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 № 3

по курсу «Теория искусственных нейронных сетей»

«Сравнительный анализ современных методов оптимизации. Использование генетического алгоритма для оптимизации гиперпараметров многослойного персептрона.»

Студент группы ИУ9-71Б Баев Д.А

Преподаватель Каганов Ю. Т.

Москва 2023

1 Задание

1. Провести сравнительный анализ современных методов оптимизации (SGD, NAG, Adagrad, ADAM) на примере многослойного персептрона.

2. Использовать генетический алгоритм для оптимизации гиперпараметров (число слоев и число нейронов) многослойного персептрона.

3. Подготовить отчет с распечаткой текста программы, графиками результатов исследования и анализом результатов.

2 Исходный код

Исходный код программы представлен в листингах 1- 7

Листинг 1: Подготовка датасета

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torchvision
from tqdm import tqdm

MNIST_train = torchvision.datasets.MNIST('./', download=True, train=True)
MNIST_test = torchvision.datasets.MNIST('./', download=True, train=False)

count = 480
count_test = 80

train_X = MNIST_train.data.numpy()[:count]
train_Y = MNIST_train.targets.numpy()[:count]
test_X = MNIST_test.data.numpy()[:count_test]
test_Y = MNIST_test.targets.numpy()[:count_test]

train_X = np.array(list(map(lambda x: x.flatten() / 256, train_X)))
train_Y = np.array([np.array([int(i == x) for i in range(10)]) for x in train_Y])
test_X = np.array(list(map(lambda x: x.flatten() / 256, test_X)))
test_Y = np.array([np.array([int(i == x) for i in range(10)]) for x in test_Y])
```

Листинг 2: Определение функций активации и функций ошибки

```
def softmax(x):
    if np.linalg.norm(x) < 0.001:
        return np.zeros(len(x))
    x = x / np.linalg.norm(x)
    return np.exp(x)/(np.exp(x)).sum() if (np.exp(x)).sum() > 0.01 else np.zeros(len(x))

def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return np.where(x > 0, 1, 0)

```

```
13 def mse(y_true, y_pred):
14     return np.sum((y_true - y_pred) ** 2) / len(y_true)
15
16 def mse_derivative(y_true, y_pred):
17     return 2 * (y_pred - y_true) / len(y_true)
18
19 def sigmoid(x): return 1/(1+np.exp(-x))
20 def sigmoid_derivative(x): return sigmoid(x)*(1-sigmoid(x))
21
22 def cross_entropy(y_true, y_pred):
23     y_pred = np.clip(y_pred, 1e-8, 1 - 1e-8)
24     return -np.mean(y_true * np.log(y_pred))
25
26 def cross_entropy_derivative(y_true, y_pred):
27     y_pred = np.clip(y_pred, 1e-8, 1 - 1e-8)
28     res = y_pred - y_true
29     return res / np.linalg.norm(res)
30
31 def kl_divergence(y_true, y_pred):
32     y_true = np.clip(y_true, 1e-8, 1- 1e-8)
33     y_pred = np.clip(y_pred, 1e-8, 1-1e-8)
34     return np.mean(y_true * np.log(y_true / y_pred))
35
36 def kl_divergence_derivative(y_true, y_pred):
37     y_pred = np.clip(y_pred, 1e-8, 1 - 1e-8)
38     res = y_pred - y_true
39     return res / np.linalg.norm(res)
```

Листинг 3: Класс линейного слоя (с реализацией современных методов оптимизации)

```
1  class LinearLayer:
2      def __init__(self, input_size, output_size, optimizer, model):
3          self.hessian = None
4          self.previous_weights = None
5          self.previous_grad = None
6          self.inputs = None
7          self.weights = np.random.rand(input_size + 1, output_size) - 0.5
8          self.optimizer = optimizer
9          self.model = model
10
11     def forward(self, inputs, train=True):
12         inputs = np.append([1], inputs)
13
14         if train:
15             self.inputs = inputs
16         return inputs @ self.weights
```

