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

```
#pragma once

#include <Eigen/Dense>
#include <iostream>

namespace nn {

```

```
class IActivationFunction {
 public:
  virtual ~IActivationFunction() = default;

 public:
  virtual Eigen::VectorXd Apply(const Eigen::VectorXd &z) = 0;
  virtual Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) = 0;
};

class Linear final : public IActivationFunction {
 public:
  Eigen::VectorXd Apply(const Eigen::VectorXd &z) override { return z; }

  Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
    return Eigen::MatrixXd::Identity(z.rows(), z.cols());
  }
};

class ReLU final : public IActivationFunction {
 public:
  Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
    return z.array().max(0.0);
  }

  Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
    return z.array().cwiseTypedGreaterOrEqual(0.0).matrix().asDiagonal();
  }
};

class LeakyReLU final : public IActivationFunction {
  std::function<double(double)> f_, f_prime_;

 public:
  LeakyReLU(const double alpha)
      : f_([alpha](const double x) { return x >= 0 ? x : alpha * x; }),
        f_prime_([alpha](const double x) { return x >= 0 ? 1 : alpha; }) {}

  Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
    return z.unaryExpr(f_);
  }
```

```

  Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
    return z.unaryExpr(f_prime_).asDiagonal();
  }
};

class Sigmoid final : public IActivationFunction {
 public:
  Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
    return 1.0 / (1.0 + (-z).array().exp());
  }

  Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
    const auto sigmoid = Apply(z);
    return (sigmoid.array() * (1 - sigmoid.array())).matrix().asDiagonal();
  }
};

class Tanh final : public IActivationFunction {
 public:
  Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
    const auto e_z = z.array().exp();
    const auto e_neg_z = (-z).array().exp();
    return (e_z - e_neg_z) / (e_z + e_neg_z);
  }

  Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
    const auto tanh = Apply(z);
    return (1 - tanh.array().square()).matrix().asDiagonal();
  }
};

class Softmax final : public IActivationFunction {
 public:
  Eigen::VectorXd Apply(const Eigen::VectorXd &z) override {
    const auto e_z = z.array().exp();
    return e_z / e_z.sum();
  }

  Eigen::MatrixXd Jacobian(const Eigen::VectorXd &z) override {
```

```
    const auto softmax = Apply(z);
    return softmax.asDiagonal().toDenseMatrix() - softmax * softmax.transpose();
  }
};

}  // namespace nn
```

Листинг 2: Файл cost_function.h

```
#pragma once

#include <Eigen/Dense>

namespace nn {

class ICostFunction {
 public:
  virtual ~ICostFunction() = default;

 public:
  virtual double Apply(const Eigen::VectorXd &y, const Eigen::VectorXd &a) = 0;
  virtual Eigen::VectorXd GradientWrtActivations(const Eigen::VectorXd &y,
                                                  const Eigen::VectorXd &a) = 0;
};

class MSE final : public ICostFunction {
 public:
  double Apply(const Eigen::VectorXd &y, const Eigen::VectorXd &a) override {
    return 0.5 * (y - a).squaredNorm();
  }

  Eigen::VectorXd GradientWrtActivations(const Eigen::VectorXd &y,
                                         const Eigen::VectorXd &a) override {
    return a - y;
  }
};

class CrossEntropy final : public ICostFunction {
 public:
  double Apply(const Eigen::VectorXd &y, const Eigen::VectorXd &a) override {
```

```
    return -(y.array() * a.array().log()).sum();
  }

  Eigen::VectorXd GradientWrtActivations(const Eigen::VectorXd &y,
                                         const Eigen::VectorXd &a) override {
    return -y.array() / a.array();
  }
};

class KLDivergence final : public ICostFunction {
 public:
  double Apply(const Eigen::VectorXd &y, const Eigen::VectorXd &a) override {
    return (y.array() * (y.array() / a.array()).log()).sum();
  }

  Eigen::VectorXd GradientWrtActivations(const Eigen::VectorXd &y,
                                         const Eigen::VectorXd &a) override {
    return -y.array() / a.array();
  }
};

}  // namespace nn
```

Листинг 3: Файл data_supplier.h

```
#pragma once

#include <Eigen/Dense>
#include <memory>
#include <vector>

#include "perceptron.h"

namespace nn {

struct Data final : nn::IData {
  Eigen::VectorXd x, y;
  std::string label;

  const Eigen::VectorXd &GetX() const override { return x; }
```

```
  const Eigen::VectorXd &GetY() const override { return y; }
  std::string_view ToString() const override { return label; }
};

class DataSupplier final : public nn::IDataSupplier {
  std::vector<std::shared_ptr<const nn::IData>> train_, test_, validation_;

 public:
  DataSupplier(const std::string &train_path, const std::string &test_path,
               const double false_score, const double true_score);

  std::size_t GetInputLayerSize() const override;
  std::size_t GetOutputLayerSize() const override;

  std::vector<std::shared_ptr<const nn::IData>> GetTrainData() const override;
  std::vector<std::shared_ptr<const nn::IData>> GetValidationData()
      const override;
  std::vector<std::shared_ptr<const nn::IData>> GetTestData() const override;
};

}  // namespace nn
```

Листинг 4: Файл data_supplier.cc

```
#include "data_supplier.h"

#include <spdlog/spdlog.h>

#include <boost/algorithm/string/classification.hpp>
#include <boost/algorithm/string/split.hpp>
#include <cassert>
#include <fstream>
#include <iterator>
#include <stdexcept>
#include <string>

#include "perceptron.h"

namespace nn {

```

```
namespace {

constexpr std::size_t kDigitsNumber = 10;
constexpr std::size_t kScanSize = 784;
constexpr std::size_t kColumnsCount = kScanSize + 1;

std::vector<std::shared_ptr<const nn::IData>> ReadMnistCsv(
    const std::string &filename, const double false_score,
    const double true_score) {
  static constexpr std::size_t kShadesCount = 255;

  auto file = std::ifstream(filename);
  if (!file.is_open()) {
    throw std::runtime_error("Failed to open MNIST CSV file " + filename);
  }

  auto instances = std::vector<std::shared_ptr<const nn::IData>>{};
  auto line = std::string{};
  while (std::getline(file, line)) {
    auto result = std::vector<std::string>{};
    result.reserve(kColumnsCount);
    boost::split(result, line, boost::is_any_of(","));

    assert(result[0].size() == 1);
    assert('0' <= result[0][0] && result[0][0] <= '9');

    auto data = Data{};
    data.label = result[0];

    data.y = Eigen::VectorXd(kDigitsNumber);
    data.y.setConstant(false_score);
    data.y(data.label[0] - '0') = true_score;

    data.x = Eigen::VectorXd(kScanSize);
    for (std::size_t i = 1; i < kColumnsCount; ++i) {
      data.x[i - 1] = std::stod(result[i]) / kShadesCount;
    }

    instances.push_back(std::make_shared<const Data>(std::move(data)));
  }
```

```
  return instances;
}

}  // namespace

DataSupplier::DataSupplier(const std::string &train_path,
                           const std::string &test_path,
                           const double false_score, const double true_score) {
  static constexpr std::size_t kTrainInitialSize = 60'000;
  static constexpr std::size_t kValidationSize = 10'000;
  static constexpr std::size_t kTestSize = 10'000;

  spdlog::info("Parsing train data...");
  train_ = ReadMnistCsv(train_path, false_score, true_score);
  assert(train_.size() == kTrainInitialSize);

  validation_ =
      std::vector(std::make_move_iterator(train_.rbegin()),
                  std::make_move_iterator(train_.rbegin() + kValidationSize));
  train_.resize(kTrainInitialSize - kValidationSize);

  spdlog::info("Parsing test data...");
  test_ = ReadMnistCsv(test_path, false_score, true_score);
  assert(test_.size() == kTestSize);
}

std::size_t DataSupplier::GetInputLayerSize() const { return kScanSize; }
std::size_t DataSupplier::GetOutputLayerSize() const { return kDigitsNumber; }

std::vector<std::shared_ptr<const nn::IData>> DataSupplier::GetTrainData()
    const {
  return train_;
}

std::vector<std::shared_ptr<const nn::IData>> DataSupplier::GetValidationData()
    const {
  return validation_;
}
```

```
std::vector<std::shared_ptr<const nn::IData>> DataSupplier::GetTestData()
    const {
  return test_;
}

}  // namespace nn
```

