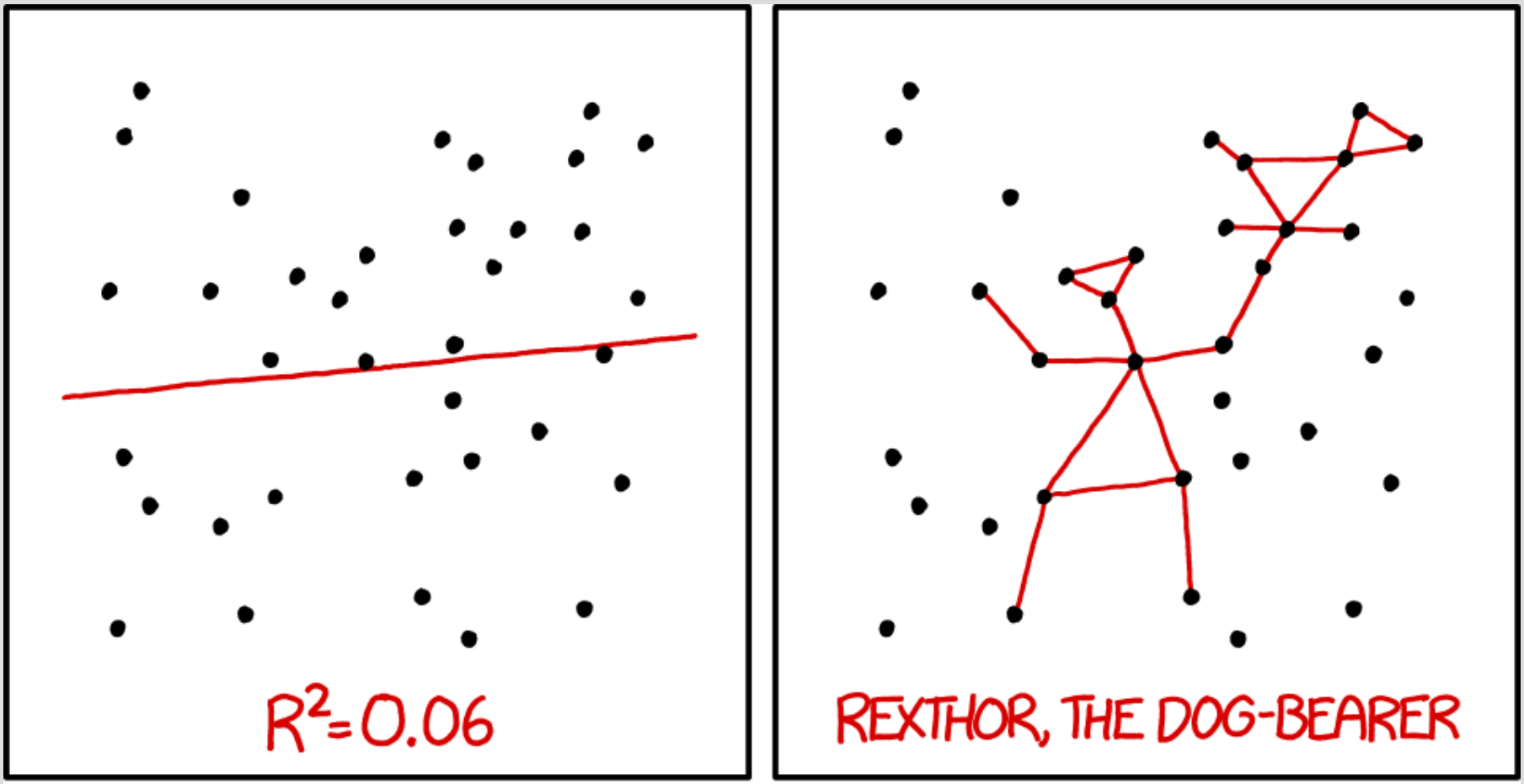
Linear Regression/Classification (Ch. 18.6-18.7)



Announcements

Homework 4 due Sunday

Test next Wednesday... covers ch 15-17 (HW 3 & 4)

Linear Regression

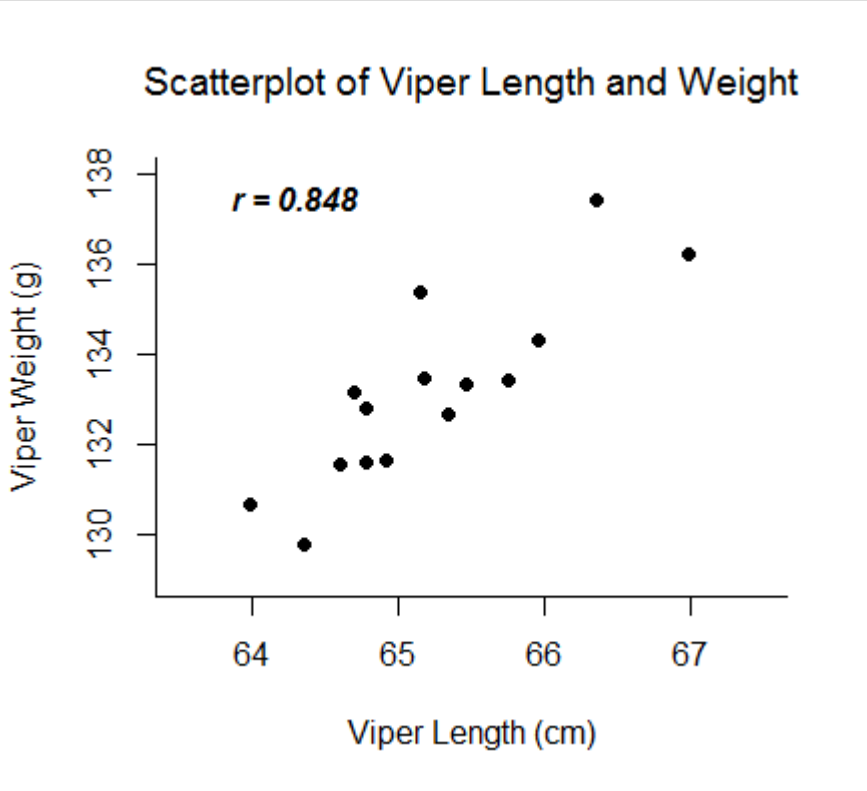
Let’s move away from decision trees (yay!) and talk about more general learning

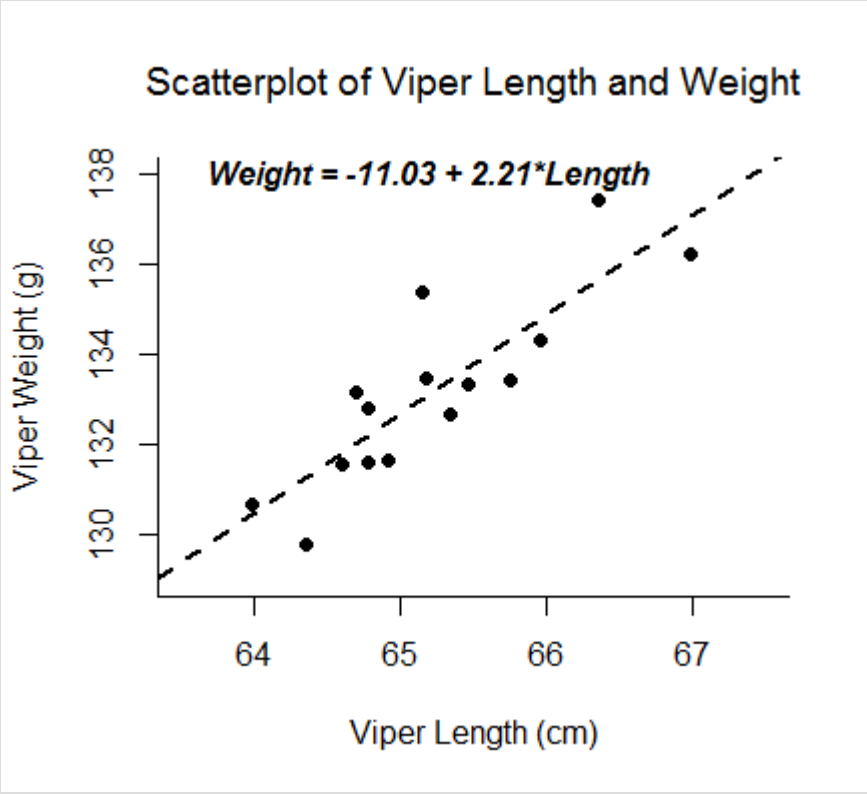
Today we will look at regression a bit (as I have been ignoring it mostly)

This is a concept that you may have encountered before, but not in the context of learning

Linear Regression

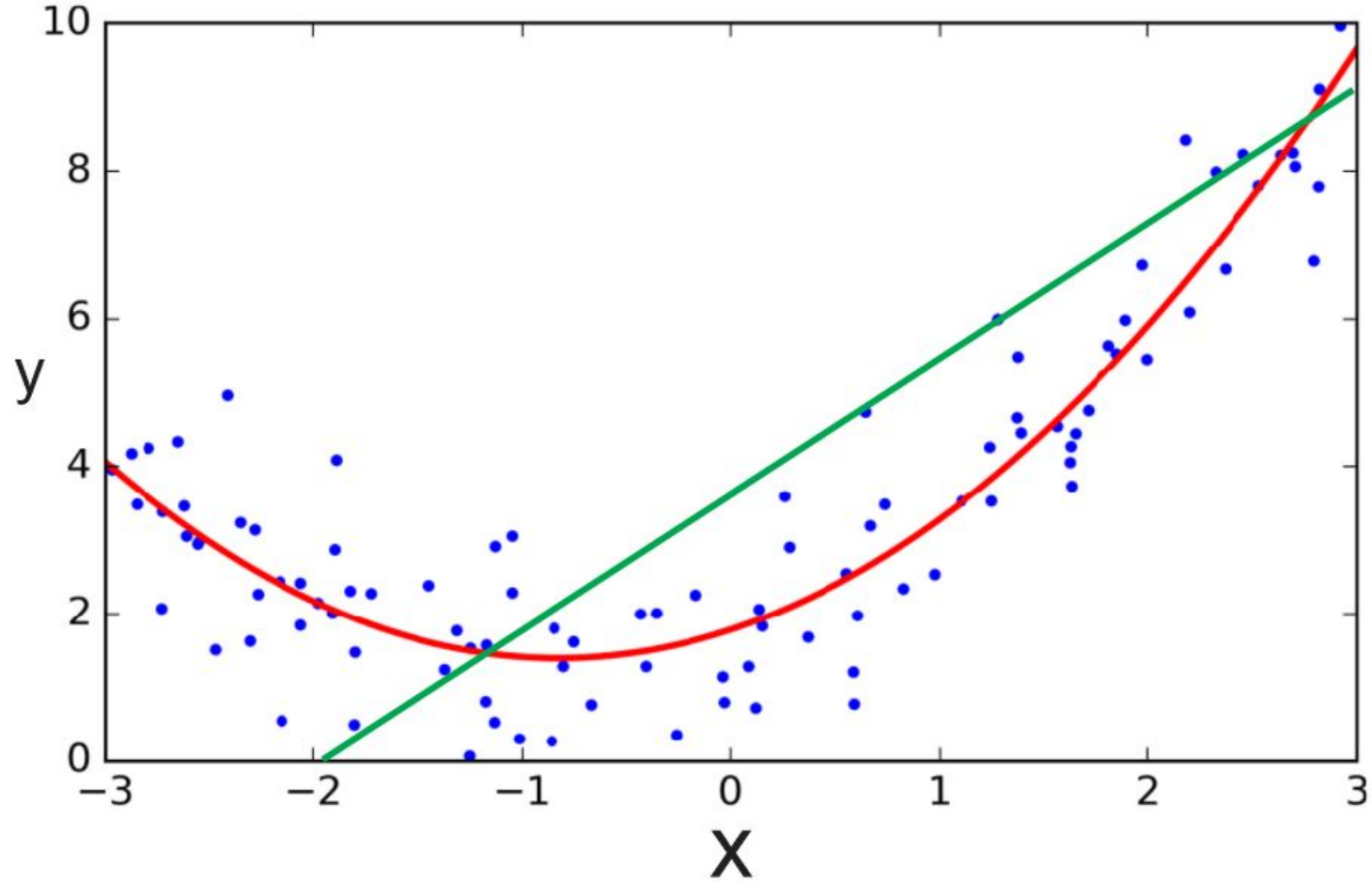
Idea: You have a bunch of data points, and you want to find the line “closest” to them





Linear Regression

Why linear and not some polynomial?



