МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра дифференциальных уравнений и системного анализа

Оптимизация производительности в BERT-подобных архитектурах нейронных сетей

Курсовая работа

Киселёва Павла
Владимировича

студента 3 курса,
специальность 1-31 03 09
Компьютерная математика
и системный анализ

Научный руководитель:
ассистент А. П. Тишуров

Минск, 2020

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В последнее время для решения задач обработки естественного языка широко используются BERT-подобные архитектуры нейронных сетей. Они показывают наилучшие результаты в тестах на понимание языка (GLUE), ответах на вопросы (SQuAD), создании логических выводов (SWAG) и т.д.

Тем не менее высокое качество часто достигается за счет огромного количества обучаемых параметров, что, в свою очередь, негативно сказывается на производительности моделей. Этот вопрос встает особенно остро при применении BERT-подобных архитектур в практических задачах, где время предсказания и объём модели играют существенную роль. Таким образом перед сообществом встал вопрос об оптимизации производительности данных архитектур.

Отдельные ученые и компании-лидеры индустрии предложили множество подходов различной сложности для решения данной проблемы. В качестве примеров можно привести оптимизаторы графов, дистилляции знаний, обрезание параметров и др.

В данной работе продемонстрированы некоторые доступные способы оптимизации BERT-подобных архитектур на примере задачи Google’s Natural Questions.

ГЛАВА 1
ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 BERT и BERT-подобные архитектуры нейронных сетей

1.1.1 BERT

BERT (анг. Bidirectional Encoder Representations from Transformers) - это модель представления языка, основанная на части кодировщика (англ. encoder) архитектуры Transformer. В отличие от предыдущих моделей представления языка, BERT спроектирован для предварительной тренировки глубоких двунаправленных представлений на основе неразмеченного текста таким образом, чтобы представления зависели как от левого, так и от правого контекста на всех слоях.

В результате, предварительно натренированная модель BERT может быть подстроена под многие задачи добавлением одного лишь выходного слоя.

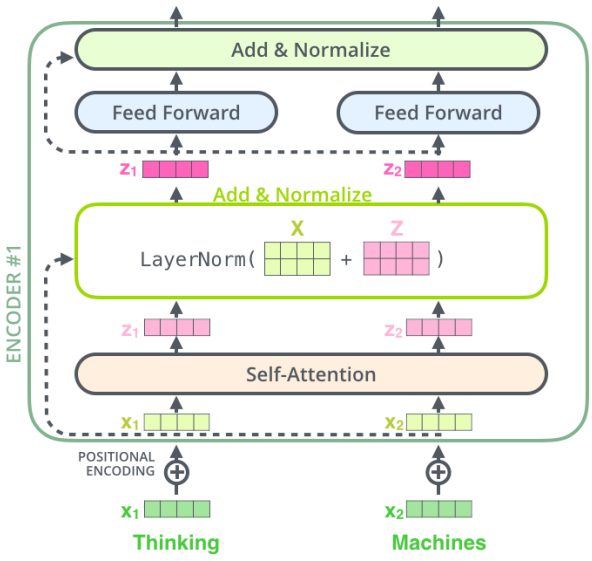

Рисунок 1 Простой пример кодировщика архитектуры Transformer

Как уже было упомянуто, в основе BERT лежит многослойный двунаправленный кодировщик из архитектуры Transformer. Только в базовой реализации используется 12 блоков, изображенных на Рисунке 1, где в каждом блоке размер скрытого слоя равен 768, а число вариантов автовнимания (англ. self-attention heads) равно 12. Таким образом BERT насчитывает 110 миллионов обучаемых параметров в своей базовой конфигурации и 340 миллионов обучаемых параметров в расширенной конфигурации.

1.1.2 BERT-подобные архитектуры

После публикации научной статьи об архитектуре BERT и соответствующей реализации появилось множество ответвлений и модификаций изначальной архитектуры. Приведем некоторые примеры.

RoBERTa использует оригинальную BERT архитектуру, но уделяет больше внимания обучению модели и настройке гиперпараметров. Использует около 355 миллионов обучаемых параметров.

ALBERT предлагает способы снижения количества параметров для увеличения производительности.

Существуют также примеры архитектур, которые не являются гибридами BERT, но похожи на него либо за счет использования составных частей Transformer, либо за счет схожести решаемых задач.

Один из примеров: **GPT-2** - языковая модель, ставшая известной благодаря качественной генерации связного текста, который сильно напоминает текст, написанный человеком. Содержит 1.5 миллиарда параметров.

