Claudio Mirabello

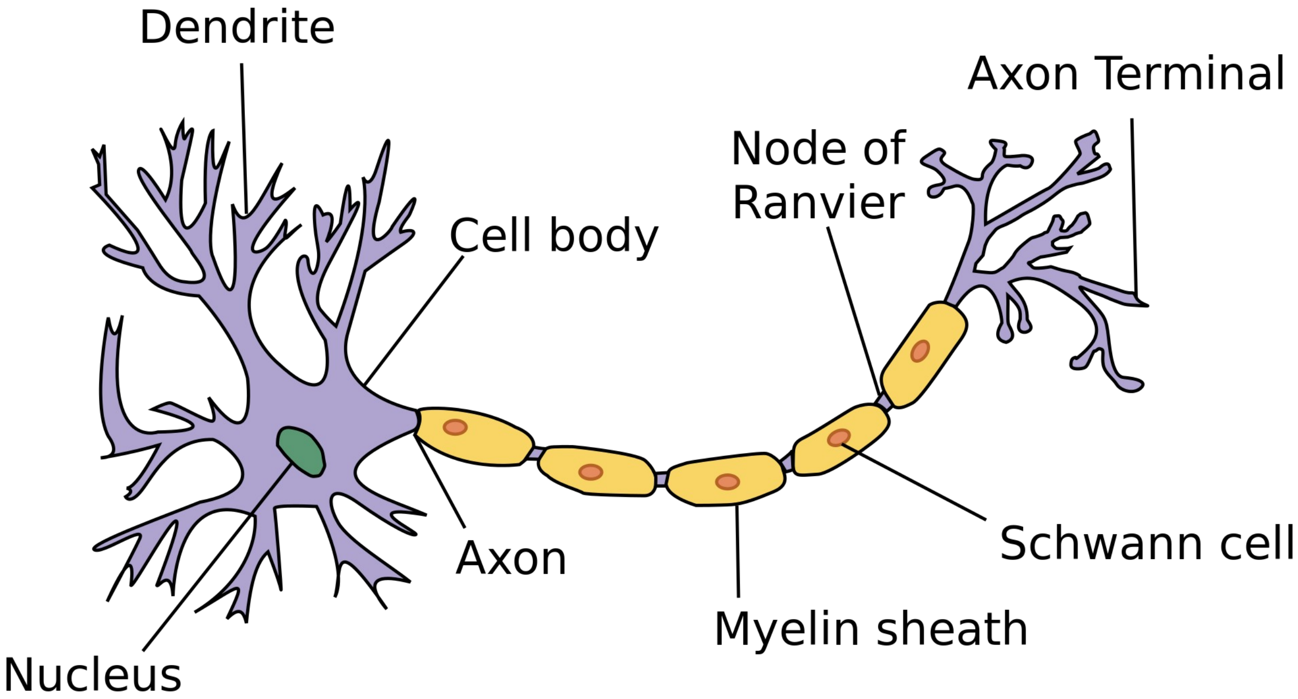
# Basic theory of Feed-forward Neural Networks

NBIS
NATIONAL BIOINFORMATICS INFRASTRUCTURE SWEDEN

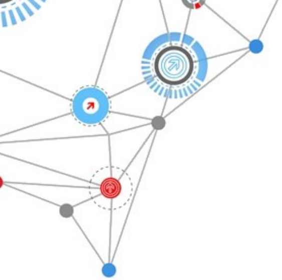
# Resources

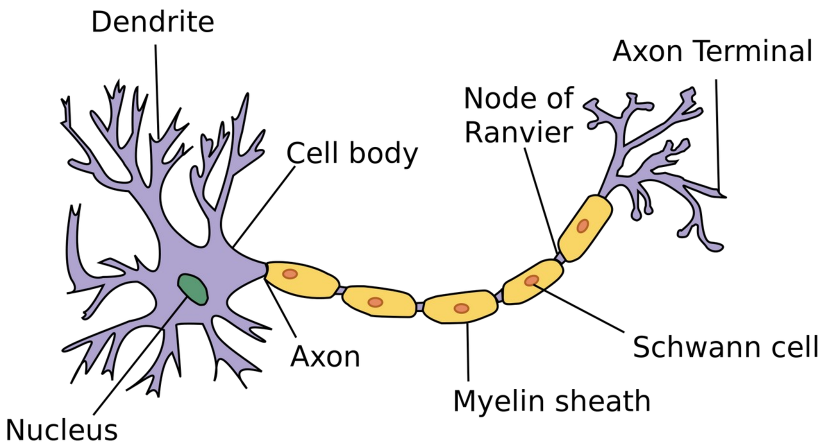
- MIT lectures on Deep Learning (http://introtodeeplearning.com/)
- TensorFlow Playground (https://playground.tensorflow.org)
- Keras Docs (https://keras.io)

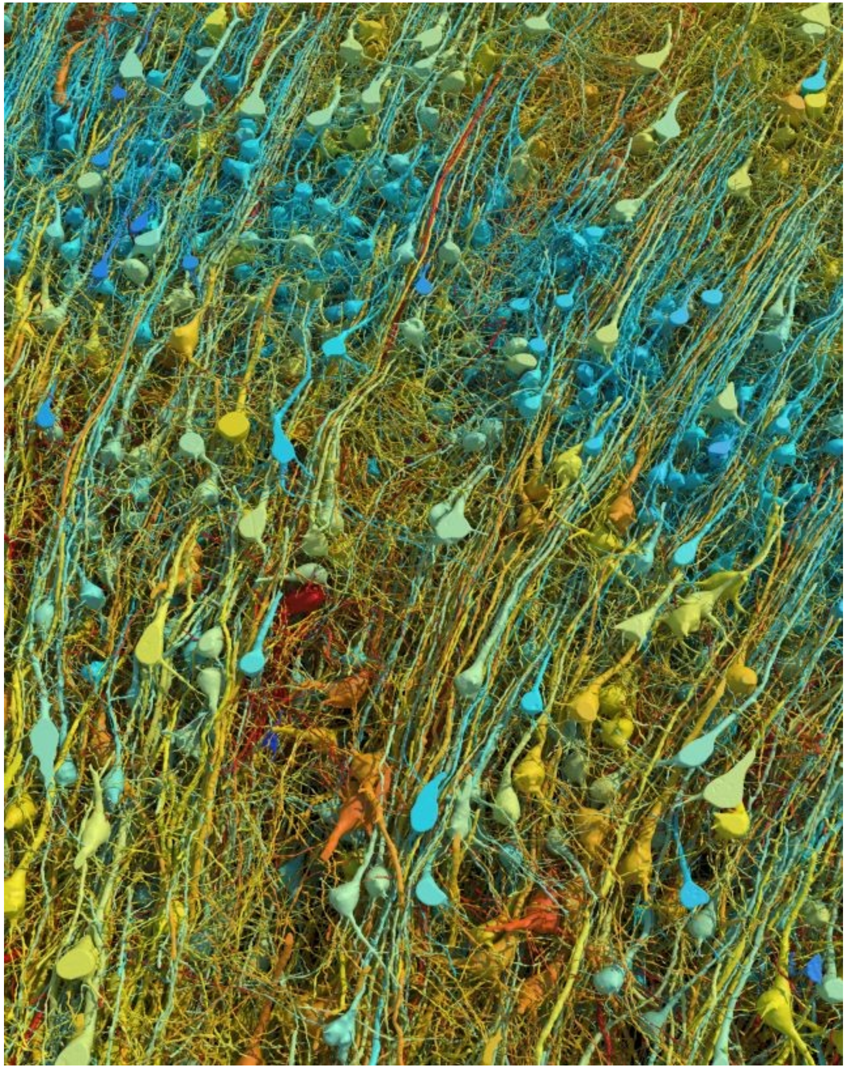
# This is a neuron




wikipedia.org

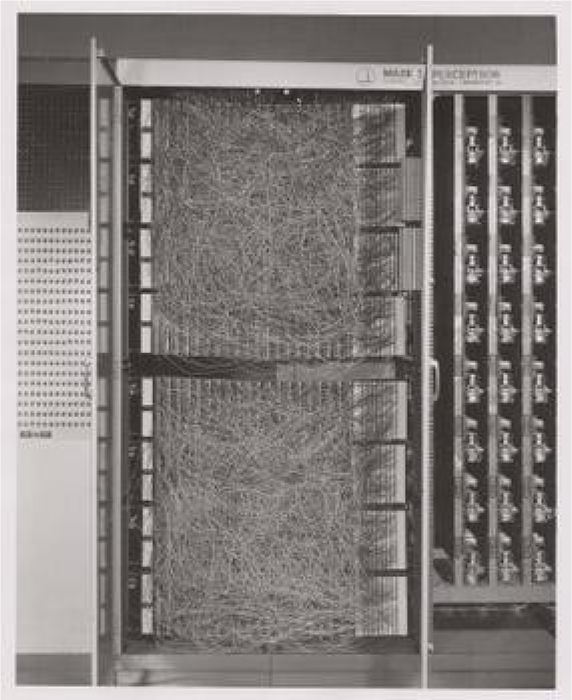




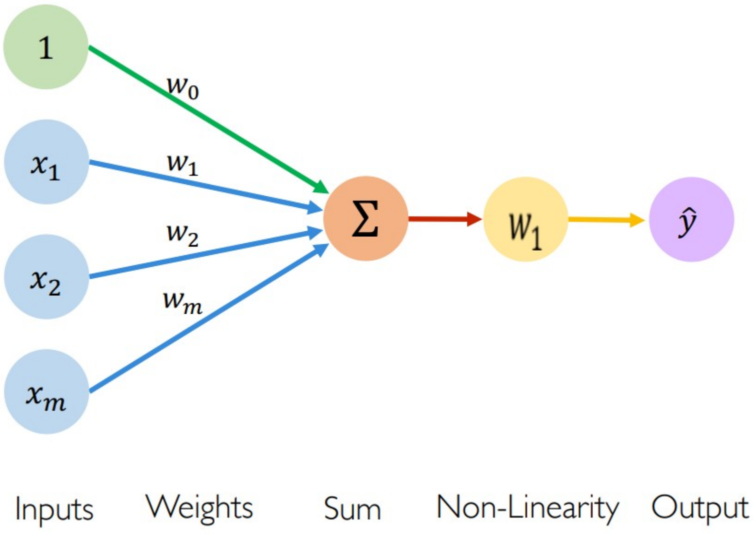



Shapson-Coe, Alexander, et al. "A petavoxel fragment of human cerebral cortex reconstructed at nanoscale resolution." Science 384.6696 (2024): eadk4858

# This is a perceptron (1958)



Mark I Perceptron machine
wikipedia.org



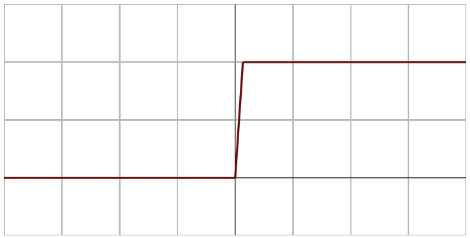

Output

Linear combination of inputs

$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i\, w_i\right)$$

Non-linear activation function

Bias



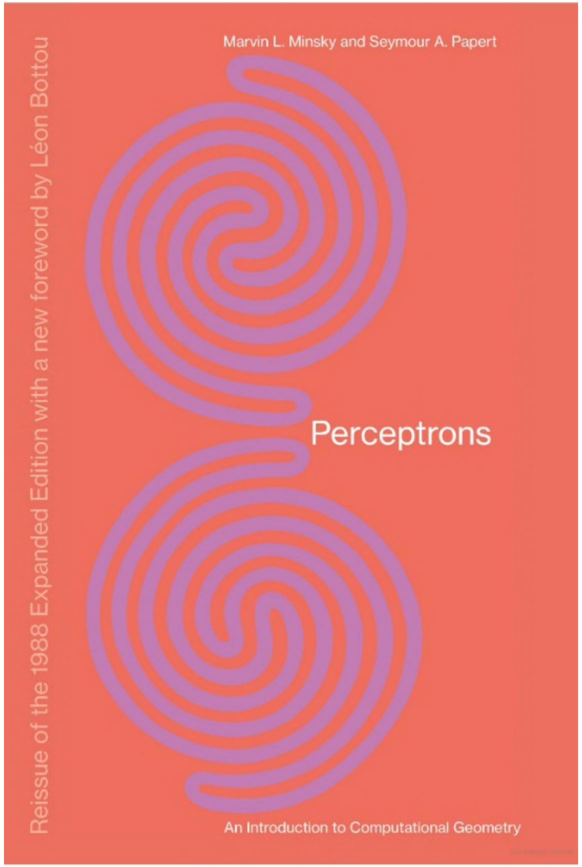
MIT “Intro to Deep Learning”

