Perceptron and Boolean Functions AND, OR, NAND and XOR.

The Perceptron is a linear machine learning algorithm for binary classification tasks.

It is the simplest type of neural network model.

(Perceptron — это линейный алгоритм машинного обучения для задач бинарной классификации.

Это самый простой тип модели нейронной сети.)

It consists of a single node or neuron that takes a row of data as input and predicts a class label. This is achieved by calculating the weighted sum of the inputs and a bias (set to 1). The weighted sum of the input of the model is called the activation.

- **Activation** = Weights * Inputs + Bias

If the activation is above 0.0, the model will output 1.0; otherwise, it will output 0.0.

- **Predict 1**: If Activation > 0.0
- **Predict 0**: If Activation <= 0.0

(Он состоит из одного узла или нейрона, который принимает строку данных в качестве входных данных и предсказывает метку класса. Это достигается путем вычисления взвешенной суммы входных данных и смещения (установленного на 1). Взвешенная сумма входных данных модели называется активацией.

- **Активация**

  = веса * входы + смещение

Если активация выше 0,0, модель выдаст 1,0; в противном случае будет выведено 0.0.

- **Прогноз 1**

  : если активация> 0,0
- **Прогноз 0**

  : если активация <= 0,0)

The Perceptron is a linear classification algorithm. This means that it learns a decision boundary that separates two classes using a line (called a hyperplane) in the feature space. As such, it is appropriate for those problems where the classes can be separated well by a line or linear model, referred to as linearly separable.

(Персептрон — это алгоритм линейной классификации. Это означает, что он изучает границу решения, которая разделяет два класса с помощью линии (называемой гиперплоскостью) в пространстве признаков. Таким образом, он подходит для тех задач, где классы могут быть хорошо разделены линейной или линейной моделью, называемой линейно разделимой.)

AND, OR, NAND and XOR.

> **Logic gates are the most basic materials to implement digital components. The use of logic gates ranges from computer architecture to the field of electronics.**
>
> **These gates deal with binary values, either 0 or 1. Different types of gates take different numbers of input, but all of them provide a single output. These logic gates when combined form complicated circuits.**

(Логические вентили — это самые основные материалы для реализации цифровых компонентов. Использование логических вентилей варьируется от компьютерной архитектуры до области электроники.

Эти вентили имеют дело с двоичными значениями, либо 0, либо 1. Различные типы вентилей принимают разное количество входных данных, но все они обеспечивают один выход. Эти логические элементы в сочетании образуют сложные схемы.)

Task:

Create and train a Single-layer Perceptron and MLP (Multi-layered perceptron) , implement an algorithm

for classifying logical functions - AND, OR, NAND and XOR.

(Задача: Создать и обучить Однослойный Персептрон и MLP (Multi-layered perceptron) , реализовать алгоритм классификации логических функций - AND, OR, NAND и XOR.)

```
# Importing Python libraries
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

from itertools import cycle
import matplotlib.pyplot as plt
import seaborn as sns
```

```
# Configuring plotting params
sns.set_style('darkgrid')
sns.set_context('poster')
plt.rcParams["figure.figsize"] = [20, 10]
```

Data (inputs)

This data is the same for each kind of logic gate, since they all take in two boolean variables as input.

(Эти данные одинаковы для каждого типа логических элементов, поскольку все они принимают на вход две булевы переменные.)

```
data = np.array([
                [0, 0],
                [0, 1],
                [1, 0],
                [1, 1]])
print(data)
print(data.shape)
```

```
[[0 0]
 [0 1]
```

```
 [1 0]
 [1 1]]
(4, 2)
```

Outputs

```
target_and = np.array(
    [
        [0],
        [0],
        [0],
        [1]])

target_or = np.array(
    [
        [0],
        [1],
        [1],
        [1]])

target_nand = np.array(
    [
        [1],
        [1],
        [1],
        [0]])

target_xor = np.array(
    [
        [0],
        [1],
        [1],
        [0]])
```

The Perceptron Class

Structure and Properties

Input Nodes

These nodes contain the input to the network. (Эти узлы содержат вход в сеть)

Weights and Biases

These parameters are what we update when we talk about “training” a model. They are initialized to some random value or set to 0 and updated as the training progresses. The bias is analogous to a weight independent of any input node. Basically, it makes the model more flexible, since you can “move” the activation function around.

(Именно эти параметры мы обновляем, когда говорим об «обучении» модели. Они инициализируются некоторым случайным значением или устанавливаются в 0 и обновляются по мере прохождения обучения. Смещение аналогично весу, независимому от любого входного узла. По сути, это делает модель более гибкой, поскольку вы можете «перемещать» функцию активации.)

Activation Function

Though there are many kinds of activation functions, we’ll be using a simple linear activation function for our perceptron. The linear activation function has no effect on its input and outputs it as is.

(Хотя существует много видов функций активации, мы будем использовать для нашего персептрона простую линейную функцию активации. Линейная функция активации не влияет на входные данные и выводит их как есть.)

Classification

We'll be modelling this as a classification problem, so Class 1 would represent an value of 1, while Class 0 would represent a value of 0.

(Мы будем моделировать это как проблему классификации, поэтому класс 1 будет представлять значение , равное 1, а класс 0 будет представлять значение 0.)

Training algorithm

Here, we cycle through the data indefinitely, keeping track of how many consecutive datapoints we correctly classified. If we manage to classify everything in one stretch, we terminate our algorithm. If not, we reset our counter, update our weights and continue the algorithm.

