



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»

Домашняя работа № 4

по курсу «Теория искусственных нейронных сетей»

Студент группы ИУ9-71Б Афанасьев И.

Преподаватель Каганов Ю.Т.

Москва 2024

1 Цель работы

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

2 Реализация

В реализации используется каркас многослойного персептрона, разработанный в домашнем задании №2. Программа написана на языке C++.

В листингах 1, 2 приводится исходный код функций активации и стоимости. В листингах 3, 4 приводится исходный код загрузки датасета MNIST. В листингах 5 и 6 приводится исходный код каркаса многослойного персептрона и методов оптимизации: SGD, NAG, Adagrad, Adam.

В листингах 7 и 8 приводятся классы хромосомы, используемые генетическим алгоритмом. В листингах 9 и 10 приводятся классы функции приспособленности. В качестве значения приспособленности используются обратные значения функций стоимости на тестовых данных. В листингах 11 и 12 приводится реализация генетического алгоритма. Отбор производится по методу рулетки. При скрещивании потомки в разных пропорциях получают родительские характеристики, а при мутации случайным образом изменяется некоторый ген хромосомы (в частности, алгоритм параметризуется долями хромосом, подвергающихся скрещиванию и мутации).

В листинге 13 приводится исходный код main-файла программы.

Листинг 1: Файл activation_function.h

```
1  #pragma once
2
3  #include <Eigen/Dense>
4  #include <iostream>
5
6  namespace nn {
7
```

```

8 class IActivationFunction {
9 public:
10     virtual ~IActivationFunction() = default;
11
12 public:
13     virtual Eigen::VectorXd Apply(const Eigen::VectorXd &z) = 0;
14     virtual Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) = 0;
15 };
16
17 class Linear final : public IActivationFunction {
18 public:
19     Eigen::VectorXd Apply(const Eigen::VectorXd &z) override { return z; }
20
21     Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
22         return Eigen::MatrixXd::Identity(z.rows(), z.cols());
23     }
24 };
25
26 class ReLU final : public IActivationFunction {
27 public:
28     Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
29         return z.array().max(0.0);
30     }
31
32     Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
33         return z.array().cwiseTypedGreaterOrEqual(0.0).matrix().asDiagonal();
34     }
35 };
36
37 class LeakyReLU final : public IActivationFunction {
38     std::function<double(double)> f_, f_prime_;
39
40 public:
41     LeakyReLU(const double alpha)
42         : f_([alpha](const double x) { return x >= 0 ? x : alpha * x; }),
43         f_prime_([alpha](const double x) { return x >= 0 ? 1 : alpha; }) {}
44
45     Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
46         return z.unaryExpr(f_);
47     }

```

```

48
49 Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
50     return z.unaryExpr(f_prime_).asDiagonal();
51 }
52 };
53
54 class Sigmoid final : public IActivationFunction {
55 public:
56     Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
57         return 1.0 / (1.0 + (-z).array().exp());
58     }
59
60     Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
61         const auto sigmoid = Apply(z);
62         return (sigmoid.array() * (1 - sigmoid.array())).matrix().asDiagonal();
63     }
64 };
65
66 class Tanh final : public IActivationFunction {
67 public:
68     Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
69         const auto e_z = z.array().exp();
70         const auto e_neg_z = (-z).array().exp();
71         return (e_z - e_neg_z) / (e_z + e_neg_z);
72     }
73
74     Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
75         const auto tanh = Apply(z);
76         return (1 - tanh.array().square()).matrix().asDiagonal();
77     }
78 };
79
80 class Softmax final : public IActivationFunction {
81 public:
82     Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
83         const auto e_z = z.array().exp();
84         return e_z / e_z.sum();
85     }
86
87     Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {

```

```

88     const auto softmax = Apply(z);
89     return softmax.asDiagonal().toDenseMatrix() - softmax * softmax.transpose();
90 }
91 };
92
93 } // namespace nn

```

Листинг 2: Файл cost_function.h

```

1  #pragma once
2
3  #include <Eigen/Dense>
4
5  namespace nn {
6
7  class ICostFunction {
8  public:
9      virtual ~ICostFunction() = default;
10
11  public:
12      virtual double Apply(const Eigen::VectorXd &y, const Eigen::VectorXd &a) = 0;
13      virtual Eigen::VectorXd GradientWrtActivations(const Eigen::VectorXd &y,
14                                                       const Eigen::VectorXd &a) = 0;
15  };
16
17  class MSE final : public ICostFunction {
18  public:
19      double Apply(const Eigen::VectorXd &y, const Eigen::VectorXd &a) override {
20          return 0.5 * (y - a).squaredNorm();
21      }
22
23      Eigen::VectorXd GradientWrtActivations(const Eigen::VectorXd &y,
24                                              const Eigen::VectorXd &a) override {
25          return a - y;
26      }
27  };
28
29  class CrossEntropy final : public ICostFunction {
30  public:
31      double Apply(const Eigen::VectorXd &y, const Eigen::VectorXd &a) override {

```

```

32     return -(y.array() * a.array().log()).sum();
33 }
34
35 Eigen::VectorXd GradientWrtActivations(const Eigen::VectorXd &y,
36                                       const Eigen::VectorXd &a) override {
37     return -y.array() / a.array();
38 }
39 };
40
41 class KLDivergence final : public ICostFunction {
42 public:
43     double Apply(const Eigen::VectorXd &y, const Eigen::VectorXd &a) override {
44         return (y.array() * (y.array() / a.array()).log()).sum();
45     }
46
47     Eigen::VectorXd GradientWrtActivations(const Eigen::VectorXd &y,
48                                           const Eigen::VectorXd &a) override {
49         return -y.array() / a.array();
50     }
51 };
52
53 } // namespace nn

```

Листинг 3: Файл data_supplier.h

```

1  #pragma once
2
3  #include <Eigen/Dense>
4  #include <memory>
5  #include <vector>
6
7  #include "perceptron.h"
8
9  namespace nn {
10
11     struct Data final : nn::IData {
12         Eigen::VectorXd x, y;
13         std::string label;
14
15         const Eigen::VectorXd &GetX() const override { return x; }

```

```

16     const Eigen::VectorXd &GetY() const override { return y; }
17     std::string_view ToString() const override { return label; }
18 };
19
20 class DataSupplier final : public nn::IDataSupplier {
21     std::vector<std::shared_ptr<const nn::IData>> train_, test_, validation_;
22
23 public:
24     DataSupplier(const std::string &train_path, const std::string &test_path,
25                 const double false_score, const double true_score);
26
27     std::size_t GetInputLayerSize() const override;
28     std::size_t GetOutputLayerSize() const override;
29
30     std::vector<std::shared_ptr<const nn::IData>> GetTrainData() const override;
31     std::vector<std::shared_ptr<const nn::IData>> GetValidationData()
32         const override;
33     std::vector<std::shared_ptr<const nn::IData>> GetTestData() const override;
34 };
35
36 } // namespace nn

```

Листинг 4: Файл data_supplier.cc

```

1  #include "data_supplier.h"
2
3  #include <spdlog/spdlog.h>
4
5  #include <boost/algorithm/string/classification.hpp>
6  #include <boost/algorithm/string/split.hpp>
7  #include <cassert>
8  #include <fstream>
9  #include <iterator>
10 #include <stdexcept>
11 #include <string>
12
13 #include "perceptron.h"
14
15 namespace nn {
16

```

```

17 namespace {
18
19 constexpr std::size_t kDigitsNumber = 10;
20 constexpr std::size_t kScanSize = 784;
21 constexpr std::size_t kColumnsCount = kScanSize + 1;
22
23 std::vector<std::shared_ptr<const nn::IData>> ReadMnistCsv(
24     const std::string &filename, const double false_score,
25     const double true_score) {
26     static constexpr std::size_t kShadesCount = 255;
27
28     auto file = std::ifstream(filename);
29     if (!file.is_open()) {
30         throw std::runtime_error("Failed to open MNIST CSV file " + filename);
31     }
32
33     auto instances = std::vector<std::shared_ptr<const nn::IData>>{};
34     auto line = std::string{};
35     while (std::getline(file, line)) {
36         auto result = std::vector<std::string>{};
37         result.reserve(kColumnsCount);
38         boost::split(result, line, boost::is_any_of(","));
39
40         assert(result[0].size() == 1);
41         assert('0' <= result[0][0] && result[0][0] <= '9');
42
43         auto data = Data{};
44         data.label = result[0];
45
46         data.y = Eigen::VectorXd(kDigitsNumber);
47         data.y.setConstant(false_score);
48         data.y(data.label[0] - '0') = true_score;
49
50         data.x = Eigen::VectorXd(kScanSize);
51         for (std::size_t i = 1; i < kColumnsCount; ++i) {
52             data.x[i - 1] = std::stod(result[i]) / kShadesCount;
53         }
54
55         instances.push_back(std::make_shared<const Data>(std::move(data)));
56     }