1.2 Проблема производительности в BERT-подобных архитектурах

1.2.1 Описание проблемы

Несмотря на то, что вышеупомянутые модели показывают лучшие результаты в задачах обработки естественного языка, они часто имеют несколько сотен миллионов параметров. Это является проблемой при их применении на практике, так как во многих случаях производительность является крайне важной характеристикой рабочей модели.

Говоря о производительности модели, можно выделить следующие важные области:

- Скорость тренировки
- Скорость предсказания
- Объём потребляемой памяти

В данной работе мы сосредоточимся на увеличении скорости предсказания.

1.2.2 Существующие подходы к решению

Большинство существующих подходов к оптимизации производительности BERT-подобных архитектур можно разделить на две группы:

- Оптимизация производительности за счет изменения архитектуры.
- Оптимизация производительности без изменения архитектуры.

К первой группе можно отнести такие подходы как ALBERT, DistillBERT, TinyBERT и другие. Главная идея здесь состоит в том, чтобы изменить архитектуру таким образом, чтобы при увеличенной производительности качество работы не пострадало.

Это может достигаться за счет продуманного уменьшения количества параметров, как, например, в случае ALBERT. Некоторые другие модели используют так называемую дистилляцию знаний, где более “легкая” в смысле производительности модель тренируется копировать поведение более “тяжелой” модели. Пример такого подхода: DistillBERT.

Подходы же из второй группы стараются оптимизировать процесс тренировки или предсказания модели без явного изменения архитектуры. Часто эти подходы носят низкоуровневый характер и сконцентрированы на таких вещах, как оптимизация кода реализации, оптимизация вычислений, работа с представлением модели в памяти и т.д.

В данной работе будут рассмотрены подходы именно из второй группы. Благодаря этим подходам, разработчик может ускорить предсказания своей модели без значительных изменений в архитектуре и без перетренировки модели.

Стоит, однако, отметить, что подходы из второй группы часто зависимы от используемого фреймворка и/или аппаратного обеспечения. Методы, рассмотренные в данной работе, были опробованы в рамках фреймворка Tensorflow 2.0. Аппаратное обеспечение: видеокарта Tesla-VCIE-V100-16GB.

1.3 Задача Google’s Natural Questions и подходы к решению

Для тестирования подходов оптимизации была использована задача Google’s Natural Questions.

1.3.1 Описание задачи

Для стимуляции развития в области ответов на вопросы в открытом домене компания Google создала Natural Question (NQ) датасет. NQ датасет содержит

вопросы настоящих пользователей и требует от системы обработать некоторую статью из Википедии, которая может как содержать, так и не содержать ответ на поставленный вопрос. Дополнительная сложность состоит в том, что от системы требуется найти:

- Длинный ответ на вопрос (абзац, кусок абзаца).
- Короткий ответ на вопрос (слово, словосочетание), если он уместен.
- Ответ “Да/Нет”, если он уместен.

В декабре 2019 года компания Google запустила соревнование на площадке Kaggle, предложив участникам решить несколько упрощенный вариант задачи (HTML страница была избавлена от лишних тегов, были предложены кандидаты на “длинные” ответы).

1.3.2 BERT Joint

Один из подходов, который, в том числе, в разных вариациях стал популярным в Kaggle соревновании, был предложен работниками Google в технической заметке “A BERT Baseline for the Natural Questions”. В репозитории на Github и в соревновании он получил название BERT Joint. Именно этот метод, как хороший пример применения BERT, использовался в данной работе. Более подробное его описание и некоторые детали реализации будут приведены далее.

1.3.3 Метрики качества в задаче Natural Questions

Для оценки качества предсказаний в задаче Natural Qusetions используются достаточно стандартные метрики: F1-score, precision и recall. При этом каждая из метрик вычисляется отдельно для “длинных” и “коротких” ответов.

Оценка качества в Kaggle соревновании отличалась тем, что объединяла качество для разных типов ответов с помощью метрики micro F1-score.

В данной работе качество работы модели оценивалось с помощью официального скрипта для оценки: nq_eval.py. Скрипт сравнивает файл с предсказанием и файл с разметкой, возвращая все ключевые метрики. Для возможности быстрой оценки улучшения или ухудшения работы модели, в скрипт был добавлен вывод micro F1-score метрики.

1.4 Резюме

В данной главе была введена проблема производительности в BERT-подобных архитектурах нейронных сетей. Затем была рассмотрена задача Google’s Natural Questions, используемые метрики, а также одно из её решений: BERT Joint.

В следующих главах остановимся более подробно на описании реализации подхода BERT Joint, а также на подходах к оптимизации его производительности.

