Н И Т У « М И С и С »

ОТЧЁТ

ПО

ЛАБОРАТОРНОЙ РАБОТЕ

«Многослойный персептрон»

Учебная дисциплина «Нейронные сети»

Группа: БПМ-16-2

Студент: Новицкий Дмитрий

Преподаватель: доц., к.т.н. Курочкин И.И.

Отметка:

Дата защиты:

2019 г.

Оглавление

Постановка задачи

Основное условие

1. Реализовать многослойный перцептрон для решения задач классификации. Для реализации можно использовать любые библиотеки и инструменты, которые позволяют выполнить условия данной лабораторной работы.
2. Количество слоев (полноценных нейронов): от 1 до 5.
3. Количество нейронов в каждом слое от 1 до 200.
4. Предусмотреть единичный вход для нейронов.
5. Функции активации могут быть различными на каждом слое.
6. Кодирование ответа в выходном слое (для задачи классификации для n классов): 3 типа (k=1, k<>n, n).
7. Разделение множества на 3 части (обучающее, валидационное и тестовое) должно происходить случайным образом.
8. Результаты задачи классификации даются на основе средней оценки (>10 запусков).

Результаты и визуализация

1. Реализовать визуализацию результатов для 2-хмерных данных с визуализацией границ разделения различных классов и истинной принадлежности точек классам. (К примеру, при разделении точек трех линейно неразделимых классов однослойным перцептроном, для точек использовать маркеры разной формы и/или разного цвета + залить области принадлежности разных классов с точки зрения перцептрона).
2. Реализовать визуализацию динамики ошибки для обучающего и валидационного множеств в процессе обучения.
3. Качество классификации с помощью: Accuracy, precision, recall, СКО (среднеквадартичной ошибки).

Демонстрация работы

Продемонстрировать работу нескольких сценариев:

1) По количеству классов: малое (2-3), среднее (7-10), большое (~30).
2) По разделимости классов: линейно разделимое множество, линейно неразделимое множество (средняя площадь пересечения классов 10-20%), линейно неразделимое множество (средняя площадь пересечения классов 50-70%).
3) По качеству обучения: переобучение, недообучение, паралич сети.
4) На эталонных множествах с количеством признаков более 5.
5) По кодированию ответа в выходном слое.

Входные/выходные данные

1. Входные данные (датасеты) в виде текстового файла. (к примеру, экспорт таблицы из Excel в формате TXT или CSV).

2. Выходные данные (результаты) сохраняются в виде, необходимом для формирования отчета PDF или DOCX.

Описание работы программы

В данной версии многослойного персептрона не были использованы сторонние библиотеки, позволяющие создать нейронную сеть «из коробки». Был написан класс, реализующий алгоритм многослойного персептрона и имеющей несколько методов для более удобной его реализации. Рассмотрим подробнее методы класса «perceptrone».

- Конструктор класса. Данный конструктор принимает и инициализирует следующие параметры: путь к датасету, количество слоёв в датасете, количество процентов от исходного датасета, которое определится для тестовой выборки, количество процентов от исходного датасета, которое определится для валидационной выборки, функция активации (либо общая для всех слоёв, либо своя функция активации для каждого слоя), количество нейронов в скрытых слоях, количество эпох обучения, скорость обучения.

```
    def __init__(self, dataset_path, count_of_layers, test_probability,
validation_probability, activation_function, neurons_hidden, count_of_eras, speed):
        self.read_from_file(dataset_path)

        self.count_of_layers = count_of_layers
        if(count_of_layers != len(neurons_hidden) + 2):
            print("Неверно определено количество слоёв и количество нейронов в слоях")
            exit()
        self.count_of_neurons = neurons_hidden
        self.count_of_eras = count_of_eras

        if(type(activation_function) == str):
            for i in range(count_of_layers):
                self.activation_function.append(activation_function)
        elif(type(activation_function) == list):
            if(len(activation_function) == count_of_layers):
                self.activation_function = activation_function
            else:
                print("Количество значений в функции активации не соответствует
количеству слоёв")
                exit()
        else:
            print("Неопознанная функция активации")
            exit()

        self.test_probability = test_probability
        self.validation_probability = validation_probability

        self.speed = speed
```

- Метод для конвертации значений датасета в тип данных float для удобства работы с данными. Кроме того, метод позволяет заменить строковые типы данных для классов датасета на значения типа int. В таком случае создаётся словарь dict, хранящий исходные значения классов и соответствующие им изменённые целые значения классов.

```
    def converting(self):
        # Конвертируем признаки датасета
        try:
            for i in range(len(self.file_matrix)):
                for j in range(len(self.file_matrix[0]) - 1):
```

```
                    self.file_matrix[i][j] = float(self.file_matrix[i][j])
        except:
            print("Плохие значения в датасете!")
            exit()

        # Конвертируем классы датасета
        try:
            for i in range(len(self.file_matrix)):
                self.file_matrix[i][len(self.file_matrix[0]) - 1] =
float(self.file_matrix[i][len(self.file_matrix[0]) - 1])
        except ValueError:
            dict = {}
            counter = 0
            for i in range(len(self.file_matrix)):
                try:
                    self.file_matrix[i][len(self.file_matrix[0]) - 1] =
dict[self.file_matrix[i][len(self.file_matrix[0]) - 1]]
                except KeyError:
                    dict[self.file_matrix[i][len(self.file_matrix[0]) - 1]] = counter
                    self.file_matrix[i][len(self.file_matrix[0]) - 1] = counter
                    counter = counter + 1
            print("Словарь для классов следующий:")
            print(dict)
```

