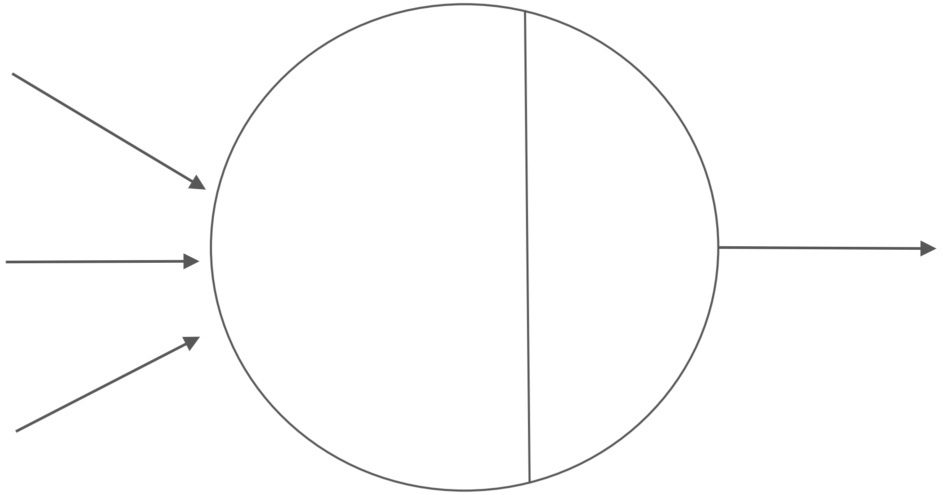
Introduction to Neural Networks for CAP4453

Single Neuron

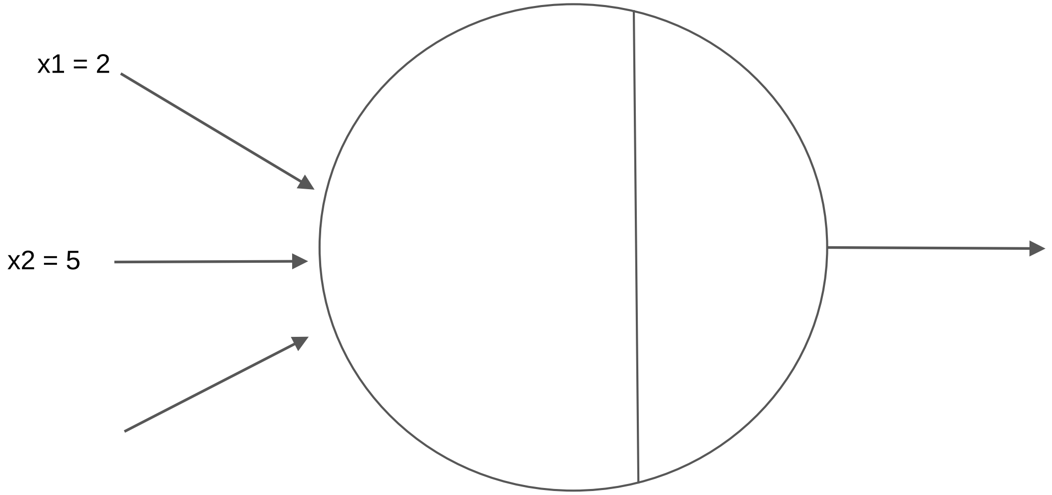
Basic computational unit of Neural Network



Single Neuron

Basic computational unit of Neural Network

Inputs (x1, x2) : Data you want to model.

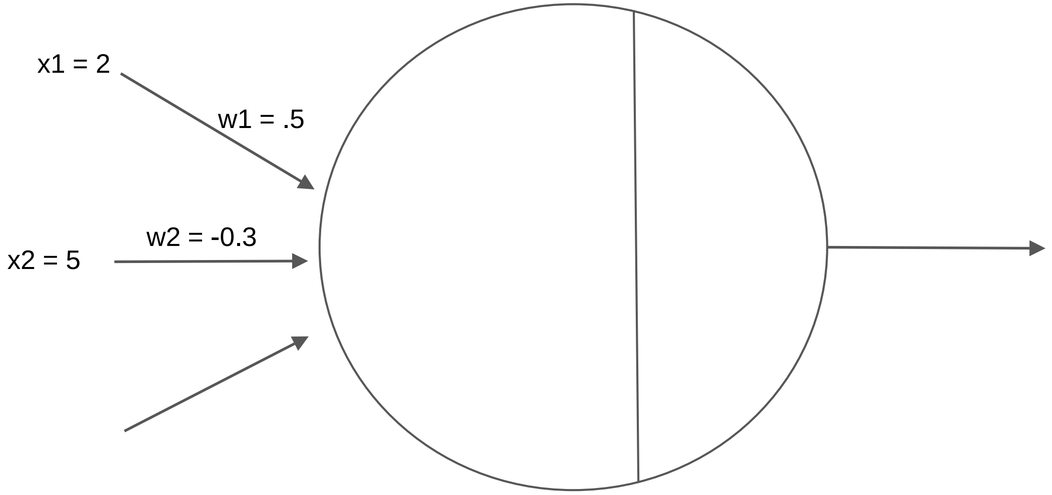


Single Neuron

Basic computational unit of Neural Network

Inputs (x1, x2) : Data you want to model.

Weights (w1, w2) : Model Parameters



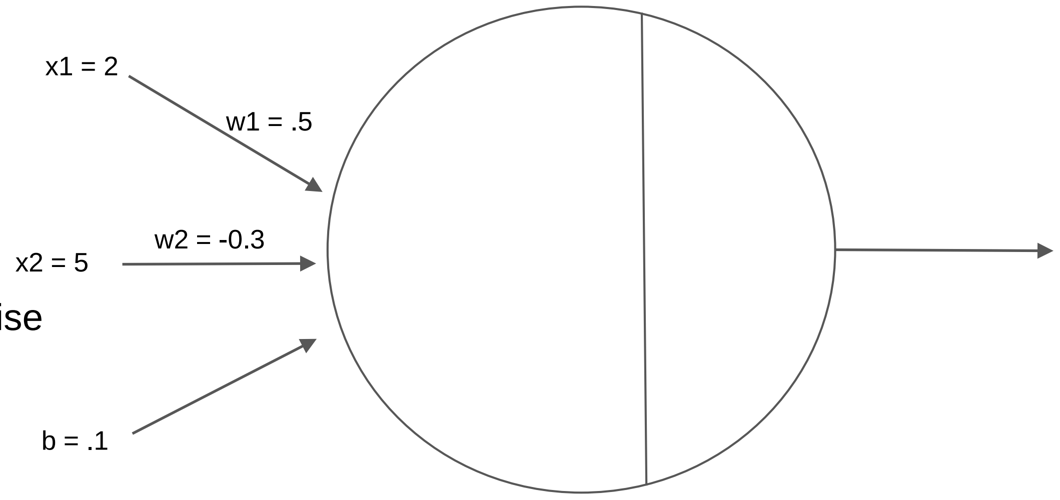
Single Neuron

Basic computational unit of Neural Network

Inputs (x1, x2) : Data you want to model.

Weights (w1, w2) : Model Parameters

Bias (b) : Model parameter to account for Noise



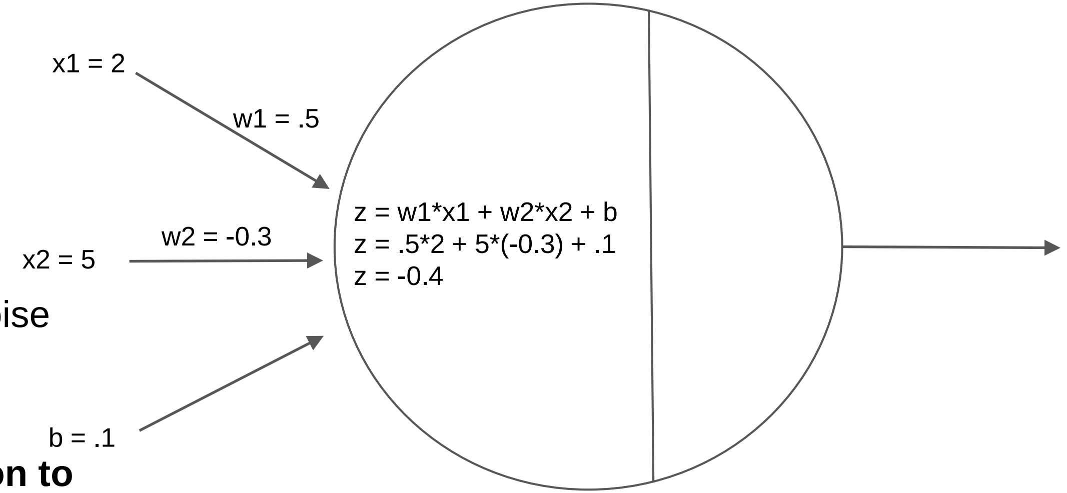
Single Neuron

Basic computational unit of Neural Network

Inputs (x1, x2) : Data you want to model.

