Записки научных
семинаров ПОМИ
Том 540, 2024 г.

В. М. Ломшаков, С. И. Николенко

ПРИМЕНЕНИЯ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ ДЛЯ ЗАДАЧ ПОРОЖДЕНИЯ И ОБРАБОТКИ ПРОГРАММНОГО КОДА

§1. Введение

В последние годы развитие больших языковых моделей (large language models, LLM) [13, 71, 74, 138] привело к значительным изменениям в различных областях, включая автоматизацию порождения программного кода. Эти модели, как семейства LLM общего назначения — семейства GPT от OpenAI [11,18,78,80], Claude от Anthropic [4,5] и Gemini от Google [106], — так и специализированные решения для кода, в частности Qwen-Coder [47] и DeepSeek-Coder [26], демонстрируют высокую способность к порождению кода, интерпретации инструкций и исправлению ошибок. Однако достижение такого уровня производительности требует применения сложных методов обучения и дообучения, таких как обучение с подкреплением на основе человеческих предпочтений (RLHF), адаптивное дообучение (PEFT), промптинг[1] (prompting) и других подходов.

Цель настоящего обзора — систематизировать текущие методы обучения и применения больших языковых моделей для порождения программного кода, рассмотреть их преимущества, ограничения и основные направления развития. Мы сосредоточимся на наиболее перспективных и активно исследуемых подходах дообучения, методах, направленных на эффективное использование вычислительных ресурсов, а

Ключевые слова: большие языковые модели, порождение программного кода, обучение с подкреплением, обучение следованию инструкциям, LLM-агенты.

Работа была поддержана грантом Российского Научного Фонда #22-11-00135, «Исследование и разработка технологий обработки и анализа мультимодальных неструктурированных данных из различных источников и их применимости для решения экономических и социальных задач».

[1] Здесь, как и в некоторых других случаях, придётся просить прощения у адмирала Шишкова — конечно, слово “prompt” по-русски можно выразить словом “запрос” или “вход модели”, но всё это более общие понятия, а для целей настоящего обзора и современного машинного обучения в целом полезно иметь отдельный термин, обозначающий именно текстовые запросы к языковым моделям.

также обсудим основные наборы данных для оценки кодовых моделей.

Большие языковые модели (LLM) можно адаптировать к решению конкретных задач через дообучение базовых моделей (foundational models), которые были перед этим предобучены на гигантских неразмеченных (точнее, автоматически размеченных — например, задачей языкового моделирования, т.е. предсказания следующего токена в тексте) датасетах. Такой подход позволяет интегрировать их в реальные сценарии программирования: от автодополнения кода до автоматизированного исправления ошибок. В настоящем обзоре мы акцентируем внимание на том, как оптимизировать взаимодействие с моделями, выбирая правильные методы дообучения и оценки для прикладных целей. Важно отметить, что наш анализ направлен на использование LLM в прикладных задачах, где необходимо не только порождать код, но и учитывать его производительность, корректность и адаптируемость. Мы сосредоточились на подходах, которые обеспечивают максимальную гибкость и минимальные затраты на ресурсы, включая промптинг, адаптивные методы обучения и RLHF.

Отметим, что основным компонентом любой такой системы всегда будет оставаться базовая модель, и в последние годы исследования в области архитектурных улучшений трансформеров и их масштабирования — в частности, исследования о законах масштабирования (scaling laws), т.е. о том, как размер модели влияет на её производительность [21, 40, 52] — достигли значительных результатов и продолжают оставаться центральной темой всего современного искусственного интеллекта [138]. Однако эти аспекты выходят за рамки настоящего обзора, так как они в основном касаются базовых моделей, а не их применения для порождения кода; здесь мы предполагаем, что большая языковая модель уже существует и предобучена как минимум на задаче предсказания следующего токена в тексте.

Вместо этого мы сосредоточились на более практических аспектах: как адаптировать уже существующие архитектуры к специфическим задачам программирования. Это отражает нашу цель — предоставить прикладное руководство по использованию существующих моделей, а не обсуждение фундаментальных улучшений в их устройстве. Кроме самих методов дообучения (fine-tuning) и эффективной адаптации больших моделей (parameter-efficient fine-tuning, PEFT), мы рассматриваем датасеты для дообучения и оценки качества моделей, работающих с программным кодом, а также приводим примеры передовых современных моделей с открытыми весами (open weights).

Обзор организован следующим образом. В разделе 2 мы даём краткий общий обзор методов дообучения больших языковых моделей в общем виде, без конкретных применений к порождению программного кода. Раздел 3 кратко описывает основные наборы данных для оценки качества моделей порождения кода и метрики, которые в этой оценке используются. Раздел 4 рассматривает порождение кода на основе методов обучения с подкреплением, как правило, вариантов обучения на основе человеческих предпочтений (RLHF). В разделе 5 мы поговорим о методах дообучения с учителем, в частности дообучения следованию инструкциям (instruction tuning) на основе датасетов, в которых задачи порождения или исправления кода представлены как наборы промптов к LLM и ответов на них. А в разделе 6 мы обсудим, как лучше строить такие промпты: какой именно контекст нужно предоставить модели для наилучших результатов в порождении кода. Раздел 7 кратко рассматривает два других важных класса методов: сэмплирование большого числа порождений с верификацией и интерактивное взаимодействие со средой разработки при помощи основанных на LLM агентов. Наконец, в разделе 8 мы приводим обзор возможностей некоторых современных индустриальных решений, основанных на LLM, а раздел 9 содержит заключение и основные выводы обзора.

§2. Обзор методов дообучения больших языковых моделей

2.1. Обучение с подкреплением на основе человеческих предпочтений. Метод *обучения с подкреплением на основе человеческих предпочтений* (RLHF) ведёт начало с работы исследователей из компании OpenAI [18]. Первый автор этой работы, Пол Кристиано (Paul Christiano), является одной из ведущих фигур в области разработки дружественного искусственного интеллекта (AI alignment), и эта работа была также мотивирована задачей, связанной с согласованием: как объяснить, например, роботу, чего именно мы от него хотим? Если мы не находимся в строго формальной системе, например в игре в шахматы, то любая функция награды, которую мы могли бы сформулировать в реальном мире, может быть поверхностно оптимизирована

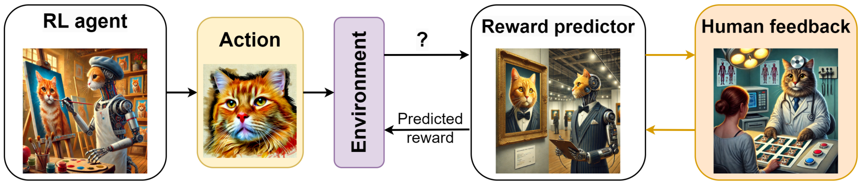


Рис. 1. Глубокое обучение с подкреплением на основе человеческих предпочтений [18]: агент обучения с подкреплением действует в среде, где награды определяются отдельной моделью, обученной с использованием человеческой обратной связи.

способами, которые трудно предсказать, но которые не дают желаемого результата [53]. Например, хорошо известно, что роботы, обучающиеся сложному поведению в моделируемых средах, часто узнают больше об ошибках и вычислительных ограничениях симулятора, чем о желаемом поведении в реальном мире.

Таким образом, в работе [18] делается следующее предположение: поскольку мы, скорее всего, не можем формально определить, чего мы хотим, лучши спросить человека напрямую. Человеческая обратная связь будет определять, насколько текущее поведение системы соответствует конечной цели, которую люди понимают, но формально выразить не могут; эта обратная связь может предоставляться в форме сравнения двух ответов или двух результатов и выбора одного из них. Однако такой подход неприменим для полного обучения: мы не можем попросить людей разметить столько данных, сколько необходимо для обучения большой языковой модели с подкреплением. Поэтому идея работы [18] заключается в обучении отдельной модели, которая кодирует предпочтения пользователя и предсказывает награду, используемую для реального обучения с подкреплением. Общая схема этого обучения представлена на рис. 1.

Человек, предоставляющий обратную связь, не может назначить числовое значение награды, поэтому вместо этого используется подход, который сравнивает пары «действий» — здесь важным также является определение одного действия, например, в работе [18] действиями были короткие последовательности игры в Atari или ходьбы робота — и устанавливает предпочтения между парами. В результате получается набор данных, представленный парами

$$D = \{(\sigma_1, \sigma_2, \mu)\}_{n=1}^{N},$$

где

$$\sigma_i = ((o_{i0}, a_{i0}), (o_{i1}, a_{i1}), \ldots, (o_{i,k_i}, a_{i,k_i}))$$

представляют собой последовательности пар наблюдение-действие, описывающих траекторию в среде обучения с подкреплением, а μ — это распределение вероятностей, определяющее, предпочитает ли пользователь σ_1 или σ_2 (или он индифферентен, что будет выражено равномерным μ).

