|  | **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 Информатики и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

**Домашнее задание № 4**

**Сравнительный анализ современных методов оптимизации. Использование генетического алгоритма для оптимизации гиперпараметров**

**ПО КУРСУ:**

***«Теория искусственных нейронных сетей»***

Студент *Караник А. А.*

Преподаватель *Каганов Ю. Т.*

*Москва, 2024 г.*

**ОГЛАВЛЕНИЕ**

[Цель работы 3](#_Toc183393278)

[Постановка задачи 3](#_Toc183393279)

[Практическая реализация 3](#_Toc183393280)

[Результаты 11](#_Toc183393281)

[Вывод 13](#_Toc183393282)

# Цель работы

1. Изучение основных методов оптимизации.
2. Изучение генетического алгоритма для оптимизации

гиперпараметров.

# Постановка задачи

1. Реализовать современные методы оптимизации: SGD, NAG, Adagrad, ADAM.
2. Реализовать генетический алгоритм для оптимизации гиперпараметров (число слоев и число нейронов) многослойного персептрона.
3. Провести сравнительный анализ работы современных методов оптимизации на примере многослойного персептрона.

# Практическая реализация

Исходный текст программы на языке программирования Python:

import sys

from matplotlib import pyplot as plt

import numpy as np

from dataclasses import dataclass

import pandas as pd

from itertools import product, chain

from IPython.display import clear\_output

from torchvision.datasets import MNIST

from torch.utils.data import DataLoader

from torch.utils.data import Subset

from tqdm import tqdm

transform = lambda img: np.array(np.asarray(img).flatten())/256

train\_dataset = MNIST('.', train=True, download=True, transform=transform)

test\_dataset = MNIST('.', train=False, transform=transform)

train\_loader = DataLoader(train\_dataset, batch\_size=32, shuffle=True)

test\_loader = DataLoader(test\_dataset, batch\_size=32, shuffle=False)

X, y = next(iter(train\_loader))

X = X.numpy()

y = y.numpy()

def train(network,train\_loader, test\_loader, epochs, loss):

    train\_loss\_epochs = []

    test\_loss\_epochs = []

    train\_accuracy\_epochs = []

    test\_accuracy\_epochs = []

    try:

        for epoch in tqdm(range(epochs)):

            losses = []

            accuracies = []

            for X, y in train\_loader:

                X = X.view(X.shape[0], -1).numpy()

                y = y.numpy()

                prediction = network.forward(X)

                loss\_batch = loss.forward(prediction, y)

                losses.append(loss\_batch)

                dLdx = loss.backward()

                network.backward(dLdx)

                network.step()

                accuracies.append((np.argmax(prediction, 1)==y).mean())

            train\_loss\_epochs.append(np.mean(losses))

            train\_accuracy\_epochs.append(np.mean(accuracies))

            losses = []

            accuracies = []

            for X, y in test\_loader:

                X = X.view(X.shape[0], -1).numpy()

                y = y.numpy()

                prediction = network.forward(X)

                loss\_batch = loss.forward(prediction, y)

                losses.append(loss\_batch)

                accuracies.append((np.argmax(prediction, 1)==y).mean())

            test\_loss\_epochs.append(np.mean(losses))

            test\_accuracy\_epochs.append(np.mean(accuracies))

            clear\_output(True)

            sys.stdout.write('\rEpoch {0}... (Train/Test) Loss: {1:.3f}/{2:.3f}\tAccuracy: {3:.3f}/{4:.3f}'.format(

