



**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»
«Сравнительный анализ современных методов оптимизации (SGD,
NAG, Adagrad, ADAM) на примере многослойного персептрона.
Использование генетического алгоритма для оптимизации
гиперпараметров многослойного персептрона»

Студент группы ИУ9-72Б Шевченко К.

Преподаватель Каганов Ю. Т.

Москва, 2023

Цель работы

1. Исследовать работу многослойного персептрона с использованием современных методов оптимизации (SGD, NAG, Adagrad, Adam).
2. Осуществить оптимизацию гиперпараметров (число слоёв, число нейронов) многослойного персептрона с помощью генетического алгоритма.

Постановка задачи

1. Требуется взять за основу многослойный персептрон, реализованный в рамках лабораторной работы №2, и осуществить его обучение на примере данных из MNIST Dataset, используя следующие оптимизаторы:
 - SGD (Stochastic gradient descent);
 - NAG (Nesterov accelerated gradient);
 - Adagrad (Adapted gradient);
 - Adam (Adaptive moment estimation).
2. Подготовить графики результатов исследования и проанализировать полученные результаты.
3. Реализовать генетический алгоритм для оптимизации гиперпараметров (число слоёв, число нейронов) многослойного персептрона.
4. Сделать выводы о настройке гиперпараметров нейронной сети при помощи генетического алгоритма.

Реализация

Исходный код программы приведён в листингах 1–10.

Листинг 1: Импорт необходимых зависимостей

```
1 import pickle
2 import matplotlib.pyplot as plt
3 import numpy as np
4 from copy import deepcopy
5 import random
6 from typing import Callable
7 import time
```

Листинг 2: Загрузка базы данных MNIST

```
1 def load_mnist(path):
2     with open(path, mode='rb') as f:
3         training_data, _, test_data = pickle.load(f, encoding='bytes')
4         return dict(
5             training_images = training_data[0],
6             training_labels = training_data[1],
7             test_images = test_data[0],
8             test_labels = test_data[1]
9         )
```

Листинг 3: Рисование цифры MNIST

```
1 def draw_mnist_digit(mnist, example):
2     label = mnist['training_labels'][example]
3     plt.title('Example: %d, label: %d' % (example, label))
4     plt.imshow(np.array(mnist['training_images'][7]).reshape((28, 28)),
5         ↪ cmap=plt.get_cmap('gray'))
```

Листинг 4: Стохастический градиентный спуск (SGD)

```
1 class SGD():
2     def __init__(self, learning_rate):
3         self.learning_rate = learning_rate
4
5     def init_network(self, network):
6         self.network = network
7
8     def update(self, grad_w, grad_b, batch, expected):
9         for i in range(len(self.network.layers) - 1, 0, -1):
10             self.network.weights[i] -= self.learning_rate * grad_w[i]
11             self.network.biases[i] -= self.learning_rate * grad_b[i]
12
13     def __str__(self):
14         return 'SGD'
```