- Метод для отрисовки массива точек на графике. Данный метод необходим для отрисовки двумерных данных для визуализации.

```
    def draw_points(points_matrix):
        fig = plt.figure()

        x1 = []
        y1 = []
        x2 = []
        y2 = []
        for i in range(len(points_matrix)):
            if(points_matrix[2] == 0):
                x1.append(points_matrix[i][0])
                y1.append(points_matrix[i][1])
            if(points_matrix[2] == 1):
                x2.append(points_matrix[i][0])
                y2.append(points_matrix[i][1])

        figure_points = fig.add_subplot(111)
        figure_points.set_title("Точки для двумерных исходных данных")
        figure_points.scatter(x1, y1, color = 'green')
        figure_points.scatter(x2, y2, color = 'blue')

        plt.show()
```

- Метод для отрисовки графиков. Данный метод необходим для отрисовки графиков зависимости среднеквадратичной ошибки для обучающего и тестового множеств от эпохи. Кроме того, метод необходим для отрисовки графиков зависимость метрики accuracy для обучающего и тестового множеств от эпохи.

```
    def graph_drawing_function(self):
        # Массив для значений x
        helper_mas = []
        for i in range(len(self.MSE_mas_learning)):
            helper_mas.append(i)

        fig = plt.figure()
```

```
        MSE_graph = fig.add_subplot(1, 2, 1)
        MSE_graph.set_title("green - MSE for learning, blue - MSE for validation")
        MSE_graph.scatter(helper_mas, self.MSE_mas_learning, color = 'green', marker =
'*')
        MSE_graph.scatter(helper_mas, self.MSE_mas_test, color = 'blue', marker = '*')
        MSE_graph.plot(helper_mas, self.MSE_mas_learning, color = 'green')
        MSE_graph.plot(helper_mas, self.MSE_mas_test, color = 'blue')

        accuracy_graph = fig.add_subplot(1, 2, 2)
        accuracy_graph.set_title("green - accuracy for learning, blue - accuracy for
validation")
        accuracy_graph.scatter(helper_mas, self.accuracy_mas_learning, color = 'green',
marker = '*')
        accuracy_graph.scatter(helper_mas, self.accuracy_mas_test, color = 'blue', marker
= '*')
        accuracy_graph.plot(helper_mas, self.accuracy_mas_learning, color = 'green')
        accuracy_graph.plot(helper_mas, self.accuracy_mas_test, color = 'blue')

        plt.show()
```

- Метод для считывания датасета с файла любого формата.

```
def read_from_file(self, path):
    file_reader = open(path)
    all_file = file_reader.read()
    file_mas = []
    file_mas = all_file.split("\n")
    for i in file_mas:
        self.file_matrix.append(i.split(","))
```

- Метод, реализующий добавление единичного входа для нейронов.

```
def single_inputs_for_neurons(self):
    for i in range(len(self.learn_inputs)):
        self.learn_inputs[i].append(1)
    for i in range(len(self.test_inputs)):
        self.test_inputs[i].append(1)
    for i in range(len(self.validation_inputs)):
        self.validation_inputs[i].append(1)
```

- Метод, позволяющий определить класс на основе выходного значения обучающей выборки.

```
def choose_sign(output, signs):
    min = abs(signs[0] - output)
    i_min = 0
    for i in range(len(signs)):
        if(abs(signs[i] - output) < min):
            min = abs(signs[i] - output)
            i_min = i
    class_output = signs[i_min]
    return class_output
```

- Метод для определения количества классов и признаков в исходном датасете. Инициализация значений количества нейронов в каждом слое, инициализация значений нейронов в каждом слое.

```
def get_classes(self):
    self.count_of_signs = len(self.file_matrix[0]) - 1
    print("Количество признаков равно ", self.count_of_signs)

    self.classes.append(self.file_matrix[0][len(self.file_matrix[0]) - 1])
```

```
        for i in range(len(self.file_matrix)):
            find = 0
            for j in range(len(self.classes)):
                if(self.classes[j] == self.file_matrix[i][len(self.file_matrix[i]) - 1]):
                    find = 1
            if(find == 0):
                self.classes.append(self.file_matrix[i][len(self.file_matrix[i]) - 1])

        self.count_of_classes = len(self.classes)
        print("Количество классов равно", self.count_of_classes)
        print(self.classes)

        print("Количество нейронов в каждом слое.")
        self.count_of_neurons.insert(0, self.count_of_signs)
        self.count_of_neurons.append(self.count_of_classes)

        for i in range(self.count_of_layers):
            print("Слой № ", i + 1, ". Количество нейронов - ", self.count_of_neurons[i])

        # Количество нейронов в первом слое равно количеству признаков
        self.count_of_neurons[0] = self.count_of_signs
        # Количество нейронов в последнем слое равно количеству классов
        self.count_of_neurons[len(self.count_of_neurons) - 1] = self.count_of_classes

        for i in range(self.count_of_layers):
            helper_mas = []
            for j in range(self.count_of_neurons[i]):
                helper_mas.append(0)
            self.neurons.append(copy.deepcopy(helper_mas))
            self.delta_neurons.append(copy.deepcopy(helper_mas))
            self.sum.append(copy.deepcopy(helper_mas))
```

- Метод для генерации случайных значений для весов.

```
    def get_random_synaptic_weights(self):
        np.random.seed(1)

        for k in range(self.count_of_layers - 1):
            helper_mas_1 = []

            for i in range(self.count_of_neurons[k]):
                helper_mas_2 = []
                for j in range(self.count_of_neurons[k + 1]):
                    helper_mas_2.append(2 * np.random.random() - 1)
                helper_mas_1.append(helper_mas_2)
            self.synaptic_weights.append(helper_mas_1)
```

