Introduction

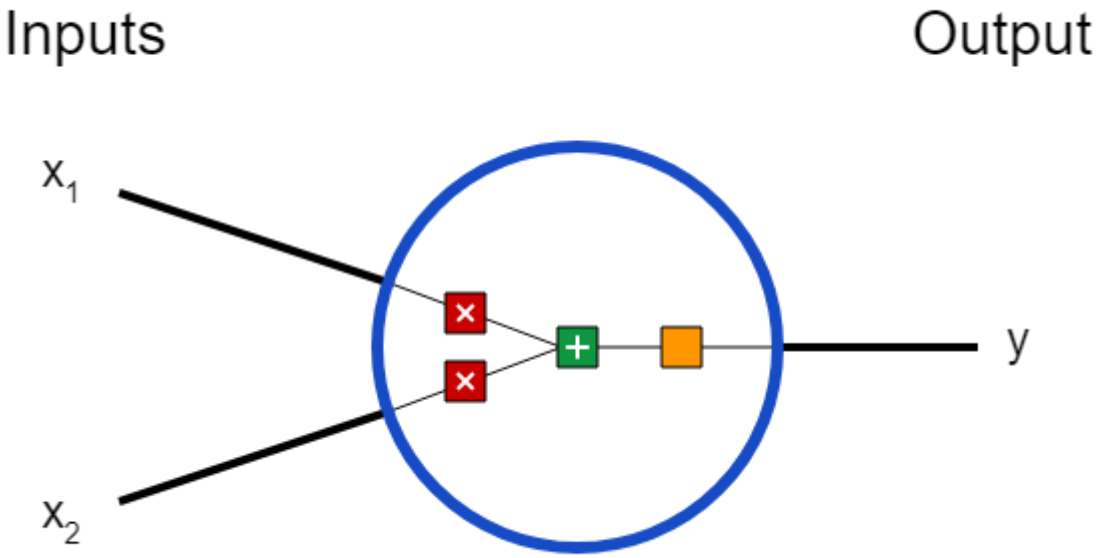
In this “project”, I will go over an **implementation code** of a neural network from scratch (Without any involvement of Keras / TensorFlow), and parse each line of code.

- **Topic:** Artificial Neural Network.
- **Purpose of the project:** Although I have already created two projects dealing with neural networks, I feel that I do not fully understand the “under the hood” of neural networks, so the process of parsing each line of implementation from scratch code **might** be helpful.

Neuron

A neural network is based on a collection of connected units called neurons. Neuron is the basic unit of a neural network.

Neuron takes at least one input, performs some calculations, and ultimately produces one output.



