



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»**

Студентка группы ИУ9-72Б Самохвалова П. С.

Преподаватель Каганов Ю. Т.

*Москва 2023*

# 1 Задание

1. Сравнительный анализ современных методов оптимизации (SGD, NAG, Adagrad, Adam) на примере многослойного персептрона.
2. Использование генетического алгоритма для оптимизации гиперпараметров (число слоев и число нейронов) многослойного персептрона.

## 2 Практическая реализация

Исходный код программы представлен в листинге 1.

Листинг 1: Методы оптимизации на примере многослойного персептрона, генетический алгоритм

```
1 import random
2 import pickle
3 import gzip
4 import numpy as np
5 from matplotlib import pyplot as plt
6
7
8 def load_data():
9     with gzip.open('../data/mnist.pkl.gz', 'rb') as f:
10         (training_data, validation_data, test_data) = pickle.load(f,
11                               encoding='latin1')
12     return (training_data, validation_data, test_data)
13
14 def load_data_wrapper():
15     tr_d, va_d, te_d = load_data()
16     training_inputs = [np.reshape(x, (784, 1)) for x in tr_d[0]]
17     training_results = [vectorized_result(y) for y in tr_d[1]]
18     training_data = list(zip(training_inputs, training_results))
19     validation_inputs = [np.reshape(x, (784, 1)) for x in va_d[0]]
20     validation_data = list(zip(validation_inputs, va_d[1]))
21     test_inputs = [np.reshape(x, (784, 1)) for x in te_d[0]]
22     test_data = list(zip(test_inputs, te_d[1]))
23     return (training_data, validation_data, test_data)
24
25
26 def vectorized_result(j):
27     e = np.zeros((10, 1))
28     e[j] = 1.0
```

```

29     return e
30
31
32 def sigmoid(x):
33     return 1 / (1 + np.exp(-x))
34
35
36 def sigmoid_der(x):
37     return sigmoid(x) * (1 - sigmoid(x))
38
39
40 def relu(x):
41     x = x.flatten()
42     return np.array([[max(c, 0)] for c in x])
43
44
45 def relu_der(x):
46     x = x.flatten()
47     return np.array([[1 if c > 0 else 0] for c in x])
48
49
50 def softmax(z):
51     e_z = np.exp(z - np.max(z))
52     return e_z / e_z.sum()
53
54
55 def softmax_der(z):
56     s = softmax(z)
57     return np.diag(s) - np.outer(s, s)
58
59
60 def mse(y_true, y_received):
61     return np.linalg.norm(y_true - y_received)
62
63
64 def mse_der(y_true, y_received):
65     return y_true - y_received
66
67
68 def categorical_cross_entropy(y0, y):
69     return -(y0 * np.log(y) + (1 - y0) * np.log(1 - y))
70
71
72 def categorical_cross_entropy_der(y0, y):
73     return -(y0 / y - (1 - y0) / (1 - y))
74

```

```

75
76 def kl_divergence(y0, y):
77     if y0 == 0:
78         return 0
79     else:
80         return y0 * np.log(y0 / y)
81
82
83 def kl_divergence_der(y0, y):
84     return np.log(y0 / y) + 1
85
86
87 class MultilayerPerceptron(object):
88
89     def __init__(self, sizes, optimization_method, activation_function,
90 loss_function):
91         self.num_layers = len(sizes)
92         self.optimization_method = optimization_method
93         self.activation_function = activation_function
94         if activation_function == sigmoid:
95             self.activation_function_der = sigmoid_der
96         elif activation_function == relu:
97             self.activation_function_der = relu_der
98         elif activation_function == softmax:
99             self.activation_function_der = softmax_der
100         self.loss_function = loss_function
101         if loss_function == mse:
102             self.loss_function_der = mse_der
103         elif loss_function == categorical_cross_entropy:
104             self.loss_function_der = categorical_cross_entropy_der
105         elif self.loss_function == kl_divergence:
106             self.loss_function_der = kl_divergence_der
107         self.sizes = sizes
108         self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
109         self.weights = [np.random.randn(y, x) for x, y in zip(sizes
110 [-1], sizes[1:])]
111         self.loss = []
112         if optimization_method == "adagrad":
113             self.G_b = [np.zeros(b.shape) for b in self.biases]
114             self.G_w = [np.zeros(w.shape) for w in self.weights]
115         if self.optimization_method == "adam":
116             self.beta1 = 0.9
117             self.beta2 = 0.999
118             self.epsilon = 1e-8
119             self.m_w = [np.zeros(w.shape) for w in self.weights]
120             self.v_w = [np.zeros(w.shape) for w in self.weights]

