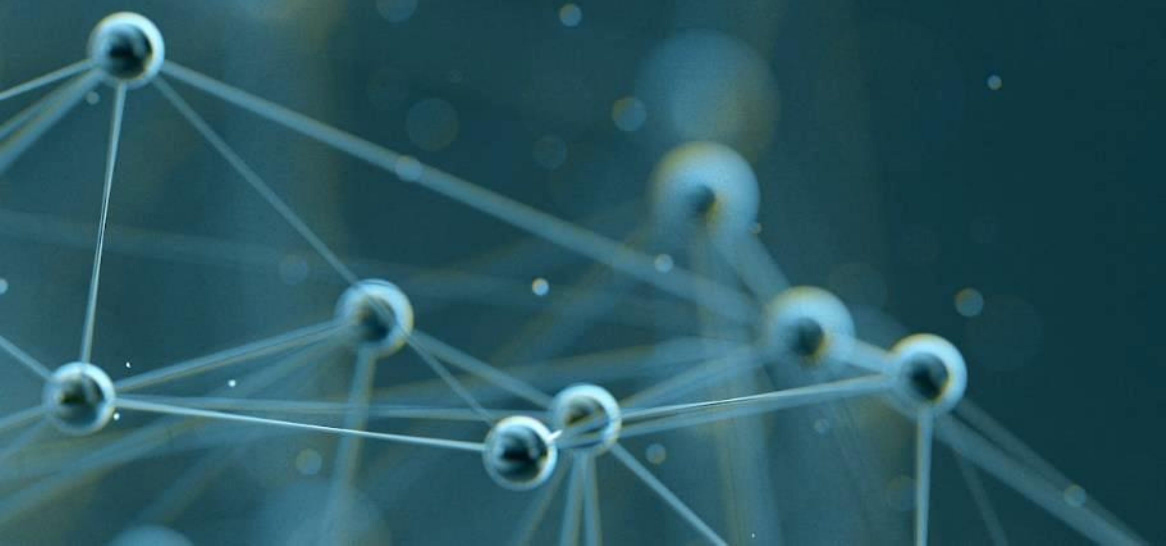
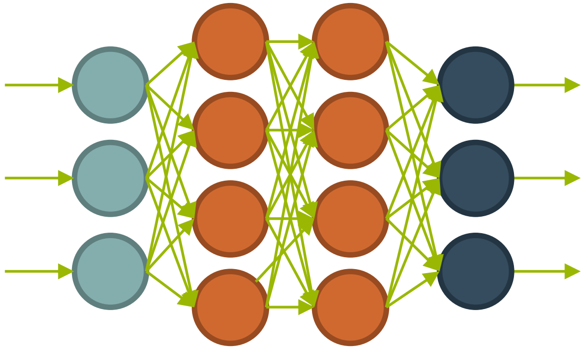
intel Nervana™ AI Academy

INTRODUCTION TO NEURAL NETS

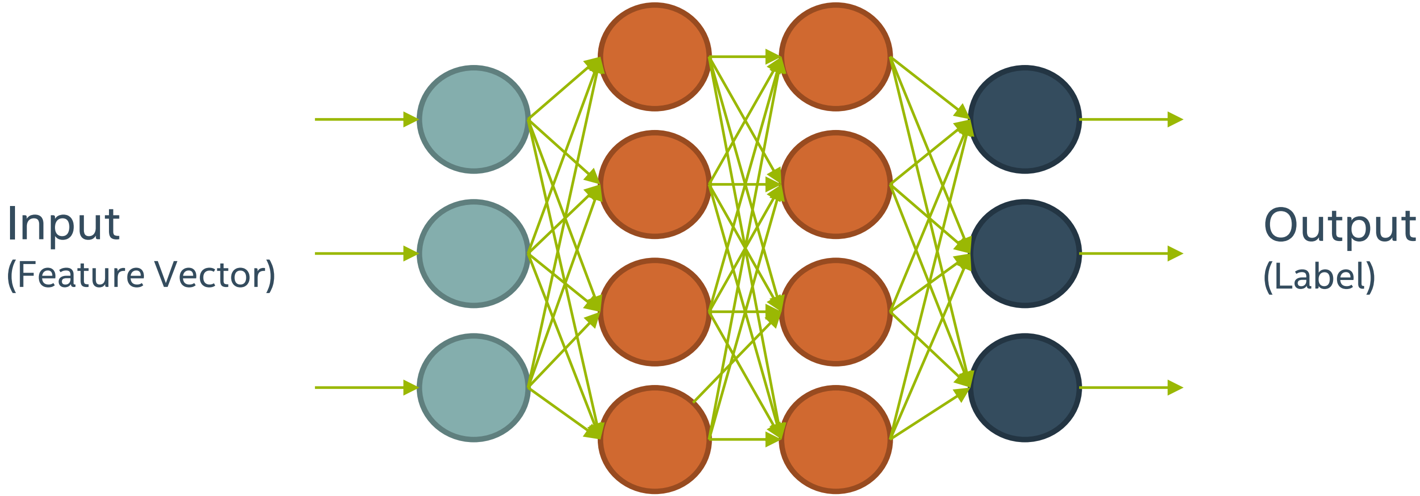


MOTIVATION FOR NEURAL NETS

- Use biology as inspiration for mathematical model
- Get signals from previous neurons
- Generate signals (or not) according to inputs
- Pass signals on to next neurons
- By layering many neurons, can create complex model

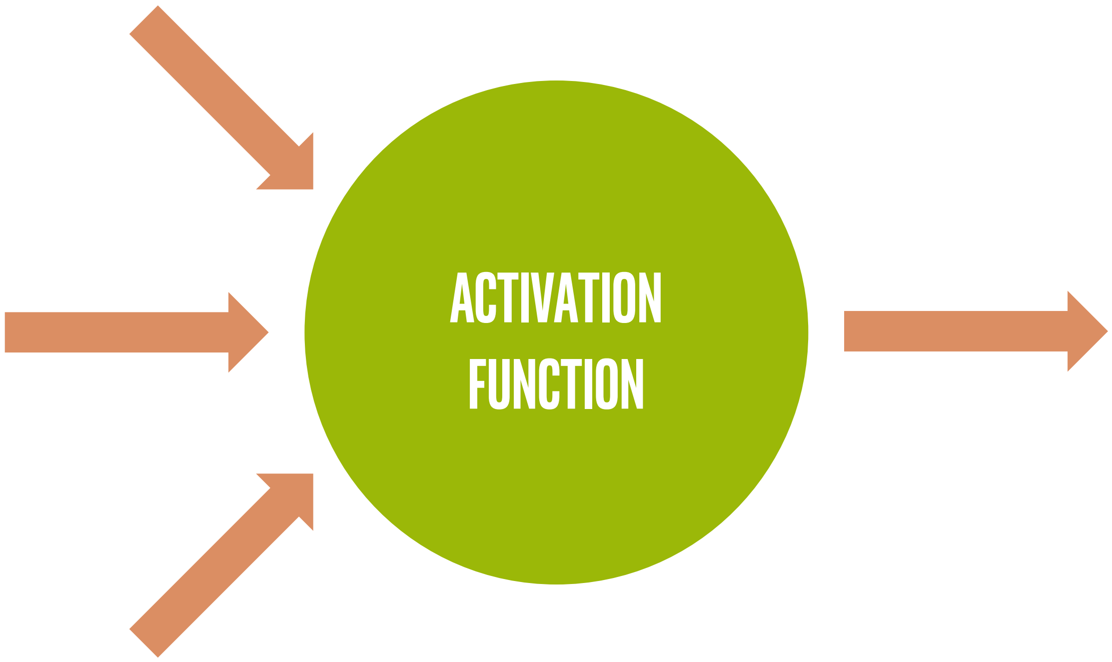


NEURAL NET STRUCTURE

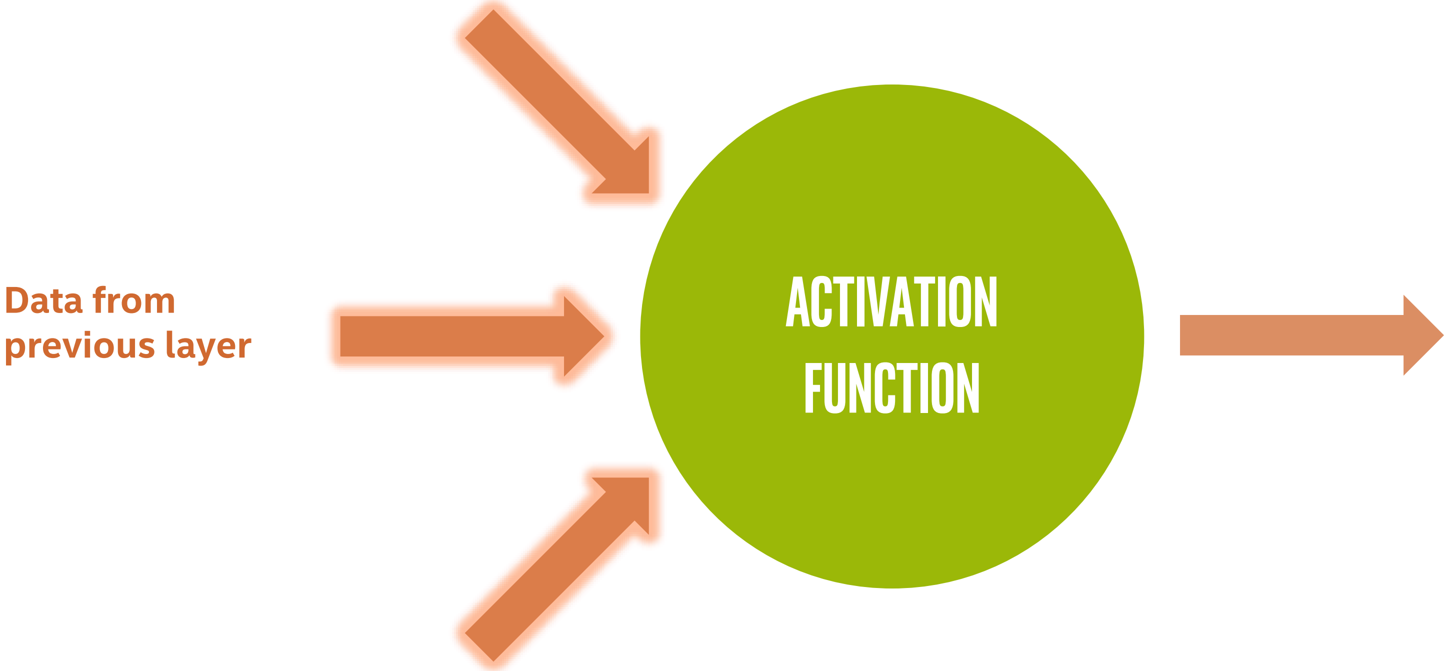


- Can think of it as a complicated computation engine
- We will ”train it” using our training data
- Then (hopefully) it will give good answers on new data