Neuron with two inputs. ©

1. Each input value x_n is multiplied by a weight w_n:

   $x_1 = x_1 \cdot w_1$

   $x_2 = x_2 \cdot w_2$

2. All the weighted input values are added together with a bias b:

   $(x_1 \cdot w_1) + (x_2 \cdot w_2) + b$

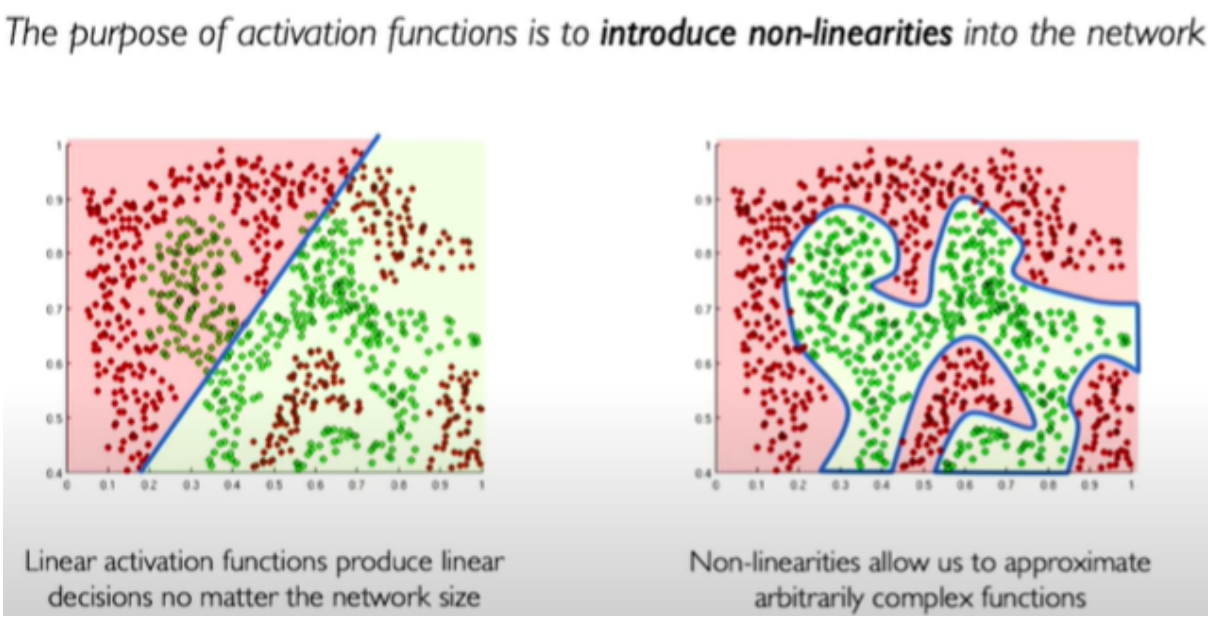
3. The sum is passed through an activation function:

   $y = g(x_1 \cdot w_1 + x_2 \cdot w_2 + b)$

Activation Function

As mentioned earlier, in artificial neural networks, each neuron forms a weighted sum of its input values and passes the resulting sum value through a function referred to as an activation function. If a neuron has n inputs which is x_1, x_2, ..., x_n then the output or activation of the neuron is $y = g(w_1 \cdot x_1 + w_2 \cdot x_2 + \ \ldots \ + w_n \cdot x_n + b)$. This function g is referred to as the activation function.

Activation functions are useful because they add non-linearities into neural networks, allowing the neural networks to learn powerful operations. If we do not apply any non-linearity in our multi-layer neural network, we are simply trying to separate the classes using a linear hyperplane. As we know, in the real world nothing is linear.



©

Commonly used activation functions are Sigmoid, ReLU, and Tanh.

For Instance: Let's assume we have a 2-input neuron that uses the sigmoid activation function, and has the following parameters:

- $w = [0, 1]$ (A way of writing $w_1 = 0, w_2 = 1$ in vector form)
- $b = 4$

We will provide the neuron with an input of $x = [2, 3]$.

As we have two vectors (w, x), we will have to use the **dot product**, what will create the following equation:

$$(w \cdot x) + b = ((w_1 \cdot x_1) + (w_2 \cdot x_2)) + b$$

Then we will set the values in the equation and get:

$$((w_1 \cdot x_1) + (w_2 \cdot x_2)) + b = 0 \cdot 2 + 1 \cdot 3 + 4 = 7$$

Finally, the sum will be passed through the sigmoid activation function, which results in the output of the neuron.

$y = g(7) = 0.999 \sim$

Implementation

```
def sigmoid(x):
    '''
    A function that will apply the sigmoid activation function.

    Parameters:
        - x (int, float): The sum of the dot product with the bias.

    Returns:
        - float: The output of the neuron.
    '''
    return 1 / (1 + np.exp(-x))
```

```
class Neuron:
    def __init__(self, weights, bias):
        self.weights = weights
        self.bias = bias

    def feed_forward(self, inputs):
        '''
        A function that will perform the process of passing inputs forward to get an output.

        Parameters:
            - inputs (int, float, list of ints / floats): Neuron's input.

        Returns:
            - float: The output of the neuron.
        '''
        result = np.dot(a = self.weights, b = inputs) + self.bias

        return sigmoid(x = result)
```