(Здесь мы бесконечно циклически перебираем данные, отслеживая, сколько последовательных точек данных мы правильно классифицировали. Если нам удастся классифицировать все за один раз, мы завершаем наш алгоритм. Если нет, мы сбрасываем наш счетчик, обновляем наши веса и продолжаем алгоритм.)

```
class Perceptron:
    """
    Create a perceptron.

    data: A 4x2 matrix with the input data. (data: Матрица 4x2 с входными данными)

    target: A 4x1 matrix with the perceptron's expected outputs  (target: матрица 4x1 с выходными данными персептрона))

    lr: the learning rate. Defaults to 0.01  (скорость обучения. По умолчанию 0,01)

    input_nodes: the number of nodes in the input layer of the perceptron.
    (input_nodes: количество узлов во входном слое персептрона)
        Should be equal to the second dimension of train_data.
        (Должно быть равно второму измерению данных.))

    """


    def __init__(self, data, target, lr=0.01, input_nodes=2):


        self.data = data
        self.target = target
        self.lr = lr
        self.input_nodes = input_nodes

        # randomly initialize the weights and set the bias to -1.
        # (случайным образом инициализируйте веса и установите смещение на -1.)

        self.w = np.random.uniform(size=self.input_nodes)
        self.b = -1

        # node_val hold the values of each node at a given point of time.
        # (node_val содержит значения каждого узла в данный момент времени.)

        self.node_val = np.zeros(self.input_nodes)

        # tracks how the number of consecutively correct classifications changes in each iteration
        # (отслеживает, как количество последовательно правильных классификаций )
        # changes in each iteration (изменения в каждой итерации)

        self.correct_iter = [0]


    def _gradient(self, node, exp, output):
        """
        Return the gradient for a weight. (Вернуть градиент для веса)
        This is the value of delta-w. (Это значение delta-w)
        """
        return node * (exp - output)
```

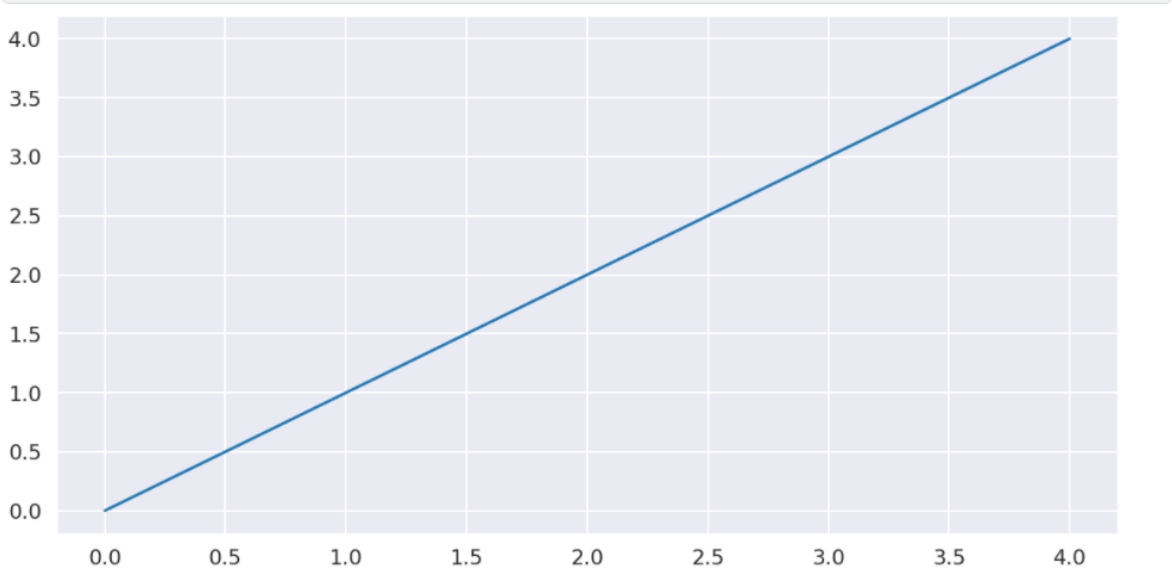
```
def update_weights(self, exp, output):
    """
    Update weights and bias based on their respective gradients
    (Обновите веса и смещения на основе их соответствующих градиентов)
    """
    for i in range(self.input_nodes):
        self.w[i] += self.lr * self._gradient(self.node_val[i], exp, output)

    # the value of the bias node can be considered as being 1 and the weight between this node
    # (значение узла смещения можно считать равным 1, а вес между этим узлом)
    # and the output node being self.b  (и выходной узел self.b)
    self.b += self.lr * self._gradient(1, exp, output)



def forward(self, datapoint):
    """
    One forward pass through the perceptron. (Один прямой проход через персептрон)
    Implementation of "wX + b". (Реализация "wX+b")
    """
    return self.b + np.dot(self.w, datapoint)


def classify(self, datapoint):
    """
    Return the class to which a datapoint belongs based on
    (Возвращает класс, к которому принадлежит точка данных, на основе)
    the perceptron's output for that point. (выход персептрона для этой точки.)
    """
    if self.forward(datapoint) >= 0:
        return 1

    return 0


def plot(self, h=0.01):
    """
    Generate plot of input data and decision boundary.
    (Сгенерируйте график входных данных и границы решения.)
    """
    # setting plot properties like size, theme and axis limits
    # (установка свойств графика, таких как размер, тема и ограничения по осям)

    sns.set_style('darkgrid')
    plt.figure(figsize=(10, 10))

    plt.axis('scaled')
    plt.xlim(-0.1, 1.1)
    plt.ylim(-0.1, 1.1)

    colors = {
        0: "ro",
        1: "go"
    }

    for i in range(len(self.data)):
        plt.plot([self.data[i][0]],
                 [self.data[i][1]],
                 colors[self.target[i][0]],
                 markersize=20)

    x_range = np.arange(-0.1, 1.1, h)
    y_range = np.arange(-0.1, 1.1, h)

    # creating a mesh to plot decision boundary (создание сетки для построения границы решения)

    xx, yy = np.meshgrid(x_range, y_range, indexing='ij')
    Z = np.array([[self.classify([x, y]) for x in x_range] for y in y_range])

    # using the contourf function to create the plot  (используя функцию контура для создания графика)

    plt.contourf(xx, yy, Z, colors=['red', 'green', 'green', 'blue'], alpha=0.4)


def train(self):
    """
    Train a single layer perceptron. ( Обучить однослойный персептрон)
    """
    # the number of consecutive correct classifications
    # (количество последовательных правильных классификаций)
    correct_counter = 0
```

```
        iterations = 0

        for train, target in cycle(zip(self.data, self.target)):
            # end if all points are correctly classified
            if correct_counter == len(self.data):
                break

            # a single layer perceptron can't model the xor function
            # (однослойный персептрон не может моделировать функцию хог)
            # so it'll never converge!  (так что никогда не сойдется!)

            if iterations > 1000:
                print("1000 iterations exceded without convergence! A single layered perceptron can't handle the XOR problem.")
                break

            output = self.classify(train)
            self.node_val = train
            iterations += 1

            # if correctly classified, increment correct_counter
            if output == target:
                correct_counter += 1
            else:
                # if incorrectly classified, update weights and reset correct_counter
                # (если неправильно классифицирован, обновить веса и сбросить правильный_счетчик)

                self.update_weights(target, output)
                correct_counter = 0

            self.correct_iter.append(correct_counter)
```