BASIC NEURON VISUALIZATION

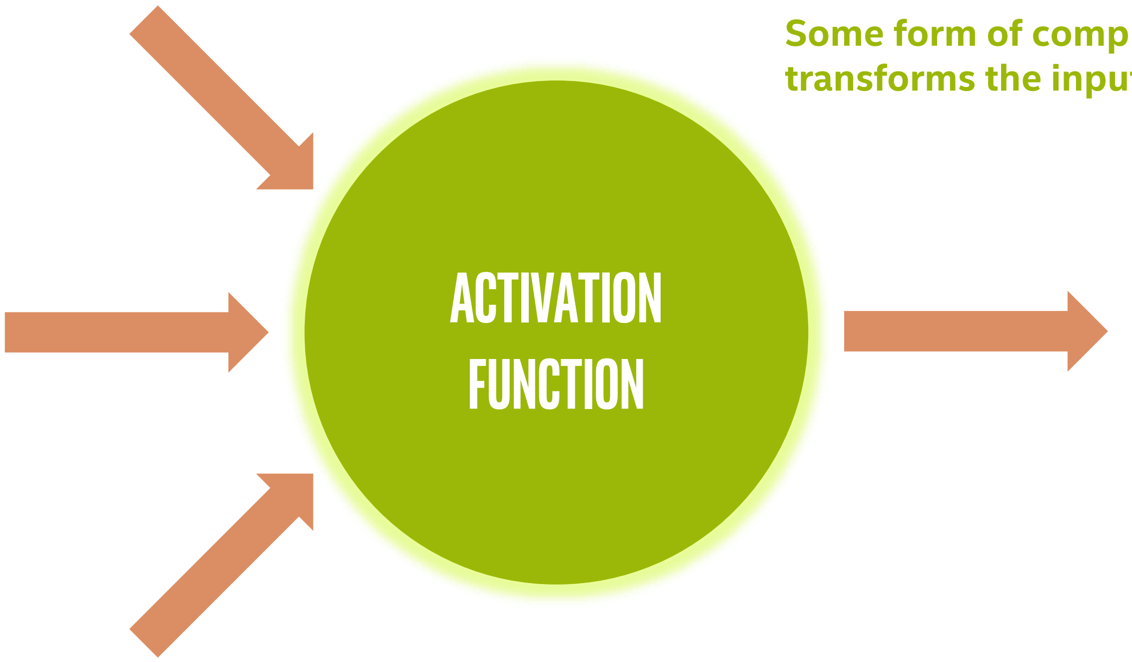


BASIC NEURON VISUALIZATION

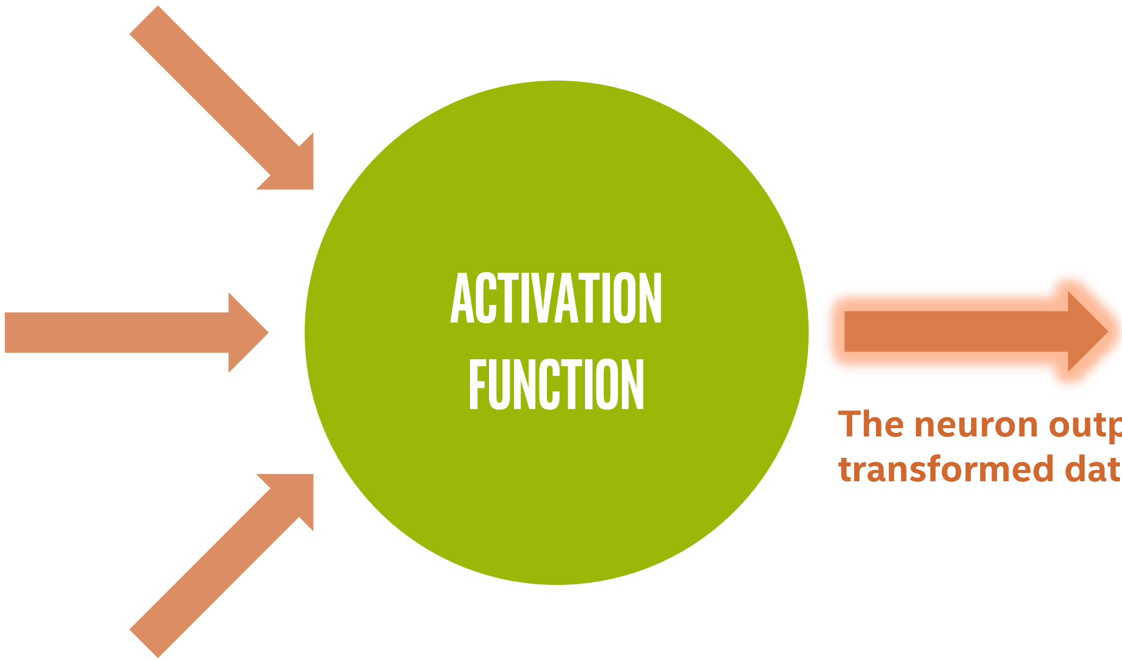


BASIC NEURON VISUALIZATION

Some form of computation transforms the inputs

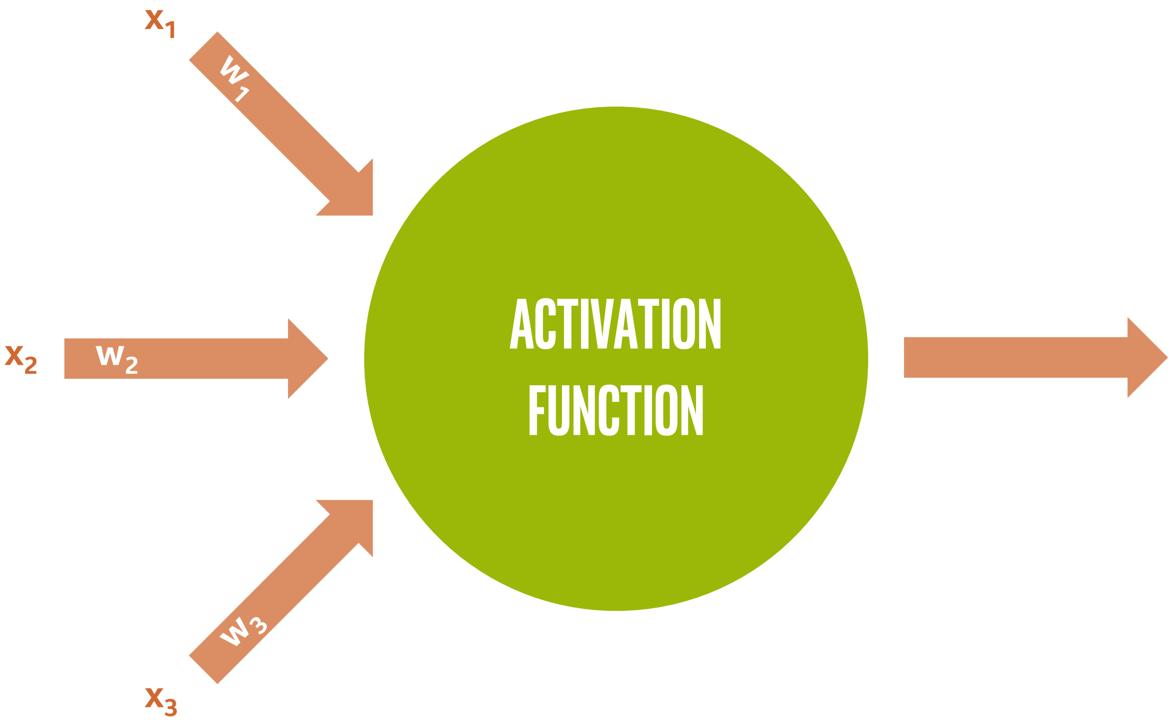


BASIC NEURON VISUALIZATION

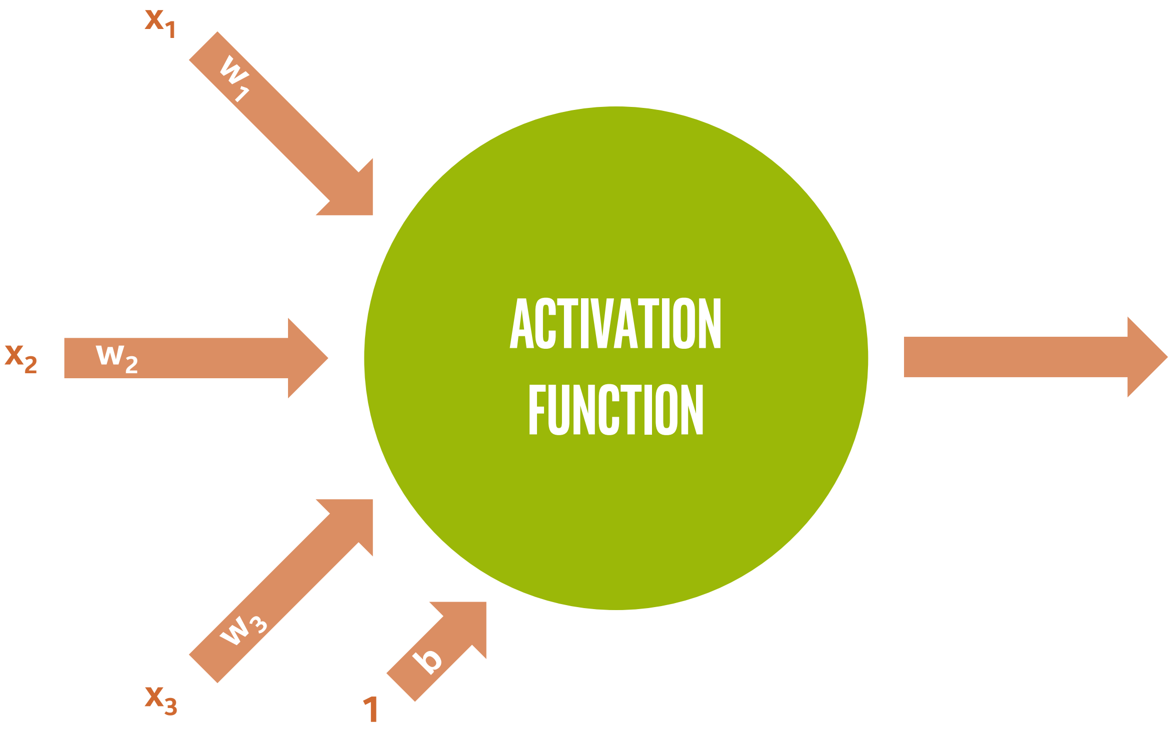


The neuron outputs the transformed data

BASIC NEURON VISUALIZATION

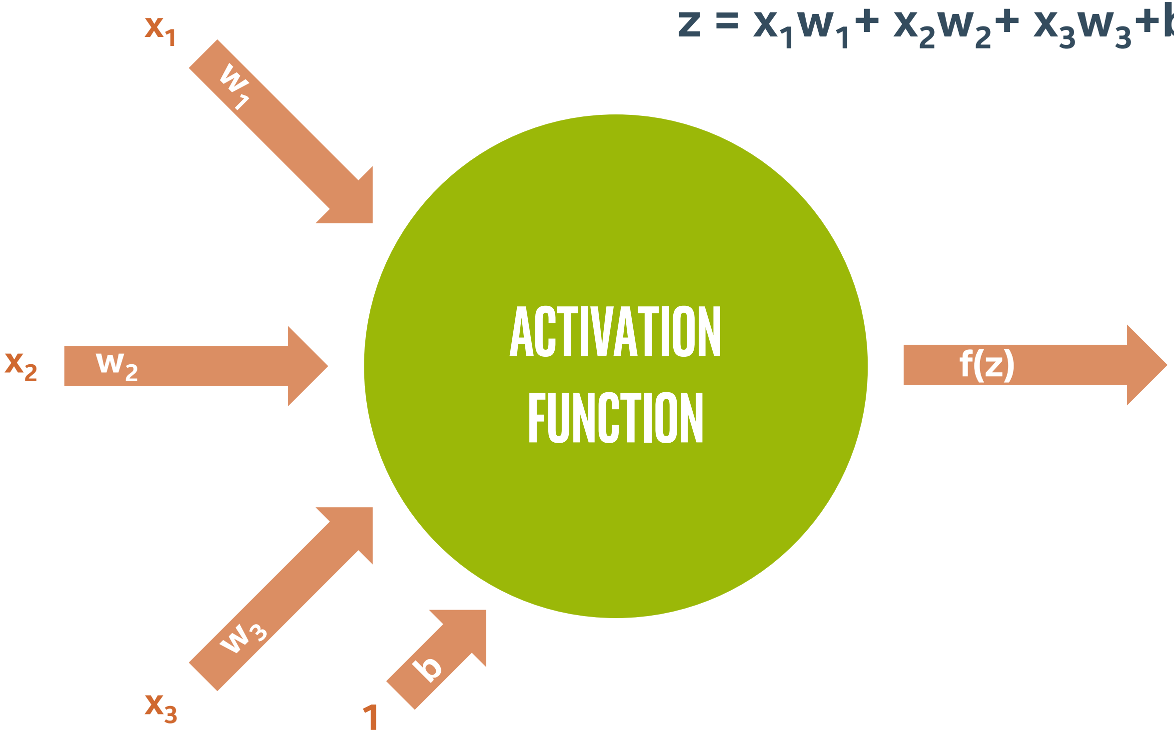


BASIC NEURON VISUALIZATION



BASIC NEURON VISUALIZATION

$$z = x_1w_1 + x_2w_2 + x_3w_3 + b$$



IN VECTOR NOTATION

z = "net input"

b = "bias term"

f = activation function

a = output to next layer

$$z = b + \sum_{i=1}^{m} x_i w_i$$

$$z = b + x^T w$$

$$a = f(z)$$

RELATION TO LOGISTIC REGRESSION

When we choose: $f(z) = \frac{1}{1+e^{-z}}$

$$z = b + \sum_{i=1}^{m} x_i w_i = x_1 w_1 + x_2 w_2 + \cdots + x_m w_m + b$$

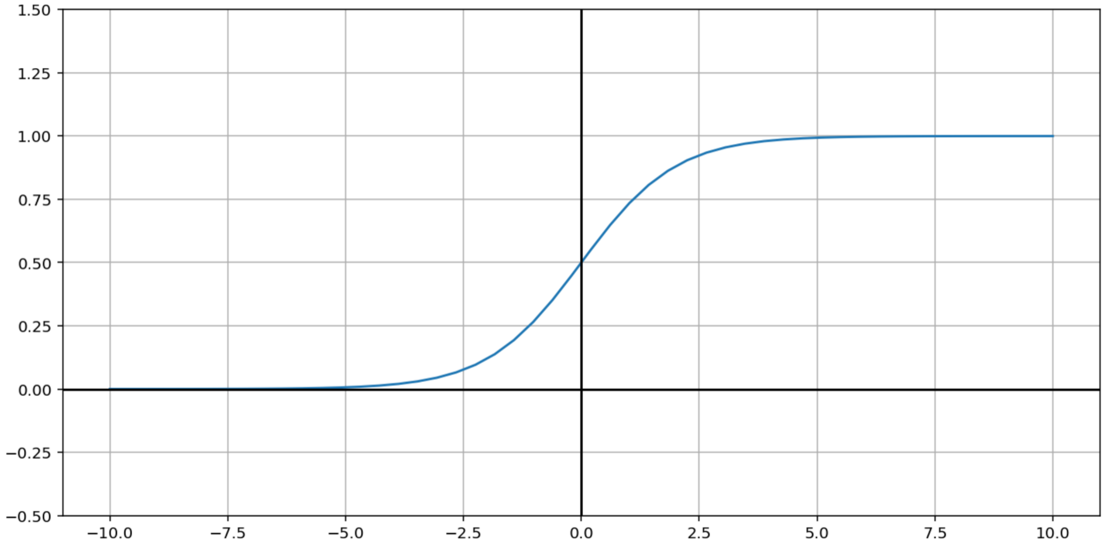
Then a neuron is simply a "unit" of logistic regression!

weights ⇔ coefficients **inputs ⇔ variables**

bias term ⇔ constant term

RELATION TO LOGISTIC REGRESSION

This is called the "sigmoid" function: $\sigma(z) = \dfrac{1}{1 + e^{-z}}$



NICE PROPERTY OF SIGMOID FUNCTION

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

Quotient rule

$$\frac{d}{dx} \cdot \frac{f(x)}{g(x)} = \frac{f'(x)g(x) - f(x)g'(x)}{g(x)^2}$$

$$\sigma'(z) = \frac{0-(-e^{-z})}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2}$$

$$= \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{\cancel{1+e^{-z}}}{(1+e^{-z})^{\cancel{2}}} - \frac{1}{(1+e^{-z})^2}$$

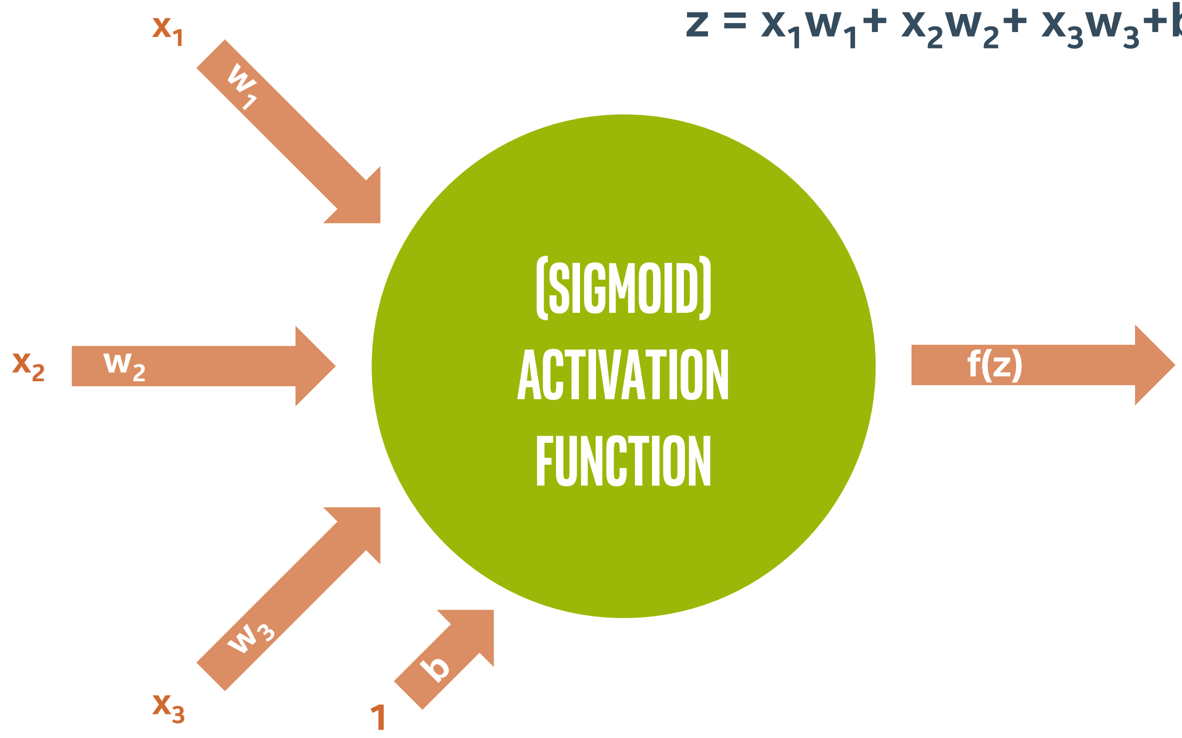
$$= \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right)$$

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

This will be helpful!

