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

```
#pragma once

#include <Eigen/Dense>

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 perceptron.h

```cpp
#pragma once

#include <memory>
#include <random>

// clang-format off
#include <Eigen/Dense>
// clang-format on

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

struct Config final {
  std::size_t epochs;
  std::size_t mini_batch_size;
  double eta;
```

```
  bool monitor_training_cost;
  bool monitor_training_accuracy;
  bool monitor_testing_cost;
  bool monitor_testing_accuracy;
};

struct Metric final {
  std::vector<double> training_cost, training_accuracy;
  std::vector<double> testing_cost, testing_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 StochasticGradientSearch(
      const std::vector<std::shared_ptr<const IData>> &training,
      const std::vector<std::shared_ptr<const IData>> &testing,
      const Config &cfg);

 private:
  template <typename Iter>
  void UpdateMiniBatch(const Iter mini_batch_begin, const Iter mini_batch_end,
                       const std::size_t mini_batch_size, const double eta);

  std::pair<std::vector<Eigen::MatrixXd>, std::vector<Eigen::VectorXd>>
  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_()),
```

```
      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::StochasticGradientSearch(
    const std::vector<std::shared_ptr<const IData>> &training,
    const std::vector<std::shared_ptr<const IData>> &testing,
    const Config &cfg) {
  const auto training_size = training.size();
  const auto whole_mini_batches_number = training_size / cfg.mini_batch_size;
  const auto remainder_mini_batch_size = training_size % cfg.mini_batch_size;

  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;
```

```
    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::Cost(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 5: Файл data_supplier.h

```
#pragma once

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

#include "perceptron.h"

namespace hw2 {

constexpr std::size_t kScanSize = 784;
constexpr std::size_t kDigitsNumber = 10;

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>> training_, testing_,
      validation_;

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

  std::vector<std::shared_ptr<const nn::IData>> GetTrainingData()
      const override {
    return training_;
  }
  std::vector<std::shared_ptr<const nn::IData>> GetValidationData()
      const override {
    return validation_;
  }
  std::vector<std::shared_ptr<const nn::IData>> GetTestingData()
      const override {
    return testing_;
  }
};

}  // namespace hw2
```

Листинг 6: Файл data_supplier.cc

```
#include "data_supplier.h"

#include <spdlog/spdlog.h>

#include <boost/algorithm/string/classification.hpp>
#include <boost/algorithm/string/split.hpp>
```

```cpp
#include <cassert>
#include <fstream>
#include <iterator>
#include <stdexcept>
#include <string>

#include "perceptron.h"

namespace hw2 {

namespace {

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;
```

```
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

```cpp
#include <matplot/matplot.h>

#include <memory>

#include "activation_function.h"
#include "cost_function.h"
#include "data_supplier.h"
#include "perceptron.h"

namespace {

const std::string kDefaultTestPath = "../data/mnist_test.csv";
const std::string kDefaultTrainPath = "../data/mnist_train.csv";

constexpr std::size_t kHiddenLayerSize = 40;
constexpr static auto kCfg = nn::Config{
    .epochs = 200,
    .mini_batch_size = 100,
    .eta = 0.025,
    .monitor_training_cost = true,
    .monitor_training_accuracy = true,
    .monitor_testing_cost = true,
    .monitor_testing_accuracy = true,
};

void RunLeakyReluSoftmaxMSE() {
  const auto data_supplier =
      hw2::DataSupplier(kDefaultTrainPath, kDefaultTestPath, 0.0, 1.0);
  const auto training = data_supplier.GetTrainingData();
  const auto testing = data_supplier.GetTestingData();

  auto cost_function = std::make_unique<nn::CrossEntropy>();
  auto activation_functions =
      std::vector<std::unique_ptr<nn::IActivationFunction>>{};
  activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
  activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
  activation_functions.push_back(std::make_unique<nn::LeakyReLU>(0.01));
  activation_functions.push_back(std::make_unique<nn::Softmax>());
  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 = matplot::linspace(0, kCfg.epochs);
81    matplot::plot(x, metrics.training_cost, x, metrics.testing_cost);
82    matplot::title(
83        "Leaky ReLU, Softmax + Cross-entropy training, testing cost");
84    matplot::show();
85
86    matplot::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
87    matplot::title(
88        "Leaky ReLU, Softmax + Cross-entropy training, testing accuracy");
89    matplot::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 = matplot::linspace(0, kCfg.epochs);
115   matplot::plot(x, metrics.training_cost, x, metrics.testing_cost);
116   matplot::title(
117       "Leaky ReLU, Softmax + K.-L. Divergence training, testing cost");
118   matplot::show();
119
```

```
120   matplot::plot(x, metrics.training_accuracy, x, metrics.testing_accuracy);
121   matplot::title(
122       "Leaky ReLU, Softmax + K.-L. Divergence training, testing accuracy");
123   matplot::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 рисунке[1] 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.

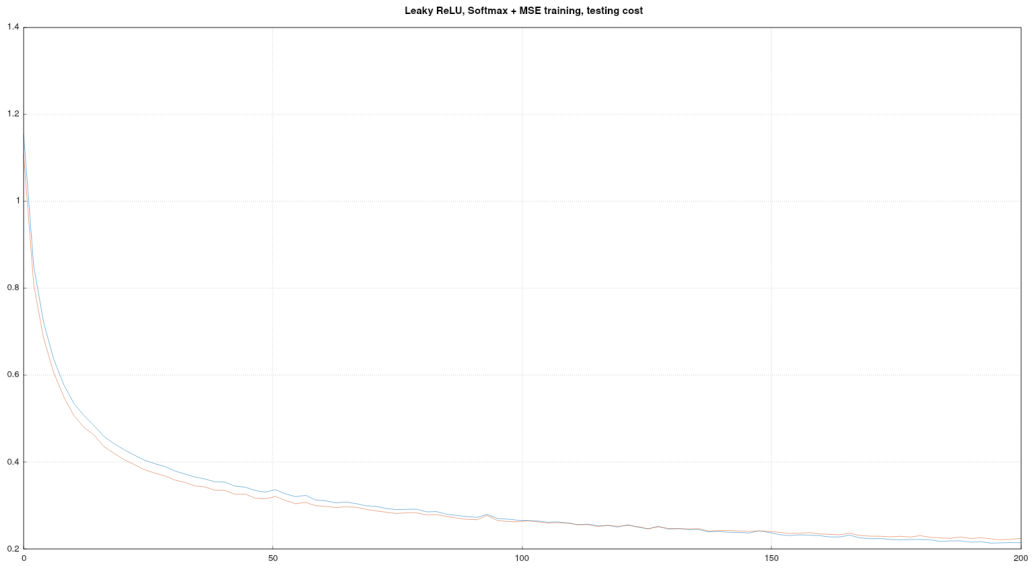

Рис. 1

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.

