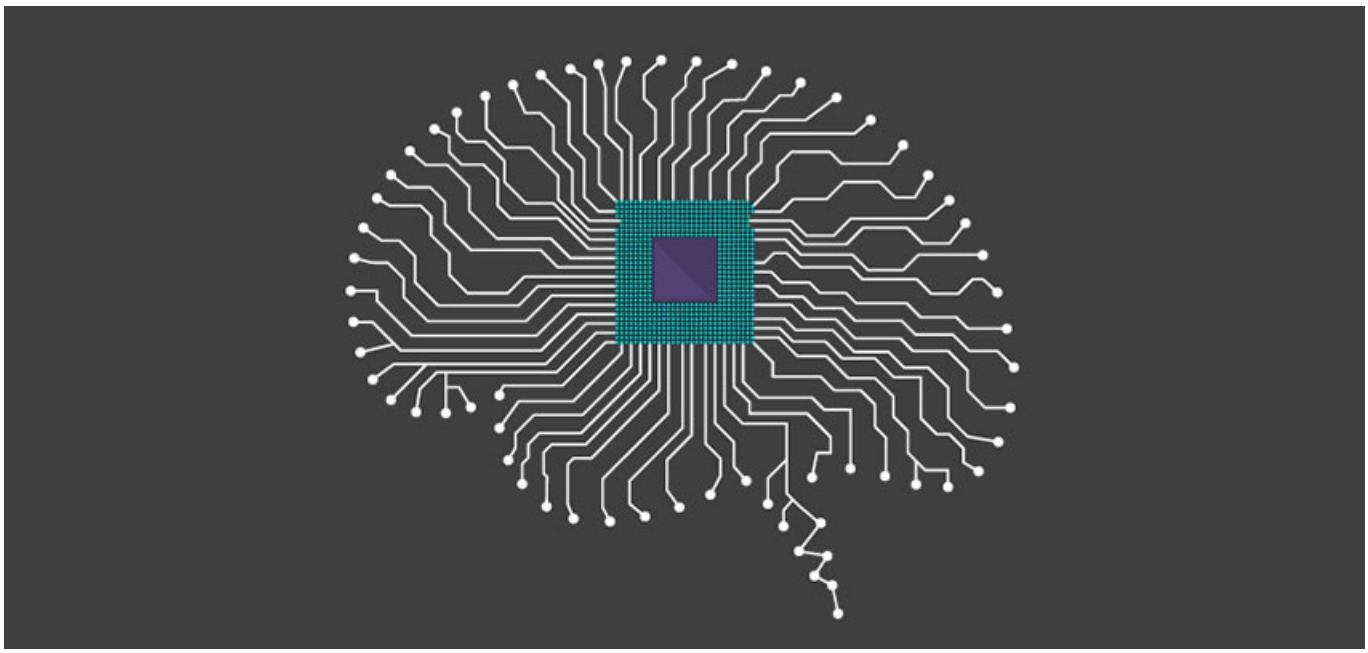
Artificial Neural Networks Explained

Aegeus Zerium Follow
Jul 24, 2018 · 9 min read

in this post we will explain everything you need to know about Artificial Neural Networks in a theoretical and programmatic way.



Definition:

Artificial neural networks (ANNs) or connectionist systems are computing systems vaguely inspired by the biological neural networks that constitute animal brains. Such systems “learn” to perform tasks by considering examples, generally without being programmed with any task-specific rules.

The Architecture of an Artificial Neural Network:

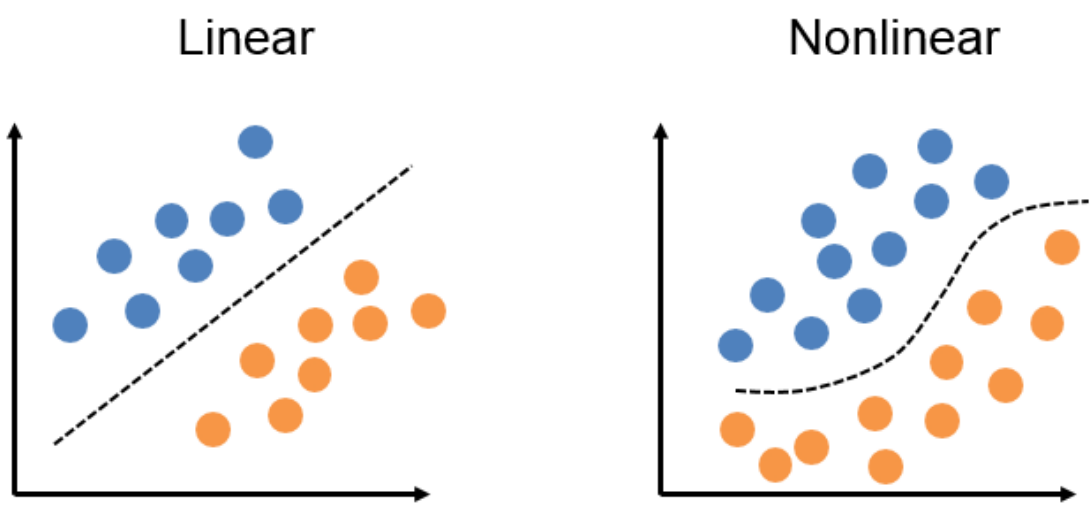
ANN is a set of connected neurons organized in layers:

- **input layer:** brings the initial data into the system for further processing by subsequent layers of artificial neurons.
- **hidden layer:** a layer in between input layers and output layers, where artificial neurons take in a set of weighted inputs and produce an output through an activation function.
- **output layer:** the last layer of neurons that produces given outputs for the program.

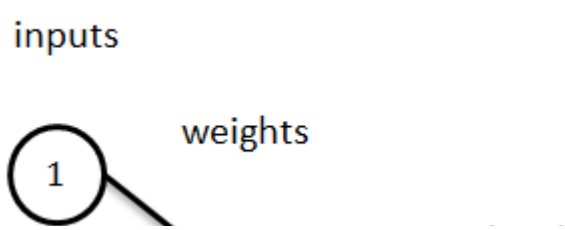
Types of ANNs:

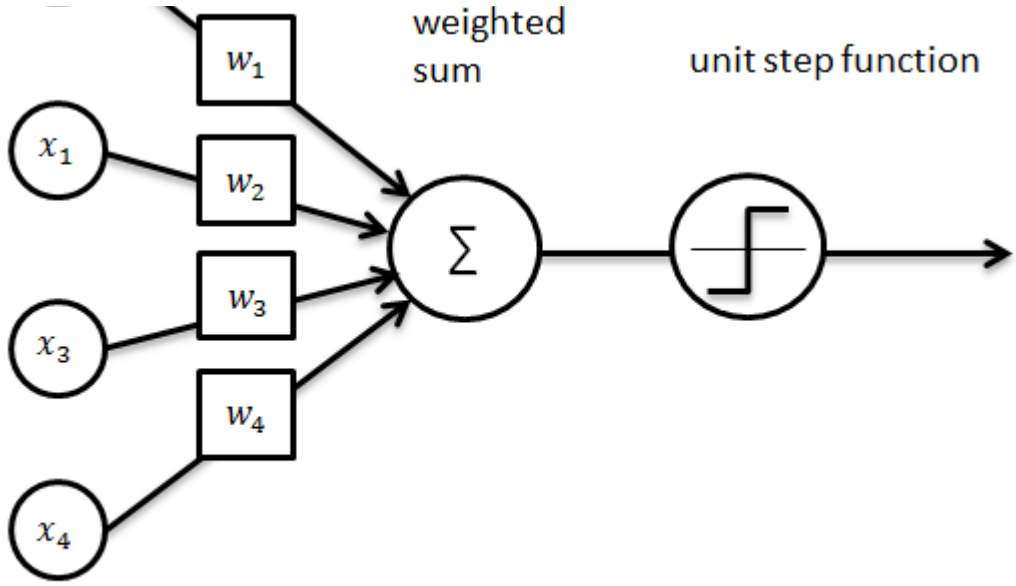
Perceptron:

The simplest and oldest model of an ANN, the Perceptron is a linear classifier used for binary predictions. This means that in order for it to work, the data must be linearly separable.



Its Architecture:

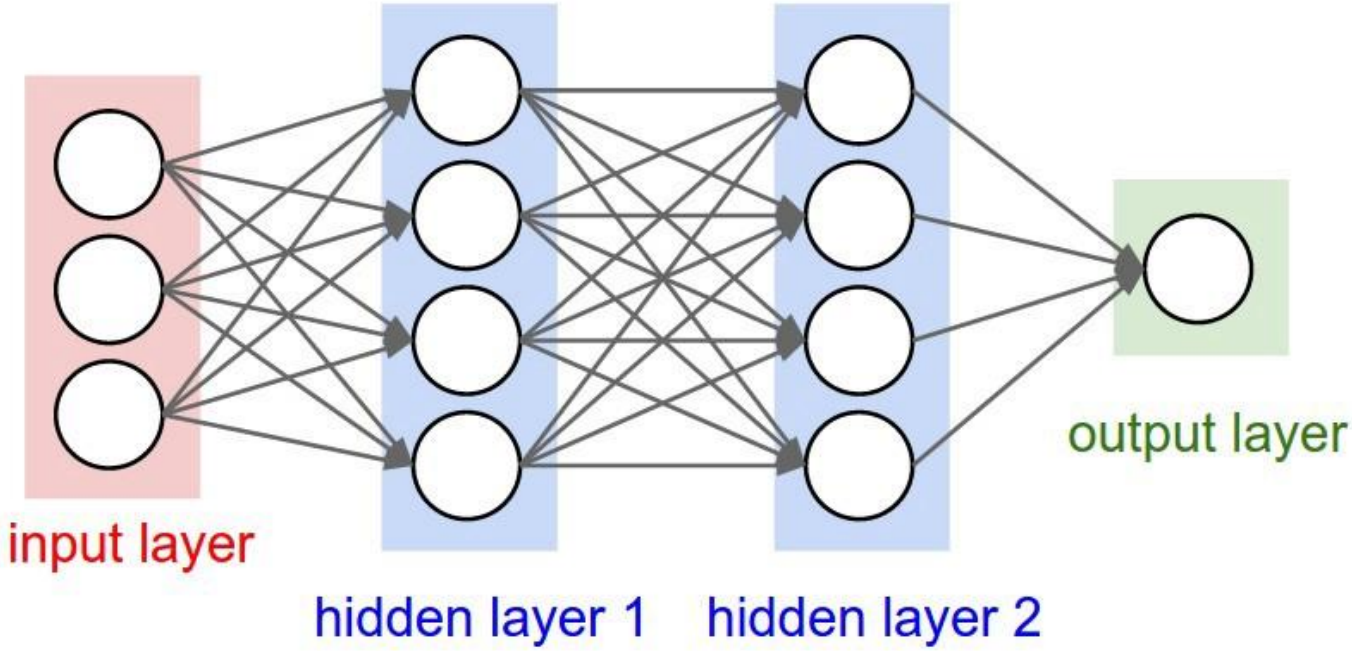




Multi-layer ANN:

More sophisticated than the perceptron, a Multi-layer ANN (e.g.: Convolutional Neural Network, Recurrent Neural Network etc …) is capable of solving more complex classification and regression tasks thanks to its hidden layer(s).

Its Architecture:



Activation Functions:

Definition:

In artificial neural networks, the **activation function** of a node defines the output of that node given an input or set of inputs.

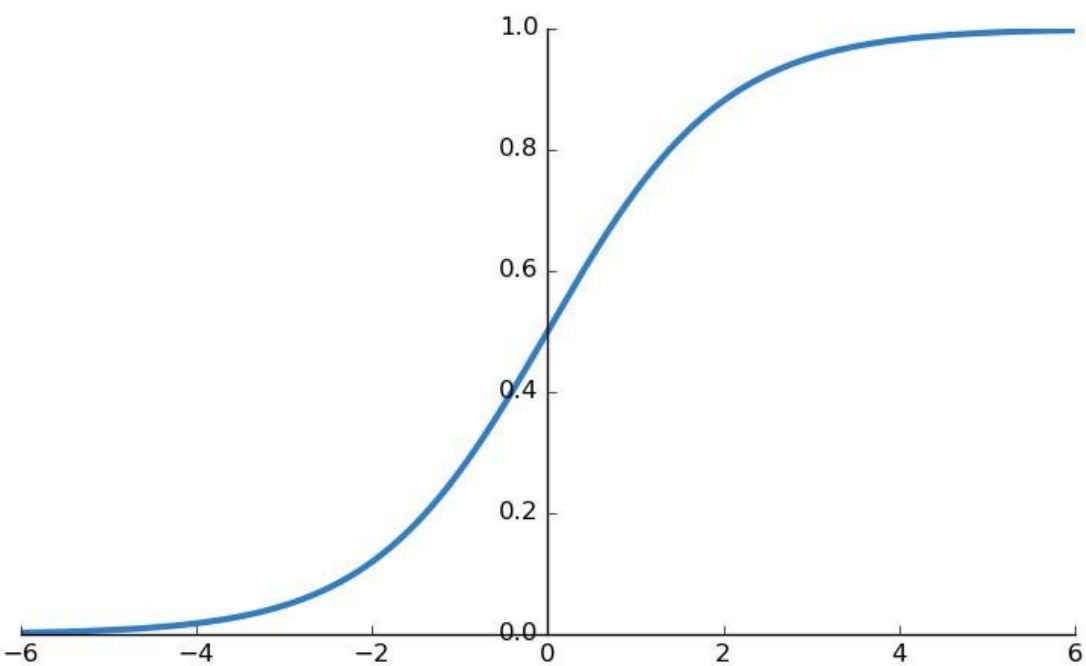
Sigmoid:

A **sigmoid function** is a mathematical function having a characteristic “S”-shaped curve or sigmoid curve. Often, sigmoid function refers to the special case of the logistic function which generate a set of probability outputs between 0 and 1 when fed with a set of inputs. The sigmoid activation function is widely used in binary classification.

