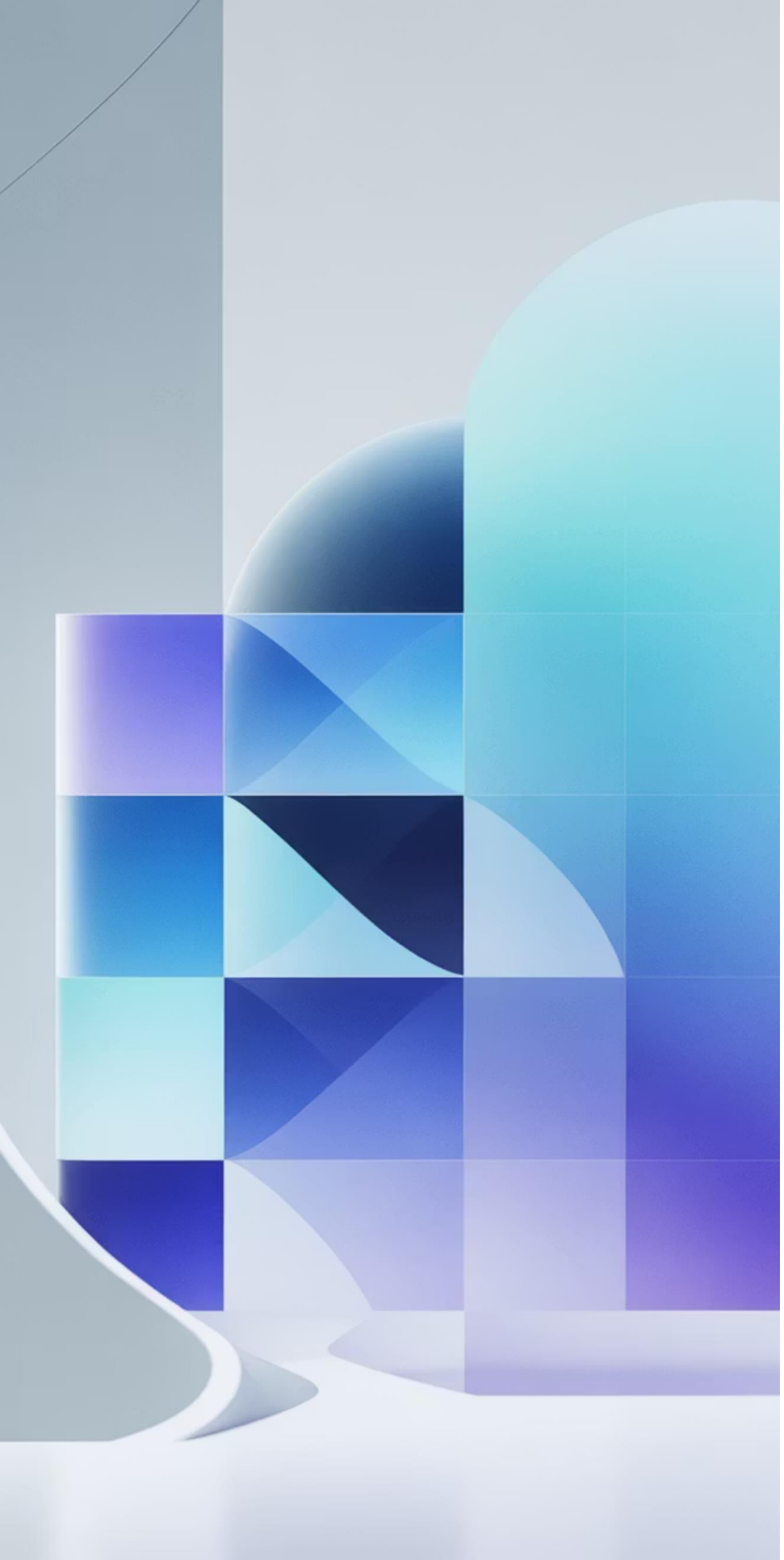


Эффективный fine-tuning больших моделей: LoRA и QLoRA

Светлана Каримова

Содержание презентации

Проблема:
Тонкая настройка (fine-tune) требует огромных ресурсов

Полная тонкая настройка

Обновление всех параметров модели требует огромной памяти GPU. Для 16-битной тонкой настройки на каждый параметр приходится:

- Веса: 16 бит (2 байта)
- Градиенты весов: 16 бит (2 байта)
- Состояния оптимизатора: 65 бит (8 байт)
- **Итого: 96 бит (12 байт) на параметр**

Требования к памяти

Модель с 65 миллиардами параметров требует:

780GB

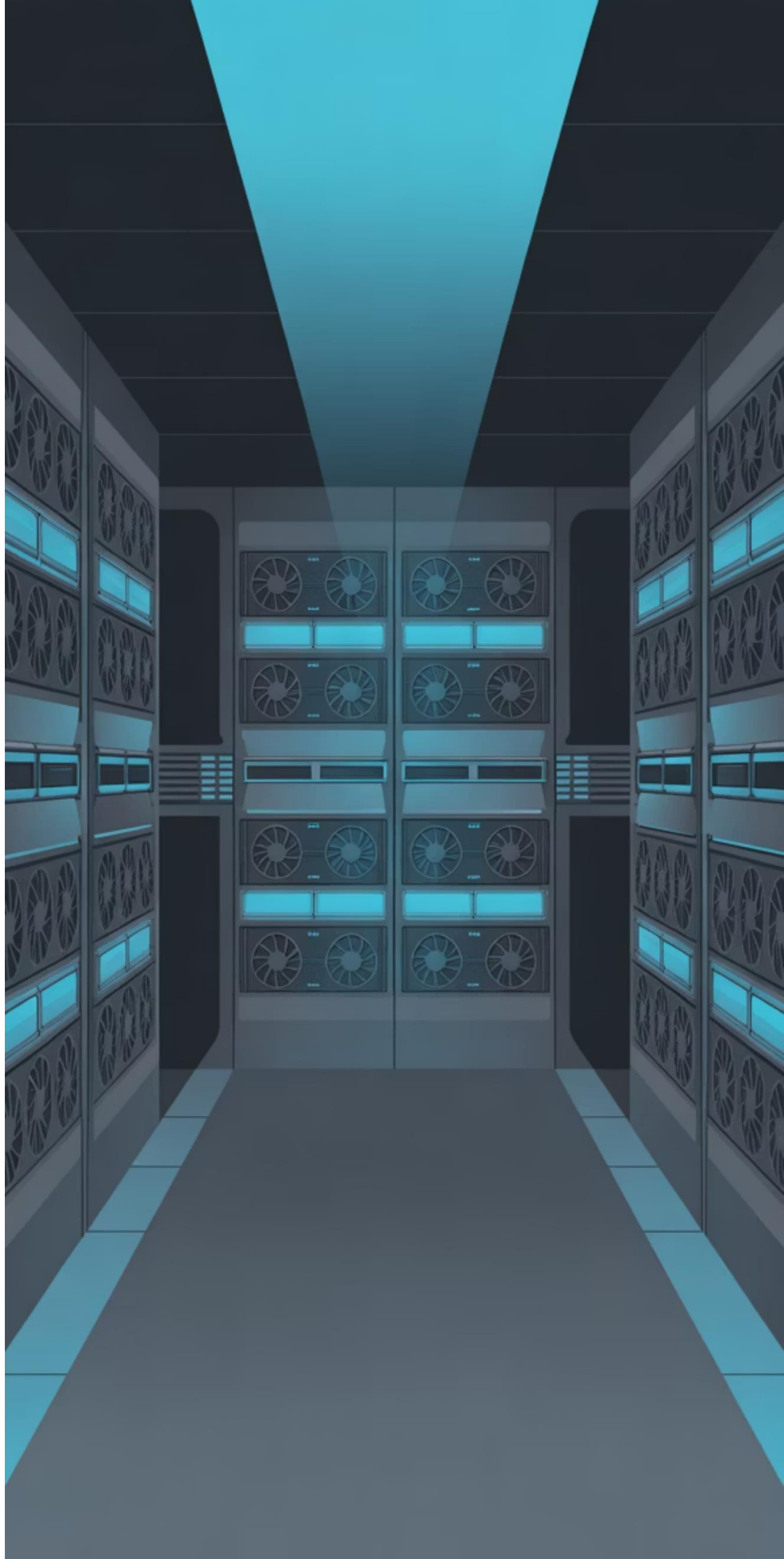
Память GPU

Для полной тонкой настройки

17x

GPU дата-центра

Или 34 потребительских GPU

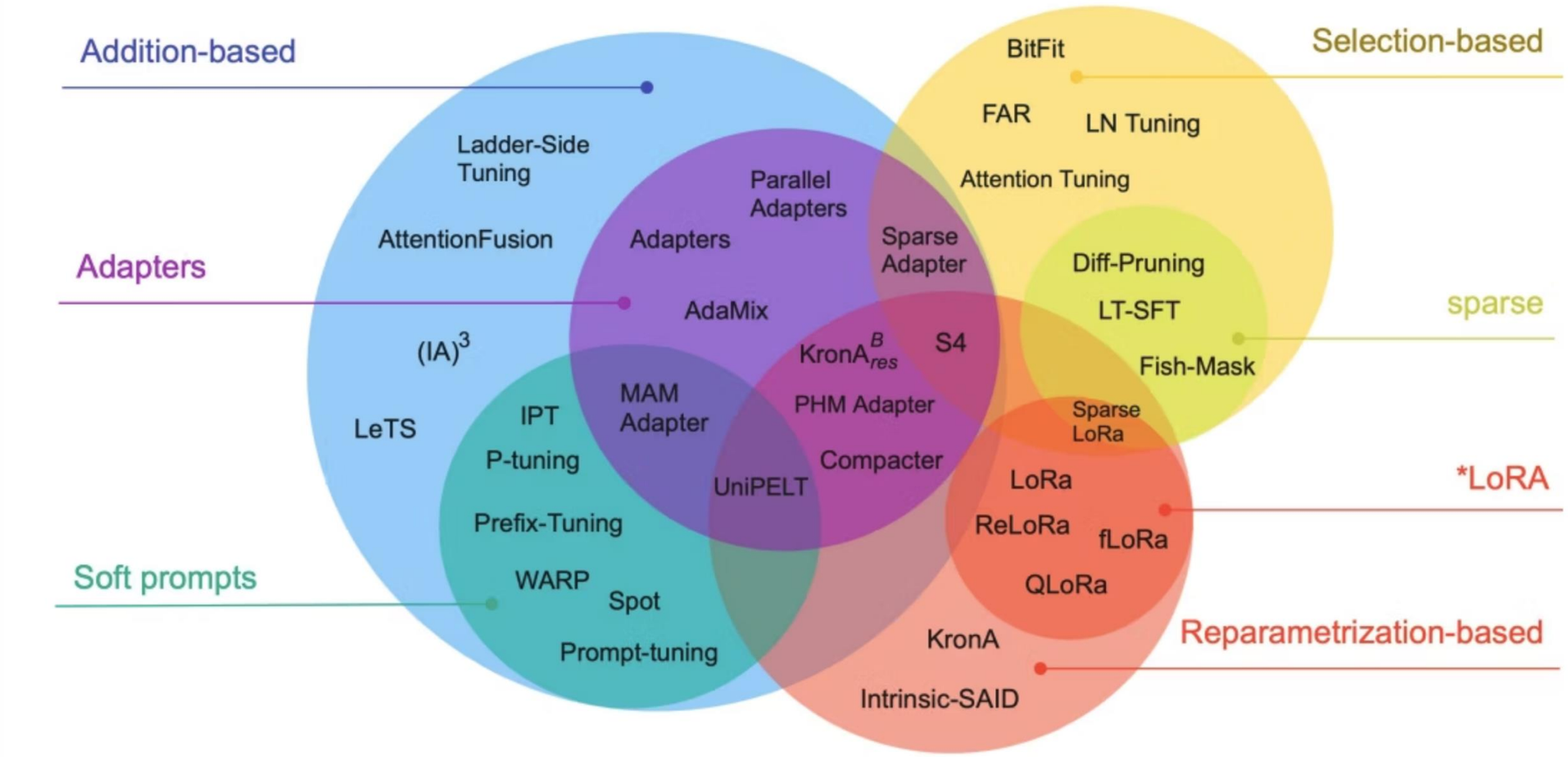


Решение: Параметрически эффективная тонкая настройка (PEFT)