Weights (w1, w2) : Model Parameters

Bias (b) : Model parameter to account for Noise



Computation step 1 - Weighted Summation to perform Linear modelling

$$z = w1 * x1 + w2 * x2 + b$$

Single Neuron

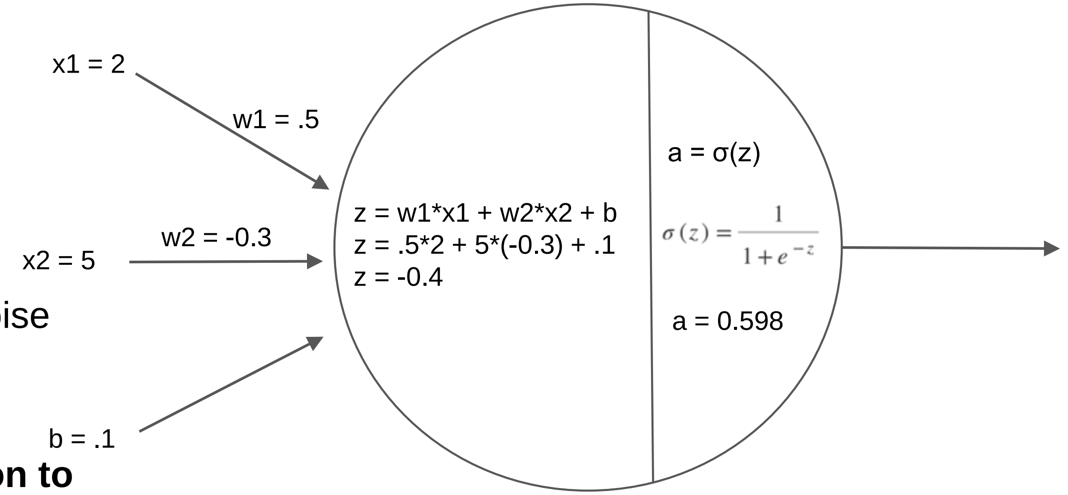
The Vertical Segment is the reminder to do the Non-linear step.

Basic computational unit of Neural Network

Inputs (x1, x2) : Data you want to model.

Weights (w1, w2) : Model Parameters

Bias (b) : Model parameter to account for Noise



Computation step 1 - Weighted Summation to perform Linear modelling

$$z = w1 * x1 + w2 * x2 + b$$

Computation step 2 - Adding Non Linearity

a = σ(z), where sigma is $\sigma(z) = \frac{1}{1+e^{-z}}$

Single Neuron

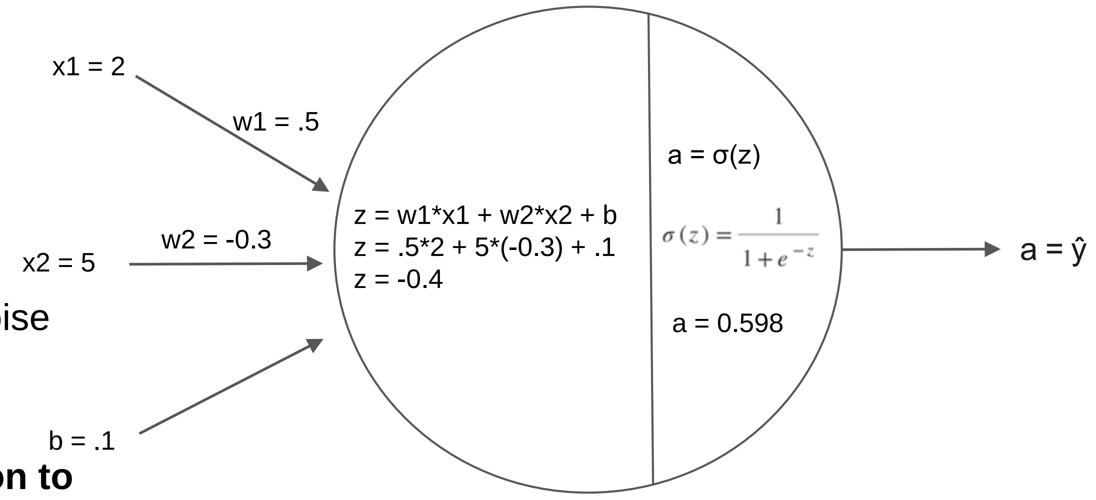
The Vertical Segment is the reminder to do the Non-linear step.

Basic computational unit of Neural Network

Inputs (x1, x2) : Data you want to model.

Weights (w1, w2) : Model Parameters

Bias (b) : Model parameter to account for Noise



Computation step 1 - Weighted Summation to perform Linear modelling

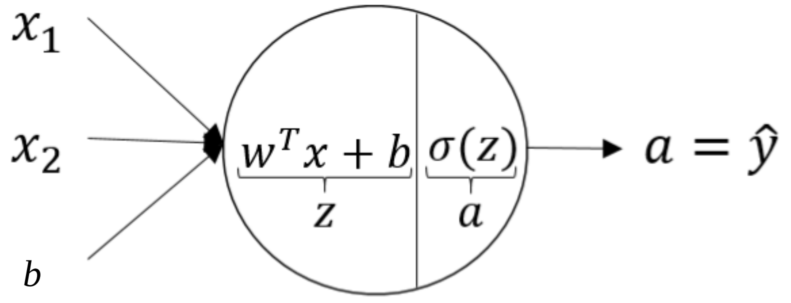
z = w1 * x1 + w2 * x2 + b or $z = w^T x + b$

Computation step 2 - Adding Non Linearity

a = σ(z), where sigma is $\sigma(z) = \dfrac{1}{1+e^{-z}}$

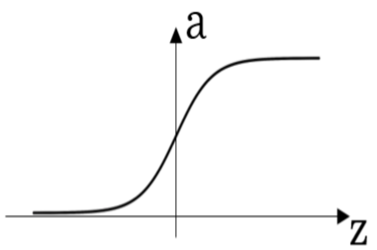
Output - ŷ

General Structure



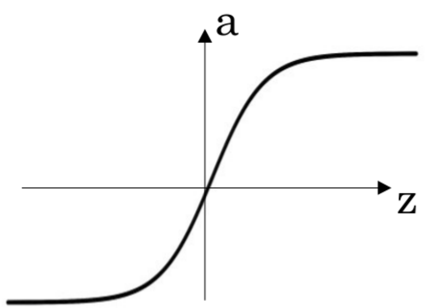
Activation Function

Adds Non-Linearity to the Neural Network to fit Non-Linear patterns



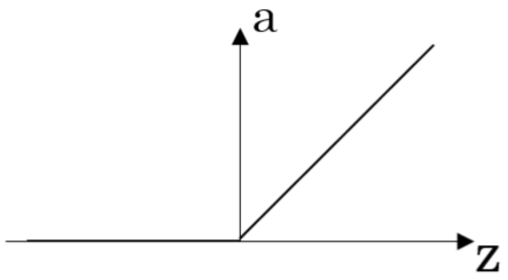
$$g(z) = \frac{1}{1+e^{-z}}$$

Sigmoid



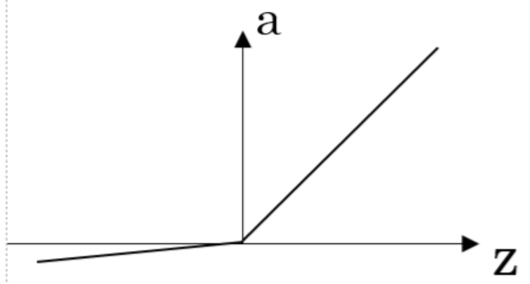
$$g(z) = \tanh(z)$$

Tanh



ReLU

g(z) = max(0,z)



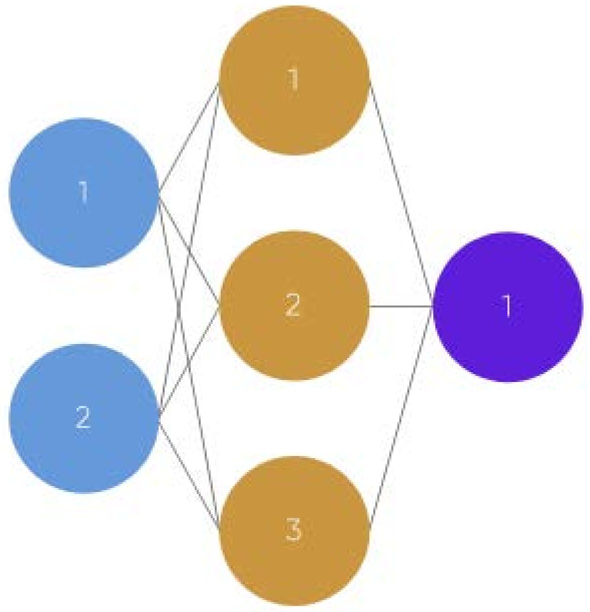
Leaky ReLU

g(z) = max(0.01z,z)

Each Activation Function has pros and cons (http://cs231n.github.io/neural-networks-1/#intro)

