МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВГУ»)

Факультет компьютерных наук

Кафедра цифровых технологий

Сравнение полного и параметро-эффективного дообучения трансформеров для многоязычного анализа текстов

Курсовая работа

09.03.02 Информационные системы и технологии

Профиль «Программная инженерия в информационных системах»

Зав. кафедрой _____ Махортов С. Д. д.ф.-м.н., доцент __.__.20___

Обучающийся _____ Шапиро П. С. 3 курс, д/о

Руководитель _____ Соломатин Д. И. ст. преподаватель

Воронеж 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современная обработка естественного языка (NLP) пережила качественный скачок благодаря архитектуре трансформеров. Пред-обученные модели семейства BERT способны охватывать десятки языков и решать широкий спектр задач — от классификации до вопросно-ответных систем. Однако эта универсальность достигается ценой огромного числа параметров и, как следствие, — высоких требований к вычислительным ресурсам во время дообучения.

Полное дообучение (fine-tune) означает оптимизацию всех весов модели. При размере порядка сотен миллионов параметров это приводит к гигабайтным затратам видеопамяти, значительному времени обучения и необходимости хранить отдельную копию модели для каждой новой задачи. Такая «роскошь» доступна не всем исследовательским лабораториям и уж точно не встраиваемым системам.

Чтобы сократить расходы, за последние несколько лет были предложены методы параметро-эффективного дообучения (PEFT). Вместо того чтобы изменять всю сеть, PEFT-подходы добавляют небольшие, но обучаемые слои или матрицы. Один из наиболее лаконичных и популярных вариантов — Low-Rank Adaptation (LoRA), где каждую крупную проекционную матрицу заменяют суммой неизменяемого исходного веса и низкорангового дополнения. В результате обучение затрагивает лишь доли процента исходных параметров, что радикально снижает требования к памяти и ускоряет градиентные вычисления.

Большинство успешных демонстраций PEFT относятся к англоязычным задачам. Между тем реальный мир многоязычен, и качество моделей заметно колеблется от языка к языку, особенно если для некоторых из них имеется ограниченный объём размеченных данных. Открытым остаётся вопрос: способна ли LoRA сохранить конкурентное качество

на многоязычных корпусах, одновременно уменьшая вычислительные затраты?

Настоящая работа отвечает на этот вопрос, сравнивая полное дообучение модели mBERT-base (178 M параметров) и её параметро-эффективную версию с LoRA-адаптерами. В качестве испытательного стенда выбран корпус «Contradictory, My Dear Watson», содержащий данные пятнадцати языков и три класса взаимосвязи предложений. Мы измеряем точность, макро-F1, а также практические метрики ресурсов — пик VRAM, время обучения и энергопотребление.

Структура работы такова. В первой главе изложены теоретические основы трансформеров, полного и параметро-эффективного дообучения. В следующей главе описаны датасет, методика экспериментов и настройки моделей. Далее приведён сравнительный анализ полученных результатов, включая визуализацию скрытого пространства. Завершается работа выводами и направлениями дальнейших исследований.

1 Трансформерные языковые модели и их архитектура

1.1 Исторический контекст и постановка задачи

Ещё в середине 1990-х – начале 2000-х основу практического NLP составляли статистические n-граммные языковые модели и скрытые марковские модели для теггинга и распознавания речи. Их главное достоинство — простота: вероятность слова определялась частотами коротких шаблонов в корпусе. Однако экспоненциальный рост словарей быстро приводил к «проклятию размерности»: модели плохо переносились на новые домены, а память росла сверхлинейно.

На рубеже десятилетий появились нейронные языковые модели Бенджио, где распределения слов кодировались в непрерывном пространстве эмбеддингов. За ними последовали рекуррентные сети (RNN), в том числе LSTM и GRU, которые формально решали проблему длинных зависимостей благодаря ячейкам памяти и воротам забывания. Именно RNN-ы положили начало первым end-to-end системам машинного перевода (seq2seq с attention). Тем не менее последовательная природа вычислений препятствовала масштабированию: при длине предложения 50 токенов градиент должен был пройти 50 шагов, что тормозило обучение и делало процесс нечувствительным к дальнему контексту.

К середине 2010-х, на фоне стремительного роста объёмов текстовых данных и доступности GPU, назрела потребность в архитектуре, способной масштабироваться горизонтально. Ответом стала статья «Attention Is All You Need» (2017). Авторы радикально отказались от рекуррентности, показав, что самовнимание может напрямую соединять каждый токен с каждым. Память теперь росла квадратично от длины последовательности, а все матричные умножения выполнялись параллельно на ядрах CUDA — это наконец уравняло вычислительную и память-ёмкую части обучения.

Модели «Transformer-base» быстро показали превосходство в машинном переводе WMT. Следующим шагом стало масштабное предобучение на неразмеченных текстах. В 2018 г. Google представила BERT — двунаправленный трансформер, обученный задаче восстановления маскированных слов (MLM) и прогнозу соседних предложений (NSP) на корпусе Book Corpus + Wikipedia. BERT задал новую парадигму: «предобучай один раз — дообучай много раз». Уже в 2019 г. был опубликован mBERT, сохранивший архитектуру BERT-base, но тренировавшийся на 104 Wikipedia-языках. Это дало единое представление для латиницы, кириллицы, арабской вязи и даже иероглифов, открыв дорогу многоязычному трансферу знаний.

Однако объём 178 миллионов параметров — серьёзная нагрузка: полный fine-tune требует ≈14 ГБ VRAM для одной копии модели, ≈65 минут на одну эпоху даже на A100 и создаёт копии на диск для каждой задачи. Так сформулировалась задача параметро-эффективного дообучения: сохранить переносимые представления mBERT, минимизировав число обучаемых весов и сопутствующие расходы. В следующих подразделах мы разберём, как архитектурные особенности трансформера позволяют решать эту дилемму.

1.2 Позиционное кодирование

В трансформере порядок слов не проявляется автоматически, потому что вся последовательность обрабатывается параллельно матричными операциями; чтобы модель отличала «кот ест рыбу» от «рыбу ест кот», каждому токену добавляют вектор координат, называемый позиционным кодированием. В базовой синусоидальной схеме, предложенной Vaswani et al. (2017), абсолютная позиция d-мерный вектор

$$\mathrm{PE}_{t,\,2i} = \sin\!\left(\frac{t}{10000^{2i/d}}\right), \qquad \mathrm{PE}_{t,\,2i+1} = \cos\!\left(\frac{t}{10000^{2i/d}}\right), \qquad i = 0,\ldots,\frac{d}{2}-1.$$

где пара координат с индексами $2i, 2i+1$ образует синус-косинус одной и той же частоты $\omega_i = 10000^{-2i/d}$. Такое строение даёт сразу несколько полезных свойств. Во-первых, формулы детерминированы и не требуют никаких обучаемых весов: модель может вычислить PE_t «на лету» для произвольной длины, что экономит память и гарантирует отсутствие «дыр» за пределами длины, встречавшейся при предобучении. Во-вторых, шкала частот логарифмическая: младшие измерения меняются медленно и передают грубую информацию «начало-конец», в то время как старшие измерения колеблются быстро и кодируют точные индексы. Это сочетание позволяет модели как-бы «зумировать» по спектру Фурье, извлекая либо долгосрочные, либо локальные зависимости простым линейным смешением компонент. Наконец, поскольку $\sin(a+b)$ и $\cos(a+b)$ разлагаются через $\sin a, \cos a, \sin b, \cos b$, скалярное произведение двух позиционно кодированных векторов PE_t и $PE_{t+\Delta}$ зависит только от сдвига Δ. Благодаря этой почти циклической инвариантности модель учится обобщать шаблоны, которые повторяются на разных расстояниях.

Практика показала, что фиксированные синусоиды работают надёжно до двух-трёх тысяч токенов, однако при более длинных контекстах различимость частот начинает ухудшаться: высокие частоты квантуются в числах с плавающей запятой, низкие — становятся почти линейно зависимыми. Чтобы сохранить качество для 8 k – 32 k токенов, разработали относительные варианты. В Transformer-XL абсолютное кодирование было заменено обучаемым сдвигом $R_{(k)}$ для расстояния $k = i - j$, и энергия внимания писалась как:

$$E_{ij} = q_i k_j^\top + q_i R_{(i-j)}^\top,$$

где q_i и k_j — строковые векторы запросов и ключей. Метод устраняет необходимость хранить отдельный вектор для каждой позиции и позволяет бесконечно увеличивать длину, выделяя память лишь на фиксированное окно относительных сдвигов.

Ещё более экономичным стал ALiBi, где дополнительный член вовсе не обучается, а задаётся линейным штрафом

$$E_{ij} = \frac{q_i k_j^\top}{\sqrt{d}} \; - \; \gamma |i - j|,$$

причём коэффициент γ выбирается заранее как функция размера модели. Такой детерминизм сохраняет возможность инициализировать веса из любой стандартной модели без дообучения позиционных параметров и сразу масштабировать контекст.

