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: Нейросети с нуля

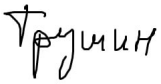
Выполнил:
Студент группы БПМИ218 — Подпись — А.В.Скрибченко (И.О.Фамилия)

23.05.2023
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 25.05. 2023 — Оценка (по 10-ти бальной шкале) — Подпись

Москва 2023

Содержание

Аннотация

В ходе данной курсовой работы будет реализована и протестирована библиотека для работы с нейросетями. Библиотека должна позволять конструировать полносвязные нейросети различной структуры и обучать их с использованием разных методов градиентного спуска.

1 Введение

В современном мире важность нейросетей не подвергается никакому сомнению. Область их применения крайне велика, но чаще всего нейросети используются для решения задач классификации, прогнозирования, распознавания объектов. Главной задачей данного курсового проекта является изучение принципа работы нейросетей с их дальнейшей имплементацией на языке C++ в виде библиотеки. Библиотека должна предоставлять возможность создать нейросеть с указанной архитектурой, а также обучить её с использованием различных модификаций градиентного спуска. В задачи проекта также входит тестирование библиотеки, написание сопроводительной документации и изложение в отчёте особенностей имплементации и архитектуры программы.

Более детальное рассмотрение поставленной задачи приводит к следующему плану работы над курсовым проектом:

1. Изучение теоретических материалов об устройстве и работе нейросетей
2. Изучение понятия градиентного спуска, а также его различных модификаций
3. Разработка программного интерфейса и проектирование архитектуры программы
4. Программная имплементация нейросети
5. Тестирование программы
6. Написание отчёта о выполненной работе

Далее отчёт структурирован следующим образом: теоретические сведения, особенности реализации и результаты тестирования программы. В первом разделе описывается математическая модель нейросети, приводятся используемые формулы. В разделе об особенностях программной реализации подробно описывается программный интерфейс, демонстрируется диаграмма классов и их взаимодействие. В разделе о тестировании программы рассказывается о тестировании компонентов библиотеки и её применении для обучения на конкретных данных.

2 Описание функциональных и нефункциональных требований к программному проекту

2.1 Описание функциональных требований к программному проекту

Библиотека должна предоставлять пользователю класс, позволяющий создать нейросеть и обучить её.

При создании объекта этого класса пользователь задаёт конструкцую нейросети. Как известно, нейросеть обычно представляет собой набор слоёв, поэтому при создании объекта вышеупомянутого класса пользователю необходимо указать размеры всех слоёв и их функции активации. Библиотека должна поддерживать следующие функции активации: сигмоида, ReLU, LeakyReLU.

У пользователя должна иметься возможность указать функцию штрафа и метод градиентного спуска, которые будут использоваться при обучении сети. Поддерживаемые функции штрафа: Mean Squared Error, Mean Absolute, Error. Поддерживаемые методы градиентного спуска: полный градиентный спуск, стохастический градиентный спуск, стохастический градиентный спуск с инерцией, Adam.

Объект класса нейросети должен обладать методом для её обучения. Метод обучения должен принимать следующие аргументы: набор векторов входных данных для обучения, набор векторов истинных значений приближаемой функции на этих входных данных и максимальное число итераций обучения.

Ещё один метод, которым должна обладать нейросеть - это метод предсказания. Он принимает вектор входных данных и возвращает предсказанный нейросетью результат.

2.2 Описание нефункциональных требований к программному проекту

Для написания библиотеки использован язык программирования C++ стандарта C++17. Для компиляции файлов проекта использован компилятор gcc версии 11.3.0. Сборка проекта осуществлялась при помощи системы сборки CMake. В проекте используется style guide компании Google, а для форматирования кода используется программа clang-format.

В процессе разработки использовалась система контроля версий git, код проекта выложен на github. Для работы с матрицами использовалась библиотека линейной алгебры Eigen [1] версии 3.4.0.

3 Имплементация нейросети с нуля

3.1 Необходимые теоретические сведения

3.1.1 Общая информация об устройстве нейросетей

Пусть у нас есть неизвестное отображение $G\colon \mathbb{R}^n \to \mathbb{R}^m$. При этом мы знаем значение отображения на векторах $x_1, \ldots, x_k$ и эти значения равны $y_1, \ldots, y_k$. Наша задача - приблизить это отображение G. Приближать его мы будем в классе полносвязных нейросетей.

Нейросеть представляет собой некоторое дифференцируемое отображение $F : \mathbb{R}^n \to \mathbb{R}^m$, параметризованное набором параметров θ. Обучение нейросети – это процесс изменения этих параметров с учётом некоторой поставленной задачи.

Для удобства далее будем называть набор $x_1, \ldots, x_k$ входными данными, а набор $y_1, \ldots, y_k$ ответами.

Нейросеть строится из слоёв. Каждый слой содержит в себе следующую информацию: матрицу A, столбец b и функцию активации σ.

Определение 1. Функция активации $\sigma\colon \mathbb{R} \to \mathbb{R}$ – некоторое нелинейное отображение, применяемое к вектору покоординатно.

Каждый слой принимает на вход вектор x, а затем передаёт его следующему слою выполнив над ним следующие операции:

$$x \mapsto Ax + b \mapsto \sigma(Ax + b) \tag{1}$$

Пусть входной вектор $x \in \mathbb{R}^l$. Тогда $A \in \mathrm{Mat}_{p\times l}$, $b \in \mathbb{R}^p$ и $\sigma(Ax + b) \in \mathbb{R}^p$. Число p характеризует рамер результирующего вектора после применения вышеописанных операций.

Если рассматривать каждый слой как отображение, то отображение F представляет собой их композицию, а матрица A и вектор b являются настраиваемыми параметрами, которые изменяются в ходе обучения нейросети.

