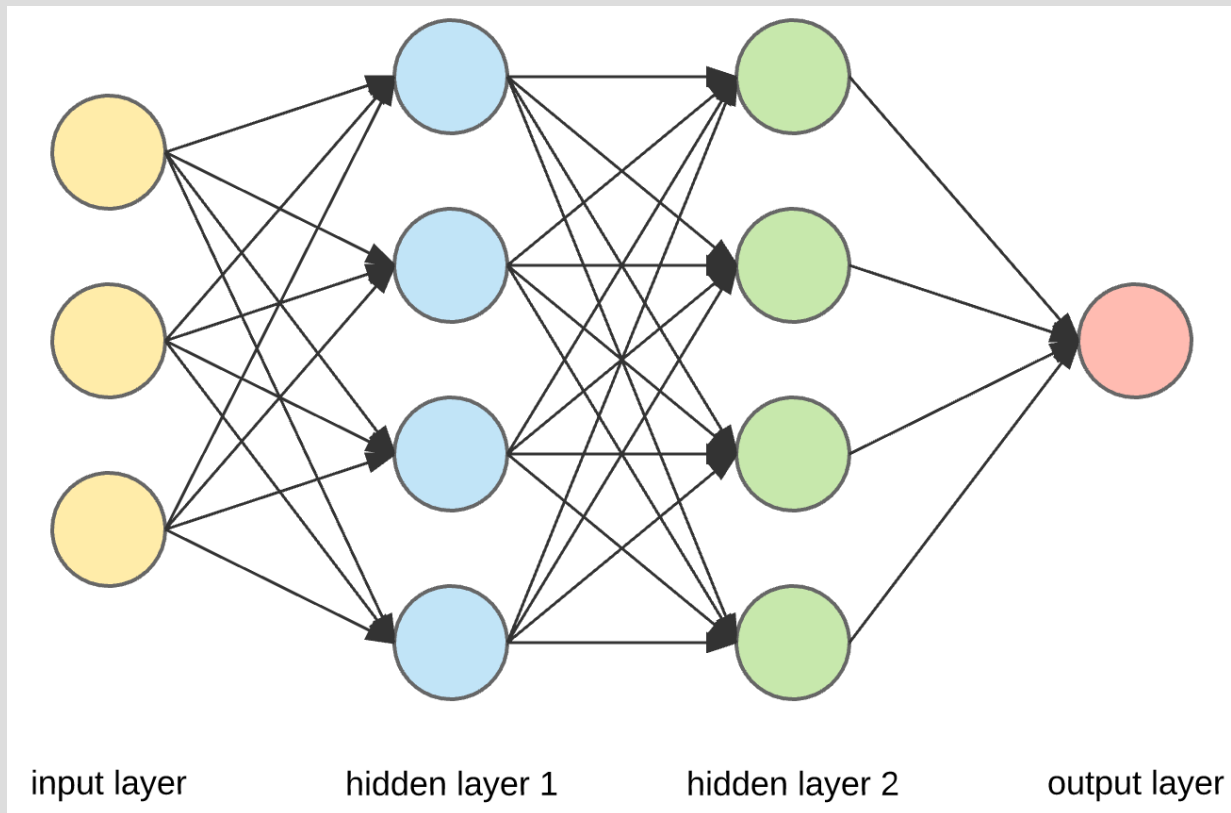
Today we will look at feed-forward NN, where information flows in a single direction

Recurrent networks can have outputs of one node loop back to inputs as previous

This can cause the NN to not converge on an answer (ask it the same question and it will respond differently) and also has to maintain some “initial state” (all around messy)

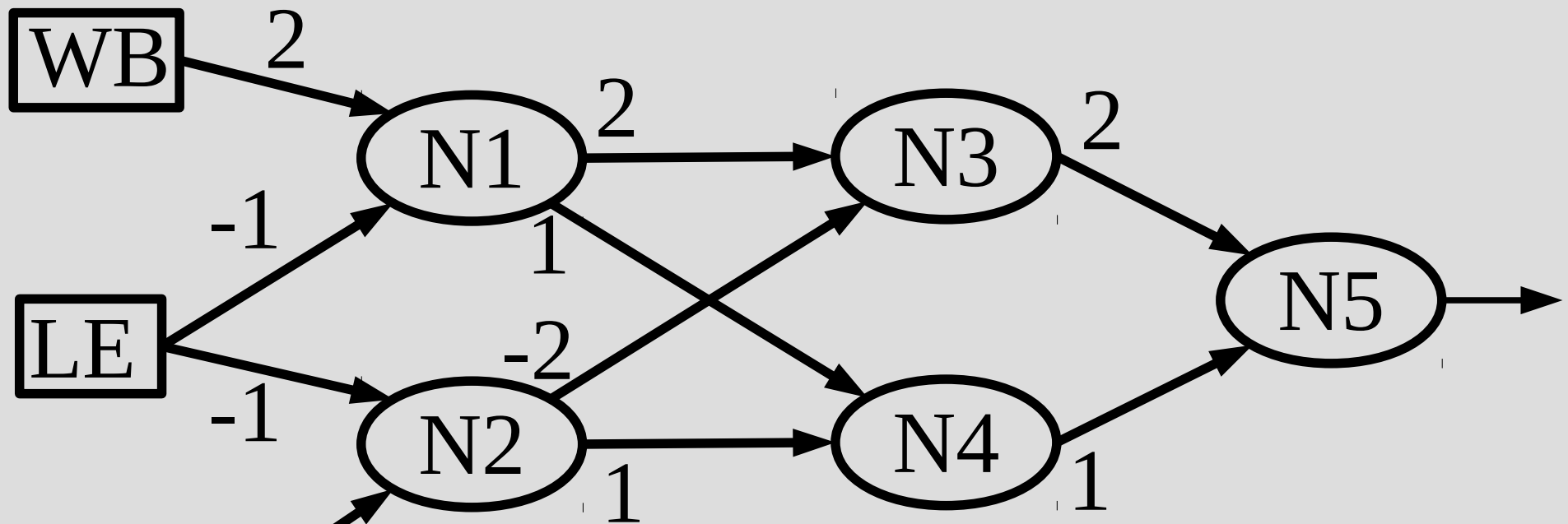
# Neural network: feed-forward

Since in feed-forward neural networks info only flows in one direction, we can group nodes into “layers” based off dependencies



