ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: Нейронные сети с нуля

Выполнил:
Студент группы БПМИ211 _____ А.В.Проскурин
Подпись И.О.Фамилия

25.05.2023
Дата

Принял:
Руководитель проекта Дмитрий Витальевич Трушин
Имя, Отчество, Фамилия

доцент, к.ф.-м.н.
Должность, ученое звание

ФКН НИУ ВШЭ
Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки _____ 2023 _____ _____
Оценка (по 10-ти бальной шкале) Подпись

Москва 2023

Содержание

Аннотация

Цель данного проекта – познакомиться с внутренним устройством нейросетей, а также написать свою простейшую нейросеть на языке C++, с последующим обучением.

1 Введение

1.1 Ссылка на репозиторий GitHub

https://github.com/K1ngNothing/Neural-Networks-from-Scratch

1.2 Задачи проекта

В задачи данного проекта входит изучение нейронных сетей:

1. Изучить и изложить необходимую теорию
2. Реализовать полносвязную нейросеть на языке C++
3. Изложить в отчете архитектуру и дизайн имплементации
4. Обучить нейросеть на базе данных MNIST

1.3 Источники информации

Главным источником информации является книга "Neural Networks from Scratch in Python"[10], последующее изложение в большей части будет основываться на ней. Математическая часть нейросети будет основываться на консультациях к проекту: [12] [13].

1.4 Краткое описание устройства нейросети

Рассмотрим устройство нейросети на примере задачи предсказания цены квартиры по списку параметров.

Внутренне устройство Нейронная сеть представляет собой ориентированный граф, где вершины – нейроны, а ребра – связи между ними. Вершины данного графа группируют в *слои*, и ребра проводят строго между соседними слоями (в частности, внутри слоя ребер нет).

В любой нейросети присутствует хотя бы два слоя: входной и выходной. Все остальные слои мы будем называть *внутренними*. Обычно в нейросети есть хотя бы один внутренний слой.

Если между любыми двумя соседними слоями проведены всевозможные ребра, то такую нейросеть называют *полносвязной*. Далее мы будем рассматривать только полносвязные сети.

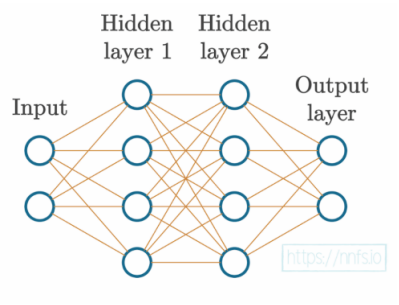

Рис. 1: Пример полносвязной нейросети ([10], Fig. 1.10)

Работа нейросети У каждого нейрона есть какое-то числовое значение, называемое *сигналом*. Изначально сигналы выставлены только во входном слое, в соответствии с входными данными задачи (в нашем примере туда заполняются параметры квартиры). Далее сигналы передаются последовательно от слоя к слою, и результатом работы нейросети считается значения сигналов на выходном слое (в нашем примере в выходном слое будет единственный нейрон, и его сигнал и будет предсказанной ценой квартиры)

Рассмотрим передачу сигнала между соседними слоями 1 и 2. Каждый нейрон слоя 2 смотрит на каждый входящий сигнал со слоя 1, умножает его на вес ребра, по которому пришёл сигнал, и суммирует полученные значения. После этого к полученному сигналу добавляется еще одно число – сдвиг нейрона.

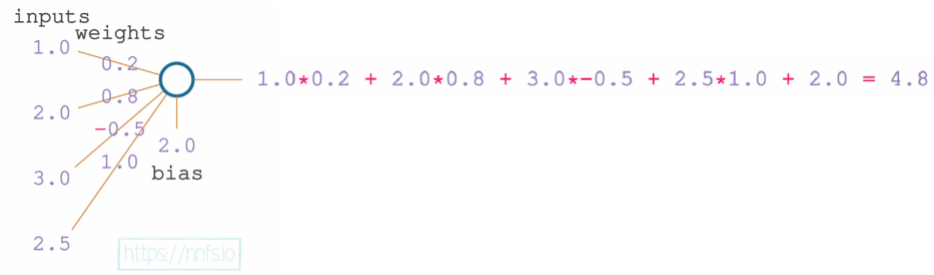

Рис. 2: Пример работы нейрона ([10], Fig. 2.02 и Fig. 2.03)

После того как все нейроны в слое посчитают сигнал, ко всем сигналам одновременно применяется функция активации. Часто она является покоординатной (т.е. применяется к каждому нейрону по-отдельности: например, так применяются функции Sigmoid и ReLU), но иногда она зависит сразу от всех нейронов в слое (например, функция SoftMax).

