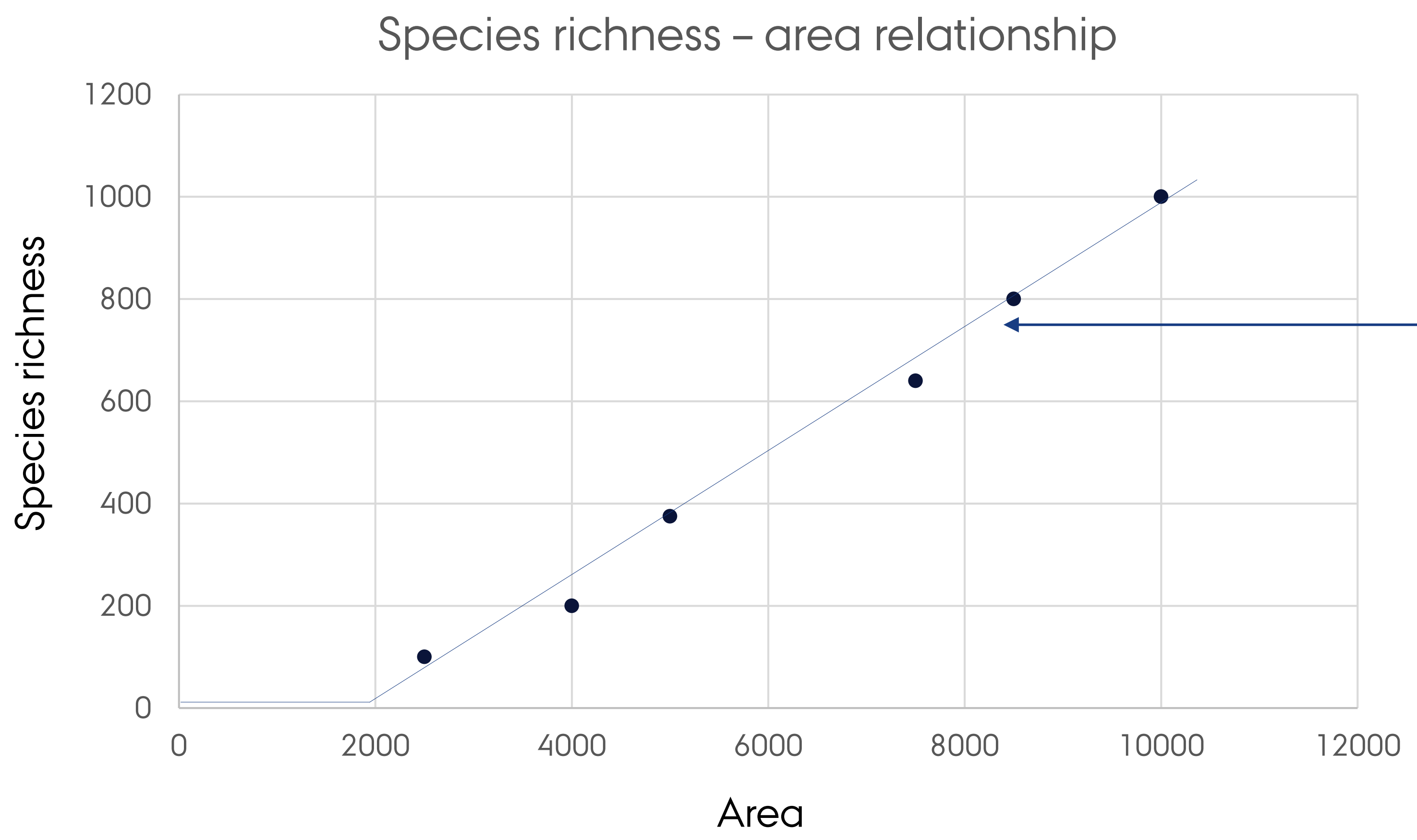
INTRODUCTION TO DEEP NEURAL NETWORKS

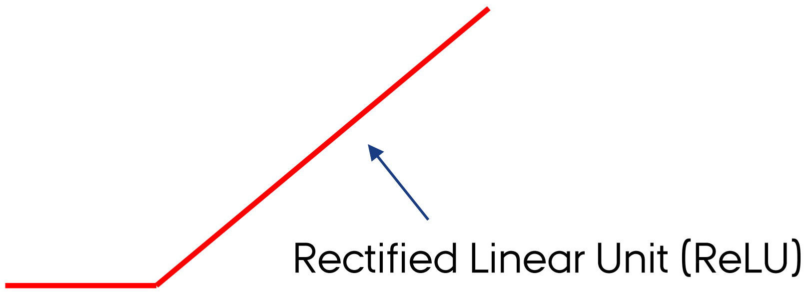
AARHUS
UNIVERSITY
DEPARTMENT OF BIOSCIENCE

MACHINE LEARNING FOR BIOLOGISTS
14 MAY 2019

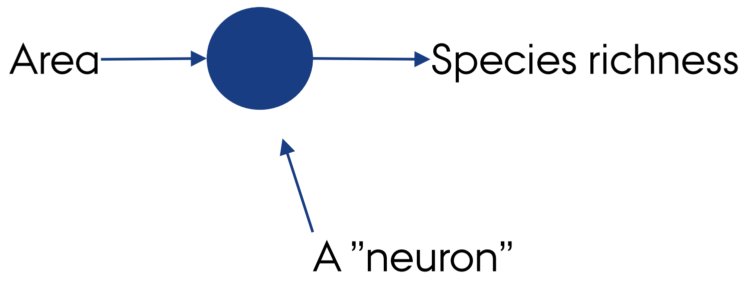
JESPER ERENSKJOLD MOESLUND
RESEARCHER

WHAT IS A NEURAL NETWORK?

