



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 № 1**  
**по курсу «Теория искусственных нейронных сетей»**  
**«Реализация однослойного персептрона»**

Студент группы ИУ9-71Б Афанасьев И.

Преподаватель Каганов Ю. Т.

*Москва 2024*

# 1 Цель работы

1. Реализовать на языке высокого уровня однослойный персептрон и проверить его работоспособность на примере искусственных данных типа цифр от 0 до 9 и букв русского алфавита. Размер поля  $5 \times 4$ .
2. Исследовать работу персептрона на основе использования различных функций активации. (Линейной, сигмоиды, гиперболического тангенса, ReLu).

## 2 Постановка задачи

Основной поставленной задачей является разработка каркаса для обучения *многослойных* персептронов — это сделано для возможности переиспользования программного кода в последующих домашних заданиях. Реализованный персептрон должен поддерживать произвольное число скрытых слоёв, параметризацию различными функциями активации и ошибки. Персептрон должен обучаться методом стохастического градиентного спуска с применением метода обратного распространения ошибки для вычисления градиентов.

## 3 Реализация

Код программы написан на языке C++. Для выполнения матричных операций используется библиотека **Eigen**.

В листинге **1** приводится реализация функций активации: линейной, ReLU, Leaky ReLU, сигмоиды, гиперболического тангенса и Softmax.

В листинге **2** приводится реализация функций ошибки: MSE, кросс-энтропии и дивергенции Кульбака-Лейблера.

В листингах **3** и **4** приводится реализация многослойного персептрона, его обучения методом стохастического градиентного спуска (с mini-batch оптимизацией) с вычислением градиентов методом обратного распространения ошибки, а также фиксирования показателей ошибки и точности персептрона на тестовых данных для каждой эпохи.

В листингах 5, 6, 7 приводится программный код генерации данных для обучения, валидации и тестирования. Задаётся по 10 образцов цифр и букв русского алфавита, для каждого из которых порождается  $2^4 = 16$  модификаций. Всего сгенерированных данных —  $20 \cdot 16 = 320$ , и они равномерно распределяются между данными для обучения, валидации и тестирования в соотношении 50%, 20% и 30% соответственно.

В листинге 8 приводится main-файл с различными конфигурациями одно-слойного персептрона.

Листинг 1: Файл activation\_function.h

```
1  #pragma once
2
3  #include <Eigen/Dense>
4
5  namespace nn {
6
7  class IActivationFunction {
8  public:
9      virtual ~IActivationFunction() = default;
10
11  public:
12      virtual Eigen::VectorXd Apply(const Eigen::VectorXd& z) = 0;
13      virtual Eigen::MatrixXd Jacobian(const Eigen::VectorXd& z) = 0;
14  };
15
16  class Linear final : public IActivationFunction {
17  public:
18      Eigen::VectorXd Apply(const Eigen::VectorXd& z) override { return z; }
19
20      Eigen::MatrixXd Jacobian(const Eigen::VectorXd& z) override {
21          return Eigen::MatrixXd::Identity(z.rows(), z.cols());
22      }
23  };
24
25  class ReLU final : public IActivationFunction {
26  public:
27      Eigen::VectorXd Apply(const Eigen::VectorXd& z) override {
28          return z.array().max(0.0);
```

```

29     }
30
31     Eigen::MatrixXd Jacobian(const Eigen::VectorXd& z) override {
32         return z.array().cwiseTypedGreaterOrEqual(0.0).matrix().asDiagonal();
33     }
34 };
35
36 class LeakyReLU final : public IActivationFunction {
37     std::function<double(double)> f_, f_prime_;
38
39     public:
40     LeakyReLU(const double alpha)
41         : f_([alpha](const double x) { return x >= 0 ? x : alpha * x; }),
42         f_prime_([alpha](const double x) { return x >= 0 ? 1 : alpha; }) {}
43
44     Eigen::VectorXd Apply(const Eigen::VectorXd& z) override {
45         return z.unaryExpr(f_);
46     }
47
48     Eigen::MatrixXd Jacobian(const Eigen::VectorXd& z) override {
49         return z.unaryExpr(f_prime_).asDiagonal();
50     }
51 };
52
53 class Sigmoid final : public IActivationFunction {
54     public:
55     Eigen::VectorXd Apply(const Eigen::VectorXd& z) override {
56         return 1.0 / (1.0 + (-z).array().exp());
57     }
58
59     Eigen::MatrixXd Jacobian(const Eigen::VectorXd& z) override {
60         const auto sigmoid = Apply(z);
61         return (sigmoid.array() * (1 - sigmoid.array())).matrix().asDiagonal();
62     }
63 };
64
65 class Tanh final : public IActivationFunction {
66     public:
67     Eigen::VectorXd Apply(const Eigen::VectorXd& z) override {
68         const auto e_z = z.array().exp();

```

```

69     const auto e_neg_z = (-z).array().exp();
70     return (e_z - e_neg_z) / (e_z + e_neg_z);
71 }
72
73 Eigen::MatrixXd Jacobian(const Eigen::VectorXd& z) override {
74     const auto tanh = Apply(z);
75     return (1 - tanh.array().square()).matrix().asDiagonal();
76 }
77 };
78
79 class Softmax final : public IActivationFunction {
80 public:
81     Eigen::VectorXd Apply(const Eigen::VectorXd& z) override {
82         const auto e_z = z.array().exp();
83         return e_z / e_z.sum();
84     }
85
86     Eigen::MatrixXd Jacobian(const Eigen::VectorXd& z) override {
87         const auto softmax = Apply(z);
88         return softmax.asDiagonal().toDenseMatrix() - softmax * softmax.transpose();
89     }
90 };
91
92 } // 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 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::vector<std::shared_ptr<const IData>> GetTrainingData() const = 0;
28     virtual std::vector<std::shared_ptr<const IData>> GetValidationData()
29         const = 0;
30     virtual std::vector<std::shared_ptr<const IData>> GetTestingData() const = 0;
31 };
32
33 struct Config final {
34     std::size_t epochs;
35     std::size_t mini_batch_size;

```

```

36     double eta;
37     bool monitor_training_cost;
38     bool monitor_training_accuracy;
39     bool monitor_testing_cost;
40     bool monitor_testing_accuracy;
41 };
42
43 struct Metric final {
44     std::vector<double> training_cost, training_accuracy;
45     std::vector<double> testing_cost, testing_accuracy;
46 };
47
48 class Perceptron final {
49     std::random_device device_;
50     std::default_random_engine generator_;
51     std::unique_ptr<ICostFunction> cost_function_;
52     std::size_t layers_number_, connections_number_;
53     std::vector<Eigen::MatrixXd> weights_;
54     std::vector<Eigen::VectorXd> biases_;
55     std::vector<std::unique_ptr<IActivationFunction>> activation_functions_;
56
57 public:
58     Perceptron(
59         std::unique_ptr<ICostFunction>&& cost_function,
60         std::vector<std::unique_ptr<IActivationFunction>>&& activation_functions,
61         const std::vector<std::size_t>& layers_sizes);
62
63     Eigen::VectorXd Feedforward(const Eigen::VectorXd& x) const;
64
65     Metric StochasticGradientSearch(
66         const std::vector<std::shared_ptr<const IData>>& training,
67         const std::vector<std::shared_ptr<const IData>>& testing,
68         const Config& cfg);
69
70     const std::vector<Eigen::MatrixXd>& get_weights() const { return weights_; }
71     const std::vector<Eigen::VectorXd>& get_biases() const { return biases_; }
72
73 private:
74     template <typename Iter>
75     void UpdateMiniBatch(const Iter mini_batch_begin, const Iter mini_batch_end,

