



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 № 2
по курсу «Теория искусственных нейронных сетей»
«Разработка многослойного персептрона на основе обратного
распространения ошибки FFNN»

Студент группы ИУ9-71Б Афанасьев И.

Преподаватель Каганов Ю. Т.

Москва 2024

1 Цель работы

1. Изучение многослойного персептрона, исследование его работы на основе использования градиентного метода оптимизации и различных целевых функций.

2 Постановка задачи

1. Реализовать на языке высокого уровня многослойный персептрон и проверить его работоспособность на примере данных, выбранных из MNIST dataset.
2. Исследовать работу персептрона на основе использования различных целевых функций (среднеквадратичная ошибка, перекрестная энтропия, дивергенция Кульбака-Лейблера).
3. Провести исследование эффективности работы многослойного персептрона при изменении гиперпараметров (количества нейронов и количества слоев).
4. Подготовить отчет с распечаткой текста программы, графиками результатов исследования и анализом результатов.

3 Реализация

Каркас для обучения многослойного персептрона на основе стохастического градиентного спуска и обратного распространения ошибки был разработан в домашнем задании №1. Программа написана 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 и его обучения.

В листингах 5 и 6 приводится программный код формирования данных для обучения, валидации и тестирования на основе исходных файлов датасета MNIST в формате CSV.

В листинге 7 приводится main-файл с различными конфигурациями 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 <memory>
4  #include <random>
5
6  // clang-format off
7  #include <Eigen/Dense>
8  // clang-format on
9
10 #include "activation_function.h"
11 #include "cost_function.h"
12
13 namespace nn {
14
15 class IData {
16 public:
17     virtual ~IData() = default;
18
19 public:
20     virtual const Eigen::VectorXd &GetX() const = 0;
21     virtual const Eigen::VectorXd &GetY() const = 0;
22     virtual std::string_view ToString() const = 0;
23 };
24
25 class IDataSupplier {
26 public:
27     virtual ~IDataSupplier() = default;
28
29 public:
30     virtual std::vector<std::shared_ptr<const IData>> GetTrainingData() const = 0;
31     virtual std::vector<std::shared_ptr<const IData>> GetValidationData()
32         const = 0;
33     virtual std::vector<std::shared_ptr<const IData>> GetTestingData() const = 0;
34 };
35
36 struct Config final {
37     std::size_t epochs;
38     std::size_t mini_batch_size;
39     double eta;

```

```

40     bool monitor_training_cost;
41     bool monitor_training_accuracy;
42     bool monitor_testing_cost;
43     bool monitor_testing_accuracy;
44 };
45
46 struct Metric final {
47     std::vector<double> training_cost, training_accuracy;
48     std::vector<double> testing_cost, testing_accuracy;
49 };
50
51 class Perceptron final {
52     std::random_device device_;
53     std::default_random_engine generator_;
54     std::unique_ptr<ICostFunction> cost_function_;
55     std::size_t layers_number_, connections_number_;
56     std::vector<Eigen::MatrixXd> weights_;
57     std::vector<Eigen::VectorXd> biases_;
58     std::vector<std::unique_ptr<IActivationFunction>> activation_functions_;
59
60 public:
61     Perceptron(
62         std::unique_ptr<ICostFunction> &&cost_function,
63         std::vector<std::unique_ptr<IActivationFunction>> &&activation_functions,
64         const std::vector<std::size_t> &layers_sizes);
65
66     Eigen::VectorXd Feedforward(const Eigen::VectorXd &x) const;
67
68     Metric StochasticGradientSearch(
69         const std::vector<std::shared_ptr<const IData>> &training,
70         const std::vector<std::shared_ptr<const IData>> &testing,
71         const Config &cfg);
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 <cassert>
2  #include <iostream>
3  #include <iterator>
4  #include <sstream>
5
6  // clang-format off
7  #include <spdlog/spdlog.h>
8  // clang-format on
9
10 #include "perceptron.h"
11
12 namespace nn {
13
14 Perceptron::Perceptron(
15     std::unique_ptr<ICostFunction> &&cost_function,
16     std::vector<std::unique_ptr<IActivationFunction>> &&activation_functions,
17     const std::vector<std::size_t> &layers_sizes)
18     : generator_(device_()),

```

```

19     cost_function_(std::move(cost_function)),
20     layers_number_(layers_sizes.size()),
21     connections_number_(layers_number_ - 1),
22     activation_functions_(std::move(activation_functions)) {
23 if (layers_number_ < 2) {
24     throw std::runtime_error("Perceptron must have at least two layers");
25 }
26
27 if (activation_functions_.size() != connections_number_) {
28     throw std::runtime_error(
29         "Activation functions number must be equal to layers number minus one");
30 }
31
32 weights_.reserve(connections_number_);
33 biases_.reserve(connections_number_);
34 for (std::size_t i = 0; i < connections_number_; ++i) {
35     weights_.push_back(
36         Eigen::MatrixXd::Random(layers_sizes[i + 1], layers_sizes[i]));
37     biases_.push_back(Eigen::VectorXd::Random(layers_sizes[i + 1]));
38 }
39 }
40
41 Eigen::VectorXd Perceptron::Feedforward(const Eigen::VectorXd &x) const {
42     auto activation = x;
43     for (std::size_t i = 0; i < connections_number_; ++i) {
44         activation =
45             activation_functions_[i]->Apply(weights_[i] * activation + biases_[i]);
46     }
47     return activation;
48 }
49
50 Metric Perceptron::StochasticGradientSearch(
51     const std::vector<std::shared_ptr<const IData>> &training,
52     const std::vector<std::shared_ptr<const IData>> &testing,
53     const Config &cfg) {
54     const auto training_size = training.size();
55     const auto whole_mini_batches_number = training_size / cfg.mini_batch_size;
56     const auto remainder_mini_batch_size = training_size % cfg.mini_batch_size;
57
58     auto training_shuffled = std::vector(training.begin(), training.end());

```

```

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

```

```

99     for (std::size_t i = 0; i < connections_number_; ++i) {
100         nabla_weights[i] += nabla_weights_part[i];
101         nabla_biases[i] += nabla_biases_part[i];
102     }
103 }
104
105 const auto learning_rate = eta / mini_batch_size;
106 for (std::size_t i = 0; i < connections_number_; ++i) {
107     weights_[i] -= learning_rate * nabla_weights[i];
108     biases_[i] -= learning_rate * nabla_biases[i];
109 }
110 }
111
112 std::pair<std::vector<Eigen::MatrixXd>, std::vector<Eigen::VectorXd>>
113 Perceptron::Backpropagation(const Eigen::VectorXd &x,
114                             const Eigen::VectorXd &y) {
115     const auto [zs, activations] = FeedforwardDetailed(x);
116     assert(zs.size() == connections_number_);
117     assert(activations.size() == layers_number_);
118
119     auto delta = static_cast<Eigen::VectorXd>(
120         activation_functions_.back()->Jacobian(zs.back()).transpose() *
121         cost_function_->GradientWrtActivations(y, activations.back()));
122
123     auto nabla_weights_reversed = std::vector<Eigen::MatrixXd>{};
124     nabla_weights_reversed.reserve(connections_number_);
125     nabla_weights_reversed.push_back(
126         delta * std::prev(activations.cend(), 2)->transpose());
127
128     auto nabla_biases_reversed = std::vector<Eigen::VectorXd>{};
129     nabla_biases_reversed.reserve(connections_number_);
130     nabla_biases_reversed.push_back(delta);
131
132     for (int i = connections_number_ - 2; i >= 0; --i) {
133         delta = (weights_[i + 1] * activation_functions_[i]->Jacobian(zs[i]))
134             .transpose() *
135             delta;
136         nabla_weights_reversed.push_back(delta * activations[i].transpose());
137         nabla_biases_reversed.push_back(delta);
138     }

```

```

139
140     return {{std::make_move_iterator(nabla_weights_reversed.rbegin()),
141         std::make_move_iterator(nabla_weights_reversed.rend())},
142         {std::make_move_iterator(nabla_biases_reversed.rbegin()),
143         std::make_move_iterator(nabla_biases_reversed.rend())}};
144 }
145
146 std::pair<std::vector<Eigen::VectorXd>, std::vector<Eigen::VectorXd>>
147 Perceptron::FeedforwardDetailed(const Eigen::VectorXd &x) {
148     std::vector<Eigen::VectorXd> zs, activations;
149     zs.reserve(connections_number_);
150     activations.reserve(layers_number_);
151
152     auto activation = x;
153     for (std::size_t i = 0; i < connections_number_; ++i) {
154         auto z =
155             static_cast<Eigen::VectorXd>(weights_[i] * activation + biases_[i]);
156         activations.push_back(std::move(activation));
157         activation = activation_functions_[i]->Apply(z);
158         zs.push_back(std::move(z));
159     }
160     activations.push_back(std::move(activation));
161
162     return {zs, activations};
163 }
164
165 Metric Perceptron::GetMetric(const Config &param) const {
166     auto metric = Metric{};
167     if (param.monitor_training_cost) {
168         metric.training_cost.reserve(param.epochs);
169     }
170     if (param.monitor_training_accuracy) {
171         metric.training_accuracy.reserve(param.epochs);
172     }
173     if (param.monitor_testing_cost) {
174         metric.testing_cost.reserve(param.epochs);
175     }
176     if (param.monitor_testing_accuracy) {
177         metric.testing_accuracy.reserve(param.epochs);
178     }

```