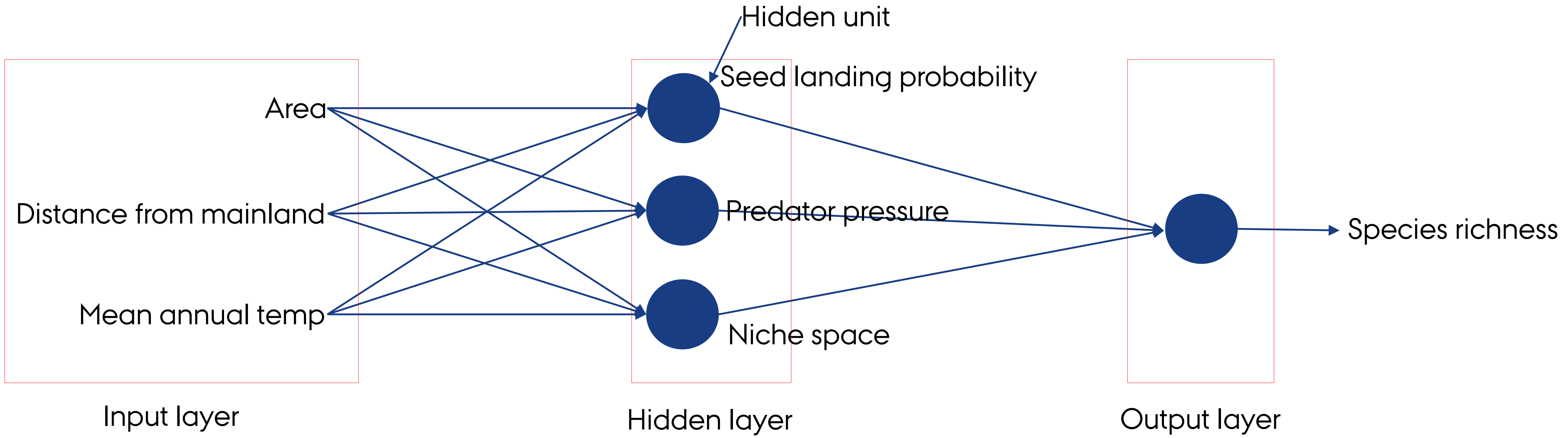




This function can be expressed as a very simple neural network:



WHAT IS A NEURAL NETWORK?



INTRO TO NOTATION AND IMPLEMENTATION

40 pixels

60 pixels

→ NN → Flower or no flower?

How can we input an image to a NN?

Vectors!

In fact each image is stored as matrices; one for each color band (RGB).

Reading these matrices in an a book-like fashion we get:

A vector representation of the image with **$\underline{n_x}$** = 3 x 40 x 60 entrances

n_x = 7200 entrances

$$x = \begin{bmatrix} 25 \\ 254 \\ 87 \\ \ldots. \\ 56 \\ 125 \\ 200 \\ \ldots \\ 96 \\ 13 \\ 177 \\ \ldots \end{bmatrix}$$

INTRO TO NOTATION AND IMPLEMENTATION

One training example can be represented by (x, y)

$x \in R^{n_x \times 1}$

$y \in \{0, 1\}$

$\mathbf{X} \in R^{n_x \times m}$

$y \in R^{1 \times m}$

In python:

x.shape = (nx, m)

y.shape = (1, m)

Say we have m = 5 training examples (i.e., x-vectors)

$$\begin{array}{c} \mathbf{X} = \left[\begin{array}{ccccc} 25 & 55 & 56 & 2 & 76 \\ 254 & 24 & 54 & 154 & 23 \\ 87 & 88 & 89 & 87 & 67 \\ \dots & \dots & \dots & \dots & \dots \\ 56 & 96 & 6 & 156 & 12 \\ 125 & 225 & 15 & 5 & 231 \\ 200 & 100 & 90 & 20 & 154 \\ \dots & \dots & \dots & \dots & \dots \\ 96 & 46 & 96 & 255 & 67 \\ 13 & 3 & 133 & 130 & 79 \\ 177 & 17 & 77 & 56 & 2 \\ \dots & \dots & \dots & \dots & \dots \end{array}\right] n_x \\ \begin{array}{ccccc} x^{(1)} & x^{(2)} & x^{(3)} & x^{(4)} & x^{(5)} \end{array} \\ m \end{array}$$

$y = \{1,0,0,1,0\}$

LOGISTIC REGRESSION

Logistic regression can be thought of as a simple neural network

Given $\vec{x}$ what is $\hat{y}$ (flower or no flower)?