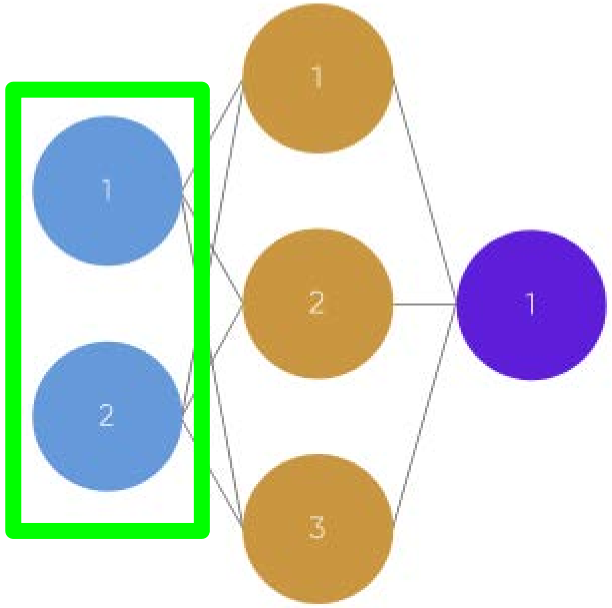
A Simple Example for Neural Network

The Collection of Neurons is organized in three main layers:



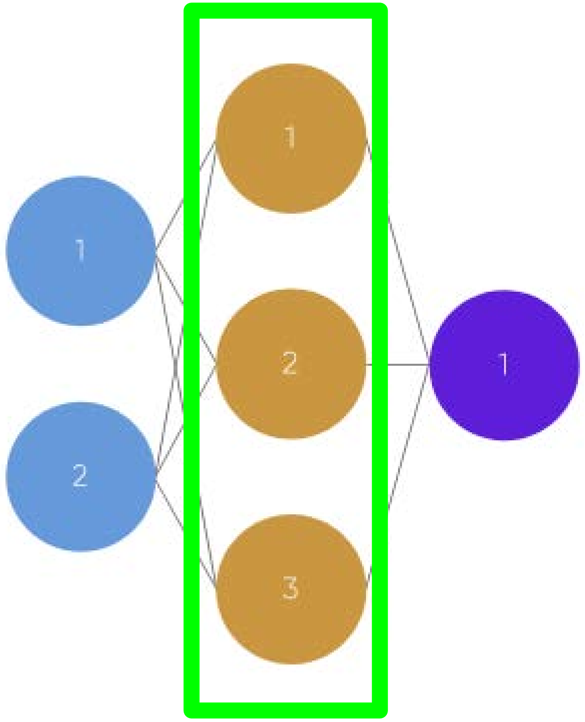
A Simple Example for Neural Network

The Collection of Neurons is organized in three main layers: the **input** layer



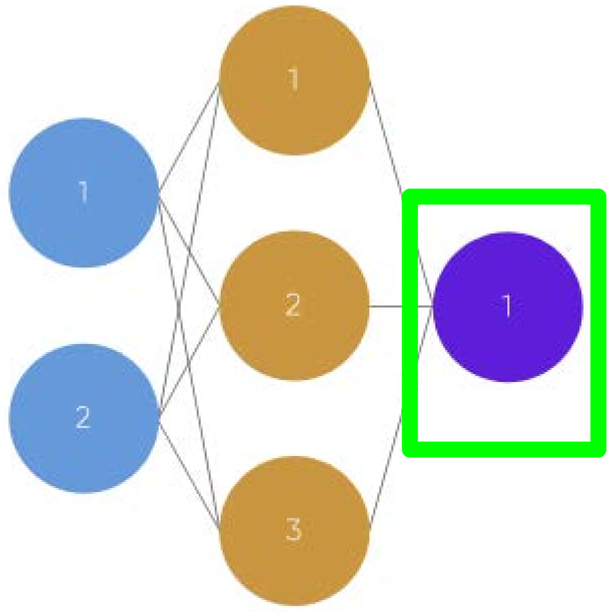
A Simple Example for Neural Network

The Collection of Neurons is organized in three main layers: the **input** layer, the **hidden** layer



A Simple Example for Neural Network

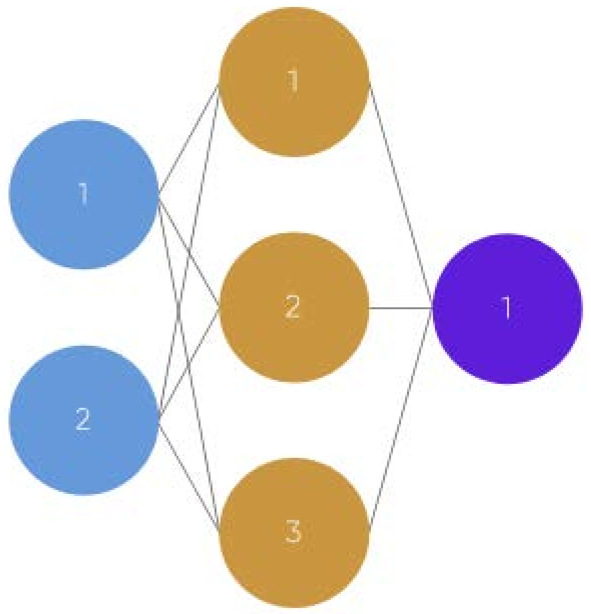
The Collection of Neurons is organized in three main layers: the **input** layer, the **hidden** layer, and the **output** layer



A Simple Example for Neural Network

The Collection of Neurons is organized in three main layers: the **input** layer, the **hidden** layer, and the **output** layer

A neural network can have many hidden layers.

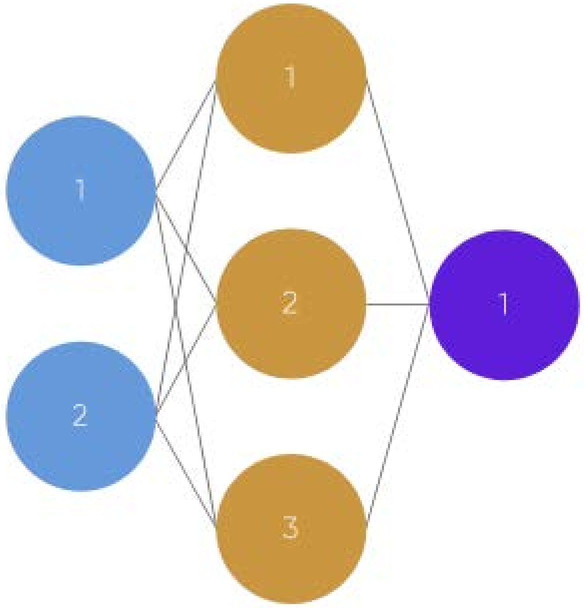


A Simple Example for Neural Network

The Collection of Neurons is organized in three main layers: the **input** layer, the **hidden** layer, and the **output** layer

A neural network can have many hidden layers.

In an **artificial neural network**, there are several inputs, which are called **features**, and produce a single output.



A Simple Example for Neural Network

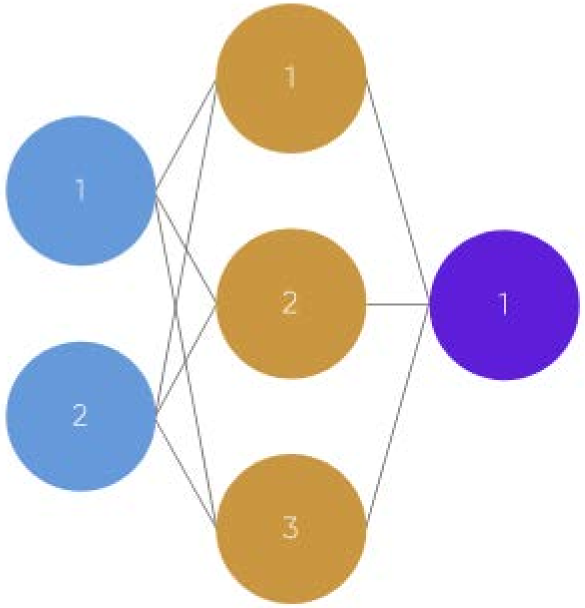
The Collection of Neurons is organized in three main layers: the **input** layer, the **hidden** layer, and the **output** layer

A neural network can have many hidden layers.

In an **artificial neural network**, there are several inputs, which are called **features**, and produce a single output.

In the figure, we model a single hidden layer with three neurons and single output.

A layer is **fully connected layer** if each neuron in the layer is connected to all neurons in the previous layer.



