



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ _____ «Информатика и системы управления»

КАФЕДРА _____ «Теоретическая информатика и компьютерные технологии»

Лабораторная работа № 4

по курсу «Теория искусственных нейронных сетей»

Студент группы ИУ9-72Б Терентьева А. С.

Преподаватель Каганов Ю. Т.

Москва 2023

1 Цель

1. Изучение алгоритмов многомерного поиска 1-го и 2-го порядка.
2. Разработка программ реализации алгоритмов многомерного поиска 1-го и 2-го порядка.
3. Вычисление экстремумов функции.

2 Задание

1. Провести сравнительный анализ современных методов оптимизации (SGD, NAG, Adagrad, ADAM) на примере многослойного персептрона.
2. Применить генетический алгоритма для оптимизации гиперпараметров (число слоев и число нейронов) многослойного персептрона.

3 Реализация

Исходный код программы представлен в листингах 1– 3.

Листинг 1: main.py

```
1 import os
2 import numpy as np
3 from src.methods import gradient, SGD, NAG, Adagrad, Adam, plt
4
5 import gzip
6 import struct
7
8 import random
9
10 n_pixels = 784 # 28*28
11
12 def relu(x):
13     return x if x > 0 else 0
14 def drelu(x):
15     return 1 if x > 0 else 0
16
17 def softmax(xs):
18     maxX = max(xs)
19     exp_values = np.exp(xs - maxX)
```

```

20
21     sum_exp_values = np.sum(exp_values)
22     return [ex / sum_exp_values for ex in exp_values]
23
24 #
25 def mse(y0, y):
26     return 1 / 2 * (y0 - y) ** 2
27 def dmse(y0, y):
28     return y - y0
29     res = []
30     for i in range(len(y)):
31         res.append(sum(y[i] - y0[i]) / len(y[i]))
32     return res
33
34
35 class Layer:
36     def __init__(self, n_neurons, n_input, activation, derivative, lr):
37         self.n_neurons = n_neurons
38         self.n_input = n_input
39         # self.w = np.array([[1/n_input for _ in range(n_input)] for _
40         in range(n_neurons)])
41         # self.w = np.array([[0.01 for _ in range(n_input)] for _ in
42         range(n_neurons)])
43         self.w = np.array([[random.uniform(1 / (2 * n_input), 2 /
44         n_input)
45         for _ in range(n_input)] for _ in range(
46         n_neurons)])
47         self.activation = activation
48         self.derivative = derivative
49         self.XW = []
50         self.out = []
51         self.lr = lr
52         self.prev_grad = []
53         self.vt = np.zeros_like(self.w) # NAG
54         self.LR = np.zeros_like(self.w) # Adagrad
55         self.m = np.zeros_like(self.w) # Adam
56         self.v = np.zeros_like(self.w) # Adam
57
58     def forward(self, x):
59         self.XW = np.dot(self.w, x)
60         if self.activation == softmax:
61             self.out = self.activation(self.XW)
62         else:
63             self.out = np.array([self.activation(xw) for xw in self.XW])
64         return self.out
65

```

```

62
63 class Perceptron:
64     def __init__(self, x_train, y_train, lr = 0.01):
65         self.layers = []
66         self.loss = mse
67         self.dloss = dmse
68         self.out = []
69         self.x_train = x_train
70         self.y_train = y_train
71         self.n_train = len(x_train)
72         self.lastDelta = []
73
74     def add_layer(self, n_neurons, n_input=-1, func_act=relu, dfunc_act=
drelu, lr=0.1):
75         if len(self.layers) == 0:
76             # 0 -
77             l0 = Layer(n_input, 0, func_act, dfunc_act, lr)
78             self.layers.append(l0)
79         n_input = (
80             n_input if n_input > 0 else len(self.layers[-1].w)
81         ) # -
82
83         self.layers.append(Layer(n_neurons, n_input, func_act, dfunc_act
, lr))
84
85     def forward(self, x):
86         self.layers[0].out = x.copy()
87         out = x.copy()
88         i = 1
89         for l in self.layers[1:]:
90             out = l.forward(out)
91             i += 1
92         self.out = out
93         return out
94
95     def change_w_nag(self, k):
96         for l in self.layers[1:]:
97             l.w += k * l.vt * 0.9
98
99     def gradient(self):
100         gradient(self)
101
102     def SGD(self):
103         SGD(self)
104
105     def NAG(self, log=True):
106         NAG(self, log)
107
108     def Adagrad(self):