PEFT обновляет только небольшое подмножество параметров, сохраняя качество модели. Сравним требования к памяти для модели с 65B параметров:

Полная настройка	LoRA	QLoRA
96 бит/параметр	**17.6 бит/параметр**	**5.2 бит/параметр**
780 GB памяти GPU	143 GB памяти GPU	42 GB памяти GPU
17 GPU дата-центра	4 GPU дата-центра	1 GPU дата-центра

Методы PEFT: Обзор подходов



Селективные методы

Выбирают подмножество параметров для тонкой настройки, замораживая остальные части модели

Методы репараметризации

Используют низкоранговое представление весов модели. Пример: LoRA

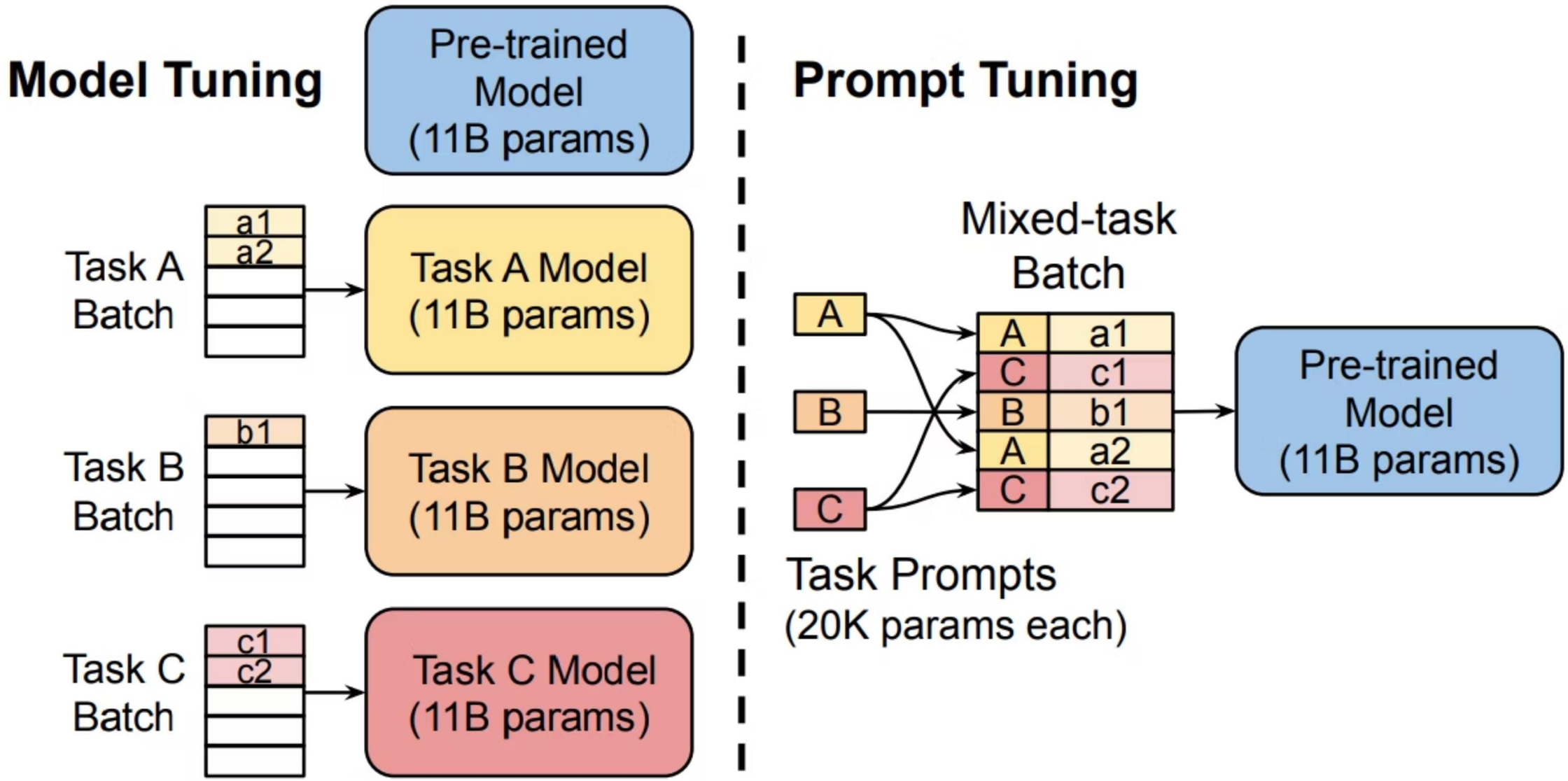
Аддитивные методы

Добавляют обучаемые слои или параметры к модели. Примеры: адаптеры, мягкие промпты

Каждый метод имеет свои компромиссы между эффективностью памяти, эффективностью параметров, производительностью модели, скоростью обучения и затратами на инференс.

Source: Lialin et al., 2023, Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning

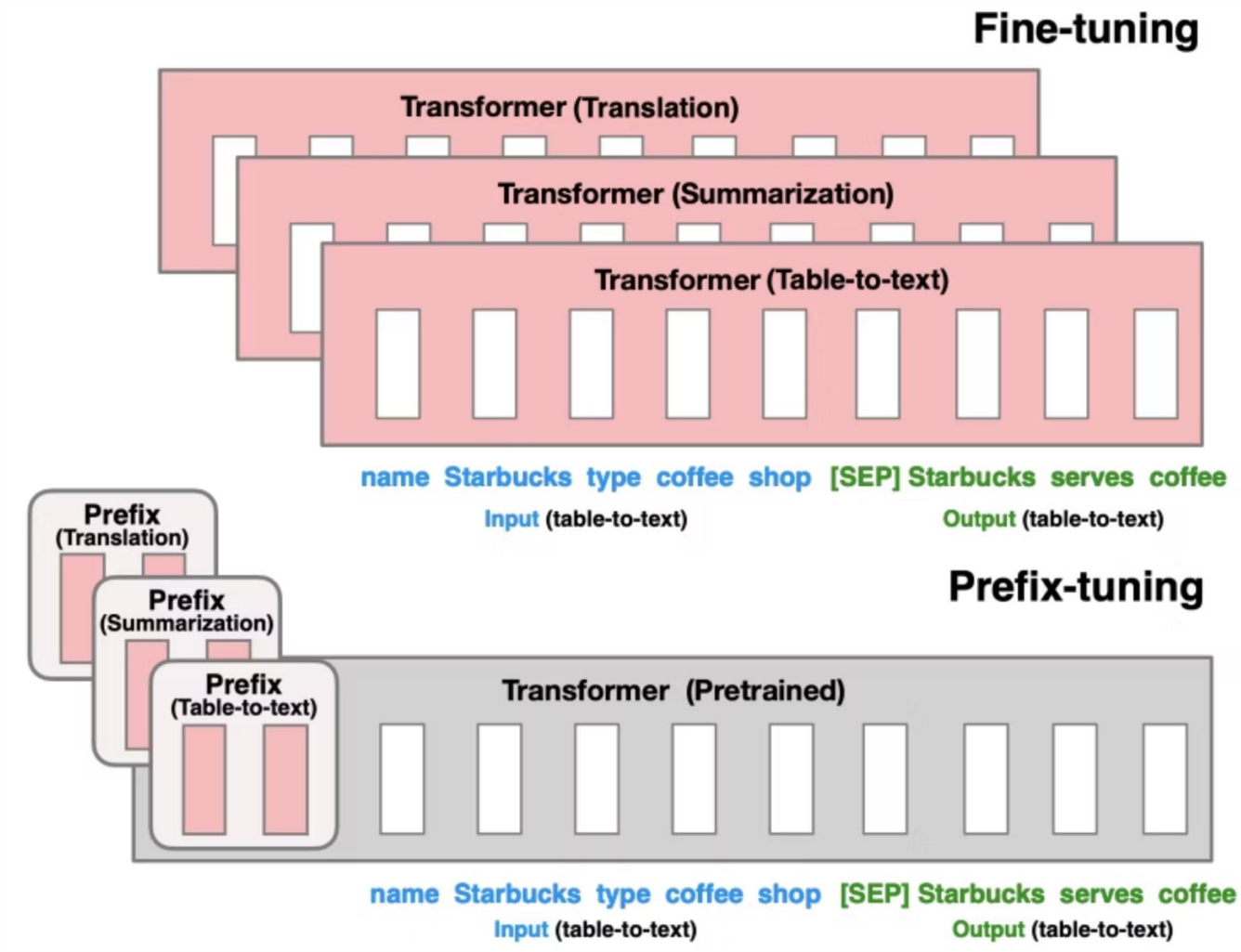
Prompt Tuning: Обучение мягких промптов



Prompt tuning добавляет обучаемые токены-промпты к входным данным, сохраняя параметры модели замороженными. Этот метод позволяет адаптировать модель к конкретным задачам с минимальными вычислительными затратами.