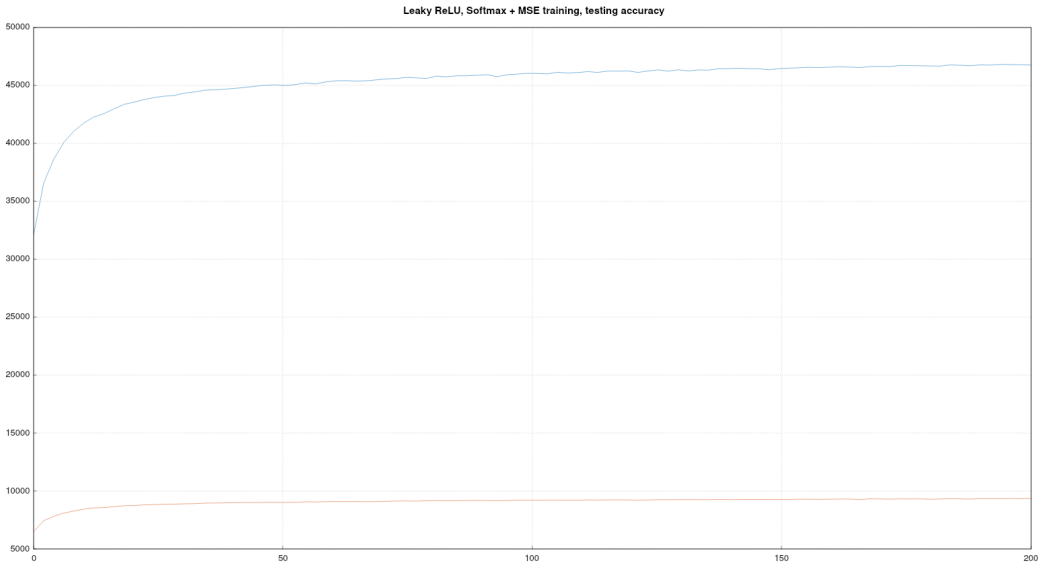

Рис. 2

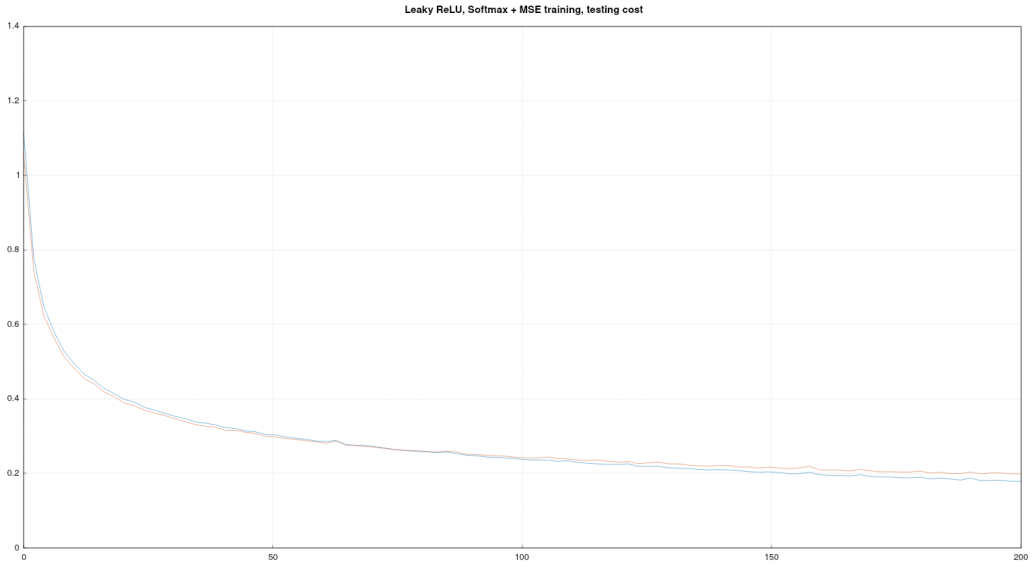

Рис. 3

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 — $\frac{9527}{10000} = 95,27\%$.

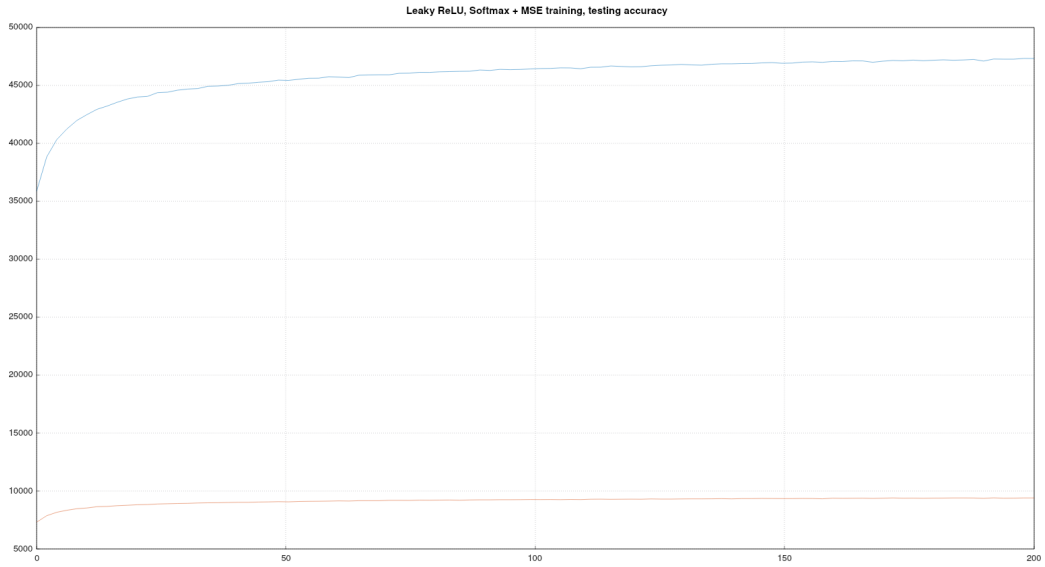

Рис. 4

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.