# Perceptrons caused excitement

- *"the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."*

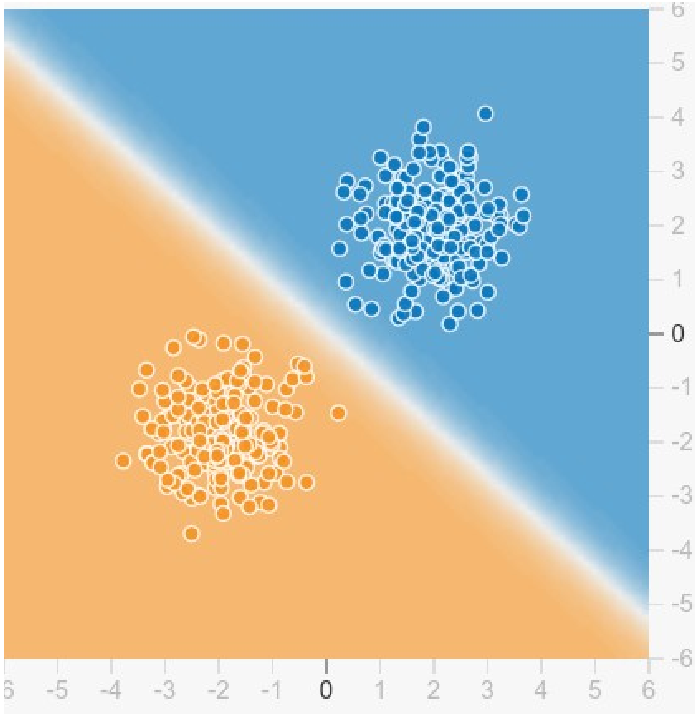
The New York Times

# Perceptrons can only learn linearly separable classes



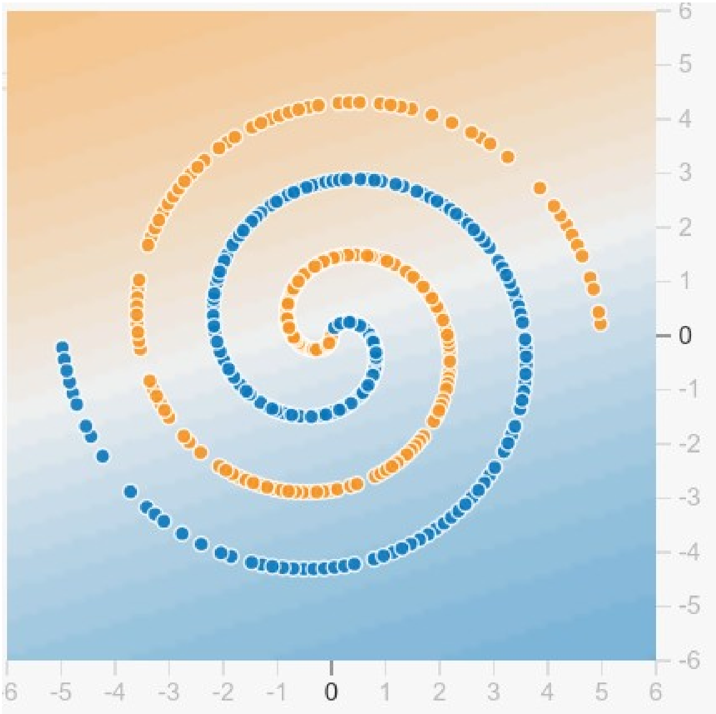

Minsky and Papert, 1969

# Perceptrons can only learn linearly separable classes



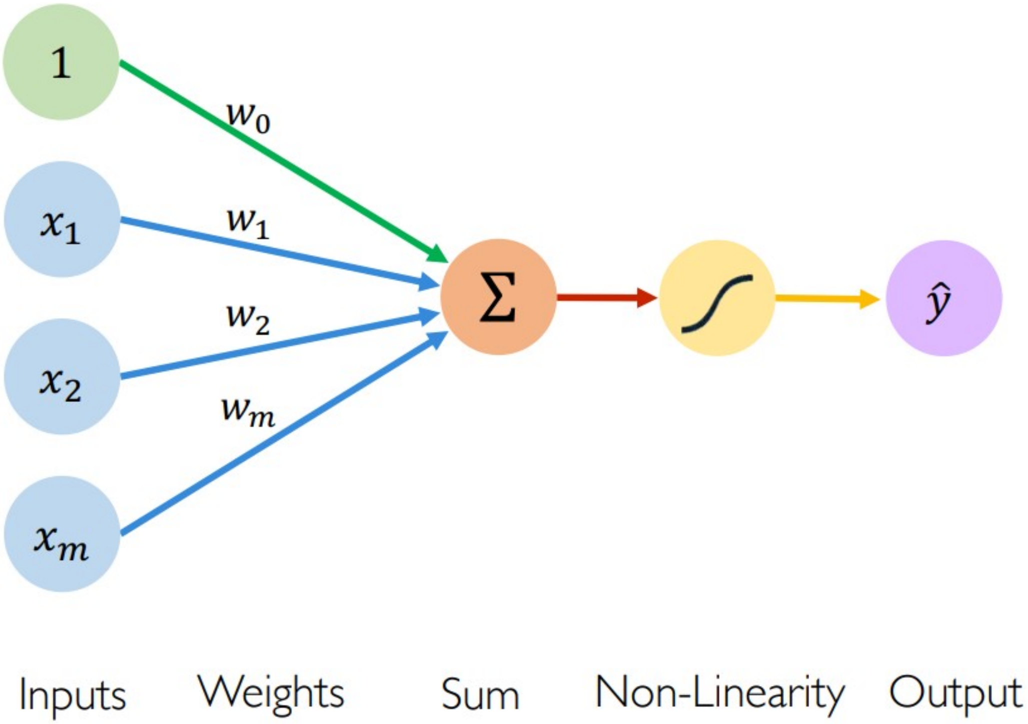

TensorFlow Playground

# But sometimes you want to model non-linear functions



How do we make this non-linear then?
Two ingredients to add

# 1: Differentiable, non-linear activation functions

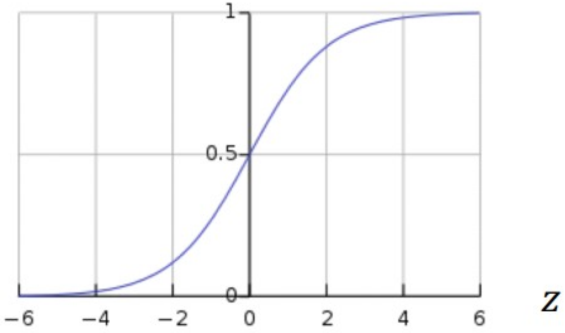



## Activation Functions

$$\hat{y} = g\left( w_0 + \boldsymbol{X}^T \boldsymbol{W} \right)$$

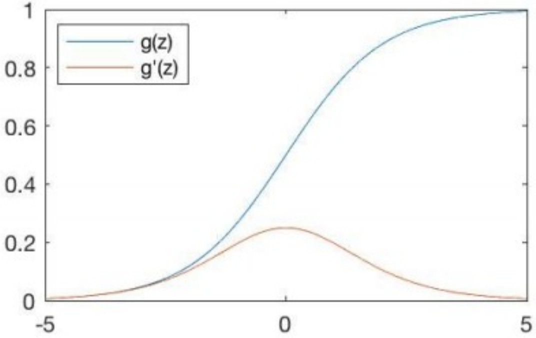
- Example: sigmoid function

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



# Common activation functions

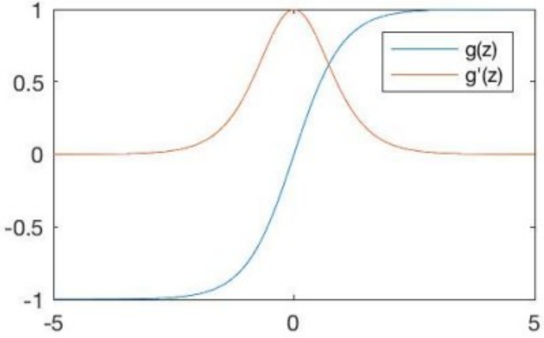
### Sigmoid Function




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

$$g'(z) = g(z)(1 - g(z))$$

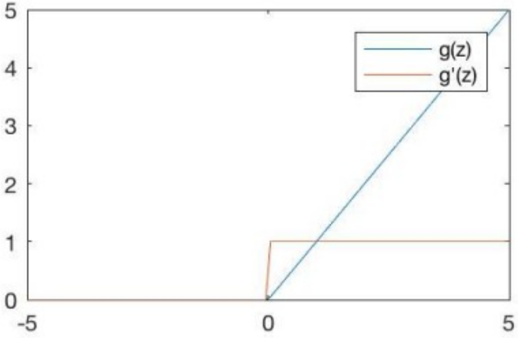
### Hyperbolic Tangent




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

$$g'(z) = 1 - g(z)^2$$

### Rectified Linear Unit (ReLU)




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

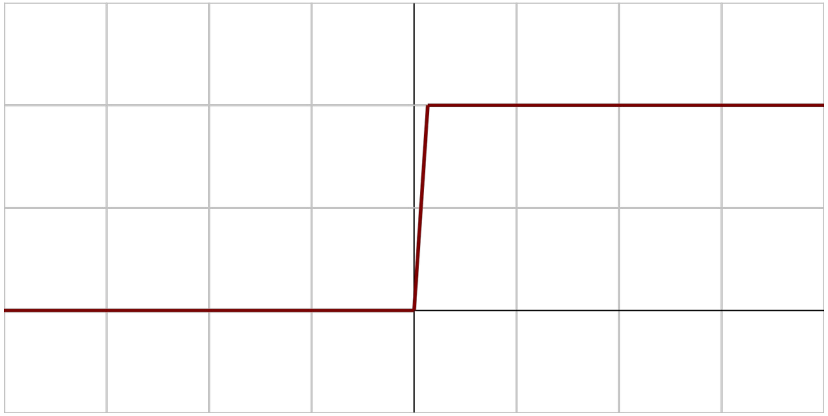
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

# Special case: softmax

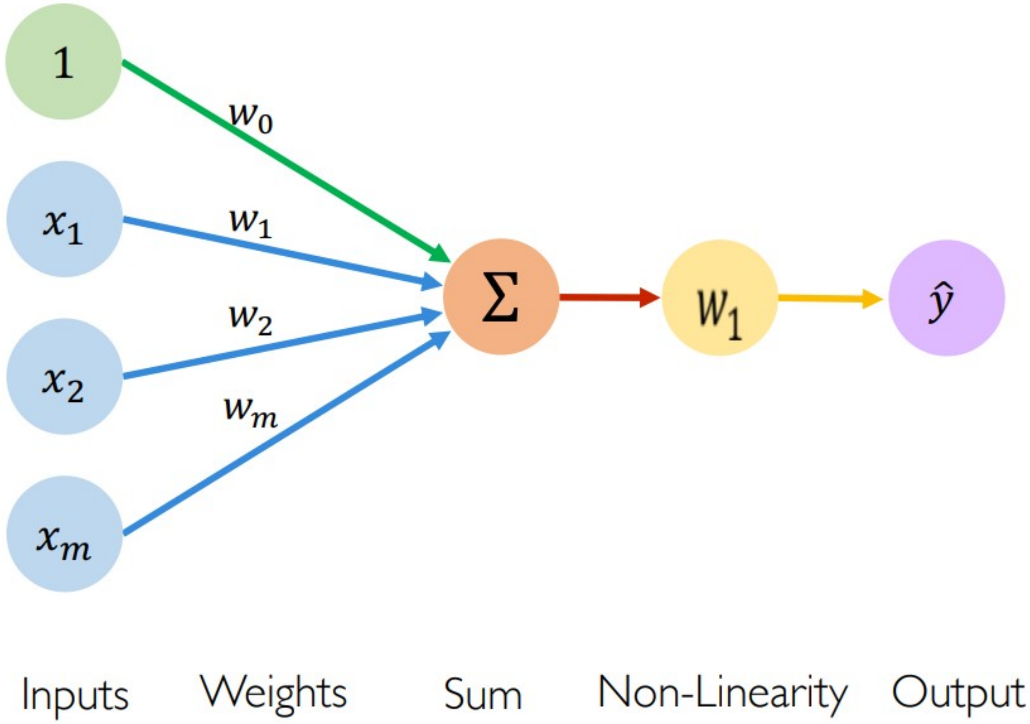
- Used in classification problems
- Given k classes, it decides which one is more likely
- One output per class, each output is assigned a probability from 0 to 1
- The sum of probabilities for all outputs is 1

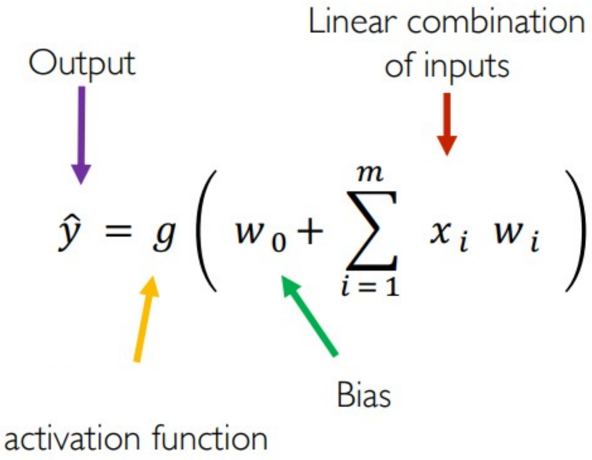
$$g(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{k} e^{z_k}} \; for\; j = 1, \ldots, k$$

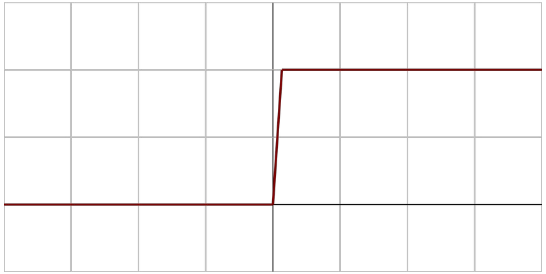
# Wait a second, the perceptron already has a non-linear (step) activation function!