Equation:

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

Graphical Representation:



Programmatically:

```
# import our dependencies
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()
from numpy import exp, array, append

# Sigmoid activation function
def Sigmoid(x):
    return 1 / (1 + exp(-x))

# Graphical representation of our Sigmoid activation function
for i in array([range(-10, 10)]):
    x = append(array([]), i)
    plt.plot(x, Sigmoid(x))
    plt.title("Sigmoid activation function")
    plt.xlabel("x")
    plt.ylabel("S(x)")
```

Sigmoid.py hosted with ♡ by GitHub view raw

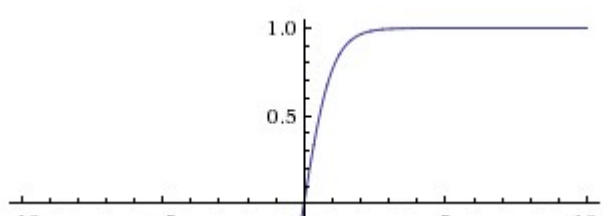
Tan-h:

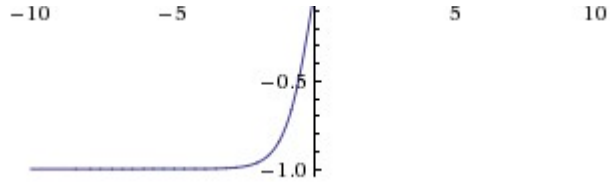
An alternative to the logistic sigmoid is the **hyperbolic tangent**, or tan-h function. Like the logistic sigmoid, the tan-h function is also sigmoidal ("S"-shaped), but instead outputs values that range [-1, 1]. Thus strongly negative inputs to the tan-h will map to negative outputs.

Equation:

$$tanh(x) = \frac{2}{1 + e^{-2x}} - 1 = 2 \times Sigmoid(2x) - 1$$

Graphical Representation:





Programmatically:

```
# Tan-h activation function
def Tanh(x):
    return 2 / (1 + exp(-2*x)) - 1

# Graphical representation of our tan-h activation function
for i in array([range(-10, 10)]):
    x = append(array([]), i)
    plt.plot(x, Tanh(x))
    plt.title("Tan-h activation function")
    plt.xlabel("x")
    plt.ylabel("tanh(x)")
```

Tan-h.py hosted with ♡ by GitHub view raw

Softmax:

Unlike the Sigmoid activation function, the **Softmax activation function** is used for multi-class classification. Softmax function calculates the probabilities distribution of the event over 'n' different events. In general way of saying, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will be helpful for determining the target class for the given inputs.

Equation:

$$Softmax(x) = \frac{e^j}{\sum_i e^i}$$

Programmatically:

```
from numpy import sum

# Softmax activation function
```

```
def Softmax(x):
    return exp(x) / sum(exp(x))

x = array([1, 2, 3, 4, 5])

Softmax(x)

# Result: array([ 0.01165623,  0.03168492,  0.08612854,  0.23412166,  0.63640865])
```

Softmax.py hosted with ♡ by GitHub view raw

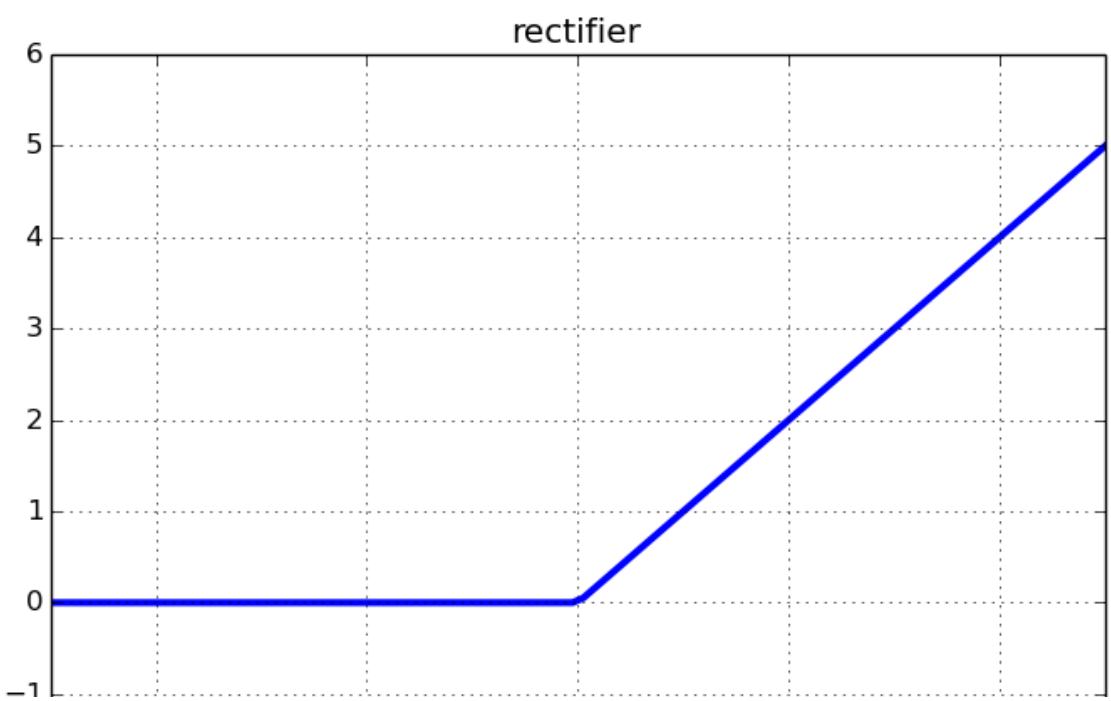
ReLU:

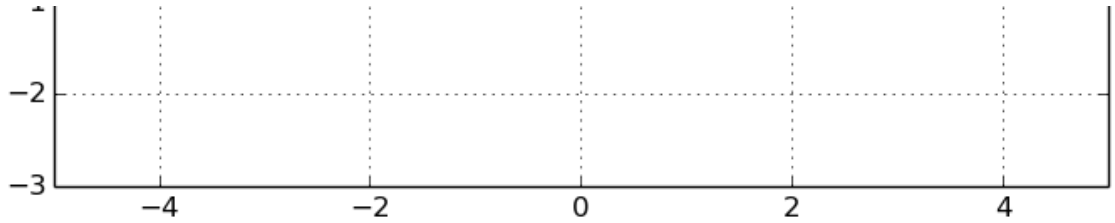
Instead of the sigmoid activation function, most recent artificial neural networks use **rectified linear units** (ReLUs) for the hidden layers. A rectified linear unit has output 0 if the input is less than 0, and raw output otherwise. That is, if the input is greater than 0, the output is equal to the input.