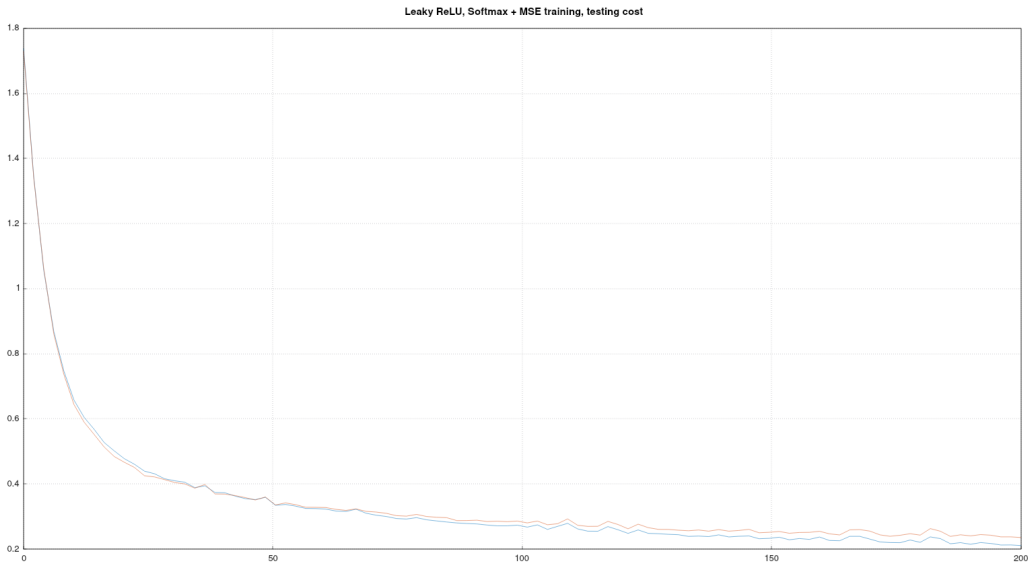

Рис. 5

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 — $\frac{9416}{10000} = 94,16\%$.

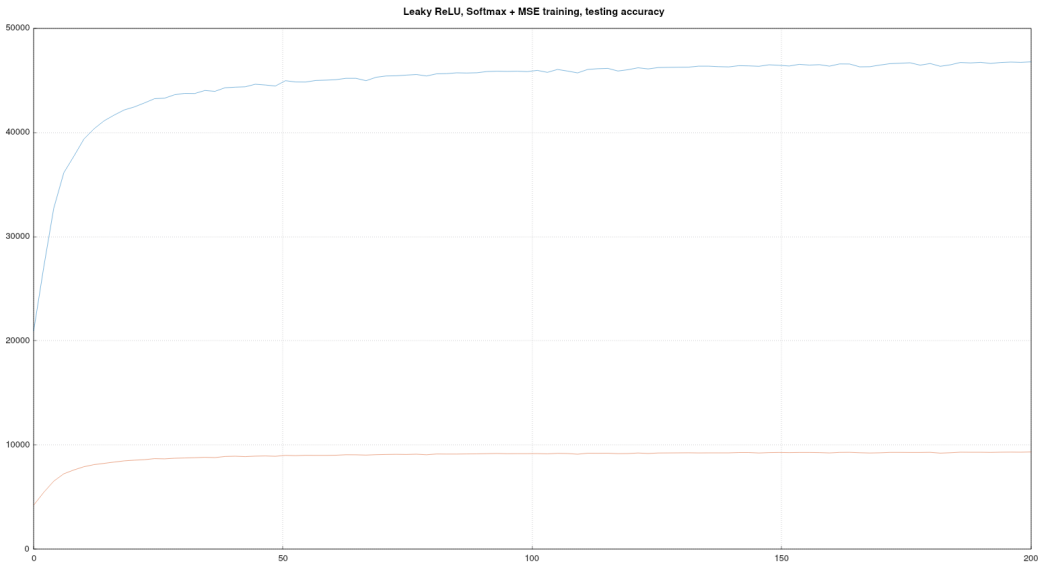

Рис. 6

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.

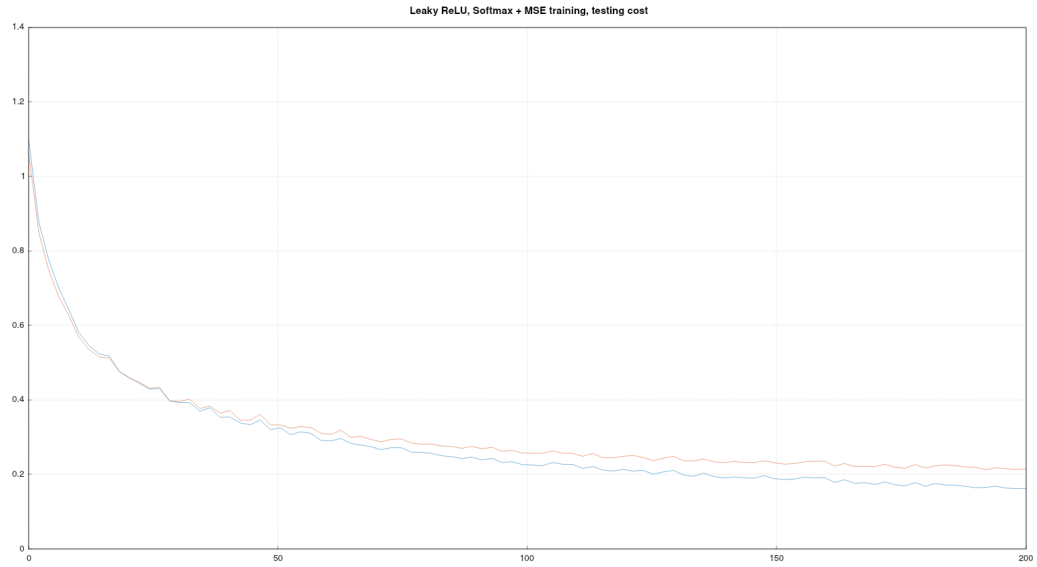

Рис. 7

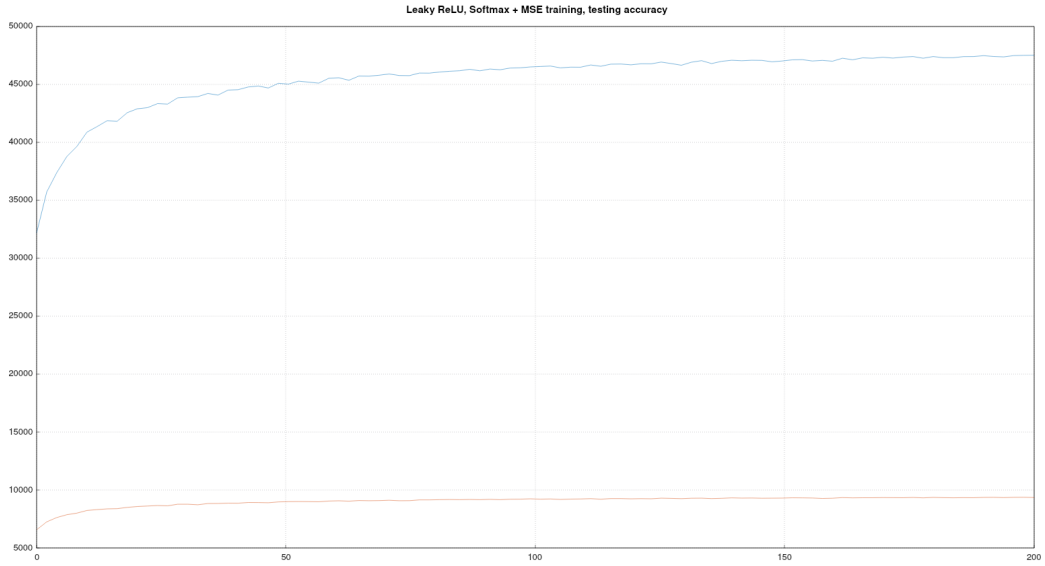

Рис. 8

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 — $\frac{9345}{10000} = 93,45\%$.

