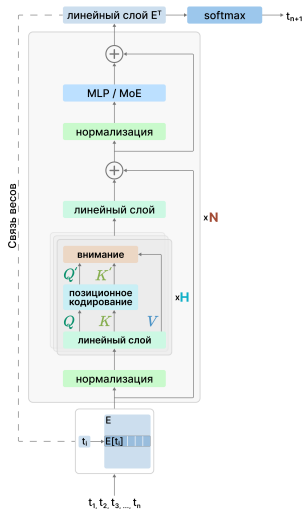


Эффективное дообучение

Юлиан Сердюк

14.10.2025

Hey everyone! :)

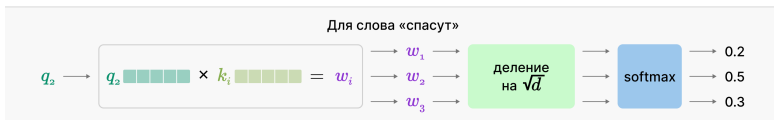


Ключевые компоненты

Следующие компоненты трансформеров являются ключевыми:

- 1 Механизмы внимания
- 2 Residual Connection
- 3 Нормализация
- 4 Функции активации
- 5 Полносвязные слои
- 6 Позиционное кодирование
- 7 Иногда выделяют энкодер и декодер
- 8 Эмбединг (которые используются дважды!)

Ванильное внимание



$$\text{Итого: } 0.2 \times \begin{matrix} v_1 \\ \vdots \end{matrix} + 0.5 \times \begin{matrix} v_2 \\ \vdots \end{matrix} + 0.3 \times \begin{matrix} v_3 \\ \vdots \end{matrix} = \begin{matrix} x'_2 \\ \vdots \end{matrix}$$

Механизм внимания используется для нахождения взаимосвязей между объектами.

Всё в той же статье¹ был предложен простой механизм внимания для трансформеров.

$$\text{attention}(Q, K, V) = \text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

Размерности d_q и d_k совпадают, а d_v может быть произвольной.

Матрицы Q , K , V получаются путем умножения эмбеддингов токенов на специальные обучаемые матрицы W^Q , W^K и W^V .

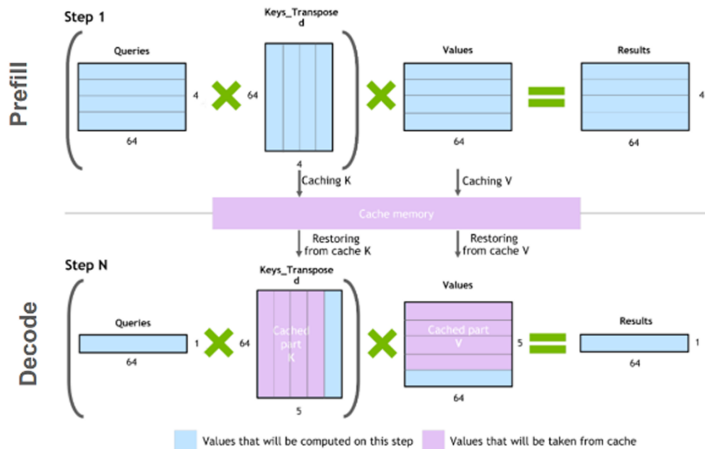
¹Vaswani, “Attention is all you need”.

Без кэша: на каждом шаге генерации пересчитываются все Q, K, V для всей последовательности \Rightarrow огромные накладные расходы.

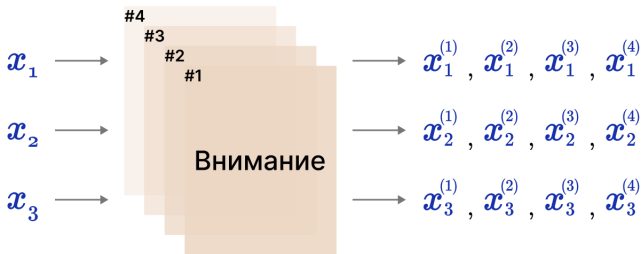
С KV-кэшем:

- Сохраняем K, V для уже сгенерированных токенов.
- На новом шаге вычисляем только Q и умножаем его на сохранённые K .
- Стоимость одного шага: $O(kd)$, где k — длина текущей последовательности.