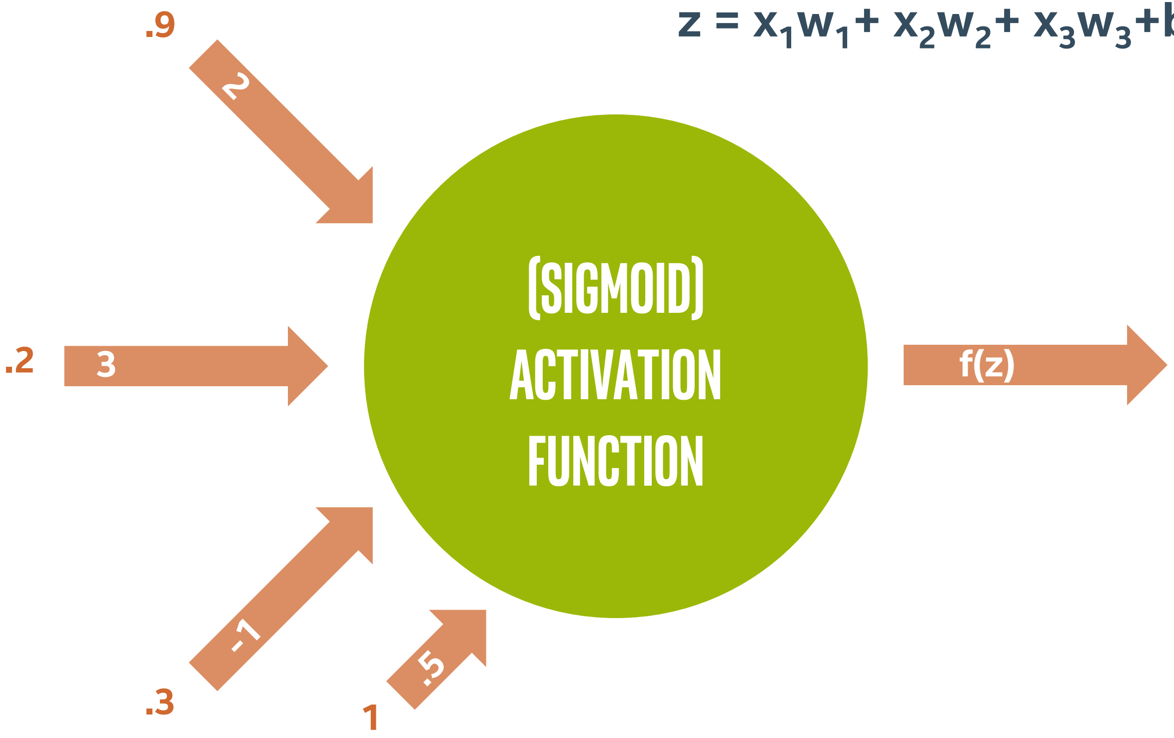
EXAMPLE NEURON COMPUTATION

$$z = x_1 w_1 + x_2 w_2 + x_3 w_3 + b$$



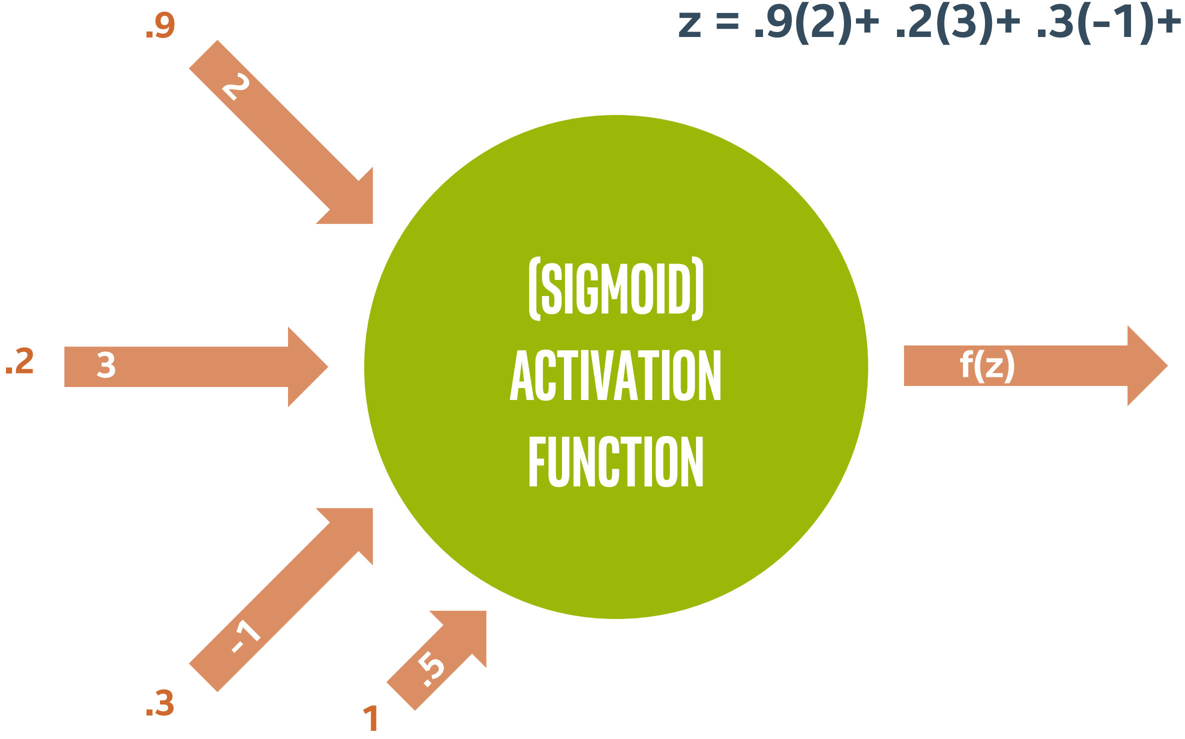
EXAMPLE NEURON COMPUTATION

$$z = x_1w_1 + x_2w_2 + x_3w_3 + b$$



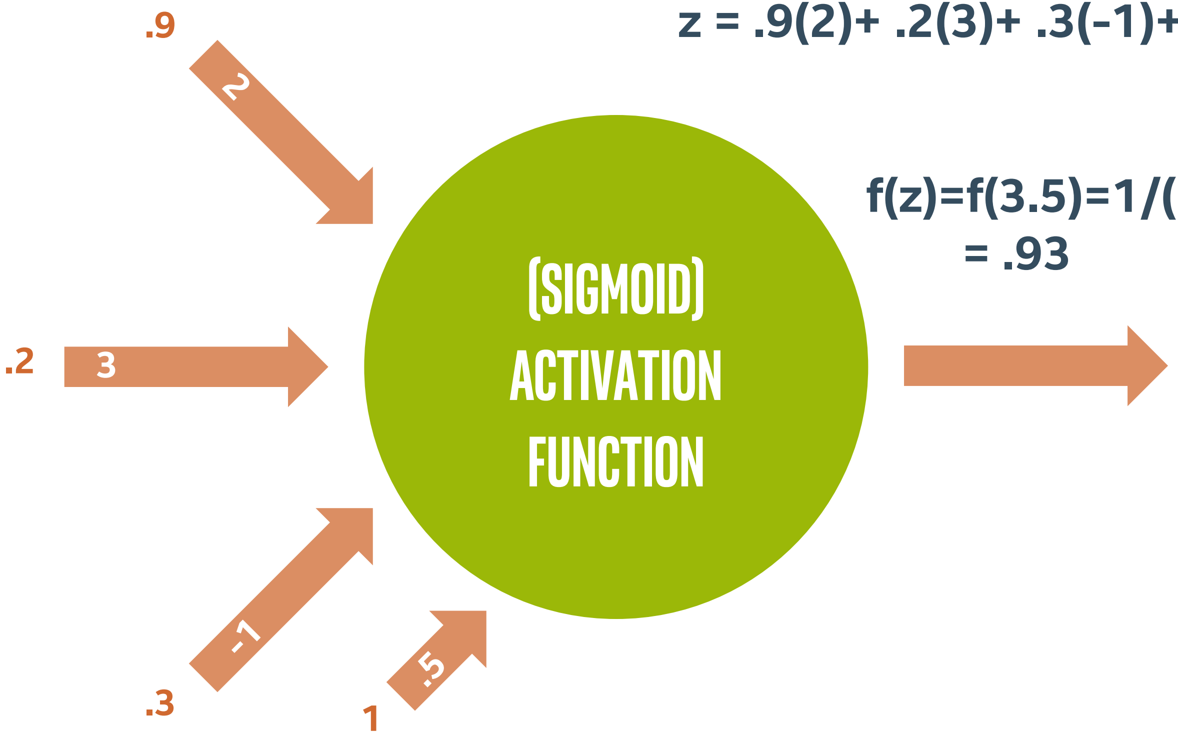
EXAMPLE NEURON COMPUTATION

$$z = .9(2) + .2(3) + .3(-1) + .5 = 2.6$$



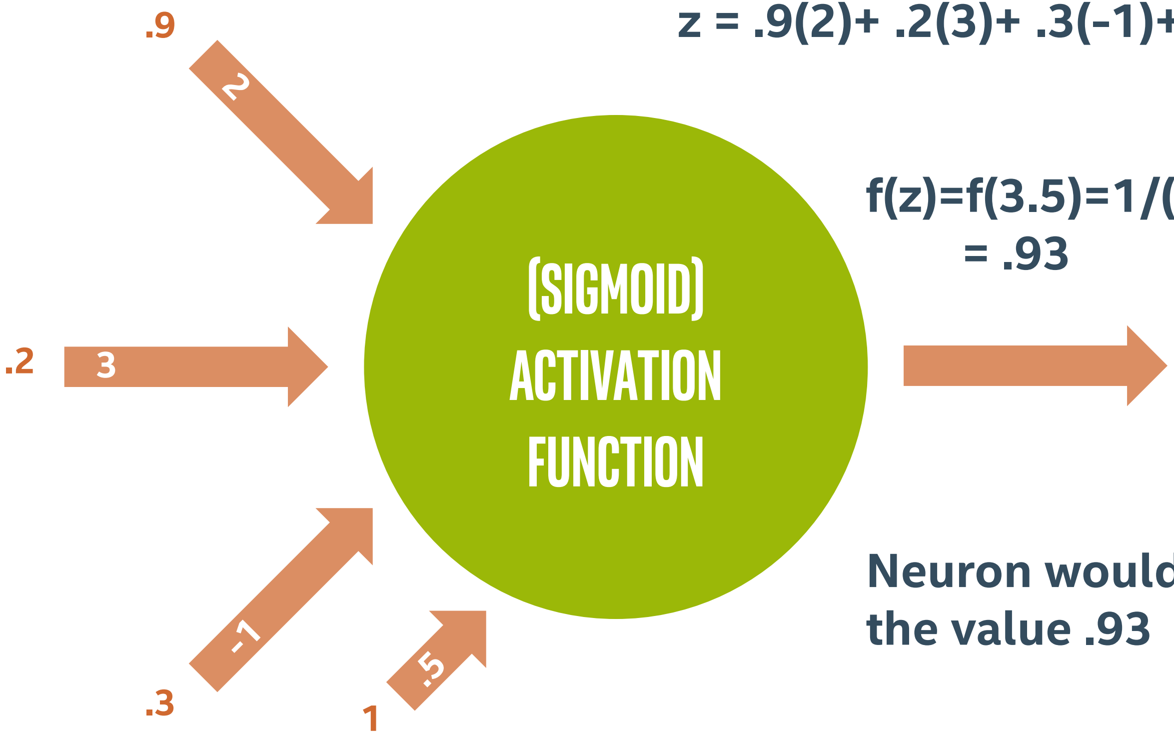
EXAMPLE NEURON COMPUTATION

$$z = .9(2) + .2(3) + .3(-1) + .5 = 2.6$$



$$f(z) = f(3.5) = 1/(1+\exp(-2.6)) = .93$$

EXAMPLE NEURON COMPUTATION



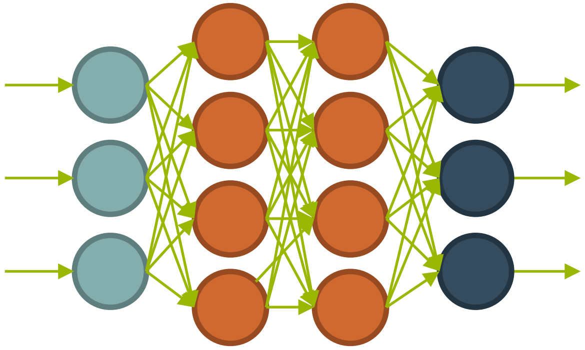
z = .9(2)+ .2(3)+ .3(-1)+.5 = 2.6

f(z)=f(3.5)=1/(1+exp(-2.6))
= .93

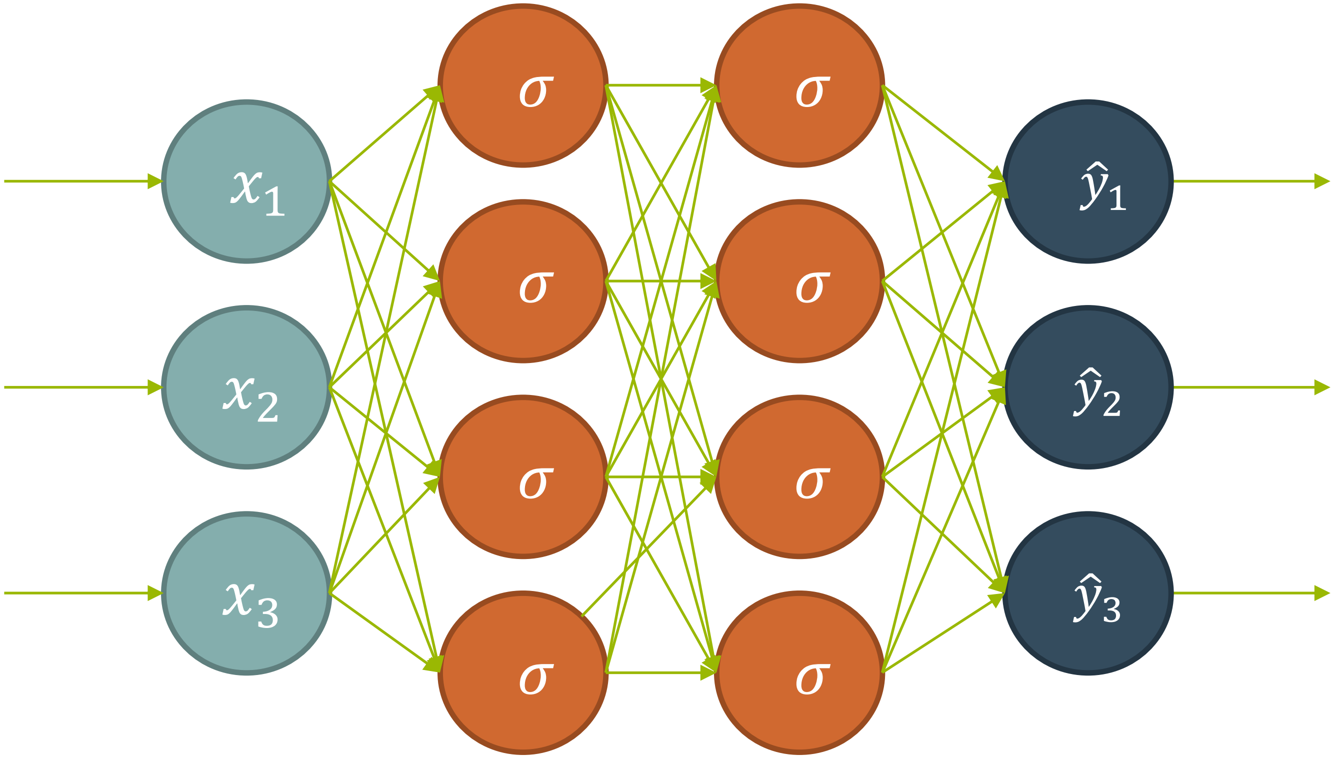
Neuron would output the value .93

WHY NEURAL NETS?

