**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

**Лабораторная работа № 3**

по курсу «Нейроинформатика»

Тема: Многослойные сети. Алгоритм обратного распространения ошибки.

Студент: Жерлыгин М.А.

Группа: 80-408б

Преподаватель: Аносова Н.П.

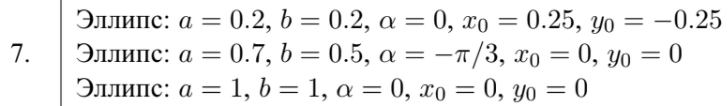
Оценка:

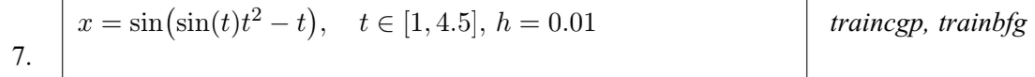
Москва, 2021

**1) Постановка задачи:** Исследование свойств многослойной нейронной сети прямого распространения и алгоритмов её обучения, применения сетей в задачах классификации и аппроксимации функции.

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

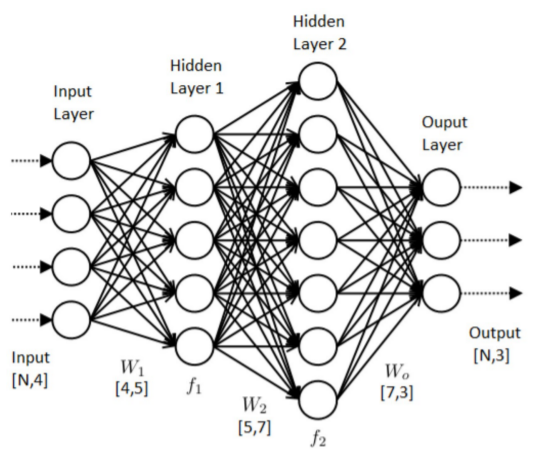
Вариант 7:



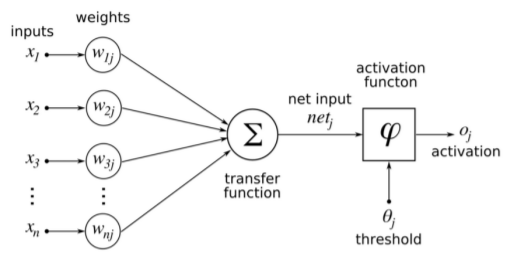


**2) Ход работы:**

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



При этом каждый нейрон такой сети может иметь следующий вид:



Чтобы реализовать слой сети можно воспользоваться представлением весов и смещений перцептронов как марицу (n+ 1)× 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 слоя нейронов:

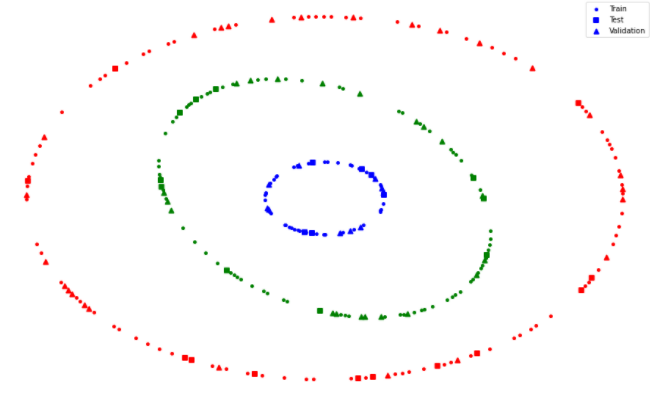
class FullyConnectedLayer:  
 def \_\_init\_\_(self, neuros = 64, activation = Sigmoid()):  
 self.W = None  
 self.X = None  
 self.S = None  
 self.activ\_f = activation  
 self.grad\_W = None  
 self.neuros = neuros  
  