Определение 2. Предсказанием нейросети будем называть вектор $w_i = F(x_i)$ из $\mathbb{R}^m$.

Определение 3. Функция штрафа – это отображение $\phi : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$, являющееся характеристикой отклонения предсказания нейросети на каком-то векторе из входных данных от соответствующего ему ответа. Пример: $\phi(w, y) = ||x - y||$.

Рассмотрим следующую функцию:

$$L(\theta) = \frac{1}{k}\sum_{i=1}^{k} \phi(w_i, y_i)$$

Данная функция зависит от θ, так как $w_i = F(x_i)$ и отображение F зависит от параметров θ.

Суть обучения нейросети заключается в минимизации функции $L(\theta)$. Для этого используется понятие градиента.

Определение 4. Пусть $f = f(x_1, \ldots, x_n)$. Градиентом функции $f : \mathbb{R}^n \to \mathbb{R}$ называется вектор из частных производных этой функции. Обозначение: ∇f. Вектор $-\nabla f$ будем называть антиградиентом.

Известно, что градиент показывает направление наискорейшего роста функции, а антиградиент – направлением наискорейшего убывания. Таким образом, для минимизации функции L нам нужно вычислить её градиент и обновить параметры слоёв нейросети в соотвествии с антиградиентом. Этот подход называется методом градиентного спуска.

Пусть A_i и b_i - параметры i-го слоя. Введём следующие понятия и обозначения:

1. $\nabla A_i = \frac{\partial L}{\partial A_i}(\theta)$ - направление вдоль градиента для параметра A_i
2. $\nabla b_i = \frac{\partial L}{\partial b_i}(\theta)$ - направление вдоль градиента для параметра b_i
3. λ - “скорость обучения”, то есть параметр, характеризующий насколько быстро мы двигаемся в направлении антиградиента. Этот параметр, как правило, подбирается эмпирически.

Используя эти обозначения, обновление параметров можно записать в виде:

$$\begin{aligned} A_i' &= A_i - \lambda\nabla A_i \\ b_i' &= b_i - \lambda\nabla b_i \end{aligned} \tag{2}$$

Для вычисления градиента и обновления параметров используется метод обратного распространения ошибки (backward propagation). Его суть заключается в следующем: в начале для вектора x_i из входных данных вычисляется предсказание $w_i = F(x_i)$ при помощи "проталкивания вперёд"по формулам 1. Затем, используя эту информацию мы вычисляем вектор U, который и будет содержать информацию об ошибке:

$$U = \frac{\partial \phi}{\partial w}(w_i, y_i) \tag{3}$$

Этот вектор содержит информацию об ошибке w_i, и его следует передать последнему слою нейросети, где тот используется для следующих вычислений:

$$\frac{\partial \phi}{\partial A_i}(\theta) = \sigma'(A_i x + b_i)Ux^T$$

$$\frac{\partial \phi}{\partial b_i}(\theta) = \sigma'(A_i x + b_i)U$$

$$U' = U^T\sigma'(A_i x + b_i)A_i \tag{4}$$

Здесь x - входной вектор для слоя, σ' - диагональная матрица из производных функции активации по координатам вектора $A_i x + b_i$. Вектор U' "проталкивается" предыщему слою в качестве вектора U, и на этом слое производятся аналогичные вычисления. "Проталкивание" происходит вплоть до первого слоя в нейросети. Данный процесс повторяется для всех векторов входых данных. Далее вычисляются ∇A_i и ∇b_i по правилу дифференцирования суммы:

$$\begin{aligned} \nabla A_i &= \frac{1}{k} \sum_{i=1}^{k} \frac{\partial \phi}{\partial A_i}(\theta) \\ \nabla b_i &= \frac{1}{k} \sum_{i=1}^{k} \frac{\partial \phi}{\partial b_i}(\theta) \end{aligned} \tag{5}$$

После этого производится обновление параметров по формулам 2.

Вышеописанный способ обновления параметров не является единственным. Далее описываются разновидности градиентного спуска, реализованные в рамках данной курсовой работы.

3.1.2 Виды градиентного спуска

Часто оказывается так, что вычисление градиента на всех векторах входных данных оказывается слишком трудоёмким. В таком случае можно производить вычисление градиента лишь на одном векторе:

$$\nabla A_i = \frac{\partial \phi}{\partial A_i}(\theta)$$

$$\nabla b_i = \frac{\partial \phi}{\partial b_i}(\theta)$$

Этот вид градиентного спуска называется стохастическим. Если мы вычисляем градиент на всей обучающей выборке, то при обновлении параметров мы гарантированно движемся в сторону минимума оптимизируемой функции. Со стохастическим градиентным спуском это не так, ведь в этом случае при каждом обновлении параметров мы будем двигаться в сторону, оптимальную с точки зрения значения функции на одном рассматриваемом объекте. Поэтому для достижения минимума может потребоваться большее количество итераций обучения. Чтобы уменьшить данный показатель, можно вычислять градиент не на одном объекте из обучающей выборки, а на каком-то её подмножестве размера k по формулам 5.Такой метод изменения параметров называют mini-batch gradient descent, а сами подмножества, на векторах из которых вычисляется градиент, называются батчами.

В стохастическом градиентом спуске может оказаться так, что направление антиградиента резко меняется от шага к шагу, что ведёт к большему числу итераций обучения. Чтобы этого избежать, можно производить обновления параметров учитывая не только текущий градиент, но и предыдущие значения. Делается это при помощи вектора инерции, вычисляемого на каждой итерации обучения.