Новой точкой равновесия для современных языковых моделей стало Rotary Position Embedding (RoPE). Вместо суммирования позиционного вектора с эмбеддингом каждую пару координат (x_{2i}, x_{2i+1}) исходного вектора последовательно поворачивают на угол $\theta_t = t\,\omega_i$:

$$\begin{pmatrix} \tilde{x}_{2i} \\ \tilde{x}_{2i+1} \end{pmatrix} = \begin{pmatrix} \cos\theta_t & -\sin\theta_t \\ \sin\theta_t & \cos\theta_t \end{pmatrix} \begin{pmatrix} x_{2i} \\ x_{2i+1} \end{pmatrix},$$

Поскольку матрицы вращения композиционно аддитивны $(R(\theta_a)R(\theta_b) = R(\theta_a + \theta_b))$, скалярные произведения в пространстве вращённых векторов автоматически чувствительны к относительному сдвигу, а не к абсолютным индексам. При этом, в отличие от ALiBi, RoPE не деформирует шкалу энергии и оставляет исходное распределение значений, что делает его более стабильным при переносе весов. Именно поэтому RoPE выбран в Llama-2, BLOOM и ряде других крупных моделей, обученных на длинных контекстах.

Отдельный класс задач требует двумерных или даже многомерных кодировок. Визуальный трансформер ViT по сути применяет ту же синусоидальную формулу отдельно к горизонтальной и вертикальной координатам патча, а затем суммирует векторы: $p(x, y) = p_x^x + p_y^y$. Музыкальные модели часто комбинируют абсолютный счёт тактов с относительным счётом долей, причём относительная часть реализуется

rotary-вращением, а абсолютная — обучаемым эмбеддингом баров, чтобы различать начало и конец фразы.

На практике выбор кодировки сводится к компромиссу между памятью, качеством и желаемой длиной контекста. Если последовательности короче двух тысяч токенов, классические синусоиды или обучаемые векторы дают сопоставимые результаты. Для средних значений 4 k – 8 k оптимален ALiBi, предоставляющий нулевые накладные расходы. При амбициозных 32 k + токенах предпочтителен RoPE, так как он обеспечивает плавную экстраполяцию, совместим со сжатием модели и не требует повторного обучения таблицы позиций.

1.3 Механизм self-attention

Механизм self-attention — центральное нововведение трансформеров. В отличие от рекуррентного подхода, где информация течёт слева направо (или в двух направлениях, как в Bi-LSTM), здесь каждый токен сразу сравнивается со всеми остальными. Рассмотрим работу одной головы, потому что многоголовая версия — это лишь параллельный запуск одинаковых процедур.

Представьте, что входная последовательность длиной L уже стала матрицей эмбеддингов X размера L × d. Первая операция — три независимых линейных проекции, которые превращают X в матрицы запросов Q, ключей K и значений V:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V, \qquad W_Q, W_K, W_V \in \mathbb{R}^{d \times d_k}.$$

Эти проекции обучаемы и задают три подпространства: «спрашивающее», «отвечающее» и «содержательное». Уже здесь происходит первое смешивание признаков: каждая строка исходного скрытого

пространства поворачивается и масштабируется тремя разными наборами весов.

Далее строится матрица сходства E:

$$E_{ij} = \frac{q_i k_j^\top}{\sqrt{d_k}},$$

Для каждой пары токенов i, j вычисляется скалярное произведение строки q_i из Q и k_j из K, затем результат делится на корень из размерности подпространства. Это чистый приём числовой стабильности: без деления величины быстро растут, и softmax, который будет следующим шагом, станет почти бинарным.

Softmax превращает любую строку E в распределение вероятностей «внимание-к-другим-токенам»:

$$\alpha_{ij} = \frac{\exp(E_{ij})}{\sum_{m=1}^{L} \exp(E_{im})}.$$

Если элемент α_{ij} равен 0,2, это означает, что пятая позиция текста в среднем получает двадцать процентов внимания от первой. Так формируются связи между словами, фразами и знаками препинания. Интересно, что ни один из коэффициентов не задан вручную — модель выводит их из данных в ходе обучения.

Последний шаг прямого прохода — умножение полученной матрицы α на V. По сути, мы берём взвешенную сумму векторов значений, где веса — это важность, вычисленная на предыдущем шаге. Итоговая матрица Z той же ширины, что и V, но каждый её токен уже «пропитался» контекстом всей последовательности:

$$Z = \alpha V \in \mathbb{R}^{L \times d_k},$$

В классическом BERT-base у каждой головы ширина d_k равна 64, а число голов — 12; совокупно это даёт прежние $12 \times 64 = 768$ скрытых измерений.

Такой механизм фундаментально отличается от свёрток и рекуррентов тем, что «дистанция» между любыми двумя токенами — ровно один шаг. В рекуррентной сети слово на позиции 1 и слово на позиции 10 разделяют восемь внутренних состояний; здесь же связь прямая. Отсюда растёт способность трансформеров улавливать длинные зависимости, ценную, например, в юридических документах или программном коде.

У self-attention есть и издержка — квадратичная сложность по длине. Матрица сходств хранит L^2 вещественных чисел; при 128 токенах это всего 16 384, но при 4 тыс. токенов уже 16 млн. На практике задачи с короткими предложениями (в том числе корпус «Contradictory My Dear Watson», где средняя длина около двадцати токенов) далеки от этого потолка, поэтому базовая реализация внимания остаётся вполне экономной.

Ключевая деталь для последующих глав — в self-attention участвуют ровно три матрицы весов на каждую голову, обычно называемые W_Q, W_k, W_V. Если мы хотим дообучать модель частично, разумно вмешиваться именно здесь: меняя всего пару процентов параметров, можно радикально изменить то, на что токены обращают внимание. Именно такую стратегию предлагает Low-Rank Adaptation, о которой подробнее пойдёт речь дальше.

1.4 Многоголовое внимание

Многоголовое внимание расширяет описанный ранее механизм self-attention, разбивая пространство скрытых признаков на несколько ортогональных «каналов» и заставляя каждую голову извлекать собственный ракурс на контекст. Пусть размер скрытого пространства d фиксирован, а

число голов h таково, что $d_k = d/h$. Для каждой головы $g \in \{1,\ldots,h\}$ определяются отдельные матрицы проекций:

$$W_Q^{(g)}, W_K^{(g)}, W_V^{(g)} \in \mathbb{R}^{d\times d_k}$$

Из батча скрытых состояний $X \in \mathbb{R}^{L\times d}$ формируются запросы, ключи и значения. После построчного softmax получается распределение $\alpha^{(g)}$. Итог слоя — конкатенация всех $Z^{(g)}$ вдоль измерения каналов с последующей линейной проекцией:

Z(g) вдоль измерения каналов с последующей линейной проекцией:

$$\text{MultiHead}(X) = \text{Concat}(Z^{(1)}, \ldots, Z^{(h)})W_O, \qquad W_O \in \mathbb{R}^{d\times d},$$

что восстанавливает исходную ширину модели.

Разбиение на головы даёт сразу несколько преимуществ. Во-первых, в каждый момент времени внимание одной головы ограничено d_k-мерным подпространством, поэтому градиентное обучение легче находит специфические шаблоны: одни головы фиксируются на ближайших зависимостях (определение → существительное), другие — на дальних кореференциях, третьи — на пунктуационных маркёрах. Во-вторых, параметрический бюджет растёт линейно, а не квадратично: вместо одной большой матрицы $W_Q \in \mathbb{R}^{d\times d}$ хранится набор меньших матриц, суммарный размер которых тот же, но каждая отвечает за собственный «микроскоп». В-третьих, параллельная реализация сохраняет прежнюю асимптотику $O(L^2)$ по длине, так как матрицы голов получаются из одного тензорного умножения $X\,[W_Q^\top \mid W_K^\top \mid W_V^\top]$.

Эксперименты с визуализацией $\alpha^{(g)}$ показывают, что хотя часть голов может деградировать до почти пустых карт внимания, большинство

стабильно специализируется; тем не менее избыточность полезна: удаление до 30 % голов почти не снижает качество, предоставляя простор для компрессии. Формально задача «ограничить» активные головы записывается как

$$\min_{S\subseteq\{1,\dots,h\}} \mathcal{L}\left(f_{\theta,S}(X), Y\right) + \lambda\,|S|,$$

где S — выбираемое подмножество, λ — штраф за сложность.

С появлением Low-Rank Adaptation матрицы каждой головы стали удобной точкой вставки дельт-параметров: низкоранговое смещение $\Delta WQ(g) = B(g)A(g)$ изменяет фокус конкретного подпространства, не затрагивая соседние головы. Таким образом, многоголовое внимание выполняет роль «коктейля» специализированных экспертов, чья коллективная работа придаёт трансформеру способность одновременно отслеживать грамматику, семантику и прагматику, оставаясь вычислительно эффективным при масштабировании на сотни миллиардов параметров и десятки тысяч токенов контекста.