ГЛАВА 2
РЕАЛИЗАЦИЯ РЕШЕНИЯ ЗАДАЧИ GOOGLE'S NATURAL

Реализация BERT и некоторых утилит были взяты из Github репозитория google-research. Некоторые элементы пред- и постобработки были позаимствованы из решения участника Marcos Martins Marchetti на Kaggle. Весь исходный код расположен в репозитории https://github.com/kisialiou/NQ/. В данной главе будут рассмотрены основные этапы решения задачи Natural Questions с помощью BERT Joint.

2.1 Предобработка данных

Основной целью предобработки данных является приведение входных данных в формат, который может потребляться моделью BERT, с сохранением важной информации. Далее будет рассматриваться обработка данных без аннотации для последующего предсказания. Подробности и соответствующий исходный код находится в файле preprocessing.py.

Изначально данные хранятся в jsonl файле, где каждая строка представляет следующую структуру:

- Текст статьи**.** Полный текст статьи из Википедии, в которой потенциально может содержаться ответ на вопрос. Присутствуют некоторые HTML-теги, несущие в себе информацию о структуре статьи.
- Уникальный идентификатор примера.
- Массив кандидатов на "длинный" ответ, где каждый кандидат представлен как:

    - Номер токена начала ответа.
    - Номер токена конца ответа.
- Текст вопроса.

Целью первого преобразования является создание так называемого контекста: последовательности токенов, среди которых будут искаться ответы на вопросы. В данном решении контекст получается объединением всех кандидатов на “длинный” ответ.

Первое преобразование приводит каждый пример к следующему виду:

- Уникальный идентификатор примера.
- Текст вопроса.
- Контекст. Текст, полученный в результате объединения всех кандидатов на “длинный” ответ.
- Последовательность номеров токенов контекста. По сути дубликат предыдущего пункта, где каждый токен представлен соответствующим ему порядковым номером в исходном тексте.

Второе преобразование использует алгоритм токенизации SentencePiece, который разбивает некоторые слова на “подслова” (англ. subwords) для получения формата входа необходимого для BERT. Таким образом каждому токену контекста ставится в соответствие один или несколько SentencePiece токенов. В дополнительной структуре сохраняется прямое и обратное соответствие между старыми и новыми токенами.

Поскольку BERT принимает на вход последовательности только фиксированной длины, то возникает необходимость в подходе для обработки контекстов, длина которых больше допустимой длины входа. Поэтому на третьем шаге для формирования входов модели выбираются части контекста определенной длины. Части формируются путем прохода по контексту интервалом

фиксированного размера с заданным шагом. Все токены оказавшиеся в границах интервала формируют подпоследовательность контекста, которую далее будем называть “спан” (англ. span - диапазон). Таким образом для каждого примера формируется некоторое количество пересекающихся спанов.

Именно из спанов и исходного вопроса формируется вход для модели. Он имеет следующую структуру:

[CLS] <question> [SEP] [SEP], где

[CLS] - специальный токен, отвечающий за классификацию всего входа,

<question> - последовательность SentencePiece токенов соответствующих тексту вопроса,

[SEP] - специальный токен, обозначающий границу между составными частями входа,

** - последовательность SentencePiece токенов, соответствующих тексту спана.

2.2 Модель

Для создания модели использовался Keras API Tensorflow 2.0. Реализация BERT была взята из репозитория Google Research.

Схематически модель представлена на Рисунке 2.

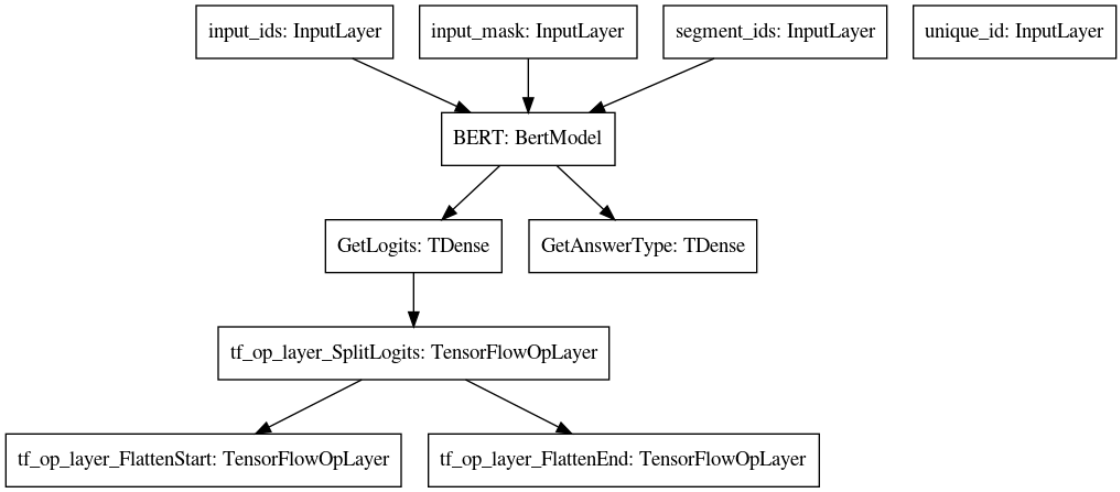

Рисунок 2 Схема модели BERT Joint

BertModel принимает на вход три последовательности одинаковой длины. Первая несет информацию непосредственно о токенах, составляющих вход, а две другие содержат дополнительную информацию. Более подробно:

input_ids - последовательность индексов, соответствующих Sentence Piece токенам.

input_mask - маска из нулей и единиц. Единицы соответствуют токенам, которые изначально присутствовали в последовательности. Нули соответствуют токенам, которые появились в результате паддинга.

segment_ids - последовательность чисел, в которой одинаковые числа обозначают принадлежность соответствующих им токенов к одному и тому же сегменту входа (вопрос или спан-ответ).

BertModel имеет два выхода:

1. Последовательность эмбеддингов соответсвующих входным токенам.
2. “Классификационный” эмбеддинг соответствующий [CLS] токену.

Каждый из выходов направляется в отдельный полносвязный слой. В результате получаем:

1. Каждому токену соответствует два числа, каждое из которых отражает с какой уверенностью модель называет этот токен началом или концом ответа соответственно.
2. Вектор из пяти элементов, где каждое число отражает с какой уверенностью модель предполагает наличие соответствующего типа ответа в данном спане.

2.3 Обработка предсказаний модели

Итак, каждое из предсказаний модели содержит:

- Уникальный идентификатор. Позволяет определить к какому из всех изначальных примеров относится данное предсказание.
- Последовательность рейтингов, отражающих уверенность модели в том, что соответствующий токен является началом ответа. Будем называть эту последовательность “start logits”.
- Последовательность рейтингов, отражающих уверенность модели в том, что соответствующий токен является концом ответа. Будем называть эту последовательность “end logits”.
- Вектор из пяти элементов, где каждое число отражает с какой уверенностью модель предполагает наличие соответствующего типа ответа в данном спане. В простейшей реализации постобработки данная информация не используется, поэтому далее опустим её.

К каждой из таких структур применяются следующие шаги:

1. В каждой из последовательностей, start logits и end logits, выбирается k наибольших значений. В нашей реализации k = 5.
2. Из выбранных значений формируются всевозможные валидные пары.
3. Для каждой пары вычисляется рейтинг на основе эвристики.

Далее вышеописанные пары делятся на две группы:

- Длинные ответы. Пара попадает в эту группу, если соответствующие ей токены содержат одного из кандидатов на длинный ответ.
- Короткие ответы - все остальные случаи.

Из каждой группы выбирается пара с лучшим рейтингом и оставляется, если рейтинг выше определенного порога. В противном случае считается, что ответ отсутствует.

2.4 Резюме

Таким образом каждому примеру из исходного датасета ставится в соответствие ноль, один или несколько ответов на вопрос.

Архитектура, описанная в данной главе, использовалась далее для тестирования подходов к оптимизации производительности.

Ключевые моменты:

1. Разбиение большого текста на маленькие кусочки для формирования входа.
2. Формирование структур вида [CLS] <question> [SEP] [SEP].
3. BERT - главная составная часть модели.
4. Интерпретация выхода на основе эвристик.

В следующей главе будет представлена статистика по базовой производительности данной модели, а также различные способы её оптимизации.

ГЛАВА 3
ОПТИМИЗАЦИЯ ПРОИЗВОДИТЕЛЬНОСТИ В ЗАДАЧЕ GOOGLE'S NATURAL QUESTIONS

3.1 Общие сведения

- Для экспериментов использовалась модель, натренированная для Kaggle соревнования участником Marcos Martins Marchetti.
- Набор данных для экспериментов представлял из себя случайную выборку из NQ датасета состоящую из 30737 примеров.
- В качестве метрики производительности использовалось время, потраченное на предсказание результатов для всего тестировочного датасета. Во всех экспериментах использовался одинаковый размер батча равный 32.
- Все эксперименты проводились на GPU Tesla-VCIE-V100-16GB.
- Для отслеживания параметров и метрик в эксперименте использовалась библиотека MLflow.
- До применения каких-либо способов оптимизации было произведено измерение времени предсказания "не оптимизированной" модели. (см. Таблицу 1)
- Все последующие эксперименты проводились в тех же условиях, что и для изначальной модели. Гиперпараметры, параметры запуска и данные не изменялись.