                            epoch, train\_loss\_epochs[-1], test\_loss\_epochs[-1],

                            train\_accuracy\_epochs[-1], test\_accuracy\_epochs[-1]))

            plt.figure(figsize=(12, 5))

            plt.subplot(1, 2, 1)

            plt.plot(train\_loss\_epochs, label='Train')

            plt.plot(test\_loss\_epochs, label='Test')

            plt.xlabel('Epochs', fontsize=16)

            plt.ylabel('Loss', fontsize=16)

            plt.legend(loc=0, fontsize=16)

            plt.grid('on')

            plt.subplot(1, 2, 2)

            plt.plot(train\_accuracy\_epochs, label='Train accuracy')

            plt.plot(test\_accuracy\_epochs, label='Test accuracy')

            plt.xlabel('Epochs', fontsize=16)

            plt.ylabel('Accuracy', fontsize=16)

            plt.legend(loc=0, fontsize=16)

            plt.grid('on')

            plt.show()

    except KeyboardInterrupt:

        pass

    return train\_loss\_epochs, \

           test\_loss\_epochs, \

           train\_accuracy\_epochs, \

           test\_accuracy\_epochs

class MSELoss:

    def \_\_init\_\_(self):

        pass

    def forward(self, X, y):

        self.X = X

        self.y = np.zeros((X.shape[0], X.shape[1]))

        self.y[np.arange(X.shape[0]), y] = 1

        return np.mean(np.square(self.X - self.y))

    def backward(self):

        return 2 \* (self.X - self.y) / self.y.shape[0]

@dataclass

class HyperParams:

    lr: float

    epochs: int

    def \_\_init\_\_(self, lr, epoch):

        self.lr = lr

        self.epochs = int(epoch)

    def as\_vec(self):

        return np.array([

            self.lr, self.epochs

        ])

class ReLU:

    def \_\_init\_\_(self):

        pass

    def step(self):

        pass

    def forward(self, X):

        self.X = X

        return np.maximum(X, 0)

    def backward(self, dLdy):

        return (self.X > 0) \* dLdy

class Adam:

    def \_\_init\_\_(self, params, beta1 = 0.9, beta2 = 0.99, nu = 1., eta = 1e-8, lr=0.001):

        self.params = params

        self.beta1 = beta1

        self.beta2 = beta2

        self.nu = nu

        self.eta = eta

        self.lr = lr

        self.m = [np.zeros(p.shape) for p in self.params]

        self.v = [np.zeros(p.shape) for p in self.params]

    def step(self, gradW, gradb):

          grads = [gradW, gradb]

          for i, p in enumerate(self.params):

              self.m[i]=self.beta1\*self.m[i]+(1-self.beta1)\*grads[i]

              self.v[i]=self.beta2\*self.v[i]+(1-self.beta2)\*grads[i]\*\*2

              m\_ = self.m[i]/(1-self.beta1\*\*(i+1))

              v\_ = self.v[i]/(1-self.beta2\*\*(i+1))

              p-=self.lr\*self.nu/(np.sqrt(v\_)+self.eta)\*m\_

class SGD:

    def \_\_init\_\_(self, params, lr=1e-2):

        self.params = params

        self.lr = lr

    def step(self,gradW, gradb):

        grads = [gradW, gradb]

        for i, p in enumerate(self.params):

            p -= self.lr \* grads[i]

class NAG:

    def \_\_init\_\_(self, params, lr=1e-2, gamma=0.9):

        self.params = params

        self.lr=lr

        self.gamma=gamma

        self.momentum = [np.zeros(p.shape) for p in self.params]

    def step(self,gradW, gradb):

        grads = [gradW, gradb]

        for i, p in enumerate(self.params):

            self.momentum[i] = self.gamma \* self.momentum[i] + self.lr \* grads[i]

            p-=self.momentum[i]

class AdaGrad:

    def \_\_init\_\_(self, params, eta=1e-8, lr=1e-2):

        self.params=params

        self.eta = eta

        self.lr = lr

        self.G = [0] \* len(self.params)

    def step(self,gradW, gradb):

        grads = [gradW, gradb]

        for i, p in enumerate(self.params):

            self.G[i] += grads[i] \*\* 2

            p -= self.lr / np.sqrt(self.G[i] + self.eta) \* grads[i]

class Linear:

    def \_\_init\_\_(self, input\_size, output\_size, optimizer):

        self.W = np.random.randn(input\_size, output\_size)\*0.01

        self.b = np.zeros(output\_size)