Листинг 5: Ускоренные градиенты Нестерова (NAG)

```
1 class NAG():
2     def __init__(self, learning_rate, gamma):
3         self.learning_rate = learning_rate
4         self.gamma = gamma
5
6     def init_network(self, network):
7         self.network = network
8         self.vt_w = [None] + [np.zeros_like(self.network.weights[i]) for i in
9             ↪ range(1, len(self.network.weights))]
9         self.vt_b = [None] + [np.zeros_like(self.network.biases[i]) for i in
10             ↪ range(1, len(self.network.biases))]
10
```

```

11 def update(self, grad_w, grad_b, batch, expected):
12     network_predicted = deepcopy(self.network)
13     for i in range(1, len(self.network.layers)):
14         network_predicted.weights[i] -= self.gamma * self.vt_w[i]
15         network_predicted.biases[i] -= self.gamma * self.vt_b[i]
16
17     weighted_inputs, predicted =
18         ↪ network_predicted.feedforward(np.transpose(batch))
19     errors = network_predicted.backpropagate_error(weighted_inputs,
20         ↪ predicted[-1], expected)
21     grad_w_predicted = [None for _ in range(len(network_predicted.layers))]
22     grad_b_predicted = [None for _ in range(len(network_predicted.layers))]
23     for i in range(len(network_predicted.layers) - 1, 0, -1):
24         grad_w_predicted[i] = np.dot(errors[i], np.transpose(predicted[i -
25             ↪ 1]))) / batch.shape[0]
26         grad_b_predicted[i] = errors[i].sum(axis=1) / batch.shape[0]
27
28     for i in range(1, len(self.network.layers)):
29         self.vt_w[i] = self.gamma * self.vt_w[i] + self.learning_rate *
30             ↪ grad_w_predicted[i]
31         self.vt_b[i] = self.gamma * self.vt_b[i] + self.learning_rate *
32             ↪ grad_b_predicted[i]
33         self.network.weights[i] -= self.vt_w[i]
34         self.network.biases[i] -= self.vt_b[i]
35
36 def __str__(self):
37     return 'NAG'

```

Листинг 6: Адаптивный градиент (Adagrad)

```

1 class Adagrad():
2     def __init__(self, learning_rate):
3         self.learning_rate = learning_rate
4
5     def init_network(self, network):
6         self.network = network
7         self.G_w = [None] + [np.zeros_like(self.network.weights[i]) for i in
8             ↪ range(1, len(self.network.weights))]
9         self.G_b = [None] + [np.zeros_like(self.network.biases[i]) for i in
10             ↪ range(1, len(self.network.biases))]
11
12     def update(self, grad_w, grad_b, batch, expected):
13         for i in range(1, len(self.network.layers)):
14             self.G_w[i] += grad_w[i] ** 2
15             self.G_b[i] += grad_b[i] ** 2
16             adapted_learning_rate_w = self.learning_rate / np.sqrt(self.G_w[i]
17                 ↪ + 1e-8)
18             adapted_learning_rate_b = self.learning_rate / np.sqrt(self.G_b[i]
19                 ↪ + 1e-8)
20             self.network.weights[i] -= adapted_learning_rate_w * grad_w[i]
21             self.network.biases[i] -= adapted_learning_rate_b * grad_b[i]
22
23     def __str__(self):
24         return 'Adagrad'

```

Листинг 7: Адаптивная оценка моментов (Adam)

```
1 class Adam():
2     def __init__(self, learning_rate, beta1, beta2):
3         self.learning_rate = learning_rate
4         self.beta1 = beta1
5         self.beta2 = beta2
6
7     def init_network(self, network):
8         self.network = network
9         self.m_w = [None] + [np.zeros_like(self.network.weights[i]) for i in
10                                ↪ range(1, len(self.network.weights))]
11         self.v_w = [None] + [np.zeros_like(self.network.weights[i]) for i in
12                                ↪ range(1, len(self.network.weights))]
13         self.m_b = [None] + [np.zeros_like(self.network.biases[i]) for i in
14                                ↪ range(1, len(self.network.biases))]
15         self.v_b = [None] + [np.zeros_like(self.network.biases[i]) for i in
16                                ↪ range(1, len(self.network.biases))]
17         self.epoch = 1
18
19     def update(self, grad_w, grad_b, batch, expected):
20         for i in range(1, len(self.network.layers)):
21             self.m_w[i] = self.beta1 * self.m_w[i] + (1 - self.beta1) *
22                 ↪ grad_w[i]
23             self.m_b[i] = self.beta1 * self.m_b[i] + (1 - self.beta1) *
24                 ↪ grad_b[i]
25             self.v_w[i] = self.beta2 * self.v_w[i] + (1 - self.beta2) *
26                 ↪ grad_w[i] ** 2
27             self.v_b[i] = self.beta2 * self.v_b[i] + (1 - self.beta2) *
28                 ↪ grad_b[i] ** 2
29
30             m_w_i = self.m_w[i] / (1 - self.beta1 ** self.epoch)
31             m_b_i = self.m_b[i] / (1 - self.beta1 ** self.epoch)
32             v_w_i = self.v_w[i] / (1 - self.beta2 ** self.epoch)
33             v_b_i = self.v_b[i] / (1 - self.beta2 ** self.epoch)
34
35             self.network.weights[i] -= self.learning_rate * m_w_i /
36                 ↪ (np.sqrt(v_w_i) + 1e-8)
37             self.network.biases[i] -= self.learning_rate * m_b_i /
38                 ↪ (np.sqrt(v_b_i) + 1e-8)
39         self.epoch += 1
40
41     def __str__(self):
42         return 'Adam'
```

