МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Московский авиационный институт
(Национальный исследовательский университет)

Факультет: «Информационные технологии и прикладная математика»
Кафедра 806: «Вычислительная математика и программирование»

Лабораторная работа № 3
по курсу «Нейроинформатика»

Студент: А. О. Тояков
Преподаватель: Н. П. Аносова
Группа: М8о-407б-18
Дата:
Оценка:
Подспись:

Москва, 2021

МНОГОСЛОЙНЫЕ СЕТИ. АЛГОРИТМ ОБРАТНОГО РАСПРОСТРАНЕНИЯ ОШИБКИ

Цель работы: исследование свойств многослойной нейронной сети прямого распространения и алгоритмов ее обучения, применение сети в задачах классификации и аппроксимации функции.

Основные этапы работы:

1. Использовать многослойную нейронную сеть для классификации точек в случае, когда классы не являются линейно разделимыми.
2. Использовать многослойную нейронную сеть для аппроксимации функции. Произвести обучение с помощью одного из методов первого порядка.
3. Использовать многослойную нейронную сеть для аппроксимации функции. Произвести обучение с помощью одного из методов второго порядка.

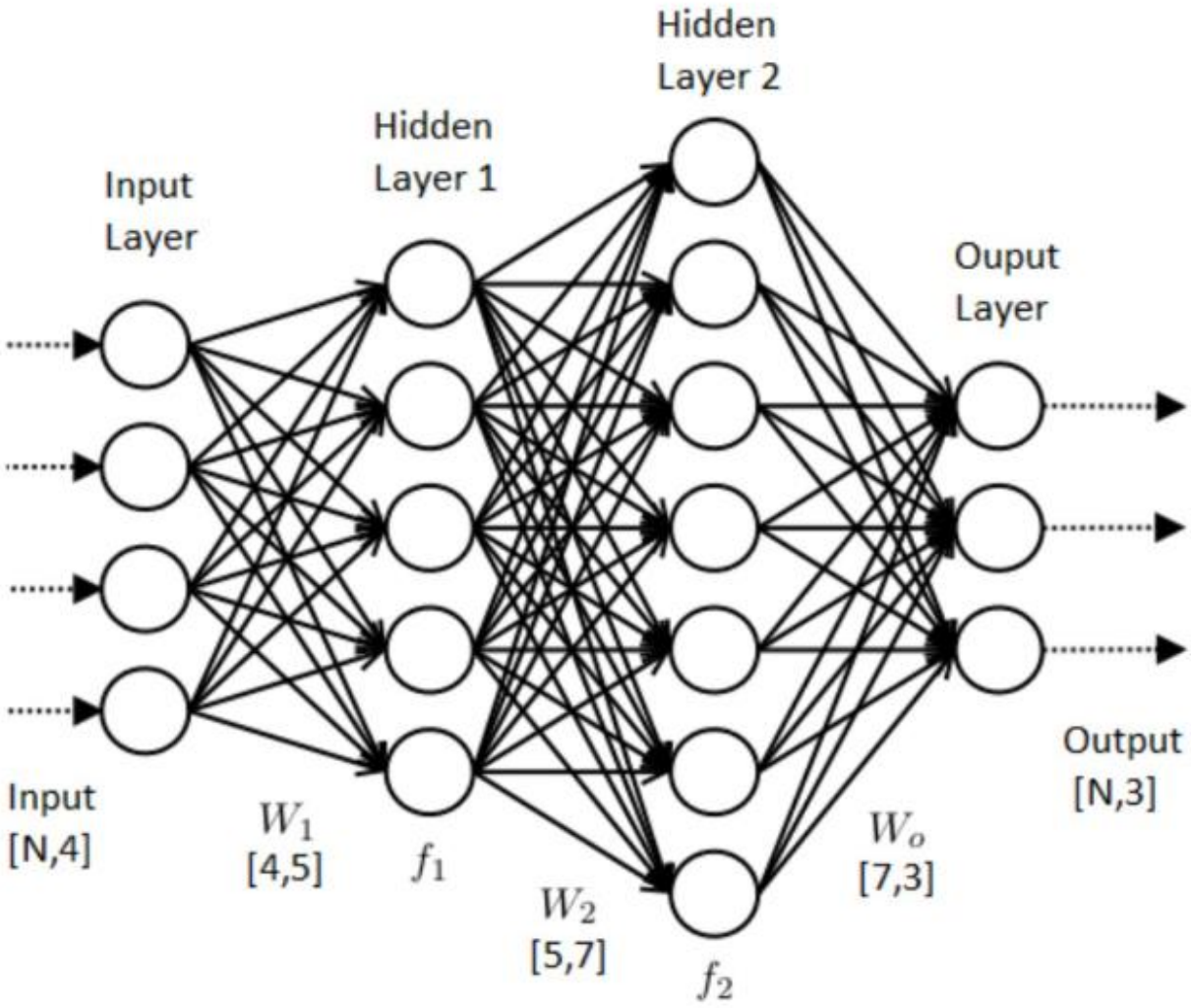
Вариант №26

1. Эллипс: a = 0.3, b = 0.3, α = 0, x0 = 0, y0 = 0.2
   Эллипс: a = 0.5, b = 0.5, α = 0, x0 = 0, y0 = 0.2
   Эллипс: a = 0.8, b = 1, α = 0, x0 = −0.1, y0 = −0.1
2. x = sin(sin(t) * t * t), t ∈ [0, 3.5], h = 0.01

Метод гибкого распространения и метод, предложенный Бройденом, Флетчером, Гольдфарбом и Шанно.