Для преобразования парных предпочтений в функцию награды этот подход использует предположения *моделей Брэдли–Терри*, предназначенных для обучения функции ранжирования на основе парных предпочтений [10]. Основное предположение таких моделей заключается в том, что вероятность победы игрока i над игроком j может быть смоделирована как

$$\hat{p}\,(i \succ j) = \frac{\gamma_i}{\gamma_i + \gamma_j}, \tag{1}$$

где $\gamma_i, \gamma_j \in \mathbb{R}$ — значения рангов. Модели Брэдли–Терри предлагают алгоритмы для максимизации полной правдоподобности набора данных с такими парными сравнениями, обычно основанные на алгоритмах минимизации-максимизации, которые являются обобщением основной идеи EM-алгоритма [48].

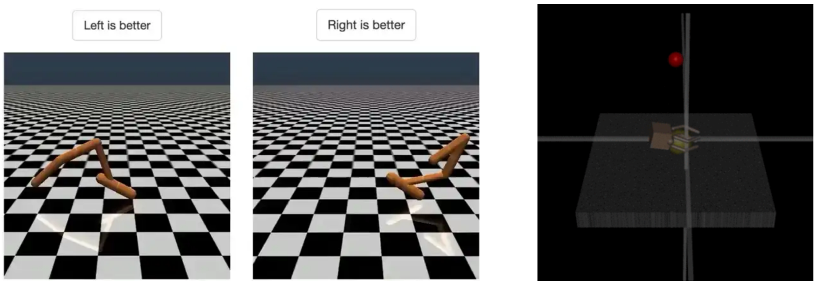
В случае RL на основе человеческих предпочтений требуется дополнительное предположение, так как γ_i должна быть функцией σ_i; в [18] это моделируется как произведение экспоненциальных наград по последовательности:

$$\gamma(\sigma_i) = e^{\sum_{t=1}^{k_i} \hat{r}(o_{it}, a_{it})}, \tag{2}$$

а функция потерь для нейронной сети может быть определена как

$$\mathcal{L} = -\sum_{(\sigma_1, \sigma_2, \mu) \in D} \left(\mu(1) \log \hat{p}\,(\sigma_1 \succ \sigma_2) + \mu(2) \log \hat{p}\,(\sigma_2 \succ \sigma_1)\right). \tag{3}$$

Может показаться, что эта идея просто переносит непрактичную задачу предоставления человеческой обратной связи во время обучения с подкреплением на столь же непрактичную задачу предоставления достаточного объема обратной связи для обучения модели предсказания награды. Однако оказалось, что с этим подходом требуется всего несколько сотен запросов к человеку, чтобы научить модель ходить



(а) (б)

Рис. 2. Глубокое обучение с подкреплением на основе человеческих предпочтений в симуляторе MuJoCo [18]: (а) пример выбора, сделанного человеком; (б) пример нежелательной манипуляции спецификацией.

или прыгать в моделируемой среде MuJoCo [107] (см. пример выбора, представленного для оценки человеком, на рис. 2а), а если вы готовы выполнить более 1000 запросов, то можете даже получить результаты лучше, чем чистое обучение с подкреплением! Этот эффект, вероятно, связан с *формированием награды* [121]: когда мы, люди, оцениваем поведение, мы устанавливаем порядок, в котором последовательности, ближе к цели, оцениваются выше, а полученные награды предоставляют агенту больше информации, чем просто бинарная метка того, выполнена ли задача успешно.

Кстати, эта работа также содержит интересный пример того, как обучение с подкреплением может пойти по неверному пути. На рис. 2b показан пример кадра из видео, где роботизированная рука пытается захватить мяч. Человеческие эксперты должны были определить, был ли захват успешным. Однако, поскольку в сцене использовалась только одна виртуальная камера, а на таком однородном фоне людям было сложно оценить глубину, робот научился располагать руку *между* мячом и камерой, чтобы людям казалось, как будто он захватывает мяч, хотя на самом деле это было не так! Это отличный пример так называемых *манипуляций спецификацией* (specification gaming), когда модели машинного обучения сходятся к поведению, которое не было задумано разработчиками, но которое действительно оптимизирует указанную ими целевую функцию; это одна из центральных проблем AI alignment.

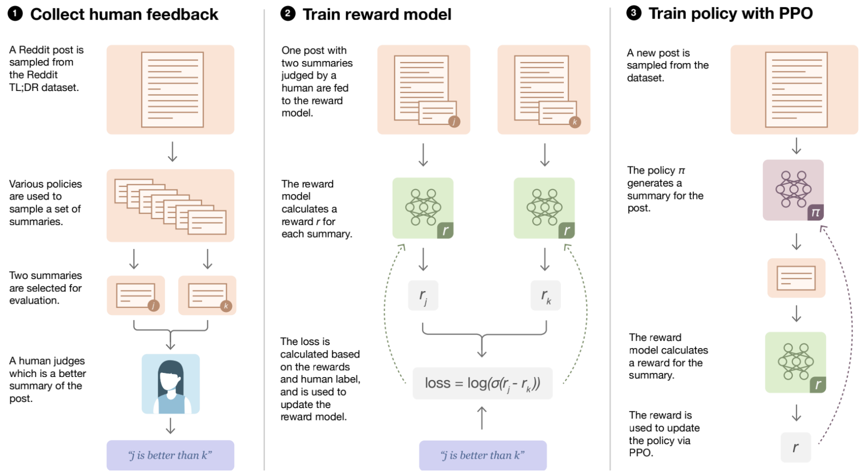


Рис. 3. Обучение резюмированию с использованием человеческой обратной связи [103].

Идеи работы [18] были продолжены и развиты в целом ряде дальнейших публикаций. В частности, были разработаны расширения для k-мерных сравнений с специально разработанным методом максимального правдоподобия [144], для неопределенной обратной связи, когда человеческий эксперт может надежно различать только те образцы, чье качество существенно отличается [12], а также для мультиагентных систем [119]. С другой стороны, это направление исследований можно рассматривать как раздел *обучения с подкреплением на основе предпочтений* (PbRL), где значения наград заменяются предпочтениями [32, 49, 123, 125].

Но наиболее важным продолжением, конечно, стало само RLHF, применение глубокого обучения с подкреплением на основе человеческих предпочтений к большим языковым моделям. Первый шаг был сделан в 2020 году, когда исследователи OpenAI в работе [103] создали модель резюмирования (summarization), основанную на человеческой обратной связи. Их подход, показанный на рис. 3, очень похож на [18]: они собирают человеческую обратную связь о том, какие резюме документов лучше, обучают модель награды (reward model) так, чтобы соответствовать этим предпочтениям, а затем используют модель награды для дообучения с использованием обучения с подкреплением. Для обучения модели награды они изменили функцию потерь (3) на функцию потерь классификации, основанную на логистическом сигмоиде:

$$\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y}_1,\boldsymbol{y}_2,\mu)\in D} \log\left(\sigma\left(\hat{r}(\boldsymbol{x},\boldsymbol{y}_\mu) - \hat{r}(\boldsymbol{x},\boldsymbol{y}_{1-\mu})\right)\right), \tag{4}$$

где $\boldsymbol{y}_1$ и $\boldsymbol{y}_2$ — это два резюме текста $\boldsymbol{x}$, а $\mu \in \{0,1\}$ показывает, какое из них предпочёл пользователь. Для обучения с подкреплением авторы использовали *проксимальную оптимизацию стратегии* (proximal policy optimization, PPO), стандартный алгоритм из обучения с подкреплением, осуществляющий эффективное обучение методом градиента по стратегии (policy gradient). Подробное описание этого метода выходит за рамки настоящего обзора; см., например, [95, 104, 140].