Листинг 8: Функции потерь

```
1 class MeanSquaredError():
2     # средняя квадратичная ошибка
3     @staticmethod
4     def calculate(expected, predicted):
5         return np.mean((expected - predicted) ** 2)
6
7 class CategoricalCrossEntropy():
8     # категориальная перекрёстная энтропия
```

```

9     @staticmethod
10    def calculate(expected, predicted):
11        return -np.sum(expected * np.log(predicted + 1e-100))
12
13    class KLDivergence():
14        # дивергенция Кульбака-Лейблера
15        @staticmethod
16        def calculate(expected, predicted):
17            return np.sum(expected * np.log((expected + 1e-100) / (predicted +
18                ↪ 1e-100)))

```

Листинг 9: Класс для представления многослойного персептрона

```

1    class Perceptron:
2        def __init__(self, layers, optimizer):
3            # layers-- это кортеж, каждый элемент которого
4            # представляет количество нейронов в соответствующем слое
5            self.layers = layers
6            self.weights = [None]
7            self.biases = [None]
8            # инициализация весов и bias-ов в каждом слое
9            # для входного слоя веса и bias-ы не нужны
10           for i in range(1, len(self.layers)):
11               self.weights.append(np.random.normal(0.0, np.sqrt(2.0 /
12                   ↪ (self.layers[i - 1] + self.layers[i])), size=(self.layers[i],
13                   ↪ self.layers[i - 1])))
14               self.biases.append(np.random.normal(0.0, np.sqrt(2.0 /
15                   ↪ (self.layers[i - 1] + self.layers[i])), size=self.layers[i]))
16           self.optimizer = optimizer
17           self.optimizer.init_network(self)
18
19       @staticmethod
20       def relu(x):
21           #return 1 / (1 + np.exp(np.clip(-x, a_min=-100, a_max=100)))
22           return np.vectorize(lambda x: x if x >= 0 else 0)(x)
23
24       @staticmethod
25       def softmax(x):
26           return np.exp(x - max(x)) / np.sum(np.exp(x - max(x)))
27
28       def activate(self, x, activation):
29           if activation not in ('relu', 'softmax'):
30               raise Exception('activation should be "relu" or "softmax"')
31           if activation == 'relu':
32               return self.relu(x)
33           return np.apply_along_axis(self.softmax, 0, x)
34
35       def activation_func_derivative(self, x, activation):
36           if activation not in ('relu', 'softmax'):
37               raise Exception('activation should be "relu" or "softmax"')
38           if activation == 'relu':
39               #return self.activate(x, 'relu') * (1 - self.activate(x, 'relu'))
40               return np.vectorize(lambda x: 1 if x >= 0 else 0)(x)
41           return self.activate(x, 'softmax') * (1 - self.activate(x, 'softmax'))
42
43       def feedforward(self, input):