```

```

119         self.m_b = [np.zeros(b.shape) for b in self.biases]
120         self.v_b = [np.zeros(b.shape) for b in self.biases]
121         self.t = 0
122         if self.optimization_method == "nag":
123             self.velocity_biases = [np.zeros(b.shape) for b in self.
biases]
124             self.velocity_weights = [np.zeros(w.shape) for w in self.
weights]
125
126         def feedforward(self, a):
127             for b, w in zip(self.biases, self.weights):
128                 a = self.activation_function(np.dot(w, a) + b)
129             return a
130
131         def train(self, training_data, epochs, mini_batch_size, eta,
test_data):
132             if self.optimization_method == "sgd":
133                 self.SGD(training_data, epochs, mini_batch_size, eta,
test_data)
134             elif self.optimization_method == "adagrad":
135                 self.Adagrad(training_data, epochs, mini_batch_size, eta,
test_data)
136             elif self.optimization_method == "adam":
137                 self.Adam(training_data, epochs, mini_batch_size, eta,
test_data)
138             elif self.optimization_method == "nag":
139                 self.NAG(training_data, epochs, mini_batch_size, eta,
test_data)
140
141         def SGD(self, training_data, epochs, mini_batch_size, eta, test_data
):
142             n = len(training_data)
143             for j in range(epochs):
144                 random.shuffle(training_data)
145                 mini_batches = [training_data[k : k+mini_batch_size] for k
in range(0, n, mini_batch_size)]
146                 for mini_batch in mini_batches:
147                     self.update_mini_batch(mini_batch, eta)
148                 self.test(test_data)
149
150         def update_mini_batch(self, mini_batch, eta):
151             nabla_b = [np.zeros(b.shape) for b in self.biases]
152             nabla_w = [np.zeros(w.shape) for w in self.weights]
153             for x, y in mini_batch:
154                 delta_nabla_b, delta_nabla_w = self.backprop(x, y)

```

```

155         nabla_b = [nb + dnb for nb, dnb in zip(nabla_b,
delta_nabla_b)]
156         nabla_w = [nw + dnw for nw, dnw in zip(nabla_w,
delta_nabla_w)]
157         self.weights = [w - (eta / len(mini_batch)) * nw for w, nw in
zip(self.weights, nabla_w)]
158         self.biases = [b - (eta / len(mini_batch)) * nb for b, nb in zip
(self.biases, nabla_b)]
159
160     def backprop(self, x, y):
161         nabla_b = [np.zeros(b.shape) for b in self.biases]
162         nabla_w = [np.zeros(w.shape) for w in self.weights]
163
164         activation = x
165         activations = [x]
166         zs = []
167         for b, w in zip(self.biases, self.weights):
168             z = np.dot(w, activation) + b
169             zs.append(z)
170             activation = self.activation_function(z)
171             activations.append(activation)
172
173         delta = self.loss_function_der(activations[-1], y) * self.
activation_function_der(zs[-1])
174         nabla_b[-1] = delta
175         nabla_w[-1] = np.dot(delta, activations[-2].transpose())
176
177         for l in range(2, self.num_layers):
178             z = zs[-1]
179             sp = self.activation_function_der(z)
180             delta = np.dot(self.weights[-l + 1].transpose(), delta) * sp
181             nabla_b[-l] = delta
182             nabla_w[-l] = np.dot(delta, activations[-l - 1].transpose())
183
184         return (nabla_b, nabla_w)
185
186     def Adagrad(self, training_data, epochs, mini_batch_size, eta,
test_data):
187         n = len(training_data)
188         for j in range(epochs):
189             random.shuffle(training_data)
190             mini_batches = [training_data[k: k + mini_batch_size] for k
in range(0, n, mini_batch_size)]
191             for mini_batch in mini_batches:
192                 self.update_mini_batch_adagrad(mini_batch, eta)
193                 self.test(test_data)