```



```

57
58     return instances;
59 }
60
61 } // namespace
62
63 DataSupplier::DataSupplier(const std::string &train_path,
64                           const std::string &test_path,
65                           const double false_score, const double true_score) {
66     static constexpr std::size_t kTrainInitialSize = 60'000;
67     static constexpr std::size_t kValidationSize = 10'000;
68     static constexpr std::size_t kTestSize = 10'000;
69
70     spdlog::info("Parsing train data...");
71     train_ = ReadMnistCsv(train_path, false_score, true_score);
72     assert(train_.size() == kTrainInitialSize);
73
74     validation_ =
75         std::vector(std::make_move_iterator(train_.rbegin()),
76                   std::make_move_iterator(train_.rbegin() + kValidationSize));
77     train_.resize(kTrainInitialSize - kValidationSize);
78
79     spdlog::info("Parsing test data...");
80     test_ = ReadMnistCsv(test_path, false_score, true_score);
81     assert(test_.size() == kTestSize);
82 }
83
84 std::size_t DataSupplier::GetInputLayerSize() const { return kScanSize; }
85 std::size_t DataSupplier::GetOutputLayerSize() const { return kDigitsNumber; }
86
87 std::vector<std::shared_ptr<const nn::IData>> DataSupplier::GetTrainData()
88     const {
89     return train_;
90 }
91
92 std::vector<std::shared_ptr<const nn::IData>> DataSupplier::GetValidationData()
93     const {
94     return validation_;
95 }
96

```

```

97 std::vector<std::shared_ptr<const nn::IData>> DataSupplier::GetTestData()
98     const {
99     return test_;
100 }
101
102 } // namespace nn

```

Листинг 5: Файл perceptron.h

```

1  #pragma once
2
3  #include <Eigen/Dense>
4  #include <memory>
5  #include <random>
6
7  #include "activation_function.h"
8  #include "cost_function.h"
9
10 namespace nn {
11
12 class IData {
13 public:
14     virtual ~IData() = default;
15
16 public:
17     virtual const Eigen::VectorXd &GetX() const = 0;
18     virtual const Eigen::VectorXd &GetY() const = 0;
19     virtual std::string_view ToString() const = 0;
20 };
21
22 class IDataSupplier {
23 public:
24     virtual ~IDataSupplier() = default;
25
26 public:
27     virtual std::size_t GetInputLayerSize() const = 0;
28     virtual std::size_t GetOutputLayerSize() const = 0;
29
30     virtual std::vector<std::shared_ptr<const IData>> GetTrainData() const = 0;
31     virtual std::vector<std::shared_ptr<const IData>> GetTestData() const = 0;

```

```

32     virtual std::vector<std::shared_ptr<const IData>> GetValidationData()
33         const = 0;
34 };
35
36 struct SgdConfiguration final {
37     std::size_t epochs;
38     std::size_t mini_batch_size;
39     double learning_rate;
40     bool monitor_train_cost;
41     bool monitor_train_accuracy;
42     bool monitor_test_cost;
43     bool monitor_test_accuracy;
44 };
45
46 struct Metric final {
47     std::vector<double> train_cost, train_accuracy;
48     std::vector<double> test_cost, test_accuracy;
49 };
50
51 class Perceptron final {
52     std::random_device device_;
53     std::default_random_engine generator_;
54     std::unique_ptr<ICostFunction> cost_function_;
55     std::size_t layers_number_, connections_number_;
56     std::vector<Eigen::MatrixXd> weights_;
57     std::vector<Eigen::VectorXd> biases_;
58     std::vector<std::unique_ptr<IActivationFunction>> activation_functions_;
59
60 public:
61     Perceptron(
62         std::unique_ptr<ICostFunction> &&cost_function,
63         std::vector<std::unique_ptr<IActivationFunction>> &&activation_functions,
64         const std::vector<std::size_t> &layers_sizes);
65
66     Eigen::VectorXd Feedforward(const Eigen::VectorXd &x) const;
67
68     Metric Sgd(const std::vector<std::shared_ptr<const IData>> &train,
69               const std::vector<std::shared_ptr<const IData>> &test,
70               const SgdConfiguration &cfig);
71

```

```

72 Metric SgdNag(const std::vector<std::shared_ptr<const IData>> &train,
73               const std::vector<std::shared_ptr<const IData>> &test,
74               const SgdConfiguration &cfg, const double gamma);
75
76 Metric SgdAdagrad(const std::vector<std::shared_ptr<const IData>> &train,
77                  const std::vector<std::shared_ptr<const IData>> &test,
78                  const SgdConfiguration &cfg, const double epsilon);
79
80 Metric SgdAdam(const std::vector<std::shared_ptr<const IData>> &train,
81                const std::vector<std::shared_ptr<const IData>> &test,
82                const SgdConfiguration &cfg, const double beta1,
83                const double beta2, const double epsilon);
84
85 private:
86     // TODO: Use concepts
87     template <typename Iter>
88     void UpdateSgd(const Iter mini_batch_begin, const Iter mini_batch_end,
89                   const std::size_t mini_batch_size, const double learning_rate);
90
91     // EMA means Exponential Moving Average
92     template <typename Iter>
93     void UpdateSgdNag(std::vector<Eigen::MatrixXd> &delta_weights_ema,
94                      std::vector<Eigen::VectorXd> &delta_biases_ema,
95                      const Iter mini_batch_begin, const Iter mini_batch_end,
96                      const std::size_t mini_batch_size,
97                      const double learning_rate, const double gamma);
98
99     template <typename Iter>
100    void UpdateSgdAdagrad(
101        std::vector<Eigen::MatrixXd> &weights_gradient_squares_sum,
102        std::vector<Eigen::VectorXd> &biases_gradient_squares_sum,
103        const Iter mini_batch_begin, const Iter mini_batch_end,
104        const std::size_t mini_batch_size, const double learning_rate);
105
106    template <typename Iter>
107    void UpdateSgdAdam(std::vector<Eigen::MatrixXd> &weights_gradient_ema,
108                      std::vector<Eigen::VectorXd> &biases_gradient_ema,
109                      std::vector<Eigen::MatrixXd> &weights_squared_gradient_ema,
110                      std::vector<Eigen::VectorXd> &biases_squared_gradient_ema,
111                      const Iter mini_batch_begin, const Iter mini_batch_end,

```

```

112         const std::size_t mini_batch_size, const std::size_t epoch,
113         const double learning_rate, const double beta1,
114         const double beta2);
115
116 private:
117     struct Parameters {
118         std::vector<Eigen::MatrixXd> weights;
119         std::vector<Eigen::VectorXd> biases;
120     };
121
122     Parameters CreateParameters(const double initial_value) const;
123
124     template <typename Iter>
125     Parameters GradientWrtParameters(const Iter mini_batch_begin,
126                                     const Iter mini_batch_end,
127                                     const std::size_t mini_batch_size) const;
128
129     Parameters Backpropagation(const Eigen::VectorXd &x,
130                               const Eigen::VectorXd &y) const;
131
132     std::pair<std::vector<Eigen::VectorXd>, std::vector<Eigen::VectorXd>>
133     FeedforwardDetailed(const Eigen::VectorXd &x) const;
134
135     Metric CreateMetric(const SgdConfiguration &cfg) const;
136
137     void WriteMetric(Metric &metric, const std::size_t epoch,
138                    const std::vector<std::shared_ptr<const IData>> &train,
139                    const std::vector<std::shared_ptr<const IData>> &test,
140                    const SgdConfiguration &cfg) const;
141
142     template <typename Iter>
143     std::size_t CalculateAccuracy(const Iter begin, const Iter end) const;
144
145     template <typename Iter>
146     double CalculateCost(const Iter begin, const Iter end) const;
147 };
148
149 } // namespace nn

```

Листинг 6: Файл perceptron.cc

```

1  #include "perceptron.h"
2
3  #include <spdlog/spdlog.h>
4
5  #include <cassert>
6  #include <cmath>
7  #include <iostream>
8  #include <iterator>
9  #include <sstream>
10
11 namespace nn {
12
13 Perceptron::Perceptron(
14     std::unique_ptr<ICostFunction> &&cost_function,
15     std::vector<std::unique_ptr<IActivationFunction>> &&activation_functions,
16     const std::vector<std::size_t> &layers_sizes)
17 : generator_(device_()),
18   cost_function_(std::move(cost_function)),
19   layers_number_(layers_sizes.size()),
20   connections_number_(layers_number_ - 1),
21   activation_functions_(std::move(activation_functions)) {
22     if (layers_number_ < 2) {
23         throw std::runtime_error("Perceptron must have at least two layers");
24     }
25     if (activation_functions_.size() != connections_number_) {
26         throw std::runtime_error(
27             "Activation functions number must be equal to layers number minus one");
28     }
29
30     weights_.reserve(connections_number_);
31     biases_.reserve(connections_number_);
32     for (std::size_t i = 0; i < connections_number_; ++i) {
33         weights_.push_back(
34             Eigen::MatrixXd::Random(layers_sizes[i + 1], layers_sizes[i]));
35         biases_.push_back(Eigen::VectorXd::Random(layers_sizes[i + 1]));
36     }
37 }
38
39 Eigen::VectorXd Perceptron::Feedforward(const Eigen::VectorXd &x) const {