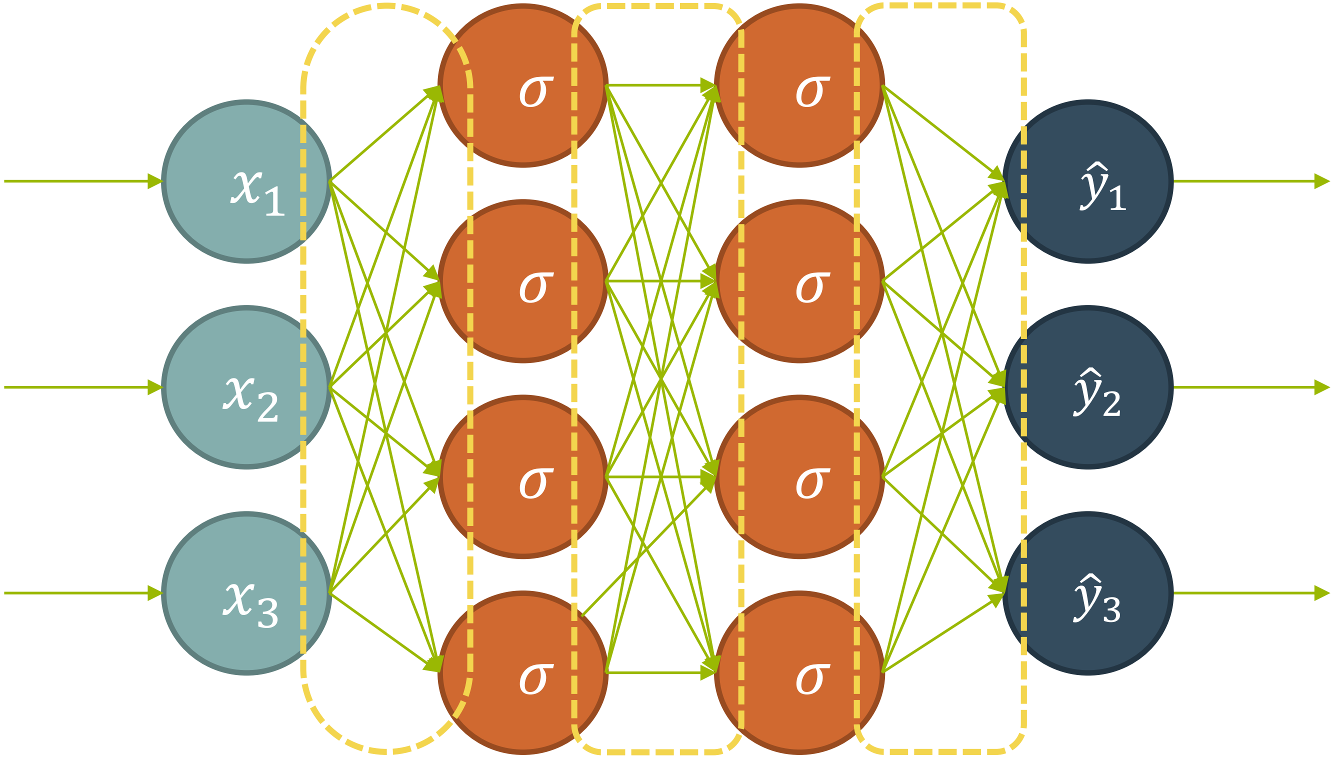
- Why not just use a single neuron? Why do we need a larger network?
- A single neuron (like logistic regression) only permits a linear decision boundary.
- Most real-world problems are considerably more complicated!



FEEDFORWARD NEURAL NETWORK

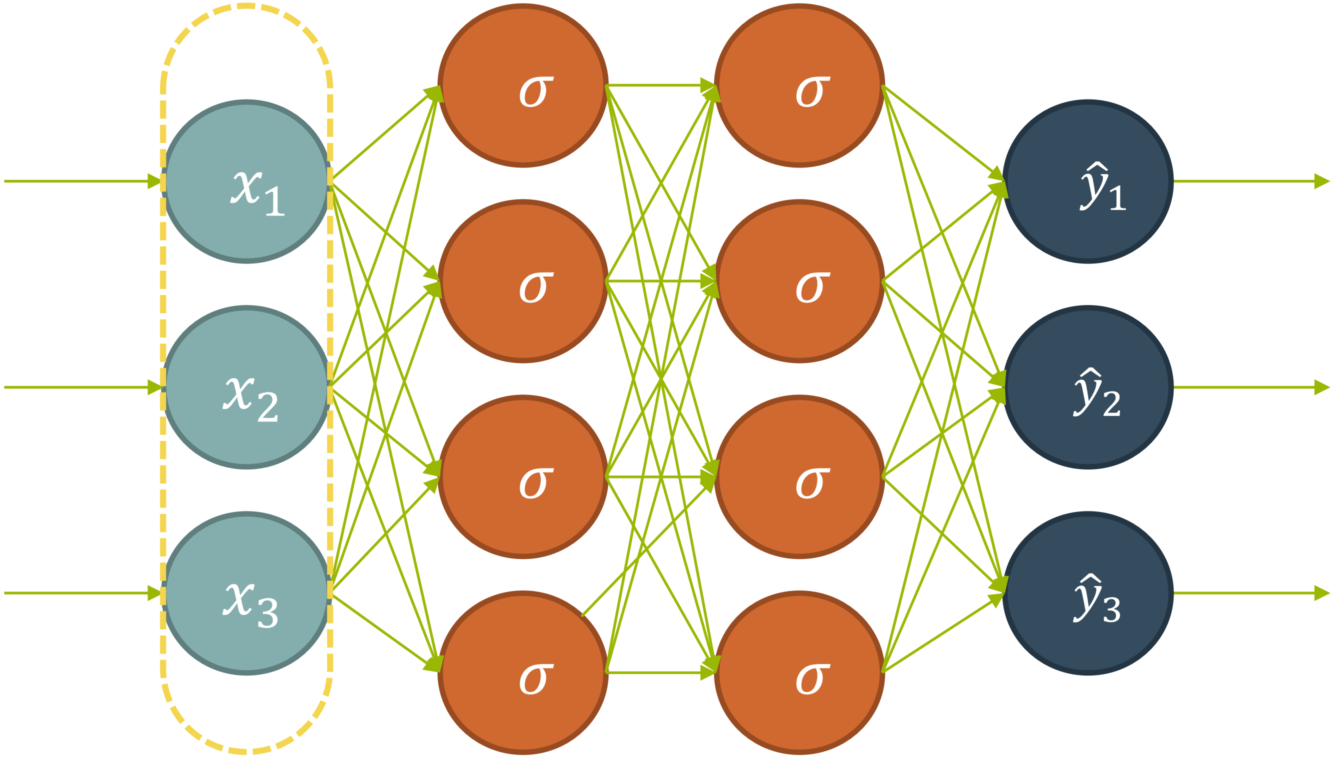


WEIGHTS



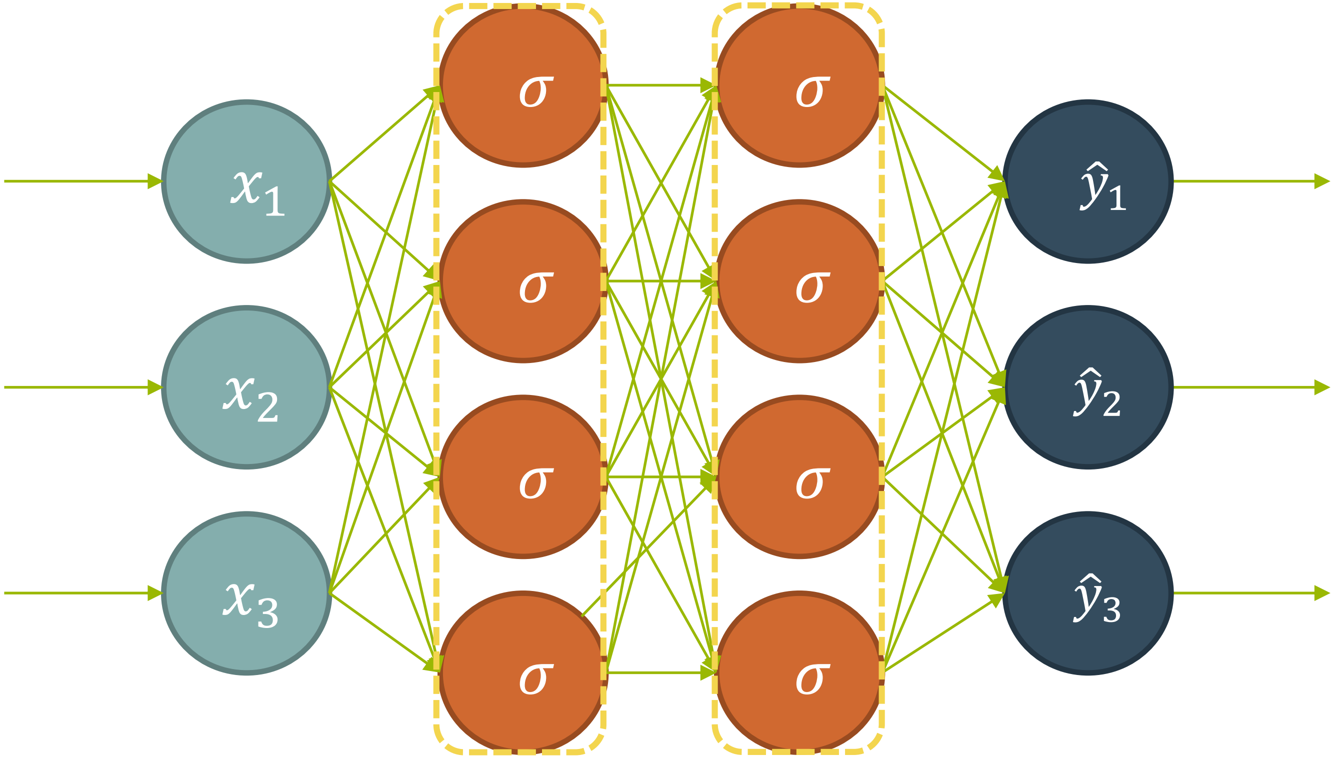
x_1
x_2
x_3
σ
σ
σ
σ
σ
σ
σ
σ
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$

INPUT LAYER

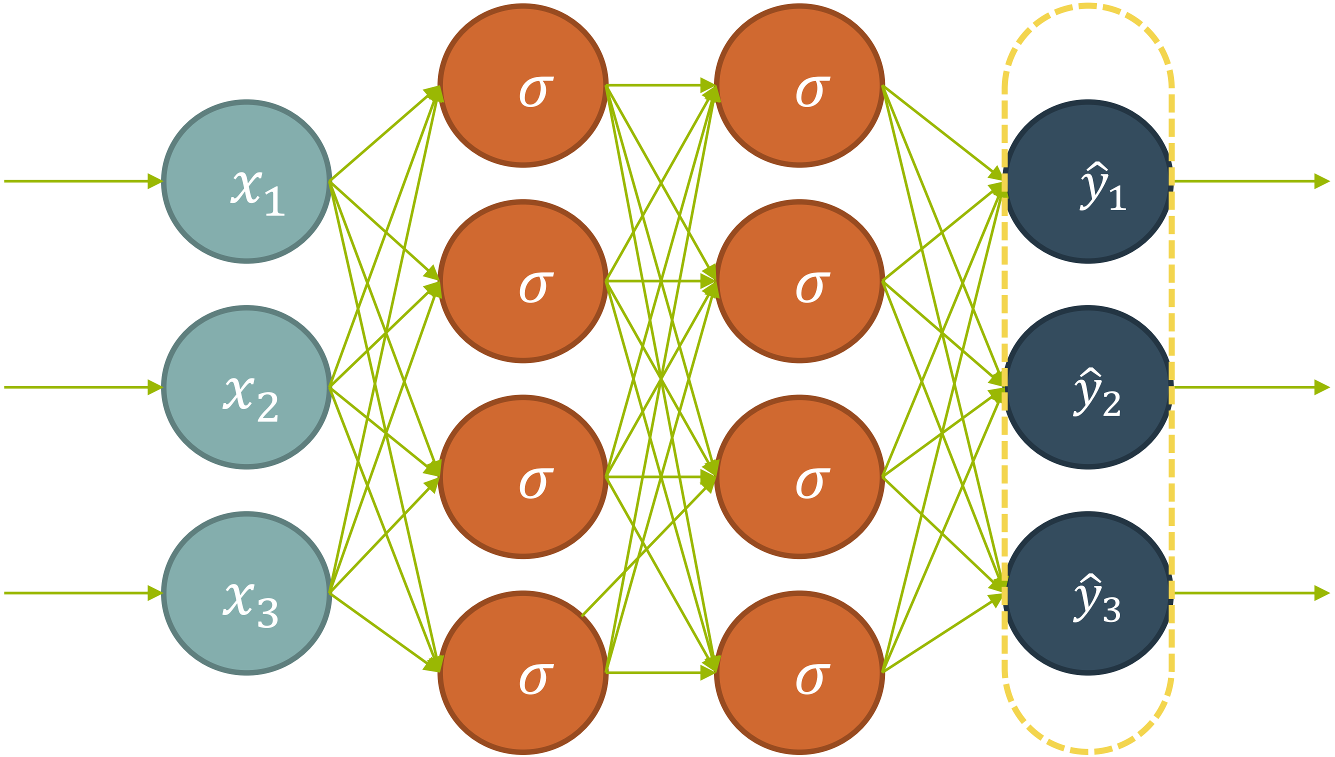


x_1
x_2
x_3
σ
σ
σ
σ
σ
σ
σ
σ
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$