        optimizer\_class = optimizer[0]

        optimizer\_options = optimizer[1] if len(optimizer) > 2 else {}

        optimizer = optimizer\_class([self.W,self.b], \*\*optimizer\_options)

        self.optimizer=optimizer

    def step(self):

        self.optimizer.step(self.dLdW,self.dLdb)

    def forward(self, X):

        self.X = X

        return X.dot(self.W)+self.b

    def backward(self, dLdy):

        self.dLdW = self.X.T.dot(dLdy)

        self.dLdb = dLdy.sum(0)

        self.dLdx = dLdy.dot(self.W.T)

        return self.dLdx

class NeuralNet:

    def \_\_init\_\_(self, modules):

        self.modules = modules

    def step(self):

        for i in range(len(self.modules)):

            self.modules[i].step()

    def forward(self, X):

        y = X

        for i in range(len(self.modules)):

            y = self.modules[i].forward(y)

        return y

    def backward(self, dLdy):

        for i in range(len(self.modules))[::-1]:

            dLdy = self.modules[i].backward(dLdy)

class Creature:

    def \_\_init\_\_(self, hp: HyperParams):

        self.hp = hp

        adam=[Adam,{'lr': hp.lr}]

        self.network = NeuralNet([

            Linear(784, 100,adam), ReLU(),

            Linear(100, 100,adam), ReLU(),

            Linear(100, 10,adam)

        ])

        self.loss = MSELoss()

        self.optimizer = 'Adam'

    def \_\_repr\_\_(self):

        return str(self.hp)

    def test(self, test\_loader):

        accuracies=[]

        for X, y in test\_loader:

            X = X.view(X.shape[0], -1).numpy()

            y = y.numpy()

            prediction = self.network.forward(X)

            loss\_batch = self.loss.forward(prediction, y)

            accuracies.append((np.argmax(prediction, 1)==y).mean())

        return np.mean(accuracies)

    def train(self, train\_loader):

        for epoch in range(self.hp.epochs):

            for X, y in train\_loader:

                X = X.view(X.shape[0], -1).numpy()

                y = y.numpy()

                prediction = self.network.forward(X)

                loss\_batch = self.loss.forward(prediction, y)

                dLdx = self.loss.backward()

                self.network.backward(dLdx)

                self.network.step()

    def fitnes(self, dl):

        return self.test(dl)

class GenAlgorithm:

    def \_\_init\_\_(self, dl\_len=1000) -> None:

        self.dl = {

            'test': DataLoader(Subset(train\_dataset, range(0, dl\_len)), shuffle=True, batch\_size=16),

            'train': DataLoader(Subset(train\_dataset, range(dl\_len, int(dl\_len\*1.2))), shuffle=True, batch\_size=16),

        }

        lrs = [0.001,0.01, 0.1]

        epochs = [10, 30]

        self.creatures = []

        self.pop\_size = 0

        for lr, ep\_num in product(lrs, epochs):

            self.creatures.append(Creature(HyperParams(lr=lr, epoch=ep\_num)))

            self.pop\_size += 1

    def train(self):

        for c in tqdm(self.creatures):

            c.train(self.dl['train'])

    def build\_df(self, creatures: list[Creature]):

        df = pd.DataFrame({'creature': creatures})

        df['fitnes'] = df.creature.map(lambda x: x.fitnes(self.dl['test']))

        df.fitnes = df.fitnes

        df['cs'] = df.fitnes / df.fitnes.sum()

        df['cs'] = df['cs'] / sum(df.cs)

        df = df.sort\_values(by=['fitnes'], axis=0, ascending=True)

        return df

    def selection(self):

        self.creatures = list(np.random.choice(

            self.df.creature,

            size=self.pop\_size,

            p=self.df.cs

        ))

    def crossing\_over(self):

        def cross(p1, p2):

            pc = 0.6

            genes1, genes2 = p1.hp.as\_vec(), p2.hp.as\_vec()

            while True:

                try:

                    ngenes1, ngenes2 = [], []

                    for g1, g2 in zip(genes1, genes2):