Введём обозначеня. Пусть w_j - параметры нейросети на j-ой итерации обучения. Тогда вектор инерции h_j на каждой итерации вычисляется по формулам:

$$\begin{aligned} h_0 &= 0 \\ h_j &= \beta h_{j-1} + \lambda \frac{1}{k} \sum_{i=1}^{k} \frac{\partial \phi}{\partial w}(w_{j-1}) \end{aligned} \tag{6}$$

Здесь λ - скорость обучения, β - параметр, определяющий скорость затухания градиентов из предыдущих итераций. Вычислив h_j, мы обновляем параметры нейросети следующим образом:

$$w_j = w_{j-1} - h_j \tag{7}$$

Описанный вид градиентного спуска обычно называется стохастический градиентный спуск с инерцией (stochastic gradient descent with momentum). Более подробная информация о нём содержится в статье [2].

Также в рамках данной курсовой работы был реализован алгоритм Adam. Он заметно отличается от вышеописанных видов градиентного спуска. Adam вычисляет скорость обучения для каждого параметра нейросети с помощью оценки первых и вторых моментов градиентов. В деталях он описывается в статье [3], здесь же приведём алгоритм и основные формулы.

На итерации обучения под номером t происходит следующее:

1. Вычисление градиента на текущем шаге:

$$g_t = \frac{1}{k}\sum_{i=1}^{k}\frac{\partial\phi}{\partial w}(w_{t-1})$$

2. Вычисление смещённой оценки математического ожидания параметров на текущем шаге:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \tag{8}$$

3. Вычисление смещённнной оценки второго момента параметров на текущем шаге:

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \tag{9}$$

4. Вычисление скорости обучения на текущем шаге:

$$\alpha_t = \alpha * \frac{\sqrt{1-\beta_2^t}}{1-\beta_1^t} \tag{10}$$

5. Обновление параметров нейросети:

$$w_t = w_{t-1} - \alpha_t \frac{m_t}{\sqrt{v_t}+\epsilon} \tag{11}$$

α, β_1, β_2 и ϵ являются настраевыми параметрами алгоритма. Также стоит отметить, что градиент g_t, как правило, вычисляется на батче, поэтому размер батча тоже является одним из параметров данного алгоритма.

3.2 Описание программы

3.2.1 Общая информация о построении библиотеки

В результате анализа функциональных требований и теоретических сведений были обозначены следующие сущности:

- слой нейросети
- блок функции ошибок
- оптимизатор градиентного спуска
- сама нейросеть, состоящая из вышеупомянутых компонентов

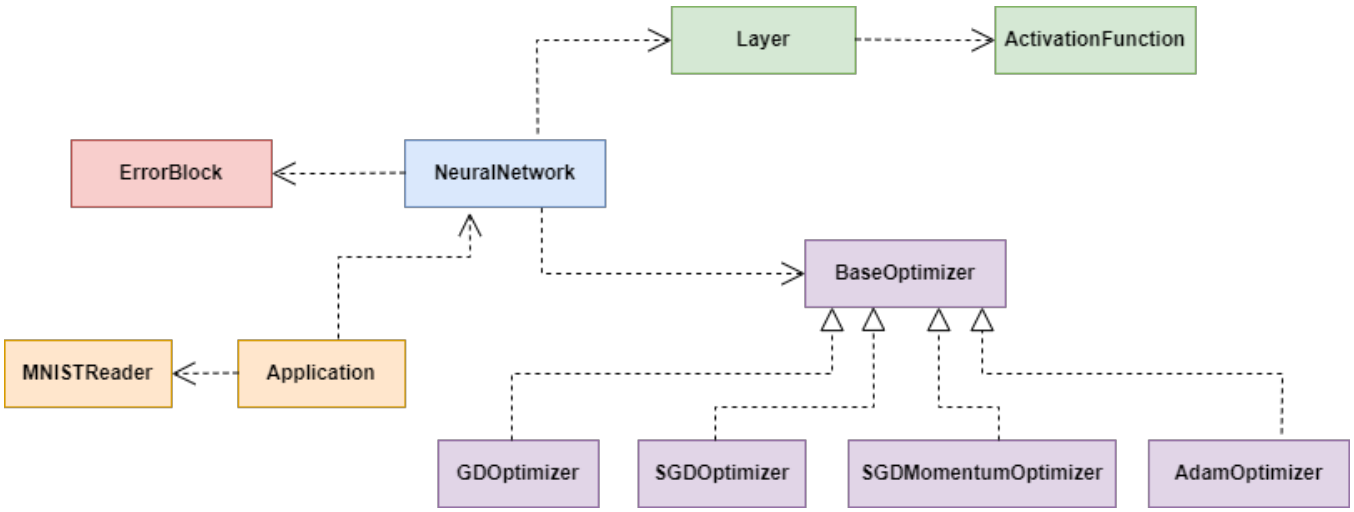
Класс Layer используется для представления в программе слоя нейросети. Данный класс содержит в себе параметры нейросети и предоставляет интерфейс для «проталкивания» векторов через этот слой вперёд и назад.

Класс ErrorBlock используется для представления в программе блока функции ошибок. Назначение этого класса – вычисление значения функции ошибок и градиента функции ошибок в соответствии с заданными векторами.

Описанные в теоретической части виды градиентного спуска реализованы в виде классов GDOptimizer, SGDOptimizer, SGDMomentumOptimizer и AdamOptimizer. Все эти классы являются наследниками абстрактного класса BaseOptimizer, который предоставляет для них общий интерфейс обучения.

Вышеописанные классы являются компонентами, из которых строится нейросеть, представленная в программе классом NeuralNetwork. Кроме того, данный класс предоставляет пользователю функции для обучения нейросети и для предсказания.

На рисунке ниже изображена диаграмма классов, присутствующих в программе:



Заметим, что на диаграмме также присутствуют классы MNISTReader и Application. Класс MNISTReader нужен для считывания информации из файлов, в которых находятся пиксели и лейблы картинок из MNIST (подробнее об этом рассказывается в разделе о тестировании библиотеки). Он не является частью нейросети, но может рассматриваться как дополнение к созданной библиотеке. Класс Application демонстрирует некоторые сценарии использования библиотеки. Использование этих классов будет продемонстрировано в разделе отчёта, посвященном тестированию.