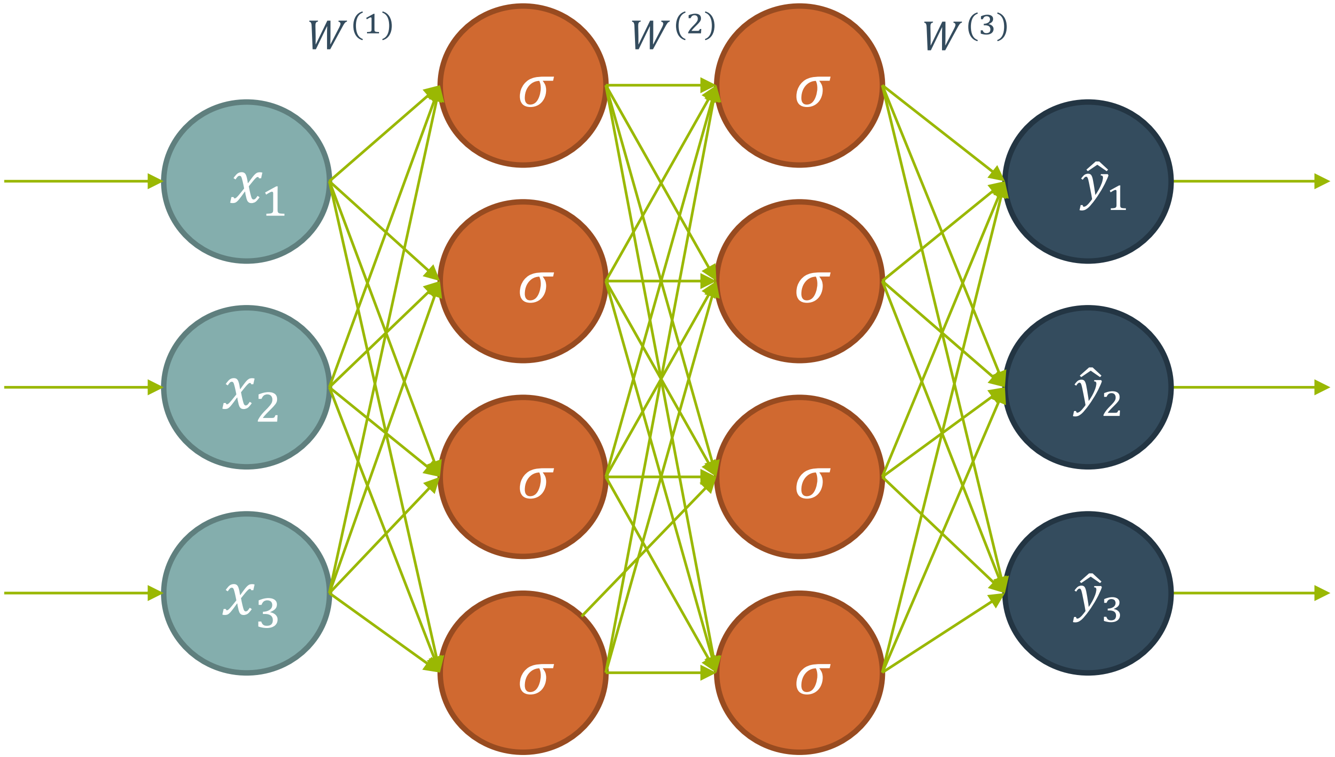
HIDDEN LAYERS



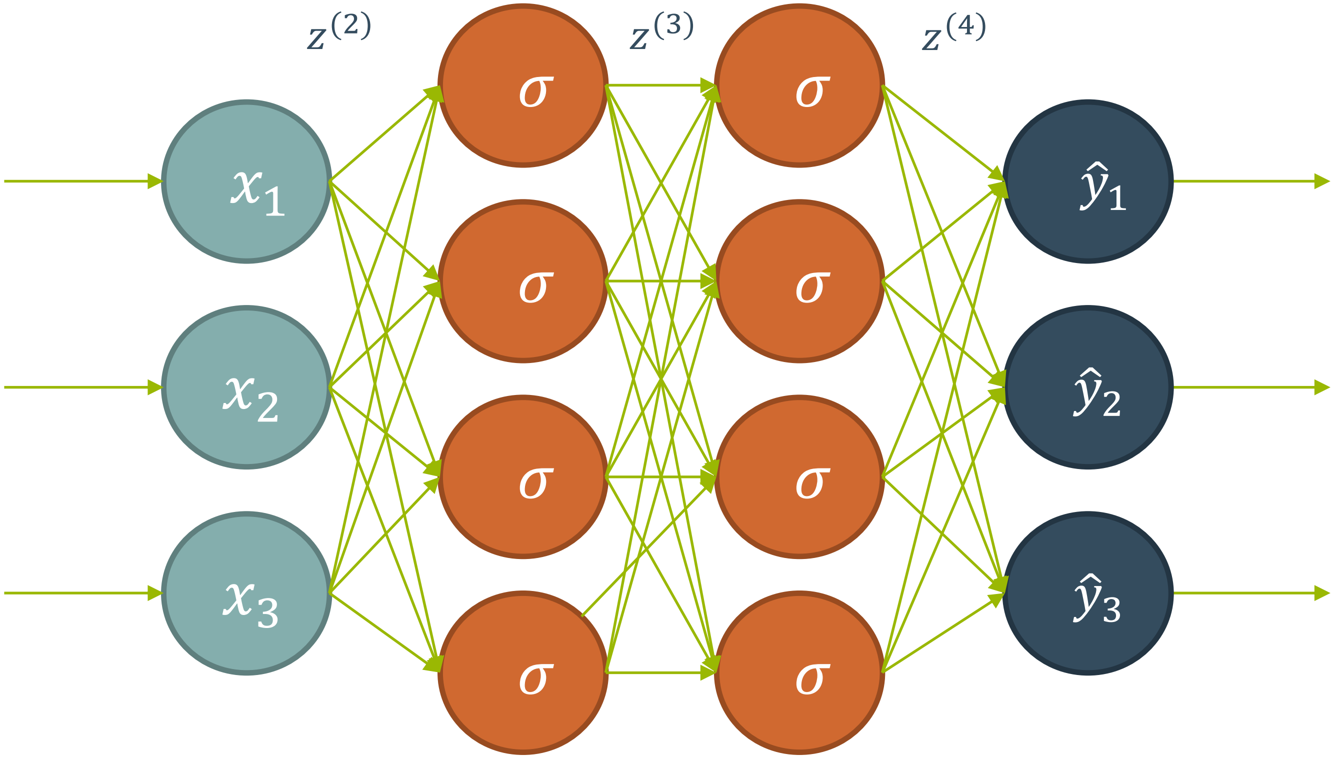
OUTPUT LAYER



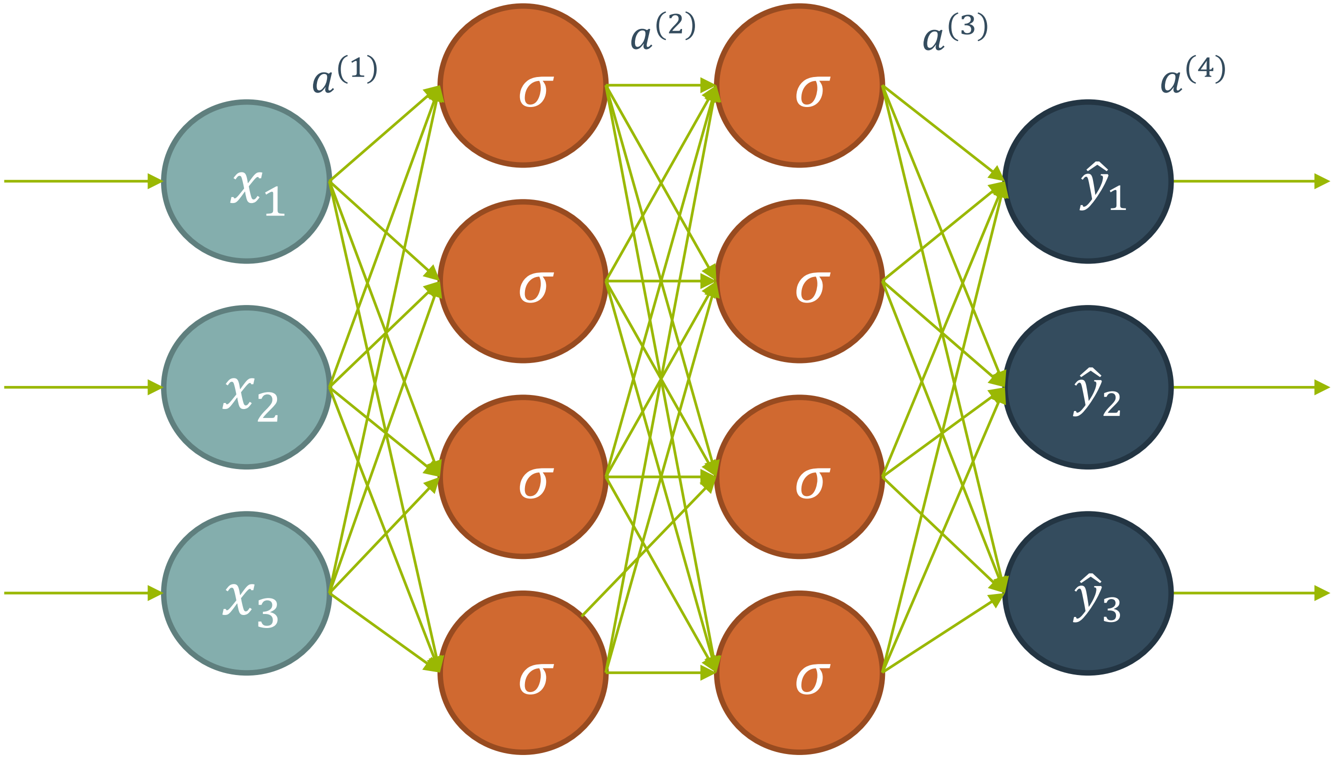
WEIGHTS (REPRESENTED BY MATRICES)



NET INPUT (SUM OF WEIGHTED INPUTS, BEFORE ACTIVATION FUNCTION)



ACTIVATIONS (OUTPUT OF NEURONS TO NEXT LAYER)



MATRIX REPRESENTATION OF COMPUTATION

x

$(x = a^{(1)})$

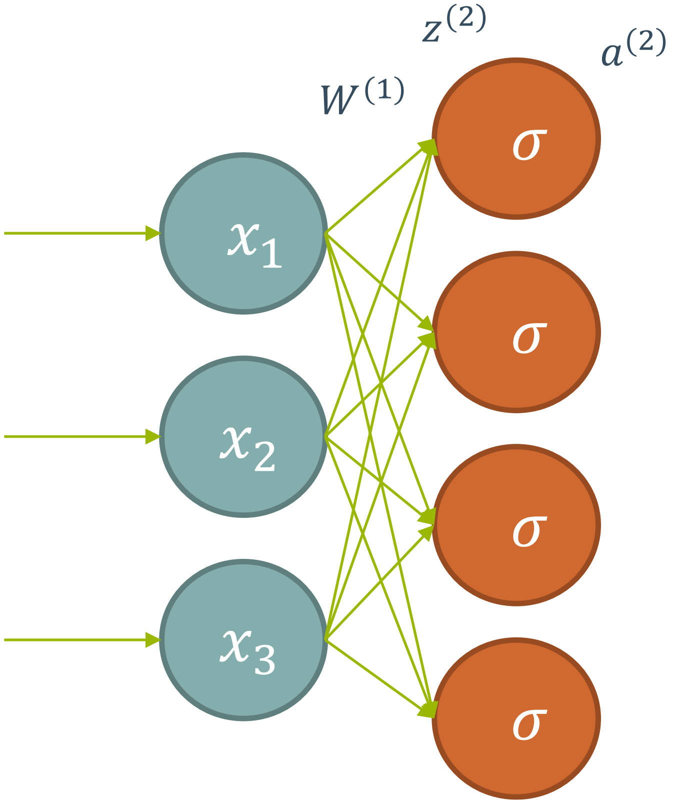
$z^{(2)} = xW^{(1)}$

$a^{(2)} = \sigma(z^{(2)})$

$W^{(1)}$ is a 3x4 matrix

$z^{(2)}$ is a 4-vector

$a^{(2)}$ is a 4-vector



CONTINUING THE COMPUTATION

For a single training instance (data point)

Input: vector x (a row vector of length 3)

Output: vector $\hat{y}$ (a row vector of length 3)

$$z^{(2)} = xW^{(1)} \qquad a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(2)} \qquad a^{(3)} = \sigma(z^{(3)})$$

$$z^{(4)} = a^{(3)}W^{(3)} \qquad \hat{y} = softmax(z^{(4)})$$

MULTIPLE DATA POINTS

In practice, we do these computation for many data points at the same time, by "stacking" the rows into a matrix. But the equations look the same!

Input: matrix x (an nx3 matrix) (each row a single instance)

Output: vector $\hat{y}$ (an nx3 matrix) (each row a single prediction)

$$z^{(2)} = xW^{(1)} \qquad a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = a^{(2)}W^{(2)} \qquad a^{(3)} = \sigma(z^{(3)})$$

$$z^{(4)} = a^{(3)}W^{(3)} \qquad \hat{y} = softmax(z^{(4)})$$

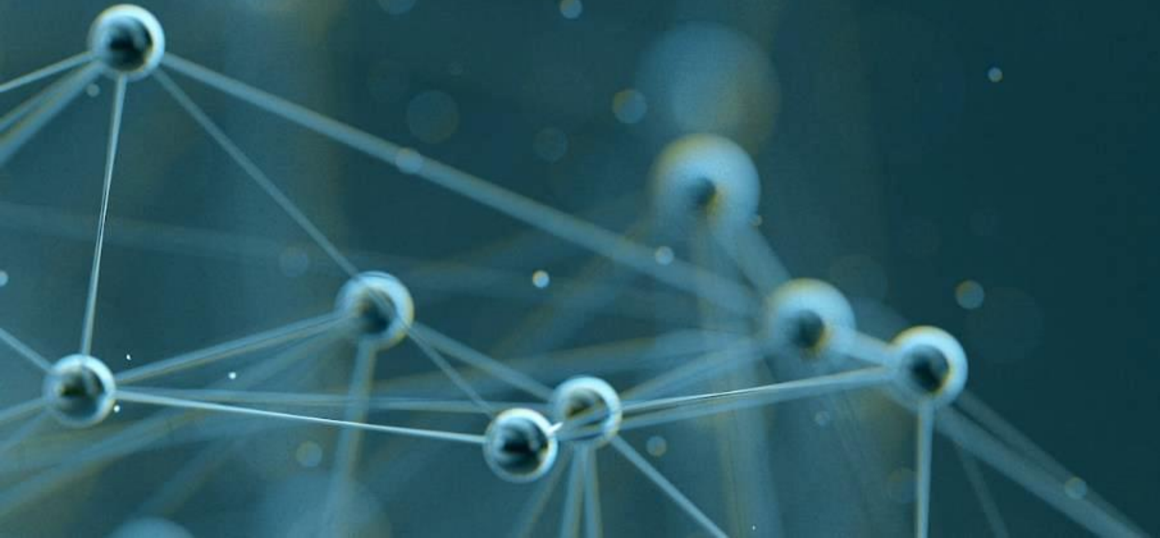
Now we know how feedforward NNs do Computations.

Next, we will learn how to adjust the weights to learn from data.

Nervana™ AI Academy

Nervana™ AI Academy

BACKPROPAGATION IN NEURAL NETS

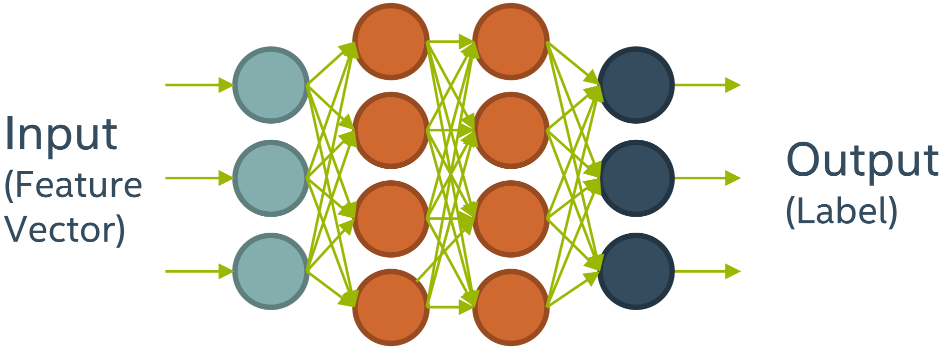


HOW TO TRAIN A NEURAL NET?

- Put in training inputs, get the output
- Compare output to correct answers: look at loss function J
- Adjust and repeat!
- Backpropagation tells us how to make a single adjustment using calculus.

HOW TO TRAIN A NEURAL NET?

- Put in training inputs, get the output
- Compare output to correct answers: look at loss function J
- Adjust and repeat!
- Backpropagation tells us how to make a single adjustment using calculus.



HOW HAVE WE TRAINED BEFORE?

Gradient Descent!

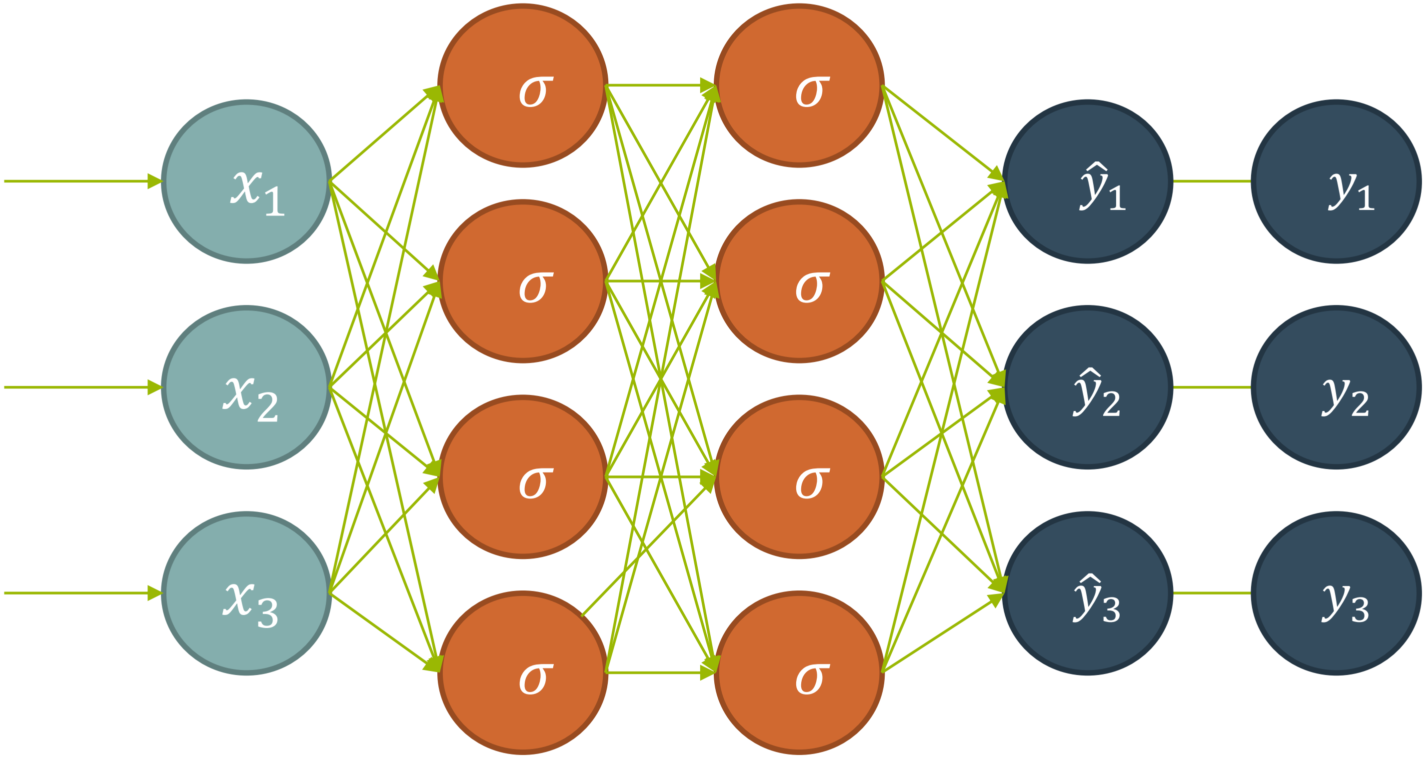
1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

HOW HAVE WE TRAINED BEFORE?