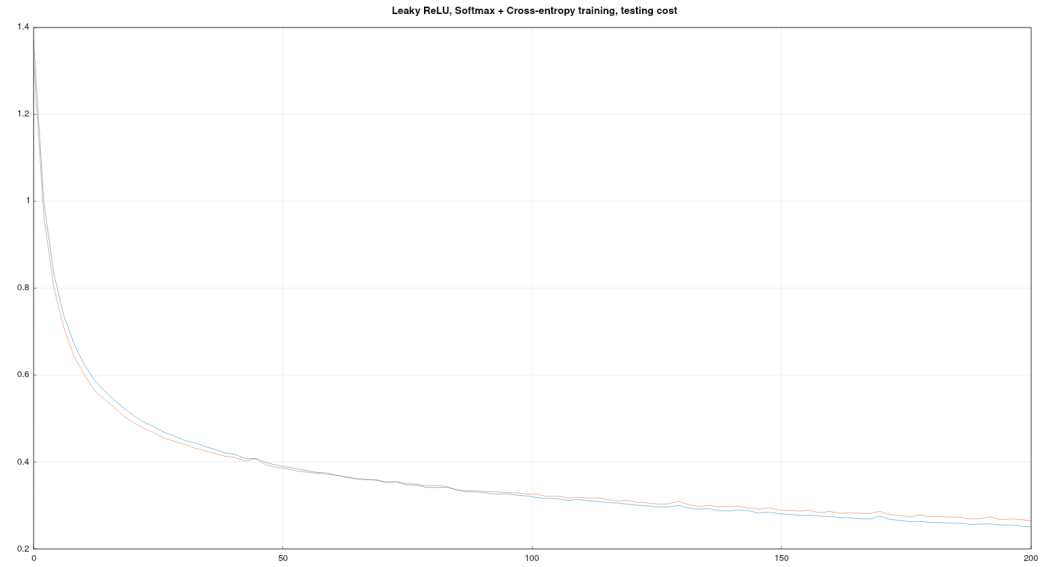

Рис. 9

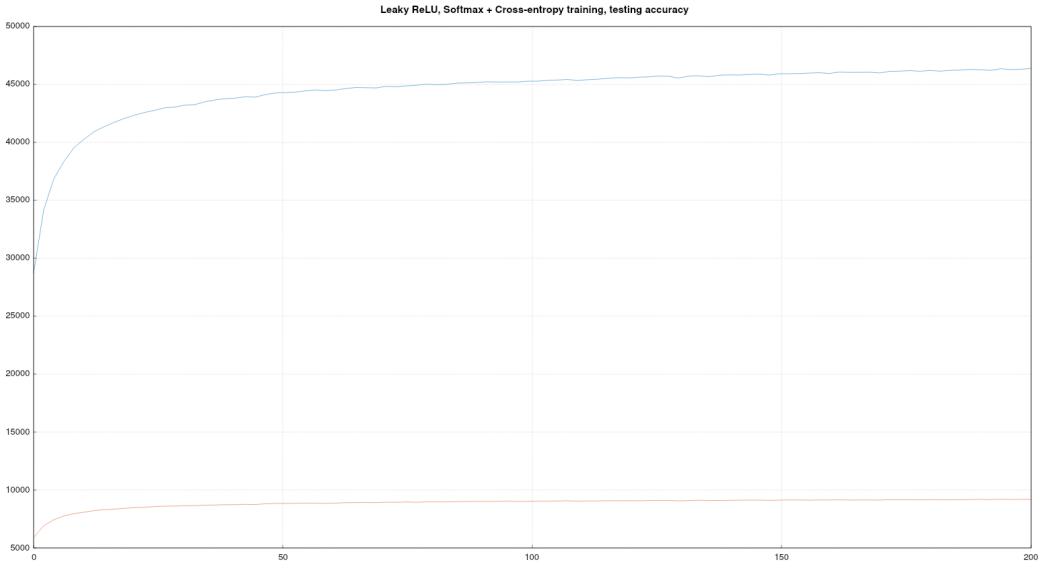

Рис. 10

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.

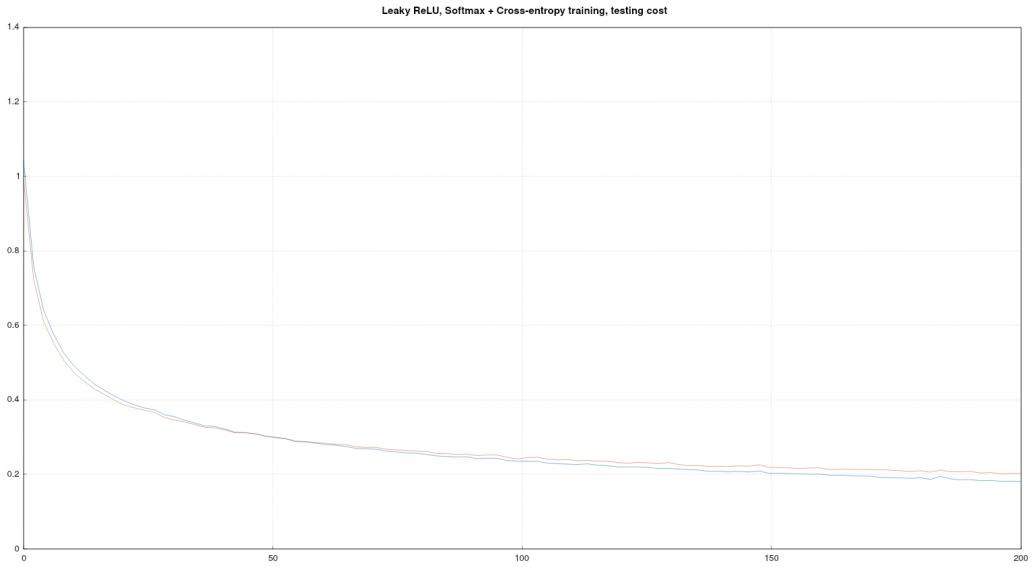

Рис. 11

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 — $\frac{9483}{10000} = 94,83\%$.