Подробное описание структуры вышеупомянутых классов содержится в следующих разделах. Важно отметить, что все вычисления с объектами линейной алгебры в данных классах производилось с помощью библиотеки Eigen. В частности, данная библиотека предоставляет классы MatrixXd и VectorXd для матриц и векторов соответственно. Для этих классов в коде используются using-и. Классу MatrixXd соответствует название Matrix, классу VectorXd соответствует название Vector (в дальнейшем повествовании будут использоваться именно эти названия).

3.2.2 Класс ActivationFunction

Для хранения функции активации используется вспомогательный класс ActivationFunction. Этот класс хранит в себе две скалярных функции типа `std::function<double(double)>`. Одна из них обозначает саму функцию активации, другая - её производную. Объект данного класса конструируется при помощи переменной типа `enum FunctionType {Sigmoid, Relu, LeakyRelu}`.

- `FunctionType::Sigmoid` соответствует функции активации сигмоида: $\sigma(x) = \frac{1}{1+e^{-x}}$
- `FunctionType::Relu` соответствует функции активации ReLU: $\sigma(x) = max(0, x)$
- `FunctionType::LeakyRelu` соответствует функции активации LeakyReLU: $\sigma(x) = max(0.01x, x)$

3.2.3 Класс Layer

Класс Layer содержит в себе параметры слоя и предоставляет функции для «проталкивания» вектора в обоих направлениях. Созданная нейросеть внутри себя хранит вектор объектов класса Layer. Обучение нейросети означает изменение параметров слоёв из данного вектора. Предсказание нейросети означает проталкивание вперёд входного вектора через эти слои.

Поля класса Layer (все поля являются приватными):

- `Matrix A_` - параметры слоя.
- `Vector b_` - параметры слоя.
- `ActivationFunction activation_` - функция активации слоя.
- `Vector input_`- входные данные слоя, необходимые для метода обратного распространения ошибки.

Функции и методы класса `Layer` (все являются публичными):

- `Vector PushForward(const Vector& x)` - "проталкивание вперёд" , т.е применение формулы 1 ко входу слоя с его дальнейшим запоминанием в поле `input_`.
- `Vector PushForwardPredict(const Vector& x) const` - константная версия предыдущего метода (без запоминания в поле `input_`).
- `Vector PushBackwards(const Vector& u, Matrix* grad_A_curr, Vector* grad_b_curr)` - "проталкивание назад", функция возвращает вектор 4. В данную функцию можно передать два указателя на объекты класса Matrix и Vector. К этим объектам будут прибавлены найденные в процессе градиенты параметров слоя.
- `void ShiftParams(const Matrix& A_update, const Vector& b_update)` - прибавление ко внутренним параметрам A_ и b_ величин A_update и b_update соответственно.
- `size_t GetInputSize() const` - получить размер входного вектора для слоя (количество столбцов у A_).
- `size_t GetOutputSize() const` - получить размер выходного вектора для слоя (количество строк у A_ или размер вектора b_).
- `Matrix GetMatrixParams() const` - получить параметры слоя, представленные в матрице A_.
- `Matrix GetVectorParams() const` - получить параметры слоя, представленные в векторе b_.

Конструктор класса `Layer` имеет следующую сигнатуру:
`Layer(size_t input_size, size_t output_size, FunctionType func)`.
Он инициализирует параметры слоя. Вектор b_ инициализируется нулевым вектором размера output_size, Матрица A_ инициализируется случайными числами из нормального распределения со средним 0 и дисперсией 2 / input_size. Подобная инициализация помогает избегать ситуаций, когда значения вычисленных градиентов получаются слишком большими или слишком маленькими. Подробнее об этом рассказывается в статье [4].

3.2.4 Класс ErrorBlock

Класс ErrorBlock предоставляет механизм функции ошибок. Он является обязательной частью нейросети и используется при её обучении для вычисления градиента функции ошибок.

Конструктор данного класса имеет сигнатуру `ErrorBlock(ErrorType err)`, где `ErrorType` - перечисляемый тип со значениями `{MSE, MAE}` (Mean Squared Error, Mean Absolute Error).

Поля этого класса (все поля являются приватными):

- `std::function<double(const Vector&, const Vector&)> error_func_` - функция ошибки
- `std::function<Vector(const Vector&, const Vector&)> gradient_` - градиент функции ошибки

Функции данного класса (все функции являются публичными):

- `double GetErrorValue(const Vector& input, const Vector& expected) const` - получить значение функции ошибки.
- `Vector GetGradientValue(const Vector& input, const Vector& expected) const` - получить вектор 3.

3.2.5 Класс BaseOptimizer и подход к обучению сети

Класс BaseOptimizer является абстрактным классом, от которого наследуются классы оптимизаторов градиентного спуска. У класса BaseOptimizer имеется единственная виртуальная функция Train с сигнатурой

`virtual void Train(std::vector<Layer>& layers, const ErrorBlock& error_block, const std::vector<Vector>& train_input, const std::vector<Vector>& train_output, size_t max_iter_count) const`.

Функция Train принимает в качестве аргументов слои, блок ошибки, вектор входных данных и вектор выходных данных. Подразумевается, что классы, реализующие интерфейс класса BaseOptimizer, переопределяют функцию Train в зависимости от типа реализуемого алгоритма.