```



```

76         const std::size_t mini_batch_size, const double eta);
77
78     std::pair<std::vector<Eigen::MatrixXd>, std::vector<Eigen::VectorXd>>
79     Backpropagation(const Eigen::VectorXd& x, const Eigen::VectorXd& y);
80
81     std::pair<std::vector<Eigen::VectorXd>, std::vector<Eigen::VectorXd>>
82     FeedforwardDetailed(const Eigen::VectorXd& x);
83
84     Metric GetMetric(const Config& cfg) const;
85
86     void WriteMetric(Metric& metric, const std::size_t epoch,
87                     const std::vector<std::shared_ptr<const IData>>& training,
88                     const std::vector<std::shared_ptr<const IData>>& testing,
89                     const Config& cfg) const;
90
91     template <typename Iter>
92     std::size_t Accuracy(const Iter begin, const Iter end) const;
93
94     template <typename Iter>
95     double Cost(const Iter begin, const Iter end) const;
96 };
97
98 } // namespace nn

```

Листинг 4: Файл perceptron.cc

```

1  #include "perceptron.h"
2
3  #include <spdlog/spdlog.h>
4
5  #include <cassert>
6  #include <iostream>
7  #include <iterator>
8  #include <sstream>
9
10 namespace nn {
11
12     Perceptron::Perceptron(
13         std::unique_ptr<ICostFunction>&& cost_function,
14         std::vector<std::unique_ptr<IActivationFunction>>&& activation_functions,

```

```

15     const std::vector<std::size_t>& layers_sizes)
16     : generator_(device_()),
17       cost_function_(std::move(cost_function)),
18       layers_number_(layers_sizes.size()),
19       connections_number_(layers_number_ - 1),
20       activation_functions_(std::move(activation_functions)) {
21     if (layers_number_ < 2) {
22         throw std::runtime_error("Perceptron must have at least two layers");
23     }
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::StochasticGradientSearch(
49     const std::vector<std::shared_ptr<const IData>>& training,
50     const std::vector<std::shared_ptr<const IData>>& testing,
51     const Config& cfg) {
52     const auto training_size = training.size();
53     const auto whole_mini_batches_number = training_size / cfg.mini_batch_size;
54     const auto remainder_mini_batch_size = training_size % cfg.mini_batch_size;

```

```

55
56 auto training_shuffled = std::vector(training.begin(), training.end());
57 auto metric = GetMetric(cfg);
58 for (std::size_t i = 0; i < cfg.epochs; ++i) {
59     std::shuffle(training_shuffled.begin(), training_shuffled.end(),
60                 generator_);
61     auto it = training_shuffled.begin();
62     for (std::size_t i = 0; i < whole_mini_batches_number; ++i) {
63         auto end = it + cfg.mini_batch_size;
64         UpdateMiniBatch(it, end, cfg.mini_batch_size, cfg.eta);
65         it = std::move(end);
66     }
67     if (remainder_mini_batch_size != 0) {
68         UpdateMiniBatch(it, it + remainder_mini_batch_size,
69                         remainder_mini_batch_size, cfg.eta);
70     }
71     WriteMetric(metric, i, training, testing, cfg);
72 }
73 return metric;
74 }
75
76 template <typename Iter>
77 void Perceptron::UpdateMiniBatch(const Iter mini_batch_begin,
78                                  const Iter mini_batch_end,
79                                  const std::size_t mini_batch_size,
80                                  const double eta) {
81     auto nabra_weights = std::vector<Eigen::MatrixXd>{};
82     nabra_weights.reserve(connections_number_);
83     for (auto&& w : weights_) {
84         nabra_weights.push_back(Eigen::MatrixXd::Zero(w.rows(), w.cols()));
85     }
86
87     auto nabra_biases = std::vector<Eigen::VectorXd>{};
88     nabra_biases.reserve(connections_number_);
89     for (auto&& b : biases_) {
90         nabra_biases.push_back(Eigen::VectorXd::Zero(b.size()));
91     }
92
93     for (auto it = mini_batch_begin; it != mini_batch_end; ++it) {
94         const auto& data = *it;

```

```

95     const auto [nabla_weights_part, nabla_biases_part] =
96         Backpropagation(data.GetX(), data.GetY());
97     for (std::size_t i = 0; i < connections_number_; ++i) {
98         nabla_weights[i] += nabla_weights_part[i];
99         nabla_biases[i] += nabla_biases_part[i];
100     }
101 }
102
103 const auto learning_rate = eta / mini_batch_size;
104 for (std::size_t i = 0; i < connections_number_; ++i) {
105     weights_[i] -= learning_rate * nabla_weights[i];
106     biases_[i] -= learning_rate * nabla_biases[i];
107 }
108 }
109
110 std::pair<std::vector<Eigen::MatrixXd>, std::vector<Eigen::VectorXd>>
111 Perceptron::Backpropagation(const Eigen::VectorXd& x,
112                             const Eigen::VectorXd& y) {
113     const auto [zs, activations] = FeedforwardDetailed(x);
114     assert(zs.size() == connections_number_);
115     assert(activations.size() == layers_number_);
116
117     auto delta = static_cast<Eigen::VectorXd>(
118         activation_functions_.back()->Jacobian(zs.back()).transpose() *
119         cost_function_->GradientWrtActivations(y, activations.back()));
120
121     auto nabla_weights_reversed = std::vector<Eigen::MatrixXd>{};
122     nabla_weights_reversed.reserve(connections_number_);
123     nabla_weights_reversed.push_back(
124         delta * std::prev(activations.cend(), 2)->transpose());
125
126     auto nabla_biases_reversed = std::vector<Eigen::VectorXd>{};
127     nabla_biases_reversed.reserve(connections_number_);
128     nabla_biases_reversed.push_back(delta);
129
130     for (int i = connections_number_ - 2; i >= 0; --i) {
131         delta = (weights_[i + 1] * activation_functions_[i]->Jacobian(zs[i]))
132             .transpose() *
133             delta;
134         nabla_weights_reversed.push_back(delta * activations[i].transpose());

```

```

135     nabla_biases_reversed.push_back(delta);
136 }
137
138 return {{std::make_move_iterator(nabla_weights_reversed.rbegin()),
139         std::make_move_iterator(nabla_weights_reversed.rend())},
140         {std::make_move_iterator(nabla_biases_reversed.rbegin()),
141         std::make_move_iterator(nabla_biases_reversed.rend())}};
142 }
143
144 std::pair<std::vector<Eigen::VectorXd>, std::vector<Eigen::VectorXd>>
145 Perceptron::FeedforwardDetailed(const Eigen::VectorXd& x) {
146     std::vector<Eigen::VectorXd> zs, activations;
147     zs.reserve(connections_number_);
148     activations.reserve(layers_number_);
149
150     auto activation = x;
151     for (std::size_t i = 0; i < connections_number_; ++i) {
152         auto z =
153             static_cast<Eigen::VectorXd>(weights_[i] * activation + biases_[i]);
154         activations.push_back(std::move(activation));
155         activation = activation_functions_[i]->Apply(z);
156         zs.push_back(std::move(z));
157     }
158     activations.push_back(std::move(activation));
159
160     return {zs, activations};
161 }
162
163 Metric Perceptron::GetMetric(const Config& param) const {
164     auto metric = Metric{};
165     if (param.monitor_training_cost) {
166         metric.training_cost.reserve(param.epochs);
167     }
168     if (param.monitor_training_accuracy) {
169         metric.training_accuracy.reserve(param.epochs);
170     }
171     if (param.monitor_testing_cost) {
172         metric.testing_cost.reserve(param.epochs);
173     }
174     if (param.monitor_testing_accuracy) {