Equation:

$$ReLU(x) = max(0, x)$$

Graphical Representation:





Programmatically:

```
# ReLU activation function
def ReLU(x):
    return x * (x > 0)

# Graphical representation of our ReLU activation function
x = array(range(-10, 10))
plt.plot(x, ReLU(x))
plt.title("ReLU activation function")
plt.xlabel("x")
plt.ylabel("ReLU(x)")
```

ReLU.py hosted with ♡ by GitHub view raw

Leaky ReLU:

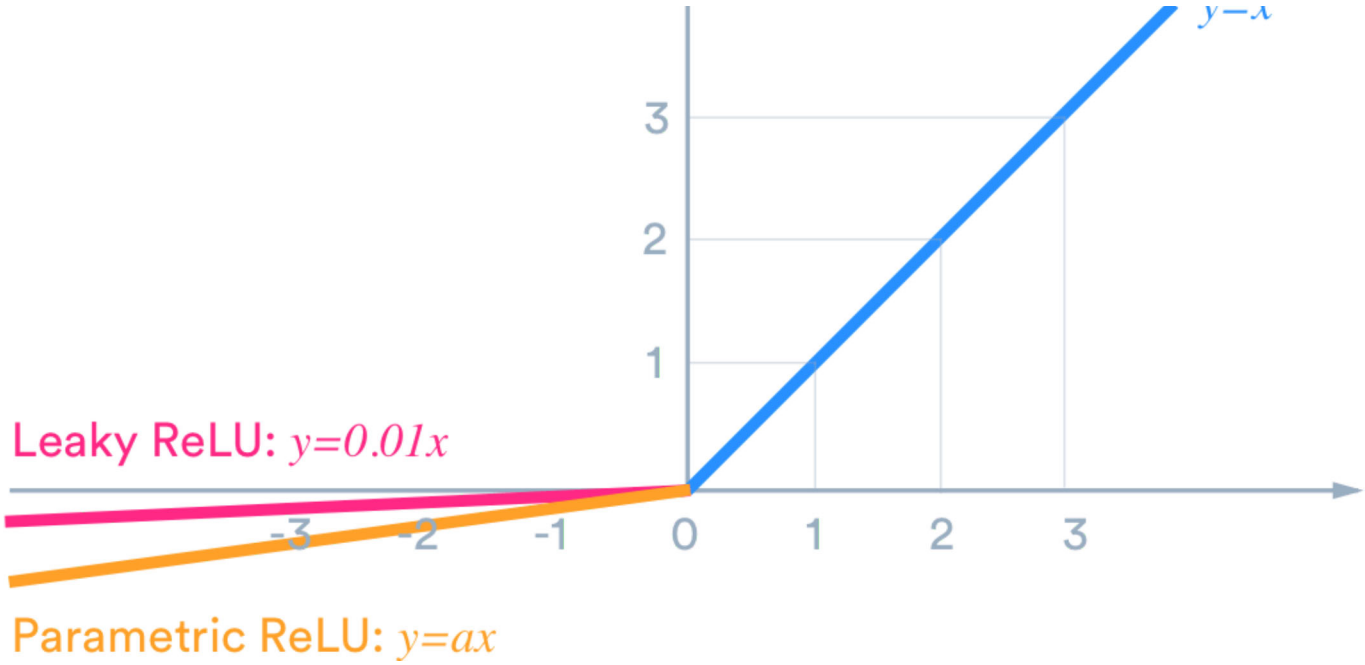
Definition:

The **Leaky ReLU** activation function works the same way as the ReLU activation function except that instead of replacing the negative values of the inputs with 0 the latter get multiplied by a small alpha value in an attempt to avoid the “dying ReLU” problem.

Equation:

$$LeakyReLU(x) = \begin{cases} x, & \text{if } x > 0. \\ \alpha x, & \text{otherwise.} \end{cases}$$

Graphical Representation:



Programmatically:

```
from numpy import where

# Leaky ReLU activation function
def LeakyReLU(x):
    return where(x > 0, x, x * 0.1)

# Graphical representation of our Leaky ReLU activation function
x = array(range(-10, 10))
plt.plot(x, LeakyReLU(x))
plt.title("Leaky ReLU activation function")
plt.xlabel("x")
plt.ylabel("Leaky ReLU(x)")
```

LeakyReLU.py hosted with ♡ by GitHub view raw

Why do we use activation functions ?

You are probably asking yourself after reading the whole Activation Function section about the useflness of the latter, why even bother using it ? why don't we just use the initial dot product output without feeding it to the activation function ?

Well, without an activation function we will fail to introduce non-linearity into the network.

An activation function will allow us to model a response variable (target variable, class label, or score) that varies non-linearly with its explanatory variables.

non-linear means that the output cannot be reproduced from a linear combination of the inputs.

another way to think of it: without a non-linear activation function in the network, an artificial neural network, no matter how many layers it has, will behave just like a single-layer perceptron, because summing these layers would give you just another linear function.

Which activation function is the "best" to use ?

There are many activation functions out there but which one is the best to use? This question has been asked by many beginners in the deep learning field (including myself).

The short answer for this question is: There is no "best" activation function.

The long answer is: The use of an activation function depends on the task you are dealing with, is the task a regression or a classification problem? if it's a classification problem then is it a binary classification or a multiclass classification task? it all comes down to the task you are dealing with.

for example we can't possibly use the Sigmoid or tan-h activation functions for a multiclass classification task we would better be using the Softmax activation function instead. right? as it would output a vector of probabilities of each class. another example, which activation would be better to use in the hidden layer part of a multi-layered neural net? if you use the sigmoid activation function then you would be hit by the vanishing gradient problem (the vanishing gradient problem is when the gradient becomes so small in the earlier layers of a deep neural network to the point that it barely affects the weights of the earlier layers thus failing at optimizing our initial weights.), should we use the tan-h activation function then? not necessarily, it's true that the tan-h function is an enhanced version of the sigmoid function but the vanishing gradient problem is still persistently there. So, the solution to our persistent problem is the the ReLU activation function, but how does it solve the vanishing gradient problem? the fact that the derivative of the ReLU activation function is 1 if its inputs are positive will

assure that the gradient will not get infinitesimally smaller passing from one layer to another but what if our inputs are negative? here we get stuck with what we call the "dying ReLU" problem (the dying ReLU problem is when the ReLU activation function always outputs the same value (zero) for any input. That means that it takes no role in discriminating between inputs. Once a ReLU ends up in this state, it is unlikely to recover, because the function gradient at 0 is also 0, so gradient descent learning will not alter the weights.), here comes the Leaky/Parametric ReLU to rescue and instead of outputting a flat out zero for the negative values the Leaky ReLU multiplies the negative values by an alpha parametric value ($\alpha=0.01$).