- Метод для разделения исходного датасета на обучающее, тестовое и валидационное множества.

```
    def devide_learning_test_validation(self):
        for i in range(len(self.file_matrix)):
            random_number = random.randint(1, 100)
            if(random_number > self.test_probability + self.validation_probability):
                self.learn_inputs.append(self.file_matrix[i])
                self.learn_outputs.append(self.file_matrix[i][len(self.file_matrix[i]) -
1])
                self.learn_inputs[len(self.learn_inputs) - 1].pop()
            elif(random_number <= self.validation_probability):
                self.validation_inputs.append(self.file_matrix[i])

self.validation_outputs.append(self.file_matrix[i][len(self.file_matrix[i]) - 1])
                self.validation_inputs[len(self.validation_inputs) - 1].pop()
            else:
```

```
                self.test_inputs.append(self.file_matrix[i])
                self.test_outputs.append(self.file_matrix[i][len(self.file_matrix[i]) -
1])
                self.test_inputs[len(self.test_inputs) - 1].pop()

        print("Размер обучающей выборки равен:", len(self.learn_inputs))
        print("Это составляет", round((len(self.learn_inputs) / len(self.file_matrix) *
100), 2), "% от исходной выборки")
        print("Размер тестовой выборки равен:", len(self.test_inputs))
        print("Это составляет", round((len(self.test_inputs) / len(self.file_matrix) *
100), 2), "% от исходной выборки")
        print("Размер валидационной выборки равен:", len(self.validation_inputs))
        print("Это составляет", round((len(self.validation_inputs) /
len(self.file_matrix) * 100), 2), "% от исходной выборки")
        print()
```

- Метод, реализующий процедуру обучения нейронной сети. Работа алгоритма прекращается в тот момент, когда текущее значение эпохи достигает изначально заданного числа эпох.

```
    def learning_procedure(self):
        counter = 0 # Счётчик эпох

        # Процесс обучения
        while(counter < self.count_of_eras):
            print("Эпоха обучения №", counter + 1)
            print("Сейчас идёт обучающая выборка")
            self.learning_function()
            print("Сейчас идёт тестовая выборка")
            self.test_function()
            counter = counter + 1
            if((self.min_MSE == -1) or (self.MSE_mas_test[len(self.MSE_mas_test) - 1] <
self.min_MSE)):
                self.min_MSE = self.MSE_mas_test[len(self.MSE_mas_test) - 1]
                self.min_MSE_era = counter
                self.best_synaptic_weights = self.synaptic_weights

        # Рисуем графики для MSE
        self.graph_drawing_function()

        # Проверка работы нейронной сети на валидационной выборке
        self.validation_function()
```

- Метод для обучения нейронной сети на обучающей выборке.

```
    def learning_function(self):
        # MSE - Mean Squared Error (среднеквадратичная ошибка)
        MSE = 0

        TP = 0 # True Positive (Правильно определена 1)
        FP = 0 # False Positive (Неправильно определена 1)
        FN = 0 # False Negative (Неправильно определён 0)
        TN = 0 # True Negative (Правильно определён 0)

        for i in range(len(self.learn_inputs)):
            # Высчитываем значения нейронов
            for j in range(self.count_of_layers):
                if(j == 0):
                    for k in range(self.count_of_signs):
                        self.neurons[0][k] = self.learn_inputs[i][k]
                else:
                    for k in range(self.count_of_neurons[j]):
                        self.sum[j][k] = self.scalar_sum_for_neuron(j, k)
```

```
                        self.neurons[j][k] =
self.choose_activation_function(self.activation_function[j], self.sum[j][k])

            # Считаем ошибку
            value_layer = self.count_of_layers - 1
            while(value_layer >= 1):
                if(value_layer == self.count_of_layers - 1):
                    for j in range(self.count_of_neurons[len(self.count_of_neurons) -
1]):
                        if(j == self.learn_outputs[i]):
                            self.delta_neurons[value_layer][j] = 1 -
self.neurons[value_layer][j]
                        else:
                            self.delta_neurons[value_layer][j] = -
self.neurons[value_layer][j]
                        MSE = MSE + (copy.deepcopy(self.delta_neurons[value_layer][j]) **
2)
                else:
                    for j in range(self.count_of_neurons[value_layer]):
                        self.delta_neurons[value_layer][j] =
self.scalar(self.delta_neurons[value_layer + 1], self.synaptic_weights[value_layer][j])
                value_layer = value_layer - 1

            for j in range(self.count_of_neurons[len(self.count_of_neurons) - 1]):
                value = round(self.neurons[self.count_of_layers - 1][j])
                if((value == 0) and (j != self.learn_outputs[i])):
                    TN = TN + 1
                if((value == 0) and (j == self.learn_outputs[i])):
                    FN = FN + 1
                if((value == 1) and (j == self.learn_outputs[i])):
                    TP = TP + 1
                if((value == 1) and (j != self.learn_outputs[i])):
                    FP = FP + 1

            # Изменяем веса
            for j in range(self.count_of_layers - 1):
                for k in range(len(self.synaptic_weights[j])):
                    for l in range(len(self.synaptic_weights[j][k])):
                        self.synaptic_weights[j][k][l] = self.synaptic_weights[j][k][l] +
self.speed * self.delta_neurons[j + 1][l] *
self.choose_derivative_activation_function(self.activation_function[j + 1], self.sum[j +
1][l]) * self.neurons[j][k]

        MSE = MSE / len(self.learn_inputs)
        self.MSE_mas_learning.append(MSE)
        print("MSE =", MSE)
        print("TP = ", TP)
        print("FP = ", FP)
        print("FN = ", FN)
        print("TN = ", TN)
        # Точность работы алгоритма
        self.accuracy_mas_learning.append((TP + TN) / (TP + TN + FP + FN))
        print("accurancy = ", (TP + TN) / (TP + TN + FP + FN))
        print("precision = ", TP / (TP + FP))
        print("recall = ", TP / (TP + FN))
```