Source: Lester et al., 2021, The Power of Scale for Parameter-Efficient Prompt Tuning

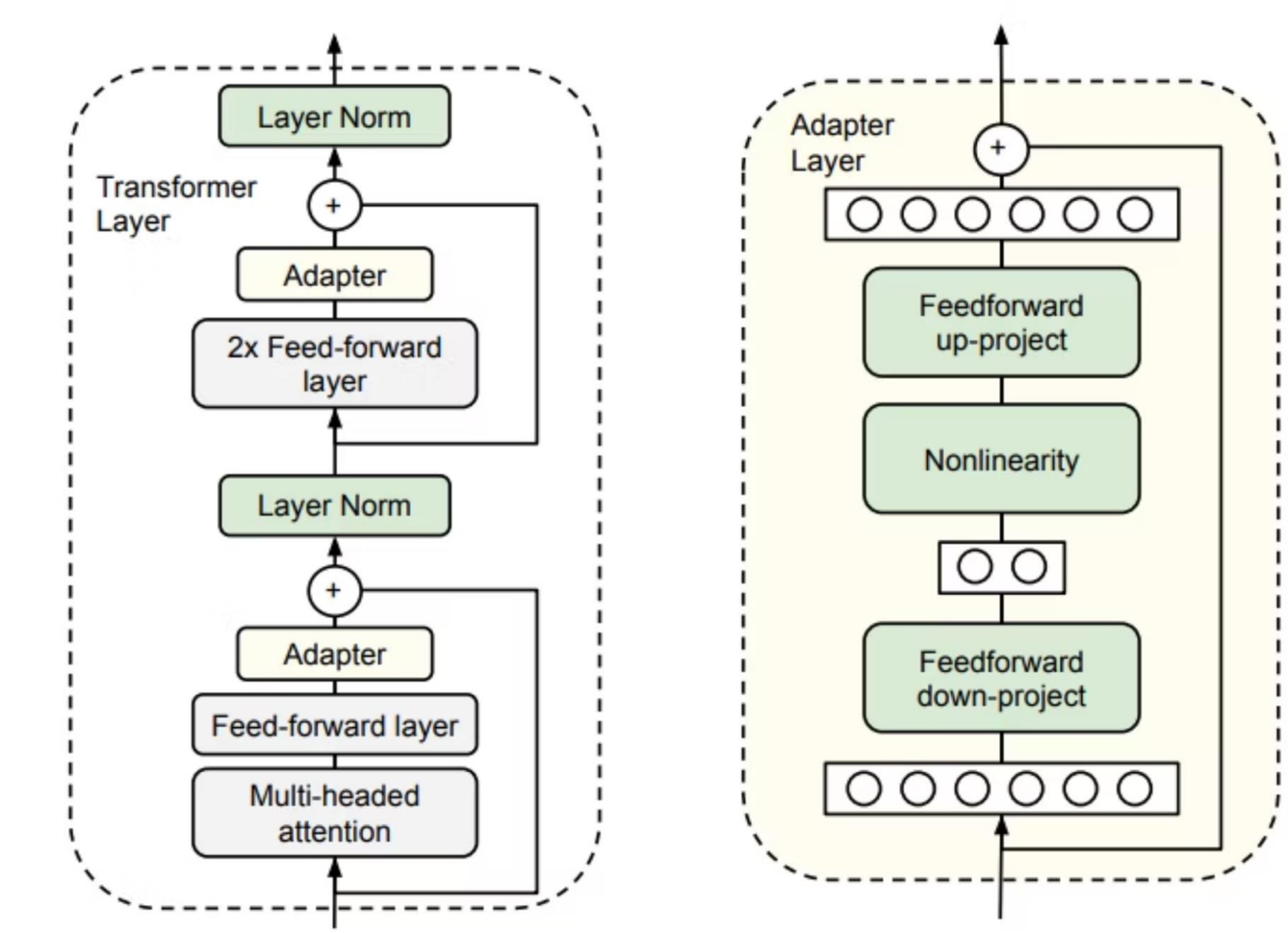
Prefix Tuning: Обучение префиксов



Prefix tuning добавляет обучаемые префиксные векторы к каждому слою трансформера, позволяя модели адаптироваться к новым задачам без изменения основных весов.

Source: Li, Liang, 2021, Prefix-Tuning: Optimizing Continuous Prompts for Generation

Adapter: Добавление адаптерных слоев



Адаптеры вставляют небольшие обучаемые модули между слоями предобученной модели. Эти модули содержат bottleneck архитектуру, которая сначала уменьшает размерность, затем увеличивает её обратно.

Source: Houlsby et al., 2019, Parameter-Efficient Transfer Learning for NLP

LoRA: Низкоранговая адаптация

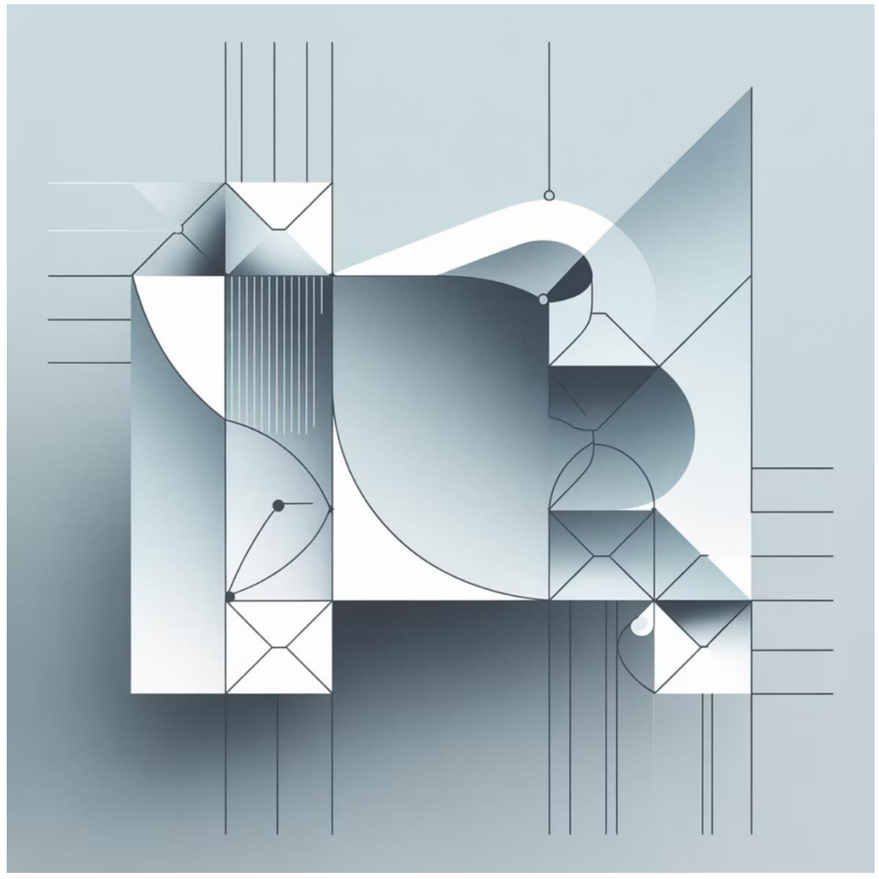
Это дорого — файнтюнить модель полностью. Дорого по времени и ресурсам.

- Во время тонкой настройки мы инициализируем предобученные параметры и обновляем их согласно целевой функции:

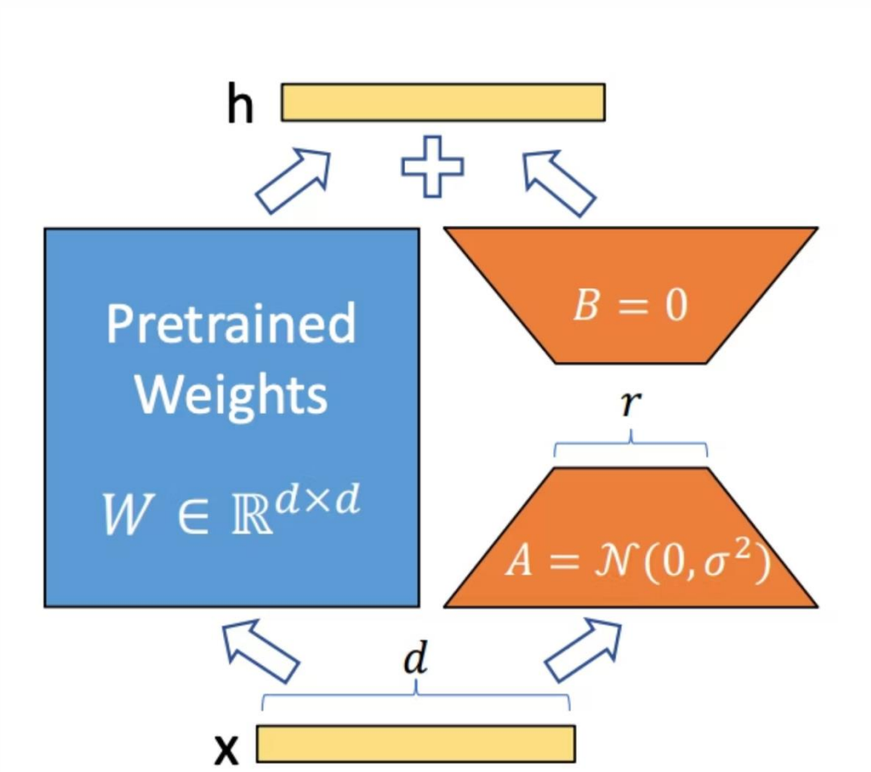
$$\max_{\Phi} \sum_{i} \sum_{t} \log p_{\Phi}(y_t \mid x, y_{<t})$$

- Мы можем предположить, что обновленные матрицы в адаптации имеют низкий "внутрений ранг", что приводит нас к подходу Low-Rank Adaptation (LoRA)
- Для каждой последующей задачи нам не нужно хранить/развертывать разные наборы параметров.

Ключевой вопрос: Можем ли мы найти параметрически эффективный подход, используя низкий внутренний ранг?



LoRA: Обучение и инференс



1. **Предобучение**

   Исследования показывают, что предобученные LLM имеют "низкую внутреннюю размерность" и могут эффективно обучаться при низкоразмерной репараметризации

2. **Обучение LoRA**

   Для предобученного веса W_0 фиксируем его и обучаем

3. **Инференс**

   Объединяем веса: $W = W_0 + BA$

Математические основы LoRA:

During training: для предобученного веса *W_0 \in \mathbb{R}^{d \times k}*, *W_0* фиксирован

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

$$B \in \mathrm{R}^{d \times r}, A \in \mathrm{R}^{r \times k}, r \ll \min(d, k)$$

During inference:

$$W = W_0 + BA$$

Обратное распространение в LoRA

W_0 — это предобученная матрица весов, которая остается фиксированной во время обучения.

Градиенты функции потерь относительно A и B вычисляются с использованием правила цепочки:

$$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial A} = \frac{\partial L}{\partial h} \frac{\partial (W_0 x + BAx)}{\partial A} = \frac{\partial L}{\partial h} Bx^T$$

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial B} = \frac{\partial L}{\partial h} \frac{\partial (W_0 x + BAx)}{\partial B} = \frac{\partial L}{\partial h} Ax^T$$

Эти частные производные вычисляются во время backward pass для обновления параметров A и B.