```

```

41     weighted_inputs, outputs = [None], [input]
42     for i in range(1, len(self.layers)):
43         weighted_input = np.dot(self.weights[i], outputs[i - 1]) +
44             ↪ self.biases[i].reshape(self.layers[i], 1)
45         output = self.activate(weighted_input, 'relu') if i !=
46             ↪ len(self.layers) - 1 else self.activate(weighted_input,
47             ↪ 'softmax')
48         weighted_inputs.append(weighted_input)
49         outputs.append(output)
50     return (weighted_inputs, outputs)
51
52 def train(self, training_data, test_data, epochs, batch_size, loss_func,
53     ↪ is_loss_funcs_plot_needed):
54     self.loss_func = loss_func
55     training_data = list(zip(training_data['training_images'],
56     ↪ training_data['training_expected']))
57     x, y = [], []
58     for epoch in range(1, epochs + 1):
59         loss = 0
60         np.random.shuffle(training_data)
61         training_images, training_expected = zip(*training_data)
62         training_images, training_expected = np.array(training_images),
63             ↪ np.array(training_expected)
64         n = 0
65         for i in range(0, training_images.shape[0], batch_size):
66             batch = training_images[i: i + batch_size]
67             expected = np.transpose(training_expected[i: i + batch_size])
68             weighted_inputs, predicted =
69                 ↪ self.feedforward(np.transpose(batch))
70             errors = self.backpropagate_error(weighted_inputs,
71                 ↪ predicted[-1], expected)
72             grad_w, grad_b = [None for _ in range(len(self.layers))], [None
73                 ↪ for _ in range(len(self.layers))]
74             for i in range(len(self.layers) - 1, 0, -1):
75                 grad_w[i] = np.dot(errors[i], np.transpose(predicted[i -
76                 ↪ 1])) / batch_size
77                 grad_b[i] = errors[i].sum(axis=1) / batch_size
78             self.optimizer.update(grad_w, grad_b, batch, expected)
79             loss += loss_func.calculate(expected=expected,
80                 ↪ predicted=predicted[-1]) / batch_size
81             n += 1
82         loss /= n
83         score, _ = self.predict(test_data=test_data)
84         # print(f'Epoch {epoch}\{epochs}\nloss: {loss: .2f}, accuracy:
85             ↪ {score: .2f}%')
86         x.append(epoch)
87         y.append(loss)
88     self.loss_func_values = y
89     if is_loss_funcs_plot_needed:
90         self.plot_loss_funcs(x, y)
91
92 def backpropagate_error(self, weighted_inputs, predicted_data,
93     ↪ expected_data):
94     errors = [None for _ in self.layers]
95     # errors[-1] = 2 * (predicted_data - expected_data) /
96     ↪ predicted_data.shape[1] ##
97     ↪ self.activation_func_derivative(weighted_inputs[-1], 'relu')

```

```

83     errors[-1] = predicted_data - expected_data
84     for i in range(len(self.layers) - 2, 0, -1):
85         errors[i] = np.dot(np.transpose(self.weights[i + 1]), errors[i +
            ↪ 1]) * \
86             self.activation_func_derivative(weighted_inputs[i], 'relu')
87     return errors
88
89     def predict(self, test_data):
90         score, loss, n = 0, 0, 0
91         for test_image, test_expected in zip(test_data['test_images'],
            ↪ test_data['test_expected']):
92             predicted = self.feedforward(test_image.reshape(784, 1))[1][-1]
93             predicted_label = np.argmax(predicted)
94             expected_label = np.argmax(test_expected)
95             score = score + 1 if predicted_label == expected_label else score
96             loss += self.loss_func.calculate(expected=test_expected,
            ↪ predicted=predicted)
97             n += 1
98         return score / n * 100, loss / test_data['test_images'].shape[0]
99
100     def plot_loss_funcs(self, x, y):
101         plt.xlabel('Epoch')
102         plt.ylabel('Loss')
103         plt.plot(x, y, label='cross_entropy_loss', c='red')
104         plt.legend(loc='upper right')
105         plt.show()
106
107     def get_loss_values(self):
108         return self.loss_func_values