В каждом из реализованных алгоритмов в том или ином виде используются градиенты по параметрам нейросети, вычисленные на всех входных данных или на батче. Получение этих градиентов осуществляется с помощью метода backward propagation, описанного в теоретической части отчёта. По этой причине функция PushBackwards класса Layer принимает в качестве аргументов указатели на объекты класса Matrix и Vector: подразумевается, что данные указатели указывают на буферы для градиентов, которые создаются и используются классами оптимизаторов градиентного спуска. После того, как градиенты были вычислены, они используются для обновления параметров слоёв в соответствии с типом алгоритма.

Таким образом, процесс обучения нейросети классом-оптимизатором можно представить в виде следующих этапов:

- Инициализация необходимых буферов.
- «Проталкивание» векторов на батче (на всей выборке для полного градиентного спуска) вперёд и назад, в ходе которого в буферах накапливаются найденные градиенты.
- Обработка вычисленных градиентов в соответствии с алгоритмом.
- Обновление параметров слоёв с использованием обработанных градиентов.

Изображение данного процесса в виде диаграммы:

3.2.6 Класс GDOptimizer

Данный класс реализует полный градиентный спуск и обладает единственным полем `double learning_speed_`. По умолчанию оно имеет значение 0.1, но можно указать другое значение с помощью конструктора `GDOptimizer(double speed)`, который создаёт объект класса GDOptimizer со значением поля learning_speed_ равным speed.

Данный класс обладает следующими вспомогательными приватными методами:

- `void CalculateGradients(std::vector<Layer>* layers, std::vector<Matrix>* grads_A, std::vector<Vector>* grads_b, const ErrorBlock& error_block, const std::vector<Vector>& train_input, const std::vector<Vector>& train_output) const` - вычисление градиентов на слоях layers с их дальнейшим накоплением в буферах grads_A и grads_b.
- `void UpdateLayerParams(std::vector<Layer>* layers, const std::vector<Matrix>& grads_A, const std::vector<Vector>& grads_b, size_t vectors_count) const` - обновление параметров слоёв layers с учётом накопленных градиентов.

Буферы для градиентов представлены в виде `std::vector<Matrix> grads_A` и `std::vector<Vector> grads_b`, где grads_A[i] и grads_b[i] соответствуют накопленным градиентам для матрицы A и вектора b i-го слоя. Такой же вид буферы имеют и в других классах, реализующих разные виды градиентного спуска.

Методы CalculateGradients и UpdateLayerParams используются для обучения следующим образом:

```
size_t iter_count = 0;
while (iter_count < max_iter_count) {
    std::vector<Matrix> grads_A(layers_count);
    std::vector<Vector> grads_b(layers_count);
    CalculateGradients(&layers, &grads_A, &grads_b, error_block, train_input, train_output);
    UpdateLayerParams(&layers, grads_A, grads_b, train_input.size());
    ++iter_count;
}
```

3.2.7 Класс SGDOptimizer

Данный класс реализует стохастический градиентный спуск. Поля этого класса (все поля являются публичными):

- `double learning_speed_` - скорость обучения, по умолчанию равная 0.1.
- `size_t batch_size_` - размер батча, по умолчанию равный 1.

У этого класса есть конструктор `SGDOptimizer(size_t batch_size, double speed)`, в котором указываются желаемые значения этих полей.

Все методы данного класса являются приватными:

- `void InitializePermutation(std::vector<size_t>* perm) const` - инициализация перестановки, нужной для генерации батча
- `void CalculateGradientsOnBatch(std::vector<Layer>* layers, std::vector<Matrix>* grads_A, std::vector<Vector>* grads_b, const ErrorBlock& error_block, const std::vector<size_t>& batch_indices, const std::vector<Vector>& train_input, const std::vector<Vector>& train_output) const` - вычисление градиента на векторах с индексами из batch_indices с дальнейшим накоплением градиентов в grads_A и grads_b.
- `void UpdateLayerParams(std::vector<Layer>* layers, const std::vector<Matrix>& grads_A, const std::vector<Vector>& grads_b) const` - обновление параметров слоёв layers с учётом накопленных градиентов.

После инициализации начальной перестановки perm вышеописанные методы используются для обучения следующим образом:

```
size_t iter_count = 0;
while (iter_count < max_iter_count) {
    std::shuffle(perm.begin(), perm.end(), mt);
    std::vector<Matrix> grads_A(layers_count);
    std::vector<Vector> grads_b(layers_count);
    CalculateGradientsOnBatch(&layers, &grads_A, &grads_b, error_block, perm, train_input,
                              train_output);
    UpdateLayerParams(&layers, grads_A, grads_b);
    ++iter_count;
}
```

3.2.8 Класс SGDMomentumOptimizer

Этот класс предназначен для реализации стохастического градиентного спуска с инерцией. Поля и конструкторы данного класса аналогичны классу SGDOptimizer.

Вспомогательные методы данного класса:

1. `void InitializePermutation(std::vector<size_t>* perm) const` - инициализация перестановки для перемешивания индексов.
2. `void InitializeInertionBuffers(const std::vector<Layer>& layers, std::vector<Matrix>* inertions_A, std::vector<Vector>* inertions_b) const` - инициализация буферов под вектор инерции. Структура этих буферов аналогична структуре буферов под градиенты.

3. `void CalculateGradientsOnBatch(std::vector<Layer>* layers, std::vector<Matrix>* grads_A, std::vector<Vector>* grads_b, const ErrorBlock& error_block, const std::vector<size_t>& batch_indices, const std::vector<Vector>& train_input, const std::vector<Vector>& train_output) const` - вычисление градиентов на батче аналогично классу SGDOptimizer

4. `void UpdateInertions(const std::vector<Matrix>& grads_A, const std::vector<Vector>& grads_b, std::vector<Matrix>* inertions_A, std::vector<Vector>* inertions_b) const` - обновление вектора инерции по формулам 6.