A Simple Example for Neural Network

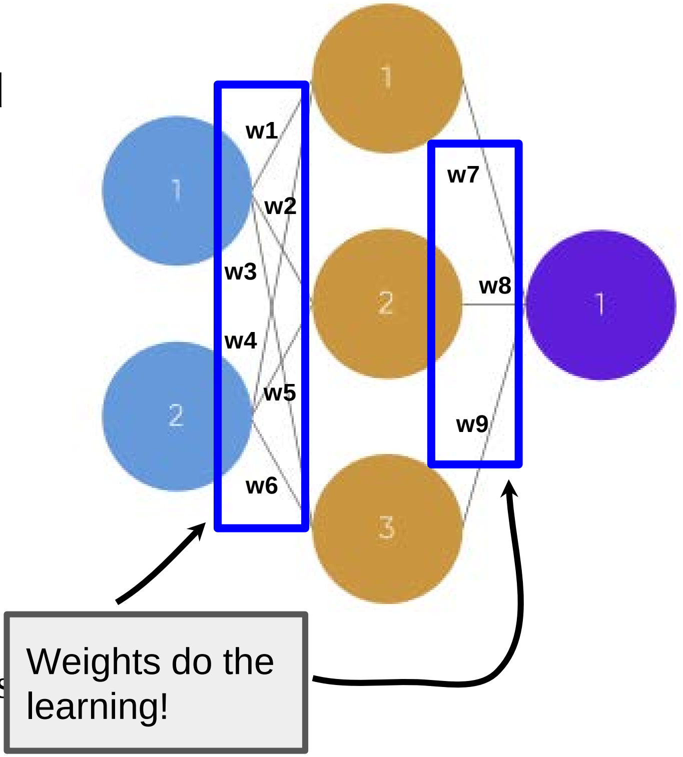
The Collection of Neurons is organized in three main layers: the **input** layer, the **hidden** layer, and the **output** layer

A neural network can have many hidden layers.

In an **artificial neural network**, there are several inputs, which are called **features**, and produce a single output.

In the figure, we model a single hidden layer with three neurons and single output.

A layer is **fully connected layer** if each neuron in the layer is connected to all neurons in the previous layer.



How to train a neural network?

Example: design a binary classifier which outputs 1 if the absolute difference in the inputs is an odd number.

Sample data:

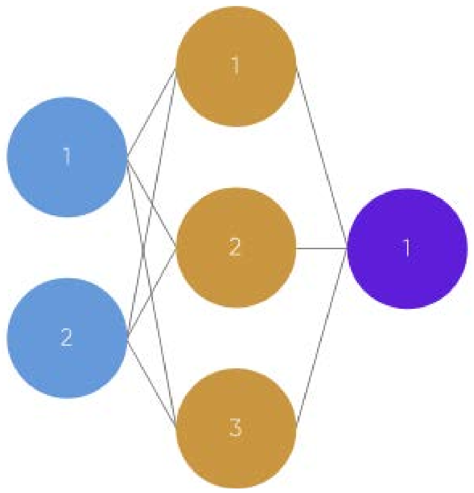
abs(2 - 5) = 3, output = 1

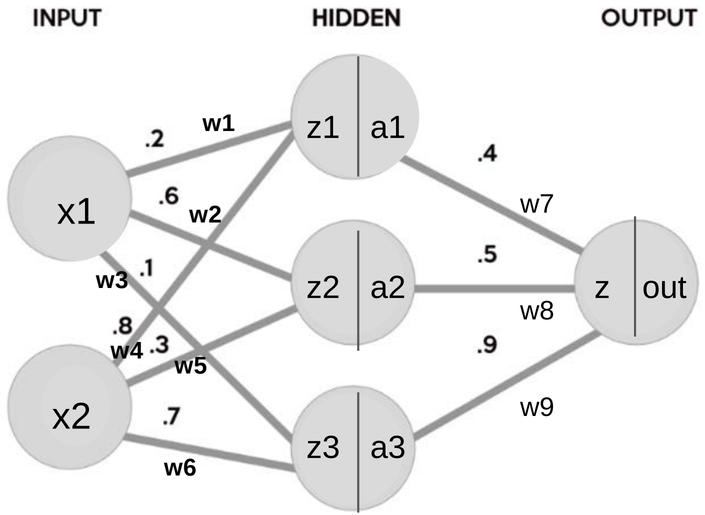
abs(5 - 3) = 2, output = 0

We start with random weights.

Note: a1,a2,.. denote single neuron's output,

whereas, **ŷ/ out denotes the final network's output.**





Forward Pass

Bias: b1 = 0.1, b2 = 0.2

Weights = Random generated

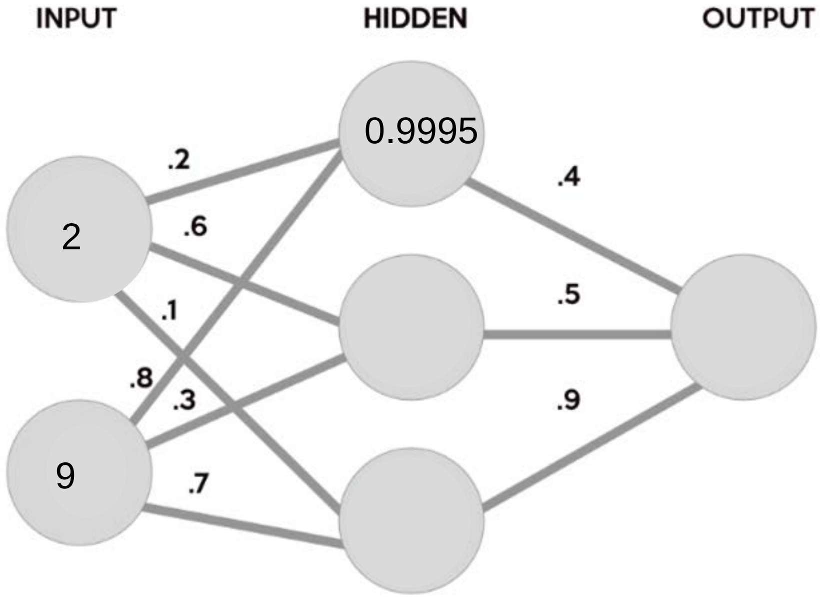
For each neuron, we calculate: $\sigma(\sum_i w_i x_i + b)$

a1 = $\sigma(\sum_i w_i x_i + b)$

a1 = σ(0.2*2+0.8*9+0.1)

a1 = σ(7.7)

a1 = 0.99 [putting 7.7 in the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$]



Forward Pass

Bias: b1 = 0.1, b2 = 0.2

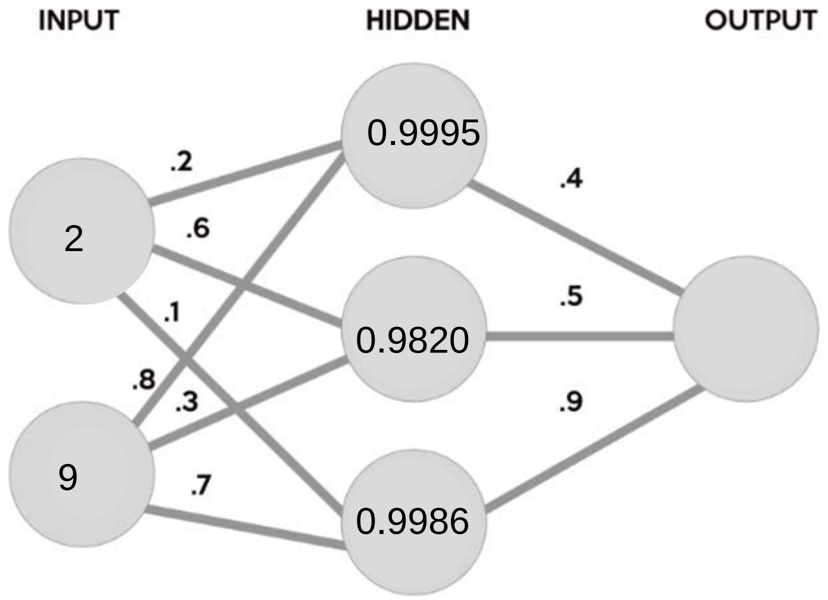
Weights = Random generated

For each neuron, we calculate: $\sigma(\sum_i w_i x_i + b)$

a = $\sigma(\sum_i w_i x_i + b)$ $\sigma(z) = \frac{1}{1+e^{-z}}$

a2 = σ(0.6*2+0.3*9+0.1) = σ(4) = 0.9820

a3 = σ(0.1*2+0.7*9+0.1) = σ(6.6) =0.9986



Forward Pass

Bias: b1 = 0.1, b2 = 0.2

Weights = Random generated

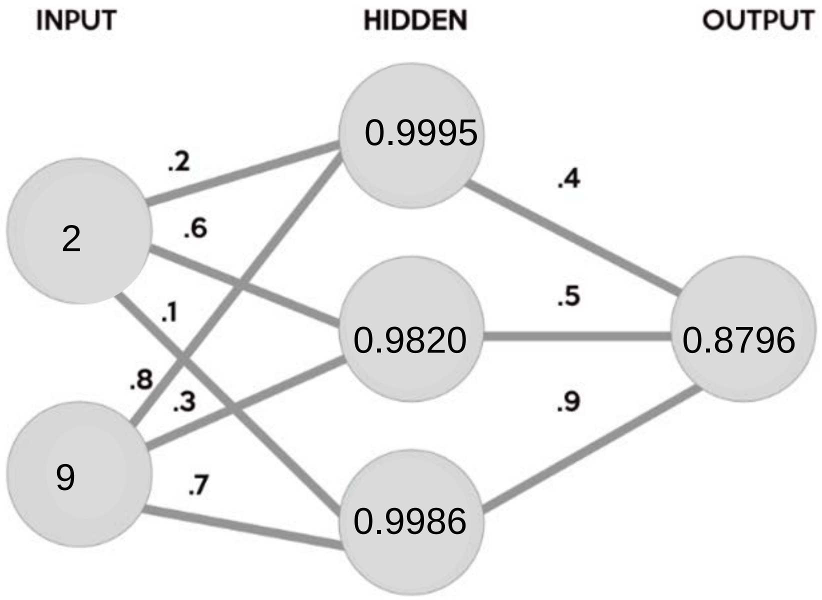
For each neuron, we calculate: $\sigma(\sum_i w_i x_i + b)$