```

```

175     metric.testing_accuracy.reserve(param.epochs);
176 }
177 return metric;
178 }
179
180 void Perceptron::WriteMetric(
181     Metric& metric, const std::size_t epoch,
182     const std::vector<std::shared_ptr<const IData>>& training,
183     const std::vector<std::shared_ptr<const IData>>& testing,
184     const Config& cfg) const {
185     std::stringstream oss;
186     oss << "Epoch " << epoch << ";";
187     if (cfg.monitor_training_cost) {
188         const auto training_cost = Cost(training.begin(), training.end());
189         metric.training_cost.push_back(training_cost);
190         oss << " training cost: " << training_cost << ";";
191     }
192     if (cfg.monitor_training_accuracy) {
193         const auto training_accuracy = Accuracy(training.begin(), training.end());
194         metric.training_accuracy.push_back(training_accuracy);
195         oss << " training accuracy: " << training_accuracy << "/" << training.size()
196             << ";";
197     }
198     if (cfg.monitor_testing_cost) {
199         const auto testing_cost = Cost(testing.begin(), testing.end());
200         metric.testing_cost.push_back(Cost(testing.begin(), testing.end()));
201         oss << " testing cost: " << testing_cost << ";";
202     }
203     if (cfg.monitor_testing_accuracy) {
204         const auto testing_accuracy = Accuracy(testing.begin(), testing.end());
205         metric.testing_accuracy.push_back(testing_accuracy);
206         oss << " testing accuracy: " << testing_accuracy << "/" << testing.size()
207             << ";";
208     }
209     spdlog::info(oss.str());
210 }
211
212 template <typename Iter>
213 std::size_t Perceptron::Accuracy(const Iter begin, const Iter end) const {
214     std::size_t right_predictions = 0;

```

```

215     for (auto it = begin; it != end; ++it) {
216         const IData& instance = **it;
217         Eigen::Index max_activation_expected, max_activation_actual;
218         instance.GetY().maxCoeff(&max_activation_expected);
219         Feedforward(instance.GetX()).maxCoeff(&max_activation_actual);
220         if (max_activation_expected == max_activation_actual) {
221             ++right_predictions;
222         }
223     }
224     return right_predictions;
225 }
226
227 template <typename Iter>
228 double Perceptron::Cost(const Iter begin, const Iter end) const {
229     double cost = 0;
230     std::size_t instances_count = 0;
231     for (auto it = begin; it != end; ++it, ++instances_count) {
232         const IData& instance = **it;
233         const auto activation = Feedforward(instance.GetX());
234         cost += cost_function_ -> Apply(instance.GetY(), activation);
235     }
236     return cost / instances_count;
237 }
238
239 } // namespace nn

```

Листинг 5: 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 hw1 {
10
11     constexpr std::size_t kScanSize = 20;
12     constexpr std::size_t kClassesCount = 20;

```

```

13
14 struct Symbol final {
15     Eigen::VectorXd scan;
16     std::string label;
17 };
18
19 class Data final : public nn::IData {
20     std::shared_ptr<const Symbol> x_;
21     std::shared_ptr<const Eigen::VectorXd> y_;
22
23 public:
24     Data(std::shared_ptr<const Symbol> x,
25         std::shared_ptr<const Eigen::VectorXd> y)
26         : x_(std::move(x)), y_(std::move(y)) {}
27
28     const Eigen::VectorXd& GetX() const override { return x_>scan; }
29     const Eigen::VectorXd& GetY() const override { return *y_; }
30     std::string_view ToString() const override { return x_>label; }
31 };
32
33 class DataSupplier final : public nn::IDataSupplier {
34     struct Parametrization;
35
36     static const Parametrization& kParametrization;
37
38     std::vector<std::shared_ptr<const nn::IData>> training_;
39     std::vector<std::shared_ptr<const nn::IData>> validation_;
40     std::vector<std::shared_ptr<const nn::IData>> testing_;
41
42 public:
43     DataSupplier(const double low_score, const double high_score);
44
45     std::vector<std::shared_ptr<const nn::IData>> GetTrainingData()
46         const override {
47         return training_;
48     }
49
50     std::vector<std::shared_ptr<const nn::IData>> GetValidationData()
51         const override {
52         return validation_;

```



```

53     }
54
55     std::vector<std::shared_ptr<const nn::IData>> GetTestingData()
56         const override {
57         return testing_;
58     }
59 };
60
61 } // namespace hw1

```

## Листинг 6: Файл data\_supplier.cc

```

1  #include "data_supplier.h"
2
3  #include <cassert>
4  #include <cmath>
5  #include <cstdlib>
6
7  #include "util.h"
8
9  namespace hw1 {
10
11     namespace {
12
13         class IPixelModifier {
14         public:
15             ~IPixelModifier() = default;
16
17         public:
18             virtual double Modify(const double value) const = 0;
19         };
20
21         class PixelReverser final : public IPixelModifier {
22         public:
23             double Modify(const double value) const override {
24                 assert(value == 0 || value == 1);
25                 return !value;
26             }
27         };
28

```

```

29 class SymbolModifier final {
30     Symbol sample_;
31     std::vector<std::size_t> indices_to_modify_;
32
33 public:
34     SymbolModifier(Symbol&& sample, std::vector<std::size_t>&& indices_to_modify)
35         : sample_(std::move(sample)),
36           indices_to_modify_(std::move(indices_to_modify)) {}
37
38     std::vector<std::shared_ptr<const Symbol>> GenerateModifications(
39         const IPixelModifier& pixel_modifier) const {
40         const auto indices_powerset = GeneratePowerset(indices_to_modify_);
41
42         auto modifications = std::vector<std::shared_ptr<const Symbol>>{};
43         modifications.reserve(indices_powerset.size());
44
45         for (auto&& indices : indices_powerset) {
46             auto variation = sample_;
47
48             for (auto&& index : indices) {
49                 auto& value = variation.scan(index);
50                 value = pixel_modifier.Modify(value);
51             }
52
53             modifications.push_back(
54                 std::make_shared<const Symbol>(std::move(variation)));
55         }
56
57         return modifications;
58     }
59 };
60
61 const std::array kSymbolModifiers{
62     SymbolModifier{{Eigen::Vector<double, kScanSize>{
63         1, 1, 1, 1, //
64         1, 0, 0, 1, //
65         1, 0, 0, 1, //
66         1, 0, 0, 1, //
67         1, 1, 1, 1, //
68     }},

```

```

69         "0"},
70         {0, 3, 16, 19}},
71     SymbolModifier{{Eigen::Vector<double, kScanSize>{
72         0, 0, 1, 0, //
73         0, 1, 1, 0, //
74         0, 0, 1, 0, //
75         0, 0, 1, 0, //
76         0, 1, 1, 1, //
77     }},
78     "1"},
79     {2, 5, 17, 19}},
80     SymbolModifier{{Eigen::Vector<double, kScanSize>{
81         1, 1, 1, 1, //
82         0, 0, 0, 1, //
83         1, 1, 1, 1, //
84         1, 0, 0, 0, //
85         1, 1, 1, 1, //
86     }},
87     "2"},
88     {3, 8, 11, 16}},
89     SymbolModifier{{Eigen::Vector<double, kScanSize>{
90         1, 1, 1, 1, //
91         0, 0, 0, 1, //
92         1, 1, 1, 1, //
93         0, 0, 0, 1, //
94         1, 1, 1, 1, //
95     }},
96     "3"},
97     {3, 8, 11, 19}},
98     SymbolModifier{{Eigen::Vector<double, kScanSize>{
99         1, 0, 0, 1, //
100        1, 0, 0, 1, //
101        1, 1, 1, 1, //
102        0, 0, 0, 1, //
103        0, 0, 0, 1, //
104    }},
105    "4"},
106    {0, 3, 8, 11}},
107    SymbolModifier{{Eigen::Vector<double, kScanSize>{
108        1, 1, 1, 1, //

```

