Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу «Нейроинформатика»

Студент: М. А. Бронников
Преподаватель: Н. П. Аносова
Группа: М8О-407Б

Дата:
Оценка:
Подпись:

Москва, 2020

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 №4

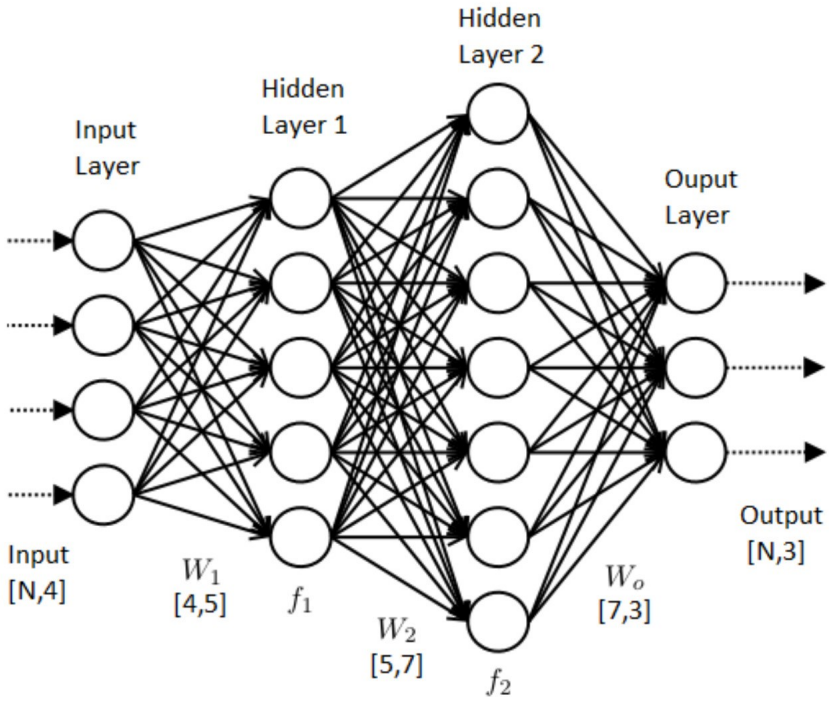
1. Эллипс: $a = 0.4, b = 0.15, \alpha = \frac{\pi}{6}, x_0 = 0, y_0 = 0$
   Эллипс: $a = 0.7, b = 0.5, \alpha = -\frac{\pi}{3}, x_0 = 0, y_0 = 0$
   Эллипс: $a = 1, b = 1, \alpha = 0, x_0 = 0, y_0 = 0$

2. $x = \cos(3t^2 + 5t + 10),\ t \in [0, 2.5], h = 0.01$

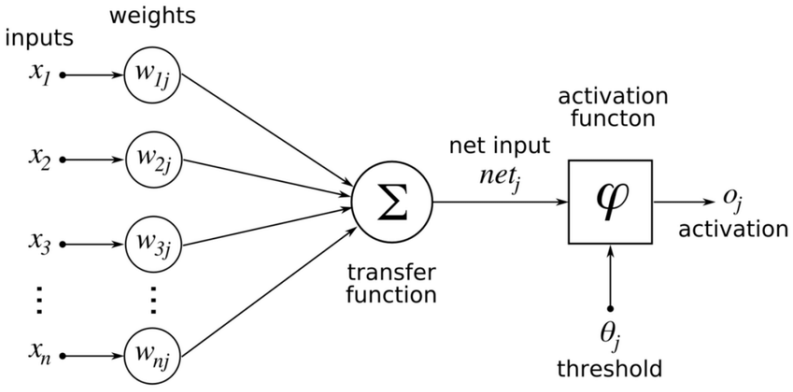
   Метод градиентного спуска и метод, предложенный Бройденом, Флетчером, Гольдфарбом и Шанно.