Листинг 5: Файл perceptron.h

```
#pragma once

#include <Eigen/Dense>
#include <memory>
#include <random>

#include "activation_function.h"
#include "cost_function.h"

namespace nn {

class IData {
 public:
  virtual ~IData() = default;

 public:
  virtual const Eigen::VectorXd &GetX() const = 0;
  virtual const Eigen::VectorXd &GetY() const = 0;
  virtual std::string_view ToString() const = 0;
};

class IDataSupplier {
 public:
  virtual ~IDataSupplier() = default;

 public:
  virtual std::size_t GetInputLayerSize() const = 0;
  virtual std::size_t GetOutputLayerSize() const = 0;

  virtual std::vector<std::shared_ptr<const IData>> GetTrainData() const = 0;
  virtual std::vector<std::shared_ptr<const IData>> GetTestData() const = 0;
```

```
  virtual std::vector<std::shared_ptr<const IData>> GetValidationData()
      const = 0;
};

struct SgdConfiguration final {
  std::size_t epochs;
  std::size_t mini_batch_size;
  double learning_rate;
  bool monitor_train_cost;
  bool monitor_train_accuracy;
  bool monitor_test_cost;
  bool monitor_test_accuracy;
};

struct Metric final {
  std::vector<double> train_cost, train_accuracy;
  std::vector<double> test_cost, test_accuracy;
};

class Perceptron final {
  std::random_device device_;
  std::default_random_engine generator_;
  std::unique_ptr<ICostFunction> cost_function_;
  std::size_t layers_number_, connections_number_;
  std::vector<Eigen::MatrixXd> weights_;
  std::vector<Eigen::VectorXd> biases_;
  std::vector<std::unique_ptr<IActivationFunction>> activation_functions_;

 public:
  Perceptron(
      std::unique_ptr<ICostFunction> &&cost_function,
      std::vector<std::unique_ptr<IActivationFunction>> &&activation_functions,
      const std::vector<std::size_t> &layers_sizes);

  Eigen::VectorXd Feedforward(const Eigen::VectorXd &x) const;

  Metric Sgd(const std::vector<std::shared_ptr<const IData>> &train,
             const std::vector<std::shared_ptr<const IData>> &test,
             const SgdConfiguration &cfg);

```

```
  Metric SgdNag(const std::vector<std::shared_ptr<const IData>> &train,
                const std::vector<std::shared_ptr<const IData>> &test,
                const SgdConfiguration &cfg, const double gamma);

  Metric SgdAdagrad(const std::vector<std::shared_ptr<const IData>> &train,
                    const std::vector<std::shared_ptr<const IData>> &test,
                    const SgdConfiguration &cfg, const double epsilon);

  Metric SgdAdam(const std::vector<std::shared_ptr<const IData>> &train,
                 const std::vector<std::shared_ptr<const IData>> &test,
                 const SgdConfiguration &cfg, const double beta1,
                 const double beta2, const double epsilon);

private:
  // TODO: Use concepts
  template <typename Iter>
  void UpdateSgd(const Iter mini_batch_begin, const Iter mini_batch_end,
                 const std::size_t mini_batch_size, const double learning_rate);

  // EMA means Exponential Moving Average
  template <typename Iter>
  void UpdateSgdNag(std::vector<Eigen::MatrixXd> &delta_weights_ema,
                    std::vector<Eigen::VectorXd> &delta_biases_ema,
                    const Iter mini_batch_begin, const Iter mini_batch_end,
                    const std::size_t mini_batch_size,
                    const double learning_rate, const double gamma);

  template <typename Iter>
  void UpdateSgdAdagrad(
      std::vector<Eigen::MatrixXd> &weights_gradient_squares_sum,
      std::vector<Eigen::VectorXd> &biases_gradient_squares_sum,
      const Iter mini_batch_begin, const Iter mini_batch_end,
      const std::size_t mini_batch_size, const double learning_rate);

  template <typename Iter>
  void UpdateSgdAdam(std::vector<Eigen::MatrixXd> &weights_gradient_ema,
                     std::vector<Eigen::VectorXd> &biases_gradient_ema,
                     std::vector<Eigen::MatrixXd> &weights_squared_gradient_ema,
                     std::vector<Eigen::VectorXd> &biases_squared_gradient_ema,
                     const Iter mini_batch_begin, const Iter mini_batch_end,
```

```
                     const std::size_t mini_batch_size, const std::size_t epoch,
                     const double learning_rate, const double beta1,
                     const double beta2);

 private:
  struct Parameters {
    std::vector<Eigen::MatrixXd> weights;
    std::vector<Eigen::VectorXd> biases;
  };

  Parameters CreateParameters(const double initial_value) const;

  template <typename Iter>
  Parameters GradientWrtParameters(const Iter mini_batch_begin,
                                   const Iter mini_batch_end,
                                   const std::size_t mini_batch_size) const;

  Parameters Backpropagation(const Eigen::VectorXd &x,
                             const Eigen::VectorXd &y) const;

  std::pair<std::vector<Eigen::VectorXd>, std::vector<Eigen::VectorXd>>
  FeedforwardDetailed(const Eigen::VectorXd &x) const;

  Metric CreateMetric(const SgdConfiguration &cfg) const;

  void WriteMetric(Metric &metric, const std::size_t epoch,
                   const std::vector<std::shared_ptr<const IData>> &train,
                   const std::vector<std::shared_ptr<const IData>> &test,
                   const SgdConfiguration &cfg) const;

  template <typename Iter>
  std::size_t CalculateAccuracy(const Iter begin, const Iter end) const;

  template <typename Iter>
  double CalculateCost(const Iter begin, const Iter end) const;
};

}  // namespace nn
```