```

109         1, 0, 0, 0, //
110         1, 1, 1, 1, //
111         0, 0, 0, 1, //
112         1, 1, 1, 1, //
113     },
114     "5"},
115     {0, 8, 11, 19}},
116 SymbolModifier{{Eigen::Vector<double, kScanSize>{
117     1, 1, 1, 1, //
118     1, 0, 0, 0, //
119     1, 1, 1, 1, //
120     1, 0, 0, 1, //
121     1, 1, 1, 1, //
122 },
123     "6"},
124     {0, 11, 16, 19}},
125 SymbolModifier{{Eigen::Vector<double, kScanSize>{
126     1, 1, 1, 1, //
127     0, 0, 0, 1, //
128     0, 0, 1, 0, //
129     0, 1, 0, 0, //
130     1, 0, 0, 0, //
131 },
132     "7"},
133     {9, 11, 14, 17}},
134 SymbolModifier{{Eigen::Vector<double, kScanSize>{
135     1, 1, 1, 1, //
136     1, 0, 0, 1, //
137     1, 1, 1, 1, //
138     1, 0, 0, 1, //
139     1, 1, 1, 1, //
140 },
141     "8"},
142     {0, 3, 16, 19}},
143 SymbolModifier{{Eigen::Vector<double, kScanSize>{
144     1, 1, 1, 1, //
145     1, 0, 0, 1, //
146     1, 1, 1, 1, //
147     0, 0, 0, 1, //
148     1, 1, 1, 1, //

```

```

149         },
150         "9"},
151         {0, 3, 8, 19}},
152     SymbolModifier{{Eigen::Vector<double, kScanSize>{
153         0, 1, 1, 0, //
154         1, 0, 0, 1, //
155         1, 0, 0, 1, //
156         1, 1, 1, 1, //
157         1, 0, 0, 1, //
158     }},
159     "A"},
160     {4, 5, 6, 7}},
161     SymbolModifier{{Eigen::Vector<double, kScanSize>{
162         1, 1, 1, 1, //
163         1, 0, 0, 0, //
164         1, 1, 1, 1, //
165         1, 0, 0, 0, //
166         1, 1, 1, 1, //
167     }},
168     "E"},
169     {0, 8, 11, 16}},
170     SymbolModifier{{Eigen::Vector<double, kScanSize>{
171         1, 0, 0, 1, //
172         1, 0, 1, 0, //
173         1, 1, 0, 0, //
174         1, 0, 1, 0, //
175         1, 0, 0, 1, //
176     }},
177     "K"},
178     {0, 3, 16, 19}},
179     SymbolModifier{{Eigen::Vector<double, kScanSize>{
180         1, 0, 0, 1, //
181         1, 0, 0, 1, //
182         1, 1, 1, 1, //
183         1, 0, 0, 1, //
184         1, 0, 0, 1, //
185     }},
186     "H"},
187     {0, 3, 16, 19}},
188     SymbolModifier{{Eigen::Vector<double, kScanSize>{

```

```

189         1, 1, 1, 1, //
190         1, 0, 0, 1, //
191         1, 0, 0, 1, //
192         1, 0, 0, 1, //
193         1, 0, 0, 1, //
194     },
195     "Π",
196     {0, 3, 16, 19}},
197 SymbolModifier{{Eigen::Vector<double, kScanSize>{
198     1, 1, 1, 1, //
199     1, 0, 0, 1, //
200     1, 1, 1, 1, //
201     1, 0, 0, 0, //
202     1, 0, 0, 0, //
203 }},
204     "P",
205     {0, 3, 11, 16}},
206 SymbolModifier{{Eigen::Vector<double, kScanSize>{
207     1, 1, 1, 1, //
208     1, 0, 0, 0, //
209     1, 0, 0, 0, //
210     1, 0, 0, 0, //
211     1, 1, 1, 1, //
212 }},
213     "C",
214     {0, 3, 16, 19}},
215 SymbolModifier{{Eigen::Vector<double, kScanSize>{
216     1, 0, 0, 1, //
217     1, 0, 0, 1, //
218     1, 1, 1, 1, //
219     0, 0, 0, 1, //
220     1, 1, 1, 1, //
221 }},
222     "Y",
223     {0, 3, 8, 19}},
224 SymbolModifier{{Eigen::Vector<double, kScanSize>{
225     1, 0, 0, 0, //
226     1, 0, 0, 0, //
227     1, 1, 1, 1, //
228     1, 0, 0, 1, //

```

```

229         1, 1, 1, 1, //
230     },
231     "Б"},
232     {0, 11, 16, 19}},
233     SymbolModifier{{Eigen::Vector<double, kScanSize>{
234         1, 1, 1, 1, //
235         1, 0, 0, 1, //
236         1, 1, 1, 1, //
237         0, 1, 0, 1, //
238         1, 0, 0, 1, //
239     },
240     "Я"},
241     {0, 3, 8, 14}},
242 };
243
244 } // namespace
245
246 struct DataSupplier::Parametrization final {
247     static constexpr double kTrainingRatio = 0.5;
248     static constexpr double kValidationRatio = 0.2;
249     static constexpr double kTestingRatio = 0.3;
250     static constexpr std::array kLabels{"0", "1", "2", "3", "4", "5", "6",
251                                         "7", "8", "9", "A", "E", "K", "H",
252                                         "П", "P", "C", "Y", "Б", "Я"};
253     static constexpr std::size_t kLabelsNumber = kLabels.size();
254
255     std::vector<std::shared_ptr<const Symbol>> training;
256     std::vector<std::shared_ptr<const Symbol>> validation;
257     std::vector<std::shared_ptr<const Symbol>> testing;
258
259     static const Parametrization& GetInstance() {
260         static Parametrization parametrization{};
261         return parametrization;
262     }
263
264     Parametrization(const Parametrization& other) = delete;
265     Parametrization& operator=(const Parametrization& other) = delete;
266
267 private:
268     std::default_random_engine generator;

```

```

269
270     Parametrization();
271 };
272
273 const DataSupplier::Parametrization& DataSupplier::kParametrization =
274     DataSupplier::Parametrization::GetInstance();
275
276 constexpr double kEpsilon = 10e-6;
277
278 bool AreEqual(const double x, const double y, const double epsilon) {
279     return std::abs(x - y) < epsilon;
280 }
281
282 DataSupplier::Parametrization::Parametrization() {
283     const auto pixel_reverser = PixelReverser{};
284     for (auto&& symbol_modifier : kSymbolModifiers) {
285         auto modifications = symbol_modifier.GenerateModifications(pixel_reverser);
286         const auto modifications_size = modifications.size();
287
288         std::shuffle(modifications.begin(), modifications.end(), generator);
289         auto begin = std::make_move_iterator(modifications.begin());
290         auto error = 0.0;
291         for (auto&& [dataset, ratio] : {
292             std::make_pair(std::reference_wrapper(training), kTrainingRatio),
293             std::make_pair(std::reference_wrapper(validation),
294                 kValidationRatio),
295             std::make_pair(std::reference_wrapper(testing), kTestingRatio),
296         }) {
297             double part_size;
298             error += std::modf(modifications_size * ratio, &part_size);
299             if (AreEqual(error, 1, kEpsilon)) {
300                 --error;
301                 ++part_size;
302             }
303
304             auto end = begin + part_size;
305             dataset.reserve(dataset.capacity() + part_size);
306             dataset.insert(dataset.end(), begin, end);
307             begin = std::move(end);
308         }

```