Analyzing the Results (AND)

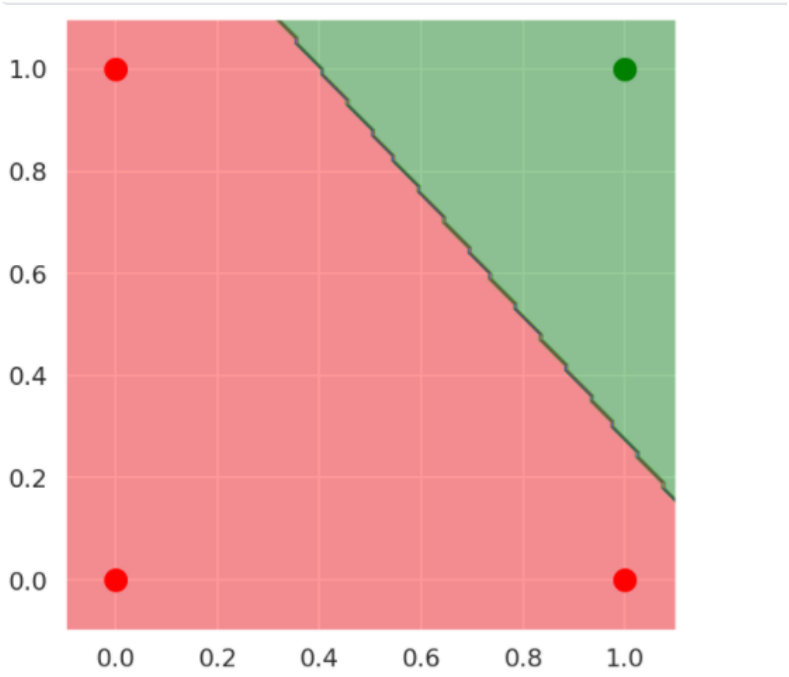
```
p_and = Perceptron(data, target_and)
p_and.train()
```

```
_ = plt.plot(p_and.correct_iter[:100])
```



AND - (0, 0, 0, 1)

```
p_and.plot()
```




```
def AND(x1, x2):

    x = [x1, x2]
    p_and = Perceptron(data, target_and)
    p_and.train()

    return p_and.classify(x)

# Test AND
AND(1, 1)
```

```
1
```

```
# Test AND
AND(0, 1)
```

```
0
```

The single-layer perceptron correctly classified the entire data set for the logical function: AND

Since the function is linearly separable

(Однослойный персептрон правильно классифицировал весь набор данных для логической функции: AND.Т. к. эта функция линейно разделима.)