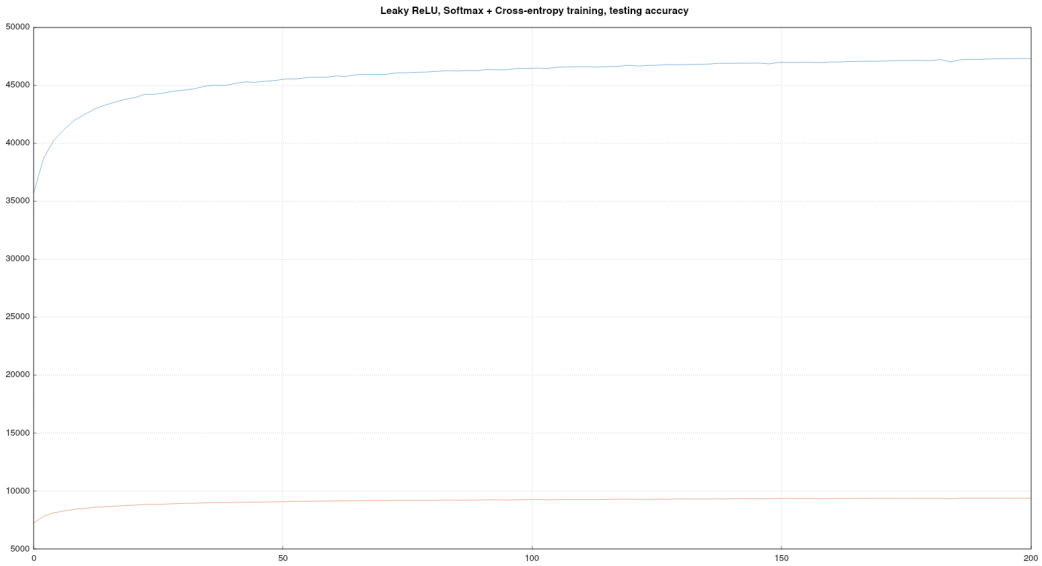

Рис. 12

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.

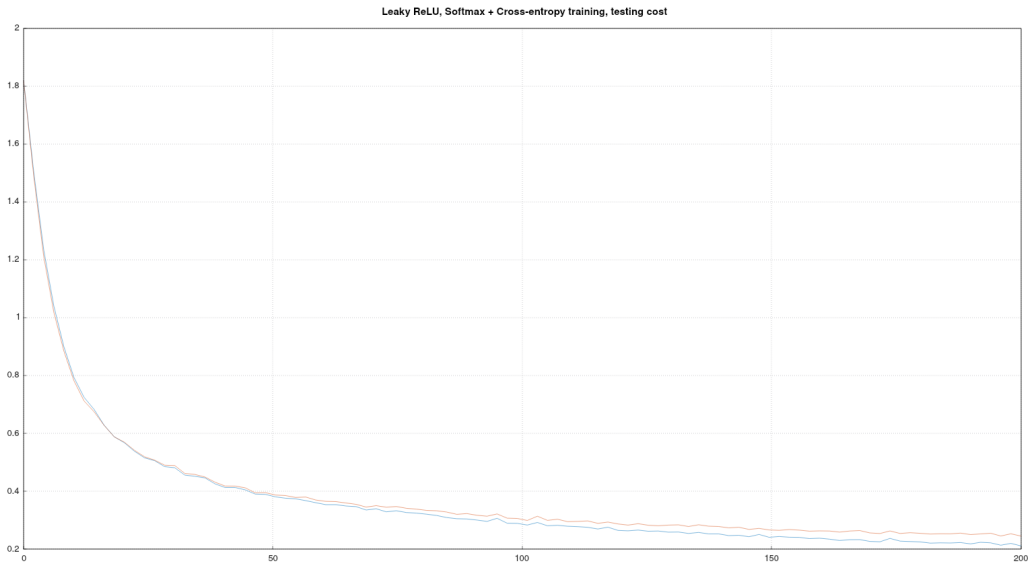

Рис. 13

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 — $\frac{9502}{10000} = 95,02\%$.

