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

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:

```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
```

```python
    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
```

```python
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

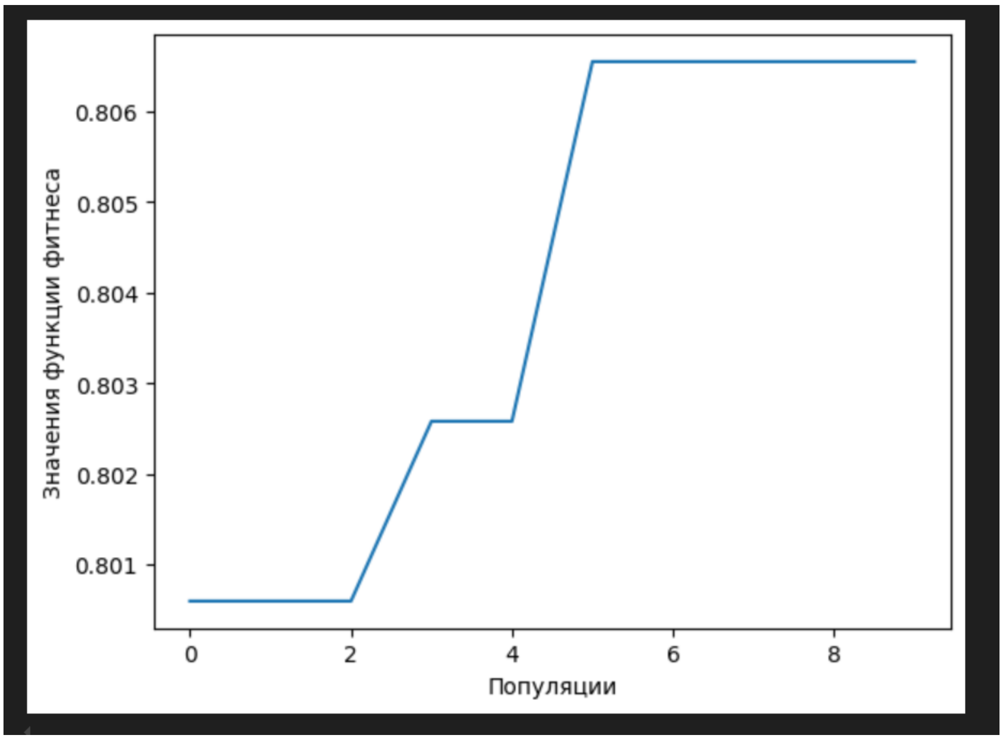
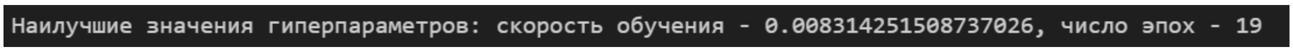


Рисунок 1 - Результат работы генетического алгоритма



```
Наилучшие значения гиперпараметров: скорость обучения - 0.008314251508737026, число эпох - 19
```

Рисунок 2- Наилучшие значения гиперпараметров

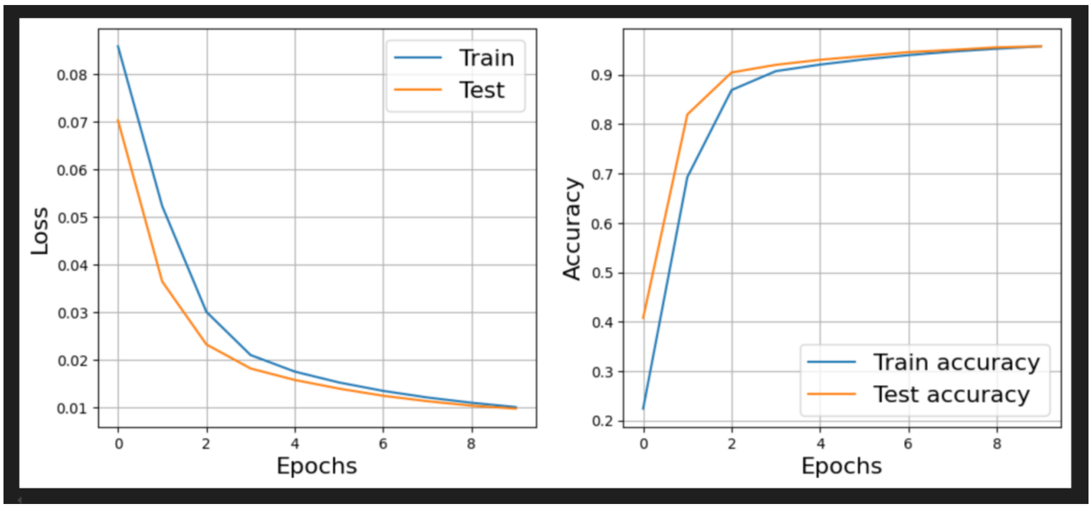