```

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

```

```

219 Eigen::Index max_activation_expected, max_activation_actual;
220 instance.GetY().maxCoeff(&max_activation_expected);
221 Feedforward(instance.GetX()).maxCoeff(&max_activation_actual);
222 if (max_activation_expected == max_activation_actual) {
223     ++right_predictions;
224 }
225 }
226 return right_predictions;
227 }
228
229 template <typename Iter>
230 double Perceptron::Cost(const Iter begin, const Iter end) const {
231     double cost = 0;
232     std::size_t instances_count = 0;
233     for (auto it = begin; it != end; ++it, ++instances_count) {
234         const IData &instance = **it;
235         const auto activation = Feedforward(instance.GetX());
236         cost += cost_function_ -> Apply(instance.GetY(), activation);
237     }
238     return cost / instances_count;
239 }
240
241 } // 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 hw2 {
10
11     constexpr std::size_t kScanSize = 784;
12     constexpr std::size_t kDigitsNumber = 10;
13
14     struct Data final : nn::IData {

```

```

15 Eigen::VectorXd x, y;
16 std::string label;
17
18 const Eigen::VectorXd &GetX() const override { return x; }
19 const Eigen::VectorXd &GetY() const override { return y; }
20 std::string_view ToString() const override { return label; }
21 };
22
23 class DataSupplier final : public nn::IDataSupplier {
24     std::vector<std::shared_ptr<const nn::IData>> training_, testing_,
25         validation_;
26
27 public:
28     DataSupplier(const std::string &train_path, const std::string &test_path,
29         const double false_score, const double true_score);
30
31     std::vector<std::shared_ptr<const nn::IData>> GetTrainingData()
32         const override {
33             return training_;
34         }
35     std::vector<std::shared_ptr<const nn::IData>> GetValidationData()
36         const override {
37             return validation_;
38         }
39     std::vector<std::shared_ptr<const nn::IData>> GetTestingData()
40         const override {
41             return testing_;
42         }
43 };
44
45 } // namespace hw2

```

Листинг 6: Файл data_supplier.cc

```

1 #include "data_supplier.h"
2
3 #include <spdlog/spdlog.h>
4
5 #include <boost/algorithm/string/classification.hpp>
6 #include <boost/algorithm/string/split.hpp>

```



```

7  #include <cassert>
8  #include <fstream>
9  #include <iterator>
10 #include <stdexcept>
11 #include <string>
12
13 #include "perceptron.h"
14
15 namespace hw2 {
16
17     namespace {
18
19         constexpr std::size_t kColumnsCount = kScanSize + 1;
20
21         std::vector<std::shared_ptr<const nn::IData>> ReadMnistCsv(
22             const std::string &filename, const double false_score,
23             const double true_score) {
24             static constexpr std::size_t kShadesCount = 255;
25
26             auto file = std::ifstream(filename);
27             if (!file.is_open()) {
28                 throw std::runtime_error("Failed to open MNIST CSV file " + filename);
29             }
30
31             auto instances = std::vector<std::shared_ptr<const nn::IData>>{};
32             auto line = std::string{};
33             while (std::getline(file, line)) {
34                 auto result = std::vector<std::string>{};
35                 result.reserve(kColumnsCount);
36                 boost::split(result, line, boost::is_any_of(","));
37
38                 assert(result[0].size() == 1);
39                 assert('0' <= result[0][0] && result[0][0] <= '9');
40
41                 auto data = Data{};
42                 data.label = result[0];
43
44                 data.y = Eigen::VectorXd(kDigitsNumber);
45                 data.y.setConstant(false_score);
46                 data.y(data.label[0] - '0') = true_score;

```

```

47
48     data.x = Eigen::VectorXd(kScanSize);
49     for (std::size_t i = 1; i < kColumnsCount; ++i) {
50         data.x[i - 1] = std::stod(result[i]) / kShadesCount;
51     }
52
53     instances.push_back(std::make_shared<const Data>(std::move(data)));
54 }
55
56 return instances;
57 }
58
59 } // namespace
60
61 DataSupplier::DataSupplier(const std::string &train_path,
62                             const std::string &test_path,
63                             const double false_score, const double true_score) {
64     static constexpr std::size_t kTrainingInitialSize = 60'000;
65     static constexpr std::size_t kValidationSize = 10'000;
66     static constexpr std::size_t kTestingSize = 10'000;
67
68     spdlog::info("Parsing training data...");
69     training_ = ReadMnistCsv(train_path, false_score, true_score);
70     assert(training_.size() == kTrainingInitialSize);
71
72     validation_ = std::vector(
73         std::make_move_iterator(training_.rbegin()),
74         std::make_move_iterator(training_.rbegin() + kValidationSize));
75     training_.resize(kTrainingInitialSize - kValidationSize);
76
77     spdlog::info("Parsing testing data...");
78     testing_ = ReadMnistCsv(test_path, false_score, true_score);
79     assert(testing_.size() == kTestingSize);
80 }
81
82 } // namespace hw2

```

Листинг 7: 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 const std::string kDefaultTestPath = "../data/mnist_test.csv";
13 const std::string kDefaultTrainPath = "../data/mnist_train.csv";
14
15 constexpr std::size_t kHiddenLayerSize = 40;
16 constexpr static auto kCfg = nn::Config{
17     .epochs = 200,
18     .mini_batch_size = 100,
19     .eta = 0.025,
20     .monitor_training_cost = true,
21     .monitor_training_accuracy = true,
22     .monitor_testing_cost = true,
23     .monitor_testing_accuracy = true,
24 };
25
26 void RunLeakyReluSoftmaxMSE() {
27     const auto data_supplier =
28         hw2::DataSupplier(kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
29     const auto training = data_supplier.GetTrainingData();
30     const auto testing = data_supplier.GetTestingData();
31
32     auto cost_function = std::make_unique<nn::CrossEntropy>();
33     auto activation_functions =
34         std::vector<std::unique_ptr<nn::IActivationFunction>>{};
35     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
36     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
37     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
38     activation_functions.push_back(std::make_unique<nn::Softmax>());
39     const auto layers_sizes = std::vector<std::size_t>{

```

```

40     hw2::kScanSize, kHiddenLayerSize, kHiddenLayerSize, kHiddenLayerSize,
41     hw2::kDigitsNumber};
42
43     auto perceptron = nn::Perceptron(
44         std::move(cost_function), std::move(activation_functions), layers_sizes);
45     const auto metrics =
46         perceptron.StochasticGradientSearch(training, testing, kCfg);
47
48     const auto x = matplot::linspace(0, kCfg.epochs);
49     matplot::plot(x, metrics.training_cost, x, metrics.testing_cost);
50     matplot::title("Leaky ReLU, Softmax + MSE training, testing cost");
51     matplot::show();
52
53     matplot::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
54     matplot::title("Leaky ReLU, Softmax + MSE training, testing accuracy");
55     matplot::show();
56 }
57
58 void RunLeakyReluSoftmaxCrossEntropy() {
59     const auto data_supplier =
60         hw2::DataSupplier(kDefaultTrainPath, kDefaultTestPath, 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::CrossEntropy>();
65     auto activation_functions =
66         std::vector<std::unique_ptr<nn::IActivationFunction>>{};
67     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
68     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
69     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
70     activation_functions.push_back(std::make_unique<nn::Softmax>());
71     const auto layers_sizes = std::vector<std::size_t>{
72         hw2::kScanSize, kHiddenLayerSize, kHiddenLayerSize, kHiddenLayerSize,
73         hw2::kDigitsNumber};
74
75     auto perceptron = nn::Perceptron(
76         std::move(cost_function), std::move(activation_functions), layers_sizes);
77     const auto metrics =
78         perceptron.StochasticGradientSearch(training, testing, kCfg);
79

```

```

80     const auto x = matplotlib::linspace(0, kCfg.epochs);
81     matplotlib::plot(x, metrics.training_cost, x, metrics.testing_cost);
82     matplotlib::title(
83         "Leaky ReLU, Softmax + Cross-entropy training, testing cost");
84     matplotlib::show();
85
86     matplotlib::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
87     matplotlib::title(
88         "Leaky ReLU, Softmax + Cross-entropy training, testing accuracy");
89     matplotlib::show();
90 }
91
92 void RunLeakyReluSoftmaxKLDivergence() {
93     const auto data_supplier =
94         hw2::DataSupplier(kDefaultTrainPath, kDefaultTestPath, 10e-6, 1.0);
95     const auto training = data_supplier.GetTrainingData();
96     const auto testing = data_supplier.GetTestingData();
97
98     auto cost_function = std::make_unique<nn::KLDivergence>();
99     auto activation_functions =
100         std::vector<std::unique_ptr<nn::IActivationFunction>>{};
101     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
102     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
103     activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
104     activation_functions.push_back(std::make_unique<nn::Softmax>());
105     const auto layers_sizes = std::vector<std::size_t>{
106         hw2::kScanSize, kHiddenLayerSize, kHiddenLayerSize, kHiddenLayerSize,
107         hw2::kDigitsNumber};
108
109     auto perceptron = nn::Perceptron(
110         std::move(cost_function), std::move(activation_functions), layers_sizes);
111     const auto metrics =
112         perceptron.StochasticGradientSearch(training, testing, kCfg);
113
114     const auto x = matplotlib::linspace(0, kCfg.epochs);
115     matplotlib::plot(x, metrics.training_cost, x, metrics.testing_cost);
116     matplotlib::title(
117         "Leaky ReLU, Softmax + K.-L. Divergence training, testing cost");
118     matplotlib::show();
119