- Метод для обучения нейронной сети на тестовой выборке. В данном методе синаптические веса не изменяются.

```
    def test_function(self):
        # MSE - Mean Squared Error (среднеквадратичная ошибка)
        MSE = 0

        TP = 0 # True Positive (Правильно определена 1)
```

```
        FP = 0 # False Positive (Неправильно определена 1)
        FN = 0 # False Negative (Неправильно определён 0)
        TN = 0 # True Negative (Правильно определён 0)

        for i in range(len(self.test_inputs)):
            # Высчитываем значения нейронов
            for j in range(self.count_of_layers):
                if(j == 0):
                    for k in range(self.count_of_signs):
                        self.neurons[0][k] = self.test_inputs[i][k]
                else:
                    for k in range(self.count_of_neurons[j]):
                        self.sum[j][k] = self.scalar_sum_for_neuron(j, k)
                        self.neurons[j][k] =
self.choose_activation_function(self.activation_function[j], self.sum[j][k])

            # Считаем ошибку
            value_layer = self.count_of_layers - 1
            while(value_layer >= 1):
                if(value_layer == self.count_of_layers - 1):
                    for j in range(self.count_of_neurons[len(self.count_of_neurons) -
1]):
                        if(j == self.test_outputs[i]):
                            self.delta_neurons[value_layer][j] = 1 -
self.neurons[value_layer][j]
                        else:
                            self.delta_neurons[value_layer][j] = -
self.neurons[value_layer][j]
                        MSE = MSE + (self.delta_neurons[value_layer][j] ** 2)
                else:
                    for j in range(self.count_of_neurons[value_layer]):
                        self.delta_neurons[value_layer][j] =
self.scalar(self.delta_neurons[value_layer + 1], self.synaptic_weights[value_layer][j])
                value_layer = value_layer - 1

            for j in range(self.count_of_neurons[len(self.count_of_neurons) - 1]):
                value = round(self.neurons[self.count_of_layers - 1][j])
                if((value == 0) and (j != self.test_outputs[i])):
                    TN = TN + 1
                if((value == 0) and (j == self.test_outputs[i])):
                    FN = FN + 1
                if((value == 1) and (j == self.test_outputs[i])):
                    TP = TP + 1
                if((value == 1) and (j != self.test_outputs[i])):
                    FP = FP + 1

        MSE = MSE / len(self.test_inputs)
        self.MSE_mas_test.append(MSE)
        print("MSE =", MSE)
        print("TP = ", TP)
        print("FP = ", FP)
        print("FN = ", FN)
        print("TN = ", TN)
        # Точность работы алгоритма
        self.accuracy_mas_test.append((TP + TN) / (TP + TN + FP + FN))
        print("accurancy = ", (TP + TN) / (TP + TN + FP + FN))
        print("precision = ", TP / (TP + FP))
        print("recall = ", TP / (TP + FN))
```

- Метод для проверки обученной нейронной сети на валидационном множестве.

```
    def validation_function(self):
        # MSE - Mean Squared Error (среднеквадратичная ошибка)
        MSE = 0
```

```
        TP = 0 # True Positive (Правильно определена 1)
        FP = 0 # False Positive (Неправильно определена 1)
        FN = 0 # False Negative (Неправильно определён 0)
        TN = 0 # True Negative (Правильно определён 0)

        for i in range(len(self.validation_inputs)):
            # Высчитываем значения нейронов
            for j in range(self.count_of_layers):
                if(j == 0):
                    for k in range(self.count_of_signs):
                        self.neurons[0][k] = self.validation_inputs[i][k]
                else:
                    for k in range(self.count_of_neurons[j]):
                        self.sum[j][k] = self.scalar_sum_for_neuron_validation(j, k)
                        self.neurons[j][k] =
self.choose_activation_function(self.activation_function[j], self.sum[j][k])

            # Считаем ошибку
            value_layer = self.count_of_layers - 1
            while(value_layer >= 1):
                if(value_layer == self.count_of_layers - 1):
                    for j in range(self.count_of_neurons[len(self.count_of_neurons) -
1]):
                        if(j == self.validation_outputs[i]):
                            self.delta_neurons[value_layer][j] = 1 -
self.neurons[value_layer][j]
                        else:
                            self.delta_neurons[value_layer][j] = -
self.neurons[value_layer][j]
                        MSE = MSE + (self.delta_neurons[value_layer][j] ** 2)
                else:
                    for j in range(self.count_of_neurons[value_layer]):
                        self.delta_neurons[value_layer][j] =
self.scalar(self.delta_neurons[value_layer + 1],
self.best_synaptic_weights[value_layer][j])
                value_layer = value_layer - 1

            for j in range(self.count_of_neurons[len(self.count_of_neurons) - 1]):
                value = round(self.neurons[self.count_of_layers - 1][j])
                if((value == 0) and (j != self.validation_outputs[i])):
                    TN = TN + 1
                if((value == 0) and (j == self.validation_outputs[i])):
                    FN = FN + 1
                if((value == 1) and (j == self.validation_outputs[i])):
                    TP = TP + 1
                if((value == 1) and (j != self.validation_outputs[i])):
                    FP = FP + 1

        MSE = MSE / len(self.validation_inputs)
        self.MSE_validation = MSE
        print("MSE =", MSE)
        print("TP = ", TP)
        print("FP = ", FP)
        print("FN = ", FN)
        print("TN = ", TN)
        # Точность работы алгоритма
        self.accuracy_validation = (TP + TN) / (TP + TN + FP + FN)
        print("accurancy = ", (TP + TN) / (TP + TN + FP + FN))
        print("precision = ", TP / (TP + FP))
        print("recall = ", TP / (TP + FN))
```