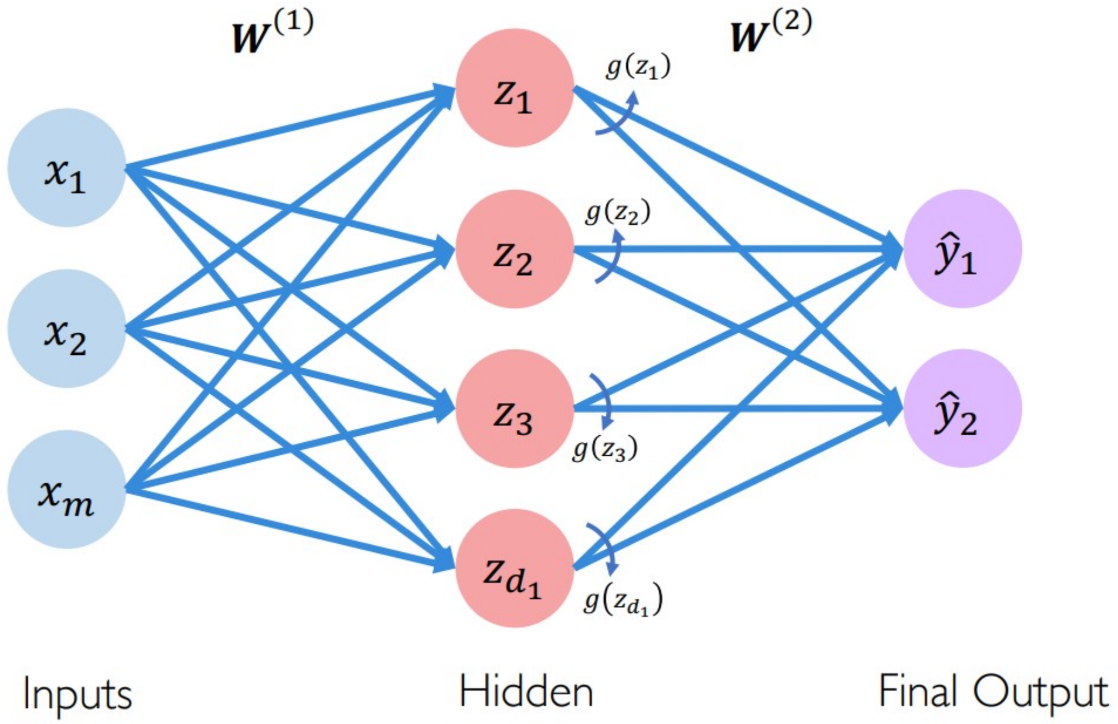
# This is a perceptron



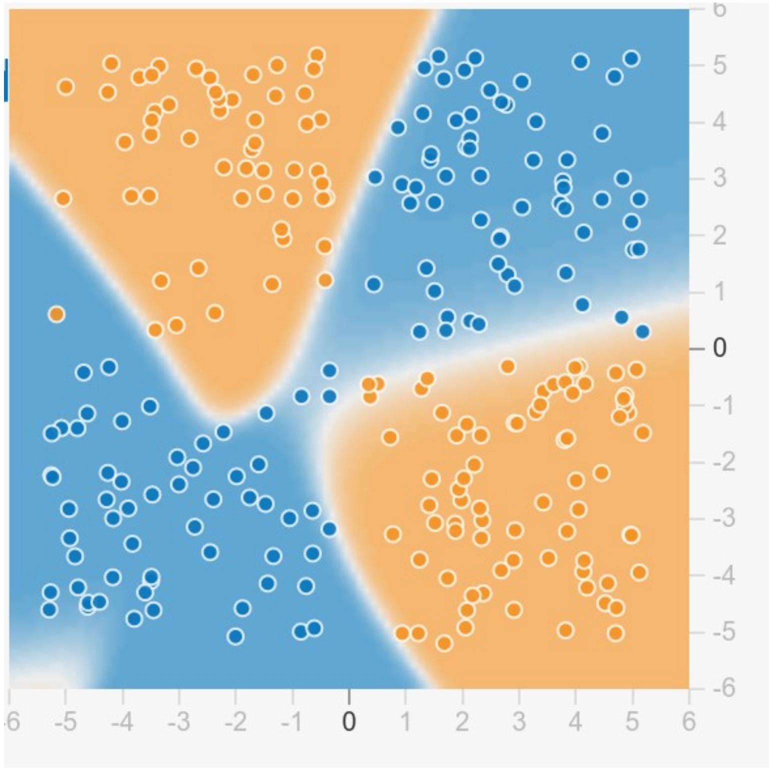



# 2: Multi-layer Perceptron (1986)

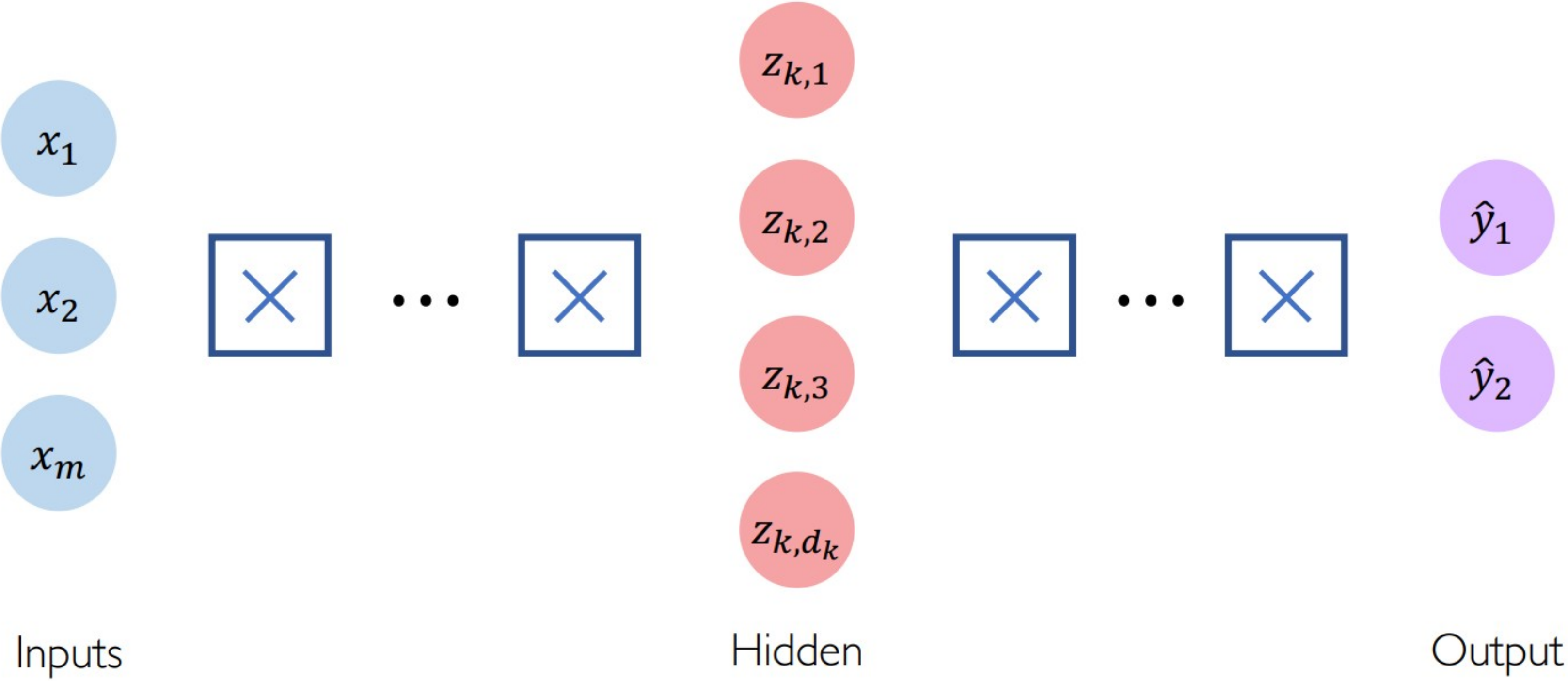



$$z_i = w_{0,i}^{(1)} + \sum_{j=1}^{m} x_j\, w_{j,i}^{(1)} \qquad \hat{y}_i = g\left(w_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j\, w_{j,i}^{(2)}\right)$$

# Now we're getting somewhere

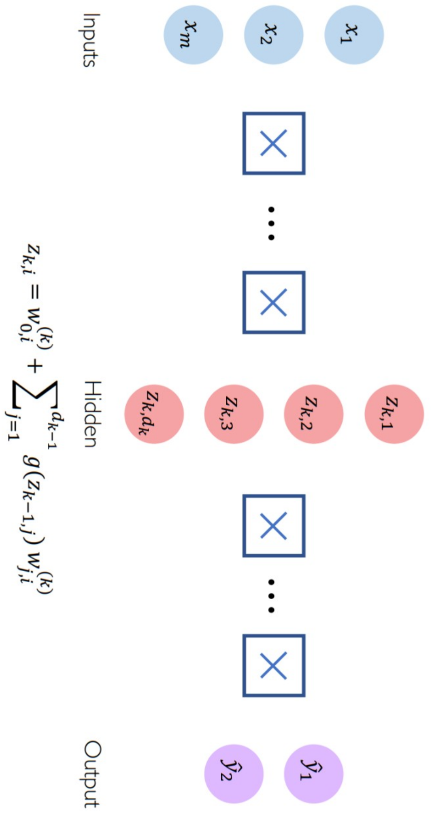


# Why stop at one hidden layer?



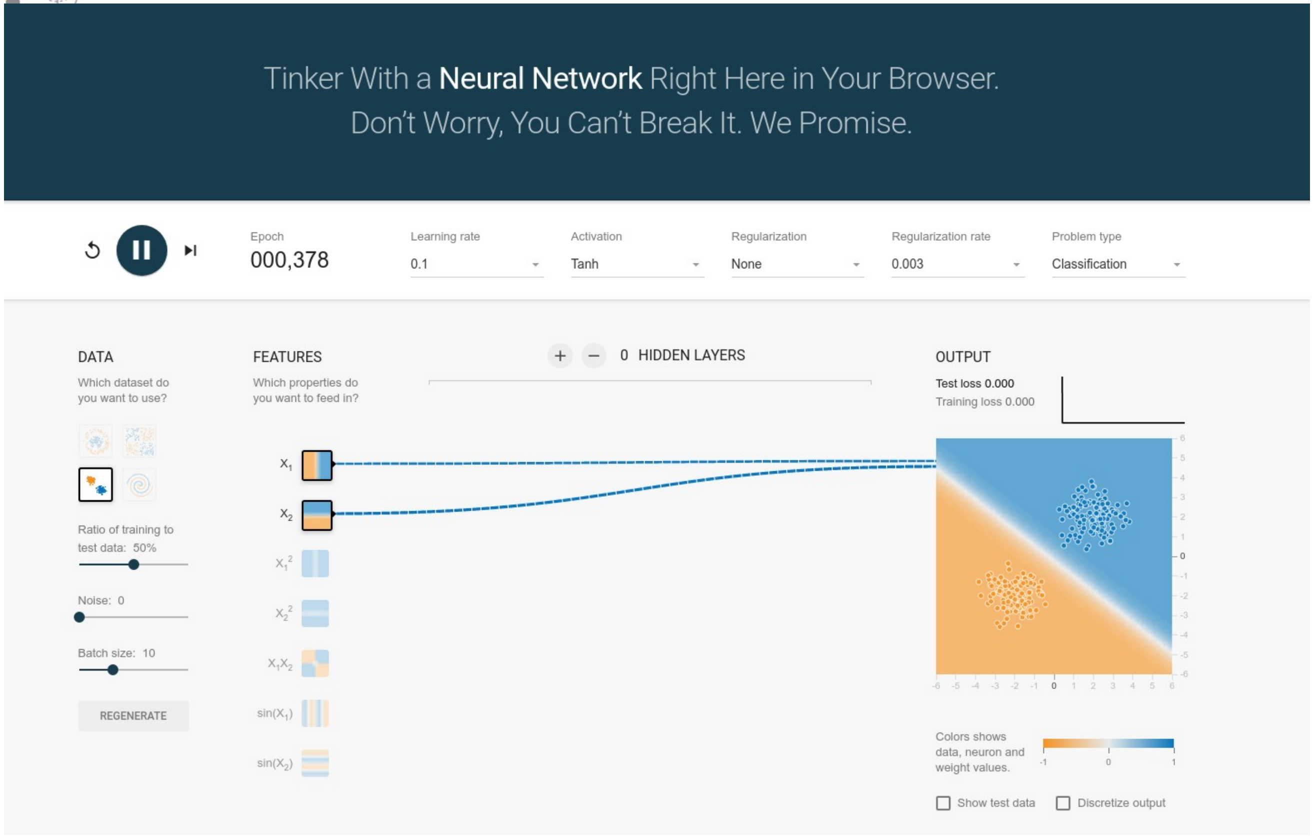

$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

# Deep Networks are simply NNs with multiple hidden layers




Deeper
this way

# https://playground.tensorflow.org




Let's review:

- Perceptron
- XOR problem
- Activations
- Multi-layer perceptron

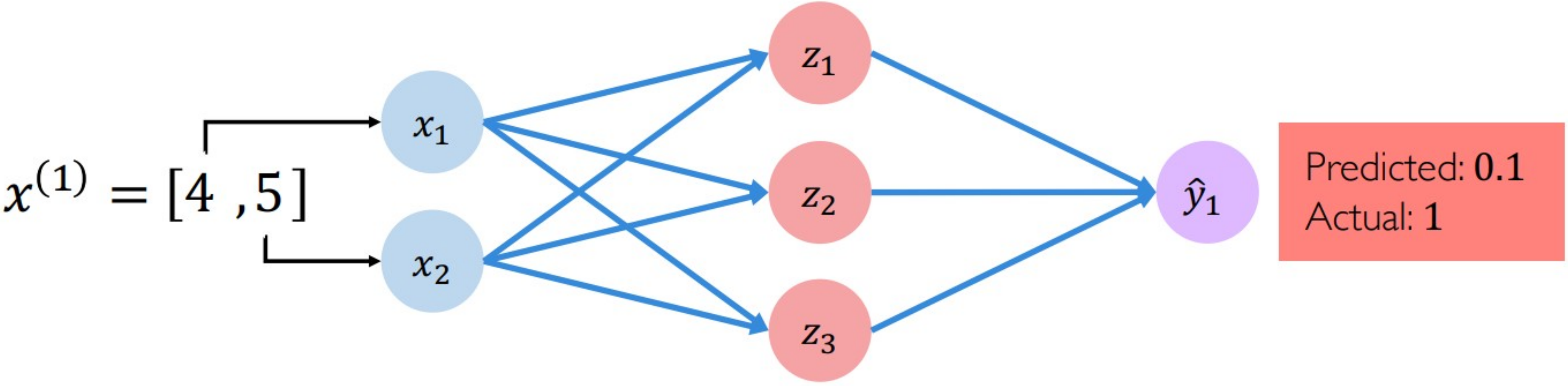
# How do we decide which weights are optimal?

- A linear regressor's weights (coefficients) are calculated in closed form

- This can't be done if you have hidden layers and non-linear activations

# How do we decide which weights are optimal?

*The **loss** of our network measures the cost incurred from incorrect predictions*

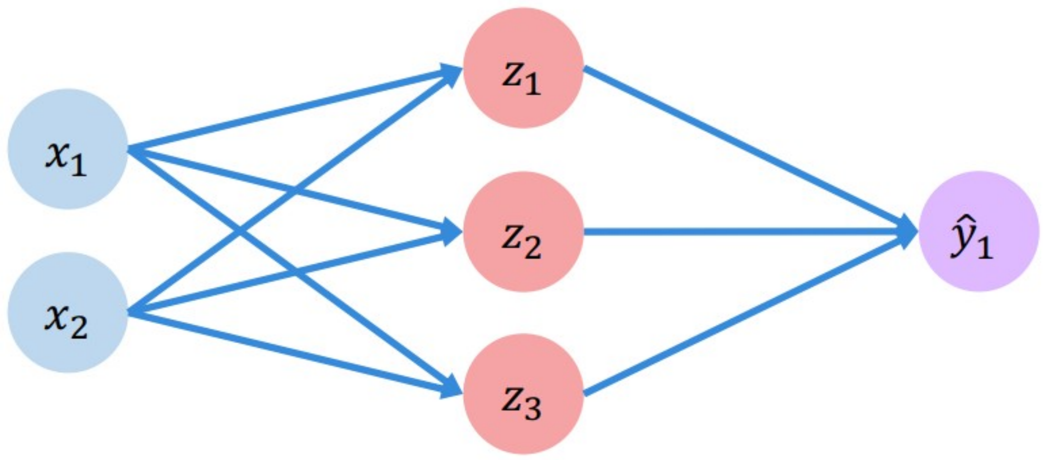



$$\mathcal{L}\big(\underbrace{f\big(x^{(i)}; \boldsymbol{W}\big)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\big)$$

# How do we decide which weights are optimal?

*The* ***empirical loss*** *measures the total loss over our entire dataset*

$$\boldsymbol{X} = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}$$




$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} \qquad y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix}$$

$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}},\underbrace{y^{(i)}}_{\text{Actual}}\Big)$$

Also known as:
- Objective function
- Cost function
- Empirical Risk

# Lower loss => better predictions

*We want to find the network weights that* ***achieve the lowest loss***

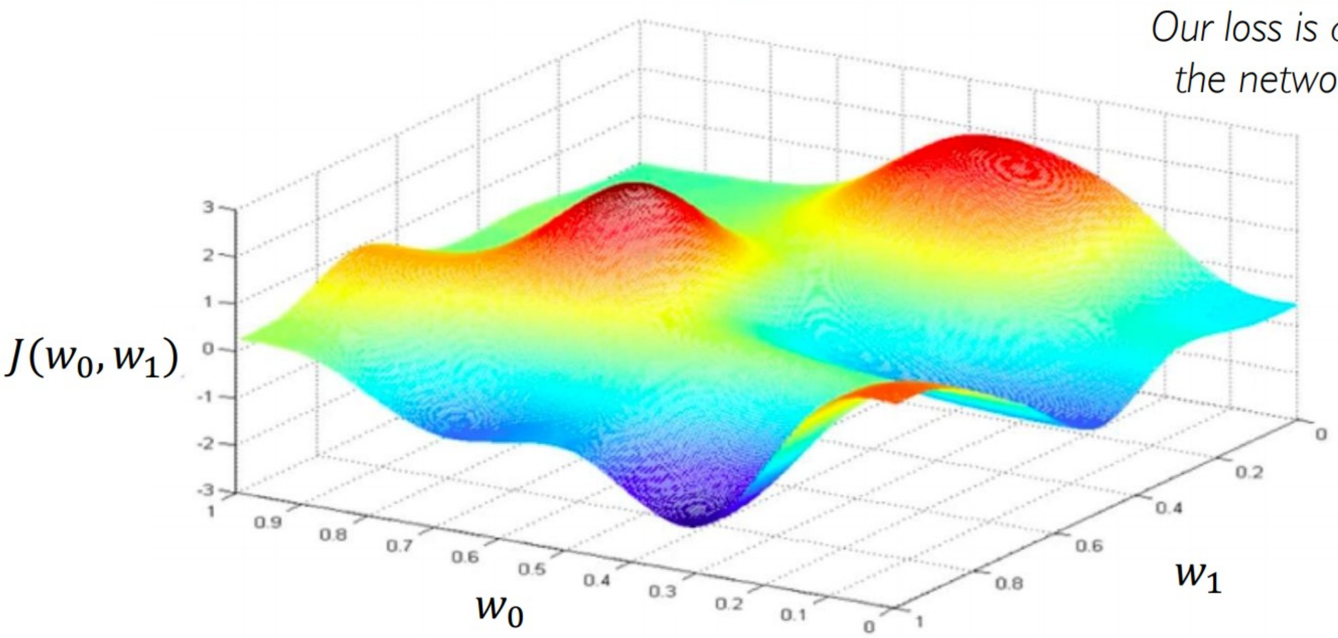
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, J(\boldsymbol{W})$$

# Minimizing loss

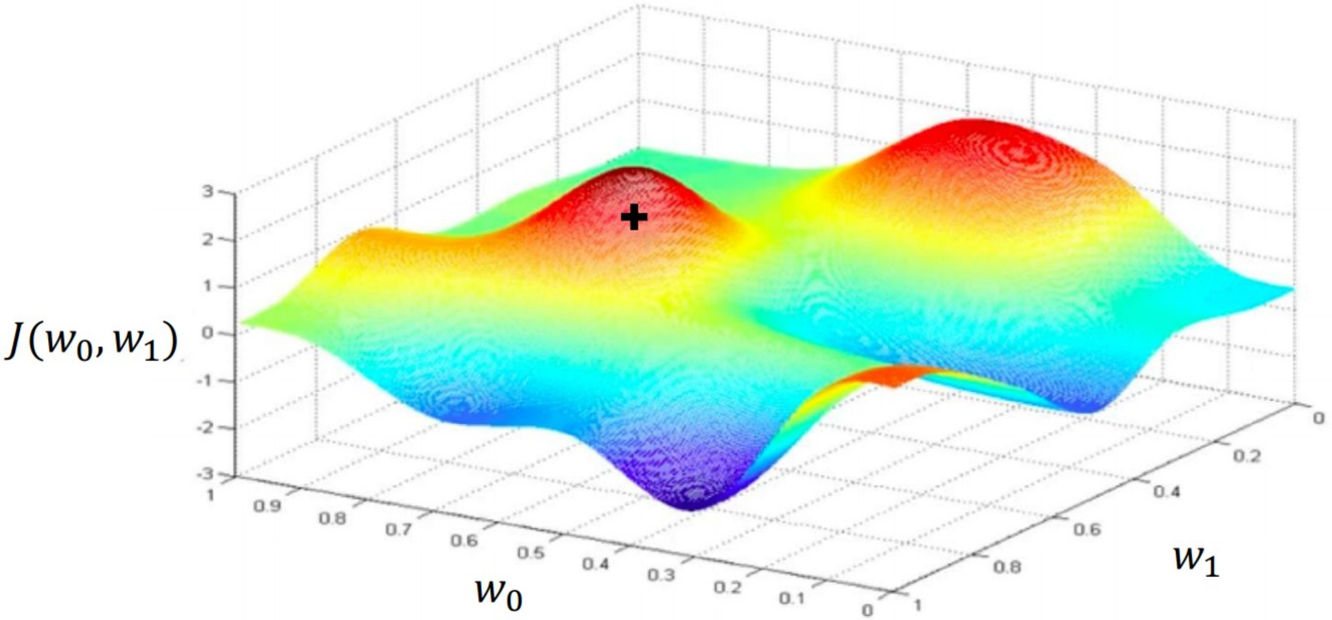
$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}}\, J(\boldsymbol{W})$$