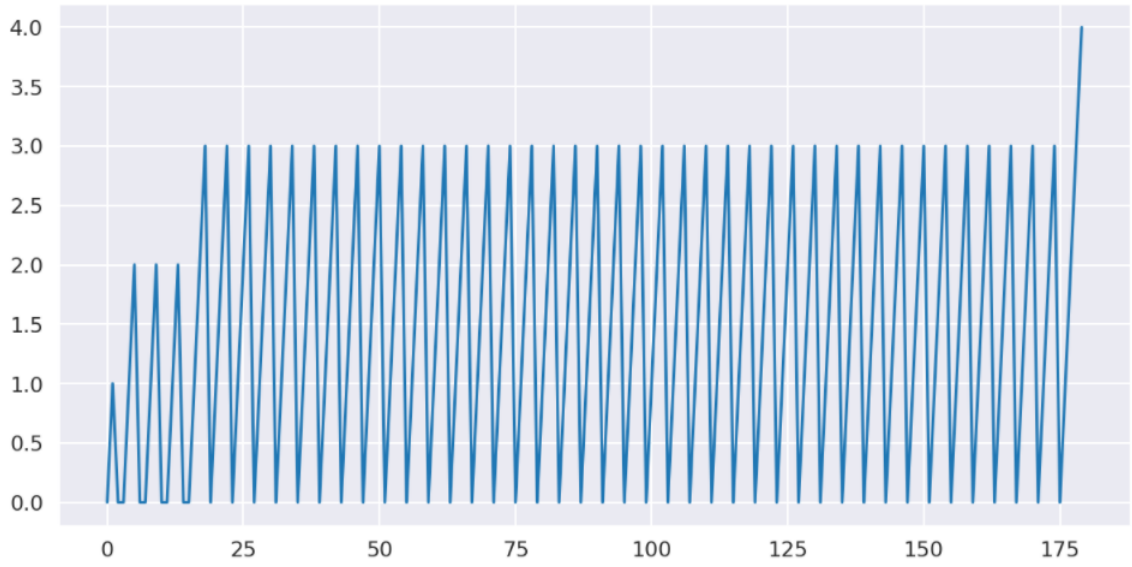
Analyzing the Results (OR)

```
p_or = Perceptron(data, target_or)
p_or.train()
```

This converges, since the data for the OR function is linearly separable. If we plot the number of correctly classified consecutive datapoints, we get the below plot. It's hits the value 4, meaning that it classified the entire dataset correctly.

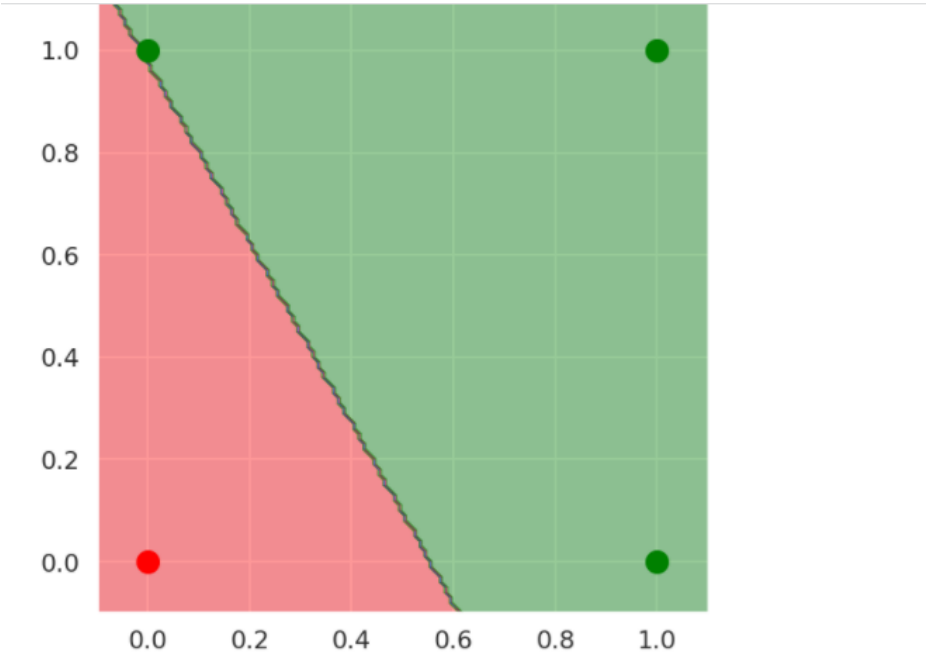
(Это сходится, поскольку данные для функции ИЛИ линейно разделимы. Если мы нанесем на график количество правильно классифицированных последовательных точек данных, мы получим приведенный ниже график. Понятно, что он достигает значения 4, что означает, что он правильно классифицировал весь набор данных.)

```
_ = plt.plot(p_or.correct_iter[:200])
```



OR -(0, 1, 1, 1)

```
p_or.plot()
```




```
def OR(x1, x2):

    x = [x1, x2]
    p_or = Perceptron(data, target_or)
    p_or.train()

    return p_or.classify(x)

# Test OR
OR(1, 1)
```

```
1
```

```
# Test OR
OR(1, 0)
```

```
1
```

The single-layer perceptron correctly classified the entire data set for the logical function: OR

