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

```
import os
import numpy as np
from src.methods import gradient, SGD, NAG, Adagrad, Adam, plt

import gzip
import struct

import random

n_pixels = 784  # 28*28

def relu(x):
    return x if x > 0 else 0
def drelu(x):
    return 1 if x > 0 else 0

def softmax(xs):
    maxX = max(xs)
    exp_values = np.exp(xs - maxX)
```

```python

    sum_exp_values = np.sum(exp_values)
    return [ex / sum_exp_values for ex in exp_values]

#                                    .
def mse(y0, y):
    return 1 / 2 * (y0 - y) ** 2
def dmse(y0, y):
    return y - y0
    res = []
    for i in range(len(y)):
        res.append(sum(y[i] - y0[i]) / len(y[i]))
    return res


class Layer:
    def __init__(self, n_neurons, n_input, activation, derivative, lr):
        self.n_neurons = n_neurons
        self.n_input = n_input
        # self.w = np.array([[1/n_input for _ in range(n_input)] for _ in range(n_neurons)])
        # self.w = np.array([[0.01 for _ in range(n_input)] for _ in range(n_neurons)])
        self.w = np.array([[random.uniform(1 / (2 * n_input), 2 / n_input)
                            for _ in range(n_input)] for _ in range(n_neurons)])
        self.activation = activation
        self.derivative = derivative
        self.XW = []
        self.out = []
        self.lr = lr
        self.prev_grad = []
        self.vt = np.zeros_like(self.w) # NAG
        self.LR = np.zeros_like(self.w) # Adagrad
        self.m = np.zeros_like(self.w) # Adam
        self.v = np.zeros_like(self.w) # Adam

    def forward(self, x):
        self.XW = np.dot(self.w, x)
        if self.activation == softmax:
            self.out = self.activation(self.XW)
        else:
            self.out = np.array([self.activation(xw) for xw in self.XW])
        return self.out

