Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Вологин Илья Олегович

Поддержка нейронных сетей на основе архитектуры Transformer в библиотеке KInference

Отчёт по учебной практике

Научный руководитель:
к. т. н., доцент Брыксин Т. А.

Консультант:
программист-исследователь JetBrains Research Тучина А. И.

Санкт-Петербург
2021

Оглавление

Введение

Нейронные сети стали неотъемлемой частью нашей жизни. Они используются для решения многих повседневных задач, например, для обработки изображений [5], распознавания речи [8], генерации текстов [4] и многих других.

В наше время существует большое количество различных архитектур построения нейронных сетей, среди которых можно выделить класс Transformer [2]. Основными представителями данного класса являются архитектуры Generative Pre-trained Transformer (GPT) [6] и Bidirectional Encoder Representations from Transformers (BERT) [3]. Модели, построенные на архитектуре Transformer, главным образом показывают высокую эффективность в задачах обработки естественных языков.

Данные модели, как правило, заранее обучают для решения определенной задачи, используя в проектах уже обученную модель.

Для запуска модели можно использовать библиотеки, ранее примененные для обучения, но тогда в проекте возникает проблема избыточной функциональности, так как большая ее часть в подобных библиотеках не требуется на этапе запуска. Также форматы моделей для этих библиотек предполагают наличие в файле модели информации, нужной только в процессе обучения. Это негативно влияет на объем потребляемых ресурсов на фазе применения модели.

Также можно использовать библиотеки, предназначенные только для запуска моделей, но процесс поддержки многих форматов моделей в данных библиотеках является довольно трудоемким. Кроме того, необходимо, чтобы они были эффективны по скорости работы, поскольку многие задачи (например, подсказки при наборе текста) требуют быстрого отклика. К тому же для оптимизации потребляемой памяти и скорости запуска модели можно квантизовать.

Для решения проблемы поддержки многих форматов моделей был разработан новый универсальный формат Open Neural Network Exchange (ONNX) [9], в который можно конвертировать модели из различных форматов. Примеров библиотек для запуска, которые поддерживают

формат ONNX, в настоящее время достаточно мало, например: ONNX Runtime, NCNN и MNN. Все данные библиотеки написаны на C++, но некоторые из них предоставляют API и для других языков.

Так как существующие решения разработаны на C++, они имеют проблемы языковой совместимости с проектами на Java и Kotlin, являющимися достаточно популярными языками разработки. Таким образом, проблема запуска моделей формата ONNX в проектах на языках Java и Kotlin остается актуальной.

В настоящее время на языке Kotlin разрабатывается библиотека KInference. Её основная цель — запуск нейронных сетей в формате ONNX в проектах на языках Kotlin и Java.

В связи со всем вышеперечисленным поддержка моделей нейронных сетей на основе Transformer архитектур в библиотеке KInference может быть полезна для проектов на языках Kotlin и Java, использующих нейронные сети.

Постановка задачи

Целью данной работы является поддержка нейронных сетей на основе архитектуры Transformer в библиотеке KInference. Для достижения поставленной цели требуется:

- реализовать основные компоненты Transformer архитектуры: кодировщик, декодировщик и механизм внимания;
- обеспечить эффективность разработанного алгоритма запуска нейронных сетей;
- реализовать несколько популярных Transformer моделей: GPT-2 и BERT;
- реализовать поддержку квантизованных моделей;
- провести апробацию полученного решения.

1. Обзор

1.1. ONNX

ONNX — универсальный формат представления модели нейронной сети, призванный упростить переносимость моделей, обученных с помощью различных библиотек, а также облегчить запуск нейронных сетей из различных библиотек для обучения. Для ONNX разработано множество утилит, которые позволяют преобразовать модель из формата библиотеки для обучения в модель формата ONNX.

Модель в формате ONNX представляется в виде графа вычислений, где вершины — это операторы, исполняющие вычисления, а рёбра определяют последовательность передачи данных между операторами.

Так как формат ONNX разрабатывался для запуска нейронных сетей, то при преобразовании в него из модели удаляются данные, которые были необходимы на этапе обучения, но не нужны для запуска нейронной сети. Это позволяет уменьшить объем потребляемых моделью ресурсов.

В связи со всем вышеперечисленным в настоящее время активно разрабатываются инструменты для запуска нейронных сетей, которые поддерживают формат ONNX.

Рассмотрим несколько таких инструментов.

- Библиотека ONNX Runtime[1]. Данная библиотека разрабатывается компанией Microsoft на языке C++, а также предоставляет API для языков Python, C, C# и Java. Разработчики библиотеки предоставляют на выбор большое количество платформ, на которых может использоваться данная библиотека, а также выбор наиболее предпочтительного аппаратного ускорения.

- Библиотека NCNN[2]. Эта библиотека разрабатывается компаний Tencent на языке C++. Данная библиотека ориентирована для работы на мобильных устройствах: оптимизирована для работы с

[1] `https://github.com/microsoft/onnxruntime`

[2] `https://github.com/Tencent/ncnn`

ARM процессорами, поддерживает мобильные графические устройства, а также не имеет дополнительных зависимостей, благодаря чему является легковесной.

- Библиотека MNN [7]. Данная библиотека разрабатывается компанией Alibaba на языке C++. Данная библиотека также ориентирована для работы на мобильных устройствах и имеет все те же преимущества, что и библиотека NCNN.

