

Perceptron

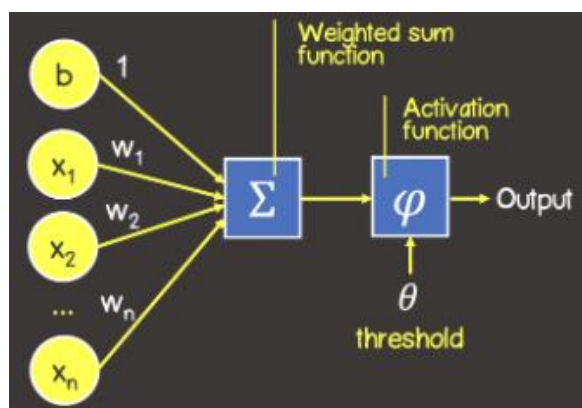
A mathematical model of a neural network consisting of a single neuron that performs two consecutive operations:

- calculates the sum of the input signals taking into account their weights (conductivity or resistance) of the connection

$$sum = \vec{X}^T \vec{W} + \vec{B} = \sum_{i=1}^n x_i w_i + b$$

- applies the activation function to the total amount of input signal exposure.

$$out = \varphi(sum)$$



Input Vector: $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$

- Output Vector: For binary classification: $y \in \{0, 1\}$

Where y is the true label.

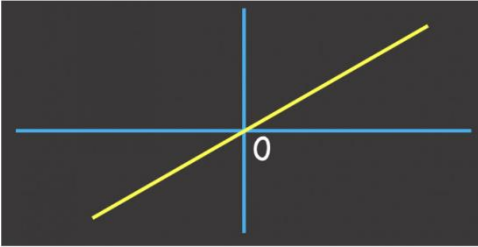
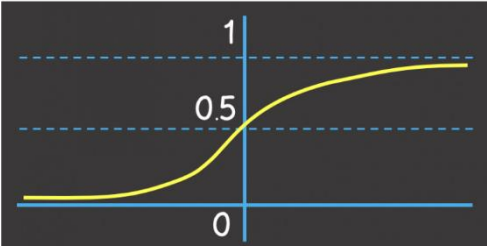
We assign a weight to each input.

$f_{W,b}(X) = h(XW + b)$, where X is the input data, W is the weight for each input neuron, b is the bias term and h is a step function.

The activation function used in a Perceptron is typically the step function (or Heaviside step function):

$$h(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ -1, & \text{if } z < 0 \end{cases}$$

Any differentiable function can also be used as an activation function. For example,

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

The loss function for a Perceptron can be defined as the perceptron loss:

$$L(\hat{y}, y) = \begin{cases} 1, & \text{if } y^y > 0 \\ -y^y, & \text{if } y^y \leq 0 \end{cases}$$

For a single-layer perceptron, the perceptron criterion is rarely used as a loss function in practice. A classification error is usually used for the perceptron.

The perceptron uses a weighted linear combination of input data to return a forecast estimate. If the forecast score exceeds the selected threshold, the perceptron predicts True, otherwise False.

Gradient descent is used to minimize the loss function by updating the weights and shifting in the direction of the negative gradient of the loss function. The gradient descent method mathematically looks like this: $\vec{w}^{(k+1)} = \vec{w}^k - \mu \nabla L(\vec{w}^k)$, where k is the k-th iteration of neural network training, μ is the learning rate, ∇L is the gradient of the error function. To find the gradient,

$$\nabla L(\vec{w}) = \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \vdots \\ \frac{\partial L}{\partial w_N} \end{bmatrix}$$

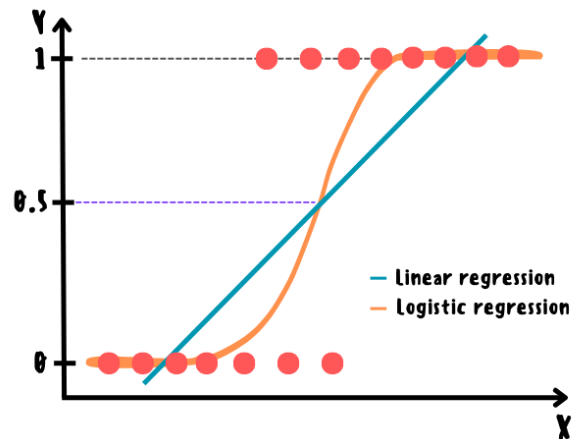
we use partial derivatives of the configurable arguments w_1, w_2 :

The weights and offsets are calculated using the formulas:

$$w_{i,j}(t+1) = w_{i,j}(t) - \alpha \frac{\partial E}{\partial w_{i,j}}$$

$$b_i(t+1) = b_i(t) - \alpha \frac{\partial E}{\partial b_i}, \text{ where } \alpha \text{ is the learning rate.}$$