```

```
class Perceptron:
    def __init__(self, x_train, y_train, lr = 0.01):
        self.layers = []
        self.loss = mse
        self.dloss = dmse
        self.out = []
        self.x_train = x_train
        self.y_train = y_train
        self.n_train = len(x_train)
        self.lastDelta = []

    def add_layer(self, n_neurons, n_input=-1, func_act=relu, dfunc_act=drelu, lr=0.1):
        if len(self.layers) == 0:
            #                    0     -
            l0 = Layer(n_input, 0, func_act, dfunc_act, lr)
            self.layers.append(l0)
        n_input = (
            n_input if n_input > 0 else len(self.layers[-1].w)
        )  #       -

        self.layers.append(Layer(n_neurons, n_input, func_act, dfunc_act, lr))

    def forward(self, x):
        self.layers[0].out = x.copy()
        out = x.copy()
        i = 1
        for l in self.layers[1:]:
            out = l.forward(out)
            i += 1
        self.out = out
        return out

    def change_w_nag(self, k):
        for l in self.layers[1:]:
            l.w += k * l.vt * 0.9

    def gradient(self):
        gradient(self)
    def SGD(self):
        SGD(self)
    def NAG(self, log=True):
        NAG(self, log)
    def Adagrad(self):
```

```
        Adagrad(self)
    def Adam(self):
        Adam(self)

    def countErr(self):
        all = 0
        for i in range(self.n_train):  #                          n
            e = 0
            res = self.forward(self.x_train[i])
            for j in range(10):  # 10
                e += self.loss(self.y_train[i][j], res[j])
            e /= 10  #
            all += e
        all /= self.n_train
        return all

    '''def countGradient(self, y):
        gradient = []
        for l_i in range(len(self.layers) - 1, 0, -1):
            l = self.layers[l_i]

            if (l_i == len(self.layers) - 1):
                lastDelta = l.out - y
            else:
                l = self.layers[l_i]
                sum = np.dot(self.layers[l_i + 1].w.T, lastDelta)
                lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for j in range(l.n_neurons)])

            gradient.insert(0, np.outer(lastDelta, self.layers[l_i - 1].out))
        return gradient'''

    def find_answ(res):
        num = 0
        min = 1
        for i in range(len(res)):
            if abs(1 - res[i]) < min:
                min = abs(1 - res[i])
                num = i
        return num

    #
    def checkCorrectness(self, x = [], y = [], log = True):
        if len(x) == 0:
```

```
            x = self.x_train
            y = self.y_train

        num = len(x)
        correct_num = 0
        for i in range(num):
            res = [0] * 10
            mas = [0] * 10

            res = self.forward(x[i])

            for j in range(10):
                mas[j] = round(res[j], 2)

            predicted = np.argmax(res)
            expected = np.argmax(y[i])
            if expected == predicted:
                correct_num += 1

            if i > num - 5 and log:
                print(mas)
                print(y[i])
                print(expected, "--->", predicted, "\n")