Данные преобразования гораздо проще записываются в матричном виде: пусть размеры слоев 1 и 2 равны n и m соответственно, а сигналы нейронов этих слоев сложим в векторы $x \in \mathbb{R}^n$ и $y \in \mathbb{R}^m$. Тогда передачу сигнала между слоями можно записать одной формулой:

$$y = \sigma(Ax + b)$$

где $A \in \mathrm{M}_{m\,n}(\mathbb{R})$ – матрица с весами ребер (в i строке лежат веса ребер, входящий в i-й нейрон второго слоя), $b \in R^m$ – вектор сдвигов нейронов второго слоя, $\sigma : \mathbb{R}^m \to \mathbb{R}^m$ – функция активации. Матрицы A и b считаются параметрами нейросети.

Обучение нейросети Процесс обучения выглядит следующим образом: нам дается *обучающая выборка* – это набор пар (входные данные, правильный ответ). В нашем примере можно посмотреть на уже оцененные квартиры, и по ним построить обучающую выборку.

Также вводится функция расстояния, называемая *функцией потерь*: она используется для того, чтобы понимать, насколько сильно предсказание нейросети отличается от правильного.

Теперь обучение происходит в несколько этапов, называемые *эпохами*. На каждой эпохе нейросеть проходит по каждому элементу обучающей выборки, делает предсказание по входным данным, после чего меняет свои параметры (матрицы A и b) в зависимости от значения функции потерь.

Проделав данный процесс много раз, мы получим нейросеть, которая хорошо предсказывает по обучающей выборке, и, как мы надеемся, будет хорошо предсказывать по реальным данным.

2 Функциональные требования

Предоставлен `class Model` – объект нейросети. Данный класс поддерживает:

1. Конструктор: принимает размеры и функции активации каждого слоя.
2. `Train`: принимает набор пар (ввод, ответ), максимальное количество эпох тренировки, порог остановки (если функция потерь падает ниже порога, обучение останавливается), начальный learning rate, learning rate decay, и функцию потерь. Производит обучение нейросети с данными параметрами и возвращает среднее значение функции потерь на последней эпохе.
3. `Predict`: по заданному вводу возвращает предсказание нейросети.
4. Сериализация: поддерживается возможность записать внутреннее состояние нейросети в файл, с последующим восстановлением.

Также предоставлены имплементации некоторых пороговых функций (Sigmoid, ReLU, Linear, SoftMax) и функций потерь (MSE, CrossEntropy)

3 Нефункциональные требования

1. Язык программирования: C++20, компилятор – GCC [1]
2. Система контроля версий: Git (GitHub [2])
3. Система сборки: CMake [3]
4. Style guide: немного модифицированный Google C++ Style Guide [4]
5. Программа форматирования кода: clang format [5]
6. Библиотеки для работы с матрицами: Eigen [6] и EigenRand [7]
7. Среда разработки: Visual Studio Code [8]

4 Математическое описание нейросети

Пусть задано отображение $\varphi\colon \mathbb{R}^n \to \mathbb{R}^m$, нам неизвестное. Мы хотим приблизить это отображение с помощью нейросети.

Структура нейросети В нейросети есть k внутренних слоев ($k \geqslant 1$), i-й слой делает преобразование $\psi_i\colon \mathbb{R}^{l_i} \to \mathbb{R}^{w_i}$, $x \mapsto \sigma(A_i x + b_i)$ ($x \in \mathbb{R}^{l_i}$ – столбец, $A_i \in \mathrm{M}_{w_i\, l_i}(\mathbb{R})$, $b_i \in \mathbb{R}^{w_i}$, $\sigma\colon \mathbb{R}^{w_i} \to \mathbb{R}^{w_i}$; $w_i = l_{i+1}$, $l_1 = n, w_k = m$), где A_i, b_i считаются параметрами. Всю работу нейросети будем обозначать $\psi\colon \mathbb{R}^n \to \mathbb{R}^m$ (т.е. $\psi = \psi_k \circ \ldots \circ \psi_1$)