- Метод, позволяющий проверить исходный датасет на корректность и наличие ошибок.

```
    def check_dataset(self):
        for i in range(len(self.file_matrix)):
```

```
            #print("len(self.file_matrix[i]) = ", len(self.file_matrix[i]))
            if(len(self.file_matrix[0]) != len(self.file_matrix[i])):
                print("Датасет испорчен. Количество столбцов различное")
                exit(0)

        print("Количество строк в исходном датасете: ", len(self.file_matrix))
        print("Количество столбцов в исходном датасете: ", len(self.file_matrix[0]))
        print()
```

- Метод для нормализации входных и выходных данных заданного типа (линейная или нелинейная нормализация).

```
    def normalization(self, type_input_normalization, type_output_normalization):
        if(type_input_normalization == "linear"):
            self.learn_inputs = self.linear_matrix_normalization(self.learn_inputs)
            self.test_inputs = self.linear_matrix_normalization(self.test_inputs)
            self.validation_inputs =
self.linear_matrix_normalization(self.validation_inputs)
        elif(type_input_normalization == "not_linear"):
            self.learn_inputs = self.not_linear_matrix_normalization(self.learn_inputs)
            self.test_inputs = self.not_linear_matrix_normalization(self.test_inputs)
            self.validation_inputs =
self.not_linear_matrix_normalization(self.validation_inputs)
        if(type_output_normalization == "linear"):
            self.learn_outputs = self.linear_mas_normalization(self.learn_outputs)
            self.test_outputs = self.linear_mas_normalization(self.test_outputs)
            self.validation_outputs =
self.linear_mas_normalization(self.validation_outputs)
        elif(type_output_normalization == "not_linear"):
            self.learn_outputs = self.not_linear_mas_normalization(self.learn_outputs)
            self.test_outputs = self.not_linear_mas_normalization(self.test_outputs)
            self.validation_outputs =
self.not_linear_mas_normalization(self.validation_outputs)
```

- Метод, реализующий линейную нормализацию массива.

```
    def linear_mas_normalization(self, file_mas):
        max = file_mas[0]
        min = file_mas[0]
        for i in range(len(file_mas)):
            if(file_mas[i] > max):
                max = file_mas[i]
            if(file_mas[i] < min):
                min = file_mas[i]
        for i in range(len(file_mas)):
            file_mas[i] = (file_mas[i] - min) / (max - min)
        return file_mas
```

- Метод, реализующий линейную нормализацию матрицы.

```
    def linear_matrix_normalization(self, file_matrix):
        for i in range(len(file_matrix[0])):
            max = file_matrix[0][i]
            min = file_matrix[0][i]
            for j in range(len(file_matrix)):
                if(file_matrix[j][i] > max):
                    max = file_matrix[j][i]
                if(file_matrix[j][i] < min):
                    min = file_matrix[j][i]
            for j in range(len(file_matrix)):
                file_matrix[j][i] = (file_matrix[j][i] - min) / (max - min)
        return file_matrix
```

- Метод, реализующий нелинейную нормализацию массива.

```
    def not_linear_mas_normalization(self, file_mas):
        a = 0.5
        average = 0
        for i in range(len(file_mas)):
            average = average + file_mas[i]
        average = average / len(file_mas)
        for i in range(len(file_matrix)):
            file_matrix[i] = 1 / (np.exp((-1) * a * (file_mas[i] - average)) + 1)
        return file_mas
```

- Метод, реализующий нелинейную нормализацию матрицы.

```
    def not_linear_matrix_normalization(self, file_matrix):
        a = 0.5 # Коэффициент нормализации
        for i in range(len(file_matrix[0])):
            average = 0
            for j in range(len(file_matrix)):
                average = average + file_matrix[j][i]
            average = average / len(file_matrix)
            for j in range(len(file_matrix)):
                file_matrix[j][i] = 1 / (np.exp((-1) * a * (file_matrix[j][i] - average))
+ 1)
        return file_matrix
```

- Метод для скалярного произведения двух векторов. Данный метод необходим для подсчёта значения, перед применением функции активации для нейрона.

```
    def scalar(self, mas_1, mas_2):
        result_mas = 0
        if((len(mas_1) == 0) | (len(mas_2) == 0)):
            print("Один из массивов пуст! Перемножать нечего!")
        elif(type(mas_1) != type(mas_2)):
            print("Массивы разных типов! Нельзя найти их скалярное произведение!")
            print("Тип массива 1 = ", type(mas_1))
            print("Тип массива 2 = ", type(mas_2))
        elif(len(mas_1) != len(mas_2)):
            print("Массивы разных размеров. Нельзя найти их скалярное произведение!")
        else:
            for i in range(len(mas_1)):
                result_mas += mas_1[i] * mas_2[i]
            return result_mas
```

- Метод для скалярного произведения двух векторов. Данный метод необходим для нахождения скалярного произведения для нейрона при обучающей и тестовой выборках.

```
    def scalar_sum_for_neuron(self, value_layer, position_weight):
        sum = 0
        for i in range(self.count_of_neurons[value_layer - 1]):
            sum = sum + self.synaptic_weights[value_layer - 1][i][position_weight] *
self.neurons[value_layer - 1][i]
        return sum
```

- Метод для скалярного произведения двух векторов. Данный метод необходим для нахождения скалярного произведения для нейрона при валидационной выборке.

```
    def scalar_sum_for_neuron_validation(self, value_layer, position_weight):
        sum = 0
        for i in range(self.count_of_neurons[value_layer - 1]):
            sum = sum + self.best_synaptic_weights[value_layer -
1][i][position_weight] * self.neurons[value_layer - 1][i]
        return sum
```

- Метод для выбора заданной функции активации.

```
def choose_activation_function(self, value_activation_function, sum):
    if(value_activation_function == "sigmoid"):
        return self.sigmoid_activation_function(sum)
    if(value_activation_function == "softsign"):
        return self.softsign_activation_function(sum)
```