        res = correct_num / num * 100
        if log:
            print(res, "% correctness")
        return res


def create_Y_ans(Y_first):
    Y_res = []
    for y in Y_first:
        mas = np.zeros(10)
        mas[y] = 1
        Y_res.append(mas)
    return np.array(Y_res)


data_folder = os.path.join(os.getcwd(), "data")

# load compressed MNIST gz files and return numpy arrays
def load_data(filename, label=False):
    with gzip.open(filename) as gz:
        struct.unpack("I", gz.read(4))
        n_items = struct.unpack(">I", gz.read(4))
```

```
        if not label:
            n_rows = struct.unpack(">I", gz.read(4))[0]
            n_cols = struct.unpack(">I", gz.read(4))[0]
            res = np.frombuffer(
                gz.read(n_items[0] * n_rows * n_cols), dtype=np.uint8)
            res = res.reshape(n_items[0], n_rows * n_cols)
        else:
            res = np.frombuffer(gz.read(n_items[0]), dtype=np.uint8)
            res = res.reshape(n_items[0], 1)
    return res

# note we also shrink the intensity values (X) from 0-255 to 0-1. This helps the model converge faster.
X_train = load_data(os.path.join(
    data_folder, "train-images.gz"), False) / 255.0
Y_train = load_data(os.path.join(
    data_folder, "train-labels.gz"), True).reshape(-1)
X_test = load_data(os.path.join(data_folder, "test-images.gz"), False) / 255.0
Y_test = load_data(os.path.join(
    data_folder, "test-labels.gz"), True).reshape(-1)

train_len = len(X_train)
n_tests = 500
X_first = X_train[:n_tests]
Y_first = Y_train[:n_tests]

Y_res = create_Y_ans(Y_first)

def createPerc(size=1, neuron_num=[10]):
    perc = Perceptron(X_first, Y_res)
    perc.neuron_num = neuron_num
    prev = n_pixels
    for i in range(size):
        perc.add_layer(n_input=prev, n_neurons=neuron_num[i], lr=0.01)
        prev = neuron_num[i]
    perc.add_layer(n_input=prev, n_neurons=10, func_act=softmax, lr=0.01)
    return perc

if __name__ == "__main__":
    perc1 = Perceptron(X_first, Y_res)
    perc1.add_layer(n_input=n_pixels, n_neurons=10, lr=0.01)
    perc1.add_layer(n_neurons=10, func_act=softmax, lr=0.01)


```

```
    W1 = [row.copy() for row in perc1.layers[1].w]
    W2 = [row.copy() for row in perc1.layers[2].w]
    perc1.gradient()
    perc1.checkCorrectness(X_first, Y_res)
    perc1.layers[1].w = [row[:] for row in W1]
    perc1.layers[2].w = [row[:] for row in W2]
    perc1.SGD()
    perc1.checkCorrectness(X_first, Y_res)
    perc1.layers[1].w = [row[:] for row in W1]
    perc1.layers[2].w = [row[:] for row in W2]
    perc1.NAG()
    perc1.checkCorrectness(X_first, Y_res)
    perc1.layers[1].w = [row[:] for row in W1]
    perc1.layers[2].w = [row[:] for row in W2]
    perc1.Adagrad()
    perc1.checkCorrectness(X_first, Y_res)
    perc1.layers[1].w = [row[:] for row in W1]
    perc1.layers[2].w = [row[:] for row in W2]
    perc1.Adam()
    perc1.checkCorrectness(X_first, Y_res)
    plt.legend()
    plt.show()

    '''print(perc1.layers[-2].w)
    print(perc1.layers[-1].w)'''

    #Y_res2 = create_Y_ans(Y_test)
    #perc1.checkCorrectness(X_test[:500], Y_res2[:500])
```