Важно отметить, что если процесс обучения с подкреплением оставить без контроля, он, скорее всего, будет склонен к переобучению, значительно отклонится от исходной модели и приведет к коллапсу в одной точке, так как человеческая обратная связь, конечно же, слишком скудна для полноценного обучения. Поэтому RLHF добавляет штрафной член в функцию награды $\hat{r}(\boldsymbol{x},\boldsymbol{y})$, который побуждает обучаемую стратегию π_{RL} не слишком сильно отличаться от исходной модели с учителем π_{SFT}, обычно в форме дивергенции Кульбака–Лейблера между ними:

$$\hat{r}'(\boldsymbol{x},\boldsymbol{y}) = \hat{r}(\boldsymbol{x},\boldsymbol{y}) - \beta \log\left(\pi_{\mathrm{RL}}\left(\boldsymbol{y}|\boldsymbol{x}\right)/\pi_{\mathrm{SFT}}\left(\boldsymbol{y}|\boldsymbol{x}\right)\right). \tag{5}$$

Настоящая революция в больших языковых моделях произошла, когда исследователи OpenAI в работе [80] применили это направление исследований непосредственно к большим языковым моделям. Их целью было сделать модели семейства GPT-3 [11] полезными и дружественными к пользователю. Проблема заключалась в том, что по умолчанию большая языковая модель обучается просто предсказывать токены, предоставлять правдоподобное продолжение текстового потока. Она «не пытается» быть полезной, неоскорбительной или даже правдивой, потому что такие продолжения, как ложь, уклонение от ответа или перевод разговора на другую тему, могут в принципе оказаться не менее правдоподобными с точки зрения обучающего набора (который для больших языковых моделей стремится включить

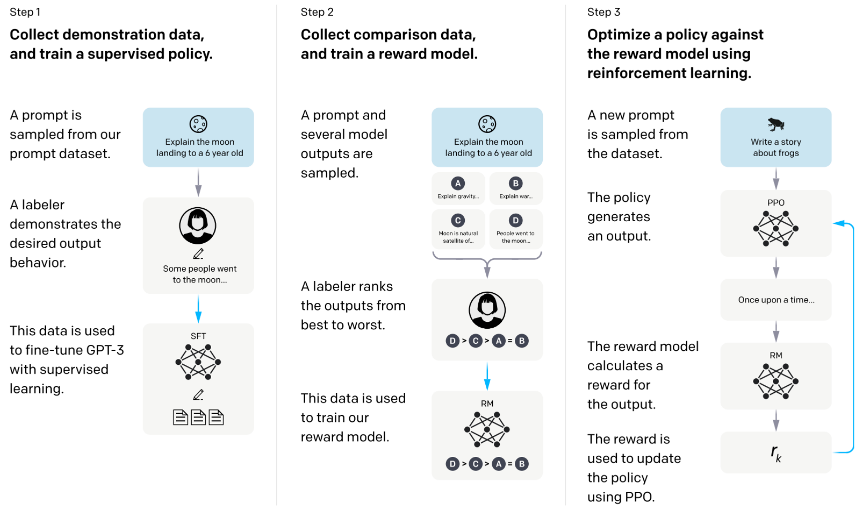


Рис. 4. RLHF, превращающий GPT-3 в InstructGPT [80].

все значимые тексты, собранные из интернета), чем честные и полные ответы на вопрос пользователя.

Поэтому в работе [80] метод RLHF, описанный выше, был применен к выходным данным большой языковой модели; общая структура этого подхода, показанная на рис. 4, очень похожа на RLHF для резюмирования, представленное на рис. 3. На этот раз человеческие эксперты должны были определить, какие из выходных данных модели являются наиболее полезными, наименее оскорбительными и наиболее правдивыми. Получившаяся языковая модель InstructGPT показала значительное улучшение в правдивости, уменьшение токсичности, более точное следование инструкциям и явным ограничениям в запросе; улучшения также оказались достаточно устойчивыми и способными к обобщению. Отметим здесь особо, что они обобщались не только на языки, отсутствовавшие в наборе данных человеческой обратной связи, но даже на языки другой природы, чем естественные, в том числе на такие задачи с программным кодом, как суммаризация и ответы на вопросы по коду [67, 80].

После InstructGPT оставался лишь небольшой шаг до ChatGPT, который вышел в ноябре 2022 года [77] и положил начало современной волне исследований и практических применений больших языковых моделей. В следующем разделе мы рассмотрим другие способы дообучения больших языковых моделей.

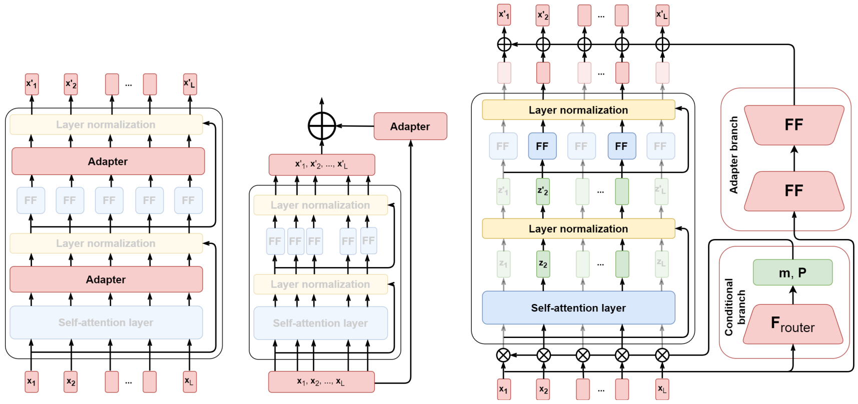
2.2. Эффективное по параметрам дообучение: аддитивные и селективные подходы. RLHF требует активной обратной связи от человека и в основном используется для целей, которые трудно определить формально, например для того, чтобы сделать большую языковую модель «более полезной».

Но на практике часто возникают ситуации, когда большая модель машинного обучения, уже предварительно обученная с использованием огромного вычислительного бюджета, должна быть дообучена для некоторой хорошо определённой задачи, для которой имеется значительно меньший доступный набор данных; такие ситуации постоянно возникают в том числе и в задачах обработки исходного кода программ. В таких случаях задача состоит в том, чтобы внести такие изменения в параметры модели, которые можно было бы обучить на небольшом наборе данных с вычислительным бюджетом на порядки меньше, чем для изначального обучения.

Этот набор методов известен как *эффективное по параметрам дообучение* (parameter-efficient fine-tuning, PEFT) [38]. На сегодняшний день это уже обширная область исследований со множеством различных подходов; здесь мы рассмотрим лишь некоторые из наиболее важных для дальнейшего изложения.

Первая категория методов, *адаптивное* PEFT, вводит некоторый новый механизм в нейронную архитектуру. Бóльшая часть весов остаётся «замороженной» (frozen), но новый механизм, называемый *адаптером* (adapter), способен модифицировать внутренние представления даже с использованием небольшого числа весов. Основное различие между методами заключается в том, где именно разместить этот новый механизм.

Адаптивное PEFT началось с работы [42], где адаптер был введён внутрь модуля трансформера [112]. В [42] адаптер размещается в двух местах: после слоя самовнимания и после слоёв прямого распространения в модуле кодировщика, оба раза перед нормализацией слоя с остаточным соединением вокруг него. На рис. 5a базовая структура модуля кодировщика трансформера показана серым, а цвет показывает, куда помещается адаптер. Сам адаптер также представляет собой



(a) (б)

Рис. 5. Адаптивное PEFT: (a) последовательный адаптер в модуле кодировщика Transformer [42]; (b) параллельный адаптер [39]; (c) условный адаптер (CoDA) [54].

миниатюрную архитектуру кодировщика–декодировщика с «бутылочным горлышком» (bottleneck) посередине: одна матрица $W_{\downarrow}$ проецирует входные вложения в пространство меньшей размерности, а другая матрица расширяет их обратно с нелинейностью h, применяемой поэлементно посередине. Формально говоря,

$$\mathrm{Adapter}(\boldsymbol{x}) = W_{\uparrow} \cdot h\left(W_{\downarrow}\boldsymbol{x}\right). \tag{6}$$

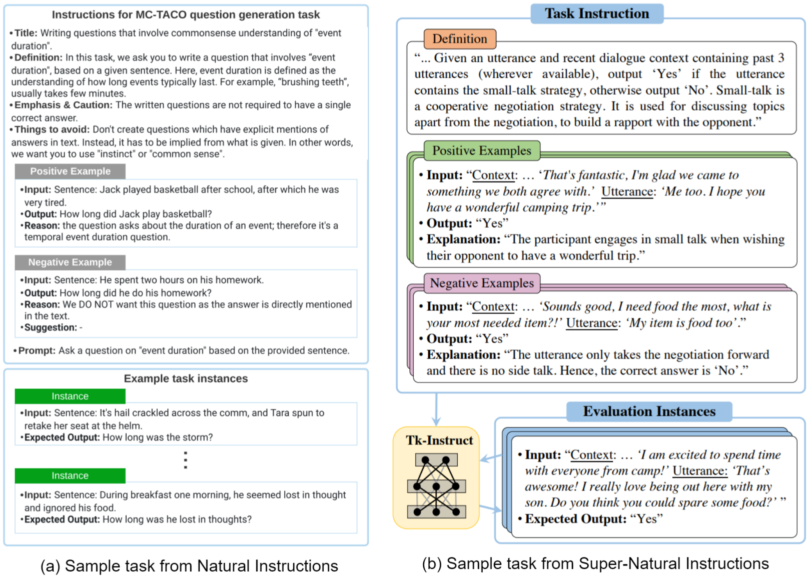
Эта архитектура содержит так называемый *последовательный адаптер*, так как новые веса подключаются последовательно внутри блоков трансформера. Альтернативой может быть использование подхода *параллельного адаптера*, где новые веса образуют побочную сеть, подключённую параллельно, как в работе [39], где эта архитектура также включает адаптер с бутылочным горлышком, определённым в (6), но на этот раз он работает параллельно с блоком кодировщика, как показано на рис. 5б. Параллельные адаптеры далее разрабатывались, например, в работе [145].