Постановка задачи Введем обучающую выборку $X = \{(x_i,\ y_i)\}$: $x_i \in \mathbb{R}^n$, $y_i \in \mathbb{R}^m$, $y_i = \varphi(x_i)$ (т.е. обучающая выборка – это набор пар (вход, правильный ответ)). Далее зафиксируем функцию расстояния $f(x, y)\colon \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$ и введем *функцию потерь* $L : \mathbb{R}^n \to \mathbb{R}^m$, $L(x_i) = f(\psi(x_i), y_i)$.

Задача: минимизировать среднее значение по всей выборке X для функции потерь $L(x)$.

Решение Зафиксируем произвольный элемент обучающей выборки $(x_0,\ y_0)$. Тогда $L(x_0) = f(\psi(x_0), y_0)$ есть функция, зависящая от параметров $\theta_i = (A_i, b_i)$. Объединим все параметры в один вектор $\theta = (\theta_1, \ldots, \theta_k)^t$ (т.к. в нейросети k внутренних слоев, то и параметров тоже k). Мы хотим посчитать градиент $\nabla L = \frac{dL(x)}{d\theta}$ и изменить параметры против этого градиента, т.е. сделать градиентный спуск.

Вывод формул градиентого спуска Введем обозначения для компонент градиента: $\nabla L = (\nabla L_1, \ldots, \nabla L_k)$, где $\nabla L_i = \frac{\partial L}{\partial \theta_i}(x_0)$. Также обозначим вход для i-го слоя: z_i (т.е. i-й слой делает преобразование $z_i \mapsto \sigma_i(A_i z_i + b_i)$)

Тогда можно записать явную формулу для ∇L_i:

$$\nabla L_i = \frac{\partial L}{\partial \theta_i}(x_0) = \frac{\partial f}{\partial x}(\psi(x_0), y_0) \cdot \prod_{j=i+1}^{k} \frac{\partial \psi_j}{\partial x}(z_j) \cdot \frac{\partial \psi_i}{\partial \theta_i}(\theta_i)$$ [1]

Обозначим $u_i = \frac{\partial f}{\partial x}(\psi(x_0), y_0) \cdot \prod\limits_{j=i+1}^{k} \frac{\partial \psi_j}{\partial x}(z_j) \in \mathrm{M}_{1\, w_i}(\mathbb{R})$. Тогда последнюю формулу можно переписать в упрощенном виде:

$$\nabla L_i = u_i \cdot \frac{\partial \psi_i}{\partial \theta_i}(\theta_i)$$

Значит, мы можем пересчитывать градиент рекурсивно: положим $u_0 = \frac{\partial f}{\partial x}(\psi(x_0), y_0)$, и будем идти по слоям от большего номера к меньшему, пересчитывая градиент:

$$\mathrm{grad}_{A_i} = \left(z_i \cdot u_i \cdot \frac{\partial \sigma_i}{\partial x}(A_i z_i + b_i) \right)^t \in \mathrm{M}_{w_i\, l_i}(\mathbb{R})$$

$$\mathrm{grad}_{B_i} = \left(u_i \cdot \frac{\partial \sigma_i}{\partial x}(A_i z_i + b_i) \right)^t \in \mathrm{M}_{w_i\, 1}(\mathbb{R})$$

$$u_{i-1} = u_i \cdot \frac{\partial \sigma_i}{\partial x}(A_i z_i + b_i) \cdot A_i \in \mathrm{M}_{1\, l_i}(\mathbb{R}) = \mathrm{M}_{1\, w_{i-1}}(\mathbb{R})$$

Данный формулы получены из формул выше, с учетом того, что i-й слой нейросети действует как $x \mapsto \sigma_i(A_i x + b_i)$, а также с учетом правил матричного дифференцирования [11]