Листинг 6: Файл perceptron.cc

```cpp
#include "perceptron.h"

#include <spdlog/spdlog.h>

#include <cassert>
#include <cmath>
#include <iostream>
#include <iterator>
#include <sstream>

namespace nn {

Perceptron::Perceptron(
    std::unique_ptr<ICostFunction> &&cost_function,
    std::vector<std::unique_ptr<IActivationFunction>> &&activation_functions,
    const std::vector<std::size_t> &layers_sizes)
    : generator_(device_()),
      cost_function_(std::move(cost_function)),
      layers_number_(layers_sizes.size()),
      connections_number_(layers_number_ - 1),
      activation_functions_(std::move(activation_functions)) {
  if (layers_number_ < 2) {
    throw std::runtime_error("Perceptron must have at least two layers");
  }
  if (activation_functions_.size() != connections_number_) {
    throw std::runtime_error(
        "Activation functions number must be equal to layers number minus one");
  }

  weights_.reserve(connections_number_);
  biases_.reserve(connections_number_);
  for (std::size_t i = 0; i < connections_number_; ++i) {
    weights_.push_back(
        Eigen::MatrixXd::Random(layers_sizes[i + 1], layers_sizes[i]));
    biases_.push_back(Eigen::VectorXd::Random(layers_sizes[i + 1]));
  }
}

Eigen::VectorXd Perceptron::Feedforward(const Eigen::VectorXd &x) const {
```

```
  auto activation = x;
  for (std::size_t i = 0; i < connections_number_; ++i) {
    activation =
        activation_functions_[i]->Apply(weights_[i] * activation + biases_[i]);
  }
  return activation;
}

Metric Perceptron::Sgd(const std::vector<std::shared_ptr<const IData>> &train,
                       const std::vector<std::shared_ptr<const IData>> &test,
                       const SgdConfiguration &cfg) {
  const auto train_size = train.size();
  const auto whole_mini_batches_number = train_size / cfg.mini_batch_size;
  const auto remainder_mini_batch_size = train_size % cfg.mini_batch_size;

  auto train_shuffled = std::vector(train.begin(), train.end());
  auto metric = CreateMetric(cfg);
  for (std::size_t i = 1; i <= cfg.epochs; ++i) {
    std::shuffle(train_shuffled.begin(), train_shuffled.end(), generator_);
    auto it = train_shuffled.begin();
    for (std::size_t j = 0; j < whole_mini_batches_number; ++j) {
      auto end = it + cfg.mini_batch_size;
      UpdateSgd(it, end, cfg.mini_batch_size, cfg.learning_rate);
      it = std::move(end);
    }

    if (remainder_mini_batch_size != 0) {
      UpdateSgd(it, it + remainder_mini_batch_size, remainder_mini_batch_size,
                cfg.learning_rate);
    }
    WriteMetric(metric, i, train, test, cfg);
  }

  return metric;
}

Metric Perceptron::SgdNag(
    const std::vector<std::shared_ptr<const IData>> &train,
    const std::vector<std::shared_ptr<const IData>> &test,
    const SgdConfiguration &cfg, const double gamma) {
```

```
  if (gamma < 0 || gamma > 1) {
    throw std::runtime_error("Gamma must belong to [0, 1]");
  }

  const auto train_size = train.size();
  const auto whole_mini_batches_number = train_size / cfg.mini_batch_size;
  const auto remainder_mini_batch_size = train_size % cfg.mini_batch_size;

  auto [delta_weights_ema, delta_biases_ema] = CreateParameters(0);
  auto train_shuffled = std::vector(train.begin(), train.end());
  auto metric = CreateMetric(cfg);
  for (std::size_t i = 1; i <= cfg.epochs; ++i) {
    std::shuffle(train_shuffled.begin(), train_shuffled.end(), generator_);
    auto it = train_shuffled.begin();
    for (std::size_t j = 0; j < whole_mini_batches_number; ++j) {
      auto end = it + cfg.mini_batch_size;
      UpdateSgdNag(delta_weights_ema, delta_biases_ema, it, end,
                   cfg.mini_batch_size, cfg.learning_rate, gamma);
      it = std::move(end);
    }

    if (remainder_mini_batch_size != 0) {
      UpdateSgdNag(delta_weights_ema, delta_biases_ema, it,
                   it + remainder_mini_batch_size, remainder_mini_batch_size,
                   cfg.learning_rate, gamma);
    }
    WriteMetric(metric, i, train, test, cfg);
  }

  return metric;
}

Metric Perceptron::SgdAdagrad(
    const std::vector<std::shared_ptr<const IData>> &train,
    const std::vector<std::shared_ptr<const IData>> &test,
    const SgdConfiguration &cfg, const double epsilon) {
  if (epsilon <= 0) {
    throw std::runtime_error("Epsilon must be strictly greater than 0");
  }

