Министерство науки и высшего образования Российской Федерации Федеральное
государственное автономное образовательное
учреждение высшего образования
“Национальный исследовательский Нижегородский государственный университет им.
Н.И. Лобачевского”
Институт информационных технологий, математики и механики

Отчет по работе
«Коллективная разработка библиотеки для вывода результатов сверточных нейронных сетей»

Выполнили:
студенты группы 3822Б1ПР2
Титов С. М., Сорокин А. А.,
студент группы 3822Б1ПМ1
Леонтьев Н. С.,
Руководители проекта:
преподаватели
Нестеров А. Ю., Оболенский А. А.

Нижний Новгород, 2024

Содержание

Введение....3
Постановка задачи и цели работы....5
Методы решения задачи....7
Основные определения и понятия....7
Слои в AlexNet, их алгоритмы....8
Программная реализация....14
Графовая модель....14
Реализация слоёв AlexNet....16
Структура проекта....20
Архитектура....20
Структура директорий....23
Зависимости проекта....24
Аппаратно-программное обеспечение....24
Документация и инструкции....25
Инфраструктура проекта....25
Организация работы команды....26
Результаты работы....28
Заключение....30
Список литературы....32
Приложения....33
Руководство пользователя....33

Введение

Промышленное программирование — это разработка приложений, ориентированных на реальные бизнес-потребности и практические задачи в разных областях и сферах жизни. Оно обычно связано с большими проектами, которые реализуют крупные приложения со сложной логикой. Для ускорения процесса, а также для улучшения качества разработки такие проекты реализует команда. В таких условиях очень важно обеспечивать слаженное управление коллективом разработчиков, согласованность командных действий, стабильное взаимодействие внутри команды. Примерами продуктов промышленного программирования могут послужить социальные сети, банковские приложения, онлайн магазины и т.п. В настоящее время очень популярным направлением промышленного программирования является разработка многослойных нейронных сетей. Данный тип нейронных сетей проникает во все сферы жизни человека. Генерирование текста и изображений, распознавание объектов на изображениях, распознавание голоса, прогнозирование вероятного будущего на основе анализа данных в различных сферах - это основные области применения многослойных нейронных сетей. Актуальность этого направления промышленного программирования очевидна. Поэтому именно его наша команда выбрала в качестве объекта разработки.

Многослойные нейронные сети являются основополагающим инструментом для обработки сложных данных в сфере глубокого обучения. Они способны самостоятельно извлекать из входных данных набор признаков и на их основе принимать различные решения. Благодаря этому, многослойные нейронные сети имеют высокую гибкость и обучающую способность, что помогает им решать сложные задачи в различных областях, таких как обработка естественного языка, обработка изображений, финансовые анализы, биоинформатика и т. п. Одним из самых распространенных, востребованных и эффективных типов многослойных нейронных сетей являются сверточные нейронные сети.

Сверточная нейронная сеть (*CNN - Convolution Neural Network*) зачастую используется для обработки изображений, а именно для решения задачи распознавания и классификации. Это означает, что обученная сверточная нейронная сеть должна уметь принять входное изображение и сопоставить ему из заранее заданного набора классов один класс, который лучше остальных характеризует это изображение. Подобное поведение сети обеспечивает последовательность слоев разных типов, одни

из которых выделяют признаки, необходимые для классификации, а другие обрабатывают полученные признаки и получают наиболее вероятный класс.

Процесс разработки такого продукта как сверточная нейронная сеть сложен. Поэтому он требует создания четкого плана действий команды разработчиков, который основан на ключевых целях и задачах реализуемого проекта.

Постановка задачи и цели работы

В рамках нашего проекта, направленного на создание библиотеки для работы со свёрточными нейронными сетями (*CNN*), мы стремимся организовать процесс разработки так, чтобы он был схож с промышленными стандартами и ориентировался на командную работу. Наша команда ставит перед собой задачу не только создать высококачественное программное обеспечение, но и обеспечить его удобство и доступность для конечных пользователей. Подобно крупным предприятиям, мы акцентируем внимание на координации усилий всех участников проекта для достижения общей цели.

Основной задачей является создание возможности для запуска и использования пред обученных нейронных сетей с пользовательскими данными.

Основные цели проекта включают:

1. Организовать эффективную коммуникацию и координацию внутри команды разработчиков.
2. Создать инструменты для инференса сверточных нейронных сетей.
3. Обеспечить возможность запуска обученных *CNN* с пользовательскими входными данными.
4. Создать простой интерфейс для работы с библиотекой, обеспечивающий удобство использования.
5. Обеспечить возможность анализа результатов работы *CNN* и предоставить соответствующие метрики для оценки качества и производительности.
6. Документировать библиотеку и предоставить примеры использования для упрощения процесса обучения пользователей.

Задачи проекта:

1. Установить регулярные встречи и отчеты для мониторинга прогресса и выявления возможных проблем на ранних стадиях.
2. Обеспечить распределение задач среди членов команды с учетом их компетенций и интересов.
3. Исследование модели *AlexNet* и инструментов для работы.

4. Проектирование архитектуры библиотеки, включая выбор используемых технологий и инструментов.
5. Реализация основных компонентов библиотеки, таких как загрузка моделей *CNN*, обработка входных данных и анализ результатов.
6. Тестирование отдельных модулей и интеграционное тестирование всей библиотеки для обеспечения её корректной работы.
7. Создание документации, включая руководства пользователя и разработчика.
8. Проведение демонстраций работы библиотеки с реальными данными для проверки ее эффективности и функциональности.

Методы решения задачи

Основные определения и понятия

Для начала определимся с основными терминами, использующимися в свёрточных нейронных сетях.

Сама свёрточная нейронная сеть (*CNN*) - это архитектура, позволяющая решить задачу классификации. Она содержит слои, являющиеся по сути своей вершинами графа всей сети, принимает входные данные и имеет возможность обучаться. Для обучения используется алгоритм обратного распространения ошибки. В проекте будет использоваться лишь прямой проход данного алгоритма.

Понятие обучения нейросети берёт своё начало от нейронов в реальном мире, которые также способны обучаться. Аналогично вводится понятие функции активации: это функция, которая поэлементно применяется к входным данным.

Слой - это компонент нейронной сети, совершающий действие над входными данными и передающий результат далее в другие слои.

Формальное определение графа - структура, задающаяся набором вершин V и набором ребер E.

Граф, используемый в сверточных нейронных сетях, представляет собой структуру, которая отображает поток данных и операций в сети. Он состоит из узлов (вершин) и ребер, которые соединяют эти узлы. Каждый узел представляет собой операцию или вычислительный элемент, такой как свертка, слой субдискретизации, активация или полносвязный слой.