Листинг 2: methods.py

```
import numpy as np
import matplotlib.pyplot as plt
import random


def gradient(perc):
    x = perc.x_train
    y = perc.y_train

    epohs = []
    errors = []

    for step in range(15):
        if step % 1 == 0:
            print(step)
```

```
            epohs.append(step)
            errors.append(perc.countErr())

        for i in range(perc.n_train):  #                    n
            out = perc.forward(x[i])

            for l_i in range(len(perc.layers) - 1, 0, -1):
                l = perc.layers[l_i]

                if l_i == len(perc.layers) - 1:
                    #
                    lastDelta = l.out - y[i]
                    # lastDelta = np.array([perc.dloss(y[i][j], l.out[j]) for j in range(l.n_neurons)])
                else:
                    #
                    l = perc.layers[l_i]
                    #                        delta
                    sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
                    lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for j in range(l.n_neurons)])

                #gradient = np.dot(np.transpose([lastDelta]), [perc.layers[l_i - 1].out])
                gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
                perc.layers[l_i].w -= l.lr * gradient

    plt.plot(epohs, errors, label="gradient")
    # plt.show()


"""
def gradient(perc):
    x = perc.x_train
    y = perc.y_train
    num = perc.n_train

    epohs = []
    errors = []

    for step in range(10):
        if step % 1 == 0:
            print(step)
            epohs.append(step)
```

```
            errors.append(perc.countErr())

        out = []
        for i in range(num):
            perc.layers[0].out = x[i]
            out.append(perc.forward(x[i]))
        out = np.array(out)

        for l_i in range(len(perc.layers) - 1, 0, -1):
            l = perc.layers[l_i]

            if (l_i == len(perc.layers) - 1):
            #
                lastDelta = perc.dloss(y.T, out.T)
                #print(lastDelta)
                #lastDelta = np.array([perc.dloss(y[i][j], l.out[j]) for j in range(l.n_neurons)])
            else:
            #
                l = perc.layers[l_i]
                #                                    delta
                sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
                lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for j in range(l.n_neurons)])

            #gradient = np.dot(np.transpose([lastDelta]), [perc.layers[l_i - 1].out])
            gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
            perc.layers[l_i].w -= l.lr * gradient


    plt.plot(epohs, errors, label=f"{perc.loss.__name__}")
    #plt.show()
"""


def SGD(perc):
    x1 = perc.x_train
    y1 = perc.y_train

    epohs = []
    errors = []

    batch_size = 200

    for step in range(15):
```

```python
        if step % 1 == 0:
            print(step)
            epohs.append(step)
            errors.append(perc.countErr())

        # rand_batch = np.random.randint(0, num - batch_size)
        batch = random.sample(list(zip(x1, y1)), batch_size)
        for x, y in batch:
            out = perc.forward(x)

            for l_i in range(len(perc.layers) - 1, 0, -1):
                l = perc.layers[l_i]

                if l_i == len(perc.layers) - 1:
                    lastDelta = l.out - y
                else:
                    l = perc.layers[l_i]
                    sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
                    lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for j in range(l.n_neurons)])

                gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
                perc.layers[l_i].w -= l.lr * gradient

    plt.plot(epohs, errors, label="SGD")


def NAG(perc, log):
    x = perc.x_train
    y = perc.y_train

    epohs = []
    errors = []

    for step in range(15):
        if step % 1 == 0 and log:
            print(step)
            epohs.append(step)
            errors.append(perc.countErr())

        for i in range(perc.n_train):
            perc.change_w_nag(-1)
            out = perc.forward(x[i])
            perc.change_w_nag(1)

            for l_i in range(len(perc.layers) - 1, 0, -1):
```

```
                l = perc.layers[l_i]

                if l_i == len(perc.layers) - 1:
                    lastDelta = l.out - y[i]
                else:
                    l = perc.layers[l_i]
                    sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
                    lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for j in range(l.n_neurons)])

                gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)

                l.vt += 0.00001 * gradient
                l.w -= l.vt

    plt.plot(epohs, errors, label="NAG")


def Adagrad(perc):
    x = perc.x_train
    y = perc.y_train

    epohs = []
    errors = []

    for step in range(15):
        if step % 1 == 0:
            print(step)
            epohs.append(step)
            errors.append(perc.countErr())

        for i in range(perc.n_train):
            out = perc.forward(x[i])

            for l_i in range(len(perc.layers) - 1, 0, -1):
                l = perc.layers[l_i]

                if l_i == len(perc.layers) - 1:
                    lastDelta = l.out - y[i]
                else:
                    l = perc.layers[l_i]
                    sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
                    lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for j in range(l.n_neurons)])

                gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
```

```
                l.LR += gradient**2
                perc.layers[l_i].w -= l.lr * gradient / (np.sqrt(l.LR) + 1e-8)

    plt.plot(epohs, errors, label="Adagrad")


beta1 = 0.99
beta2 = 0.9
def Adam(perc):
    x = perc.x_train
    y = perc.y_train

    epohs = []
    errors = []

    for step in range(15):
        if step % 1 == 0:
            print(step)
            epohs.append(step)
            errors.append(perc.countErr())

        for i in range(perc.n_train):
            out = perc.forward(x[i])

            for l_i in range(len(perc.layers) - 1, 0, -1):
                l = perc.layers[l_i]

                if l_i == len(perc.layers) - 1:
                    lastDelta = l.out - y[i]
                else:
                    l = perc.layers[l_i]
                    sum = np.dot(perc.layers[l_i + 1].w.T, lastDelta)
                    lastDelta = np.array([sum[j] * l.derivative(l.XW[j]) for j in range(l.n_neurons)])

                gradient = np.outer(lastDelta, perc.layers[l_i - 1].out)
                l.m = beta1 * l.m + (1 - beta1) * gradient
                l.v = beta2 * l.v + (1 - beta2) * (gradient**2)
                m_ = l.m / (1 - beta1**(step + 1))
                v_ = l.v / (1 - beta2**(step + 1))

                perc.layers[l_i].w -= 0.00001 * m_ / (np.sqrt(v_) + 1e-8)

    plt.plot(epohs, errors, label="Adam")
```