Рисунок 3 - SGD

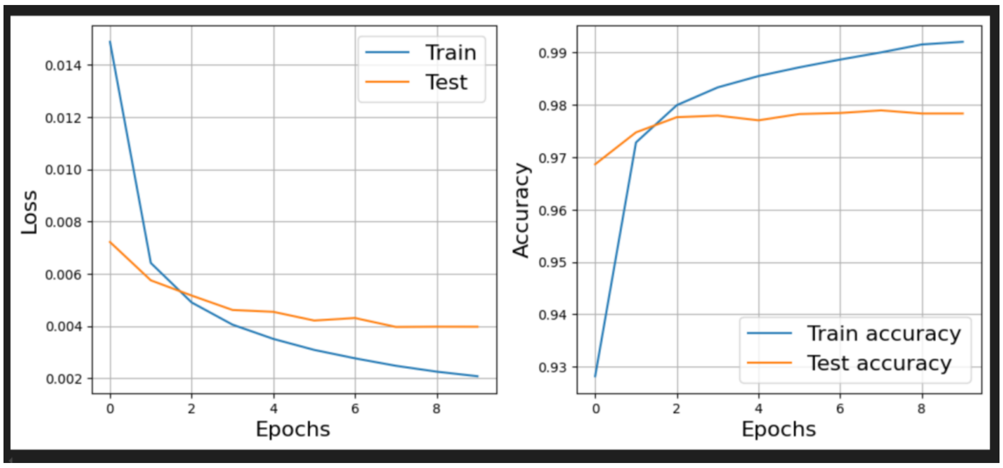


Рисунок 4 - Adam

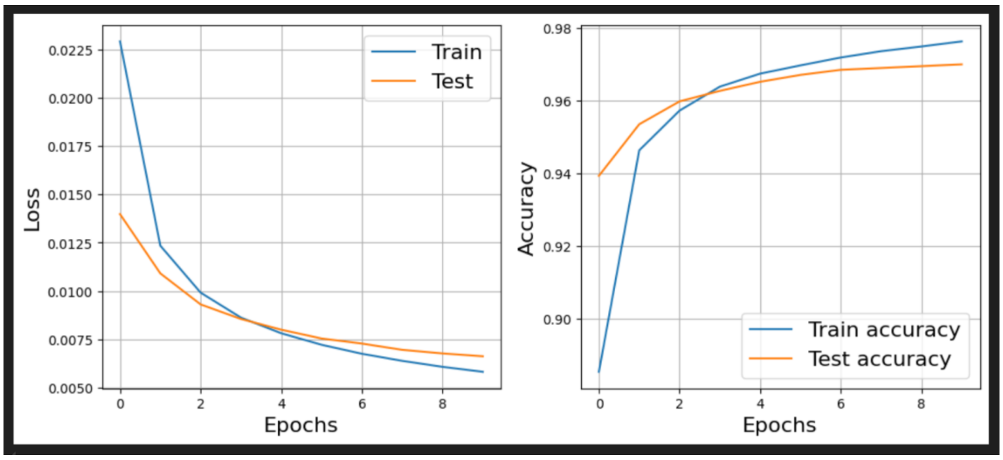


Рисунок 5 - Adagrad

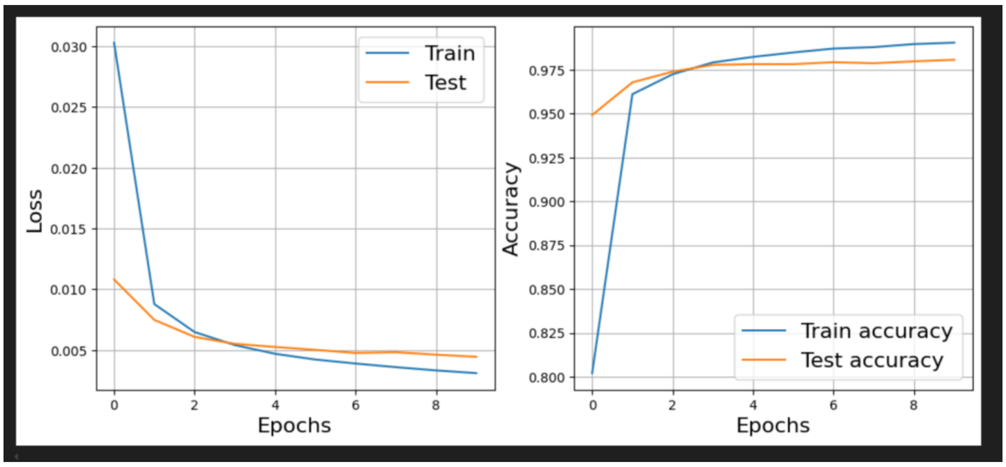


Рисунок 6 – NAG

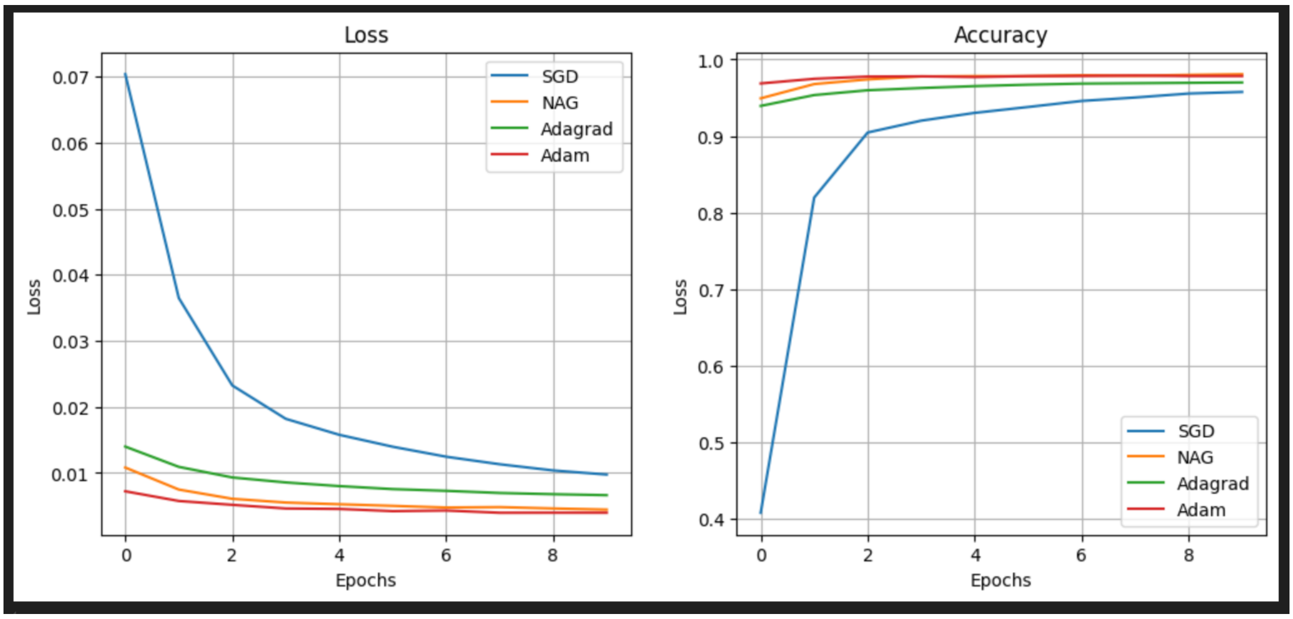


Рисунок 7 - Общий график

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.