Входные данные проходят через граф, подвергаясь различным операциям в каждом слое, пока не достигнут конечного слоя, который представляет собой выход сети.

Эффективное построение и обучение таких графов позволяет сверточным нейронным сетям эффективно извлекать признаки из изображений или других типов данных, что делает их мощным инструментом в задачах компьютерного зрения, распознавания образов и других областях.

Слои в AlexNet, их алгоритмы

В *AlexNet* существуют такие слои, как:

- входной слой (*input layer*)

- сверточный слой (*convolution layer*)
- полносвязный слой (*fully-connected layer*)
- поэлементный слой или слой активации (*element-wise layer*)
- слой субдискретизации (*pooling layer*)
- выходной слой (*output layer*)
- выпадающий слой (*dropout layer*)
- слой нормализации (*normalization layer*)

Входной слой:

Входной слой (*Input layer*) является первым слоем в нейронной сети и предназначен для приема входных данных. Он принимает входные сигналы и передает их дальше в сеть для обработки и вычислений.

Входные данные могут требовать предварительной обработки или преобразования перед передачей их внутрь нейронной сети. Это может включать в себя нормализацию (например, приведение значений пикселей изображений к диапазону от 0 до 1) или любые другие операции, необходимые для корректной работы сети.

Входной слой также определяет ожидаемую размерность входных данных для нейронной сети. Например, для изображений это может быть размер изображения и количество каналов (например, *RGB* изображение имеет три канала), для текста это может быть количество слов в предложении и так далее.

Сверточный слой:

Основным слоем в любой сверточной нейронной сети является сверточный слой. Задача этого слоя — выполнить операцию свертки над указанным тензором. Свертка — операция последовательного применения фильтра (ядра) ко всем покрываемым им частям трехмерной матрицы. В результате операции свертки образуется тензор меньшего размера. Фильтр или ядро — это тензор коэффициентов. При применении фильтра к участку тензора происходит поэлементное произведение каждого элемента фильтра на каждый соответствующий ему покрываемый элемент тензора. Полученные результаты умножения складываются, и сумма является новым значением, которое помещается в результирующий тензор.

Операция свертки имеет 4 основных параметра: ядро свертки, шаг (*stride*), отступы (*paddings*), растяжение (*dilation*).

Ядро свертки — матрица, используемая как фильтр.

Шаг — количество элементов обрабатываемого тензора, на которое перемещается фильтр от предыдущего участка к следующему. Иногда отдельно задается различный вертикальный и горизонтальный шаг.

Отступы — обозначают количество добавляемых фиктивных слоев пикселей с каждой из сторон тензора. Необходим для того, чтобы краевые элементы обрабатываемого тензора покрывались фильтром столько же раз, сколько и центральные. Отступы могут задаваться различными для каждой стороны тензора.

Растяжение — увеличивает размер фильтра, оставляя количество коэффициентов фильтра неизменным. Следовательно, коэффициенты «растягиваются», а в элементы между ними записываются нулевые значения. Растяжение также может задаваться различное для каждого направления.

Полносвязный слой:

Полносвязный слой - это слой, в котором каждый входной элемент (нейрон) связан с каждым элементом на выходе, отсюда и название. Периодически можно встретить название «“плотный” слой» (*dense layer*). Он играет ключевую роль в решении задачи классификации, а внутри себя использует веса и сдвиги для корректной работы. Во время обучения нейронной сети важно подобрать такие веса и сдвиги, чтобы добиться максимальной точности классификации.

Алгоритм, используемый внутри полносвязного слоя, как ни странно, имеет математическую основу. А именно - умножение матрицы весов на вектор входных данных и прибавление к результату вектора сдвига:

$$res = \begin{pmatrix} w_{11} & \cdots & w_{1n} \\ \vdots & \ddots & \vdots \\ w_{m1} & \cdots & w_{mn} \end{pmatrix} * \begin{pmatrix} a_1 \\ \vdots \\ a_n \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix}$$

$$a - \text{входной вектор}$$

$$w - \text{матрица весов}$$

$$b - \text{вектор сдвига}$$

Поэлементный слой:

Поэлементный слой - это слой, используемый для применения к каждому элементу тензора определённой унарной операции. В частности, это может быть *ReLU* - операция, изменяющая все отрицательные значения на 0. Такие функции обычно называют функциями активации. Данные поэлементные преобразования имеют большой смысл во время запуска прямого прохода *CNN*, так как приводят данные к более информативному виду, а также определяют их границы значений.

Пример работы слоя на входном векторе представлен ниже:

$$res = \begin{pmatrix} f(a_1) \\ \vdots \\ f(a_n) \end{pmatrix}$$

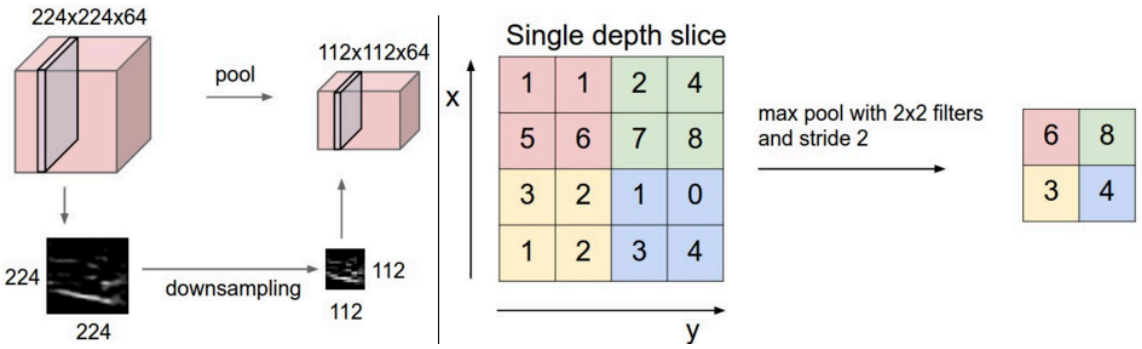
$$a - \text{входной вектор}$$

$$f - \text{функция активации}$$

Слой субдискретизации:

Данный слой имеет схожую природу со слоем свёртки. Его цель - сжать входные данные (вектор или матрицу), уменьшив размер. Под "сжатием" здесь понимается либо нахождение максимального среди пачки элементов, либо нахождение среднего. То, какая будет пачка элементов, определяется размером ядра (например 2x2, 3x3).

С точки зрения всей сети, субдискретизация помогает сократить количество вычислений. Часто она используется сразу после слоя свёртки. На рисунке ниже представлена работа данного слоя:



(Рис. 1 Работа слоя субдискретизации) [2]

Как и в слое свёртки, здесь присутствует понятие шага, определяющее сдвиг ядра во время выполнения алгоритма.