Преимущества LoRA

Применение к трансформеру

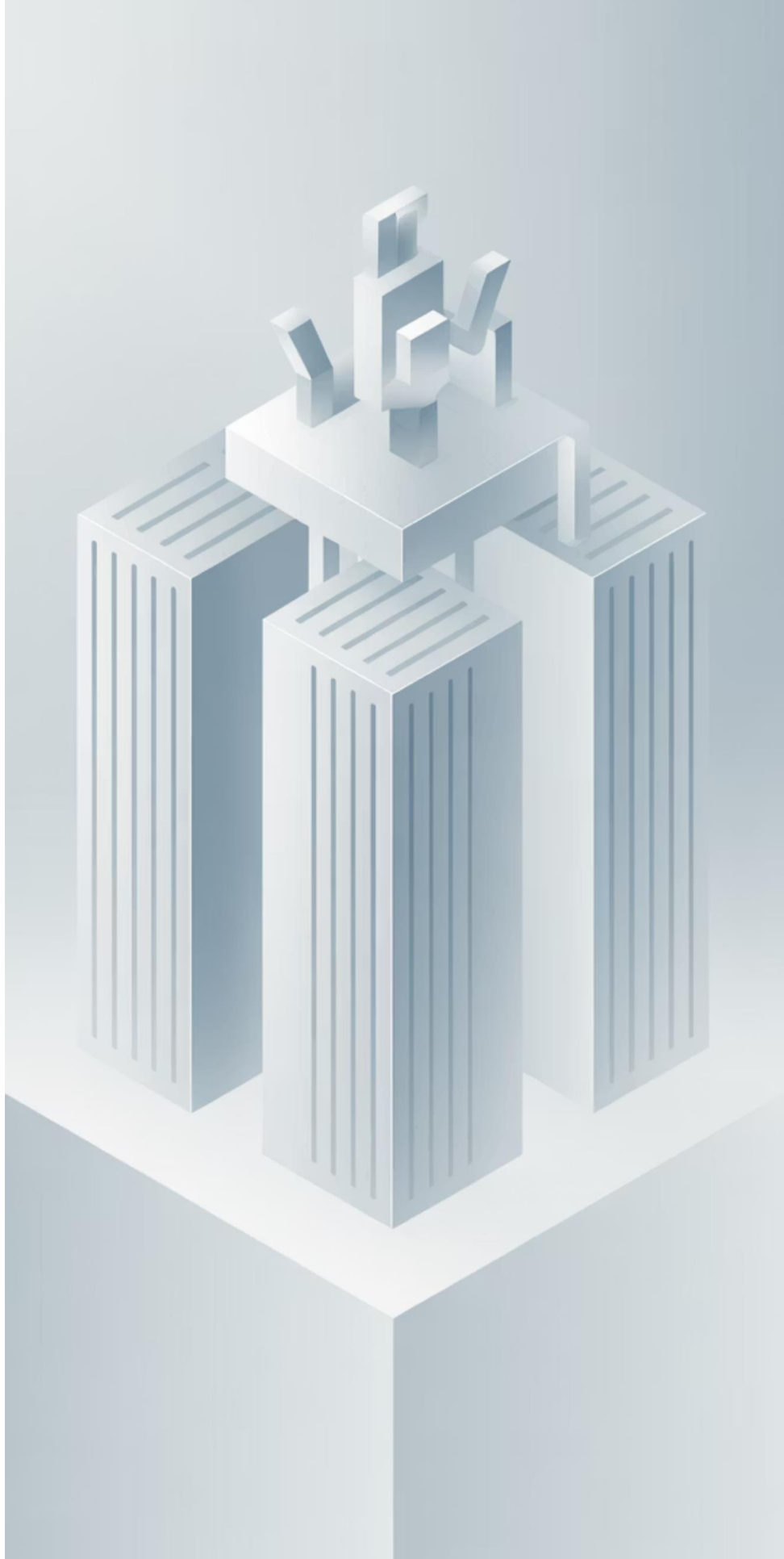
В принципе, LoRA можно применить к любым весовым матрицам в глубоком обучении.

Оригинальная статья изучает только изменение весов внимания:

$$W_q, W_k, W_v \in \mathrm{R}^{d_{model} \times d_{model}}$$

Ключевые преимущества

- Не нужно отслеживать состояния оптимизатора для замороженных параметров
- Меньший размер чекпоинта (GPT-3: 1.2TB → 350GB → 35MB)
- Нет дополнительной задержки инференса
- Ускорение во время обучения



Оптимальная конфигурация LoRA

Два ключевых вопроса при применении LoRA к трансформеру:

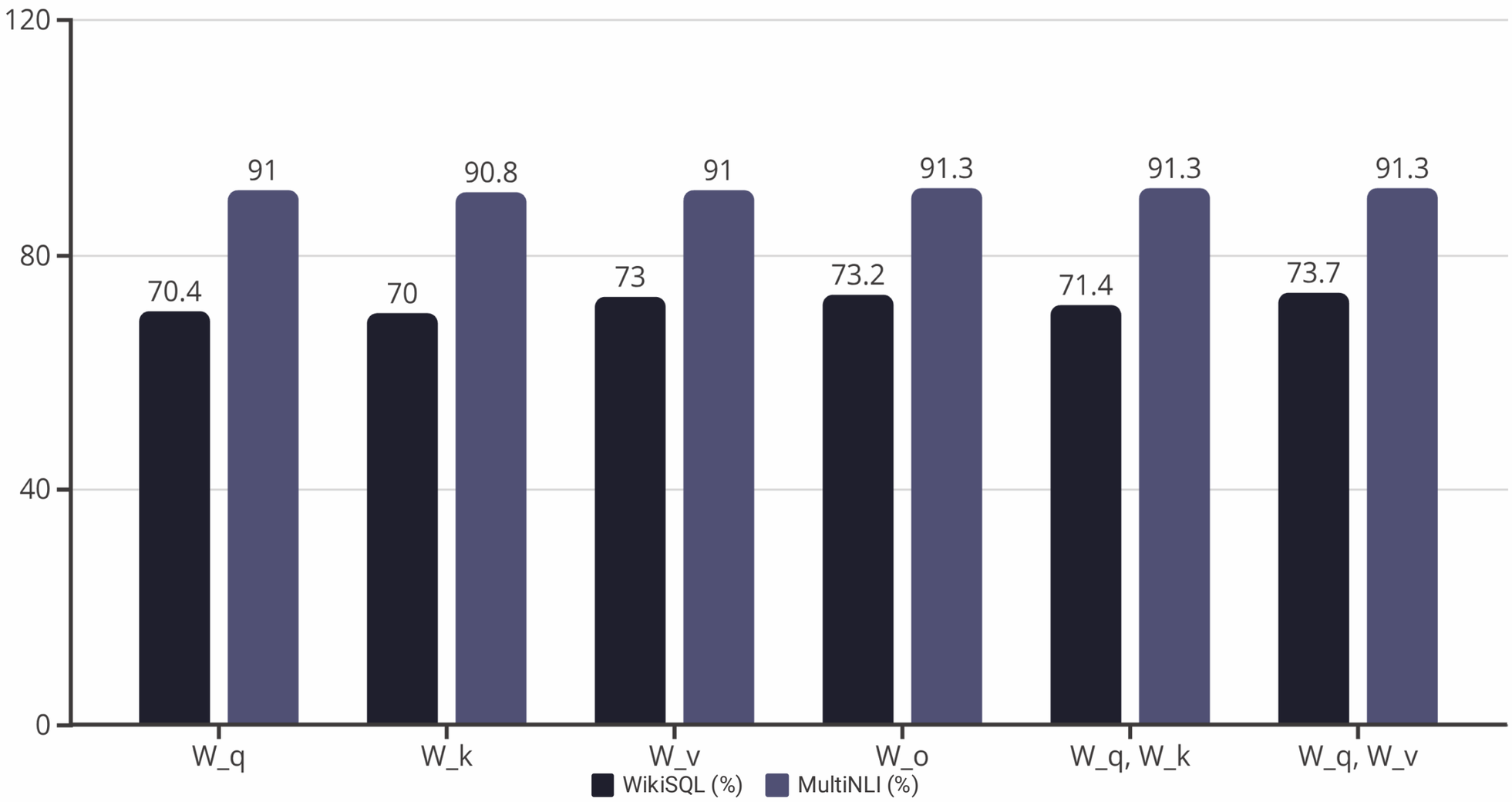
Куда?

К каким весовым матрицам в трансформере следует применять LoRA для достижения наилучших результатов?

Какой ранг?

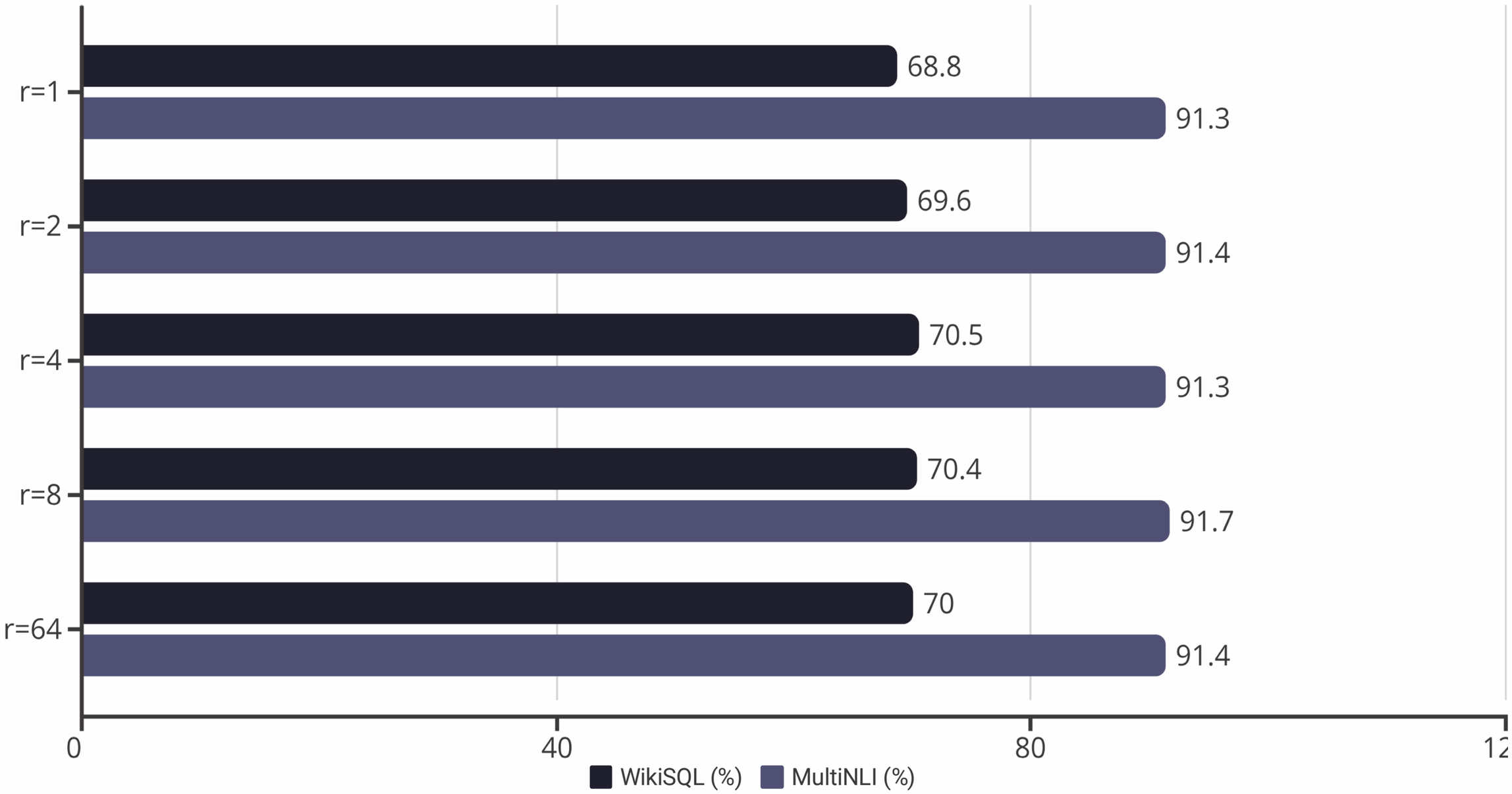
Какой оптимальный ранг r для LoRA? Увеличение ранга не охватывает более значимые подпространства, что предполагает достаточность низкоранговой адаптационной матрицы.