```

```

105         Adagrad(self)
106     def Adam(self):
107         Adam(self)
108
109     def countErr(self):
110         all = 0
111         for i in range(self.n_train): # n
112
113             e = 0
114             res = self.forward(self.x_train[i])
115             for j in range(10): # 10
116                 e += self.loss(self.y_train[i][j], res[j])
117             e /= 10 #
118             all += e
119         all /= self.n_train
120         return all
121
122     '''def countGradient(self, y):
123         gradient = []
124         for l_i in range(len(self.layers) - 1, 0, -1):
125             l = self.layers[l_i]
126
127             if (l_i == len(self.layers) - 1):
128                 lastDelta = l.out - y
129             else:
130                 l = self.layers[l_i]
131                 sum = np.dot(self.layers[l_i + 1].w.T, lastDelta)
132                 lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for
133                                     j in range(l.n_neurons)])
134
135             gradient.insert(0, np.outer(lastDelta, self.layers[l_i - 1].
136 out))
137         return gradient'''
138
139     def find_answ(res):
140         num = 0
141         min = 1
142         for i in range(len(res)):
143             if abs(1 - res[i]) < min:
144                 min = abs(1 - res[i])
145                 num = i
146         return num
147
148 #
149     def checkCorrectness(self, x = [], y = [], log = True):
150         if len(x) == 0:

```

```

148         x = self.x_train
149         y = self.y_train
150
151         num = len(x)
152         correct_num = 0
153         for i in range(num):
154             res = [0] * 10
155             mas = [0] * 10
156
157             res = self.forward(x[i])
158
159             for j in range(10):
160                 mas[j] = round(res[j], 2)
161
162             predicted = np.argmax(res)
163             expected = np.argmax(y[i])
164             if expected == predicted:
165                 correct_num += 1
166
167             if i > num - 5 and log:
168                 print(mas)
169                 print(y[i])
170                 print(expected, "--->", predicted, "\n")
171
172         res = correct_num / num * 100
173         if log:
174             print(res, "%%% correctness")
175         return res
176
177
178 def create_Y_ans(Y_first):
179     Y_res = []
180     for y in Y_first:
181         mas = np.zeros(10)
182         mas[y] = 1
183         Y_res.append(mas)
184     return np.array(Y_res)
185
186
187 data_folder = os.path.join(os.getcwd(), "data")
188
189 # load compressed MNIST gz files and return numpy arrays
190 def load_data(filename, label=False):
191     with gzip.open(filename) as gz:
192         struct.unpack("I", gz.read(4))
193         n_items = struct.unpack(">I", gz.read(4))

```

```

194         if not label:
195             n_rows = struct.unpack(">I", gz.read(4))[0]
196             n_cols = struct.unpack(">I", gz.read(4))[0]
197             res = np.frombuffer(
198                 gz.read(n_items[0] * n_rows * n_cols), dtype=np.uint8)
199             res = res.reshape(n_items[0], n_rows * n_cols)
200         else:
201             res = np.frombuffer(gz.read(n_items[0]), dtype=np.uint8)
202             res = res.reshape(n_items[0], 1)
203     return res
204
205 # note we also shrink the intensity values (X) from 0-255 to 0-1. This
    helps the model converge faster.
206 X_train = load_data(os.path.join(
207     data_folder, "train-images.gz"), False) / 255.0
208 Y_train = load_data(os.path.join(
209     data_folder, "train-labels.gz"), True).reshape(-1)
210 X_test = load_data(os.path.join(data_folder, "test-images.gz"), False) /
    255.0
211 Y_test = load_data(os.path.join(
212     data_folder, "test-labels.gz"), True).reshape(-1)
213
214 train_len = len(X_train)
215 n_tests = 500
216 X_first = X_train[:n_tests]
217 Y_first = Y_train[:n_tests]
218
219 Y_res = create_Y_ans(Y_first)
220
221 def createPerc(size=1, neuron_num=[10]):
222     perc = Perceptron(X_first, Y_res)
223     perc.neuron_num = neuron_num
224     prev = n_pixels
225     for i in range(size):
226         perc.add_layer(n_input=prev, n_neurons=neuron_num[i], lr=0.01)
227         prev = neuron_num[i]
228     perc.add_layer(n_input=prev, n_neurons=10, func_act=softmax, lr
    =0.01)
229     return perc
230
231 if __name__ == "__main__":
232     perc1 = Perceptron(X_first, Y_res)
233     perc1.add_layer(n_input=n_pixels, n_neurons=10, lr=0.01)
234     perc1.add_layer(n_neurons=10, func_act=softmax, lr=0.01)
235
236