В проектах на Java и Kotlin из данных примеров можно использовать только библиотеку ONNX Runtime. Но для работы с данными проектами ONNX Runtime только запускает скомпилированную на C++ библиотеку при помощи Java Native Interface (JNI)[3]. Это, в свою очередь, влияет на стабильность работы, а также может ухудшить производительность, так как при работе с данной библиотекой происходят дополнительные копирования памяти. К тому же данная библиотека не является легковесной, что является важным фактором для многих проектов.

1.2. KInference

KInference[4] — библиотека для запуска нейронных сетей, разрабатываемая в лаборатории методов машинного обучения в программной инженерии JetBrains Research. Эта библиотека написана на языке Kotlin и имеет несколько преимуществ:

- поддержка универсального формата ONNX;
- легковесность;
- возможность в будущем стать мультиплатформенной.

Так как данная библиотека полностью написана на языке Kotlin, то у неё нет проблем с языковой совместимостью с проектами на Kotlin и

[3] https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

[4] https://github.com/JetBrains-Research/kinference

Java, которые сейчас являются довольно популярными языками разработки.

1.3. Архитектура Transformer

Архитектура Transformer впервые была представлена в недавнем исследовании [2]. Эта архитектура состоит главным образом из кодирующего и декодирующего модуля. На Рис. 1 изображен общий вид архитектуры Transformer.

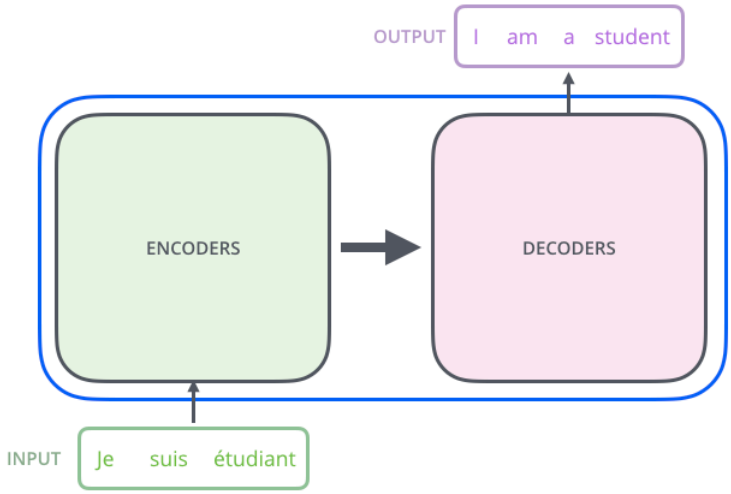

Рис. 1: Общий вид архитектуры Transformer (рисунок взят из [1]).

Кодирующий и декодирующий модули представляют из себя последовательности повторяющихся кодировщиков и декодировщиков, причем количество кодировщиков и декодировщиков всегда равно. В последствии выходы кодирующего модуля передаются всем декодировщикам. На Рис. 2 изображена структура кодирующих и декодирующих модулей.

Кодировщик состоит из двух частей: механизма самовнимания и линейного слоя. Декодировщик состоит из трех частей: механизма самовнимания, механизма внимания, который на вход дополнительно получает выходы кодирующего модуля и линейного слоя. Также входы и выходы каждого слоя складываются между собой и нормализуются. На Рис. 3 изображена структура кодировщика и декодировщика.

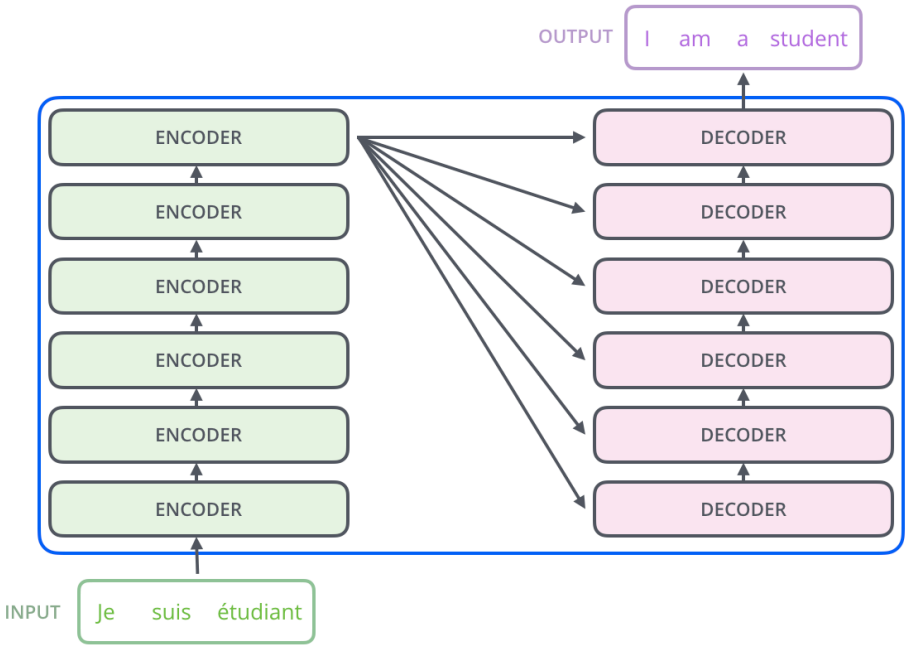

Рис. 2: Общий вид кодирующего и декодирующего модуля (рисунок взят из [1]).

Рассмотрим некоторые компоненты подробнее.

1.3.1. Механизмы внимания

В первую очередь для каждого входного вектора создаются три вектора: вектор запроса, вектор ключа и вектор значения. Они вычисляются по формулам (1), (2) и (3) соответственно.

$$q_i = x_i \times W_Q \tag{1}$$

$$k_i = x_i \times W_K \tag{2}$$

$$v_i = x_i \times W_V, \tag{3}$$

где W_Q, W_K, W_V — матрицы весов;
q_i, k_i, v_i — векторы запроса, ключа и значения;
x_i — входной вектор.