```

```

120  matplotlib::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
121  matplotlib::title(
122      "Leaky ReLU, Softmax + K.-L. Divergence training, testing accuracy");
123  matplotlib::show();
124  }
125
126  } // namespace
127
128  int main(int argc, char *argv[]) {
129      RunLeakyReluSoftmaxMSE();
130      RunLeakyReluSoftmaxCrossEntropy();
131      RunLeakyReluSoftmaxKLDivergence();
132  }

```

4 Результаты экспериментов

На протяжении всех экспериментов значения некоторых гиперпараметров были фиксированными:

- количество эпох: 200;
- размер пакета данных (mini-batch) для стохастического градиентного спуска: 100;
- коэффициент обучения (learning rate): 0.025.

Изменялись значения следующих гиперпараметров:

- количество скрытых слоёв: 1 или 3;
- количество нейронов в скрытых слоях: 20 или 40.

Таким образом, для каждой функции ошибки — среднеквадратичной, перекрёстной энтропии и дивергенции Кульбака-Лейблера — рассматривались четыре конфигурации с разным числом скрытых слоёв и нейронов в них. В скрытых слоях всегда используется функция активации Leaky ReLU, на выходном слое — Softmax.

4.1 Среднеквадратичная ошибка (MSE)

4.1.1 1 скрытый слой, 20 нейронов

На рисунке¹ 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 0.166286, ошибка на тестовых данных — 0.193274.

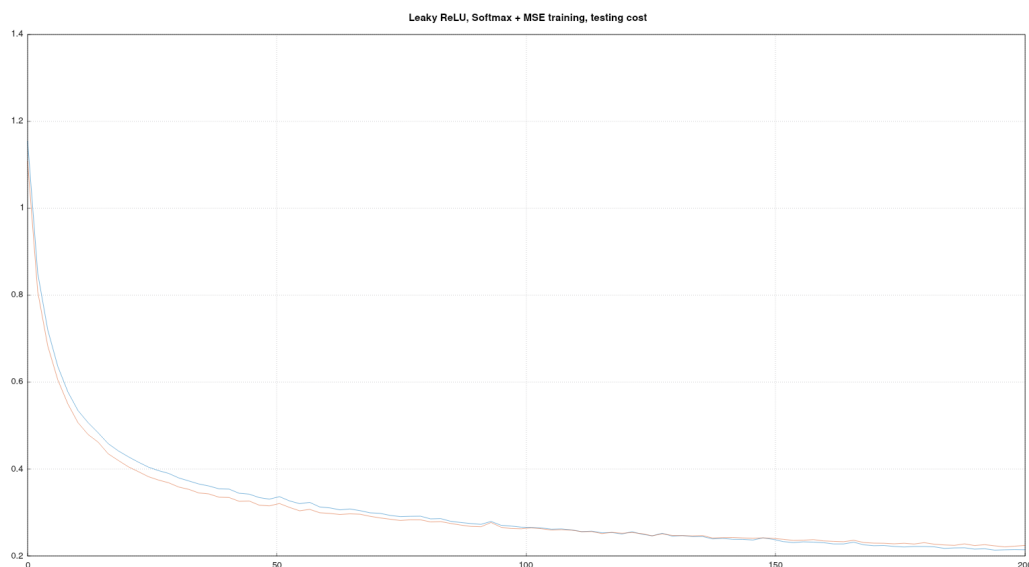


Рис. 1

На рисунке 2 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{47518}{50000} \approx 95,04\%$, на тестовых данных — $\frac{9421}{10000} = 94,21\%$.

4.1.2 1 скрытый слой, 40 нейронов

На рисунке 3 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.123245, ошибка на тестовых данных — 0.161724.

¹ На некоторых рисунках шкала оси *Oy* начинается с 0.2, а не 0.0.

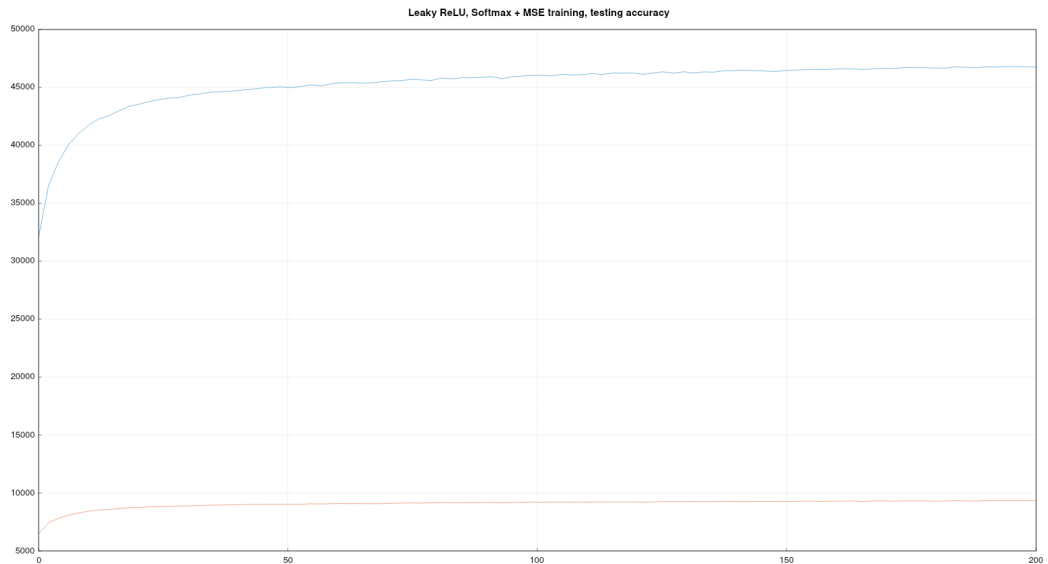


Рис. 2

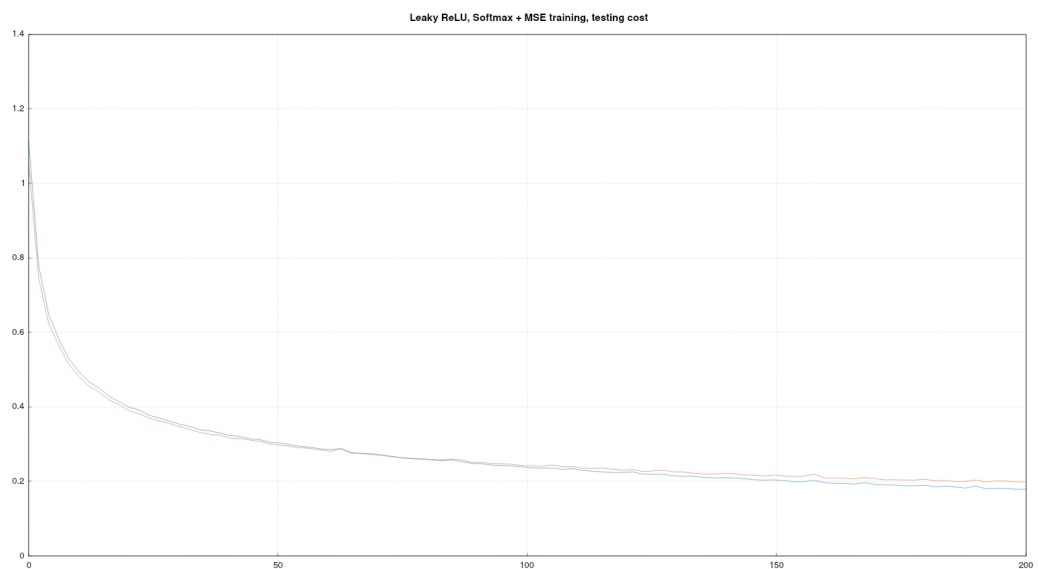


Рис. 3

На рисунке 4 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{48121}{50000} \approx 96,24\%$, на тестовых данных — $\frac{9527}{10000} = 95,27\%$.