```
17
18      def calculate_forward(self):
19          return self.model.calculate_forward(self, self.inputs[1:])
20
21      def backward(self, grad):
22          accum_grad = (grad @ self.weights.T)[1:]
23          step_grad = np.outer(self.inputs, grad)
24          if np.linalg.norm(step_grad) != 0:
25              step_grad /= np.linalg.norm(step_grad)
26          step = None
27          match self.optimizer:
28              case 0:
29                  # GD
30                  step = step_grad
31              case 1:
32                  # FR
33                  if self.previous_grad is None:
34                      self.previous_grad = step_grad
35                      w = 0
36                  else:
37                      grad = step_grad.flatten()
38                      previous_grad = self.previous_grad.flatten()
39                      w = max((np.linalg.norm(grad)/np.linalg.norm(
    previous_grad)) ** 2, 0)
40                      if np.isnan(w) or w == np.Inf:
41                          w = 0
42                  step = (step_grad + w * self.previous_grad)
43                  self.previous_grad = step_grad
44              case 2:
45                  # BFGS
46                  if self.hessian is None:
47                      self.hessian = np.identity(len(step_grad))
48                      self.previous_grad = np.zeros(step_grad.shape)
49                      self.previous_weights = np.zeros(self.weights.shape)
50
51                  step = self.hessian @ step_grad
52
53                  s = self.weights - self.previous_weights
54                  y = step_grad - self.previous_grad
55
56                  w = np.identity(len(step_grad)) - s @ y.T
57                  self.hessian = w @ self.hessian @ w.T
58
59                  self.previous_grad = step_grad
60                  self.previous_weights = self.weights
61
```

```
62
63          step *= self.model.lr
64
65          before = self.calculate_forward()
66          self.weights -= step
67          after = self.calculate_forward()
68          if before > after and abs(after - before) > 0.0001:
69              while before > after and abs(after - before) > 0.0001:
70                  self.weights -= step
71                  before = after
72                  after = self.calculate_forward()
73              self.weights += step
74          return accum_grad
```

Листинг 4: Класс слоя активации

```
1  class ActivationLayer:
2      def __init__(self, activation, activation_derivative):
3          self.inputs = None
4          self.activation = activation
5          self.activation_derivative = activation_derivative
6      def forward(self, inputs, train=True):
7          self.inputs = inputs
8          return self.activation(inputs)
9      def backward(self, grad):
10         return grad * self.activation_derivative(self.inputs)
```

Листинг 5: Класс перцептрона

```
1  class Perceptron:
2      def __init__(self, input_size, sizes, loss, loss_derivative,
       optimizer, lr):
3          self.last_true = None
4          self.layers = []
5          prev_size = input_size
6          for size in sizes:
7              self.layers.append(LinearLayer(prev_size, size, optimizer,
       self))
8              self.layers.append(ActivationLayer(sigmoid,
       sigmoid_derivative))
9              prev_size = size
10         self.layers.append(LinearLayer(prev_size, 10, optimizer, self))
11         self.layers.append(ActivationLayer(softmax, lambda x: softmax(x)
        * (1 - softmax(x))))
12         self.loss = loss
13         self.loss_derivative = loss_derivative
14         self.lr = lr
```

```
15
16      def forward(self, inputs, train=True):
17          result = inputs
18          for layer in self.layers:
19              result = layer.forward(result, train)
20          return result
21
22      def backward(self, y_true, y_pred):
23          grad = self.loss_derivative(y_true, y_pred)
24          for layer in self.layers[::-1]:
25              grad = layer.backward(grad)
26
27      def fit(self, inputs, y_true):
28          self.last_true = y_true
29          y_pred = self.forward(inputs, True)
30          loss = self.loss(y_true, y_pred)
31          self.backward(y_true, y_pred)
32          return y_pred, loss
33
34      def calculate(self, layer, inputs):
35          place = self.layers.index(layer)
36          result = inputs
37          for layer in self.layers[place:]:
38              result = layer.forward(result, False)
39          return self.loss(self.last_true, result)
40
41
42      def train(self, epochs):
43          accuracy = []
44          loss_arr = []
45          for _ in tqdm(range(epochs)):
46              running_accuracy = 0
47              running_loss = 0
48              for inputs, y_true in zip(train_X, train_Y):
49                  y_pred, loss = self.fit(inputs, y_true)
50                  pred = np.argmax(y_pred)
51                  running_loss += loss
52                  running_accuracy += (np.argmax(y_true) == pred)
53              accuracy.append(running_accuracy / len(train_X))
54              loss_arr.append(running_loss / len(train_X))
55          return accuracy, loss_arr
56
57      def validate(self):
58          running_accuracy = 0
59          running_loss = 0
60          for inputs, y_true in zip(test_X, test_Y):
```