Выходной слой:

Так как *AlexNet* решает задачу классификации, необходимо в правильном виде представить результаты работы нейросети. Для этого в модели присутствует данный слой, содержащий наименования для каждого элемента результирующих данных. В случае, когда на вход приходит изображение, результат будет представлен в понятном для пользователя виде. К примеру, если сеть будет отличать кошек от собак, результирующие данные можно представить в следующем виде:

0.87 - кошка

0.13 - собака

Результат интерпретируется так, что легко понять, что наша сеть больше уверена в том, что на изображении - кошка. В общем случае нас больше интересует алгоритм *TopK* (вывод *K* наиболее вероятных исходов).

Его принцип заключается в сортировке выходных данных по вероятностям и вывода результатов пользователю:

0.08	Акула
0.25	Краб
0.48	Медуза
0.11	Морская звезда
0.09	Морской конёк

0.48	Медуза
0.25	Краб
0.11	Морская звезда
0.09	Морской конёк
0.08	Акула

(Рис. 2 Алгоритм *TopK* для пяти элементов)

Выпадающий слой:

Альтернативное названия - исключающий, выбрасывающий. Данный слой используется в обучении сети, так как позволяет более качественно обучать конкретные нейроны. Его принцип заключается в отключении N случайных нейронов на разных эпохах обучения нейросети. Этот метод предотвращает переобучение.

Слой нормализации:

Слой нормализации - это один из слоёв, подготавливающих данные к дальнейшей обработке. Его цель - ограничить значения до нужного диапазона. Делается это для того, чтобы все входящие данные были примерно однотипны и последующие слои могли их использовать без излишнего роста значений.

Пример нормализации - приведение всех элементов к диапазону [0;1] или [0;255]. Второй диапазон имеет для нас больше смысла, так как в случае изображений на вход подразумевается, что они имеют конкретные RGB значения.

Проблемы отсутствия нормализации заметны как и во время операций полносвязного слоя, так и при обучении. Они могут привести к неконтролируемому и нестабильному росту значений и необходимости переучивать сеть.

Программная реализация

Графовая модель

В библиотеке проекта класс *Graph* представляет собой реализацию направленного ациклического графа (*Directed Acyclic Graph, DAG*) для нейронной сети. Такой граф не содержит циклов и предназначен для представления потока данных и операций в нейронной сети.

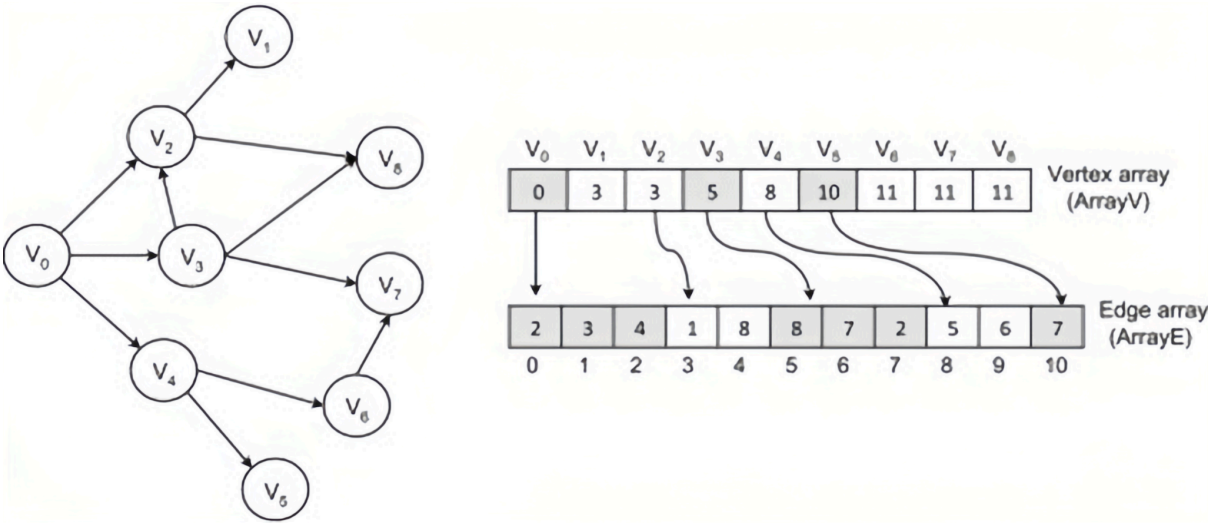
В данной реализации граф организован следующим образом:

- Узлы графа представлены объектами класса *Layer*, которые содержат операции над данными.
- Рёбра графа задаются методом *makeConnection*, который соединяет слои сети.
- Для выполнения прямого прохода по графу и выполнения инференса используется метод *inference*.

Отдельное внимание стоит уделить способу хранения информации о связанности слоев. Для экономии места и удобства организуются два массива:

1. содержит подряд идущие номера слоев, к которым мы можем идти из данного слоя
2. хранит индексы первого массива чтобы не путать какие слои к каким относятся.

Для большей наглядности представлено изображение .



(Рис. 3 Сжатые списки смежности)

Также был реализован алгоритм, напоминающий поиск в ширину для генерации маршрута, чтобы пройтись по слоям в правильной последовательности.

Форма

Форма (*shape*) позволяет явно определить, какой размерности тензор и какие размеры он имеет, а также позволяет итерироваться по нему.

В нашем проекте значения в тензоре представляются в виде одномерного вектора, так что возникла необходимость в выводе алгоритма вычисления индекса тензора:

Для тензора 2 ранга:

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

$$\downarrow$$

$$(a_{11} \quad a_{12} \quad a_{13} \quad a_{14} \quad a_{21} \quad a_{22} \quad a_{23} \quad a_{24} \quad a_{31} \quad a_{32} \quad a_{33} \quad a_{34})$$

$$index(a_{ij}) = (i-1) * width + (j-1)$$

Далее по индукции ...

$$index(a_{i_1 i_2 \dots i_n}) = (i_1 - 1) * \prod_{j=2}^{n} d_j + (i_2 - 1) * \prod_{j=3}^{n} d_j + \dots + (i_{n-1} - 1) * d_n + (i_n - 1) =$$

$$= \sum_{k=1}^{n} \left((i_k - 1) * \prod_{j=k+1}^{n} d_j \right) + (i_n - 1)$$

где $d = (d_1 \quad d_2 \quad \cdots \quad d_n)$ – форма тензора (максимально возможные значения соотв. индексов)

Тензор

Универсальной структурой, которая будет использована для хранения данных внутри слоя и для передачи данных между слоями является тензор. Тензор - это N-мерная матрица, которая позволяет хранить элементы разных типов данных.