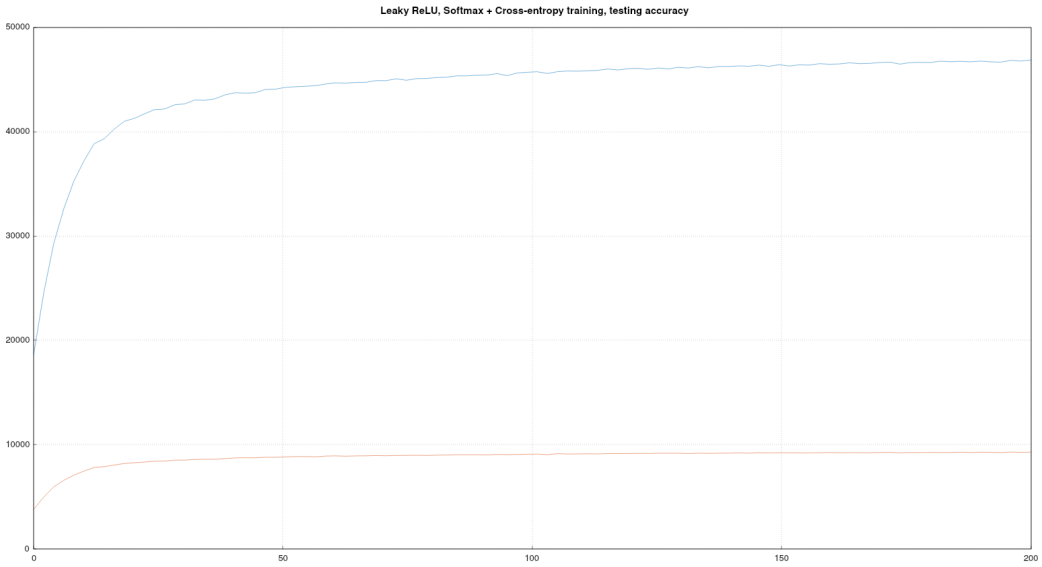

Рис. 14

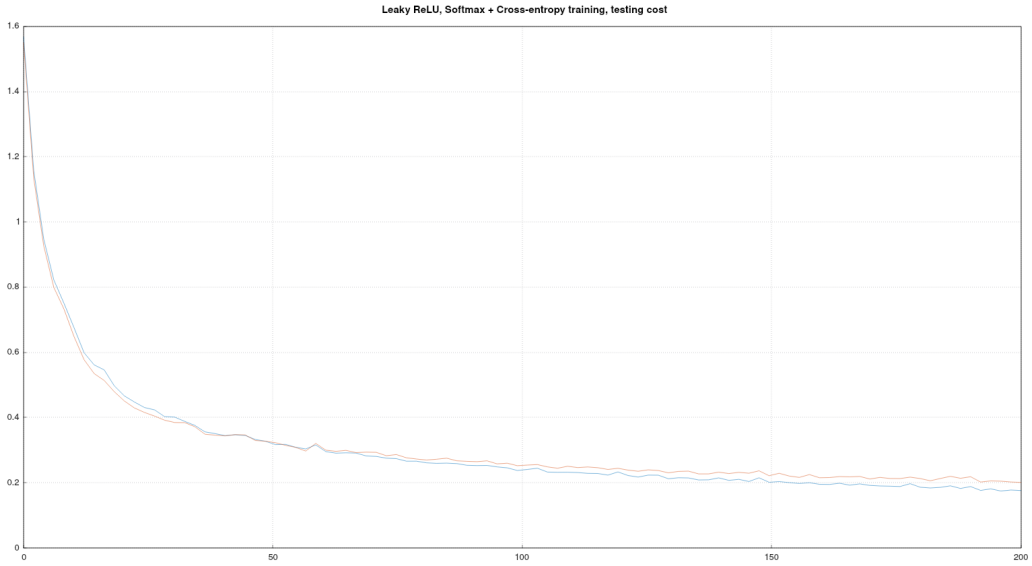

Рис. 15

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.

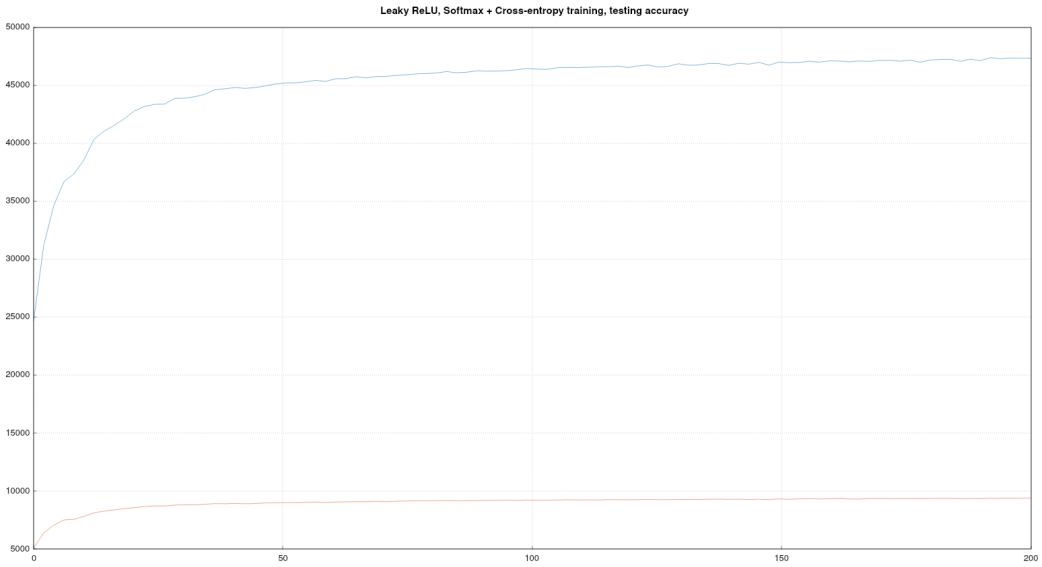

Рис. 16

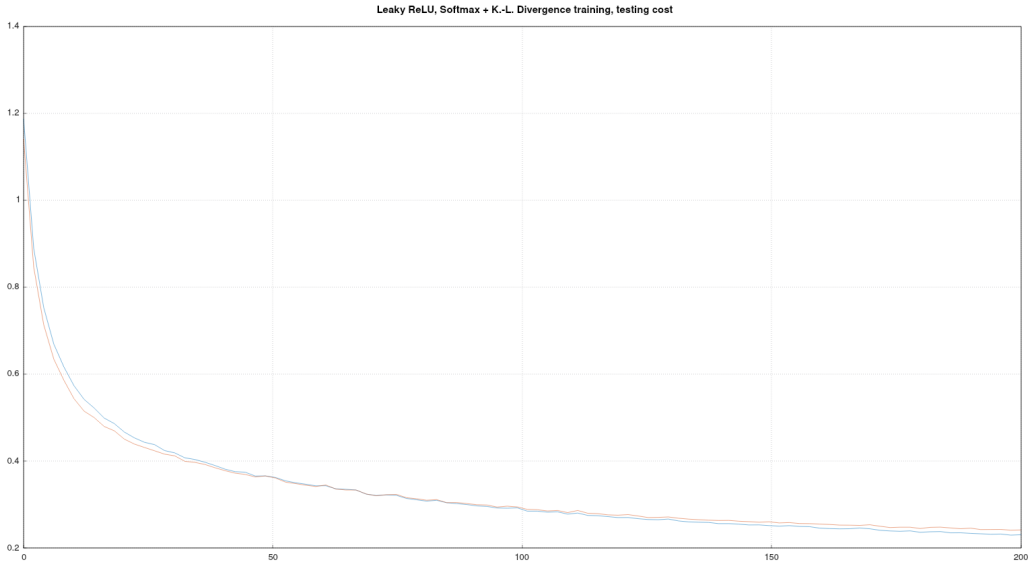

Рис. 17

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 — $\frac{9415}{10000} = 94,15\%$.