```
            y_pred = self.forward(inputs, False)
            loss = self.loss(y_true, y_pred)
            pred = np.argmax(y_pred)
            running_loss += loss
            running_accuracy += (np.argmax(y_true) == pred)
        return running_loss / len(test_X), running_accuracy / len(test_X
    )
```

Листинг 6 — Функция эксперимента

```
def experiment(learning_rate, layer_count, layer_neurons, epochs,
    optimizer, loss, loss_der):
    perceptron = Perceptron(28 * 28, [layer_neurons for _ in range(
    layer_count)], loss, loss_der, optimizer, learning_rate)

    accuracy, loss = perceptron.train(epochs)


    plt.plot(np.arange(len(accuracy)), accuracy)
    plt.title("Accuracy")
    plt.show()
    plt.plot(np.arange(len(loss)), loss)
    plt.title("Loss")
    plt.show()

    print(perceptron.validate())
```

Листинг 7: Генетический алгоритм

```
class Individual:
    def __init__(self, num_hidden_layers, num_neurons):
        self.num_hidden_layers = num_hidden_layers
        self.num_neurons = num_neurons

def evaluate_individual(individual, loss, loss_der, optimizer,
    learning_rate, gamma, beta1, beta2):
    perceptron = Perceptron(28 * 28, [individual.num_neurons for _ in
    range(individual.num_hidden_layers)], loss, loss_der, optimizer,
    learning_rate, gamma, beta1, beta2)

    accuracy, loss = perceptron.train(5)

    return accuracy[-1], loss[-1]


def genetic_algorithm(population_size, generations, loss, loss_der,
    optimizer, learning_rate, gamma, beta1, beta2):
    best_individual = None
```