```

```

237
238 W1 = [row.copy() for row in perc1.layers[1].w]
239 W2 = [row.copy() for row in perc1.layers[2].w]
240 perc1.gradient()
241 perc1.checkCorrectness(X_first, Y_res)
242 perc1.layers[1].w = [row[:] for row in W1]
243 perc1.layers[2].w = [row[:] for row in W2]
244 perc1.SGD()
245 perc1.checkCorrectness(X_first, Y_res)
246 perc1.layers[1].w = [row[:] for row in W1]
247 perc1.layers[2].w = [row[:] for row in W2]
248 perc1.NAG()
249 perc1.checkCorrectness(X_first, Y_res)
250 perc1.layers[1].w = [row[:] for row in W1]
251 perc1.layers[2].w = [row[:] for row in W2]
252 perc1.Adagrad()
253 perc1.checkCorrectness(X_first, Y_res)
254 perc1.layers[1].w = [row[:] for row in W1]
255 perc1.layers[2].w = [row[:] for row in W2]
256 perc1.Adam()
257 perc1.checkCorrectness(X_first, Y_res)
258 plt.legend()
259 plt.show()
260
261 '''print(perc1.layers[-2].w)
262 print(perc1.layers[-1].w)'''
263
264 #Y_res2 = create_Y_ans(Y_test)
265 #perc1.checkCorrectness(X_test[:500], Y_res2[:500])

```

Листинг 2: methods.py

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import random
4
5
6 def gradient(perc):
7     x = perc.x_train
8     y = perc.y_train
9
10    epohs = []
11    errors = []
12
13    for step in range(15):
14        if step % 1 == 0:
15            print(step)

```



```

16         epohs.append(step)
17         errors.append(perc.countErr())
18
19         for i in range(perc.n_train): # n
20
21             out = perc.forward(x[i])
22
23             for l_i in range(len(perc.layers) - 1, 0, -1):
24                 l = perc.layers[l_i]
25
26                 if l_i == len(perc.layers) - 1:
27                     #
28                     lastDelta = l.out - y[i]
29                     # lastDelta = np.array([perc.dloss(y[i][j], l.out[j
30                     ]) for j in range(l.n_neurons)])
31                 else:
32                     #
33                     l = perc.layers[l_i]
34                     #
35                     delta
36
37                     sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
38                     lastDelta = np.array([sum[j] * l.derivative(l.XW[j])
39                     for j in range(l.n_neurons)])
40
41                     #gradient = np.dot(np.transpose([lastDelta]), [perc.
42                     layers[l_i - 1].out])
43                     gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
44                     perc.layers[l_i].w -= l.lr * gradient
45
46         plt.plot(epohs, errors, label="gradient")
47         # plt.show()
48
49
50 """
51 def gradient(perc):
52     x = perc.x_train
53     y = perc.y_train
54     num = perc.n_train
55
56     epohs = []
57     errors = []
58
59     for step in range(10):
60         if step % 1 == 0:
61             print(step)
62             epohs.append(step)

```