```

```
  const auto train_size = train.size();
  const auto whole_mini_batches_number = train_size / cfg.mini_batch_size;
  const auto remainder_mini_batch_size = train_size % cfg.mini_batch_size;

  auto [weights_gradient_squares_sum, biases_gradient_squares_sum] =
      CreateParameters(epsilon);
  auto train_shuffled = std::vector(train.begin(), train.end());
  auto metric = CreateMetric(cfg);
  for (std::size_t i = 1; i <= cfg.epochs; ++i) {
    std::shuffle(train_shuffled.begin(), train_shuffled.end(), generator_);
    auto it = train_shuffled.begin();
    for (std::size_t i = 0; i < whole_mini_batches_number; ++i) {
      auto end = it + cfg.mini_batch_size;
      UpdateSgdAdagrad(weights_gradient_squares_sum,
                       biases_gradient_squares_sum, it, end,
                       cfg.mini_batch_size, cfg.learning_rate);
      it = std::move(end);
    }

    if (remainder_mini_batch_size != 0) {
      UpdateSgdAdagrad(weights_gradient_squares_sum,
                       biases_gradient_squares_sum, it,
                       it + remainder_mini_batch_size,
                       remainder_mini_batch_size, cfg.learning_rate);
    }
    WriteMetric(metric, i, train, test, cfg);
  }

  return metric;
}

Metric Perceptron::SgdAdam(
    const std::vector<std::shared_ptr<const IData>> &train,
    const std::vector<std::shared_ptr<const IData>> &test,
    const SgdConfiguration &cfg, const double beta1, const double beta2,
    const double epsilon) {
  if (beta1 < 0 || beta1 > 1) {
    throw std::runtime_error("Beta1 must belong to [0, 1]");
  }
  if (beta2 < 0 || beta2 > 1) {
```

```
    throw std::runtime_error("Beta2 must belong to [0, 1]");
  }
  if (epsilon <= 0) {
    throw std::runtime_error("Epsilon must be strictly greater than 0");
  }

  const auto train_size = train.size();
  const auto whole_mini_batches_number = train_size / cfg.mini_batch_size;
  const auto remainder_mini_batch_size = train_size % cfg.mini_batch_size;

  auto [weights_gradient_ema, biases_gradient_ema] = CreateParameters(0);
  auto [weights_squared_gradient_ema, biases_squared_gradient_ema] =
      CreateParameters(epsilon);
  auto train_shuffled = std::vector(train.begin(), train.end());
  auto metric = CreateMetric(cfg);
  for (std::size_t i = 1; i <= cfg.epochs; ++i) {
    std::shuffle(train_shuffled.begin(), train_shuffled.end(), generator_);
    auto it = train_shuffled.begin();
    for (std::size_t j = 0; j < whole_mini_batches_number; ++j) {
      auto end = it + cfg.mini_batch_size;
      UpdateSgdAdam(weights_gradient_ema, biases_gradient_ema,
                    weights_squared_gradient_ema, biases_squared_gradient_ema,
                    it, end, cfg.mini_batch_size, i, cfg.learning_rate, beta1,
                    beta2);
      it = std::move(end);
    }

    if (remainder_mini_batch_size != 0) {
      UpdateSgdAdam(weights_gradient_ema, biases_gradient_ema,
                    weights_squared_gradient_ema, biases_squared_gradient_ema,
                    it, it + remainder_mini_batch_size,
                    remainder_mini_batch_size, i, cfg.learning_rate, beta1,
                    beta2);
    }
    WriteMetric(metric, i, train, test, cfg);
  }

  return metric;
}
```

```cpp
template <typename Iter>
void Perceptron::UpdateSgd(const Iter mini_batch_begin,
                           const Iter mini_batch_end,
                           const std::size_t mini_batch_size,
                           const double learning_rate) {
  const auto [weights_gradient, biases_gradient] =
      GradientWrtParameters(mini_batch_begin, mini_batch_end, mini_batch_size);

  for (std::size_t i = 0; i < connections_number_; ++i) {
    weights_[i] -= learning_rate * weights_gradient[i];
    biases_[i] -= learning_rate * biases_gradient[i];
  }
}

template <typename Iter>
void Perceptron::UpdateSgdNag(std::vector<Eigen::MatrixXd> &delta_weights_ema,
                              std::vector<Eigen::VectorXd> &delta_biases_ema,
                              const Iter mini_batch_begin,
                              const Iter mini_batch_end,
                              const std::size_t mini_batch_size,
                              const double learning_rate, const double gamma) {
  for (std::size_t i = 0; i < connections_number_; ++i) {
    weights_[i] -= gamma * delta_weights_ema[i];
    biases_[i] -= gamma * delta_biases_ema[i];
  }

  auto [weights_gradient, biases_gradient] =
      GradientWrtParameters(mini_batch_begin, mini_batch_end, mini_batch_size);

  for (std::size_t i = 0; i < connections_number_; ++i) {
    const auto saved_delta_weights_ema = gamma * delta_weights_ema[i];
    delta_weights_ema[i] =
        saved_delta_weights_ema + learning_rate * weights_gradient[i];
    weights_[i] += saved_delta_weights_ema - delta_weights_ema[i];

    const auto saved_delta_biases_ema = gamma * delta_biases_ema[i];
    delta_biases_ema[i] =
        saved_delta_biases_ema + learning_rate * biases_gradient[i];
    biases_[i] += saved_delta_biases_ema - delta_biases_ema[i];
  }
```

```
}

template <typename Iter>
void Perceptron::UpdateSgdAdagrad(
    std::vector<Eigen::MatrixXd> &weights_gradient_squares_sum,
    std::vector<Eigen::VectorXd> &biases_gradient_squares_sum,
    const Iter mini_batch_begin, const Iter mini_batch_end,
    const std::size_t mini_batch_size, const double learning_rate) {
  auto [weights_gradient, biases_gradient] =
      GradientWrtParameters(mini_batch_begin, mini_batch_end, mini_batch_size);

  for (std::size_t i = 0; i < connections_number_; ++i) {
    weights_gradient_squares_sum[i] +=
        weights_gradient[i].array().pow(2).matrix();
    weights_[i] -= learning_rate / weights_gradient_squares_sum[i].lpNorm<2>() *
                   weights_gradient[i];

    biases_gradient_squares_sum[i] +=
        biases_gradient[i].array().pow(2).matrix();
    biases_[i] -= learning_rate / biases_gradient_squares_sum[i].lpNorm<2>() *
                  biases_gradient[i];
  }
}

template <typename Iter>
void Perceptron::UpdateSgdAdam(
    std::vector<Eigen::MatrixXd> &weights_gradient_ema,
    std::vector<Eigen::VectorXd> &biases_gradient_ema,
    std::vector<Eigen::MatrixXd> &weights_squared_gradient_ema,
    std::vector<Eigen::VectorXd> &biases_squared_gradient_ema,
    const Iter mini_batch_begin, const Iter mini_batch_end,
    const std::size_t mini_batch_size, const std::size_t epoch,
    const double learning_rate, const double beta1, const double beta2) {
  auto [weights_gradient, biases_gradient] =
      GradientWrtParameters(mini_batch_begin, mini_batch_end, mini_batch_size);

  for (std::size_t i = 0; i < connections_number_; ++i) {
    weights_gradient_ema[i] =
        beta1 * weights_gradient_ema[i] + (1 - beta1) * weights_gradient[i];
    biases_gradient_ema[i] =
```

```
        beta1 * biases_gradient_ema[i] + (1 - beta1) * biases_gradient[i];

    const auto adjusted_weights_gradient_ema =
        weights_gradient_ema[i] / (1 - std::pow(beta1, epoch));
    const auto adjusted_biases_gradient_ema =
        biases_gradient_ema[i] / (1 - std::pow(beta1, epoch));

    weights_squared_gradient_ema[i] =
        beta2 * weights_squared_gradient_ema[i] +
        (1 - beta2) * weights_gradient[i].array().pow(2).matrix();
    biases_squared_gradient_ema[i] =
        beta2 * biases_squared_gradient_ema[i] +
        (1 - beta2) * biases_gradient[i].array().pow(2).matrix();

    const auto adjusted_weights_squared_gradient_ema =
        weights_squared_gradient_ema[i] / (1 - std::pow(beta2, epoch));
    const auto adjusted_biases_squared_gradient_ema =
        biases_squared_gradient_ema[i] / (1 - std::pow(beta2, epoch));

    weights_[i] -= learning_rate /
                   adjusted_weights_squared_gradient_ema.lpNorm<2>() *
                   adjusted_weights_gradient_ema;
    biases_[i] -= learning_rate /
                  adjusted_biases_squared_gradient_ema.lpNorm<2>() *
                  adjusted_biases_gradient_ema;
  }
}

Perceptron::Parameters Perceptron::CreateParameters(
    const double initial_value) const {
  auto weights = std::vector<Eigen::MatrixXd>{};
  weights.reserve(weights_.size());
  for (auto &&w : weights_) {
    auto m = Eigen::MatrixXd(w.rows(), w.cols());
    m.setConstant(initial_value);
    weights.push_back(std::move(m));
  }

  auto biases = std::vector<Eigen::VectorXd>{};
  biases.reserve(biases_.size());
```

```
  for (auto &&b : biases_) {
    auto v = Eigen::VectorXd(b.size());
    v.setConstant(initial_value);
    biases.push_back(std::move(v));
  }

  return {std::move(weights), std::move(biases)};
}

template <typename Iter>
Perceptron::Parameters Perceptron::GradientWrtParameters(
    const Iter mini_batch_begin, const Iter mini_batch_end,
    const std::size_t mini_batch_size) const {
  auto [weights_gradient, biases_gradient] = CreateParameters(0);

  for (auto it = mini_batch_begin; it != mini_batch_end; ++it) {
    const auto &data = **it;
    const auto [weights_gradient_contribution, biases_gradient_contribution] =
        Backpropagation(data.GetX(), data.GetY());
    for (std::size_t i = 0; i < connections_number_; ++i) {
      weights_gradient[i] += weights_gradient_contribution[i];
      biases_gradient[i] += biases_gradient_contribution[i];
    }
  }

  const auto factor = 1.0 / mini_batch_size;
  for (std::size_t i = 0; i < connections_number_; ++i) {
    weights_gradient[i] *= factor;
    biases_gradient[i] *= factor;
  }

  return {std::move(weights_gradient), std::move(biases_gradient)};
}

Perceptron::Parameters Perceptron::Backpropagation(
    const Eigen::VectorXd &x, const Eigen::VectorXd &y) const {
  const auto [linear_values, activations] = FeedforwardDetailed(x);
  assert(linear_values.size() == connections_number_);
  assert(activations.size() == layers_number_);