Remember:
*Our loss is a function of the network weights!*



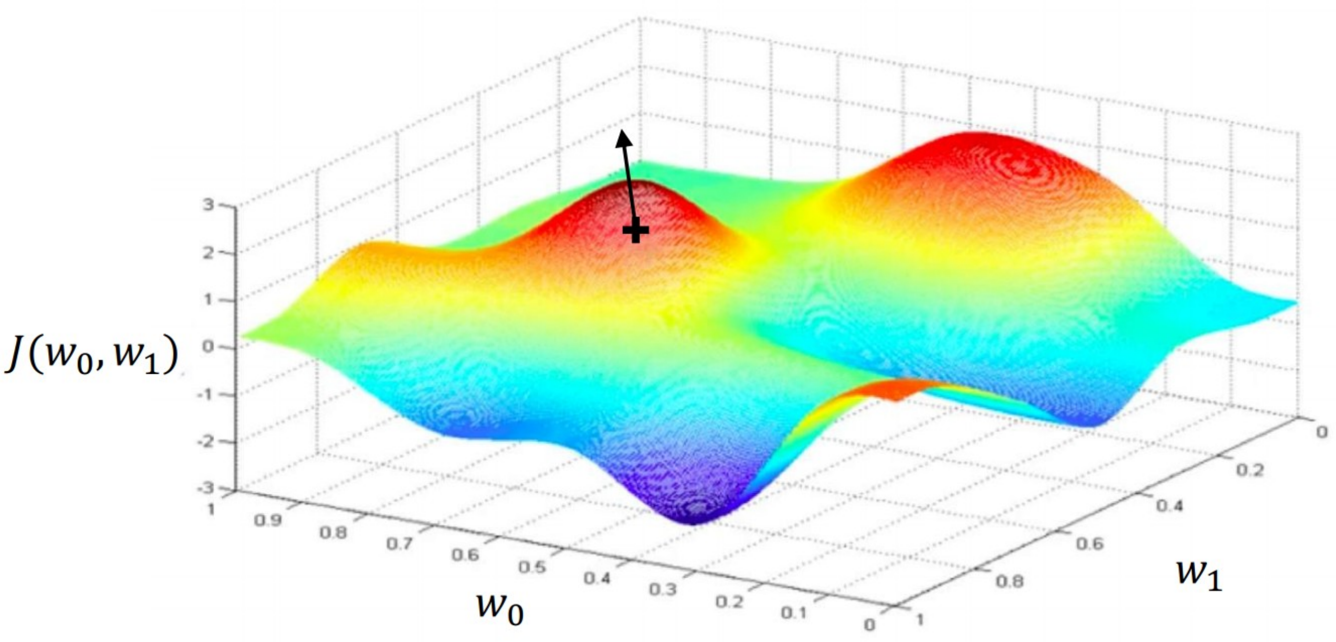
# Minimizing loss

Randomly pick an initial $(w_0, w_1)$



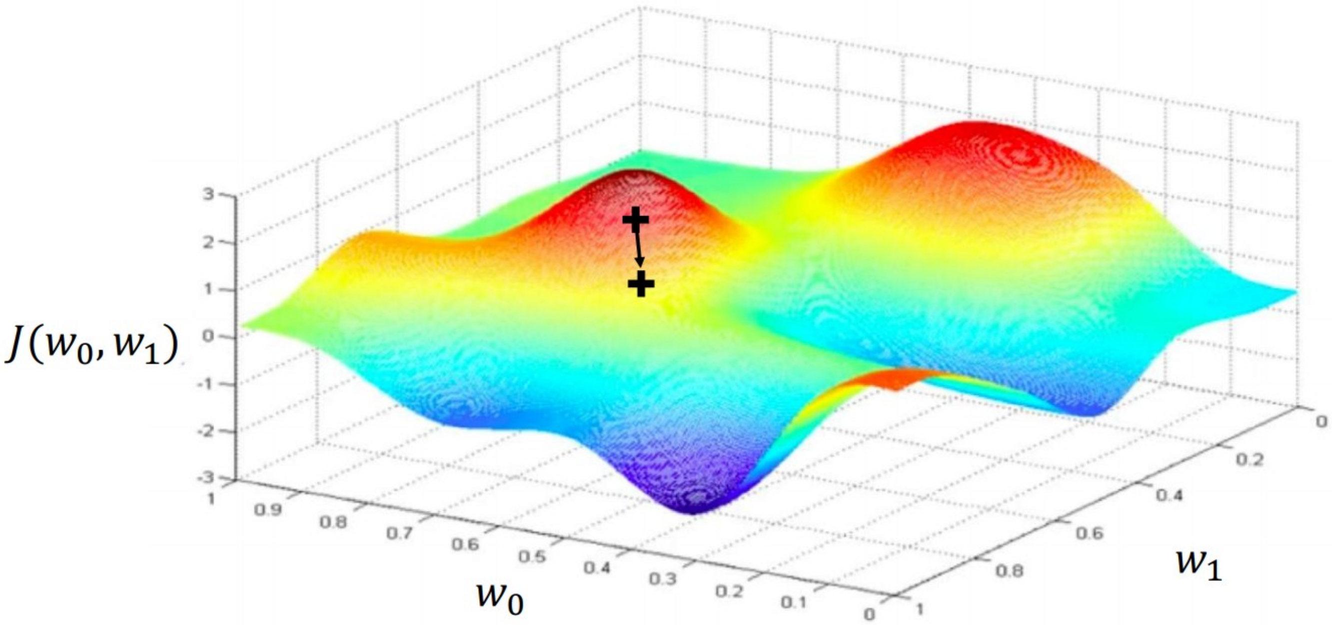
# Minimizing loss

Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

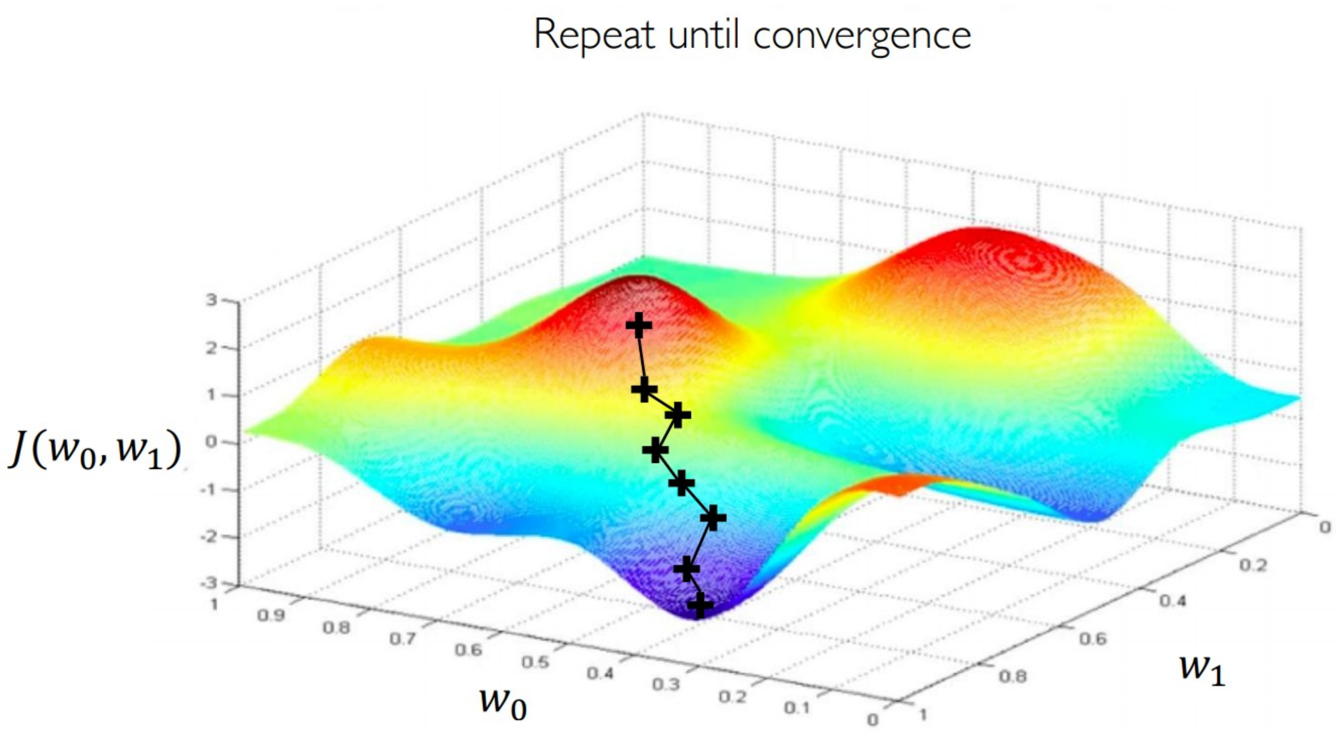


# Minimizing loss

Take small step in opposite direction of gradient



# Minimizing loss



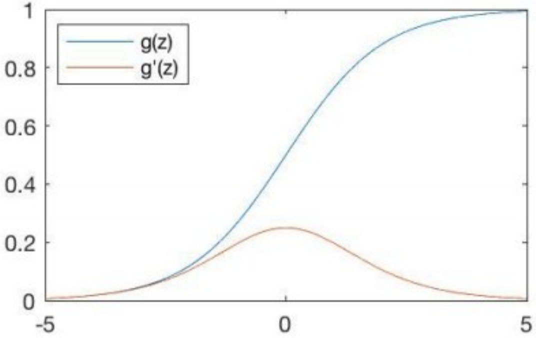
# Gradient descent

**Algorithm**

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

# Activation functions have to be differentiable!

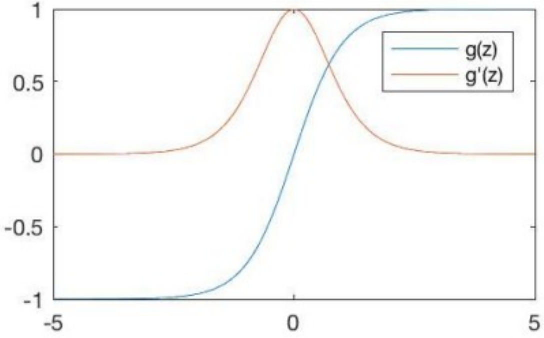
### Sigmoid Function




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

$$g'(z) = g(z)(1 - g(z))$$

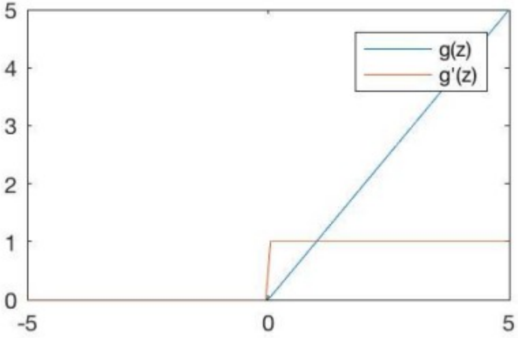
### Hyperbolic Tangent




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

$$g'(z) = 1 - g(z)^2$$

### Rectified Linear Unit (ReLU)




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

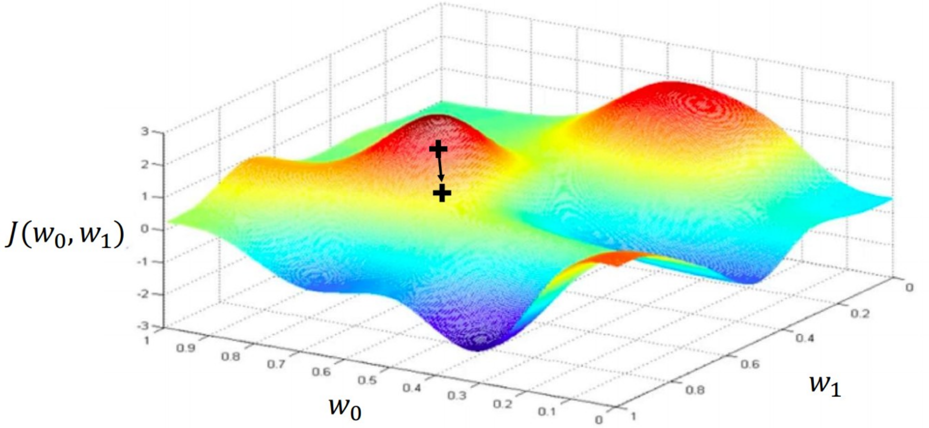
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

# The learning rate η

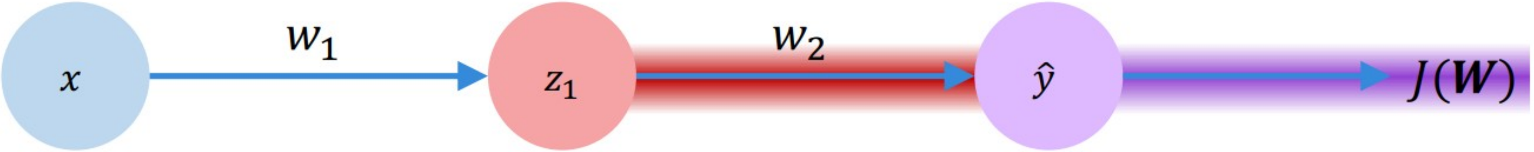
**Algorithm**

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\boxed{\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights

Take small step in opposite direction of gradient

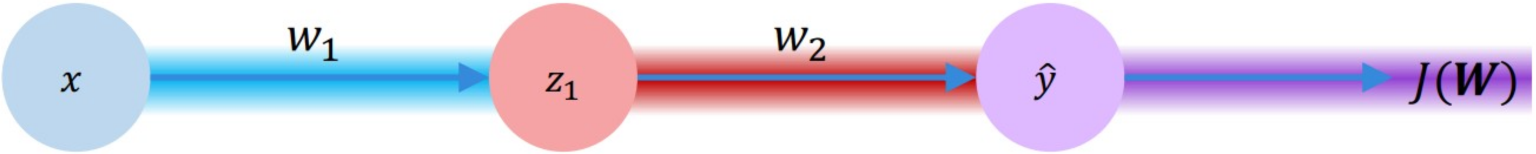


# Backpropagation




$$\frac{\partial J(\boldsymbol{W})}{\partial w_2} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

# Backpropagation

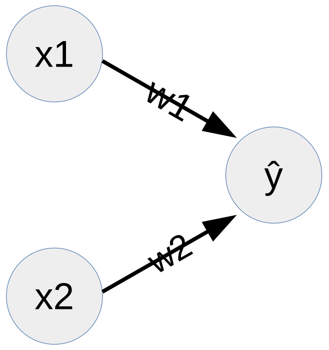


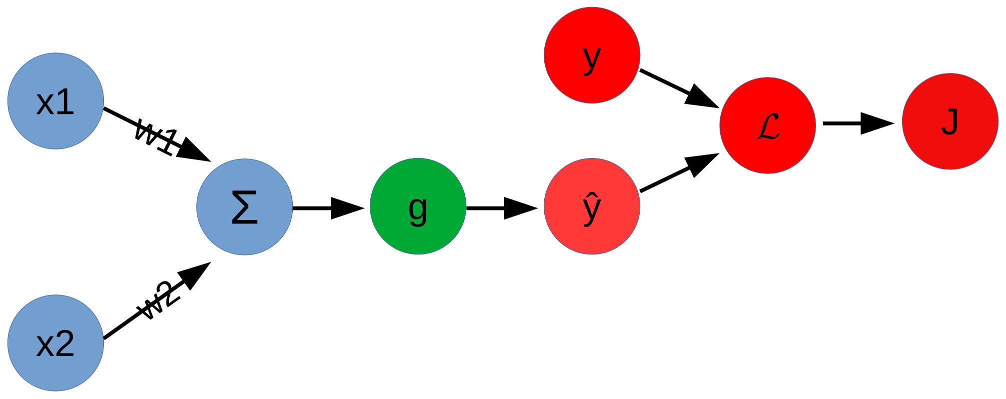

$$\frac{\partial J(\boldsymbol{W})}{\partial w_1} = \frac{\partial J(\boldsymbol{W})}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