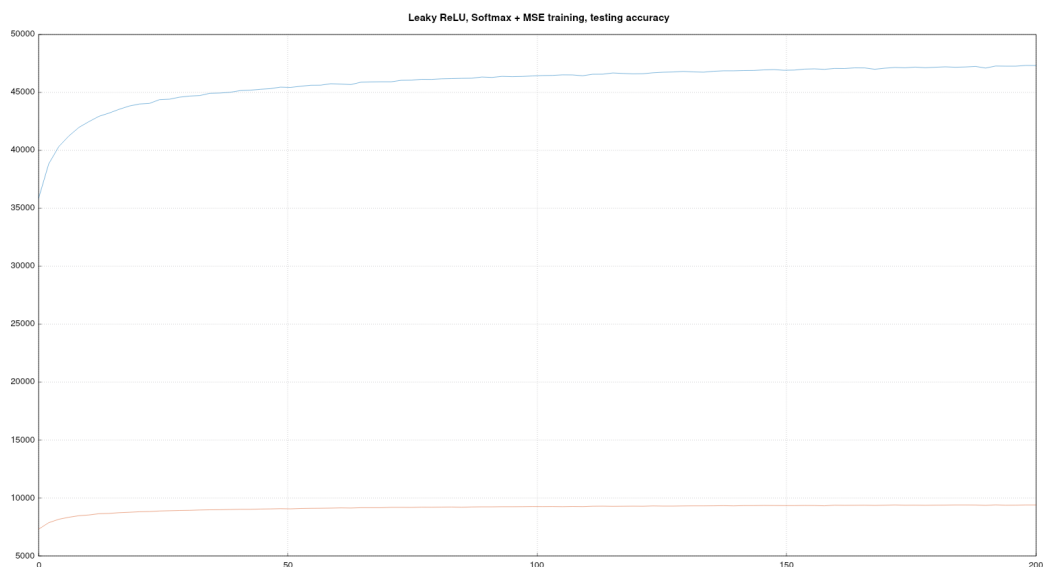


Рис. 4

4.1.3 3 скрытых слоя, 20 нейронов

На рисунке 5 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.163502, ошибка на тестовых данных — 0.206251.

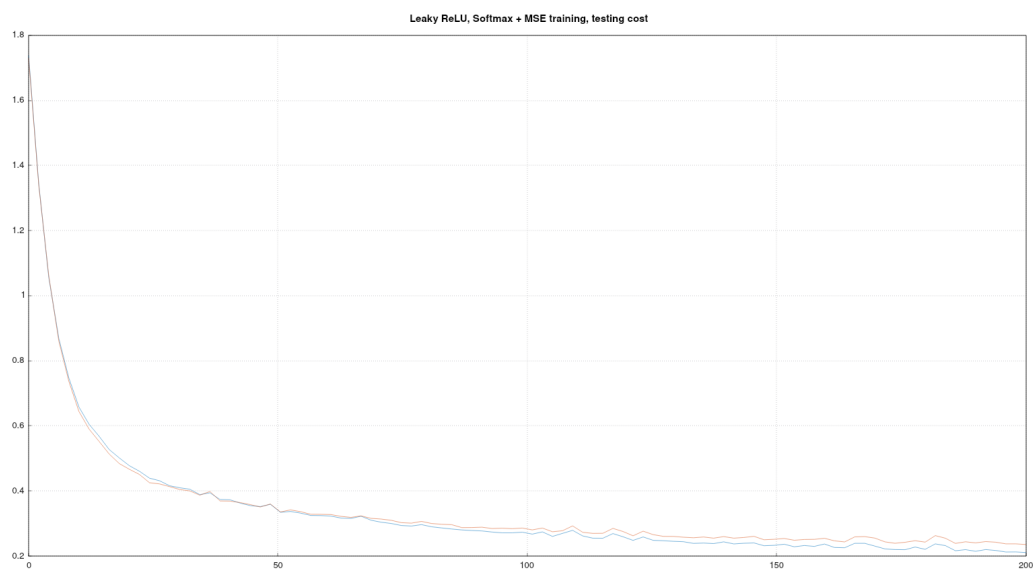


Рис. 5

На рисунке 6 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{47504}{50000} \approx 95,01\%$, на тестовых данных — $\frac{9416}{10000} = 94,16\%$.

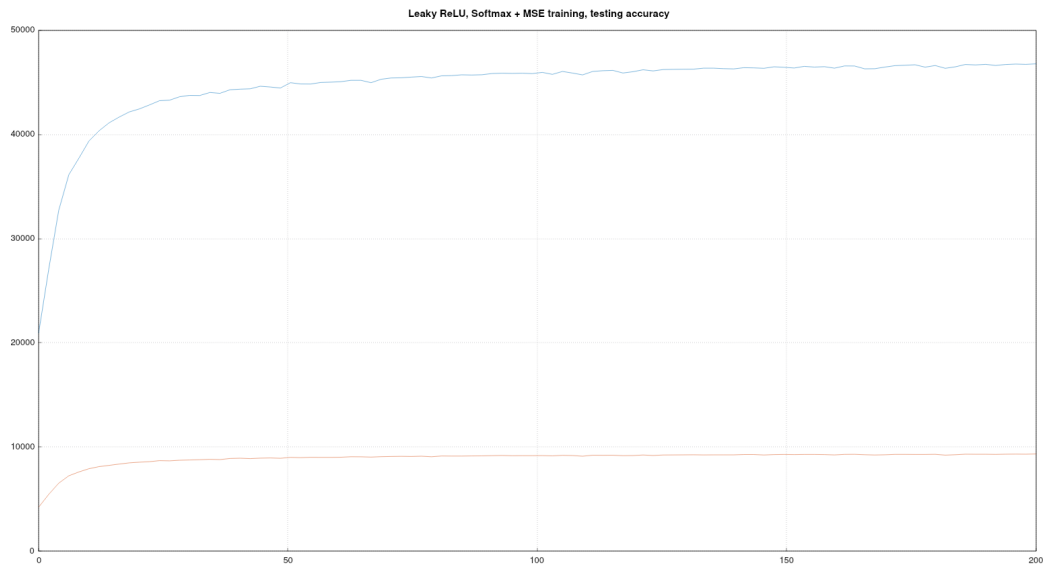


Рис. 6

4.1.4 3 скрытых слоя, 40 нейронов

На рисунке 7 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.105351, ошибка на тестовых данных — 0.190718.

На рисунке 8 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{48422}{50000} \approx 96,84\%$, на тестовых данных — $\frac{9481}{10000} = 94,81\%$.

4.2 Перекрёстная энтропия

4.2.1 1 скрытый слой, 20 нейронов

На рисунке 9 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.192402, ошибка на тестовых данных — 0.217352.