```
16      population = [Individual(num_hidden_layers=np.random.randint(1, 4),
        num_neurons=np.random.randint(16, 256)) for _ in range(
        population_size)]
17      for generation in tqdm(range(generations)):
18          fitness = np.array([evaluate_individual(individual, loss,
        loss_der, optimizer, learning_rate, gamma, beta1, beta2) for
        individual in population], dtype=[("first", float), ("second", float
        )])
19
20          sorted_indices = np.argsort(fitness, order='second')
21          population = [population[i] for i in sorted_indices]
22          elite = population[:int(0.2 * population_size)]
23
24          offspring = []
25          for _ in range(int(0.8*population_size)):
26              parent1 = np.random.choice(elite)
27              parent2 = np.random.choice(elite)
28              child = Individual(
29                  num_hidden_layers=np.random.choice([parent1.
        num_hidden_layers, parent2.num_hidden_layers]),
30                  num_neurons=np.random.choice([parent1.num_neurons,
        parent2.num_neurons])
31              )
32              offspring.append(child)
33
34          for child in offspring:
35              if np.random.rand() < 0.15:
36                  child.num_hidden_layers = np.random.randint(1, 4)
37              if np.random.rand() < 0.15:
38                  child.num_neurons = np.random.randint(16, 256)
39
40          population = elite + offspring
41
42          best_individual = population[0]
43          print(f"Generation {generation + 1}, Best accuracy: {fitness
        [0][0]}, Best_loss: {fitness[0][1]}")
44
45      return best_individual
```

3 Результаты

В качестве начального эксперимента были выбраны следующие параметры: learning rate - 0.009, количество скрытых слоев (без учета входного и выходного

слоев) - 1, количество нейронов в скрытом слое - 64, количество эпох - 40, функция потерь - перекрестная энтропия, оптимизатор - SGD.

График точности этой модели от количества эпох приведен на рисунке 1

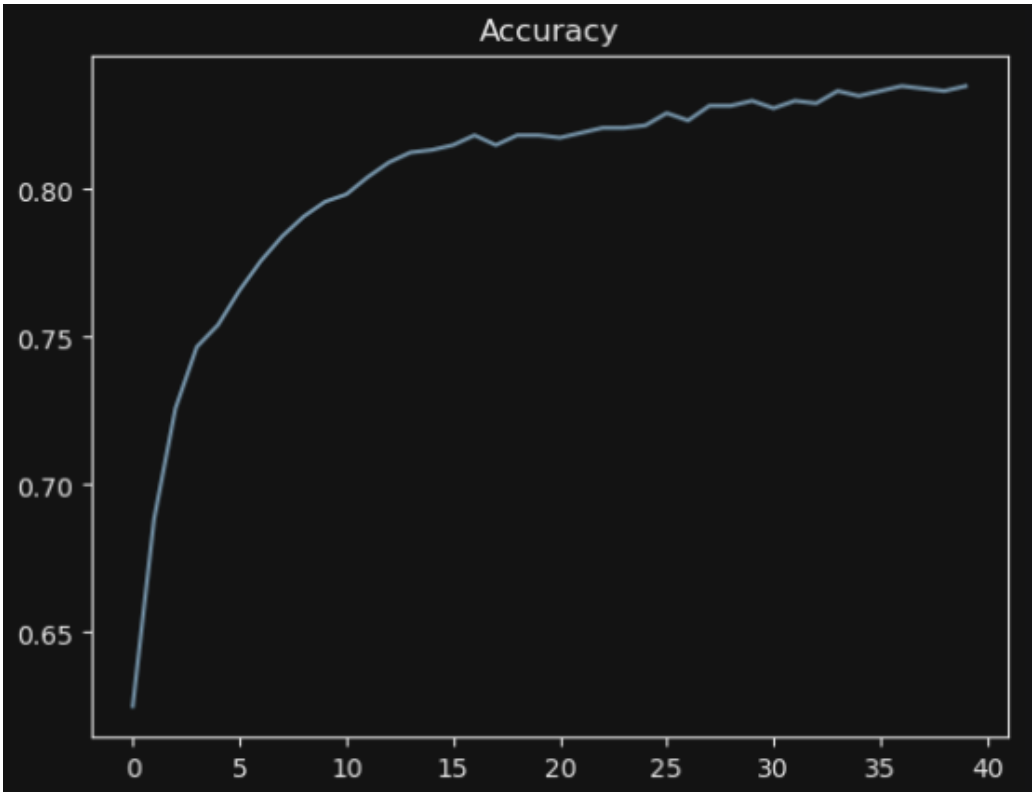

Рис. 1

Теперь были выбраны следующие параметры: learning rate - 0.0065, количество скрытых слоев (без учета входного и выходного слоев) - 1, количество нейронов в скрытом слое - 64, количество эпох - 40, функция потерь - перекрестная энтропия, оптимизатор - NAG, gamma - 0.95. Видно серьезное улучшение результата по сравнению с SGD.