Итого: за k шагов генерации сложность $O(k^2d)$. Асимптотика та же, но вычисления на каждом шаге сильно ускоряются.

$(Q * K^T) * V$ computation process with caching

Многоголовое внимание



Для x_i

The equation shows the linear transformation of the i -th input vector x_i . The input vector x_i is represented as a sequence of four vectors: $x_i^{(1)}, x_i^{(2)}, x_i^{(3)}, x_i^{(4)}$. This is multiplied by a weight matrix W^O , which is shown as a purple grid with four rows and four columns. The result is a new vector x'_i , represented as a sequence of four blue boxes.

$$\begin{bmatrix} x_i^{(1)} & x_i^{(2)} & x_i^{(3)} & x_i^{(4)} \end{bmatrix} \times \begin{bmatrix} \dots \\ W^O \end{bmatrix} = \begin{bmatrix} x'_i \end{bmatrix}$$

Механизм MultiHead Attention позволяет учитывать множество различных связей между токенами.

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1 \text{head}_2 \dots \text{head}_h) W^O,$$

$$\text{head}_i = \text{attention}(QW_i^Q, KW_i^K, VW_i^V).$$

Привет суровая реальность :(

И вот тут начинается грустная история – у нас есть огромная классная моделька, которую нам нужно дообучить под наши нужды. И, к сожалению, дообучить её мы не можем из-за нехватки ресурсов...

Few-Shot Learning

Идея: Модель не переобучается, а получает *несколько примеров прямо в промпте* — и по ним обобщает на новый случай.

Пример few-shot промпта

Переведи слово на французский:

```
-- cat → chat  
-- dog → chien  
-- bird → ?
```

Что делает модель:

- Распознаёт шаблон «английское слово → французский перевод»
- Строит внутреннее соответствие между примерами
- Генерирует ответ: bird → oiseau

→ Модель “учится на лету”, не обновляя веса.

Что есть дообучение

В статье² обосновывается идея, что использование промпта (в частности few-shot learning) сопоставимо с полным обучением SFT, просто без градиента и без обновления весов модели...

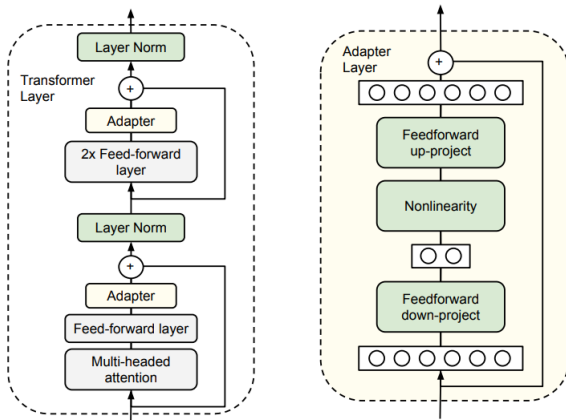
²Dherin и др., *Learning without training: The implicit dynamics of in-context learning*.

В статье³ была впервые описана идея добавления небольшого числа параметров к модели и дообучения только их. Метод был предложен для использования в области обработки изображений.

³Rebuffi, Bilen и Vedaldi, *Learning multiple visual domains with residual adapters.*

Адаптеры

В статье⁴ эта идея была впервые применена к трансформерам.



⁴Houlsby и др., *Parameter-Efficient Transfer Learning for NLP*.

Остаточные связи

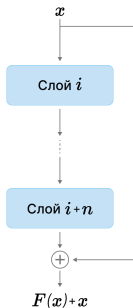
Зачем нужны остаточные связи?

$$x' = F(x) + x$$

Нейронная сеть
без остаточных связей



Нейронная сеть
с остаточными связями

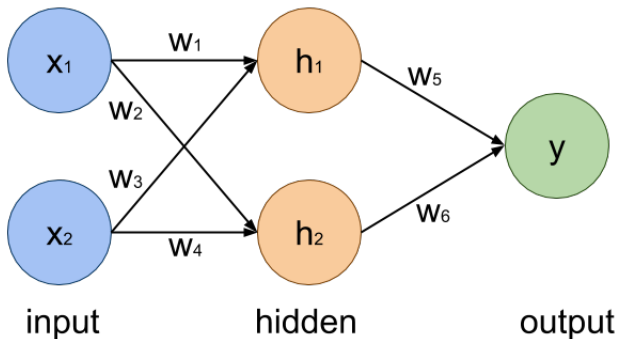


Ответ гугла:

Residual connections have provided a huge boost to networks' ability to overcoming vanishing gradients, and hence, to ever-deeper networks' expertise on the problem.