Вторым шагом является подсчет внутренних коэффициентов внимания. Данные коэффициенты показывают степень связи входа i со

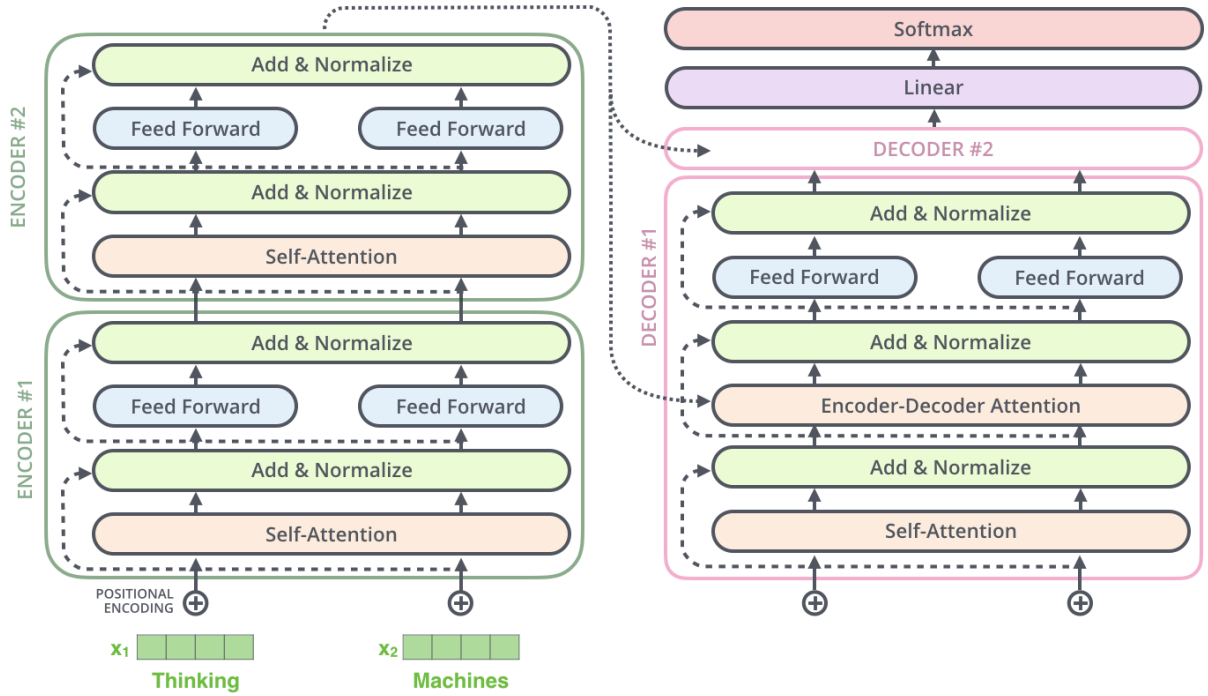

Рис. 3: Внутренняя структура кодировщика и декодировщика (рисунок взят из [1]).

входом j. Они вычисляются по формуле (4).

$$c_{ij} = (q_i; k_j) \tag{4}$$

где $(a; b)$ — скалярное произведение векторов a и b.

На выходе получается N векторов c_i размерности N, где N — это количество входных векторов, а c_i — это вектор коэффициентов для входа i.

Далее необходимо нормализовать каждый вектор по формуле (5).

$$sc_i = Softmax\left(\frac{c_i}{\sqrt{d_k}}\right) \tag{5}$$

где d_k — размерность вектора ключа;

sc_i — нормализованный вектор коэффициентов.

Выходные векторы механизма внимания считаются по формуле (6).

$$z_i = \sum_{j=1}^{N} sc_{ij} * v_j \tag{6}$$

Если представлять набор входных векторов как матрицу, то вычисления сводятся к нескольким простым формулам. Изначально вычисляются матрицы запроса, ключа и значения по формулам (7), (8) и (9) соответственно.

$$Q = X \times W_Q \tag{7}$$

$$K = X \times W_K \tag{8}$$

$$V = X \times W_V \tag{9}$$

где X — матрица входных векторов.

Матрица выходов, в свою очередь, считаются по формуле (10).

$$Z = Softmax\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) \times V \tag{10}$$

где Z — матрица выходных векторов.

1.3.2. Механизм множественного внимания

Множественное внимание представляет из себя несколько параллельных механизмов внимания, которые имеют свои собственные веса. Это позволяет определять больше различных связей между элементами входных данных, тем самым более точно решать поставленную задачу. В механизме множественного внимания каждый отдельный механизм внимания называют «головы».

Принцип работы данного механизма заключается в том, что один и тот же вход поступает на каждую голову и выходы считаются параллельно. Далее выходы из каждой головы конкатенируются, а затем умножаются на выходную матрицу, полученную во время обучения. Это позволяет объединить информацию, полученную из всех голов, и нормализовать размерность выхода. На Рис. 4 изображен весь процесс работы механизма множественного внимания.

Именно множественное внимание чаще всего встречается в моделях, построенных на архитектуре Transformer.