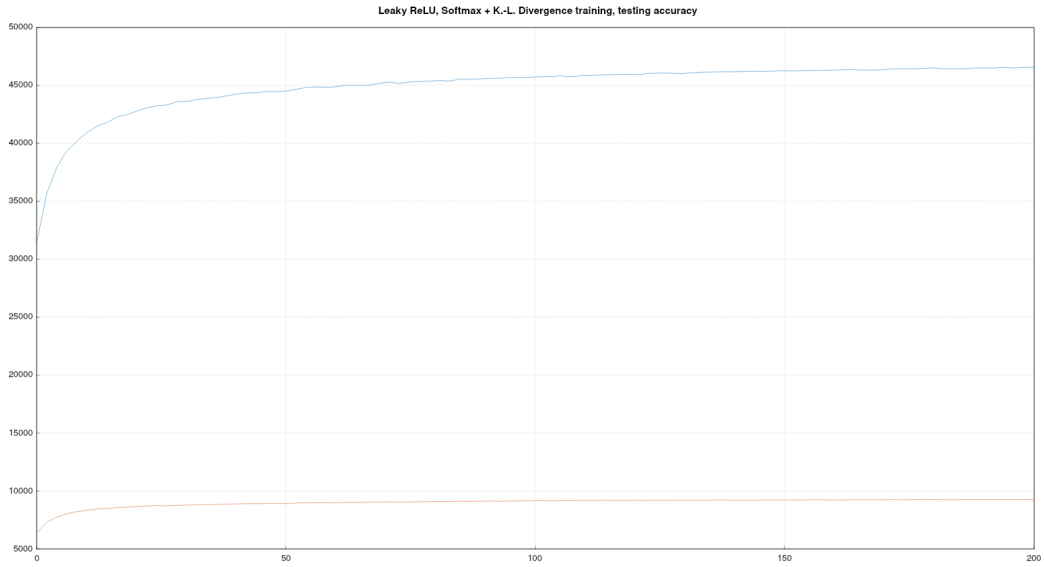

Рис. 18

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.

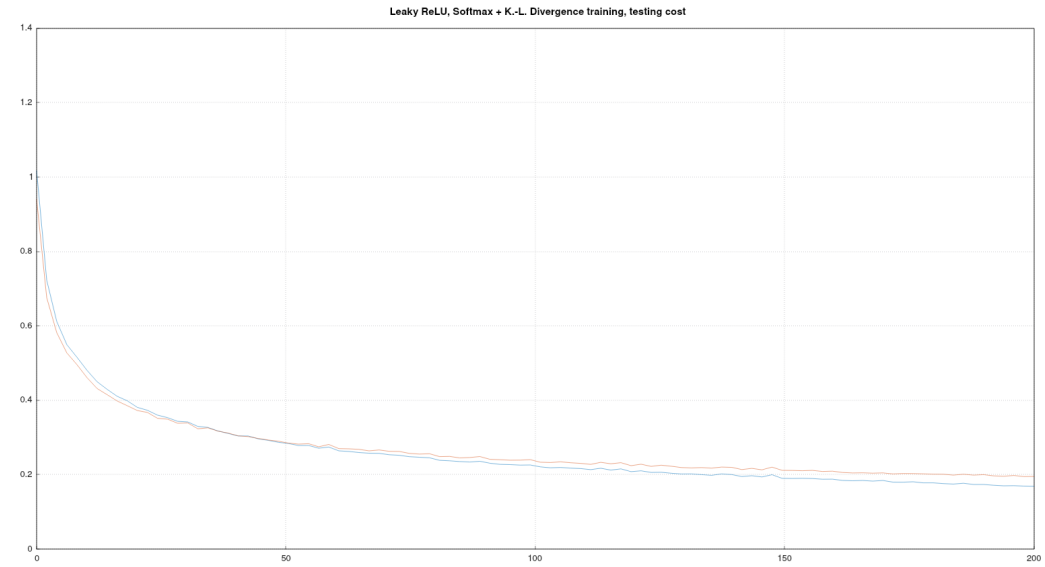

Рис. 19

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 — $\frac{9506}{10000} = 95,06\%$.

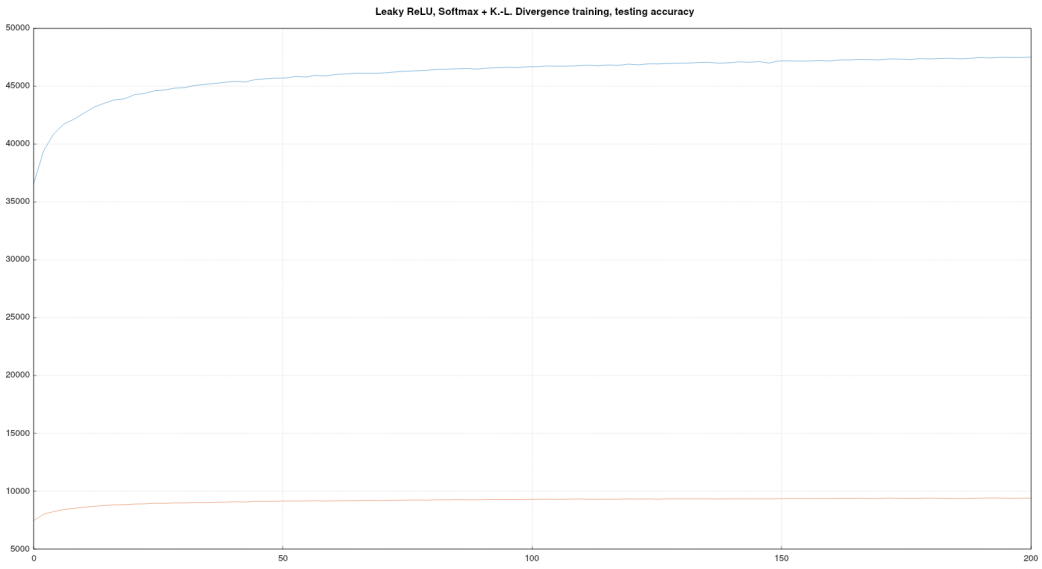

Рис. 20

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 — $\frac{9457}{10000} = 94,57\%$.