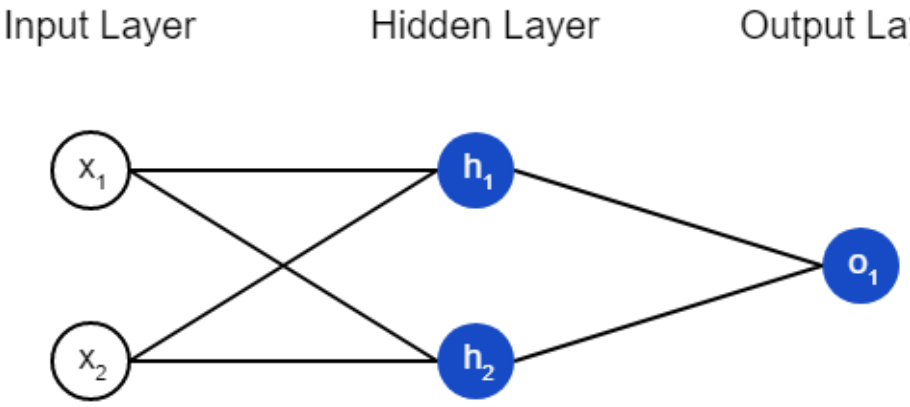
```
#w_1 = 0, w_2 = 1
weights = [0, 1]

#b = 4
bias = 4
neuron = Neuron(weights = weights, bias = bias)
```

```
#x_1 = 2, x_2 = 3
inputs = [2, 3]
neuron.feed_forward(inputs = inputs)
```

```
0.9990889488055994
```

Forming the Neural Network

As mentioned in the previous section, a neural network is based on a collection of connected neurons.



©

The neural network shown above has two input values (x_1, x_2), one hidden layer with two neurons (h_1, h_2), and an output layer with one neuron (o_1). The inputs to the output layer are the outputs of the hidden layer, which makes this a network.

- **Hidden Layer:** Any layer in between the input (First) and the output (Last) layers. There can be multiple hidden layers.

For Instance: We will use the neural network shown above, with the following parameters:

- $w = [0, 1]$ (A way of writing $w_1 = 0, w_2 = 1$ in vector form)
- $b = 0$

We will provide the neural network with an input of $x = [2, 3]$.

As we have two vectors (w, x), we will have to use the **dot product**, what will create the following equation for both h_1 and h_2:

$$h_1 = h_2 = (w \cdot x) + b = ((w_1 \cdot x_1) + (w_2 \cdot x_2)) + b$$

Then we will set the values in the equation for each hidden neuron and get:

$$h_1 = h_2 = ((w_1 \cdot x_1) + (w_2 \cdot x_2)) + b = 0 \cdot 2 + 1 \cdot 3 + 0 = 3$$

Each hidden neuron's sum will be passed through his activation function (Which is the sigmoid activation function in our case), which results in the inputs for the output layer.

$$h_1 = h_2 = g(3) = 0.9526$$

$$o_1 = ((w_1 \cdot h_1) + (w_2 \cdot h_2)) + b = 0 \cdot 0.9526 + 1 \cdot 0.9526 + 0 = 0.9526$$

Finally, the sum will be passed through the sigmoid activation function of the output neuron, which results in the output of the neural network.

$$o_1 = g(0.9526) = 0.7216 \sim$$

Implementation

```python
class Neural_Network():
    '''
    A neural network in the form of:
        - 2 input values
        - 1 hidden layer with 2 neurons
        - An output layer with 1 neuron

    For simplicity, each neuron will have the same weights and bias as follows:
        - weights = [0, 1]
        - bias = 0
    '''
    def __init__(self):
        weights = [0, 1]
        bias = 0

        self.h1 = Neuron(weights = weights, bias = bias)
        self.h2 = Neuron(weights = weights, bias = bias)
        self.o1 = Neuron(weights = weights, bias = bias)

    def feed_forward(self, inputs):
        '''
        A function that will perform the process of passing inputs forward to get an output.

        Parameters:
            - inputs (list of ints / floats): Neural network's input.

        Returns:
            - float: The output of the neural network.
        '''
        output_h1 = self.h1.feed_forward(inputs = inputs)
        output_h2 = self.h2.feed_forward(inputs = inputs)

        #The inputs to the output layer are the outputs of the hidden layer.
        output_o1 = self.o1.feed_forward(inputs = [output_h1, output_h2])

        return output_o1
```

```python
ann = Neural_Network()
```

```python
inputs = [2, 3]
ann.feed_forward(inputs = inputs)
```

```
0.7216325609518421
```