a = $\sigma(\sum_i w_i x_i + b)$ $\sigma(z) = \frac{1}{1+e^{-z}}$

a2 = σ(0.6*2+0.3*9+0.1) = σ(4) = 0.9820

a3 = σ(0.1*2+0.7*9+0.1) = σ(6.6) =0.9986

ŷ = σ(0.9995*0.4 + 0.9820*0.5 + 0.9986*0.9 + 0.2) = σ(1.9895) = 0.8796



Calculating the error

The true label for output = 1

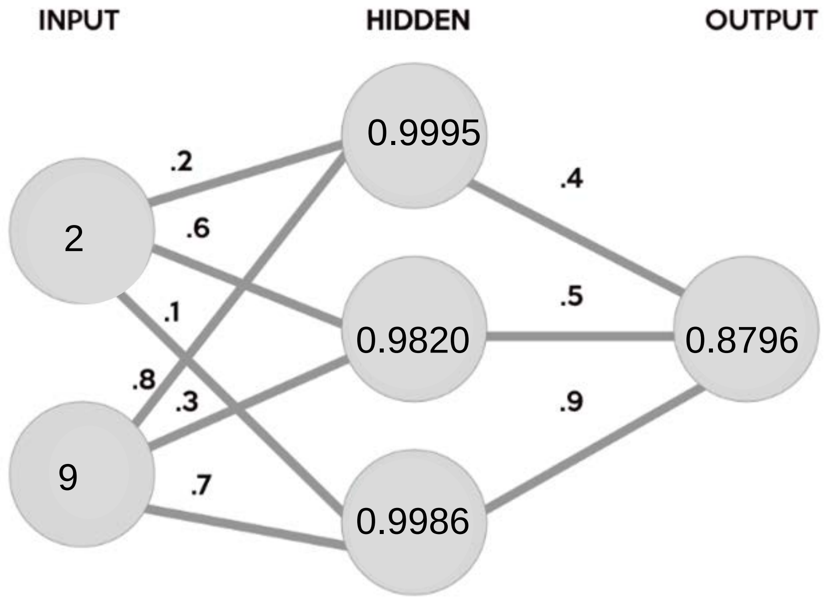
We have two input numbers and two classes:

Odd (1) and Even (0)

we need to alter weights to make our inputs(e.g., 2 and 9) equal to the corresponding output(i.e., 1).

This is done through a method called backpropagation.

Works by using a **loss function** to calculate how far the network was from the target output.



Loss Function

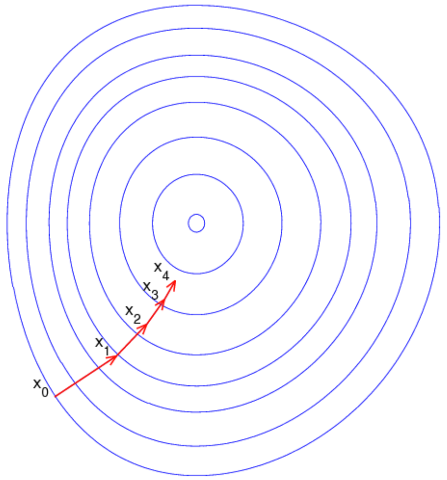
- Calculates error between the actual output and the predicted output.
- The error is back-propagated to update the weights.
- Ideally, if model (the weights and bias) is perfect then the error should be zero.
- We choose loss function based on the application.
- For example
  - Binary Classification - Cross Entropy (or log loss)
  - Multiclass Classification - Multi-class Cross Entropy
  - Regression - Mean Square Error

Gradient Descent (GD)

- A gradient measures how much the output of a function changes if you change the inputs a little bit.
- Commonly used optimization algorithm while training a machine learning model.
- It tweaks model parameters iteratively to minimize a given function to its local minimum.
- As shown, at each step GD tries to converge to minimum.

Steps in GD:

- Perform forward pass.
- Calculate error.
- Back propagate error as gradients.
- These gradients at each step update weights using the equation ($W^{k+1} = W^{k}$ - learning_rate*(gradient))
- Perform above steps iteratively until error reaches minimal value.

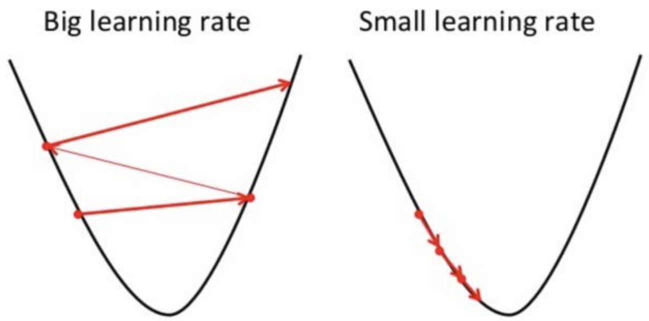


Source: https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0

Learning Rate

- A major component of Gradient descent is learning rate.
- Learning rate decides how big the steps are that the GD takes in the direction of the local-minimum.
- In order for Gradient Descent to reach the local minimum, we have to set the learning rate to an appropriate value, which is neither too low nor too high.

- If the steps it takes are too big, it maybe will not reach the local minimum because it just bounces back and forth between the convex function of gradient descent
- If you set the learning rate to a very small value, gradient descent will eventually reach the local minimum but it might take too much time as you can see (may happen) on the right side of the figure.



Source: https://towardsdatascience.com/gradient-descent-in-a-nutshell-eaf8c18212f0

Back to our example: Calculating the error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

In our case, we have single output neuron. The target output = 1, but the neural network output = 0.8796

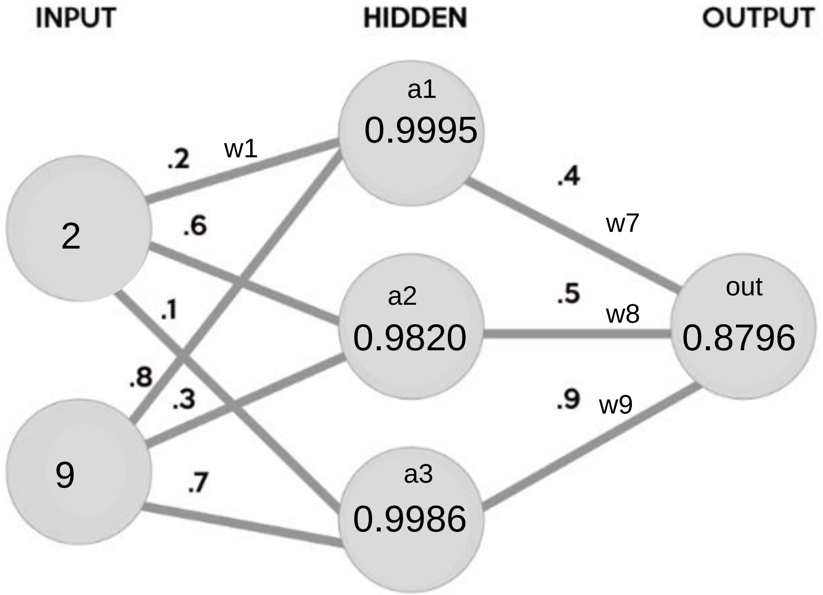
Therefore, its error is $E = (1/2)(1 - 0.8796)^2 = 0.00724808$

backpropagation: to update each of the weights in the network so that they cause the actual output to be closer to the target output, thereby minimizing the error for each output neuron and the network as a whole.

The Backward Pass

Remember the update step in the Gradient descent algorithm.

