

## **2. Генерация выборок. Элементарный перцептрон.**

**Цель работы** - смоделировать и исследовать поведение простейших моделей классификации на синтетических двумерных выборках. В первой части требуется реализовать генераторы данных для четырех типов распределений, добавляя случайную ошибку измерения по признакам  $x = (x_1, x_2)$ , и сформировать обучающую и тестовую выборки. Во второй части необходимо реализовать элементарный перцептрон для бинарной классификации в двух вариантах: со ступенчатой функцией активации и с сигмоидой. Дополнительно необходимо провести анализ качества и обобщающей способности: построение вычислительного графа и локальных производных для случая обратного распространения ошибки, представление результатов через матрицу ошибок (confusion matrix) и сравнение времени обучения и метрик классификации для обеих реализаций на разных типах данных, включая демонстрацию переобучения в playground.tensorflow.

### **Задания:**

1. Генерация обучающих и тестовых выборок
2. Реализация элементарного перцептрана

### **Ссылка на исходный код лабораторной работы:**

[https://github.com/F1ameX/Modern-Methods-of-Deep-Machine-Learning/blob/main/2\\_elementary\\_perceptron/2\\_elementary\\_perceptron.ipynb](https://github.com/F1ameX/Modern-Methods-of-Deep-Machine-Learning/blob/main/2_elementary_perceptron/2_elementary_perceptron.ipynb)

### **Ход работы** **Генерация обучающих и тестовых выборок**

В ходе проведения эксперимента были реализованы функции для генерации выборок с распределениями представленными на рис. 1 ниже. Каждый генератор реализует матрицу признаков  $X$  и вектор меток  $y$  по заданному алгоритму, затем к  $X$  добавляется некоторый шум измерения, после чего данные перемешиваются и разбиваются на обучающую и тестовые части. Далее будут подробно рассматриваться четыре типа распределений:

концентрические окружности (circles), XOR, гауссовские кластеры (blobs) и две спирали (spiral) в порядке слева направо на рис. 1.

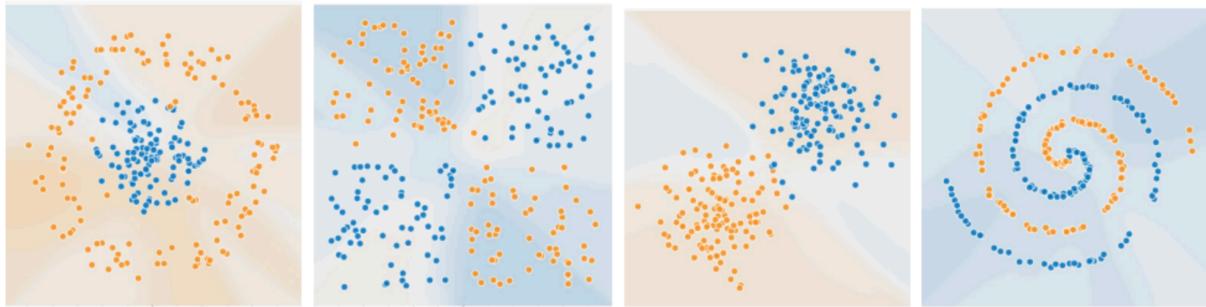


Рисунок 1 - Распределения для генераторов выборок

### Концентрические окружности

Генерация концентрических окружностей математически представляет из себя генерацию значений на интервале  $[0, 2\pi)$  и представление их через синусы и косинусы. Рассмотрим формальную математическую запись:

Пусть  $N = \text{n\_samples}$ ,  $f = \text{factor} \in (0, 1)$ ,  $N_{\text{out}} = \lfloor \frac{N}{2} \rfloor$ ,  $N_{\text{in}} = N - N_{\text{out}}$ .  
Зададим углы

$$\theta_i^{\text{out}} = \frac{2\pi i}{N_{\text{out}}}, \quad i = 0, \dots, N_{\text{out}} - 1, \quad \theta_j^{\text{in}} = \frac{2\pi j}{N_{\text{in}}}, \quad j = 0, \dots, N_{\text{in}} - 1.$$

Точки двух окружностей:

$$x_i^{\text{out}} = (\cos \theta_i^{\text{out}}, \sin \theta_i^{\text{out}}), \quad x_j^{\text{in}} = (f \cos \theta_j^{\text{in}}, f \sin \theta_j^{\text{in}}).$$

Итоговая выборка

$$X = \{x_i^{\text{out}}\}_{i=0}^{N_{\text{out}}-1} \cup \{x_j^{\text{in}}\}_{j=0}^{N_{\text{in}}-1}.$$

Метки классов

$$y = (\underbrace{0, \dots, 0}_{N_{\text{out}}}, \underbrace{1, \dots, 1}_{N_{\text{in}}})^\top.$$

Если задан  $\sigma = \text{noise}$ , то  $X \leftarrow X + \varepsilon$ , где  $\varepsilon_k \sim \mathcal{N}(0, \sigma^2 I_2)$ .

Рисунок 2 - Определение генерации распределения концентрических окружностей

Рассмотрим код генерации выборки данного распределения:

```
def make_circles(n_samples : int = 100,
                 shuffle : bool = True,
                 noise : float = None,
                 random_state : int = None,
                 factor : float = 0.8):

    rng = np.random.default_rng(seed = random_state)

    n_samples_out = n_samples // 2
    n_samples_in = n_samples - n_samples_out

    linspace_out = np.linspace(0, 2 * np.pi, n_samples_out, endpoint = False)
    linspace_in = np.linspace(0, 2 * np.pi, n_samples_in, endpoint = False)

    X1_out = np.cos(linspace_out) * 5
    X2_out = np.sin(linspace_out) * 5

    X1_in = np.cos(linspace_in) * 5 * factor
    X2_in = np.sin(linspace_in) * 5 * factor

    X = np.vstack(
        [np.append(X1_out, X1_in), np.append(X2_out, X2_in)])
    .T

    y = np.hstack (
        [np.zeros(n_samples_out, dtype = int), np.ones(n_samples_in, dtype = int)])
    )

    if noise is not None:
        X += rng.normal(scale = noise, size = X.shape)

    if shuffle:
        permutation = rng.permutation(n_samples)
        X = X[permutation]
        y = y[permutation]

    return X, y
```

Рисунок 3 - Код для генерации распределения концентрических окружностей

В функции `make_circles` выборка формируется как объединение двух окружностей: внешней радиуса 5 и внутренней радиуса  $5 * \text{factor}$ . Для обеих окружностей равномерно задаются углы на интервале  $[0, 2\pi)$ , после чего координаты точек вычисляются через  $\cos$  и  $\sin$ . Метки классов присваиваются как  $y = 0$  для внешней окружности и  $y = 1$  для внутренней соответственно. При заданном параметре `noise` к каждой точке добавляется гауссов шум,

моделирующий ошибку измерения признаков. Опционально выполняется случайная перестановка объектов для устранения влияния порядка генерации на обучение модели.

Таким образом, получаем бинарную задачу классификации с нелинейно разделимыми классами. Задача является нелинейно разделимой, потому что линейный классификатор строит лишь границу вида  $w_0 + w_1x_1 + w_2x_2 = 0$ , что является прямой, в то время как для окружностей естественная граница выглядит как  $x_1^2 + x_2^2 = R^2$ .

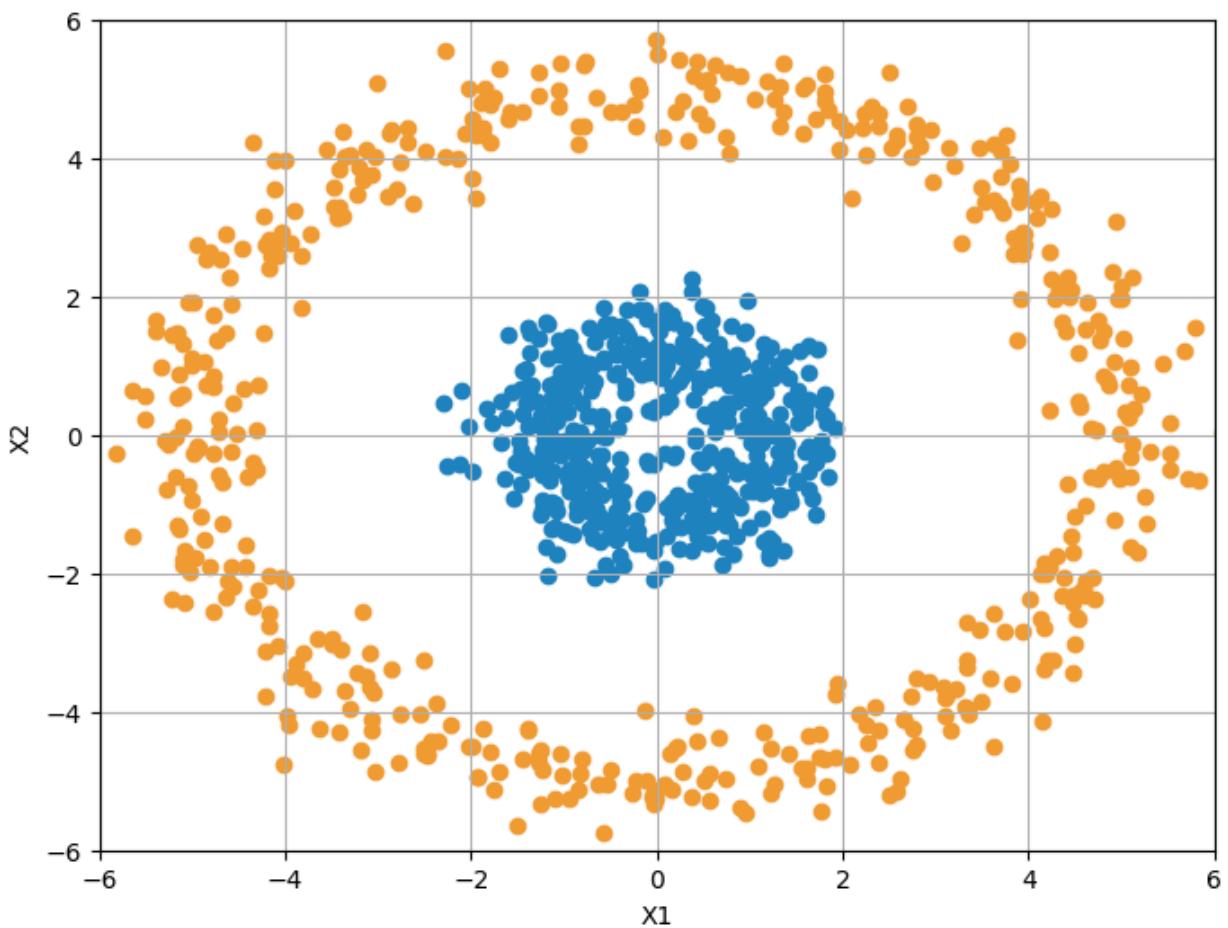


Рисунок 4 - График распределения концентрических окружностей

После этого на сайте [playground.tensorflow.org](http://playground.tensorflow.org) были подобраны такие параметры нейронной сети, благодаря которым было получено переобучение:

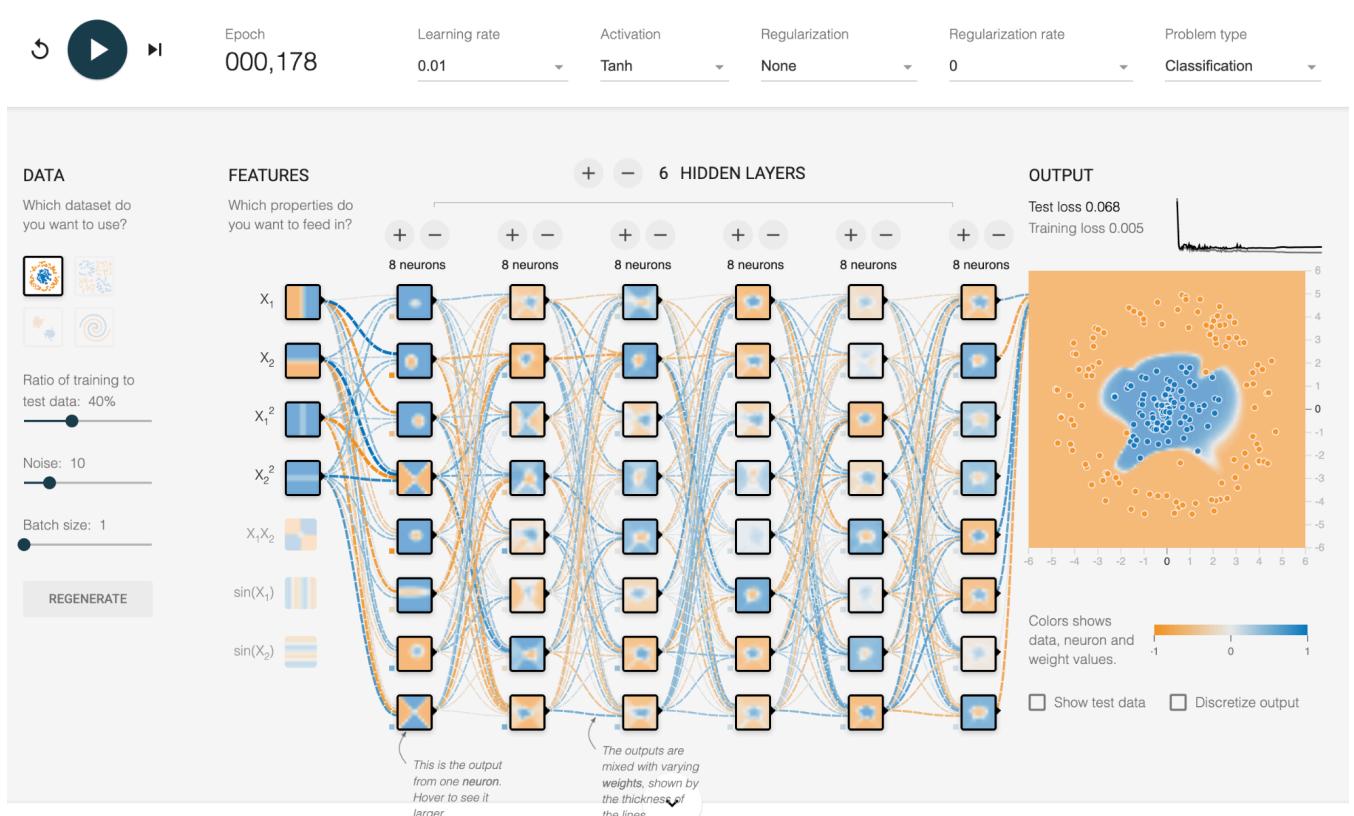


Рисунок 5 - Параметры сети для распределения концентрических окружностей

Для получения переобучения использовалась обучающая выборка размером 40% от всей полученной со значением шумом 10 и размером батча 1. На вход подаются значения  $x_1$ ,  $x_2$ ,  $x_1^2$ ,  $x_2^2$ . Добавление новых фичей (feature-engineering) связано с тем, что таким образом граница по окружности становится линейной по данным признакам. Сеть состоит из 6 скрытых слоёв по 8 нейронов на каждом. Используется learning rate = 0.01 и функция активации гиперболического тангенса. Переобучение фиксируется по разрыву между ошибками на обучающей и тестовой выборках: L\_train = 0.005 при L\_test = 0.068. Существенный разрыв (более чем на порядок) означает, что модель хорошо подгоняет обучающие данные, но хуже переносит знания на новые примеры.

### XOR распределение

Генерация XOR-распределения математически представляет собой формирование четырёх компактных кластеров в вершинах квадрата ( $\pm s, \pm s$ ) присвоение меток по правилу исключающего «ИЛИ» (XOR): класс зависит от

комбинации знаков координат. Каждая точка получается добавлением гауссовского шума к одному из центров, после чего выборка объединяется и при необходимости случайно перемешивается. Рассмотрим формальную математическую запись:

Пусть  $N = \text{n\_samples}$ ,  $\sigma = \text{noise}$ ,  $s = \text{scale}$ . Если  $\text{centers} = \emptyset$ , то зададим четыре центра  $C = \{c_0, c_1, c_2, c_3\} \subset \mathbb{R}^2$ , где  $c_0 = (s, s)$ ,  $c_1 = (s, -s)$ ,  $c_2 = (-s, s)$ ,  $c_3 = (-s, -s)$ . Положим  $N_0 = \lfloor \frac{N}{4} \rfloor$ ,  $r = N \bmod 4$  и определим размеры кластеров  $n_k = N_0 + \mathbb{I}[k < r]$  для  $k = 0, 1, 2, 3$ . Для каждого  $k$  сгенерируем  $n_k$  точек:  $x_i^{(k)} = c_k + \sigma z_i^{(k)}$ , где  $z_i^{(k)} \sim \mathcal{N}(0, I_2)$  независимо,  $i = 1, \dots, n_k$ . Метки зададим как  $y_i^{(k)} = 0$  при  $k \in \{0, 3\}$  и  $y_i^{(k)} = 1$  при  $k \in \{1, 2\}$ . Итоговую выборку получим конкатенацией всех кластеров  $(X, y)$  с последующей случайной перестановкой объектов.

Рисунок 6 - Определение генерации XOR распределения

Рассмотрим код генерации для данного распределения:

```
def make_xor(n_samples : int = 100,
             noise : float = 0.2,
             scale : float = 1.0,
             centers : ArrayLike | None = None,
             shuffle : bool = True,
             random_state : int = 42,
             return_centers : bool = False):

    rng = np.random.default_rng(seed = random_state)

    if centers is None:
        centers = np.array([
            (scale, scale),
            (scale, -scale),
            (-scale, scale),
            (-scale, -scale),
        ],
        dtype = float
    )

    n_centers = centers.shape[0]

    base = n_samples // n_centers
    extra = n_samples % n_centers

    clusters = np.zeros(n_centers, dtype = int)
```

Рисунок 7 - Первая часть кода генерации XOR распределения

```

for k in range(n_centers):
    if k < extra:
        clusters[k] = base + 1
    else:
        clusters[k] = base

X = np.empty((n_samples, 2))
y = np.empty(n_samples, dtype = int)

pos = 0
for k in range(n_centers):
    normal_matrix = rng.normal(loc = 0, scale = 1, size = (clusters[k], 2))
    X_k = centers[k] + noise * normal_matrix

    if k in [0, 3]:
        y[pos : pos + clusters[k]] = 0
    else:
        y[pos : pos + clusters[k]] = 1

    X[pos : pos + clusters[k]] = X_k
    pos += clusters[k]

if shuffle:
    permutation = rng.permutation(n_samples)
    X = X[permutation]
    y = y[permutation]

if return_centers:
    return X, y, centers
return X, y

```

Рисунок 8 - вторая часть кода генерации XOR распределения

В функции `make_xor` выборка формируется как объединение четырёх компактных кластеров, расположенных в вершинах квадрата с координатами  $(\pm \text{scale}, \pm \text{scale})$ . Если параметры `centers` не заданы, центры кластеров фиксируются как  $(s, s)$ ,  $(s, -s)$ ,  $(-s, s)$ ,  $(-s, -s)$ , где  $s = \text{scale}$ . Общее число объектов  $N = n_{\text{samples}}$  распределяется по кластерам максимально равномерно (разница в размерах кластеров не превышает 1). Далее для каждого кластера генерируются точки добавлением гауссовского шума к соответствующему центру: к каждой координате прибавляется случайное отклонение с масштабом `noise`, что моделирует “размытие” точек вокруг вершин квадрата.

Метки классов присваиваются по правилу XOR: два диагонально противоположных кластера относятся к одному классу, а два оставшихся - к другому. В реализованной схеме кластеры  $(s, s)$  и  $(-s, -s)$  получают метку  $y = 0$ , а  $(s, -s)$  и  $(-s, s)$  - метку  $y = 1$ . После генерации всех точек выборка при необходимости случайно перемешивается параметром `shuffle`, чтобы порядок генерации не влиял на обучение модели. Опционально можно вернуть также массив центров (`return_centers=True`).

Таким образом, получаем бинарную задачу классификации с нелинейно разделимыми классами. Задача является нелинейно разделимой, поскольку линейный классификатор строит лишь одну границу вида  $w_0 + w_1x_1 + w_2x_2 = 0$  (прямую), а в XOR-распределении точки одного класса расположены в двух кластерах по диагонали, то есть кластеры одного класса лежат в двух диагонально противоположных квадрантах, поэтому любая прямая, отделяющая один из них, неизбежно захватит один кластер.