Linear Regression

Why linear and not some polynomial?

It is a lot harder to “overfit” with a linear fit, yet it still gets the major “trend” of data

Also a bit to “visualize” data if high dimension

Another bonus is that it makes the calculations much easier (which is always nice...)

Linear Regression: How To

To find this line, let's start with the simple case: only one variable

Then our line will look like (call them "h" just like our learning trees):

$$h_w(x) = y = w_0 + w_1 \cdot x$$

w is $\{w_0, w_1\}$ as parameters

Then we need to define what "fit to data" means (i.e. how do we calculate how "wrong" a line is)

Linear Regression: How To

There are multiple options, but a common choice is the square difference (called “loss”):

$$Loss(h_w) = \sum_{j=1}^{N} (y_j - h_w(x_j))^2 = \sum_{j=1}^{N} (y_j - w_0 + w_1 \cdot x_j)^2$$

y_j is actual y-coordinate

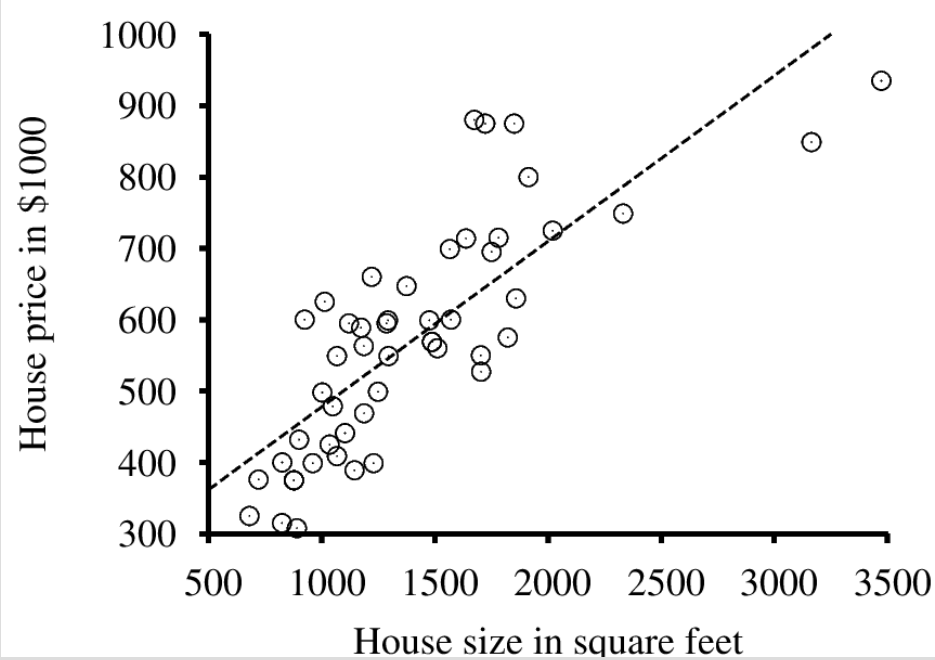
$h_w(x_j)$ is approximated (line) y-coordinate

... where N is the number of examples/points

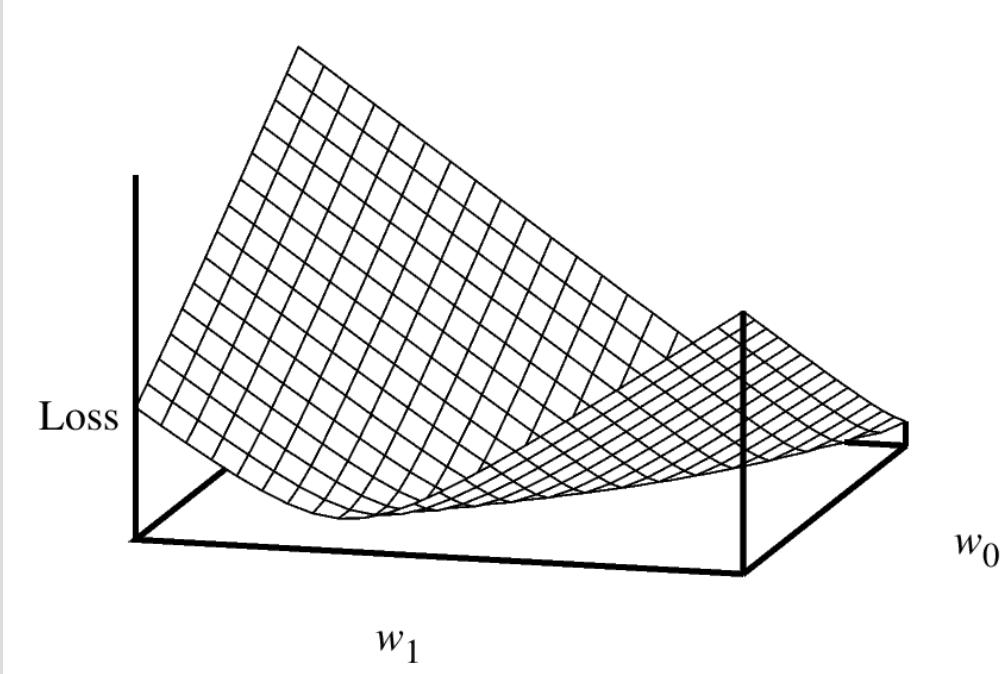
This makes sense as it penalizes “wrong” answers more the further they are away (two points off by 1 better than one off by 2)

Linear Regression: How To

You can plot this loss function (z-axis) with respect to the choice of w_0 and w_1



Regression



Loss

Linear Regression: How To

We want the regression line (w_0, w_1) to have the lowest loss possible

As the loss function looks convex (it is), the minimum is unique, so from calculus we want:

$$\frac{\partial}{\partial w_0} Loss(h_w) = 0 = \frac{\partial}{\partial w_0} \sum_{j=1}^{N} (y_j - w_0 + w_1 \cdot x_1)^2$$

bottom is when both w_0 and w_1 derivatives zero

$$\frac{\partial}{\partial w_1} Loss(h_w) = 0 = \frac{\partial}{\partial w_1} \sum_{j=1}^{N} (y_j - w_0 + w_1 \cdot x_1)^2$$

Linear Regression: How To

It is not too hard to do a bit of calculus to find the unique solution for w_0 and w_1:

$$w_1 = \frac{N \cdot (\sum_j x_j \cdot y_j) - (\sum_j x_j) \cdot (\sum_j y_j)}{N \cdot (\sum_j x_j^2) - (\sum_j x_j)^2}$$

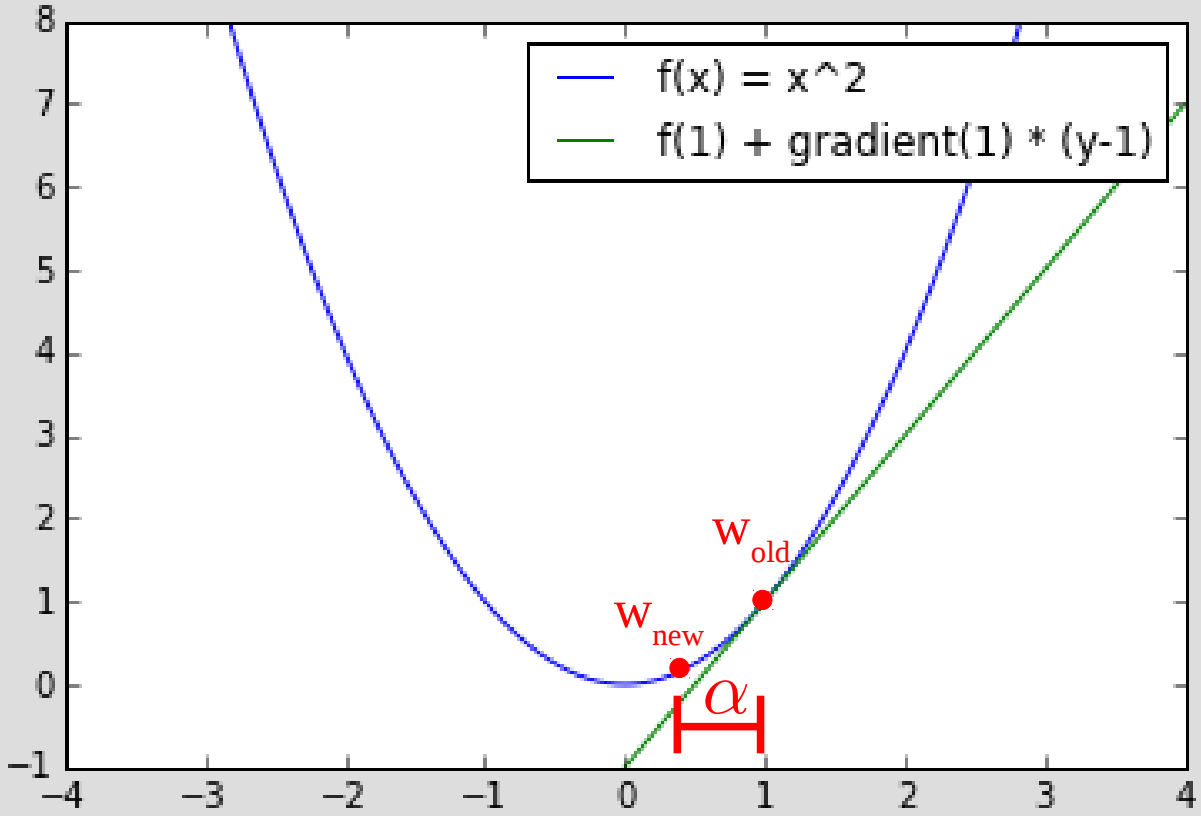
all sum from j=1 to N

$$w_0 = \frac{\sum_j y_j - w_1 \cdot (\sum_j x_j)}{N}$$

Unfortunately, if you want to do polynomials, you might not have a closed form solution like this (i.e. no “easy” exact answer)

Linear Regression: Estimate

You can do a gradient descent (much like Newton's method)(similar to "hill-climbing")