- Метод для выбора заданной производной от функции активации.

```
def choose_derivative_activation_function(self, value_activation_function, sum):
    if(value_activation_function == "sigmoid"):
        return self.derivative_sigmoid_activation_function(sum)
    if(value_activation_function == "softsign"):
        return self.derivative_softsign_activation_function(sum)
```

- Реализация функции активации «softsign».

```
def softsign_activation_function(self, x):
    return (x / (1 + np.abs(x)))
```

- Реализация производной от функции активации «softsign».

```
def derivative_softsign_activation_function(self, x):
    if(x >= 0):
        return (1 / ((1 + x) * (1 + x)))
    else:
        return (1 / ((1 - x) * (1 - x)))
```

- Реализация функции активации «sigmoid».

```
def sigmoid_activation_function(self, x):
    return (1 / (1 + np.exp(-x)))
```

- Реализация производной от функции активации «sigmoid».

```
def derivative_sigmoid_activation_function(self, x):
    return (np.exp(-x) / ((1 + np.exp(-x)) * (1 + np.exp(-x))))
```

Демонстрация работы программы

Проверим работу программы, меняя различные параметры многослойного персептрона. Будем идти от простого сложного. Поскольку в моём датасете достаточно много данных и признаков, то для начала используем более простой датасет. Возьмём датасет Рачеева Романа – «data_banknote_authentication». На вход для многослойного персептрона пададим следующие параметры:

- Общее количество слоёв – 4
- Количество нейронов в скрытых слоях – 8, 8
- Количество процентов из всего датасета для тестового множества – 20
- Количество процентов из всего датасета для валидационного множества – 10.
- Функция активации для каждого слоя – сигмоида.
- Количество эпох обучения – 150
- Скорость обучения – 0.1

Результаты получились следующие:

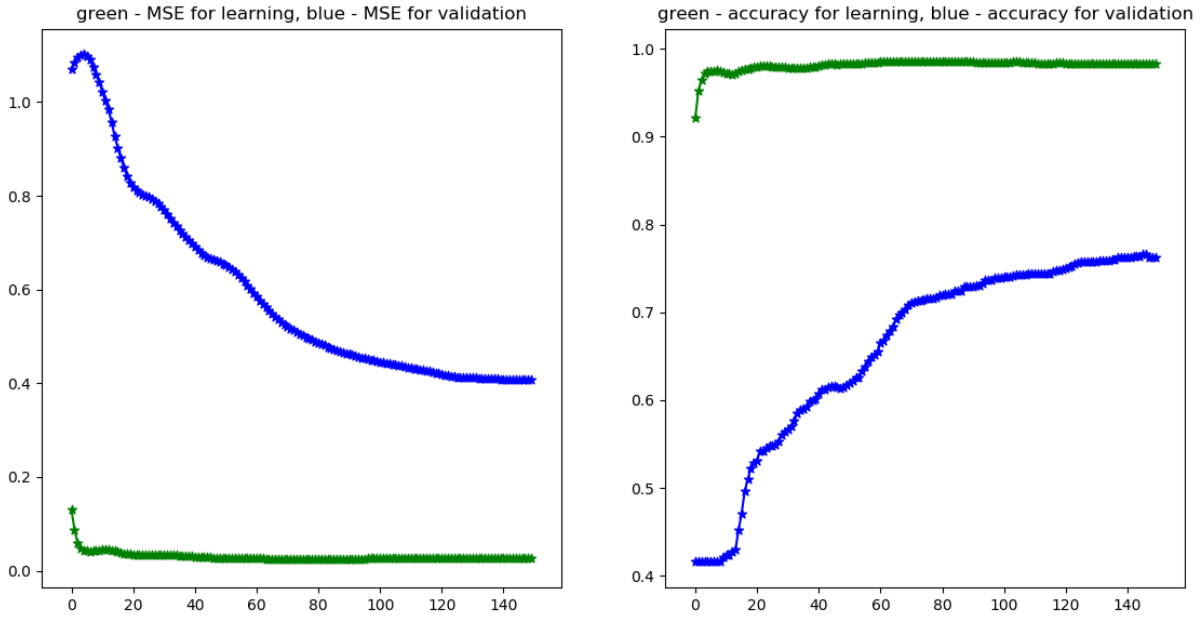


Рис. 1. Графики зависимости среднеквадратичной ошибки и accuracy для обучающей и тестовой выборок от эпохи.

```
MSE = 0.33581109380919655
TP =  229
FP =  52
FN =  52
TN =  229
accurancy =  0.8149466192170819
precision =  0.8149466192170819
recall =  0.8149466192170819
Для продолжения нажмите любую клавишу . . .
```

Рис. 2. Метрики для валидационной выборки.

Из данных значений можно сделать вывод, что нейронная сеть недообучилась, так как значение среднеквадратичной ошибки для тестового множества постепенно снижалась на протяжении всего периода обучения.