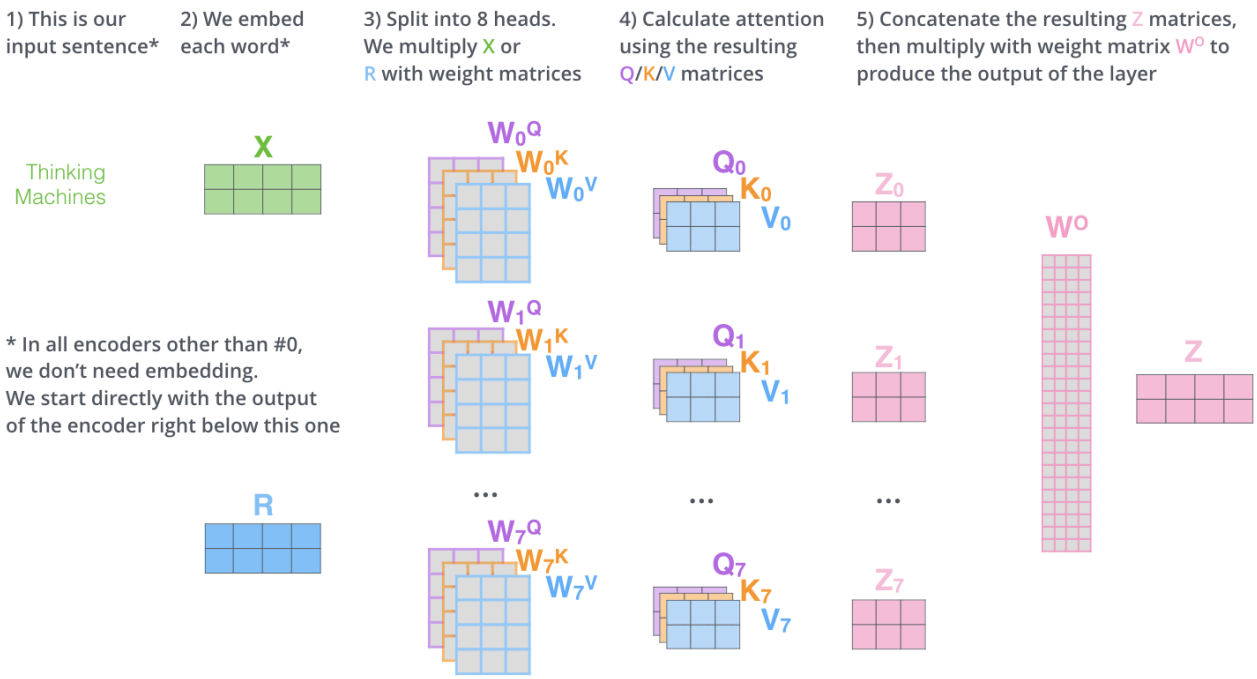

Рис. 4: Процесс механизма множественного внимания (рисунок взят из [1]).

1.3.3. Суммирование и нормализация

Данный слой складывает матрицы входов и выходов предыдущего слоя, а потом применяет к полученному вектору функцию $LayerNorm$. На Рис. 5 изображен данный слой подробнее.

Рассмотрим подробнее функцию $LayerNorm$. Обозначим сумму матриц выходов и входов за Y. Данная функция выполняется последовательно для всех векторов Y_i в матрице Y. В первую очередь вычисляются средний коэффициент μ_i и дисперсия σ_i^2 по формулам (11) и (12) соответственно.

$$\mu_i = \frac{1}{K}\sum_{k=1}^{K} Y_{ik} \tag{11}$$

$$\sigma_i^2 = \frac{1}{K}\sum_{k=1}^{K}(Y_{ik} - \mu_i)^2 \tag{12}$$

где K — размерность вектора Y_i.

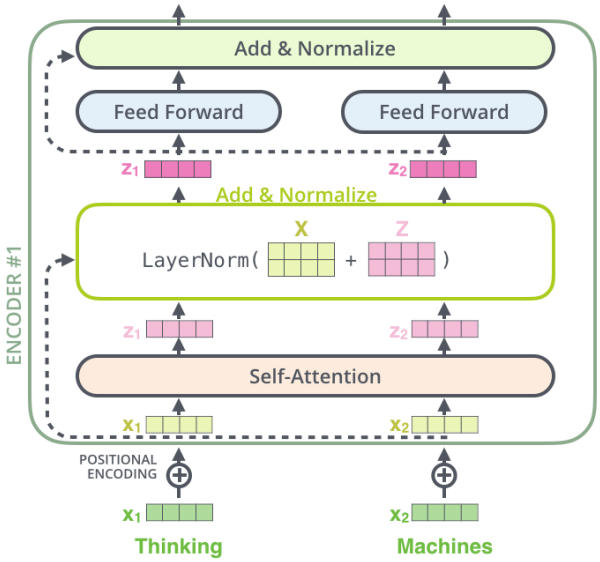

Рис. 5: Процесс суммы и нормализации (рисунок взят из [1]).

Далее происходит сама нормализация вектора Y_i. Она вычисляется по формуле (13). ϵ необходимо, чтобы не произошло деление на ноль.

$$\hat{Y}_i = \frac{Y_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}, \tag{13}$$

где ϵ — достаточно малое ненулевое число;
$\hat{Y}_i$ — нормализованный вектор Y_i.

В конце нормализованный вектор масштабируется, сдвигается и передается в следующий слой. Векторы масштабирования и сдвига являются параметрами модели, которые были получены во время обучения. Выходной вектор вычисляется по формуле (14).

$$N_i = \gamma \circ \hat{Y}_i + \beta, \tag{14}$$

где γ — вектор масштабирования;
β — вектор сдвига;
$\circ$ — произведение Адамара (поэлементное произведение).