```

```

40     auto activation = x;
41     for (std::size_t i = 0; i < connections_number_; ++i) {
42         activation =
43             activation_functions_[i]->Apply(weights_[i] * activation + biases_[i]);
44     }
45     return activation;
46 }
47
48 Metric Perceptron::Sgd(const std::vector<std::shared_ptr<const IData>> &train,
49                       const std::vector<std::shared_ptr<const IData>> &test,
50                       const SgdConfiguration &cfg) {
51     const auto train_size = train.size();
52     const auto whole_mini_batches_number = train_size / cfg.mini_batch_size;
53     const auto remainder_mini_batch_size = train_size % cfg.mini_batch_size;
54
55     auto train_shuffled = std::vector(train.begin(), train.end());
56     auto metric = CreateMetric(cfg);
57     for (std::size_t i = 1; i <= cfg.epochs; ++i) {
58         std::shuffle(train_shuffled.begin(), train_shuffled.end(), generator_);
59         auto it = train_shuffled.begin();
60         for (std::size_t j = 0; j < whole_mini_batches_number; ++j) {
61             auto end = it + cfg.mini_batch_size;
62             UpdateSgd(it, end, cfg.mini_batch_size, cfg.learning_rate);
63             it = std::move(end);
64         }
65
66         if (remainder_mini_batch_size != 0) {
67             UpdateSgd(it, it + remainder_mini_batch_size, remainder_mini_batch_size,
68                     cfg.learning_rate);
69         }
70         WriteMetric(metric, i, train, test, cfg);
71     }
72
73     return metric;
74 }
75
76 Metric Perceptron::SgdNag(
77     const std::vector<std::shared_ptr<const IData>> &train,
78     const std::vector<std::shared_ptr<const IData>> &test,
79     const SgdConfiguration &cfg, const double gamma) {

```

```

80     if (gamma < 0 || gamma > 1) {
81         throw std::runtime_error("Gamma must belong to [0, 1]");
82     }
83
84     const auto train_size = train.size();
85     const auto whole_mini_batches_number = train_size / cfg.mini_batch_size;
86     const auto remainder_mini_batch_size = train_size % cfg.mini_batch_size;
87
88     auto [delta_weights_ema, delta_biases_ema] = CreateParameters(0);
89     auto train_shuffled = std::vector(train.begin(), train.end());
90     auto metric = CreateMetric(cfg);
91     for (std::size_t i = 1; i <= cfg.epochs; ++i) {
92         std::shuffle(train_shuffled.begin(), train_shuffled.end(), generator_);
93         auto it = train_shuffled.begin();
94         for (std::size_t j = 0; j < whole_mini_batches_number; ++j) {
95             auto end = it + cfg.mini_batch_size;
96             UpdateSgdNag(delta_weights_ema, delta_biases_ema, it, end,
97                         cfg.mini_batch_size, cfg.learning_rate, gamma);
98             it = std::move(end);
99         }
100
101         if (remainder_mini_batch_size != 0) {
102             UpdateSgdNag(delta_weights_ema, delta_biases_ema, it,
103                         it + remainder_mini_batch_size, remainder_mini_batch_size,
104                         cfg.learning_rate, gamma);
105         }
106         WriteMetric(metric, i, train, test, cfg);
107     }
108
109     return metric;
110 }
111
112 Metric Perceptron::SgdAdagrad(
113     const std::vector<std::shared_ptr<const IData>> &train,
114     const std::vector<std::shared_ptr<const IData>> &test,
115     const SgdConfiguration &cfg, const double epsilon) {
116     if (epsilon <= 0) {
117         throw std::runtime_error("Epsilon must be strictly greater than 0");
118     }
119

```



```

120 const auto train_size = train.size();
121 const auto whole_mini_batches_number = train_size / cfg.mini_batch_size;
122 const auto remainder_mini_batch_size = train_size % cfg.mini_batch_size;
123
124 auto [weights_gradient_squares_sum, biases_gradient_squares_sum] =
125     CreateParameters(epsilon);
126 auto train_shuffled = std::vector(train.begin(), train.end());
127 auto metric = CreateMetric(cfg);
128 for (std::size_t i = 1; i <= cfg.epochs; ++i) {
129     std::shuffle(train_shuffled.begin(), train_shuffled.end(), generator_);
130     auto it = train_shuffled.begin();
131     for (std::size_t i = 0; i < whole_mini_batches_number; ++i) {
132         auto end = it + cfg.mini_batch_size;
133         UpdateSgdAdagrad(weights_gradient_squares_sum,
134             biases_gradient_squares_sum, it, end,
135             cfg.mini_batch_size, cfg.learning_rate);
136         it = std::move(end);
137     }
138
139     if (remainder_mini_batch_size != 0) {
140         UpdateSgdAdagrad(weights_gradient_squares_sum,
141             biases_gradient_squares_sum, it,
142             it + remainder_mini_batch_size,
143             remainder_mini_batch_size, cfg.learning_rate);
144     }
145     WriteMetric(metric, i, train, test, cfg);
146 }
147
148 return metric;
149 }
150
151 Metric Perceptron::SgdAdam(
152     const std::vector<std::shared_ptr<const IData>> &train,
153     const std::vector<std::shared_ptr<const IData>> &test,
154     const SgdConfiguration &cfg, const double beta1, const double beta2,
155     const double epsilon) {
156     if (beta1 < 0 || beta1 > 1) {
157         throw std::runtime_error("Beta1 must belong to [0, 1]");
158     }
159     if (beta2 < 0 || beta2 > 1) {

```

```

160     throw std::runtime_error("Beta2 must belong to [0, 1]");
161 }
162 if (epsilon <= 0) {
163     throw std::runtime_error("Epsilon must be strictly greater than 0");
164 }
165
166 const auto train_size = train.size();
167 const auto whole_mini_batches_number = train_size / cfg.mini_batch_size;
168 const auto remainder_mini_batch_size = train_size % cfg.mini_batch_size;
169
170 auto [weights_gradient_ema, biases_gradient_ema] = CreateParameters(0);
171 auto [weights_squared_gradient_ema, biases_squared_gradient_ema] =
172     CreateParameters(epsilon);
173 auto train_shuffled = std::vector(train.begin(), train.end());
174 auto metric = CreateMetric(cfg);
175 for (std::size_t i = 1; i <= cfg.epochs; ++i) {
176     std::shuffle(train_shuffled.begin(), train_shuffled.end(), generator_);
177     auto it = train_shuffled.begin();
178     for (std::size_t j = 0; j < whole_mini_batches_number; ++j) {
179         auto end = it + cfg.mini_batch_size;
180         UpdateSgdAdam(weights_gradient_ema, biases_gradient_ema,
181             weights_squared_gradient_ema, biases_squared_gradient_ema,
182             it, end, cfg.mini_batch_size, i, cfg.learning_rate, beta1,
183             beta2);
184         it = std::move(end);
185     }
186
187     if (remainder_mini_batch_size != 0) {
188         UpdateSgdAdam(weights_gradient_ema, biases_gradient_ema,
189             weights_squared_gradient_ema, biases_squared_gradient_ema,
190             it, it + remainder_mini_batch_size,
191             remainder_mini_batch_size, i, cfg.learning_rate, beta1,
192             beta2);
193     }
194     WriteMetric(metric, i, train, test, cfg);
195 }
196
197 return metric;
198 }
199

```

```

200 template <typename Iter>
201 void Perceptron::UpdateSgd(const Iter mini_batch_begin,
202                             const Iter mini_batch_end,
203                             const std::size_t mini_batch_size,
204                             const double learning_rate) {
205     const auto [weights_gradient, biases_gradient] =
206         GradientWrtParameters(mini_batch_begin, mini_batch_end, mini_batch_size);
207
208     for (std::size_t i = 0; i < connections_number_; ++i) {
209         weights_[i] -= learning_rate * weights_gradient[i];
210         biases_[i] -= learning_rate * biases_gradient[i];
211     }
212 }
213
214 template <typename Iter>
215 void Perceptron::UpdateSgdNag(std::vector<Eigen::MatrixXd> &delta_weights_ema,
216                               std::vector<Eigen::VectorXd> &delta_biases_ema,
217                               const Iter mini_batch_begin,
218                               const Iter mini_batch_end,
219                               const std::size_t mini_batch_size,
220                               const double learning_rate, const double gamma) {
221     for (std::size_t i = 0; i < connections_number_; ++i) {
222         weights_[i] -= gamma * delta_weights_ema[i];
223         biases_[i] -= gamma * delta_biases_ema[i];
224     }
225
226     auto [weights_gradient, biases_gradient] =
227         GradientWrtParameters(mini_batch_begin, mini_batch_end, mini_batch_size);
228
229     for (std::size_t i = 0; i < connections_number_; ++i) {
230         const auto saved_delta_weights_ema = gamma * delta_weights_ema[i];
231         delta_weights_ema[i] =
232             saved_delta_weights_ema + learning_rate * weights_gradient[i];
233         weights_[i] += saved_delta_weights_ema - delta_weights_ema[i];
234
235         const auto saved_delta_biases_ema = gamma * delta_biases_ema[i];
236         delta_biases_ema[i] =
237             saved_delta_biases_ema + learning_rate * biases_gradient[i];
238         biases_[i] += saved_delta_biases_ema - delta_biases_ema[i];
239     }