В нашей библиотеке тензор представляется одномерным динамическим массивом. Для обеспечения возможности хранения в тензоре данных различных типов и в целях оптимизации хранения данных вместо шаблонов было решено использовать динамический массив байт. Таким образом мы будем иметь возможность преобразовывать массив байт в массив элементов необходимого размера, в зависимости от типа данных конкретного тензора. Для этого тензор дополнительно должен хранить свой тип данных. В нашей библиотеке в данный момент поддерживаются два типа данных для тензора: 32-битный целочисленный тип и 32-битный вещественный тип. Кроме преобразований типа массива мы должны иметь возможность получить доступ к конкретному элементу тензора по N координатам. Для этого в тензоре будет сохранена его размерность, которая позволит преобразовать N входных координат элемента в индекс массива, где этот элемент хранится.

Реализация слоёв AlexNet

Входной слой:

В слое, который реализован нами, предусмотрена как нормализация, так и передаваемая размерность. Это сделано для того, чтобы нейронная сеть принимала на вход как формат *NCHW(Number, channels, height, width)*, так и *NHWC(Number, height, width, channels)*. Изначально проект направлен на инференс любых сверточных нейронных сетей, а в разных сетях разные параметры для входа.

Сначала изменяются значения пикселей в зависимости от параметров, которые мы подаем на вход, а затем, получая информацию о нужном формате, заносим значения в новый тензор в необходимом порядке.

Сверточный слой:

В сверточном слое, написанном нами используются все параметры, перечисленные выше в методах решения. Изначально происходит увеличение размеров матрицы, за счет добавления нулей во все три потока, это необходимо для того, чтобы

при необходимости можно было использовать ядро большего размера. После этого немного иным способом происходит расширение ядра, по заданным параметрам, также с добавлением нулей. Затем начинается перемножение матриц по всем правилам математики с заполнением нового вектора, а также подсчет параметров для тензора, так как при умножении на ядро размер тензора должен быть скорректирован. Изначально данный слой принимал на вход только формат *NHWC*, а на данный момент, чтобы перейти ближе к структуре AlexNet, переведён на *NCHW*. Также для него есть подкласс, который выполняет сами операции с вектором, а все параметры тензора извлекаются на входе и используются по отдельности.

Полносвязный слой:

Перед тем, как реализовать полносвязный слой, необходимо было разобраться с его основной операцией: умножением матрицы на вектор и прибавление вектора сдвига. Для этого была создана функция *mat_vec_mul*, использующая стандартный алгоритм умножения. Сам класс *FCLayerImpl* принимает одномерный вектор, веса, сдвиги и возвращает опять же одномерный вектор для передачи далее. *FCLayer* же работает схожим образом, но работает на тензорах. Проверяются все исключительные случаи и покрываются различными тестами.

Поэлементный слой:

EWLayerImpl работает по простому принципу: он принимает название функции типа *std::string*, параметры альфа и бета (необязательные). На данный момент доступны функции *tanh, sin, minus, linear, relu*. Некоторые из них нужны лишь для тестов. Параметры альфа и бета нужны исключительно для линейной (*linear*) функции. Её можно описать так: *linear(x) = alpha*x + b*.

Данный функционал также описан тестами.

Слой субдискретизации:

Написание кода для слоя субдискретизации требует хорошего понимания процесса. В данный момент подразумевается, что шаг выбран так, чтобы не было пересечений во время движения ядра по тензору. Подход к такому в *PoolingLayerImpl* - это в конструкторе принимать форму ядра и вид субдискретизации (с функцией среднего или максимального). Это позволяет до вычислений понять, какой формы

будет выходной тензор, ведь помимо этого в *PoolingLayerImpl* принимается форма входного тензора.

Зная конечную форму тензора, можно итерироваться по этой форме, лишь тщательно выбирая индексы для перемещения. В процессе разработки появилась необходимость в функции *OutOfBounds*. Она определяет, не вышли ли мы за пределы размеров нашего тензора во время выполнения алгоритма, подразумевая, что следующие и предыдущие по очереди измерения тензора имеют размер 1 (нетрудно догадаться, что любой тензор можно представить тензором более высокого ранга, как и математический вектор можно представить матрицей n x 1).

Выходной слой:

В соответствии с концепцией выходного слоя и с базовым классом *Layer* был создан выходной слой и покрыт тестами. Для реализации сортировки была использована стандартная функция *std::sort*.

Структура проекта

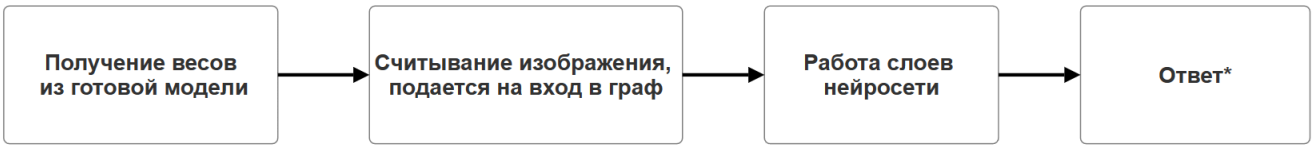
Проект организован так, чтобы обеспечивалось удобство в управлении и поддержке кодовой базы. В ходе работы команда столкнулась с проблемой, что все слои были реализованы с помощью шаблонов (*template в C++*). Это не позволяло работать с разными типами данных в пределах одной сети и уменьшало читаемость кода.

По этой причине каждый слой делится на два: *...LayerImpl* (изначальная реализация) и *...Layer* (удобный интерфейс). LayerImpl содержит реализацию слоя через шаблоны, а другой - это работа со слоем через тензор (который сам по себе не шаблонный). Отсутствие такого разделения с одновременным удалением шаблонов бы породило разрастание кода, поэтому все алгоритмы были сосредоточены в классе реализации. Интерфейс лишь в соответствии с типом данных создает экземпляры класса реализации.

Разделение также помогает при оптимизации кода, так как позволяет создать *LayerImplOptimizedTBB*, *LayerImplOptimizedOmp* и так далее без необходимости в разрастании классов.