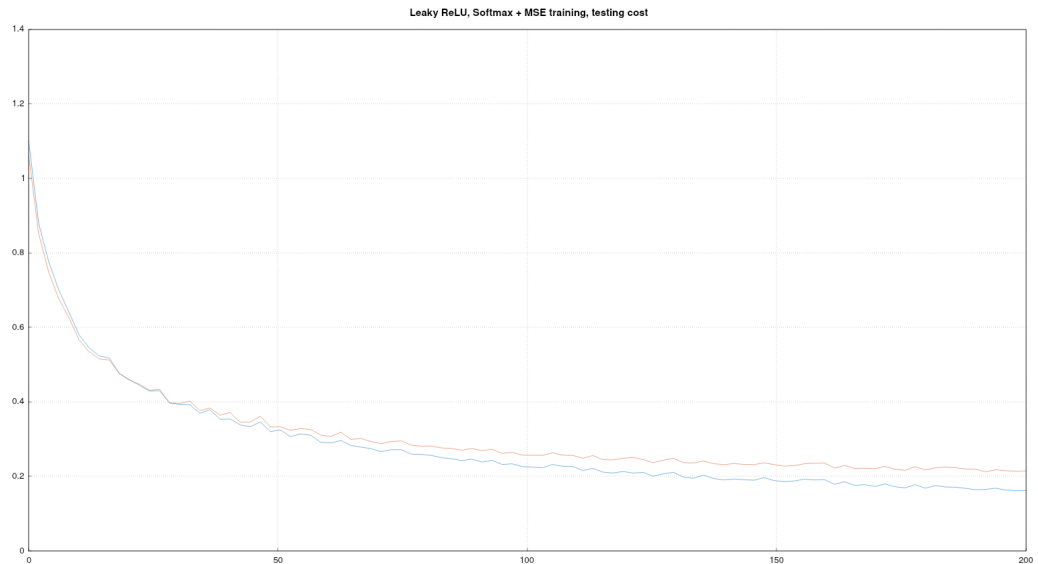


Рис. 7

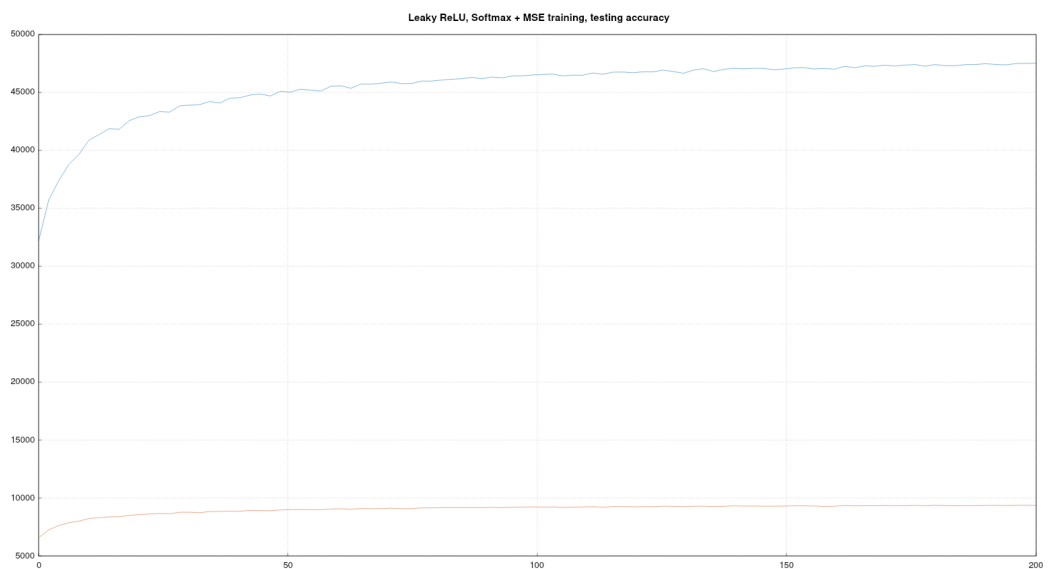


Рис. 8

На рисунке 10 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{47194}{50000} \approx 94,34\%$, на тестовых данных — $\frac{9345}{10000} = 93,45\%$.

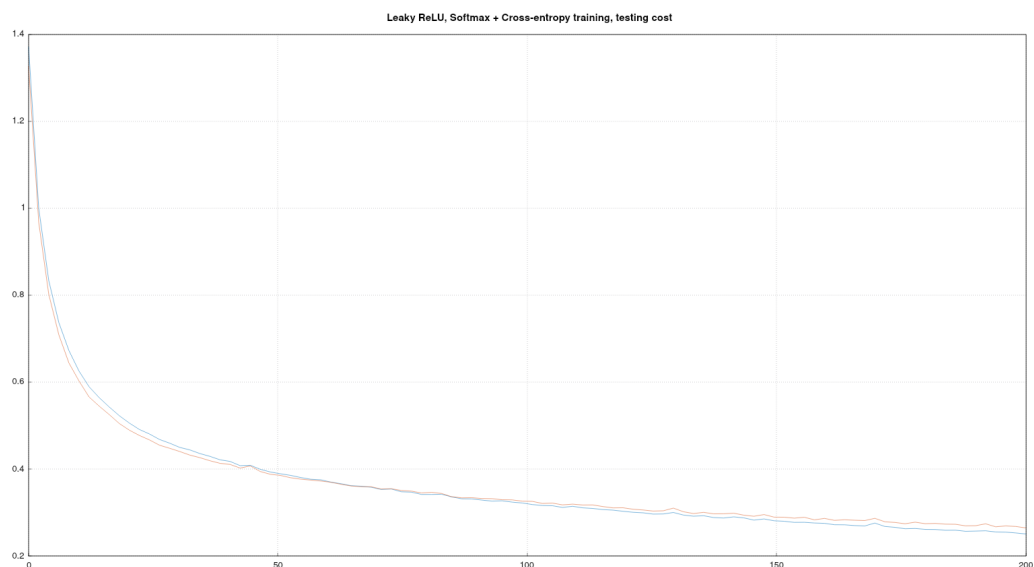


Рис. 9

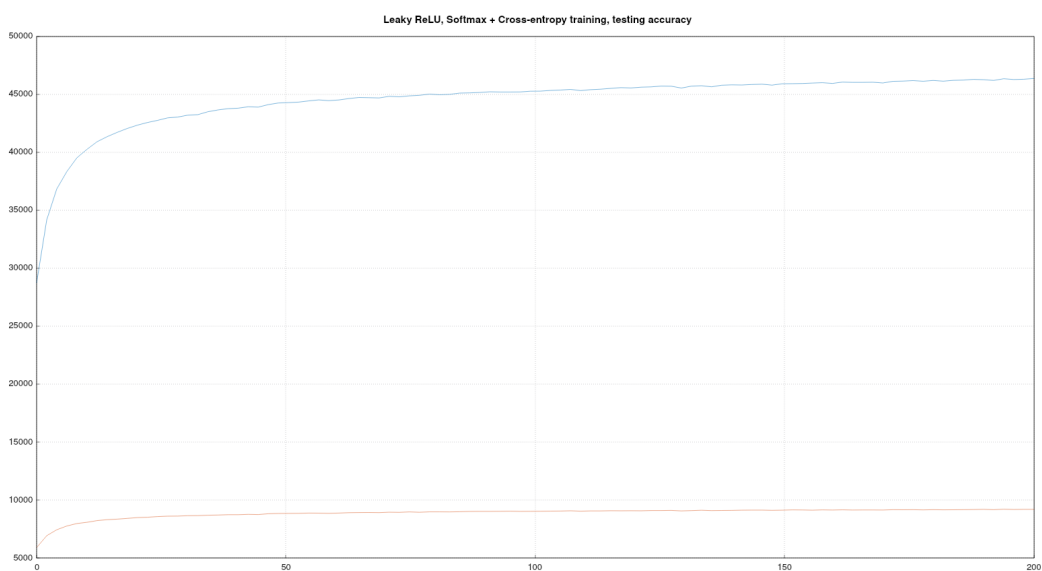


Рис. 10

4.2.2 1 скрытый слой, 40 нейронов

На рисунке 11 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.131022, ошибка на тестовых данных — 0.173189.

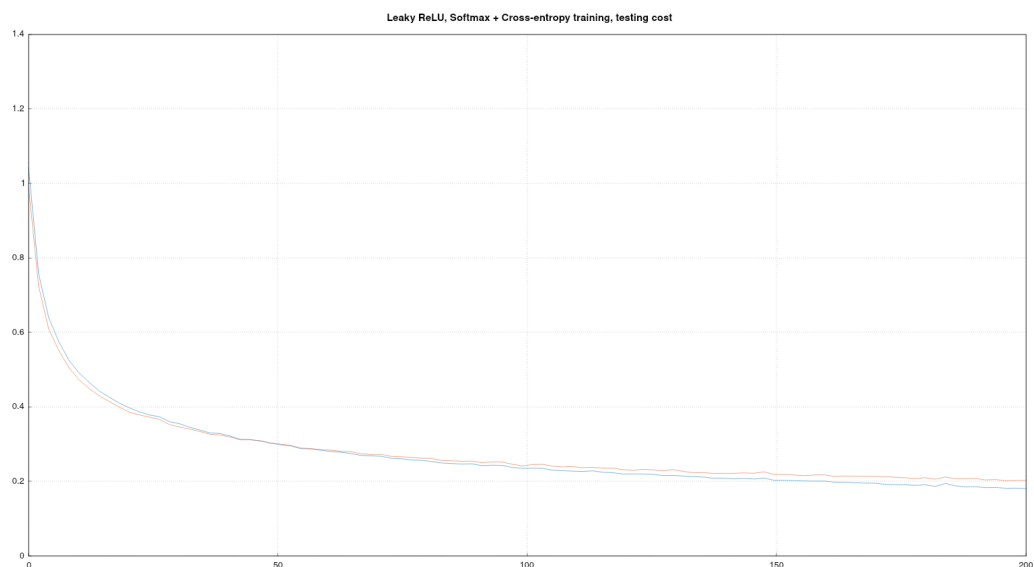


Рис. 11

На рисунке 12 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{48097}{50000} \approx 96,19\%$, на тестовых данных — $\frac{9483}{10000} = 94,83\%$.

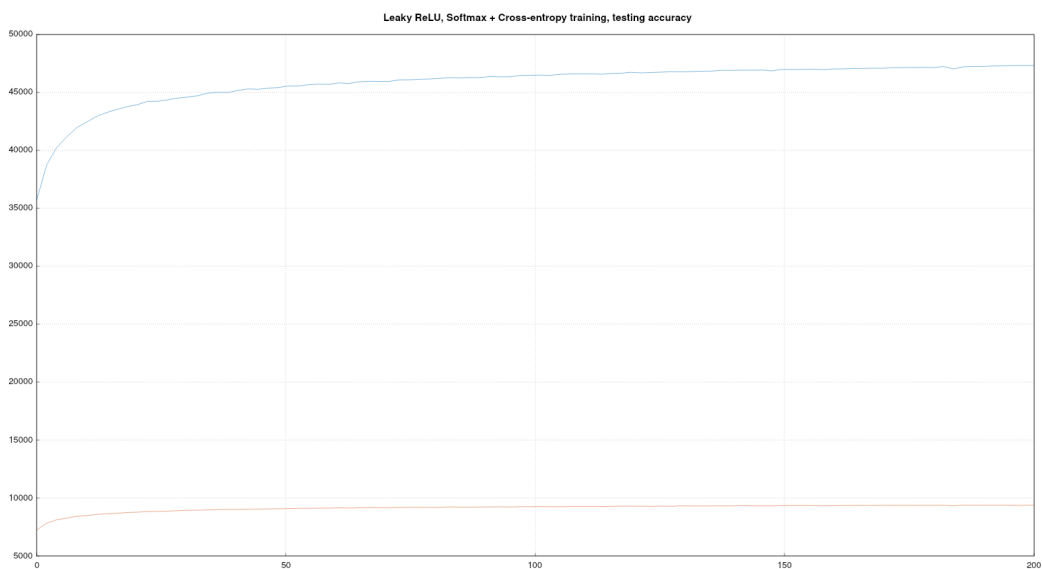


Рис. 12

4.2.3 3 скрытых слоя, 20 нейронов

На рисунке 13 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.157613, ошибка на тестовых данных — 0.216353.