Количество обучаемых параметров: 18M

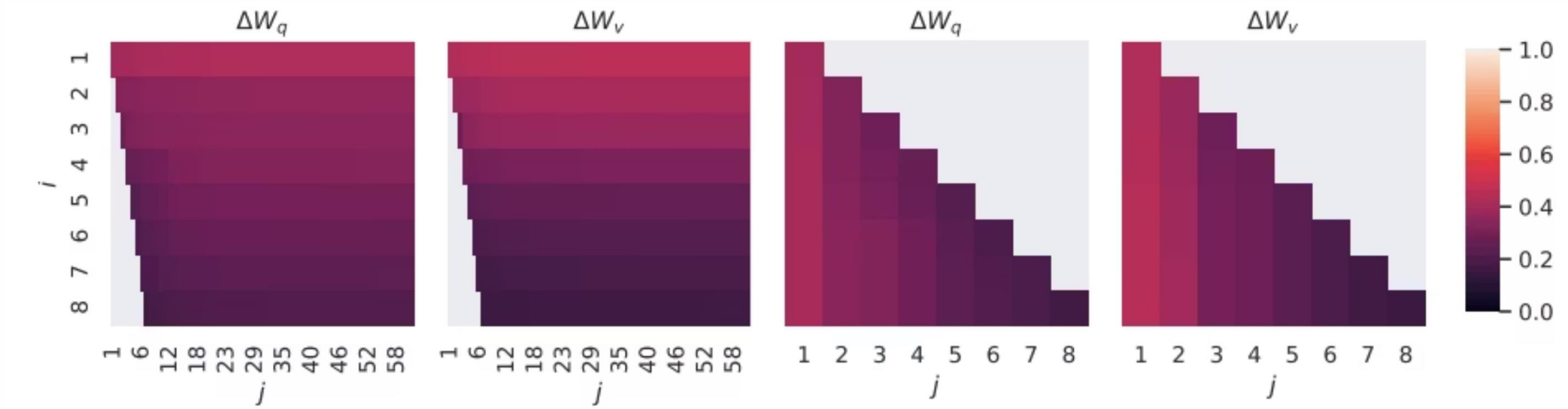


Влияние ранга на производительность

Эксперименты показывают, что производительность стабилизируется при относительно низких значениях ранга:



Сходство подпространств между разными рангами



Сравнение r=8 и r=64 после SVD-декомпозиции показывает интересные закономерности:

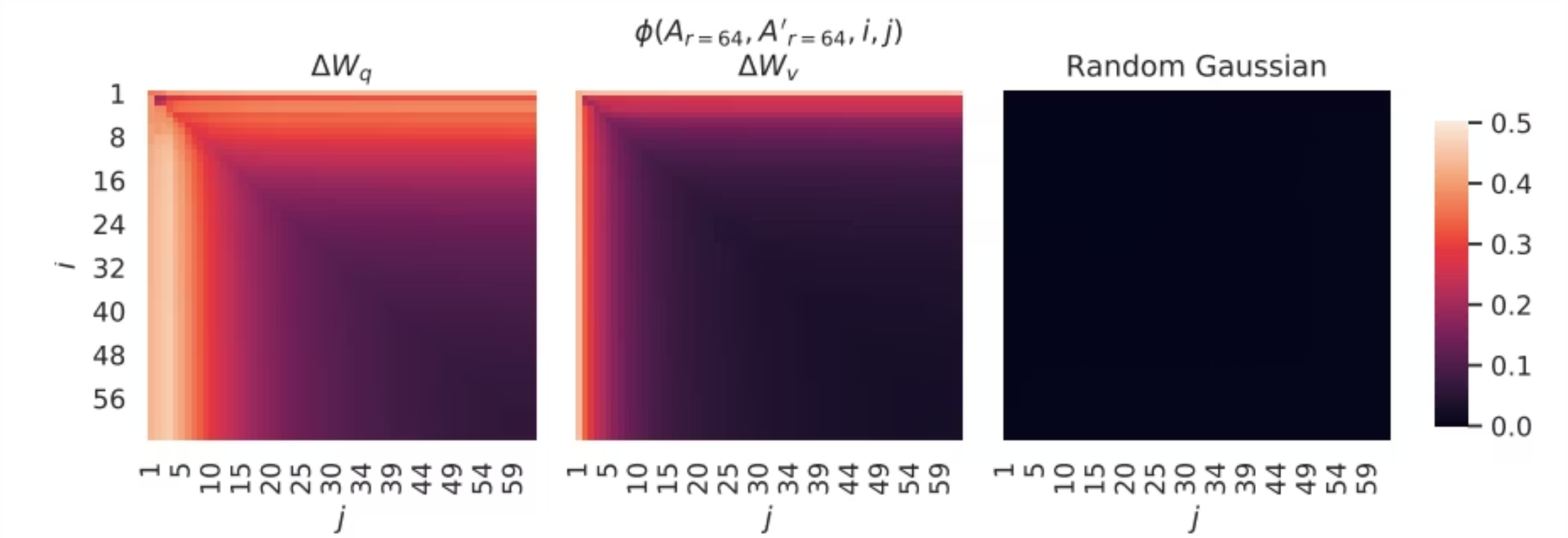
1. ## Перекрытие верхних векторов

   Верхние сингулярные векторы значительно перекрываются между r=8 и r=64, в то время как остальные направления не показывают такого сходства

2. ## Важность направлений

   Это указывает, что направления верхних сингулярных векторов ΔW наиболее полезны, в то время как другие направления потенциально содержат в основном случайный шум, накопленный во время обучения

Роль ΔW в адаптации модели



ΔW усиливает важные направления: Матрица обновлений ΔW усиливает направления, которые важны, но не подчеркнуты в исходной матрице W.

Эффект большего ранга: ΔW с большим r имеет тенденцию выбирать направления, уже подчеркнутые в W, что может быть менее эффективным.

Экспериментальная установка

Понимание языка (NLU)

Модели: RoBERTa, DeBERTa

Задачи: MNLI, SST-2, MRPC, CoLA, QNLI, QQP, RTE, STS-B

Генерация языка (NLG)

Модели: GPT-2, GPT-3

Метрики: BLEU, NIST, MET, ROUGE-L, CIDEr

Базовые методы

Fine-Tuning, Bias-only (BitFit), Prefix-embedding tuning, Prefix-layer tuning, Adapter tuning, LoRA

Результаты на задачах NLU

LoRA демонстрирует превосходную производительность с меньшим количеством обучаемых параметров:

Модель и метод	Параметры	MNLI	SST-2	MRPC	Среднее
RoBERTa base (FT)	125.0M	87.6	94.8	90.2	86.4
RoBERTa base (BitFit)	0.1M	84.7	93.7	92.7	85.2
RoBERTa base (Adapter)	0.9M	87.3	94.7	88.4	85.4
RoBERTa base (LoRA)	0.3M	87.5	95.1	89.7	87.2

LoRA достигает лучшей средней производительности с всего 0.3M обучаемых параметров — в 416 раз меньше, чем полный fine-tune

Результаты на задачах NLG

LoRA показывает превосходные результаты в генерации естественного языка:

Модель и метод	Параметры	BLEU	NIST	MET	ROUGE-L
GPT-2 M (Fine-Tune)	354.92M	68.2	8.62	46.2	71.0
GPT-2 M (Adapter)	11.48M	68.9	8.71	46.1	71.3
GPT-2 M (Prefix)	0.35M	69.7	8.81	46.1	71.4
GPT-2 M (LoRA)	0.35M	70.4	8.85	46.8	71.8

LoRA превосходит все базовые методы, включая полную тонкую настройку, используя в 1000 раз меньше параметров.

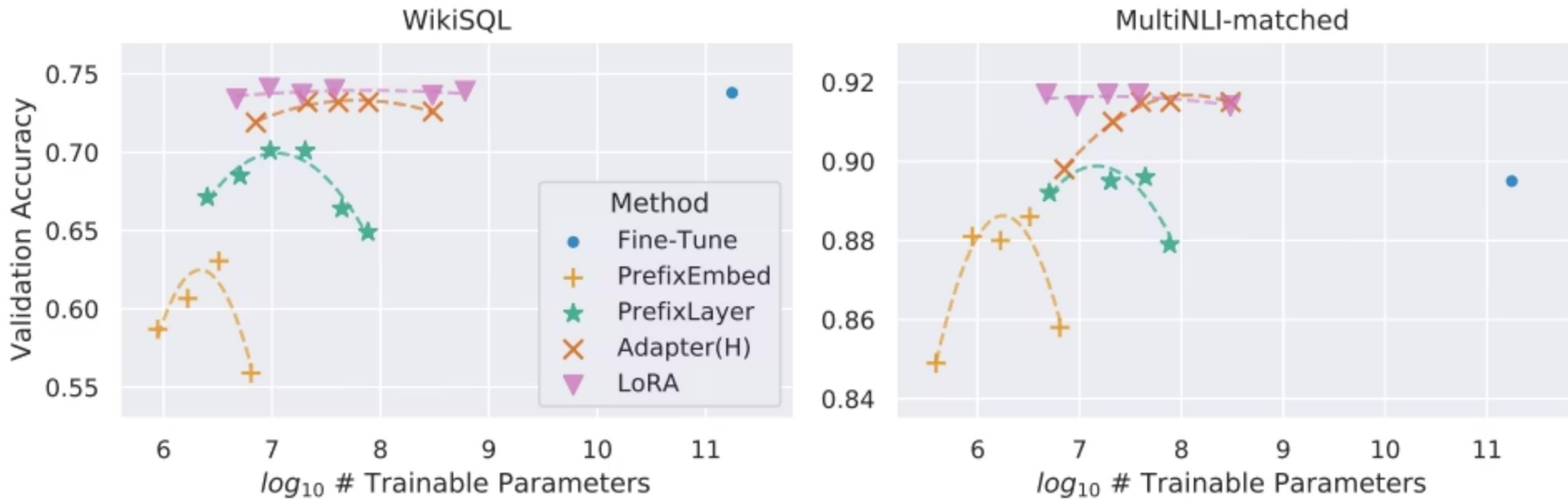
Масштабирование до GPT-3 175B

Стресс-тест NLG: LoRA успешно масштабируется до моделей с 175 миллиардами параметров:

175B	4.7M	91.7%
Параметры GPT-3	Обучаемые параметры	Точность MNLI
Крупнейшая протестированная модель	Всего 0.0027% от общего числа	Превосходит полную настройку

Метод	Параметры	WikiSQL	MNLI-m	SAMSum
GPT-3 (Fine-Tune)	175,255.8M	73.8	89.5	52.0/28.0/44.5
GPT-3 (PrefixLayer)	20.2M	70.1	89.5	50.8/27.3/43.5
GPT-3 (LoRA)	4.7M	73.4	91.7	53.8/29.8/45.9

Не все методы выигрывают от увеличения параметров



Увеличение количества обучаемых параметров не всегда приводит к улучшению производительности. Это подчеркивает важность эффективности метода, а не просто количества параметров.

График показывает, что LoRA поддерживает стабильную производительность при различных размерах параметров, в то время как другие методы демонстрируют непредсказуемое поведение.