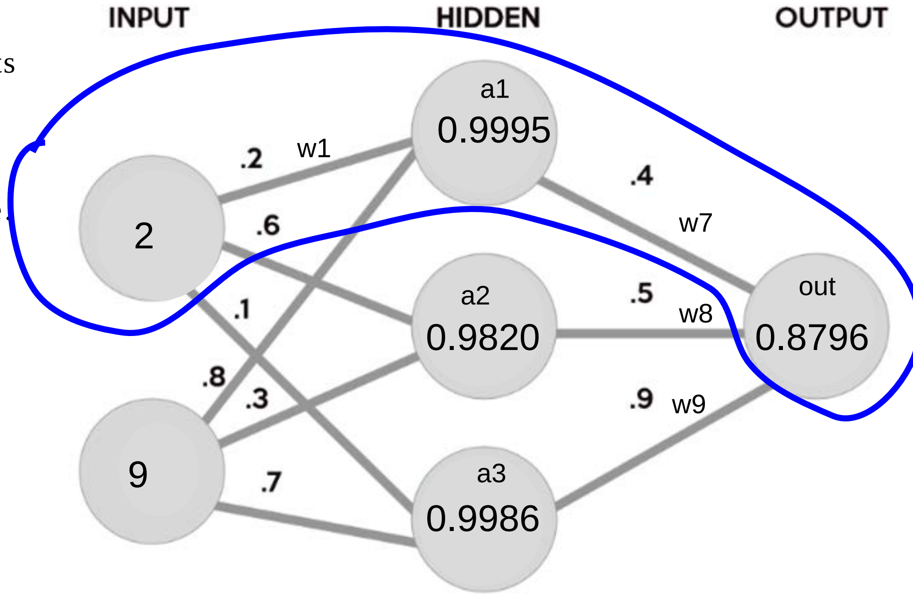
- $W^{k+1} = W^{k}$ - learning_rate*(gradient)
- To calculate gradients:
  - We want to know how much a change in weights affects the error.



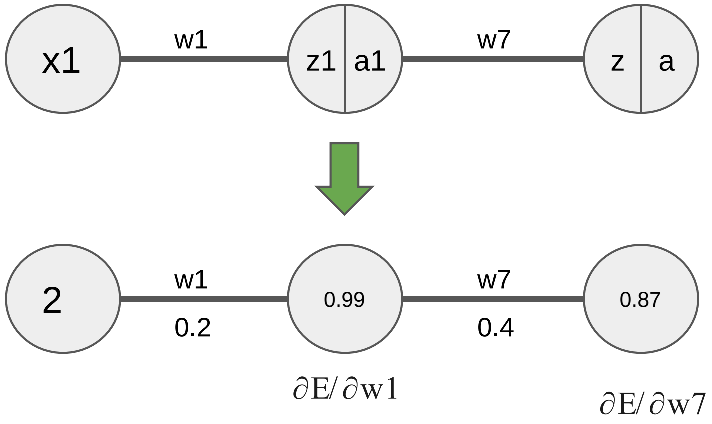
The Backward Pass

Remember the update step in the Gradient descent algorithm.

- $W^{k+1} = W^{k}$ - learning_rate*(gradient)
- To calculate gradients:
  - We want to know how much a change in weights affects the error.
  - In this example, we will just show backpropagation for the highlighted subgraph.
  - You can complete rest of the calculation as an exercise.



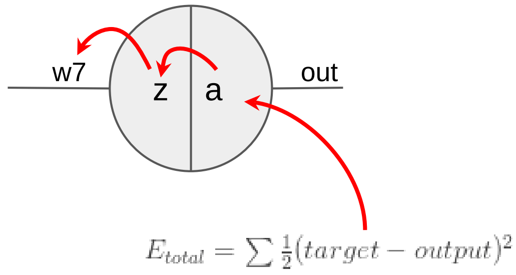
The Backward Pass



Remember the update step in the Gradient descent algorithm.

- $W^{k+1} = W^{k}$ - learning_rate*(gradient)
- For backward pass, we need to calculate derivatives (gradient) i.e. Æ/ ∂w7 and Æ/ ∂w1
- Using chain rule: $\partial E/\partial w7 = \partial E/\partial out * \partial out/\partial z * \partial z/\partial w7$
- $\partial E/\partial out = \partial[1/2(target - out)^2]/\partial out$

$$\partial E/\partial out = ½ * 2(target - out)^{2-1}(-1)+0$$
$$= -(target - out) = -(1 - 0.8796) = -0.1204$$



Chain rule illustration

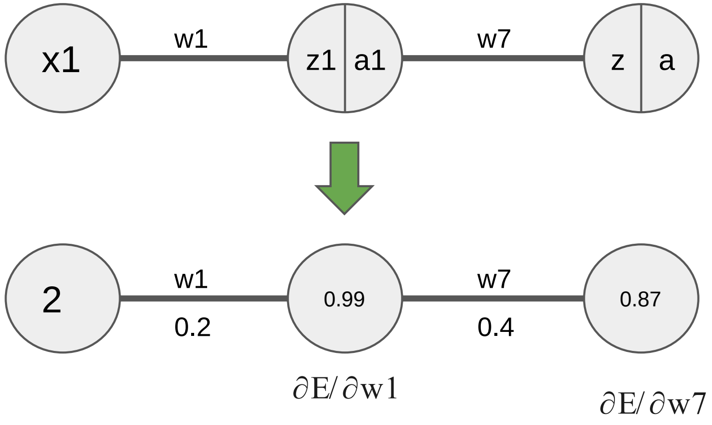
x1, x2 are inputs.

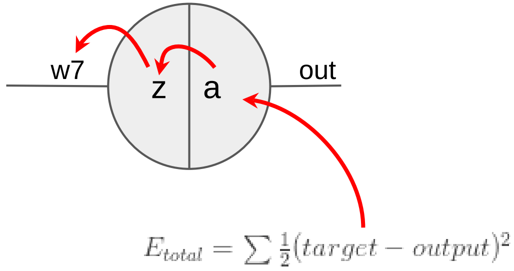
z = a1*w7 + a2*w8 + a3*w9 + b2 * 1
z1 = w1*x1 + w4* x2 + b1*1
(refer to slide 18 for full example)

The Backward Pass

Remember the update step in the Gradient descent algorithm.

- $W^{k+1} = W^k$ - learning_rate*(gradient)
- For backward pass, we need to calculate derivatives (gradient) i.e. $\partial E/\partial w7$ and $\partial E/\partial w1$
- Using chain rule: $\partial E/\partial w7 = \partial E/\partial out * \partial out/\partial z * \partial z/\partial w7$
- Out = Sigmoid function = $\sigma(z) = \frac{1}{1+e^{-z}}$
- $\partial\sigma/\partial z = \sigma(1-\sigma)$
- [For complete derivation, see this link: https://beckernick.github.io/sigmoid-derivative-neural-network/]
- $\partial out/\partial z = \partial\sigma/\partial z = 0.8796\ (1 - 0.8796) = 0.1059$





Chain rule illustration

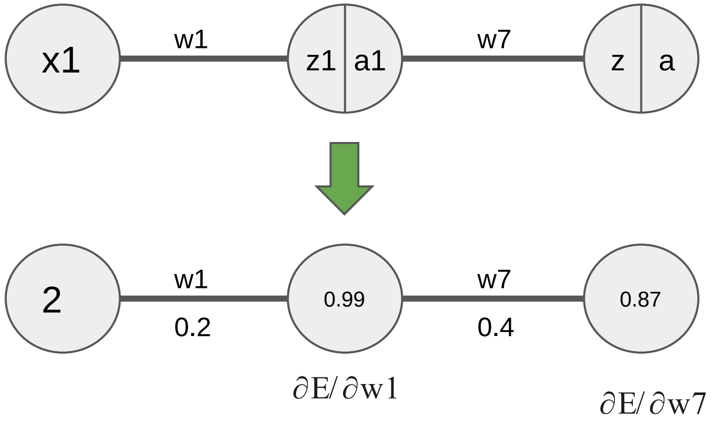
The Backward Pass

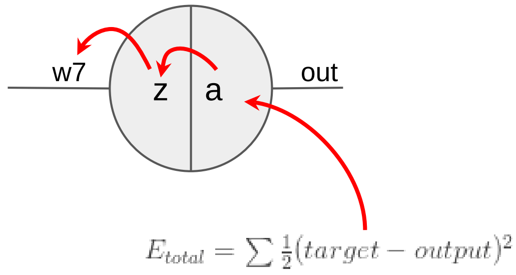
Remember the update step in the Gradient descent algorithm.