Обучение Обучение происходит в несколько этапов, называемых *эпохами*. Каждую эпоху нейросеть проходит по всем элементам обучающей выборки (x_j, y_j), вычисляет $\psi(x_j)$, после чего вычисляет градиент, одновременно оптимизируя параметры слоев:

$$\Delta A_i = -\,\mathrm{grad}_{A_i} \cdot \mathit{LearningRate}(\mathit{epoch})$$

$$\Delta b_i = -\,\mathrm{grad}_{B_i} \cdot \mathit{LearningRate}(\mathit{epoch})$$

, где *learing rate* – скорость обучения нейросети. Это функция $\mathbb{N} \to \mathbb{R}$, которая подбирается в зависимости от задачи. Обычно её делают убывающей, вводя дополнительный параметр *learning rate decay*. Тогда

$$\mathit{LearningRate}(\mathit{epoch}) = \frac{\mathit{StartingLearningRate}}{1 + \mathit{LearningRateDecay} \cdot \mathit{epoch}}$$

Обучение происходит до тех пор, пока среднее значение функции потерь по выборке не будет ниже заданного уровня, или пока не будет превышено максимальное количество эпох обучения.

[1] Для лучшего понимания напомню, что $f\colon (x, y) \mapsto f(x, y)$; $\psi_i\colon x \mapsto \sigma(A_i x + b_i)$; $\theta_i = (A_i, b_i)$

5 Имплементация

5.1 class Model

Предоставляет класс для работы с произвольной полносвязной нейросетью. Он поддерживает конструктор, принимающий список размеров слоев и список функций активаций, а также ряд методов:

- `Train` для обучения нейросети: принимает обучающую выборку `training_data`, максимальное количество эпох обучения `epoch_count`, критерий остановки `stop_threshold` (если значение функции потерь падает ниже этого уровня, обучение останавливается), параметры скорости обучения `starting_learning_rate` и `learning_rate_decay`, функцию потерь `loss_function`; возвращает среднее значение функции потерь на последней эпохе тренировки. Дополнительно каждую эпоху выводит на стандартный поток вывода текущее значение уровня потерь и текущую скорость обучения.
- `Predict` для получения предсказаний нейросети: принимает входные данные в виде вектора, возвращает предсказание нейросети.
- `GetAverageLoss` и `GetAccuracy` для оценки эффективности нейросети: принимают данные в формате обучающей выборки и функцию потерь, возвращают среднее значение функции потерь по всей выборке и долю правильных предсказаний соответственно.
- `Serialize` и конструктор по имени файла – подробно описаны в разделе про сериализацию

Данный класс хранит в себе список слоев и поддерживает все перечисленные в функциональных требованиях методы. Во время обучения изменения в слоях применяются не сразу, а только при вызове `ApplyDeltas`: сделано это для того, чтобы поддерживать тренировку батчами.

5.2 class Layer

Предоставляет класс для работы со слоем нейросети. **Данный класс не предполагается для создания пользователем**, поэтому вынесен в `namespace impl`. Имеет конструктор, принимающий размер входных данных, размер выходных данных и функцию активации. Предоставляет методы `PushVector`, `PushGradient`, `UpdateDelta` для обучения нейросети, а также метод `ApplyToVector`, используемый в предсказаниях нейросети. Класс хранит в себе параметры слоя (линейные параметры `A_` и `b_`, функцию активации `sigma_`), а также поля, необходимый для обучения слоя (`last_input_, delta_A_, delta_b_, delta_count_`).

В конструкторе от размеров слоя и функции активации используется генератор случайных чисел. Для его получения используется функция `GetRNG`: она единожды создает статический генератор, который возвращается по ссылке.

Метод `PushVector` помимо вычисления $\sigma(Ax + b)$ также сохраняет x в поле `last_input_`. Это используется в методах `PushGradient` и `UpdateDelta`, поэтому данные методы **не являются безопасными без предварительного вызова** `PushVector` (чего при обучении нейросети не происходит, ведь мы всегда сначала делаем `PushVector`, а потом `PushGradient`). Этой особенности нет в методе `ApplyToVector`, что отличает методы между собой.

Поля `delta_A_, delta_b_, delta_count_`, а также метод `ApplyChanges` сделаны для поддержки тренировки батчами: изменения в слое применяются не сразу, а записываются в `delta_A_, delta_b_`. В момент вызова `ApplyChanges` они усредняются и применяются к параметрам слоев.