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

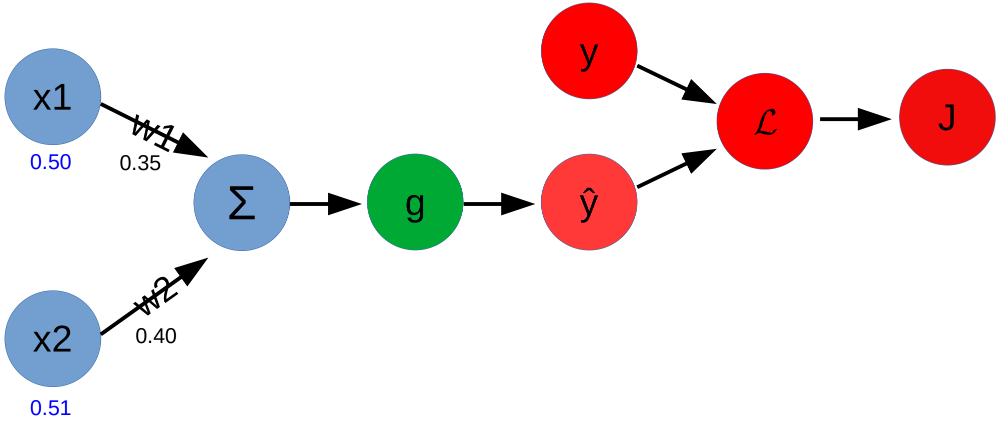
# Practical example

- Simple network:
  - Two inputs [x1, x2]
  - Two weights [w1, w2]
  - No bias
  - Activation function g( )
  - One output ŷ
  - One label y
  - Loss function $\mathcal{L}$( )
  - Weight-dependent error J(W)



# 1. Forward pass

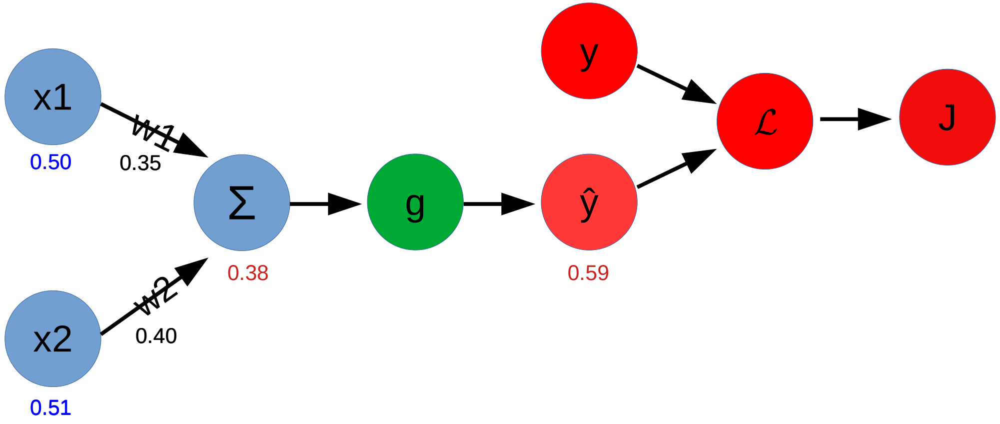



**x1** = 0.50
**x2** = 0.51

**w1** = 0.35
**w2** = 0.40

**Σ** = ?
**ŷ** = **g(Σ) =** $1/(1+e^{-\Sigma})$ = ?

# 1. Forward pass

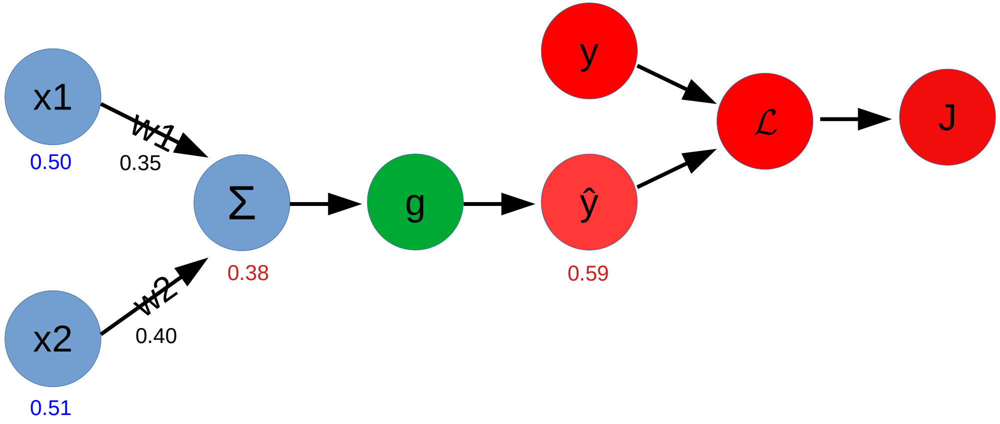



**x1** = 0.50
**x2** = 0.51

**w1** = 0.35
**w2** = 0.40

**Σ** = 0.35 * x1 + 0.4 * x2 = 0.38
**ŷ** = g(Σ) = $1/(1+e^{-\Sigma})$ = 0.59

# 2. Calculate Loss



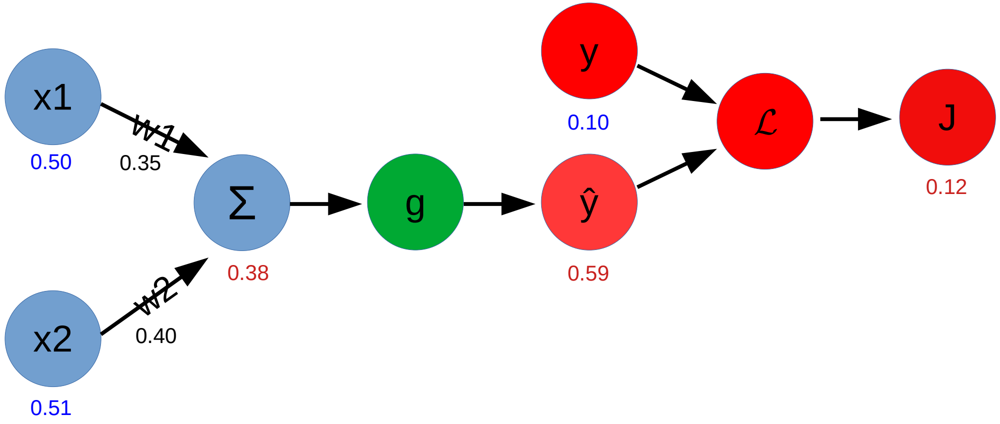

**x1** = 0.50
**x2** = 0.51

**w1** = 0.35
**w2** = 0.40

**Σ** = 0.35 * x1 + 0.4 * x2 = 0.38
**ŷ** = g(Σ) = $1/(1+e^{-\Sigma})$ = 0.59

**Y** = 0.10
**J(W)** = $\mathcal{L}(y, \hat{y}) = ½ * (y - \hat{y})^2 = ?$

# 2. Calculate Loss



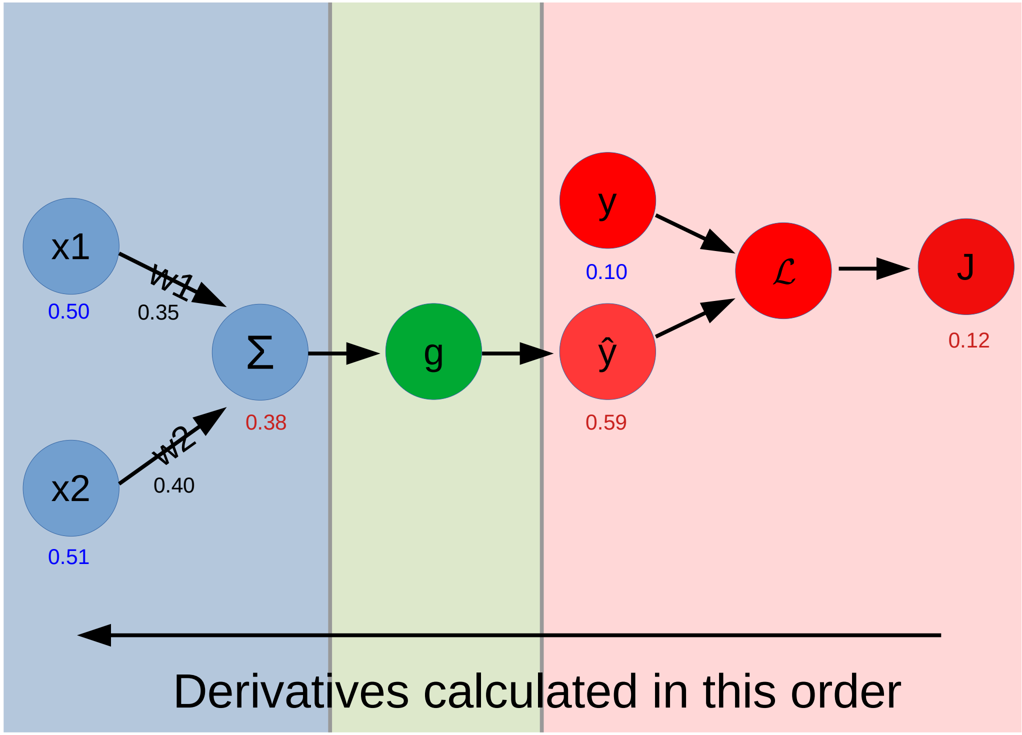

**x1** = 0.50
**x2** = 0.51

**w1** = 0.35
**w2** = 0.40

**Σ** = 0.35 * x1 + 0.4 * x2 = 0.38
**ŷ** = g(Σ) = $1/(1+e^{-\Sigma})$ = 0.59

**Y** = 0.10
**J(W)** = $\mathcal{L}$(y, ŷ) = ½ * $(y - \hat{y})^2$ = 0.12

# 3. Backpropagate the error



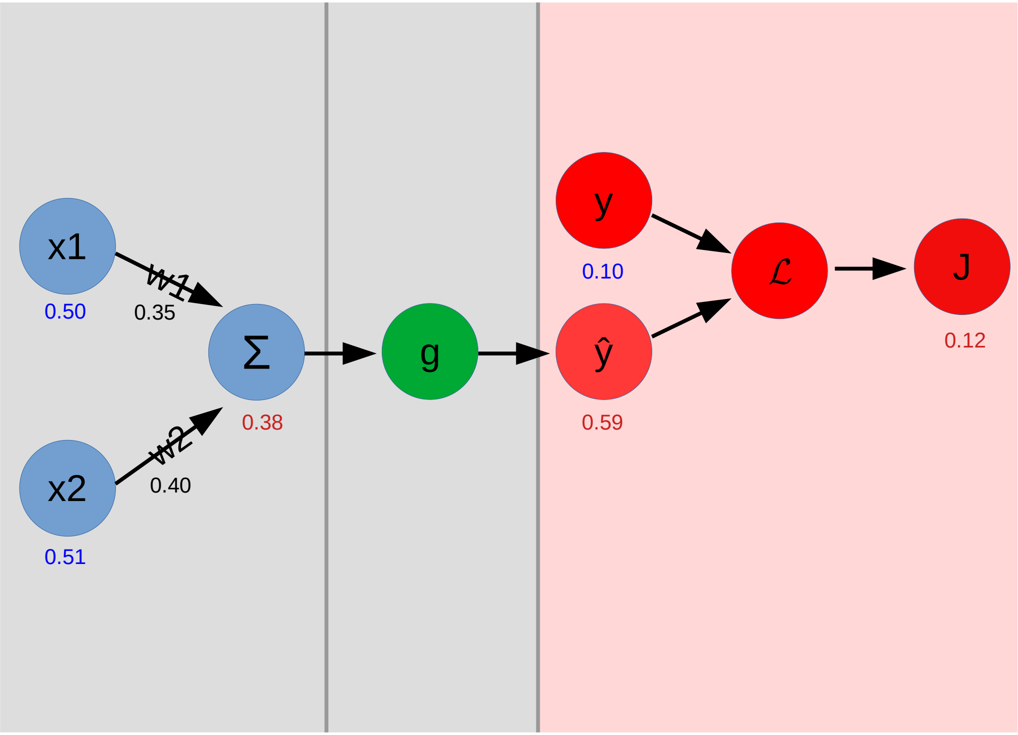

$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = \partial J(W) / \partial \hat{y}$ (loss)

$* \ \partial \hat{y} / \Sigma$ (activation)

$* \ \partial \Sigma / \partial w1$ (weight)

# 3. Backpropagate the error: loss




$$J(W) = \mathcal{L}(g(X * W))$$

$$\partial J(W)/\partial w1 = \partial J(W) / \partial \hat{y} \quad \text{(loss)}$$