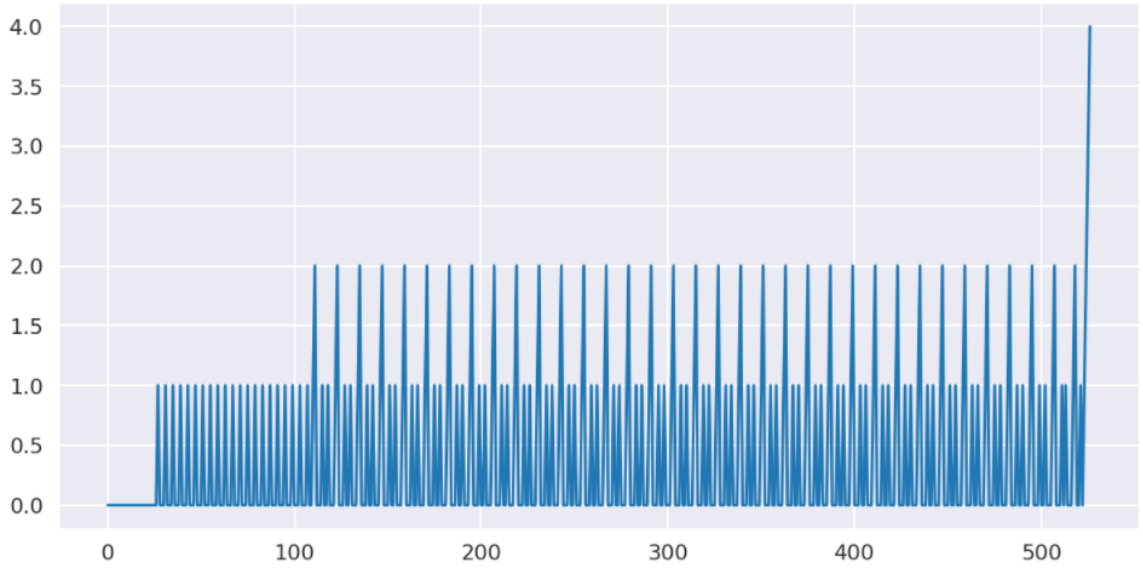
Since the function is linearly separable

(Однослойный персептрон правильно классифицировал весь набор данных для логической функции: OR.т. к. эта функция линейно разделима.)

Analyzing the Results (NAND)

```
p_nand = Perceptron(data, target_nand)
p_nand.train()
```

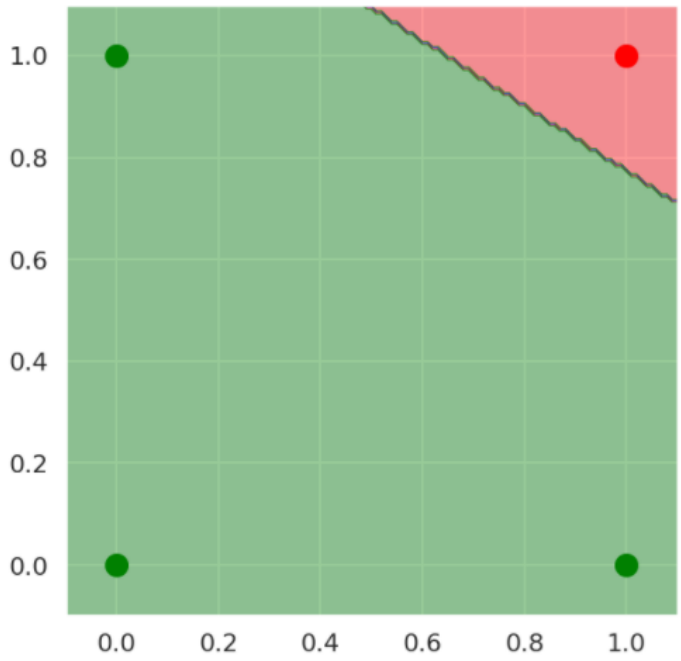
```
_ = plt.plot(p_nand.correct_iter[:1000])
```



NAND - (1, 1, 1, 0)

The plot shows perfect convergence. (График показывает идеальную сходимость.)

```
_ = p_nand.plot()
```




```
def NAND(x1, x2):

    x = [x1, x2]
    p_nand = Perceptron(data, target_nand)
    p_nand.train()

    return p_nand.classify(x)

# Test NAND
NAND(1, 1)
```

```
0
```

```
# Test NAND
NAND(0, 0)
```

```
1
```

The single-layer perceptron correctly classified the entire data set for the logical functions: AND, OR, and NAND.

Since these functions are linearly separable

(Однослойный персептрон правильно классифицировал весь набор данных для логических функций: AND, OR и NAND.

т. к. эти функции линейно разделимы.)

Analyzing the Results (XOR)

A single-layered perceptron isn't enough to model the 2d XOR function, so training will never converge. Here, convergence means correctly classifying all training examples consecutively.

(Однослойного персептрона недостаточно для моделирования функции 2d XOR, поэтому обучение никогда не сойдется. Здесь конвергенция означает правильную последовательную классификацию всех обучающих примеров.)

```
p_xor = Perceptron(data, target_xor)
p_xor.train()
```

```
1000 iterations exceded without convergence! A single layered perceptron can't handle the XOR problem.
```

The training loop never terminates, since a perceptron can only converge on linearly separable data. Linearly separable data basically means that you can separate data with a point in 1D, a line in 2D, a plane in 3D and so on.

(Цикл обучения никогда не заканчивается, поскольку персептрон может сходиться только на линейно разделимых данных. Линейно разделяемые данные в основном означают, что вы можете разделить данные с помощью точки в 1D, линии в 2D, плоскости в 3D и так далее.)