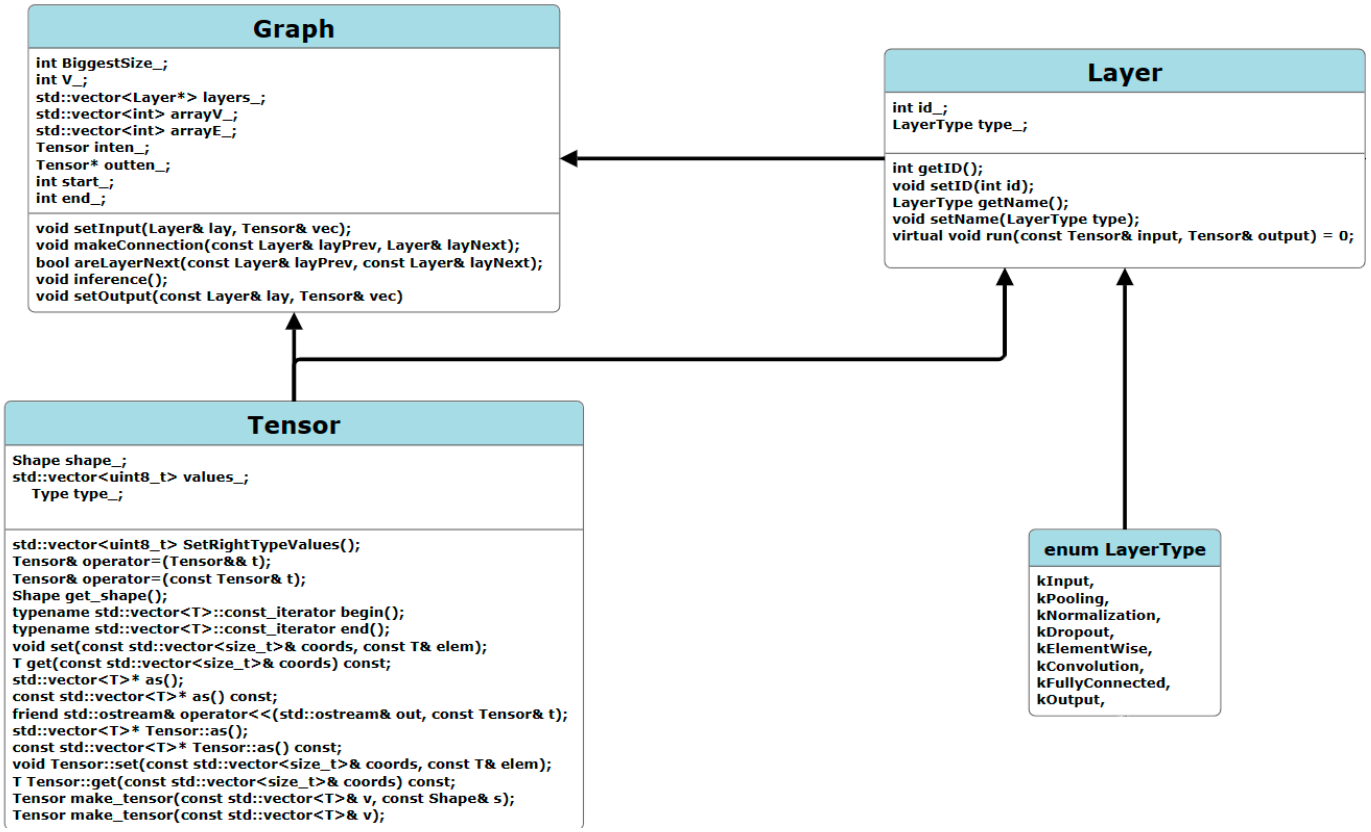
Ниже представлена детальная информация о структуре проекта и его зависимостях.

Архитектура



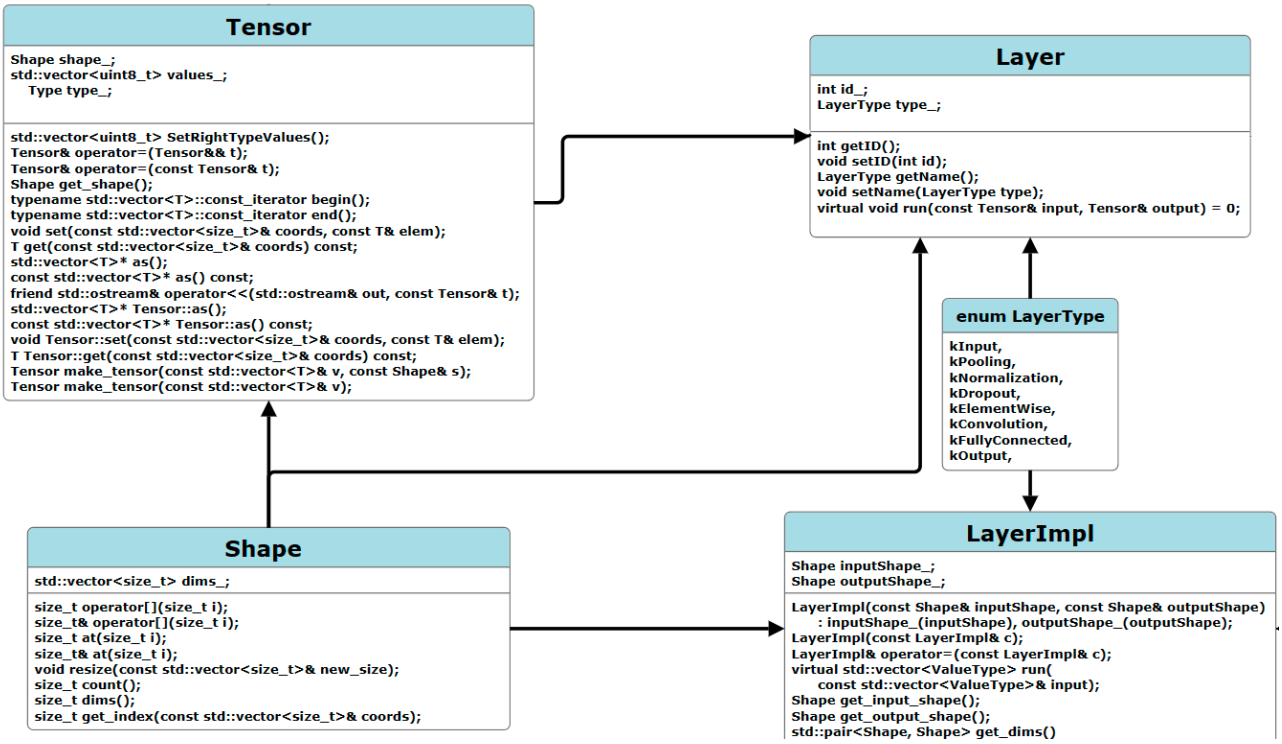
(рис. 4 Схема работы библиотеки)

Схема классов библиотеки: основа сети - класс *Graph* содержит в себе вектор классов *Layer* и 2 класса типа *Tensor* (вход и выход). *Layer* в свою очередь может быть одним из 8 видов слоев: *kInput, kPooling, kNormalization, kDropout, kElementWise, kConvolution, kFullyConnected, kOutput*.