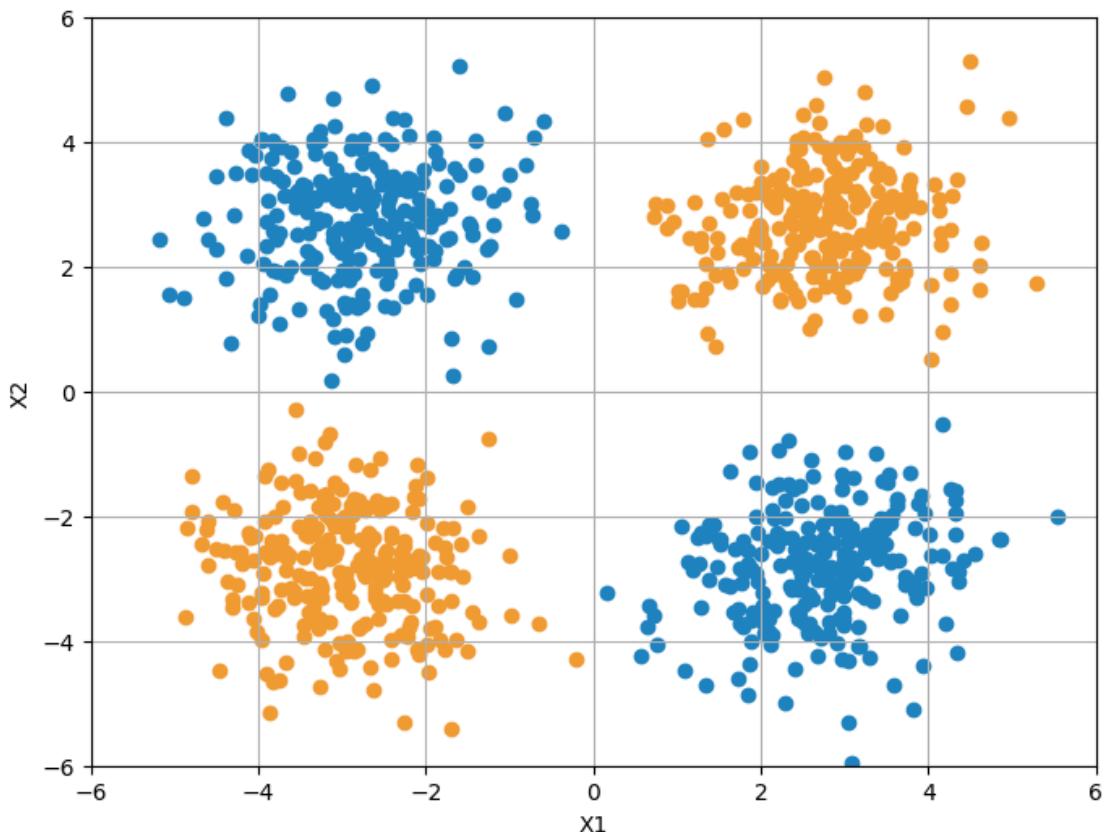


Рисунок 9 - График XOR распределения

После этого на сайте [playground.tensorflow.org](http://playground.tensorflow.org) были подобраны такие параметры нейронной сети, благодаря которым было получено переобучение:

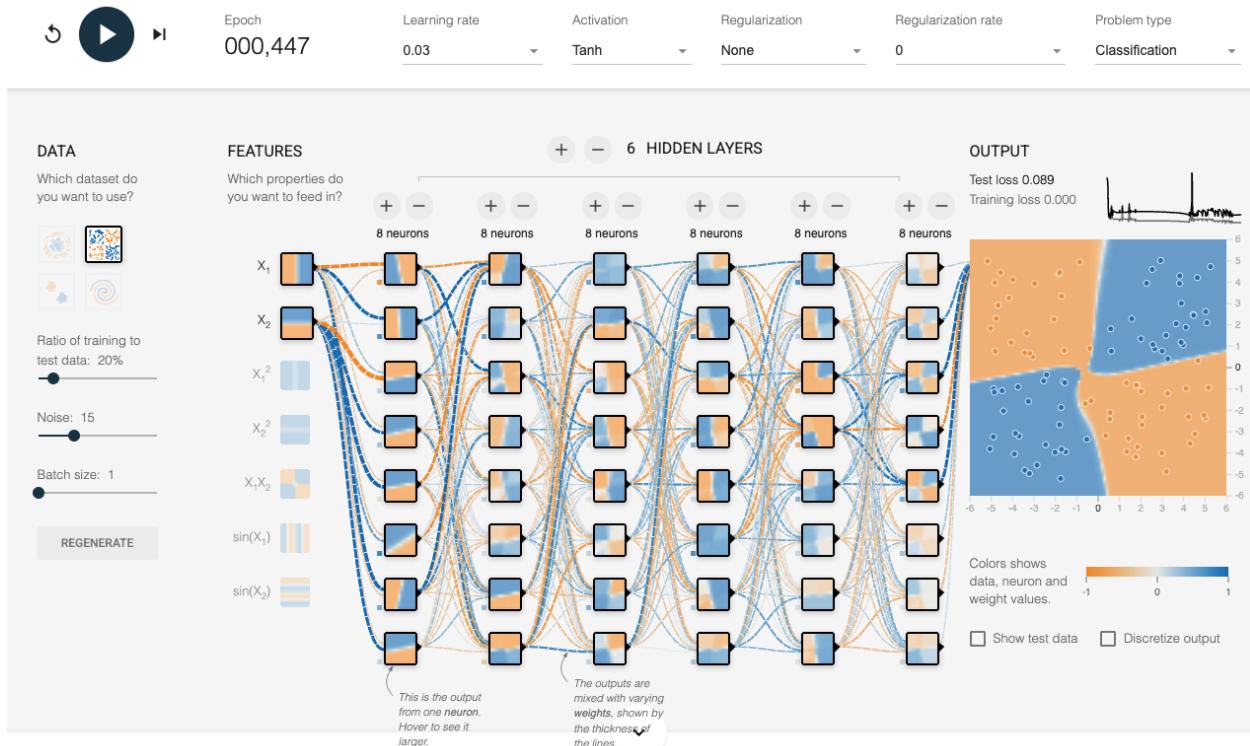


Рисунок 10 - Параметры сети для XOR распределения

Для получения переобучения использовалась обучающая выборка размером 20% от всей полученной со значением шума 15 и размером батча 1. На вход подаются значения  $x_1$ ,  $x_2$ . Сеть состоит из 6 скрытых слоёв по 8 нейронов на каждом. Используется learning rate = 0.03 и функция активации гиперболического тангенса. Переобучение фиксируется по разрыву между ошибками на обучающей и тестовой выборках:  $L_{train} = 0.0$  при  $L_{test} = 0.089$ . Наличие нулевой ошибки на обучающей выборке говорит о том, что построенная сеть подогнала параметры так, что ошибка на обучающих данных обнулилась. Это означает, что модель запомнила обучающие данные, однако на тестовых данных качество заметно хуже, что указывает на слабую обобщающую способность.

## Гауссовские кластеры

Генерация гауссовых кластеров математически представляет собой формирование нескольких компактных групп точек вокруг заранее заданных

(или случайно выбранных) центров в пространстве признаков. Для каждого центра  $C_k$  точки генерируются добавлением гауссовского шума, причём разброс внутри кластера регулируется параметром `cluster_std`. Далее объекты всех кластеров объединяются в одну выборку, им присваиваются метки по номеру кластера, а при необходимости выполняется случайная перестановка объектов для устранения влияния порядка генерации. Рассмотрим формальную математическую запись:

Пусть  $N = \text{n\_samples}$ ,  $d = \text{n\_features}$ ,  $K$  — число кластеров,  $[a, b] = \text{center\_box}$ . Если  $\text{centers} = \emptyset$ , то  $K = 2$  и  $c_k \sim U([a, b]^d)$ ,  $k = 0, \dots, K - 1$ ; если  $\text{centers} \in \mathbb{N}$ , то  $K = \text{centers}$  и центры задаются так же; иначе  $c_k$  берутся из заданного массива `centers`.

Положим  $q = \lfloor \frac{N}{K} \rfloor$ ,  $r = N \bmod K$ ,  $n_k = q + \mathbb{I}[k < r]$ . Если `cluster_std = σ`, то  $\sigma_k = \sigma$  для всех  $k$ , иначе  $\sigma_k$  задаются вектором `cluster_std`. Для каждого  $k$  генерируем независимые  $z_i^{(k)} \sim \mathcal{N}(0, I_d)$ ,  $i = 1, \dots, n_k$ , и задаём

$$x_i^{(k)} = c_k + \sigma_k z_i^{(k)}, \quad y_i^{(k)} = k.$$

Итоговая выборка:

$$X = \bigcup_{k=0}^{K-1} \{x_i^{(k)}\}_{i=1}^{n_k}, \quad y = \bigcup_{k=0}^{K-1} \{y_i^{(k)}\}_{i=1}^{n_k}.$$

Рисунок 12 - Определение генерации гауссовых кластеров

Рассмотрим код для генерации данного распределения:

```
def make_blobs(n_samples : int = 100,
               n_features : int = 2,
               centers : int | ArrayLike | None = None,
               cluster_std : float | ArrayLike = 1.0,
               center_box : tuple[float, float] = (-1.0, 1.0),
               shuffle : bool = True,
               random_state : int = 42,
               return_centers : bool = False):

    rng = np.random.default_rng(seed = random_state)

    if centers is None:
        n_centers = 2

        centers = rng.uniform(
            center_box[0], center_box[1], size = (n_centers, n_features)
        )

    elif isinstance(centers, int):
        n_centers = centers
        centers = rng.uniform(
            center_box[0], center_box[1], size = (n_centers, n_features)
        )
```

Рисунок 13 - Первая часть кода генерации гауссовых кластеров

```

else:
    centers = np.asarray(centers, dtype = float)
    n_centers = centers.shape[0]

base = n_samples // n_centers
extra = n_samples % n_centers

clusters = np.zeros(n_centers, dtype = int)

for k in range(n_centers):
    if k < extra:
        clusters[k] = base + 1
    else:
        clusters[k] = base

if isinstance(cluster_std, int | float):
    cluster_std_sigma = cluster_std
    cluster_std = np.repeat(cluster_std_sigma, n_centers)

X = np.empty((n_samples, n_features))
y = np.empty(n_samples, dtype = int)

pos = 0

for k in range(n_centers):
    normal_matrix = rng.normal(loc = 0, scale = 1, size = (clusters[k], n_features))
    X_k = centers[k] + cluster_std[k] * normal_matrix

    y[pos : pos + clusters[k]] = k
    X[pos : pos + clusters[k]] = X_k
    pos += clusters[k]

if shuffle:
    permutation = rng.permutation(n_samples)
    X = X[permutation]
    y = y[permutation]

if return_centers:
    return X, y, centers

return X, y

```

Рисунок 14 - Вторая часть кода для генерации гауссовских кластеров

В функции `make_blobs` выборка формируется как объединение нескольких гауссовских кластеров в пространстве признаков размерности `n_features`. Центры кластеров задаются параметром `centers`: если `centers=None`, используется два кластера `n_centers = 2`, а их координаты генерируются равномерно в прямоугольнике `center_box`; если `centers` — целое число, оно трактуется как количество кластеров, и центры также случайно выбираются в пределах `center_box`; если `centers` задан массивом, он напрямую используется как матрица центров кластеров.