$\hat{y}$ = P(y = 1 | $\vec{x}$)

weights

this is what we typically consider the intersection

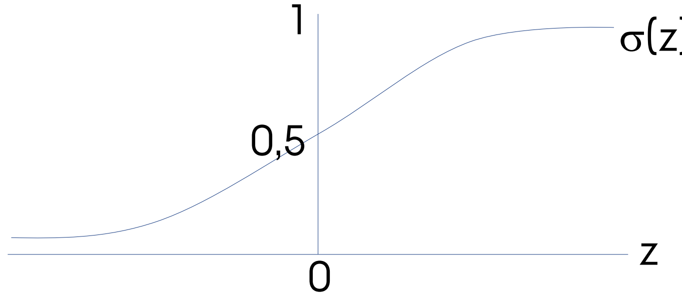
In logistic regression we have two parameters, lets call them $\vec{w}$ ($\in R^{n_x \times 1}$) and b ($\in$ R)

In linear regression we would just do:

$\cancel{\hat{y} = \vec{w}^{\mathsf{T}}\vec{x} + b}$

Instead:

$\hat{y} = \sigma(\underbrace{\vec{w}^{\mathsf{T}}\vec{x} + b}_{z})$, σ is the logistic or sigmoid function



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

LOGISTIC REGRESSION

For each training example i, we want to find the $\vec{w}^{(i)}$ and $b^{(i)}$ causing our $\hat{y}^{(i)}$ to be as close to $y^{(i)}$ (the "truth") as possible

How is this optimization undertaken? We need a **loss-function**!

Most simple would be to use $\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$ but this may be non-convex

Instead: $\mathcal{L}(\hat{y}, y) = -(y*\log(\hat{y}) + (1-y)\log(1-\hat{y}))$

We want this function as small as possible. What if y = 1? y = 0?

How to optimize across the whole training set?

Cost function:

$$\mathcal{J}(\vec{w}, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

GRADIENT DESCENT

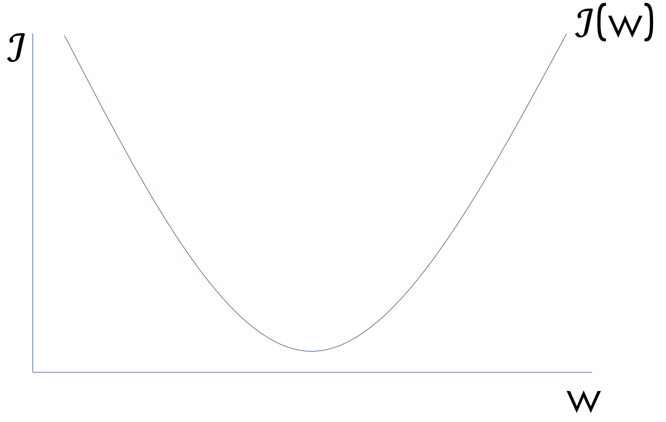
How to find the $\vec{w}$ and b that minimize $\mathcal{J}(\vec{w}, \mathrm{b}) = \frac{1}{m}\sum_{i=1}^{m} -(\mathrm{y}^{(\mathrm{i})} * \log(\hat{y}^{(\mathrm{i})}) + (1-\mathrm{y}^{(\mathrm{i})})\log(1-\hat{y}^{(\mathrm{i})}))$?

For illustration purposes let's say that w is just a real number not a vector

Gradient descent: we iteratively take one step downhill in the steepest direction

Clearly we have problems if the function is not convex!

GRADIENT DESCENT MORE FORMALLY



If we wish to find the most optimal w we update w in steps:

w = w – $\alpha \frac{d\mathcal{J}(w)}{dw}$, where α is the learning rate (step-length)

In logistic regression the cost-function is a function of both w and b:

w = w - $\alpha \frac{\partial\mathcal{J}(w,b)}{\partial w}$

b = b - $\alpha \frac{\partial\mathcal{J}(w,b)}{\partial b}$

LOGISTIC REGRESSION GRADIENT DESCENT

Let's go back a few slides and consider the case where we only have **one training example**

Let's also say that in this example we only have two features (explanatory factors):

x_1

w_1

x_2

w_2

b

$$z = w_1x_1 + w_2x_2 + b \longrightarrow \hat{y} = \sigma(z) \longrightarrow \mathcal{L}(\hat{y}, y) = -(y * \log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

$$\frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{d\hat{y}}\frac{d\hat{y}}{dz} = \hat{y} - y \qquad \frac{d\hat{y}}{dz} = \hat{y}(1-\hat{y}) \qquad \frac{d\mathcal{L}}{d\hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$$

$$\boxed{\begin{aligned} dw_1 &= x_1 dz \\ dw_2 &= x_2 dz \\ db &= dz \end{aligned}}$$

Now we can do gradient descent with logistic regression!

We want dw_1, dw_2 and db, but neither w_1, w_2 nor b is part of the loss function. Hence we need calculus!

GRADIENT DESCENT ON MULTIPLE TRAINING SETS

Now we consider the case where we have **m training sets**

$$\mathcal{J}(w, b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

We now how to calculate $dw_1^{(i)}$, $dw_2^{(i)}$ and $db^{(i)}$ for one training set $(x^{(i)}, y^{(i)})$

We want:

Average of this

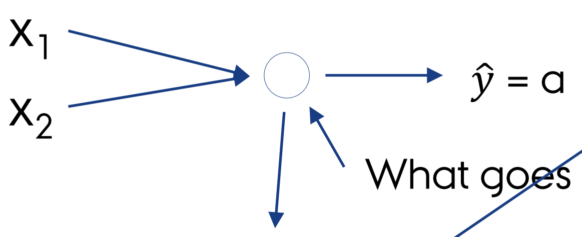
$$\frac{\partial \mathcal{J}(w, b)}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial \mathcal{L}(\hat{y}^{(i)}, y^{(i)})}{\partial w_1}$$

In traditional programming one would typically calculate this mean in a for loop

In modern machine learning software all this is done for you and it's vectorized to optimize performance!