Генетический алгоритм применялся к персептрону 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

```
from main import createPerc, plt

perc1 = createPerc()

import numpy as np
import time
import random

class Individ:
    def __init__(self, layers_num, neuron_num):
        self.layers_num = layers_num
        self.neuron_num = neuron_num
        self.params = [layers_num] + neuron_num
        self.p = createPerc(layers_num, neuron_num)
        self.p.NAG(log=False)

def createIndivid(params):
    layers_num = int(params[0])
    neuron_num = params[1:]
    neuron_num = neuron_num[:layers_num] + [10] * (layers_num - len(neuron_num))
    return Individ(layers_num, neuron_num)

def generate_perc_population(size):
    population = []
    for _ in range(size):
        layers_num = random.randint(1, 10)
        neuron_num = []
        for _ in range(layers_num):
            neuron_num.append(random.randint(1, 30))
        p = Individ(layers_num, neuron_num)
        population.append(p)
    return population


def F(x):
```

```
    return x.p.checkCorrectness(log = False)

#
def fitness_function(x):
    return 1 / F(x)

#
def selection(population, fitness_func, retain_ratio=0.5):
    fitness_scores = [(fitness_func(ind), ind) for ind in population]
    sorted_population = [ind for _, ind in sorted(fitness_scores, key=lambda x: x[0])]
    retain_length = int(len(sorted_population) * retain_ratio)
    retain_length = 2 if retain_length < 2 else retain_length
    parents = sorted_population[:retain_length]
    return parents

#
def crossover(parents):
    parents = [p.params for p in parents]
    children = []
    while len(children) < len(parents):
        id1 = random.randint(0, len(parents)-1)
        id2 = random.randint(0, len(parents)-1)
        if id1 != id2:
            parent1 = parents[id1]
            parent2 = parents[id2]
            maxlen = min(len(parent1), len(parent2))
            crossover_point = random.randint(0, maxlen)
            child1 = parent1[:crossover_point] + parent2[crossover_point:]
            child2 = parent2[:crossover_point] + parent1[crossover_point:]
            children.extend([createIndivid(child1), createIndivid(child2)])
    return children

#
def mutate(params):
    rand_index = random.randint(0, len(params)-1)
    params[rand_index] += random.randint(-2, 2)
    params[rand_index] = max(1, params[rand_index])
    return createIndivid(params)

def mutation(children, mutation_chance=0.2):
    for child in children:
        if random.random() < mutation_chance:
```

```
                child = mutate(child.params)

    return children

xs = []
ys = []

#
def genetic_algorithm(population_size, generations, mutation_rate, crossover_rate):
    population = generate_perc_population(size=population_size)
    for i in range(generations):
        print(i)
        parents = selection(population, fitness_function, crossover_rate)
        offspring = crossover(parents)
        offspring = mutation(offspring, mutation_rate)
        population = parents + offspring
        result = population[np.argmin([fitness_function(x) for x in population])]
        xs.append(i)
        ys.append(F(result))

    for p in population:
        print("          :", p.layers_num, "      -                    :", p.neuron_num)
        print("                     :", F(p))
    best_solution = population[np.argmin([fitness_function(x) for x in population])]
    return best_solution

start_time = time.time()
print("                                :")
result = genetic_algorithm(population_size=20, generations=10, mutation_rate=0.2, crossover_rate=0.5)
print("                          :", time.time() - start_time, "c")
plt.plot(xs, ys)
plt.show()
print("      -                           :", result.layers_num, "      -                                   :", result.neuron_num)
print("                           :", F(result))
print()
```