```

```

240 }
241
242 template <typename Iter>
243 void Perceptron::UpdateSgdAdagrad(
244     std::vector<Eigen::MatrixXd> &weights_gradient_squares_sum,
245     std::vector<Eigen::VectorXd> &biases_gradient_squares_sum,
246     const Iter mini_batch_begin, const Iter mini_batch_end,
247     const std::size_t mini_batch_size, const double learning_rate) {
248     auto [weights_gradient, biases_gradient] =
249         GradientWrtParameters(mini_batch_begin, mini_batch_end, mini_batch_size);
250
251     for (std::size_t i = 0; i < connections_number_; ++i) {
252         weights_gradient_squares_sum[i] +=
253             weights_gradient[i].array().pow(2).matrix();
254         weights_[i] -= learning_rate / weights_gradient_squares_sum[i].lpNorm<2>() *
255             weights_gradient[i];
256
257         biases_gradient_squares_sum[i] +=
258             biases_gradient[i].array().pow(2).matrix();
259         biases_[i] -= learning_rate / biases_gradient_squares_sum[i].lpNorm<2>() *
260             biases_gradient[i];
261     }
262 }
263
264 template <typename Iter>
265 void Perceptron::UpdateSgdAdam(
266     std::vector<Eigen::MatrixXd> &weights_gradient_ema,
267     std::vector<Eigen::VectorXd> &biases_gradient_ema,
268     std::vector<Eigen::MatrixXd> &weights_squared_gradient_ema,
269     std::vector<Eigen::VectorXd> &biases_squared_gradient_ema,
270     const Iter mini_batch_begin, const Iter mini_batch_end,
271     const std::size_t mini_batch_size, const std::size_t epoch,
272     const double learning_rate, const double beta1, const double beta2) {
273     auto [weights_gradient, biases_gradient] =
274         GradientWrtParameters(mini_batch_begin, mini_batch_end, mini_batch_size);
275
276     for (std::size_t i = 0; i < connections_number_; ++i) {
277         weights_gradient_ema[i] =
278             beta1 * weights_gradient_ema[i] + (1 - beta1) * weights_gradient[i];
279         biases_gradient_ema[i] =

```

```

280     beta1 * biases_gradient_ema[i] + (1 - beta1) * biases_gradient[i];
281
282     const auto adjusted_weights_gradient_ema =
283         weights_gradient_ema[i] / (1 - std::pow(beta1, epoch));
284     const auto adjusted_biases_gradient_ema =
285         biases_gradient_ema[i] / (1 - std::pow(beta1, epoch));
286
287     weights_squared_gradient_ema[i] =
288         beta2 * weights_squared_gradient_ema[i] +
289         (1 - beta2) * weights_gradient[i].array().pow(2).matrix();
290     biases_squared_gradient_ema[i] =
291         beta2 * biases_squared_gradient_ema[i] +
292         (1 - beta2) * biases_gradient[i].array().pow(2).matrix();
293
294     const auto adjusted_weights_squared_gradient_ema =
295         weights_squared_gradient_ema[i] / (1 - std::pow(beta2, epoch));
296     const auto adjusted_biases_squared_gradient_ema =
297         biases_squared_gradient_ema[i] / (1 - std::pow(beta2, epoch));
298
299     weights_[i] -= learning_rate /
300         adjusted_weights_squared_gradient_ema.lpNorm<2>() *
301         adjusted_weights_gradient_ema;
302     biases_[i] -= learning_rate /
303         adjusted_biases_squared_gradient_ema.lpNorm<2>() *
304         adjusted_biases_gradient_ema;
305 }
306 }
307
308 Perceptron::Parameters Perceptron::CreateParameters(
309     const double initial_value) const {
310     auto weights = std::vector<Eigen::MatrixXd>{};
311     weights.reserve(weights_.size());
312     for (auto &&w : weights_) {
313         auto m = Eigen::MatrixXd(w.rows(), w.cols());
314         m.setConstant(initial_value);
315         weights.push_back(std::move(m));
316     }
317
318     auto biases = std::vector<Eigen::VectorXd>{};
319     biases.reserve(biases_.size());

```

```

320     for (auto &&b : biases_) {
321         auto v = Eigen::VectorXd(b.size());
322         v.setConstant(initial_value);
323         biases.push_back(std::move(v));
324     }
325
326     return {std::move(weights), std::move(biases)};
327 }
328
329 template <typename Iter>
330 Perceptron::Parameters Perceptron::GradientWrtParameters(
331     const Iter mini_batch_begin, const Iter mini_batch_end,
332     const std::size_t mini_batch_size) const {
333     auto [weights_gradient, biases_gradient] = CreateParameters(0);
334
335     for (auto it = mini_batch_begin; it != mini_batch_end; ++it) {
336         const auto &data = **it;
337         const auto [weights_gradient_contribution, biases_gradient_contribution] =
338             Backpropagation(data.GetX(), data.GetY());
339         for (std::size_t i = 0; i < connections_number_; ++i) {
340             weights_gradient[i] += weights_gradient_contribution[i];
341             biases_gradient[i] += biases_gradient_contribution[i];
342         }
343     }
344
345     const auto factor = 1.0 / mini_batch_size;
346     for (std::size_t i = 0; i < connections_number_; ++i) {
347         weights_gradient[i] *= factor;
348         biases_gradient[i] *= factor;
349     }
350
351     return {std::move(weights_gradient), std::move(biases_gradient)};
352 }
353
354 Perceptron::Parameters Perceptron::Backpropagation(
355     const Eigen::VectorXd &x, const Eigen::VectorXd &y) const {
356     const auto [linear_values, activations] = FeedforwardDetailed(x);
357     assert(linear_values.size() == connections_number_);
358     assert(activations.size() == layers_number_);
359

```

```

360 auto delta = static_cast<Eigen::VectorXd>(
361     activation_functions_.back()->Jacobian(linear_values.back()).transpose() *
362     cost_function_->GradientWrtActivations(y, activations.back()));
363
364 auto nabla_weights_reversed = std::vector<Eigen::MatrixXd>{};
365 nabla_weights_reversed.reserve(connections_number_);
366 nabla_weights_reversed.push_back(
367     delta * std::prev(activations.cend(), 2)->transpose());
368
369 auto nabla_biases_reversed = std::vector<Eigen::VectorXd>{};
370 nabla_biases_reversed.reserve(connections_number_);
371 nabla_biases_reversed.push_back(delta);
372
373 for (int i = connections_number_ - 2; i >= 0; --i) {
374     delta =
375         (weights_[i + 1] * activation_functions_[i]->Jacobian(linear_values[i]))
376         .transpose() *
377         delta;
378     nabla_weights_reversed.push_back(delta * activations[i].transpose());
379     nabla_biases_reversed.push_back(delta);
380 }
381
382 return {{std::make_move_iterator(nabla_weights_reversed.rbegin()),
383         std::make_move_iterator(nabla_weights_reversed.rend())},
384         {std::make_move_iterator(nabla_biases_reversed.rbegin()),
385         std::make_move_iterator(nabla_biases_reversed.rend())}};
386 }
387
388 std::pair<std::vector<Eigen::VectorXd>, std::vector<Eigen::VectorXd>>
389 Perceptron::FeedforwardDetailed(const Eigen::VectorXd &x) const {
390     std::vector<Eigen::VectorXd> linear_values, activations;
391     linear_values.reserve(connections_number_);
392     activations.reserve(layers_number_);
393
394     auto activation = x;
395     for (std::size_t i = 0; i < connections_number_; ++i) {
396         auto linear_value =
397             static_cast<Eigen::VectorXd>(weights_[i] * activation + biases_[i]);
398         activations.push_back(std::move(activation));
399         activation = activation_functions_[i]->Apply(linear_value);