Pulsatilla vulgaris

FROM LOGISTIC REGRESSION TO NEURAL NETWORK

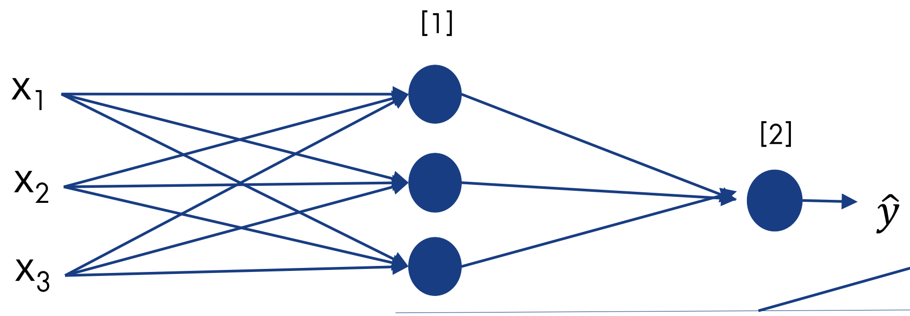


Now vectors

What goes on here in logistic regression?

$$z = w^T x + b \longrightarrow \hat{y} = \sigma(z) = a \longrightarrow \mathcal{L}(\hat{y}, y)$$

In a neural network:



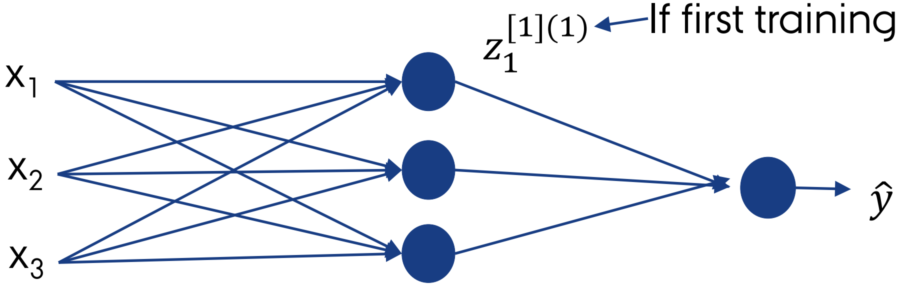
So in neural networks we just do this multiple times…

$$z^{[1]} = W^{[1]}x + b^{[1]} \longrightarrow a^{[1]} = \sigma(z^{[1]}) \longrightarrow z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \longrightarrow a^{[2]} = \sigma(z^{[2]}) = \hat{y} \longrightarrow \mathcal{L}(\hat{y}, y)$$

$dW^{[1]} + db^{[1]}$ $\longleftarrow$ $dz^{[2]}$ $\longleftarrow$ $da^{[2]}$ $\longleftarrow$

REPRESENTATION AND NOTATION

Input layer [0] Hidden layer [1....] Output layer [depth]



If first training example

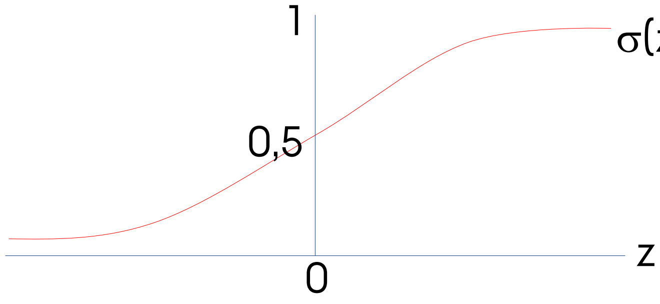
2 layer neural network

One training example: $z^{[1]} = W^{[1]}x + b^{[1]} \longrightarrow a^{[1]} = \sigma(z^{[1]}) \longrightarrow z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \longrightarrow a^{[2]} = \sigma(z^{[2]})$

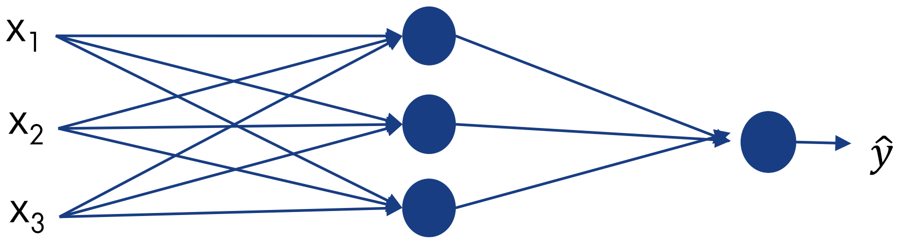
Multiple training examples: $Z^{[1]} = W^{[1]}X + b^{[1]} \longrightarrow A^{[1]} = \sigma(Z^{[1]}) \longrightarrow Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \longrightarrow A^{[2]} = \sigma(Z^{[2]})$

Now each training example is represented as a column in a matrix called X

ACTIVATION FUNCTIONS



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



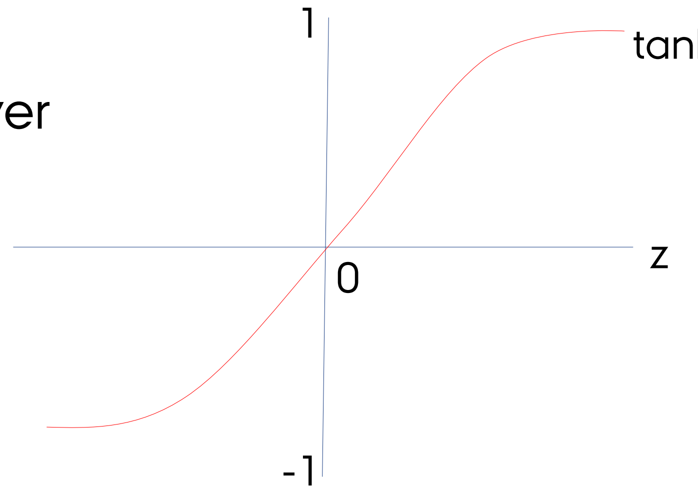
$Z^{[1]} = W^{[1]}X + b^{[1]} \longrightarrow A^{[1]} = \sigma(Z^{[1]}) \longrightarrow Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \longrightarrow A^{[2]} = \sigma(Z^{[2]})$