```

```
  auto delta = static_cast<Eigen::VectorXd>(
      activation_functions_.back()->Jacobian(linear_values.back()).transpose() *
      cost_function_->GradientWrtActivations(y, activations.back()));

  auto nabla_weights_reversed = std::vector<Eigen::MatrixXd>{};
  nabla_weights_reversed.reserve(connections_number_);
  nabla_weights_reversed.push_back(
      delta * std::prev(activations.cend(), 2)->transpose());

  auto nabla_biases_reversed = std::vector<Eigen::VectorXd>{};
  nabla_biases_reversed.reserve(connections_number_);
  nabla_biases_reversed.push_back(delta);

  for (int i = connections_number_ - 2; i >= 0; --i) {
    delta =
        (weights_[i + 1] * activation_functions_[i]->Jacobian(linear_values[i]))
            .transpose() *
        delta;
    nabla_weights_reversed.push_back(delta * activations[i].transpose());
    nabla_biases_reversed.push_back(delta);
  }

  return {{std::make_move_iterator(nabla_weights_reversed.rbegin()),
           std::make_move_iterator(nabla_weights_reversed.rend())},
          {std::make_move_iterator(nabla_biases_reversed.rbegin()),
           std::make_move_iterator(nabla_biases_reversed.rend())}};
}

std::pair<std::vector<Eigen::VectorXd>, std::vector<Eigen::VectorXd>>
Perceptron::FeedforwardDetailed(const Eigen::VectorXd &x) const {
  std::vector<Eigen::VectorXd> linear_values, activations;
  linear_values.reserve(connections_number_);
  activations.reserve(layers_number_);

  auto activation = x;
  for (std::size_t i = 0; i < connections_number_; ++i) {
    auto linear_value =
        static_cast<Eigen::VectorXd>(weights_[i] * activation + biases_[i]);
    activations.push_back(std::move(activation));
    activation = activation_functions_[i]->Apply(linear_value);
```

```
    linear_values.push_back(std::move(linear_value));
  }
  activations.push_back(std::move(activation));

  return {std::move(linear_values), std::move(activations)};
}

Metric Perceptron::CreateMetric(const SgdConfiguration &cfg) const {
  auto metric = Metric{};
  if (cfg.monitor_train_cost) {
    metric.train_cost.reserve(cfg.epochs);
  }
  if (cfg.monitor_train_accuracy) {
    metric.train_accuracy.reserve(cfg.epochs);
  }
  if (cfg.monitor_test_cost) {
    metric.test_cost.reserve(cfg.epochs);
  }
  if (cfg.monitor_test_accuracy) {
    metric.test_accuracy.reserve(cfg.epochs);
  }
  return metric;
}

void Perceptron::WriteMetric(
    Metric &metric, const std::size_t epoch,
    const std::vector<std::shared_ptr<const IData>> &train,
    const std::vector<std::shared_ptr<const IData>> &test,
    const SgdConfiguration &cfg) const {
  std::stringstream oss;
  oss << "Epoch " << epoch << "/" << cfg.epochs << ";";
  if (cfg.monitor_train_cost) {
    const auto train_cost = CalculateCost(train.begin(), train.end());
    metric.train_cost.push_back(train_cost);
    oss << " train cost: " << train_cost << ";";
  }
  if (cfg.monitor_train_accuracy) {
    const auto train_accuracy = CalculateAccuracy(train.begin(), train.end());
    metric.train_accuracy.push_back(train_accuracy);
    oss << " train accuracy: " << train_accuracy << "/" << train.size() << ";";
```

```
  }
  if (cfg.monitor_test_cost) {
    const auto test_cost = CalculateCost(test.begin(), test.end());
    metric.test_cost.push_back(CalculateCost(test.begin(), test.end()));
    oss << " test cost: " << test_cost << ";";
  }
  if (cfg.monitor_test_accuracy) {
    const auto test_accuracy = CalculateAccuracy(test.begin(), test.end());
    metric.test_accuracy.push_back(test_accuracy);
    oss << " test accuracy: " << test_accuracy << "/" << test.size() << ";";
  }
  spdlog::info(oss.str());
}

template <typename Iter>
std::size_t Perceptron::CalculateAccuracy(const Iter begin,
                                          const Iter end) const {
  std::size_t right_predictions = 0;
  for (auto it = begin; it != end; ++it) {
    const IData &instance = **it;
    Eigen::Index max_activation_expected, max_activation_actual;
    instance.GetY().maxCoeff(&max_activation_expected);
    Feedforward(instance.GetX()).maxCoeff(&max_activation_actual);
    if (max_activation_expected == max_activation_actual) {
      ++right_predictions;
    }
  }
  return right_predictions;
}

template <typename Iter>
double Perceptron::CalculateCost(const Iter begin, const Iter end) const {
  double cost = 0;
  std::size_t instances_count = 0;
  for (auto it = begin; it != end; ++it, ++instances_count) {
    const IData &instance = **it;
    const auto activation = Feedforward(instance.GetX());
    cost += cost_function_->Apply(instance.GetY(), activation);
  }
  return cost / instances_count;
```

```
}

}  // namespace nn
```

Листинг 7: Файл chromosome.h

```
#pragma once

#include <memory>
#include <vector>

#include "fitness_function.h"

namespace nn {

enum class ChromosomeSubclass {
  kSgdHyperparametersKit,
};

class IChromosome {
 public:
  static std::shared_ptr<IChromosome> Create(std::vector<double>&& genes,
                                             const ChromosomeSubclass subclass);

 public:
  virtual ~IChromosome() = default;

  virtual std::string ToString() const = 0;

 public:
  const std::vector<double>& get_genes() const;

 protected:
  IChromosome(std::vector<double>&& genes);

  std::vector<double> genes_;
};

class SgdHyperparametersKit final : public IChromosome {
  enum Index : std::size_t {
```

```
    kLearningRate,
    kEpochs,
    kMiniBatchSize,
    kHiddenLayers,
    kNeuronsPerHiddenLayer,
  };

  static constexpr std::size_t kHyperparametersNumber = 5;

 public:
  SgdHyperparametersKit(std::vector<double>&& hyperparameters);

  double get_learning_rate() const;
  std::size_t get_epochs() const;
  std::size_t get_mini_batch_size() const;
  std::size_t get_hidden_layers() const;
  std::size_t get_neurons_per_hidden_layer() const;

  std::string ToString() const override;
};

}  // namespace nn
```

Листинг 8: Файл chromosome.cc

```
#include "chromosome.h"

#include <sstream>
#include <stdexcept>

namespace nn {

std::shared_ptr<IChromosome> IChromosome::Create(
    std::vector<double>&& genes, const ChromosomeSubclass subclass) {
  switch (subclass) {
    case ChromosomeSubclass::kSgdHyperparametersKit:
      return std::make_shared<SgdHyperparametersKit>(std::move(genes));
  }
}

```

```
const std::vector<double>& IChromosome::get_genes() const { return genes_; }

IChromosome::IChromosome(std::vector<double>&& genes)
    : genes_(std::move(genes)) {}

SgdHyperparametersKit::SgdHyperparametersKit(
    std::vector<double>&& hyperparameters)
    : IChromosome(std::move(hyperparameters)) {
  if (genes_.size() != kHyperparametersNumber) {
    throw std::runtime_error(
        "Got " + std::to_string(genes_.size()) + " SGD hyperparameters, " +
        std::to_string(kHyperparametersNumber) + " expected");
  }
}

double SgdHyperparametersKit::get_learning_rate() const {
  return genes_.at(Index::kLearningRate);
}

std::size_t SgdHyperparametersKit::get_epochs() const {
  return genes_.at(Index::kEpochs);
}

std::size_t SgdHyperparametersKit::get_mini_batch_size() const {
  return genes_.at(Index::kMiniBatchSize);
}

std::size_t SgdHyperparametersKit::get_hidden_layers() const {
  return genes_.at(Index::kHiddenLayers);
}

std::size_t SgdHyperparametersKit::get_neurons_per_hidden_layer() const {
  return genes_.at(Index::kNeuronsPerHiddenLayer);
}

std::string SgdHyperparametersKit::ToString() const {
  auto oss = std::ostringstream{};
  oss << "- Learning rate: " << get_learning_rate()
      << ";\n- Epochs: " << get_epochs()
      << ";\n- Mini-batch size: " << get_mini_batch_size()
```

```
      << ";\n- Hidden layers: " << get_hidden_layers()
      << ";\n- Neurons per hidden layer: " << get_neurons_per_hidden_layer();
  return oss.str();
}

}  // namespace nn
```

Листинг 9: Файл fitness_function.h

```
#pragma once

#include <memory>

#include "perceptron.h"

namespace nn {

class IChromosome;

class IFitnessFunction {
 public:
  virtual ~IFitnessFunction() = default;

  virtual double Assess(const IChromosome& chromosome) const = 0;
};

class ISgdFitness : public IFitnessFunction {
 public:
  virtual ~ISgdFitness() = default;

  ISgdFitness(std::unique_ptr<IDataSupplier>&& data_supplier);

 protected:
  std::unique_ptr<IDataSupplier> data_supplier_;
};

class SgdFitness final : public ISgdFitness {
 public:
  using ISgdFitness::ISgdFitness;