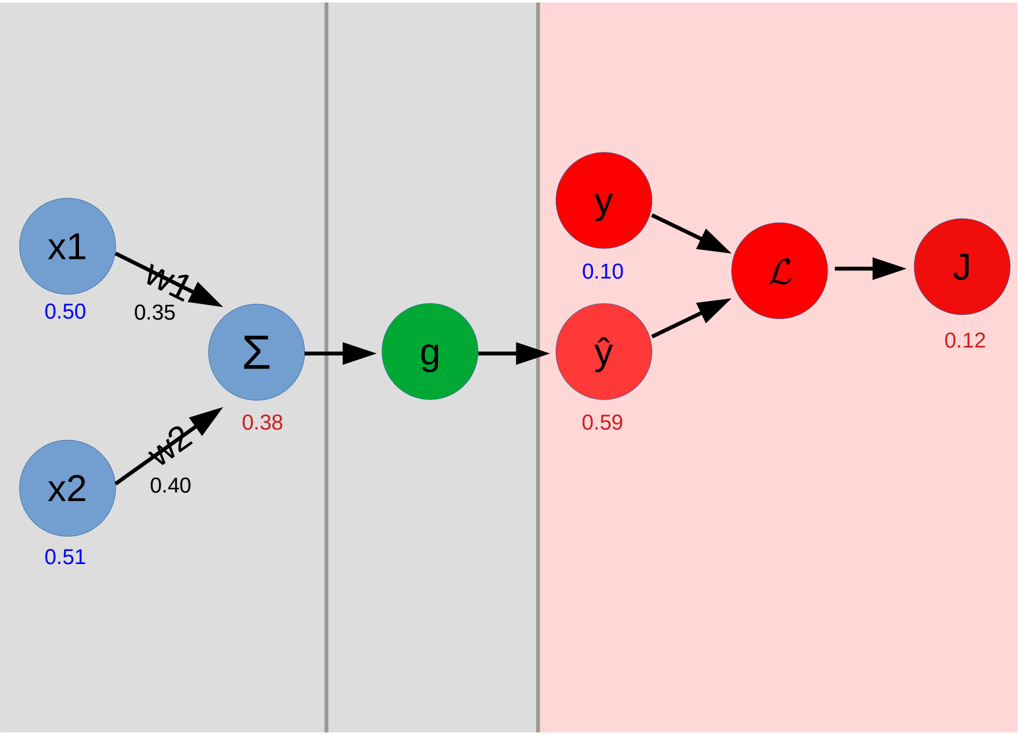
$$* \ \partial \hat{y} / \Sigma \quad \text{(activation)}$$

$$* \ \partial \Sigma / \partial w1 \quad \text{(weight)}$$

$$J(W) = \mathcal{L}(y, \hat{y}) = ½ * (y - \hat{y})^2$$

$$\partial J(W) / \partial \hat{y} = \partial \mathcal{L}(y, \hat{y}) / \partial \hat{y} = ?$$

# 3. Backpropagate the error: loss




$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = \partial J(W) / \partial \hat{y}$ (loss)

$* \; \partial \hat{y} / \Sigma$ (activation)

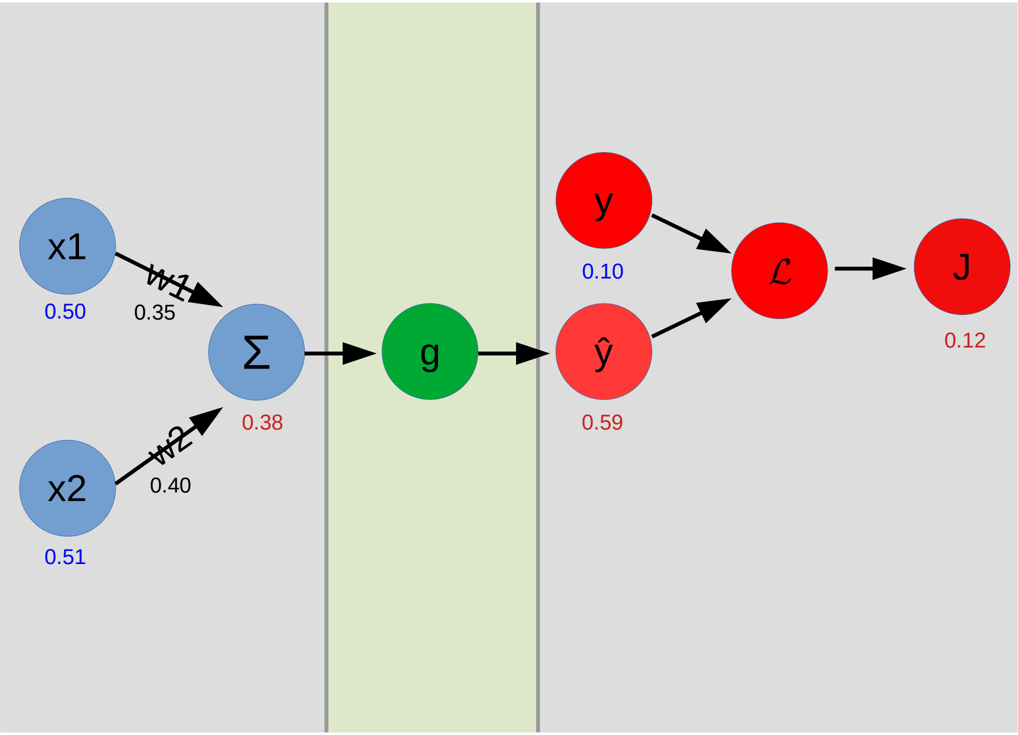
$* \; \partial \Sigma / \partial w1$ (weight)

$J(W) = \mathcal{L}(y, \hat{y}) = ½ * (y – \hat{y})^2$

$\partial J(W) / \partial \hat{y} = 2 * ½ * (y – \hat{y}) * -1$

$= -y + \hat{y} = -0.1 + 0.59 = 0.49$

# 3. Backpropagate the error: activation




$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

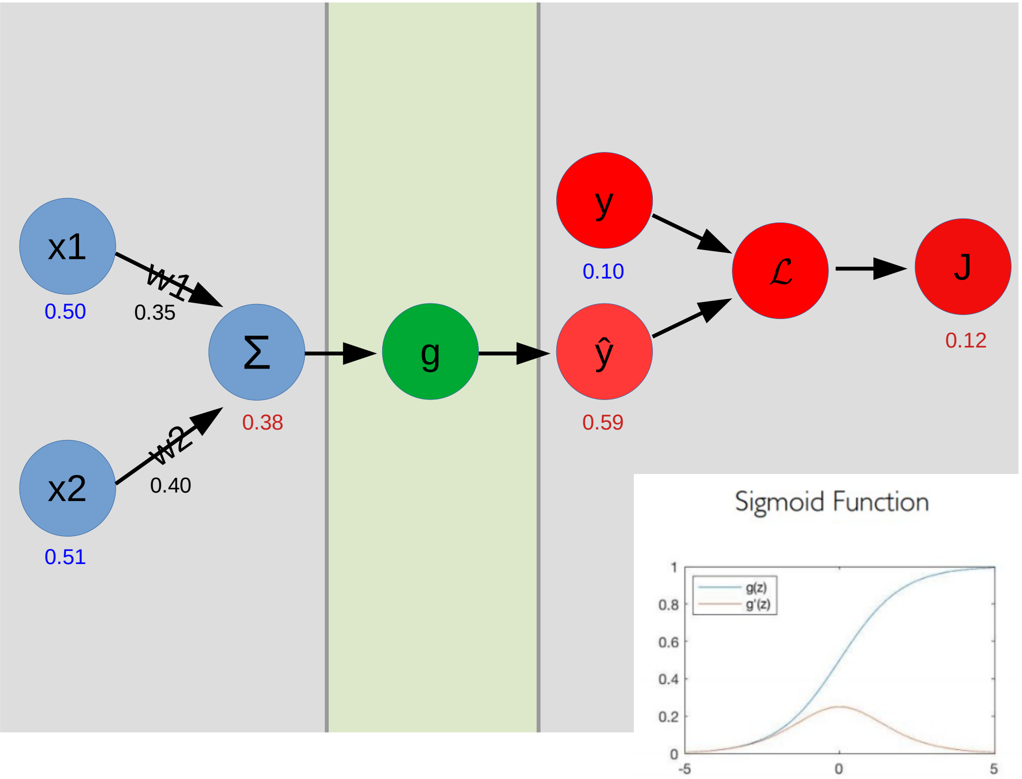
$* \ \partial \hat{y} / \partial \Sigma$ (activation)

$* \ \partial \Sigma / \partial w1$ (weight)

$\hat{y} = g(\Sigma) = 1/(1+e^{-\Sigma})$

$\partial \hat{y} / \partial \Sigma = ?$

# 3. Backpropagate the error: activation




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

$$g'(z) = g(z)(1 - g(z))$$

$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

$* \, \partial \hat{y} / \partial \Sigma$ (activation)

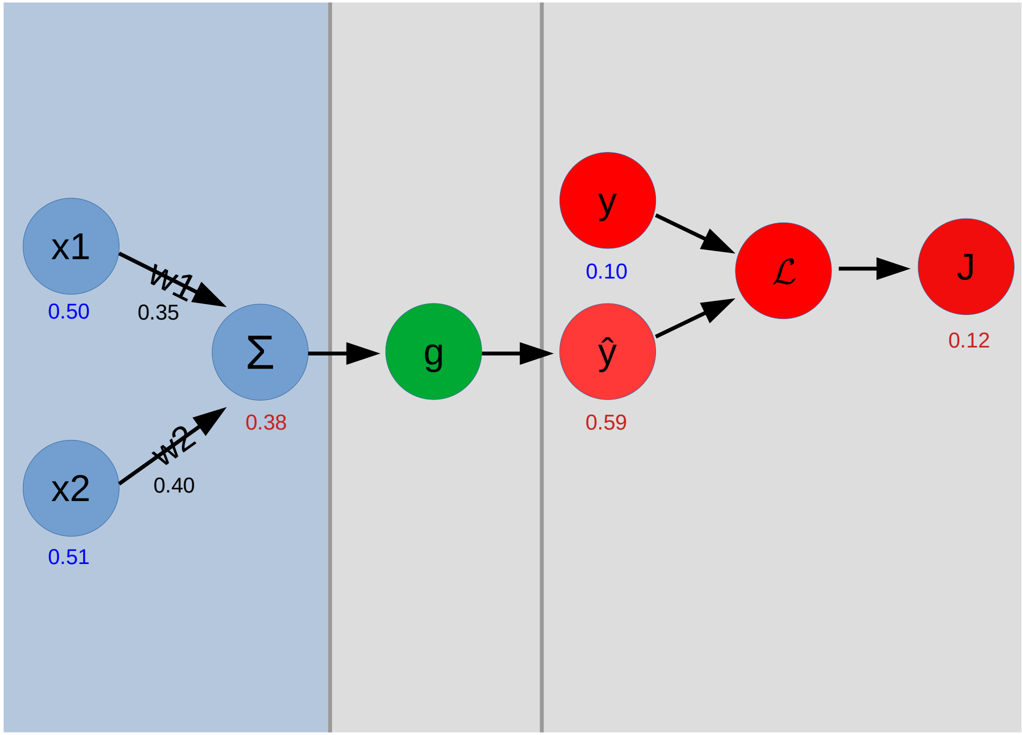
$* \, \partial \Sigma / \partial w1$ (weight)

$\hat{y} = g(\Sigma) = 1/(1+e^{-\Sigma})$

$\partial \hat{y} / \partial \Sigma = 1/(1+e^{-\Sigma}) * (1 - 1/(1+e^{-\Sigma}))$

$= 0.24$

# 3. Backpropagate the error: weight




$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

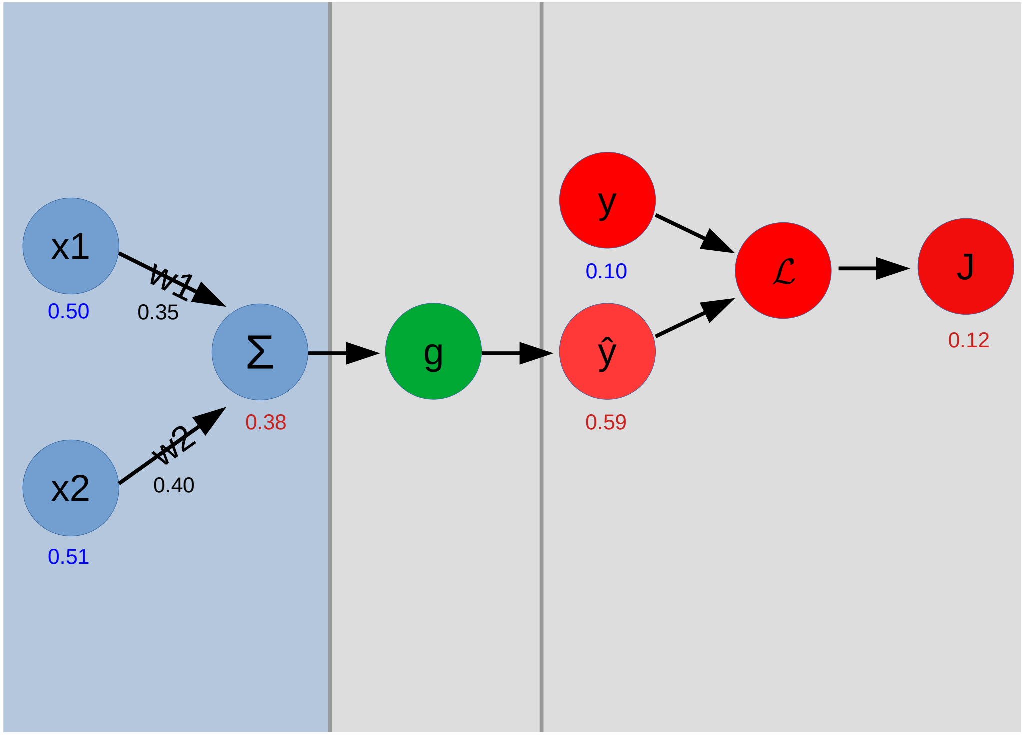
$* \ 0.24$ (activation)