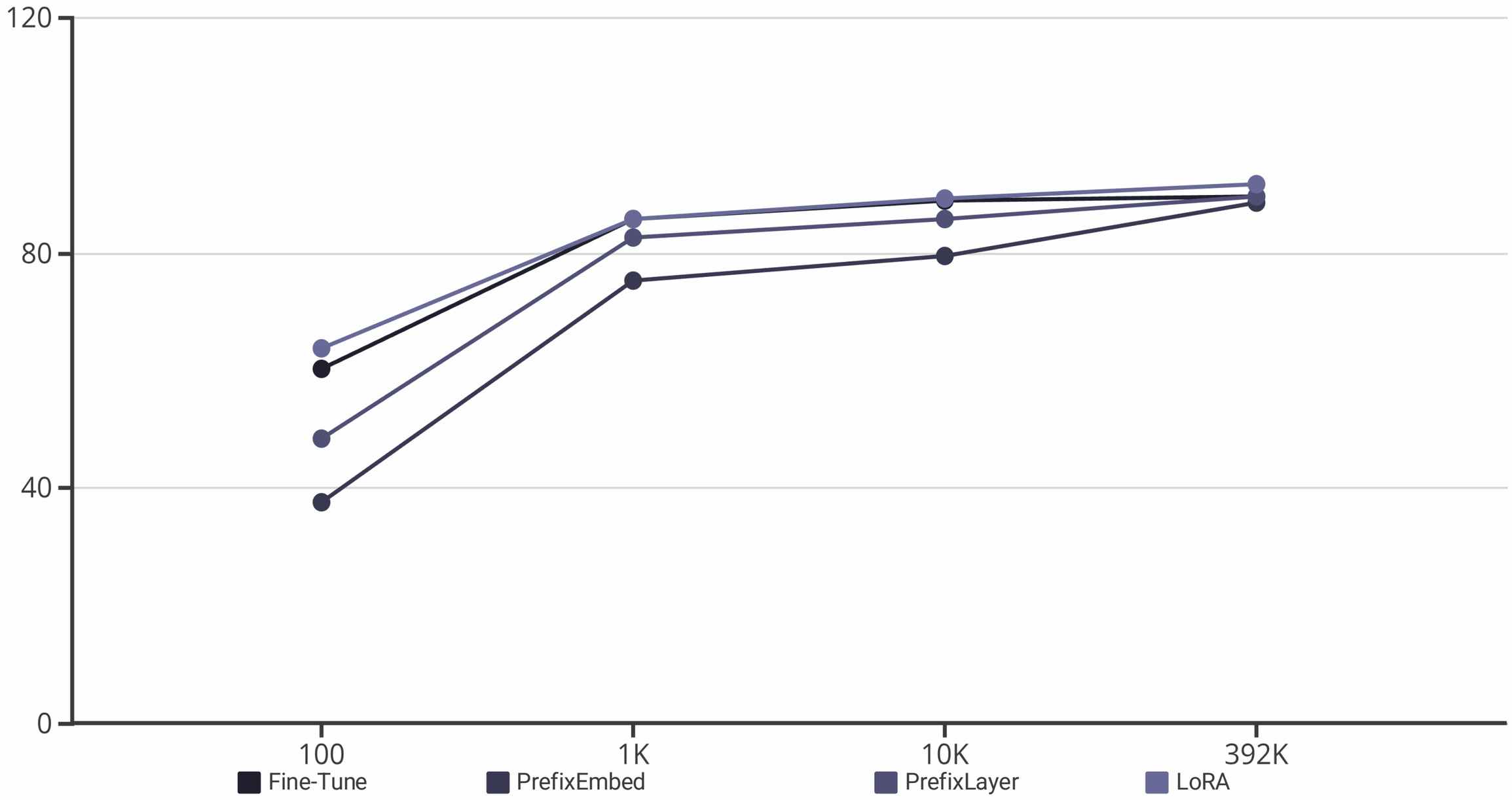
PrefixEmbed показал результаты едва ли лучше, чем случайный выбор (точность 37,6% против 33,3%)

PrefixLayer показал результаты лучше, чем PrefixEmbed, но всё ещё значительно хуже по сравнению с тонкой настройкой и LoRA.

По мере увеличения количества обучающих примеров разрыв в производительности между подходами на основе **префиксов** и такими методами, как **fine-tuning** или **LoRA**, уменьшался, что позволяет предположить, что подходы на основе **префиксов** могут быть не очень подходящими для сценариев с очень малым объёмом данных в таких моделях, как GPT-3.

Производительность при ограниченных данных

LoRA особенно эффективна в сценариях с малым количеством обучающих данных:



PrefixEmbed показывает результаты едва лучше случайного угадывания (37.6% против 33.3%) при 100 примерах. LoRA превосходит все методы во всех сценариях.

Выводы по LoRA

Эффективность

LoRA обеспечивает экономически эффективную адаптацию больших моделей, изменяя меньше параметров

Производительность

Соответствует или превосходит полную тонкую настройку в различных задачах с меньшим количеством обучаемых параметров

Масштабируемость

Эффективна для гигантских моделей вроде GPT-3, делая адаптацию более доступной

Эффективность при малых данных

Превосходна в условиях ограниченных данных, снижая потребность в больших датасетах

Нулевая задержка

Сохраняет качество модели без добавления задержки инференса или уменьшения длины входной последовательности

Широкая применимость

Принципы LoRA адаптируемы к различным архитектурам нейронных сетей за пределами языковых моделей

QLoRA: Квантованная LoRA

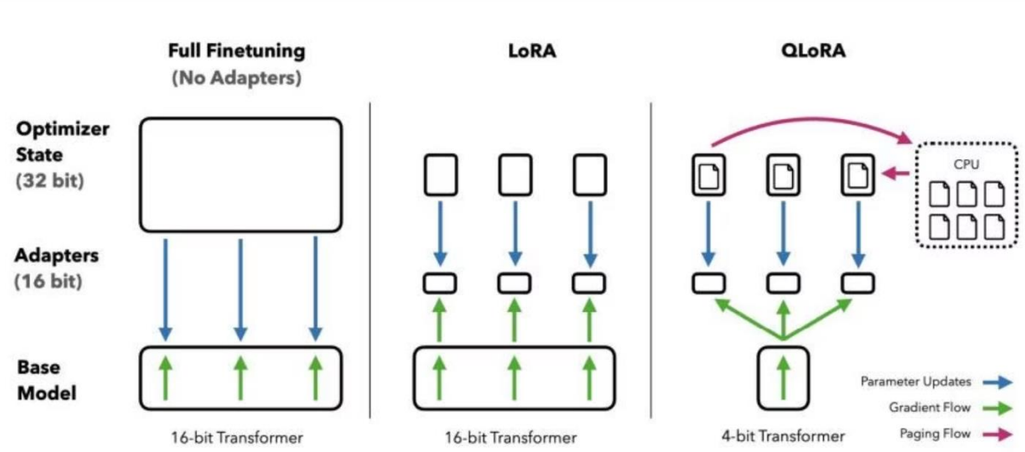
QLoRA = Квантованная предобученная LLM + LoRA

Основные инновации:

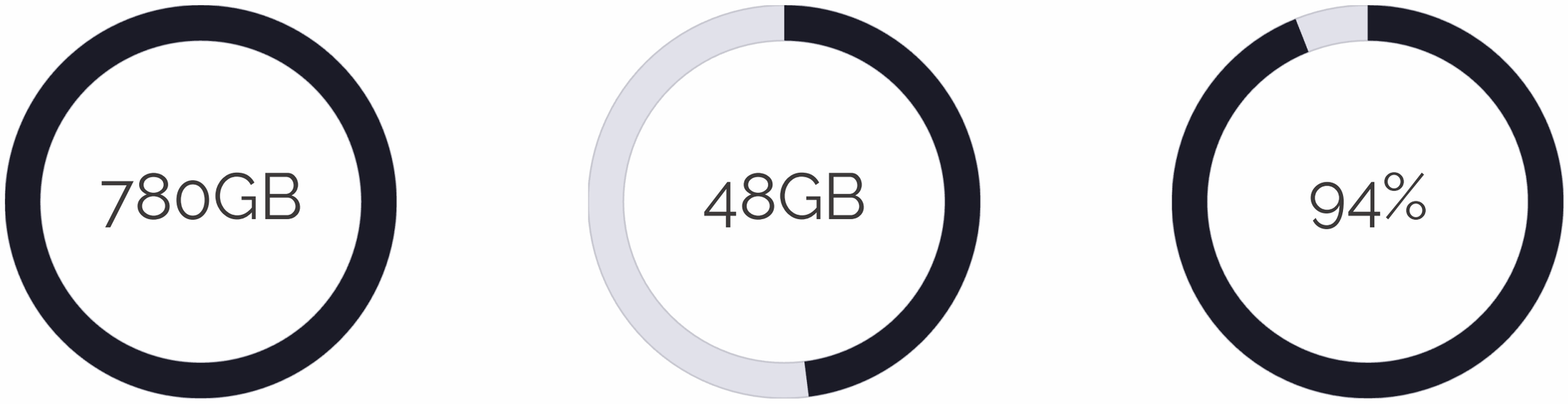
- 4-битный нормальный Float (NF4)
- Двойная квантизация
- Страничные оптимизаторы

Типы данных:

- 4-битный тип хранения
- Bfloat16 вычислительный тип



Революционное сокращение памяти с QLoRA



До QLoRA

Требования к памяти для модели 65B

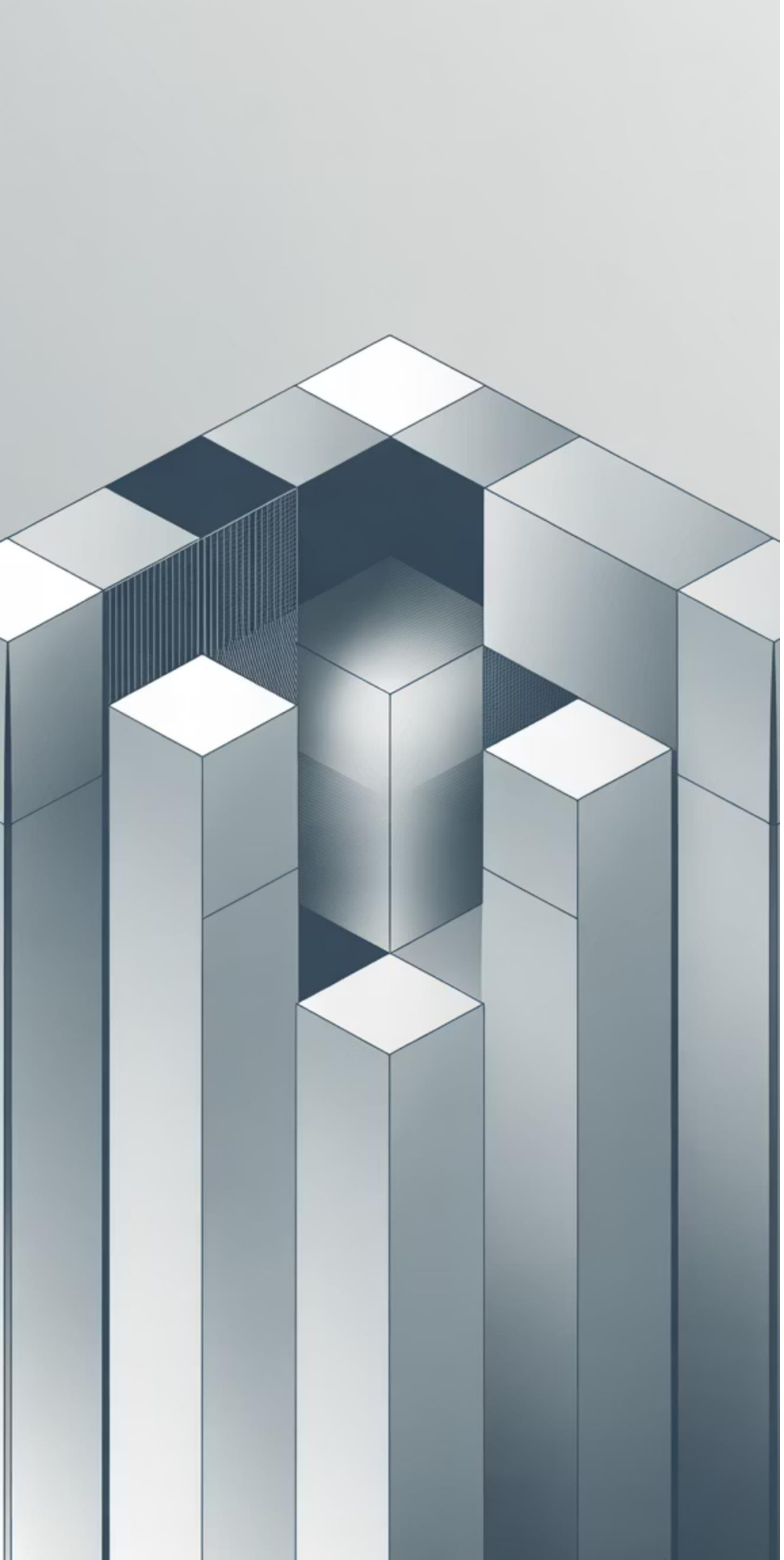
После QLoRA

На одном GPU!