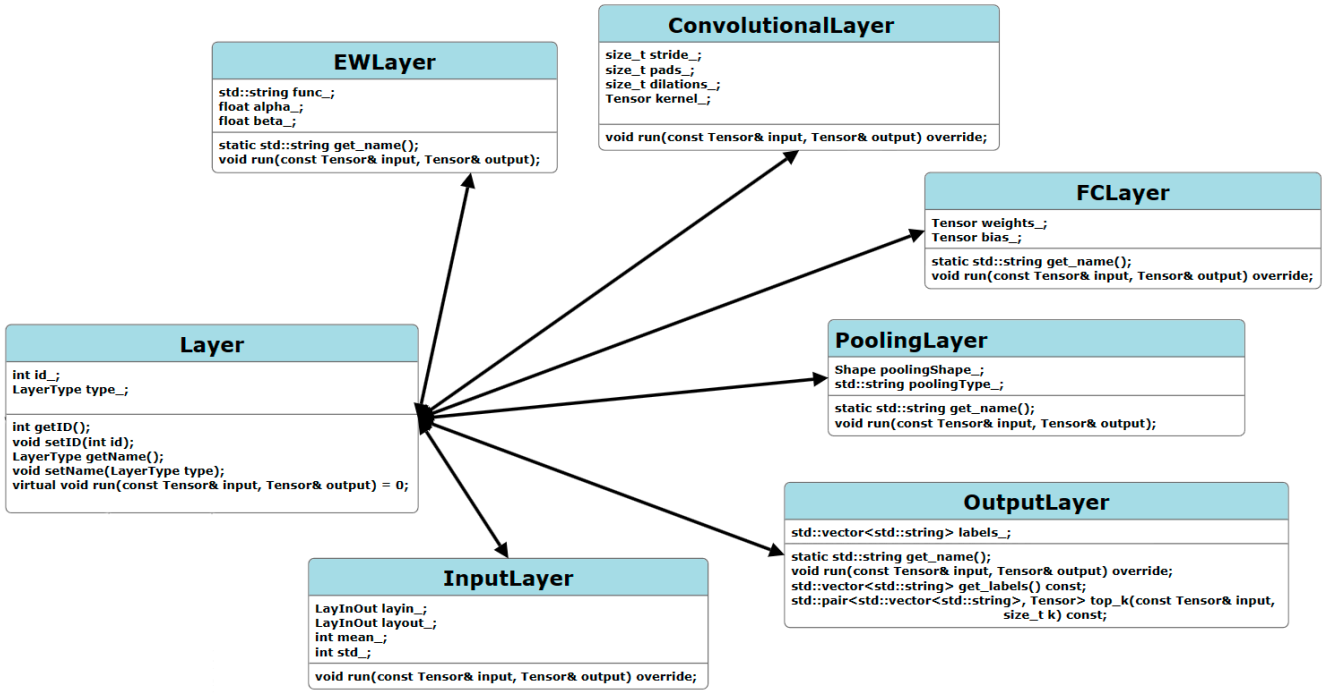
(рис. 5 Схема классов ч.1)

Класс *Tensor* содержит в себе экземпляр *Shape*. Layer работает со структурой *Tensor* и ,следовательно, с классом *Shape*. *LayerImpl* работает напрямую с классом *Shape* и содержит реализацию слоя через шаблоны, а *Layer* работает со структурой *Tensor.*



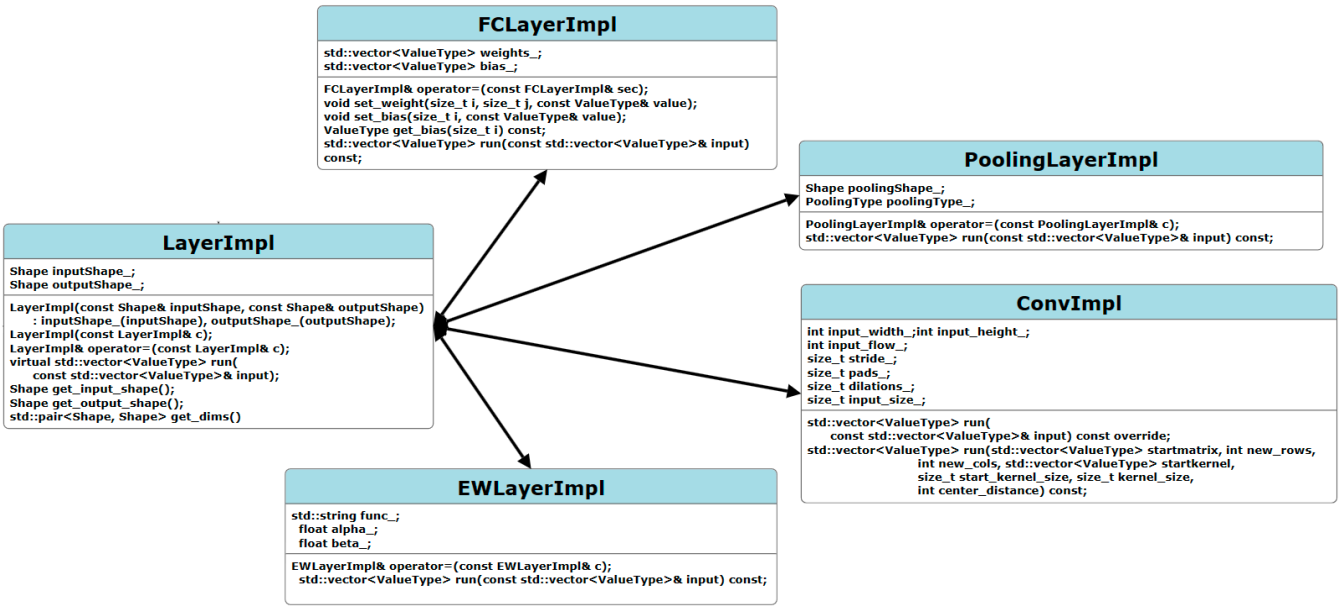
(рис. 6 Схема классов ч.2)

На схеме ниже показано наследование классов различных слоев от класса *Layer*.



(рис. 7 Схема классов ч.3)

И также схема наследования классов типа ...*LayerImpl* от класса *LayerImpl*.



(рис. 8 Схема классов ч.4)

Полная схема библиотеки размещена в репозитории проекта.

Структура директорий

Структура организована следующим образом: в папке *app* хранятся примеры использования проекта, запуск нейросети на интересующих изображениях, а также проверка accuracy библиотеки. В *include* содержатся заголовочные файлы, в таких папках как: *graph* - структура нейросети, *layers* - описание всех слоев содержащихся в графе, *perf* - заголовочные файлы для измерения производительности всех функций. В папке *src* лежит реализация файлов библиотеки. В папке *3rdparty* хранятся сторонние библиотеки, используемые в нашем проекте. Тесты находятся в папке *test* внутри проекта и обеспечивают проверку правильности работы основных компонентов приложения.