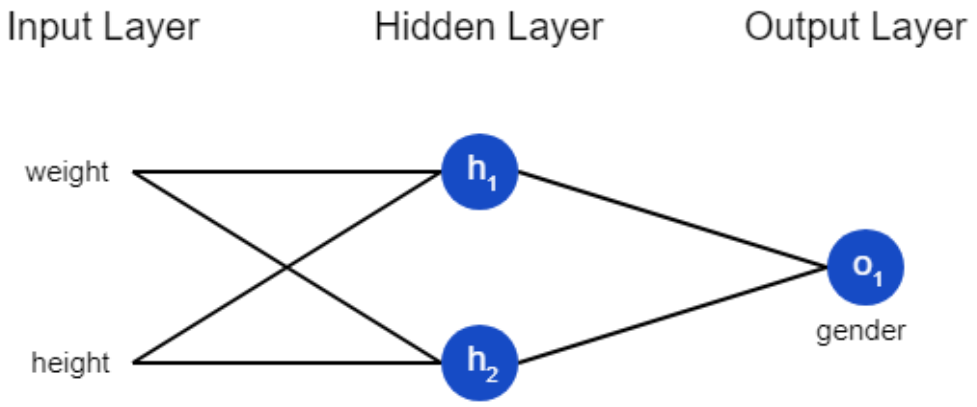
Training the Neural Network

Throughout this section, we will work with the following dataset:

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F

©

We want to train the neural network shown in the previous section, in order to predict someone's gender given their weight and height.



©

Male will be represented as 0 and Female as 1, and the data also has been shifted (Arbitrarily) for convenience.

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1

©

Cost (or loss) Function

As a preliminary step to the training of the neural network, we need a way to quantify how accurate the neural network is, so we can improve it as much as possible. To achieve the ability of quantifying, we will use a cost (or loss) function.

We will use the mean squared error (MSE) [MSE is a widely used cost (or loss) function].

Formula: $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

- n: The number of samples (4 in our case - Alice, Bob, Charlie, and Diana).
- y: The target variable being predicted (Gender in our case).
- y_i: The actual value of the target variable for sample i [The actual value for sample 0 (Which is Alice, because she is first) is 1 (Female)].
- $\hat{y}_i$: The predicted (By the neural network) value of the target variable for sample i.
- $(y_i - \hat{y}_i)^2$: The squared error.

As the name implies, MSE is simply taking the average over all squared errors.

Ultimately, the goal is to minimize as much as possible our cost (or loss) function to ensure as much as possible correctness for any given input value.

In conclusion,

$$Training\ a\ neural\ network = Trying\ to\ minimize\ the\ cost\ (or\ loss)\ function$$

For Instance: Let's assume that our neural network always outputs 0.

Name	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
Alice	1	0	1
Bob	0	0	0
Charlie	0	0	0
Diana	1	0	1

©

So our loss will be as following:

$$MSE = \frac{1}{4}\sum_{i=1}^{4}(y_i - \hat{y}_i)^2 = \frac{1}{4}(1 + 0 + 0 + 1) = 0.5$$

Implementation

```python
def mse_loss(y_true, y_pred):
    '''
    A function that will calculate the loss in the form of mean squared error (MSE).

    Parameters:
        - y_true (numpy array): The actual values of the target variable.
        - y_pred (numpy array): The predicted values of the target variable.

    Returns:
        - int / float: The loss.
    '''
    return ((y_pred - y_true) ** 2).mean()
```

```python
y_true = np.array([1, 0, 0, 1])
y_pred = np.array([0, 0, 0, 0])
mse_loss(y_true = y_true, y_pred = y_pred)
```

```
0.5
```

Now that we are familiar with the cost (or loss) function, and also with the fact that we can change the neural network's weights and biases to influence the predictions, we have to understand how to do so in a way that decreases loss.

For simplicity, we assume that we only have Alice in our dataset.