```

```

400     linear_values.push_back(std::move(linear_value));
401 }
402 activations.push_back(std::move(activation));
403
404     return {std::move(linear_values), std::move(activations)};
405 }
406
407 Metric Perceptron::CreateMetric(const SgdConfiguration &cfg) const {
408     auto metric = Metric{};
409     if (cfg.monitor_train_cost) {
410         metric.train_cost.reserve(cfg.epochs);
411     }
412     if (cfg.monitor_train_accuracy) {
413         metric.train_accuracy.reserve(cfg.epochs);
414     }
415     if (cfg.monitor_test_cost) {
416         metric.test_cost.reserve(cfg.epochs);
417     }
418     if (cfg.monitor_test_accuracy) {
419         metric.test_accuracy.reserve(cfg.epochs);
420     }
421     return metric;
422 }
423
424 void Perceptron::WriteMetric(
425     Metric &metric, const std::size_t epoch,
426     const std::vector<std::shared_ptr<const IData>> &train,
427     const std::vector<std::shared_ptr<const IData>> &test,
428     const SgdConfiguration &cfg) const {
429     std::stringstream oss;
430     oss << "Epoch " << epoch << "/" << cfg.epochs << ";";
431     if (cfg.monitor_train_cost) {
432         const auto train_cost = CalculateCost(train.begin(), train.end());
433         metric.train_cost.push_back(train_cost);
434         oss << " train cost: " << train_cost << ";";
435     }
436     if (cfg.monitor_train_accuracy) {
437         const auto train_accuracy = CalculateAccuracy(train.begin(), train.end());
438         metric.train_accuracy.push_back(train_accuracy);
439         oss << " train accuracy: " << train_accuracy << "/" << train.size() << ";";

```



```

440 }
441 if (cfg.monitor_test_cost) {
442     const auto test_cost = CalculateCost(test.begin(), test.end());
443     metric.test_cost.push_back(CalculateCost(test.begin(), test.end()));
444     oss << " test cost: " << test_cost << ",";
445 }
446 if (cfg.monitor_test_accuracy) {
447     const auto test_accuracy = CalculateAccuracy(test.begin(), test.end());
448     metric.test_accuracy.push_back(test_accuracy);
449     oss << " test accuracy: " << test_accuracy << "/" << test.size() << ",";
450 }
451 spdlog::info(oss.str());
452 }
453
454 template <typename Iter>
455 std::size_t Perceptron::CalculateAccuracy(const Iter begin,
456                                         const Iter end) const {
457     std::size_t right_predictions = 0;
458     for (auto it = begin; it != end; ++it) {
459         const IData &instance = **it;
460         Eigen::Index max_activation_expected, max_activation_actual;
461         instance.GetY().maxCoeff(&max_activation_expected);
462         Feedforward(instance.GetX()).maxCoeff(&max_activation_actual);
463         if (max_activation_expected == max_activation_actual) {
464             ++right_predictions;
465         }
466     }
467     return right_predictions;
468 }
469
470 template <typename Iter>
471 double Perceptron::CalculateCost(const Iter begin, const Iter end) const {
472     double cost = 0;
473     std::size_t instances_count = 0;
474     for (auto it = begin; it != end; ++it, ++instances_count) {
475         const IData &instance = **it;
476         const auto activation = Feedforward(instance.GetX());
477         cost += cost_function_ -> Apply(instance.GetY(), activation);
478     }
479     return cost / instances_count;

```

```

480 }
481
482 } // namespace nn

```

Листинг 7: Файл chromosome.h

```

1  #pragma once
2
3  #include <memory>
4  #include <vector>
5
6  #include "fitness_function.h"
7
8  namespace nn {
9
10 enum class ChromosomeSubclass {
11     kSgdHyperparametersKit,
12 };
13
14 class IChromosome {
15 public:
16     static std::shared_ptr<IChromosome> Create(std::vector<double>&& genes,
17                                                const ChromosomeSubclass subclass);
18
19 public:
20     virtual ~IChromosome() = default;
21
22     virtual std::string ToString() const = 0;
23
24 public:
25     const std::vector<double>& get_genes() const;
26
27 protected:
28     IChromosome(std::vector<double>&& genes);
29
30     std::vector<double> genes_;
31 };
32
33 class SgdHyperparametersKit final : public IChromosome {
34     enum Index : std::size_t {

```

```

35     kLearningRate,
36     kEpochs,
37     kMiniBatchSize,
38     kHiddenLayers,
39     kNeuronsPerHiddenLayer,
40 };
41
42 static constexpr std::size_t kHyperparametersNumber = 5;
43
44 public:
45     SgdHyperparametersKit(std::vector<double>&& hyperparameters);
46
47     double get_learning_rate() const;
48     std::size_t get_epochs() const;
49     std::size_t get_mini_batch_size() const;
50     std::size_t get_hidden_layers() const;
51     std::size_t get_neurons_per_hidden_layer() const;
52
53     std::string ToString() const override;
54 };
55
56 } // namespace nn

```

Листинг 8: Файл chromosome.cc

```

1  #include "chromosome.h"
2
3  #include <sstream>
4  #include <stdexcept>
5
6  namespace nn {
7
8  std::shared_ptr<IChromosome> IChromosome::Create(
9      std::vector<double>&& genes, const ChromosomeSubclass subclass) {
10     switch (subclass) {
11         case ChromosomeSubclass::kSgdHyperparametersKit:
12             return std::make_shared<SgdHyperparametersKit>(std::move(genes));
13     }
14 }
15

```

```

16  const std::vector<double>& IChromosome::get_genes() const { return genes_; }
17
18  IChromosome::IChromosome(std::vector<double>&& genes)
19      : genes_(std::move(genes)) {}
20
21  SgdHyperparametersKit::SgdHyperparametersKit(
22      std::vector<double>&& hyperparameters)
23      : IChromosome(std::move(hyperparameters)) {
24      if (genes_.size() != kHyperparametersNumber) {
25          throw std::runtime_error(
26              "Got " + std::to_string(genes_.size()) + " SGD hyperparameters, " +
27              std::to_string(kHyperparametersNumber) + " expected");
28      }
29  }
30
31  double SgdHyperparametersKit::get_learning_rate() const {
32      return genes_.at(Index::kLearningRate);
33  }
34
35  std::size_t SgdHyperparametersKit::get_epochs() const {
36      return genes_.at(Index::kEpochs);
37  }
38
39  std::size_t SgdHyperparametersKit::get_mini_batch_size() const {
40      return genes_.at(Index::kMiniBatchSize);
41  }
42
43  std::size_t SgdHyperparametersKit::get_hidden_layers() const {
44      return genes_.at(Index::kHiddenLayers);
45  }
46
47  std::size_t SgdHyperparametersKit::get_neurons_per_hidden_layer() const {
48      return genes_.at(Index::kNeuronsPerHiddenLayer);
49  }
50
51  std::string SgdHyperparametersKit::ToString() const {
52      auto oss = std::ostringstream{};
53      oss << "- Learning rate: " << get_learning_rate()
54          << "; \n- Epochs: " << get_epochs()
55          << "; \n- Mini-batch size: " << get_mini_batch_size()

```

```

56     << "< \n- Hidden layers: " << get_hidden_layers()
57     << "< \n- Neurons per hidden layer: " << get_neurons_per_hidden_layer();
58     return oss.str();
59 }
60
61 } // namespace nn

```

Листинг 9: Файл fitness_function.h

```

1  #pragma once
2
3  #include <memory>
4
5  #include "perceptron.h"
6
7  namespace nn {
8
9  class IChromosome;
10
11  class IFitnessFunction {
12  public:
13      virtual ~IFitnessFunction() = default;
14
15      virtual double Assess(const IChromosome& chromosome) const = 0;
16  };
17
18  class ISgdFitness : public IFitnessFunction {
19  public:
20      virtual ~ISgdFitness() = default;
21
22      ISgdFitness(std::unique_ptr<IDataSupplier>&& data_supplier);
23
24  protected:
25      std::unique_ptr<IDataSupplier> data_supplier_;
26  };
27
28  class SgdFitness final : public ISgdFitness {
29  public:
30      using ISgdFitness::ISgdFitness;
31

```

```

32     double Assess(const IChromosome& chromosome) const override;
33 };
34
35 class SgdNagFitness final : public ISgdFitness {
36 public:
37     using ISgdFitness::ISgdFitness;
38
39     double Assess(const IChromosome& chromosome) const override;
40 };
41
42 class SgdAdagradFitness final : public ISgdFitness {
43 public:
44     using ISgdFitness::ISgdFitness;
45
46     double Assess(const IChromosome& chromosome) const override;
47 };
48
49 class SgdAdamFitness final : public ISgdFitness {
50 public:
51     using ISgdFitness::ISgdFitness;
52
53     double Assess(const IChromosome& chromosome) const override;
54 };
55
56 } // namespace nn

```

Листинг 10: Файл fitness_function.cc

```

1  #include "fitness_function.h"
2
3  #include <cassert>
4
5  #include "chromosome.h"
6  #include "cost_function.h"
7  #include "perceptron.h"
8
9  namespace nn {
10
11  namespace {
12