We've till now only considered the logistic or sigmoid function

Not the best choice! Only for binary classification in output layer

tanh is almost always better, helps learning faster

Problem when z is very high or low, slow learning



$\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

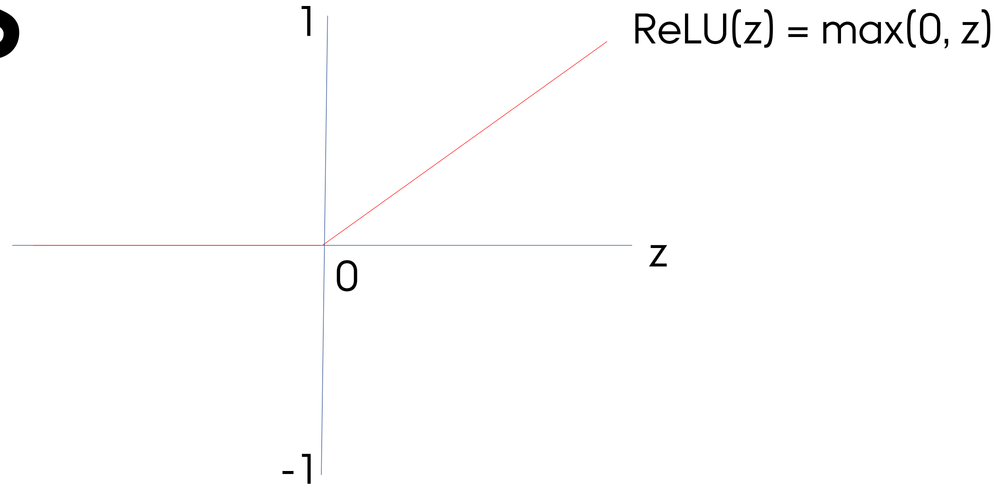
ACTIVATION FUNCTIONS

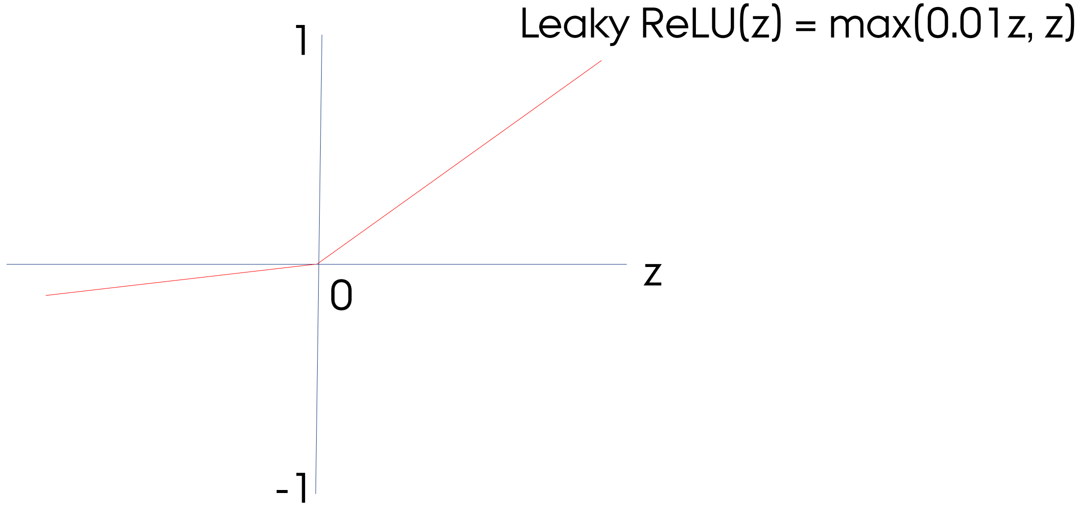
ReLU is therefore used even more often

Leaky ReLU also a posibility

Often just use ReLU but if in doubt try out the different functions

More generally, instead of $\sigma(z)$: $\mathbf{g(z)}$, where g is a non-linear function





ACTIVATION FUNCTIONS

Why does the activation function have to be non-linear?

$$Z^{[1]} = W^{[1]}X + b^{[1]} \longrightarrow A^{[1]} = Z^{[1]} \longrightarrow Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \longrightarrow A^{[2]} = Z^{[2]}$$

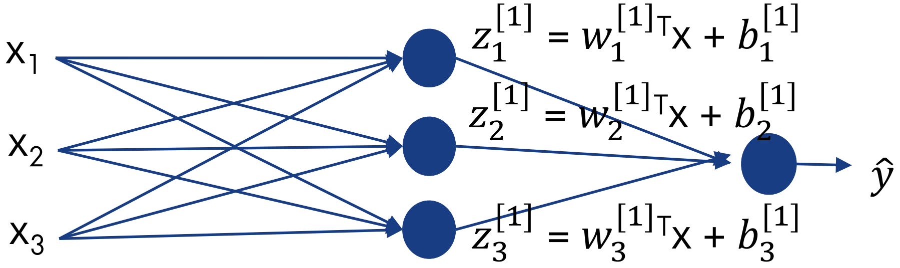
Otherwise it's just a linear combination of input and not better than a standard GLM

But, if you predict real numbers, you could have a linear function in output layer to allow for negative and positive real number output

Non-linear activation functions is critical in NNs!