# Neural network: feed-forward

Let's expand our mammal classification to 5 nodes in 3 layers (weights on edges):

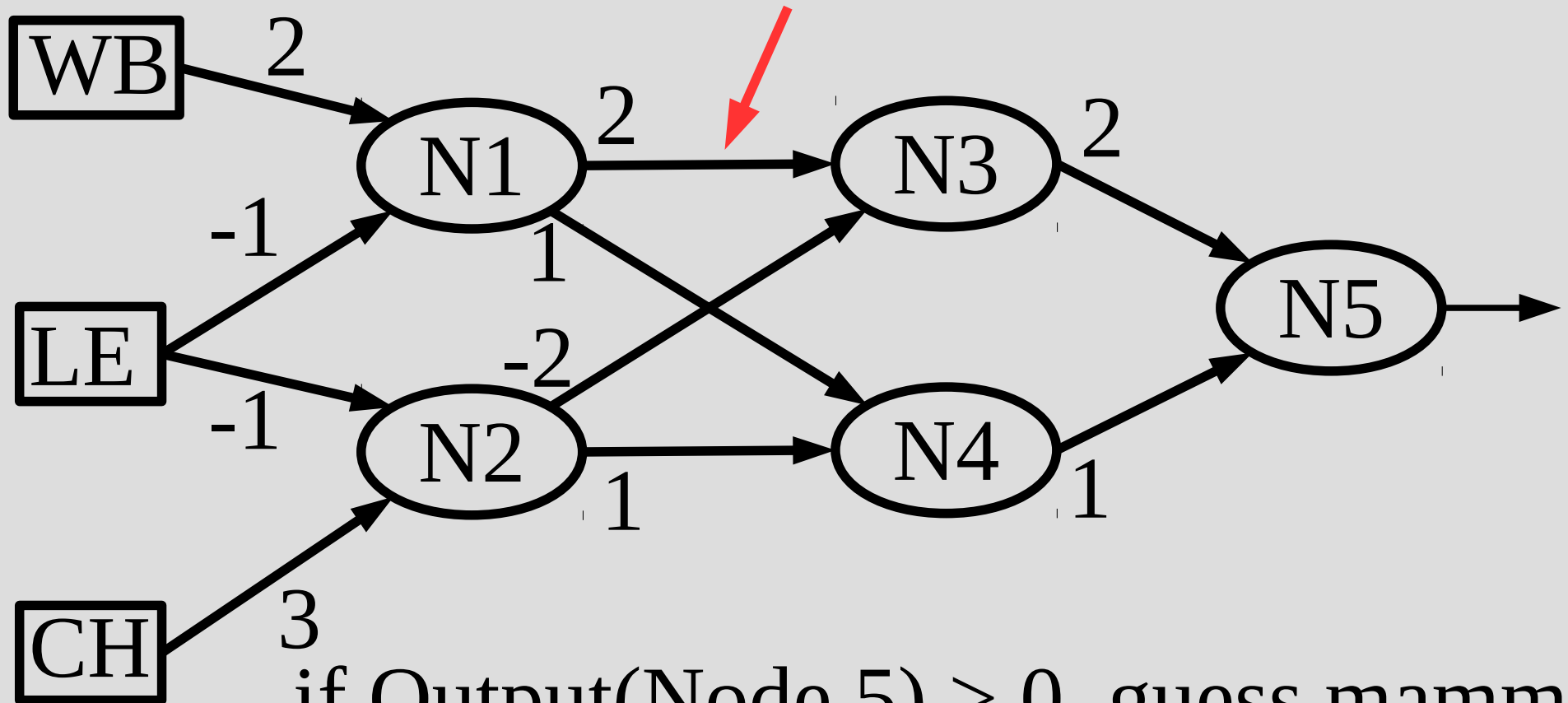


if  $\text{Output}(\text{Node } 5) > 0$ , guess mammal

# Neural network: feed-forward

You try Bat on this: {WB=0, LE=-1, CH=1}

Assume (for now) output = sum input

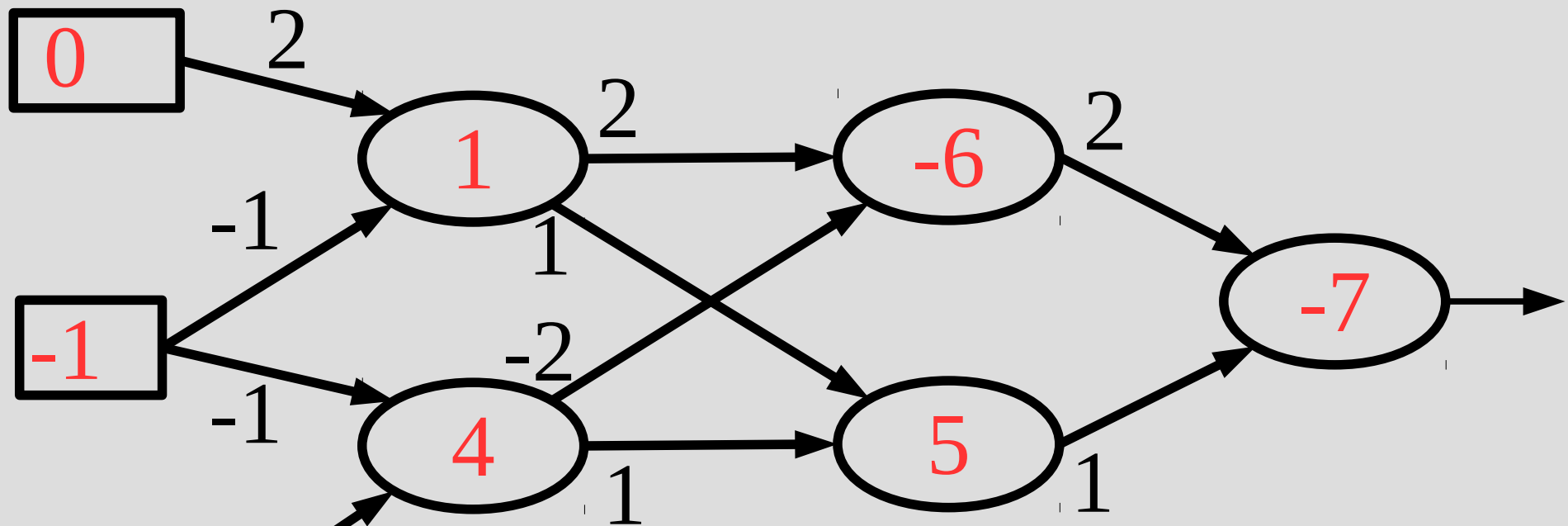


if Output(Node 5) > 0, guess mammal



# Neural network: feed-forward

Output is -7, so bats are not mammal... Oops...



if Output(Node 5) > 0, guess mammal

# Neural network: feed-forward

In fact, this is no better than our 1 node NN

This is because we simply output a linear combination of weights into a linear function (i.e. if  $f(x)$  and  $g(x)$  are linear... then  $g(x)+f(x)$  is also linear)