1.3.4. Квантизация

Квантизация — это метод оптимизации модели нейронной сети, при котором параметры модели преобразуются из типа Float к типам Int8

или UInt8. Данный метод позволяет ускорить запуск модели и уменьшить размер модели при незначительной потери в точности. Ускорение достигается за счёт того, что операции сложения и умножения для целочисленных типов выполняются быстрее, чем для типов с плавающей запятой. Оптимизация размера модели достигается за счёт того, что типы Int8 и UInt8 занимают 1 байт памяти, в то время как Float занимает 4 байта памяти.

Процесс квантизации числа $x \in [\alpha; \beta]$ в целое число $x_q \in [\alpha_q; \beta_q]$ описывается формулой (15).

$$x_q = \text{clip}\Big(\text{round}\big(\frac{1}{s}x + z\big), \alpha_q, \beta_q\Big)$$

$$\text{clip}(x, l, u) = \begin{cases} l & \text{если } x < l \\ x & \text{если l} \leq x \leq u \\ u & \text{если } x > u \end{cases} \tag{15}$$

Обратный процесс описывается формулой (16).

$$x = s(x_q - z) \tag{16}$$

Параметры s и z являются дополнительными параметрами квантизации и вычисляются отдельно для каждой матрицы. Они вычисляются по формулам (17) и (18) соответственно.

$$s = \frac{\beta - \alpha}{\beta_q - \alpha_q} \tag{17}$$

$$z = \text{round}\left(\frac{\beta\alpha_q - \alpha\beta_q}{\beta - \alpha}\right) \tag{18}$$

Стандартно параметры α и β определяют как максимальное и минимальное значения в квантизуемой матрице, а α_q и β_q – как максимальное и минимальное значения целочисленной переменной.

2. Разработанное решение

Для реализации компонентов кодировщика и декодировщика в формате ONNX необходимо реализовать несколько операторов:

- Attention;
- QAttention;
- LayerNormalization;
- SkipLayerNormalization;
- EmbedLayerNormalization;
- DynamicQuantizeLinear;
- DequantizeLinear.

Реализованные операторы были интегрированы в библиотеку KInference.

2.1. Реализация операторов

На Рис. 6 изображена архитектура реализации требуемых операторов внутри библиотеки (голубым цветом выделены существующие классы библиотеки).

В момент загрузки модели в библиотеку с помощью парсера считываются и создаются все необходимые операторы и их атрибуты. Для инициализации операторов используется фабрика OperatorFactory.

В библиотеке реализован специальный абстрактный класс Operator, наследниками которого представлены реализованные в библиотеке операторы. В данном классе определена абстрактная функция `apply`, которая вызывается библиотекой во время вычислений в узле оператора. Данная функция производит необходимые вычисления для определенного оператора и возвращает результат. Таким образом, классы новых операторов должны быть наследниками класса Operator и реализовывать собственную версию функции `apply`.

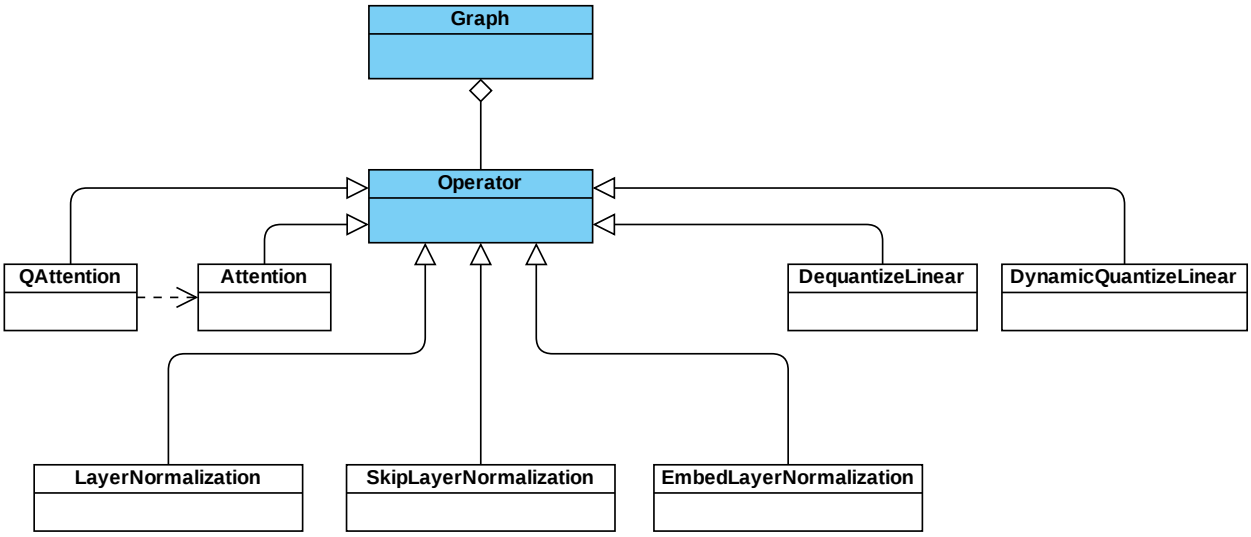

Рис. 6: Архитектура решения.

2.1.1. Операторы внимания

Операторы Attention и QAttention реализуют механизм множественного внимания в формате ONNX. На Рис. 7 изображена архитектура реализации механизма внимания.