Остаточные связи

Также такие связи помогают избавиться от симметрии в оптимизационной задаче.



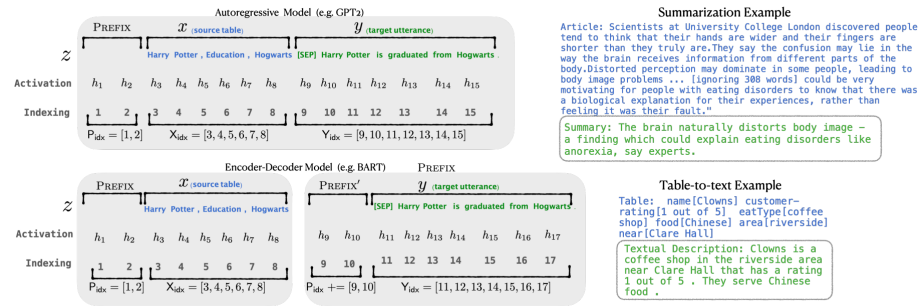
Структура адаптера (bottleneck)

$$x' = x + W_{\text{up}} \sigma(W_{\text{down}} x)$$

- $W_{\text{down}} \in \mathbb{R}^{r \times d}$ — понижение размерности ($r \ll d$)
- $W_{\text{up}} \in \mathbb{R}^{d \times r}$ — восстановление размерности
- $\sigma(\cdot)$ — нелинейность (обычно ReLU)
- Добавляется **residual** $h + \dots$ и near-identity инициализация

Prefix-tuning

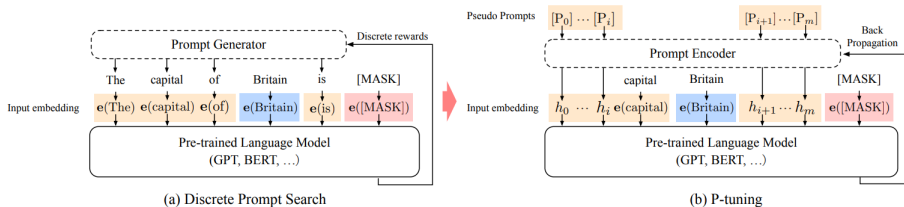
В статье⁵ была предложена альтернативная идея к дообучению модели. К каждой голове внимания добавлялись специальные токены.



⁵Li и Liang, *Prefix-Tuning: Optimizing Continuous Prompts for Generation*.

P-tuning

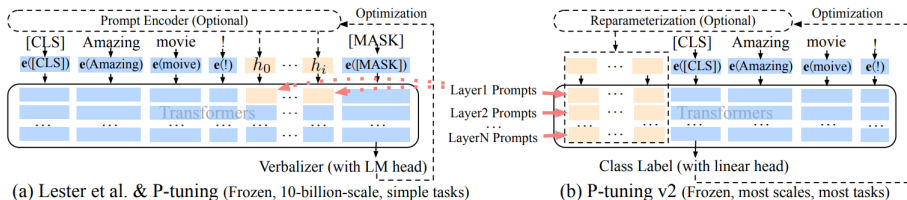
В другой статье⁶ Вектора p_i обучаются для каждого слоя отдельно. Авторы отметили, что обучение векторов p_i напрямую даёт нестабильные результаты, поэтому вместо обучения всех векторов p_i для всех слоёв.



⁶Liu и др., *GPT Understands, Too*.

P-tuning2

В другой статье⁷ эту идею доработали, добавив обучаемые токены к каждому слою.



⁷Liu и др., *P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks*.