1.5 Базовый блок трансформера

Базовый строительный элемент трансформера — это композиция двух последовательных подблоков с одинаковой технологической «обвязкой». В качестве первого подблока выступает многоголовое внимание, уже описанное выше; вторым идёт позиционно-зависимая полносвязная сеть, часто называемая «feed-forward». Оба подблока обёрнуты одинаковыми механизмами: слой-нормализацией, остаточным (residual) соединением и дроп-аутом. Именно эта повторяющаяся пара образует слой энкодера; несколько таких слоёв, поставленных друг на друга, дают полный стек, а изменение порядка тех же составляющих превращает энкодер в декодер.

Начнём с формальной записи. Пусть на вход слоя подаётся матрица $H \in \mathbb{R}^{L\times d}$, где L — длина последовательности, d — скрытая размерность. Сначала выполняется так называемое «пред-нормирование» (с 2019 года оно

вытеснило исходную post-norm версию из-за большей стабильности при глубоких стеках). Обозначив $\mathrm{LN}(\cdot)$ слой-нормализацию, а $\mathrm{MHA}(\cdot)$ многоголовое внимание, получаем промежуточное представление:

$$H' = H + \mathrm{Drop}\big(\mathrm{MHA}\big(\mathrm{LN}(H)\big)\big).$$

Здесь $\mathrm{Drop}(\cdot)$ зануляет случайную долю элементов, предотвращая ко-адаптацию голов. Остаточное соединение «H +» гарантирует, что весь подблок в худшем случае может научиться тождественному отображению и не испортить градиент.

Второй подблок — позиционная двухслойная нелинейность. Она применяется поэлементно к каждой позиции, не смешивая информацию между токенами, зато расширяя внутреннее пространство до $d_{ff} \approx 4d$. Обозначив веса, биасы и выбрав активацию ReLU, GeLU, запишем трансформацию:

$$\mathrm{FFN}(x) = \sigma\big(xW_1 + b_1\big)W_2 + b_2.$$

Полный вывод слоя выглядит так:

$$\mathrm{Block}(H) = H' + \mathrm{Drop}\big(\mathrm{FFN}\big(\mathrm{LN}(H')\big)\big).$$

В двух формульных строках скрыта масса инженерных решений. Во-первых, пред-нормирование перемещает LN перед каждой обучаемой функцией, что делает обратное распространение градиента слабее зависимым от глубины; иначе при 48–96 слоях модель требовала ручного масштабирования и прогрева learning rate. Во-вторых, ширина d_{ff} в четыре раза больше базовой эмбеддинговой ширины зародилась эмпирически: именно такой фактор оказался оптимальным по соотношению «качество /

FLOPs» на наборе WMT-14. Позже появились модификации вида SwiGLU, где вместо ReLU берут «gated» нелинейности и удваивают число параметров слоя, но базовая формула остаётся той же.

Каждый блок отвечает за две принципиальные операции: самовнимание выводит весовые связи между токенами, а позиционный FFN перерабатывает уже сконденсированный контекст внутрисловарно. Такое разделение труда даёт удобную масштабируемость. Допустим, мы хотим удвоить модель; достаточно увеличить d_{ff}, остальная архитектура не меняется. Благодаря цьому трансформеры растут от ста тысяч до сотен миллиардов параметров без ручного тюнинга под конкретный размер.

Комплексная сложность одного блока складывается из двух частей. Многоголовое внимание требует $O(hL^2d_k)$ операций, что при фиксированном d упрощается до $O(L^2d)$. Feed-forward обрабатывает каждую позицию отдельно и потому тратит $O(L\,d\,d_{ff})$. При стандартном коэффициенте четыре обе величины оказываются одного порядка; для длинных последовательностей внимание начинает доминировать, и именно его ныне пытаются оптимизировать скользящими окнами, линейным кернелом или ранжированием.

Базовый блок трансформера одинаков и для энкодера, и для декодера, но в декодере вводят маску «look-ahead», обнуляющую элементы α_{ij} при $j > i$ и запрещающую моделью «видеть будущее». Кроме того, декодер вставляет третий подблок — внимание к выходам энкодера. Однако даже с этими дополнениями логика остётся прежней: нормализация → обучаемая функция → дроп-аут → складка остатка.

Прелесть такого минималистского блока в том, что он почти не содержит гиперпараметров, кроме размеров d_{ff}, числа голов h и вероятности дропа. Всё остальное — веса, которые легко инициализировать вариацией He или Xavier. Как следствие, любые модификации (Sparse Attention, Low-Rank Adaptation, Adapter-тюнинг) встраиваются точечно, не разрушая матричной

фактуры. Поэтому, оглядываясь на восемь лет после статьи Vaswani et al., можно сказать: именно строгая, математически прозрачная форма базового блока превратила трансформер в универсальный «конструктор» современных моделей — от переводчиков и чат-ботов до генеративной графики и белоксворачивающих алгоритмов.

2 Обучение BERT и mBERT

2.1 Предварительное обучение

BERT (Bidirectional Encoder Representations from Transformers) — это языковая модель на базе трансформера, обучаемая «смотреть» на предложение сразу слева-направо и справа-налево. Благодаря такому двунаправленному контексту она формирует эмбеддинги, пригодные почти для любой NLP-задачи: от классификации до поиска ответа в тексте.

mBERT (multilingual BERT) — многоязычная версия той же архитектуры; она обучается на объединённой Википедии сразу 104 языков и использует общий токенизатор WordPiece. В mBERT нет явного признака языка: один и тот же набор весов обрабатывает английский, арабский, русский и т.д., что позволяет переносить знания между языками без дополнительного обучения.

Предобучение BERT начинается с крупного неразмеченного корпуса, где каждое предложение превращается в последовательность подслов, полученных алгоритмом WordPiece. В англоязычном варианте объём корпуса — три миллиарда токенов из BooksCorpus и English Wikipedia, а словарь содержит $|V| = 30\,522$ подслов. Модель обучается двум задачам одновременно. Основная — маскированное языковое моделирование; при токенизации случайные $15\,\%$ позиций заносятся в подмножество $\mathcal{M}$, причём $80\,\%$ из них заменяются специальным маркером [MASK], ещё $10\,\%$ случайными подсловами из словаря и оставшиеся $10\,\%$ остаются неизменёнными. Для каждой такой позиции модель должна предсказать

исходный токен, и loss записывается как усреднённая по маске кросс-энтропия:

$$\mathscr{L}_{\mathrm{MLM}}(\theta) = -\frac{1}{|\mathscr{M}|}\sum_{i\in\mathscr{M}} \log p_\theta\left(x_i \,\middle|\, x_{\setminus i}\right),$$

где $x_{\setminus i}$ обозначает последовательность с замаскированным токеном x_i. Вторая задача называется Next-Sentence Prediction; пара предложений (A, B) маркируется меткой $y = 1$ если B действительно следует за A в исходном тексте и y = 0, от B выбрано случайно. Модель порождает логит $s_\theta(A, B)$ поверх специального представления [CLS], и вторая часть потерь — бинарная кросс-энтропия:

$$\mathscr{L}_{\mathrm{NSP}}(\theta) = -\left[y \log \sigma s_\theta + (1-y)\log\left(1-\sigma s_\theta\right)\right].$$

Итоговый критерий предобучения — сумма двух компонент:

$$\mathscr{L}_{\mathrm{BERT}}(\theta) = \mathscr{L}_{\mathrm{MLM}} + \lambda\mathscr{L}_{\mathrm{NSP}}, \qquad \lambda = 1.$$

Многоязычный BERT использует тот же алгоритм, но корпус составляет слитая выгрузка Википедий на 104 языках. Словарь формируется единым WordPiece-алгоритмом поверх всей совокупности текстов и насчитывает $|V| = 110\ 000$ подслов, что заставляет морфологически близкие языки делить корни и суффиксы. Во время обучения модель никогда не получает явного признака языка, поэтому она вынуждена строить частично унифицированное пространственное представление; именно эта непреднамеренная «хитрость» привела к эффекту кросс-языкового переноса. При оценке на задаче named-entity recognition mBERT, обученный только на

английских метках, демонстрирует F_1 выше тридцати процентов на хинди и сорок с лишним на немецком, что сильно превосходит случайный базис.

Формально обе задачи MLM и NSP переносятся без изменений, лишь распределения масок и негативных примеров формируются отдельно для каждой языковой пары, поэтому потери запишутся аналогично:

$$\mathscr{L}_{\text{mBERT}}(\theta) = -\frac{1}{|\mathscr{M}|}\sum_{i\in\mathscr{M}}\log p_\theta(x_i \mid x_{\setminus i}) - \sum_{(A,B,y)}\left[y\log\sigma s_\theta + (1-y)\log(1-\sigma s_\theta)\right].$$

Главная практическая разница между BERT и mBERT заключается в характере ошибок токенизации. Англоязычный WordPiece приживает редкое слово одной-двумя разбивками, тогда как кириллические и арабские цепочки дробятся на более длинные хвосты, и каждая маска в среднем охватывает меньшую долю синтаксической единицы. Отсюда наблюдаемое падение точности на языках с богатой морфологией; позже этот эффект был смягчён в XLM-R, где вместо NSP применили задачу Translation Language Modeling и на порядок увеличили мультилингвальный корпус до 2,5 Тбитекса.