Метрика \ Модель	BERT Joint Baseline
Время предсказания	36 087 с
Micro F1-score	0.466

Таблица 1 Базовая производительность

3.2 Графовые оптимизаторы

3.2.1 Описание подхода

В Tensorflow существует два режима вычислений:

- Графовый режим.
- Eager режим (англ. eager - нетерпеливый).

Графовый режим представляет из себя декларативную парадигму, где разработчик описывает составные части графа и связи между ними. Во время выполнения программы описанный граф компилируется, а затем производятся все необходимые вычисления.

Eager режим представляет императивную парадигму, где все описанные команды выполняются последовательно, без предварительной компиляции. Такой подход является более медленным, но, в то же время, более удобным при отлавливании ошибок.

Несмотря на то, что графовый режим уже является своего рода оптимизацией производительности, существуют модули, реализованные в рамках Tensorflow, которые позволяют ускорить работу модели в графовом режиме за счет различных

оптимизаций. Подробное описание данных модулей выходит за рамки данной работы, поэтому ограничимся кратким обзором использованных оптимизаторов:

- XLA (Accelerated Linear Algebra) компилятор - компилятор для алгебраических вычислений.
- Grappler - набор графовых оптимизаторов, анализирующих граф и изменяющих его структуру для улучшения производительности.

3.2.2 Реализация подхода

В случае с BERT добавление XLA и Grappler не требует от разработчика изменения никакого исходного кода. Главное требование: наличие графа. При использовании Keras API в Tensorflow 2.0 статический граф формируется по умолчанию.

Таким образом для активации XLA компилятора требуется лишь прописать соответствующую команду. Активация Grappler отличается лишь тем, что в команду необходимо передать желаемые графовые оптимизаторы. Их исчерпывающий список может быть найден в официальной документации.

3.2.3 Результаты

Для тестирования данного подхода было проведено два эксперимента:

1. Эксперимент включающий только XLA компилятор.
2. Эксперимент включающий как XLA компилятор, так и Grappler.

Результаты приведены в Таблице 2.

Метрика \ Модель	BERT Joint Baseline	BERT Joint XLA	BERT Joint XLA+Grappler
Время предсказания	36 087 с	-11.83 %	-14.83 %
Micro F1-score	0.466	0.466	0.466

Таблица 2 Результаты экспериментов с графовыми оптимизаторами

Из таблицы видно, что использование XLA компилятора позволило уменьшить время предсказания на 10%, а использование пары XLA-Grappler почти на 15%. Подведем итоги.

Преимущества подхода:

- Исключительная простота использования.
- Полное сохранение качества.

Недостатки подхода:

- Незначительное уменьшение времени предсказания (10-15%).

3.3 Смешанная точность значений с плавающей точкой

3.3.1 Описание подхода

Смешанная точность - это использование одновременно 16-битных и 32-битных типов с плавающей точкой для ускорения работы модели и меньшего потребления памяти.

На сегодняшний день большинство моделей в Tensorflow использует тип float32, который занимает, соответственно, 32 бита памяти. Однако, существует два других типа с меньшей точностью: float16 и bfloat16, которые уже занимают 16 бит в памяти.

Отличие между float16 и bfloat16 состоит в диапазоне значений. Тип bfloat16, впервые предложенный компанией Google, расширяет диапазон доступных значений за счет выделения большего числа бит под экспоненту и меньшего числа бит под мантиссу. В данной работе, однако, этот тип рассматриваться не будет, т.к. может быть использован лишь на ограниченном спектре аппаратного обеспечения.

У 16-битных типов есть два важных преимущества:

1. Существует специализированное аппаратное обеспечение, позволяющее выполнять операции с 16-битными значениями быстрее, чем с 32-битными.
2. 16-битные значения считываются из памяти быстрее.

Таким образом можно достичь ускорения работы перейдя с 32-битных типов на 16-битные. Однако, переменные модели и некоторые операции все же должны использовать 32-битные типы для того, чтобы качество работы модели не ухудшалось. В качестве компромисса Keras API позволяет использовать как 16-битные типы, так и 32-битные для того, чтобы улучшить производительность благодаря первым и сохранить качество работы благодаря вторым.

Несмотря, однако, на то, что данный подход применим практически на любом аппаратном обеспечении, желаемый эффект улучшения производительности может быть достигнут только на последних NVIDIA GPU и Cloud TPU. В частности, среди NVIDIA GPU наилучший эффект может быть достигнут у моделей с “compute capability” 7.0 или выше, поскольку в них присутствуют специальные модули, Tensor Cores, для ускорения вычислений с типом float16. Более старые модели не обладают способностью к ускорению вычислений с данными типами, однако некоторые улучшения все же могут быть заметны за счет экономии памяти.