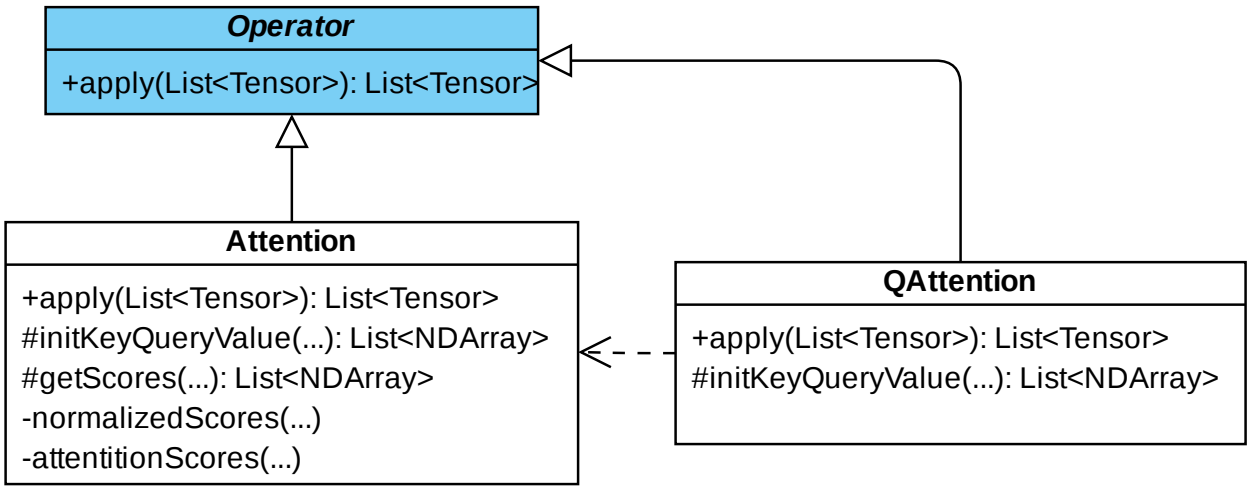

Рис. 7: Архитектура реализации механизма внимания.

Оператор Attention на вход функции `apply` получает матрицы весов, смещений и входные данные. В этой функции данные обрабатываются, и при помощи функции `initKeyQueryValue` создаются матрицы запросов, ключей и значений. Далее вызывается функция `getScores`, в которую передаются вычисленные матрицы. Она, в свою очередь, при помощи функции `normalizedScores` вычисляет нормализованные

коэффициенты, а потом при помощи функции `attentionScores` вычисляет выходы механизма внимания. В конце данная функция приводит ответы к указанному в спецификации виду и возвращает обратно в функцию `apply`, а она возвращает полученные выходы обратно в модель.

Оператор QAttention выполняет аналогичные действия, но на вход функции `apply` получает квантизованные веса и входы.

2.1.2. Операторы нормализаций

Оператор LayerNormalization реализует механизм нормализации векторов в формате ONNX.

Данный оператор на вход функции `apply` получает входную матрицу, вектор масштабирования и вектор смещения, далее для каждого вектора в матрице вычисляются средние коэффициенты и дисперсия. В конце все векторы нормализуются, конкатенируются и возвращаются обратно в модель.

Оператор SkipLayerNormalization является слиянием операторов суммирования и нормализации и выполняет аналогичные действия.

Оператор EmbedLayerNormalization является слиянием механизмов векторного представления слова и нормализации и выполняет аналогичные действия.

2.1.3. Операторы квантизации

Оператор DynamicQuantizeLinear реализует механизм квантизации. Данный оператор на вход функции `apply` получает матрицу, которую необходимо квантизовать. Для данной матрицы вычисляются параметры s и z по формулам (17) и (18) для типа UInt8, по формуле (15) вычисляется квантизованная версия матрицы и возвращается обратно в модель.

Оператор DequantizeLinear реализует механизм деквантизации. На вход функции `apply` передается квантизованная матрица и параметры s и z, матрица деквантизуется и возвращается обратно в модель.

2.2. Оптимизация представления многомерных массивов

В ходе тестирования было замечено, что Transformer модели работают достаточно медленно. Проанализировав работу библиотеки, было замечено, что производительность теряется в операции умножения больших матриц, которые представлены многомерными массивами. Стандартная реализация подразумевает хранение всех данных многомерного массива в одном массиве примитивов. Было установлено, что начиная с определенного размера массива примитивов платформа JVM недостаточно оптимизирует работу с кэш-памятью процессора, из-за чего происходит большое количество обращений к оперативной памяти.

Экспериментально было замечено, что если разбивать массив на блоки размера 512 и меньше, то платформа JVM достаточно хорошо оптимизирует работу с кэш-памятью процессора и обращений к оперативной памяти становится меньше. В ходе эксперимента измерялась скорость работы операции умножения для матриц $N \times K$ и $K \times M$. В Таблице 1 представлены средние значения на 100 запусков и разброс значений для данных замеров.

Параметры матриц			Стандартная реализация массивов, мс	Блочная реализация массивов, мс			
N	M	K		размер блока 256	размер блока 512	размер блока 1024	размер блока 2048
2048	2048	2048	5315 ± 53	1125 ± 3	1063 ± 3	1084 ± 4	1658 ± 12
1024	2048	1024	1240 ± 15	276 ± 1	212 ± 2	211 ± 2	252 ± 2
1024	1024	1024	612 ± 4	119 ± 1	108 ± 1	106 ± 1	106 ± 1
2048	1024	2048	2502 ± 28	477 ± 2	430 ± 2	541 ± 3	538 ± 3

Таблица 1: Результаты замеров скорости умножения матриц для различных реализаций.

Для решения проблемы с производительностью было принято решение изменить представление многомерных массивов в библиотеке.