1 Структура модели

Для решения этой задачи необходимо воспользоваться нейронной сетью из нескольких слоев, которую можно представить 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 смеще-

ний перцептронов как марицу $(n+1) \times m$, где n - число входов, а m - число выходов. При этом градиент ошибки я определяю по **правилу обратного распространения ошибки** иттерационно для каждого слоя, начиная с последнего (выходного).

Реализация полносвязного слоя:

```
1
2  # Fully conncted Layer with biases
3  class FullyConnectedLayer:
4      def __init__(self, neuros = 64, activation = Sigmoid()):
5          self.W = None
6          self.X = None
7          self.S = None
8          self.activ_f = activation
9          self.grad_W = None # current gradient for training
10         self.neuros = neuros
11
12     # weights initializer
13     def weights_random_init(shape, limits):
14         mult = limits[1] - limits[0]
15         return np.random.random(shape)*mult + limits[0]
16
17     # w_diap - diapazon of weights in init, solver - gradient method
18     def compilation(self, prev_neuros, w_diap = (-5, 5)):
19         # init weight by random (+ 1 for bias)
20         self.W = self.weights_random_init((prev_neuros + 1, self.neuros), w_diap)
21
22
23     # forward step - just multiply matrix and store result
24     def forward_step(self, X):
25         # add column of ones for bias weights
26         self.X = np.append(X, np.ones((X.shape[0], 1)), axis = 1)
27         # compute net(X) function
28         self.S = self.X.dot(self.W)
29         # output - activation(net(X))
30         return self.activ_f(self.S)
31
32     # backward step - computes gradients of loss by each weights
33     def backward_step(self, L_grad):
34         L_grad *= self.activ_f.grad(self.S) # gradient from next layer
35         next_grad = L_grad.dot(self.W[:-1].T) # gradient for next layer
36         self.grad_W = np.zeros(self.W.shape) # init grad by (m, k) of 0.0
37         # (dL/dw) - sum (avg) by all input data
38         for i in range(self.X.shape[0]):
39             # X[i] - (1, m), (dL/ds) - (1, k) =>
40             # we need dot (m, 1) on (1, k) for get gradient by all weights of (m, k)
41             self.grad_W += \
42                 self.X[i].reshape(
43                     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)

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

Тогда нейронную сеть из нескольких слоев 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
```

```
41                  Y_out = self.forward_pass(X_pass)
42                  # compute gradient of last output
43                  Y_grad = self.output.loss_grad(Y_out, Y_pass)
44                  # compuite gradients for each layer in graph
45                  self.backward_pass(Y_grad)
46                  # solver has graph, call will update weights of layers
47                  self.solver()
48              # append val loss to history if exist
49              if hist is not None:
50                  hist.append(self.loss(X_val, Y_val))
51          # return history of train
52          return hist
53
54      # forward pass - just matrix mults from start to end
55      def forward_pass(self, X_pass):
56          X_pass = np.copy(X_pass)
57          for layer in self.graph:
58              X_pass = layer(X_pass)
59
60          return self.output(X_pass)
61
62      # backward pass - backdirection move, which computes derivatives for layers
63      def backward_pass(self, Y_grad):
64          # getback direction
65          back_direction = reversed(self.graph)
66          for layer in back_direction:
67              # compute gradient for next layer
68              Y_grad = layer.backward_step(Y_grad)
69
70      def classify(self, X):
71          return self.output.classify(self(X))
72
73      def classify_task(self, X):
74          return self.output.classify_task(self(X))
75
76      def loss(self, X, Y):
77          return self.output.loss(self(X), Y)
78
79      def __call__(self, X):
80          if len(X.shape) == 1:
81              X = X.reshape(1, X.shape[0])
82          return self.forward_pass(X)
```

Как можно заметить, для обучения я использую один из заданных алгоритмов оптимизации на основе подсчитанных градиентов для каждого из слоев на основе функции потерь, которая выбирается в зависимости от задач, решаемых нейронной сетью. Реализации методов обучения я в отчете приводить не стал, их можно посмотреть у меня в исходных файлах.