Ideally, we want a activation function that has a limited range so large signals do not always dominate... what should we use?

# Neural network: feed-forward

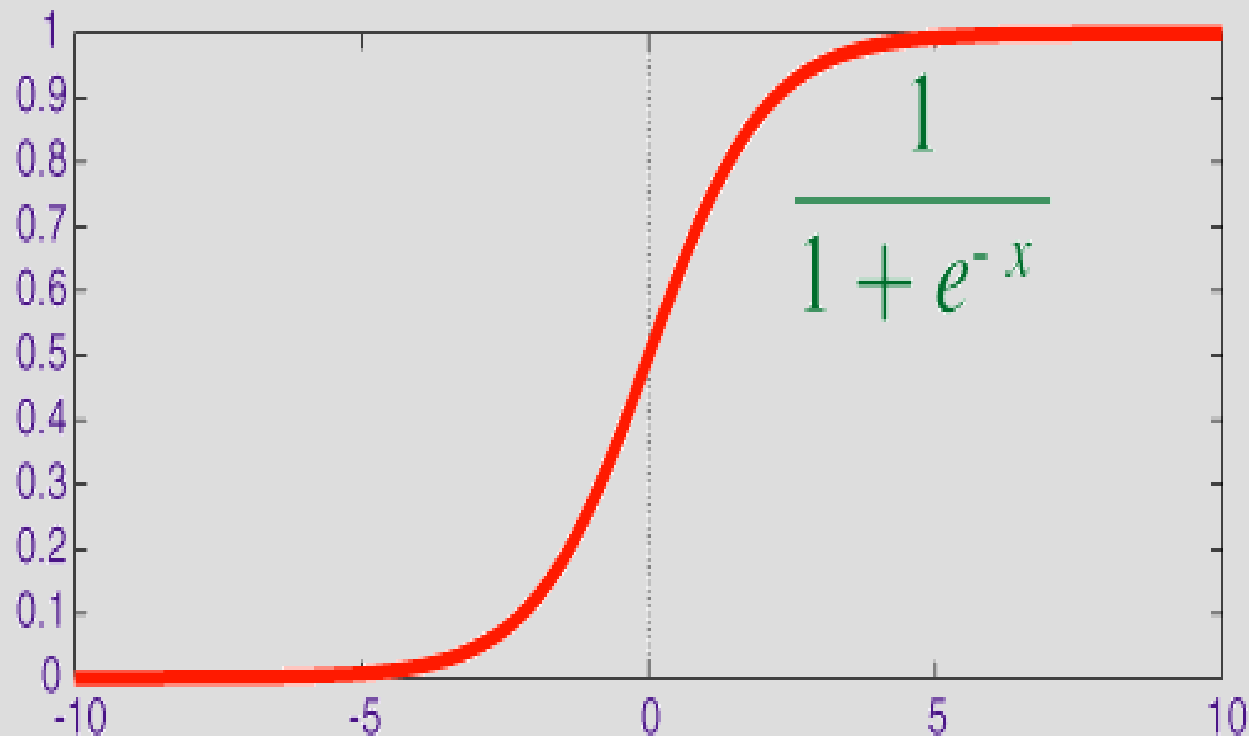
One commonly used function is the sigmoid:

$$S(x) = \frac{1}{1+e^{-x}} \text{ (in Logistic function family)}$$

Why good?

1. Continuous  
(derivatives exist)

2. Tells you  
“how similar”  
not just T/F



# Back-propagation

The neural network is as good as its structure and weights on edges

Structure we will ignore (more complex), but there is an automated way to learn weights

Whenever a NN incorrectly answer a problem, the weights play a “blame game”...