Попробуем увеличить скорость обучения нейронной сети с 0.1 до 0.5. Получим следующие результаты:

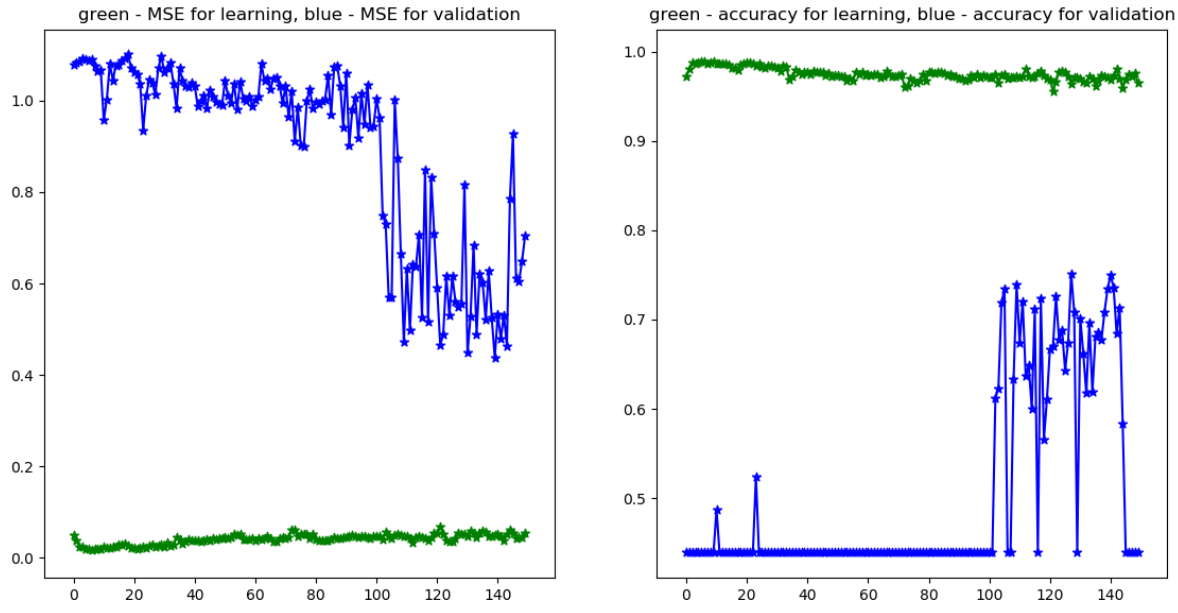


Рис. 3. Графики зависимости среднеквадратичной ошибки и accuracy для обучающей и тестовой выборок от эпохи.

```
MSE = 0.6803493237843182
TP =  105
FP =  148
FN =  148
TN =  105
accurancy =  0.4150197628458498
precision =  0.4150197628458498
recall =  0.4150197628458498
Для продолжения нажмите любую клавишу . . .
```

Рис. 4. Метрики для валидационной выборки.

Полученные результаты говорят о большой скорости обучения, из-за чего значение среднеквадратичной ошибки динамично меняются на протяжении обучения.

Оставим значение скорости обучения на уровне 0.1, но изменим количество нейронов в скрытых слоях. Поставим количество нейронов в скрытых слоях – 15, 15. Также в скрытых слоях поставим функцию активации – softsign. Результаты работы нейронной сети будут следующими:

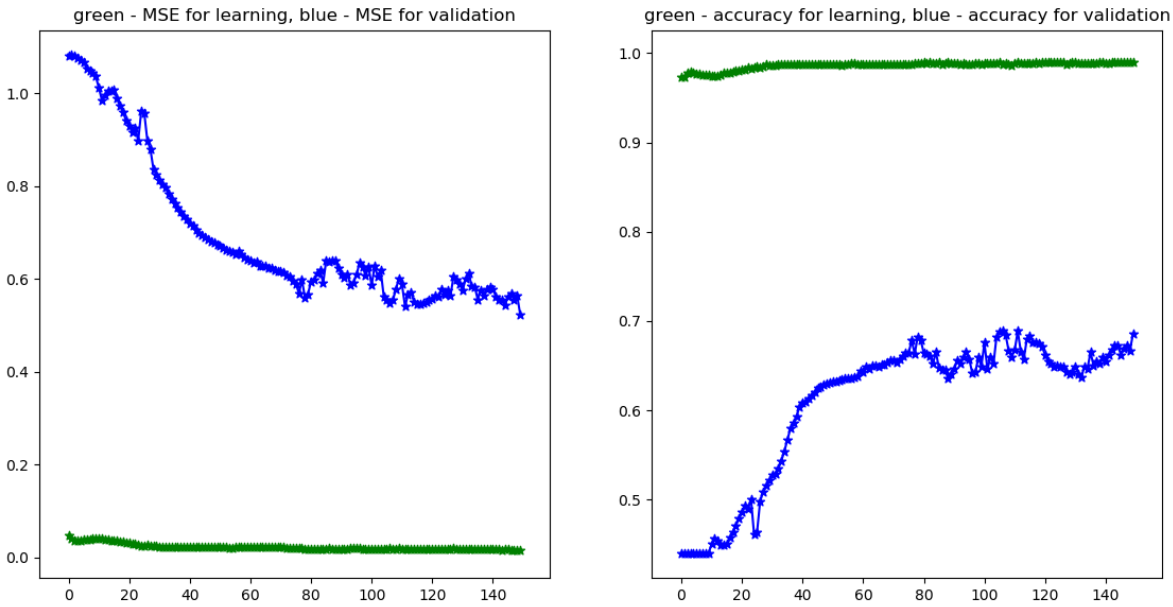


Рис. 5. Графики зависимости среднеквадратичной ошибки и accuracy для обучающей и тестовой выборок от эпохи.

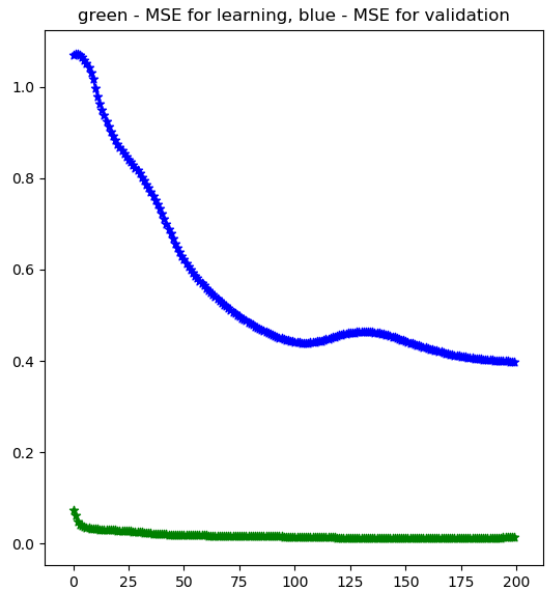
```
MSE = 0.5030120198242877
TP =  210
FP =  87
FN =  87
TN =  210
accurancy =  0.7070707070707071
precision =  0.7070707070707071
recall =  0.7070707070707071
Для продолжения нажмите любую клавишу . . .
```

Рис. 6. Метрики для валидационной выборки.