Конструктор по `FileReader` и метод `Serialize` описаны в отдельном пункте про сериализацию.

5.3 class ActivationFunction

Предоставляет класс для хранения функции активации – нелинейного параметра слоя нейросети. Предоставляет оператор `()` для взятия значения функции в точке и оператор `[]` для взятия производной функции в точке. Представляет собой стирающий тип, т.е. он позволяет хранить в себе произвольный класс с сигнатурой функции активации (с небольшой оговоркой, подробности в пункте про сериализацию). Также реализовано несколько экземпляров классов для различных функций активации: `Sigmoid`, `ReLU`, `Linear`, `SoftMax`.

Функции активации sigmoid и ReLU применяются покоординатно. Для избежания дублирования кода были созданы функции `ApplyCoordinateWise` и `GetJacobianMatrix` – обе они принимают функцию, применяемую к одной координате вектора. Для избежания коллизии имен функций были созданы неймспейсы `calculate_one_coordinate` и `calculate_one_derivative`: благодаря им функции взятия значения в точке и

взятия производной в точке можно называть по названию класса, а различать их между собой по названию неймспейса. Поле `type_` и метод `GetType` используется при сериализации нейросети – об этом подробнее в отдельном пункте.

5.4 class LossFunction

Предоставляет класс для хранения функции потерь – функции штрафа при обучении и тестировании нейросети. Предоставляет оператор `()` для взятия значения функции в точке `(x, y)` и метод `GetGradientX` для взятия градиента по x в точке `(x, y)`. Также, как и `ActivationFunction`, является стирающим типом. Также реализовано несколько экземпляров классов для различных функций потерь: `MSE`, `CrossEntropy`.

Использовать метод `GetGradientX` вместо оператора `[]` пришлось из-за того, что c++ не позволяет передавать несколько аргументов в оператор `[]`.

5.5 Стирающие типы

Стирающие типы позволяют хранить в переменной данного типа переменную любого другого типа с заданной сигнатурой. Реализованы классы `ActivationFunction`, позволяющий хранить произвольный класс с сигнатурой функции активации, и класс `LossFunction`, позволяющий хранить произвольный класс с сигнатурой функции потерь.

Стирание типа достигается следующим образом: пусть мы хотим реализовать `class TypeErasure`, который умеет хранить произвольный объект с методом `Method`. Заведем класс-интерфейс `InnerBase`, от которого наследуется шаблонный класс `Base`. Класс `Base` честно хранит объект, и соответствующие методы вызывает напрямую. А класс `TypeErasure` хранит указатель на `InnerBase` и вызывает методы по указателю.

Т.к. `InnerBase` не является шаблонным, то и `TypeErasure` не шаблонный класс. При этом по указателю на `InnerBase` хранится не класс `InnerBase`, а класс-наследник `Base`. И именно потому, что c++ позволяет по указателю на родителя хранить наследника, и достигается стирание типа.

Псевдокод для лучшего понимания:

```
class TypeErasure {
private:
    class InnerBase {
    public:
        virtual Method(...) const = 0;
        virtual std::unique_ptr<InnerBase> Clone() const = 0;

        virtual ~InnerBase() = default;
    };

    template <typename T>
    class Inner : public InnerBase {
    public:
        Inner(const T& object) = default;
        Inner(T&& object) noexcept = default;

        Method(...) const override {
            return object.Method(...);
        }
        std::unique_ptr<InnerBase> Clone() const override {
            return std::make_unique<Inner>(object_);
        }

    private:
        T object_;
    };

public:
    template <typename T>
```

```
    TypeErasure(T object) : ptr_(std::make_unique<Inner<T>>(std::move(object))) {
    }
    TypeErasure(const TypeErasure& other)
        : ptr_(other.ptr_ == nullptr ? nullptr : other.ptr_->Clone()) {
    }
    TypeErasure(TypeErasure&& other) noexcept = default;
    TypeErasure& operator=(const TypeErasure& other) {
        return *this = TypeErasure(other);
    }
    TypeErasure& operator=(TypeErasure&& other) noexcept = default;

    Method(...) const {
        return ptr_->Method(...);
    }

private:
    std::unique_ptr<InnerBase> ptr_;
};
```

Из нетривиального здесь конструктор копирования: действительно, мы не можем просто скопировать указатель (этого не позволяет `std::unique_ptr`): нам нужно скопировать объект, который лежит по указателю `ptr_`, что не получится сделать напрямую из-за отсутствия информации о типе хранимого объекта. Для этого мы введем метод `Clone`, который возвращает указатель на уже скопированный объект. Реализован он в классе `Inner` (у которого есть доступ к хранимому объекту, так что реализация тривиальна), а в классе `TypeErasure` мы вызываем его по указателю. Оператор присваивания с копированием реализован через конструктор копирования.