                        r = np.random.random()

                        if r < pc:

                            ngenes1.append(g1)

                            ngenes2.append(g2)

                        else:

                            c = np.random.random()

                            ngenes1.append(g1\*c + (1-c)\*g2)

                            ngenes2.append(g2\*c + (1-c)\*g1)

                except AssertionError:

                    continue

                else:

                    return [Creature(HyperParams(\*ngenes1)), Creature(HyperParams(\*ngenes2))]

        np.random.shuffle(self.creatures)

        pairs = [tuple(self.creatures[i:i+2]) for i in range(0, 2\*len(self.creatures)//2-1, 2)] + \

            [tuple(self.creatures[-2:])]

        offsprings = list(map(lambda x: cross(\*x), pairs))

        self.creatures = list(chain(\*offsprings))[:self.pop\_size]

    def mutation(self):

        pm = 0.4

        def mutate(c):

            if np.random.random() > pm:

                return c

            gens = c.hp.as\_vec()

            i = np.random.randint(0, len(gens))

            gens[i] = np.random.uniform(\*[(0.001, 0.01),(10, 30),][i])

            try:

                return Creature(HyperParams(\*gens))

            except:

                return c

        self.creatures = list(map(mutate, self.creatures))

    def replace\_with\_new\_gen(self):

        new\_df = self.build\_df(self.creatures)

        all\_df = pd.concat([self.df, new\_df], axis=0)

        all\_df.fitnes

        all\_df.sort\_values(by='fitnes', ascending=True, inplace=True)

        self.df = all\_df.tail(self.pop\_size)

        self.df.cs = self.df.fitnes / self.df.fitnes.sum()

        print(self.df)

    def evolve(self, N):

        best = []

        self.train()

        self.df = self.build\_df(self.creatures)

        for i in range(N):

            print(f'-> generation {i+1} of {N}')

            self.selection()

            self.crossing\_over()

            self.mutation()

            self.train()

            self.replace\_with\_new\_gen()

            best.append(self.df.iloc[-1].fitnes)

        print(self.df)

        row = self.df['fitnes'].idxmax()

        plt.plot(range(len(best)), best)

        plt.xlabel("Популяции")

        plt.ylabel("Значения функции фитнеса")

        return self.df.iloc[row].creature, self.df.iloc[row].fitnes

algorithm = GenAlgorithm()

result = algorithm.evolve(10)

print("Наилучшие значения гиперпараметров: скорость обучения - {0}, число эпох - {1}".format(result[0].hp.lr, result[0].hp.epochs))

sgd=[SGD, {'lr': 0.005}]

network = NeuralNet([

    Linear(784, 100, sgd), ReLU(),

    Linear(100, 100, sgd), ReLU(),

    Linear(100, 10, sgd)

])

loss = MSELoss()

train\_mse\_sgd, test\_mse\_sgd, train\_ac\_mse\_sgd, test\_ac\_mse\_sgd = train(network,train\_loader,test\_loader, 10, loss=loss)

adam=[Adam,{'lr': 0.005}]

network = NeuralNet([

    Linear(784, 100, adam), ReLU(),

    Linear(100, 100, adam), ReLU(),

    Linear(100, 10, adam)

])

loss = MSELoss()

train\_mse\_adam, test\_mse\_adam, train\_ac\_mse\_adam, test\_ac\_mse\_adam = train(network, train\_loader, test\_loader, 10, loss=loss)

adagrad = [AdaGrad,{'lr': 0.005}]

network = NeuralNet([

    Linear(784, 100, adagrad), ReLU(),

    Linear(100, 100, adagrad), ReLU(),

    Linear(100, 10, adagrad)

])

loss = MSELoss()

train\_mse\_adg, test\_mse\_adg, train\_ac\_mse\_adg, test\_ac\_mse\_adg = train(network, train\_loader, test\_loader, 10, loss=loss)

nag = [NAG,{'lr': 0.005}]

network = NeuralNet([

    Linear(784, 100, nag), ReLU(),

    Linear(100, 100, nag), ReLU(),

    Linear(100, 10, nag)