2.2.1. Реализация внутреннего представления данных для многомерных массивов

Многомерные массивы в библиотеке реализованы классом PrimitiveNDArray. На этапе компиляции при помощи внешнего инструмента[5] на основе данного класса генерируются реализации для всех примитивных типов, таких как Int, Float, Double и так далее. Это позволяет не изменять реализацию многомерных массивов для каждого типа отдельно, а генерировать из общего кода для всех типов сразу.

Для оптимизации работы многомерных массивов в рамках данной работы была разработала новую архитектура представления внутренних данных в виде блочных массивов. На Рис. 8 изображена данная архитектура.

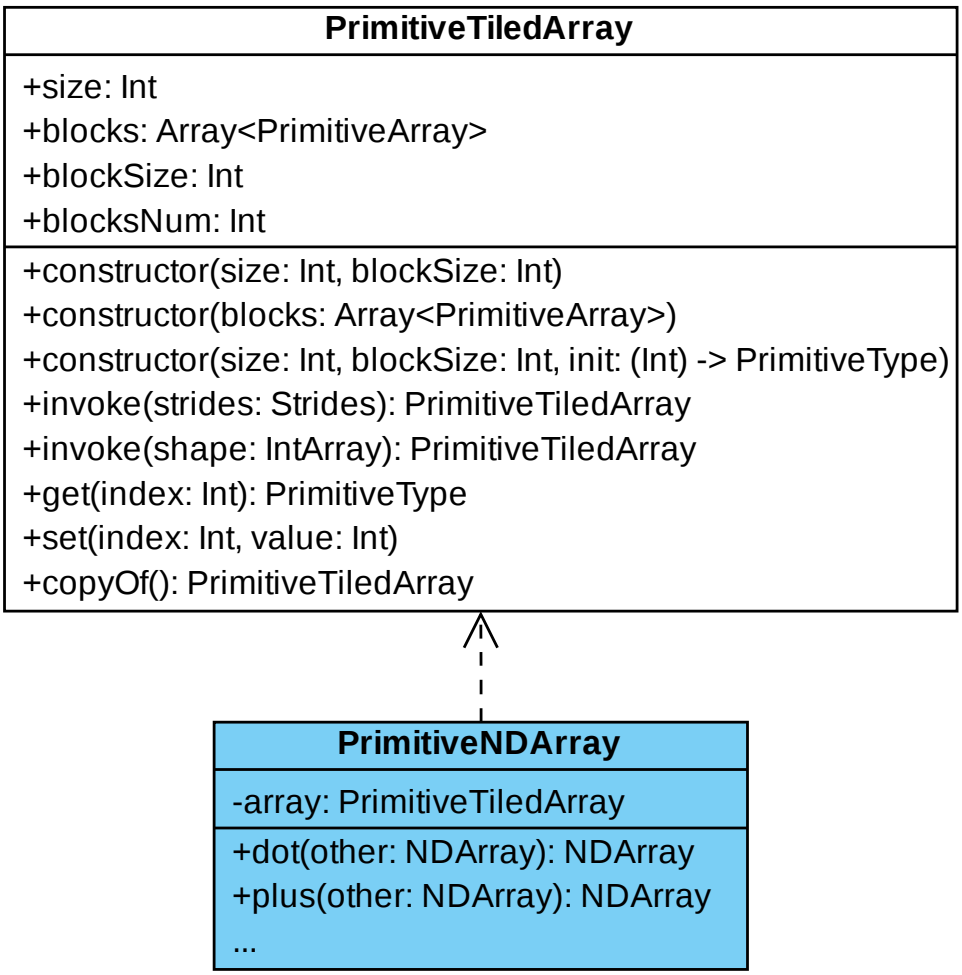

Рис. 8: Архитектура реализации многомерных массивов.

[5] https://github.com/JetBrains-Research/kinference-primitives

При создании многомерного массива для инициализации переменной `array` используется функция `invoke` для блочных массивов, в которую передается размерность многомерного массива. В данной функции вычисляется размер блока относительно последнего измерения многомерного массива. Если размер последнего измерения меньше 512, то размер блока берется равным данному размеру, иначе берется наименьшее, больше или равное 512, число, которое делит размер последнего измерения без остатка.

После реализации блочных массивов была произведена оптимизация всех операций многомерных массивов, используя новое блочное представление. В итоге это позволило достичь необходимой скорости работы Transformer моделей. Как мы видим в Таблице 1, реализация с блочным представлением массивов с размером блока 512 работает в несколько раз быстрее стандартной реализации.

3. Апробация

Для проверки реализованных операторов на корректность и соответствие спецификациям были созданы тестовые модели, состоящие из тестируемого оператора. Для создания моделей была использована библиотека ONNX для языка Python. Далее были сгенерированы необходимые входы для каждой модели и экспортированы в формат ONNX. Для создания заведомо верных результатов операторов использовалась библиотека ONNX Runtime. Данные тесты были подключены к библиотеке. К тому же было проведего тестирование библиотеки на реальных Transformer моделях: GPT-2 и BERT. Для этого аналогичным образом были сгенерированы заведомо верные результаты моделей при помощи ONNX Runtime и также подключены к нашей библиотеке. Все подключенные тесты успешно выполняются для KInference.

3.1. Использование в Grazie Platform

На данный момент библиотека KInference используется для запуска моделей GPT-2 и BERT в Grazie Platform[6] — платформе для работы с текстами на естественном языке, которая разрабатывается в JetBrains.

3.1.1. Автодополнение текста

Grazie Platform предоставляет возможность автодополнения текста на английском языке при помощи модели GPT-2. На основе данной модели был проведён эксперимент, в котором измерялась скорость работы библиотеки на стандартной реализации и на блочной реализации массивов. Для измерения скорости использовались внутренние возможности языка Kotlin.