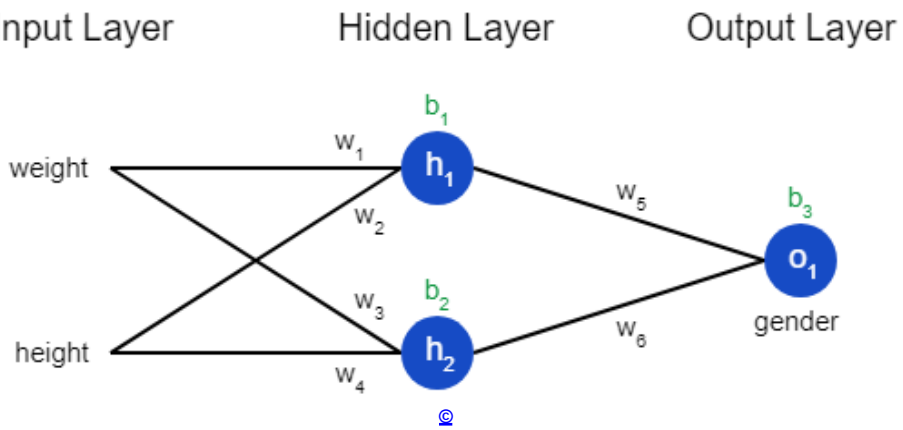
Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

©

If we compute the loss only for Alice, we will just get her squared error.

$$MSE = \frac{1}{1}\sum_{i=1}^{1}(y_i - \hat{y}_i)^2 = (y_i - \hat{y}_i)^2 = (1 - \hat{y}_i)^2$$

Another way to represent the loss, is as a function of all the weights and biases in the neural network (Because the loss is dependent upon all the weights and biases within the neural network).



©

$$L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$$

Out of those variables, let's assume we want to tweak w_1. That means, that we would want to figure out how much loss L change if we will change w_1.

To tackle this, we would have to calculate the **partial derivative** $\frac{\partial L}{\partial w_1}$.

As a preliminary step, by applying the **chain rule** we can rewrite the partial derivative $\frac{\partial L}{\partial w_1}$ in terms of $\frac{\partial \hat{y}}{\partial w_1}$ as follows:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_1}$$

Next, we can calculate the derivative of $\frac{\partial L}{\partial \hat{y}}$ because **we know** that $L = (1 - \hat{y}_i)^2$.

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial (1-\hat{y}_i)^2}{\partial \hat{y}} = -2(1 - \hat{y})$$ (We derived $(1 - \hat{y}_i)^2$ with respect to $\hat{y}_i$).

```
dL_dypred = -2 * (y_true - y_pred)
```

Now we will have to decompose the second term $\frac{\partial \hat{y}}{\partial w_1}$ similar to the first term in order to calculate the derivative.

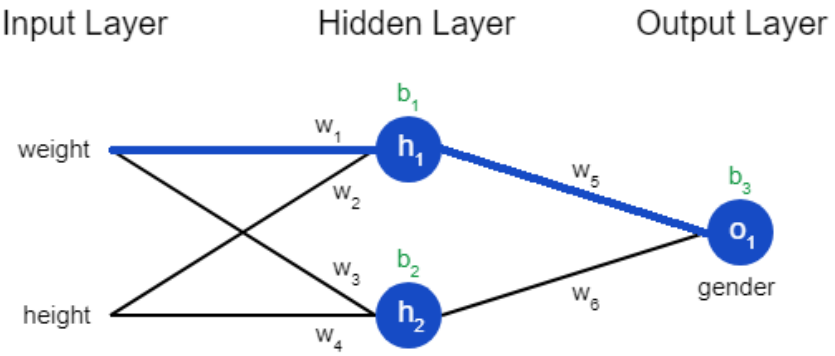
As mentioned above, the input of o_1 are the outputs of both hidden layers h_1 and h_2, so we can rewrite o_1 as a function of them $(o_1 = \hat{y})$.

$$o_1 = \hat{y} = g(h_1 \cdot w_5 + h_2 \cdot w_6 + b_3)$$ (Recall that $g()$ is the activation function of o_1).

```
# The inputs to the output layer are the outputs of the hidden layer.
sum_o1 = self.w5 * h1 + self.w6 * h2 + self.b3
o1 = sigmoid(x = sum_o1)
# o1 is assigned to y_pred, as he is the last neuron, and hence it will be the neural network's prediction.
y_pred = o1
```

Since w_1 affects only h_1, we can write $\frac{\partial \hat{y}}{\partial w_1}$ and replace $\hat{y}$ as follows:

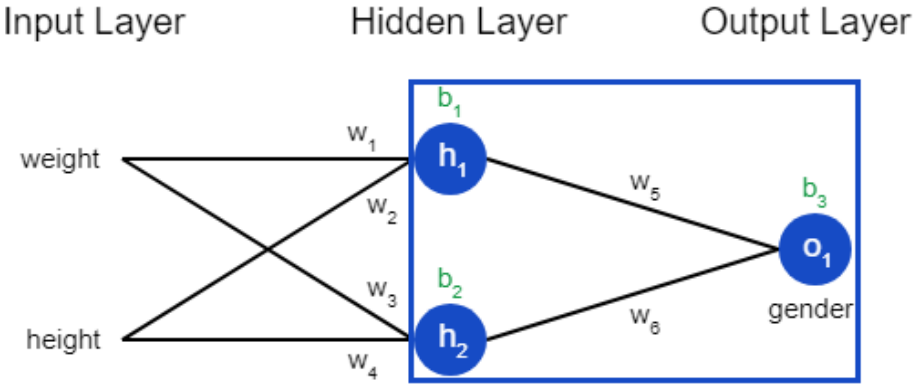
$$\frac{\partial \hat{y}}{\partial w_1} = \frac{\partial \hat{y}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$



$$\frac{\partial \hat{y}}{\partial h_1} = \frac{\partial g(h_1 \cdot w_5 + h_2 \cdot w_6 + b_3)}{\partial h_1} = w_5 \cdot g'(h_1 \cdot w_5 + h_2 \cdot w_6 + b_3)$$

```
# h1, h2 -> o1.
dypred_dw5 = h1 * sigmoid_derivative(x = sum_o1)
dypred_dw6 = h2 * sigmoid_derivative(x = sum_o1)
dypred_db3 = 1 * sigmoid_derivative(x = sum_o1)

dypred_dh1 = self.w5 * sigmoid_derivative(x = sum_o1)
dypred_dh2 = self.w6 * sigmoid_derivative(x = sum_o1)
```

In this document we are tweaking w_1, but the same process will be applied upon all the weights and biases.



On the same principle, we can rewrite h_1 as a function of the input values [Weight (x_1) and height (x_2)], weights (w_1 and w_2), and bias (b_1).

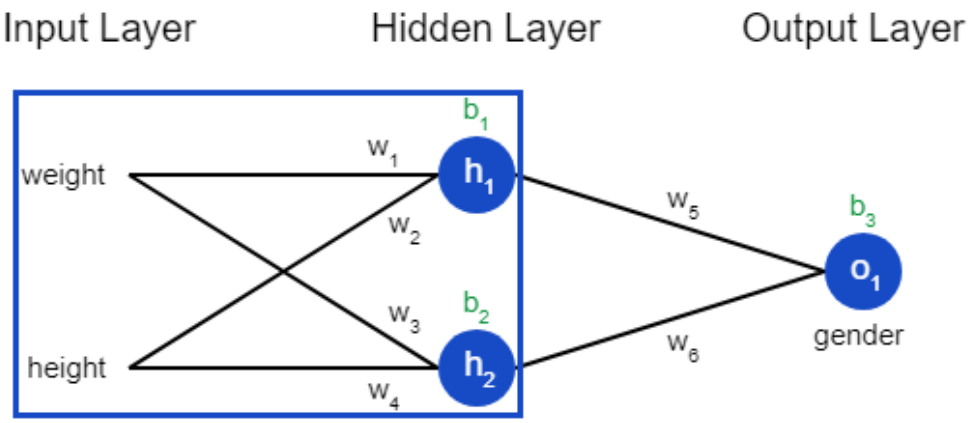
$$h_1 = g(x_1 \cdot w_1 + x_2 \cdot w_2 + b_1)$$ (Recall that $g()$ is the activation function of h_1).