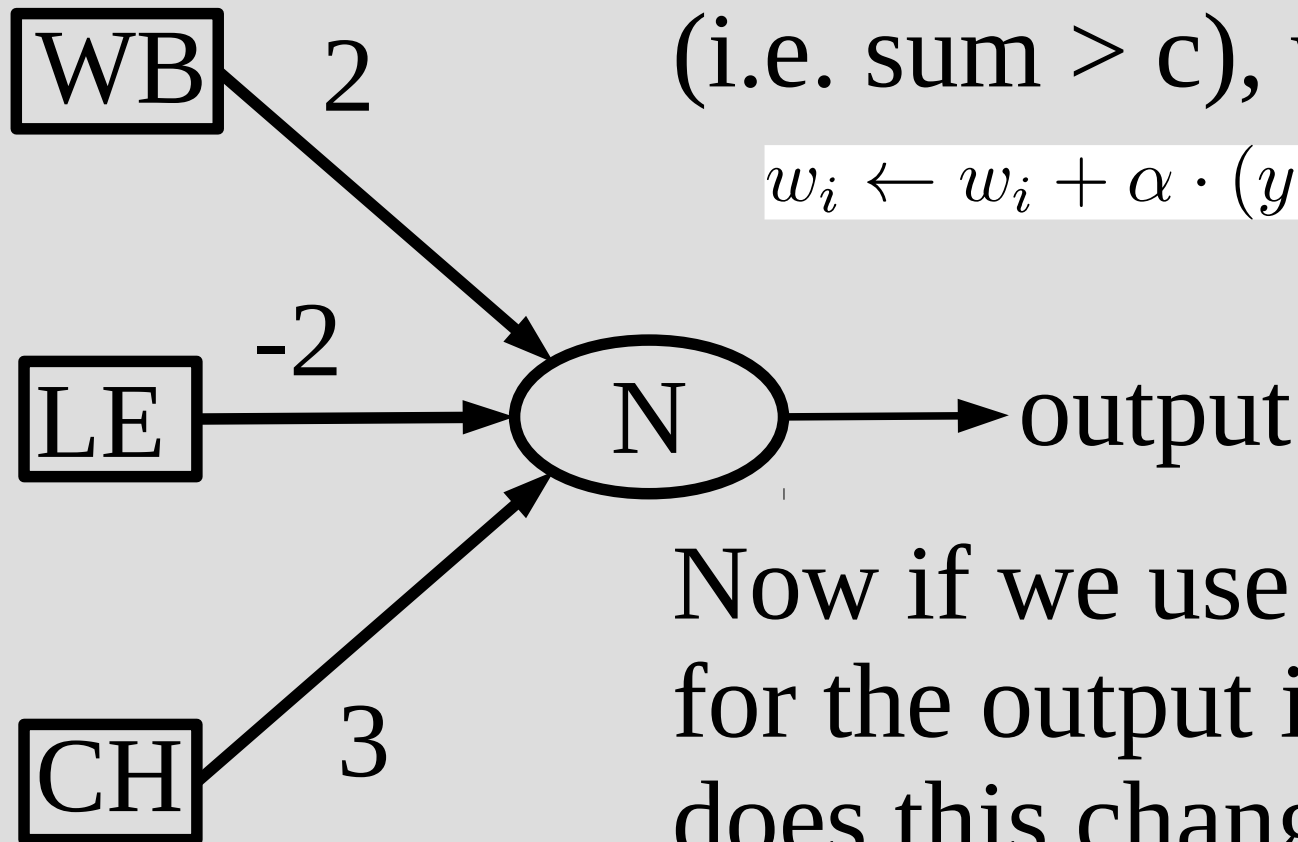
- Weights that have a big impact to the wrong answer are reduced

# Back-propagation

Let's go back to our simple Neural Network:

When output was threshold  
(i.e.  $\text{sum} > c$ ), we had:

$$w_i \leftarrow w_i + \alpha \cdot (y - h_w(x)) \cdot x_i$$



Now if we use the sigmoid  
for the output instead... how  
does this change?

# Back-propagation

Basically we used to have:

$$Loss(w) = (y - h_w(x))^2 = (y - w \cdot x)^2$$

Now we have:

$$Loss(w) = \frac{(y - h_w(x))^2}{2} = \frac{(y - S(w \cdot x))^2}{2}$$

compare line output

compare output after sigmoid

So we have to use our good old friend, the chain rule! So...

$$w_i \leftarrow w_i + \alpha \cdot (y - h_w(x)) \cdot x_i$$

... turns into (math with chain rule) ...

$$w_i \leftarrow w_i + \alpha \cdot S'(w \cdot x) \cdot (y - h_w(x)) \cdot x_i$$

# Back-propagation

So if we had input:

WB = 1, LE = -1, CH = 0.5

... and we expected output “1”

$$w_i \leftarrow w_i + \alpha \cdot S'(w \cdot x) \cdot (y - h_w(x)) \cdot x_i$$

Then we would update the WB weight as:

$$\begin{aligned} w_{WB} &= 2 + \alpha \cdot S'(1 \cdot 2 + -1 \cdot 2 + 0.5 \cdot 3) \cdot (1 - output) \cdot 1 \\ &= 2 + 0.5 \cdot S'(5.5) \cdot (1 - output) \quad S'(x) = S(x) \cdot (1 - S(x)) \\ &= 2 + 0.5 \cdot S(5.5) \cdot (1 - S(5.5)) \cdot (1 - S(5.5)) \\ &= 2 + 0.5 \cdot 0.9959 \cdot 0.004070 \cdot (1 - 0.9959) \\ &= 2.00000830929 \end{aligned}$$

# Back-propagation

The neural network is as good as its structure and weights on edges

Structure we will ignore (more complex), but there is an automated way to learn weights