3.3.2 Реализация подхода

Официальное руководство по использованию модуля tensorflow.keras.mixed_precision утверждает, что разработчик может ощутить все преимущества смешанной точности после добавления нескольких строк кода. Однако, на данный момент такая простота в применении не доступна для BERT, по крайней мере в популярных реализациях от Google Research и Hugging Face.

Рассмотрим данную проблему подробнее. Допустим, мы установили политику использования смешанных типов так, как описано в вышеупомянутом руководстве. С этого момента все вычисления в слоях производятся в типе float16, а переменные модели продолжают храниться в float32. Следовательно, выходом любого слоя должен являться тензор с типом float16. Однако, в вышеупомянутых реализациях некоторые слои модели принудительно вычисляются в 32-битных типах и, соответственно, возвращают тензоры с типом float32. Таким образом, внутри модели возникают промежуточные тензоры разных типов: одни - результат заданной политики, другие - результат принудительных вычислений в типе float32. Проблема возникает, когда к тензорам с разными типами применяется одна из встроенных операций (tf.math.add, например), типы входов для которой должны совпадать. В результате мы получаем ошибку похожую на следующее: *“Input 'y' of Op <operation name> has type float32 that does not match type float16 of argument 'x'.”*

Решение в данном случае:

1. Выявление тех мест, где происходит конфликт типов.
2. Ликвидация конфликтов посредством явного приведения тензора с типом float32 к типу float16.

Примеры приведения могут быть найдены в файле modelling.py, строки 389, 967, 970 и др.

Безусловно, данное решение не является абсолютно безопасным в смысле стабильности модели, т.к. в некоторых местах нарушается логика разработчиков. Тем не менее, оно позволяет применить подход со смешанной точностью к BERT.

3.3.3 Результаты

Результаты эксперимента для данного подхода приведены в Таблице 3.

Метрика / Модель	BERT Joint Baseline	BERT Joint Mixed Precision
Время предсказания	36 087 с	-63.1 %
Micro F1-score	0.466	0.466

Таблица 3 Результаты эксперимента со смешанной точностью

Из таблицы видно, что практически без ухудшения качества нам удалось уменьшить время предсказания на 63%. Подведем итоги.

Преимущества подхода:

- Значительное ускорение модели (почти в 3 раза).
- Отсутствие ухудшения качества.

Недостатки подхода:

- Требовательность к аппаратному обеспечению.
- Необходимость правки исходного кода реализации BERT.

3.4 Комбинация подходов

После проведения экспериментов для каждого из подходов было бы логично попробовать их комбинацию. Результаты эксперимента приведены в Таблице 4.

Метрика / Модель	BERT Joint Baseline	BERT Joint Mixed Precision +XLA + Grappler
Время предсказания	36 087 с	-74.24 %
Micro F1-score	0.466	0.466

Таблица 4 Результаты комбинации подходов

Из таблицы видно, что комбинация подходов показала себя хорошо и позволила снизить время предсказания на 74%, что лучше, чем каждый из подходов в отдельности.

3.5 Резюме

Рассмотрим результаты всех подходов в одной сводной таблице с абсолютными значениями.

Метрика / Модель	Baseline	XLA	XLA + Grappler	Mixed Precision	Mixed Precision +XLA +Grappler
Время предсказания	36 087 с	31 816 с (-11.83 %)	30 737 с (-14.83 %)	13 316 с (-63.1 %)	9 296 с (-74.24 %)
Micro F1-score	0.466	0.466	0.466	0.466	0.466

Таблица 5 Итоговые результаты

ЗАКЛЮЧЕНИЕ

В данной работе было рассмотрено два подхода к оптимизации времени предсказания модели BERT на примере задачи Google’s Natural Questions.

Каждый из подходов имеет как преимущества, так и недостатки. Тем не менее, было продемонстрировано, что существует возможность уменьшить время предсказания модели BERT на 70% ценой правки исходного кода и на 10-15% практически без усилий со стороны разработчика.

С другой стороны, эффект от различных способов оптимизации может достаточно сильно зависеть от решаемой задачи и аппаратного обеспечения.

В любом случае, описанные в данной работе методы могут использоваться разработчиком как отправная точка в попытке быстро улучшить производительность обученной модели.