```

309     }
310
311     for (auto&& [begin, end] :
312         {std::make_pair(training.begin(), training.end()),
313          std::make_pair(validation.begin(), validation.end()),
314          std::make_pair(testing.begin(), testing.end())}) {
315         std::shuffle(begin, end, generator);
316     }
317 }
318
319 DataSupplier::DataSupplier(const double low_score, const double high_score) {
320     auto label_to_y =
321         std::unordered_map<std::string, std::shared_ptr<const Eigen::VectorXd>>{};
322     label_to_y.reserve(Parametrization::kLabelsNumber);
323     for (std::size_t i = 0; i < Parametrization::kLabelsNumber; ++i) {
324         auto y = Eigen::VectorXd(Parametrization::kLabelsNumber);
325         y.fill(low_score);
326         y(i) = high_score;
327         label_to_y.insert({Parametrization::kLabels[i],
328                             std::make_shared<const Eigen::VectorXd>(std::move(y))});
329     }
330
331     for (auto&& [dataset, symbol_dataset] :
332         {std::make_pair(std::reference_wrapper(training_),
333                          std::reference_wrapper(kParametrization.training)),
334          std::make_pair(std::reference_wrapper(validation_),
335                          std::reference_wrapper(kParametrization.validation)),
336          std::make_pair(std::reference_wrapper(testing_),
337                          std::reference_wrapper(kParametrization.testing))}) {
338         dataset.reserve(symbol_dataset.size());
339         for (auto&& symbol : symbol_dataset) {
340             dataset.push_back(
341                 std::make_shared<const Data>(symbol, label_to_y.at(symbol->label)));
342         }
343     }
344 }
345
346 } // namespace hw1

```

Листинг 7: Файл util.h

```

1  #pragma once
2
3  #include <vector>
4
5  namespace hw1 {
6
7  template <typename T>
8  std::vector<std::vector<T>> GeneratePowerset(const std::vector<T>& set) {
9      const auto set_size = set.size();
10     const auto powerset_size = static_cast<std::size_t>(1 << set_size);
11
12     auto powerset = std::vector<std::vector<T>>{};
13     powerset.reserve(powerset_size);
14
15     for (std::size_t i = 0; i < powerset_size; ++i) {
16         auto subset = std::vector<T>{};
17
18         for (std::size_t j = 0; j < set_size; ++j) {
19             if (i & (1 << j)) {
20                 subset.push_back(set.at(j));
21             }
22         }
23
24         powerset.push_back(std::move(subset));
25     }
26
27     return powerset;
28 }
29
30 } // namespace hw1

```

Листинг 8: Файл main.cc

```

1  #include <matplot/matplot.h>
2
3  #include <memory>
4
5  #include "activation_function.h"
6  #include "cost_function.h"

```

```

7  #include "data_supplier.h"
8  #include "perceptron.h"
9
10 namespace {
11
12 void RunLinearMse() {
13     constexpr static auto kCfg = nn::Config{
14         .epochs = 200,
15         .mini_batch_size = 10,
16         .eta = 0.1,
17         .monitor_training_cost = true,
18         .monitor_training_accuracy = true,
19         .monitor_testing_cost = true,
20         .monitor_testing_accuracy = true,
21     };
22
23     const auto data_supplier = hw1::DataSupplier(0.0, 1.0);
24     const auto training = data_supplier.GetTrainingData();
25     const auto testing = data_supplier.GetTestingData();
26
27     auto cost_function = std::make_unique<nn::MSE>();
28     auto activation_functions =
29         std::vector<std::unique_ptr<nn::IActivationFunction>>{};
30     activation_functions.push_back(std::make_unique<nn::Linear>());
31     const auto layers_sizes =
32         std::vector<std::size_t>{hw1::kScanSize, hw1::kClassesCount};
33
34     auto perceptron = nn::Perceptron(
35         std::move(cost_function), std::move(activation_functions), layers_sizes);
36     const auto metrics =
37         perceptron.StochasticGradientSearch(training, testing, kCfg);
38
39     const auto x = matplotlib::linspace(0, kCfg.epochs);
40     matplotlib::plot(x, metrics.training_cost, x, metrics.testing_cost);
41     matplotlib::title("Linear + MSE training, testing cost");
42     matplotlib::show();
43
44     matplotlib::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
45     matplotlib::title("Linear + MSE training, testing accuracy");
46     matplotlib::show();

```

```

47 }
48
49 void RunReluMse() {
50     constexpr static auto kCfg = nn::Config{
51         .epochs = 200,
52         .mini_batch_size = 10,
53         .eta = 0.1,
54         .monitor_training_cost = true,
55         .monitor_training_accuracy = true,
56         .monitor_testing_cost = true,
57         .monitor_testing_accuracy = true,
58     };
59
60     const auto data_supplier = hw1::DataSupplier(0.0, 1.0);
61     const auto training = data_supplier.GetTrainingData();
62     const auto testing = data_supplier.GetTestingData();
63
64     auto cost_function = std::make_unique<nn::MSE>();
65     auto activation_functions =
66         std::vector<std::unique_ptr<nn::IActivationFunction>>{};
67     activation_functions.push_back(std::make_unique<nn::ReLU>());
68     const auto layers_sizes =
69         std::vector<std::size_t>{hw1::kScanSize, hw1::kClassesCount};
70
71     auto perceptron = nn::Perceptron(
72         std::move(cost_function), std::move(activation_functions), layers_sizes);
73     const auto metrics =
74         perceptron.StochasticGradientSearch(training, testing, kCfg);
75
76     const auto x = matplotlib::linspace(0, kCfg.epochs);
77     matplotlib::plot(x, metrics.training_cost, x, metrics.testing_cost);
78     matplotlib::title("ReLU + MSE training, testing cost");
79     matplotlib::show();
80
81     matplotlib::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
82     matplotlib::title("ReLU + MSE training, testing accuracy");
83     matplotlib::show();
84 }
85
86 void RunSigmoidMse() {

```

```

87 constexpr static auto kCfg = nn::Config{
88     .epochs = 200,
89     .mini_batch_size = 10,
90     .eta = 5,
91     .monitor_training_cost = true,
92     .monitor_training_accuracy = true,
93     .monitor_testing_cost = true,
94     .monitor_testing_accuracy = true,
95 };
96
97 const auto data_supplier = hw1::DataSupplier(0.0, 1.0);
98 const auto training = data_supplier.GetTrainingData();
99 const auto testing = data_supplier.GetTestingData();
100
101 auto cost_function = std::make_unique<nn::MSE>();
102 auto activation_functions =
103     std::vector<std::unique_ptr<nn::IActivationFunction>>{};
104 activation_functions.push_back(std::make_unique<nn::Sigmoid>());
105 const auto layers_sizes =
106     std::vector<std::size_t>{hw1::kScanSize, hw1::kClassesCount};
107
108 auto perceptron = nn::Perceptron(
109     std::move(cost_function), std::move(activation_functions), layers_sizes);
110 const auto metrics =
111     perceptron.StochasticGradientSearch(training, testing, kCfg);
112
113 const auto x = matplotlib::linspace(0, kCfg.epochs);
114 matplotlib::plot(x, metrics.training_cost, x, metrics.testing_cost);
115 matplotlib::title("Sigmoid + MSE training, testing cost");
116 matplotlib::show();
117
118 matplotlib::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
119 matplotlib::title("Sigmoid + MSE training, testing accuracy");
120 matplotlib::show();
121 }
122
123 void RunTanhMse() {
124     constexpr static auto kCfg = nn::Config{
125         .epochs = 200,
126         .mini_batch_size = 10,

```