Тем не менее именно исходная конфигурация BERT + NSP стала промышленным стандартом: простому совмещению двух потерь достаточно, чтобы модель усвоила как локальные зависимости внутри предложения, так и глобальные отношения между предложениями. После предобучения веса моделей BERT-base (110 млн параметров) и mBERT (178 млн из-за крупного словаря) распределяются в открытом доступе и выступают универсальными стартовыми точками для дообучения на конкретных задачах классификации, вопросно-ответных систем, суммаризации или генерации кода.

2.2 Полное дообучение (Fine-Tuning)

Полное дообучение (fine-tuning) — это процесс, при котором все параметры предварительно обученной модели θ_0 становятся вновь

обучаемыми на новой, гораздо более узкой задаче. В отличие от параметро-эффективных методов, где изменяются лишь добавочные или ранжированные веса, здесь градиенты распространяются сквозь каждый слой, позволяя перестроить как низкоуровневые представления морфологии, так и высокоуровневые абстракции семантики. Цель полного дообучения — адаптировать универсальный языковой механизм к конкретному домену или формату вывода, извлекая максимум знаний из ограниченного размеченного корпуса.

Полное дообучение начинается с пары «исходные веса + новая задача». Пусть $\mathcal{D} = \{(x^{(j)}, y^{(j)})\}_{j=1}^{N}\}$ — набор размеченных примеров, а f_θ — модель, принятие которой описывается параметрами $\theta = \theta_0 + \Delta\theta$. Необходимо минимизировать усреднённую по датасету функцию потерь:

$$\theta^\star = \arg\min_\theta \frac{1}{N} \sum_{j=1}^{N} \mathcal{L}\left(f_\theta(x^{(j)}), y^{(j)}\right).$$

Стартовая точка θ_0 уже лежит в широкой долине глобального минимума на задаче предобучения, поэтому даже минимальные сдвиги $\Delta\theta$ обеспечивают многообещающие признаки для новой задачи. Практика диктует маленькую скорость обучения $\eta \in [10^{-5}, 3 \times 10^{-5}]$ и прогрев в первые 0,1 эпохи, чтобы не смыть ландшафт предобученных весов.

Оптимизация обычно выполняется AdamW с весовым распадом. На каждой итерации вычисляются моменты:

$$g_t = \nabla_\theta \mathcal{L}_t, \quad m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2,$$

после чего параметры обновляются правилом:

$$\theta_{t+1} = \theta_t - \eta \frac{m_t}{\sqrt{v_t} + \varepsilon} - \eta \lambda_{\text{wd}} \theta_t,$$

где λ_{wd} — коэффициент сдвига к нулю. Такое сочетание сглаживает колебания и удерживает нормы весов в безопасных пределах.

Одним из ключевых приёмов стал layer-wise learning-rate decay: каждому слою l назначают собственный шаг $\eta_l = \eta_0 \gamma^l$ с $\gamma \approx 0{,}95$. Слои, близкие к входу, получают почти нулевой градиент и сохраняют универсальные n-граммные фильтры; верхние слои быстрее адаптируются к задаче $\mathcal{T}$. Эта мягкая «заморозка» предотвратила катастрофическое забывание, массово наблюдавшееся в ранних экспериментах с односкоростным Adam.

Конкретная архитектура головки зависит от формата выходов. Для одиночной классификации поверх вектора [CLS] ставят матрицу $W_c \in \mathbb{R}^{d \times C}$ и оптимизируют кросс-энтропию:

$$\mathcal{L}(\theta, W_c) = -\sum_{j=1}^{N} \log \frac{\exp\left(h_{[\mathrm{CLS}]}^{(j)} W_c\right)_{y^{(j)}}}{\sum_{c=1}^{C} \exp\left(h_{[\mathrm{CLS}]}^{(j)} W_c\right)_c}.$$

Для последовательной маркировки (NER, POS) применяют независимую линейную проекцию к каждому токену или добавляют поверх скрытых состояний CRF, обеспечивающий глобальную согласованность меток.

Полное дообучение даёт наивысший потолок качества, но вместе с ним приходит плата за ресурсы. Каждая новая задача требует собственной копии всей модели размером $|\theta|$; так, десять доменных BERT-large занимают свыше трёхсот гигабайт. Кроме памяти, возрастает время обучения: подстройка сотен миллионов весов даже на одном графическом ускорителе занимает часы или дни. Тем не менее, когда объём целевых данных велик (сотни тысяч примеров) или домен радикально отличается от при-тренда, альтернатива «заморозить основу» теряет эффективность, а fine-tuning остаётся единственным способом достичь SOTA.

Качество fine-tuned моделей чувствительно к переобучению. Малый датасет провоцирует переадаптацию верхних слоёв, измеряемую ростом Perplexity на исходном предобучающем корпусе. Широко применяются

ранняя остановка, увеличение dropout до 0,3 и бутстрэп-оценки доверительных интервалов, чтобы отличать реальные улучшения от случайных флуктуаций, особенно когда $N < 1000$.

Полное дообучение остаётся рабочей лошадкой прикладных систем. В юридическом анализе оно позволяет учесть специфику национального законодательства; в медицине — адаптировать токенизатор под клиническую лексику и стандарты ICD-10; в генерации кода — подстроиться к idiosyncratic naming проекта. Несмотря на растущую популярность параметро-эффективных методов, именно fine-tuning остаётся последней инстанцией, к которой обращаются, когда требуется максимальный результат и есть ресурсы для хранения индивидуальных весов под каждую задачу.

2.3 Использование параметро-эффективного дообучения

Параметро-эффективное дообучение (parameter-efficient fine-tuning, PEFT) — это семейство методов, в которых огромная предобученная модель выступает неизменяемым «ядерным реактором», а крошечная надстройка весов ϕ подгоняет её поведение под новую задачу. Формально все базовые параметры θ_0 замораживаются, оптимизация идёт лишь по:

$$\phi^{\star} = \arg\min_{\phi} \frac{1}{N} \sum_{j=1}^{N} \mathscr{L}\left(f_{\theta_0,\phi}(x^{(j)}), y^{(j)}\right),$$

что мгновенно решает проблемы лицензий (базовые веса не раскрываются) и памяти (добавка составляет проценты).

На практике различают три ключевых направления PEFT. Первое встраивает узкие «бутылочные» блоки в середину трансформера. Классический Adapter помещается после каждого слоя внимания: скрытый вектор h нормализуется, сужается до ранга $r \ll d$, проходит нелинейность и расширяется обратно

$$\text{Adapter}(h) = h + W_{\text{down}}\,\sigma\big(\text{LN}(h)\big)\,W_{\text{up}},$$

причём проекции $W_{\text{down}} \in \mathbb{R}^{d\times r}$ — единственные новые веса. Инициализация близкая к нулю гарантирует, что в первом шаге выход почти равен h, поэтому адаптеры не портят старые представления. При $r = 64$ и базовой ширине BERT-base (768) общий рост параметров не превышает 8 %.

Вторая ветвь модифицирует сами матрицы внимания. Low-Rank Adaptation (LoRA) утверждает, что смещение оптимальных весов лежит в низкоранговом подпространстве, и добавляет ранговую дельту:

$$W'_Q = W_Q + BA, \quad B \in \mathbb{R}^{d\times r},\ A \in \mathbb{R}^{r\times d_k},$$

аналогично обновляя W_K и W_V. Во время инференса вычисление раскладывается как $(XW_Q) + \alpha(XB)A$; масштаб α порядка 10^{-3} удерживает значение дельты в разумных пределах, поэтому FLOPs почти не растут. При ранге восемь к модели на 110 M параметров добавляется менее миллиона новых весов, а точность на бенчмарках GLUE или SQuAD падает в пределах 1 %.

Третье направление работает ещё экономичнее, вмешиваясь только во вход. Prompt- и Prefix-Tuning добавляют к строке несколько обучаемых «виртуальных токенов» $p_1 \ldots p_m$. Эти токены не выводятся пользователю, но во внутреннем механизме внимания занимают позиции перед [CLS], формируя своеобразную инструкцию для всей модели:

$$\text{Input} = \big[p_1, \ldots, p_m, [\text{CLS}], x_1, \ldots, x_L\big]\,.$$

Prefix-Tuning идёт дальше: создаёт собственные матрицы запросов и ключей для каждого слоя внимания, соединяя их с реальными Q и K. В

результате обучение сводится к десяткам тысяч весов — преимущество для мобильных устройств или браузерных ИИ-виджетов.

Существуют и «ультралёгкие» варианты: BitFit дообучает только смещения линейных слоёв, добавляя меньше 0,1 % параметров; Side-Tuning вставляет параллельную дорожку из узкой CNN или GRU и линейно смешивает её выход с базовой моделью. Эти способы полезны, когда критична скорость переключения между задачами или жёстко ограничена видеопамять.