```

```
  double Assess(const IChromosome& chromosome) const override;
};

class SgdNagFitness final : public ISgdFitness {
 public:
  using ISgdFitness::ISgdFitness;

  double Assess(const IChromosome& chromosome) const override;
};

class SgdAdagradFitness final : public ISgdFitness {
 public:
  using ISgdFitness::ISgdFitness;

  double Assess(const IChromosome& chromosome) const override;
};

class SgdAdamFitness final : public ISgdFitness {
 public:
  using ISgdFitness::ISgdFitness;

  double Assess(const IChromosome& chromosome) const override;
};

}  // namespace nn
```

Листинг 10: Файл fitness_function.cc

```
#include "fitness_function.h"

#include <cassert>

#include "chromosome.h"
#include "cost_function.h"
#include "perceptron.h"

namespace nn {

namespace {

```

```
double CostToFitness(const double value) {
  return std::exp(1 / (value + 1e-8));
}

}  // namespace

ISgdFitness::ISgdFitness(std::unique_ptr<IDataSupplier>&& data_supplier)
    : data_supplier_(std::move(data_supplier)) {}

double SgdFitness::Assess(const IChromosome& chromosome) const {
  const auto kit = static_cast<const SgdHyperparametersKit&>(chromosome);

  auto cost_function = std::make_unique<CrossEntropy>();

  auto activation_functions =
      std::vector<std::unique_ptr<IActivationFunction>>{};
  const auto hidden_layers = kit.get_hidden_layers();
  activation_functions.reserve(hidden_layers + 1);
  for (std::size_t i = 0; i < hidden_layers; ++i) {
    activation_functions.push_back(std::make_unique<LeakyReLU>(0.01));
  }
  activation_functions.push_back(std::make_unique<Softmax>());

  auto layers_sizes = std::vector<std::size_t>{};
  layers_sizes.reserve(hidden_layers + 2);
  layers_sizes.push_back(data_supplier_->GetInputLayerSize());
  const auto neurons_per_hidden_layer = kit.get_neurons_per_hidden_layer();
  for (std::size_t i = 0; i < hidden_layers; ++i) {
    layers_sizes.push_back(neurons_per_hidden_layer);
  }
  layers_sizes.push_back(data_supplier_->GetOutputLayerSize());

  auto perceptron = Perceptron(std::move(cost_function),
                               std::move(activation_functions), layers_sizes);
  const auto cfg = SgdConfiguration{
      .epochs = kit.get_epochs(),
      .mini_batch_size = kit.get_mini_batch_size(),
      .learning_rate = kit.get_learning_rate(),
      .monitor_train_cost = true,
      .monitor_train_accuracy = true,
```

```
    .monitor_test_cost = true,
    .monitor_test_accuracy = true,
  };

  const auto train_data = data_supplier_->GetTrainData();
  const auto test_data = data_supplier_->GetTestData();

  auto metrics = perceptron.Sgd(train_data, test_data, cfg);
  return CostToFitness(metrics.test_cost.back());
}

double SgdNagFitness::Assess(const IChromosome& chromosome) const {
  constexpr double kGamma = 0.9;

  const auto kit = static_cast<const SgdHyperparametersKit&>(chromosome);

  auto cost_function = std::make_unique<CrossEntropy>();

  auto activation_functions =
      std::vector<std::unique_ptr<IActivationFunction>>{};
  const auto hidden_layers = kit.get_hidden_layers();
  activation_functions.reserve(hidden_layers + 1);
  for (std::size_t i = 0; i < hidden_layers; ++i) {
    activation_functions.push_back(std::make_unique<LeakyReLU>(0.01));
  }
  activation_functions.push_back(std::make_unique<Softmax>());

  auto layers_sizes = std::vector<std::size_t>{};
  layers_sizes.reserve(hidden_layers + 2);
  layers_sizes.push_back(data_supplier_->GetInputLayerSize());
  const auto neurons_per_hidden_layer = kit.get_neurons_per_hidden_layer();
  for (std::size_t i = 0; i < hidden_layers; ++i) {
    layers_sizes.push_back(neurons_per_hidden_layer);
  }
  layers_sizes.push_back(data_supplier_->GetOutputLayerSize());

  auto perceptron = Perceptron(std::move(cost_function),
                               std::move(activation_functions), layers_sizes);
  const auto cfg = SgdConfiguration{
      .epochs = kit.get_epochs(),
```

```
    .mini_batch_size = kit.get_mini_batch_size(),
    .learning_rate = kit.get_learning_rate(),
    .monitor_train_cost = true,
    .monitor_train_accuracy = true,
    .monitor_test_cost = true,
    .monitor_test_accuracy = true,
  };

  const auto train_data = data_supplier_->GetTrainData();
  const auto test_data = data_supplier_->GetTestData();

  auto metrics = perceptron.SgdNag(train_data, test_data, cfg, kGamma);
  return CostToFitness(metrics.test_cost.back());
}

double SgdAdagradFitness::Assess(const IChromosome& chromosome) const {
  constexpr double kEpsilon = 1e-8;

  const auto kit = static_cast<const SgdHyperparametersKit&>(chromosome);

  auto cost_function = std::make_unique<CrossEntropy>();

  auto activation_functions =
    std::vector<std::unique_ptr<IActivationFunction>>{};
  const auto hidden_layers = kit.get_hidden_layers();
  activation_functions.reserve(hidden_layers + 1);
  for (std::size_t i = 0; i < hidden_layers; ++i) {
    activation_functions.push_back(std::make_unique<LeakyReLU>(0.01));
  }
  activation_functions.push_back(std::make_unique<Softmax>());

  auto layers_sizes = std::vector<std::size_t>{};
  layers_sizes.reserve(hidden_layers + 2);
  layers_sizes.push_back(data_supplier_->GetInputLayerSize());
  const auto neurons_per_hidden_layer = kit.get_neurons_per_hidden_layer();
  for (std::size_t i = 0; i < hidden_layers; ++i) {
    layers_sizes.push_back(neurons_per_hidden_layer);
  }
  layers_sizes.push_back(data_supplier_->GetOutputLayerSize());

```

```
  auto perceptron = Perceptron(std::move(cost_function),
                               std::move(activation_functions), layers_sizes);
  const auto cfg = SgdConfiguration{
      .epochs = kit.get_epochs(),
      .mini_batch_size = kit.get_mini_batch_size(),
      .learning_rate = kit.get_learning_rate(),
      .monitor_train_cost = true,
      .monitor_train_accuracy = true,
      .monitor_test_cost = true,
      .monitor_test_accuracy = true,
  };

  const auto train_data = data_supplier_->GetTrainData();
  const auto test_data = data_supplier_->GetTestData();

  auto metrics = perceptron.SgdAdagrad(train_data, test_data, cfg, kEpsilon);
  return CostToFitness(metrics.test_cost.back());
}

double SgdAdamFitness::Assess(const IChromosome& chromosome) const {
  constexpr double kEpsilon = 1e-8;
  constexpr double kBeta1 = 0.9;
  constexpr double kBeta2 = 0.999;

  const auto kit = static_cast<const SgdHyperparametersKit&>(chromosome);

  auto cost_function = std::make_unique<CrossEntropy>();

  auto activation_functions =
      std::vector<std::unique_ptr<IActivationFunction>>{};
  const auto hidden_layers = kit.get_hidden_layers();
  activation_functions.reserve(hidden_layers + 1);
  for (std::size_t i = 0; i < hidden_layers; ++i) {
    activation_functions.push_back(std::make_unique<LeakyReLU>(0.01));
  }
  activation_functions.push_back(std::make_unique<Softmax>());

  auto layers_sizes = std::vector<std::size_t>{};
  layers_sizes.reserve(hidden_layers + 2);
  layers_sizes.push_back(data_supplier_->GetInputLayerSize());
```

```
  const auto neurons_per_hidden_layer = kit.get_neurons_per_hidden_layer();
  for (std::size_t i = 0; i < hidden_layers; ++i) {
    layers_sizes.push_back(neurons_per_hidden_layer);
  }
  layers_sizes.push_back(data_supplier_->GetOutputLayerSize());

  auto perceptron = Perceptron(std::move(cost_function),
                               std::move(activation_functions), layers_sizes);
  const auto cfg = SgdConfiguration{
      .epochs = kit.get_epochs(),
      .mini_batch_size = kit.get_mini_batch_size(),
      .learning_rate = kit.get_learning_rate(),
      .monitor_train_cost = true,
      .monitor_train_accuracy = true,
      .monitor_test_cost = true,
      .monitor_test_accuracy = true,
  };

  const auto train_data = data_supplier_->GetTrainData();
  const auto test_data = data_supplier_->GetTestData();

  auto metrics =
      perceptron.SgdAdam(train_data, test_data, cfg, kBeta1, kBeta2, kEpsilon);
  return CostToFitness(metrics.test_cost.back());
}

}  // namespace nn
```