Linear Regression: Estimate

Again, you need calculus to figure out what direction is "down hill", so to move the weights (w_0, w_1, ...) towards the bottom:

$$w_i \leftarrow w_i - \alpha \cdot \frac{\partial}{\partial w_i} Loss(h_{w_i})$$

w_{new} w_{old} w_{old}

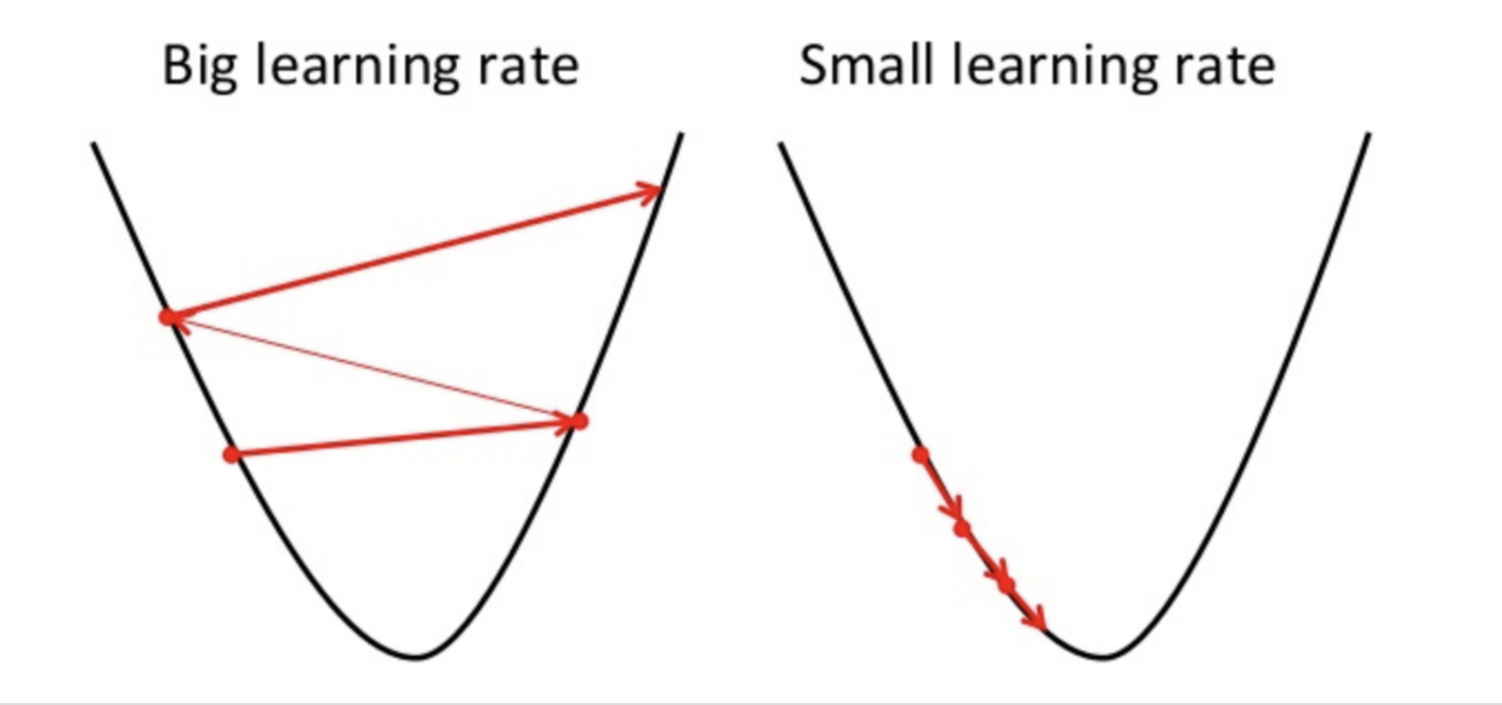
Loss function is what we minimizing (convex), so derivative of it

... where α is basically the "step size" (we will often use alpha in a similar fashion, but call it the "learning factor/rate")

Linear Regression: Estimate

The choice of α is somewhat arbitrary...

You don’t want it too big, but anything small is fine (even better if you shrink it over time)



Linear Regression: Estimate

You can extend this to more than just one variable (or attribute) in a similar fashion

If we X as (for attributes a,b,c ...):

$$X = \begin{bmatrix} 1 & a_1 & b_1 & c_1 & \dots \\ 1 & a_2 & b_2 & c_2 & \dots \\ 1 & a_3 & b_3 & c_3 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

... and w as:

$$w^T = \begin{bmatrix} w_0 & w_1 & w_2 & w_3 & \dots \end{bmatrix}$$

Linear Regression: Estimate

Then if x_j is a single row of X:

$$(x_j)^T = \begin{bmatrix} 1 & a_j & b_j & c_j & \dots \end{bmatrix}$$

Then our "line" is just the dot product:

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

Just like for the single variable case, we update our w's as: $w_i \leftarrow w_i - \alpha \cdot \frac{\partial}{\partial w_i} Loss(h_w)$

attribute for the corresponding weight in example, so if updating "w_2" then "b_j" as in line we do "$w_2 * b_j$"

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

y_j is actual output for example/point number j

Linear Regression: Exact

However, you can solve for linear regression exactly even with multiple inputs

Specifically, you can find optimal weights as:

$$w = (X^T X)^{-1} X^T y$$

This requires you to find a matrix inverse, which can be a bit ugly... but do-able

matrix multiplication

Thus we estimate our line as: $Xw \approx y$

Linear Regression: Overfitting

You actually still can overfit even with a linear approximation by using too many variables (can't overfit "trend")

Another option to minimize (rather than loss):

$$Cost(h_w) = Loss(h_w) + \lambda \cdot Complexity(h_w)$$

as before for line fit

... where we will treat λ as some constant

and: $Complexity(h_w) = L_p(w)$

... where $L_p(w)$ is similar to the p-norm

Side note: “Distance”

The p-norm is a generalized way of measuring distance (you already know some of these)

The definition is of a p-norm:

$$||x||_p = (|x_1|^p + |x_2|^p + ...)^{1/p}$$

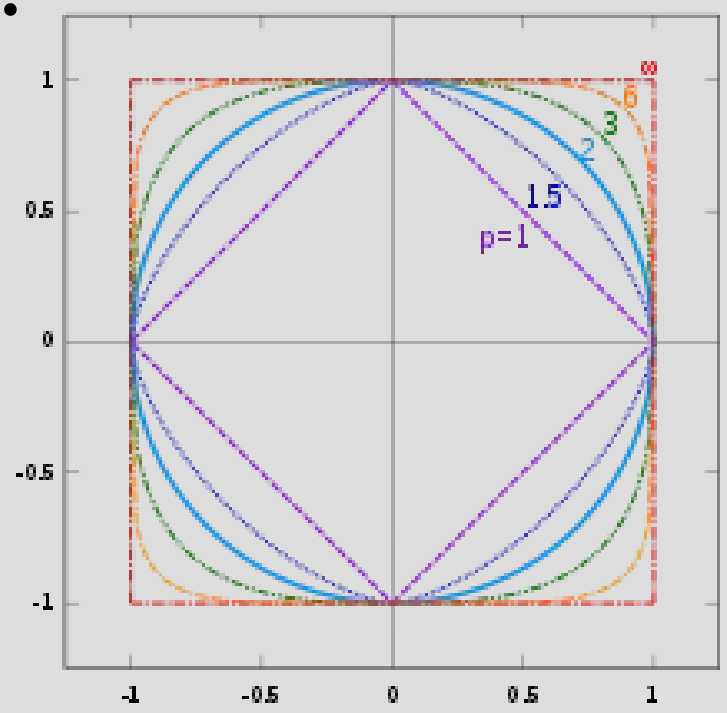
Specifically in 2 dimensions:

$$||(x, y)||_1 = |x| + |y|$$

(Manhattan distance)

$$||(x, y)||_2 = \sqrt{x^2 + y^2}$$

(Euclidean distance)



Linear Regression: Overfitting

We drop the exponent for L's, so in 2D:

$$||(x,y)||_2 = \sqrt{x^2+y^2}$$

$$L_2(x,y) = x^2+y^2$$

So we treat the weight vector's "distance" as the complexity (to minimize)

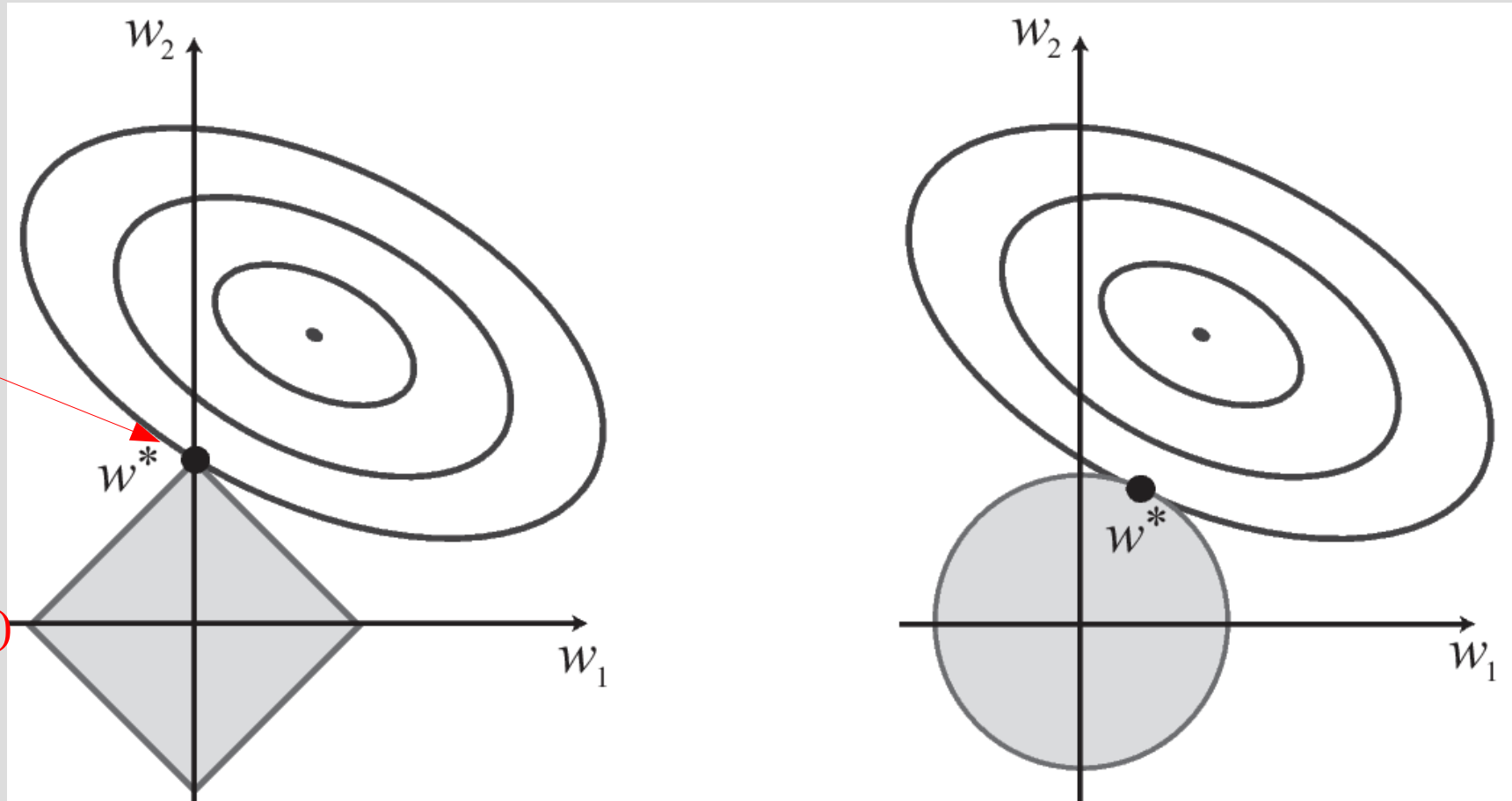
Here L_1 is often the best choice as it tends to have 0's on "irrelevant" attributes/varaibles

... why are 0's good? Why does it happen?