```

57         errors.append(perc.countErr())
58
59     out = []
60     for i in range(num):
61         perc.layers[0].out = x[i]
62         out.append(perc.forward(x[i]))
63     out = np.array(out)
64
65     for l_i in range(len(perc.layers) - 1, 0, -1):
66         l = perc.layers[l_i]
67
68         if (l_i == len(perc.layers) - 1):
69             #
70             lastDelta = perc.dloss(y.T, out.T)
71             #print(lastDelta)
72             #lastDelta = np.array([perc.dloss(y[i][j], l.out[j]) for
j in range(l.n_neurons)])
73         else:
74             #
75             l = perc.layers[l_i]
76             #
77             sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
78             lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for
j in range(l.n_neurons)])
79
80             #gradient = np.dot(np.transpose([lastDelta]), [perc.layers[
l_i - 1].out])
81             gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
82             perc.layers[l_i].w -= l.lr * gradient
83
84
85     plt.plot(epohs, errors, label=f"{perc.loss.__name__}")
86     #plt.show()
87 """
88
89
90 def SGD(perc):
91     x1 = perc.x_train
92     y1 = perc.y_train
93
94     epohs = []
95     errors = []
96
97     batch_size = 200
98
99     for step in range(15):

```

```

100         if step % 1 == 0:
101             print(step)
102             epohs.append(step)
103             errors.append(perc.countErr())
104
105         # rand_batch = np.random.randint(0, num - batch_size)
106         batch = random.sample(list(zip(x1, y1)), batch_size)
107         for x, y in batch:
108             out = perc.forward(x)
109
110             for l_i in range(len(perc.layers) - 1, 0, -1):
111                 l = perc.layers[l_i]
112
113                 if l_i == len(perc.layers) - 1:
114                     lastDelta = l.out - y
115                 else:
116                     l = perc.layers[l_i]
117                     sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
118                     lastDelta = np.array([sum[j] * l.derivative(l.XW[j])
119 for j in range(l.n_neurons)])
120
121                     gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
122                     perc.layers[l_i].w -= l.lr * gradient
123
124         plt.plot(epohs, errors, label="SGD")
125
126 def NAG(perc, log):
127     x = perc.x_train
128     y = perc.y_train
129
130     epohs = []
131     errors = []
132
133     for step in range(15):
134         if step % 1 == 0 and log:
135             print(step)
136             epohs.append(step)
137             errors.append(perc.countErr())
138
139         for i in range(perc.n_train):
140             perc.change_w_nag(-1)
141             out = perc.forward(x[i])
142             perc.change_w_nag(1)
143
144         for l_i in range(len(perc.layers) - 1, 0, -1):

```

```

145         l = perc.layers[l_i]
146
147         if l_i == len(perc.layers) - 1:
148             lastDelta = l.out - y[i]
149         else:
150             l = perc.layers[l_i]
151             sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
152             lastDelta = np.array([sum[j] * l.derivative(l.XW[j])
for j in range(l.n_neurons)])
153
154         gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
155
156         l.vt += 0.00001 * gradient
157         l.w -= l.vt
158
159     plt.plot(epohs, errors, label="NAG")
160
161
162 def Adagrad(perc):
163     x = perc.x_train
164     y = perc.y_train
165
166     epohs = []
167     errors = []
168
169     for step in range(15):
170         if step % 1 == 0:
171             print(step)
172             epohs.append(step)
173             errors.append(perc.countErr())
174
175         for i in range(perc.n_train):
176             out = perc.forward(x[i])
177
178             for l_i in range(len(perc.layers) - 1, 0, -1):
179                 l = perc.layers[l_i]
180
181                 if l_i == len(perc.layers) - 1:
182                     lastDelta = l.out - y[i]
183                 else:
184                     l = perc.layers[l_i]
185                     sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
186                     lastDelta = np.array([sum[j] * l.derivative(l.XW[j])
for j in range(l.n_neurons)])
187
188             gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)

```