Интересное дальнейшее развитие метода было предложено в подходе *условных адаптеров* (conditional adapters, CoDA) [54]. Этот подход использует те же параллельные адаптеры, что и [39], но также стремится улучшить производительность ***вывода*** (inference), отмечая, что для обработки конкретного запроса не требуется вся большая предварительно обученная модель. В этом подходе параллельный адаптер (рис. 5б) состоит из двух ветвей. Адаптерная ветвь, аналогично [39], представляет собой сеть прямого распространения с бутылочным горлышком посередине. Условная ветвь, с другой стороны, нацелена на сокращение объёма вычислений, необходимых от предварительно обученных замороженных слоёв трансформера. Она начинается с функции маршрутизации F_{router}, которая выбирает k «лучших» токенов из входных L токенов:

$$F_{\mathrm{router}}(X) = (\boldsymbol{m}, P)\,, \tag{7}$$

где $P \in \{0,1\}^{k\times L}$ — матрица векторов с одной единицей (one-hot кодирование), показывающих, какие токены были выбраны, а $\boldsymbol{m} \in [0,1]^L$ — весовая маска для выбранных токенов; F_{router} реализован через оператор ***мягкого выбора top-k*** (soft top-k), который является обобщением функции softmax, естественного «мягкого выбора top-1», для $k > 1$ [54]. Затем k выбранных векторов используются в стандартном слое самовнимания, либо друг с другом (k-to-k внимание), либо как запросы к оригинальным L векторам (k-to-all внимание). В результате вычисления, требуемые от замороженных слоёв трансформера, значительно сокращаются (ещё больше в случае k-на-k внимания, хотя результаты, как правило, лучше при k-to-all внимании), а функция маршрутизации с весами, обученными в процессе дообучения, позволяет выбрать токены, наиболее соответствующие поставленной задаче. По сути, CoDA выбирает только часть токенов для обработки на каждом слое самовнимания.

С помощью адаптеров одну предварительно обученную модель на основе трансформера можно адаптировать к различным задачам, при этом обучая только веса модулей адаптеров и оставляя большую часть предобученных весов замороженной. Однако передача информации между различными задачами затруднена из-за явления, известного как ***катастрофическое забывание*** (catastrophic forgetting): когда модель обучается на новой задаче, она, как правило, теряет знания, полученные из предыдущих задач, так что вместо совместного использования информации просто получается модель для новой задачи. Этот эффект известен по меньшей мере с 1980-х годов [31, 70] и до сих пор остаётся серьёзной проблемой в глубоком обучении [3, 86]; было



Instructions for MC-TACO question generation task

- **Title:** Writing questions that involve commonsense understanding of "event duration".
- **Definition:** In this task, we ask you to write a question that involves "event duration", based on a given sentence. Here, event duration is defined as the understanding of how long events typically last. For example, "brushing teeth", usually takes few minutes.
- **Emphasis & Caution:** The written questions are not required to have a single correct answer.
- **Things to avoid:** Don't create questions which have explicit mentions of answers in text. Instead, it has to be implied from what is given. In other words, we want you to use "instinct" or "common sense".

Positive Example

- **Input:** Sentence: Jack played basketball after school, after which he was very tired.
- **Output:** How long did Jack play basketball?
- **Reason:** the question asks about the duration of an event; therefore it's a temporal event duration question.

Negative Example

- **Input:** Sentence: He spent two hours on his homework.
- **Output:** How long did he do his homework?
- **Reason:** We DO NOT want this question as the answer is directly mentioned in the text.
- **Suggestion:** -

- **Prompt:** Ask a question on "event duration" based on the provided sentence.

Example task instances

Instance

- **Input:** Sentence: It's hail crackled across the comm, and Tara spun to retake her seat at the helm.
- **Expected Output:** How long was the storm?

⋮

Instance

- **Input:** Sentence: During breakfast one morning, he seemed lost in thought and ignored his food.
- **Expected Output:** How long was he lost in thoughts?

(a) Sample task from Natural Instructions

Task Instruction

Definition

"... Given an utterance and recent dialogue context containing past 3 utterances (wherever available), output 'Yes' if the utterance contains the small-talk strategy, otherwise output 'No'. Small-talk is a cooperative negotiation strategy. It is used for discussing topics apart from the negotiation, to build a rapport with the opponent."

Positive Examples

- **Input:** "Context: … *'That's fantastic, I'm glad we came to something we both agree with.'* Utterance: *'Me too. I hope you have a wonderful camping trip.'*"
- **Output:** "Yes"
- **Explanation:** "The participant engages in small talk when wishing their opponent to have a wonderful trip."

Negative Examples

- **Input:** "Context: … *'Sounds good, I need food the most, what is your most needed item?!'* Utterance: *'My item is food too'*."
- **Output:** "Yes"
- **Explanation:** "The utterance only takes the negotiation forward and there is no side talk. Hence, the correct answer is 'No'."

Evaluation Instances

- **Input:** "Context: … *'I am excited to spend time with everyone from camp!'* Utterance: *'That's awesome! I really love being out here with my son. Do you think you could spare some food?'* "
- **Expected Output:** "Yes"

(b) Sample task from Super-Natural Instructions

(а) (б)

Рис. 6. Многозадачность с адаптерами: (а) *AdapterFusion* объединяет выходы нескольких адаптеров, обученных на отдельных задачах [85]; (б) *HyperAdapter* создаёт вход для гиперсети, которая порождает необходимый адаптер [135].

разработано несколько различных классов подходов для его смягчения [24, 69, 81, 90], но он всё ещё мешает проектам, связанным, например, с *пожизненным обучением* (lifelong learning) нейронных сетей, где основной целью является обучение различным задачам в разные моменты времени [82].

AdapterFusion, представленный в работе [85], решает проблему катастрофического забывания, предлагая не только использовать несколько различных адаптеров, обученных для различных задач, но и объединять результаты этих адаптеров, как показано на рис. 6а.

Отдельные адаптеры обучаются либо отдельно для каждой задачи, либо параллельно с функцией потерь, которая комбинирует все задачи и использует для них разные адаптеры, как это было предложено ранее в [102]. Затем мы фиксируем как предварительно обученные веса, так и отдельные адаптеры и обучаем веса модуля объединения адаптеров, снова используя данные из отдельных задач; таким образом, этапы извлечения знаний в адаптерах и объединения знаний в модуле объединения разделены, что помогает справляться с катастрофическим забыванием. С архитектурной точки зрения, модуль объединения адаптеров снова является слоем самовнимания, где ключи и значения берутся из адаптеров, а запросы извлекаются из получаемых на вход вложений.

Другой подход состоит в том, чтобы *составить* адаптер на основе задачи. Характерным примером здесь является *гиперадаптер на основе прототипов* (prototype-based hyperadapter, PHA) [135], который использует подход *гиперсетей* (hypernetworks) [37], то есть нейронных сетей, которые выводят параметры для других нейронных сетей. Как показано на рис. 6б, HyperAdapter состоит из двух основных частей:

- выходы кодировщика поступают в специально обученный модуль *ретривера* (retriever), который обучается проецировать входные данные в латентное пространство; ретривер обучается с использованием контрастивной функции ошибке (noise-contrastive estimation, NCE) [111] и, в частности, прототипического контрастивного обучения (prototypical contrastive learning, PCL) [57], чтобы кластеризовать экземпляры одной и той же задачи в латентном пространстве, одновременно отдаляя экземпляры разных задач друг от друга;
- PCL также обучает прототипы (аналогичные центроидам кластеров), которые нацелены на выделение «сути» каждой задачи в латентном пространстве; эти прототипы служат входными данными для гиперсети, которая создаёт параметры для модуля адаптера; адаптер здесь снова состоит из двух слоёв прямого распространения с бутылочным горлышком, как в [42], но веса теперь создаются гиперсетью;
- во время вывода новый экземпляр проходит через тот же замороженный энкодер и обученный модуль ретривера; затем ближайший прототип в латентном пространстве используется для порождения весов адаптера через гиперсеть, а адаптер применяется к декодировщику.