Linear Regression: Overfitting

This is because the L_1 (Manhattan distance) has a sharper "angle" than a circle (L_2)

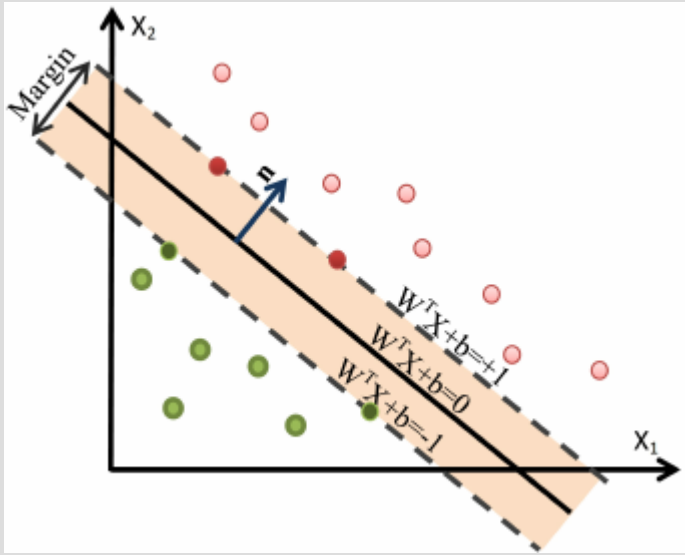
has $w_1 = 0$, as on y-axis ... so w_1 seems irrelevant (less overfit)



Linear Classification

A similar problem is instead of finding the "closest" line, we want to find a line that separates the data points (assume T/F for data)

This is more similar to what we were doing with decision trees, except we will use lines rather than trees



Linear Classification

This is actually a bit harder than linear regression as you can wiggle the line, yet the classification stays the same

This means, most places the derivative are zero, so we cannot do simple gradient descent

To classify we check if:

same as before: line defined by weights

$$h_w(x) = w \cdot x > y$$

... if yes, then guess True... otherwise guess F

Linear Classification

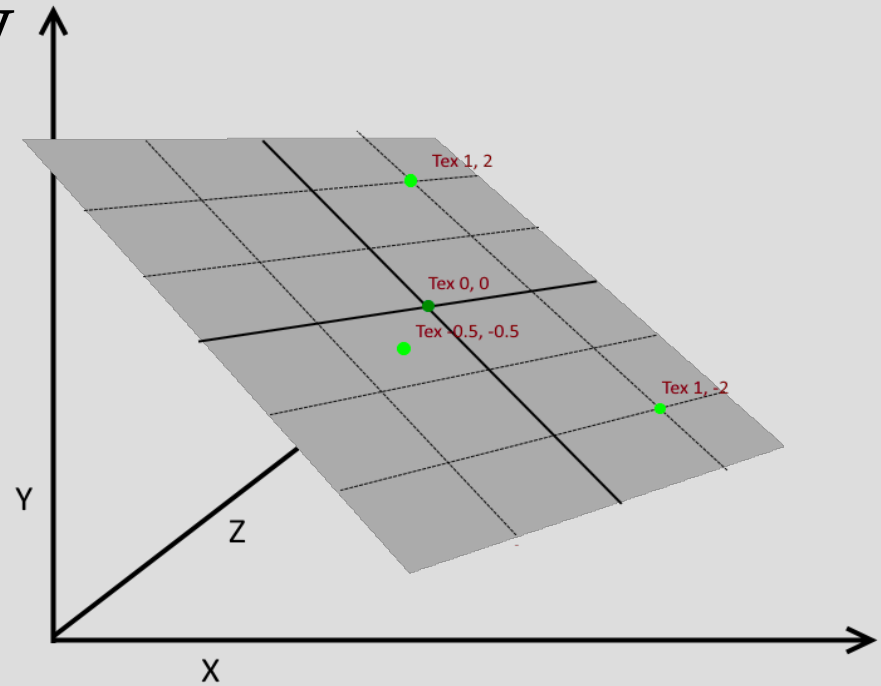
For example, in three dimensions:

$$w_1 x + w_2 \cdot y + w_3 \cdot z > c$$

y is not “output” atm

c = -w_0

This is simply one side of a plane in 3D, so this is trying to classify all possible points using a single plane...



Linear Classification

Despite gradient descent not working, we can still “update” weights until convergence as:

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

Start weight randomly, then update weight for every example with above equation

... what does this equation look like?

Linear Classification

Despite gradient descent not working, we can still “update” weights until convergence as:

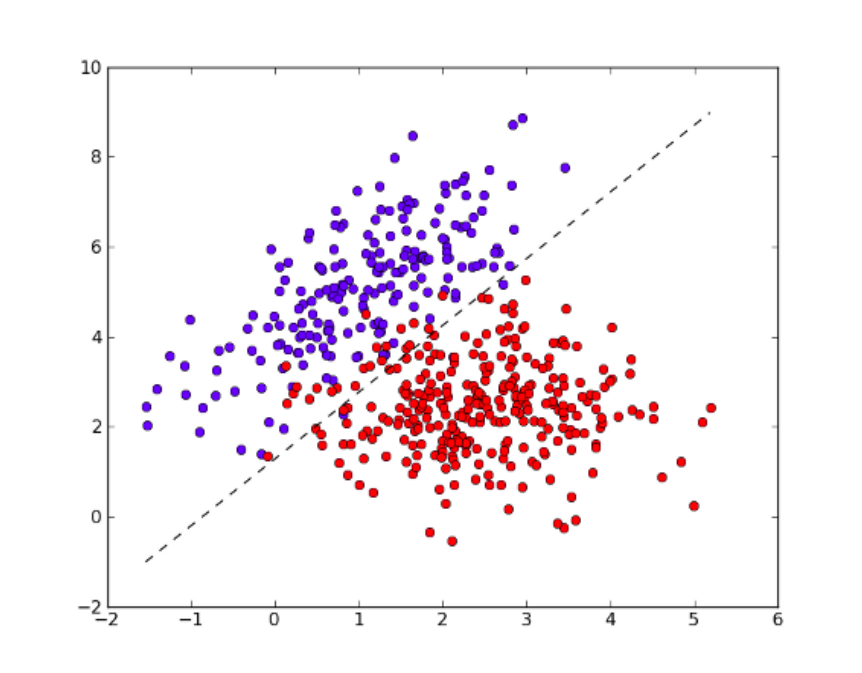
$$w_i \leftarrow w_i + \alpha \cdot (y - h_w(x)) \cdot x_i$$

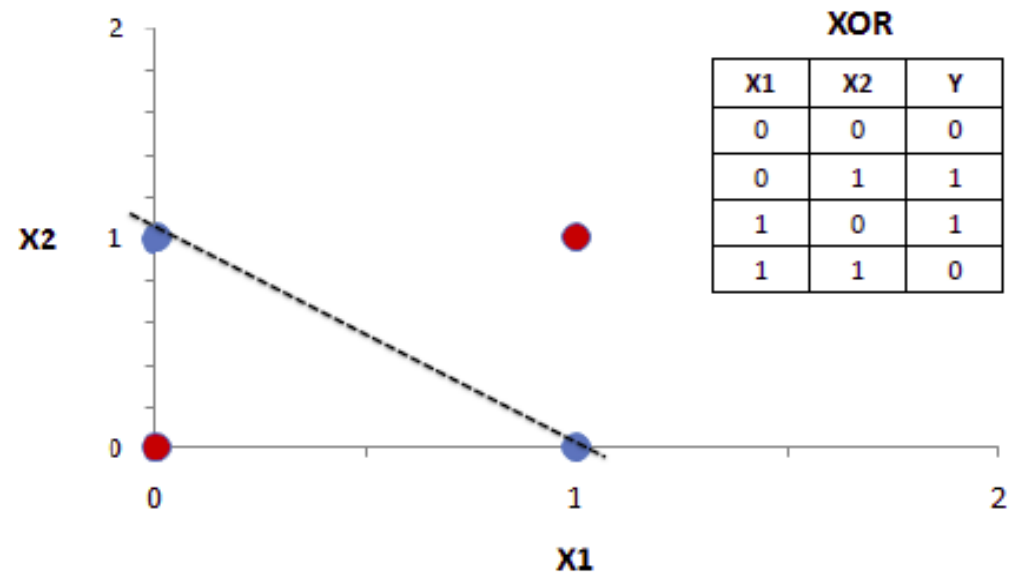
Start weight randomly, then update weight for every example with above equation

... what does this equation look like?
Just the gradient descent (but I thought you said we couldn’t since derivative messed up!)

Linear Classification

If we had only 2 inputs, it would be everything above a line in 2D, but consider XOR on right



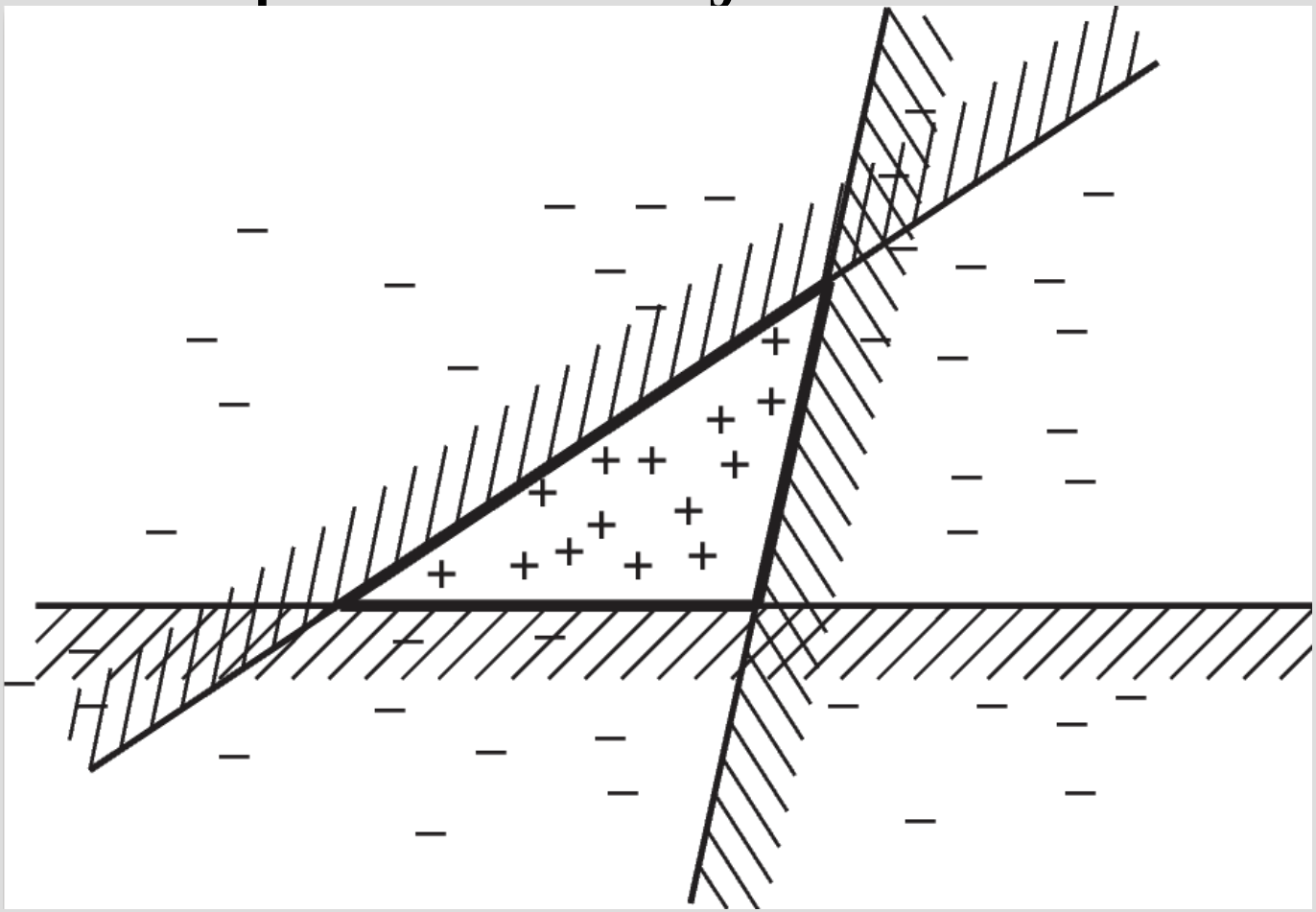


There is no way a single line can classify XOR ... what should we do?