```

189         l.LR += gradient**2
190         perc.layers[l_i].w -= l.lr * gradient / (np.sqrt(l.LR) +
191         1e-8)
192
193     plt.plot(epohs, errors, label="Adagrad")
194
195     beta1 = 0.99
196     beta2 = 0.9
197     def Adam(perc):
198         x = perc.x_train
199         y = perc.y_train
200
201         epohs = []
202         errors = []
203
204         for step in range(15):
205             if step % 1 == 0:
206                 print(step)
207                 epohs.append(step)
208                 errors.append(perc.countErr())
209
210             for i in range(perc.n_train):
211                 out = perc.forward(x[i])
212
213                 for l_i in range(len(perc.layers) - 1, 0, -1):
214                     l = perc.layers[l_i]
215
216                     if l_i == len(perc.layers) - 1:
217                         lastDelta = l.out - y[i]
218                     else:
219                         l = perc.layers[l_i]
220                         sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
221                         lastDelta = np.array([sum[j] * l.derivative(l.XW[j])
222
223                     for j in range(l.n_neurons)])
224
225                     gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
226                     l.m = beta1 * l.m + (1 - beta1) * gradient
227                     l.v = beta2 * l.v + (1 - beta2) * (gradient**2)
228                     m_ = l.m / (1 - beta1**(step + 1))
229                     v_ = l.v / (1 - beta2**(step + 1))
230
231                     perc.layers[l_i].w -= 0.00001 * m_ / (np.sqrt(v_) + 1e
232                     -8)
233
234     plt.plot(epohs, errors, label="Adam")

```

Генетический алгоритм применялся к персептрон с выходным слоем из 10 нейронов, целевая функция - mse, а в качестве гиперпараметров рассматривались количество скрытых слоев и количество нейронов на них. В качестве метода оптимизации использовался метод Нестерова, а скорость обучения составляла 10^{-4} . Функции активации: на выходном слое - Softmax, на скрытых слоях - ReLu. Количество эпох обучения - 15.

Листинг 3: gen.py

```

1 from main import createPerc, plt
2
3 perc1 = createPerc()
4
5 import numpy as np
6 import time
7 import random
8
9 class Individ:
10     def __init__(self, layers_num, neuron_num):
11         self.layers_num = layers_num
12         self.neuron_num = neuron_num
13         self.params = [layers_num] + neuron_num
14         self.p = createPerc(layers_num, neuron_num)
15         self.p.NAG(log=False)
16
17 def createIndivid(params):
18     layers_num = int(params[0])
19     neuron_num = params[1:]
20     neuron_num = neuron_num[:layers_num] + [10] * (layers_num - len(
21         neuron_num))
22     return Individ(layers_num, neuron_num)
23
24 def generate_perc_population(size):
25     population = []
26     for _ in range(size):
27         layers_num = random.randint(1, 10)
28         neuron_num = []
29         for _ in range(layers_num):
30             neuron_num.append(random.randint(1, 30))
31         p = Individ(layers_num, neuron_num)
32         population.append(p)
33     return population
34
35 def F(x):

```

```

36     return x.p.checkCorrectness(log = False)
37
38 #
39 def fitness_function(x):
40     return 1 / F(x)
41
42 #
43 def selection(population, fitness_func, retain_ratio=0.5):
44     fitness_scores = [(fitness_func(ind), ind) for ind in population]
45     sorted_population = [ind for _, ind in sorted(fitness_scores, key=
lambda x: x[0])]
46     retain_length = int(len(sorted_population) * retain_ratio)
47     retain_length = 2 if retain_length < 2 else retain_length
48     parents = sorted_population[:retain_length]
49     return parents
50
51 #
52 def crossover(parents):
53     parents = [p.params for p in parents]
54     children = []
55     while len(children) < len(parents):
56         id1 = random.randint(0, len(parents)-1)
57         id2 = random.randint(0, len(parents)-1)
58         if id1 != id2:
59             parent1 = parents[id1]
60             parent2 = parents[id2]
61             maxlen = min(len(parent1), len(parent2))
62             crossover_point = random.randint(0, maxlen)
63             child1 = parent1[:crossover_point] + parent2[crossover_point
:]
64             child2 = parent2[:crossover_point] + parent1[crossover_point
:]
65             children.extend([createIndivid(child1), createIndivid(child2
)])
66     return children
67
68 #
69 def mutate(params):
70     rand_index = random.randint(0, len(params)-1)
71     params[rand_index] += random.randint(-2, 2)
72     params[rand_index] = max(1, params[rand_index])
73     return createIndivid(params)
74
75 def mutation(children, mutation_chance=0.2):
76     for child in children:
77         if random.random() < mutation_chance:

```