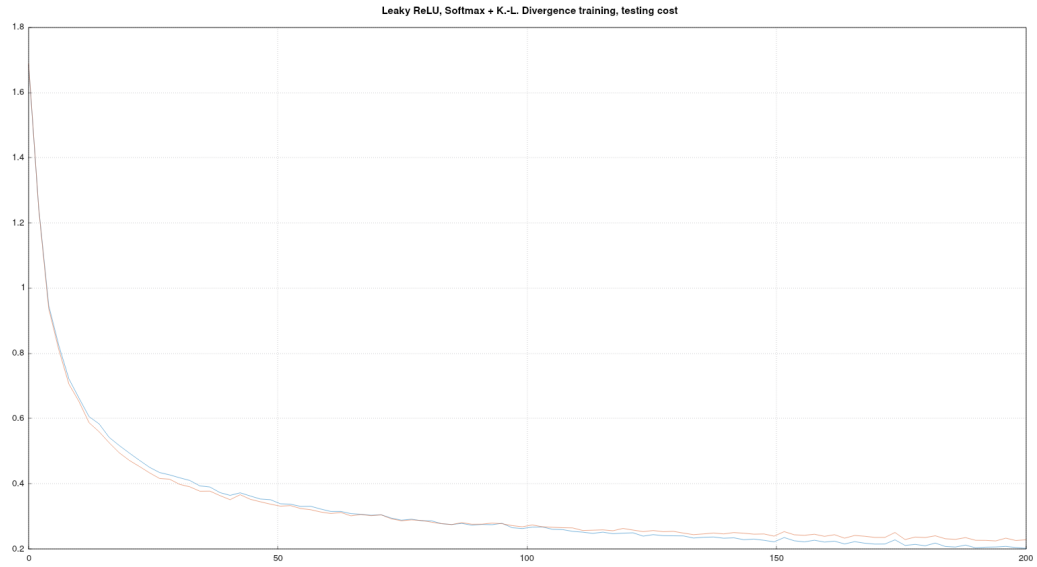

Рис. 21

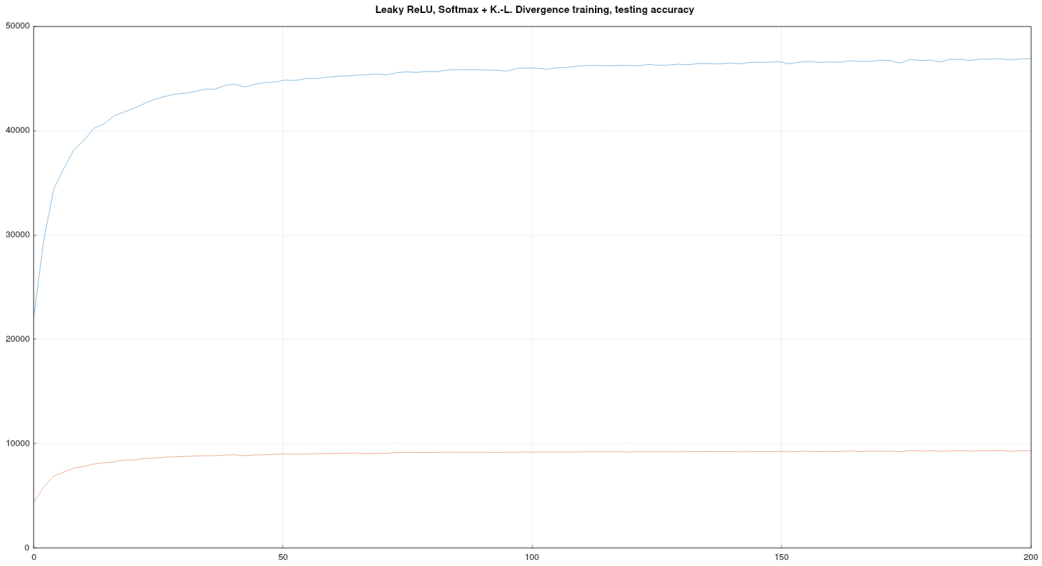

Рис. 22

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 $95,06\%$

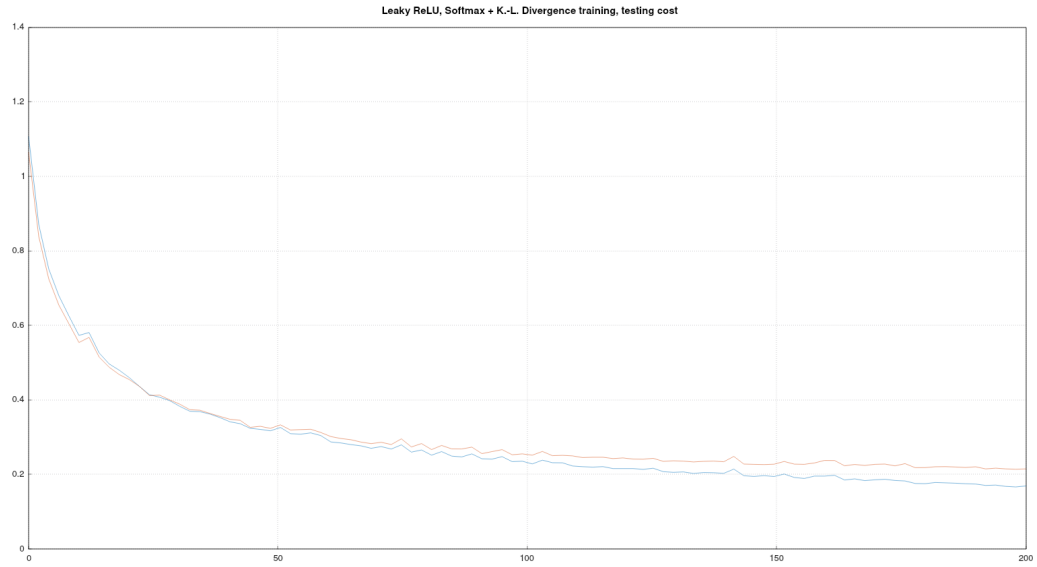

Рис. 23

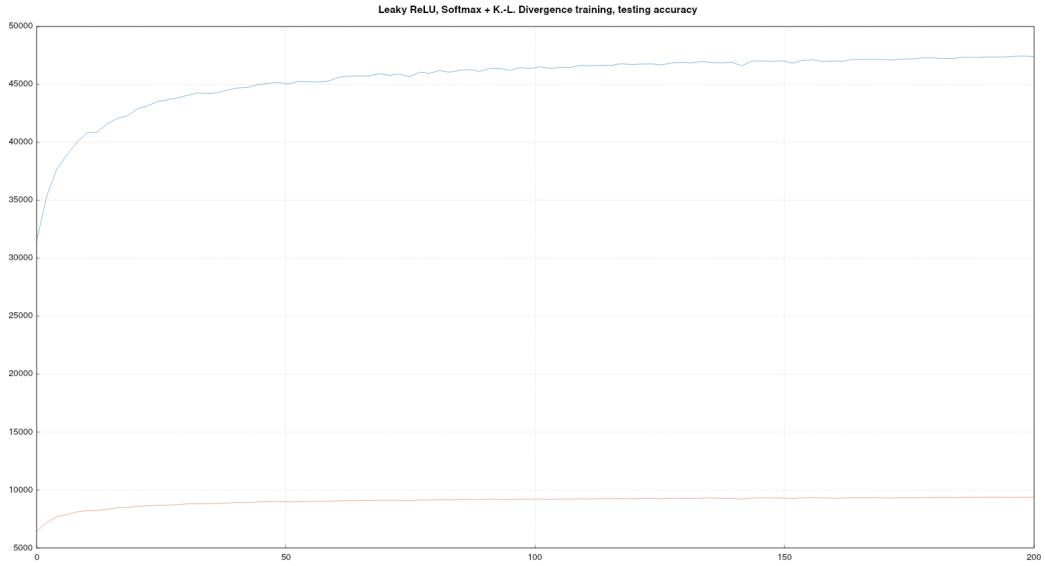

Рис. 24

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, особенных различий нет; здесь ошибка в целом выше, чем в предыдущих двух случаях.