Linear Classification

If one line isn’t enough... use more!

Our next topic will do just this...



Biology: brains

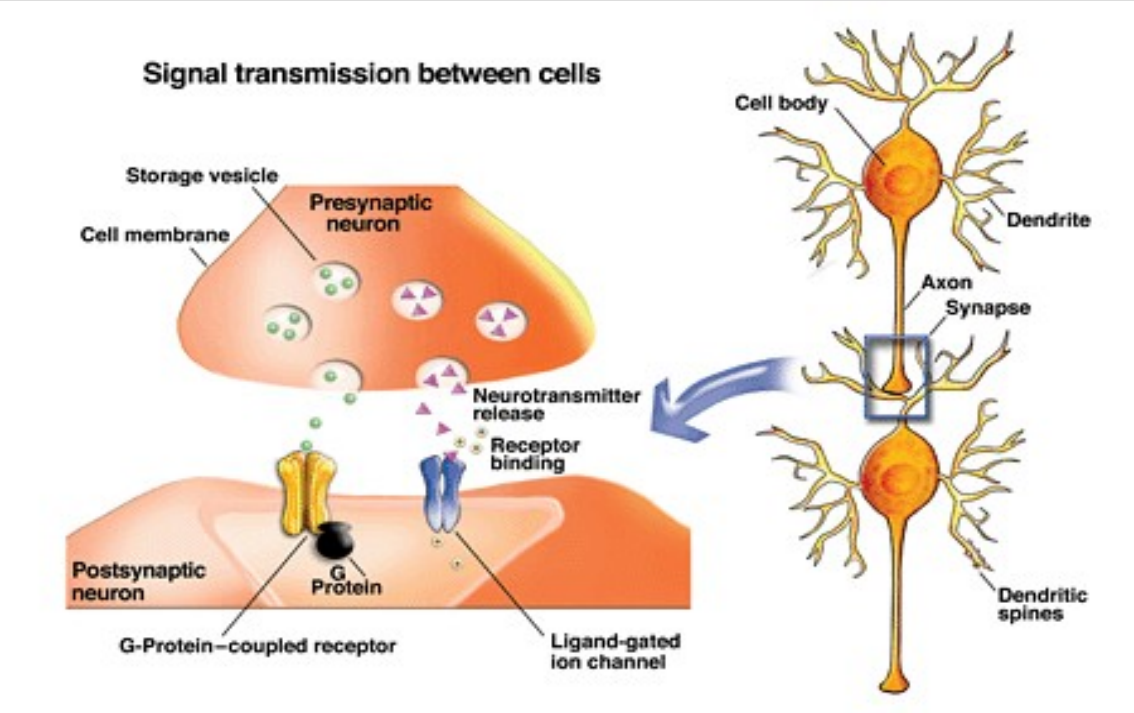
Computer science is fundamentally a creative process: building new & interesting algorithms

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

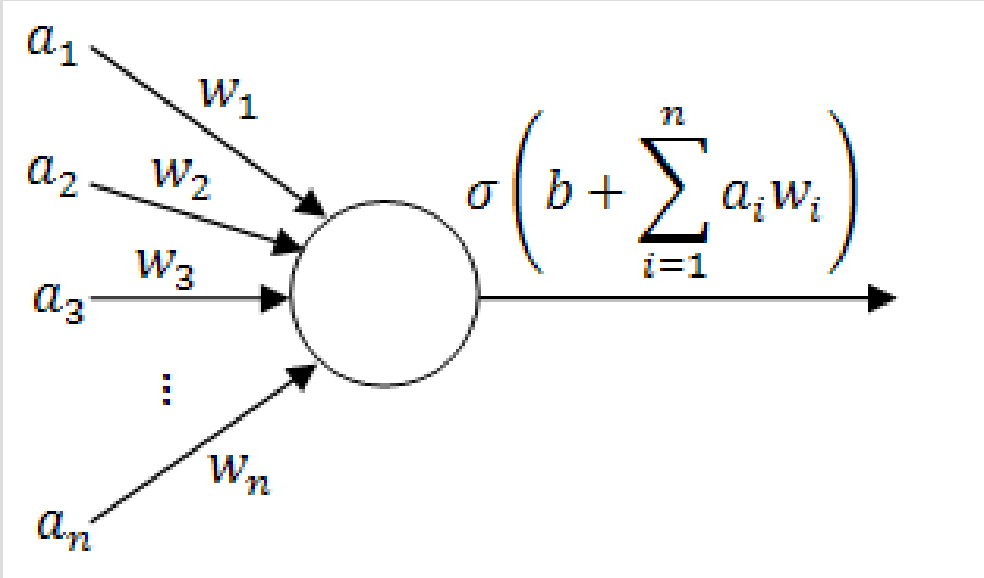
Other sounds might help you go to sleep (my majestic voice?)

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)

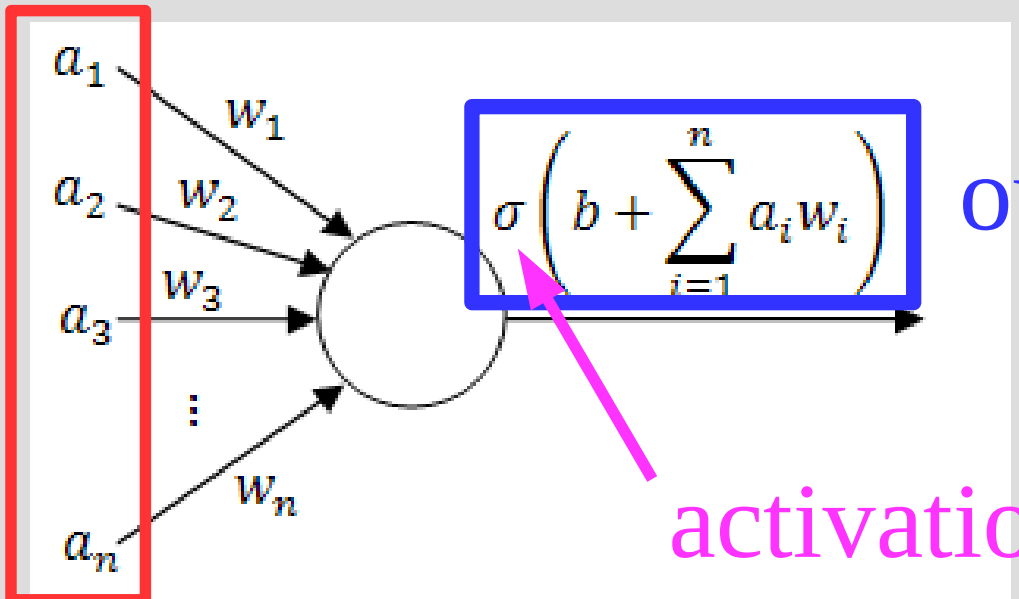
First is an example of a perceptron, the most simple NN; a single node on a single layer



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



inputs

output

activation function

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

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

Perceptrons

Consider the following animals:

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

$2(1) + -2(-1) + 3(1) = 7 > 1$ ... Correct!

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

$2(0.5) + -2(-1) + 3(1) = 6 > 1$ ... 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

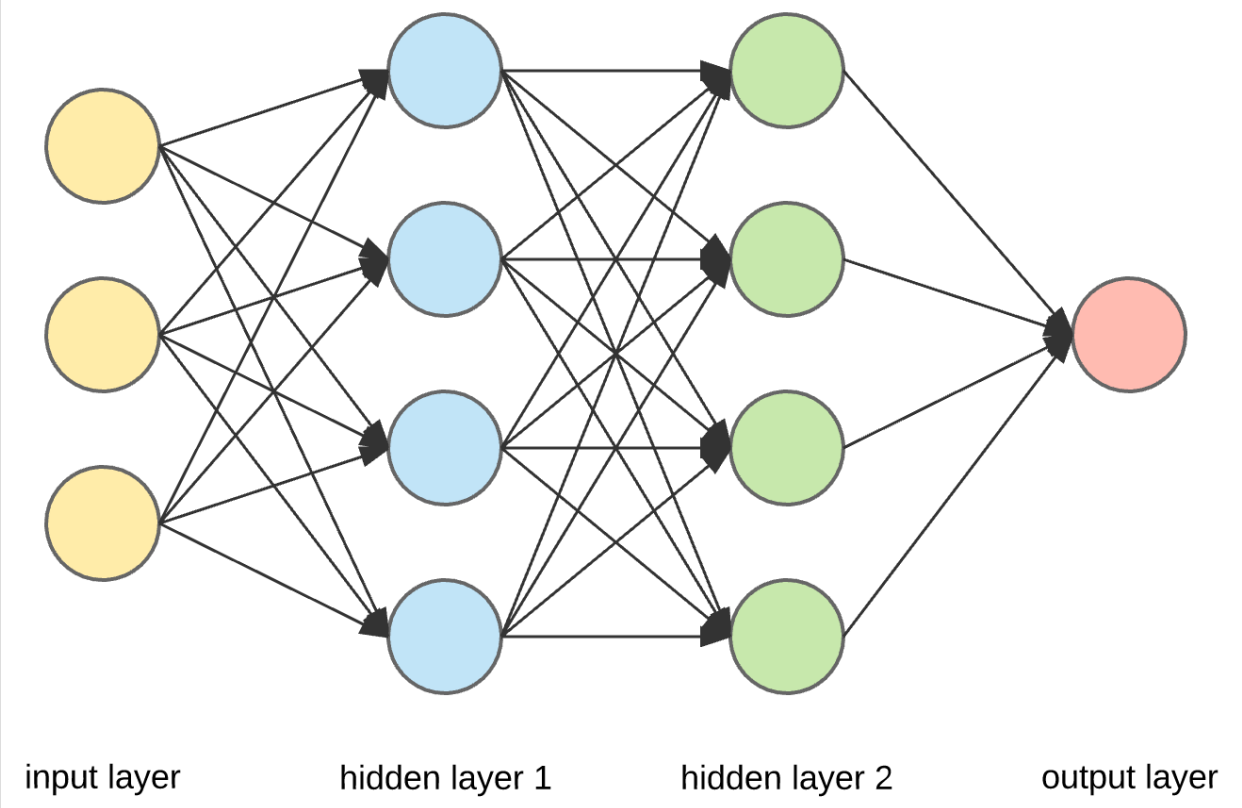
Today we will look at feed-forward NN, where information flows in a single direction

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)