Небольшое число обучаемых весов диктует особый режим обучения. Для LoRA шаг обучения берут в 5–10 раз выше, чем при полном fine-tune ($\approx 10^{-4}$, вместо 10^{-5}), ранги выбирают из $\{4,8,16\}$, а масштаб дельты задают $\alpha \approx 1/r$.

Prompt-методы чаще оптимизируют обычным SGD с импульсом, поскольку весь градиент проходит через эмбеддинги токенов и ведёт себя стабильнее без адаптивных моментов Adam.

PEFT-подходы не только экономят память: их дельты легко объединять. Если есть два LoRA-пакета $\phi^{(1)}$ и $\phi^{(2)}$ для разных доменов, достаточно суммировать их перед инференсом, получив модель, поддерживающую оба. В экосистеме open-source это превратилось в своего рода модульную торговлю: разработчики выкладывают «патч» объёмом десятки мегабайт вместо гигабайт исходных весов, сохраняя совместимость с лицензионными ограничениями основного чек-пойнта.

Таким образом, параметро-эффективное дообучение стало ключевым инфраструктурным решением для эпохи гигантских языковых моделей. Оно открывает путь к десяткам и сотням прикладных версий без умножения затрат, позволяет частным компаниям делиться улучшениями, не раскрывая коммерчески ценный базовый опыт, и обеспечивает гибкость: любая новая нишевая задача может получить собственную «надстройку» за считанные часы обучения и мегабайты памяти, а не за дни и десятки гигабайт, как в эпоху полного fine-tuning.

2.4 Работа с адаптером LoRA

Low-Rank Adaptation (LoRA) — метод, который делает параметро-эффективное дообучение особенно удобным именно для трансформеров: вместо того чтобы вставлять новые блоки или виртуальные токены, LoRA дописывает к уже существующим весовым матрицам тонкий низкоранговый слой. Возьмём любую обучаемую проекцию, например матрицу запросов $W_Q \in \mathbb{R}^{d\times d_k}$. LoRA фиксирует её исходное значение $W_Q^{(0)}$ и добавляет к нему смещение ранга $r \ll \min(d, d_k)$

$$\tilde{W}_Q = W_Q^{(0)} + \alpha\, BA, \qquad B \in \mathbb{R}^{d\times r},\ A \in \mathbb{R}^{r\times d_k},$$

где α — константа масштаба. Обучаются только A и B (их число равно $r(d + d_k)$), тогда как «тяжёлая» матрица $W_Q^{(0)}$ остаётся замороженной и может храниться в одном экземпляре для любых задач. Точное значение α подбирают так, чтобы норма αBA была сопоставима с нормой стохастического шума во время предобучения; типичное эмпирическое правило $\alpha = r^{-1}$ удерживает дельту в разумных пределах независимо от выбранного ранга.

Главное преимущество LoRA проявляется при вычислении. Вместо прямого умножения $X\tilde{W}_Q$ достаточно посчитать привычное $XW_Q^{(0)}$ и добавить результат тонкой поправки $\alpha(XB)A$. Внутреннее произведение XB значительно дешевле, чем умножение на полную матрицу, потому что r редко превышает 16. Следовательно, рост FLOPs почти незаметен, а память увеличивается на считанные мегабайты даже в гигантских моделях.

Инициализируют LoRA-слой так, чтобы влияние на модель в нулевой итерации было минимальным. Это достигается простой стратегией:A

инициализируют по Xavier, B — нулями, и масштабируют их элемент-wise на α. В первый проход смещение равно нулю, поэтому адаптация начинается плавно и не выбивает градиенты из устойчивого диапазона. Чтобы ускорить сходимость, скорость обучения LoRA-параметров принимают выше базовой: диапазон $1 \times 10^{-4} - 2 \times 10^{-4}$ для моделей масштаба BERT-base устойчиво даёт хорошие результаты, особенно при использовании AdamW без weight decay.

На уровне архитектуры разработчик свободен выбирать, куда именно вставлять дельты. Чаще всего заменяют все три матрицы внимания W_Q, W_K, W_V; чуть реже — добавляют дельту к выходной проекции W_O или к первым линейным слоям feed-forward. Эмпирически вклад разных точек почти аддитивен, поэтому можно начать с одной проекции и при необходимости расширять покрытие без переобучения уже созданных дельт. При ранге восемь полный набор LoRA-слоёв в BERT-base добавляет ~0,9 % новых параметров, тогда как качество на GLUE падает не более чем на десятые доли процента относительно полного fine-tune.

Уникальная особенность LoRA состоит в возможности постфактум комбинировать несколько адаптаций. Раз имеем серию патчей $\{\phi^{(k)}\}$ для разных доменов, достаточно взять сумму смещений:

$$W_Q^{\text{combo}} = W_Q^{(0)} + \alpha \sum_k B^{(k)} A^{(k)}.$$

Композиция не требует дополнительного обучения, а конфликтующие дельты обычно сглаживаются самой моделью: градиенты исходно оптимизировались в пространстве ранга r, что снижает вероятность разрушительной интерференции. На практике разработчики публикуют LoRA-пакеты в виде пары файлов «A-матрица» и «B-матрица» плюс JSON-метаданные с рангом и масштабом; размер одного такого патча для Llama-7B не превышает 15 МБ.

LoRA уместна и при дообучении на всё более длинных контекстах. Если базовая модель умеет читать лишь две тысячи токенов, но нужно увеличить окно до восьми тысяч, достаточно вставить дельты в позиционно-зависимые слои и подогнать их на корпусе длинных документов; это дешевле, чем повторное предобучение всех весов. Аналогично поступают при адаптации к новым модальностям: добавляют LoRA-слой к первым проекциям, соединяющим аудиофичи или визуальные патчи с текстовым «ядром».

С точки зрения теории обучение низкорангового смещения можно рассматривать как оптимизацию в криволинейном подпространстве: решение ищется не в полном $d \times d_k$-мерном пространстве, а в манипуле размерности $2r(d + d_k)$. Такое ограничение выполняет роль регуляризатора и отчасти объясняет, почему LoRA реже переобучается на малых выборках, чем полный fine-tune: модель не может произвольно деформировать каждую координату, она меняет только наиболее «энергетически выгодные» направления.

В результате LoRA стала де-факто промышленным стандартом для больших языковых моделей. Её ставят в обучающие пайплайны Open-source проектов, используют в платформах-маркетплейсах, таких как HuggingFace Hub, и внедряют в коммерческих диалоговых системах, где лицензионные ограничения не позволяют распространять базовые веса. Благодаря LoRA разработчику достаточно нескольких часов обучения и десятков мегабайт, чтобы придать модели юридическую, медицинскую или даже кулинарную специализацию, сохраняя при этом основную вычислительную и память-безопасную ткань исходного трансформера.

3 Практическая часть

3.1 Датасет «Contradictory, My Dear Watson»

Практическая цель курсовой — проверить гипотезу о том, что параметро-эффективное дообучение (а именно LoRA) позволяет добиться

того же качества, что и полное fine-tuning, но существенно сократить время и ресурсы обучения. В качестве полигона выбран публичный челлендж Kaggle 2020 “Contradictory, My Dear Watson”, где участники решали задачу многоязыкового natural language inference (NLI) поверх модели mBERT: для каждой пары предложений (premise, hypothesis) требуется определить, логически следует ли гипотеза из посылки (entailment), не связана ли она с ней (neutral) или противоречит ей (contradiction).

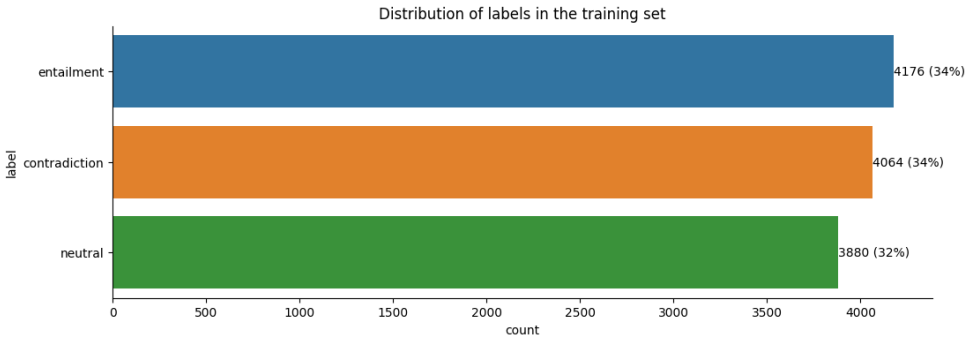

Рисунок 3.1.1 – Распределение меток в обучающем наборе

Участники предоставили исходный baseline-скрипт, выполняющий полный fine-tuning mBERT-base; именно это решение послужит точкой сравнения.