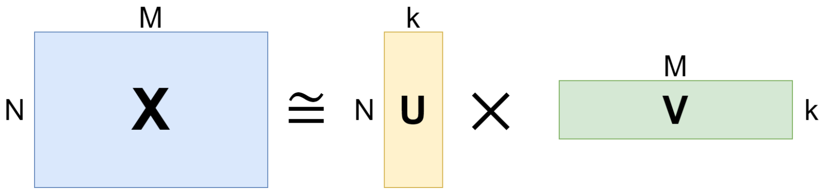


Рис. 7. Низкоранговые аппроксимации.

Можно также объединить несколько различных адаптеров в *смешанную экспертную модель* (mixture of experts, MoE) [75,130]. В частности, *Adamix* [115] обучает несколько матриц проекций вниз и вверх, а затем выбирает одну из каждой случайным образом во время вывода. *DEMix* [36] обучает коллекцию экспертных сетей прямого распространения и слой, который учитывает домен ввода во время обучения и оценивает его во время вывода. Работа [19] строит иерархию доменов в виде дерева с набором весов адаптера, связанных с каждым узлом. *AdapterSoup* [20] обучает несколько адаптеров и усредняет веса адаптеров, которые наиболее связаны с новым доменом, тем самым расширяя PEFT на настройки вне домена или задачи с слишком большим количеством различных доменов. Также были разработаны другие способы комбинирования адаптеров. Например, в работе [145] добавляются параллельные адаптеры, дообученные для конкретных языковых пар, к большой многоязычной модели Transformer для улучшения машинного перевода.

Итак, мы обсудили последовательные и параллельные адаптеры, а также способы выбора или комбинирования нескольких различных адаптеров, в основном независимо от их конкретной архитектуры. Однако существует важный класс адаптеров, который приводит к эффективным алгоритмам PEFT. Они обучают низкоранговые аппроксимации изменений в полной матрице весов, тем самым позволяя изменять все веса, но сохраняя низкие вычислительные затраты. Мы рассмотрим их в следующем разделе.

2.3. Дообучение с ограниченными ресурсами через низкоранговые аппроксимации: LoRA.

Низкоранговая адаптация (low-rank adaptation, LoRA) — это техника, разработанная для эффективного дообучения больших предварительно обученных моделей за счёт сокращения числа обучаемых параметров с помощью низкоранговых аппроксимаций. Представленная исследователями из Microsoft в 2022 году [43], она начинается с классической идеи низкоранговой декомпозиции матрицы, как показано на рис. 7: большая $N \times M$ матрица X аппроксимируется произведением двух прямоугольных матриц,

$$X \approx UV, \quad \text{где } U \in \mathbb{R}^{N\times k},\ V \in \mathbb{R}^{k\times M} \text{ для } k \ll N, M.$$

Произведение UV по определению является матрицей ранга k, и существуют эффективные алгоритмы для поиска таких U и V, что UV является наилучшей аппроксимацией для X ранга k, где «наилучшая» обычно понимается в терминах L^2-нормы разности $\|X - UV^\top\|_2^2$.

В машинном обучении методы, основанные на низкоранговых аппроксимациях, имеют долгую историю и всегда представляют интерес: если можно предположить, что большая матрица, которая нас интересует, имеет ранг k, мы сможем практически «бесплатно» заменить сложность $O(NM)$ при обучении на сложность $O((N+M)k)$ для обучения матриц U и V. Для больших языковых моделей и больших нейронных сетей в целом ранее проводились исследования, показывающие, что пространство параметров в больших моделях обычно слишком велико:

- в работе [55] введено понятие *внутренней размерности* (intrinsic dimension) путём обучения нейронных сетей в случайных подпространствах пространства параметров с постепенно увеличивающейся размерностью; было показано, что размерность, при которой начинают появляться решения, неизменно намного меньше числа параметров, и часто это разница оказывается весьма значительной;
- в работе [1] это понятие было применено к дообучению языковых моделей; было показано, что стандартные предварительно обученные модели, например RoBERTa, имеют очень низкие внутренние размерности, что означает, что небольшое дообучение в правильном подпространстве может дать значительный результат.

Последний пункт как раз и лежит в основе LoRA. LoRA делает предположение, что *изменения*, внесённые дообучением, могут быть представлены матрицей низкого ранга. Иными словами, она фиксирует предварительно обученную матрицу весов $W \in \mathbb{R}^{N\times M}$ и ищет

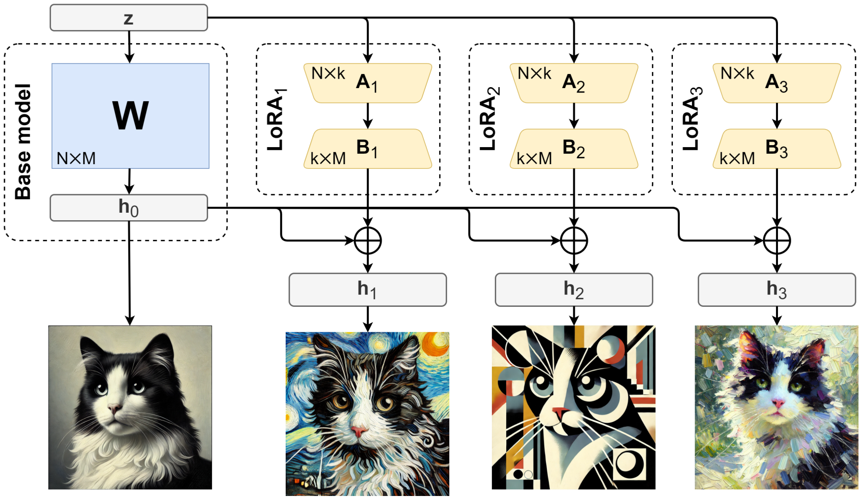


Рис. 8. Различные добавления LoRA B_iA_i к одной и той же предварительно обученной матрице весов W.

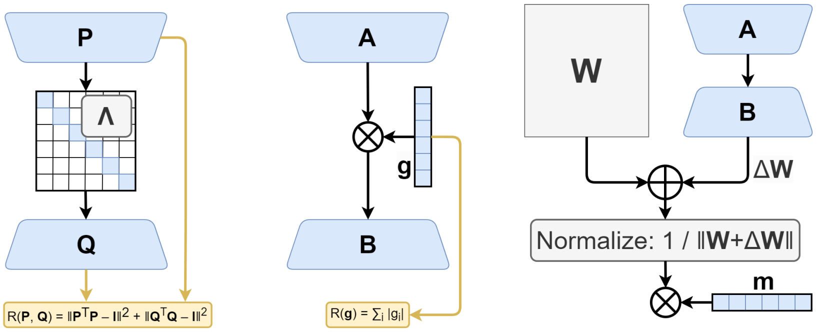
обновление дообучения ΔW в форме низкоранговой аппроксимации:

$$\Delta W = BA, \quad \text{где } B \in \mathbb{R}^{N\times k},\ A \in \mathbb{R}^{k\times M}.$$

Для обучения LoRA использует случайную инициализацию A по нормальному распределению и начальные нули для B, что означает, что в начале обучения ΔW равна нулю. Затем модель дообучается на новом наборе данных, и используется новая матрица весов $W + \Delta W$.

Фокусируясь на низкоранговых обновлениях, LoRA значительно снижает вычислительные и ресурсные затраты по сравнению с традиционными методами дообучения. В работе [43] отмечается, что даже очень маленькие значения k могут быть достаточны; например, авторы приводят контрольную точку LoRA для большой модели Transformer с $k = 4$, где модифицируются только матрицы весов запросов и значений, что уменьшает размер контрольной точки с 350 ГБ для полной матрицы весов до 35 МБ — сокращение в 10000 раз!

После обучения нет необходимости хранить A и B, так как можно просто использовать модифицированную матрицу весов $W' = W + BA$, ведь матрицу весов $N \times M$ всё равно нужно хранить. Но с помощью LoRA можно выполнять дообучение для нескольких различных задач,



(а) (б) (в)

Рис. 9. Вариации метода LoRA; синие модули содержат обучаемые параметры: (а) AdaLoRA [133]; (б) SoRA [27]; (в) DoRA [64].

используя одну и ту же базовую матрицу весов W. Базовую матрицу нужно хранить только один раз, а новые вариации можно хранить в виде набора разных A_i и B_i, как показано на рис. 8. Низкие требования к памяти и значительно уменьшенные вычислительные затраты на обучение позволяют обучать LoRA-обновления даже для больших моделей на пользовательском оборудовании, без дорогих кластеров или нескольких GPU. Это уже привело к появлению множества модификаций на основе LoRA для открытых моделей генерации изображений, особенно *Stable Diffusion* [92], и больших языковых моделей, таких как семейство Llama [108, 109].