В итоге полученная конструкция позволяет хранить любой объект с заданной сигнатурой. Причем при попытке сохранить объект, не подходящий под сигнатуру, произойдет ошибка компиляции (ведь в шаблонном классе `Inner` не скомпилируется соответсвующий метод), а не ошибка исполнения.

5.6 Сериализация

После обучения нейросети возникает естественное желание сохранить параметры нейросети в файл, чтобы потом можно было пользоваться нейросетью без повторного обучения. За это отвечают конструктор `Model(filename)` и метод `Model::Serialize(filename)`.

Формат файла: сначала записывается количество слоев, потом информация о слоях. Линейные параметры слоя `A_` и `b_` записываются в естественном формате: пишутся размеры матрицы `A_`, коэффициенты `A_`, коэффициенты `b_` (размер вектора `b_` совпадает с количеством строк `A_`, поэтому нет нужды записывать его отдельно). От функции активации `sigma_` записывается `sigma_.GetType()`: об этом подробнее в отдельном параграфе. Все, что записывается в файл, записывается в бинарном виде (это сделано для того, чтобы не терять точность типа `double` при записи в текстовом виде). Для чтения / записи в файл в бинарном виде создан класс `FileReader`: в конструкторе он создает `std::fstream`, а методы `Read` и `Write` позволяют читать / записывать заданную переменную в файл (переменная передается как аргумент по ссылке)

Процесс сериализации выглядит следующим образом: `Model::Serialize(filename)` создает объект `FileReader`, записывает количество слоев и вызывает `Layer::Serialize(FileReader)` для каждого из них. Тот в свою очередь записывает информацию о слое в описанном выше формате. Конструирование по сериализации работает абсолютно аналогично, но только вместо записи происходит чтение из файла.

Сериализация ActivationFunction Для сериализации `ActivationFunction` добавлены поле `AFType type_` и метод `GetType()`, возвращающее это поле. Также реализована функция-фабрика `AFFabric(AFType type)`, конструирующая функцию активации типа `type`. `AFType` – это `enum`, в котором перечислены все функции активации. `AFFabric` представляет собой захардкоженный перебор значений `enum AFType`.

В такой реализации сериализации и содержится та самая оговорка из части про `ActivationFunction`: если пользователь хочет написать свою функцию активации, он должен добавить ее в `enum AFType`, а также добавить в фабрику `AFFabric`. Если же пользователь не будет пользоваться сериализацией, достаточно просто как угодно реализовать метод `GetType` (в таком случае при попытке сериализации скорее всего выкинется `runtime error`, но никаких гарантий не дается).