Примерно в то же время появился альтернативный вариант обучения адаптеров⁸. Данная идея состояла в добавлении обучаемых весов к весам проекций W^Q, W^K, W^V .

$$\begin{matrix} n \\ \text{ } \\ m \\ W \end{matrix} = \begin{matrix} n \\ \text{ } \\ m \\ W_o \end{matrix} + \begin{matrix} r \\ n \dots \\ A \end{matrix} \begin{matrix} r & m \\ \text{ } & B \\ \left[\begin{matrix} \text{ } & \text{ } \\ \text{ } & \text{ } \\ \text{ } & \text{ } \end{matrix} \right] \\ n & m \\ AB \end{matrix}$$

⁸Hu и др., *LoRA: Low-Rank Adaptation of Large Language Models*.

LoRA: Low-Rank Adaptation of Large Language Models

Идея: вместо дообучения всех весов $\mathbf{W}_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, мы аппроксимируем их изменение с помощью низкорангового приращения:

$$\mathbf{W} = \mathbf{W}_0 + \alpha \Delta \mathbf{W}, \quad \Delta \mathbf{W} = \mathbf{B} \mathbf{A}, \quad \mathbf{A} \in \mathbb{R}^{r \times d_{\text{in}}}, \quad \mathbf{B} \in \mathbb{R}^{d_{\text{out}} \times r}.$$

Во время обучения:

- \mathbf{W}_0 заморожена, обучаются только \mathbf{A} и \mathbf{B} ;
- обычно одна из матриц \mathbf{A} или \mathbf{B} иницируется нулями;
- количество обучаемых параметров сокращается с $d_{\text{in}} d_{\text{out}}$ до $(d_{\text{in}} + d_{\text{out}})r$.

Проекция на подпространство ранга r захватывает основные направления градиентов. Плюс, дообучение никак не влияет на архитектуру модели.

В методе AdaLora⁹ предлагается оптимизировать ранги под каждый слой отдельно.

- Во время обучения оценивается **важность** каждого слоя: насколько изменение его LoRA-весов влияет на loss.

$$I_k = \|\nabla_{\text{LoRA}_k} \mathcal{L}\| \Rightarrow \text{«чувствительность слоя»}$$

- Слоям с высоким I_k выделяется **большой ранг** r_k ;
- Менее важные слои получают меньший ранг (их низкоранговое приближение ужимается);
- Общий бюджет параметров сохраняется:

$$\sum_k r_k = R_{\text{budget}}.$$

⁹Zhang и др., *AdaLoRA: Adaptive Budget Allocation for Parameter-Efficient Fine-Tuning*.


Идея LoRA была развита в QLoRA¹⁰. Одна из основных идей - двойное квантование модели и “деквантование” на лету.

Основной принцип:

- Веса модели разбиваются на блоки (обычно по 64 элемента);
- Каждый блок нормализуется и хранится как набор 4-битовых индексов q_i ;
- Для блока сохраняется *масштаб* s_b (scale), который восстанавливает диапазон значений;
- При прямом проходе выполняется **деквантование на лету**:

$$\widehat{w}_i = s_b \cdot \text{LUT}_{NF4}[q_i],$$

где LUT_{NF4} — фиксированная таблица из 16 нормализованных float-значений.

¹⁰Dettmers и др., *QLoRA: Efficient Finetuning of Quantized LLMs*. 

Double Quantization:

- Даже масштабы s_b сами квантуются в 8 бит:

$$s_b \approx s_{\text{scale}} \cdot t_{pb}, \quad t_{pb} \in \text{LUT}_{8b};$$

- Ещё больше снижает память (20–30%) при почти неизменной точности.

Итого:

$$\widetilde{\mathbf{W}} = \text{DeQuant}(\text{Quant}(\mathbf{W}_0)) + \alpha \mathbf{B}\mathbf{A},$$

LoRA Without Regret

Статья¹¹ дает несколько практических советов по использованию LoRA.

- LoRA хорошо при умеренном размере и достаточном r . Но на сложных задачах SFT лучше.
- LoRA полезна не только для Attention (W_q, W_v), но и для MLP-слоёв.
- Обучение LoRA устойчиво при большем LR, чем SFT.
- При больших batch size точность немного снижается (эффект усреднения градиентов на малом числе параметров).
- LoRA можно подключать и менять “на лету”

¹¹Schulman и Lab, “LoRA Without Regret”.