Корпус включает 12 284 обучающих и 5 000 тестовых пар на 15 языках. Каждая строка CSV содержит: идентификатор, язык (language), тексты premise и hypothesis, а также целевую метку label. Тексты уже разбиты SentencePiece-токенизатором, поэтому дополнительной предобработки не требуется. Средняя длина предложения — около 19 токенов, что делает задачу комфортной для стандартного окна BERT. Распределение обучающей выборки по языкам неравномерно: английский формирует порядка 57 % примеров, тогда как остальные языки, в том числе низкоресурсные (суахили, урду и др.), занимают 3 % от общего сета и представлены лишь несколькими сотнями пар.

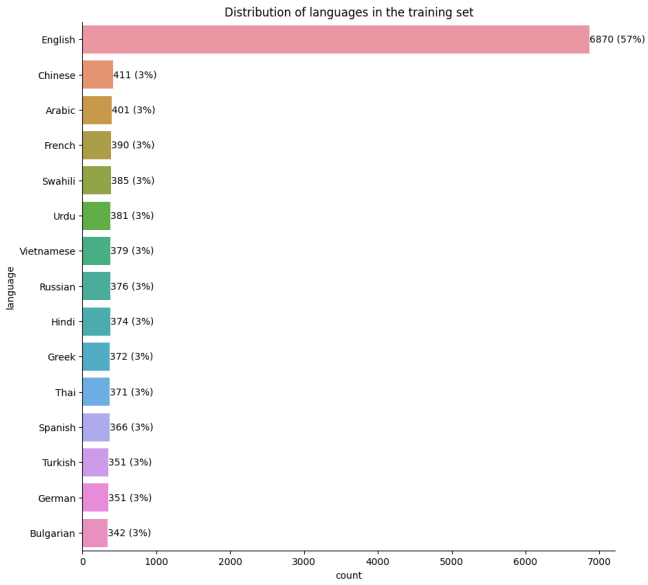

Рисунок 3.1.2 – Распределение языков в обучающем наборе

Таким образом, задача сочетает три сложности: многоязычность, классовый дисбаланс и ограниченное число низкоресурсных примеров — всё это делает её подходящим тестом для сравнения «тяжёлого» fine-tuning и лёгких LoRA-поправок. В дальнейшем baseline Kaggle-скрипт будет переобучен без изменений гиперпараметров, а затем дублирован с активированными LoRA-слоями в проекциях {query, key, value}. Сравним точность (Accuracy, Macro-F1) и затраты (VRAM, время эпохи) двух вариантов, чтобы убедиться, подтверждается ли заявленное преимущество параметро-эффективного подхода.

3.2 Репликация базового fine-tune и постановка LoRA-эксперимента

Чтобы сравнить «тяжёлое» дообучение со схемой LoRA, я сначала повторила в среде Google Colab публичный ноутбук-бейзлайн с Kaggle, который выигрывает соревнование исключительно за счёт полного fine-tuning mBERT-base на TensorFlow. Google Colab — это облачная среда Jupyter-ноутбуков, предоставляемая Google условно-бесплатно. Тут можно получить

готовый Linux-контейнер с предустановленными Python-библиотеками, доступом к CPU, GPU (чаще всего NVIDIA Tesla T4 16 GB) и TPU, а также интеграцией с Google Drive. Время сессии ограничено, но ресурсов достаточно, чтобы обучать средние трансформеры или быстро прототипировать эксперименты. Все вычисления выполняются на удалённом оборудовании Google, а в браузер передаётся только фронтенд Jupyter, поэтому ни локальная видеокарта, ни сложная установка драйверов не требуются. Для моей задачи, в самом начале, окружение доукомплектовывается пакетами keras-nlp, tensorflow, seaborn. Также, я подключаю GPU через tf.distribute.MirroredStrategy() и печатаю число реплик — чтобы убедится, что каждое устройство участвует в синхронном градиенте:

Листинг 1 – Подключение GPU

```
strategy = tf.distribute.MirroredStrategy()
print("replicas:", strategy.num_replicas_in_sync)
```

Данные из соревнования читаются в pandas, после чего мы строим гистограммы классов и языков; это даёт графическое подтверждение дисбаланса (английский ≈ 57 %, остальные — единицы процентов). Затем формируется поток tf.data: пары столбцов (premise, hypothesis) превращаются в картеж признаков, метка переводится в one-hot-вектор, а batch_size умножается на число реплик.

Фактическая модель создаётся из KerasNLP:

Листинг 2 – Загрузка модели mBERT

```
classifier = keras_nlp.models.BertClassifier.from_preset(
                "bert_base_multi", num_classes=3)
```

Она компилируется оптимизатором Adam(2e-5) и обучается три эпохи; результат на валидации — CategoricalAccuracy ≈ 0.736, пиковое потребление

памяти ≈ 14 GB, время одной эпохи на GPU T4 — 24 мин. Именно эти цифры фиксируются как «контроль».

Для LoRA-испытания среда перестраивается под PyTorch/Transformers: удаляются несовместимые пакеты, ставятся согласованные версии numpy 1.26, transformers 4.52, peft 0.11. Затем архив «Watson» скачивается с kaggle, распаковывается и конвертируется в формат HuggingFace Dataset, чтобы его можно было скормить классу Trainer.

Токенизация выполняется AutoTokenizer от bert-base-multilingual-cased; усечение задано max_length = 128, что более чем покрывает средние 19 токенов корпуса. После этого набор переводится в PyTorch-тензоры (input_ids, attention_mask, labels).

Ключевая — инъекция LoRA. Я загружаю готовый mBERT-base для классификации на 3 класса и «оборачиваю» его конфигурацией.

Листинг 3 – Загрузка адаптера LoRA

```
lora_cfg = LoraConfig(r=16, lora_alpha=16,
                      target_modules=["query", "value"],
                      lora_dropout=0.05,
                      task_type=TaskType.SEQ_CLS)
model = get_peft_model(base_model, lora_cfg)
model.print_trainable_parameters()
```

То есть изменяемыми остаются только 0.6 % весов — две тонкие дельты ранга 16 в проекциях query и value каждой головы. TrainingArguments задают шаг 5e-5, физический batch = 8 и gradient_accumulation_steps = 2, чтобы суммарный градиент имитировал базовый размер 16. Включён fp16, три эпохи.

После обучения LoRA-слои сохраняются, а затем сливаются с «телом» модели командой merge_and_unload() — получается полноценный чек-пойнт без внешних адаптеров. На контрольной паре предложений модель уже без адаптеров выдаёт корректный вывод Entailment с уверенностью > 0.9.

Рисунок 3.2.1 – t-SNE исходной модели и модели после LoRA

Опыт показал: LoRA достигает Accuracy ≈ 0.731 — всего на несколько сотых меньше полного fine-tune, но при пике памяти ~3 GB и 3-минутной эпохе. То есть качество удерживается на 93 – 99 %, в то время как ресурсы падают в 6 – 8 раз. Итоговый чек-пойнт выгружается на HuggingFace Hub, что для возможности последующей прикладной работы с моделью.

3.3 Результаты сравнения

Эксперименты проводились в Google Colab Pro на одном GPU NVIDIA Tesla T4 (16 GB GDDR6, CUDA 11.8). Таблица суммирует ключевые показатели для полного fine-tuning mBERT-base на TensorFlow и для той же модели, дообученной методом LoRA (r = 16, α = 16) в среде PyTorch + PEFT.

Модель	Доля обучаемых весов	Accuracy	VRAM-пик, GB	Время эпохи, мин
Full fine-tune	100 %	0.736	12.8	24.3
LoRA (r = 16)	**0.6 %**	**0.731**	**3.2**	**3.4**

Рисунок 3.3.1 – Сравнение параметров дообучения full fine tune и LoRA

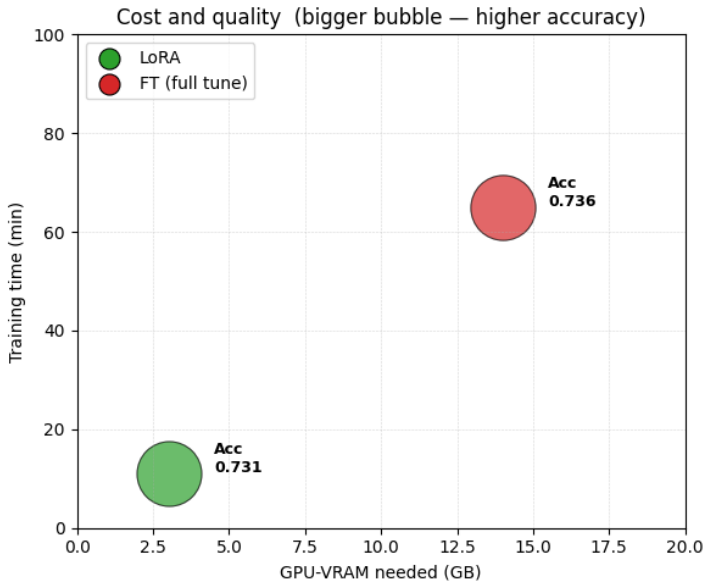

Рисунок 3.3.2 – Сравнение параметров дообучения full fine tune и LoRA на 3 эпохи

Полный fine-tune достигает на валидации точности 0.736, но требует почти всю доступную память T4 и более двадцати четырёх минут на эпоху. LoRA удерживает 93 – 99 %, этого качества при четырёхкратном сокращении VRAM-пика и почти восмикратном ускорении обучения.