INITIALIZATION

In logistic regression it would work if you initialize your weights (w) to all zero, but not in neural networks:



Now if $w_1^{[1]} = w_2^{[1]} = w_3^{[1]} = \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$, then all hidden units will be equal (symmetric) too since the x vector is the same in all units!

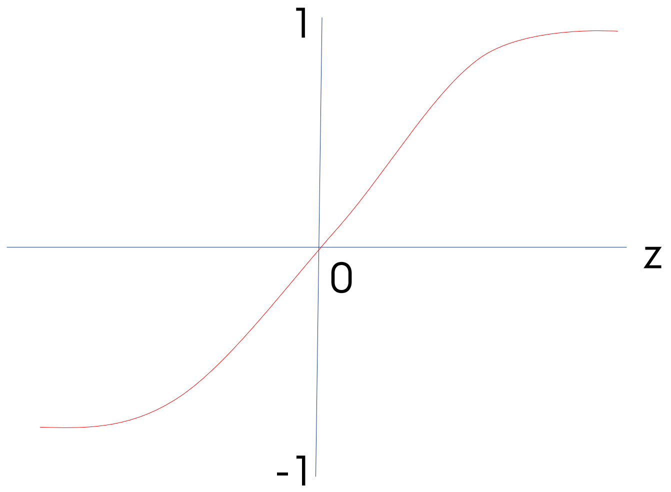
Then also all dz's will be equal and and the w's will keep being equal no matter how many gradient descents are performed

No point in having more than one hidden unit then

INITIALIZATION

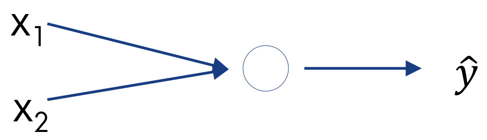
Then what to do?

Pick random numbers multiplied by e.g. 0.01 or some small number



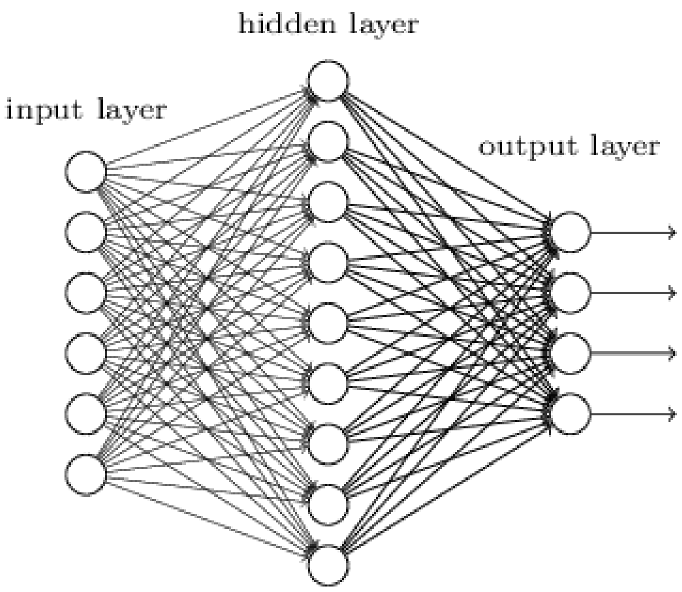
Why not a big number like 1000?