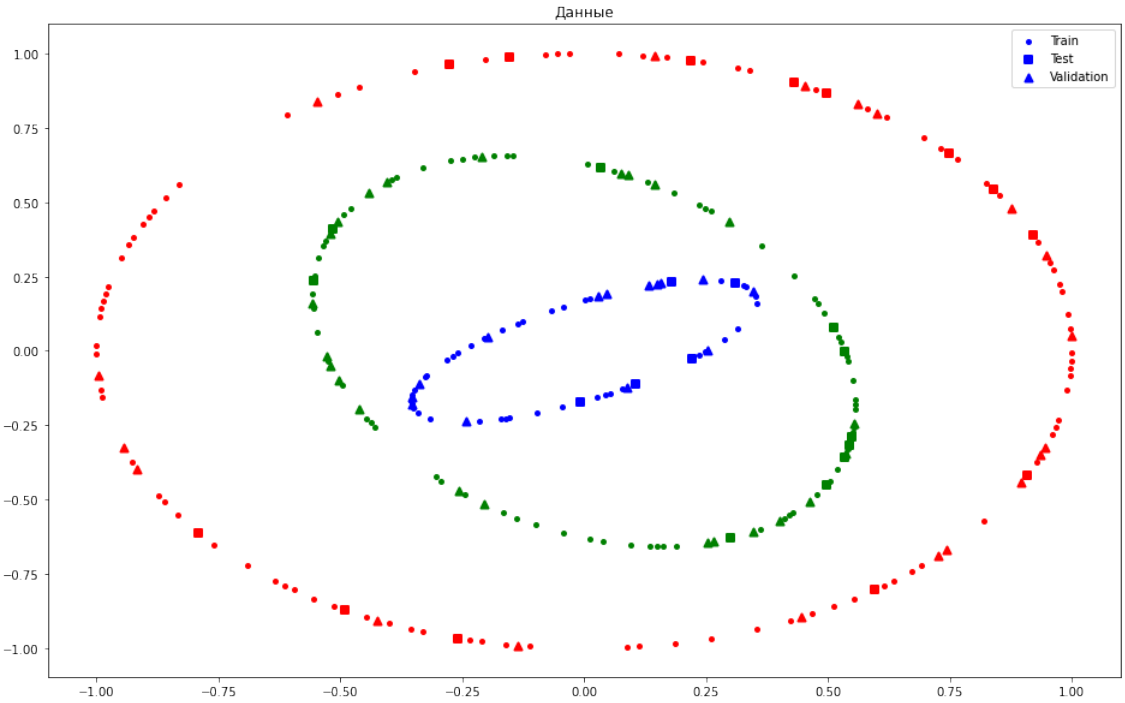
2 Ход работы

Этап №1

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

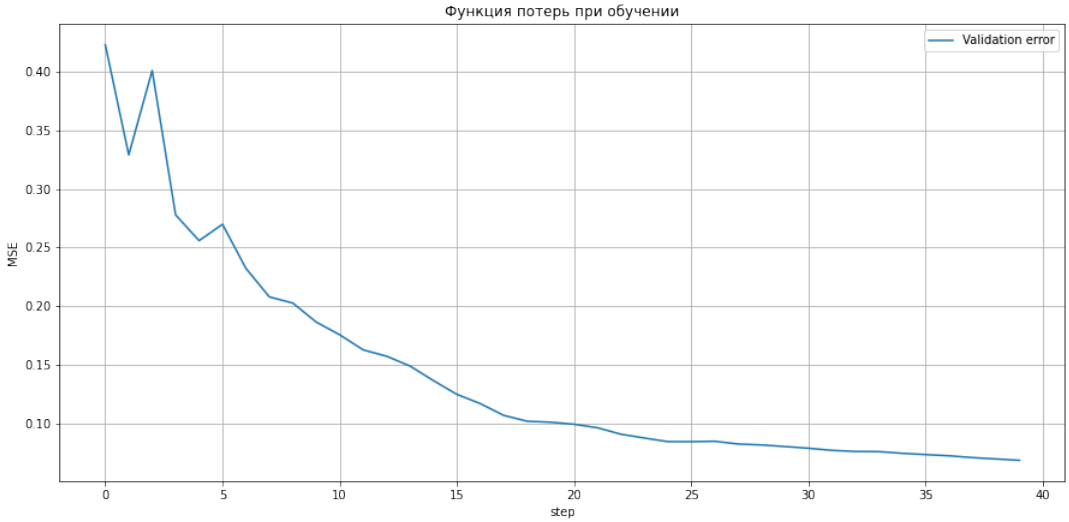
```
cls1 = gen_line(a1, b1, alpha1)
cls2 = gen_line(a2, b2, alpha2)
cls3 = gen_line(a3, b3, alpha3)