```

78         child = mutate(child.params)
79
80     return children
81
82 xs = []
83 ys = []
84
85 #
86 def genetic_algorithm(population_size, generations, mutation_rate,
87                        crossover_rate):
88     population = generate_perc_population(size=population_size)
89     for i in range(generations):
90         print(i)
91         parents = selection(population, fitness_function, crossover_rate
92                             )
93         offspring = crossover(parents)
94         offspring = mutation(offspring, mutation_rate)
95         population = parents + offspring
96         result = population[np.argmin([fitness_function(x) for x in
97                                       population])]
98         xs.append(i)
99         ys.append(F(result))
100
101     for p in population:
102         print("          :", p.layers_num, "          -          :",
103               , p.neuron_num)
104         print("          :", F(p))
105     best_solution = population[np.argmin([fitness_function(x) for x in
106                                           population])]
107     return best_solution
108
109 start_time = time.time()
110 print("          :")
111 result = genetic_algorithm(population_size=20, generations=10,
112                            mutation_rate=0.2, crossover_rate=0.5)
113 print("          :", time.time() - start_time, "c")
114 plt.plot(xs, ys)
115 plt.show()
116 print("          -          :", result.layers_num, "
117       -          :",
118       result.neuron_num)
119 print("          :", F(result))
120 print()

```


4 Результат работы

```
gradient:  
88.0 %% correctness  
SGD:  
87.2 %% correctness  
NAG:  
97.8 %% correctness  
Adagrad:  
83.6 %% correctness  
Adam:  
87.4 %% correctness
```

Рис. 1 — Результат вычислений: точность каждого метода

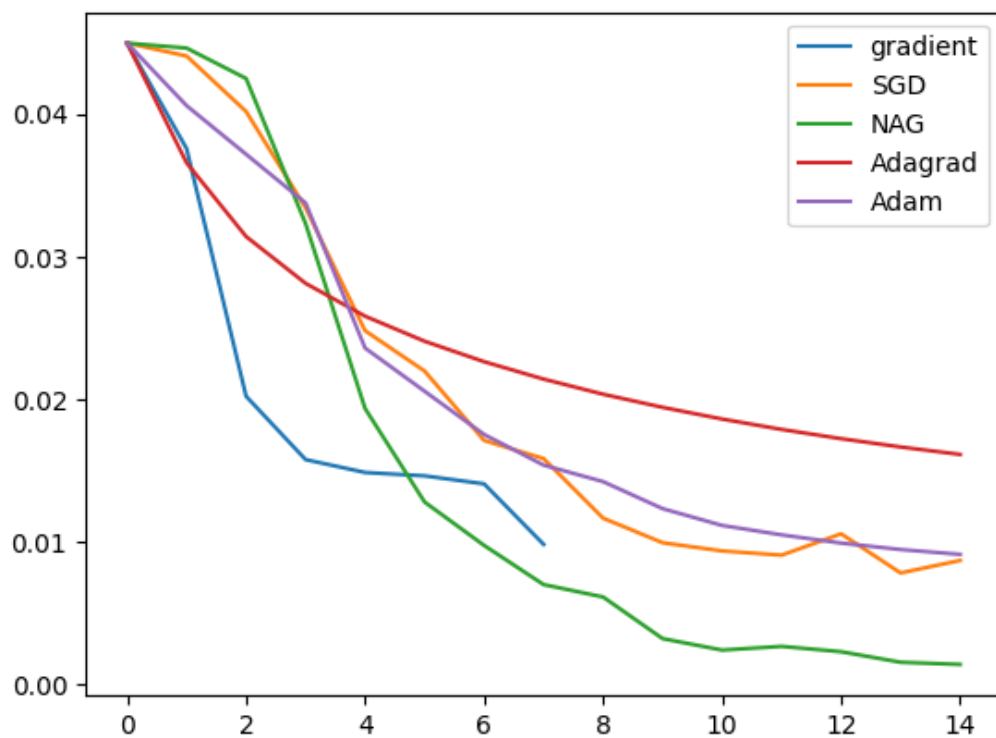


Рис. 2 — Сравнительный график сходимости методов оптимизации: зависимость значения ошибки от количества эпох