DEEP NEURAL NETWORKS

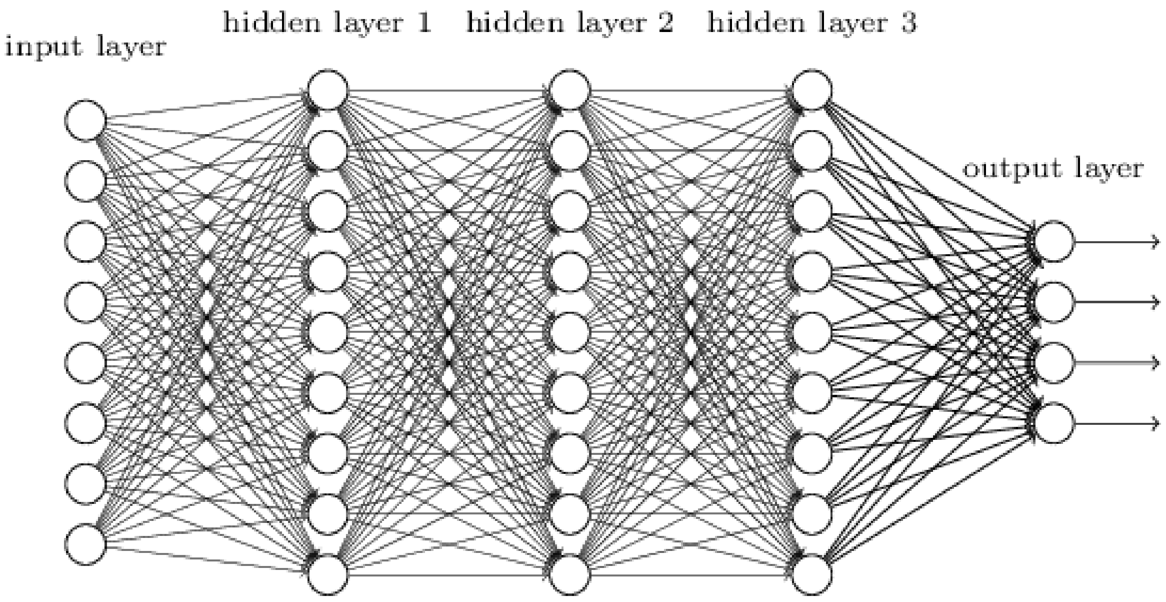


Shallow neural network

Non-deep neural network



Deep neural network

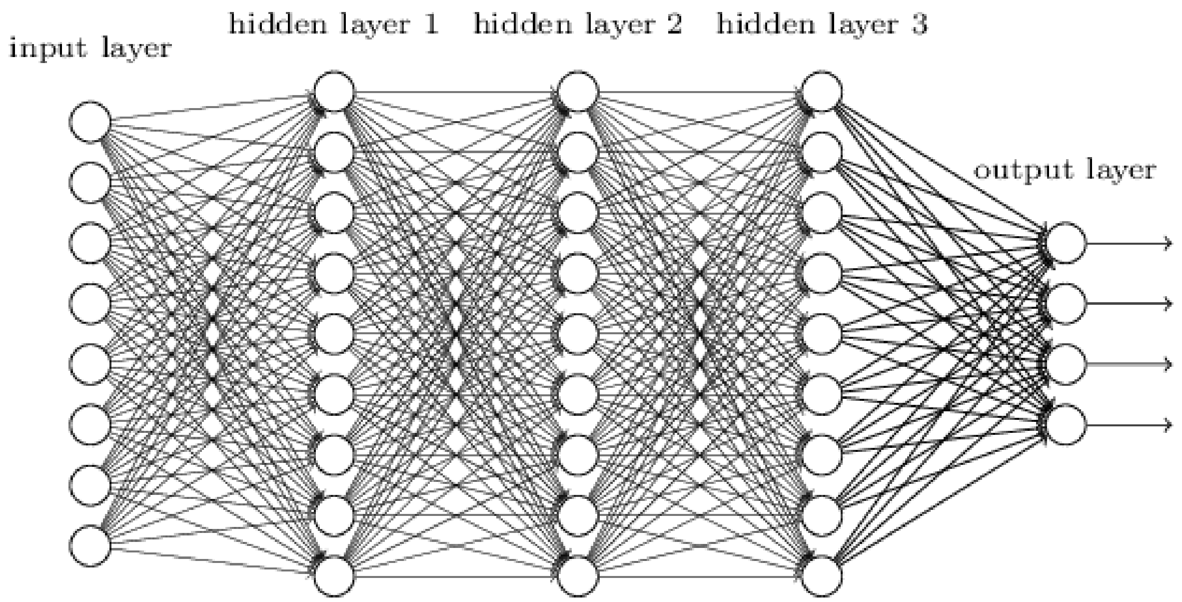


DEEP NEURAL NETWORKS NOTATION

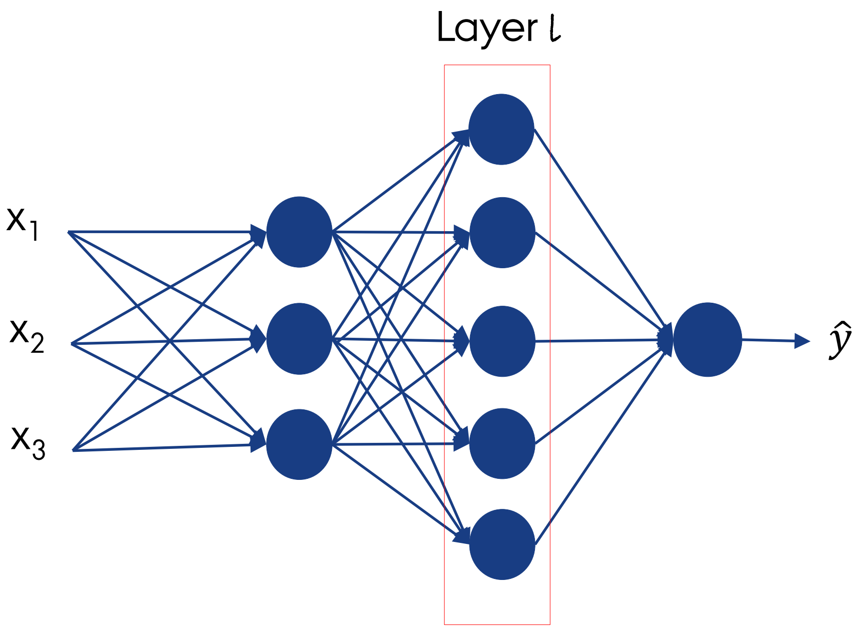
L = 4 (number of layers)

$n^{[l]}$ = number of nodes in layer l

$n^{[0]} = n_x = 8$ $n^{[1]} = 9$



FORWARD PROPAGATION IN DEEP NNS



$$a^{[l-1]} \longrightarrow z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$$