X, Y = assimetric_dataset_classify((cls1, cls2, cls3), (60, 100, 120))

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 полученого разбиения:

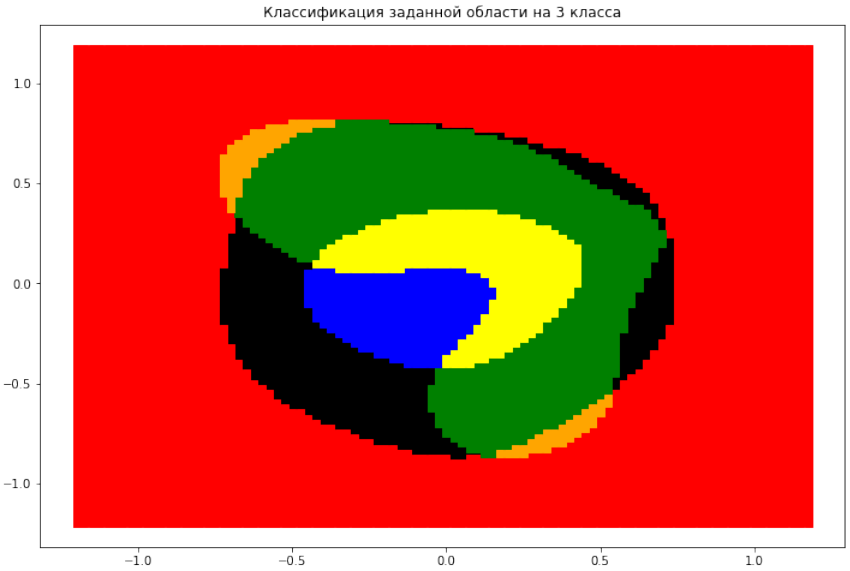


Далее я построил и обучил нейронную сеть с двумя слоями с выходом *Sigmoid*:

```
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])
```

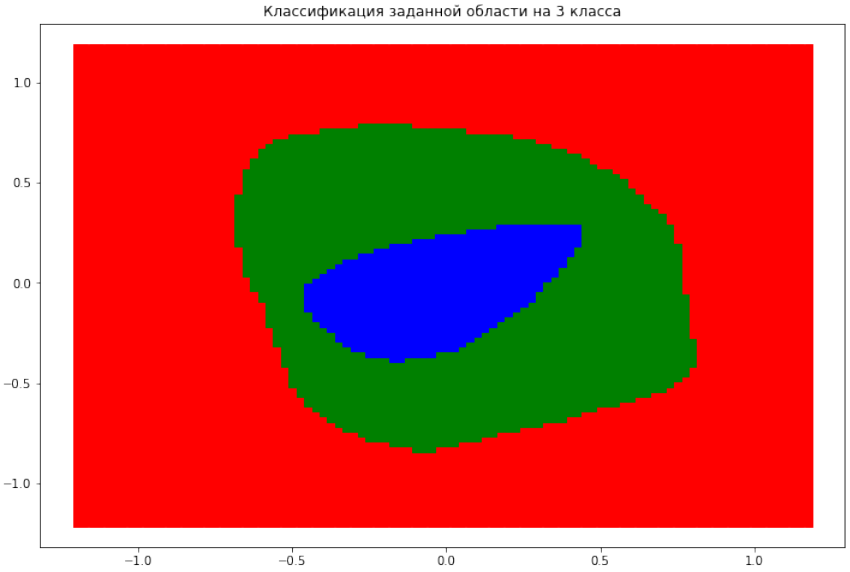


И классифицировал область $[-1.2, 1.2]x[-1.2, 1.2]$:



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*.

Результат для такого слоя:



Этапы №2-3

Я сгенерировал обучающее множество на основе заднного уравнения для задачи регресии и разбил его на тренировочное и контрольное:

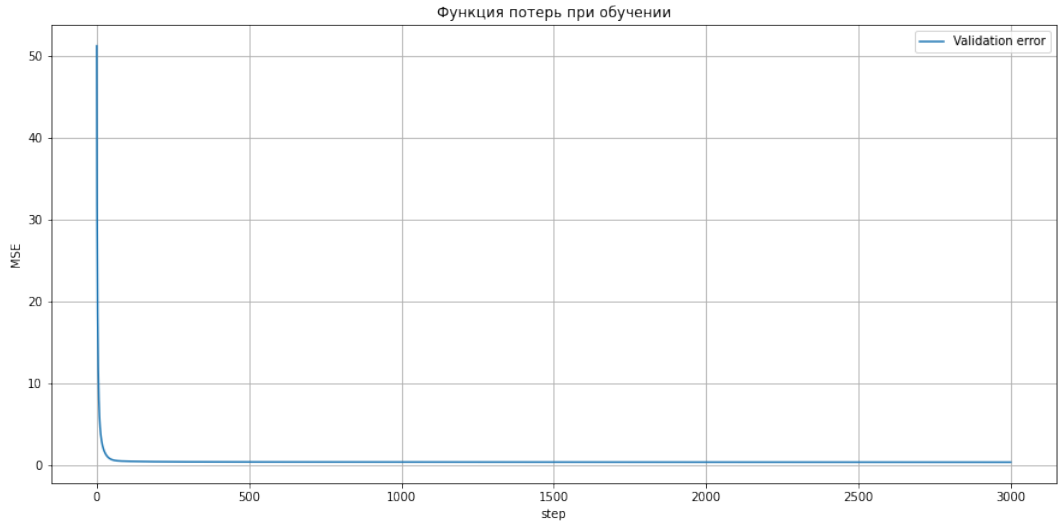
```
x = np.arange(*interval, step_h)
y = func_t(x)

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:

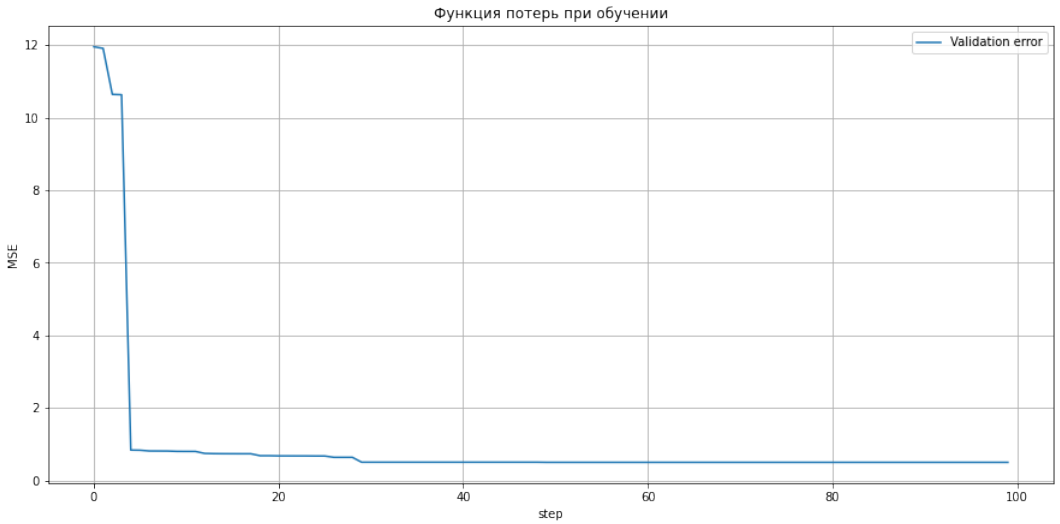
```
model1 = NeuralNetwork()
# model1.add(FullyConnectedLayer(neuros=32, activation=Tansig()))
# model1.add(FullyConnectedLayer(neuros=16, activation=Tansig()))
model1.add(FullyConnectedLayer(neuros=10, activation=Tansig()))
model1.add(FullyConnectedLayer(neuros=1, activation=Purelin()))
model1.compilation(solver=GradientDescent(learn_rate=0.00005), out_layer=
    Linear_with_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=0.000005), out_layer=Linear_with_MSE(),
    data_dim=1)
```

И обучил их:

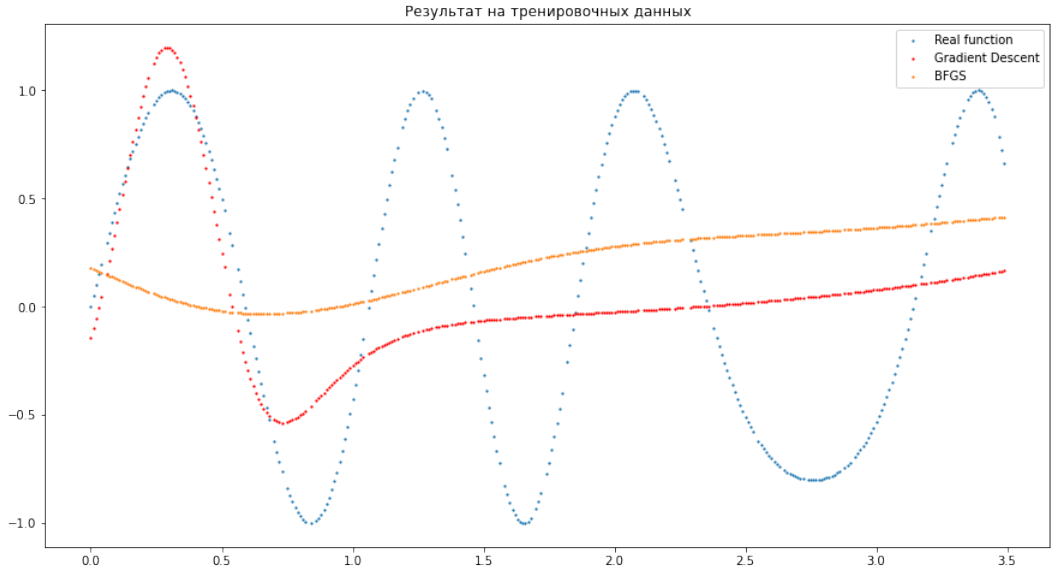
```
hist = model1.fit(X_train, Y_train, X_valid, Y_valid, 3000, X_train.shape[0])
plot_history(hist)
```




```
hist = model2.fit(X_train, Y_train, X_valid, Y_valid, 100, X_train.shape[0])
plot_history(hist)
```



Результат обучения на тренировочных данных:



3 Выводы

Выполнив третью лабораторную работу по курсу «Нейроинформатика», я узнал о многослойных сетях, методе обратного распространения ошибки, а также методах их обучения.

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

Эта работа была интересной, но оказалась не такой простой, как первые две лабораторные работы, но я с ней справился, и в будущем я ожидаю более трудных и интересных задач.