СТРУКТУРА МОДЕЛИ

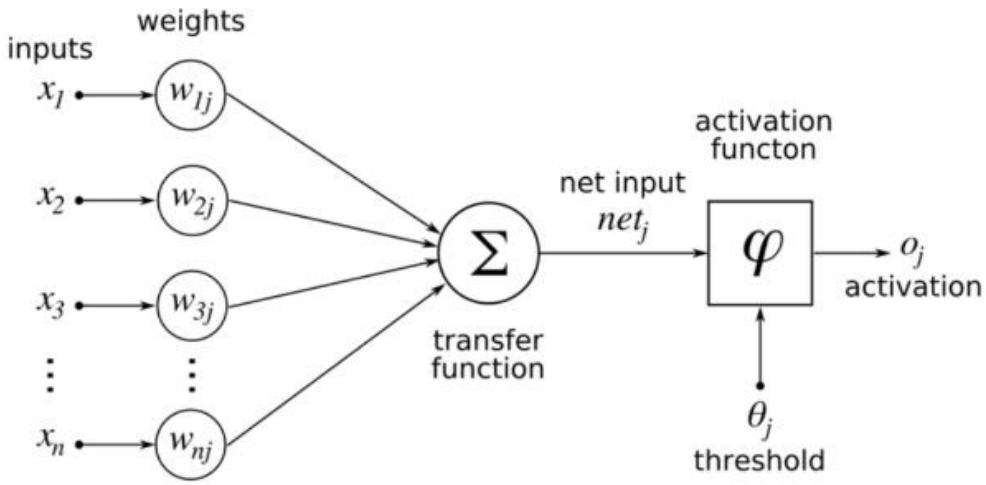
Для решения этой задачи необходимо воспользоваться нейронной сетью из нескольких слоёв, которую можно задать в виде:



При этом каждый нейрон такой сети определяется функцией:

$$o = \sigma(\sum_{i=0}^{n} w_i x_i + b)$$

и может иметь следующий вид:



Чтобы реализовать слой сети можно воспользоваться представлением весов и смещений перцептронов, как матрицу (n + 1) * m, где n – число входов, а m – число выходов. При этом градиент ошибки я определяю по правилу обратного распространения ошибки итерационно для каждого слоя, начиная с последнего (выходного).

Реализация полносвязного слоя:

```
# Fully conncted Layer with biases
class FullyConnectedLayer:
    def __init__(self, neuros = 64, activation = Sigmoid()):
        self.W = None
        self.X = None
        self.S = None
        self.activ_f = activation
        self.grad_W = None # current gradient for training
        self.neuros = neuros

    # weights initializer
    @staticmethod
    def weights_random_init(shape, limits):
        mult = limits[1] - limits[0]
        return np.random.random(shape)*mult + limits[0]

    # w diap - diapazon of weights in init, solver - gradient method
    def compilation(self, prev_neuros, w_diap = (-5, 5)):
        # init weight  by random (+ 1 for bias)
        self.W = self.weights_random_init((prev_neuros + 1, self.neuros), w_diap)


    # forward step - just multiply matrix and store result
    def forward_step(self, X):
        # add column of ones for bias weights
        self.X = np.append(X, np.ones((X.shape[0], 1)), axis = 1)
        # compute net(X) function
        self.S = self.X.dot(self.W)
        # output - activation(net(X))
        return self.activ_f(self.S)

    # backward step - computes gradients of loss by each weights
    def backward_step(self, L_grad):
        L_grad *= self.activ_f.grad(self.S) # gradient from next layer
        next_grad = L_grad.dot(self.W[:-1].T) # gradient for next layer
        self.grad_W = np.zeros(self.W.shape) # init grad by (m, k) of 0.0
        # (dL/dw) - sum (avg) by all input data
        for i in range(self.X.shape[0]):
            # X[i] - (1, m), (dL/ds) - (1, k) =>
            # we need dot (m, 1) on (1, k) for get gradient by all weights of (m, k)
            self.grad_W += \
                self.X[i].reshape(
                    self.X.shape[1], 1
                ).dot(L_grad[i].reshape(1, L_grad.shape[1]))
        # self.opt_method(d_w) # update weights with some gradient optimize method
        return next_grad # return grad by funtion of next layer (dL/df)

    # Magic methods:
    def __str__(self):
        return "FullyConnected({})".format(self.neuros)

    def __repr__(self):
        return self.__str__()

    def __call__(self, X):
        return self.forward_step(X)
```