 @staticmethod  
 def weights\_random\_init(shape, limits):  
 mult = limits[1] - limits[0]  
 return np.random.random(shape)\*mult + limits[0]  
   
 def compilation(self, prev\_neuros, w\_diap = (-5, 5)):  
 self.W = self.weights\_random\_init((prev\_neuros + 1, self.neuros), w\_diap)  
   
   
 def forward\_step(self, X):  
 self.X = np.append(X, np.ones((X.shape[0], 1)), axis = 1)  
 self.S = self.X.dot(self.W)  
 return self.activ\_f(self.S)  
   
 def backward\_step(self, L\_grad):  
 L\_grad \*= self.activ\_f.grad(self.S)  
 next\_grad = L\_grad.dot(self.W[:-1].T)  
 self.grad\_W = np.zeros(self.W.shape)  
 for i in range(self.X.shape[0]):  
 self.grad\_W += \  
 self.X[i].reshape(  
 self.X.shape[1], 1  
 ).dot(L\_grad[i].reshape(1, L\_grad.shape[1]))  
 return next\_grad  
   
 def \_\_str\_\_(self):  
 return "FullyConnected({})".format(self.neuros)  
   
 def \_\_repr\_\_(self):  
 return self.\_\_str\_\_()  
   
 def \_\_call\_\_(self, X):  
 return self.forward\_step(X)

Реализация нейросети:

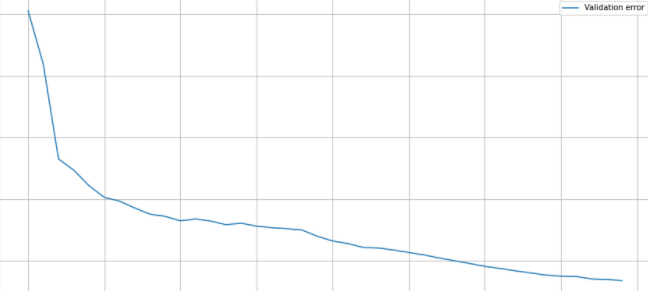
class NeuralNetwork:  
 def \_\_init\_\_(self):  
 self.graph = []  
 self.solver = None  
 self.output = None  
   
 def add(self, layer):  
 self.graph.append(layer)  
   
 def compilation(self, solver, out\_layer, data\_dim):  
 prev\_neuros = data\_dim  
 for layer in self.graph:  
 layer.compilation(prev\_neuros)  
 prev\_neuros = layer.neuros  
 self.output = out\_layer  
 solver.set\_network(self.graph)  
 self.solver = solver  
   
 def fit(self, X, Y, X\_val=None, Y\_val=None, steps=600, batch\_size=1):  
 hist = []  
 if X\_val is None and Y\_val is None:  
 hist = None   
 if len(Y.shape) == 1:  
 Y = np.reshape(Y, (Y.shape[0], 1))  
 for \_ in tqdm(range(steps)):  
 for i in range(0, X.shape[0] - batch\_size + 1, batch\_size):  
 X\_pass = X[i: i + batch\_size]  
 Y\_pass = Y[i: i + batch\_size]  
 Y\_out = self.forward\_pass(X\_pass)  
 Y\_grad = self.output.loss\_grad(Y\_out, Y\_pass)  
 self.backward\_pass(Y\_grad)  
 self.solver()  
 if hist is not None:  
 hist.append(self.loss(X\_val, Y\_val))  
 return hist  
   
 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)  
   
 def backward\_pass(self, Y\_grad):  
 back\_direction = reversed(self.graph)  
 for layer in back\_direction:  
 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)  
   
 def \_\_str\_\_(self):  
 return "Neural Network"  
   
 def \_\_repr\_\_(self):  
 return self.\_\_str\_\_()

**Этап 1:**

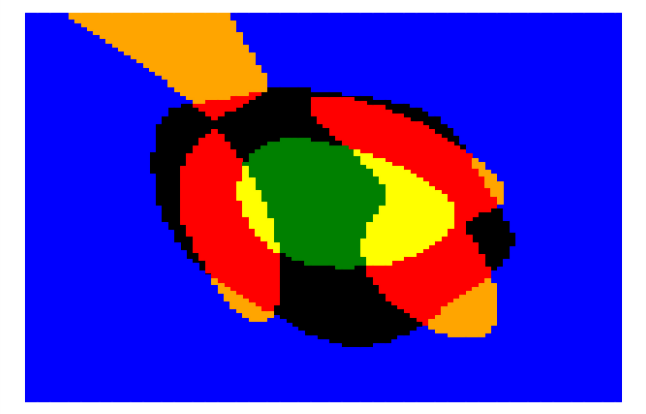
Разбиваем выборки:



Строим и обучаем нейросеть. График потерь:

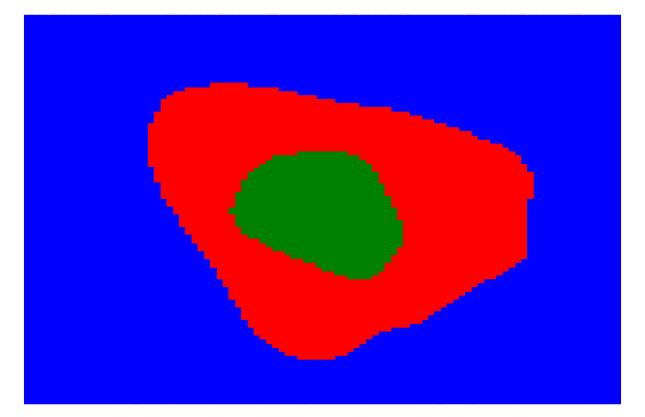


И классифицируем области:



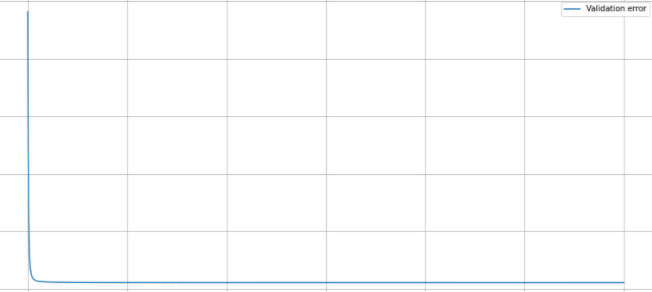
Такой результат получился из-за того, что выход Sigmoid с порогом 0.5 может отнести объект сразу к нескольким классам (или вообще ни к одному).

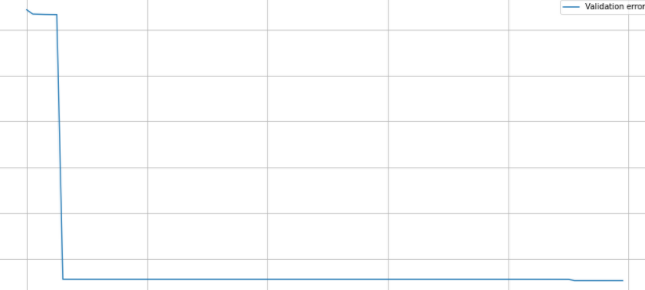
Используем функцию ошибки – кросс-энтропия:



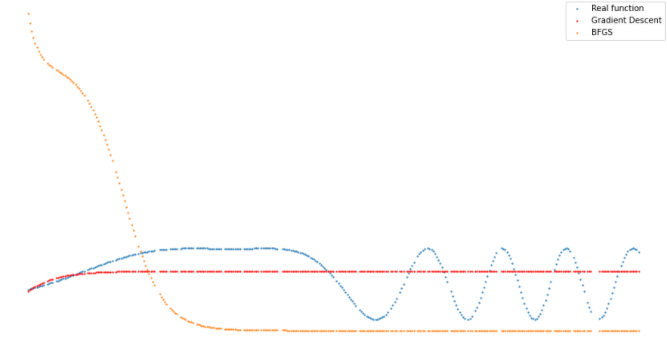
**Этап 2-3:**

Генерируем обучающее множество на основе заданного уравнения для задачи регрессии и разбиваем его на тренировочное и контрольное. Строим нейронную сеть с указанными в задании алгоритмами оптимизации и обучаем их:





Результат обучения на тренировочных данных:



**3) Вывод:**

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 рассмотренная 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.