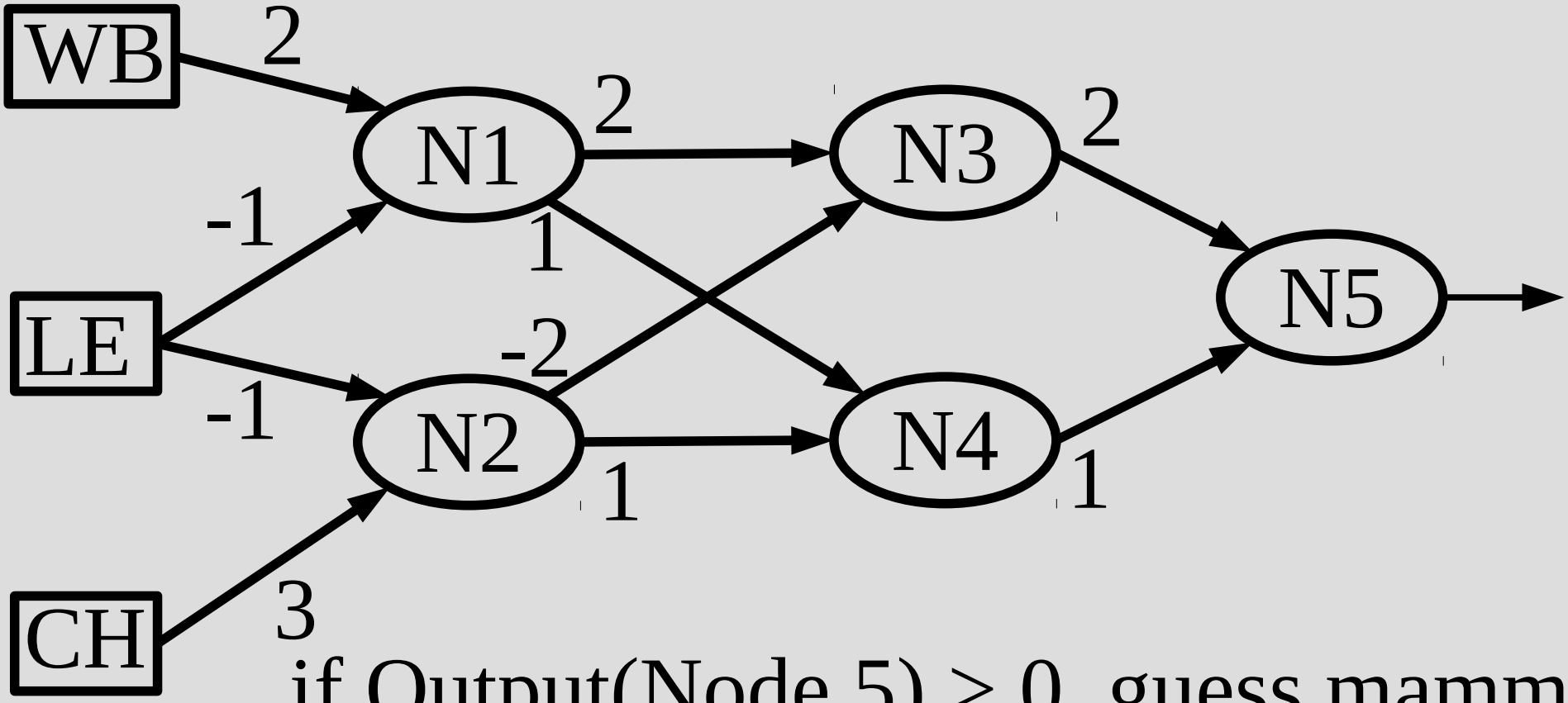
Neural network: feed-forward

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):