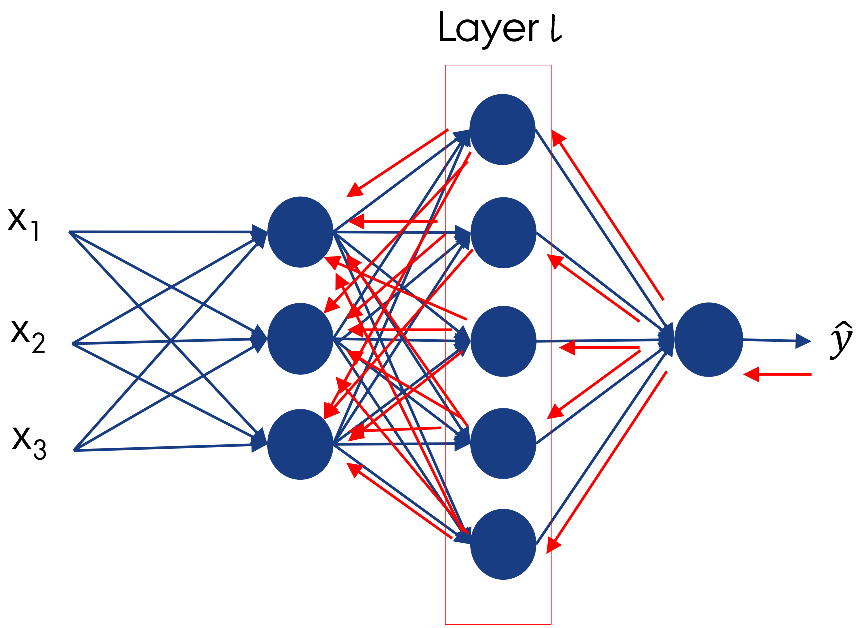
$$a^{[l]} = g(z^{[l]})$$

$$A^{[l-1]} \longrightarrow Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

$$A^{[l]} = g(Z^{[l]})$$

Then move on to next layers

BACKWARD PROPAGATION IN DEEP NNS



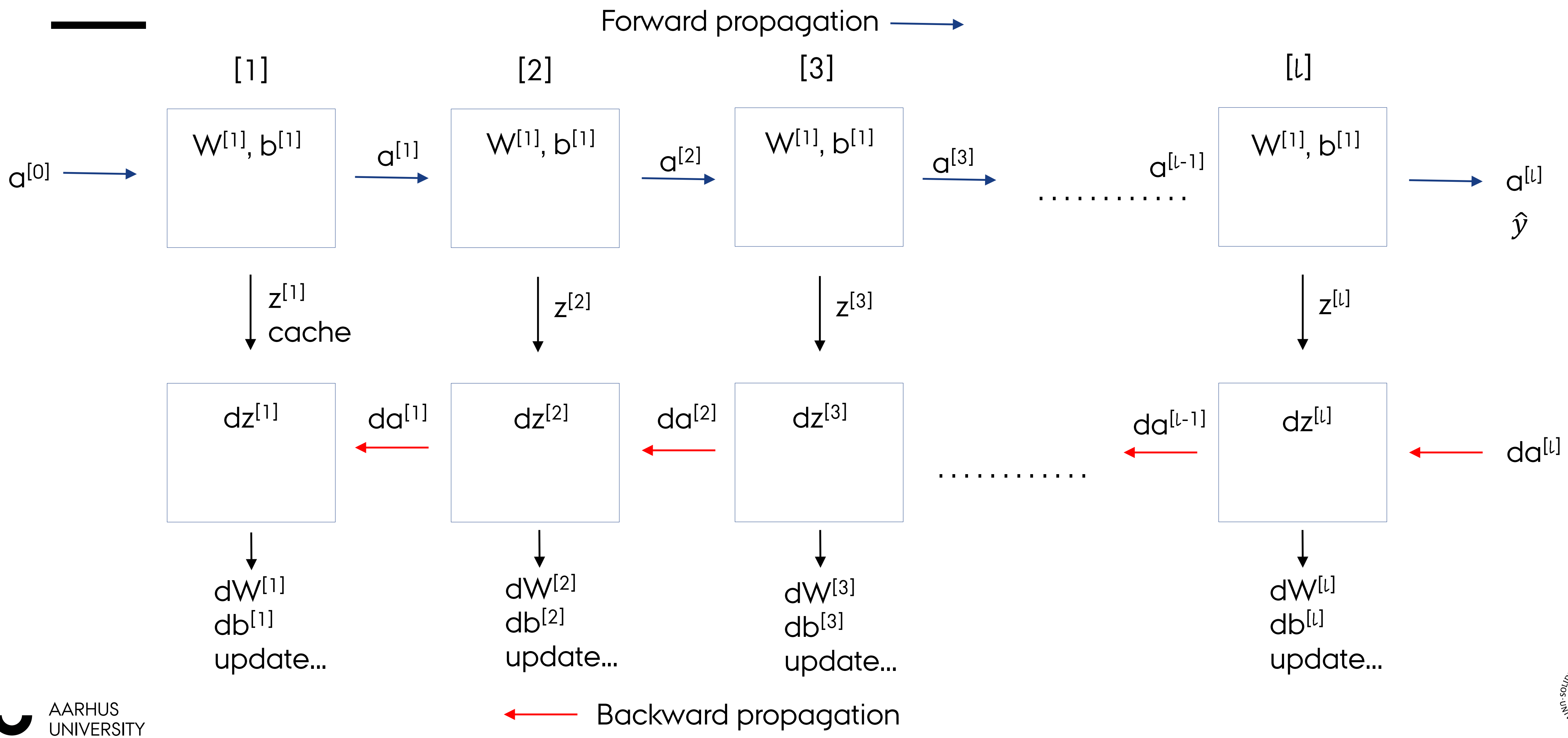
Then move on to previous layers ← $da^{[l-1]}$ ← $da^{[l]}$

$dz^{[l]}$

$dW^{[l]}$ → $W^{[l]}$ and $b^{[l]}$ gets updated

$db^{[l]}$

GRADIENT DESCENT IN NNS



AARHUS
UNIVERSITY