We can then replace h_1, and calculate the derivative of $\frac{\partial h_1}{\partial w_1}$.

$$\frac{\partial h_1}{\partial w_1} = \frac{\partial g(x_1 \cdot w_1 + x_2 \cdot w_2 + b_1)}{\partial w_1} = x_1 \cdot g'(x_1 \cdot w_1 + x_2 \cdot w_2 + b_1)$$

```
# x1, x2 -> h1.
dh1_dw1 = inputs[0] * sigmoid_derivative(x = sum_h1)
dh1_dw2 = inputs[1] * sigmoid_derivative(x = sum_h1)
dh1_db1 = 1 * sigmoid_derivative(x = sum_h1)

# x1, x2 -> h2.
dh2_dw3 = inputs[0] * sigmoid_derivative(x = sum_h2)
dh2_dw4 = inputs[1] * sigmoid_derivative(x = sum_h2)
dh2_db2 = 1 * sigmoid_derivative(x = sum_h2)
```

In this document we are tweaking w_1, but the same process will be applied upon all the weights and biases.



We can see in both $\frac{\partial \hat{y}}{\partial h_1}$ and $\frac{\partial h_1}{\partial w_1}$ derivatives that we also have to derive the activation function $(g'())$, which in our case is the sigmoid activation function.

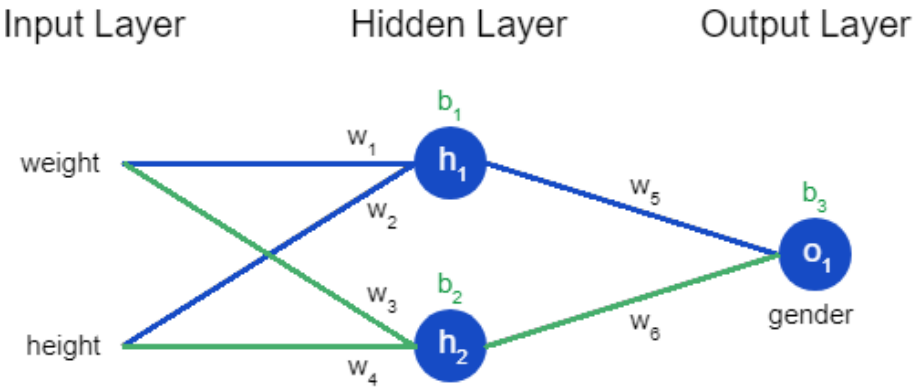
$$g(x) = \frac{1}{1+e^{-x}}$$

$$g'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = f(x) \cdot (1 - f(x))$$

```
def sigmoid_derivative(x):
    '''
    A function that will apply the derivative of the sigmoid activation function.

    Parameters:
        - x (int, float): The sum of the dot product with the bias.

    Returns:
        - float.
    '''
    fx = sigmoid(x = x)

    return fx * (1 - fx)
```

To summarize the calculation of the partial derivative of $\frac{\partial L}{\partial w_1}$:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$



The calculation of the partial derivative of $\frac{\partial L}{\partial w_1}$ is also referred to as the **gradient** of the cost (or loss) function with respect to w_1. The gradient tells us the direction that we need to go to maximize our loss. So we will take steps in the opposite direction because we want to find the lowest loss for a given set of weights.

The whole process described above in which the gradient was calculated is referred to as **backpropagation**.

For Instance: We still assume that we only have Alice in our dataset.

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

©

For simplicity, all the weights will be initialized to 1, and all the biases to 0.

We will perform a feedforward through the neural network, and get the following:

- The output of h_1:

$$h_1 = g(x_1 \cdot w_1 + x_2 \cdot w_2 + b_1) = g(-2 \cdot 1 - 1 \cdot 1 + 0) = g(-3) = 0.0474$$

- The output of h_2:

$$h_2 = g(x_1 \cdot w_3 + x_2 \cdot w_4 + b_2) = g(-2 \cdot 1 - 1 \cdot 1 + 0) = g(-3) = 0.0474$$

- The output of o_1:

$$o_1 = g(h_1 \cdot w_5 + h_2 \cdot w_6 + b_3) = g(0.0474 \cdot 1 + 0.0474 \cdot 1 + 0) = g(0.0948) = 0.524$$

The neural network outputs $o_1 = \hat{y} = 0.524$, which does not strongly favor Male (0) or Female (1).

We now want to figure out how much loss L change if we will change w_1, so we will calculate $\frac{\partial L}{\partial w_1}$.