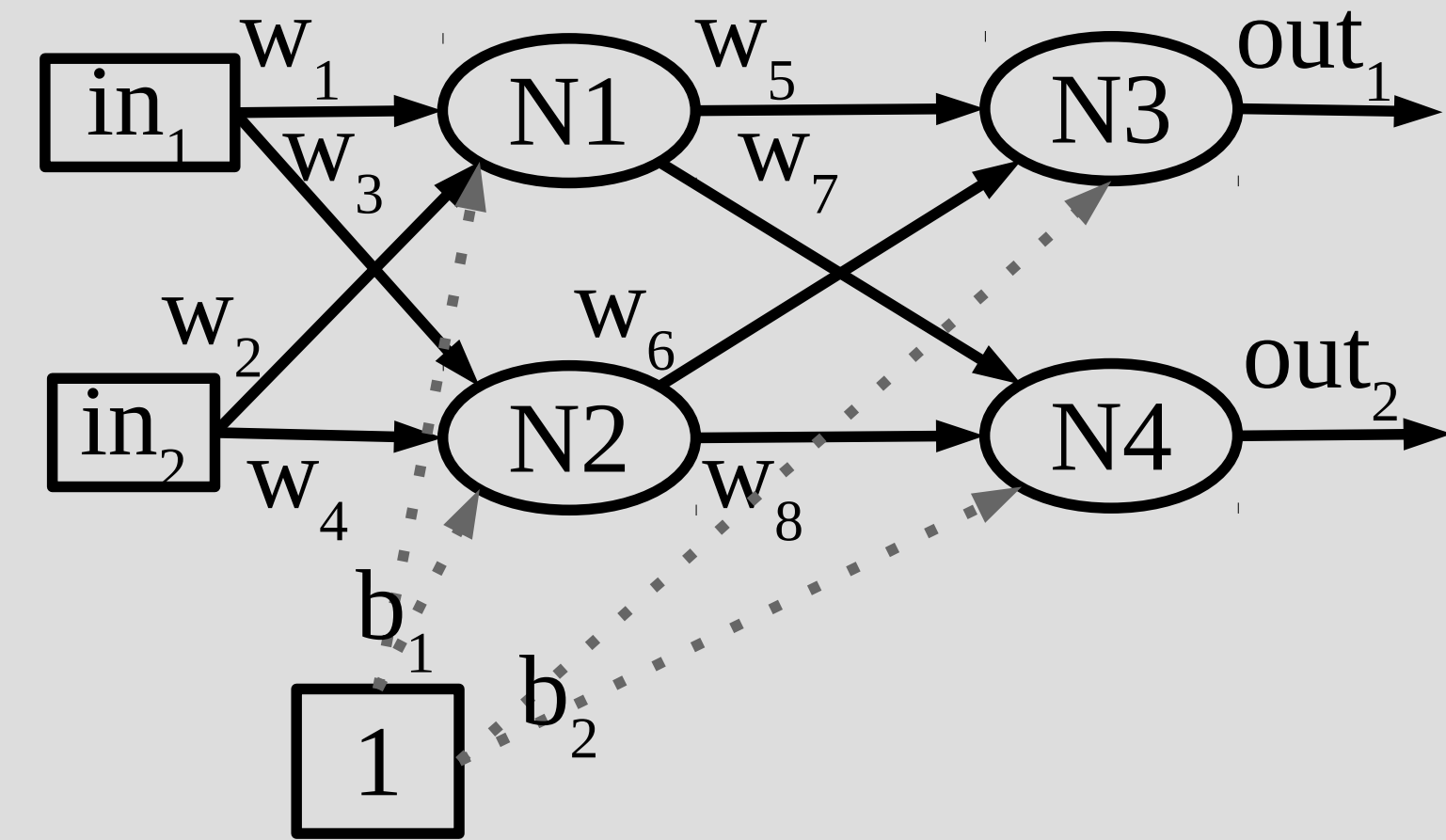
Whenever a NN incorrectly answer a problem, the weights play a “blame game”...