```

```

194
195 def update_mini_batch_adagrad(self, mini_batch, eta):
196     epsilon = 1e-8
197     for x, y in mini_batch:
198         nabla_b, nabla_w = self.backprop(x, y)
199         for l in range(self.num_layers - 1):
200             self.G_b[l] += nabla_b[l] ** 2
201             self.G_w[l] += nabla_w[l] ** 2
202         for l in range(self.num_layers - 1):
203             self.biases[l] -= (eta / (np.sqrt(self.G_b[l] + epsilon)
204 )) * nabla_b[l]
205             self.weights[l] -= (eta / (np.sqrt(self.G_w[l] + epsilon
206 ))) * nabla_w[l]
207
208 def Adam(self, training_data, epochs, mini_batch_size, eta,
209 test_data):
210     n = len(training_data)
211     for j in range(epochs):
212         random.shuffle(training_data)
213         mini_batches = [training_data[k: k + mini_batch_size] for k
214 in range(0, n, mini_batch_size)]
215         for mini_batch in mini_batches:
216             self.update_mini_batch_adam(mini_batch, eta)
217             self.test(test_data)
218
219 def update_mini_batch_adam(self, mini_batch, eta):
220     self.t += 1
221
222     for x, y in mini_batch:
223         delta_nabla_b, delta_nabla_w = self.backprop(x, y)
224         self.m_w = [(self.beta1 * mw + (1 - self.beta1) * dw) for mw
225 , dw in zip(self.m_w, delta_nabla_w)]
226         self.v_w = [(self.beta2 * vw + (1 - self.beta2) * (dw ** 2))
227 for vw, dw in zip(self.v_w, delta_nabla_w)]
228         self.m_b = [(self.beta1 * mb + (1 - self.beta1) * db) for mb
229 , db in zip(self.m_b, delta_nabla_b)]
230         self.v_b = [(self.beta2 * vb + (1 - self.beta2) * (db ** 2))
231 for vb, db in zip(self.v_b, delta_nabla_b)]
232
233         m_w_corrected = [mw / (1 - self.beta1 ** self.t) for mw in self.
234 m_w]
235         v_w_corrected = [vw / (1 - self.beta2 ** self.t) for vw in self.
236 v_w]
237         m_b_corrected = [mb / (1 - self.beta1 ** self.t) for mb in self.
238 m_b]

```

```

228         v_b_corrected = [vb / (1 - self.beta2 ** self.t) for vb in self.
v_b]
229
230         self.weights = [w - (eta / (np.sqrt(vw) + self.epsilon)) *
mw_corr for w, vw, mw_corr in zip(self.weights, v_w_corrected,
m_w_corrected)]
231         self.biases = [b - (eta / (np.sqrt(vb) + self.epsilon)) *
mb_corr for b, vb, mb_corr in zip(self.biases, v_b_corrected,
m_b_corrected)]
232
233     def NAG(self, training_data, epochs, mini_batch_size, eta, test_data
):
234         n = len(training_data)
235         for j in range(epochs):
236             random.shuffle(training_data)
237             mini_batches = [training_data[k : k+mini_batch_size] for k
in range(0, n, mini_batch_size)]
238             for mini_batch in mini_batches:
239                 self.update_mini_batch_nag(mini_batch, eta)
240                 self.test(test_data)
241
242     def update_mini_batch_nag(self, mini_batch, eta, gamma=0.9):
243         nabla_b = [np.zeros(b.shape) for b in self.biases]
244         nabla_w = [np.zeros(w.shape) for w in self.weights]
245         for x, y in mini_batch:
246             delta_nabla_b, delta_nabla_w = self.backprop(x, y)
247             nabla_b = [nb + dnb for nb, dnb in zip(nabla_b,
delta_nabla_b)]
248             nabla_w = [nw + dnw for nw, dnw in zip(nabla_w,
delta_nabla_w)]
249
250         self.velocity_biases = [gamma * vb + (eta / len(mini_batch)) *
nb for vb, nb in zip(self.velocity_biases, nabla_b)]
251         self.velocity_weights = [gamma * vw + (eta / len(mini_batch)) *
nw for vw, nw in zip(self.velocity_weights, nabla_w)]
252
253         self.weights = [w - vw for w, vw in zip(self.weights, self.
velocity_weights)]
254         self.biases = [b - vb for b, vb in zip(self.biases, self.
velocity_biases)]
255
256     def test(self, test_data):
257         n_test = len(test_data)
258         s = 0
259         for i in range(n_test):
260             y_receivied = self.feedforward(test_data[i][0])