```

Листинг 10: Генетический алгоритм для оптимизации гиперпараметров многослойного персептрона

```

1  epochs = 100
2
3  training_data['training_images'] = training_data['training_images'][:2000]
4  training_data['training_expected'] = training_data['training_expected'][:2000]
5  test_data['test_images'] = test_data['test_images'][:2000]
6  test_data['test_expected'] = test_data['test_expected'][:2000]
7
8  def create_neural_network(genome):
9      # genome[0] - количество слоев, genome[1] - количество нейронов в
            ↪ промежуточном слое
10     network = Perceptron(layers=[784] + genome[0] * [genome[1]] + [10],
            ↪ optimizer=SGD(0.01))
11     return network
12
13     def train_and_evaluate(genome):
14         network = create_neural_network(genome)
15         network.train(
16             training_data=training_data,
17             test_data=test_data,
18             epochs=epochs,
19             batch_size=70,
20             loss_func=CategoricalCrossEntropy(),
21             is_loss_funcs_plot_needed=False

```



```

22     )
23     accuracy, _ = network.predict(test_data=test_data)
24     return accuracy
25
26 def fitness_func(genome):
27     start_time = time.time()
28     accuracy = train_and_evaluate(genome)
29     end_time = time.time()
30     completion_time = end_time - start_time
31
32     return accuracy / 10 if accuracy < 74 else 10 + (accuracy - 74) ** 4 -
        ↪ (completion_time / 60)
33
34 def genetic_algorithm() -> tuple[np.ndarray, float]:
35     t, Np, k, Mp = 0, 4, 1, 10
36     # размер генома
37     n = 2
38     # формируем исходную популяцию с количеством особей Mp
39     population = []
40     Mp = 0
41     for i in range(1, 3):
42         for j in range(1, 5):
43             Mp += 1
44             population.append([i, j])
45
46     population = np.array(population)
47     fitness_values_individuals_hist = {}
48     fitness_values_individuals = []
49
50     for genome, fitness_value in zip(population,
        ↪ np.array(list(map(fitness_func, population)))):
51         fitness_values_individuals_hist[str(genome)] = fitness_value
52         fitness_values_individuals.append(fitness_value)
53
54     # print(population)
55     # print(fitness_values_individuals_hist)
56
57     fitness_value_population = np.sum(fitness_values_individuals)
58
59     while t != Np:
60         k = 1
61         while k < Mp:
62             # этап селекции (отбора особей) для последующего их скрещивания
63             q = fitness_values_individuals / fitness_value_population
64             new_population = [population[np.random.choice(len(population),
        ↪ p=q)] for _ in range(Mp)]
65
66             # этап скрещивания
67             Pc = 0.5
68             parent_indices = []
69             for i in range(len(new_population)):
70                 r = np.random.random()
71                 if r < Pc:
72                     parent_indices.append(i)
73
74             for parent1_index, parent2_index in zip(parent_indices[0::2],
        ↪ parent_indices[1::2]):

```

```

75         c = np.random.random()
76         parent1, parent2 = new_population[parent1_index],
           ↪ new_population[parent2_index]
77         parent1 = c * parent1 + (1 - c) * parent2
78         parent2 = (1 - c) * parent1 + c * parent2
79
80         # эман мутацuu
81         Pm = 0.2
82         for i in range(len(new_population)):
83             r = np.random.random()
84             if r < Pm:
85                 p = np.random.randint(0, n, 1)
86                 if p == 0:
87                     new_population[i][p] = np.random.randint(1, 16)
88                 else:
89                     new_population[i][p] = np.random.randint(1, 33)
90
91         population[:] = new_population
92
93         fitness_values_individuals = []
94         for genome in population:
95             if fitness_values_individuals_hist.get(str(genome)) is None:
96                 fitness_values_individuals_hist[str(genome)] =
           ↪ fitness_func(genome)
97
           ↪ fitness_values_individuals.append(fitness_values_individuals_hist[genome])
98
99         fitness_value_population = np.sum(fitness_values_individuals)
100         k += 1
101         t += 1
102         # print(list(zip(population, fitness_values_individuals)))
103         best_individual = population[np.argmax(fitness_values_individuals)]
104         return best_individual,
           ↪ fitness_values_individuals[np.argmax(fitness_values_individuals)]
105
106 x, y = genetic_algorithm()
107 print(f'Best individual: {x}')

```

Тестирование

Исходный код для тестирования разработанной программы представлен в листинге 11.