Листинг 11: **Файл** genetic_algorithm.h

```
#pragma once

#include <memory>
#include <random>
#include <vector>

#include "chromosome.h"
#include "fitness_function.h"

namespace nn {
```

```cpp

class Segment final {
  double left_, right_;

 public:
  Segment(const double left, const double right);

  double get_left() const;
  double get_right() const;
};

class GeneticAlgorithm final {
 public:
  // TODO: Validate the configuration
  struct Configuration final {
    std::size_t populations_number;
    std::size_t population_size;
    double crossover_proportion;
    double mutation_proportion;
  };

 private:
  std::mt19937 engine_{std::random_device{}()};

  std::unique_ptr<IFitnessFunction> fitness_function_;
  ChromosomeSubclass chromosome_subclass_;
  std::vector<std::uniform_real_distribution<double>> genes_distributions_;
  std::vector<std::shared_ptr<IChromosome>> population_;
  Configuration cfg_;
  std::size_t genes_number_;

 public:
  GeneticAlgorithm(std::unique_ptr<IFitnessFunction>&& fitness_function,
                   const ChromosomeSubclass subclass,
                   const std::vector&lt;Segment&gt;& segments,
                   const Configuration& cfg);

  std::shared_ptr<IChromosome> Run();

 private:
```

```
  std::vector<std::shared_ptr<IChromosome>> RouletteWheelSelection();
  void Crossover(std::vector<std::shared_ptr<IChromosome>>& population);
  void Mutate(std::vector<std::shared_ptr<IChromosome>>& population);

  std::vector<double> CalculateFitnessValue() const;
};

}  // namespace nn
```

Листинг 12: Файл genetic_algorithm.cc

```
#include "genetic_algorithm.h"

#include <spdlog/spdlog.h>

#include <algorithm>
#include <cassert>
#include <cmath>
#include <map>
#include <numeric>
#include <random>
#include <stdexcept>

#include "chromosome.h"

namespace nn {

Segment::Segment(const double left, const double right)
    : left_(left), right_(right) {
  if (left_ > right_) {
    throw std::runtime_error(
        "The left border must be less or equal than the right one");
  }
}

double Segment::get_left() const { return left_; }
double Segment::get_right() const { return right_; }

GeneticAlgorithm::GeneticAlgorithm(
    std::unique_ptr<IFitnessFunction>&& fitness_function,
```

```
    const ChromosomeSubclass subclass, const std::vector&lt;Segment&gt;& segments,
    const GeneticAlgorithm::Configuration& cfg)
    : fitness_function_(std::move(fitness_function)),
      chromosome_subclass_(subclass),
      cfg_(cfg),
      genes_number_(segments.size()) {
  genes_distributions_.reserve(genes_number_);
  for (auto&& segment : segments) {
    genes_distributions_.push_back(std::uniform_real_distribution<>(
        segment.get_left(), segment.get_right()));
  }

  population_.reserve(cfg.population_size);
  for (std::size_t i = 0; i < cfg.population_size; ++i) {
    auto genes = std::vector<double>{};
    genes.reserve(genes_number_);
    for (auto&& distribution : genes_distributions_) {
      genes.push_back(distribution(engine_));
    }
    population_.push_back(
        IChromosome::Create(std::move(genes), chromosome_subclass_));
  }
}

std::shared_ptr<IChromosome> GeneticAlgorithm::Run() {
  for (std::size_t i = 0; i < cfg_.populations_number; ++i) {
    spdlog::info("Population {}/{}:", i, cfg_.populations_number);
    for (std::size_t j = 0; j < cfg_.population_size; ++j) {
      spdlog::info("Chromosome {}/{}:\n{}", j + 1, cfg_.population_size,
                   population_[j]->ToString());
    }

    auto new_population = RouletteWheelSelection();
    Crossover(new_population);
    std::shuffle(new_population.begin(), new_population.end(), engine_);
    Mutate(new_population);
    std::shuffle(new_population.begin(), new_population.end(), engine_);

    population_ = std::move(new_population);
  }
```

```
  spdlog::info("Population {}/{}:", cfg_.populations_number,
               cfg_.populations_number);
  for (std::size_t j = 0; j < cfg_.population_size; ++j) {
    spdlog::info("Chromosome {}/{}:\n{}", j + 1, cfg_.population_size,
                 population_[j]->ToString());
  }

  const auto fitness_values = CalculateFitnessValue();
  const auto fittest_chromosome_index = std::distance(
      fitness_values.cbegin(),
      std::max_element(fitness_values.cbegin(), fitness_values.cend()));

  spdlog::info("Chromosome {} (the fittest one):\n{}",
               fittest_chromosome_index + 1,
               population_[fittest_chromosome_index]->ToString());
  return population_[fittest_chromosome_index];
}

std::vector<std::shared_ptr<IChromosome>>
GeneticAlgorithm::RouletteWheelSelection() {
  auto fitness_values = CalculateFitnessValue();
  for (std::size_t i = 0; i < cfg_.population_size; ++i) {
    if (!std::isfinite(fitness_values[i])) {
      fitness_values[i] = 0;
    }
  }

  auto partial_sum = std::vector<double>(cfg_.population_size);
  std::partial_sum(fitness_values.cbegin(), fitness_values.cend(),
                   partial_sum.begin());

  auto partial_sum_to_chromosome =
      std::map<double, std::shared_ptr<IChromosome>>{};
  for (std::size_t i = 0; i < cfg_.population_size; ++i) {
    partial_sum_to_chromosome.insert({partial_sum[i], population_[i]});
  }

  auto selected_chromosomes = std::vector<std::shared_ptr<IChromosome>>{};
  selected_chromosomes.reserve(cfg_.population_size);
```

```
  auto distribution =
      std::uniform_real_distribution<double>{0, partial_sum.back()};
  for (std::size_t i = 0; i < cfg_.population_size; ++i) {
    const auto value = distribution(engine_);
    const auto it = partial_sum_to_chromosome.upper_bound(value);
    assert(it != partial_sum_to_chromosome.end());
    selected_chromosomes.push_back(it->second);
  }

  return selected_chromosomes;
}

void GeneticAlgorithm::Crossover(
    std::vector<std::shared_ptr<IChromosome>>& population) {
  static const auto parents_number = static_cast<std::size_t>(
      cfg_.crossover_proportion * cfg_.population_size);
  static auto distribution = std::uniform_real_distribution<>{0.0, 1.0};
  for (std::size_t i = 0; i + 1 < parents_number; i += 2) {
    const auto alpha = distribution(engine_);

    const auto& parent1_genes = population[i]->get_genes();
    const auto& parent2_genes = population[i + 1]->get_genes();

    auto offspring1_genes = std::vector<double>{};
    offspring1_genes.reserve(genes_number_);
    for (std::size_t j = 0; j < genes_number_; ++j) {
      offspring1_genes.push_back(alpha * parent1_genes[j] +
                                 (1 - alpha) * parent2_genes[j]);
    }

    auto offspring2_genes = std::vector<double>{};
    offspring2_genes.reserve(genes_number_);
    for (std::size_t j = 0; j < genes_number_; ++j) {
      offspring2_genes.push_back((1 - alpha) * parent1_genes[j] +
                                 alpha * parent2_genes[j]);
    }

    population[i] =
        IChromosome::Create(std::move(offspring1_genes), chromosome_subclass_);
    population[i + 1] =
```

```
      IChromosome::Create(std::move(offspring2_genes), chromosome_subclass_);
  }
}

void GeneticAlgorithm::Mutate(
    std::vector<std::shared_ptr<IChromosome>>& population) {
  static const auto mutants_number =
      static_cast<std::size_t>(cfg_.mutation_proportion * cfg_.population_size);
  static auto distribution =
      std::uniform_int_distribution<>{0, static_cast<int>(genes_number_) - 1};
  for (std::size_t i = 0; i < mutants_number; ++i) {
    const auto mutated_gene_index = distribution(engine_);

    auto genes = population[i]->get_genes();
    genes[mutated_gene_index] =
        genes_distributions_[mutated_gene_index](engine_);
    population[i] = IChromosome::Create(std::move(genes), chromosome_subclass_);
  }
}

std::vector<double> GeneticAlgorithm::CalculateFitnessValue() const {
  auto fitness_values = std::vector<double>{};
  fitness_values.reserve(cfg_.population_size);
  for (std::size_t i = 0; i < cfg_.population_size; ++i) {
    fitness_values.push_back(fitness_function_->Assess(*population_[i]));
    spdlog::info("Chromosome {}/{} fittness value: {}", i + 1,
                 cfg_.population_size, fitness_values.back());
  }
  return fitness_values;
}

}  // namespace nn
```