Gradient Descent!

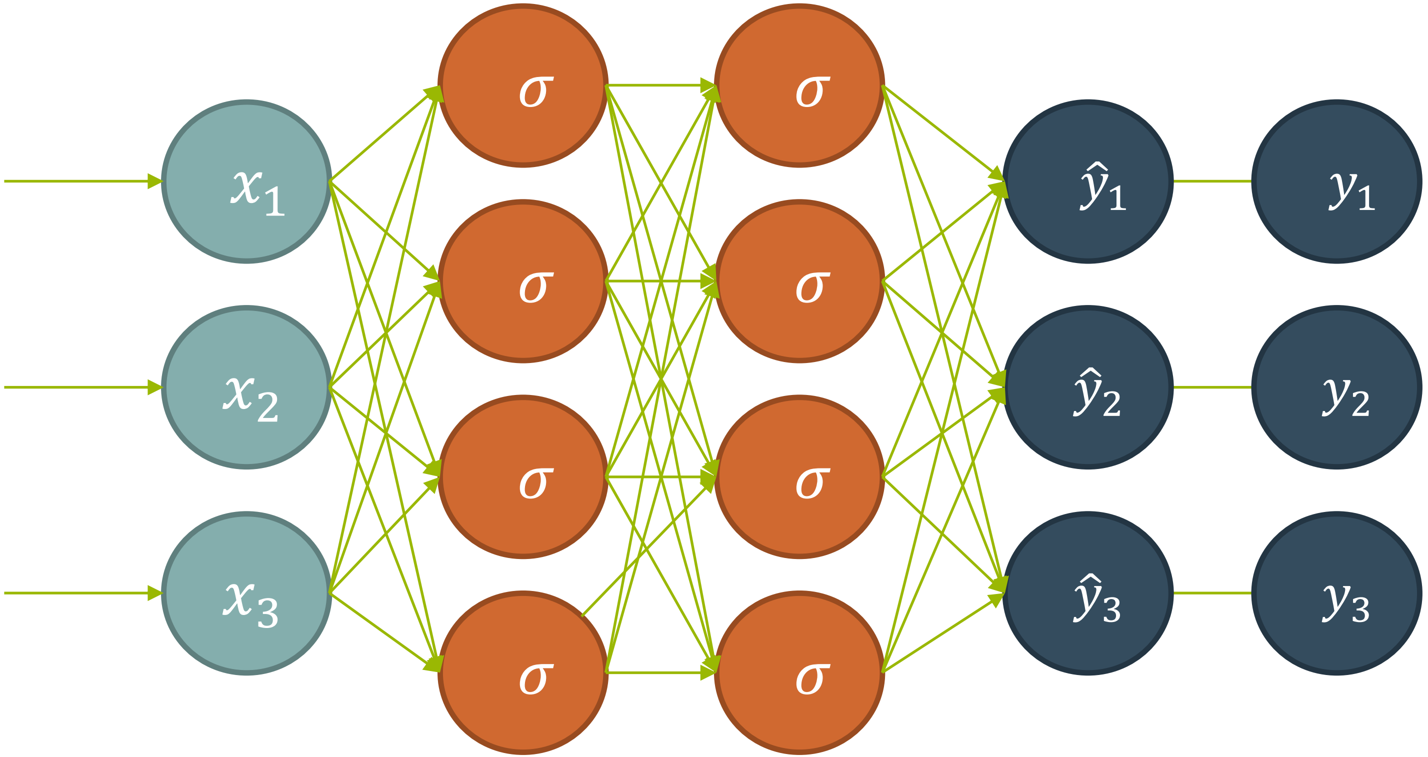
1. **Make prediction**
2. **Calculate Loss**
3. Calculate gradient of the loss function w.r.t. parameters
4. Update parameters by taking a step in the opposite direction
5. Iterate

FEEDFORWARD NEURAL NETWORK



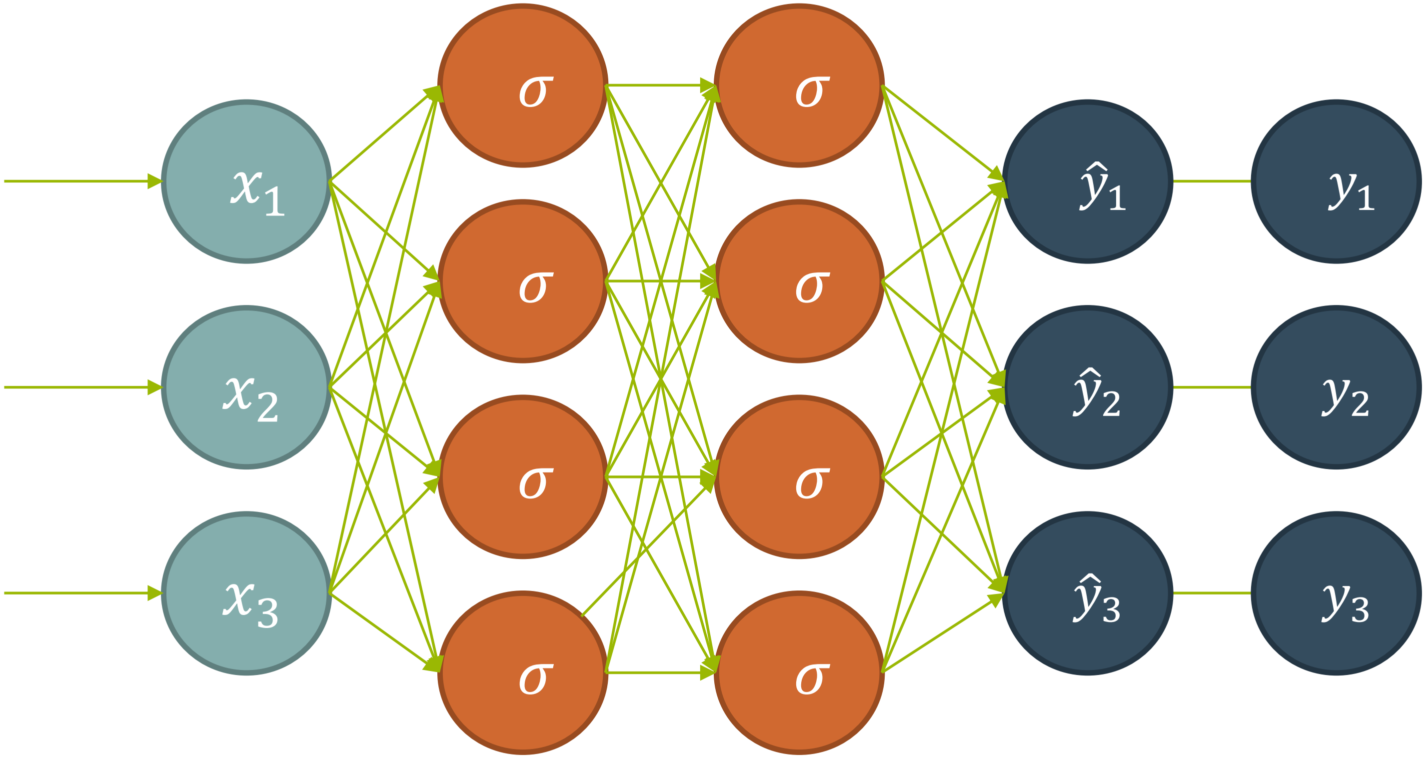
FORWARD PROPAGATION

Pass in Input

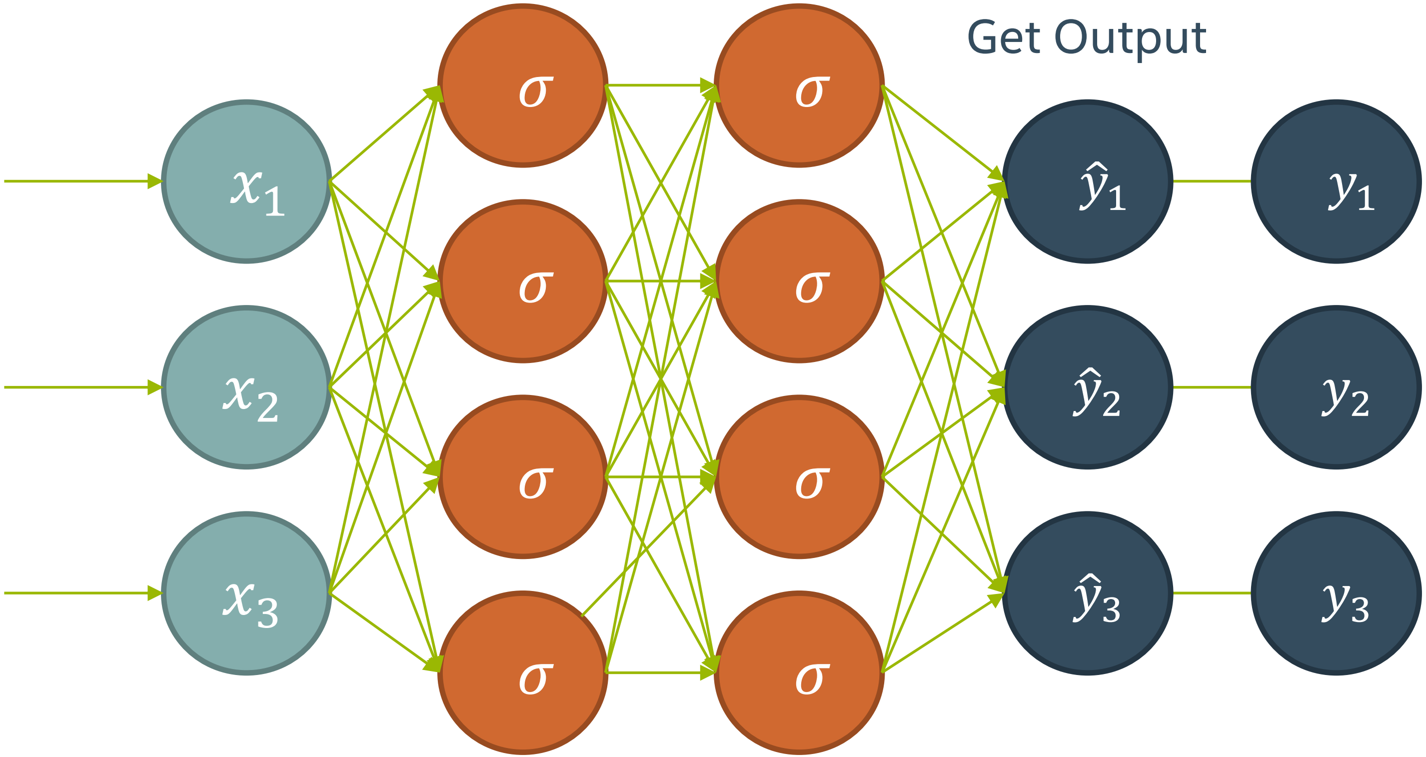


FORWARD PROPAGATION

Calculate each Layer

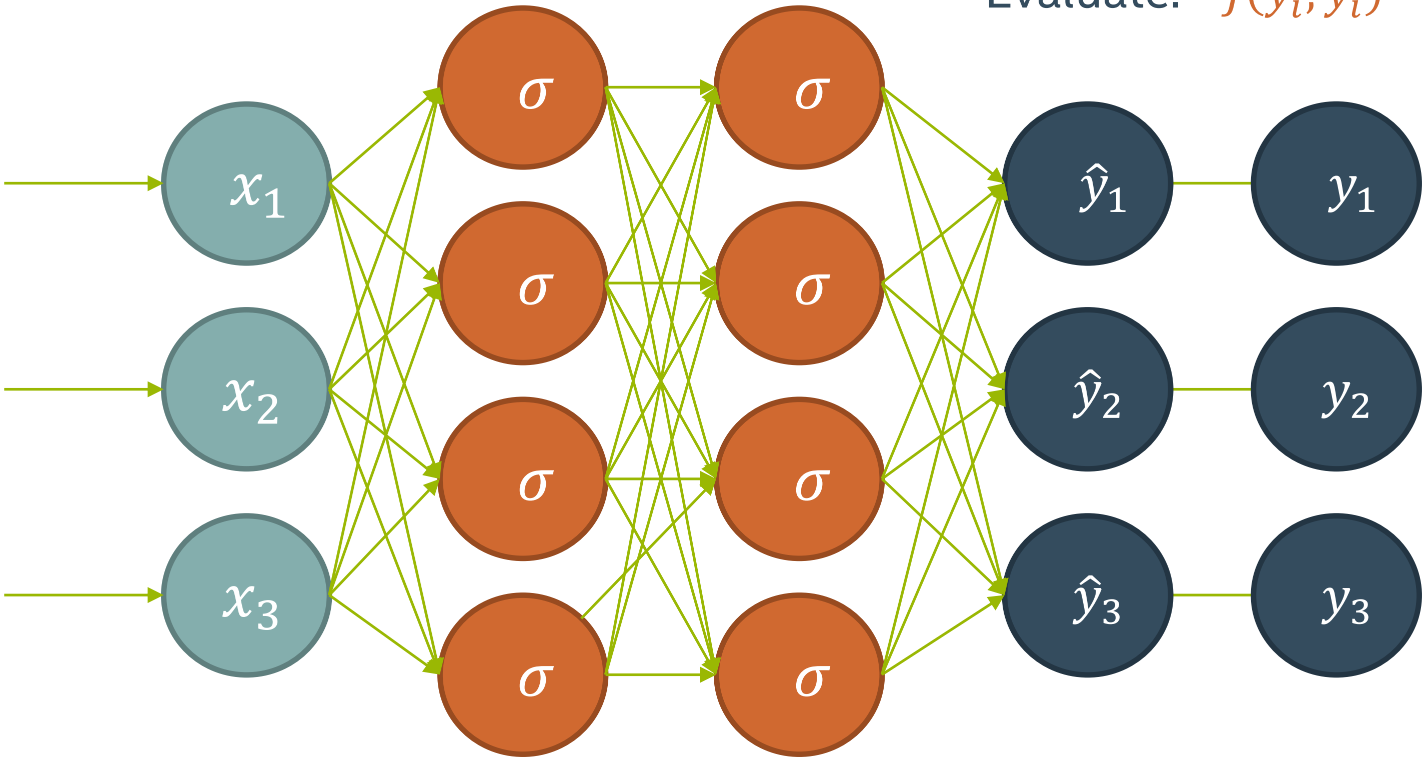


FORWARD PROPAGATION



FORWARD PROPAGATION

Evaluate: $J(y_i, \hat{y_i})$



HOW HAVE WE TRAINED BEFORE?

Gradient Descent!

1. Make prediction
2. Calculate Loss
3. **Calculate gradient of the loss function w.r.t. parameters**
4. Update parameters by taking a step in the opposite direction
5. Iterate

HOW TO CALCULATE GRADIENT?

Chain rule

HOW TO TRAIN A NEURAL NET?