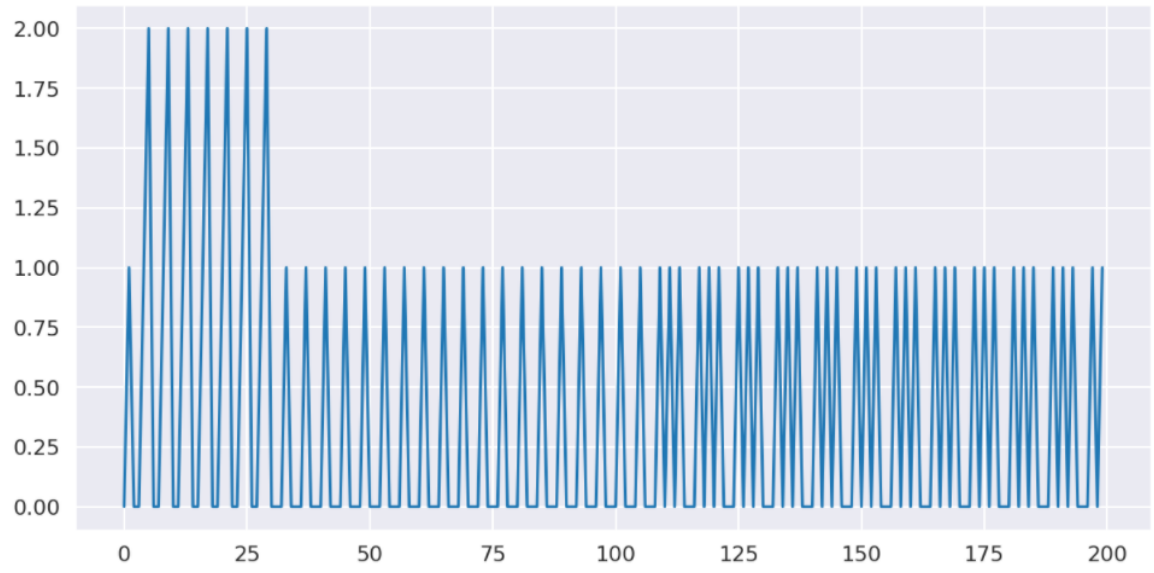
A perceptron can only converge on linearly separable data. Therefore, it isn't capable of imitating the XOR function.

(Персептрон может сходиться только на линейно разделимых данных. Следовательно, он не способен имитировать функцию XOR.)

A perceptron must correctly classify the entire training data in one go. If we keep track of how many points it correctly classified consecutively, we get something like this.

(персептрон должен правильно классифицировать все обучающие данные за один раз. Если мы отследим, сколько точек правильно классифицировано последовательно, мы получим что-то вроде этого.)

```
_ = plt.plot(p_xor.correct_iter[:200])
```



The algorithm only terminates when correct_counter hits 4 — which is the size of the training set — so this will go on indefinitely. (Алгоритм завершается только тогда, когда correct_counter достигает 4 — это размер тренировочного набора — так что это будет продолжаться бесконечно.)

A single layer perceptron is not enough to model the 2d XOR function, so the learning cycle never ends. this function is not linearly separable. But the XOR function can be modeled by combining AND, OR and NAND single layer perceptrons

(Однослойного персептрона недостаточно для моделирования функции 2d XOR, поэтому цикл обучения никогда не заканчивается, т.к. эта функция линейно не разделима. Но функцию XOR можно моделировать путем объединения однослойных персептронов AND, OR и NAND.)

Modeling the XOR function by combining single-layered perceptrons

(Моделирование функции XOR путем объединения однослойных персептронов)

The XOR function is basically a combination of an AND, OR and NAND. gate.

(Функция XOR в основном представляет собой комбинацию AND, OR и NAND.

XOR(x1, x2) = AND(OR(x1, x2), NAND(x1, x2))

```
def XOR(x1, x2):
    """
    Return the boolean XOR of x1 and x2 by combining individual single-layered perceptrons
    (Верните логическое XOR x1 и x2, объединив отдельные однослойные персептроны.)
    """

    x = [x1, x2]
    p_or = Perceptron(data, target_or)
    p_nand = Perceptron(data, target_nand)
    p_and = Perceptron(data, target_and)

    p_or.train()
    p_nand.train()
    p_and.train()

    return p_and.classify([p_or.classify(x),
                           p_nand.classify(x)
```

```
# Test.
# When we try and find the XOR of 1 and 1, we get 0 just like we expected.
# Когда мы пытаемся найти XOR 1 и 1, мы получаем 0, как и ожидали.

XOR(1, 1)
```

```
0
```

```
# Test.
XOR(0, 1)
```

```
1
```

```
# Test.
XOR(0, 0)
```

```
0
```

This model correctly classified the entire data set for the logical function: XOR.

(Такая модель правильно классифицировала весь набор данных для логической функции: XOR.)

MLP (Multi-layered perceptron)

A multi-layered perceptron can have hidden layers. (Многослойный персептрон может иметь скрытые слои.)

There are no fixed rules on the number of hidden layers or the number of nodes in each layer of a network. The best performing models are obtained through trial and error.

(Не существует фиксированных правил по количеству скрытых слоев или количеству узлов в каждом слое сети. Лучшие модели получаются методом проб и ошибок.)

Let’s go with a single hidden layer with two nodes in it. We’ll be using the sigmoid function in each of our hidden layer nodes and of course, our output node.

(Давайте возьмем один скрытый слой с двумя узлами в нем. Мы будем использовать сигмовидную функцию в каждом из наших узлов скрытого слоя и, конечно же, в нашем выходном узле.)

The MLP Class

This class houses each component of our MLP, from training and the forward pass, to classifying and plotting the decision boundary.