Тогда нейронную сеть из нескольких слоёв можно реализовать следующим образом:

```
class NeuralNetwork:
    def __init__(self):
        self.graph = []
        self.solver = None
        self.output = None

    #  add layer to model
    def add(self, layer):
        self.graph.append(layer)

    # compile sequential network
    def compilation(self, solver, out_layer, data_dim):
        # data_dim - dimention of input vectors
        prev_neuros = data_dim
        # for each layer define Weights
        for layer in self.graph:
            layer.compilation(prev_neuros)
            prev_neuros = layer.neuros
        # set loss funstion with solver
        self.output = out_layer
        solver.set_network(self.graph) #init solver data
        self.solver = solver # set solver

    # train network
    def fit(self, X, Y, X_val=None, Y_val=None, steps=600, batch_size=1):
        hist = []
        # compute history if validation data exist
        if X_val is None and Y_val is None:
            hist = None
        # reshape Y
        if len(Y.shape) == 1:
            Y = np.reshape(Y, (Y.shape[0], 1))
        # itterative weights updating
        for _ in tqdm(range(steps)):
            # for each batch in data do pass train pass in model
            for i in range(0, X.shape[0] - batch_size + 1, batch_size):
                # extract batch (X, Y)
                X_pass = X[i: i + batch_size]
                Y_pass = Y[i: i + batch_size]
                # do forward pass which compute intermideate values for update
                Y_out = self.forward_pass(X_pass)
                # compute gradient of last output
                Y_grad = self.output.loss_grad(Y_out, Y_pass)
                # compuite gradients for each layer in graph
                self.backward_pass(Y_grad)
                # solver has graph, call will update weights of layers
                self.solver()
            # append val loss to history if exist
            if hist is not None:
                hist.append(self.loss(X_val, Y_val))
        # return history of train
        return hist

    # forward pass - just matrix mults from start to end
    def forward_pass(self, X_pass):
        X_pass = np.copy(X_pass)
        for layer in self.graph:
            X_pass = layer(X_pass)

        return self.output(X_pass)

    # backward pass - backdirection move, which computes derivatives for layers
    def backward_pass(self, Y_grad):
        # getback direction
        back_direction = reversed(self.graph)
        for layer in back_direction:
            # compute gradient for next layer
            Y_grad = layer.backward_step(Y_grad)

    def classify(self, X):
        return self.output.classify(self(X))

    def classify_task(self, X):
        return self.output.classify_task(self(X))

    def loss(self, X, Y):
        return self.output.loss(self(X), Y)

    def __call__(self, X):
```

```
        if len(X.shape) == 1:
            X = X.reshape(1, X.shape[0])
        return self.forward_pass(X)
```

Как можно заметить, для обучения я использую один из заданных алгоритмов оптимизации на основе подсчитанных градиентов для каждого из слоёв на основе функции потерь, которая выбирается в зависимости от задач, решаемых нейронной сетью.