5. `void UpdateLayerParams(std::vector<Layer>* layers, const std::vector<Matrix>& inertions_A, const std::vector<Vector>& inertions_b) const` - обновление параметров слоёв layers по формулам 7.

После инициализации начальной перестановки perm и буферов inertions_A и inertions_b вышеописанные методы используются для обучения следующим образом:

```
size_t iter_count = 0;
while (iter_count < max_iter_count) {
    std::shuffle(perm.begin(), perm.end(), mt);
    std::vector<Matrix> grads_A(layers_count);
    std::vector<Vector> grads_b(layers_count);
    CalculateGradientsOnBatch(&layers, &grads_A, &grads_b, error_block, perm, train_input,
                              train_output);
    UpdateInertions(grads_A, grads_b, &inertions_A, &inertions_b);
    UpdateLayerParams(&layers, inertions_A, inertions_b);
    ++iter_count;
}
```

3.2.9 Класс AdamOptimizer

Этот класс реализует алгоритм Adam. Поля данного класса соответствуют настраевым параметрам данного алгоритма:

- `double learning_speed_` соответствует параметру α и по умолчанию равно 0.0001
- `double beta1_` соответствует параметру β_1 и по умолчанию равно 0.9
- `double beta2_` соответствует параметру β_2 и по умолчанию равно 0.999
- `double eps_` соответствует параметру ϵ и по умолчанию равно 10e-8.
- `size_t batch_size_` соответствует размеру батча для вычисления градиента.

Значения параметров устанавливаются с помощью конструктора `AdamOptimizer(double learning_speed, double beta1, double beta2, double eps, size_t batch_size)`.

Вспомогательные методы класса AdamOptimizer:

- `void InitializePermutation(std::vector<size_t>* perm) const` - инициализация перестановки для перемешивания индексов.
- `void InitializeMomentsBuffers(const std::vector<Layer>& layers, std::vector<Matrix>* current_m_A, std::vector<Vector>* current_m_b, std::vector<Matrix>* current_v_A, std::vector<Vector>* current_v_b) const` - инициализация буферов под моменты m_t и v_t.
- `void CalculateGradientsOnBatch(std::vector<Layer>* layers, std::vector<Matrix>* grads_A, std::vector<Vector>* grads_b, const ErrorBlock& error_block, const std::vector<size_t>& batch_indices, const std::vector<Vector>& train_input, const std::vector<Vector>& train_output) const` - вычисление градиента на батче.
- `void UpdateMoments(const std::vector<Matrix>& grads_A, const std::vector<Vector>& grads_b, std::vector<Matrix>* current_m_A, std::vector<Vector>* current_m_b, std::vector<Matrix>* current_v_A, std::vector<Vector>* current_v_b) const` - вычисление моментов на текущей итерации по формулам 8 и 9.

- `void UpdateLayerParams(std::vector<Layer>* layers, double current_learning_speed, const std::vector<Matrix>& current_m_A, const std::vector<Vector>& current_m_b, const std::vector<Matrix>& current_v_A, const std::vector<Vector>& current_v_b) const` - обновление параметров слоёв layers по формуле 11.

После инициализации необходимых буферов, методы из списка выше используются для обучения следующим способом:

```
size_t iter_count = 0;
while (iter_count < max_iter_count) {
    std::shuffle(perm.begin(), perm.end(), mt);
    std::vector<Matrix> grads_A(layers_count);
    std::vector<Vector> grads_b(layers_count);
    CalculateGradientsOnBatch(&layers, &grads_A, &grads_b, error_block, perm, train_input,
                              train_output);
    UpdateMoments(grads_A, grads_b, &current_m_A, &current_m_b, &current_v_A, &current_v_b);
    current_learning_speed = learning_speed_ * sqrt(1 - pow(beta2_, iter_count + 1)) /
                             (1 - pow(beta1_, iter_count + 1));
    UpdateLayerParams(&layers, current_learning_speed, current_m_A, current_m_b, current_v_A,
                      current_v_b);
    ++iter_count;
}
```

Значение current_learning_speed вычисляется по формуле 10.

3.2.10 Класс NeuralNetwork

Класс NeuralNetwork объединяет в себе все ранее упомянутые компоненты и предоставляет пользователю интерфейс для их использования. У класса NeuralNetwork имеются следующие поля, все из которых являются приватными:

- `std::vector<Layer> layers_` - набор слоёв нейросети.
- `ErrorBlock error_block_` - блок функции ошибок.
- `std::unique_ptr<BaseOptimizer> optimizer_` - указатель на оптимизатор градиентного спуска.

У данного класса есть два конструктора:

- `NeuralNetwork(std::initializer_list<size_t> layers_sizes, std::initializer_list<FunctionType> functions);` - этот конструктор создаёт слои нейросети по заданным размерам и функциям активации. В этом случае созданная нейросеть имеет блок ошибок с функцией MSE и оптимизатор AdamOptimizer.
- `NeuralNetwork(std::initializer_list<size_t> layers_sizes, std::initializer_list<FunctionType> functions, T optimizer, ErrorType type)` - этот конструктор создаёт слои нейросети по заданным размерам и функциям активации, а также инициализирует блок ошибок и оптимизатор переданными значениями.

Вспомогательные функции и методы класса NeuralNetwork:

- `void CreateLayers(const std::initializer_list<size_t>& layers_sizes, const std::initializer_list<FunctionType>& functions)` - создание слоёв по переданным размерам и функциям активации. Этот метод вызывается в обоих конструкторах класса NeuralNetwork.
- `std::vector<Vector> TransformData(const std::vector<std::vector<double»& data) const` - преобразование вектора векторов из стандартной библиотеки к вектору векторов из библиотеки Eigen. Этот метод вызывается перед началом обучения нейросети.