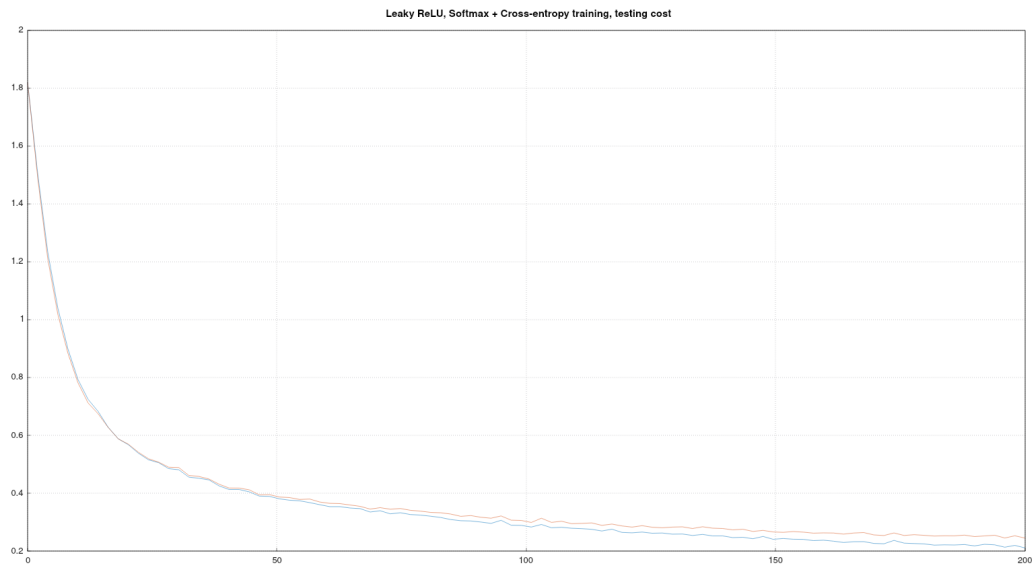


Рис. 13

На рисунке 14 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{47681}{50000} \approx 95,36\%$, на тестовых данных — $\frac{9386}{10000} = 93,86\%$.

4.2.4 3 скрытых слоя, 40 нейронов

На рисунке 15 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.118457, ошибка на тестовых данных — 0.17582.

На рисунке 16 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{48232}{50000} \approx 96,46\%$, на тестовых данных — $\frac{9502}{10000} = 95,02\%$.

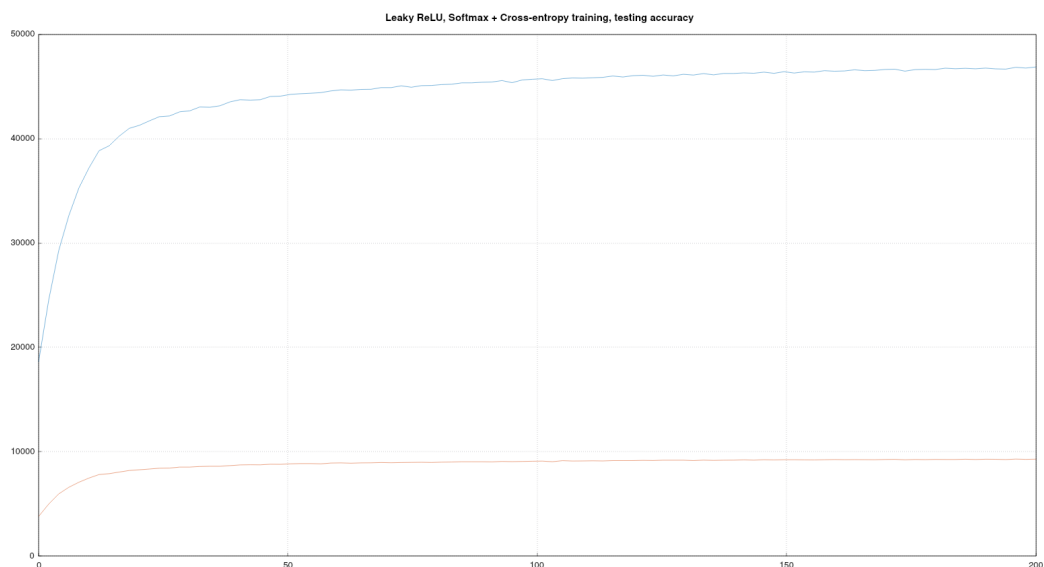


Рис. 14

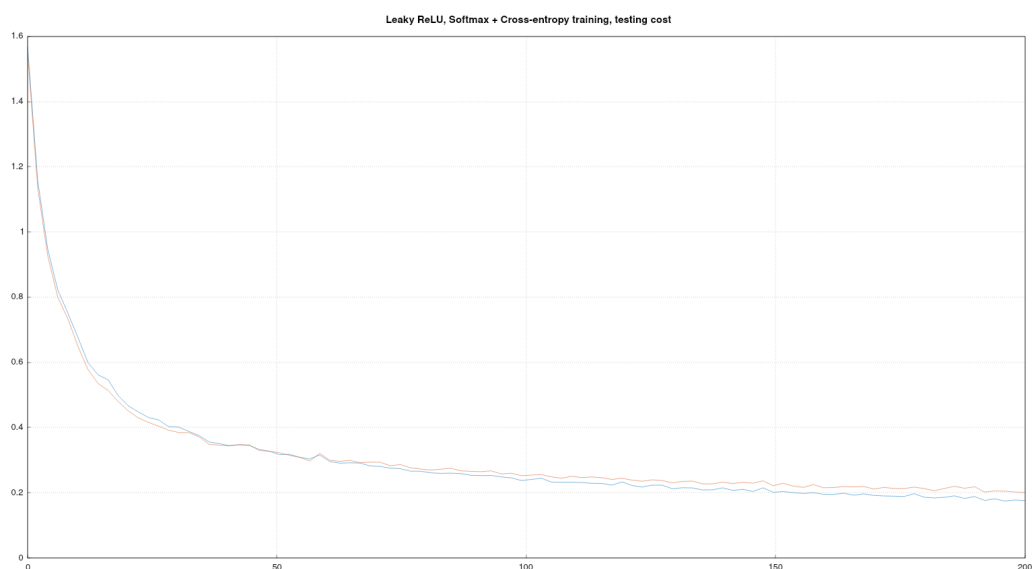


Рис. 15

4.3 Дивергенция Кульбака-Лейблера

4.3.1 1 скрытый слой, 20 нейронов

На рисунке 17 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.17601, ошибка на тестовых данных — 0.201559.