4 Результат работы

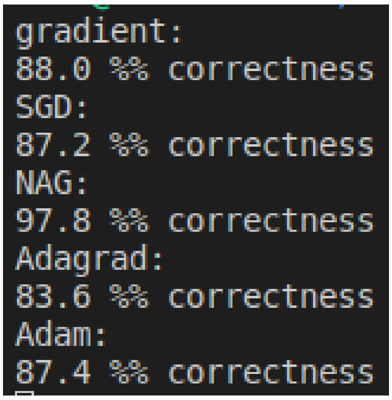

Рис. 1 — Результат вычислений: точность каждого метода

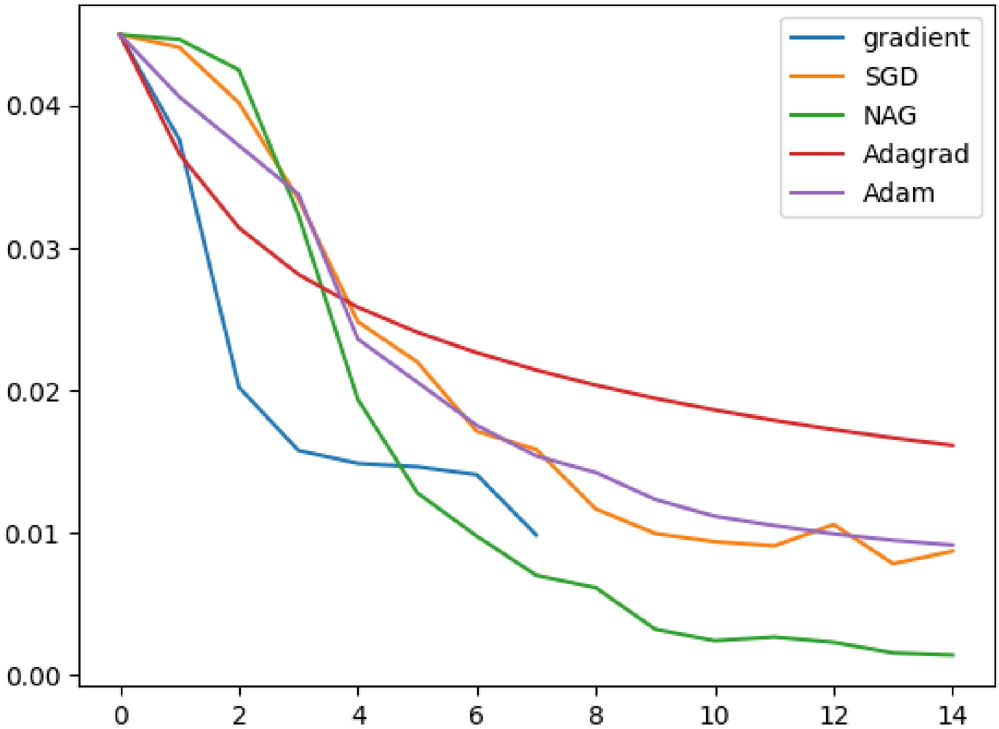

Рис. 2 — Сравнительный график сходимости методов оптимизации: зависимость значения ошибки от количества эпох

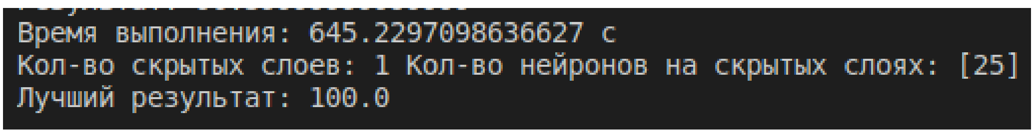

Рис. 3 — Результат работы генетического алгоритма

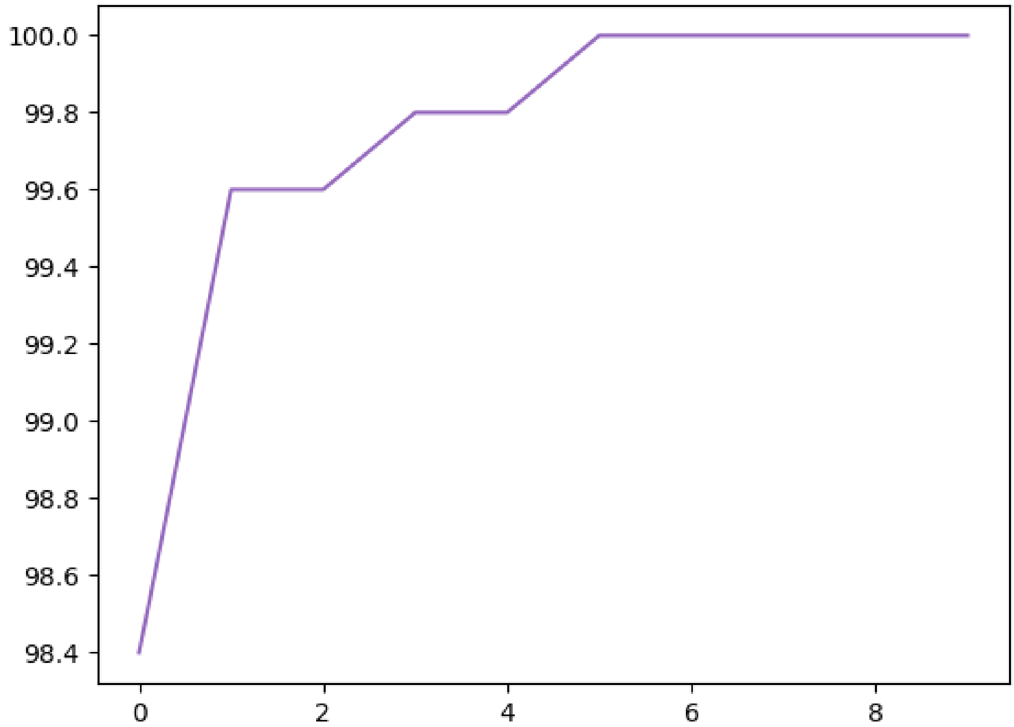

Рис. 4 — График генетического алгоритма:
зависимость точности от кол-ва итераций (поколений)

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.