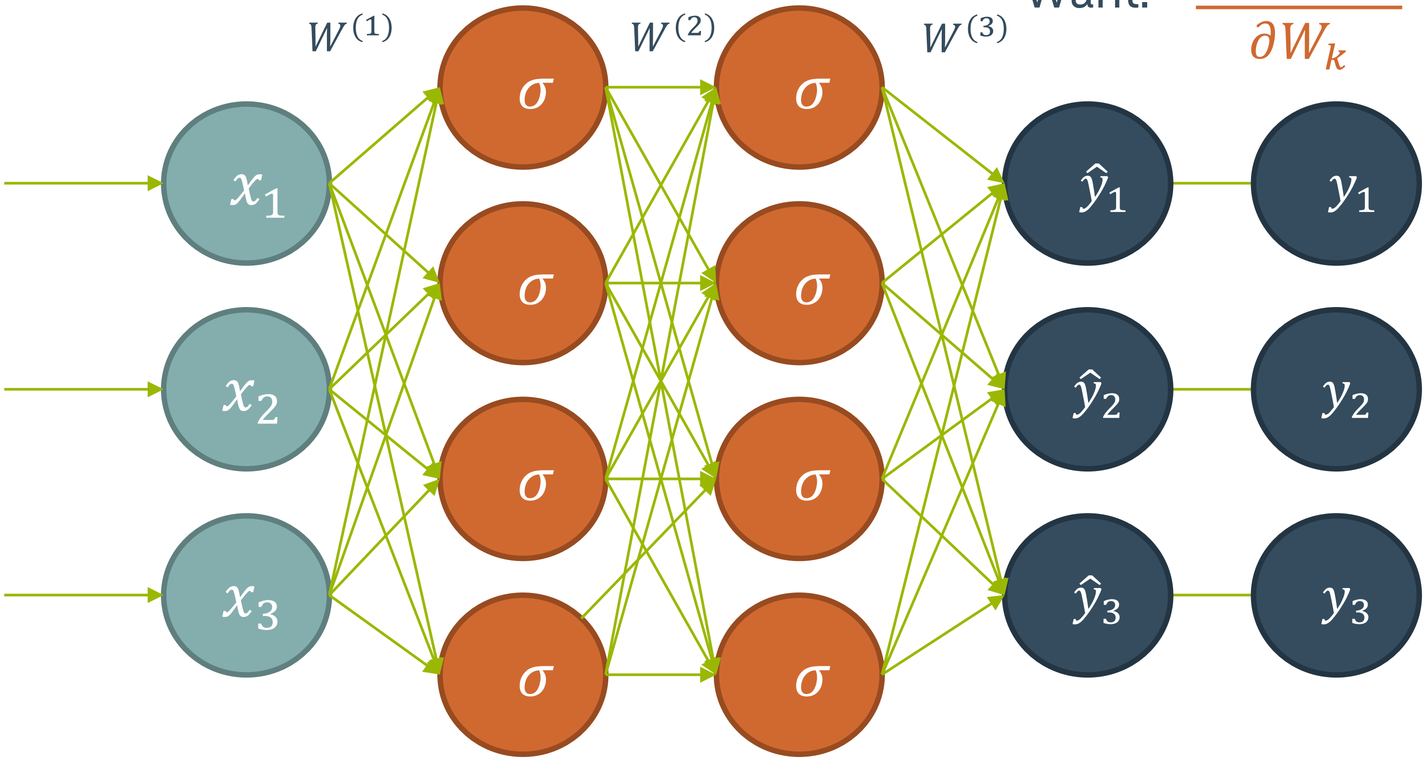
- How could we change the weights to make our Loss Function lower?
- Think of neural net as a function F: X -> Y
- F is a complex computation involving many weights W_k
- Given the structure, the weights "define" the function F (and therefore define our model)
- Loss Function is J(y,F(x))

HOW TO TRAIN A NEURAL NET?

- Get $\frac{\partial J}{\partial W_k}$ for every weight in the network.
- This tells us what direction to adjust each Wk if we want to lower our loss function.
- Make an adjustment and repeat!

FEEDFORWARD NEURAL NETWORK

Want: $\frac{\partial J(y_i, \hat{y}_i)}{\partial W_k}$



CALCULUS TO THE RESCUE

- Use calculus, chain rule, etc. etc.
- Functions are chosen to have “nice” derivatives
- Numerical issues to be considered

PUNCHLINE

$$\frac{\partial J}{\partial W^{(3)}} = (\hat{y} - y) \cdot a^{(3)}$$

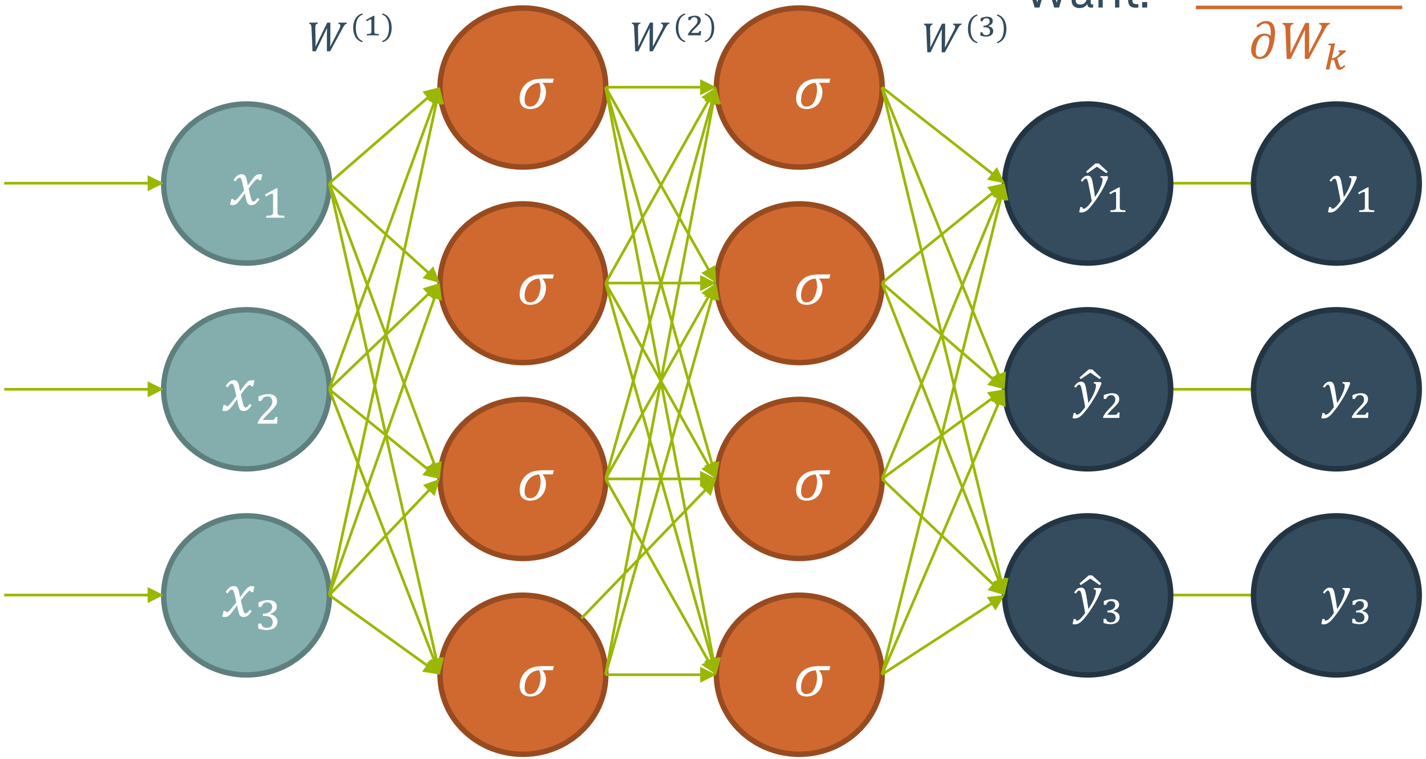
$$\frac{\partial J}{\partial W^{(2)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot a^{(2)}$$

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

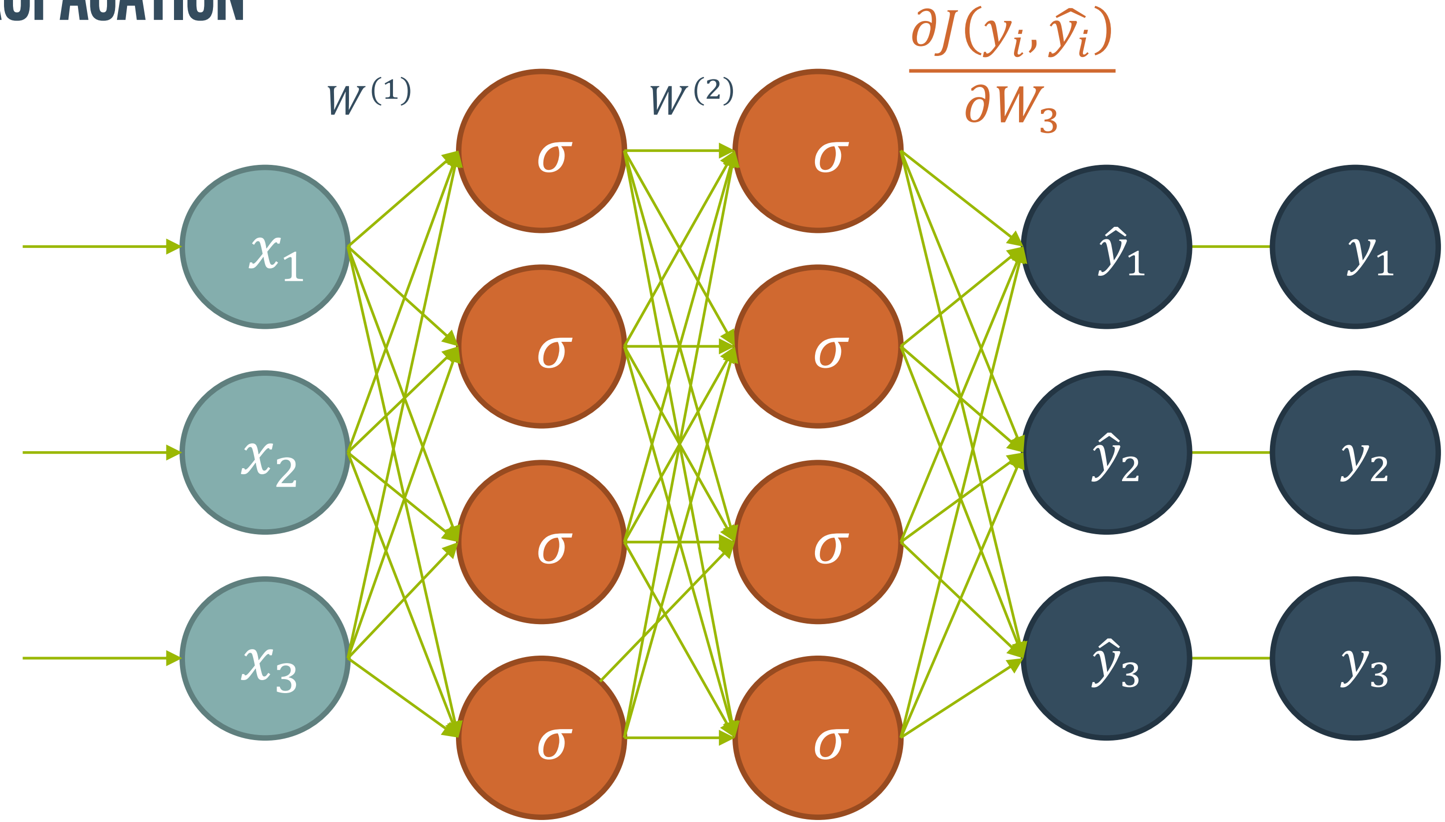
- Recall that: $\sigma'(z) = \sigma(z)(1-\sigma(z))$
- Though they appear complex, above are easy to compute!

BACKPROPAGATION

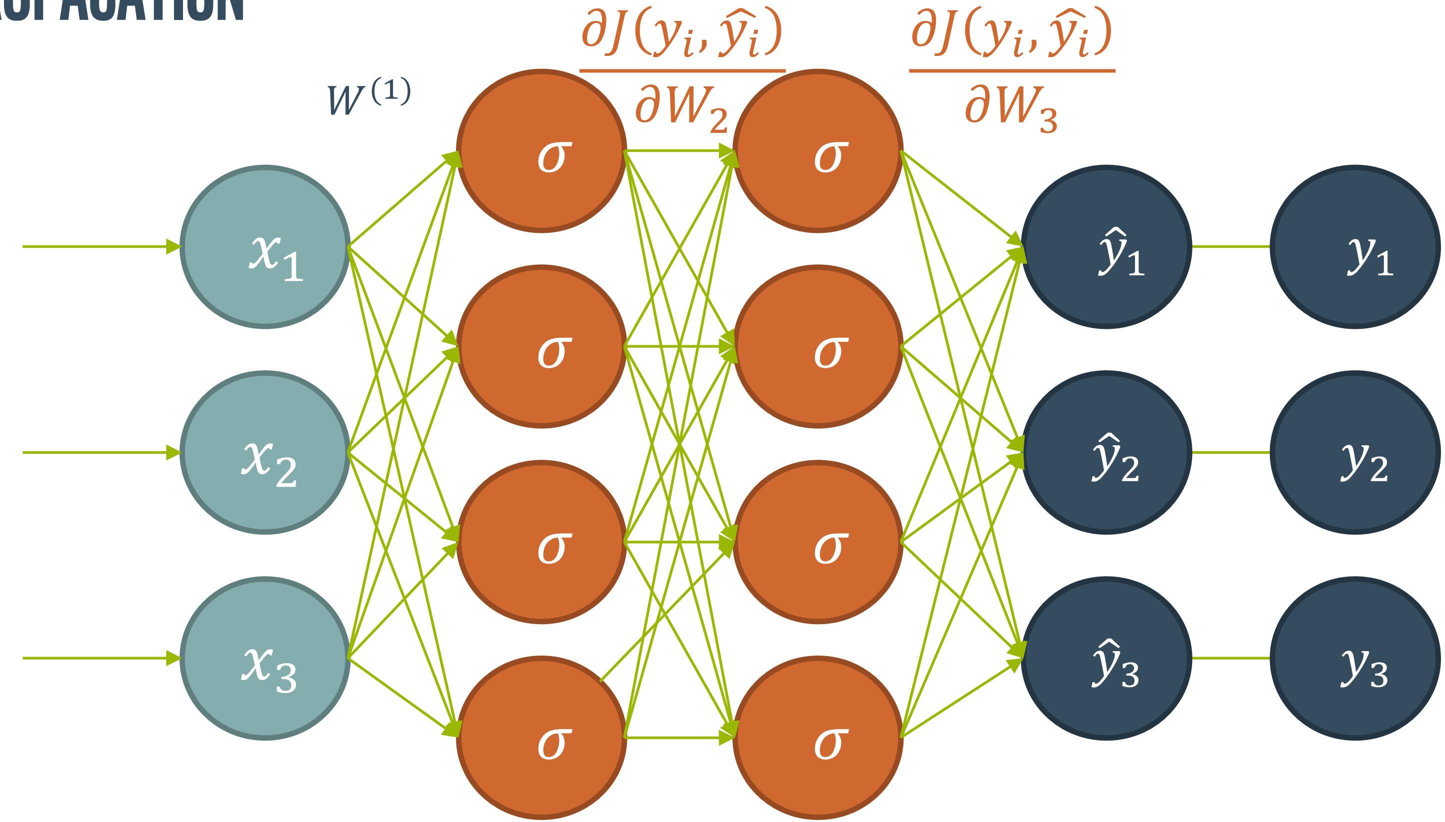
Want: $\dfrac{\partial J(y_i, \hat{y}_i)}{\partial W_k}$