])

loss = MSELoss()

train\_mse\_nag, test\_mse\_nag, train\_ac\_mse\_nag, test\_ac\_mse\_nag = train(network, train\_loader, test\_loader, 10, loss=loss)

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.title('Loss')

plt.plot(test\_mse\_sgd, label='SGD')

plt.plot(test\_mse\_nag, label='NAG')

plt.plot(test\_mse\_adg, label='Adagrad')

plt.plot(test\_mse\_adam, label='Adam')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend(loc=0)

plt.grid()

plt.subplot(1, 2, 2)

plt.title('Accuracy')

plt.plot(test\_ac\_mse\_sgd, label='SGD')

plt.plot(test\_ac\_mse\_nag, label='NAG')

plt.plot(test\_ac\_mse\_adg, label='Adagrad')

plt.plot(test\_ac\_mse\_adam, label='Adam')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend(loc=0)

plt.grid()

plt.show()

# Результаты

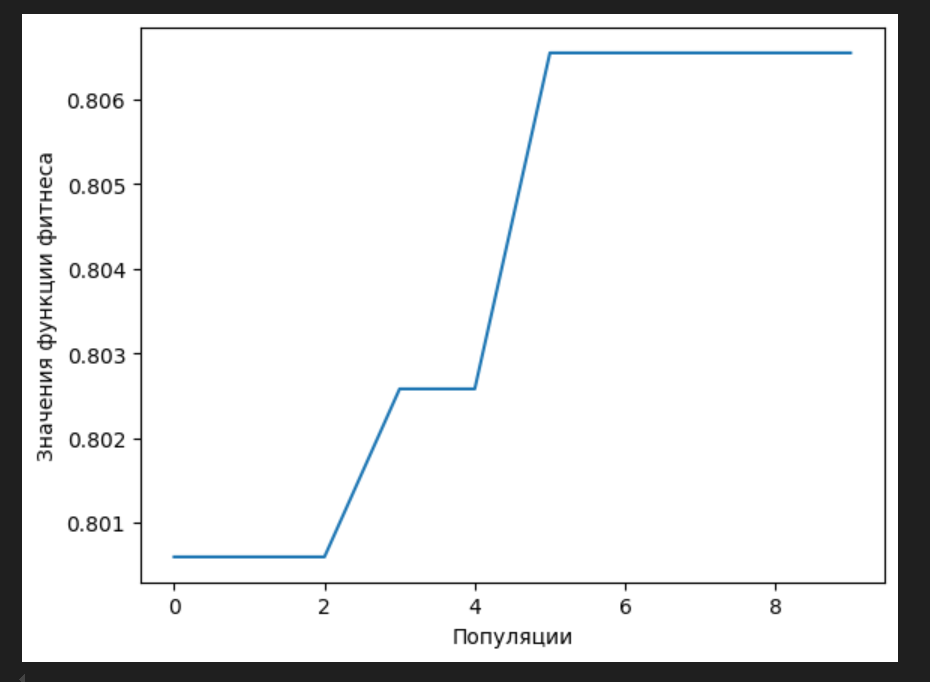