6 Демонстрация создания и обучения нейросети

Приведем пример кода по созданию и обучению нейросети:

```cpp
#include <ActivationFunction/predefined.h>
#include <LossFunction/predefined.h>
#include <model.h>

#include <iostream>

void TrainModel() {
    // Создадим нейросеть с тремя слоями размера 4, 2, 1
    // В качестве функций активации будем использовать ReLU и Linear
    // (ко входному слою функция активации не применяется, поэтому указываем две вместо трех)
    size_t input_layer_size = 4;
    size_t hidden_layer_size = 2;
    size_t output_layer_size = 1;
    model::Model model({input_layer_size, hidden_layer_size, output_layer_size},
                       {model::ReLU(), model::Linear()});

    // Зададим параметры обучения
    size_t epoch_count = 100;
    double stop_threshold = 1e-12;
    size_t batch_size = 10;
    double starting_learning_rate = 0.1;
    double learning_rate_decay = 0.01;
    LossFunction loss_function = model::MSE()

    // Обучим нейросеть на обучающей выборке training_set
    double training_set_loss = model.Train(training_set, epoch_count, stop_threshold,
                                           batch_size, starting_learning_rate,
                                           learning_rate_decay, loss_function);

    // Выведем среднее значение функции потерь на последней эпохе обучения
    std::cout << "Average loss on training set: " << training_set_loss << std::endl;

    // Запишем слои обученной нейросети в файл layers.txt
    model.Serialize("layers.txt");
}

void TestModel() {
    // Восстановим записанные слои из файла layers.txt
    model::Model model("layers.txt");

    // Протестируем модель: посчитаем среднее значение функций потерь
    // В качестве функции потерь будем использовать MSE
    double testing_set_loss = model.GetAverageLoss(testing_set, model::MSE());
    std::cout << "Average loss on testing set: " << testing_set_loss << std::endl;
    // Также выведем долю правильных предсказаний
    std::cout << "Accuracy on testing set: " << model.GetAccuracy(testing_set) << std::endl;

    // Рассмотрим пример предсказания нейросети: возьмем первый элемент тестовой выборки
    // и выведем входные данные, предсказание и правильный ответ
    const model::Vector& input = testing_set[0].input;
    const model::Vector& answer = testing_set[0].output;
    model::Vector prediction = model.Predict(input);
```

```
    std::cout << "Example of prediction:" << std::endl;
    std::cout << "Input:\n" << input << std::endl;
    std::cout << "Prediction:\n" << prediction << std::endl;
    std::cout << "Correct answer:\n" << answer << std::endl;
}

int main() {
    TrainModel();
    TestModel();
}
```

7 Обучение нейросети

Задача Задачей нейросети было научиться распознавать рукописные цифры по картинке. Для этого она обучалась на базе данных MNIST [9]. Данная база содержит 70000 размеченных картинок с рукописными цифрами (60000 картинок в обучающей выборке, 10000 – в тестовой).

Параметры нейросети

1. Размеры слоев: 784, 100, 10
2. Функции активации слоев: ReLU и softmax
3. Функция потерь: cross-entropy loss function
4. Значение функции потерь, при котором обучение останавливалось: 0.01
5. Batch size: 10
6. Starting learning rate: 0.001
7. Learning rate decay: 0.01

Функция активации softmax и функция потерь cross-entropy были выбраны потому, что они идеально подходят для задачи классификации. Softmax позволяет нейросети "голосовать" за то, какая цифра нарисована на картинке: чем больше число в нейроне i, тем более нейросеть уверена, что на картинке изображена цифра i. А функция cross-entropy позволяет поощрять нейросеть за большую уверенность в правильном ответе.

Достаточно низкий learning rate выбран из-за того, что при больших значениях происходил "взрыв" параметров: изменения параметров становились настолько большими, что они быстро уходили в inf, и нейросеть не обучалась.

Результаты За 9 эпох обучения достигнута точность в 94.97% на тестовой выборке со значением функции потерь 0.27 (на обучающей выборке эти показатели равны 99.85% и 0.008 соответственно). Обучение заняло 1 час 48 минут.

Список литературы

[1] URL: https://gcc.gnu.org/.

[2] URL: https://github.com/.

[3] URL: https://cmake.org/.

[4] URL: https://google.github.io/styleguide/cppguide.html.

[5] URL: https://clang.llvm.org/docs/ClangFormat.html.

[6] URL: https://eigen.tuxfamily.org/index.php?title=Main_Page.

[7] URL: https://bab2min.github.io/eigenrand/v0.5.0/en/index.html.

[8] URL: https://code.visualstudio.com/.

[9] URL: http://yann.lecun.com/exdb/mnist/.

[10] Harrison Kinsley and Daniel Kukieła. *Neural Networks from Scratch in Python*. Harrison Kinsley, 2020.

[11] Трушин Д. В. Матричные дифференцирования. URL: https://disk.yandex.ru/i/YynctjURriDSaQ.

[12] Трушин Д. В. Консультация к проекту «Нейросети с нуля», часть 1. 2022. URL: https://disk.yandex.ru/i/MeYOKmKmA-viWw.

[13] Трушин Д. В. Консультация к проекту «Нейросети с нуля», часть 2. 2022. URL: https://disk.yandex.ru/i/UR37foU7v8H9Gw.