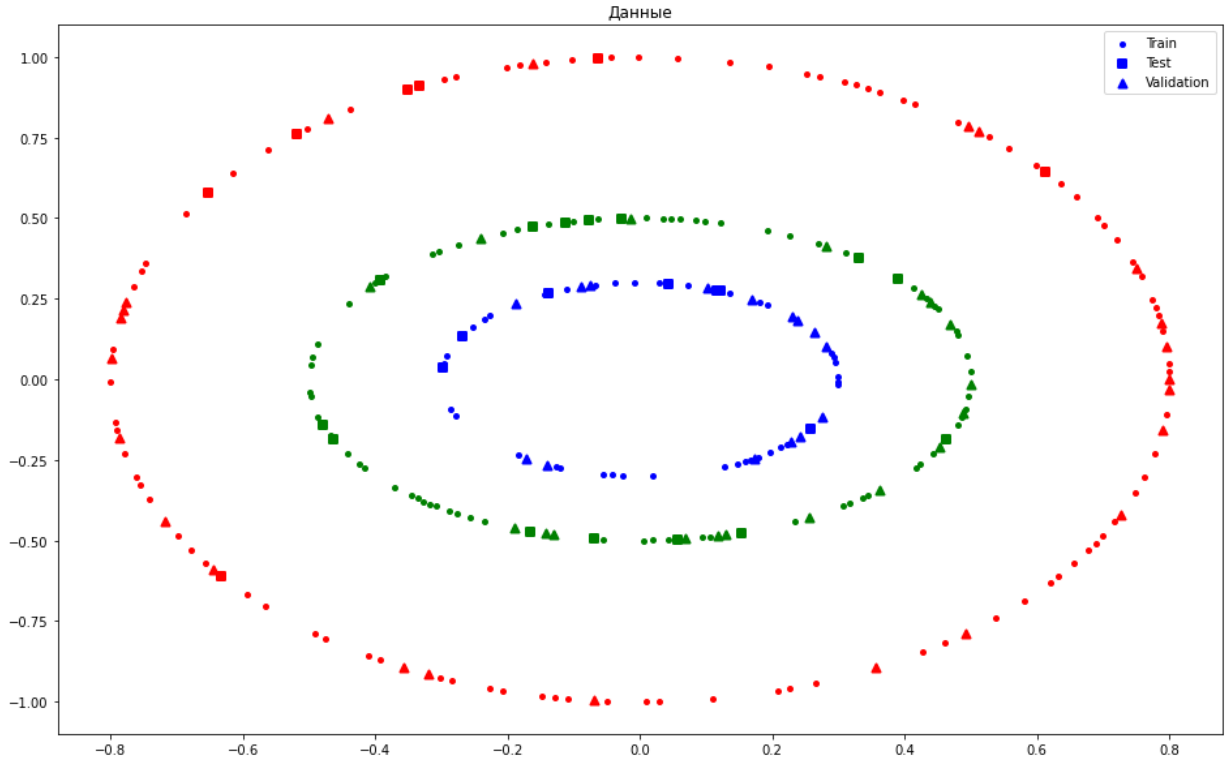
ХОД РАБОТЫ

Этап №1

Я сгенерировал обучающее множество на основе заданных уравнений кривых для каждого из классов и разделил его случайным образом на обучающее, тестовое и контрольное.

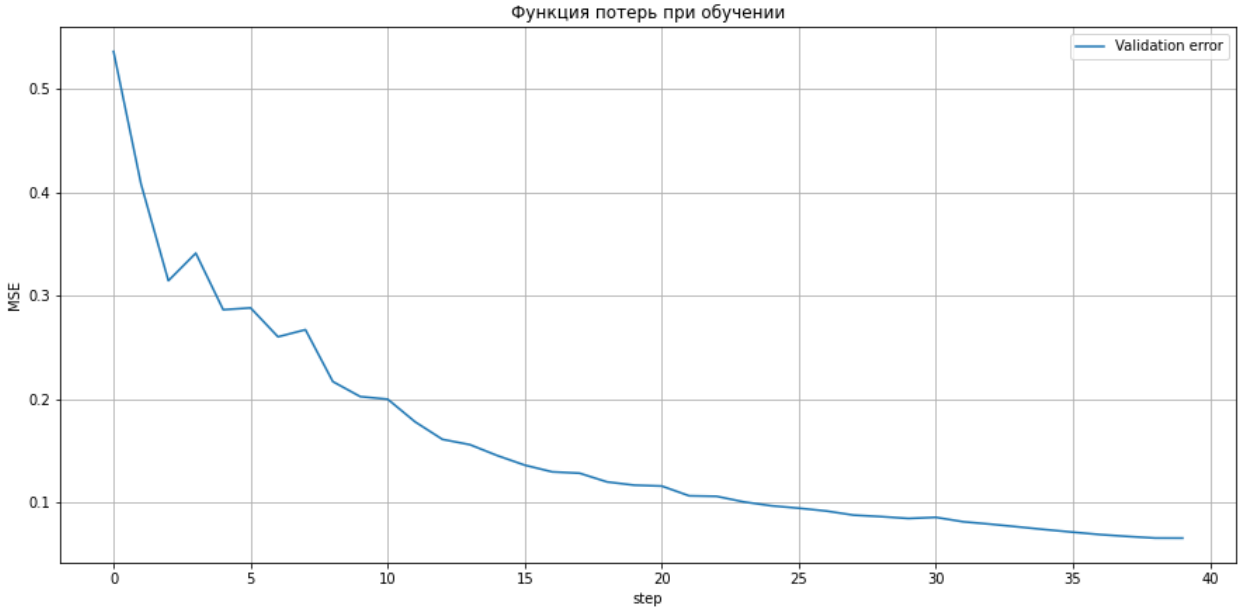
```
cls1 = gen_line(a1, b1, alpha1)
cls2 = gen_line(a2, b2, alpha2)
cls3 = gen_line(a3, b3, alpha3)
X, Y = assimetric_dataset_classify((cls1, cls2, cls3), (60, 100, 120))

train_p = 0.7
valid_p = 0.2
test_p = 0.1

size_train = int(train_p * X.shape[0])
bound_valid = size_train + int(valid_p * X.shape[0])

X_train, Y_train = X[:size_train], Y[:size_train]
X_valid, Y_valid = X[size_train:bound_valid], Y[size_train:bound_valid]
X_test, Y_test = X[bound_valid:], Y[bound_valid:]
```

Иллюстрация полученного разбиения:

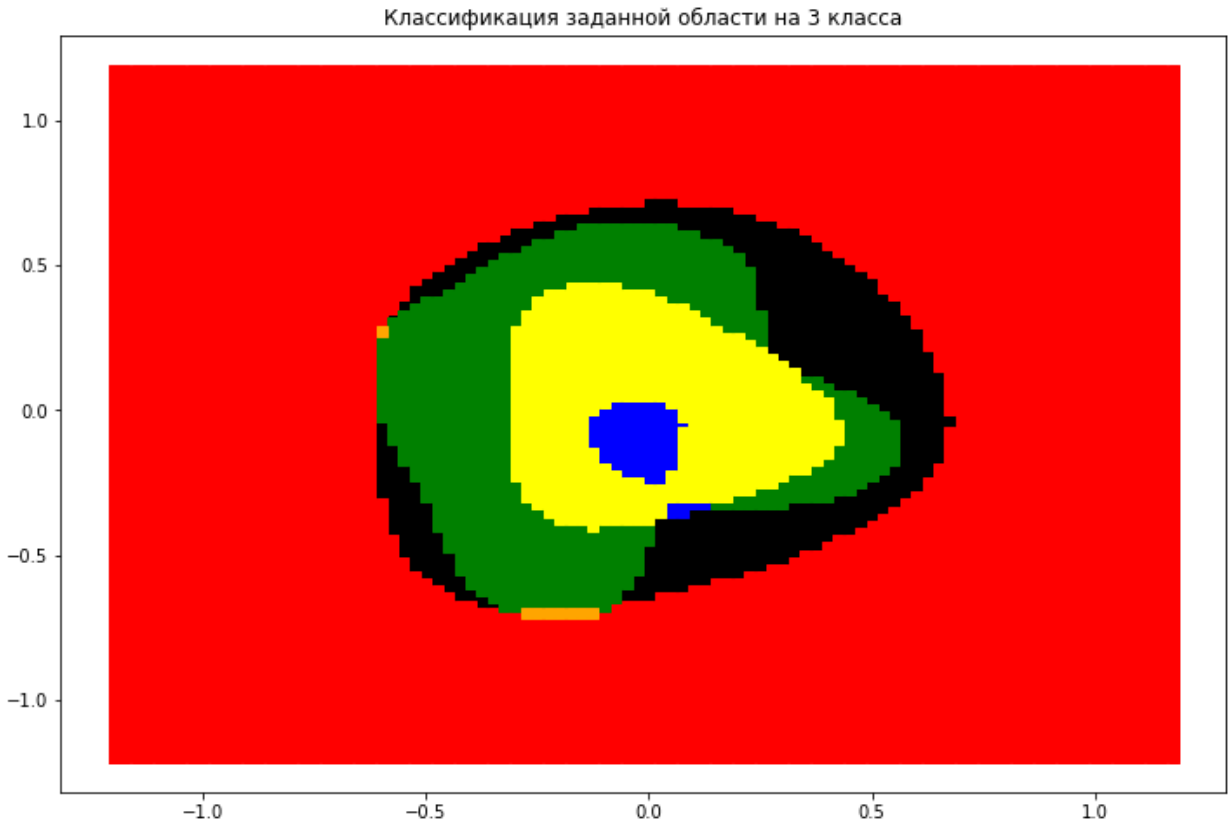


Далее я построил и обучил нейронную сеть с двумя слоями с выходом Sigmoid:

```
model = NeuralNetwork()
model.add(FullyConnectedLayer(neuros=20, activation=Sigmoid()))
model.add(FullyConnectedLayer(neuros=3, activation=Sigmoid()))

model.compilation(solver=RProp(), out_layer=Linear_with_MSE(), data_dim=2)
hist = model.fit(X_train, Y_train, X_valid, Y_valid, 40, X_train.shape[0])
```