As you can see from the previous explanation, the use of a certain activation function depends on the task you are dealing with and the problems you face while building your neural network.

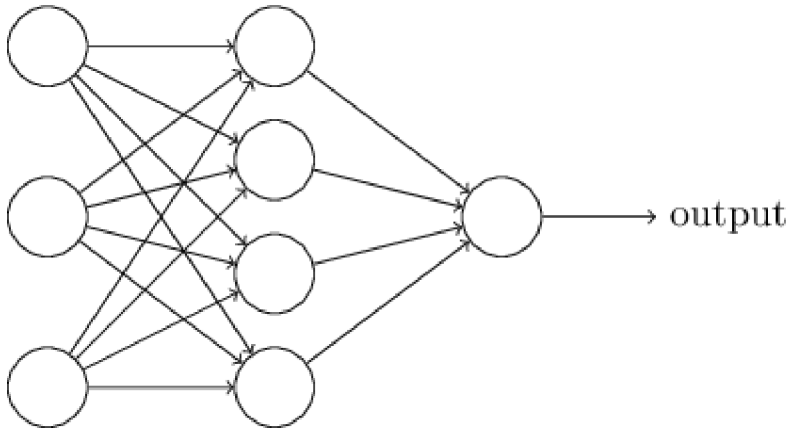
Forward Propagation:

Definition:

Forward propagation is the process of feeding the Neural Network with a set of inputs to get their dot product with their weights then feeding the latter to an activation function and comparing its numerical value to the actual output called "the ground truth".

Calculations:

Let's demonstrate how the output gets calculated in a forward pass in a 3–4–1 Artificial Neural Network:



The initial weights of the first and second layer:

$$W_{ij} = \begin{bmatrix} 0.1 & 0.2 & 0.4 & 0.3 \\ 0.5 & 0.4 & 0.7 & 0.9 \\ 0.2 & 0.6 & 0.3 & 0.8 \end{bmatrix} and\ W_{jk} = \begin{bmatrix} 0.2 \\ 0.3 \\ 0.6 \\ 0.1 \end{bmatrix}$$

Input:

$$i = \begin{bmatrix} 0.1 \\ 0.4 \\ 0.5 \end{bmatrix}$$

output:

$$o = \begin{bmatrix} 1 \end{bmatrix}$$

The activation function of the hidden layer will be the ReLU function and the activation function of the output layer will be the Sigmoid function.

let's begin our calculculations:

```
# import our dependencies
from numpy import dot

i = array([[0.1, 0.4, 0.5]])
w_ij = array([[0.1, 0.2, 0.4, 0.3],
              [0.5, 0.4, 0.7, 0.9],
              [0.2, 0.6, 0.3, 0.8]])
w_jk = array([[0.2],
              [0.3],
              [0.6],
              [0.1]])

z_ij = dot(i, w_ij) # calculating the dot product of the inputs and weights of the hidden layer
a_ij = ReLU(z_ij) # applying the ReLU activation function
z_jk = dot(a_ij, w_jk) # the dot product of the inputs in the hidden layer after activation and
```

```
a_jk = sigmoid(z_jk) # applying the sigmoid activation function
# The final output of the feed forward neural network
a_jk
# the predicted output "a_jk" equals 0.63807065
```

ForwardPropagation.py hosted with ♡ by GitHub view raw

Cross entropy error:

Cross-entropy loss, or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. Cross-entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value. A perfect model would have a log loss of 0.

Equation:

$$L = -\sum_i (y'_i log(y_i) + (1 - y'_i) log(1 - y_i))$$

> Where ‘yi’ is the predicted probability value for class ‘i’ and ‘y′i’ is the true probability for that class.

Programmatically:

```
from numpy import log10

# cross entropy error function
def crossentropyerror(a, y):
    return - sum(y * log10(a) + (1 - y) * log10(1 - a))

a = a_jk # the output of the neural network
y = array([[1]]) # the ground truth
crossentropyerror(a, y)

# the cross entropy error equals 0.19513122954797735
```

CrossEntropyError.py hosted with ♡ by GitHub view raw

Conclusion:

This is in a nutshell how a forward propagation works and how a neural network generates its predictions, the forward propagation step has always been less mathematically intense than the backpropagation step where most students and beginners find legitimately harder to grasp.

Back-propagation:

Definition:

Backpropagation is a method used in artificial neural networks to calculate a gradient that is needed in the calculation of the weights to be used in the network.

Calculations:

first let's lay some important derivatives:

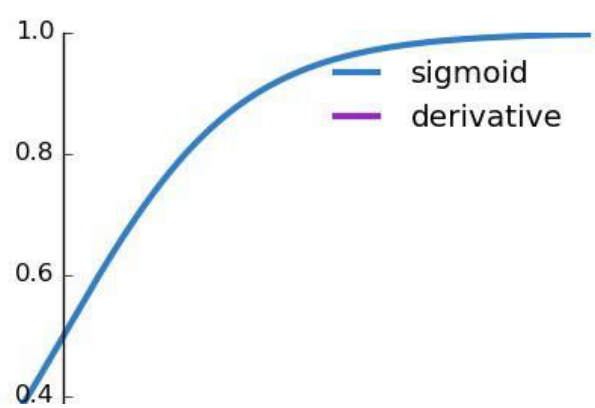
Cross entropy error derivative:

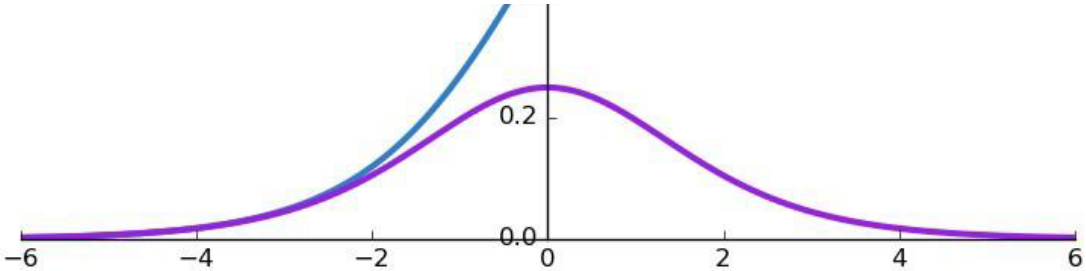
$$\frac{\partial L}{\partial y_i} = \frac{-y'_i}{y_i} + \frac{(1 - y'_i)}{(1 - y_i)}$$