Рисунок - Результат работы генетического алгоритма



Рисунок - Наилучшие значения гиперпараметров

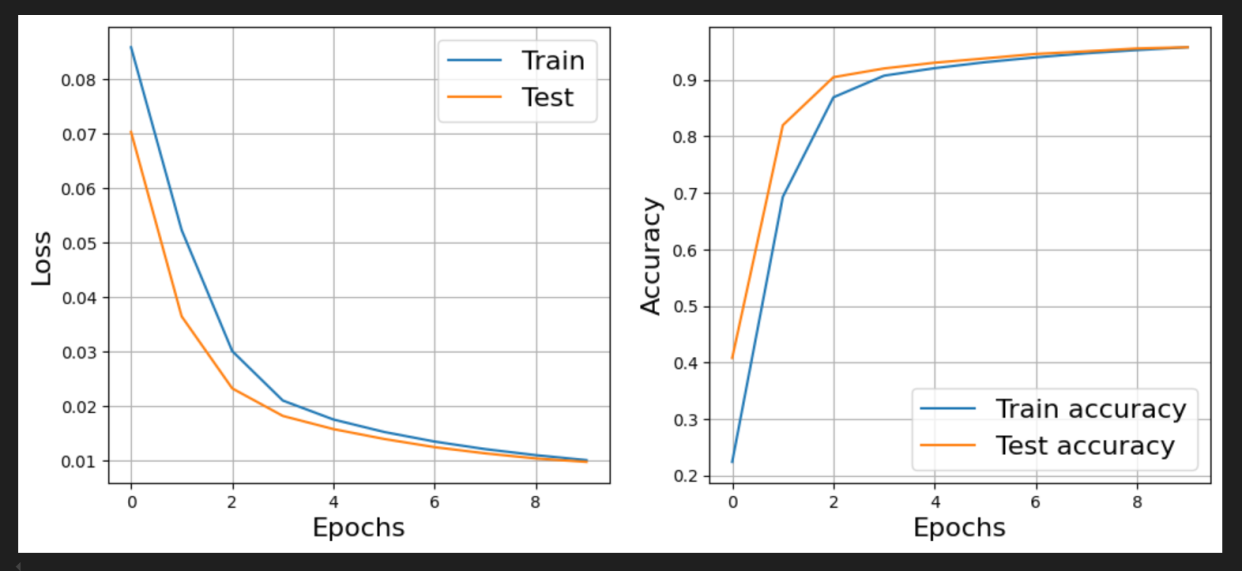


Рисунок - SGD

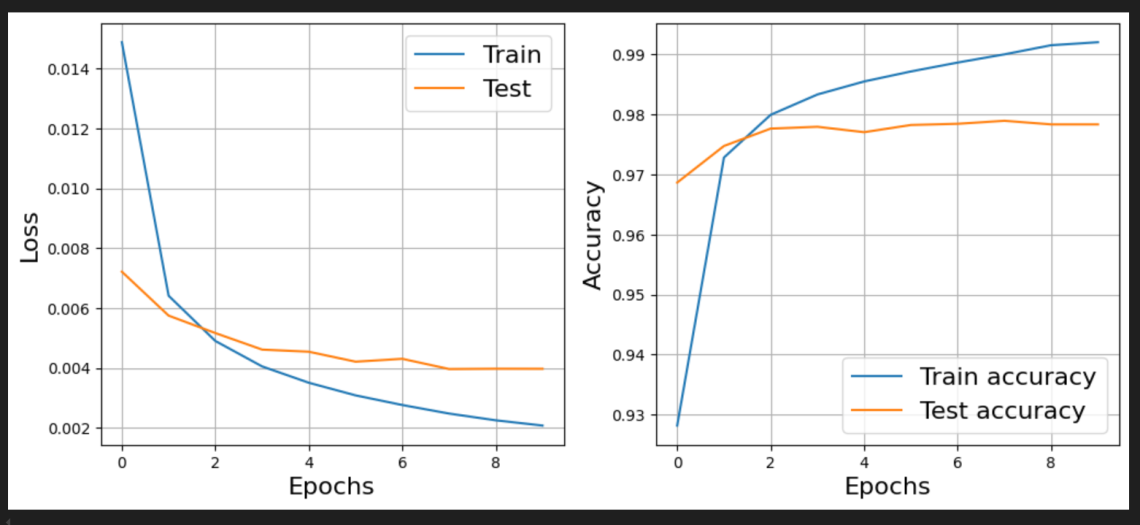


Рисунок - Adam

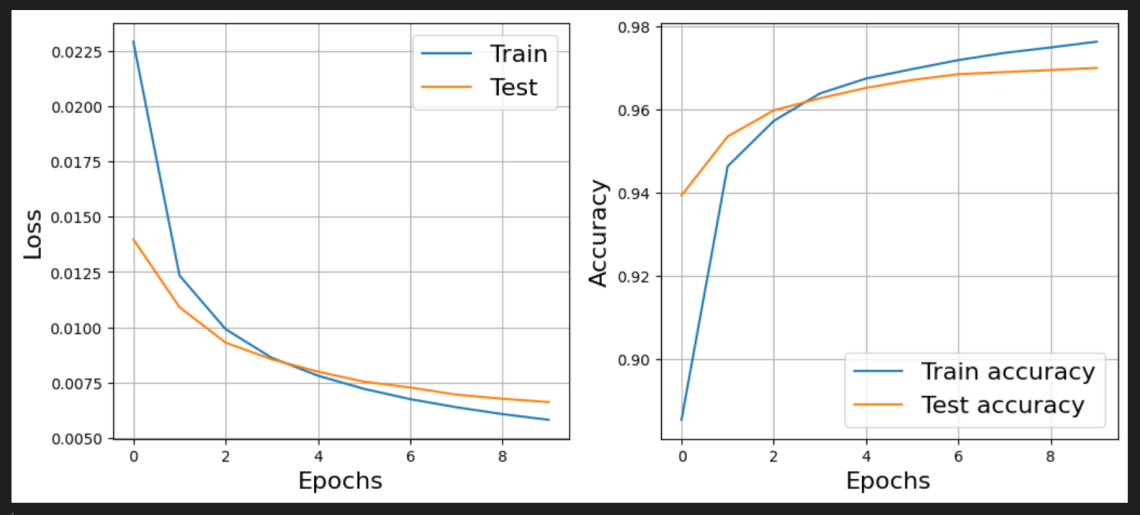


Рисунок - Adagrad

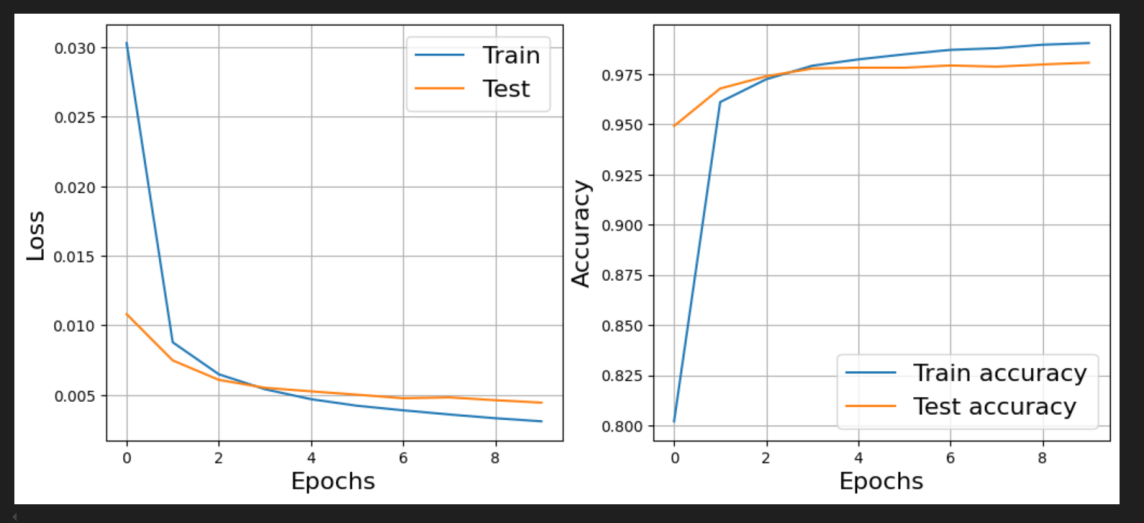


Рисунок – NAG

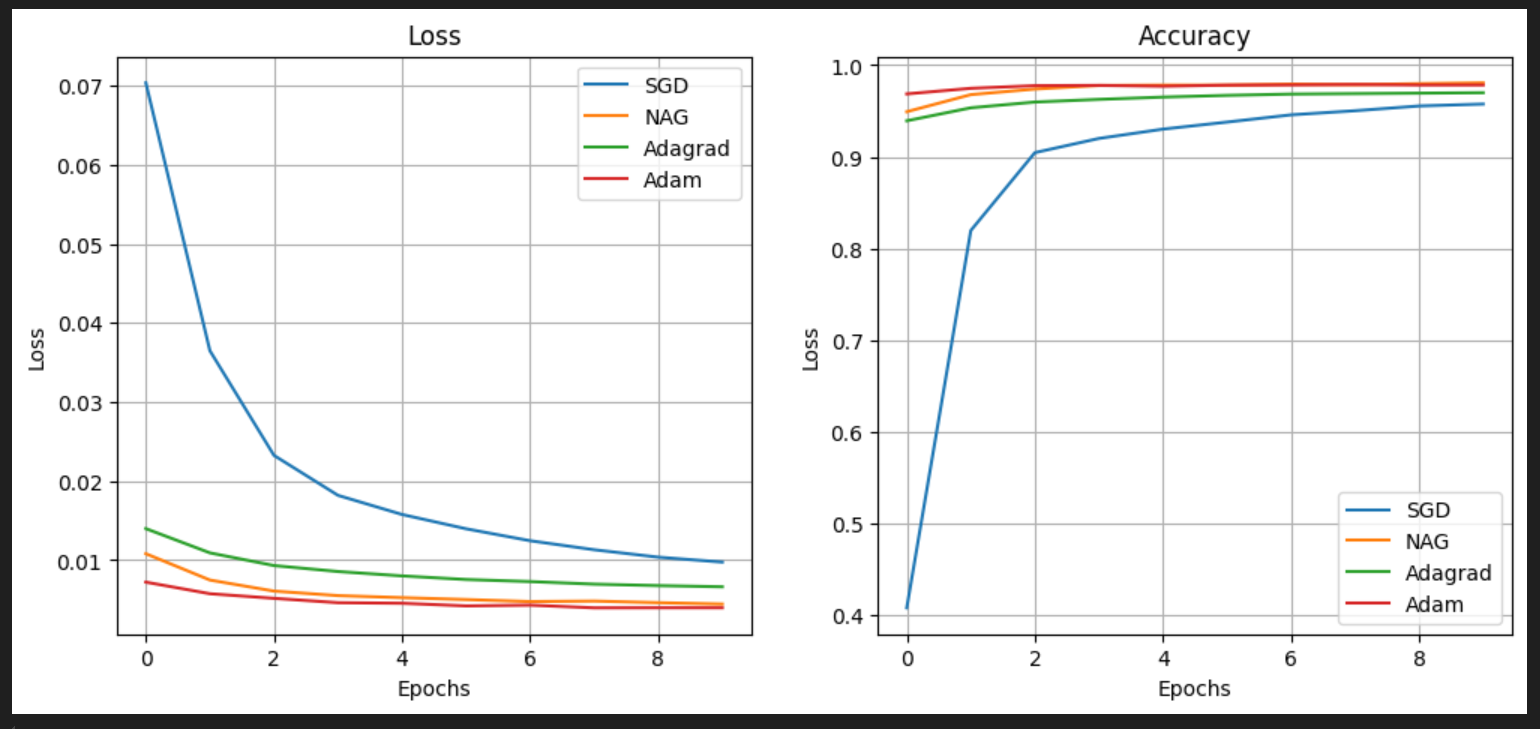


Рисунок - Общий график

# Вывод

В ходе выполнения данной работы был разработан генетический алгоритм для оптимизации гиперпараметров нейронной сети, таких как количество скрытых слоев и число нейронов в них. Кроме того, были реализованы различные алгоритмы оптимизации. Тестирование показало, что методы Adam и NAG демонстрируют лучшую скорость сходимости по сравнению с другими рассмотренными алгоритмами.