Листинг 11: Тестирование

```

1 optimizers = (
2     SGD(0.01),
3     NAG(0.01, 0.9),
4     Adagrad(0.01),
5     Adam(0.01, 0.9, 0.99)
6 )
7
8 y_s = list()

```

```

9  for optimizer in optimizers:
10     perceptron = Perceptron(layers=(784, 18, 10), optimizer=optimizer)
11     perceptron.train(
12         training_data=training_data,
13         test_data=test_data,
14         epochs=epochs,
15         batch_size=70,
16         loss_func=CategoricalCrossEntropy(),
17         is_loss_funcs_plot_needed=False
18     )
19     y_s.append(perceptron.get_loss_values())
20     score, _ = perceptron.predict(test_data=test_data)
21     print(f'Optimization method: {str(optimizer)}')
22     print(f'Accuracy of guessed numbers: {score}%')
23
24 for i, optimizer in enumerate(optimizers):
25     x, y = range(1, epochs + 1), y_s[i]
26     plt.plot(x, y, label=str(optimizer))
27 plt.legend(bbox_to_anchor=(1, 1), loc='upper left')
28 plt.show()

```

Тестирование проводилось путём обучения многослойного персептрона на 10 эпохах. В качестве функции активации нейронов использовалась функция *ReLU*, а на последнем слое – функция *Softmax* для получения вероятностей принадлежности объектов, соответствующих изображениям, классам цифр. В качестве функции потерь применялась категориальная перекрёстная энтропия (categorical cross entropy), хорошо показавшая себя для обучения персептрона в рамках лабораторной работы №2.

В данной работе обучение персептрона осуществлялось на каждой итерации не на всём наборе данных из *MNIST dataset*, а путём обработки за одну итерацию относительно небольшого набора обучающих данных из всего набора данных – пакета (*batch*), размер которого в данной работе выбран равным 70. Использование пакетной обработки позволило избежать недостатка традиционного градиентного метода, увеличив скорость обучения нейронной сети, уменьшив затраты оперативной памяти, а также вероятность попадания в локальный минимум в процессе спуска.

Рисунок 1 демонстрирует результаты обучения многослойного персептрона (точность распознавания цифр на тестовой выборке) при использовании современных методов оптимизации (SGD, NAG, Adagrad, Adam).

```
Optimization method: SGD
Accuracy of guessed numbers: 92.07%
Optimization method: NAG
Accuracy of guessed numbers: 95.09%
Optimization method: Adagrad
Accuracy of guessed numbers: 93.63%
Optimization method: Adam
Accuracy of guessed numbers: 95.26%
```

Рисунок 1 — Результаты обучения многослойного персептрона с использованием современных методов оптимизации

На рисунке 2 представлены графики зависимости функции потерь от числа эпох при использовании различных методов оптимизации.