Sigmoid derivative:

$$\frac{\partial Sigmoid(x)}{\partial x} = Sigmoid(x) \times (1 - Sigmoid(x))$$

Graphical Representation:





Programmatically:

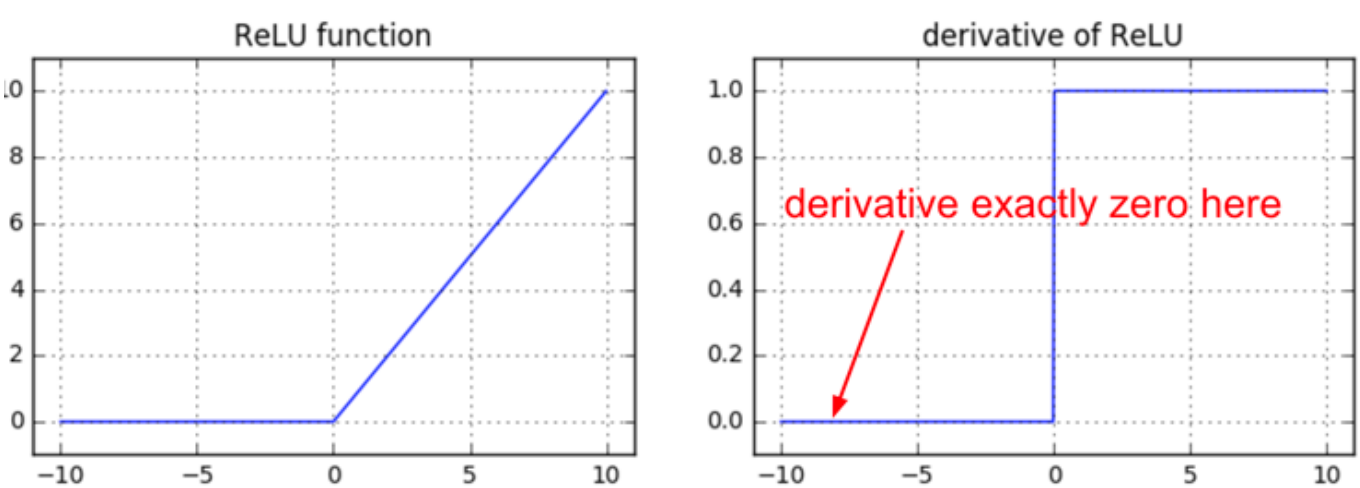
```
# Derivative of the Sigmoid activation function
def SigmoidDerivative(x):
    return Sigmoid(x) * (1 - Sigmoid(x))

# Graphical representation of the Sigmoid derivative
x = array(range(-10, 10))
plt.plot(x, SigmoidDerivative(x))
plt.title("Derivative of Sigmoid")
plt.xlabel("x")
plt.ylabel("Sigmoid Derivative")
```

SigmoidDerivative.py hosted with ♡ by GitHub view raw

ReLU derivative:

$$\frac{\partial ReLU(x)}{\partial x} = \begin{cases} 1, & \text{if } x > 0. \\ 0, & \text{otherwise.} \end{cases}$$

Graphical Representation:



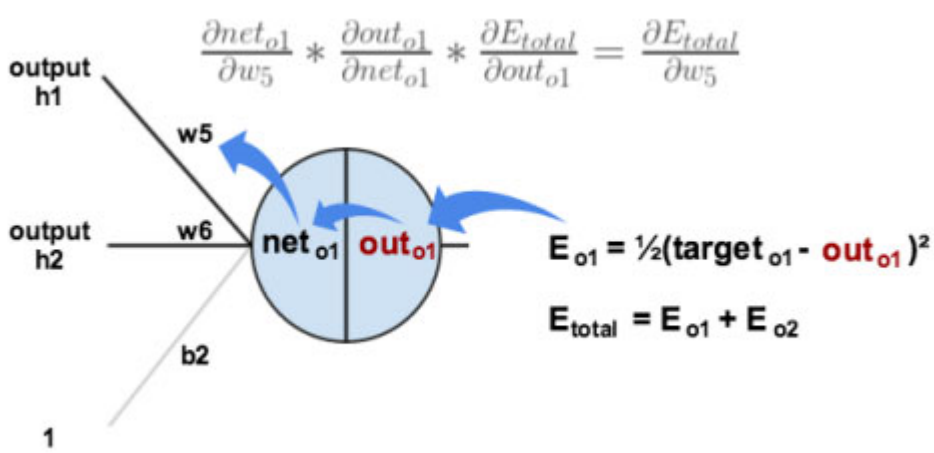
Programmatically:

```
# Derivative of the ReLU activation function
def ReLUDerivative(x):
    return 1. * (x > 0)

# Graphical representation of the ReLU derivative
x = array(range(-10, 10))
plt.plot(x, ReLUDerivative(x))
plt.title("Derivative of ReLU")
plt.xlabel("x")
plt.ylabel("ReLU Derivative")
```

ReLUDerivative.py hosted with ♡ by GitHub view raw

Let's now demonstrate the calculations that go underneath the Backpropagation process:

An illustration of how a neural network backpropagates its error:



Programmatically:

```
# Backpropagation from the output layer to the hidden layer
dl_jk = -y/a_jk + (1 - y)/(1 - a_jk) # the derivative of the cross entropy loss wrt output
da_jk = SigmoidDerivative(z_jk) # the derivative of Sigmoid wrt the input (before activation) of
dz_jk = a_ij # the derivative of the inputs of the hidden layer (before activation) wrt weights
gradient_jk = dot(dz_jk.T , dl_jk * da_jk) # chain rule
gradient_jk = array([[-0.1121981 ],
                     [-0.17372609],
```

```
                    [-0.17010679],
                    [-0.28592418]])

# Backpropagation from the hidden layer to the output layer
dl_ij = dot(da_jk * dl_jk, w_jk.T) # the derivative of the cross entropy loss wrt hidden layer i
da_ij = ReLUDerivative(z_ij) # the derivative of the ReLU function wrt the input (before activat
dz_ij = i # the derivative of the inputs of the hidden layer (before activation) wrt weights of
gradient_ij = dot(dz_ij.T , dl_ij * da_ij) # chain rule
gradient_ij = array([[-0.00723859, -0.01085788, -0.02171576, -0.00361929],
       [-0.02895435, -0.04343152, -0.08686304, -0.01447717],
       [-0.03619293, -0.0542894 , -0.1085788 , -0.01809647]])

# The new weights we get after the backpropagation process
w_ij = w_ij - gradient_ij
w_jk = w_jk - gradient_jk

print("New synaptic weights after the backpropagation process: ")
print("w_ij: ", w_ij)
print("w_jk: ", w_jk)

New synaptic weights after the backpropagation process:
w_ij:  [[ 0.10723859  0.21085788  0.42171576  0.30361929]
        [ 0.52895435  0.44343152  0.78686304  0.91447717]
        [ 0.23619293  0.6542894   0.4085788   0.81809647]]
w_jk:  [[ 0.3121981 ]
        [ 0.47372609]
        [ 0.77010679]
        [ 0.38592418]]
```