(Этот класс содержит каждый компонент нашего MLP, от обучения и прямого прохода до классификации и построения границы решения.)

```
class MLP:
    """
    Create a multi-layer perceptron.

    data: A 4x2 matrix with the input data.

    target: A 4x1 matrix with expected outputs

    lr: the learning rate. Defaults to 0.1

    num_epochs: the number of times the training data goes through the model
        while training
    (num_epochs: сколько раз обучающие данные проходят через модель
         во время тренировки)

    num_input: the number of nodes in the input layer of the MLP.
        Should be equal to the second dimension of data.
```

```
    (num_input: количество узлов во входном слое MLP.
         Должен быть равен второму измерению data.)

    num_hidden: the number of nodes in the hidden layer of the MLP.
    (num_hidden: количество узлов в скрытом слое MLP.)

    num_output: the number of nodes in the output layer of the MLP.
        Should be equal to the second dimension of target.
     (num_output: количество узлов в выходном слое MLP.
         Должен быть равен второму измерению цели.)

    """


    def __init__(self, data, target, lr=0.1, num_epochs=100, num_input=2, num_hidden=2, num_output=1):
        self.data = data
        self.target = target
        self.lr = lr
        self.num_epochs = num_epochs

        # initialize both sets of weights and biases randomly
        # (инициализировать оба набора весов и смещений случайным образом)

            # - weights_01: weights between input and hidden layer
            # (weights_01: веса между входным и скрытым слоями)

            # - weights_12: weights between hidden and output layer
            # (weights_12: веса между скрытым и выходным слоями)

        self.weights_01 = np.random.uniform(size=(num_input, num_hidden))
        self.weights_12 = np.random.uniform(size=(num_hidden, num_output))

        # - b01: biases for the  hidden layer
        # (b01: смещения для скрытого слоя)

        # - b12: bias for the output layer
        # (# - b12: смещение выходного слоя)

        self.b01 = np.random.uniform(size=(1,num_hidden))
        self.b12 = np.random.uniform(size=(1,num_output))

        self.losses = []

    def update_weights(self):

        # Calculate the squared error
        # (Вычислить квадрат ошибки)

        loss = 0.5 * (self.target - self.output_final) ** 2
        self.losses.append(np.sum(loss))

        error_term = (self.target - self.output_final)

        # the gradient for the hidden layer weights
        # (градиент веса скрытого слоя)

        grad01 = self.data.T @ (((error_term * self._delsigmoid(self.output_final)) * self.weights_12.T) *
                                    self._delsigmoid(self.hidden_out))

        # the gradient for the output layer weights
        # (градиент для весов выходного слоя)

        grad12 = self.hidden_out.T @ (error_term * self._delsigmoid(self.output_final))

        # updating the weights by the learning rate times their gradient
        # обновление весов по скорости обучения, умноженной на их градиент

        self.weights_01 += self.lr * grad01
        self.weights_12 += self.lr * grad12

        # update the biases the same way
        # (обновите biases таким же образом)

        self.b01 += np.sum(self.lr * ((error_term * self._delsigmoid(self.output_final)) * self.weights_12.T) *
                           self._delsigmoid(self.hidden_out), axis=0)
        self.b12 += np.sum(self.lr * error_term * self._delsigmoid(self.output_final), axis=0)

    def _sigmoid(self, x):
        """
        The sigmoid activation function.
        """
        return 1 / (1 + np.exp(-x))
```

```
    def _delsigmoid(self, x):
        """
        The first derivative of the sigmoid function wrt x
        (Первая производная сигмовидной функции)
        """
        return x * (1 - x)

    def forward(self, batch):
        """
        A single forward pass through the network. (Один прямой проход по сети.)
        Implementation of wX + b  (Реализация wX+b)
        """

        self.hidden_ = np.dot(batch, self.weights_01) + self.b01
        self.hidden_out = self._sigmoid(self.hidden_)

        self.output_ = np.dot(self.hidden_out, self.weights_12) + self.b12
        self.output_final = self._sigmoid(self.output_)

        return self.output_final

    def classify(self, datapoint):
        """
        Return the class to which a datapoint belongs based on
        the perceptron's output for that point.
        (Возвращает класс, к которому принадлежит точка данных, на основе
         выхода персептрона для этой точки.)
        """
        datapoint = np.transpose(datapoint)
        if self.forward(datapoint) >= 0.5:
            return 1

        return 0

    def plot(self, h=0.01):
        """
        Generate plot of input data and decision boundary.
        (Сгенерируйте график входных данных и границы решения.)
        """
        # setting plot properties like size, theme and axis limits
        sns.set_style('darkgrid')
        plt.figure(figsize=(10, 10))

        plt.axis('scaled')
        plt.xlim(-0.1, 1.1)
        plt.ylim(-0.1, 1.1)

        colors = {
            0: "ro",
            1: "go"
        }

        # plotting the four datapoints
        for i in range(len(self.data)):
            plt.plot([self.data[i][0]],
                     [self.data[i][1]],
                     colors[self.target[i][0]],
                     markersize=20)

        x_range = np.arange(-0.1, 1.1, h)
        y_range = np.arange(-0.1, 1.1, h)

        # creating a mesh to plot decision boundary
        xx, yy = np.meshgrid(x_range, y_range, indexing='ij')
        Z = np.array([[self.classify([x, y]) for x in x_range] for y in y_range])

        # using the contourf function to create the plot
        plt.contourf(xx, yy, Z, colors=['red', 'green', 'green', 'blue'], alpha=0.4)

    def train(self):
        """
        Train an MLP. Runs through the data num_epochs number of times.
        (Обучение МЛП. Проходит через данные num_epochs количество раз.)
        A forward pass is done first, followed by a backward pass (backpropagation)
        (Сначала выполняется прямой проход, а затем обратный проход (обратное распространение).)
        where the networks parameter's are updated.
        (где параметры сети обновляются)
        """
        for epoch in range(self.num_epochs):

            self.forward(self.data)
```

```
            self.update_weights()

            if epoch % 500 == 0:
                print("Loss: ", self.losses[epoch])
```

Analyzing the Results (XOR)

Let’s train our MLP with a learning rate of 0.25 over 9000 epochs.