Logistic Regression

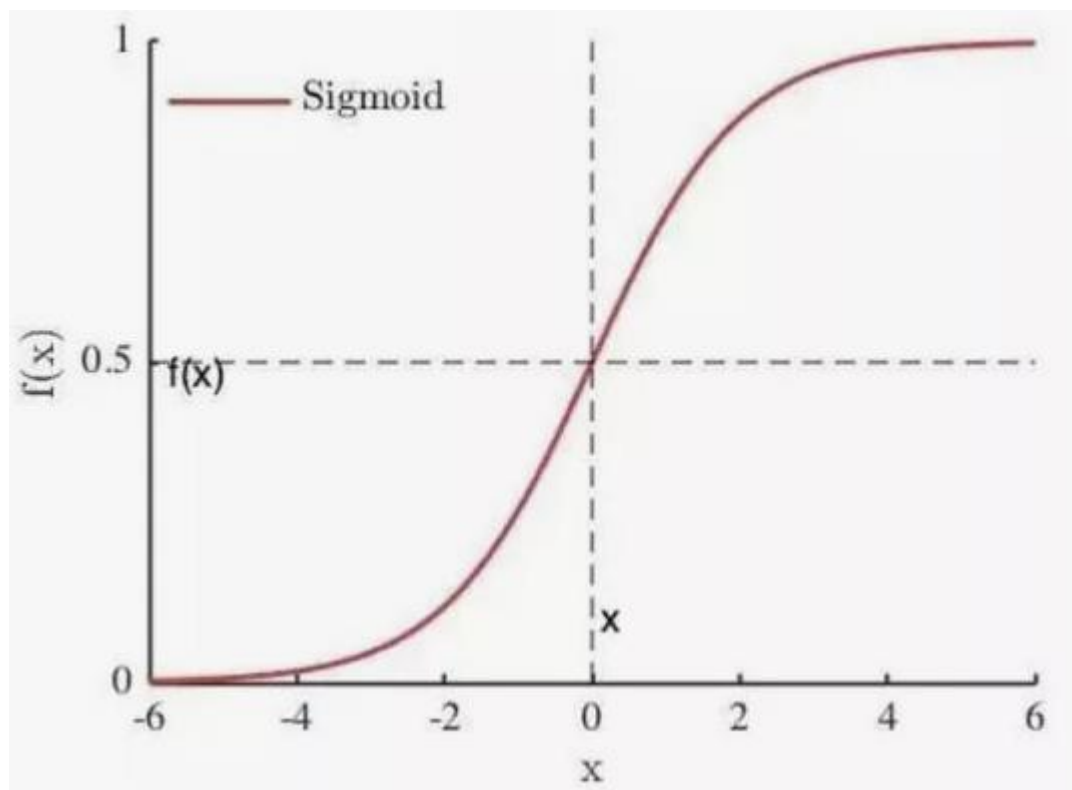


Input Vector: $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$

Output Vector: Produces the final prediction y_{hat} (or \hat{y}), which is the probability of the positive class: $y_{\text{hat}} \in [0, 1]$.

Output (prediction) formula: Linear combination: $\hat{y} = \sigma(w_1x_1 + w_2x_2 + b)$, where: $W = [w_1, w_2, \dots, w_n]^T$ is the weight vector, b is the bias term.

The activation function used in logistic regression is the sigmoid function: $\sigma(x) = \frac{1}{1+e^{-x}}$



The sigmoid shows the confidence level of the model and tends to unity, which allows you to estimate the probability.

The initial form of logistical losses:

$$L(\sigma(z), y_i) = \begin{cases} -\ln \sigma(z) & \text{if } y_i = 1 \\ -\ln(1 - \sigma(z)) & \text{if } y_i = 0 \end{cases}$$

Error function: Cross entropy Loss for the entire training set:

$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i=1}^N [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

In logistic regression, the calculation of forecasts (\hat{y}) takes place in two stages: first, using weights and bias, a linear forecast is calculated, which is then transformed using a sigmoidal function. The probabilities obtained are then interpreted as classes, where the value is rounded to 0 if activation ≤ 0.5 , and to 1 if activation > 0.5 , which allows you to determine the final result.

As with the perceptron, gradient descent is used to minimize the loss function.

Gradients of the loss function for displacement and weights, respectively

$$\frac{\partial J(w, b)}{\partial b} = \left(\frac{1}{n} \sum_{i=1}^n L_{\log}(\sigma(z), y_i) \right)' = \frac{1}{n} \sum_{i=1}^n \frac{\partial L_{\log}(\sigma(z), y_i)}{\partial b} = \frac{1}{n} \sum_{i=1}^n (\sigma(z) - y_i)$$