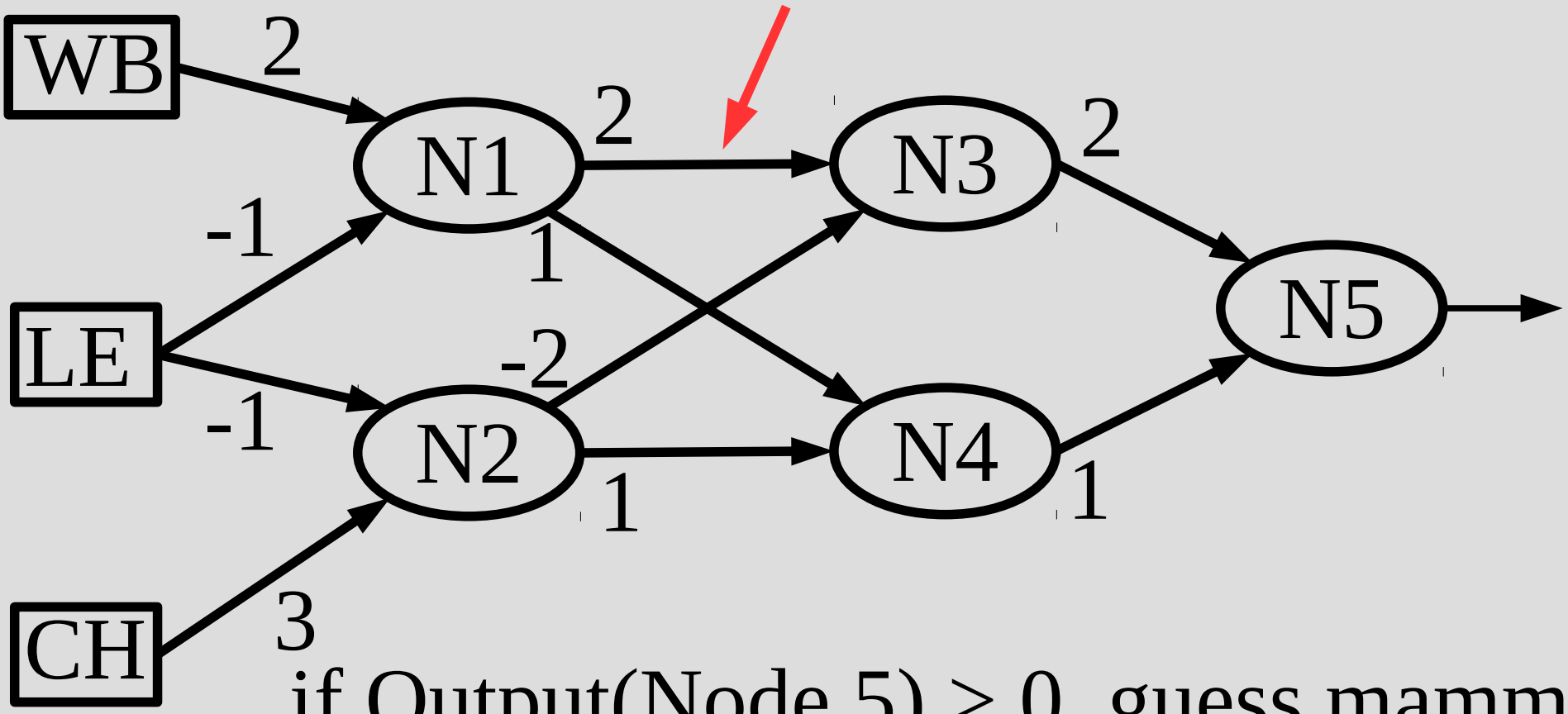


if Output(Node 5) > 0, guess mammal

Neural network: feed-forward

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

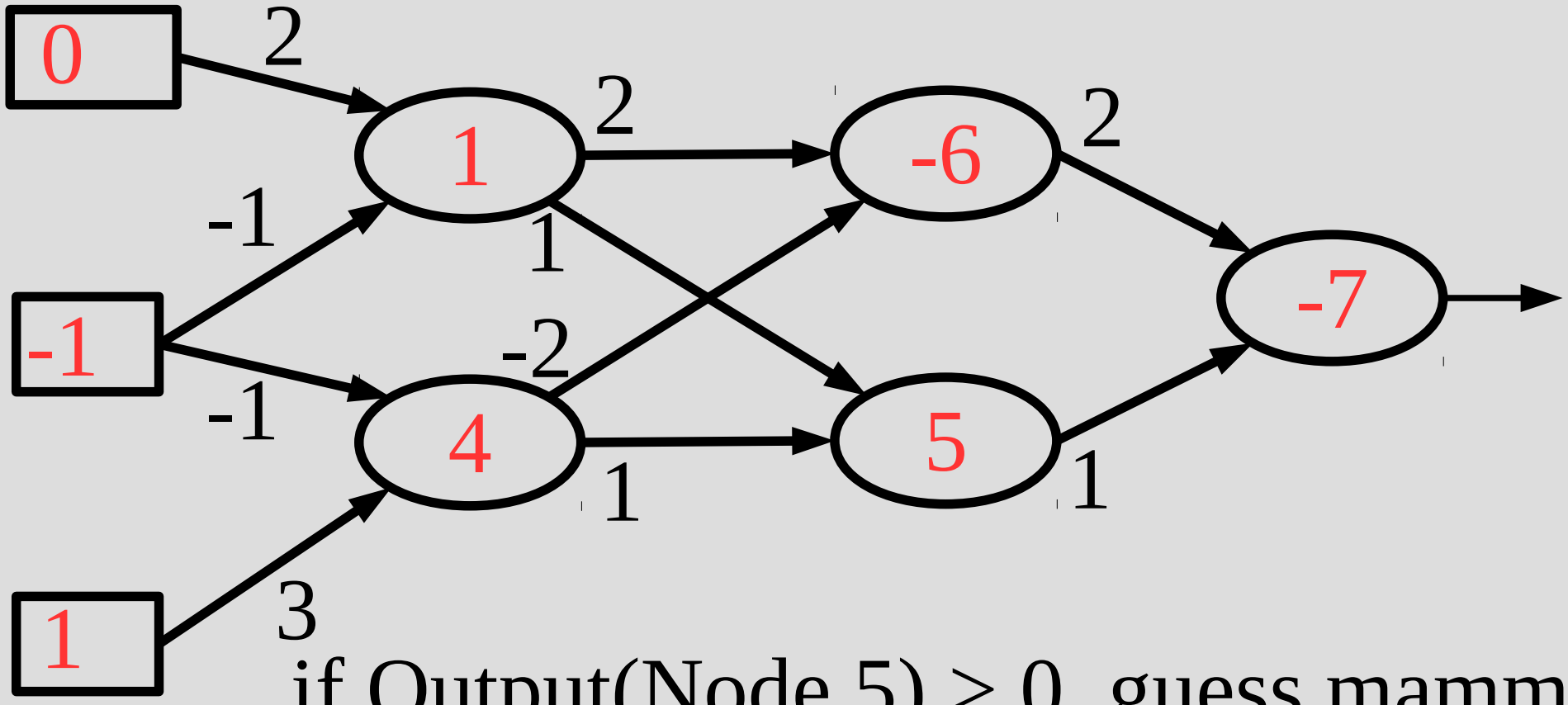
Assume (for now) output = sum input



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

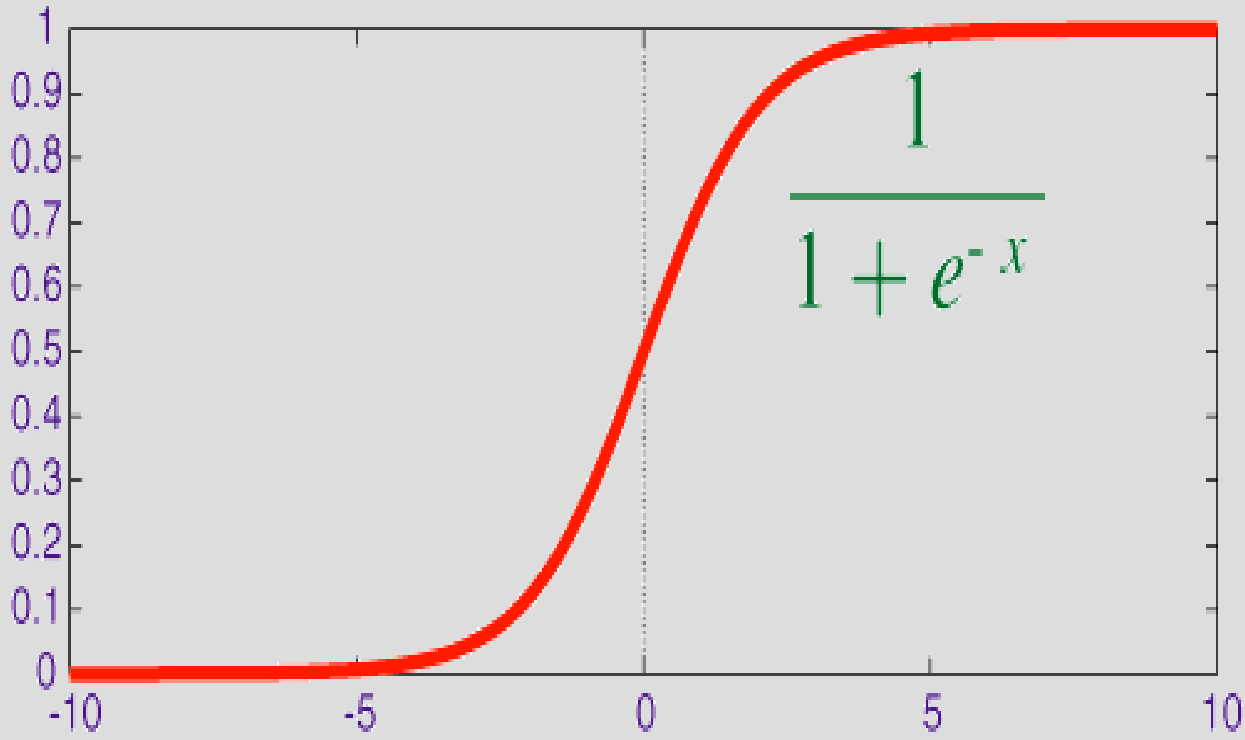
This is because we simply output a linear combination of weights into a linear function (i.e. if f(x) and g(x) are linear... then g(x)+f(x) is also linear)

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:

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

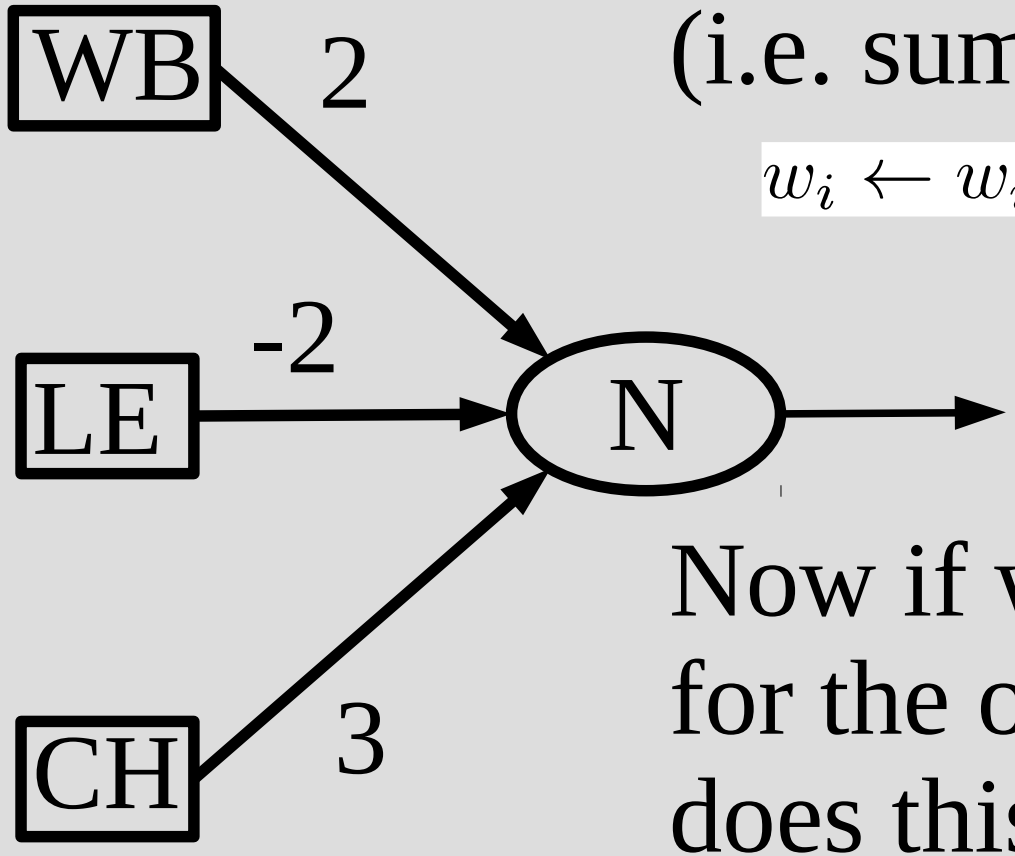
Structure we will ignore (more complex), but there is an automated way to learn weights

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. 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:

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

compare line output

Now we have:

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

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 needed) ...

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

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:

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

$S'(x) = S(x) \cdot (1 - S(x))$

Back-propagation

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

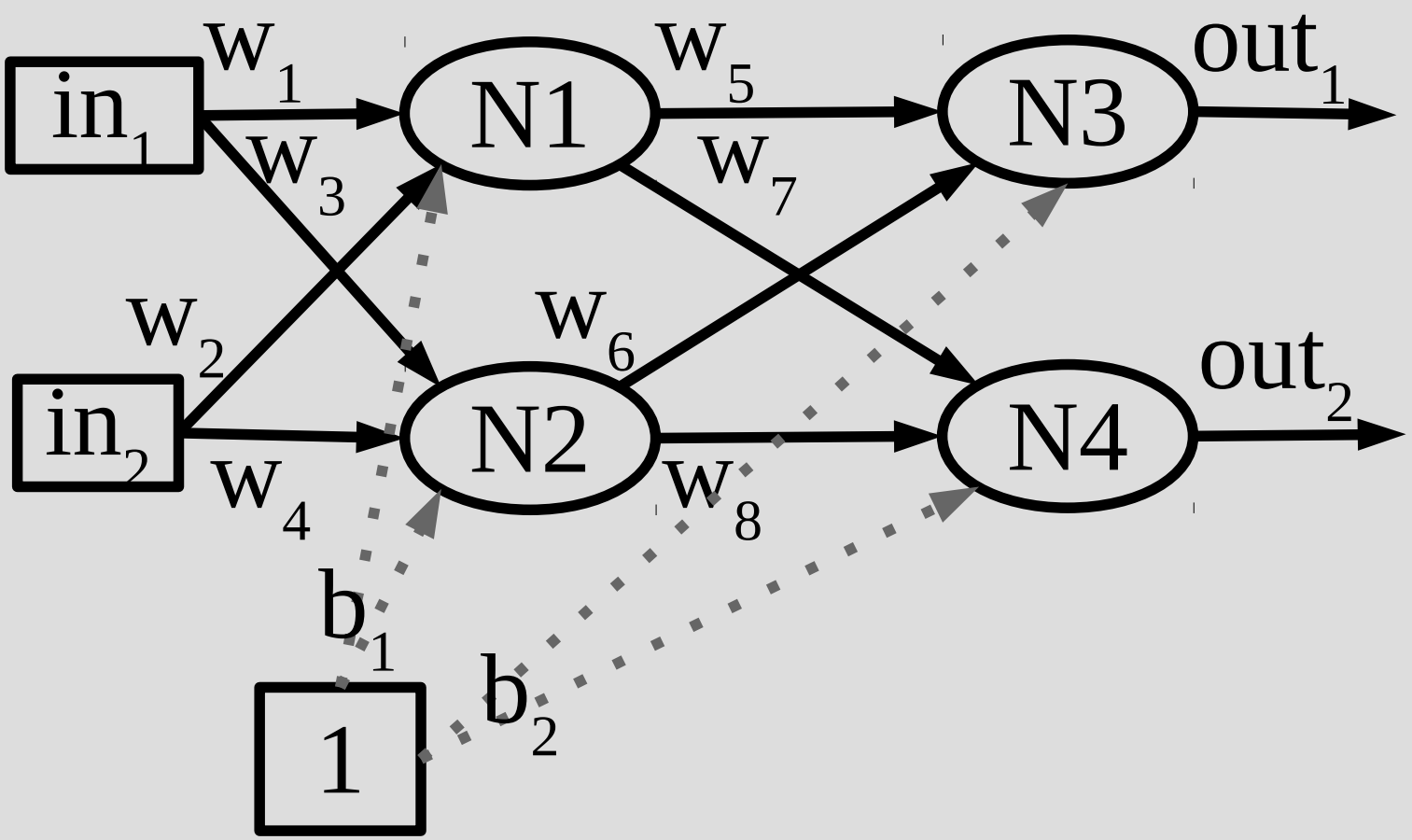
Structure we will ignore (more complex), but there is an automated way to learn weights