```

127     .eta = 0.5,
128     .monitor_training_cost = true,
129     .monitor_training_accuracy = true,
130     .monitor_testing_cost = true,
131     .monitor_testing_accuracy = true,
132 };
133
134 const auto data_supplier = hw1::DataSupplier(-1.0, 1.0);
135 const auto training = data_supplier.GetTrainingData();
136 const auto testing = data_supplier.GetTestingData();
137
138 auto cost_function = std::make_unique<nn::MSE>();
139 auto activation_functions =
140     std::vector<std::unique_ptr<nn::IActivationFunction>>{};
141 activation_functions.push_back(std::make_unique<nn::Tanh>());
142 const auto layers_sizes =
143     std::vector<std::size_t>{hw1::kScanSize, hw1::kClassesCount};
144
145 auto perceptron = nn::Perceptron(
146     std::move(cost_function), std::move(activation_functions), layers_sizes);
147 const auto metrics =
148     perceptron.StochasticGradientSearch(training, testing, kCfg);
149
150 const auto x = matplotlib::linspace(0, kCfg.epochs);
151 matplotlib::plot(x, metrics.training_cost, x, metrics.testing_cost);
152 matplotlib::title("Tanh + MSE training, testing cost");
153 matplotlib::show();
154
155 matplotlib::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
156 matplotlib::title("Tanh + MSE training, testing accuracy");
157 matplotlib::show();
158 }
159
160 void RunSoftmaxCrossEntropy() {
161     constexpr static auto kCfg = nn::Config{
162         .epochs = 200,
163         .mini_batch_size = 10,
164         .eta = 10,
165         .monitor_training_cost = true,
166         .monitor_training_accuracy = true,

```