- Weights that have a big impact to the wrong answer are reduced



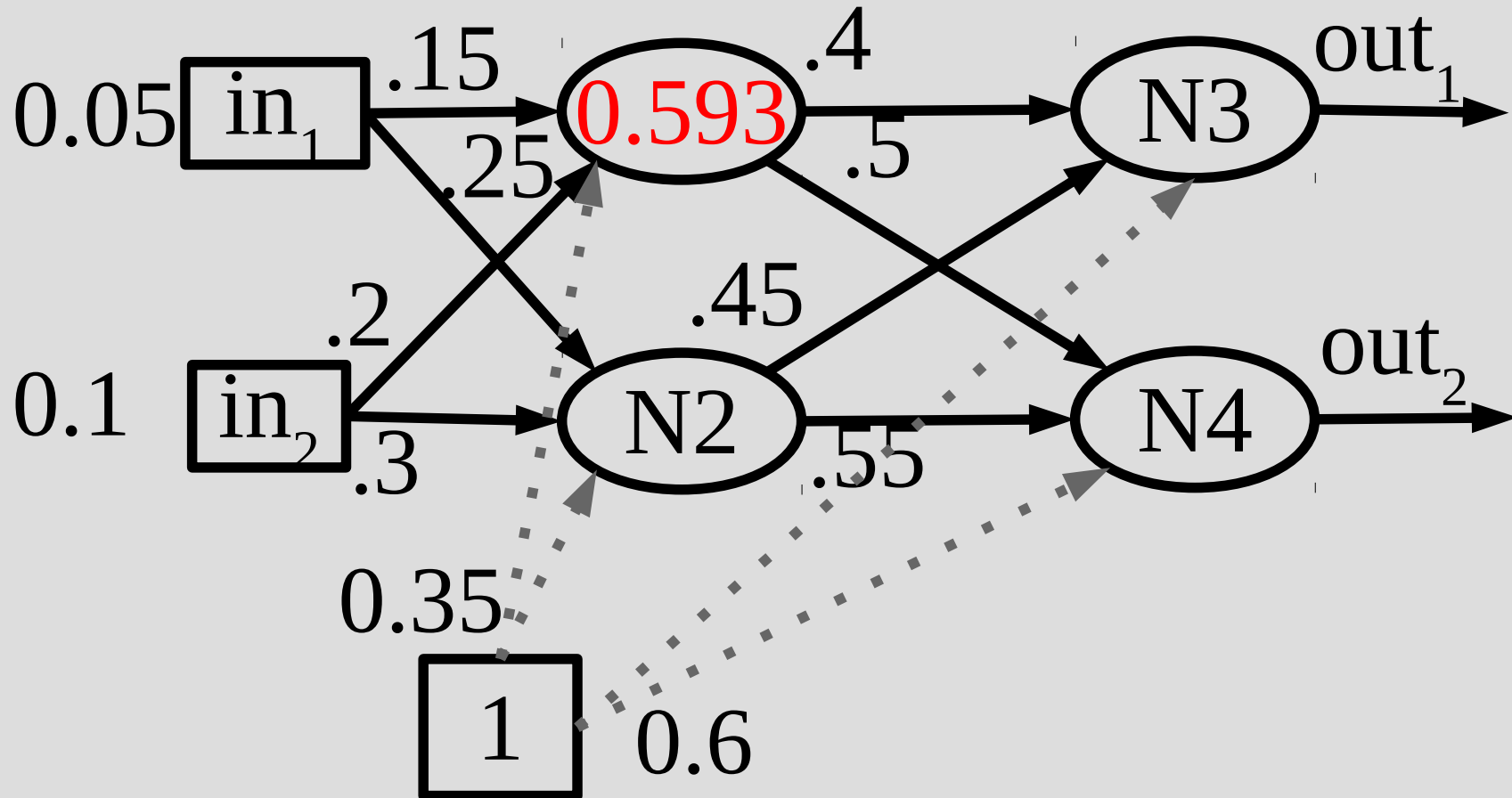
# Back-propagation

Consider this example: 4 nodes, 2 layers



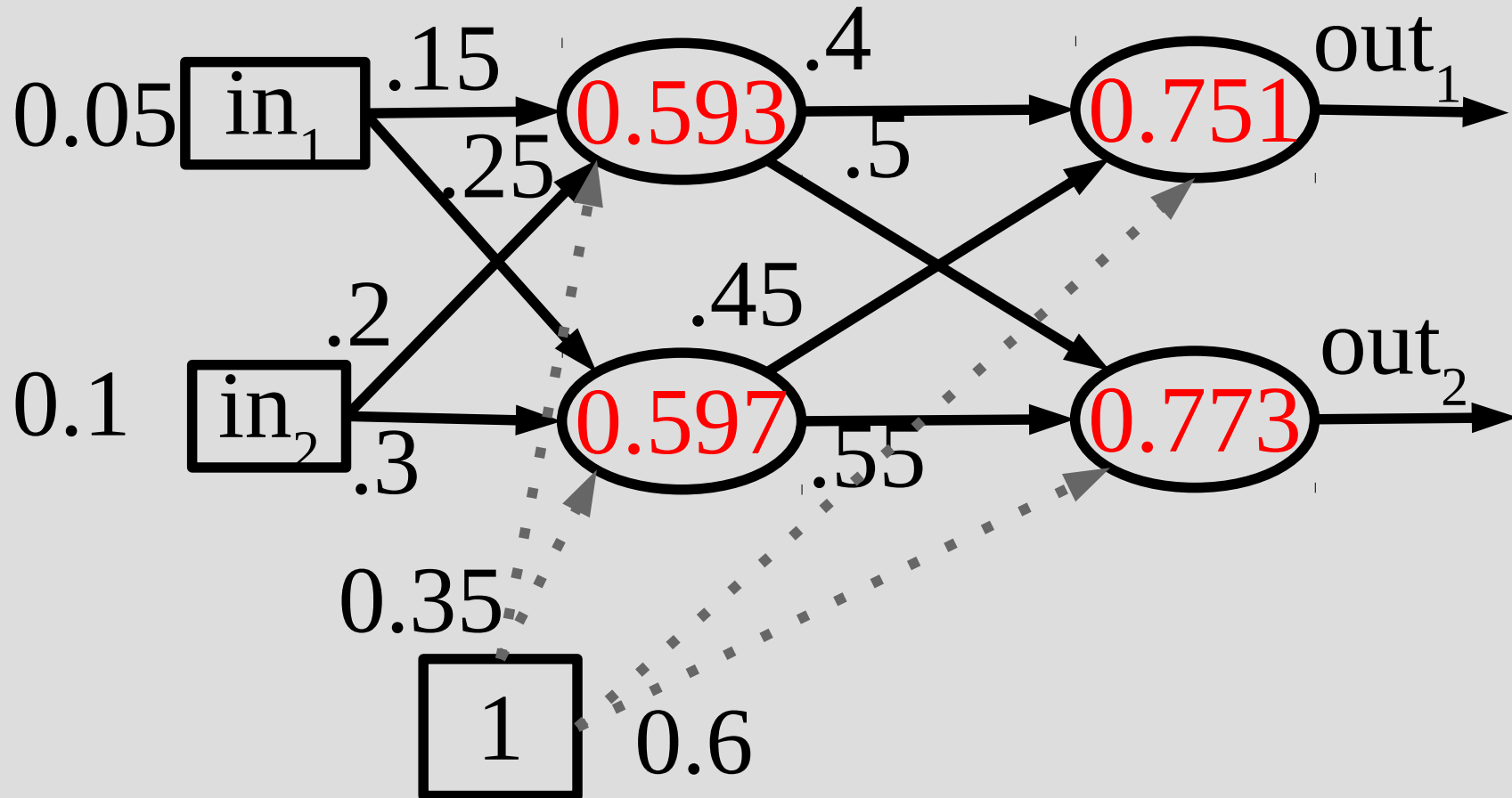
This node as a constant bias of 1

# Back-propagation



Node 1:  $0.15 * 0.05 + 0.2 * 0.1 + 0.35 = 0.3775$  input  
thus it outputs (all edges)  $S(0.3775) = 0.59327$

# Back-propagation



Eventually we get:  $out_1 = 0.751$ ,  $out_2 = 0.773$

Suppose wanted:  $out_1 = 0.01$ ,  $out_2 = 0.99$

# Back-propagation

We will define the error as:  $\frac{\sum_i (\text{correct}_i - \text{output}_i)^2}{2}$   
(you will see why shortly)