```

```

13 double CostToFitness(const double value) {
14     return std::exp(1 / (value + 1e-8));
15 }
16
17 } // namespace
18
19 ISgdFitness::ISgdFitness(std::unique_ptr<IDataSupplier>&& data_supplier)
20     : data_supplier_(std::move(data_supplier)) {}
21
22 double SgdFitness::Assess(const IChromosome& chromosome) const {
23     const auto kit = static_cast<const SgdHyperparametersKit&>(chromosome);
24
25     auto cost_function = std::make_unique<CrossEntropy>();
26
27     auto activation_functions =
28         std::vector<std::unique_ptr<IActivationFunction>>{};
29     const auto hidden_layers = kit.get_hidden_layers();
30     activation_functions.reserve(hidden_layers + 1);
31     for (std::size_t i = 0; i < hidden_layers; ++i) {
32         activation_functions.push_back(std::make_unique<LeakyReLU>(0.01));
33     }
34     activation_functions.push_back(std::make_unique<Softmax>());
35
36     auto layers_sizes = std::vector<std::size_t>{};
37     layers_sizes.reserve(hidden_layers + 2);
38     layers_sizes.push_back(data_supplier_->GetInputLayerSize());
39     const auto neurons_per_hidden_layer = kit.get_neurons_per_hidden_layer();
40     for (std::size_t i = 0; i < hidden_layers; ++i) {
41         layers_sizes.push_back(neurons_per_hidden_layer);
42     }
43     layers_sizes.push_back(data_supplier_->GetOutputLayerSize());
44
45     auto perceptron = Perceptron(std::move(cost_function),
46                                   std::move(activation_functions), layers_sizes);
47     const auto cfg = SgdConfiguration{
48         .epochs = kit.get_epochs(),
49         .mini_batch_size = kit.get_mini_batch_size(),
50         .learning_rate = kit.get_learning_rate(),
51         .monitor_train_cost = true,
52         .monitor_train_accuracy = true,

```

```

53     .monitor_test_cost = true,
54     .monitor_test_accuracy = true,
55 };
56
57 const auto train_data = data_supplier_->GetTrainData();
58 const auto test_data = data_supplier_->GetTestData();
59
60 auto metrics = perceptron.Sgd(train_data, test_data, cfg);
61 return CostToFitness(metrics.test_cost.back());
62 }
63
64 double SgdNagFitness::Assess(const IChromosome& chromosome) const {
65     constexpr double kGamma = 0.9;
66
67     const auto kit = static_cast<const SgdHyperparametersKit&>(chromosome);
68
69     auto cost_function = std::make_unique<CrossEntropy>();
70
71     auto activation_functions =
72         std::vector<std::unique_ptr<IActivationFunction>>{};
73     const auto hidden_layers = kit.get_hidden_layers();
74     activation_functions.reserve(hidden_layers + 1);
75     for (std::size_t i = 0; i < hidden_layers; ++i) {
76         activation_functions.push_back(std::make_unique<LeakyReLU>(0.01));
77     }
78     activation_functions.push_back(std::make_unique<Softmax>());
79
80     auto layers_sizes = std::vector<std::size_t>{};
81     layers_sizes.reserve(hidden_layers + 2);
82     layers_sizes.push_back(data_supplier_->GetInputLayerSize());
83     const auto neurons_per_hidden_layer = kit.get_neurons_per_hidden_layer();
84     for (std::size_t i = 0; i < hidden_layers; ++i) {
85         layers_sizes.push_back(neurons_per_hidden_layer);
86     }
87     layers_sizes.push_back(data_supplier_->GetOutputLayerSize());
88
89     auto perceptron = Perceptron(std::move(cost_function),
90                                   std::move(activation_functions), layers_sizes);
91     const auto cfg = SgdConfiguration{
92         .epochs = kit.get_epochs(),

```



```

93     .mini_batch_size = kit.get_mini_batch_size(),
94     .learning_rate = kit.get_learning_rate(),
95     .monitor_train_cost = true,
96     .monitor_train_accuracy = true,
97     .monitor_test_cost = true,
98     .monitor_test_accuracy = true,
99 };
100
101 const auto train_data = data_supplier_->GetTrainData();
102 const auto test_data = data_supplier_->GetTestData();
103
104 auto metrics = perceptron.SgdNag(train_data, test_data, cfg, kGamma);
105 return CostToFitness(metrics.test_cost.back());
106 }
107
108 double SgdAdagradFitness::Assess(const IChromosome& chromosome) const {
109     constexpr double kEpsilon = 1e-8;
110
111     const auto kit = static_cast<const SgdHyperparametersKit&>(chromosome);
112
113     auto cost_function = std::make_unique<CrossEntropy>();
114
115     auto activation_functions =
116         std::vector<std::unique_ptr<IActivationFunction>>{};
117     const auto hidden_layers = kit.get_hidden_layers();
118     activation_functions.reserve(hidden_layers + 1);
119     for (std::size_t i = 0; i < hidden_layers; ++i) {
120         activation_functions.push_back(std::make_unique<LeakyReLU>(0.01));
121     }
122     activation_functions.push_back(std::make_unique<Softmax>());
123
124     auto layers_sizes = std::vector<std::size_t>{};
125     layers_sizes.reserve(hidden_layers + 2);
126     layers_sizes.push_back(data_supplier_->GetInputLayerSize());
127     const auto neurons_per_hidden_layer = kit.get_neurons_per_hidden_layer();
128     for (std::size_t i = 0; i < hidden_layers; ++i) {
129         layers_sizes.push_back(neurons_per_hidden_layer);
130     }
131     layers_sizes.push_back(data_supplier_->GetOutputLayerSize());
132

```

```

133     auto perceptron = Perceptron(std::move(cost_function),
134                                   std::move(activation_functions), layers_sizes);
135     const auto cfg = SgdConfiguration{
136         .epochs = kit.get_epochs(),
137         .mini_batch_size = kit.get_mini_batch_size(),
138         .learning_rate = kit.get_learning_rate(),
139         .monitor_train_cost = true,
140         .monitor_train_accuracy = true,
141         .monitor_test_cost = true,
142         .monitor_test_accuracy = true,
143     };
144
145     const auto train_data = data_supplier_ -> GetTrainData();
146     const auto test_data = data_supplier_ -> GetTestData();
147
148     auto metrics = perceptron.SgdAdagrad(train_data, test_data, cfg, kEpsilon);
149     return CostToFitness(metrics.test_cost.back());
150 }
151
152 double SgdAdamFitness::Assess(const IChromosome& chromosome) const {
153     constexpr double kEpsilon = 1e-8;
154     constexpr double kBeta1 = 0.9;
155     constexpr double kBeta2 = 0.999;
156
157     const auto kit = static_cast<const SgdHyperparametersKit&>(chromosome);
158
159     auto cost_function = std::make_unique<CrossEntropy>();
160
161     auto activation_functions =
162         std::vector<std::unique_ptr<IActivationFunction>>{};
163     const auto hidden_layers = kit.get_hidden_layers();
164     activation_functions.reserve(hidden_layers + 1);
165     for (std::size_t i = 0; i < hidden_layers; ++i) {
166         activation_functions.push_back(std::make_unique<LeakyReLU>(0.01));
167     }
168     activation_functions.push_back(std::make_unique<Softmax>());
169
170     auto layers_sizes = std::vector<std::size_t>{};
171     layers_sizes.reserve(hidden_layers + 2);
172     layers_sizes.push_back(data_supplier_ -> GetInputLayerSize());

```

```

173     const auto neurons_per_hidden_layer = kit.get_neurons_per_hidden_layer();
174     for (std::size_t i = 0; i < hidden_layers; ++i) {
175         layers_sizes.push_back(neurons_per_hidden_layer);
176     }
177     layers_sizes.push_back(data_supplier_ -> GetOutputLayerSize());
178
179     auto perceptron = Perceptron(std::move(cost_function),
180                                   std::move(activation_functions), layers_sizes);
181     const auto cfg = SgdConfiguration{
182         .epochs = kit.get_epochs(),
183         .mini_batch_size = kit.get_mini_batch_size(),
184         .learning_rate = kit.get_learning_rate(),
185         .monitor_train_cost = true,
186         .monitor_train_accuracy = true,
187         .monitor_test_cost = true,
188         .monitor_test_accuracy = true,
189     };
190
191     const auto train_data = data_supplier_ -> GetTrainData();
192     const auto test_data = data_supplier_ -> GetTestData();
193
194     auto metrics =
195         perceptron.SgdAdam(train_data, test_data, cfg, kBeta1, kBeta2, kEpsilon);
196     return CostToFitness(metrics.test_cost.back());
197 }
198
199 } // namespace nn

```

ЛИСТИНГ 11: Файл genetic_algorithm.h

```

1  #pragma once
2
3  #include <memory>
4  #include <random>
5  #include <vector>
6
7  #include "chromosome.h"
8  #include "fitness_function.h"
9
10 namespace nn {