- $W^{k+1} = W^{k}$ - learning_rate*(gradient)
- For backward pass, we need to calculate derivatives (gradient) i.e. $\partial E/\partial w7$ and $\partial E/\partial w1$
- Using chain rule: $\partial E/\partial w7 = \partial E/\partial out * \partial out/\partial z * \partial z/\partial w7$
- $\partial z/\partial w7 = \partial(a1*w7 + a2*w8 + a3*w9 + b2 * 1)/\partial w7$

  $= a1 + 0 + 0$

  $= a1 = 0.9995$





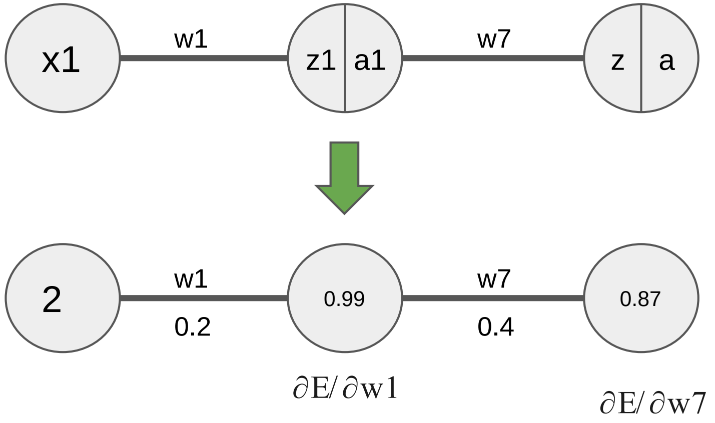
Chain rule illustration

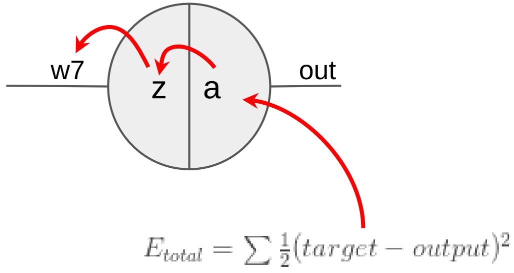
The Backward Pass

- $W^{k+1} = W^{k}$ - learning_rate*(gradient)
- For backward pass, we need to calculate derivatives (gradient) i.e. $\partial E/\partial w7$ and $\partial E/\partial w1$
- Using chain rule: $\partial E/\partial w7 = \partial E/\partial out * \partial out/\partial z * \partial z/\partial w7$

Putting it all together:

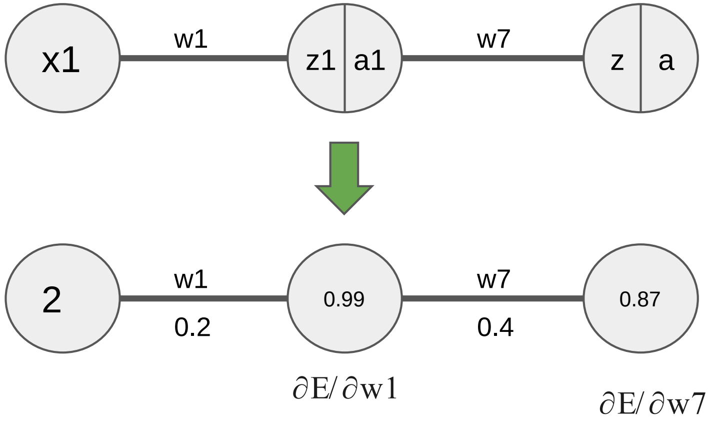
- $\partial E/\partial w7 = -0.1204 * 0.1059 * 0.9995$
  $= -0.0127$





Chain rule illustration

The Backward Pass



- $W^{k+1} = W^k$ - learning_rate*(gradient)
- For backward pass, we need to calculate derivatives (gradient) i.e. $\partial E/\partial w7$ and $\partial E/\partial w1$
- Using chain rule: $\partial E/\partial w7 = \partial E/\partial out * \partial out/\partial z * \partial z/\partial w7$

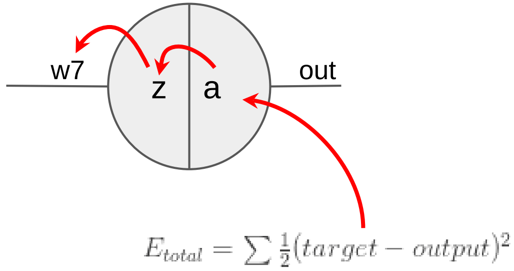
Putting it all together:

- $\partial E/\partial w7 = -0.1204 * 0.1059 * 0.9995$
  $= -0.0127$

To decrease the error, we then subtract this value from the current weight.

- w7 = w7 - learning_rate * (-0.0127)

- **w7 = 0.4 - 0.1 * (-0.0127) = 0.40127** (assume learning_rate = 0.1)

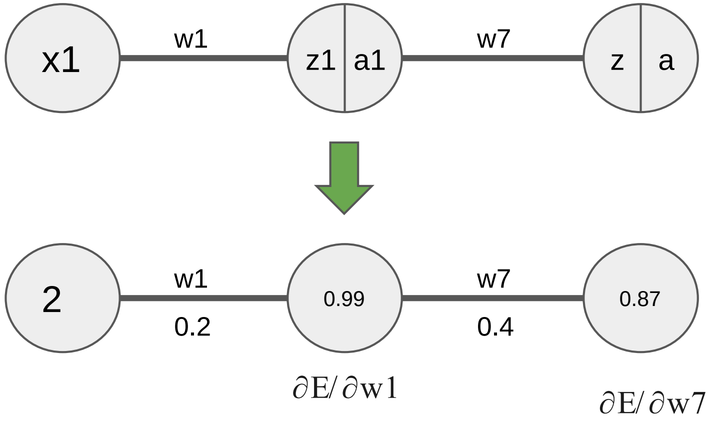


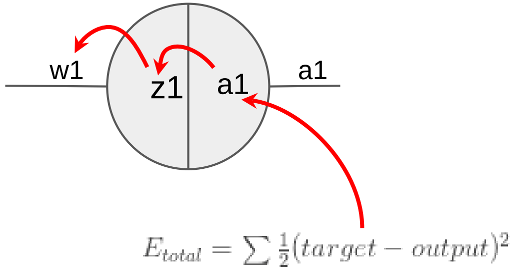
Chain rule illustration

The Backward Pass

- For backward pass, we need to calculate derivatives (gradient) i.e. ∂E/∂w7 and ∂E/∂w1
- Using chain rule for hidden layer: ∂E/∂w1 = ∂E/∂a1 * ∂a1/∂z1 * ∂z1/∂w1
- ∂E/∂a1 = ∂E/∂z * ∂z/∂a1

  Using previously calculated values we have
  ∂E/∂z = ∂E/∂out * ∂out/∂z = -0.1204 * 0.1059 = −0.01275
- ∂z/∂a1 = ∂(a1*w7 + a2*w8 + a3*w9 + b2 * 1)/∂a1 = w7+0+0+0 = w7 = 0.4
- ∂E/∂a1 = ∂E/∂z * ∂z/∂a1 = −0.01275*0.4 = −0.0051





Chain rule illustration

The Backward Pass

- **PLEASE NOTE:** Using chain rule for hidden layer:

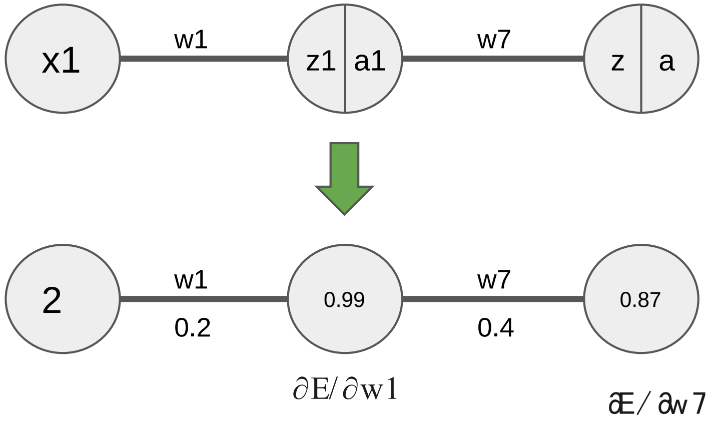
$$\boxed{\partial E/\partial w1 = \partial E/\partial a1 * \partial a1/\partial z1 * \partial z1/\partial w1}$$

simply expands to

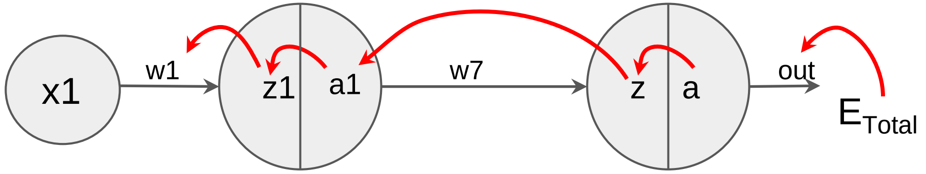
$$\boxed{\partial E/\partial w1 = \partial E/\partial out * \partial out/\partial z * \partial z/\partial a1 * \partial a1/\partial z1 * \partial z1/\partial w1}$$

Compare this to below

$$\boxed{\partial E/\partial w7 = \partial E/\partial out * \partial out/\partial z * \partial z/\partial w7}$$



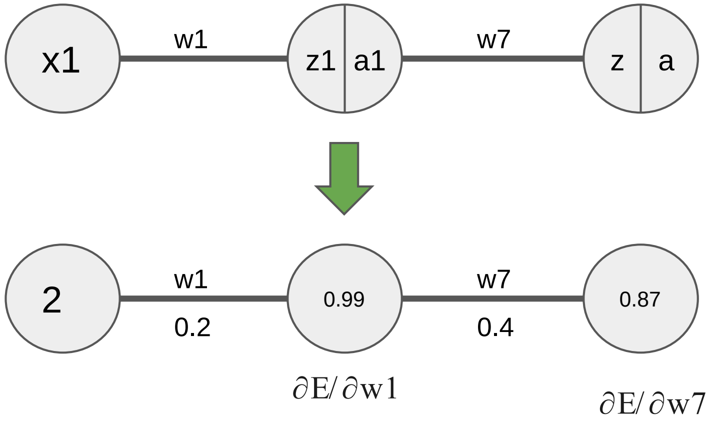
Chain rule illustration for $\partial E/\partial w1$:

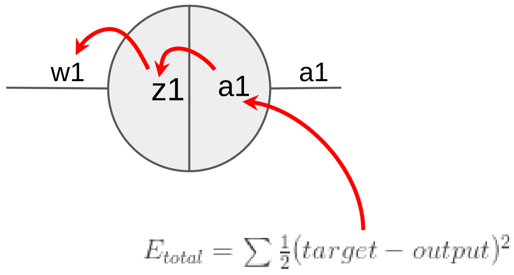


The Backward Pass

- For backward pass, we need to calculate derivatives (gradient) i.e. $\partial E/\partial w7$ and $\partial E/\partial w1$
- Using chain rule for hidden layer: $\partial E/\partial w1 = \partial E/\partial a1 * \partial a1/\partial z1 * \partial z1/\partial w1$
- $\partial a1/\partial z1 = \sigma(1\text{-}\sigma)$ [a1 is also an output of the sigmoid function]
  $= 0.9995\ (1\text{-}0.9995) = 0.00049975$
- $\partial z1/\partial w1 = \partial(w1*x1 + w4*x2 + b1*1)/\partial w1$
  $= x1 + 0 + 0 = 2$

Putting it all together: $\partial E/\partial w1 = \partial E/\partial a1 * \partial a1/\partial z1 * \partial z1/\partial w1$
$= -0.0051 * 0.00049975 * 2 =$ - 5.097E-6





Chain rule illustration

The Backward Pass

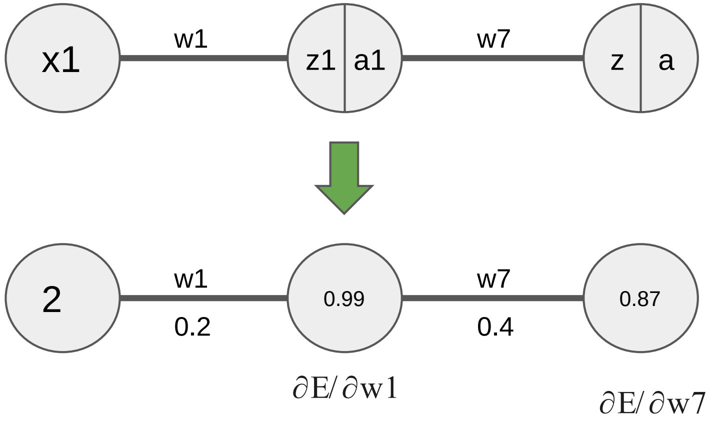
- For backward pass, we need to calculate derivatives (gradient) i.e. ∂E/∂w7 and ∂E/∂w1
- Using chain rule for hidden layer: ∂E/∂w1 = ∂E/∂a1 * ∂a1/∂z1 * ∂z1/∂w1
- ∂a1/∂z1 = σ(1-σ) [a1 is also an output of the sigmoid function]
  = 0.9995 (1-0.9995) = 0.00049975
- ∂z1/∂w1 = ∂(w1*x1 + w4* x2 + b1*1)/∂w1
  = x1 + 0 + 0 = 2

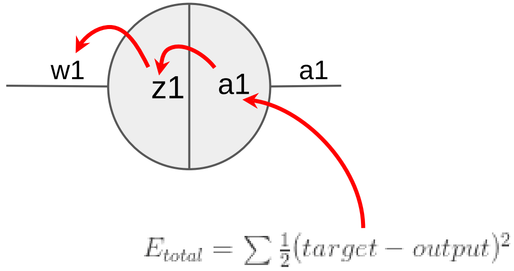
Putting it all together: ∂E/∂w1 = ∂E/∂a1 * ∂a1/∂z1 * ∂z1/∂w1

= −0.0051 * 0.00049975 * 2 = - 5.097E-6

Updating w1 = w1 - learning_rate * gradient

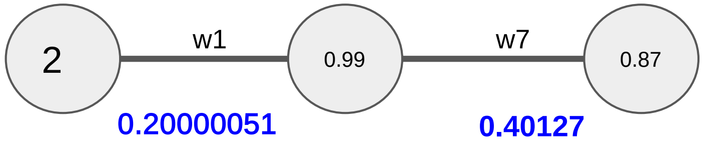
w1 = 0.2 - 0.1*(-5.097E-6) = 0.20000051





Chain rule illustration

After update, new weights would be

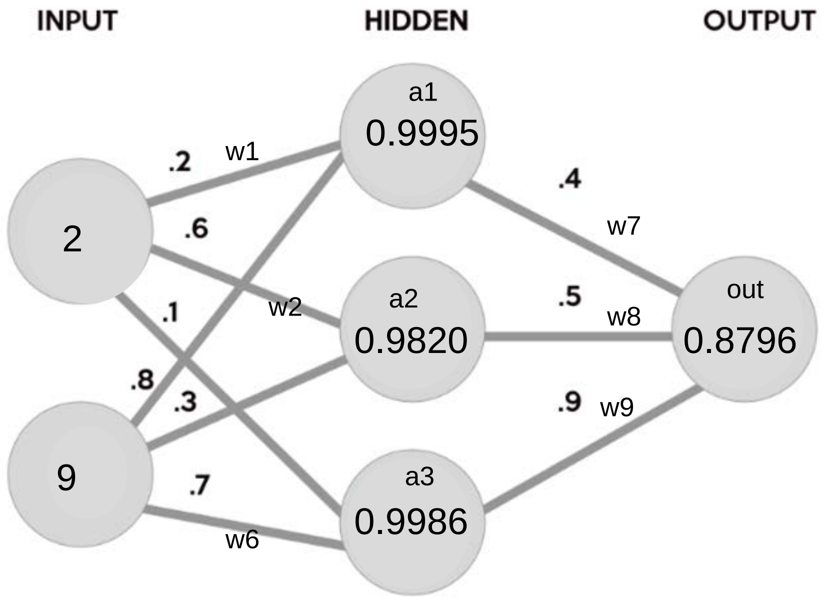


- Perform the forward pass and backward pass steps iteratively until the loss reaches minimal value.

Practice Calculations:

After one iteration, the updated weights for w2, w6, w8, and w9 are:

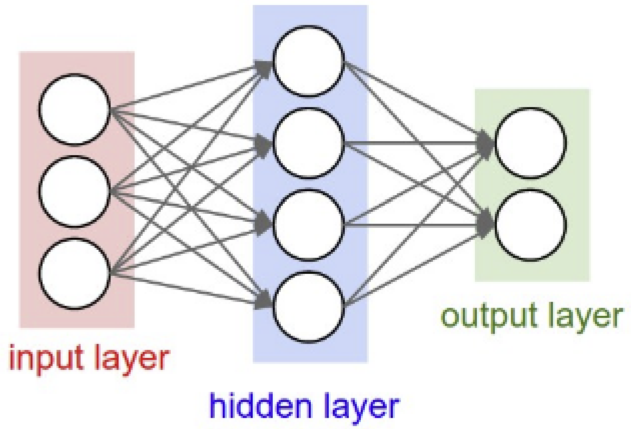
- w8 = 0.50125
- w9 = 0.90127325094
- w2 = 0.60002253753
- w6 = 0.70001443866

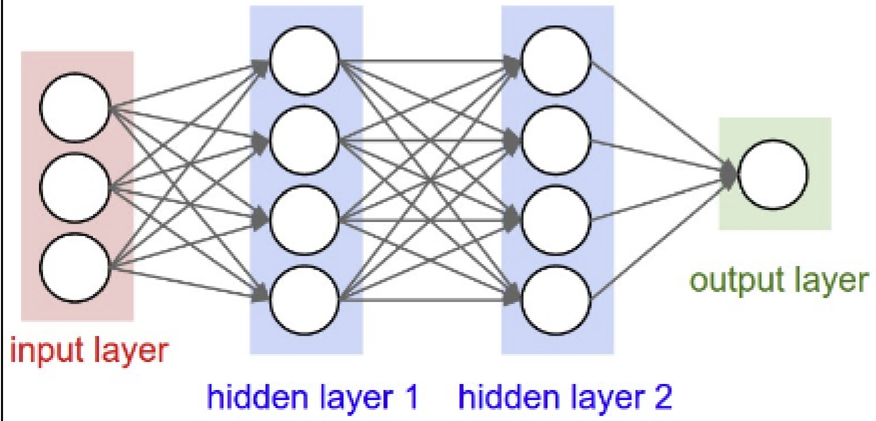


Summary

- The whole process of Forward propagation and backpropagation constitute a single iteration.
- This process is iteratively repeated until loss value reaches a minimum value and weights become stable.
- In practice, we don’t train the model on single example, rather we train it on many many different examples and the model weights are updated slowly towards a convergence point.
- When we train the model on different examples, the model learns weights to produce the output close to the target output.

Layers organization in a Neural Network





Source: http://cs231n.github.io/neural-networks-1/#intro