Suppose we want to find how much  $w_5$  is to blame for our incorrectness

We then need to find:  $\frac{\partial \text{Error}}{\partial w_5}$

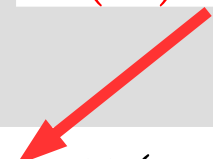
Apply the chain rule:

$$\frac{\partial \text{Error}}{\partial \text{out}_1} \cdot \frac{\partial S(\text{In}(N_3))}{\partial \text{In}(N_3)} \cdot \frac{\partial \text{In}(N_3)}{\partial w_5}$$

# Back-propagation

$$Error = \frac{\sum_i (correct_i - output_i)^2}{2}$$

$$\begin{aligned} \frac{\partial Error}{\partial out_1} &= -(correct_1 - out_1) \\ &= -(0.01 - 0.751) = 0.741 \end{aligned}$$

$$\text{As } S'(x) = S(x) \cdot (1 - S(x))$$


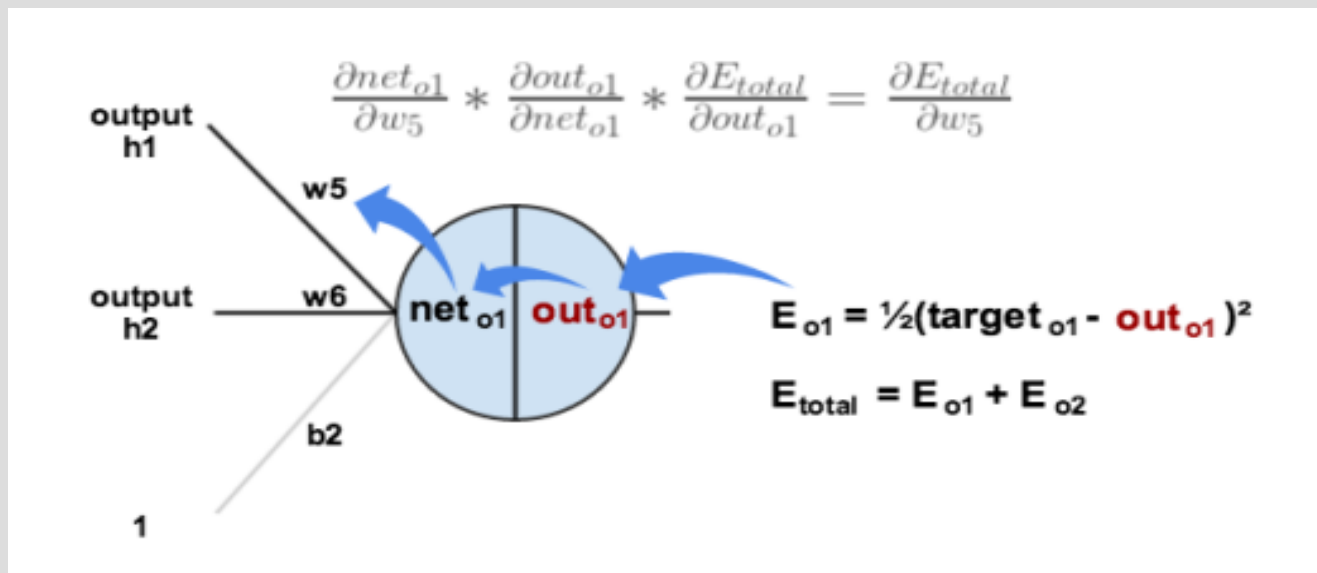
$$\begin{aligned} \frac{\partial S(In(N_3))}{\partial In(N_3)} &= S(In(N_3)) \cdot (1 - S(In(N_3))) \\ &= 0.751 \cdot (1 - 0.751) = 0.187 \end{aligned}$$

$$\begin{aligned} \frac{\partial In(N_3)}{\partial w_5} &= \frac{\partial w_5 \cdot Out(N_1) + w_6 \cdot Out(N_2) + b_2 \cdot 1}{\partial w_5} \\ &= Out(N_1) = 0.593 \end{aligned}$$

$$\text{Thus, } \frac{\partial Error}{\partial w_5} = 0.593 \cdot 0.187 \cdot 0.741 = 0.0822$$

# Back-propagation

In a picture we did this:



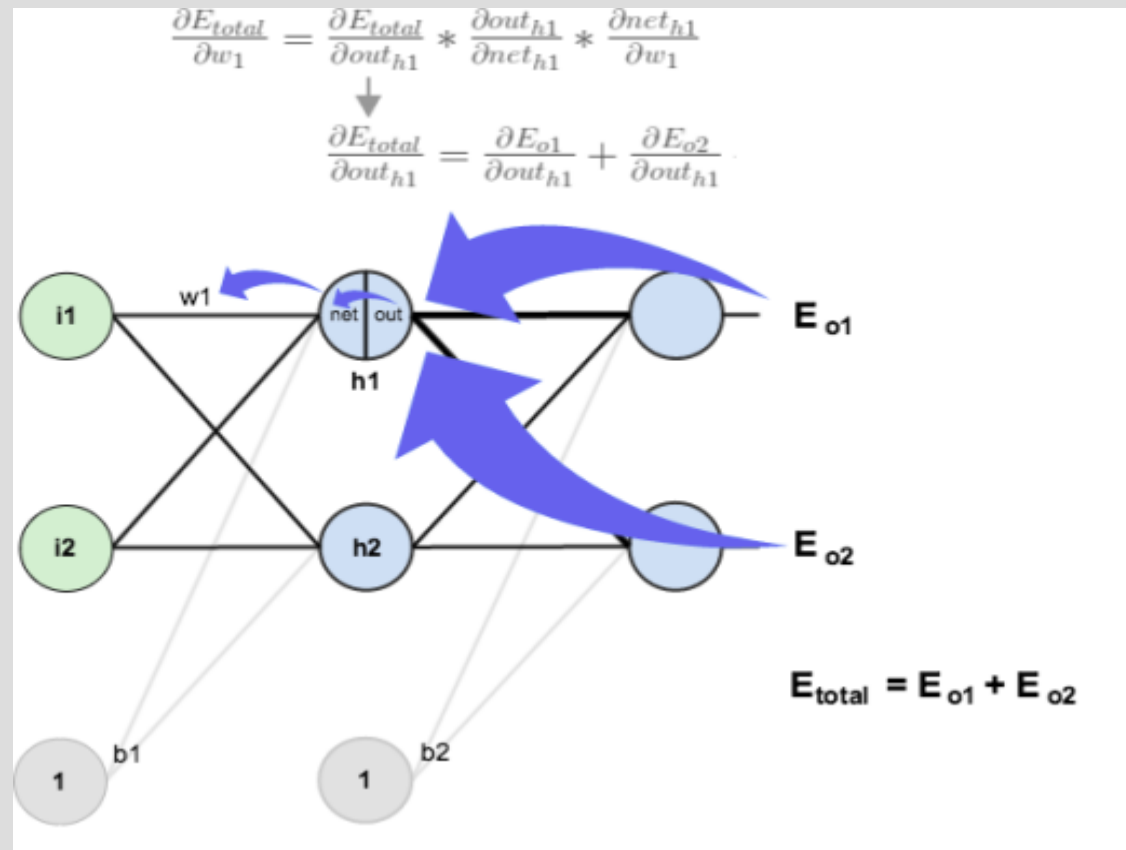
Now that we know  $w_5$  is 0.08217 part responsible, we update the weight by:

$$w_5 \leftarrow w_5 - \alpha * 0.0822 = 0.3589 \text{ (from 0.4)}$$

$\alpha$  is learning rate, set to 0.5

# Back-propagation

For  $w_1$  it would look like:



(book describes how to dynamic program this)

# Back-propagation

Specifically for  $w_1$  you would get:

$$\frac{\partial Error}{\partial S(In(N_1))} = \frac{\partial Error_1}{\partial S(In(N_1))} + \frac{\partial Error_2}{\partial S(In(N_1))}$$

$$\begin{aligned} \frac{\partial S(In(N_1))}{\partial In(N_1)} &= S(In(N_1)) \cdot (1 - S(In(N_1))) \\ &= 0.593 \cdot (1 - 0.593) = 0.241 \end{aligned}$$

$$\begin{aligned} \frac{\partial In(N_3)}{\partial w_5} &= \frac{\partial w_1 \cdot In_1 + w_2 \cdot In_2 + b_1 \cdot 1}{\partial w_5} \\ &= In_1 = 0.05 \end{aligned}$$

Next we have to break down the top equation...



# Back-propagation

$$\frac{\partial Error}{\partial S(In(N_1))} = \frac{\partial Error_1}{\partial S(In(N_1))} + \frac{\partial Error_2}{\partial S(In(N_1))}$$

$$\frac{\partial Error_1}{\partial S(In(N_1))} = \frac{\partial Error_1}{\partial S(In(N_3))} \cdot \frac{\partial S(In(N_3))}{\partial In(N_3)} \cdot \frac{\partial In(N_3)}{\partial S(In(N_1))}$$

From before...  $\frac{\partial Error_1}{\partial S(In(N_3))} \cdot \frac{\partial S(In(N_3))}{\partial In(N_3)}$   
 $= 0.7414 \cdot 0.1868 = 0.1385$

$$\frac{\partial In(N_3)}{\partial S(In(N_1))} = \frac{\partial w_5 \cdot S(In(N_1)) + w_6 \cdot S(In(N_2)) + b_1 \cdot 1}{\partial S(In(N_1))}$$

$= w_5 = 0.4$

Thus,  $\frac{\partial Error_1}{\partial S(In(N_1))} = 0.1385 \cdot 0.4 = 0.05540$

# Back-propagation

Similarly for  $Error_2$  we get:

$$\begin{aligned}\frac{\partial Error}{\partial S(In(N_1))} &= \frac{\partial Error_1}{\partial S(In(N_1))} + \frac{\partial Error_2}{\partial S(In(N_1))} \\ &= 0.05540 + -0.01905 = 0.03635\end{aligned}$$

$$\text{Thus, } \frac{\partial Error}{\partial w_1} = 0.03635 \cdot 0.2413 \cdot 0.05 = 0.0004386$$

$$\text{Update } w_1 \leftarrow w_1 - \alpha \frac{\partial Error}{\partial w_1} = 0.15 - 0.5 \cdot 0.0004386 = 0.1498$$

You might notice this is small...

This is an issue with neural networks, deeper the network the less earlier nodes update

# NN examples

Despite this learning shortcoming, NN are useful in a wide range of applications:

Reading handwriting

Playing games

Face detection

Economic predictions

Neural networks can also be very powerful when combined with other techniques (genetic algorithms, search techniques, ...)

# NN examples

Examples:

<https://www.youtube.com/watch?v=umRdt3zGgpU>

<https://www.youtube.com/watch?v=qv6UV0Q0F44>

<https://www.youtube.com/watch?v=xcIBoPuNIiw>

<https://www.youtube.com/watch?v=0Str0Rdkxxo>

[https://www.youtube.com/watch?v=l2\\_CPB0uBkc](https://www.youtube.com/watch?v=l2_CPB0uBkc)

<https://www.youtube.com/watch?v=0VTI1BBLydE>

# NN examples

AlphaGo/Zero has been in the news recently, and is also based on neural networks

AlphaGo uses Monte-Carlo tree search guided by the neural network to prune useless parts

Often limiting Monte-Carlo in a static way reduces the effectiveness, much like mid-state evaluations can limit algorithm effectiveness

# NN examples

Basically, AlphaGo uses a neural network to “prune” parts for a Monte-carlo search