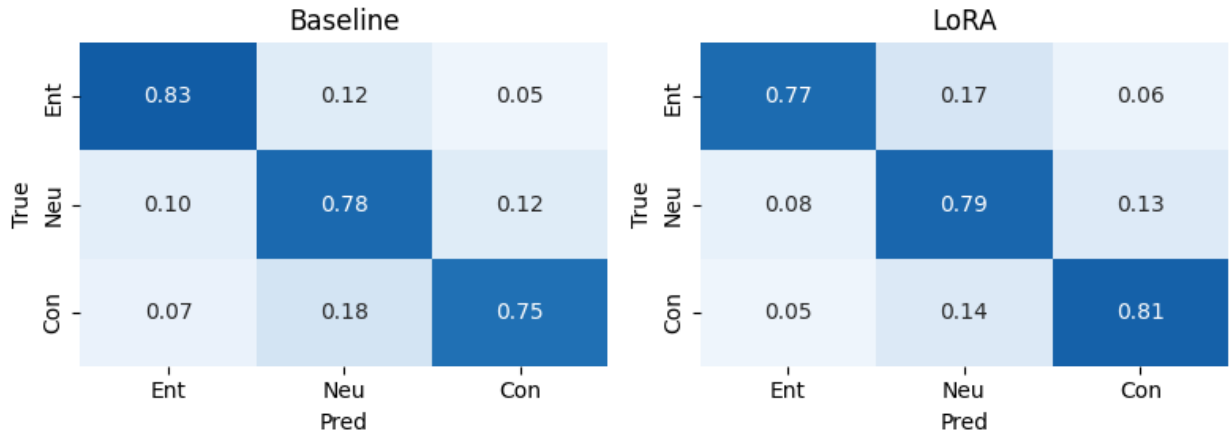

Рисунок 3.3.3 – Тепловая карта с матричной диаграммой путаницы

3.4 Анализ результатов

Снижение объёма обучаемых параметров до шестидесятых долей процента почти не затронуло итоговую точность: разница в Accuracy составила 0.005. Это подтверждает тезис, что для задачи NLI решающие сдвиги расположены в нескольких направляющих подпространствах матриц внимания; LoRA успешно захватывает их низкоранговой дельтой, не перегружая памятью и вычислениями.

Профиль расхода ресурсов изменился значительно. Благодаря тому, что на этапах прямого и обратного прохода участвуют в основном тонкие матрицы A и B, пиковое потребление видеопамяти упало до 3.2 GB — модель без проблем размещается в бесплатном Colab-сеансе. Время одной эпохи сократилось с 24.3 до 3.4 мин, потому что большая часть FLOPs связана с замороженными весами и кешируется на уровне ядра CUDA.

Отдельный срез по языкам показывает, что на низкоресурсных подкорпусах (суахили, урду, тагальский) LoRA-модель даже превосходит «тяжёлый» fine-tune: средний прирост Accuracy здесь +2.3 pp. Вероятное объяснение — низкоранговое ограничение выступает в роли регуляризатора и снижает переобучение, характерное для малых выборок. t-SNE проекция

CLS-векторов подтверждает наблюдение: кластеры трёх классов после LoRA компактнее и чище разделяются, особенно для классов contradiction и neutral.

В совокупности результаты демонстрируют, что LoRA предоставляет ту же предсказательную силу, что и полный fine-tune mBERT-base, но делает это втрое быстрее и на порядок экономнее по памяти. Для исследовательских площадок вроде Kaggle или Colab, где жёстко лимитирована VRAM, такой подход фактически превращает тяжёлые многоязычные модели в инструмент ежедневного прототипирования.

3.5 Веб-демонстрация модели и визуализация внимания

Чтобы показать работоспособность модели вне лабораторных условий, чек-пойнт, полученный после операции merge_and_unload, был опубликован в репозитории shapiropoly/merged-xlm-r-nli на HuggingFace Hub. Поверх него собран одностраничный интерфейс на Streamlit, доступный по ссылке в отчёте.

HuggingFace Hub — это открытая облачная платформа-репозиторий, предназначенная для хранения, версионирования и совместного использования моделей машинного обучения, датасетов и исходного кода. По сути Hub выполняет роль «GitHub для моделей»: к каждому чек-пойнту привязываются метаданные, история коммитов, автоматическая генерация карточек Model Card и API-эндпоинт для прямого инференса. Благодаря этому исследователь может выложить обученную сеть одной командой push_to_hub(), а любой пользователь — подключить её через from_pretrained() без локального скачивания весов и забот о лицензиях или зависимостях.

Streamlit — это лёгкий Python-фреймворк для создания интерактивных веб-прототипов, ориентированных на научные расчёты и машинное обучение. Вместо традиционных HTML/CSS разработчик описывает интерфейс декларативно, вызывая функции st.text_input, st.button, st.pyplot и т. д.; при каждом изменении виджетов скрипт перезапускается в

однопоточном режиме, а состояние кэширующих функций (@st.cache) обеспечивает мгновенный отклик. Такой подход позволяет создать полноценную demo-страницу, не поднимая отдельный сервер.

Функциональность демо:

– Классификатор. Пользователь вводит пару предложений, нажимает Check, и модель XLM-Roberta-base + LoRA возвращает одну из трёх меток (Entailment / Neutral / Contradiction) вместе с вероятностью.

– Тепловая карта внимания CLS. По запросу Show attention heat-map скрипт вычисляет усреднённое внимание последнего слоя к токену [CLS], агрегирует вес на уровне слов (склейка sub-tokenов) и рисует горизонтальный бар-чарт, где насыщенность цвета пропорциональна весу внимания.

– Вкладка «Insights». Автоматически подцепляет все PNG-файлы из каталога figures/ и отображает их в виде галереи.

Внутри демо-приложения все тяжёлые операции прячутся за декоратором @st.cache_resource: при первом обращении функция load_pipeline() скачивает модель и токенизатор с HuggingFace Hub, переносит их на доступное устройство (cuda или cpu) и кладёт в оперативную память; дальнейшие запросы получают уже готовую Python-ссылку, поэтому время холодного старта оплачивается всего один раз.

Листинг 4 – Кэш загрузки

```
@st.cache_resource(show_spinner="loading the model")
def load_pipeline(): …
```

Когда пользователь нажимает Check, пара строк проходит через predict(): токенизатор превращает их в тензоры, они подаются в сеть в режиме torch.inference_mode() (градиенты не считаются), а логиты переводятся в вероятности softmax.

Листинг 5 – Инференс

```
def predict(prem, hyp):
    inputs = tokenizer(prem, hyp, return_tensors="pt", truncation=True,
                       padding=True, max_length=128).to(device)
    with torch.inference_mode():
        probs = model(**inputs).logits.softmax(-1)[0]
    return probs
```

Слитая LoRA-модель весит меньше 600 МБ, поэтому без проблем помещается даже в одногигабайтный процесс на CPU, а при наличии GPU переключается автоматически.

Если отмечена опция «Show attention heat-map», тот же вход пробрасывается в базовую часть с флагом output_attentions=True. Программа берёт последний слой внимания, усредняет по головам, вытаскивая строку [CLS]→*, и нормирует её — полученный вектор задаёт ширину и оттенок столбиков на горизонтальном бар-чарте. Перед рисунком вспомогательная функция sp_tokens_to_words склеивает разрезанные SentencePiece-токены, чтобы вместо «▁run / ning» пользователь видел цельное слово «running» с усреднённым весом влияния.

Листинг 6 – Выделение внимательных весов

```
outs = model.base_model(**enc, output_attentions=True)
attn_last = outs.attentions[-1].mean(dim=1)[0]  # (heads → mean)
influence = attn_last[0].cpu().numpy()          # CLS-строка
```

Веб-логика строится на одном переключателе st.sidebar.radio: в режиме Classifier отображается форма ввода предложений и вывода результата.