BackPropagation.py hosted with ♡ by GitHub view raw

let's now predict the output with the new updated weights.

```
i = array([[0.1, 0.4, 0.5]])

# new hidden layer weights
w_ij = array([[ 0.10723859,  0.21085788,  0.42171576,  0.30361929],
              [ 0.52895435,  0.44343152,  0.78686304,  0.91447717],
              [ 0.23619293,  0.6542894,   0.4085788,   0.81809647]])

# new output layer weights
w_jk = array([[ 0.3121981 ],
              [ 0.47372609],
```

```
                [ 0.77010679],
                [ 0.38592418]])

z_ij = dot(i, w_ij) # calculating the dot product of the inputs and weights of the hidden layer
a_ij = ReLU(z_ij) # applying the ReLU activation function
z_jk = dot(a_ij, w_jk) # the dot product of the inputs in the hidden layer after activation and
a_jk = Sigmoid(z_jk) # applying the Sigmoid activation function

# the new predicted output as you will see is closer to the ground truth than the initial output
a_jk = array([[ 0.74992269]])
```

ResultsafterOptimization.py hosted with ♡ by GitHub view raw

Congratulations! you've just done forward and backpropagation with just a few lines of code.

Building a one hidden layer Neural Network:

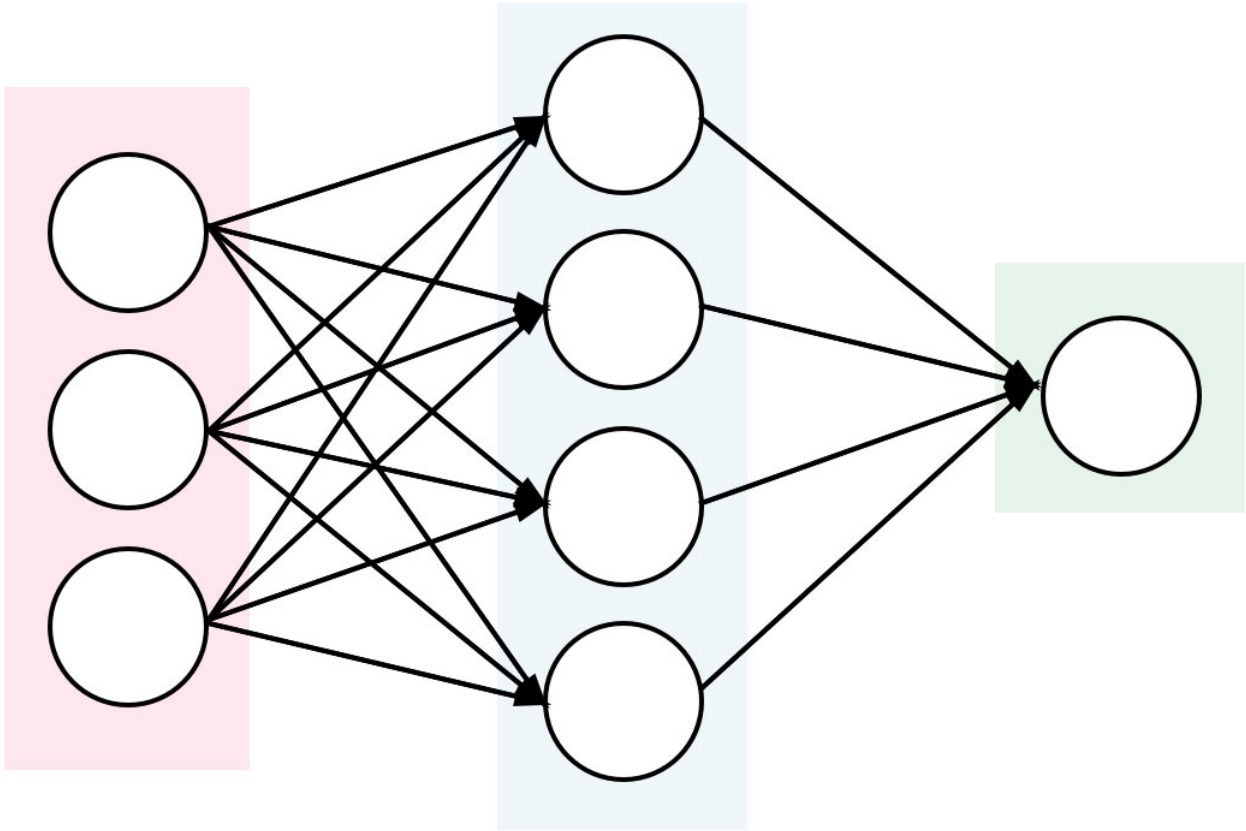
let's now build a fully functional multilayer artificial neural network, our neural network will tackle the XOR task (where the third column of the table is irrelevant).

	Input			Output
Example 1	0	0	1	0
Example 2	0	1	1	1
Example 3	1	0	1	1
Example 4	0	1	0	1
Example 5	1	0	0	1
Example 6	1	1	1	0
Example 7	0	0	0	0

New situation	1	1	0	?

The classification task

the ANN's architecture will look like this:



The ANN we'll build

Code:

```
# Import our dependencies
from numpy import exp, array, random, dot, ones_like, where, log10

# Create our Artificial Neural Network class
class ArtificialNeuralNetwork():

    # initializing the class
    def __init__(self):

        # generating the same synaptic weights every time the program runs
        random.seed(1)