```
Время выполнения: 645.2297098636627 с  
Кол-во скрытых слоев: 1 Кол-во нейронов на скрытых слоях: [25]  
Лучший результат: 100.0
```

Рис. 3 — Результат работы генетического алгоритма

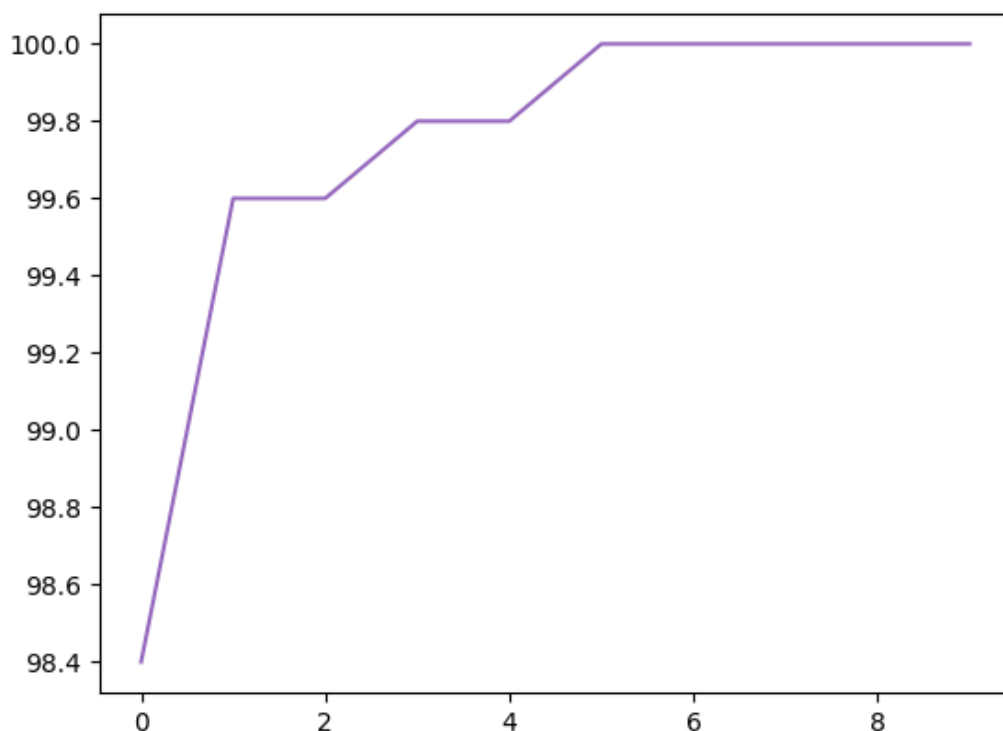


Рис. 4 — График генетического алгоритма:
зависимость точности от кол-ва итераций (поколений)

5 Выводы

В ходе выполнения лабораторной работы был проведен сравнительный анализ современных методов оптимизации на примере многослойного персептрона, была написана их реализация на языке программирования Python.

В ходе эксперимента по исследованию работы программы на основе различных методов оптимизации (SGD, NAG, Adagrad, ADAM), наилучший результат по точности показал метод Нестерова.

По результатам применения генетического алгоритма для оптимизации гиперпараметров многослойного персептрона, был сделан вывод, что в текущей конфигурации лучший результат достигается при 1 скрытом слое и 25-30 нейронах на нем.