График точности этой модели от количества эпох приведен на рисунке 2

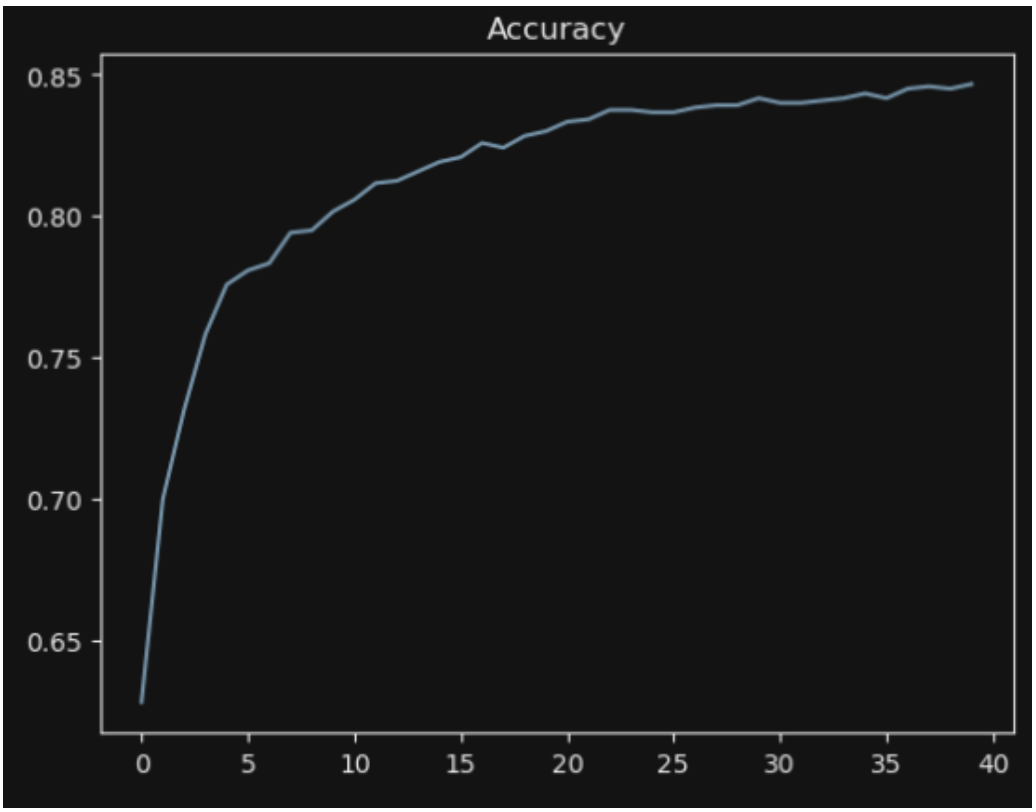

Рис. 2

Теперь были выбраны следующие параметры: learning rate - 0.15, количество скрытых слоев (без учета входного и выходного слоев) - 1, количество нейронов в скрытом слое - 64, количество эпох - 40, функция потерь - перекрестная энтропия, оптимизатор - Adadelta. Результат примерно такой же, как и с NAG.

График точности этой модели от количества эпох приведен на рисунке 3

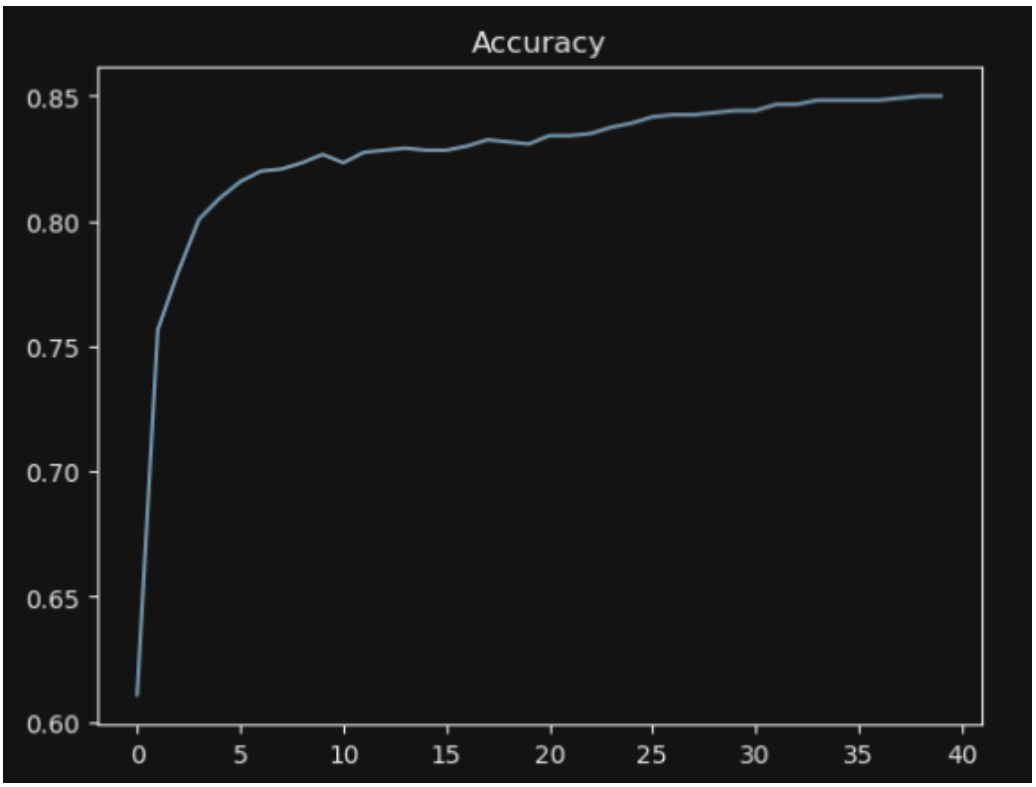

Рис. 3

Теперь были выбраны следующие параметры: learning rate - 0.0065, количество скрытых слоев (без учета входного и выходного слоев) - 1, количество нейронов в скрытом слое - 64, количество эпох - 40, функция потерь - перекрестная энтропия, оптимизатор - Adam. Получился результат, немного лучший, чем с предыдущими двумя оптимизаторами.

График точности этой модели от количества эпох приведен на рисунке 4