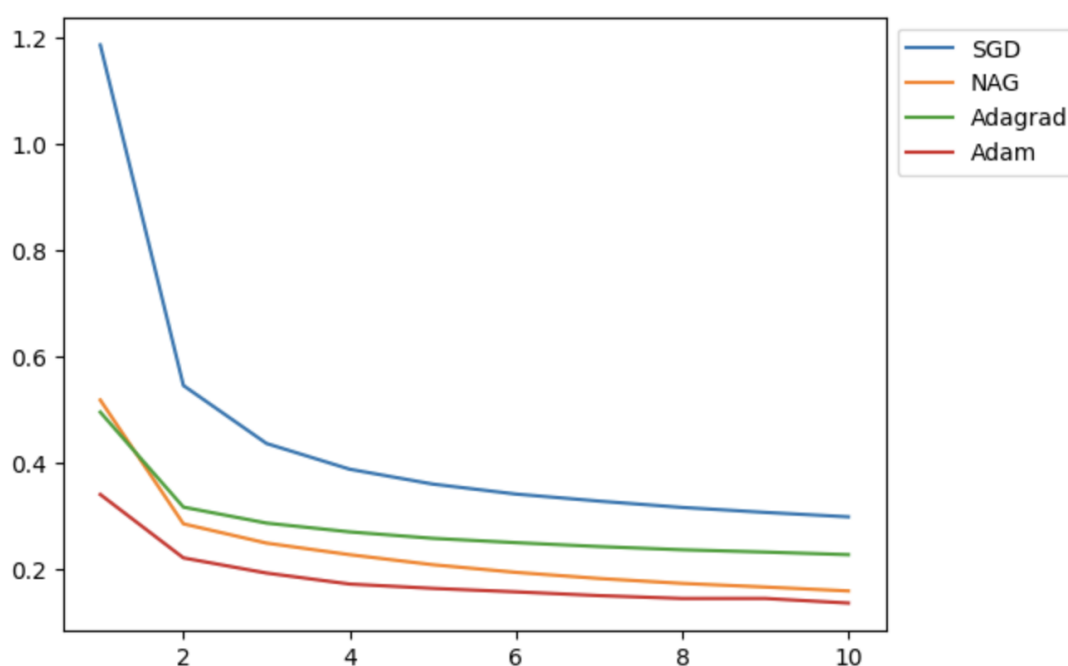


Рисунок 2 — Графики зависимости функции потерь от числа эпох при использовании различных методов оптимизации

Стохастический градиентный спуск (SGD) показал худшие результаты среди рассмотренных методов оптимизации, что оказалось вполне ожидаемо. Вторым после SGD оказался алгоритм Adagrad, суть которого заключается в адаптивном выборе шага обучения. Данный метод оптимизации является крайне эффективным при обучении сложных нейронных сетей, состоящих из множества промежуточных слоёв и нейронов, поскольку позволяет адаптировать скорость изменения каждого параметра. Хорошие

результаты показал метод ускоренных градиентов Нестерова (NAG), позволивший увеличить шаг градиентного спуска путём учёта сдвигов на предыдущих итерациях. Наибольшую же эффективность показал метод адаптивной оценки моментов (Adam), что также оказалось ожидаемым результатом, поскольку в основе данного метода лежат алгоритмы NAG и Adagrad. То есть Adam сочетает в себе учёт импульса на предыдущих итерациях (NAG) и адаптивный шаг обучения (Adagrad), что и приводит к высокой точности обучения.

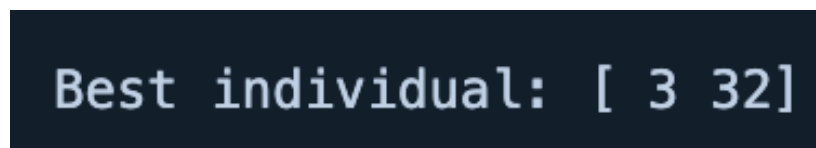
Тестирование генетического алгоритма для оптимизации гиперпараметров многослойного персептрона (число слоёв и число нейронов) проводилось следующим образом:

1. формировалась начальная популяция из 15 особей. Геном каждой особи представлен двумя генами. Первый ген отвечал за число слоёв, второй – за число нейронов в слое;
2. для каждой особи происходило создание и обучение нейронной сети с соответствующими гиперпараметрами;
3. оценивалась приспособленность каждой особи при помощи функции фитнеса, а также приспособленность всей популяции в целом;
4. применялись генетические операторы: отбор, селекция (вероятность 0.5), мутация (вероятность 0.2).

Функция фитнеса $f(accuracy, time)$ генетического алгоритма была подобрана таким образом, чтобы учитывалась не только точность предсказания, но и время обучения нейронной сети:

$$f(accuracy, time) = \begin{cases} \frac{accuracy}{10}, & accuracy < 74, \\ 10 + (accuracy - 74)^4 - (\frac{time}{60}). \end{cases} \quad (1)$$

Результат работы генетического алгоритма представлен на рисунке 3.