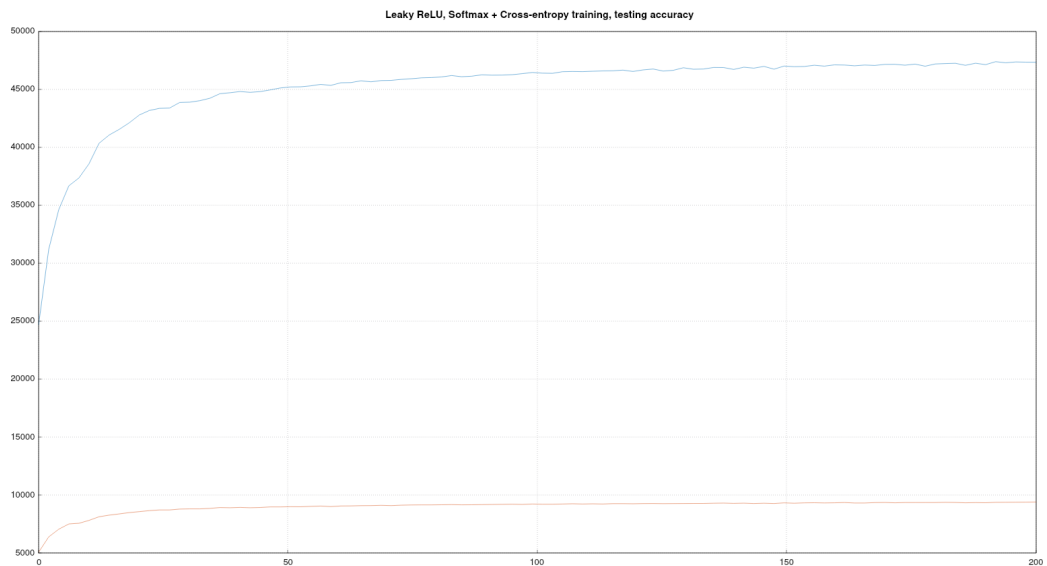


Рис. 16

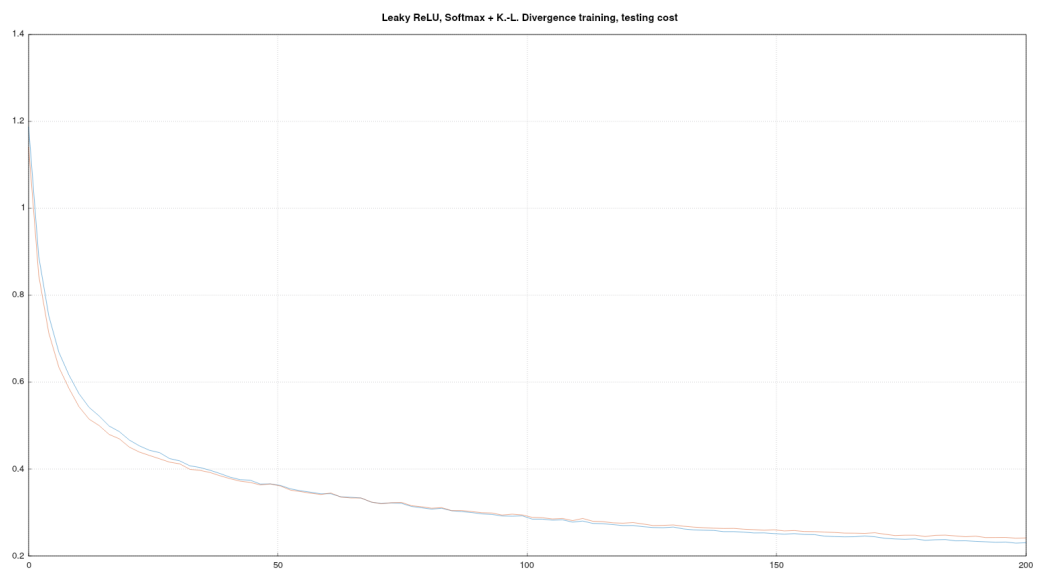


Рис. 17

На рисунке 18 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{47346}{50000} \approx 94,69\%$, на тестовых данных — $\frac{9415}{10000} = 94,15\%$.

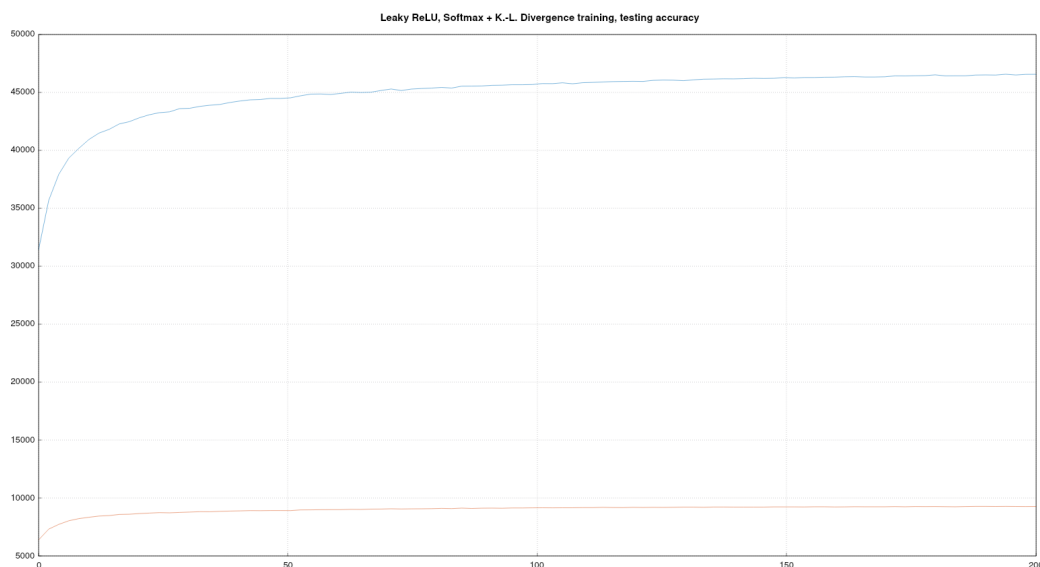


Рис. 18

4.3.2 1 скрытый слой, 40 нейронов

На рисунке 19 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.118821, ошибка на тестовых данных — 0.168346.

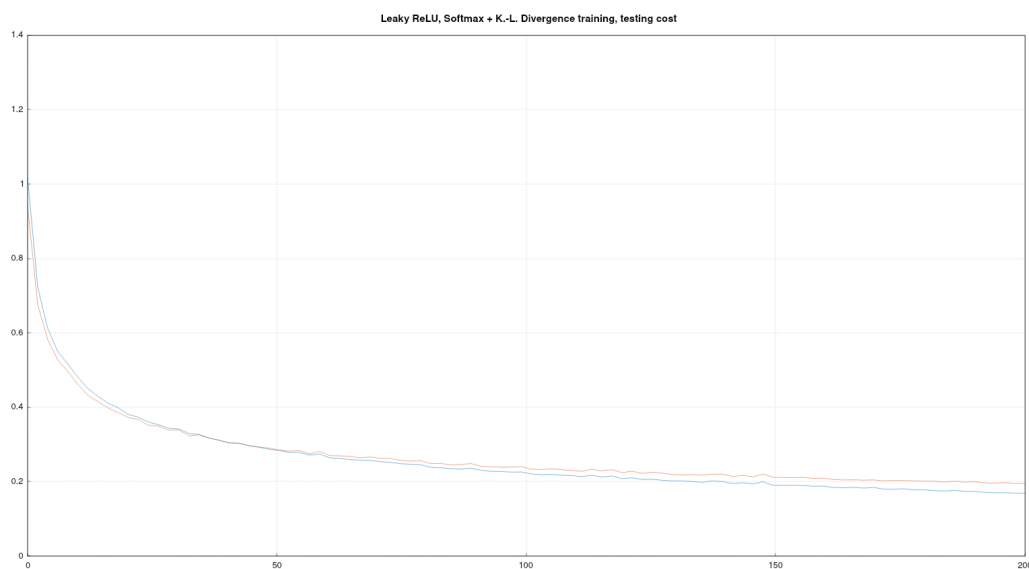


Рис. 19

На рисунке 20 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{48233}{50000} \approx 96,47\%$, на тестовых данных — $\frac{9506}{10000} = 95,06\%$.

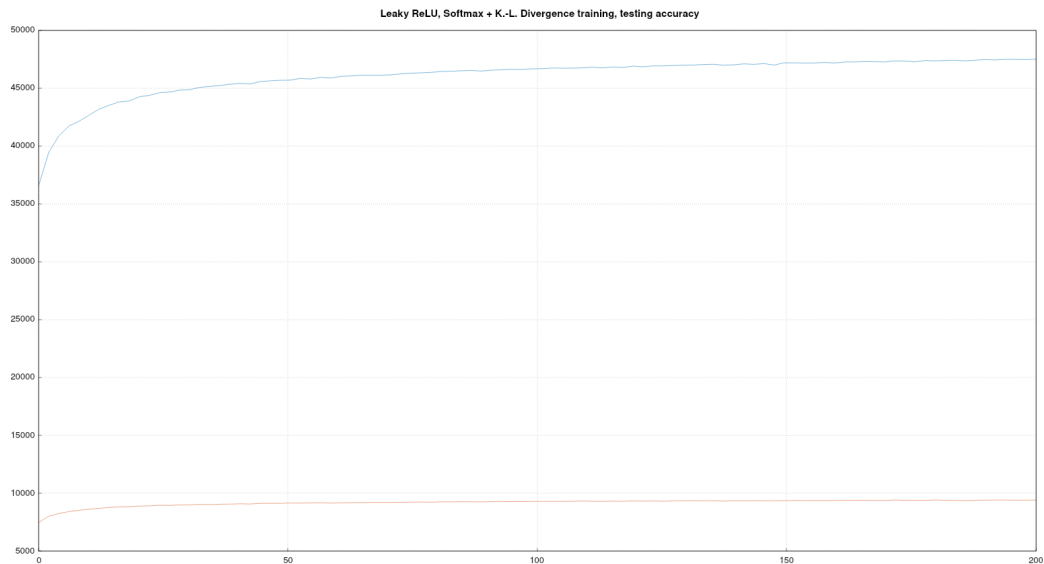


Рис. 20

4.3.3 3 скрытых слоя, 20 нейронов

На рисунке 21 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.163792, ошибка на тестовых данных — 0.213412.

На рисунке 22 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{47487}{50000} \approx 94,97\%$, на тестовых данных — $\frac{9369}{10000} = 93,69\%$.

4.3.4 3 скрытых слоя, 40 нейронов

На рисунке 23 изображено изменение функции ошибки за период обучения персептрона. В результате, ошибка на данных для обучения составляет 0.111865, ошибка на тестовых данных — 0.197848.