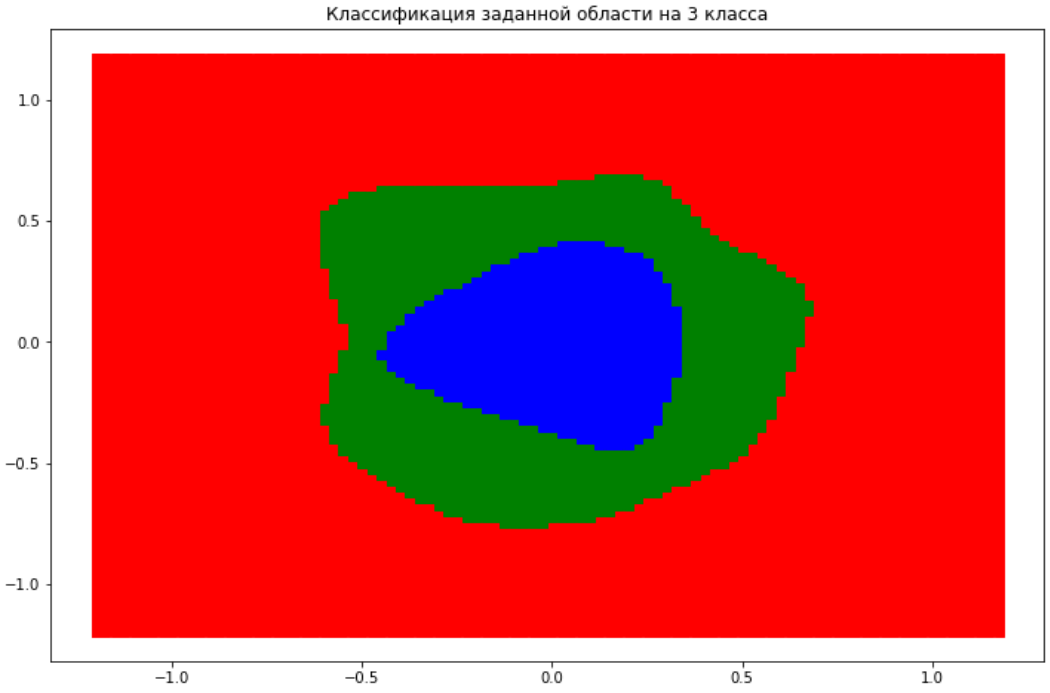
И классифицировал область [-1.2, 1.2]x[-1.2, 1.2]:



Такой результат получился из-за того, что выход Sigmoid с порогом 0.5 может отнести объект сразу к нескольким классам (или вообще ни к одному).

Для задач классификации рекомендуется использовать выходной слой Softmax с функцией ошибки – кросс-энтропия.

Результаты такого слоя:



Этапы №2-3

Я сгенерировал обучающее множество на основе заданного уравнения для задачи регрессии и разбил его на тренировочное и контрольное:

```
x = np.arange(*interval, step_h)
y = func_t(x)

permut = permutation(len(x))
x, y = x[permut].reshape(x.shape[0], 1), y[permut].reshape(y.shape[0], 1)
```

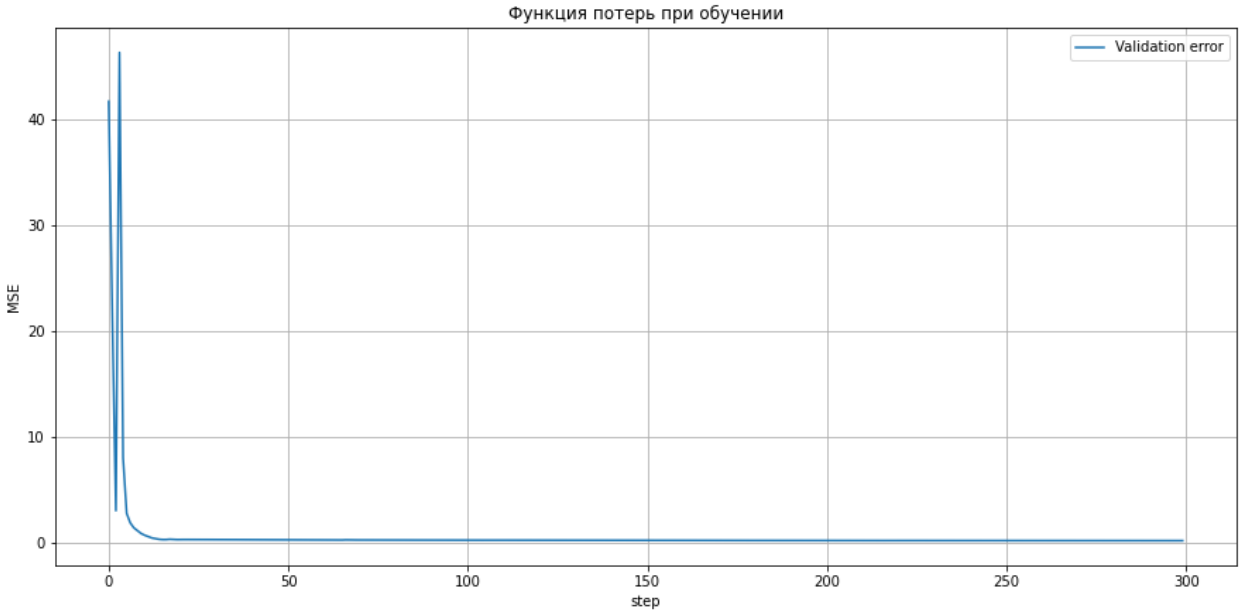
```
train_p = 0.9
valid_p = 0.1

size_train = int(train_p * x.shape[0])

X_train, Y_train = x[:size_train], y[:size_train]
X_valid, Y_valid = x[size_train:], y[size_train:]
```

Далее я построил нейронные сети с двумя слоями и выходом Purelin с указанными в задании алгоритмами оптимизации:

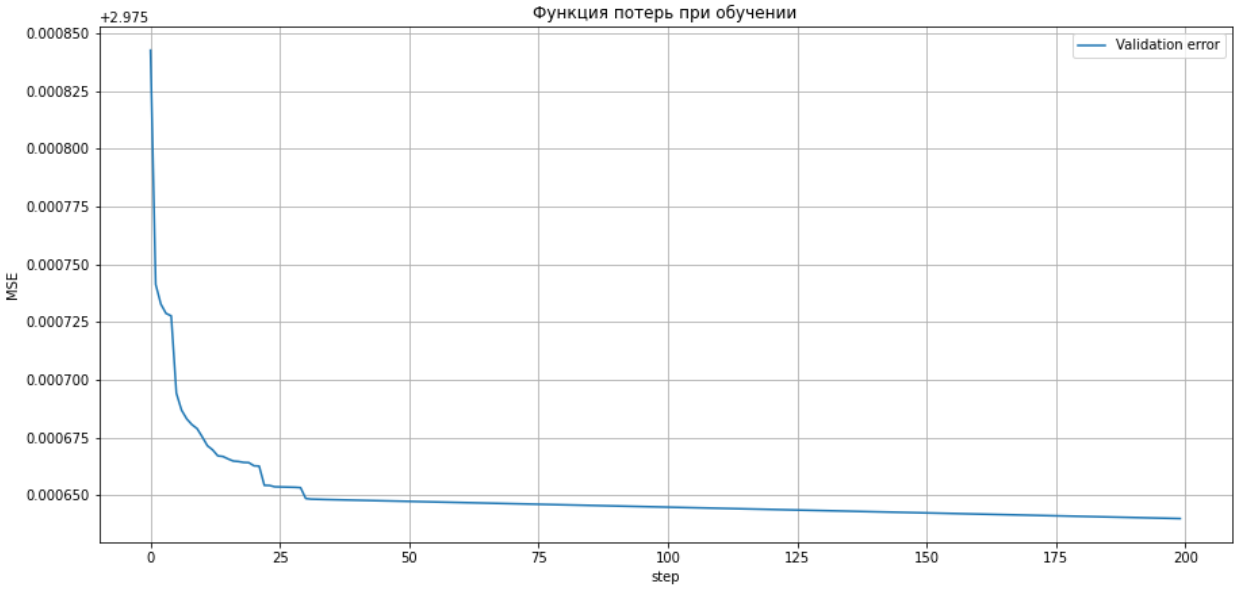
```
model1 = NeuralNetwork()
model1.add(FullyConnectedLayer(neuros=10, activation=Tansig()))
model1.add(FullyConnectedLayer(neuros=1, activation=Purelin()))
model1.compilation(solver=RProp(inf = 1e-6, sup = 50.0, n_neg=0.5, n_pos=1.2), out_layer=Linear_w
ith_MSE(), data_dim=1)

model2 = NeuralNetwork()
model2.add(FullyConnectedLayer(neuros=10, activation=Tansig()))
model2.add(FullyConnectedLayer(neuros=1, activation=Purelin()))
model2.compilation(solver=BFGS(learn_rate=0.000000005), out_layer=Linear_with_MSE(), data_dim=1)
```

И обучил их:

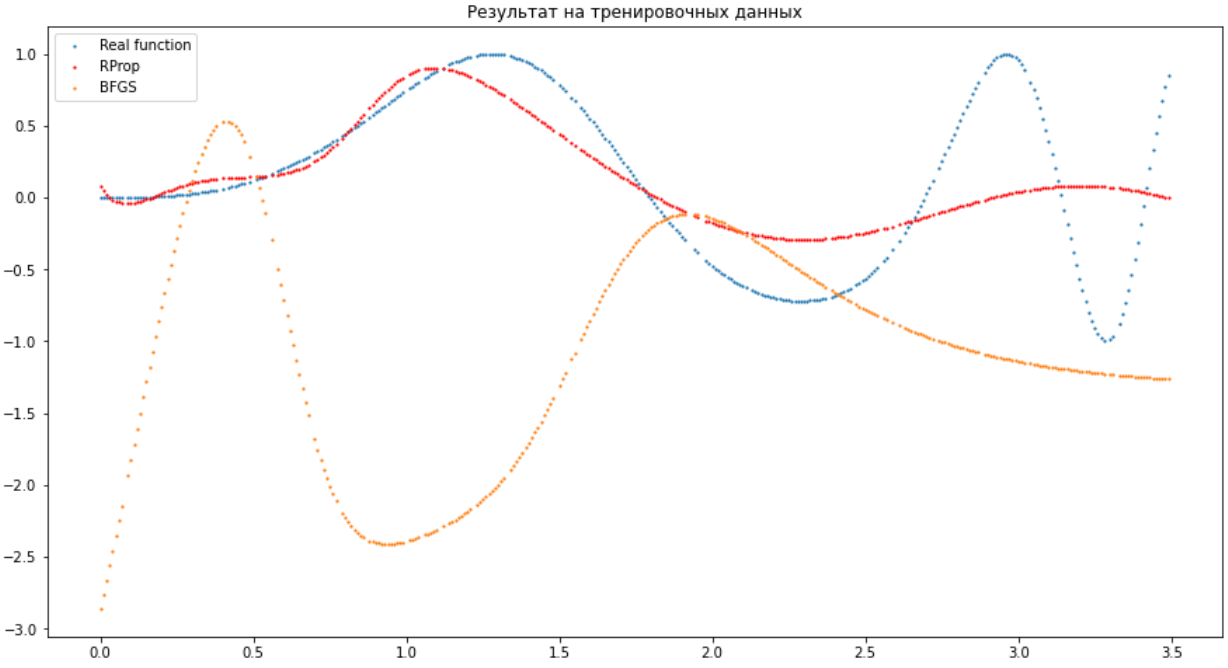
```
hist = model1.fit(X_train, Y_train, X_valid, Y_valid, 300, X_train.shape[0])
plot_history(hist)
```



