The Perceptron: the "not so artificial", artificial neuron

Berke Aslan

December 2024

Disclaimer: these are learning notes meant as a **work in progress**. As I learn more about the field of AI and all its intricacies, this and my other docs will be expanded on. This is by no means meant to be an exhaustive overview.

What is a perceptron?

The perceptron (or single-layer perceptron) is a supervised learning algorithm designed for binary classification and is one of the simplest forms of artificial neural networks. By some it is considered to be the first mathematical model that mimics the neuron's in the human brain (albeit not fully).

How it works

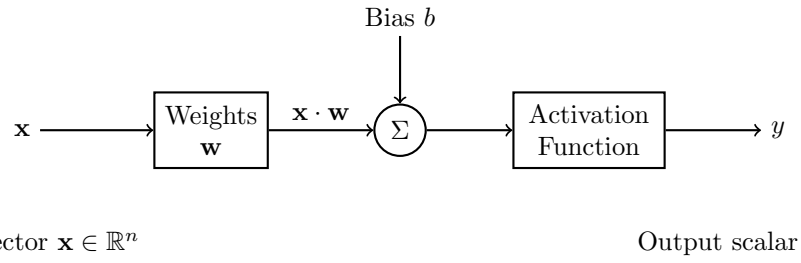

Figure 1: A schematic overview of the single-layer perceptron.

So how does a perceptron work? It's actually quite straightforward: a perceptron works by passing the sum of an input vector $\mathbf{x} \in \mathbb{R}^n$ (multiplied by weights $\mathbf{w}$) and a bias term b trough an activation function. This activation function is often the Heaviside step function $H : \mathbb{R} \to \{0, 1\}$ defined as:

$$H(x) = \begin{cases} 1, & \text{if } x \geq 0, \\ 0, & \text{if } x < 0. \end{cases}$$

In other words, our model is given by $f : \mathbb{R} \to \{0, 1\}$ with f defined as:

$$f(x) = H(\mathbf{w^T x} + b)$$

Alternatively, we could add the bias b as an element of vector $\mathbf{w}$. This can be done by setting the first element of $\mathbf{x}$ as 1. Then the weight corresponding to this element would be the bias b. More formally:

$$\mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} b \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix}$$

Now, you may wonder how the weights $\mathbf{w}$ and the bias term b are determined. Recall that this is a supervised learning algorithm, so we have labeled training data. In other words, we already now the output scalars y for all input vectors $\mathbf{x}$ in our training data set. The model learns by applying the "perceptron learning rule". Before we continue with an explanation, let us define the following (adapted from `https://en.wikipedia.org/wiki/Perceptron`):

- The learning rate $\alpha > 0$
- The output of the perceptron $\hat{y}_i = f(\mathbf{x}_i)$
- The training set $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ..., (\mathbf{x}_k, y_k)\}$ with $\mathbf{x}_i \in \mathbb{R}^n$ and $y_i \in \mathbb{R}$ for $\forall i \in \{1, ..., k\}$

Then the steps of the learning algorithm can be given by:

1. Initialize the weights: either to 0 or another small random value.
2. For each sample $j \in D$ perform the following steps:
   (a) Compute the output $\hat{y}_j$ of the perceptron for the current sample j: $\hat{y}_j(t) = f(\mathbf{w(t)} \cdot \mathbf{x}) = f(w_0(t)x_{j,0} + w_1(t)x_{j,1} + ... + w_n(t)x_{j,n})$ with $x_{j,0} = 1$ and w_0 is the bias term.
   (b) Update the weights using the following rule: $w_i(t+1) = w_i(t) + \alpha(y_j - \hat{y}_j(t))x_{j,i}$.
3. Repeat the previous step until either a predetermined number of epochs is ran or until the termination condition (iteration error) is met: $\frac{1}{k}\sum_{i=1}^{k} |y_j - \hat{y}_j(t)|$.

Linear separability

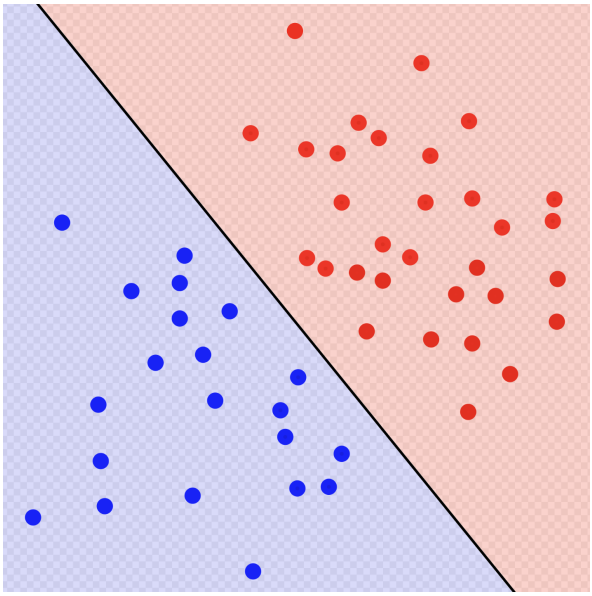

Figure 2: Graphic showing the linear separability of a feature space with a hyperplane (in this case a line) dividing the space in the separate classes (blue and red).

Note, that, the data has to be linearly separable (`https://en.wikipedia.org/wiki/Linear_separability`) in order to utilize the perceptron. This means that there should be a hyperplane that divides the feature space such that points above and below the hyperplane belong to a certain class (that we try to predict). An example is shown in Figure 2.

Implementation in Python

In this section we will implement the perceptron in Python. Note, that, many different implementations are possible. We have not included any data to play with, as this learning doc is more meant as a high-level explanation to understand the algorithm itself and its functioning.

```
1  import numpy as np
2
3  class Perceptron:
4      # Initialize the initial parameters:
5      def __init__(self, alpha = 0.1, epochs = 100, gamma = 0.1,
       ↪  weights = None) -> None:
```

```
        self.alpha = alpha
        self.epochs = epochs
        self.gamma = gamma
        self.weights = weights

    # The Heaviside step function that we use as the activation function:
    def activation(self, x) -> int:
        return 1 if x >= 0 else 0

    def predict(self, X) -> int:
        # We add the 1 for the bias term as a constant to the X vector:
        X = np.insert(X, 0, 1)
        weighted_sum = np.dot(self.weights, X)
        return self.activation(weighted_sum)

    def fit(self, X, y) -> None:
        self.weights = np.zeros(X.shape[1] + 1)  # + 1 for the bias term

        # We run the training algorithm for a set number of epochs
        for epoch in range(self.epochs):
            total_error = 0
            for i in range(len(X)):
                x_i = np.insert(X[i], 0, 1)  # Add bias term
                weighted_sum = np.dot(self.weights, x_i)
                prediction = self.activation(weighted_sum)
                error = y[i] - prediction
                total_error += abs(error)
                self.weights += self.alpha * error * x_i  # Weight update function

            # We check if the termination condition is reached before the epochs are done:
            iteration_error = total_error / len(X)
            if iteration_error < self.gamma:
                print(f"Stopping early at epoch {epoch + 1}: iteration error {iteration_error} < gamma {self.gamma}")
                break
```