Best individual: [3 32]

Рисунок 3 — Результат работы генетического алгоритма

С увеличением размера популяции и итераций возможно добиться более высокой точности работы алгоритма. Однако такой подход сильно затратен по времени и

используемым ресурсам. Хорошего результата удалось добиться, ограничившись размером популяции, равным 15.

Ниже на рисунке 4 представлен результат обучения в виде точности предсказания с использованием полученных гиперпараметров многослойного персептрона. Число эпох по-прежнему равно 10.

Accuracy of guessed numbers: 92.33%

Рисунок 4 — Результат обучения многослойного персептрона с использованием гиперпараметров, полученных путём оптимизации с помощью генетического алгоритма

Также на рисунке 5 представлен соответствующий гиперпараметрам график зависимости функции потерь от числа эпох обучения.

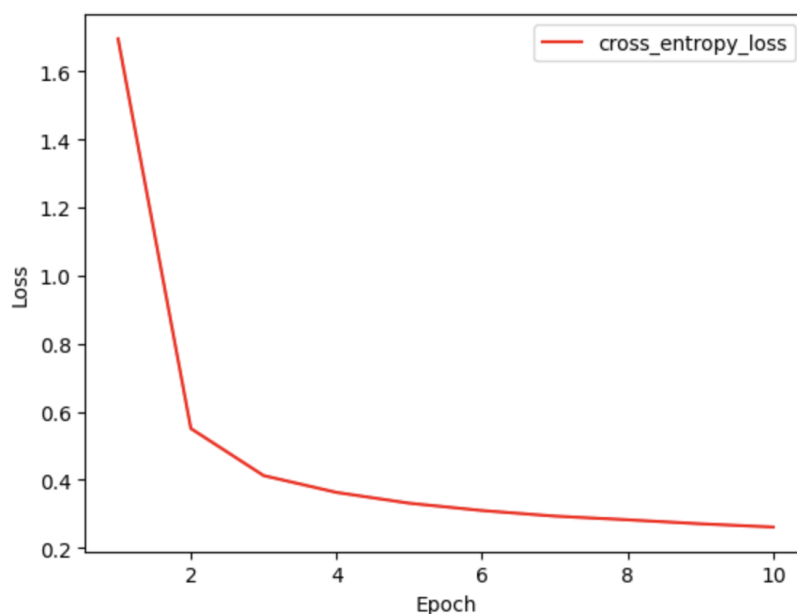


Рисунок 5 — График зависимости функции потерь от числа эпох обучения многослойного персептрона (гиперпараметры, оптимизированные генетическим алгоритмом)

Выводы

В ходе выполнения лабораторной работы были реализованы современные методы оптимизации на примере многослойного персептрона, разработанного в лабораторной

работе №2, а также осуществлена оптимизация гиперпараметров (число слоёв, число нейронов) с помощью генетического алгоритма.

Полученные результаты позволяют сделать следующие выводы:

1. Все рассмотренные современные методы оптимизации дают хорошие результаты обучения, однако самым эффективным является метод адаптивной оценки моментов (Adam) в силу сочетания в себе сразу двух алгоритмов оптимизации: NAG и Adagrad;
2. Генетический алгоритм для оптимизации гиперпараметров многослойного персептрона показал хорошие результаты работы, которые можно улучшить, увеличив размер популяции и число итераций алгоритма. Однако даже при рассмотренных в работе параметрах получение результатов потребовало больших вычислительных ресурсов и заняло длительное время. С увеличением числа слоёв нейронной сети при фиксированном количестве эпох требуется больше итераций на обучение, поэтому для более точной оптимизации хорошим тоном является оптимизация большего числа гиперпараметров (число эпох, функция активации, функция потерь, размер пакетов, оптимизатор).