```

```
        # synaptic weights (3 × 4 Matrix) of the hidden layer
        self.w_ij = 2 * random.rand(3, 4) - 1

        # synaptic weights (4 × 1 Matrix) of the output layer
        self.w_jk = 2 * random.rand(4, 1) - 1

    def Sigmoid(self, x):

        # The Sigmoid activation function will turn every input value into probabilities betwee
        # the probabilistic values help us assert which class x belongs to

        return 1 / (1 + exp(-x))

    def SigmoidDerivative(self, x):

        # The derivative of Sigmoid will be used to calculate the gradient during the backpropa
        # and help optimize the random starting synaptic weights

        return x * (1 - x)

    def crossentropyerror(self, a, y):

        # The cross entropy loss function
        # we use it to evaluate the performance of our model

        return - sum(y * log10(a) + (1 - y) * log10(1 - a))

    def train(self, x, y, learning_rate, iterations):

        # x: training set of data
        # y: the actual output of the training data

        for i in range(iterations):

            z_ij = dot(x, self.w_ij) # the dot product of the weights of the hidden layer and t
            a_ij = self.Sigmoid(z_ij) # applying the Sigmoid activation function

            z_jk = dot(a_ij, self.w_jk) # the same previous process will be applied to find the
            a_jk = self.Sigmoid(z_jk)

            dl_jk = -y/a_jk + (1 - y)/(1 - a_jk) # the derivative of the cross entropy loss wrt
            da_jk = self.SigmoidDerivative(a_jk) # the derivative of Sigmoid  wrt the input (be
            dz_jk = a_ij # the derivative of the inputs of the hidden layer (before activation)

            dl_ij = dot(da_jk * dl_jk, self.w_jk.T) # the derivative of cross entropy loss wrt
```

```
            dl_ij = dot(da_jk * dl_jk, self.w_jk.T) # the derivative of cross entropy loss wrt
            da_ij = self.SigmoidDerivative(a_ij) # the derivative of Sigmoid wrt the inputs of
            dz_ij = x # the derivative of the inputs of the hidden layer (before activation) wr

            # calculating the gradient using the chain rule
            gradient_ij = dot(dz_ij.T , dl_ij * da_ij)
            gradient_jk = dot(dz_jk.T , dl_jk * da_jk)

            # calculating the new optimal weights
            self.w_ij = self.w_ij - learning_rate * gradient_ij
            self.w_jk = self.w_jk - learning_rate * gradient_jk

            # printing the loss of our neural network after each 1000 iteration
            if i % 1000 == 0 in range(iterations):
                print("loss: ", self.crossentropyerror(a_jk, y))

    def predict(self, inputs):

        # predicting the class of the input data after weights optimization

        output_from_layer1 = self.Sigmoid(dot(inputs, self.w_ij)) # the output of the hidden la

        output_from_layer2 = self.Sigmoid(dot(output_from_layer1, self.w_jk)) # the output of t

        return output_from_layer1, output_from_layer2

    # the function will print the initial starting weights before training
    def SynapticWeights(self):

        print("Layer 1 (4 neurons, each with 3 inputs): ")

        print("w_ij: ", self.w_ij)

        print("Layer 2 (1 neuron, with 4 inputs): ")

        print("w_jk: ", self.w_jk)


def main():

    ANN = ArtificialNeuralNetwork()

    ANN.SynapticWeights()

    # the training inputs
```

```
    # the last column is used to add non linearity to the clasification task
    x = array([[0, 0, 1],
               [0, 1, 1],
               [1, 0, 1],
               [0, 1, 0],
               [1, 0, 0],
               [1, 1, 1],
               [0, 0, 0]])

    # the training outputs
    y = array([[0, 1, 1, 1, 1, 0, 0]]).T

    ANN.train(x, y, 1, 10000)

    # Printing the new synaptic weights after training
    print("New synaptic weights after training: ")
    print("w_ij: ", ANN.w_ij)
    print("w_jk: ", ANN.w_jk)

    # Our prediction after feeding the ANN with new set of data
    print("Considering new situation [1, 1, 0] -> ?: ")
    print(ANN.predict(array([[1, 1, 0]])))

if __name__=="__main__":
    main()
```

Artificial Neural Network.py hosted with ♡ by GitHub view raw

Your results should be like this:

```
Layer 1 (4 neurons, each with 3 inputs):
w_ij:  [[-0.16595599  0.44064899 -0.99977125 -0.39533485]
 [-0.70648822 -0.81532281 -0.62747958 -0.30887855]
 [-0.20646505  0.07763347 -0.16161097  0.370439  ]]
Layer 2 (1 neuron, with 4 inputs):
w_jk:  [[-0.5910955 ]
 [ 0.75623487]
 [-0.94522481]
 [ 0.34093502]]
loss:  2.07175185207
loss:  0.0252235360038
loss:  0.00812448219046
loss:  0.00483745527192
```

```
loss:  0.00344285541416
loss:  0.00267203784984
loss:  0.00218307936662
loss:  0.0018453146627
loss:  0.00159802272115
loss:  0.0014091540602
New synaptic weights after training:
w_ij:  [[  0.15412064   5.61526375  -6.92492097 -10.55136518]
 [  0.13145897 -10.51956567  -6.93646143   5.44436653]
 [ -0.01611526  -0.75294181   0.19272139  -0.54865495]]
w_jk:  [[-13.08177073]
 [ 15.15669436]
 [-31.14768507]
 [ 15.0867679 ]]
Considering new situation [1, 1, 0] -> ?:
(array([[  5.70913606e-01,   7.36004539e-03,   9.55163456e-07,
          6.01779348e-03]]), array([[ 0.00069824]]))
```

output.txt hosted with ♡ by GitHub view raw

Vizualization of the loss function:

```
# Visualizing the loss function
weights = array(range(10)) # the weights after each 1000 iteration
loss = array([2.0717518, 0.0252235, 0.0081244, 0.0048374, 0.0034428, 0.0026720, 0.0021830, 0.0018
plt.plot(weights, loss)
plt.title("Loss evaluation")
plt.xlabel("Weights")
plt.ylabel("Loss")
```

Visualization.py hosted with ♡ by GitHub view raw

As you can see in the results the cross entropy loss has dramatically decreased from the first 1000th iteration to the last 1000th iteration which showcase how good our neural network is doing to find the optimal weights.

the prediction output of our Neural Network is 0.00069824 which is very close to the ground truth 0.

Note:

I strongly recommend you check out the notebook to visualize the plots that are missing in this blog post.

Link to Jupyter Notebook:
https://github.com/AegeusZerium/DeepLearning/blob/master/Deep%20Learning/Artificial%20Neural%20Network.ipynb

Conclusion:

With the appearance of Big Data, more Computational power and more challenging and mathematically intense tasks, Neural Networks will be used more than ever.

References:

- https://en.wikipedia.org/wiki/Artificial_neural_network
- https://en.wikipedia.org/wiki/Activation_function
- https://en.wikipedia.org/wiki/Sigmoid_function
- https://theclevermachine.wordpress.com/tag/tanh-function/
- http://dataaspirant.com/2017/03/07/difference-between-softmax-function-and-sigmoid-function/
- https://github.com/Kulbear/deep-learning-nano-foundation/wiki/ReLU-and-Softmax-Activation-Functions
- http://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html
- https://en.wikipedia.org/wiki/Backpropagation

For Contacts:

- Gmail: EternalZer0DayX@gmail.com
- Facebook: https://www.facebook.com/belghouat.khalil
- Twitter: @EternalBlue__

GOOD AUDIENCE

Our newsletter features the latest advancements in deep tech.

Join over 22,000 brilliant minds who have hands-on experience in pushing the limits of innovation in artificial intelligence, blockchain, space technology, agriculture, pandemic relief, and much more.

Email

Sign up

Formed on Upscribe

Connect with the Raven team on Telegram

Machine Learning Deep Learning Neural Networks Artificial Intelligence Python

About Help Legal

Get the Medium app

Download on the App Store

GET IT ON Google Play