На рисунке 24 изображено изменение точности за период обучения персептрона. В результате, точность на данных для обучения составляет $\frac{48315}{50000} \approx 96,63\%$, на тестовых данных — $\frac{9457}{10000} = 94,57\%$.

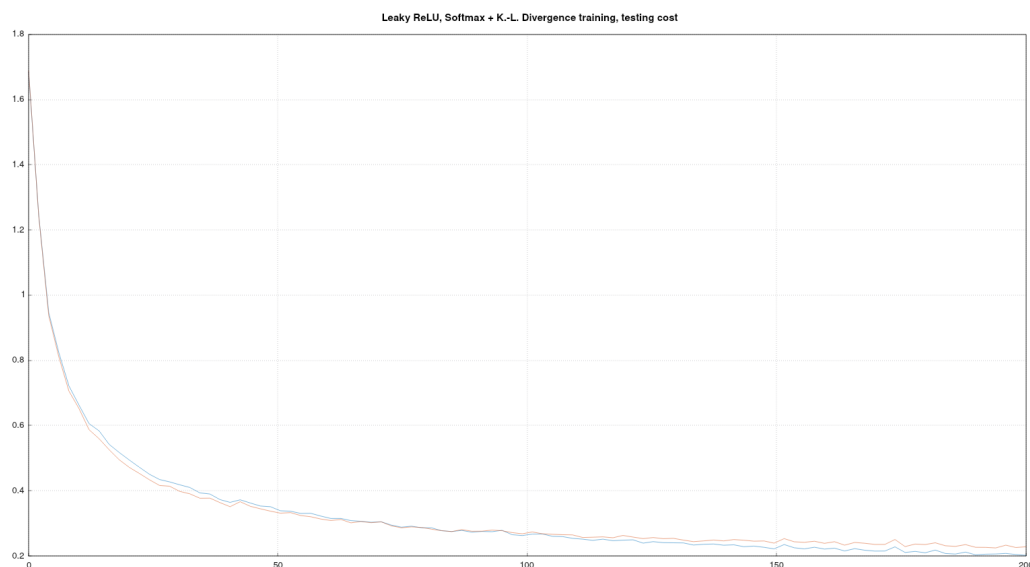


Рис. 21

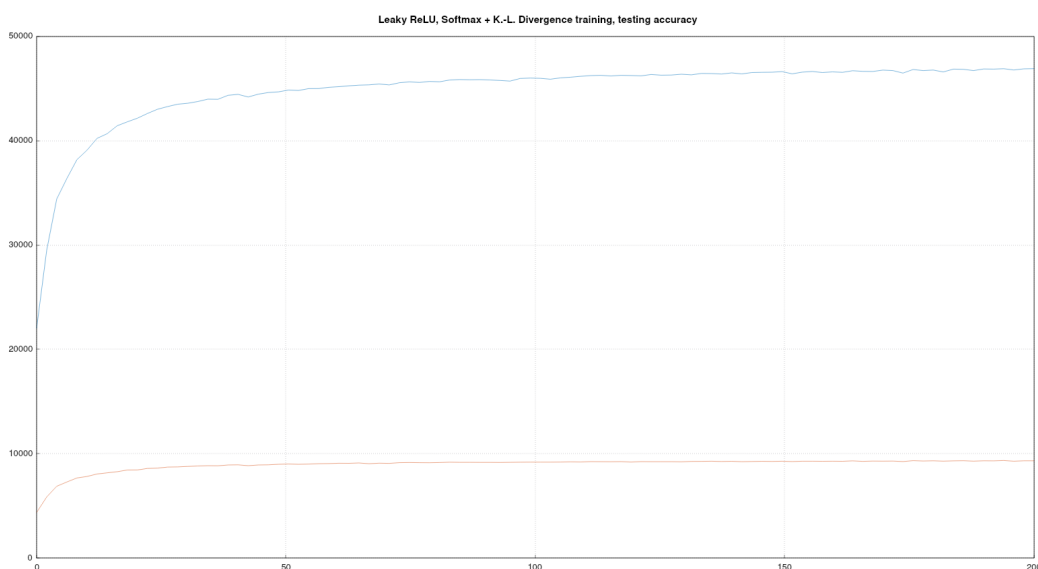


Рис. 22

5 Вывод

Исходя из полученных результатов видно, что наименьшая ошибка на тестовых данных достигается при наличии 1-го скрытого слоя с 40 нейронами: для среднеквадратичной функции, перекрёстной энтропии, дивергенции Кульбака-Лейблера точность на тестовых данных составляет 95,27%, 94,83% и 95,06%

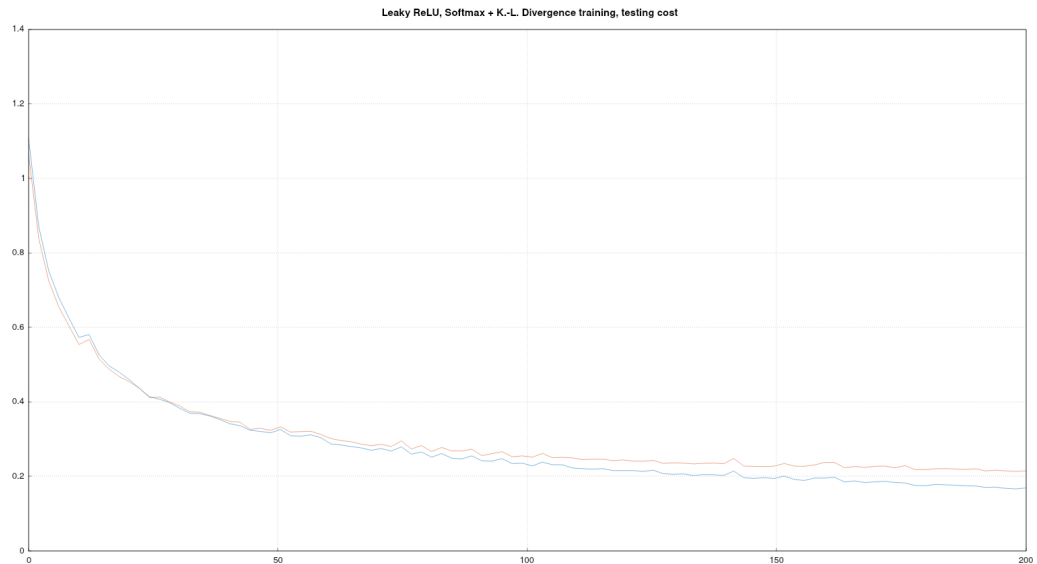


Рис. 23

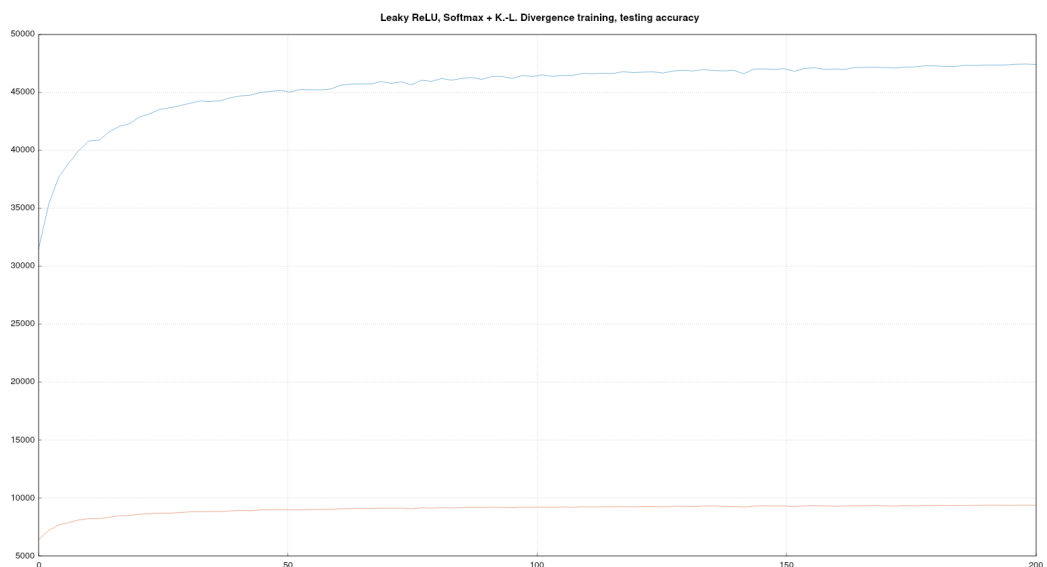


Рис. 24

соответственно. Увеличение числа нейронов в скрытых слоях может способствовать уменьшению ошибки и повышению точности персептрона.

При 3-х скрытых слоях с 40 нейронами ошибка на данных для обучения меньше, чем в предыдущем случае, однако ошибка на тестовых данных выше — это может свидетельствовать о переобучении персептрона; более простая архитектура сети (с 1-м скрытым слоем, а не 3-мя) показывает лучший результат.

Между конфигурациями, где скрытые слои содержат по 20 нейронов, особых различий нет; здесь ошибка в целом выше, чем в предыдущих двух случаях.