Идея LoRA была представлена в 2021 году, поэтому, естественно, уже появилось множество исследований, расширяющих эти идеи. Рассмотрим несколько важных новых расширений LoRA.

Во-первых, важной проблемой в любой схеме низкоранговой аппроксимации является выбор ранга k. Если ранг слишком велик, мы расходуем лишние вычислительные ресурсы и память, а если слишком мал, теряем ценную выразительную способность, которая могла бы стоить нам очень дёшево. Поэтому многие расширения LoRA сосредоточены на автоматическом выборе ранга k:

- *адаптивная низкоранговая адаптация* (adaptive low-rank adaptation, AdaLoRA) [133] параметризует ΔW как сингулярное разложение, $\Delta W = P\Lambda Q$, где матрицы $P \in \mathbb{R}^{N\times k}$ и $Q \in \mathbb{R}^{k\times M}$ являются *ортогональными*, а $\Lambda = \mathrm{diag}\left(\lambda_1, \ldots, \lambda_k\right)$ — диагональной матрицей сингулярных значений (см. рис. 9а); в SVD величины сингулярных значений $|\lambda_i|$ отражают значимость соответствующих компонент разложения, и можно исключить сингулярные значения с малыми величинами; однако выполнение полного SVD для матриц размера $N \times M$ на каждом шаге слишком вычислительно затратно, поэтому AdaLoRA аппроксимирует разложение, добавляя регуляризаторы ортогональности для P и Q в функцию потерь:

$$R(P,Q) = \left\|P^\top P - \mathbf{I}\right\|^2 + \left\|Q^\top Q - \mathbf{I}\right\|^2; \tag{8}$$

- *разреженная низкоранговая адаптация* (sparse low-rank adaptation, SoRA) [27] отмечает, что ключевая часть разложения SVD состоит в том, что матрица Λ выступает в роли механизма фильтрации для строк и столбцов матриц A и B: если сингулярное значение равно нулю, соответствующее измерение можно исключить; поэтому этот этап фильтрации делают явным, рассматривая ΔW как композицию с покоординатным умножением в середине,

$$\Delta W\boldsymbol{x} = B \cdot \left(\boldsymbol{g} \odot \left(A \cdot \boldsymbol{x}\right)\right), \tag{9}$$

  а затем компоненты $\boldsymbol{g}$ оптимизируются с использованием L_1-регуляризации для разреженности (рис. 9б);
- *структурно-осознанная низкоранговая адаптация* (structure-aware low-rank adaptation, SaLoRA) [44] также обучает ранговые значения адаптивно, удаляя компоненты из диагональной матрицы в середине разложения;
- *распределяющая низкоранговая адаптация* (allocating low-rank adaptation, ALoRA) [66] также добавляет диагональную матрицу Λ между A и B, но не стремится делать A и B ортогональными; вместо этого она предлагает отдельный алгоритм абляции AB-LoRA для оценки важности отдельных рангов в Λ, а затем удаляет ранги с низкой важностью или увеличивает размерность матриц, где каждый ранг важен; здесь, как обычно в подходах LoRA, разложение $\Delta W = BA$ выполняется

отдельно для различных матриц весов в сети, и ранги могут отличаться между ними;

- *динамическая низкоранговая адаптация без поиска* (dynamic search-free low-rank adaptation, DyLoRA) [110] выбирает ранг k на каждом шаге обучения и обучает усечённые версии A и B; авторы показывают, что этот подход может привести к значительной экономии времени при обучении LoRA;
- *низкоранговая адаптация с декомпозицией весов* (weight-decomposed low-rank adaptation, DoRA) [64] раскладывает каждый предварительно обученный вес на две составляющие, величину и направление, и обучает их отдельно; формально это означает, что матрица весов W раскладывается как $W = \|W\| \cdot \frac{W}{\|W\|}$, и LoRA применяется только к части, соответствующей направлению:

$$W' = \|W\| \cdot \left(\frac{W}{\|W\|} + \Delta W\right); \qquad (10)$$

  величина $\boldsymbol{m}$, изначально равная вектору норм столбцов матрицы W, также обучается в DoRA (как показано на рис. 9в); в целом этот подход может сократить число обучаемых параметров, а авторы показывают, что он соответствует или превосходит базовый LoRA в различных задачах без дополнительных затрат на обучение.

Применение низкоранговых аппроксимаций для дообучения больших языковых моделей остаётся активной областью исследований, и в ближайшем будущем здесь наверняка появятся новые идеи.

Итак, в этом разделе мы рассмотрели основные методы дообучения больших языковых моделей: на основе обучения с подкреплением, на основе дообучения адаптеров и, в частности, адаптеры на основе низкоранговых приближений. Все они были успешно использованы для дообучения моделей для работы с исходным кодом программ, о чём пойдёт речь в следующих разделах.

§3. Оценка качества кодовых моделей

В этом разделе мы обсудим существующие способы оценки качества кодовых моделей, метрики и открытые данные для их тестирования и дообучения. Тестовые наборы данных, которые описаны в этом разделе, будут появляться в описаниях моделей далее, ведь именно на них модели сравниваются друг с другом.

3.1. HumanEval. *HumanEval* — это один из самых популярных датасетов для моделей, работающих с исходным кодом; он был опубликован исследователями из OpenAI [17]. HumanEval создан для оценки качества задачи порождения Python-функций по документации. Корректность кода проверяется с помощью модульных тестов, в отличие от порождения текста на естественном языке, где оценка обычно проводится по эвристикам или вручную. Для тестирования были вручную созданы 164 задачи по программированию, покрывающие понимание языка, алгоритмов и базовую математику; эти задачи сопоставимы с простыми вопросами на собеседованиях на позицию разработчика.

В качестве основной метрики используется pass@k, которая вычисляется по формуле

$$\mathrm{pass@}k := \mathbb{E}_{\mathrm{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{11}$$

где n — общее число порождённых решений, c — число успешных решений, прошедших тесты, k — число выбранных решений, $\binom{n}{k}$ — биномиальный коэффициент. Метрика показывает вероятность того, что хотя бы одно из k случайно выбранных решений окажется корректным.

3.2. MBPP. *MBPP* (Mostly Basic Programming Problems) — ещё один популярный набор данных для задачи порождения отдельных функций на языке *Python*, на этот раз от *Google Research* [7]. Бенчмарк MBPP содержит 974 короткие программы на Python, созданные с использованием краудсорсинга среди людей с базовыми знаниями Python.

Каждая запись в датасете содержит описание задачи, её решение в виде функции на Python и три теста для проверки корректности. Анализ показал, что 58% задач имеют математический характер, 43% связаны с обработкой списков, 19% с обработкой строк; средняя длина решения составляет 6,8 строк. После ручной проверки и доработки было отобрано 426 гарантированно ясных и корректных вопросов, образующих отредактированный набор данных. В качестве метрики измеряется доля решённых задач, и задача считается решённой, если хотя бы одно из K решений (в работе [7] $K = 80$) успешно проходит тесты.

3.3. APPS. Следующий бенчмарк *APPS* (Automated Programming Progress Standard) содержит 10 000 задач по программированию, собранных с таких открытых платформ, как *CodeForces* и *Kattis*. Он предназначен для оценки навыков программирования и решения задач; в нём содержатся задачи на естественном языке и 131 777 тестовых случаев для проверки правильности решений. В наборе данных содержится 232 421 эталонное решение, написанное людьми, а уровень сложности задач варьируется от начального до уровня университетских соревнований.

Средняя длина описания задачи в APPS составляет 293,2 слова, что подчёркивает их сложность. Набор данных равномерно разделён на 5000 задач для обучения и 5000 для тестирования, при этом у тестовых задач в среднем 21,2 модульных теста на задачу, что обеспечивает тщательную проверку функциональности программ.