Общее число объектов  $N = n_{samples}$  распределяется по кластерам максимально равномерно: каждый кластер получает либо  $N / n_{centers}$ , либо  $N / n_{centers} + 1$  точек (разница между кластерами не превышает 1). Далее точки каждого кластера генерируются как центр плюс гауссовское отклонение: к каждой координате прибавляется нормально распределенный шум. Масштаб разброса задаётся параметром `cluster_std`: если он задан одним числом, одинаковое стандартное отклонение используется для всех кластеров; если передан массив, то для каждого кластера применяется своё значение стандартного отклонения.

Метки классов формируются естественным образом по номеру кластера: для  $k$ -го кластера всем его объектам присваивается метка  $y = k$ . После генерации всех кластеров выборка при необходимости случайно перемешивается параметром `shuffle`, чтобы порядок генерации не влиял на дальнейшее обучение моделей. Опционально можно вернуть также массив центров кластеров, если установлен флаг `return_centers=True`.

Таким образом, получаем задачу классификации с классами, имеющими гауссовскую структуру вокруг центров. В отличие от окружностей и XOR, такие данные часто являются линейно разделимыми при достаточно разнесенных центрах и умеренном шуме, однако при больших значениях разброса кластеры начинают перекрываться, и задача становится существенно сложнее (вплоть до сильной неоднозначности границы).

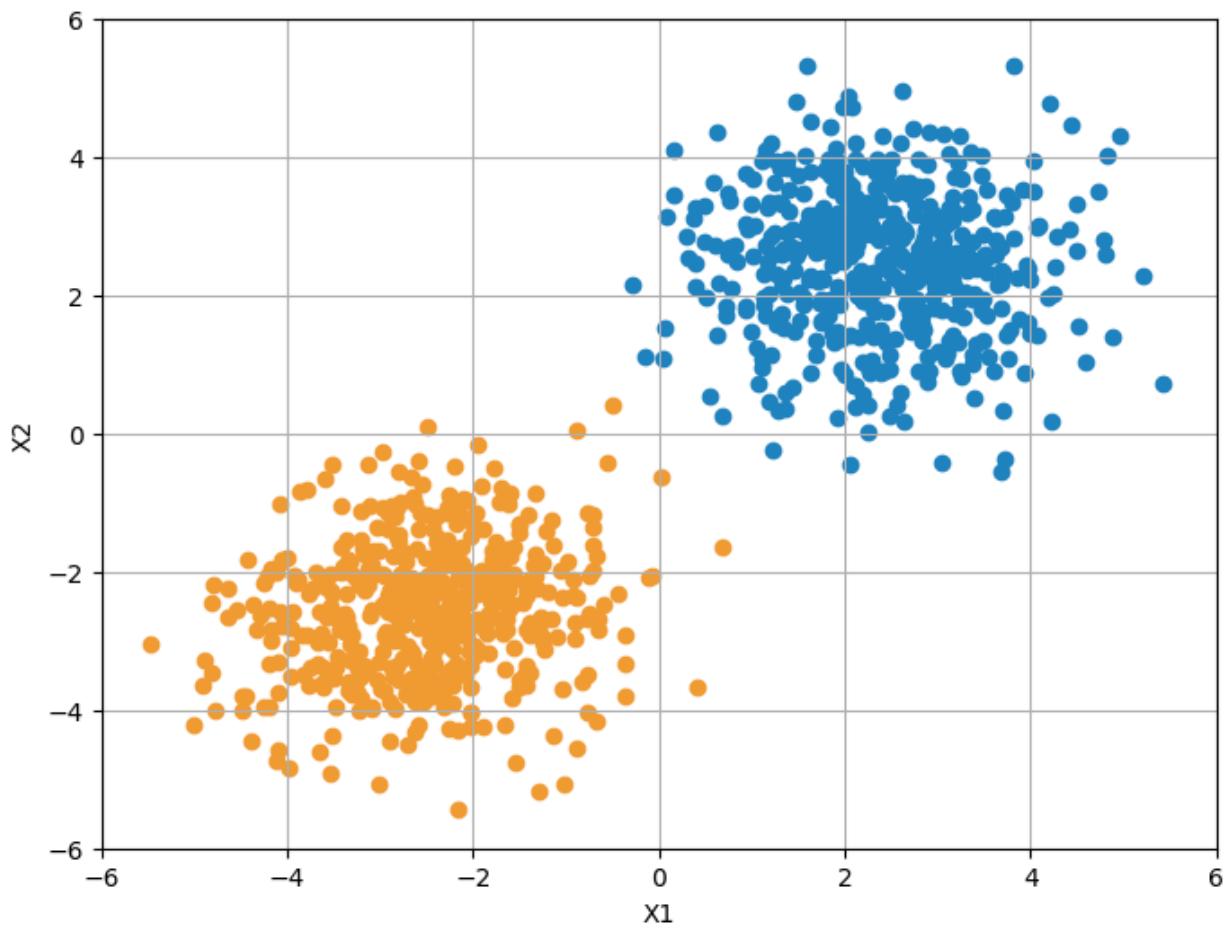


Рисунок 15 - График гауссовских кластеров

После этого на сайте [playground.tensorflow.org](https://playground.tensorflow.org) были подобраны такие параметры нейронной сети, благодаря которым было получено переобучение:

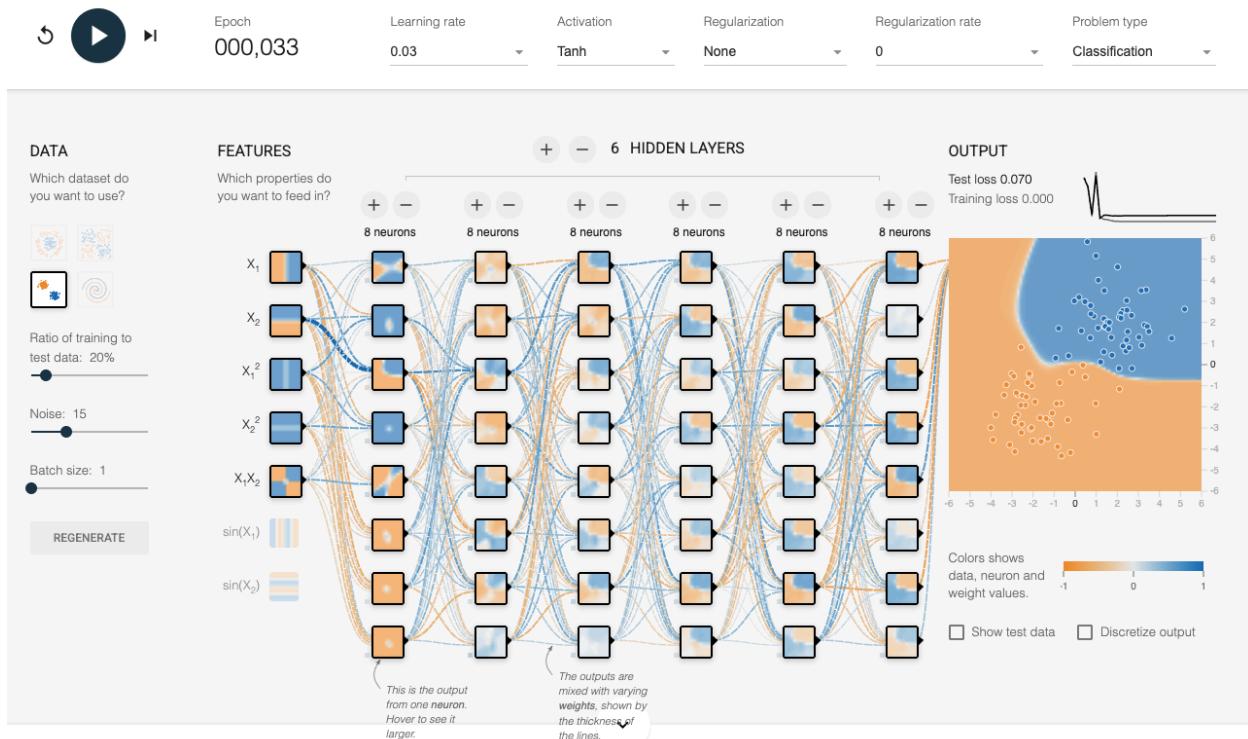


Рисунок 16 - Параметры сети для гауссовских кластеров

Для получения переобучения использовалась обучающая выборка размером 20% от всей сгенерированной, со значением шума 15 и размером батча 1. На вход сети подавались признаки  $x_1$ ,  $x_2$ ,  $x_1^2$ ,  $x_2^2$ ,  $x_1x_2$ , что повышает выразительность модели даже для простых гауссовых кластеров. Нейронная сеть состояла из 6 скрытых слоев по 8 нейронов в каждом, использовались  $learning\ rate = 0.03$  и функция активации гиперболического тангенса.

Переобучение фиксируется по существенному разрыву между функцией потерь на обучающей и тестовой выборках:  $L_{train} = 0.000$  при  $L_{test} = 0.070$ . Нулевая ошибка на обучающей выборке означает, что сеть фактически запомнила обучающие примеры вплоть до шума и локальных особенностей распределения, однако на тестовых данных качество заметно хуже. Это указывает на низкую обобщающую способность модели при выбранных гиперпараметрах и подтверждает наличие переобучения.