```

167     .monitor_testing_cost = true,
168     .monitor_testing_accuracy = true,
169 };
170
171 const auto data_supplier = hw1::DataSupplier(0.0, 1.0);
172 const auto training = data_supplier.GetTrainingData();
173 const auto testing = data_supplier.GetTestingData();
174
175 auto cost_function = std::make_unique<nn::CrossEntropy>();
176 auto activation_functions =
177     std::vector<std::unique_ptr<nn::IActivationFunction>>{};
178 activation_functions.push_back(std::make_unique<nn::Softmax>());
179 const auto layers_sizes =
180     std::vector<std::size_t>{hw1::kScanSize, hw1::kClassesCount};
181
182 auto perceptron = nn::Perceptron(
183     std::move(cost_function), std::move(activation_functions), layers_sizes);
184 const auto metrics =
185     perceptron.StochasticGradientSearch(training, testing, kCfg);
186
187 const auto x = matplotlib::linspace(0, kCfg.epochs);
188 matplotlib::plot(x, metrics.training_cost, x, metrics.testing_cost);
189 matplotlib::title("Softmax + Cross-entropy training, testing cost");
190 matplotlib::show();
191
192 matplotlib::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
193 matplotlib::title("Softmax + Cross-entropy training, testing accuracy");
194 matplotlib::show();
195 }
196
197 } // namespace
198
199 int main(int argc, char* argv[]) {
200     RunLinearMse();
201     RunReluMse();
202     RunSigmoidMse();
203     RunTanhMse();
204     RunSoftmaxCrossEntropy();
205 }

```

## 4 Результаты экспериментов

При проведении экспериментов некоторые гиперпараметры были фиксированными: количество эпох — 200, размер пакета данных (mini-batch), используемого при обучении, — 10. Коэффициент обучения (learning rate) подбирался индивидуально для каждой конфигурации персептрона.

Количество данных для обучения составляет 160, валидационных данных — 60, тестовых данных — 100. Вместе с обозначенными в условии функциями активации (линейная, ReLU, сигмоида, гиперболический тангенс) используется функция ошибки MSE.

### 4.1 Линейная функция активации

Линейную функцию активации нельзя использовать с алгоритмом обратного распространения ошибки, поскольку её производная — константа и не зависит от входных данных. Результат в этом случае получается бессмысленным.

### 4.2 Функция ReLU

Коэффициент обучения — 0.1.

На рисунке 1 изображено изменение функции ошибки за период обучения персептрона. Здесь и далее график синей функции соответствует данным для обучения, график оранжевой функции — тестовым данным. В результате, ошибка на данных для обучения составляет 0.400494, ошибка на тестовых данных — 0.40741.

На рисунке 2 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет  $\frac{40}{160}$ , на тестовых данных —  $\frac{25}{100}$ .

### 4.3 Функция сигмоиды

Коэффициент обучения — 5.