Для оценки моделей используются две основные метрики: среднее число пройденных тестов (Test Case Average) и “строгая точность” (Strict Accuracy). Первая метрика измеряет долю тестов, успешно пройденных порождёнными решениями; для задачи p с набором тестов C_p эта метрика вычисляется по формуле

$$\mathrm{TestCaseAverage} = \frac{1}{P}\sum_{p=1}^{P}\frac{1}{C_p}\sum_{c=1}^{C_p}\mathbb{1}\{\mathrm{eval}(\langle \mathrm{code}_p\rangle, x_{p,c}) = y_{p,c}\},$$

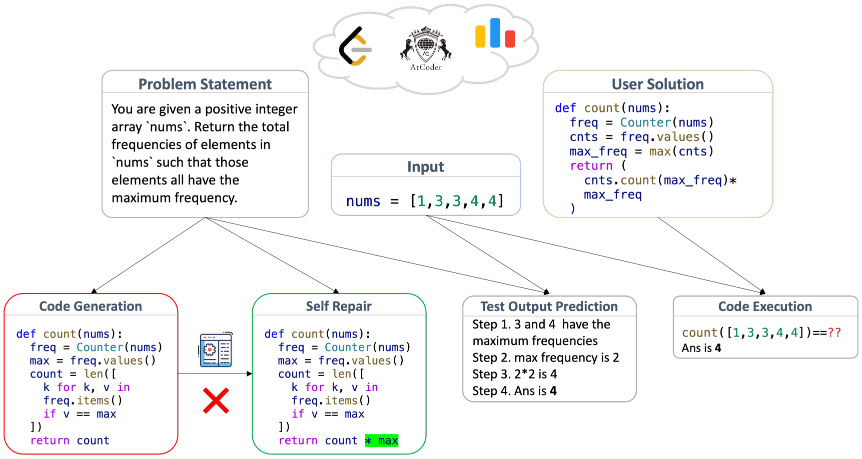
где P — общее число задач, $\mathrm{eval}(\langle code_p\rangle, x_{p,c})$ — результат выполнения кода на входных данных $x_{p,c}$, а $y_{p,c}$ — ожидаемый (правильный) результат.

“Строгая точность” требует, чтобы порождённые решения прошли все тесты для задачи. Она вычисляется как

$$\mathrm{StrictAccuracy} = \frac{1}{P}\sum_{p=1}^{P}\prod_{c=1}^{C_p}\mathbb{1}\{\mathrm{eval}(\langle \mathrm{code}_p\rangle, x_{p,c}) = y_{p,c}\}.$$

Эта метрика более строгая и учитывает только полностью корректные решения.

Таким образом, среднее число пройденных тестов — это менее требовательная метрика; она позволяет фиксировать улучшения и тех моделей, которые пока не могут полностью решить задачи. “Строгая



You are given a positive integer array `nums`. Return the total frequencies of elements in `nums` such that those elements all have the maximum frequency.

```
nums = [1,3,3,4,4]
```

```
def count(nums):
  freq = Counter(nums)
  cnts = freq.values()
  max_freq = max(cnts)
  return (
    cnts.count(max_freq)*
    max_freq
  )
```

```
def count(nums):
  freq = Counter(nums)
  max = freq.values()
  count = len([
    k for k, v in
    freq.items()
    if v == max
  ])
  return count
```

```
def count(nums):
  freq = Counter(nums)
  max = freq.values()
  count = len([
    k for k, v in
    freq.items()
    if v == max
  ])
  return count * max
```

Step 1. 3 and 4 have the maximum frequencies
Step 2. max frequency is 2
Step 3. 2*2 is 4
Step 4. Ans is **4**

```
count([1,3,3,4,4])==??
```

Ans is **4**

Рис. 10. Типы задач в датасете LiveCodeBench [50].

точность", напротив, отражает способность модели решать задачи полностью, включая сложные крайние случаи, что важно для реальных применений.

3.4. LiveCodeBench. *LiveCodeBench* — ещё один часто встречающийся бенчмарк для языка Python, который решает важные проблемы его предшественников [50]. В этой работе авторы боролись с риском утечки тестовых данных в обучающую выборку (contamination). Для этой цели были собраны 511 задач по спортивному программированию с платформ *LeetCode* [2], *AtCoder* [3] и *CodeForces* [4], опубликованных в период с мая 2023 по май 2024 года.

Кроме того, для более полного сравнения кодовых моделей, помимо задачи порождения функции по документации авторы добавили три дополнительные категории задач:

- исправление ошибок в программе по некорректно пройденным тестам;
- виртуальное "исполнение" программы с конкретным входом;

---

[2] https://leetcode.com/
[3] https://atcoder.jp/
[4] https://codeforces.com/

- предсказание выходных данных по описанию задачи и входным данным.

Все виды задач проиллюстрированы на рис. 10.

Для задач порождения и исправления программ использовалась метрика pass@k из *HumanEval*, определённая выше в (11), а для "исполнения" программы и предсказания выходных данных — бинарная метрика сопоставления с эталонным ответом. Для первых двух категорий на одну задачу в среднем приходится около 17 тестов. Кроме того, авторы классифицировали все задачи по уровню сложности (easy, medium, hard) и сбалансировали их количество, что позволяет детальнее сравнивать модели с одинаковым среднем значением метрики.

Интересно, что в зависимости от даты выхода модели было замечено резкое снижение качества для моделей *DeepSeek*, *GPT-4o*, *Codestral* и других: если задачи не попадают в массив данных, доступных для (до)обучения в момент выхода модели, то некоторые модели начинают заметно хуже с ними справляться. Поэтому выбор более свежих задач (опубликованных уже после выхода модели) позволяет получать более реалистичную оценку качества. Для примера таблицы 1 и 2 показывают состояние дел на бенчмарке LCB в начале декабря 2024 года[5] для задач автоматического порождения и автоматического исправления кода соответственно; в таблицах 1 и 2 оценка ведётся только по задачам, которые добавлены после 1 января 2024 года. Курсивом выделены модели, которые могут быть контаминированы частью этих задач в обучающей выборке, и они в итоге не получают ранг в списке.

3.5. HumanEval+ и MBPP+. Следующие бенчмарки *HumanEval+* и *MBPP+* — это продолжения и варианты *HumanEval* и *MBPP* соответственно, в которые авторы добавили необходимые тесты и исправили неоднозначные формулировки заданий [63].

3.6. CodeContests. *CodeContests* — это набор данных, который был собран для дообучения и тестирования AlphaCode модели [60]. Он содержит задачи на спортивное программирование, взятые с платформы *CodeForces* [6], с небольшим набором открытых тестов. На каждую задачу в среднем приходится 203,7 закрытых теста. Тестовая часть *CodeContests* содержит 165 задач. В качестве метрик использовались следующие:

---

[5] `https://livecodebench.github.io/leaderboard.html`

[6] https://codeforces.com/

Таблица 1. Сравнение кодовых моделей на добавленных в 2024 году задачах датасета *LiveCodeBench* по порождению кода, по состоянию на начало декабря 2024 года [50].

	Модель	**Метрика Pass@1, %**			
		All	**Easy**	**Medium**	**Hard**
1	O1-Mini-2024-09-12 (N=1)	67.2	93.5	74.1	31.7
2	O1-Preview-2024-09-12 (N=1)	52.5	90.3	53.1	12.5
	Claude-3.5-Sonnet-20240620	42.6	85.6	36.9	5.1
	Claude-3.5-Sonnet-20241022	42.5	92	30.6	5.5
	Qwen2.5-Ins-72B	41.7	85.6	36.3	2.7
3	GPT-4O-2024-08-06	41.3	87.5	33.6	2.6
4	ChatGPT-4O-Latest-08-24	41.2	88.5	31.7	3.6
5	GPT-4-Turbo-2024-04-09	40.7	84.5	33.1	4.5
	Qwen2.5-Ins-32B	40.1	84	32.7	3.6
6	GPT-4O-2024-05-13	40.1	86.9	30	3.9
	Qwen2.5-Coder-Ins-32B	40	84.2	34.8	0.4
7	LLama3.1-405b-Ins-FP8	37.3	81	27.7	3.5
8	Gemini-Pro-1.5-002	36.4	82.5	24.1	3.5
9	GPT-4-Turbo-1106	35.7	81.8	23.6	2.7
10	Mistral-Large	35.6	82.9	22.1	2.7
11	DeepSeek-V2	34.8	80.6	23.4	1
12	GPT-4O-mini-2024-07-18	34.8	78.2	23.6	3.1
13	DeepSeek-Coder-V2-Instruct-0724	34.6	79.6	24.1	0.6
14	DeepSeek-Coder-V2-Instruct	34.4	80.2	22.1	1.8
15	Gemini-Flash-1.5-002	34.4	81	21.3	1.7
16	DeepSeek-V2-Chat-0628	33.2	78.1	20.6	1.8
17	GPT-4-0613	32.5	74.9	20.6	2.8
18	Claude-3-Opus	31.9	76	18.7	2.1
	Qwen2.5-Base-72B	28.2	64	20.4	0.5
	Codestral-Latest	27.4	67	15.9	0.3
19	LLama3.1-70b-Ins	26.3	62	15.6	2
	Qwen2.5-Coder-Ins-7B	26.2	62.7	16.5	0.1
	Qwen2.5-Base-32B	26	62.6	15.7	0.6
20	Qwen2-Ins-72B	25.3	63.1	13.1	0.9

- pass@k — процент решённых задач при протоколе, в котором модель порождает k решений для каждой задачи, они тестируются на скрытых тестах, и если какое-либо решение из k проходит тесты, проблема считается решенной;

Таблица 2. Сравнение кодовых моделей на добавленных в 2024 году задачах датасета *LiveCodeBench* по автоматическому исправлению кода, по состоянию на начало декабря 2024 года [50].