Изменим параметры следующим образом:

- Общее количество слоёв в персептроне – 4.
- Количество нейронов в скрытых слоях – 16, 16.
- Функция активации для каждого слоя – сигмоида.
- Количество эпох обучения – 200.
- Скорость обучения – 0.1

Получим следующие результаты:



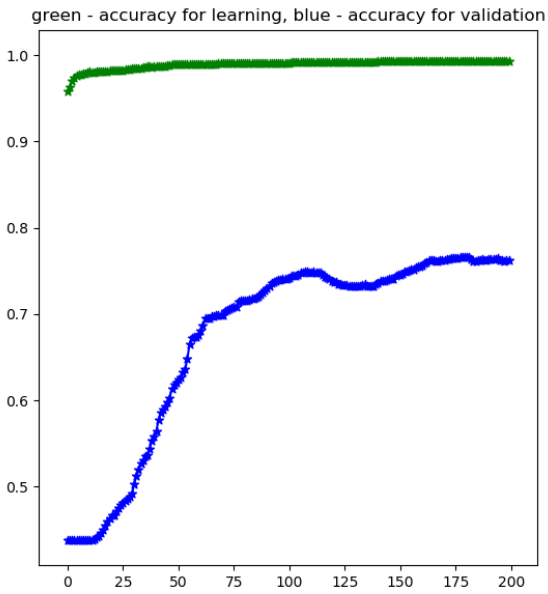


Рис. 7. Графики зависимости среднеквадратичной ошибки и accuracy для обучающей и тестовой выборок от эпохи.

```
MSE = 0.3068723383569957
TP =  252
FP =  53
FN =  56
TN =  255
accurancy =  0.823051948051948
precision =  0.8262295081967214
recall =  0.8181818181818182
Press any key to continue . . .
```

Рис. 8. Метрики для валидационной выборки.

Теперь проверим работу нейронной сети на моём датасете – «predictiong a pulsar star». Зададим следующие параметры нейронной сети:

- Общее количество слоёв в нейронной сети – 4
- Количество нейронов в скрытых слоях – 8, 8
- Количество процентов из исходного датасета для тестового множества – 20
- Количество процентов из исходного датасета для валидационного множества – 10
- Функция активации для каждого слоя – сигмоида
- Количество эпох обучения – 25 (данное значение выбрано исходя из большого количества данных в исходном датасете – 17898)
- Скорость обучения нейронной сети – 0.1
- Нормализация входных данных – нелинейная
- Нормализация выходных данных – отсутствует (так как в датасете 2 класса – 0 и 1)

Результаты работы нейронной сети будут следующими:

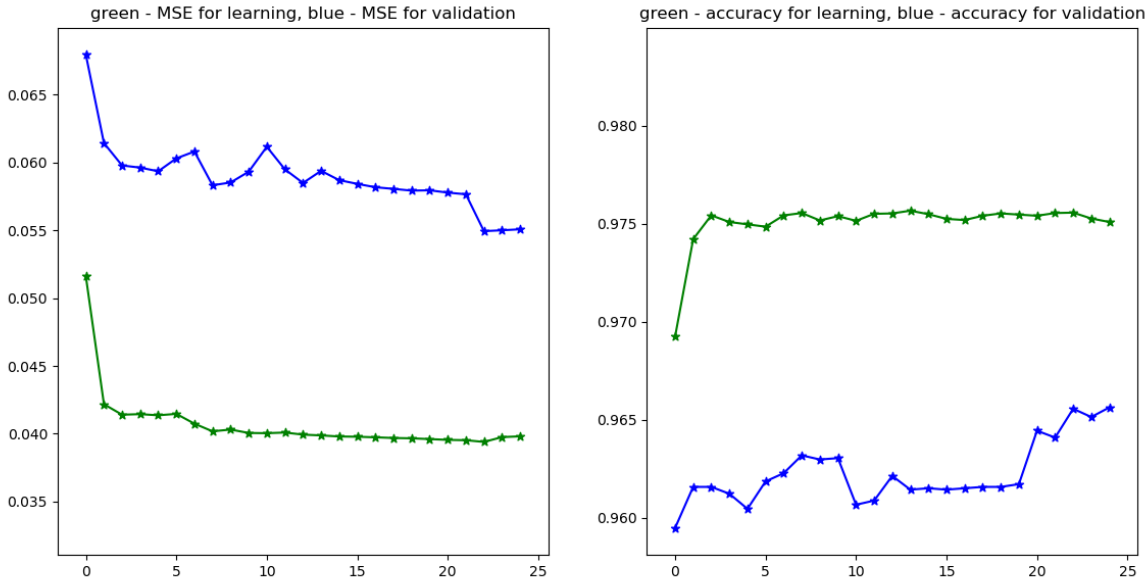


Рис. 9. Графики зависимости среднеквадратичной ошибки и accuracy для обучающей и тестовой выборок от эпохи.

```
MSE = 0.05929636045323017
TP =  3349
FP =  125
FN =  125
TN =  3349
accurancy =  0.9640184225676454
precision =  0.9640184225676454
recall =  0.9640184225676454
Для продолжения нажмите любую клавишу . . .
```

Рис. 10. Метрики для валидационной выборки.