Класс NeuralNetwork предоставляет следующий пользовательский интерфейс:

- `void AddNextLayer(size_t input_size, size_t output_size, FunctionType func)` - добавить слой с заданными данными в конец нейросети.
- `void SetOptimizer(T optimizer)` - сеттер для оптимизатора.
- `void SetError(ErrorType type)` - сеттер для блока ошибки.
- `void Train(const std::vector<std::vector<double»& train_input, const std::vector<std::vector<double»& train_output, size_t max_iter_count)` - обучение нейросети на заданных данных в течение max_iter_count итераций.
- `std::vector<double> Predict(const std::vector<double>& data) const` - функция для получения предсказания нейросети на конкретном векторе data.

Метод Train класса NeuralNetwork преобразует векторы с помощью функции TransformData и передаёт необходимые параметрых в оптимизаторы для обучения:

```
void NeuralNetwork::Train(const std::vector<std::vector<double>>& train_input,
const std::vector<std::vector<double>>& train_output,
                          size_t max_iter_count) {
    std::vector<Vector> x = TransformData(train_input);
    std::vector<Vector> y = TransformData(train_output);
    optimizer_->Train(layers_, error_block_, x, y, max_iter_count);
}
```

Функция Predict "проталкивает" входной вектор через слои layers_ с помощью метода PushForwardPredict класса Layer, и возвращает результат.

3.3 Тестирование

3.3.1 Модульное тестирование компонентов нейросети

Классы Layer, ActivationFunction и ErrorBlock являются основными составляющими конструкции нейросети. Внутрь данных классов заложены вычисления, от корректности которых зависит весь процесс её обучения. Для тестирования данных классов были написаны модульные тесты с использованием библиотеки Google Test [5].

Тестирование класса ActivationFunction заключалось в проверке правильности вычисления значения функции и производной для разных поддерживаемых вариантов (Sigmoid, ReLU, LeakyReLU). Написанные для этого тесты находятся в файле tests/activation-function/activation_function_tests.cpp. Их можно разделить на группы:

- SigmoidTests - тестирование сигмоиды
  - SigmoidValueCorrectness - проверка значения функции
  - SigmoidDerivativeCorrectness - проверка производной
- ReluTests - тестирование ReLU
  - ReluValueCorrectness - проверка значения функции
  - ReluDerivativeCorrectness - проверка производной
- LeakyReluTests - тестирование LeakyReLU
  - LeakyReluValueCorrectness - проверка значения функции
  - LeakyReluDerivativeCorrectness - проверка производной

Аналогичным образом устроено тестирование класса ErrorBlock. Тесты находятся в файле tests/error-block/error_block_tests.cpp. Они проверяют правильность вычисления значения и градиента функции для MSE и MAE.

- MSETests - группа тестов MSE

  - MSEValueCorrectness - проверка значения функции
  - MSEGradientCorrectness - проверка градиента

- MAETests - группа тестов MAE
  - MAEValueCorrectness - проверка значения функции
  - MAEGradientCorrectness - проверка градиента

При тестировании класса Layer было важно проверить правильность "проталкивания" вектора вперёд (функция PushForward) и назад (функция PushBackwards), так как от этого зависит работоспособность реализации метода backward propagation. Тесты класса Layer находятся в файле tests/layer/layer_tests.cpp. Тест Basic проверяет работу PushForward и PushBackwards для случая одного слоя, а тест Multiple проверяет их работу для случая нескольких слоёв.

3.3.2 Обучение нейросети для вычисления XOR двух чисел

Для демонстрации конкретных сценариев работы библиотеки используется класс Application. Этот класс имеет публичные методы `void Run1()`, `void Run2()`, `void Run3()`, `void Run4()` и `void Run5()`. Первые четыре из них демонстрируют обучение нейросети для вычисления XOR двух чисел с использованием разных оптимизаторов градиентного спуска.

У класса Application есть поля `std::vector<std::vector<double» xor_train_input_` и `std::vector<std::vector<double» xor_train_output_` , в которых хранятся данные для обучения. Они инициализируются с помощью метода `void InitializeXORData()`. Данный метод инициализирует вектор xor_train_input_ двумястами случайными парами из нулей и единиц. В вектор xor_train_output_ записываются соответствующие этим парам значения функции XOR.

1. `void Run1()`

   Этот метод демонстрирует использование полного градиентного спуска. После инициализации необходимых данных код выглядит следующим образом:

```
NeuralNetwork network({2, 4, 4, 4, 1},
                          {
                              FunctionType::LeakyRelu,
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                          },
                          GDOptimizer(), ErrorType::MSE);
    network.Train(xor_train_input_, xor_train_output_, 1000);
    std::vector<double> prediction = network.Predict({0, 0});
    std::cout << "0 0 " << prediction[0] << "\n";
    prediction = network.Predict({0, 1});
    std::cout << "0 1 " << prediction[0] << "\n";
    prediction = network.Predict({1, 0});
    std::cout << "1 0 " << prediction[0] << "\n";
    prediction = network.Predict({1, 1});
    std::cout << "1 1 " << prediction[0] << "\n";
```

   Сначала мы создаём объект нейросети с указанием размеров слоёв, функций активаций, оптимизатора и функции ошибки. Далее мы запускаем процесс обучения на имеющиъся данных, после чего предсказываем значения функции XOR для всех пар из нулей и единиц. Результат:

```
0 0 0.0627224
0 1 0.971313
1 0 0.911356
1 1 0.0627224
```

2. `void Run2()`

   Здесь используется алгоритм Adam:

```
NeuralNetwork network({2, 4, 4, 4, 1},
                          {
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                          },
                          AdamOptimizer(0.001, 0.9, 0.999, 10e-8, 8), ErrorType::MSE);
network.Train(xor_train_input_, xor_train_output_, 10000);
```