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial (1-\hat{y}_i)^2}{\partial \hat{y}} = -2(1-\hat{y}) = -2(1-0.524) = -0.952$$

$$\frac{\partial \hat{y}}{\partial h_1} = \frac{\partial g(h_1 \cdot w_5 + h_2 \cdot w_6 + b_3)}{\partial h_1} = w_5 \cdot g'(h_1 \cdot w_5 + h_2 \cdot w_6 + b_3) = 1 \cdot g'(0.0474 \cdot 1 + 0.0474 \cdot 1 + 0) =$$
$$= 1 \cdot g(0.0948) \cdot (1 - g(0.0948)) = 0.524 \cdot 0.476 = 0.25$$

$$\frac{\partial h_1}{\partial w_1} = \frac{\partial g(x_1 \cdot w_1 + x_2 \cdot w_2 + b_1)}{\partial w_1} = x_1 \cdot g'(x_1 \cdot w_1 + x_2 \cdot w_2 + b_1) = -2 \cdot g'(-2 \cdot 1 - 1 \cdot 1 + 0) =$$
$$-2 \cdot g(-3) \cdot (1 - g(-3)) = -0.0904$$

$$\frac{\partial L}{\partial w_1} = -0.952 \cdot 0.25 \cdot (-0.0904) = 0.0215$$

So 0.0215 indicates that if we increase w_1, the loss will increase (Because the result is positive) by this amount [If the partial derivative was negative, it means that if we increase w_1, the loss will increase by this amount (Which is negative)].

Gradient Descent

The gradient descent is an optimization algorithm that helps us to determine how (Increase or decrease weight's value) every single weight in our neural network needs to be changed in order to decrease our loss on the next training iteration. It is basically the following update equation:

Formula: $w_1 = w_1 - \eta \cdot \frac{\partial L}{\partial w_1}$

η is a constant called the **learning rate**, and it determines how much step we take in our gradient. In other words, the learning rate controls the size of the step we will take from the previous point. Common values for a learning rate are usually in the range of 0.01 to 0.1.

As mentioned above, the calculated gradient tells us the direction in which the loss will be maximized, therefore we will have to take steps in the opposite direction. In the update equation discussed above, the minus (-) will change the direction of the calculated gradient in order to achieve the opposite direction.

- If $\frac{\partial L}{\partial w_1}$ is positive, it means that if we will increase w_1, the loss will be increased by $\frac{\partial L}{\partial w_1}$. Therefore, we have to decrease w_1, and we achieve this with the help of the minus (-) in the gradient descent's formula.
- If $\frac{\partial L}{\partial w_1}$ is negative, it means that if we will increase w_1, the loss will be increased by $\frac{\partial L}{\partial w_1}$ (As $\frac{\partial L}{\partial w_1}$ is now negative, the loss will be increased by a negative number, or in other words, the loss will decrease by $\frac{\partial L}{\partial w_1}$). Therefore, we have to increase w_1, and we achieve this with the help of the minus (-) in the gradient descent's formula.

```
# Weights and biases updation.
# h1 related - The parameters which affect only h1.
self.w1 -= learning_rate * dL_dypred * dypred_dh1 * dh1_dw1
self.w2 -= learning_rate * dL_dypred * dypred_dh1 * dh1_dw2
self.b1 -= learning_rate * dL_dypred * dypred_dh1 * dh1_db1

# h2 related - The parameters which affect only h2.
self.w3 -= learning_rate * dL_dypred * dypred_dh2 * dh2_dw3
self.w4 -= learning_rate * dL_dypred * dypred_dh2 * dh2_dw4
self.b2 -= learning_rate * dL_dypred * dypred_dh2 * dh2_db2

# o1 related - The parameters which affect only o1.
self.w5 -= learning_rate * dL_dypred * dypred_dw5
self.w6 -= learning_rate * dL_dypred * dypred_dw6
self.b3 -= learning_rate * dL_dypred * dypred_db3
```

In this document we are tweaking w_1, but the same process will be applied upon all the weights and biases.

As the gradient is very computationally expensive to compute, because it is computed as summation over our entire data set (The neural network will have to calculate the gradients of all the n inputs), it is often common to use the **stochastic gradient descent (SGD)**. That is, a gradient that will be computed on a single random input from the data set.