## Сpirали Архимеда

Генерация спиралей Архимеда математически представляет собой построение двух кривых в полярных координатах вида  $r(\theta) = r_0 + a\theta$ , где  $r_0 = \text{radius}$ ,  $a = \text{sweep}$ , а  $\theta$  равномерно пробегает интервал  $[0, 2 * \pi * \text{turns}]$ . Для получения двух классов строятся две спирали: первая задается параметрически как  $(x_1, x_2) = (r(\theta)\cos\theta; r(\theta)\sin\theta)$ , а вторая — как та же спираль, но повернутая на  $\pi$ :  $(x_1, x_2) = (r(\theta)\cos(\theta+\pi); r(\theta)\sin(\theta+\pi))$ . Далее точки обеих спиралей объединяются в одну выборку, метки присваиваются как  $y = 0$  для первой спирали и  $y = 1$  для второй. При заданном параметре  $\text{noise}$  к каждой точке добавляется гауссов шум, моделирующий ошибку измерения признаков, после чего при необходимости выполняется случайная перестановка объектов  $\text{shuffle}$  для устранения влияния порядка генерации на обучение модели. Рассмотрим формальную математическую запись:

Пусть  $N = \text{n\_samples}$ ,  $T = \text{turns}$ ,  $r_0 = \text{radius}$ ,  $a = \text{sweep}$ . Положим

$$N_{\text{out}} = \left\lfloor \frac{N}{2} \right\rfloor, \quad N_{\text{in}} = N - N_{\text{out}}.$$

Зададим углы

$$\theta_i^{\text{out}} = \frac{2\pi T}{N_{\text{out}}} i, \quad i = 0, \dots, N_{\text{out}} - 1, \quad \theta_j^{\text{in}} = \frac{2\pi T}{N_{\text{in}}} j, \quad j = 0, \dots, N_{\text{in}} - 1.$$

Радиальная функция спирали Архимеда:

$$r(\theta) = r_0 + a\theta.$$

Точки двух спиралей:

$$x_i^{\text{out}} = \left( r(\theta_i^{\text{out}}) \cos \theta_i^{\text{out}}, r(\theta_i^{\text{out}}) \sin \theta_i^{\text{out}} \right), \quad x_j^{\text{in}} = \left( r(\theta_j^{\text{in}}) \cos(\theta_j^{\text{in}} + \pi), r(\theta_j^{\text{in}}) \sin(\theta_j^{\text{in}} + \pi) \right).$$

Итоговая выборка и метки:

$$X = \{x_i^{\text{out}}\}_{i=0}^{N_{\text{out}}-1} \cup \{x_j^{\text{in}}\}_{j=0}^{N_{\text{in}}-1}, \quad y = (\underbrace{0, \dots, 0}_{N_{\text{out}}}, \underbrace{1, \dots, 1}_{N_{\text{in}}})^\top.$$

Если задано  $\sigma = \text{noise}$ , то  $X \leftarrow X + \varepsilon$ , где строки  $\varepsilon_k \sim \mathcal{N}(0, \sigma^2 I_2)$  независимы.

Рисунок 17 - Определение генерации спиралей Архимеда

Рассмотрим код для генерации данного распределения:

```
def make_spiral(n_samples : int = 100,
                 turns : int = 2,
                 radius : float = 0.0,
                 sweep : float = 0.15,
                 shuffle : bool = True,
                 noise : float = None,
                 random_state : int = 42):

    rng = np.random.default_rng(seed = random_state)

    n_samples_out = n_samples // 2
    n_samples_in = n_samples - n_samples_out

    theta_out = np.linspace(0, turns * 2 * np.pi, n_samples_out, endpoint = False)
    theta_in = np.linspace(0, turns * 2 * np.pi, n_samples_in, endpoint = False)

    spiral_out = radius + sweep * theta_out
    spiral_in = radius + sweep * theta_in

    X1_out = spiral_out * np.cos(theta_out)
    X2_out = spiral_out * np.sin(theta_out)

    X1_in = spiral_in * np.cos(theta_in + np.pi)
    X2_in = spiral_in * np.sin(theta_in + np.pi)

    X = np.vstack(
        [np.append(X1_out, X1_in), np.append(X2_out, X2_in)])
    ).T

    y = np.hstack(
        [np.zeros(n_samples_out, dtype = int), np.ones(n_samples_in, dtype = int)])
    )

    if noise is not None:
        X += rng.normal(scale = noise, size = X.shape)

    if shuffle:
        permutation = rng.permutation(n_samples)
        X = X[permutation]
        y = y[permutation]

return X, y
```

Рисунок 18 - Код для генерации спиралей Архимеда

В функции `make_spiral` выборка формируется как объединение двух спиралей Архимеда (два класса), заданных параметрически в двумерном пространстве. Сначала общее число объектов  $N = n\_samples$  делится на две части:  $N_{out} = N/2$  точек для первой спирали и  $N_{in} = N - N_{out}$  точек для второй. Для

каждой части независимо задаётся равномерная сетка углов  $\theta$  на интервале  $[0;2\pi)$ , что определяет «количество витков» спирали.

Далее для каждого значения угла строится радиальная координата по закону спирали Архимеда:  $r(\theta) = \text{radius} + \text{sweep} * \theta$ , после чего вычисляются декартовы координаты точек первой спирали как  $(x_1, x_2) = (r(\theta)\cos\theta; r(\theta)\sin\theta)$ .

Вторая спираль строится по той же формуле радиуса, но поворачивается на  $\pi$  (т.е. отражается относительно начала координат): вместо  $\theta$  используется  $\theta + \pi$ , поэтому ее точки имеют вид  $(x_1, x_2) = (r(\theta)\cos(\theta + \pi); r(\theta)\sin(\theta + \pi))$ .

Затем точки обеих спиралей объединяются в одну матрицу признаков  $X$ . Метки классов присваиваются так:  $y = 0$  для точек первой спирали и  $y = 1$  для точек второй. При заданном параметре `noise` к каждой точке добавляется двумерный гауссов шум, моделирующий ошибку измерения признаков.

После этого, если задан флаг `shuffle`, применяется случайная перестановка объектов, чтобы порядок генерации не влиял на последующее обучение модели.

Таким образом, получаем бинарную задачу классификации с нелинейной границей: классы представляют собой две взаимно «переплетенные» кривые, поэтому одной прямой вида  $w_0 + w_1x_1 + w_2x_2 = 0$  их корректно разделить нельзя, и для качественной классификации требуется нелинейная модель/признаки.

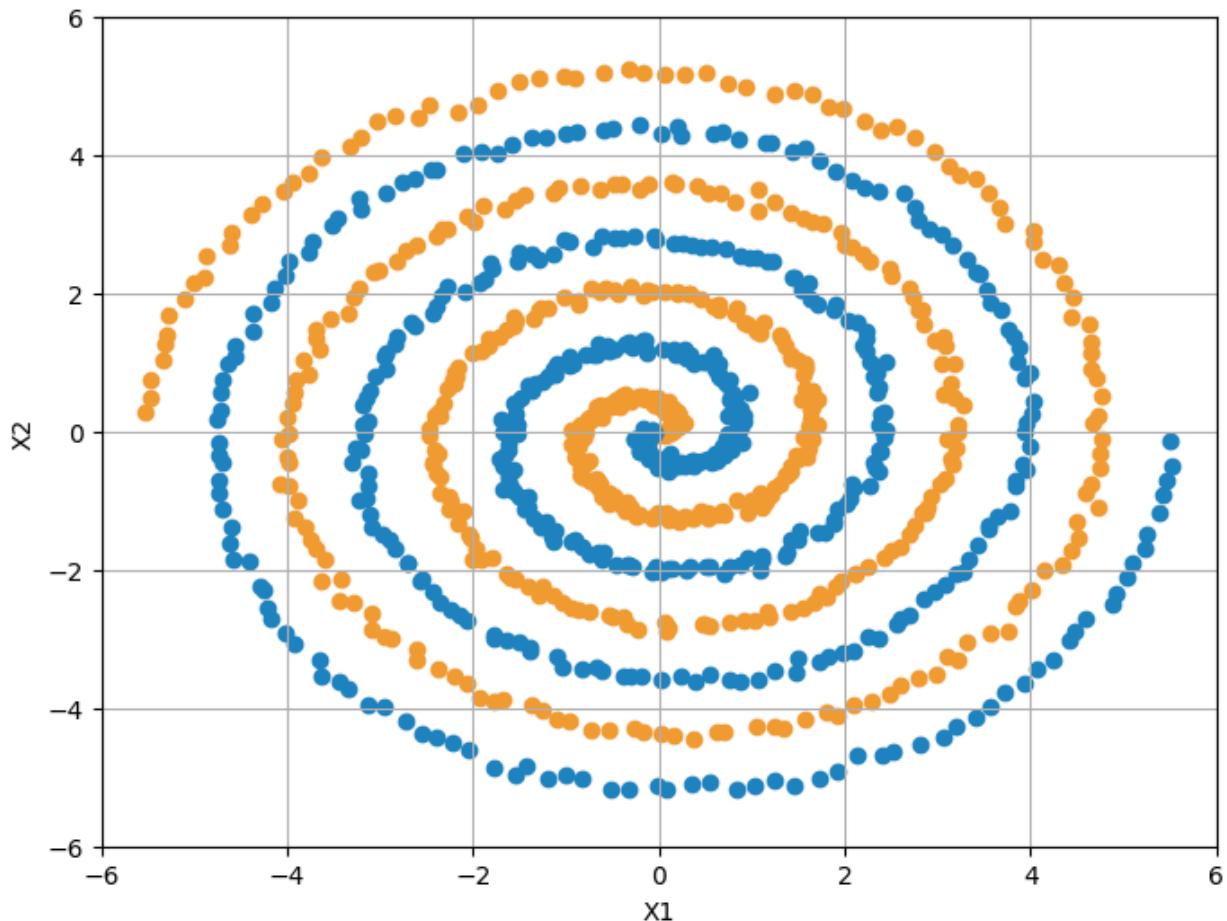


Рисунок 19 - График спиралей Архимеда

После этого на сайте [playground.tensorflow.org](https://playground.tensorflow.org) были подобраны такие параметры нейронной сети, благодаря которым было получено переобучение:

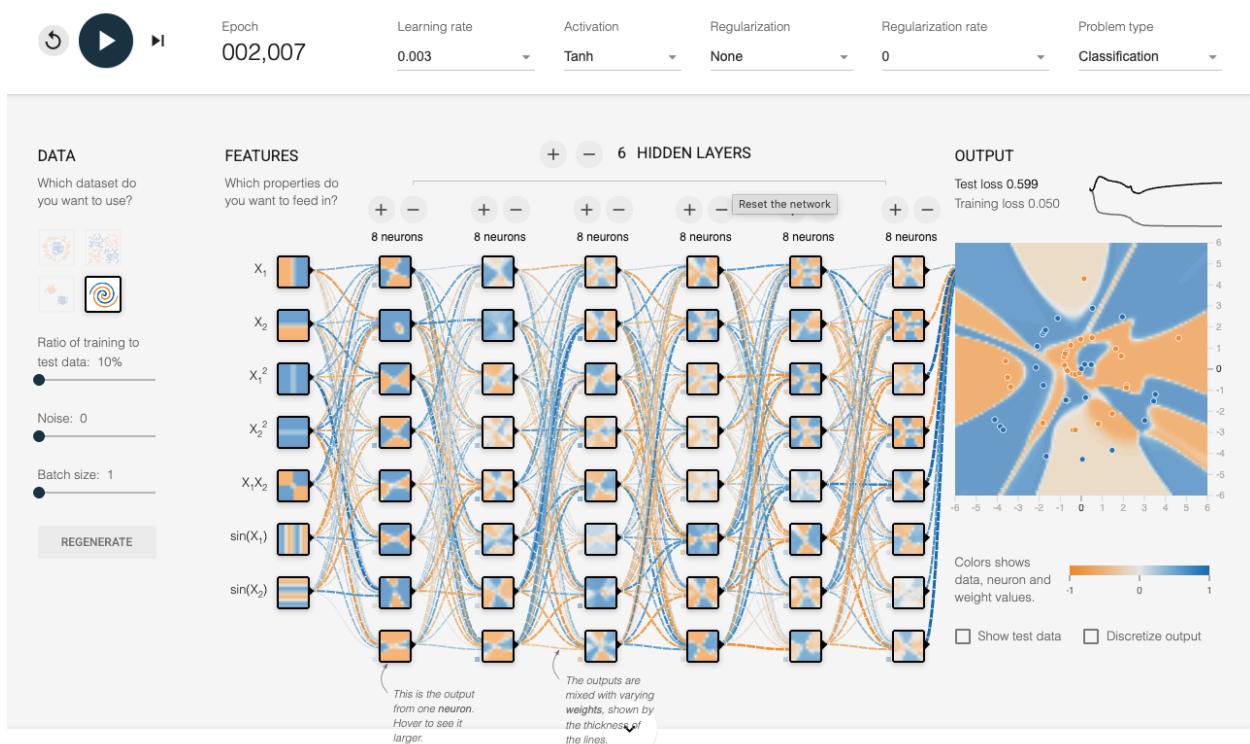


Рисунок 20 - Параметры сети для спиралей Архимеда

Для получения переобучения использовалась обучающая выборка размером 10% от всей сгенерированной, при нулевом шуме и размере батча 1. На вход сети подавались признаки  $x_1$ ,  $x_2$ ,  $x_1^2$ ,  $x_2^2$ ,  $x_1x_2$ ,  $\sin(x_1)$ ,  $\sin(x_2)$ , что позволяет модели описывать периодическую и сильно нелинейную структуру спиралей. Нейронная сеть состояла из 6 скрытых слоев по 8 нейронов в каждом, использовались learning rate = 0.003 и функция активации гиперболического тангенса.

Переобучение фиксируется по разрыву между функцией потерь на обучающей и тестовой выборках:  $L_{\text{train}} = 0.050$  при  $L_{\text{test}} = 0.599$ . Существенно меньшая ошибка на обучающей выборке означает, что сеть хорошо подогнала границу под обучающие точки, однако на тестовых данных качество заметно хуже. Это указывает на слабую обобщающую способность модели при выбранных гиперпараметрах и подтверждает наличие переобучения.

## Реализации элементарного перцептрана

В ходе выполнения второй части работы был реализован элементарный перцептрон для бинарной классификации. Рассматривались две функции активации: ступенчатая и сигмоида. Для ступенчатой активации обучение проводилось по классическому правилу перцептрана (алгоритм из теоремы о сходимости), а для сигмоидальной — методом градиентного спуска с вычислением градиентов через обратное распространение ошибки. Качество моделей оценивалось по матрице ошибок (confusion matrix), а также выполнялось сравнение времени обучения и итоговой точности классификации.

### Элементарный перцептрон

Рассмотрим математическую модель элементарного перцептрана:

Пусть на вход подаётся вектор признаков  $x \in \mathbb{R}^d$ . Введём фиктивный признак  $x_0 = 1$  (bias) и расширенный вектор  $\tilde{x} = (x_0, x_1, \dots, x_d)^\top \in \mathbb{R}^{d+1}$ , а также веса  $\tilde{w} = (w_0, w_1, \dots, w_d)^\top$ .

$$\begin{cases} a = \sum_{i=1}^d w_i x_i + w_0 = \tilde{w}^\top \tilde{x}, \\ z = \varphi(a). \end{cases}$$

Рассматривались две основные две функции активации  $\varphi(\cdot)$ :

**Ступенчатая:**  $\varphi_{\text{step}}(a) = \begin{cases} 1, & a \geq 0, \\ 0, & a < 0, \end{cases} \Rightarrow \hat{y} = \varphi_{\text{step}}(\tilde{w}^\top \tilde{x}).$

**Сигмоида:**  $\varphi_\sigma(a) = \sigma(a) = \frac{1}{1 + e^{-a}}, \Rightarrow p(y = 1 | x) = \sigma(\tilde{w}^\top \tilde{x}), \hat{y} = \mathbb{I}[p \geq \frac{1}{2}]$ .

Рисунок 21 - Математическая модель элементарного перцептрана

Данная модель была реализована при помощи следующего кода:

```
class Perceptron:
    def __init__(self, activation_function: str = 'step', fit_intercept: bool = True):
        assert activation_function in ['step', 'sigmoid']
        self.activation_function = activation_function
        self.fit_intercept = fit_intercept

        self.weights = None

    def _add_bias(self, X: np.ndarray) -> np.ndarray:
        if self.fit_intercept:
            return np.hstack([np.ones((X.shape[0], 1)), X])
        return X
```

Рисунок 22 - Первая часть кода реализации элементарного перцептрана

```

def decision_function(self, X : np.ndarray) -> np.ndarray:
    Xb = self._add_bias(X)
    return Xb @ self.weights

def predict_proba(self, X : np.ndarray) -> np.ndarray:
    if self.activation_function != 'sigmoid':
        raise ValueError("predict_proba method available only for sigmoid function!")
    z = np.clip(self.decision_function(X), -50, 50)
    return 1 / (1 + np.exp(-z))

def predict(self, X : np.ndarray) -> np.ndarray:
    if self.activation_function == 'step':
        return (self.decision_function(X) >= 0).astype(int)
    else:
        return (self.predict_proba(X) >= 0.5).astype(int)

def fit(self, X: np.ndarray,
        y: np.ndarray,
        random_state : int = 42,
        learning_rate : float = 0.01,
        n_epochs : int = 100,
        ):
    X = np.asarray(X, dtype = float)
    y = np.asarray(y, dtype = int).ravel()

    rng = np.random.default_rng(seed = random_state)

    Xb = self._add_bias(X)
    n_features = Xb.shape[1]

    self.weights = np.zeros(n_features, dtype = float)

```

Рисунок 23 - Вторая часть кода реализации элементарного перцептрана

```

if self.activation_function == 'step':
    Xb_base = Xb.copy()
    y_norm_base = 2 * y - 1

    for epoch in tqdm(
        range(n_epochs),
        desc = "Fitting perceptron",
        unit = "epoch",
        total = n_epochs,
        dynamic_ncols = True,
        leave = True,
        mininterval = 0.1):
        updates_epoch = 0

        permutation = rng.permutation(X.shape[0])
        Xb = Xb_base[permutation]
        y_norm = y_norm_base[permutation]

        for i in range(Xb.shape[0]):
            if y_norm[i] * (Xb[i] @ self.weights) <= 0:
                self.weights += learning_rate * y_norm[i] * Xb[i]
            updates_epoch += 1

        if updates_epoch == 0:
            break

```

Рисунок 24 - Третья часть кода реализации элементарного перцептрана

```

else:
    for epoch in tqdm(
        range(n_epochs),
        desc = "Fitting perceptron",
        unit = "epoch",
        total = n_epochs,
        dynamic_ncols = True,
        leave = True,
        mininterval = 0.1):

        z = np.clip(Xb @ self.weights, -50, 50)
        p = 1 / (1 + np.exp(-z))
        grad = (Xb.T @ (p - y)) / Xb.shape[0]
        self.weights -= learning_rate * grad

    return self

```

Рисунок 25 - Четвертая часть кода реализации элементарного перцептрана

Рассмотрим вычислительный граф данного перцептрана:

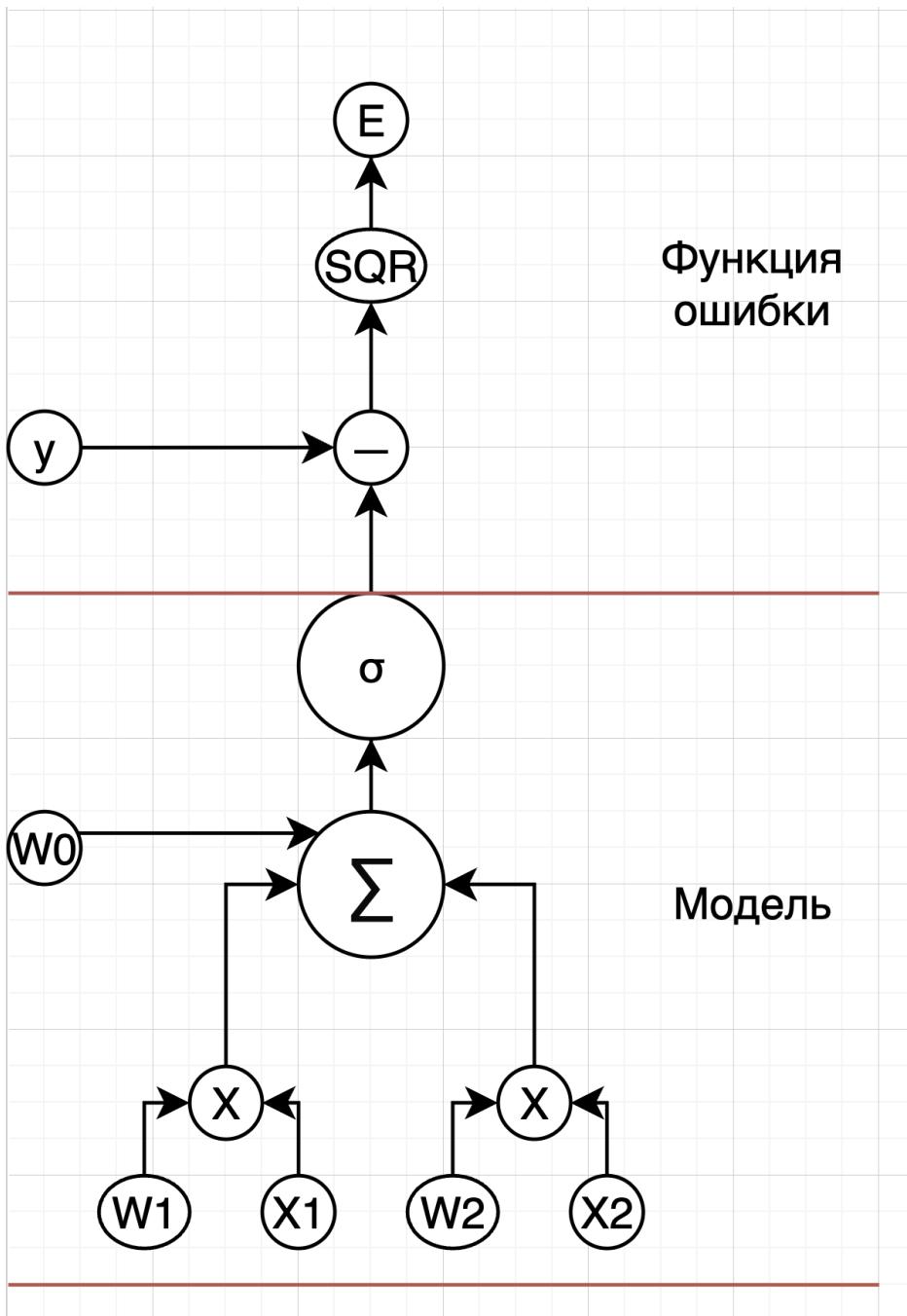


Рисунок 26 - Вычислительный граф элементарного перцептрана

Ниже также приведен расчёт локальных производных в ходе обратного распространения ошибки:

$$\hat{y} = \sigma(\Sigma), \quad \delta = y - \hat{y}, \quad E = \delta^2.$$

$$\frac{\partial E}{\partial \delta} = 2\delta, \quad \frac{\partial \delta}{\partial y} = 1, \quad \frac{\partial \delta}{\partial \hat{y}} = -1$$

$$\frac{\partial \hat{y}}{\partial \Sigma} = \sigma(\Sigma)(1 - \sigma(\Sigma)) = \hat{y}(1 - \hat{y})$$

$$\frac{\partial \Sigma}{\partial w_0} = 1, \quad \frac{\partial \Sigma}{\partial (w_1 x_1)} = 1, \quad \frac{\partial \Sigma}{\partial (w_2 x_2)} = 1$$

$$\frac{\partial (w_1 x_1)}{\partial w_1} = x_1, \quad \frac{\partial (w_1 x_1)}{\partial x_1} = w_1, \quad \frac{\partial (w_2 x_2)}{\partial w_2} = x_2, \quad \frac{\partial (w_2 x_2)}{\partial x_2} = w_2$$

$$\frac{\partial E}{\partial \hat{y}} = -2(y - \hat{y}), \quad \frac{\partial E}{\partial \Sigma} = -2(y - \hat{y})\hat{y}(1 - \hat{y})$$

$$\frac{\partial E}{\partial w_0} = \frac{\partial E}{\partial \Sigma}, \quad \frac{\partial E}{\partial w_1} = \frac{\partial E}{\partial \Sigma} x_1, \quad \frac{\partial E}{\partial w_2} = \frac{\partial E}{\partial \Sigma} x_2$$

Рисунок 27 - Расчёт локальных производных в ходе обратного распространения ошибки

Для ступенчатой активации градиентный подход неприменим (производная не существует в нуле и равна 0 почти всюду), поэтому используется правило перцептрона; вывод локальных производных далее относится к сигмоидальному случаю.

### Обучение элементарного перцептрана

После этого, для каждого обеих функции активации были произведены итерации обучения элементарного перцептрана на всех полученных выборках. Выборки изначально были разделены на train и test выборки (в отношении 70 на 30 соответственно) при помощи следующей функции:

```

def split_data(X: np.array, y: np.array, ratio: float = 0.30, random_state: int = 42):
    rng = np.random.default_rng(seed=random_state)
    idx = np.arange(X.shape[0])
    rng.shuffle(idx)
    left_share = int((1 - ratio) * X.shape[0])

    train_idx = idx[:left_share]
    test_idx = idx[left_share:]

    return train_idx, test_idx

```

Рисунок 28 - Функция разделения выборок

Все перцептроны учились с параметрами  $\text{learning\_rate} = 0.03$  и  $n_{\text{epochs}} = 1000$ . Рассмотрим полученные результаты обучения для функции активации “ступенька”:

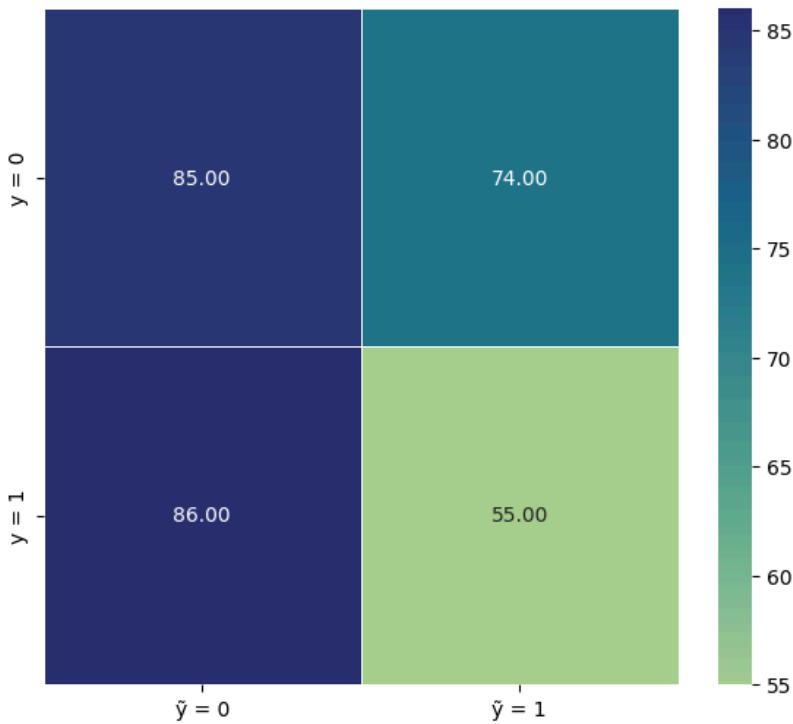


Рисунок 29 - Confusion Matrix для распределения концентрических окружностей

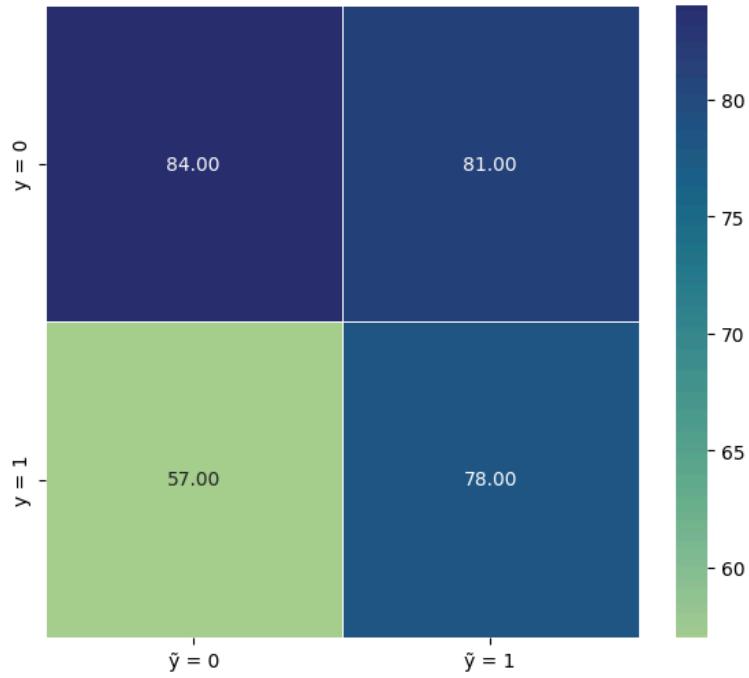


Рисунок 30 - Confusion Matrix для XOR распределения

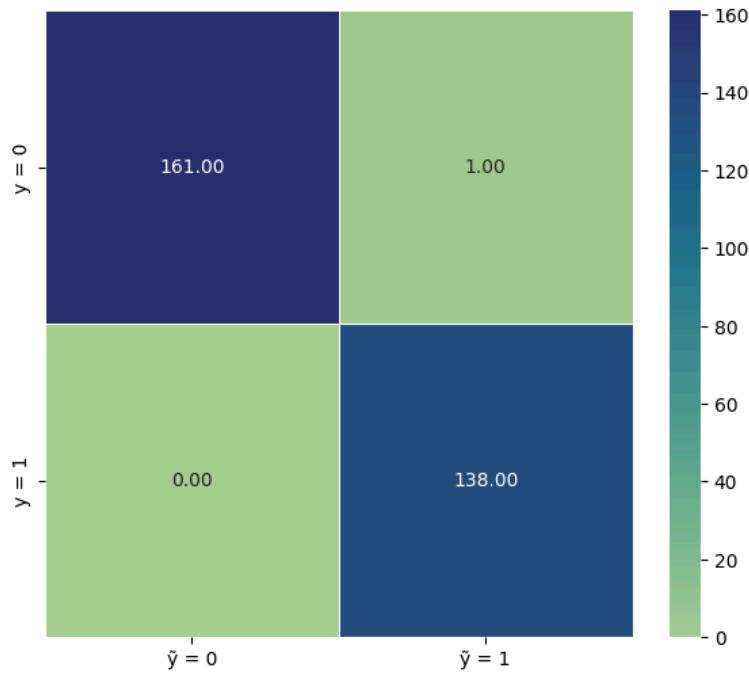


Рисунок 31 - Confusion Matrix для гауссовских кластеров

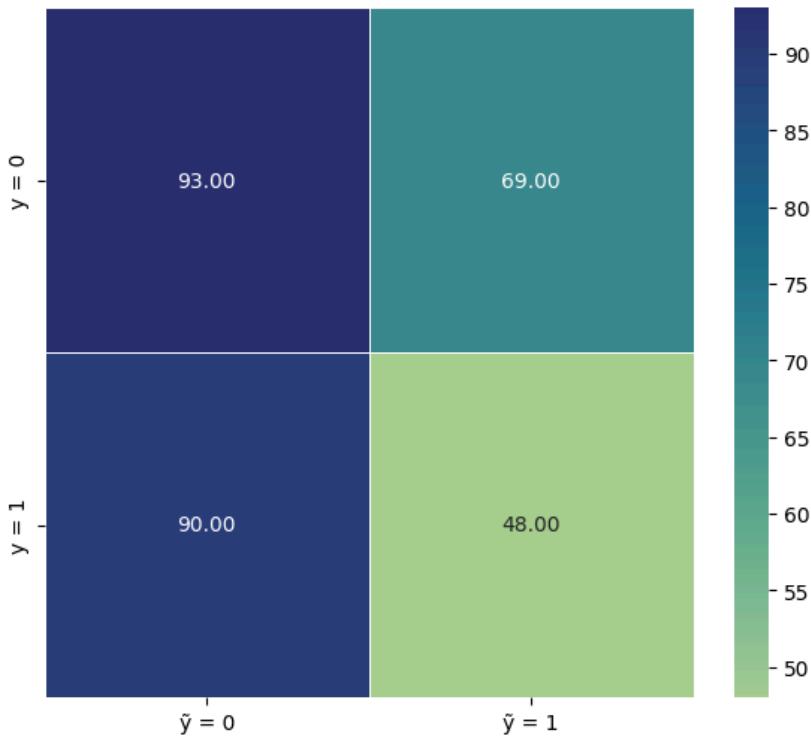


Рисунок 32 - Confusion Matrix для спиралей Архимеда

	Accuracy	Precision	Recall	F1-score
<b>spirals</b>	0.466667	0.426357	0.390071	0.407407
<b>XOR</b>	0.540000	0.490566	0.577778	0.530612
<b>Blobs</b>	0.996667	0.992806	1.000000	0.996390
<b>Spirals</b>	0.470000	0.410256	0.347826	0.376471

Рисунок 33 - Classification report для функции активации “Ступенька”

Теперь рассмотрим результаты обучения для функции активации sigmoid:

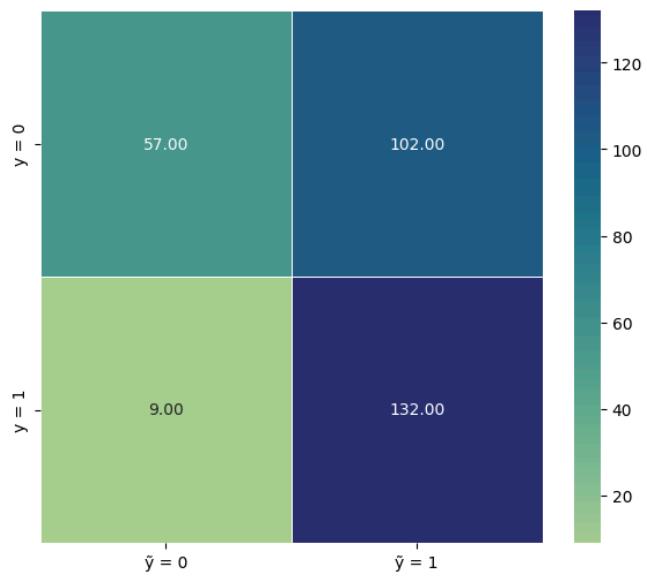


Рисунок 34 - Confusion Matrix для распределения концентрических окружностей

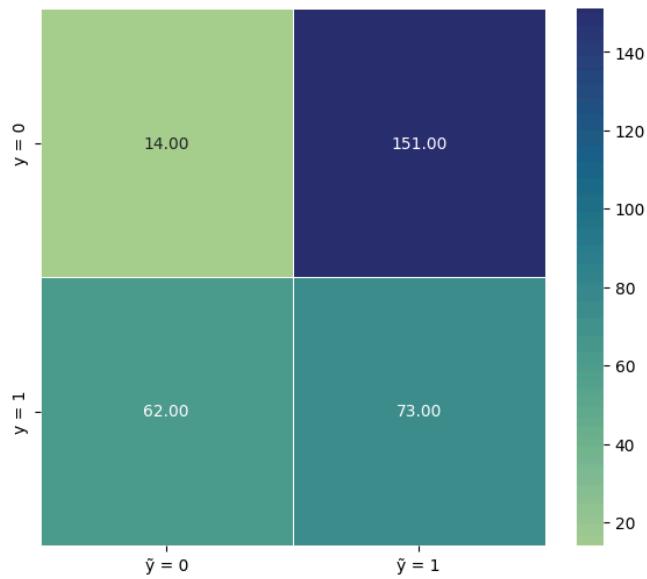


Рисунок 35 - Confusion Matrix для XOR распределения

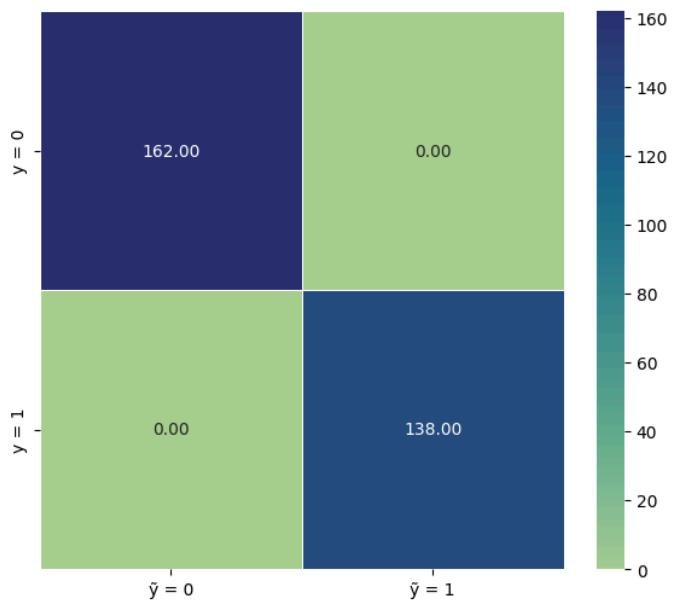


Рисунок 36 - Confusion Matrix для гауссовских кластеров

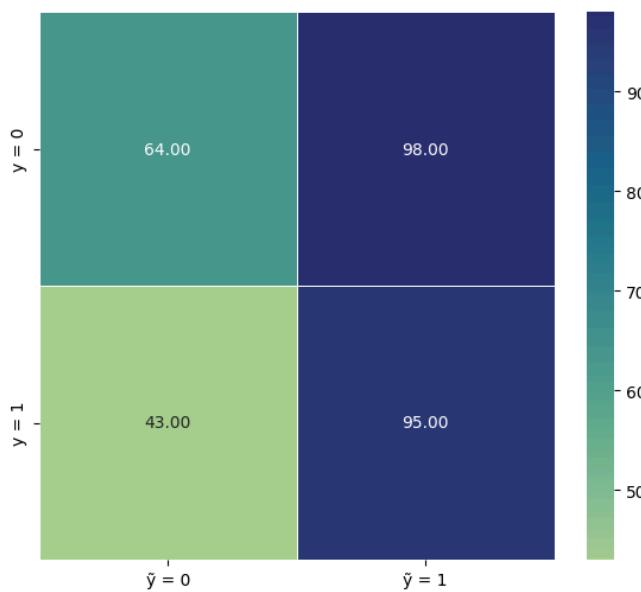


Рисунок 37 - Confusion Matrix для спиралей Архимеда

	Accuracy	Precision	Recall	F1-score
<b>spirals</b>	0.63	0.564103	0.936170	0.704000
<b>XOR</b>	0.29	0.325893	0.540741	0.406685
<b>Blobs</b>	1.00	1.000000	1.000000	1.000000
<b>Spirals</b>	0.53	0.492228	0.688406	0.574018

Рисунок 38 - Classification report для функции активации “Сигмоида”

Экспериментальные результаты (confusion matrix и classification report) показывают, что перцептрон хорошо работает на задачах, близких к линейно разделимым (гауссовские кластеры при достаточном разнесении центров): качество близко к идеальному. В то же время на нелинейно разделимых распределениях (XOR, концентрические окружности, спирали Архимеда) однослойная линейная модель принципиально ограничена: она строит лишь одну разделяющую гиперплоскость, поэтому качество заметно ниже и зависит от конкретного распределения и шума.

Также видно, что сигмоидальная версия часто ведет себя стабильнее в плане оптимизации (за счет непрерывности и наличия градиента), однако сама по себе сигмоида не “чинит” нелинейную разделимость: для задач типа XOR/спиралей нужна либо нелинейная инженерия признаков, либо многослойная сеть.

## Выводы

В первой части были реализованы генераторы четырёх типов данных (концентрические окружности, XOR, гауссовские кластеры, спирали Архимеда) и на TensorFlow Playground подобраны параметры нейросети, при которых наблюдается переобучение: малая ошибка на train при заметно большей ошибке на test. Это подтвердило ключевую идею: рост сложности модели и уменьшение доли обучающей выборки/рост шума повышают риск того, что модель начнет запоминать обучающие точки вместо извлечения устойчивых закономерностей.

Во второй части показано, что элементарный перцептрон - это линейный классификатор, поэтому он демонстрирует высокое качество на данных, которые линейно разделимы (или близки к этому), но системно проигрывает на нелинейных распределениях (окружности, XOR, спирали). Таким образом, результаты обеих частей согласуются:

- сложность данных (нелинейность границы + шум) требует более выразительных моделей;
- сложность модели без контроля (регуляризация/валидация/достаточный объем данных) легко приводит к переобучению;

- качество обобщения определяется балансом между выразительностью модели, уровнем шума и объемом обучающей выборки.