Сокращение памяти

При сохранении производительности

QLoRA сокращает средние требования к памяти для тонкой настройки модели с 65B параметров с 780GB до 48GB на **одном GPU**, сохраняя производительность по сравнению с 16-битным базовым уровнем полной тонкой настройки.



Квантизация модели: Основы

Квантование сохраняет модель неизменной, но уменьшает количество бит для представления параметров.

Посттренинговая квантизация (PTQ)

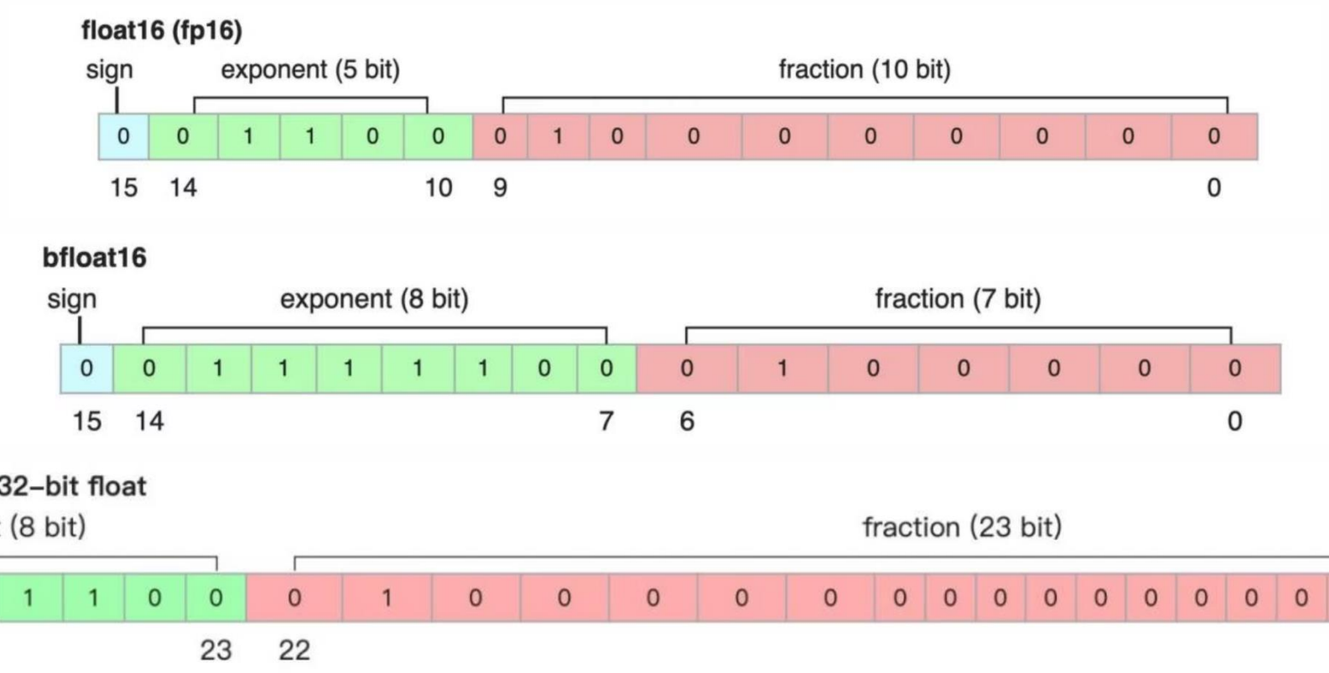
Преобразование весов уже обученной модели в более низкую точность без переобучения. PTQ может ухудшить производительность модели.

Квантизация с учетом обучения (QAT)

Интегрирует процесс преобразования весов на этапе обучения. Приводит к превосходной производительности модели. QLoRA использует QAT.

Числа с плавающей точкой: Структура

Понимание представления чисел с плавающей точкой критично для квантования:



Число с плавающей точкой состоит из трех компонентов: знак (1 бит), экспонента (определяет диапазон) и мантисса (определяет точность).

Пример квантования: Absolute Maximum

Использование метода Absolute Maximum (absmax) для квантования 32-битного тензора FP32 в тензор Int8 с диапазоном [-127, 127]:

$$\mathbf{X}_{\text{quant}} = \text{round}\left(\frac{127}{\max|\mathbf{X}|}\mathbf{X}\right)$$

Пример: Дан FP32 тензор [1.2, -3.1, 0.8, 2.4, 5.4]

01

Найти максимум

max|X| = 5.4

02

Вычислить масштабный коэффициент

127 / 5.4 ≈ 23.52

03

Квантовать

[28, -73, 19, 56, 127]

4-битное нормальное Float квантование (NF4)

Мотивация и преимущества

Мотивация: Веса в предобученных LLM обычно имеют нормальное распределение с центром в нуле.

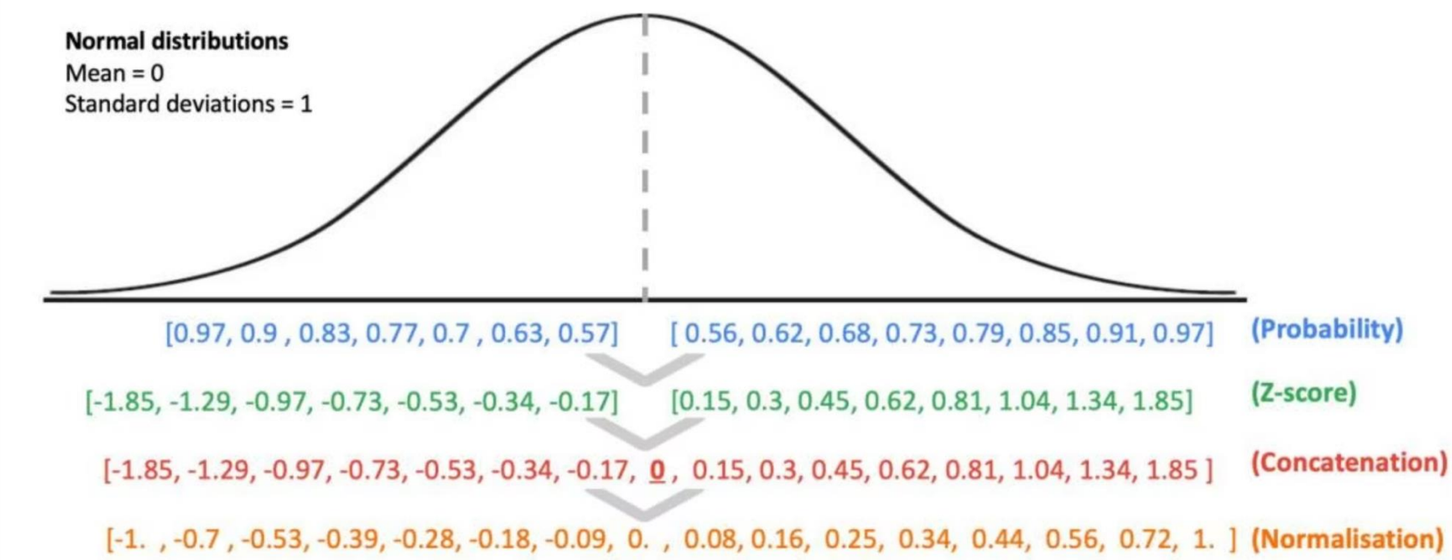
Преимущество NF-4: Это информационно-теоретически оптимальный тип данных квантования для нормально распределенных данных, который дает лучшие эмпирические результаты, чем 4-битные целые числа и 4-битные Float.

Процесс вычисления (k=4)

1. Оценить 16+1 квантилей теоретического распределения N(0,1)
2. Нормализовать значения в диапазон [-1, 1]
3. Квантовать входной тензор весов в диапазон [-1, 1]

$$q_i = \frac{1}{2}(Q_X(\frac{i}{2^k + 1}) + Q_X(\frac{i + 1}{2^k + 1}))$$

Точные значения типа данных NF4



NF4 использует 16 дискретных значений, оптимизированных для нормального распределения:

Эти значения тщательно выбраны для минимизации ошибки квантования при работе с весами, следующими нормальному распределению, что типично для предобученных языковых моделей.

Блочное квантование: Решение проблемы выбросов

Проблема

Выбросы во входном тензоре приводят к неэффективному использованию квантовых бинов, так как большинство значений сжимается в узкий диапазон.

Решение

Разбиваем входной тензор на n непрерывных блоков размера B, каждый со своей константой

Двойная квантизация: Экономия памяти на константах

В методах квантования, таких как блочное квантование, константы масштабирования (c1 и c2) обычно хранятся в формате полной точности. Для больших моделей эти константы, хоть и меньше основных весов, могут значительно увеличивать потребление памяти.