Зависимости проекта

Проект использует стороннюю библиотеку *OpenCV* в качестве субмодуля для работы с изображениями, необходимыми для входного слоя. Также используется библиотека *TBB* для эффективного распараллеливания вычислений, особенно в операциях с большими объемами данных. Во время написания кода каждому необходимо было внедрять тесты с использованием *Google C++ Testing Framework*

(gtest). Это помогало не только проверять код на ошибки, но и в общем представить картину каждого из модулей и всей библиотеки.

Для настройки работы нейросети в проект была интегрирована *TensorFlow*, чтобы написать парсер модели *protocol buffer alexnet_frozen.pb* — предварительно обученной модели. Это позволило бы использовать уже обученную сеть и ее веса для работы нейросети. Однако, мы столкнулись с существенными трудностями при использовании *TensorFlow* на *C++*. Сложность интеграции, несовместимость версий и другие технические проблемы сделали этот подход неэффективным для нашего проекта [4].

В результате, было принято решение сменить фреймворк и написать парсер на *Python*. Была выбрана модель — *Alexnet-model.h5*. Получение весов и графа этой модели прошло успешно. Это позволило нам продолжить разработку и интеграцию без значительных задержек и проблем.

Аппаратно-программное обеспечение

Для выполнения большинства задач проекта было необходимо четко определить обязанности каждого участника, поэтому во время работы был использован сервис *GitHub*, основанный, в свою очередь, на *Git*. Это распространённая практика для разработки библиотек, приложений, сайтов и так далее.

Определение основных функциональных требований к библиотеке происходило с опорой на другие подобные библиотеки (*OpenVINO, TensorFlow,* их документации в Интернете). Еженедельно проводились собрания, где обсуждалась сама архитектура библиотеки и то, какой мы её видим.

В проекте используется система сборки *CMake* для автоматизации процесса компиляции и сборки проекта. Это позволяет обеспечить переносимость проекта на различные операционные системы такие как *Windows, Linux* и *MacOS* и архитектуры x64 и x86. Сборка проекта разделена на несколько этапов. Сначала происходит сборка сторонних библиотек, таких как *OpenCV*, которые необходимы для работы нашей библиотеки. Затем собирается сам проект, включая создание исполняемых файлов и динамических библиотек [3].

Вся логика сборки и зависимости описаны в файлах *CMakeLists.txt.*

Документация и инструкции

На странице проекта на *GitHub* предоставлена документация, включая инструкции по установке, сборке, запуску и локальному тестированию библиотеки, а также некоторые бинарные файлы участвовавшие в разработке.

Вся документация написана на понятном языке с примерами использования и подробными пояснениями, чтобы обеспечить удобство в работе с проектом.

Инфраструктура проекта

В процессе разработки проекта были использованы следующие инструменты и практики:

Непрерывная интеграция (CI)

Для обеспечения чистоты и читабельности кода применялись следующие инструменты:

- *clang-format checks* на *GitHub Actions* для проверки форматирования кода в соответствии с *Google Style Guide*.
- *clang-tidy* бот для анализа кода и выявления потенциальных ошибок и проблем.
- *codecov* для проверки покрытия кода тестами и отслеживания изменений в покрытии. В нашем случае - 87%.

В целом, созданная инфраструктура обеспечивает эффективную и надежную работу нашего проекта на платформе *GitHub*, что позволяет нам фокусироваться на разработке и улучшении проекта, а не на решении технических проблем.

Организация работы команды

- Еженедельные созвоны для обсуждения прогресса, постановки задач и решения возникающих проблем. Порядок прохождения собрания был заранее определен и сохранялся на протяжении всей работы: сначала каждый отчитывался о своем статусе выполнения той или иной задачи, задавал вопросы и получал пояснения или корректировки, далее выдавались новые задачи и дополнительный материал к ним.
- Рабочий чат для оперативного общения и задавания вопросов.
- Доска заданий на *GitHub* для отслеживания и распределения задач.
- Ветвление в *Git*: для каждой задачи создавалась отдельная ветка, что обеспечивало изолированную разработку функциональности. После завершения

работы над задачей создавался *Pull Requests* для объединения изменений с основной веткой (*main*).

- Ревью кода: каждый пул реквест проходил процесс ревью, в ходе которого код проверялся другими членами команды.
- Объединение с главной веткой: изменения объединялись с основной веткой только после успешного прохождения всех проверок и получения трех одобрений (*approvals*) от других участников команды.

Использование этих инструментов и практик способствовало повышению качества кода, облегчению совместной работы и отслеживанию прогресса разработки.

Результаты работы

Наша команда успешно завершила разработку **C++ библиотеки AlexMNIST** для эффективного инференса на MNIST. Ключевые итоги:

- Все запланированные компоненты реализованы
- Достигнута точность **98.02%**
- Производительность:
    - 30 секунд на инференс 10 тысяч изображений (CPU, Intel i7 11700)
    - Ускорение на 97% благодаря множественному входному слою и распараллеливанию с помощью TBB слоя свёртки.

Вклад команды:

Составляющие библиотеки	Статус	Автор
Add openCV	Выполнено	Титов С. М.
Add tensorflow	Выполнено	Титов С. М.
Implement TBB	Выполнено	Титов С. М.
Documentation	Выполнено	Титов С. М.
Parsing weights from a model	Выполнено	Титов С. М.
Weights from JSON to tensors	Выполнено	Титов С. М.
Graph structure	Выполнено	Сорокин А. А.
Input layer	Выполнено	Сорокин А. А.
Convolution Layer	Выполнено	Сорокин А. А.
Implement per-layer statistics	Выполнено	Сорокин А. А.
Fully-connected layer	Выполнено	Леонтьев Н. С.
Element-wise layer	Выполнено	Леонтьев Н. С.
Pooling layer	Выполнено	Леонтьев Н. С.
Output layer	Выполнено	Леонтьев Н. С.

Function wall execution time	Выполнено	Леонтьев Н. С.
Tensor	Выполнено	Суворов Д. И. Леонтьев Н. С.
Refactor layer and graph classes	Выполнено	Сорокин А. А. Леонтьев Н. С.
Normalization layer	Выполнено	Леонтьев Н. С.
Flatten Layer	Выполнено	Леонтьев Н. С.
Dropout layer	Выполнено	Сорокин А. А.
Accuracy verification	Выполнено	Сорокин А. А.
The reading function	Выполнено	Сорокин А. А. Титов С. М.
Graph-inference	Выполнено	Сорокин А. А. Титов С. М.
Library optimization	Выполнено	Сорокин А. А.