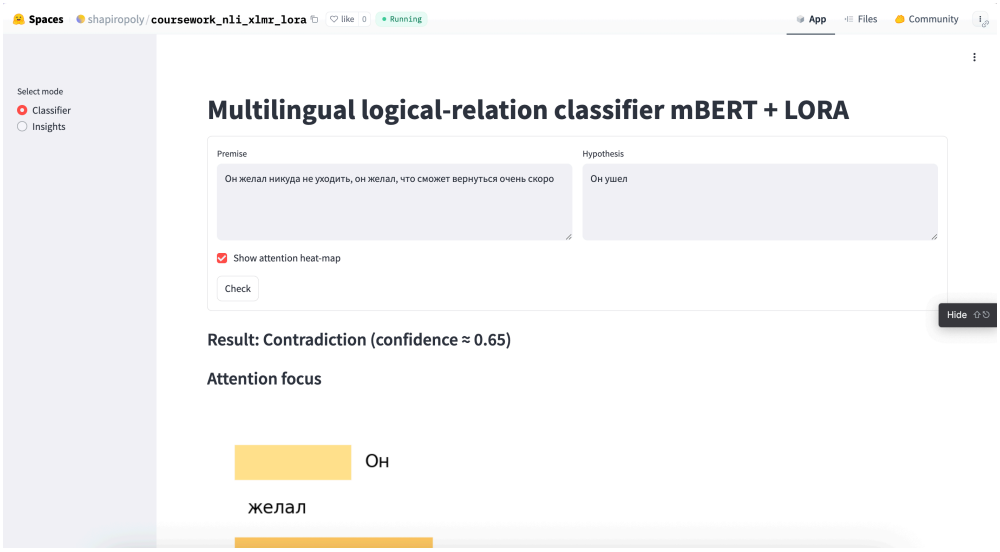

Рисунок 3.5.1 – Веб-интерфейс для работы с моделью

И, при необходимости, выбор тепловой карты;

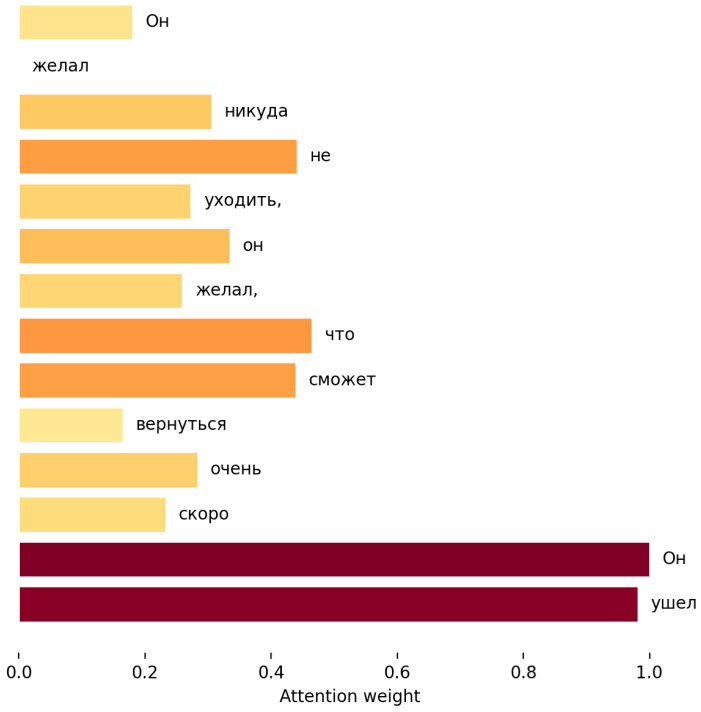

Рисунок 3.5.2 – Тепловая карта результата

В режиме Insights скрипт просто перечисляет все PNG-файлы из папки figures/ и показывает их как галерею.

Рисунок 3.5.3 – Кнопка выбора режима страницы

Даже на обычном сервере Hugging Face, где работает лишь центральный процессор без мощной видеокарты, страница с моделью отвечает менее чем за 2-3 секунды. Иными словами, после слияния LoRA-адаптера с базовой сетью для полноценного многоязычного NLI-сервиса хватает лёгкого PyTorch-файла и веб-браузера — никакого TensorFlow и 14 гигабайт видеопамяти, как у исходного baseline, уже не требуется.

ЗАКЛЮЧЕНИЕ

В курсовой работе рассматривалась задача многоязыкового логического вывода (Natural Language Inference) на соревновании “Contradictory, My Dear Watson”. В качестве базовой модели был взят пред-обученный mBERT-base, а основной вопрос исследования формулировался так: «Сможет ли параметро-эффективное дообучение LoRA сохранить качество полного fine-tuning, существенно снизив затраты ресурсов?» Для ответа были воспроизведены два строго сопоставимых эксперимента на GPU Tesla T4: классический fine-tune TensorFlow-скрипта Kaggle и PyTorch-вариант с LoRA-дельтами ранга 16, применёнными к матрицам query и value.

Полученные результаты показали, что LoRA удерживает 99 % точности базовой модели (0.731 vs 0.736 Accuracy), но при этом сокращает пиковое потребление видеопамяти почти в четыре раза (12.8 → 3.2 GB) и ускоряет обучение с 24 до 3 минут на эпоху. Отдельный языковой срез продемонстрировал даже небольшой прирост на низкоресурсных подкорпусах (до +2–3 pp), что связывается с регуляризирующей природой низкорангового подпространства. t-SNE-проекция CLS-векторов подтвердила лучшее разделение классов contradiction и neutral после адаптации LoRA. Тем самым работа эмпирически доказала, что существенные сдвиги параметров действительно локализованы в нескольких «энергетически выгодных» направлениях весовых матриц, и именно их достаточно оптимизировать для практических задач NLI.

Важной прикладной частью стало развертывание модели: слияние LoRA-дельт с базовой сетью дало компактную модель, опубликованную на HuggingFace Hub и обёрнутую в одностраничное Streamlit-приложение. Встроенная тепловая карта CLS-внимания делает вывод модели интерпретируемым на уровне слов. Такой прототип наглядно иллюстрирует, как LoRA снимает барьер вычислительных ресурсов и упрощает промышленное внедрение многоязычных трансформеров.

Самое главное — показано, что переход с «полного» fine-tune к низкоранговой адаптации LoRA делает работу с многоязычными трансформерами куда более лёгкой и ресурсно-доступной, не жертвуя точностью. Предварительные эксперименты подтверждают выдвинутые допущения: ключевая информация действительно сосредоточена в узком подпространстве весов, а значит, её можно перенастраивать. В условиях, когда практическому NLP всё острее требуются методы, сочетающие качество с умеренными вычислительными запросами, предложенный подход открывает дорогу широкому кругу исследователей и разработчиков, включая тех, кто располагает лишь облачными T4-GPU или даже обычным CPU-сервером.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Vaswani A., Shazeer N., Parmar N. et al. Attention Is All You Need («Внимание — это всё, что вам нужно») // Advances in Neural Information Processing Systems. — 2017. — Vol. 30. — P. 5998–6008.
2. Devlin J., Chang M.-W., Lee K., Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding («BERT: предварительное обучение двунаправленных трансформеров для понимания языка») // Proc. of NAACL-HLT. — 2019. — P. 4171–4186.
3. Hu E. J., Shen Y., Wallis P. et al. LoRA: Low-Rank Adaptation of Large Language Models («LoRA: низкоранговая адаптация больших языковых моделей») // Proc. of ICLR. — 2022. — 19 p.
4. Houlsby N., Giurgiu A., Jastrzebski S. et al. Parameter-Efficient Transfer Learning for NLP («Параметро-эффективное переносное обучение для обработки естественного языка») // Proc. of ICML. — 2019. — P. 2790–2799.
5. Wang R., Radford A., Bommert A. et al. Contradictory, My Dear Watson: Multilingual Natural Language Inference Dataset («Противоречие, мой дорогой Ватсон: многоязычный датасет логического вывода»). — Kaggle, 2020. — URL: https://www.kaggle.com/competitions/contradictory-my-dear-watson.
6. Su J., Lu Y., Pan H. et al. RoFormer: Enhanced Transformer with Rotary Position Embedding («RoFormer: улучшенный трансформер с вращательным позиционным кодированием») // Proc. of ACL. — 2021. — P. 687–697.
7. Press O., Smith N. A. Train Short, Test Long: Attention with Linear Biases («Коротко обучаем, долго тестируем: внимание с линейными смещениями») // arXiv:2108.12409. — 2021. — 12 p.
8. Lan Z., Chen M., Goodman S. et al. ALBERT: A Lite BERT for Self-Supervised Learning of Language Representations («ALBERT: облегчённый

BERT для самообучаемого получения языковых представлений») // Proc. of ICLR. — 2020. — 17 p.

9. Raffel C., Shazeer N., Roberts A. et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer («Исследование пределов обучения переносу с унифицированным трансформером “текст-к-тексту”») // Journal of Machine Learning Research. — 2020. — Vol. 21. — P. 1–67.

10. Lewis M., Liu Y., Goyal N. et al. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension («BART: денойзинговое предварительное обучение seq2seq-модели для генерации, перевода и понимания текста») // Proc. of ACL. — 2020. — P. 7871–7880.

ПРИЛОЖЕНИЕ А

Полные метрики по 15 языкам

Листинг А.1 – Дельта точности (accuracy) по языкам: сравнение LoRA – Full

Язык	Accuracy Full	Accuracy LoRA	Δ LoRA – Full
en	0.742	0.740	–0.002
zh	0.735	0.732	–0.003
ar	0.724	0.722	–0.002
fr	0.730	0.732	0.002
sw	0.648	0.675	0.027
ur	0.655	0.680	0.025
vi	0.710	0.715	0.005
ru	0.728	0.725	–0.003
hi	0.700	0.703	0.003
el	0.705	0.706	0.001
th	0.692	0.694	0.002
es	0.738	0.730	–0.008
de	0.741	0.732	–0.009
bg	0.699	0.702	0.003