   Предсказание результатов выполняется аналогично предыдущему пункту. Результат:

```
0 0 0.0146757
0 1 0.99423
1 0 0.98532
1 1 0.0140515
```

3. `void Run3()`

   Демонстрация использования SGDOptimizer:

```
NeuralNetwork network({2, 4, 4, 4, 1},
                          {
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                          },
                          SGDOptimizer(16, 0.5), ErrorType::MSE);
network.Train(xor_train_input_, xor_train_output_, 5000);
```

   Результат:

```
0 0 0.0206889
0 1 0.973193
1 0 0.973193
1 1 0.0329795
```

4. `void Run4()` Использование SGDMomentumOptimizer:

```
NeuralNetwork network({2, 4, 4, 4, 1},
                          {
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                              FunctionType::Sigmoid,
                          },
                          SGDMomentumOptimizer(16, 0.1, 0.4), ErrorType::MSE);
network.Train(xor_train_input_, xor_train_output_, 5000);
```

   Результат:

```
0 0 0.0321191
0 1 0.968076
1 0 0.966509
1 1 0.0321267
```

3.3.3 MNIST и класс MNISTReader

MNIST – это база данных картинок рукописных цифр. Она состоит из картинок и лейблов к ним (лейбл картинки – цифра, которая на самом деле на ней изображена). Картинки представляют собой набор из 28x28 пикселей, каждый из которых имеет значение от 0 до 255. Значение 0 соответствует чёрному пикселю, 255 - белому пикселю.

Загрузить базу данных MNIST можно с ресурса [6]. Загруженные данные сразу разделены на обучающую выборку (60000 картинок) и тестовую выборку (10000 данных). Картинки и лейблы находятся в файлах со специальным форматом, описание которого находится на этом же ресурсе. Для считывания этого формата был написан класс MNISTReader.

Класс MNISTReader имеет конструктор со следующей сигнатурой:

```
MNISTReader(const std::string& images_filepath, const std::string& labels_filepath)
```

images_filepath – путь к файлу с картинками, labels_filepath – путь к файлу с лейблами.

Для считывания картинок у данного класса есть функция `std::vector<std::vector<double» GetImagesData() const`. Для считывания лейблов используется функция `std::vector<double> GetLabelsData() const`.

3.3.4 Обучение нейросети на базе данных MNIST

Входные данные для нейросети – набор картинок. Каждая картинка состоит из 28x28=784 пикселей. Значит, первый слой нейросети должен принимать на вход вектор размера 784.

Нам нужно, чтобы обученная нейросеть по полученной картинке умела предсказывать нарисованную на ней цифру. Может показаться, что в таком случае последний слой нейросети должен в качестве ответа выдавать какое-то число от 0 до 9. Однако у такой модели есть проблема: если на картинке изображена, к примеру, цифра 7, то нейросеть может начать предсказывать по модулю близкие значения (например, 6, или 8), ведь это может быть выгодно с точки зрения минимизации функции ошибки. На практике же разница между предсказанной цифрой и истинным результатом нас не волнует – нам важно только наличие или отсутствие совпадения. Таким образом, перед нами стоит задача классификации, а значит выходные данные последнего слоя нейросети можно представить в виде вектора v размера 10, где значение v[i] обозначает "вероятность" того, что на картинке изображена цифра i.

Метод `void InitializeMNISTData()` производит считывание обучающих и тестовых данных из файлов. Теперь с ними нужно выполнить следующие действия:

1. Пиксели картинок нужно поделить на 255.0. Это делается для того, чтобы избежать чрезмерно больших значений градиента в течение обучения. Для данного процесса используется функция `std::vector<std::vector<double» NormalizeMNISTImages( const std::vector<std::vector<double»& images)`.

2. Каждый лейбл нужно преобразовать в вектор v, такой что v[i] = 1 если лейбл равен i и v[i] = 0 иначе. Для этой операции используется функция `std::vector<std::vector<double» OneHotEncodeMNISTLabels(const std::vector<double>& train_labels)`.

После того, как обработка данных была выполнена, можно переходить к обучению (это сценарий из метода `void Run5()` класса Application):

```
auto train_input = NormalizeMNISTImages(mnist_train_images_);
auto train_output = OneHotEncodeMNISTLabels(mnist_train_labels_);
NeuralNetwork network({784, 256, 10},
                      {
                          FunctionType::Relu,
                          FunctionType::Sigmoid,
                      },
                      AdamOptimizer(0.0001, 0.9, 0.999, 10e-8, 4), ErrorType::MSE);
network.Train(train_input, train_output, 4000);
auto test_images_normalized = NormalizeMNISTImages(mnist_test_images_);
CalculateMNISTAccuracy(test_images_normalized, mnist_test_labels_, network);
```

В конце обучения вызывается функция `void CalculateMNISTAccuracy(const std::vector<std::vector<double»& images, const std::vector<double>& labels, const NeuralNetwork& network) const`. Эта функция

с помощью нейросети network предсказывает цифру на каждой картинке из images и сравнивает предсказанное значение с истинным результатом из labels. После рассмотрения всех картинок из images данная функция вычисляет точность обученной нейросети как количество совпадений, делённое на размер рассмотренной выборки.

Построенная нейросеть достигла точности в 93,13% на тестовых данных.

Список литературы

[1] Eigen. https://eigen.tuxfamily.org/index.php?title=Main_Page. (дата обр. 23.05.2023).

[2] An overview of gradient descent optimization algorithms. https://www.ruder.io/optimizing-gradient-descent/. (дата обр. 23.05.2023).

[3] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, 2015.

[5] Googletest user's guide | googletest. https://google.github.io/googletest/. (дата обр. 23.05.2023).

[6] Mnist handwritten digit database, yann lecun, corinna cortes and chris burges. http://yann.lecun.com/exdb/mnist/. (дата обр. 23.05.2023).