Проблема

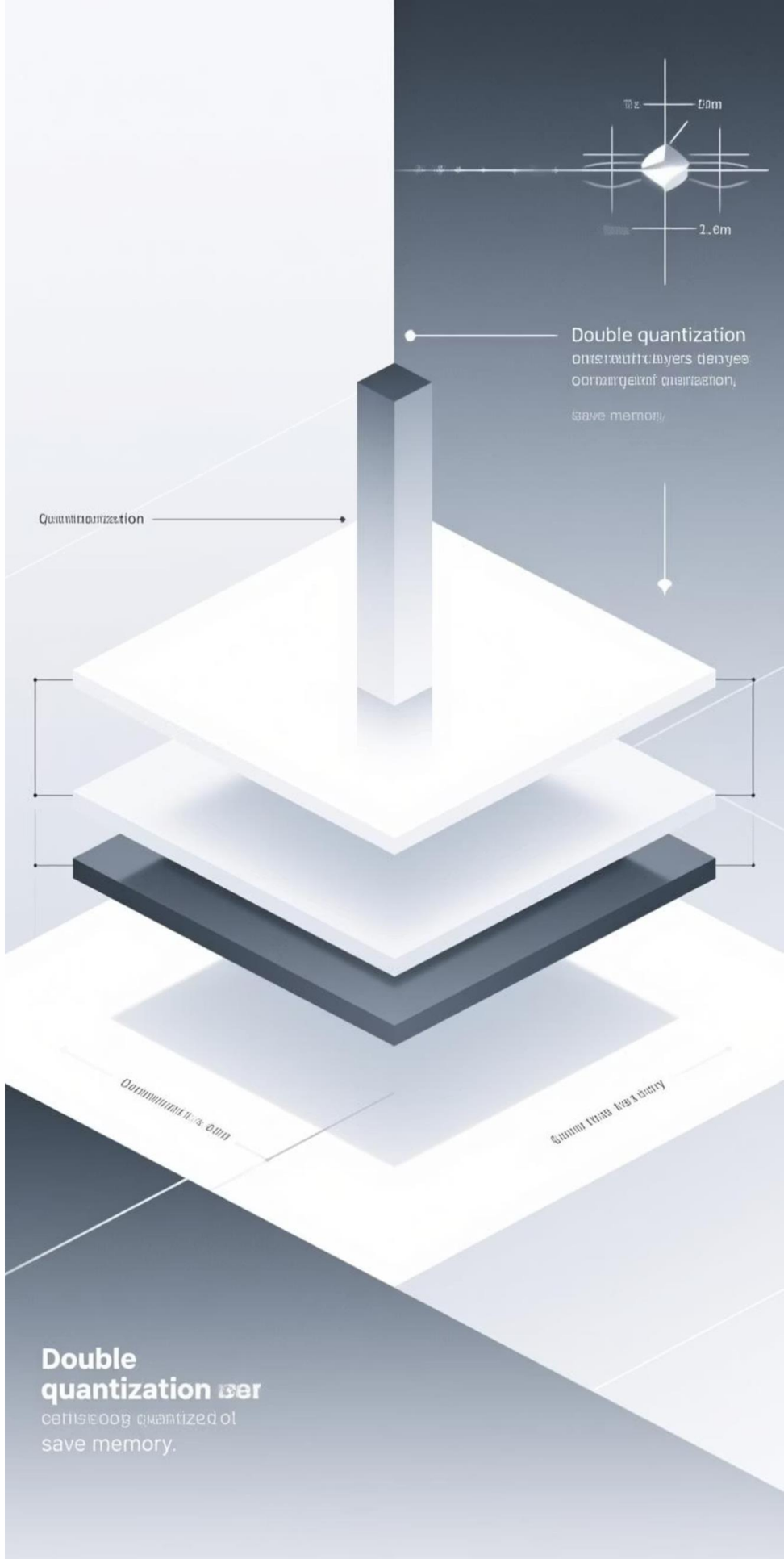
Константы квантизации, такие как c1 и c2, используемые для масштабирования весов в блочном квантовании, сами по себе занимают дополнительную память. Для большой модели с многочисленными блоками эти константы могут накапливаться и значительно увеличивать требования к памяти.

Решение

Двойная квантизация решает эту проблему путем квантования самих этих констант. Вместо хранения их в полном 32-битном формате, они квантуются в 8-битные числа с плавающей точкой (Float) с использованием блочного размера 256.

$$c_2 = \text{Quantize8bitFloat}(c_1)$$

Это приводит к дополнительной экономии памяти до 0.5 бит на параметр, что в совокупности позволяет добиться значительного снижения общего объема потребляемой памяти для всей модели.



Страничные оптимизаторы: Управление пиками памяти

Проблема

Во время тонкой настройки больших языковых моделей (LLM) оптимизатор Adam, часто используемый в QLoRA, требует значительного объема памяти для хранения своих состояний (моменты градиента) для каждого параметра. Эти состояния оптимизатора могут вызывать пики памяти, особенно когда модель большая, приводя к ошибкам нехватки памяти (OOM).

Решение

Страничные оптимизаторы решают эту проблему, интеллектуально перемещая состояния оптимизатора, которые не используются в данный момент, из GPU памяти в CPU память. Когда эти состояния требуются снова, они переносятся обратно в GPU. Этот механизм работает аналогично страничной памяти в операционных системах.

В результате этого процесса GPU память используется более эффективно, поскольку на GPU хранятся только активно используемые состояния. Это позволяет тонкой настройке гораздо больших моделей, чем было возможно ранее, на том же оборудовании, значительно сокращая общее потребление GPU памяти и позволяя обучать модели, которые иначе не поместились бы в доступную VRAM.

Экспериментальные результаты QLoRA

Ниже представлены экспериментальные результаты, демонстрирующие эффективность QLoRA по сравнению с полной тонкой настройкой на различных задачах для моделей LLaMA разных размеров.

LLaMA 7B	75.2%	75.0%	4
LLaMA 13B	78.5%	78.3%	8
LLaMA 30B	81.1%	80.9%	24
LLaMA 65B	83.0%	82.8%	48

Как видно из таблицы, QLoRA обеспечивает производительность, очень близкую к полной тонкой настройке, при значительном сокращении требований к памяти, что позволяет работать с более крупными моделями на доступном оборудовании.

Сравнение методов PEFT: Производительность vs Эффективность

Для тонкой настройки больших языковых моделей (LLM) существует несколько методов эффективной настройки параметров (PEFT), каждый из которых предлагает уникальный баланс между количеством обучаемых параметров, требованиями к памяти, качеством результатов и скоростью обучения. Ниже приведена сравнительная таблица наиболее популярных методов.

	Количество обучаемых параметров	Требования к памяти	Качество результатов	Скорость обучения	Преимущества / Недостатки
LoRA	Очень мало (0.01% - 0.1% от исходной модели)	Низкие	Высокое (близко к полной тонкой настройке)	Быстрая	**Преимущества:** Высокая производительность, низкие затраты памяти. **Недостатки:** Требует хранения адаптеров в полной точности.
QLoRA	Чрезвычайно мало (аналогично LoRA, но с 4-битной квантизацией)	Чрезвычайно низкие	Очень высокое (почти идентично LoRA и полной тонкой настройке)	Быстрая	**Преимущества:** Максимально эффективное использование памяти, позволяет дообучать огромные модели на потребительских GPU. **Недостатки:** Более сложная реализация из-за квантизации.
Adapter	Мало (добавление новых слоев между существующими)	Низкие	Хорошее (иногда немного ниже, чем LoRA/QLoRA)	Быстрая	**Преимущества:** Модульность, легкость замены адаптеров. **Недостатки:** Добавляет задержку из-за дополнительных слоев.
Prefix Tuning	Мало (обучение виртуальных токенов-префиксов)	Низкие	Хорошее (эффективно для генерации)	Быстрая	**Преимущества:** Не изменяет внутренние параметры модели, эффективно для генеративных задач. **Недостатки:** Чувствительность к длине и инициализации префикса.
Prompt Tuning	Очень мало (обучение только "мягкого" промпта)	Очень низкие	Хорошее (но может быть ниже, чем у других PEFT)	Самая быстрая	**Преимущества:** Простейшая реализация, минимальный объем памяти, хорош для обучения с малым количеством примеров. **Недостатки:** Менее выразительный, чем другие методы.

Выбор оптимального метода PEFT зависит от конкретных задач и доступных ресурсов. QLoRA часто является предпочтительным выбором, когда критически важна экономия памяти при сохранении высокой производительности.

Практические рекомендации по применению LoRA/QLoRA

Для эффективного использования LoRA/QLoRA и достижения оптимальных результатов необходимо учитывать несколько ключевых практических рекомендаций:

Применение этих рекомендаций поможет оптимизировать процесс тонкой настройки LLM с использованием LoRA/QLoRA, добиваясь высокого качества при минимизации использования ресурсов.

Инструменты и библиотеки для LoRA/QLoRA

Для эффективной реализации и работы с методами PEFT, такими как LoRA и QLoRA, существует ряд ключевых инструментов и библиотек, которые значительно упрощают процесс тонкой настройки больших языковых моделей.

Hugging Face PEFT (Parameter-Efficient Fine-Tuning)

Библиотека Hugging Face PEFT предоставляет унифицированный и простой в использовании интерфейс для применения различных методов эффективной тонкой настройки параметров, включая LoRA, Prefix Tuning, P-Tuning и другие. Она легко интегрируется с экосистемой Hugging Face Transformers.

Документация: PEFT Documentation**Репозиторий:** GitHub Repository

Поддерживаемые модели: Широкий спектр моделей на основе трансформеров, включая GPT, Llama, T5, BERT и многие другие, доступные через библиотеку Transformers.

```
from peft import LoraConfig, get_peft_model
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "mistralai/Mistral-7B-v0.1"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# Определение конфигурации LoRA
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

# Применение LoRA к модели
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
```

bitsandbytes: Квантизация для эффективности

bitsandbytes: Квантизация для эффективности

bitsandbytes — это библиотека для оптимизации глубокого обучения, которая позволяет эффективно работать с моделями, квантованными в 8-битном или 4-битном формате. Она играет ключевую роль в QLoRA, обеспечивая обучение больших моделей с минимальным потреблением памяти на потребительских GPU, при этом сохраняя высокую точность.

Документация: bitsandbytes Documentation (GitHub)**Репозиторий:** GitHub Repository

Поддерживаемые модели: Совместима с любыми моделями PyTorch, которые могут использовать квантованные слои, особенно популярна для LLM.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer
from bitsandbytes.quantization import quantize_model_fp4

model_name = "meta-llama/Llama-2-7b-hf"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name, torch_dtype=torch.float16)

# Применение 4-битной квантизации
model = quantize_model_fp4(model)

# Теперь модель использует 4-битные веса, что значительно снижает потребление памяти.
print(model)
```

Hugging Face Transformers: Основа для LLM

Hugging Face Transformers: Основа для LLM

Хотя Transformers не является библиотекой PEFT напрямую, она является основой для большинства современных LLM и тесно интегрирована с PEFT и bitsandbytes. Она предоставляет готовые архитектуры моделей, предобученные веса, токенизаторы и инструменты для обучения, которые используются при тонкой настройке с LoRA/QLoRA.

Документация: Transformers Documentation**Репозиторий:** GitHub Repository

Поддерживаемые модели: Огромное количество предобученных моделей, включая BERT, GPT-2, GPT-J, Llama, T5, Falcon и многие другие, для различных задач NLP.

```
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from peft import LoraConfig, get_peft_model
import torch

model_name = "mistralai/Mistral-7B-v0.1" # Пример модели
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Загрузка модели с 4-битной квантизацией (для QLoRA)
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=False,
)
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto"
)

# Определение конфигурации LoRA (для PEFT)
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.05,
    bias="none",
    task_type="CAUSAL_LM"
)

# Применение LoRA к квантованной модели
model = get_peft_model(model, lora_config)
model.print_trainable_parameters()
```

Есть вопросы?

Полезные ресурсы:

- Документация PEFT: Подробные руководства по использованию библиотеки PEFT.
- Репозиторий PEFT на GitHub: Исходный код, примеры и активное сообщество разработчиков.
- Научные статьи о LoRA и QLoRA: Актуальные исследования и публикации по теме.
    - PEFT: Lialin et al., 2023, Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning
    - Prompt Tuning: Lester et al., 2021, The Power of Scale for Parameter-Efficient Prompt Tuning
    - Prefix Tuning: Li, Liang, 2021, Prefix-Tuning: Optimizing Continuous Prompts for Generation
    - Adapters: Houlsby et al., 2019, Parameter-Efficient Transfer Learning for NLP