Естественно, что методы оптимизации производительности BERT-подобных архитектур не исчерпываются подходами, описанными в данной работе. Отличным примером подхода, который может быть применен без изменения архитектуры модели, является ONNX Runtime. К тому же достаточно эффективны и методы основанные на изменении архитектуры такие как DistillBERT, ALBERT и другие.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of deep bidirectional transformers for language understanding.”
2. Chris Alberti, Kenton Lee Michael Collins. “A BERT Baseline for the Natural Questions.”
3. https://ai.google.com/research/NaturalQuestions/
4. https://github.com/google-research-datasets/natural-questions/blob/master/nq_eval.py
5. https://www.kaggle.com/c/tensorflow2-question-answering/overview
6. https://github.com/google-research/language/tree/master/language/question_answering/bert_joint
7. https://github.com/google-research/bert
8. Taku Kudo. “Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates.”
9. Rico Sennrich, Barry Haddow, Alexandra Birch “Neural Machine Translation of Rare Words with Subword Units.”
10. https://www.tensorflow.org/guide/keras/mixed_precision
11. https://www.tensorflow.org/guide/graph_optimization
12. https://www.tensorflow.org/xla

ПРИЛОЖЕНИЕ А.

Код программы

Ниже приведены выдержки из кода расположенного в репозитории https://github.com/kisialiou/NQ/

answerNQ.py

```
import tensorflow as tf
import sys
import importlib
import subprocess


import json
import argparse
import os
import re


import mlflow
import time


sys.path.append('/home/bsucm/NQ/')


import src.utils.bert_utils as bert_utils
from src.utils.utils import *
import src.utils.modeling as modeling
import src.utils.tokenization as tokenization
from src.utils.preprocessing import preprocess




MODELS_DIR = '/home/bsucm/NQ/models/'
NQ_EVAL = '/home/bsucm/NQ/src/scripts/nq_eval.py'
```

```
def create_name(name, ext=''):

    if not os.path.exists(name + '.' + ext):
        return name + '.' + ext

    i = 2
    while(os.path.exists(name + str(i) + '.' + ext)):
        i += 1

    return name + str(i) + '.' + ext



parser = argparse.ArgumentParser(description='Answers question after analyzing the text')
parser.add_argument('--run_id', type=str, help='Unique id for current run')
parser.add_argument('--model', type=str, help='Name of the model')
parser.add_argument('--data', type=str, help='Jsonl file to measure quality and
performance')
parser.add_argument('--use_feats', type=str, default='', help='Path to existing features')
parser.add_argument('--no_track', default=False, action='store_true')
parser.add_argument('--outdir', type=str, default='/home/bsucm/NQ/data/outputs/')
args = parser.parse_args()

run_name = args.run_id + '_' + args.model

model_pack = 'models.' + args.model
model = importlib.import_module('.model', model_pack)
model_path = MODELS_DIR + args.model + '/'

gpu = bool(tf.config.experimental.list_physical_devices('GPU'))

if not args.no_track:
    mlflow.set_tracking_uri('/home/bsucm/NQ/mlruns')
    mlflow.start_run(run_name=run_name)

    mlflow.log_param('gpu', gpu)
```

```
    mlflow.log_param('model', args.model)
    mlflow.log_param('data', args.data)


# Loading model
curr_model = model.mk_model()
cpkt = tf.train.Checkpoint(model=curr_model)
cpkt.restore(model_path + 'model_cpkt-1').assert_consumed()


# Data preprocessing
if not args.use_feats:


    feat_records = args.outdir + run_name + '.tfrecords'


    if gpu:
        with tf.device('/GPU:0',):
            preprocess(args.data, feat_records, model_path + 'vocab.txt')
    else:
        preprocess(args.data, feat_records, model_path + 'vocab.txt')
else:
    feat_records = args.use_feats




raw_ds = tf.data.TFRecordDataset(feat_records)
token_map_ds = raw_ds.map(decode_tokens)
decoded_ds = raw_ds.map(decode_record)
ds = decoded_ds.batch(batch_size=32, drop_remainder=False)


start = time.time()
if gpu:
    with tf.device('/GPU:0',):
        result = curr_model.predict_generator(ds,verbose=1)
else:
    result = curr_model.predict_generator(ds,verbose=1)


if not args.no_track:
    mlflow.log_metric('prediction_time', time.time()- start)
```

```
np.savez_compressed(args.outdir + run_name + '-output' + '.npz',

**dict(zip(['uniqe_id','start_logits','end_logits','answer_type_logits'],
                                result)))


# result = []
# with np.load('/home/bsucm/NQ/data/outputs/bert_joint_baseline-output.npz') as data:
#     result.append(data["uniqe_id"])
#     result.append(data["start_logits"])
#     result.append(data["end_logits"])
#     result.append(data["answer_type_logits"])


all_results = [bert_utils.RawResult(*x) for x in zip(*result)]

print ("Going to candidates file")


candidates_dict = read_candidates(args.data)


print ("setting up eval features")


eval_features = list(token_map_ds)


print ("compute_pred_dict")
# For every example dict in appropriate form with answers is returned


nq_pred_dict = compute_pred_dict(candidates_dict,
                                 eval_features,
                                 all_results,
                                 tqdm=None)


predictions_json = {"predictions": list(nq_pred_dict.values())}


print ("writing json")
```

```
prediction_name = args.outdir +  run_name + '_predictions' + '.json'


with tf.io.gfile.GFile(prediction_name, "w") as f:
    json.dump(predictions_json, f, indent=4)
print('done!')
print('Evaluating...')


nq_eval_command = f'python3 {NQ_EVAL} --gold_path {args.data} --predictions_path
{prediction_name}'


eval_results = subprocess.check_output(nq_eval_command, shell=True,
stderr=subprocess.STDOUT)
eval_results = json.loads(re.sub('>=', '-noless-', re.findall('\{.*\}',
eval_results.decode('utf-8'))[0]))


if not args.no_track:
    mlflow.log_metrics(eval_results)
    mlflow.log_artifact(prediction_name)
    mlflow.end_run()


print(eval_results)
```