(Давайте обучим наш MLP со скоростью обучения 0,25 за 9000 эпох.)

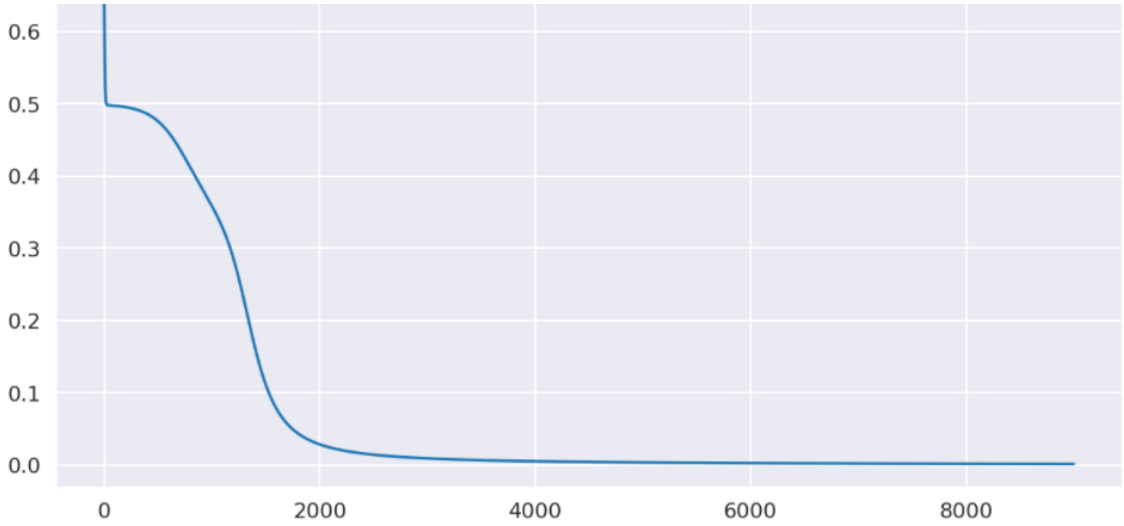
```
mlp = MLP(data, target_xor, 0.25, 9000)
mlp.train()
```

```
Loss:  0.656195847366289
Loss:  0.4760176756647404
Loss:  0.3588188934764573
Loss:  0.11155132719993556
Loss:  0.0286857179765561
Loss:  0.014387753765296347
Loss:  0.009281599407692962
Loss:  0.00675542164732832
Loss:  0.005271758142378163
Loss:  0.0043038170224446275
Loss:  0.0036259878334236563
Loss:  0.0031265326871645904
Loss:  0.002744138369190125
Loss:  0.002442494006589588
Loss:  0.0021987877970801387
Loss:  0.0019979978049526467
Loss:  0.0018298443837310332
Loss:  0.001687064633266797
```

If we plot the values of our loss function, we get the following plot after about 9000 iterations, showing that our model has indeed converged.

(Если мы построим значения нашей функции потерь, мы получим следующий график примерно после 9000 итераций, показывающий, что наша модель действительно сошлась.)

```
_ = plt.plot(mlp.losses)
```

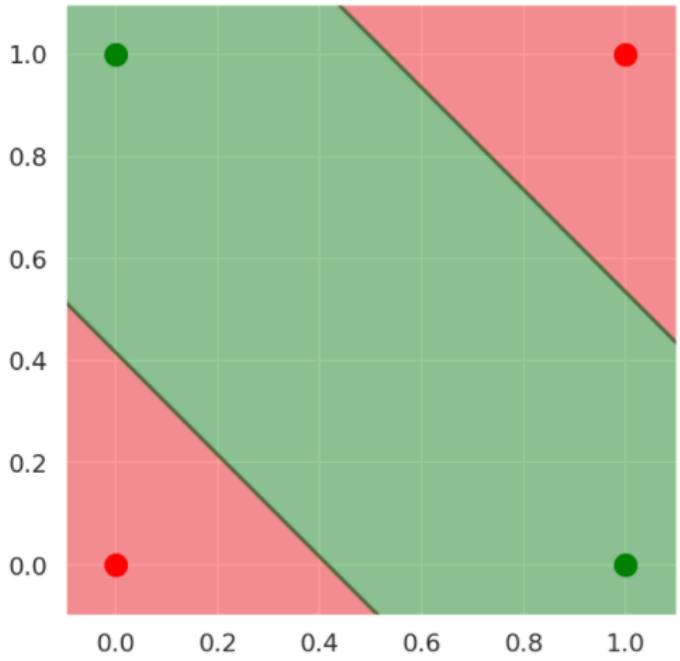


A clear non-linear decision boundary is created here with our generalized neural network, or MLP.

(Четкая нелинейная граница решения создается здесь с помощью нашей обобщенной нейронной сети или MLP.)

XOR - (0, 1, 1, 0)

```
mlp.plot()
```



MLP (Multi-layered perceptron) correctly classified the entire data set for the logical function: XOR.

(MLP (Multi-layered perceptron) правильно классифицировал весь набор данных для логической функции: XOR.)