В работе WeightLoRA¹² была дополнительно развита идея оценки важности адаптеров.

- Добавляем обучаемый коэффициент ω_i для каждого слоя:

$$h_i(x) = W_i x + \omega_i (B_i A_i x)$$

- Во время обучения ω_i показывает *насколько адаптер слоя важен*;
- Оцениваем нормализованную важность:

$$s_i = |\omega_i| \cdot \frac{\|B_i A_i\|_F}{\sum_j \|B_j A_j\|_F}$$

- Выбираем **топ-К адаптеров по s_i** — остальные отключаем ($\omega_i = 0$).

¹²Veprikov и др., *WeightLoRA: Keep Only Necessary Adapters*

Скоро можно будет выдохнуть

Конечно, LoRA не единственный способ обучить модель с ограниченным набором памяти.

Gradient Accumulation

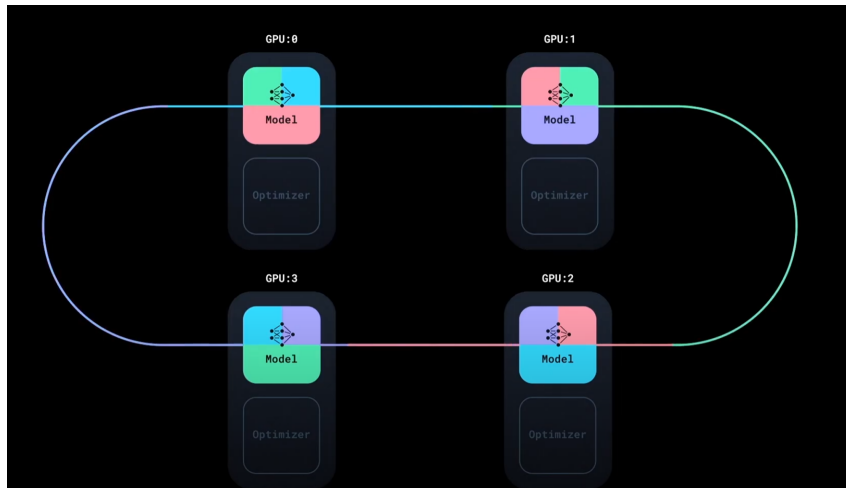
Когда GPU не помещает большой batch, можно разбить его на несколько микробатчей. Для каждого микробатча считаем градиенты, но не делаем шаг оптимизатора.

Формула итогового градиента считается как

$$g = \frac{1}{N} \sum_{i=1}^N \nabla_{\theta} \mathcal{L}_i \Rightarrow \text{оптимизация после } N \text{ микробатчей.}$$

- Работает совместно с DDP/FSDP;
- Увеличивает стабильность, но требует аккуратного скейлинга learning rate.

Gradient Accumulation



Gradient Checkpointing

Gradient Checkpointing (activation checkpointing) – хранение только части активаций и пересчет их во время обратного прохода.

Применение: `torch.utils.checkpoint` или
`model.gradient_checkpointing_enable();`

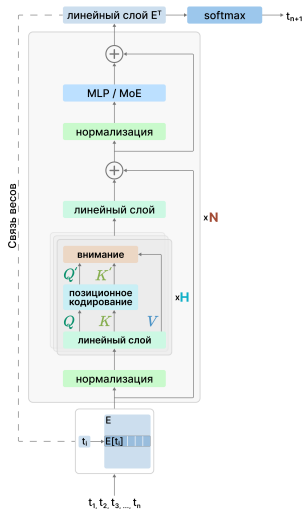
Не все тензоры обязаны храниться в GPU-памяти, некоторые можно **оффлоадить** часть данных на CPU или NVMe-диск.

Виды оффлоада:

- **Parameter Offload** — веса (при FSDP, DeepSpeed Zero-3);
- **Optimizer Offload** — моменты оптимизатора на CPU/NVMe;
- **Activation Offload** — активации (редко, дорого).

Также нужно учесть что слои можно подгружать по очереди...

Offloading



Ура! :)

В? П? П?