Bert_joint_baseline: model.py

```
import sys
import tensorflow as tf


sys.path.append('/home/bsucm/NQ/')
import src.utils.modeling as modeling


config = {'attention_probs_dropout_prob':0.1,
'hidden_act':'gelu', # 'gelu',
'hidden_dropout_prob':0.1,
'hidden_size':1024,
'initializer_range':0.02,
'intermediate_size':4096,
'max_position_embeddings':512,
'num_attention_heads':16,
'num_hidden_layers':24,
'type_vocab_size':2,
'vocab_size':30522}


class TDense(tf.keras.layers.Layer):
    def __init__(self,
                 output_size,
                 kernel_initializer=None,
                 bias_initializer="zeros",
                **kwargs):
        super().__init__(**kwargs)
        self.output_size = output_size
        self.kernel_initializer = kernel_initializer
        self.bias_initializer = bias_initializer
    def build(self,input_shape):
        dtype = tf.as_dtype(self.dtype or tf.keras.backend.floatx())
        if not (dtype.is_floating or dtype.is_complex):
          raise TypeError("Unable to build `TDense` layer with "
                          "non-floating point (and non-complex) "
                          "dtype %s" % (dtype,))
        input_shape = tf.TensorShape(input_shape)
        if tf.compat.dimension_value(input_shape[-1]) is None:
          raise ValueError("The last dimension of the inputs to "
                           "`TDense` should be defined. "
                           "Found `None`.")
```

```
        last_dim = tf.compat.dimension_value(input_shape[-1])
        self.input_spec = tf.keras.layers.InputSpec(min_ndim=3, axes={-1: last_dim})
        self.kernel = self.add_weight(
            "kernel",
            shape=[self.output_size,last_dim],
            initializer=self.kernel_initializer,
            dtype=self.dtype,
            trainable=True)
        self.bias = self.add_weight(
            "bias",
            shape=[self.output_size],
            initializer=self.bias_initializer,
            dtype=self.dtype,
            trainable=True)
        super(TDense, self).build(input_shape)
    def call(self,x):
        return tf.matmul(x,self.kernel,transpose_b=True)+self.bias

def mk_model():
    seq_len = config['max_position_embeddings']
    unique_id  = tf.keras.Input(shape=(1,),dtype=tf.int64,name='unique_id')
    input_ids   = tf.keras.Input(shape=(seq_len,),dtype=tf.int32,name='input_ids')
    input_mask  = tf.keras.Input(shape=(seq_len,),dtype=tf.int32,name='input_mask')
    segment_ids = tf.keras.Input(shape=(seq_len,),dtype=tf.int32,name='segment_ids')
    BERT = modeling.BertModel(config=config,name='bert')
    pooled_output, sequence_output = BERT(input_word_ids=input_ids,
                                          input_mask=input_mask,
                                          input_type_ids=segment_ids)

    logits = TDense(2,name='logits')(sequence_output)
    start_logits,end_logits = tf.split(logits,axis=-1,num_or_size_splits= 2,name='split')
    start_logits = tf.squeeze(start_logits,axis=-1,name='start_squeeze')
    end_logits   = tf.squeeze(end_logits,  axis=-1,name='end_squeeze')

    ans_type      = TDense(5,name='ans_type')(pooled_output)
    return tf.keras.Model([input_ for input_ in
[unique_id,input_ids,input_mask,segment_ids]
                           if input_ is not None],
                          [unique_id,start_logits,end_logits,ans_type],
                          name='bert-baseline')
```