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

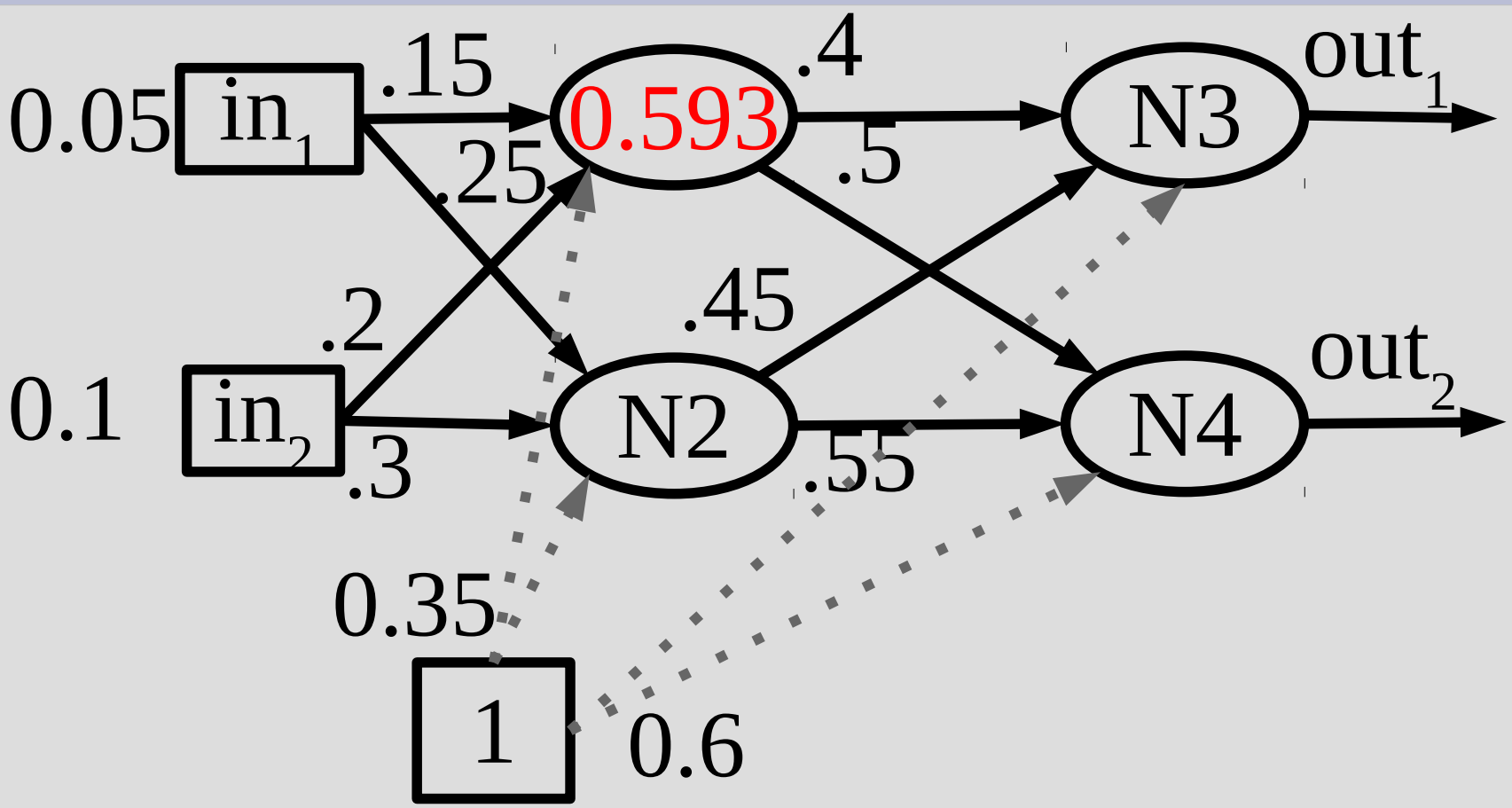
Back-propagation

Consider this example: 4 nodes, 2 layers



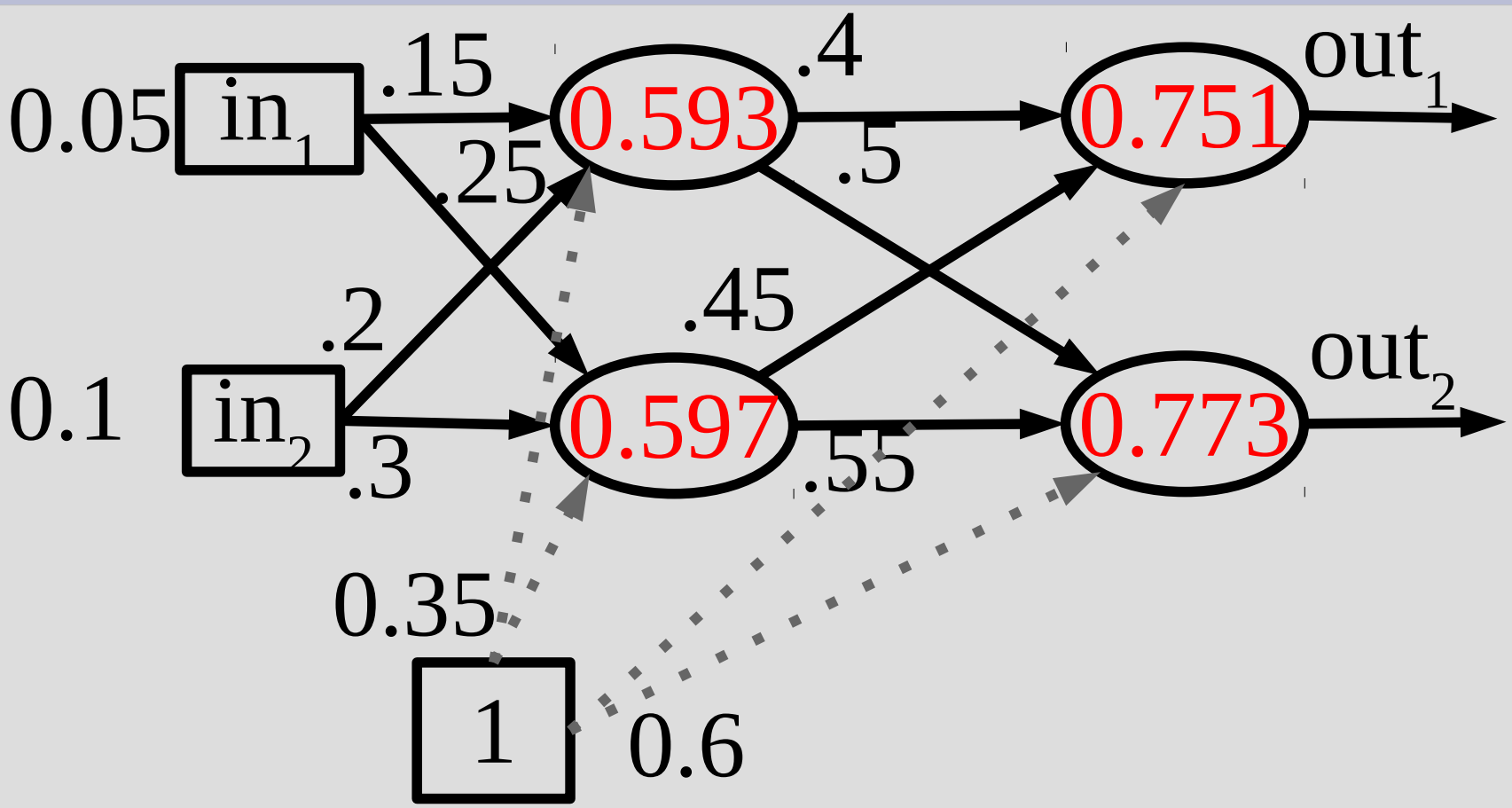
This node as a constant bias of 1

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: $\frac{\sum_i (correct_i - output_i)^2}{2}$
(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: $\frac{\partial Error}{\partial w_5}$
Apply the chain rule:

$$\frac{\partial Error}{\partial out_1} \cdot \frac{\partial S(In(N_3))}{\partial In(N_3)} \cdot \frac{\partial In(N_3)}{\partial w_5}$$

Back-propagation

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

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

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

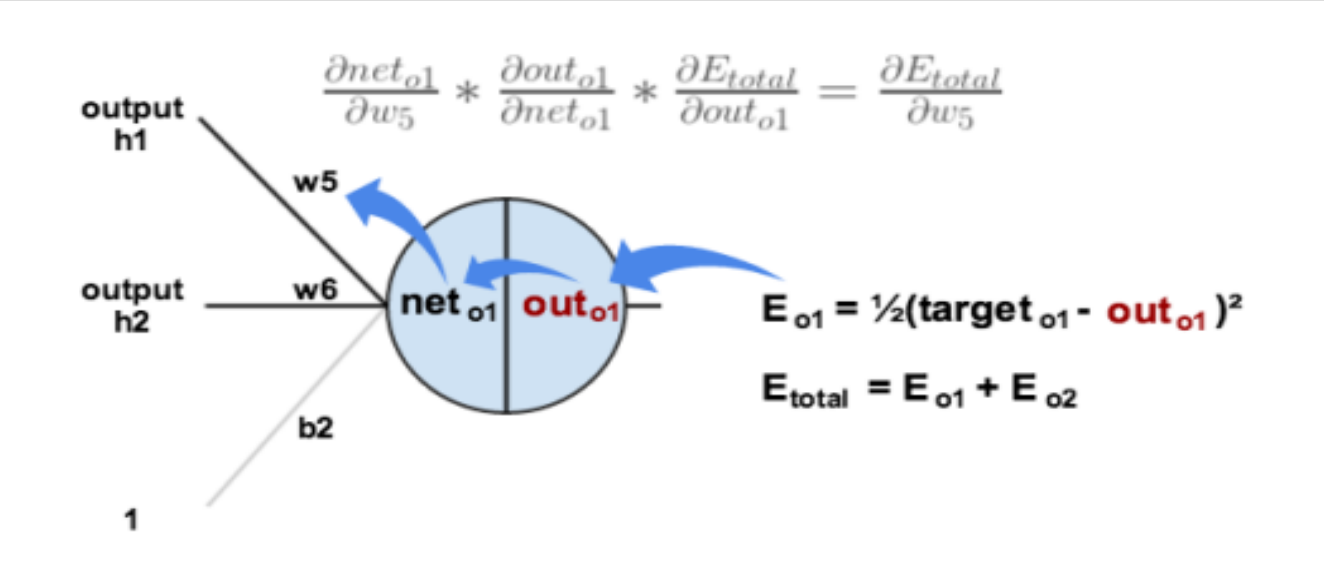
$$\frac{\partial S(In(N_3))}{\partial In(N_3)} = S(In(N_3)) \cdot (1 - S(In(N_3))$$
$$= 0.751 \cdot (1 - 0.751) = 0.187$$

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

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:



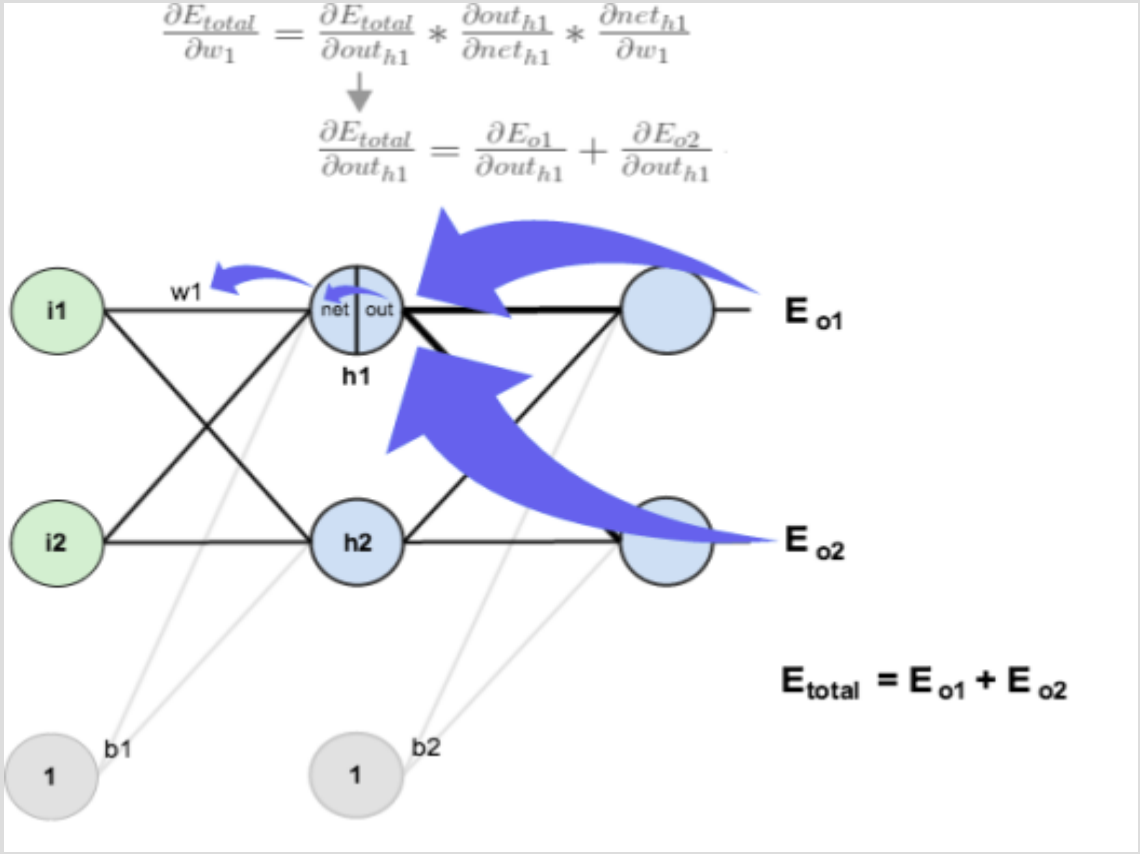
Now that we know w5 is 0.08217 part responsible, we update the weight by:

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

α 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:

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

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

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

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:

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

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

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

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=qv6UVOQ0F44

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=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