```

```

11
12 class Segment final {
13     double left_, right_;
14
15 public:
16     Segment(const double left, const double right);
17
18     double get_left() const;
19     double get_right() const;
20 };
21
22 class GeneticAlgorithm final {
23 public:
24     // TODO: Validate the configuration
25     struct Configuration final {
26         std::size_t populations_number;
27         std::size_t population_size;
28         double crossover_proportion;
29         double mutation_proportion;
30     };
31
32 private:
33     std::mt19937 engine_{std::random_device{}}();
34
35     std::unique_ptr<IFitnessFunction> fitness_function_;
36     ChromosomeSubclass chromosome_subclass_;
37     std::vector<std::uniform_real_distribution<double>> genes_distributions_;
38     std::vector<std::shared_ptr<IChromosome>> population_;
39     Configuration cfg_;
40     std::size_t genes_number_;
41
42 public:
43     GeneticAlgorithm(std::unique_ptr<IFitnessFunction>&& fitness_function,
44                     const ChromosomeSubclass subclass,
45                     const std::vector<Segment>& segments,
46                     const Configuration& cfg);
47
48     std::shared_ptr<IChromosome> Run();
49
50 private:

```

```

51     std::vector<std::shared_ptr<IChromosome>> RouletteWheelSelection();
52     void Crossover(std::vector<std::shared_ptr<IChromosome>>& population);
53     void Mutate(std::vector<std::shared_ptr<IChromosome>>& population);
54
55     std::vector<double> CalculateFitnessValue() const;
56 };
57
58 } // namespace nn

```

Листинг 12: Файл genetic_algorithm.cc

```

1  #include "genetic_algorithm.h"
2
3  #include <spdlog/spdlog.h>
4
5  #include <algorithm>
6  #include <cassert>
7  #include <cmath>
8  #include <map>
9  #include <numeric>
10 #include <random>
11 #include <stdexcept>
12
13 #include "chromosome.h"
14
15 namespace nn {
16
17 Segment::Segment(const double left, const double right)
18     : left_(left), right_(right) {
19     if (left_ > right_) {
20         throw std::runtime_error(
21             "The left border must be less or equal than the right one");
22     }
23 }
24
25 double Segment::get_left() const { return left_; }
26 double Segment::get_right() const { return right_; }
27
28 GeneticAlgorithm::GeneticAlgorithm(
29     std::unique_ptr<IFitnessFunction>&& fitness_function,

```

```

30     const ChromosomeSubclass subclass, const std::vector<Segment>& segments,
31     const GeneticAlgorithm::Configuration& cfg)
32     : fitness_function_(std::move(fitness_function)),
33       chromosome_subclass_(subclass),
34       cfg_(cfg),
35       genes_number_(segments.size()) {
36     genes_distributions_.reserve(genes_number_);
37     for (auto&& segment : segments) {
38         genes_distributions_.push_back(std::uniform_real_distribution<>(
39             segment.get_left(), segment.get_right()));
40     }
41
42     population_.reserve(cfg.population_size);
43     for (std::size_t i = 0; i < cfg.population_size; ++i) {
44         auto genes = std::vector<double>{};
45         genes.reserve(genes_number_);
46         for (auto&& distribution : genes_distributions_) {
47             genes.push_back(distribution(engine_));
48         }
49         population_.push_back(
50             IChromosome::Create(std::move(genes), chromosome_subclass_));
51     }
52 }
53
54 std::shared_ptr<IChromosome> GeneticAlgorithm::Run() {
55     for (std::size_t i = 0; i < cfg_.populations_number; ++i) {
56         spdlog::info("Population {}/{}:", i, cfg_.populations_number);
57         for (std::size_t j = 0; j < cfg_.population_size; ++j) {
58             spdlog::info("Chromosome {}/{}:\n{}", j + 1, cfg_.population_size,
59                 population_[j]->ToString());
60         }
61
62         auto new_population = RouletteWheelSelection();
63         Crossover(new_population);
64         std::shuffle(new_population.begin(), new_population.end(), engine_);
65         Mutate(new_population);
66         std::shuffle(new_population.begin(), new_population.end(), engine_);
67
68         population_ = std::move(new_population);
69     }

```

```

70
71 spdlog::info("Population {}/{}:", cfg_.populations_number,
72             cfg_.populations_number);
73 for (std::size_t j = 0; j < cfg_.population_size; ++j) {
74     spdlog::info("Chromosome {}/{}:\n{}", j + 1, cfg_.population_size,
75                 population_[j]->ToString());
76 }
77
78 const auto fitness_values = CalculateFitnessValue();
79 const auto fittest_chromosome_index = std::distance(
80     fitness_values.cbegin(),
81     std::max_element(fitness_values.cbegin(), fitness_values.cend()));
82
83 spdlog::info("Chromosome {} (the fittest one):\n{}",
84             fittest_chromosome_index + 1,
85             population_[fittest_chromosome_index]->ToString());
86 return population_[fittest_chromosome_index];
87 }
88
89 std::vector<std::shared_ptr<IChromosome>>
90 GeneticAlgorithm::RouletteWheelSelection() {
91     auto fitness_values = CalculateFitnessValue();
92     for (std::size_t i = 0; i < cfg_.population_size; ++i) {
93         if (!std::isfinite(fitness_values[i])) {
94             fitness_values[i] = 0;
95         }
96     }
97
98     auto partial_sum = std::vector<double>(cfg_.population_size);
99     std::partial_sum(fitness_values.cbegin(), fitness_values.cend(),
100                     partial_sum.begin());
101
102     auto partial_sum_to_chromosome =
103         std::map<double, std::shared_ptr<IChromosome>>{};
104     for (std::size_t i = 0; i < cfg_.population_size; ++i) {
105         partial_sum_to_chromosome.insert({partial_sum[i], population_[i]});
106     }
107
108     auto selected_chromosomes = std::vector<std::shared_ptr<IChromosome>>{};
109     selected_chromosomes.reserve(cfg_.population_size);

```

```

110 auto distribution =
111     std::uniform_real_distribution<double>{0, partial_sum.back()};
112 for (std::size_t i = 0; i < cfg_.population_size; ++i) {
113     const auto value = distribution(engine_);
114     const auto it = partial_sum_to_chromosome.upper_bound(value);
115     assert(it != partial_sum_to_chromosome.end());
116     selected_chromosomes.push_back(it->second);
117 }
118
119 return selected_chromosomes;
120 }
121
122 void GeneticAlgorithm::Crossover(
123     std::vector<std::shared_ptr<IChromosome>>& population) {
124     static const auto parents_number = static_cast<std::size_t>(
125         cfg_.crossover_proportion * cfg_.population_size);
126     static auto distribution = std::uniform_real_distribution<>{0.0, 1.0};
127     for (std::size_t i = 0; i + 1 < parents_number; i += 2) {
128         const auto alpha = distribution(engine_);
129
130         const auto& parent1_genes = population[i]->get_genes();
131         const auto& parent2_genes = population[i + 1]->get_genes();
132
133         auto offspring1_genes = std::vector<double>{};
134         offspring1_genes.reserve(genes_number_);
135         for (std::size_t j = 0; j < genes_number_; ++j) {
136             offspring1_genes.push_back(alpha * parent1_genes[j] +
137                                     (1 - alpha) * parent2_genes[j]);
138         }
139
140         auto offspring2_genes = std::vector<double>{};
141         offspring2_genes.reserve(genes_number_);
142         for (std::size_t j = 0; j < genes_number_; ++j) {
143             offspring2_genes.push_back((1 - alpha) * parent1_genes[j] +
144                                     alpha * parent2_genes[j]);
145         }
146
147         population[i] =
148             IChromosome::Create(std::move(offspring1_genes), chromosome_subclass_);
149         population[i + 1] =

```



```

150     IChromosome::Create(std::move(offspring2_genes), chromosome_subclass_);
151 }
152 }
153
154 void GeneticAlgorithm::Mutate(
155     std::vector<std::shared_ptr<IChromosome>>& population) {
156     static const auto mutants_number =
157         static_cast<std::size_t>(cfg_.mutation_proportion * cfg_.population_size);
158     static auto distribution =
159         std::uniform_int_distribution<>{0, static_cast<int>(genes_number_) - 1};
160     for (std::size_t i = 0; i < mutants_number; ++i) {
161         const auto mutated_gene_index = distribution(engine_);
162
163         auto genes = population[i]->get_genes();
164         genes[mutated_gene_index] =
165             genes_distributions_[mutated_gene_index](engine_);
166         population[i] = IChromosome::Create(std::move(genes), chromosome_subclass_);
167     }
168 }
169
170 std::vector<double> GeneticAlgorithm::CalculateFitnessValue() const {
171     auto fitness_values = std::vector<double>{};
172     fitness_values.reserve(cfg_.population_size);
173     for (std::size_t i = 0; i < cfg_.population_size; ++i) {
174         fitness_values.push_back(fitness_function_->Assess(*population_[i]));
175         spdlog::info("Chromosome {}/{} fitness value: {}", i + 1,
176             cfg_.population_size, fitness_values.back());
177     }
178     return fitness_values;
179 }
180
181 } // namespace nn

```

Листинг 13: Файл main.cc

```

1  #include <matplotlib/matplotlib.h>
2  #include <spdlog/common.h>
3  #include <spdlog/spdlog.h>
4
5  #include <memory>