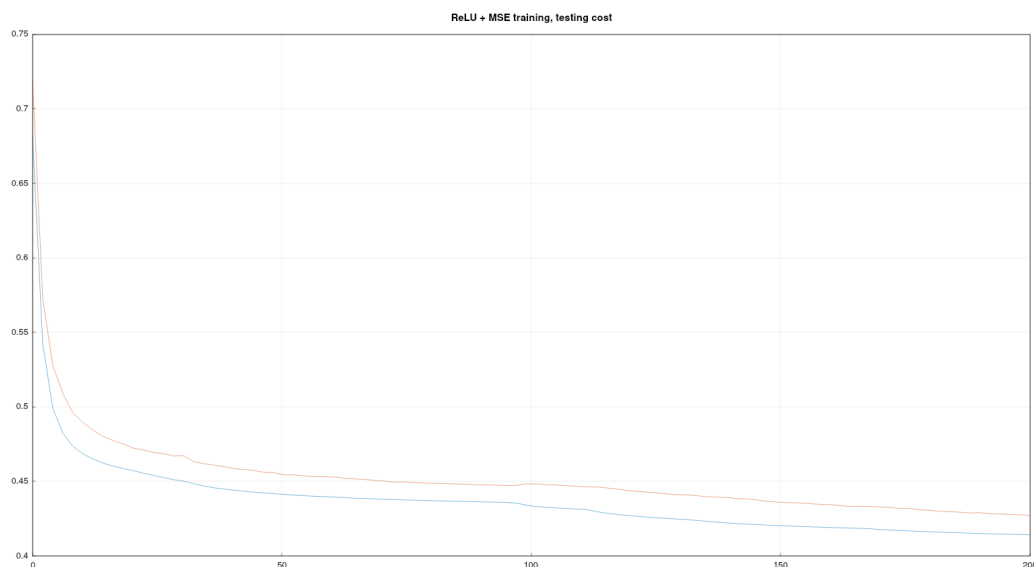


Рис. 1

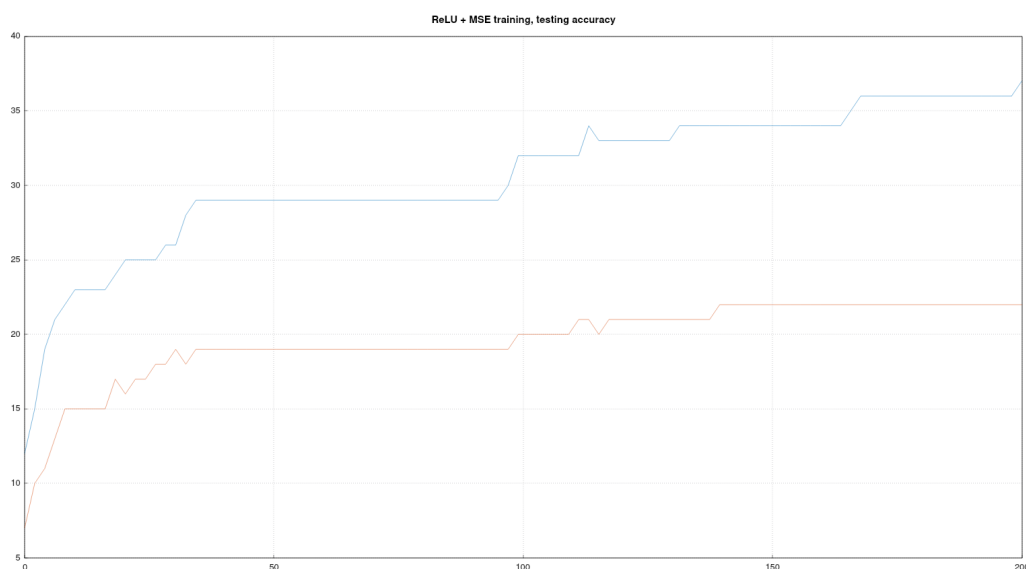


Рис. 2

На рисунке 3 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.028494, ошибка на тестовых данных — 0.03111.

На рисунке 4 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет  $\frac{152}{160}$ , на тестовых данных —  $\frac{95}{100}$ .

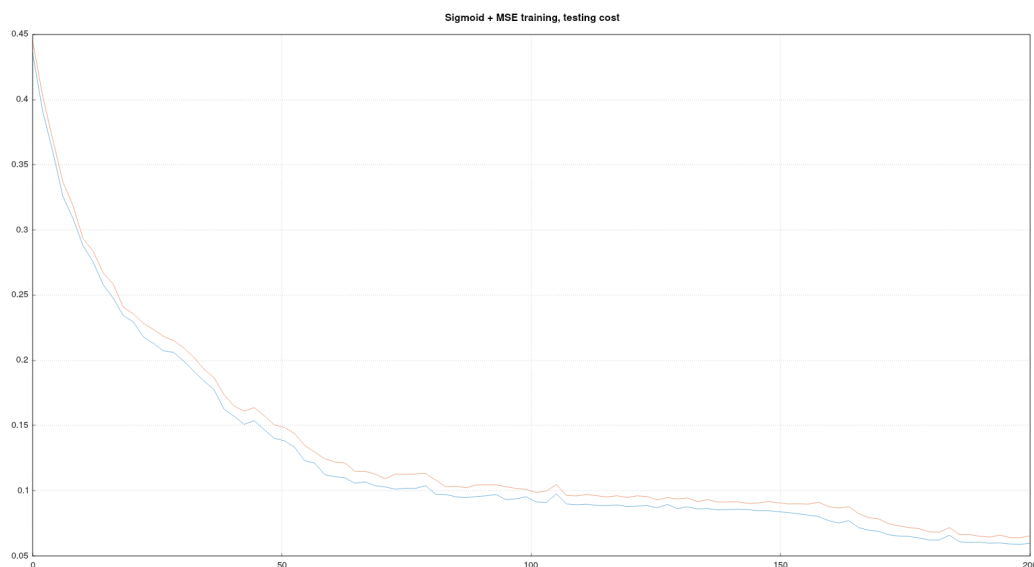


Рис. 3

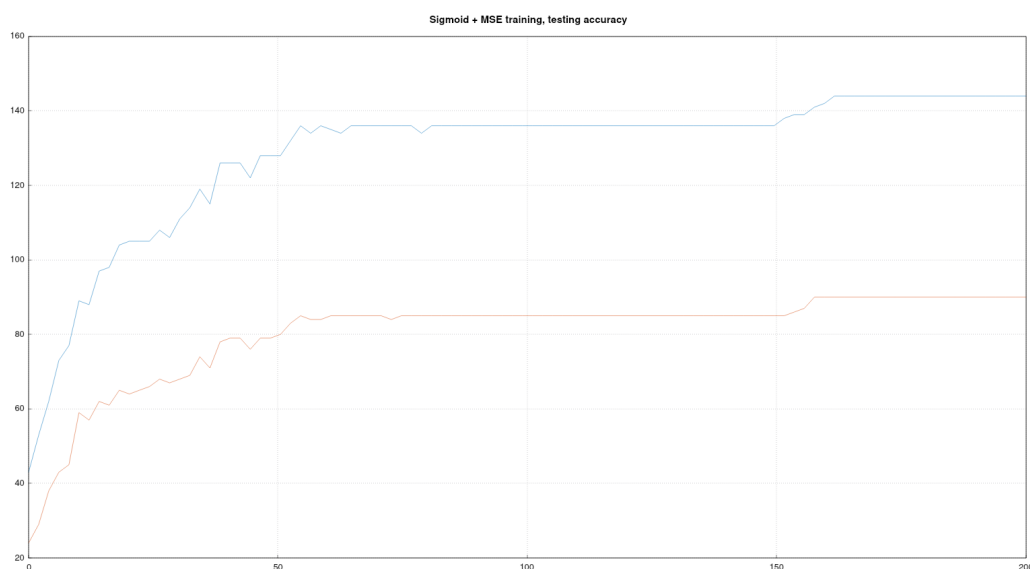


Рис. 4

#### 4.4 Функция гиперболического тангенса

Коэффициент обучения — 0.5.

На рисунке 5 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.70365, ошибка на тестовых данных — 0.710095.

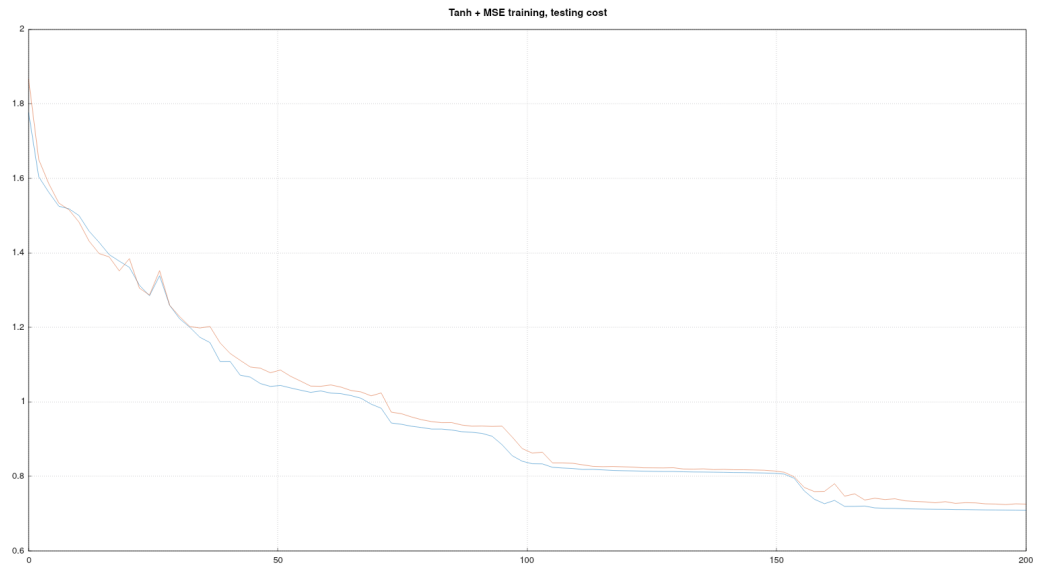


Рис. 5

На рисунке 6 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет  $\frac{104}{160}$ , на тестовых данных —  $\frac{65}{100}$ .

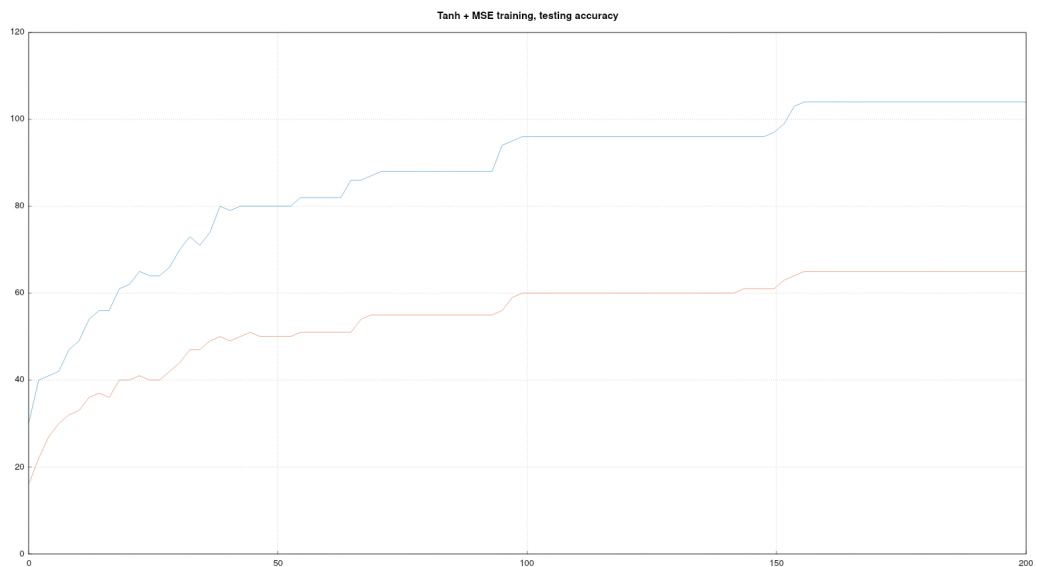


Рис. 6

## 4.5 Функция Softmax

Коэффициент обучения — 10.

Дополнительно произведено обучение персептрона с функцией активации Softmax и кросс-энтропией в качестве функции ошибки.

На рисунке 7 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.000196681, ошибка на тестовых данных — 0.0104362.

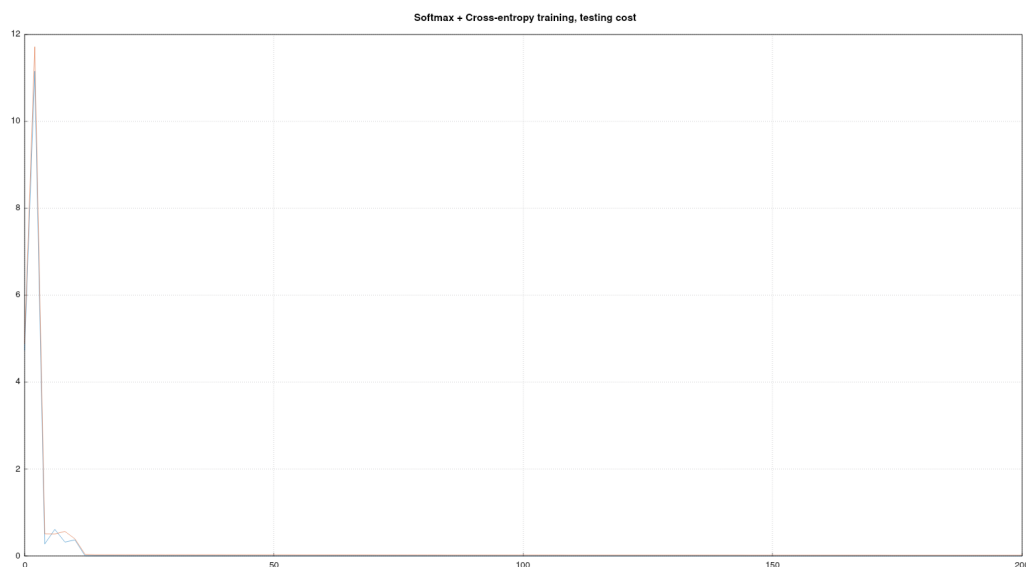


Рис. 7

На рисунке 8 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет  $\frac{160}{160}$ , на тестовых данных —  $\frac{100}{100}$ .

## 5 Вывод

В результате выполнения домашней работы мне удалось реализовать каркас многослойного персептрона с обучением методом стохастического градиентного спуска и вычислением градиентов методом обратного распространения ошибки. Персептрон может иметь произвольное число скрытых слоёв и быть параметризован различными функциями активации и ошибки. Полученный каркас может быть расширен и совершенствован (например, с добавлением техник регуляризации).

По результатам экспериментов для однослойного персептрона видно:

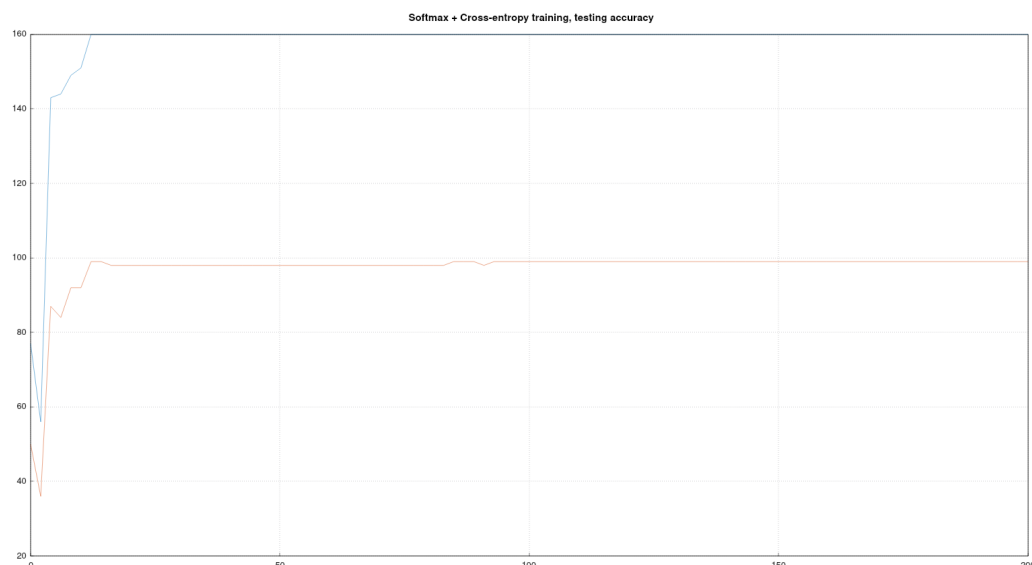


Рис. 8

- линейная функция активации непригодна для использования на выходном слое в рамках поставленной задачи классификации;
- функция ReLU на выходном слое также показывает низкий результат точности в 25% (на тестовых данных) — функция предназначена для использования в *скрытых* слоях персептрона;
- сигмоида показывает хороший результат точности в 95%.
- гиперболический тангенс показывает посредственный результат точности в 65%;
- дополнительно протестированные Softmax + кросс-энтропия показывают отличный результат точности 100%, который достигается всего за 10 эпох обучения.

Мне показался удивительным разрыв в точности между персептронами с сигмоидой и гиперболическим тангенсом. Вероятно, с лучшей настройкой гиперпараметров результат для гиперболического тангенса может быть улучшен.