Подробности можно узнать в разделе «*Pull Requests*» репозитория *GitHub*.

Заключение

В рамках нашего проекта по созданию библиотеки для инференса нейронной сети *AlexNet*, команда достигла поставленной цели благодаря слаженной работе и эффективной организации разработки.

Проект выполнен на высоком уровне, что стало возможным благодаря четкому распределению задач, регулярному взаимодействию внутри команды и использованию современных инструментов и технологий.

Ключевые аспекты успешной реализации проекта:

1. Техническая реализация:
    - Полностью завершена разработка всех компонентов:
        - Сверточные слои
        - Пулинговые слои (*MaxPooling, AveragePooling*)
        - Полносвязные слои с оптимизированными матричными операциями
        - Нормализационные слои
        - Слой Dropout
        - Input, Output слои
2. Оптимизация производительности:
    - Реализована эффективная параллелизация вычислений с использованием *Intel TBB*
    - Оптимизированы операции матричного умножения
    - Достигнуто время инференса 3 мс на изображение на *CPU*
3. Точность работы:
    - Подтверждена точность 98.02% на тестовом наборе *MNIST*
    - Проведено сравнение с эталонными реализациями
4. Качество кода:
    - Соблюдены принципы *Google Style Code*
    - Реализована модульная архитектура
    - Обеспечена кроссплатформенная совместимость
5. Документация:
    - Полностью описаны *API* библиотеки

- Созданы примеры использования

Проект выполнен на профессиональном уровне благодаря:

- Четкому распределению задач между участниками
- Регулярному код-ревью
- Использованию современных практик разработки (*CI/CD*, юнит-тестирование)
- Применению актуальных инструментов (*CMake, Git*)

Библиотека готова к использованию в промышленных и исследовательских задачах, демонстрируя высокие показатели как по точности, так и по производительности.

Список литературы

1. Netron, программа для визуализации моделей нейронных сетей. - [Электронный ресурс]. URL: https://netron.app.
2. Pooling Layer. - [Электронный ресурс]. URL: https://leonardoaraujosantos.gitbook.io/artificial-inteligence/machine_learning/deep_learning/pooling_layer.
3. Документация языка сборки CMake. - [Электронный ресурс]. URL: https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html.
4. Нейросетевые модели и наборы данных. - [Электронный ресурс]. URL: https://www.tensorflow.org/resources/models-datasets?hl=ru.
5. Yu W. et al. Visualizing and comparing AlexNet and VGG using deconvolutional layers. - [Электронный ресурс] //Proceedings of the 33 rd International Conference on Machine Learning. – 2016. URL: https://icmlviz.github.io/icmlviz2016/assets/papers/4.pdf
6. Kaehler A., Bradski G. Learning OpenCV 3: computer vision in C++ with the OpenCV library. - [Электронный ресурс] – " O'Reilly Media, Inc.", 2016. URL: https://books.google.ru/books?id=LPm3DQAAQBAJ&printsec=frontcover&hl=ru

Приложения

Репозиторий GitHub: https://github.com/embedded-dev-research/itlab_2023.

Руководство пользователя

Как мне запустить инференс?

1. Убедитесь, что вы установили зависимости проекта, выполнив команду:

   pip install -r requirements.txt

2. Вам нужно запустить скрипт *parser.py* который находится в *app/AlexNet*, чтобы считывать веса из модели *Alexnet-model.h5*, а json-файл с весами будет сохранен в папке *docs*, в ней же и находится модель.
3. Затем поместите тестовые изображения в формате *png* в папку *docs/input.*
4. После сборки проекта, которая описана ниже запустите файл *Graph_build* из *build/bin.*

Сборка библиотеки

Windows

Для сборки и локального запуска этого проекта на *Windows* следуйте этим шагам:

1. Клонируйте этот репозиторий на свою локальную машину, используя следующую команду:

   git clone https://github.com/embedded-dev-research/itlab_2023.git

2. Перейдите в директорию проекта и обновите субмодули:

   cd itlab_2023
   git submodule update --init --recursive

3. Настройте проект: Создайте папку *build* для настройки проекта и его компиляции:

   mkdir build
   cd build
   cmake .. -DCMAKE_BUILD_TYPE=Release

   Примечание: Убедитесь, что у вас установлен *CMake* для сборки проекта.
4. Соберите проект: Далее, чтобы собрать проект, нам нужно ввести команду:

cmake --build . --config Release

Примечание: Если вы хотите собрать в режиме отладки, замените *release* на *debug.*

5. Запустите проект: После сборки проекта вы можете найти исполняемые файлы по следующему пути:

   cd build/bin

Linux/macOS

Для сборки и локального запуска этого проекта на *Linux* или *macOS* следуйте этим шагам:

1. Клонируйте этот репозиторий на свою локальную машину, используя следующую команду:

   git clone https://github.com/embedded-dev-research/itlab_2023.git

2. Перейдите в директорию проекта и обновите субмодули:

   cd itlab_2023
   git submodule update --init --recursive

3. Установите необходимые зависимости:

   OpenMP Debian/Ubuntu:
   sudo apt-get install -y libomp-dev

   macOS:
   brew install libomp

4. Настройте проект: Создайте отдельную директорию для настройки проекта и его компиляции:

   cmake -S . -B build

Примечание: Убедитесь, что у вас установлен *CMake* для сборки проекта. Для *macOS* необходимо указать путь к файлу *omp.h*:

cmake -S . -B build -DCMAKE_CXX_FLAGS="-I$(brew --prefix libomp)/include" -DCMAKE_C_FLAGS="-I$(brew --prefix libomp)/include"

5. Соберите проект: Далее, чтобы собрать проект, нам нужно ввести команду:

cmake --build build --config Release

Если вы хотите собрать в режиме отладки, замените release на debug

6. Запустите проект: После сборки проекта вы можете найти исполняемые файлы в папке *build.*

Процесс тестирования

Этот проект содержит тесты для проверки функциональности. Для тестирования проекта используется *Google Test Framework* как субмодуль проекта.

Windows

Чтобы начать процесс тестирования локально, вам нужно перейти в директорию

cd build/bin

и запустить следующие файлы:

run_test.exe

Linux

Чтобы начать процесс тестирования локально, вам нужно перейти в директорию

cd build/bin

и запустить следующие файлы:

chmod +x run_test
./run_test

Проверка Accuracy

Для проверки точности вам необходимо использовать набор данных *MNIST*, который вы можете скачать здесь и поместить в папку *docs/mnist/mnist/test* Теперь вы можете запустить проверку точности:

build\bin\ACC_MNIST.exe

Точность должна составлять 98,02%