$* \ \partial\Sigma / \partial w1$ (weight)

$\Sigma = X * W = x1 * w1 + x2 * w2$

$\partial\Sigma/\partial w1 = ?$

# 3. Backpropagate the error: weight




$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

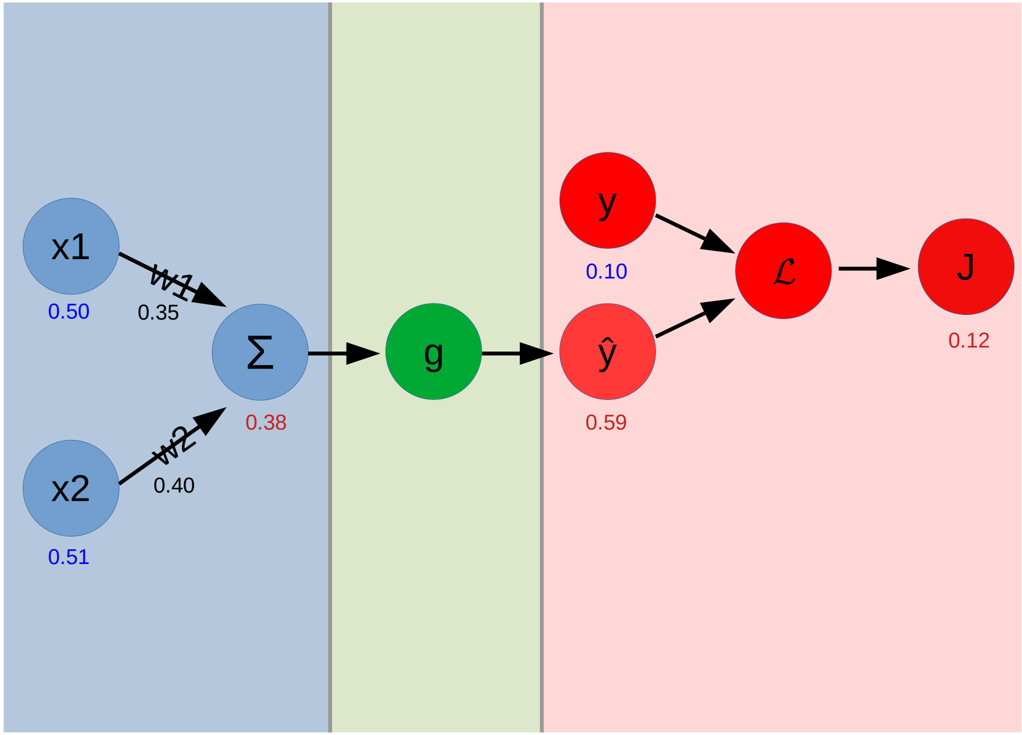
$* \ 0.24$ (activation)

$* \ \partial \Sigma / \partial w1$ (weight)

$\Sigma = X * W = x1 * w1 + x2 * w2$

$\partial \Sigma / \partial w1 = x1 + 0 = 0.5$

# 4. Weight update




$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

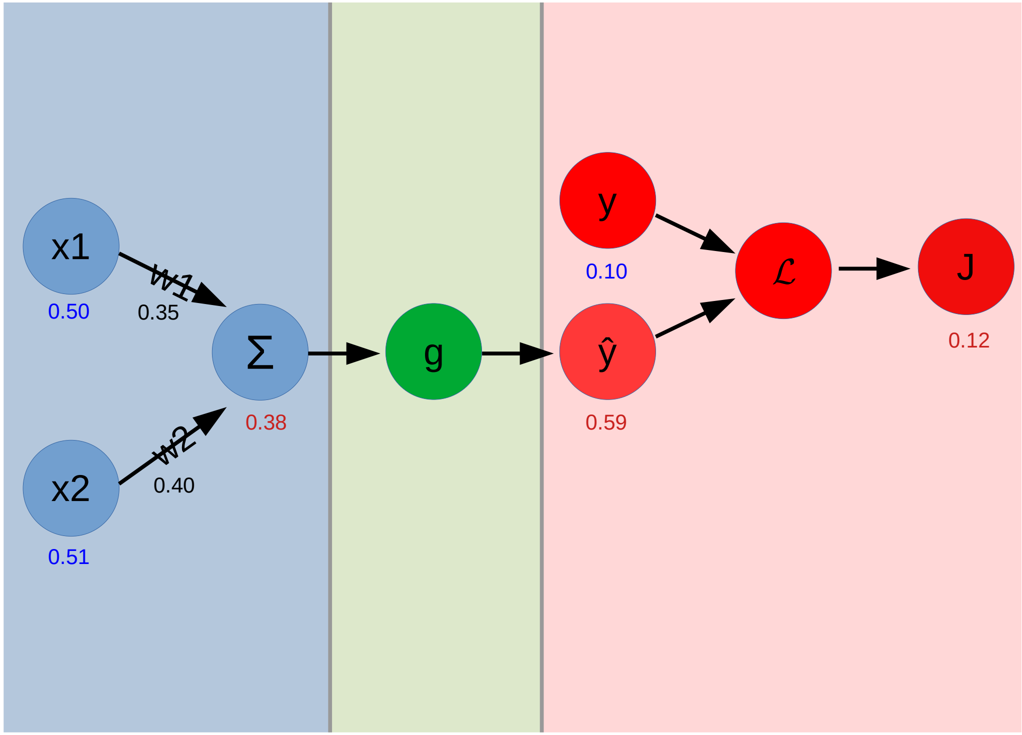
$* \ 0.24$ (activation)

$* \ 0.51$ (weight)

$= 0.06$ (gradient)

w1’ = ?

# 4. Weight update




$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

$* \; 0.24$ (activation)

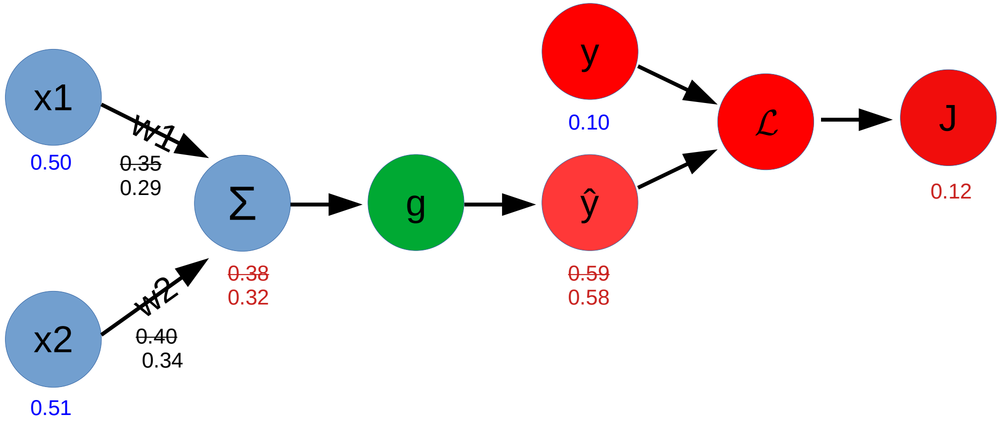
$* \; 0.51$ (weight)

$= 0.06$ (gradient)

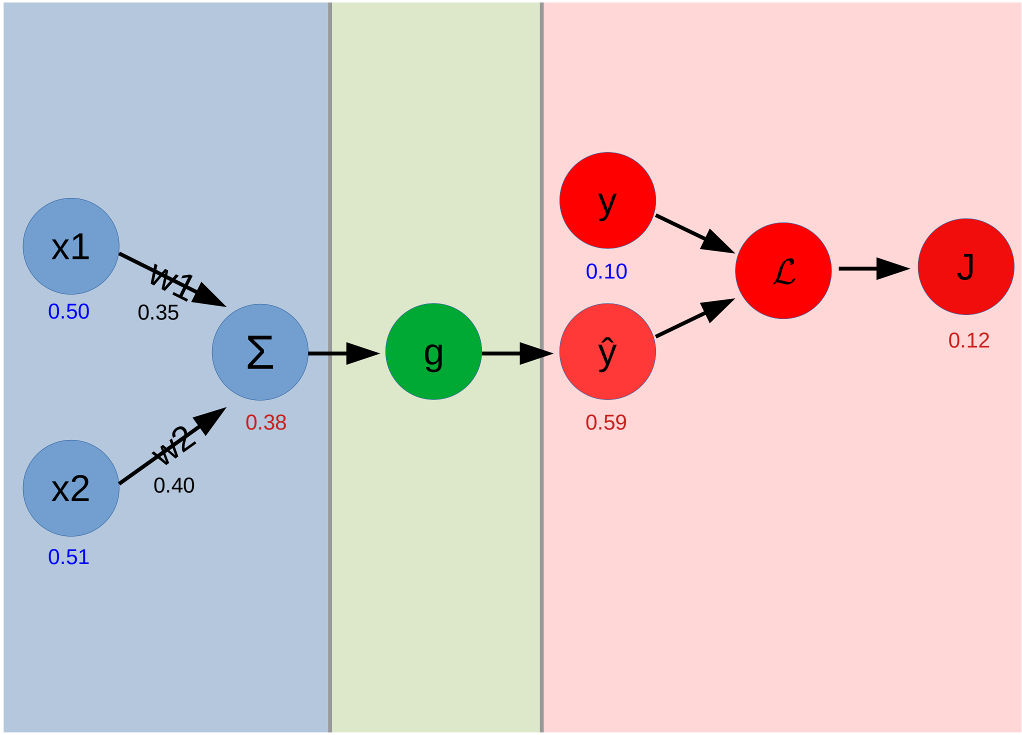
$w1' = w1 - \eta * 0.06 = 0.35 - 0.06 = 0.29$

# Exercise

- Can you calculate the weight update for w2? How many new gradients do you need to calculate?
- What is the new predicted output? Has the error gone down?
- What if I had another layer before this one?



# 4. Weight update




$J(W) = \mathcal{L}(g(X * W))$

$\partial J(W)/\partial w1 = 0.49$ (loss)

$* \ 0.24$ (activation)

$* \ 0.51$ (weight)

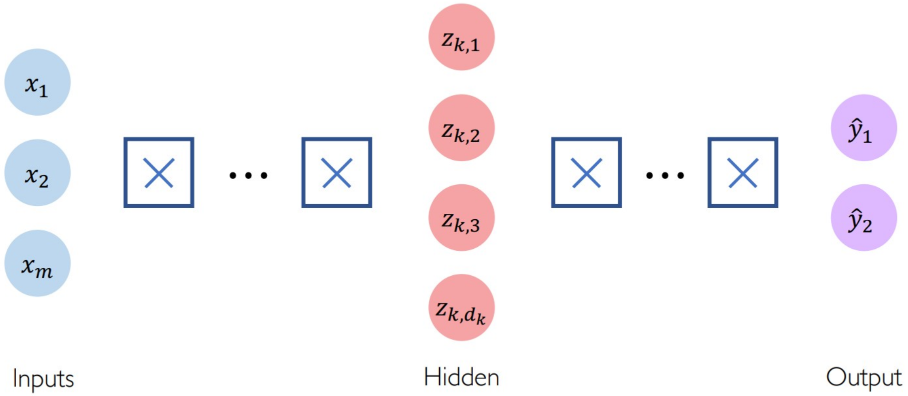
$= 0.06$ (gradient)

$w1' = w1 - \eta * 0.06 = 0.35 - 0.06 = 0.29$

$W2' = w2 - \eta * 0.06 = 0.4 - 0.06 = 0.34$

# Gradient vanishing

- What happens if we backpropagate on a network with many (N > k > 1) hidden layers?




$$z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j})\, w_{j,i}^{(k)}$$

$$\partial J/\partial W1 = \partial J/\partial \hat{Y} * \partial \hat{Y}/\partial \Sigma_N * \partial \Sigma_N/\partial W_N * \ldots * \partial Z_k/\partial \Sigma_k * \partial \Sigma_k/\partial W_k * \ldots \partial Z_1/\partial \Sigma_1 * \partial \Sigma_1/\partial W_1$$

# Gradient vanishing

- These are all “zero-point-somethings” multiplied by each other
- So the gradient becomes smaller by orders of magnitudes as we go back more and more layers until it’s so small that the network is stuck

$$\partial E/\partial W1 = \partial J/\partial \hat{Y} * \partial \hat{Y}/\partial \Sigma_N * \partial \Sigma_N/\partial W_N * \ldots * \partial Z_k/\partial \Sigma_k * \partial \Sigma_k/\partial W_k * \ldots \partial Z_1/\partial \Sigma_1 * \partial \Sigma_1/\partial X_1 = O(10^{-N})$$

initial w1 = 0.5
optimal w1 = -0.2
5-layer gradient ~0.00001

How many iterations do we need to get from 0.5 to -0.2?