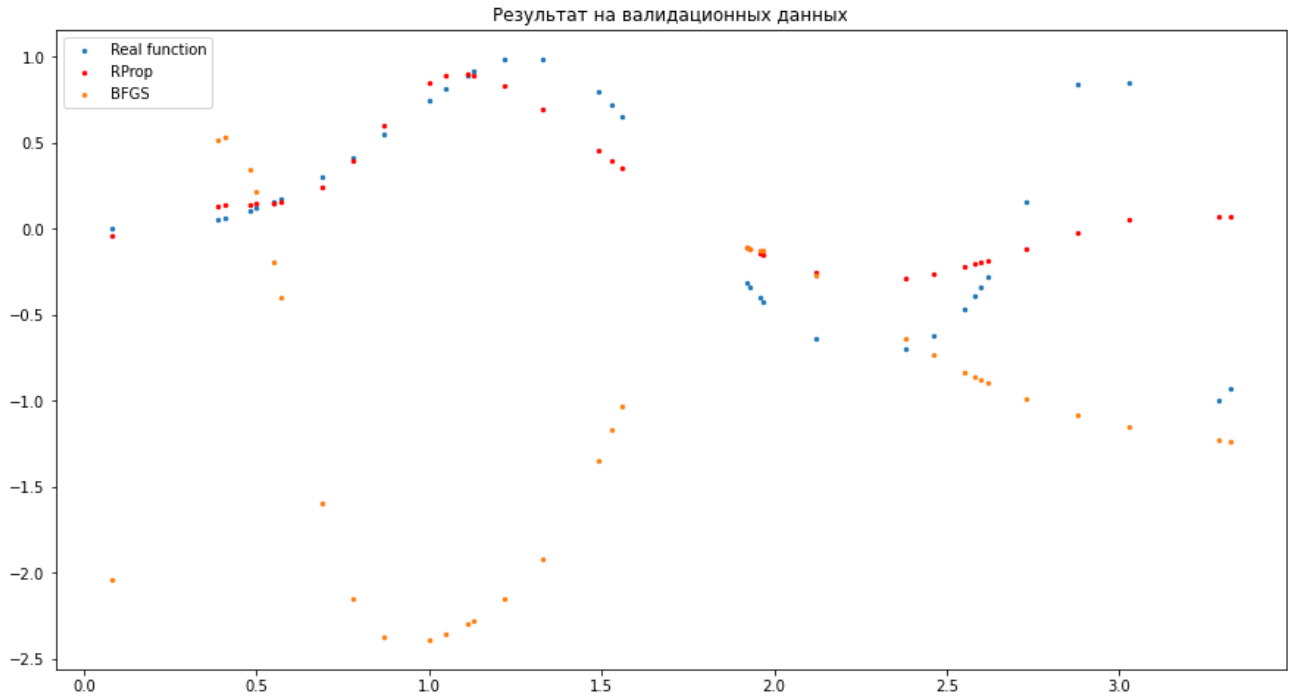

```
hist = model2.fit(X_train, Y_train, X_valid, Y_valid, 200, X_train.shape[0
])
plot_history(hist)
```



Результат обучения на тренировочных данных:



Результат на контрольных данных:



ВЫВОДЫ

Выполнив третью лабораторную работу по курсу «Нейроинформатика», я узнал о многослойных сетях, методах их обучения, а также укрепил знания о методе обратного распространения ошибки, о котором нам рассказывали на курсе «Искусственный интеллект».

Полученные результаты сообщают, что рассмотренная нами модель хорошо справляется как с задачами классификации, так и с задачами регрессии, но полученный результат сильно зависит не только от выбранных параметров сети, но и от функций активации и функций ошибок сети. Так, например, для задач регрессии лучше подходит MSE ошибка, когда для задач классификации следует выбрать функцию ошибки – кросс-энтропию с выходом сети Softmax.