```

```

6
7 #include "chromosome.h"
8 #include "data_supplier.h"
9 #include "fitness_function.h"
10 #include "genetic_algorithm.h"
11
12 namespace {
13
14 const std::string kDefaultTestPath = "../datasets/MNIST_CSV/test.csv";
15 const std::string kDefaultTrainPath = "../datasets/MNIST_CSV/train.csv";
16
17 void RunGeneticAlgorithmSgd() {
18     /*
19      * Train cost: 0.0964943;
20      * Train accuracy: 48532/50000;
21      * Test cost: 0.14492;
22      * Test accuracy: 9580/10000;
23      *
24      * Learning rate: 0.0959074;
25      * Epochs: 100;
26      * Mini-batch size: 100;
27      * Hidden layers: 1;
28      * Neurons per hidden layer: 28
29      */
30
31     auto data_supplier = std::make_unique<nn::DataSupplier>(
32         kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
33     auto fitness_function =
34         std::make_unique<nn::SgdFitness>(std::move(data_supplier));
35     const auto segments = std::vector<nn::Segment>{
36         {0.001, 1}, // kLearningRate
37         {100, 100}, // kEpochs
38         {100, 100}, // kMiniBatchSize
39         {0, 4}, // kHiddenLayer
40         {10, 40}, // kNeuronsPerHiddenLayer
41     };
42     const auto cfg = nn::GeneticAlgorithm::Configuration{
43         .populations_number = 10,
44         .population_size = 60,
45         .crossover_proportion = 0.4,

```

```

46     .mutation_proportion = 0.15,
47 };
48 auto genetic_algorithm = nn::GeneticAlgorithm(
49     std::move(fitness_function),
50     nn::ChromosomeSubclass::kSgdHyperparametersKit, segments, cfg);
51 genetic_algorithm.Run();
52 }
53
54 void RunGeneticAlgorithmSgdNag() {
55     /*
56     * Train cost: 0.0957484;
57     * Train accuracy: 48562/50000;
58     * Test cost: 0.146654;
59     * Test accuracy: 9565/10000;
60     *
61     * Learning rate: 0.0100863;
62     * Epochs: 100;
63     * Mini-batch size: 100;
64     * Hidden layers: 1;
65     * Neurons per hidden layer: 28
66     */
67
68     auto data_supplier = std::make_unique<nn::DataSupplier>(
69         kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
70     auto fitness_function =
71         std::make_unique<nn::SgdNagFitness>(std::move(data_supplier));
72     const auto segments = std::vector<nn::Segment>{
73         {0.001, 1}, // kLearningRate
74         {100, 100}, // kEpochs
75         {100, 100}, // kMiniBatchSize
76         {0, 4},     // kHiddenLayer
77         {10, 40},   // kNeuronsPerHiddenLayer
78     };
79     const auto cfg = nn::GeneticAlgorithm::Configuration{
80         .populations_number = 10,
81         .population_size = 60,
82         .crossover_proportion = 0.4,
83         .mutation_proportion = 0.15,
84     };
85     auto genetic_algorithm = nn::GeneticAlgorithm(

```

```

86     std::move(fitness_function),
87     nn::ChromosomeSubclass::kSgdHyperparametersKit, segments, cfg);
88     genetic_algorithm.Run();
89 }
90
91 void RunGeneticAlgorithmSgdAdagrad() {
92     /*
93      * Train cost: 0.181089;
94      * Train accuracy: 47361/50000;
95      * Test cost: 0.214604;
96      * Test accuracy: 9395/10000;
97      *
98      * Learning rate: 0.936544;
99      * Epochs: 100;
100     * Mini-batch size: 100;
101     * Hidden layers: 1;
102     * Neurons per hidden layer: 36
103     */
104
105     auto data_supplier = std::make_unique<nn::DataSupplier>(
106         kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
107     auto fitness_function =
108         std::make_unique<nn::SgdAdagradFitness>(std::move(data_supplier));
109     const auto segments = std::vector<nn::Segment>{
110         {0.001, 1}, // kLearningRate
111         {100, 100}, // kEpochs
112         {100, 100}, // kMiniBatchSize
113         {0, 4},     // kHiddenLayer
114         {10, 40},   // kNeuronsPerHiddenLayer
115     };
116     const auto cfg = nn::GeneticAlgorithm::Configuration{
117         .populations_number = 10,
118         .population_size = 60,
119         .crossover_proportion = 0.4,
120         .mutation_proportion = 0.15,
121     };
122     auto genetic_algorithm = nn::GeneticAlgorithm(
123         std::move(fitness_function),
124         nn::ChromosomeSubclass::kSgdHyperparametersKit, segments, cfg);
125     genetic_algorithm.Run();

```

```

126 }
127
128 void RunGeneticAlgorithmSgdAdam() {
129     /*
130      * Train cost: 0.0886979;
131      * Train accuracy: 48692/50000;
132      * Test cost: 0.143935;
133      * Test accuracy: 9620/10000;
134      *
135      * Learning rate: 0.0462101;
136      * Epochs: 100;
137      * Mini-batch size: 100;
138      * Hidden layers: 3;
139      * Neurons per hidden layer: 35
140      */
141
142     auto data_supplier = std::make_unique<nn::DataSupplier>(
143         kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
144     auto fitness_function =
145         std::make_unique<nn::SgdAdamFitness>(std::move(data_supplier));
146     const auto segments = std::vector<nn::Segment>{
147         {0.001, 1}, // kLearningRate
148         {100, 100}, // kEpochs
149         {100, 100}, // kMiniBatchSize
150         {0, 4},     // kHiddenLayer
151         {10, 40},   // kNeuronsPerHiddenLayer
152     };
153     const auto cfg = nn::GeneticAlgorithm::Configuration{
154         .populations_number = 10,
155         .population_size = 60,
156         .crossover_proportion = 0.4,
157         .mutation_proportion = 0.15,
158     };
159     auto genetic_algorithm = nn::GeneticAlgorithm(
160         std::move(fitness_function),
161         nn::ChromosomeSubclass::kSgdHyperparametersKit, segments, cfg);
162     genetic_algorithm.Run();
163 }
164
165 } // namespace

```

```

166
167 int main() {
168     RunGeneticAlgorithmSgd();
169     RunGeneticAlgorithmSgdNag();
170     RunGeneticAlgorithmSgdAdagrad();
171     RunGeneticAlgorithmSgdAdam();
172 }

```

3 Результаты сравнения

Оптимальные гиперпараметры для обучения многослойного персептрона с различными оптимизаторами определялись с помощью генетического алгоритма. Фиксированными были количество эпох обучения (100) и размер пакета данных (100). Коэффициент обучения определялся в промежутке $[0.001, 1]$, количество скрытых слоёв — от 0 до 4, количество нейронов в скрытых слоях — от 10 до 40. Параметры генетического алгоритма:

- количество популяций: 10;
- размер популяции: 60;
- доля особей, подвергающихся скрещиванию: 40%;
- доля особей, подвергающихся мутации: 15%.

Функции активации скрытых слоёв персептрона — Leaky ReLU, выходного слоя — Softmax. Функция стоимости — перекрёстная энтропия.

3.1 SGD

Для SGD-оптимизатора определены следующие оптимальные гиперпараметры:

- коэффициент обучения: 0.0959074;
- количество скрытых слоёв: 1;

- количество нейронов в скрытых слоях: 28.

Точность работы пересептрона с указанными гиперпараметрами на тренировочных данных составляет 97.06%, на тестовых данных — 95.80%.

3.2 NAG

Для NAG-оптимизатора с коэффициентом $\gamma = 0.9$ определены следующие оптимальные гиперпараметры:

- коэффициент обучения: 0.0100863;
- количество скрытых слоёв: 1;
- количество нейронов в скрытых слоях: 28.

Точность работы пересептрона с указанными гиперпараметрами на тренировочных данных составляет 97.12%, на тестовых данных — 95.65%.

3.3 Adagrad

Для Adagrad-оптимизатора с $\varepsilon = 10^{-8}$ определены следующие оптимальные гиперпараметры:

- коэффициент обучения: 0.936544;
- количество скрытых слоёв: 1;
- количество нейронов в скрытых слоях: 36.

Точность работы пересептрона с указанными гиперпараметрами на тренировочных данных составляет 94.72%, на тестовых данных — 93.95%.

3.4 Adam

Для Adam-оптимизатора с $\varepsilon = 10^{-8}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ определены следующие оптимальные гиперпараметры:

- коэффициент обучения: 0.0462101;

- количество скрытых слоёв: 3;
- количество нейронов в скрытых слоях: 35.

Точность работы персептрона с указанными гиперпараметрами на тренировочных данных составляет 97.38%, на тестовых данных — 96.20%.

4 Вывод

Наилучший результат в 96.20% точности на тестовых данных был достигнут с оптимизатором Adam, который сочетает в себе идеи оптимизаторов NAG и Adagrad. Благодаря использованию генетического алгоритма удалось автоматически определить оптимальные гиперпараметры для обучения персептрона с различными оптимизаторами.