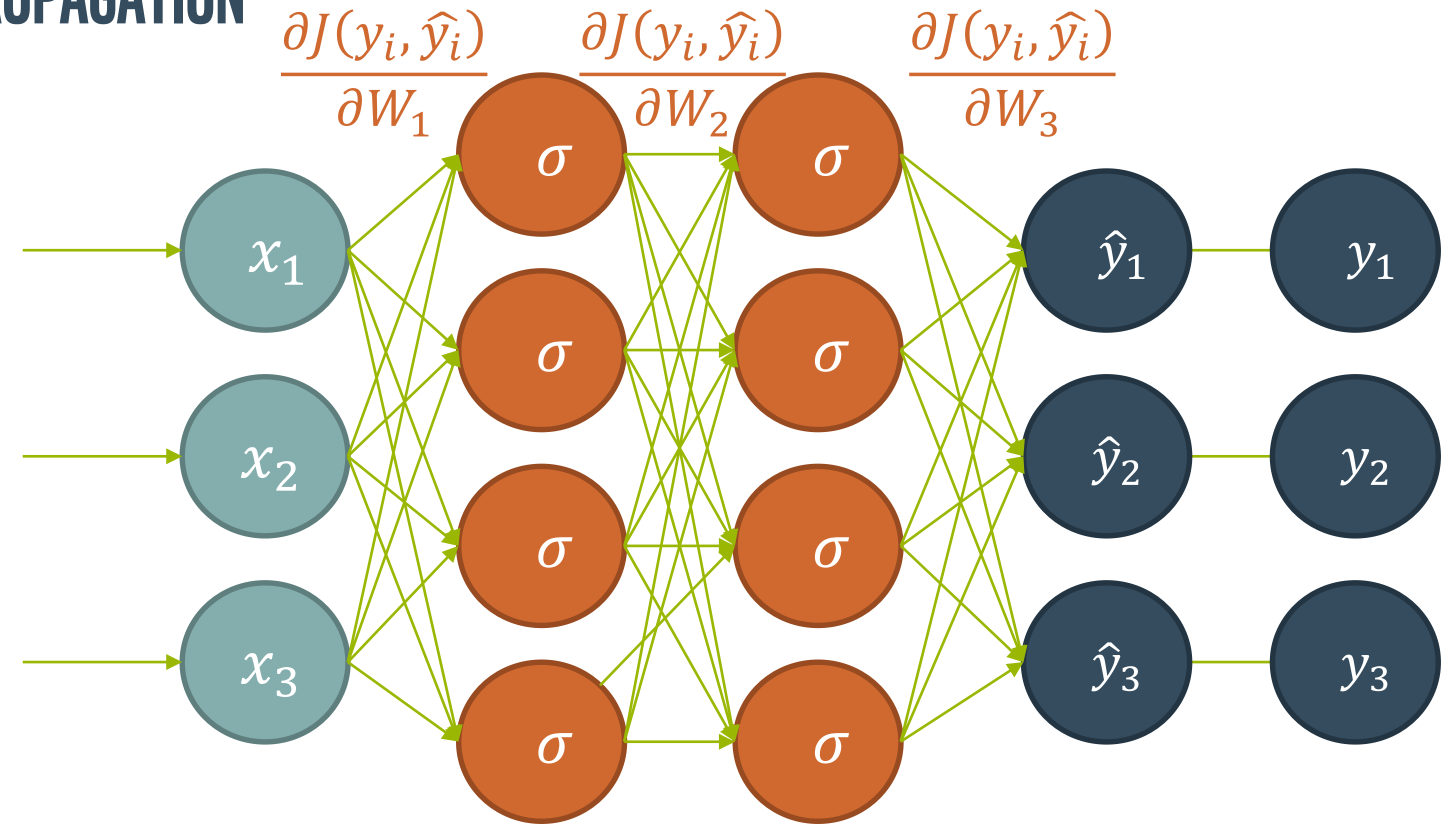
BACKPROPAGATION



BACKPROPAGATION



BACKPROPAGATION



HOW HAVE WE TRAINED BEFORE?

Gradient Descent!

1. Make prediction
2. Calculate Loss
3. Calculate gradient of the loss function w.r.t. parameters
4. **Update parameters by taking a step in the opposite direction**
5. Iterate

VANISHING GRADIENTS

Recall that:

$$\frac{\partial J}{\partial W^{(1)}} = (\hat{y} - y) \cdot W^{(3)} \cdot \sigma'\left(z^{(3)}\right) \cdot W^{(2)} \cdot \sigma'\left(z^{(2)}\right) \cdot X$$

- Remember: $\sigma'(z) = \sigma(z)(1-\sigma(z)) \leq .25$
- As we have more layers, the gradient gets very small at the early layers.
- This is known as the "vanishing gradient" problem.
- For this reason, other activations (such as ReLU) have become more common.

OTHER ACTIVATION FUNCTIONS

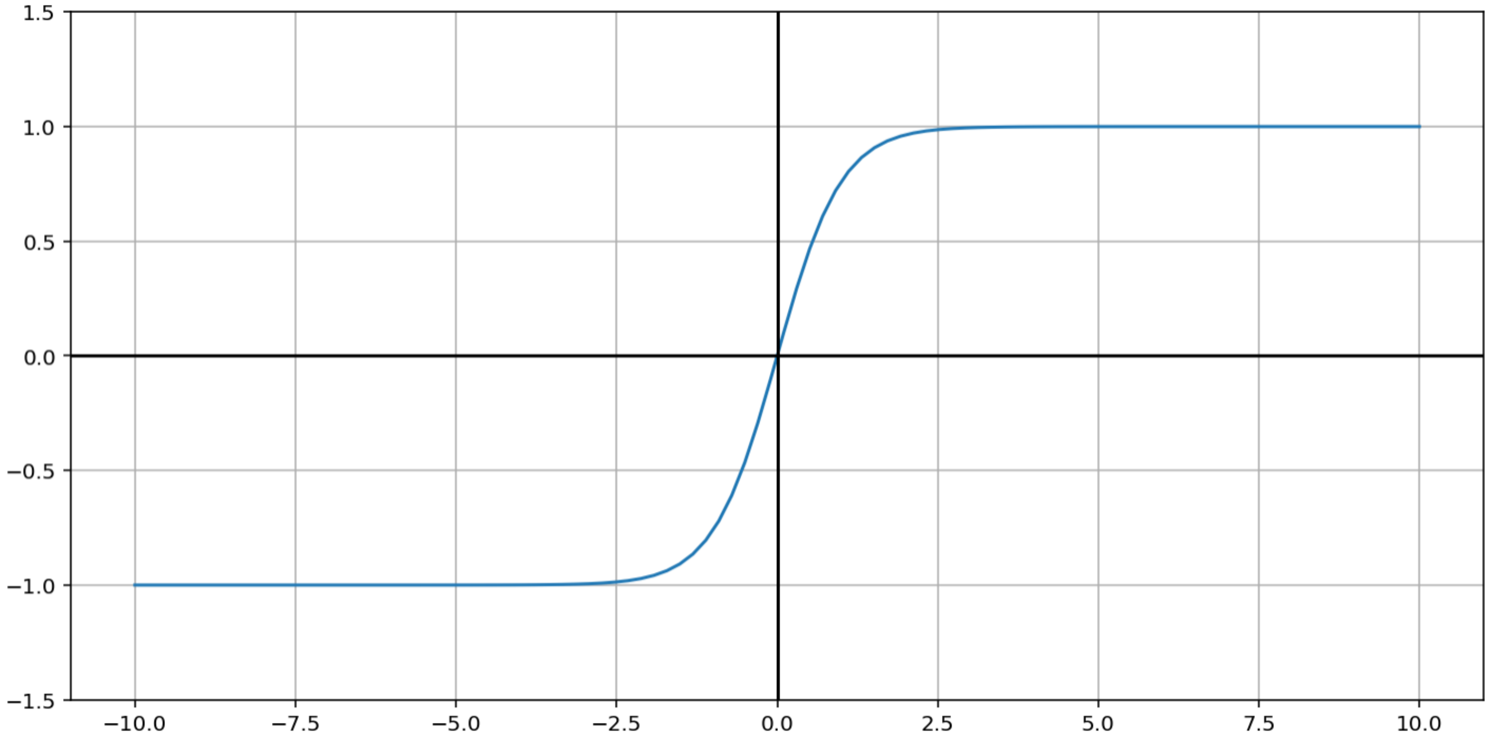
HYPERBOLIC TANGENT FUNCTION

- Hyperbolic tangent function
- Pronounced “tanch”

$$tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

$$tanh(0) = 0$$
$$tanh(\infty) = 1$$
$$tanh(-\infty) = -1$$

HYPERBOLIC TANGENT FUNCTION



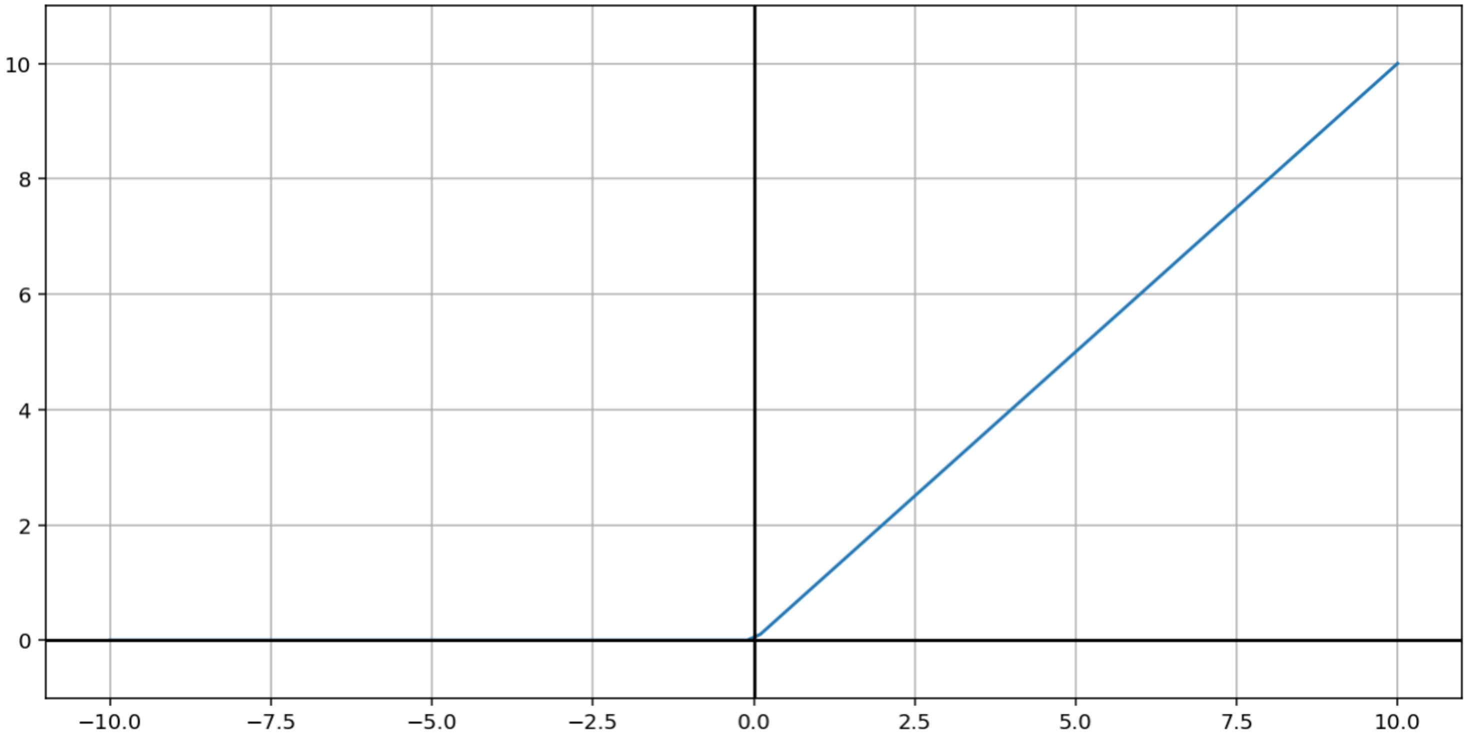
RECTIFIED LINEAR UNIT (RELU)

$$ReLU(z) = \begin{cases} 0, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$= \max(0, z)$$

$$ReLU(0) = 0$$
$$ReLU(z) = z \qquad \text{for } (z \gg 0)$$
$$ReLU(-z) = 0$$

RECTIFIED LINEAR UNIT (RELU)



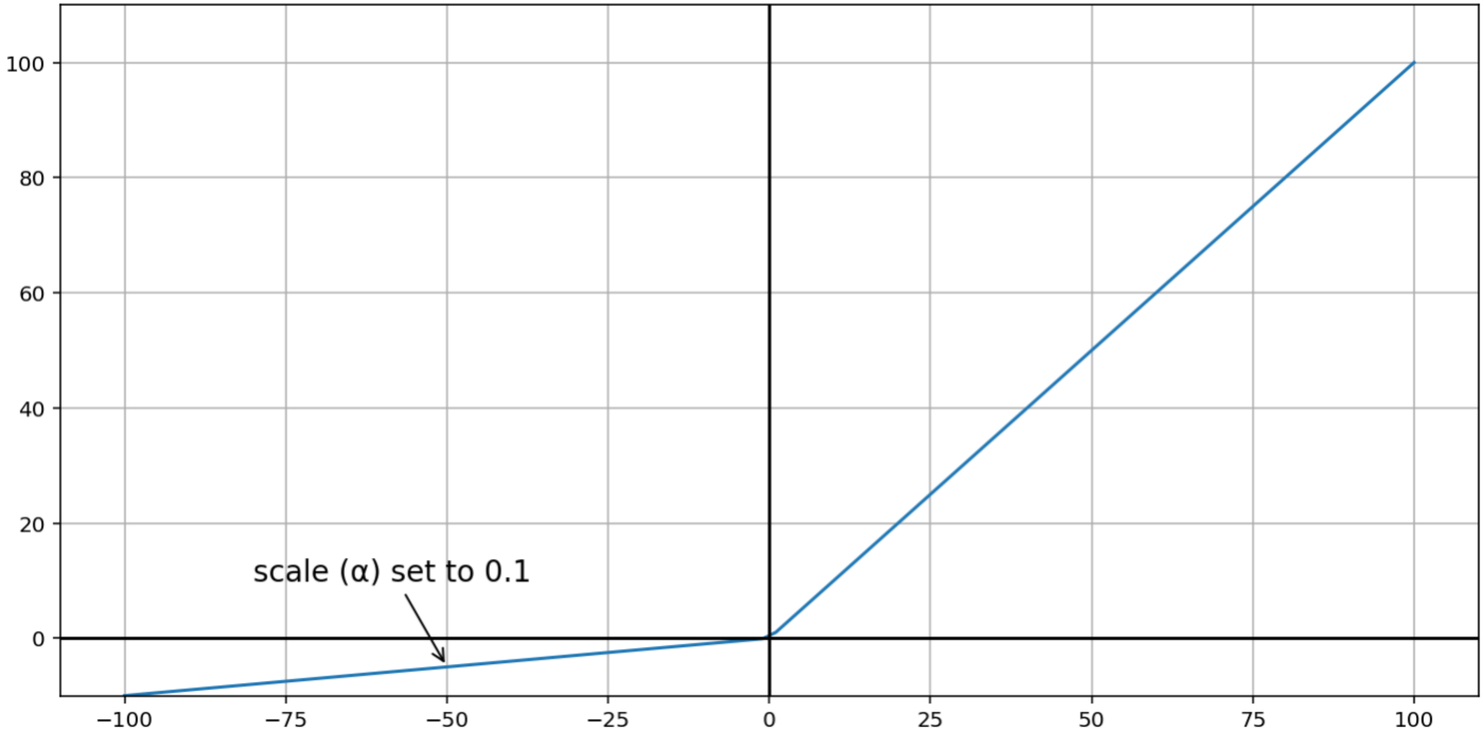
"LEAKY" RECTIFIED LINEAR UNIT (RELU)

$$LReLU(z) = \begin{cases} \alpha z, & z < 0 \\ z, & z \geq 0 \end{cases}$$

$$= \max(\alpha z, z) \qquad \text{for } (\alpha < 1)$$

$$LReLU(0) = 0$$
$$LReLU(z) = z \qquad \text{for } (z \gg 0)$$
$$LReLU(-z) = -\alpha z$$

"LEAKY" RECTIFIED LINEAR UNIT (RELU)



WHAT NEXT?

We now know how to make a single update to a model given some data.

But how do we do the full training?

We will dive into these details in the next lecture.

intel® Nervana™ AI Academy