```



```

261         y_true = vectorized_result(test_data[i][1])
262         s += self.loss_function(y_true, y_receivied)
263     s /= n_test
264     self.loss.append(s)
265
266     def get_loss(self, i):
267         return self.loss[i]
268
269     def recognize(self, test_example):
270         return np.argmax(self.feedforward(test_example))
271
272
273 def genetic_algorithm(Mp, Np, f):
274
275     population = [[random.uniform(0, 1), random.randint(1, 10)] for _ in
276                   range(Mp)]
277
278     for k in range(Np):
279         fitness = [1 / f(population[i]) for i in range(Mp)]
280         fit = sum(fitness)
281         p = [0] * Mp
282         for i in range(Mp):
283             for j in range(i + 1):
284                 p[i] += fitness[j] / fit
285         p = [0] + p
286         cross = []
287         for i in range(Mp):
288             r = random.uniform(1e-7, 1.0)
289             for j in range(1, Mp + 1):
290                 if p[j - 1] < r <= p[j]:
291                     cross.append(population[j - 1])
292         population_n = []
293         for i in range(Mp):
294             r = random.uniform(1e-7, 1 - 1e-7)
295             new_fraction = r * cross[i][0] + (1 - r) * cross[i][1]
296             new_integer = random.randint(1, 10)
297             population_n.append([new_fraction, new_integer])
298         pm = random.uniform(0.05, 0.2)
299         mutations = []
300         for i in range(Mp):
301             r = random.uniform(0, 1)
302             if r < pm:
303                 mutations.append(population_n[i])
304         for i in range(len(mutations)):
305             index = random.randint(0, 1)
306             if index == 0:

```

```

306         mutations[i][0] = random.uniform(0, 1)
307     else:
308         mutations[i][1] = random.randint(1, 10)
309     if len(mutations) > 0:
310         fitness = [1 / f(population_n[i]) for i in range(Mp)]
311         min_fitness_idx = np.argmin(fitness)
312         population_n[min_fitness_idx] = mutations[random.randint(0,
len(mutations) - 1)]
313     for i in range(Mp):
314         population[i] = population_n[i]
315
316     fitness = [1 / f(population[i]) for i in range(Mp)]
317     max_fitness_idx = np.argmax(fitness)
318     print("Genetic algorithm")
319     print(population[max_fitness_idx])
320
321
322 def function(x):
323     global training_data, test_data
324     eta = x[0]
325     num_neurons = x[1]
326     epochs = 2
327     perceptron = MultilayerPerceptron([784, num_neurons, 10], "sgd",
sigmoid, mse)
328     perceptron.train(training_data, epochs, 10, eta, test_data)
329     return perceptron.get_loss(epochs - 1)
330
331
332 epochs = 100
333 eta = 0.01
334
335
336 training_data, validation_data, test_data = load_data_wrapper()
337
338 perceptron = MultilayerPerceptron([784, 8, 10], "sgd", sigmoid, mse)
339
340 perceptron.train(training_data, epochs, 10, eta, test_data)
341
342 print(perceptron.weights)
343
344 plt.plot([i for i in range(epochs)], [perceptron.get_loss(i) for i in
range(epochs)], label='SGD', color='blue')
345 plt.xlabel('Epochs')
346 plt.ylabel('Loss')
347 plt.title('Loss and Epochs')
348 plt.legend()

```

```

349 plt.show()
350
351 print()
352 for i in range(5):
353     print("Expected:", test_data[i][1])
354     print("Received:", perceptron.recognize(test_data[i][0]))
355     print()
356
357
358 # training_data, validation_data, test_data = load_data_wrapper()
359 #
360 # perceptron = MultilayerPerceptron([784, 8, 10], "adagrad", sigmoid,
361 #                                     mse)
362 #
363 # perceptron.train(training_data, epochs, 10, eta, test_data)
364 #
365 # print(perceptron.weights)
366 #
367 # plt.plot([i for i in range(epochs)], [perceptron.get_loss(i) for i in
368 #                                     range(epochs)], label='Adagrad', color='red')
369 # plt.xlabel('Epochs')
370 # plt.ylabel('Loss')
371 # plt.title('Loss and Epochs')
372 # plt.legend()
373 # plt.show()
374 #
375 # print()
376 # for i in range(5):
377 #     print("Expected:", test_data[i][1])
378 #     print("Received:", perceptron.recognize(test_data[i][0]))
379 #     print()
380 #
381 # training_data, validation_data, test_data = load_data_wrapper()
382 #
383 # perceptron = MultilayerPerceptron([784, 8, 10], "adam", sigmoid, mse)
384 #
385 # perceptron.train(training_data, epochs, 10, eta, test_data)
386 #
387 # print(perceptron.weights)
388 #
389 # plt.plot([i for i in range(epochs)], [perceptron.get_loss(i) for i in
390 #                                     range(epochs)], label='Adam', color='green')
391 # plt.xlabel('Epochs')
392 # plt.ylabel('Loss')
393 # plt.title('Loss and Epochs')