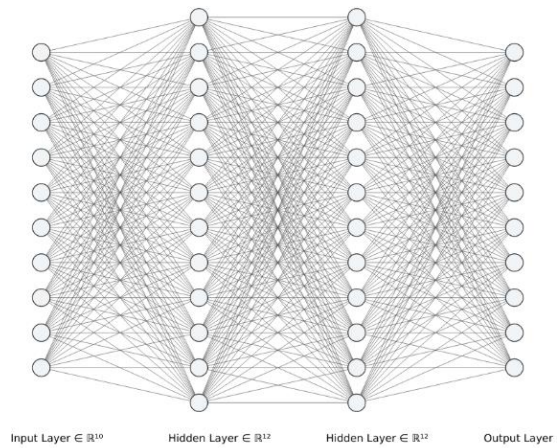
$$\frac{\partial J(w, b)}{\partial w} = \left(\frac{1}{n} \sum_{i=1}^n L_{\log}(\sigma(z), y_i) \right)' = \frac{1}{n} \sum_{i=1}^n \frac{\partial L_{\log}(\sigma(z), y_i)}{\partial w} = \frac{1}{n} \sum_{i=1}^n (\sigma(z) - y_i) x_i$$

The function that updates the weights

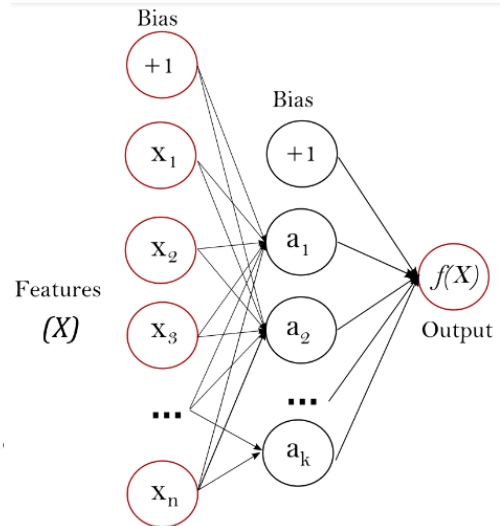
$$w_i \leftarrow w_i + \eta(y - \hat{y})x_i$$

$$b \leftarrow b + \eta(y - \hat{y})$$

Multilayer Perceptron



Multi-layer perceptron



Input Vector: $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$

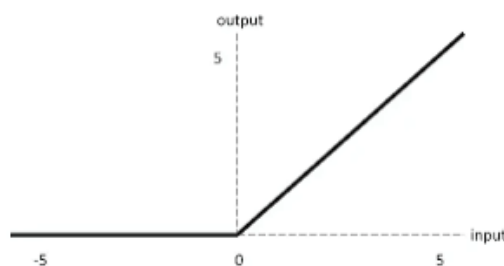
Output Vector: For a single output or multi-class classification: $\hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \dots, \hat{y}_m]^T$, where m is the number of output classes.

The leftmost layer is the input layer, which consists of a set of neurons $\{x_i | x_1, x_2, \dots, x_m\}$ representing the input characteristics. Each neuron in the hidden layer transforms the values of the previous layer using weighted linear summation $x_1 w_1 + x_2 w_2 + \dots + x_m w_m$, followed by a nonlinear activation function $g(\cdot): \mathbb{R} \rightarrow \mathbb{R}$. The output layer gets the values from the last hidden layer and converts them to output values.

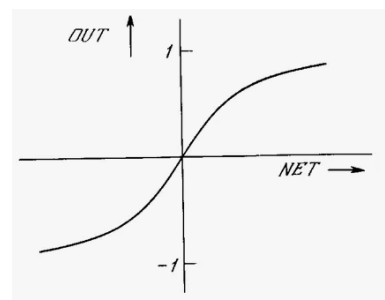
Linear Combination: $z^i = W^i a^{i-1} + b^i$, where W^i is the weight matrix for layer i , a^{i-1} is the activation vector from the previous layer, b^i is the bias vector for layer i .

Common activation functions are the sigmoid shown in logistic regression, the hyperbolic tangent in the range $(-1;1)$: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, ReLU (Rectified Linear Unit): $f(z) = \max(0, z)$,

Softmax (for multi-class classification): $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}}$



ReLU (Rectified Linear Unit)



Softmax

The most common loss function in classification problems is binary cross-entropy:

$$\mathcal{L}(y; w) := -y_i * \log(p(x_i)) - (1 - y_i) * \log(1 - p(x_i)) \text{ or } Loss = -\sum_{i=1}^{output\ size} y_i \cdot \log \hat{y}_i$$

Using gradient descent, the change in each weight w_{ij} is equal $\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial v_j(n)} y_i(n)$ or

$w \leftarrow w - \eta (\alpha \frac{\partial R(w)}{\partial w} + \frac{\partial Loss}{\partial w})$, where $y_i(n)$ – s the output of the previous neuron i, and η is the learning rate, which is chosen so that the weights converge quickly to the answer without hesitation

$\frac{\partial \mathcal{E}(n)}{\partial v_j(n)}$ denotes the partial derivative of the error $E(n)$ by the weighted sum $v_j(n)$ of the input connections of the neuron i. The derivative to be calculated depends on the induced local field v_j ,

which itself changes. This derivative can be simplified to $-\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} = e_j(n) \phi'(v_j(n))$, where ϕ' – is the derivative of the activation function, which itself does not change. To change the weights of the hidden layer, the weights of the output layer are changed according to the derivative of the activation function, and this algorithm is a reverse propagation of the activation function.