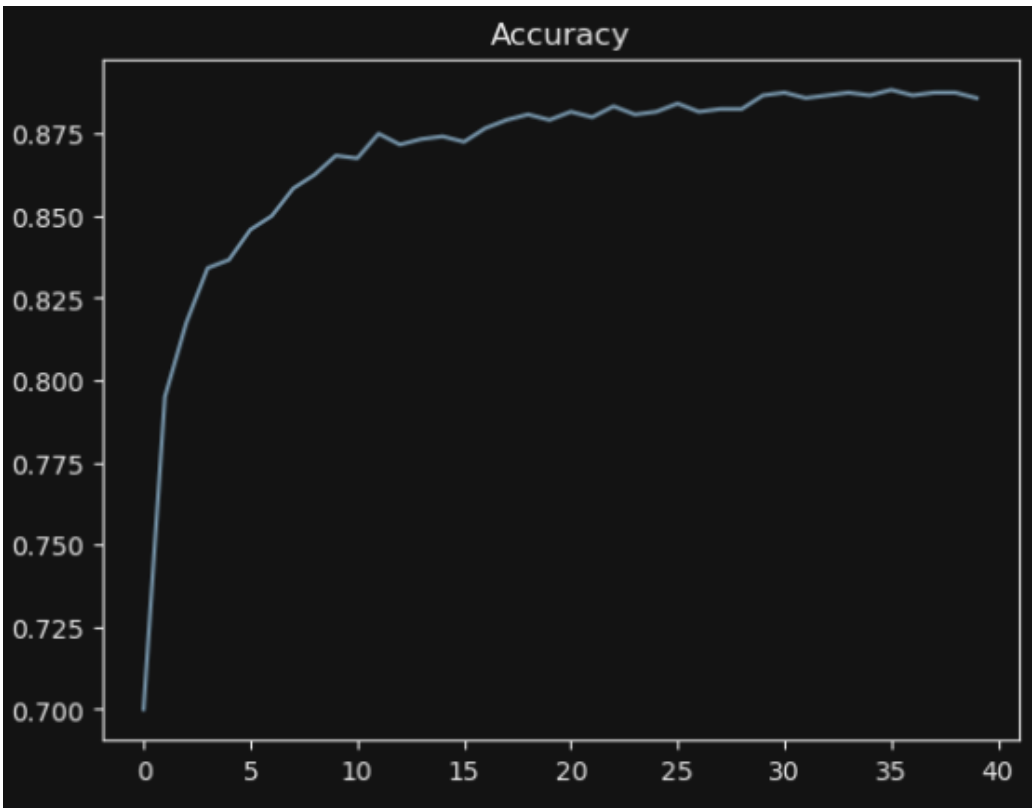

Рис. 4

При проведении генетического алгоритма были выбраны: размер популяции - 25 особей, количество поколений - 5, отбор лучших происходит по наименьшему значению loss функции (1/5 особей), шанс мутации - 0.15. В результате были получены следующие гиперпараметры: количество нейронов - 248, количество слоев - 2.

Результат работы генетического алгоритма приведен на рисунке 5

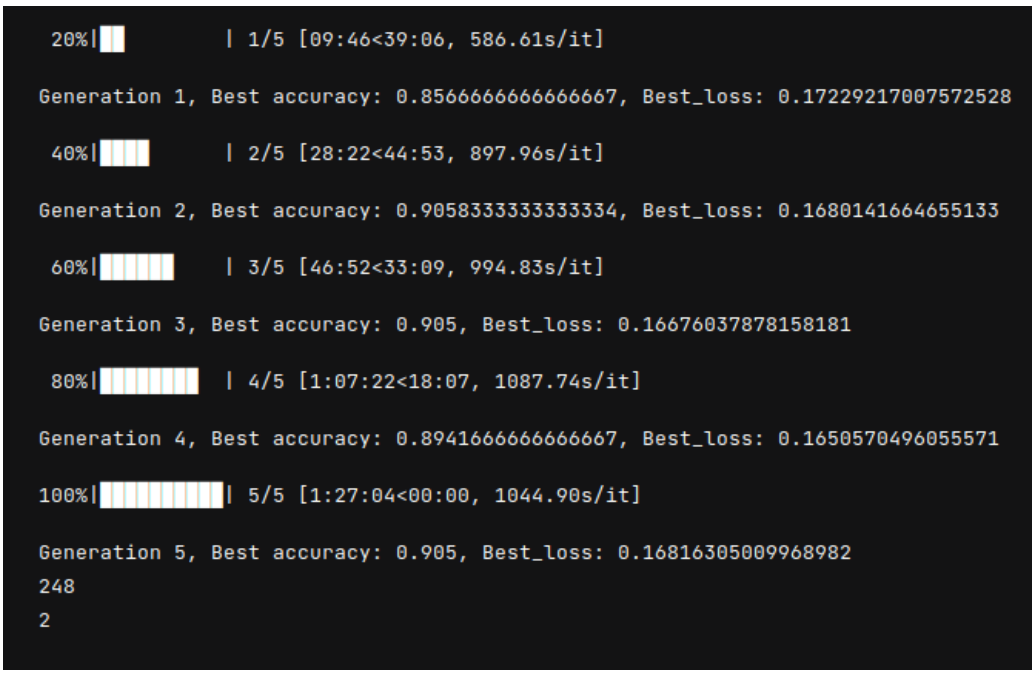

Рис. 5

Далее была обучена модель с подобранными гиперпараметрами. По графику видно, что удалось добиться значительного повышения точности, по сравнению с предыдущими результатами.

График точности этой модели от количества эпох приведен на рисункее 6

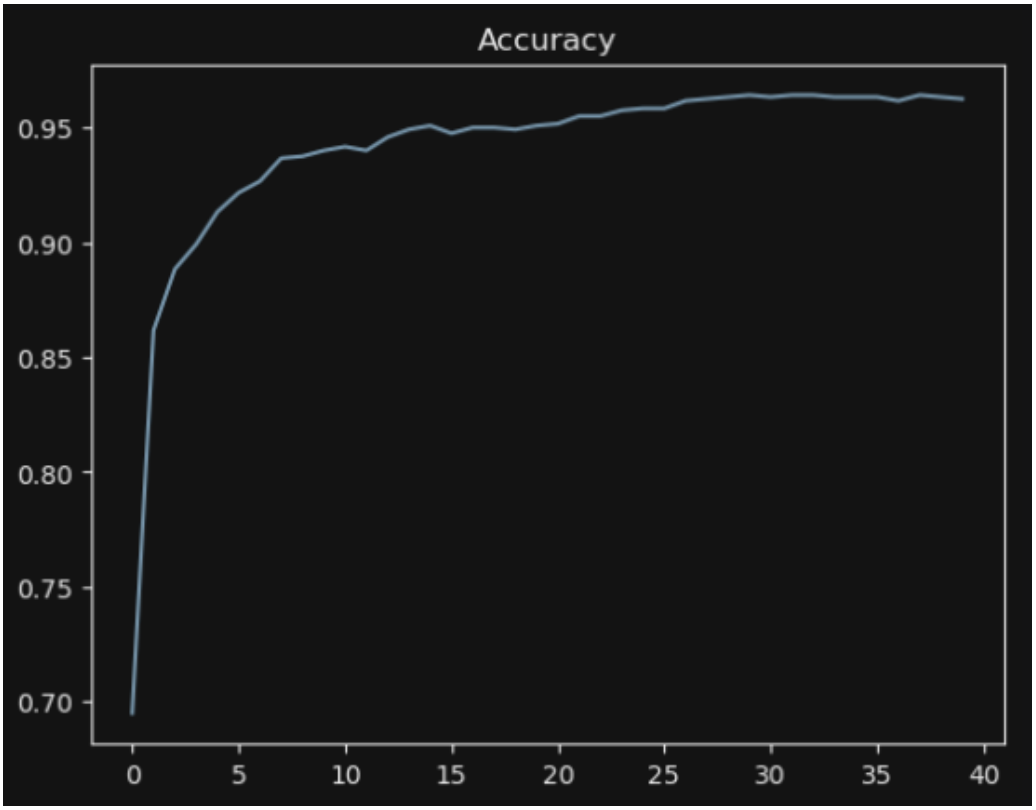

Рис. 6

4 Выводы

В рамках данной лабораторной работы были реализованы и сравнены современные методы оптимизации на примере многослойного перцептрона. В результате экспериментов лучший результат показал оптимизатор Adam. Также был реализован генетический алгоритм для подбора гиперпараметров в многослойном перцептроне. Благодаря этому алгоритму удалось серьезно улучшить результат.