```

```

392 # plt.legend()
393 # plt.show()
394 #
395 # print()
396 # for i in range(5):
397 #     print("Expected:", test_data[i][1])
398 #     print("Receivied:", perceptron.recognize(test_data[i][0]))
399 #     print()
400
401
402 # training_data, validation_data, test_data = load_data_wrapper()
403 #
404 # perceptron = MultilayerPerceptron([784, 8, 10], "nag", sigmoid, mse)
405 #
406 # perceptron.train(training_data, epochs, 10, eta, test_data)
407 #
408 # print(perceptron.weights)
409 #
410 # plt.plot([i for i in range(epochs)], [perceptron.get_loss(i) for i in
411 #     range(epochs)], label='NAG', color='brown')
412 # plt.xlabel('Epochs')
413 # plt.ylabel('Loss')
414 # plt.title('Loss and Epochs')
415 # plt.legend()
416 # plt.show()
417 #
418 # print()
419 # for i in range(5):
420 #     print("Expected:", test_data[i][1])
421 #     print("Receivied:", perceptron.recognize(test_data[i][0]))
422 #     print()
423
424 # Mp = 3
425 # Np = 5
426 #
427 # training_data, validation_data, test_data = load_data_wrapper()
428 #
429 # genetic_algorithm(Mp, Np, function)

```

### 3 Результаты

Результаты работы программы представлены на рисунках 1- 2.

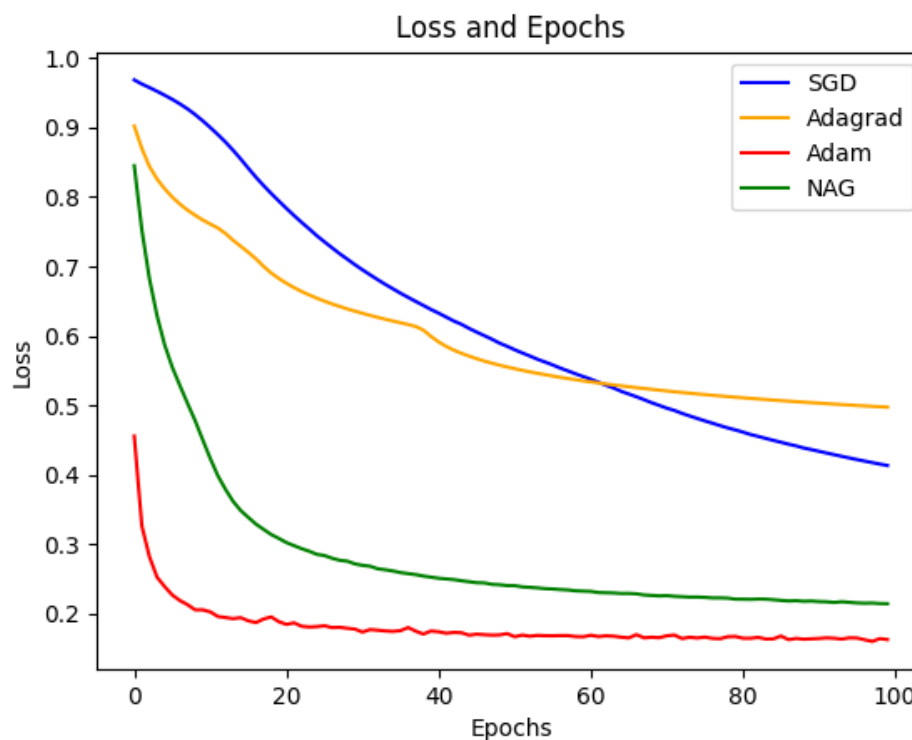


Рис. 1 — Результат работы методов оптимизации SGD, Adagrad, Adam, NAG для многослойного персептрона

```
Genetic algorithm
[0.180427542823264, 9]
```

Рис. 2 — Результат работы генетического алгоритма для оптимизации гиперпараметров многослойного персептрона

## 4 Выводы

В результате выполнения лабораторной работы были реализованы методы оптимизации SGD, Adagrad, Adam, NAG для многослойного персептрона, был реализован генетический алгоритм для оптимизации гиперпараметров многослойного персептрона.