Листинг 13: Файл main.cc

```
#include <matplot/matplot.h>
#include <spdlog/common.h>
#include <spdlog/spdlog.h>

#include <memory>
```

```
#include "chromosome.h"
#include "data_supplier.h"
#include "fitness_function.h"
#include "genetic_algorithm.h"

namespace {

const std::string kDefaultTestPath = "../../datasets/MNIST_CSV/test.csv";
const std::string kDefaultTrainPath = "../../datasets/MNIST_CSV/train.csv";

void RunGeneticAlgorithmSgd() {
  /*
   * Train cost: 0.0964943;
   * Train accuracy: 48532/50000;
   * Test cost: 0.14492;
   * Test accuracy: 9580/10000;
   *
   * Learning rate: 0.0959074;
   * Epochs: 100;
   * Mini-batch size: 100;
   * Hidden layers: 1;
   * Neurons per hidden layer: 28
   */

  auto data_supplier = std::make_unique<nn::DataSupplier>(
      kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
  auto fitness_function =
      std::make_unique<nn::SgdFitness>(std::move(data_supplier));
  const auto segments = std::vector<nn::Segment>{
      {0.001, 1},  // kLearningRate
      {100, 100},  // kEpochs
      {100, 100},  // kMiniBatchSize
      {0, 4},      // kHiddenLayer
      {10, 40},    // kNeuronsPerHiddenLayer
  };
  const auto cfg = nn::GeneticAlgorithm::Configuration{
      .populations_number = 10,
      .population_size = 60,
      .crossover_proportion = 0.4,
```

```
    .mutation_proportion = 0.15,
  };
  auto genetic_algorithm = nn::GeneticAlgorithm(
    std::move(fitness_function),
    nn::ChromosomeSubclass::kSgdHyperparametersKit, segments, cfg);
  genetic_algorithm.Run();
}

void RunGeneticAlgorithmSgdNag() {
  /*
   * Train cost: 0.0957484;
   * Train accuracy: 48562/50000;
   * Test cost: 0.146654;
   * Test accuracy: 9565/10000;
   *
   * Learning rate: 0.0100863;
   * Epochs: 100;
   * Mini-batch size: 100;
   * Hidden layers: 1;
   * Neurons per hidden layer: 28
   */

  auto data_supplier = std::make_unique<nn::DataSupplier>(
    kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
  auto fitness_function =
    std::make_unique<nn::SgdNagFitness>(std::move(data_supplier));
  const auto segments = std::vector<nn::Segment>{
    {0.001, 1},  // kLearningRate
    {100, 100},  // kEpochs
    {100, 100},  // kMiniBatchSize
    {0, 4},      // kHiddenLayer
    {10, 40},    // kNeuronsPerHiddenLayer
  };
  const auto cfg = nn::GeneticAlgorithm::Configuration{
    .populations_number = 10,
    .population_size = 60,
    .crossover_proportion = 0.4,
    .mutation_proportion = 0.15,
  };
  auto genetic_algorithm = nn::GeneticAlgorithm(
```

```
      std::move(fitness_function),
      nn::ChromosomeSubclass::kSgdHyperparametersKit, segments, cfg);
  genetic_algorithm.Run();
}

void RunGeneticAlgorithmSgdAdagrad() {
  /*
   * Train cost: 0.181089;
   * Train accuracy: 47361/50000;
   * Test cost: 0.214604;
   * Test accuracy: 9395/10000;
   *
   * Learning rate: 0.936544;
   * Epochs: 100;
   * Mini-batch size: 100;
   * Hidden layers: 1;
   * Neurons per hidden layer: 36
   */

  auto data_supplier = std::make_unique<nn::DataSupplier>(
      kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
  auto fitness_function =
      std::make_unique<nn::SgdAdagradFitness>(std::move(data_supplier));
  const auto segments = std::vector<nn::Segment>{
      {0.001, 1},  // kLearningRate
      {100, 100},  // kEpochs
      {100, 100},  // kMiniBatchSize
      {0, 4},      // kHiddenLayer
      {10, 40},    // kNeuronsPerHiddenLayer
  };
  const auto cfg = nn::GeneticAlgorithm::Configuration{
      .populations_number = 10,
      .population_size = 60,
      .crossover_proportion = 0.4,
      .mutation_proportion = 0.15,
  };
  auto genetic_algorithm = nn::GeneticAlgorithm(
      std::move(fitness_function),
      nn::ChromosomeSubclass::kSgdHyperparametersKit, segments, cfg);
  genetic_algorithm.Run();
```

```
}

void RunGeneticAlgorithmSgdAdam() {
  /*
   * Train cost: 0.0886979;
   * Train accuracy: 48692/50000;
   * Test cost: 0.143935;
   * Test accuracy: 9620/10000;
   *
   * Learning rate: 0.0462101;
   * Epochs: 100;
   * Mini-batch size: 100;
   * Hidden layers: 3;
   * Neurons per hidden layer: 35
   */

  auto data_supplier = std::make_unique<nn::DataSupplier>(
      kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
  auto fitness_function =
      std::make_unique<nn::SgdAdamFitness>(std::move(data_supplier));
  const auto segments = std::vector<nn::Segment>{
      {0.001, 1},  // kLearningRate
      {100, 100},  // kEpochs
      {100, 100},  // kMiniBatchSize
      {0, 4},      // kHiddenLayer
      {10, 40},    // kNeuronsPerHiddenLayer
  };
  const auto cfg = nn::GeneticAlgorithm::Configuration{
      .populations_number = 10,
      .population_size = 60,
      .crossover_proportion = 0.4,
      .mutation_proportion = 0.15,
  };
  auto genetic_algorithm = nn::GeneticAlgorithm(
      std::move(fitness_function),
      nn::ChromosomeSubclass::kSgdHyperparametersKit, segments, cfg);
  genetic_algorithm.Run();
}

}  // namespace
```

```

int main() {
  RunGeneticAlgorithmSgd();
  RunGeneticAlgorithmSgdNag();
  RunGeneticAlgorithmSgdAdagrad();
  RunGeneticAlgorithmSgdAdam();
}
```

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 пересептрона 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.