Данная модель запускается в момент генерации текста для автодополнения. Сначала с помощью внешних утилит текст разбивается на токены, извлекаются последние N токенов, составляющие контекст, и преобразуются в численные вектора. Далее в модель передаются по-

[6] `https://github.com/JetBrains/intellij-community/tree/master/plugins/grazie`

лученные вектора, на основе которых она считает первый выходной токен. К тому же матрицы ключей и значений из всех механизмов внимания, сохраняются и переиспользуются для вычисления последующих выходных токенов.

В последующих запусках матрицы ключей и значений вычисляются только для предыдущего созданного выходного токена и конкатенируются с матрицами, полученными в предыдущем запуске. Все последующие вычисления остаются неизменными. Полученные токены декодируются обратно в слова и выводятся на экран в качестве подсказки. Первый запуск является наиболее трудоемким, так как на нем обрабатывается наибольшее количество входных токенов. Замеры проводились для различных размеров контекста для первого запуска и последующих.

В Таблице 2 представлены средние значения на 100 запусков и разброс значений для данных замеров.

Длина контекста, количество токенов	Стандартная реализация массивов, мс		Блочная реализация массивов, мс	
	Первый запуск	Последующие запуски	Первый запуск	Последующие запуски
50	278 ± 2	34 ± 1	56 ± 1	27 ± 1
100	610 ± 3	37 ± 1	95 ± 1	27 ± 1
200	1488 ± 5	44 ± 1	180 ± 1	27 ± 1
256	2105 ± 3	49 ± 1	228 ± 1	29 ± 1

Таблица 2: Результаты замеров на модели GPT-2.

В итоге эксперимента было установлено, что использование блочных массивов при первом запуске во много раз эффективнее, чем стандартных, и незначительно лучше для последующих. Это обусловлено тем, что при первом запуске объемы входных данных значительно больше, чем на последующих.

3.1.2. Исправление грамматических ошибок

Также в Grazie Platform осуществляется исправление грамматических ошибок в тексте на английском языке при помощи модели BERT.

На основе данной модели был проведен аналогичный эксперимент, в котором измерялась скорость работы библиотеки на стандартной реализации и на блочной реализации массивов.

Текст, который необходимо проверить на ошибки, аналогично разбивается на токены и передается в модель. В ответ модель возвращает токены слов, которые необходимо исправить, и несколько вариантов исправлений для каждого слова. Замеры проводились для различного количества входных токенов.

В Таблице 3 представлены средние значения на 100 запусков и разброс значений для данных замеров.

Количество токенов	Стандартная реализация массивов, мс	Блочная реализация массивов, мс
32	70 ± 1	24 ± 1
64	148 ± 2	40 ± 1
128	309 ± 4	77 ± 1
256	665 ± 7	155 ± 1
512	1525 ± 6	347 ± 1

Таблица 3: Результаты замеров на модели BERT

В итоге данного эксперимента было установлено, что использование блочных массивов в несколько раз эффективнее, чем стандартных, аналогично предыдущему эксперименту. Таким образом, проблема производительности была устранена при реализации блочных массивов.

Все замеры проводились на стенде со следующими техническими характеристиками:

- операционная система macOS Big Sur 11,3;
- 8-ядерный процессор Intel Core i9 с тактовой частотой 2,3 ГГц;
- 16 Гб оперативной памяти DDR4 с тактовой частотой 2667 МГц.

4. Заключение

В ходе выполнения данной работы были достигнуты следующие результаты:

- реализованы и интегрированы основные компоненты Transformer архитектуры: кодировщик, декодировщик и механизм внимания;
- обеспечена эффективность разработанного алгоритма запуска нейронной сети;
  - реализовано блочное представление многомерных массивов в памяти;
- реализована поддержка квантизованных моделей;
- проведена апробация полученного решения для Transformer моделей: GPT-2 и BERT.

Ссылка на репозиторий с исходным кодом: `https://github.com/JetBrains-Research/kinference`, разработка велась под аккаунтом cupertank.

Список литературы

[1] Alammar Jay. The Illustrated Transformer. — URL: https://jalammar.github.io/illustrated-transformer (дата обращения: 2020-12-12).

[2] Attention is All You Need / Ashish Vaswani, Noam Shazeer, Niki Parmar et al. — 2017. — Access mode: https://arxiv.org/pdf/1706.03762.pdf.

[3] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding / Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova // arXiv preprint arXiv:1810.04805. — 2018. — Access mode: https://arxiv.org/pdf/1810.04805.pdf.

[4] Groenwold Sophie, Ou Lily, Parekh Aesha et al. Investigating African-American Vernacular English in Transformer-Based Text Generation. — 2020. — 2010.02510.

[5] Kim Jiwon, Lee Jung Kwon, Lee Kyoung Mu. Deeply-Recursive Convolutional Network for Image Super-Resolution. — 2016. — 1511.04491.

[6] Language Models are Unsupervised Multitask Learners / Alec Radford, Jeff Wu, Rewon Child et al. — 2019. — Access mode: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[7] MNN: A Universal and Efficient Inference Engine / Xiaotang Jiang, Huan Wang, Yiliu Chen et al. // MLSys. — 2020.

[8] Zhang Shucong, Loweimi Erfan, Bell Peter, Renals Steve. On the Usefulness of Self-Attention for Automatic Speech Recognition with Transformers. — 2020. — 2011.04906.

[9] ONNX. — Open Neural Network Exchange. — URL: https://onnx.ai/ (дата обращения: 2020-12-12).