	Модель	**Метрика Pass@1, %**			
		All	**Easy**	**Medium**	**Hard**
1	O1-Mini-2024-09-12 (N=1)	67.7	99.2	76.9	24.2
2	O1-Preview-2024-09-12 (N=1)	57.4	98.4	59.4	12.5
	Claude-3.5-Sonnet-20241022	51	96	49.4	6.4
	Claude-3.5-Sonnet-20240620	47.4	94.3	43.4	3.8
	Qwen2.5-Ins-32B	44.3	90.8	39	2.8
3	GPT-4-Turbo-2024-04-09	44.3	91.5	37.3	3.8
4	GPT-4O-2024-08-06	44.1	91.9	38	2.1
5	GPT-4O-2024-05-13	43.3	90.9	36.2	2.6
6	LLama3.1-405b-Ins-FP8	40.9	88	32.4	2.4
7	Gemini-Pro-1.5-002	40.7	86.5	32.9	2.6
8	GPT-4O-mini-2024-07-18	38	85.1	27.6	1.8
9	Gemini-Flash-1.5-002	36.7	85.4	23.5	2.2
10	Mistral-Large	36.1	87.4	20.4	1.9
11	LLama3.1-70b-Ins	32.1	75.2	20.7	1.2
12	Qwen2-Ins-72B	30.5	72.6	19.2	0.5
13	Claude-3-Haiku	25.4	66.7	10.9	0
14	LLama3.1-8b-Ins	18.8	50.8	6.1	0.8

- 10@k — процент решённых задач при протоколе, в котором модель порождает k решений для каждой задачи, но может запустить скрытые тесты только на 10 из них; это отражает разницу между относительно «дешёвыми» дополнительными порождениями из модели и относительно «дорогим» процессом их перепроверки и тестирования: если модель сама может выбрать лучшие из порождённых, будет несложно попросить её порождать больше.

Также стоит отметить, что на каждую задачу отводилось фиксированное время для прохождения тестов.

3.7. Super-NaturalInstructions. В сравнении моделей будет встречаться и бенчмарк более общего назначения *Super-NaturalInstructions*, созданный для проверки обобщающих способностей моделей; в нём модель просят решать различные задачи по инструкции на естественном языке [117]. Датасет содержит 1616 различных задач из 76 уникальных типов задач, в частности классификации, редактирования текстов, написания краткого пересказа и т.д., включая и задачи, связанные с программным кодом — описание кода и порождение программы по текстовому описанию.

3.8. TransCoder. Следующий бенчмарк, появившийся в работе о модели *TransCoder*, был создан для оценки качества трансляции программ с одного языка программирования на другой [93]. Для этой цели авторы собрали с сайта *GeeksforGeeks* примеры реализаций одной и той же функции на языках C++, Java и Python. В итоге было собрано порядка 460 функций для каждого языка и 10 тестов для каждой функции. В качестве метрики качества подсчитывается процент решений, которые успешно прошли модульные тесты.

3.9. tldr. Бенчмарк *tldr* [143] содержит пары из человеческих запросов на английском языке и названия команды консоли bash с нужной комбинацией флагов. Всего он содержит 9187 пар и 1879 уникальных команд bash. Бенчмарк позволяет оценивать качество порождения команд bash моделью по инструкции от пользователя. В качестве метрик используются:

- совпадение имени команды,
- точное совпадение команды с эталоном,
- F_1-мера на токенах,
- метрика charBLEU, вариант стандартного BLEU, считающийся по n-граммам символов, а не токенов [87, 88].

3.10. SWE-bench. Бенчмарк *SWE-bench* был разработан для оценки способности больших языковых моделей решать реальные кодовые задачи, взятые с GitHub. Он включает в себя 2294 пары описания проблемы (issue) и соответствующего решения (pull request) из 12 популярных репозиториев на языке Python. Моделям предоставляется кодовая база и описание проблемы, и от них требуется породить исправление (patch), которое решает указанную проблему.

SWE-bench проверяет, могут ли модели понимать и координировать изменения в нескольких функциях, классах и даже файлах одновременно, что требует сложных рассуждений и взаимодействия с программным окружением. Оценка проводится с помощью модульных тестов: если предложенное исправление проходит как новые, так и существующие тесты, оно считается успешным.

§4. Дообучение LLM методами обучения с подкреплением

4.1. Постановка задачи обучения с подкреплением. Обучение с подкреплением (reinforcement learning, RL) для порождения кода рассматривается как задача последовательного принятия решений, формализуемая через марковский процесс принятия решений (Markov decision process, MDP). MDP задаётся пятёркой $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma)$, где:

- $\mathcal{S}$ — пространство состояний, где состояние $s_t = (y_{<t}, x)$ включает некоторый префикс из токенов $y_{<t}$ и описание задачи x;
- $\mathcal{A}$ — пространство действий, соответствующее выбору следующего токена y_t;
- $\mathcal{P}(s' \mid s, a)$ — функция переходов, определяющая вероятность перехода в состояние s' после выполнения действия a в состоянии s;
- $r(s, a)$ — функция вознаграждения, задающая оценку действия a в состоянии s;
- $\gamma \in [0, 1]$ — коэффициент дисконтирования будущих наград.

Цель обучения с подкреплением заключается в максимизации ожидаемого суммарного вознаграждения:

$$J(\pi_\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right], \tag{12}$$

где $\pi_\theta(a \mid s)$ — параметризованная стратегия (policy), определяющая распределение вероятностей действий в каждом состоянии.

Функции значения состояний и пар состояние-действие определяются как

$$V^\pi(s) = \mathbb{E}_{a\sim\pi}\left[Q^\pi(s, a)\right], \quad Q^\pi(s, a) = \mathbb{E}\left[r(s, a) + \gamma V^\pi(s')\right], \tag{13}$$

где s' — следующее состояние.

Обучение с подкреплением обычно делается одним из двух способов:

- обучение функций значений (value-based RL), в котором обучаются функции V и Q, введённые в (13), в частности функции V^* и Q^*, соответствующие оптимальной стратегии π^*, а затем сама стратегия π^* выводится из них; к этому классу относятся методы Монте-Карло в обучении с подкреплением и обучение на основе временных разностей (TD-обучение), в частности Q-обучение [120];

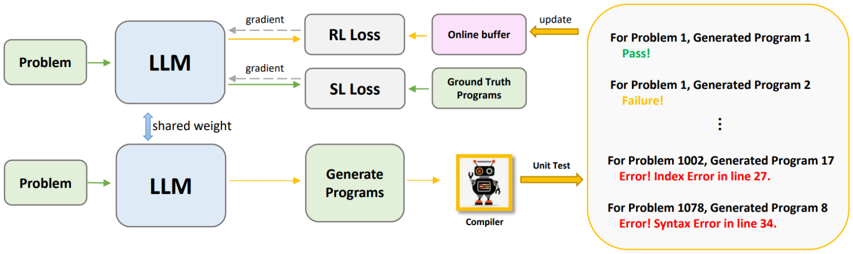


Рис. 11. Обучение с подкреплением на основе юнит-тестов (RLTF) [62].

- обучение стратегии напрямую методами градиента по стратегиям (policy gradient) [9, 105]; к этому классу относится классический алгоритм REINFORCE [122], а также современные методы оптимизации стратегии на основе доверительных областей (trust region policy optimization, TRPO) [94], проксимальной оптимизации стратегии (proximal policy optimization, PPO) [95] и другие.

Каждый из рассматриваемых ниже методов использует один из этих алгоритмов обучения; подробное их изложение выходит за рамки обзора, см., например, [95, 104, 140].

4.2. RLTF. В работе [62] предложен метод обучения с подкреплением на основе данных из юнит-тестов (Reinforcement Learning from Unit Test Feedback, RLTF), который позволяет улучшить качество порождения программного кода с использованием предобученных языковых моделей (LLM). Основная идея RLTF, проиллюстрированная на рис. 11, заключается в том, чтобы интегрировать механизм порождения данных в реальном времени с многоуровневой обратной связью от юнит-тестов, что позволяет модели эффективно обучаться за счёт более разнообразных и релевантных примеров.

Задача синтеза программного кода здесь формализуется как задача последовательного порождения кода W, удовлетворяющего высокоуровневому описанию задачи D. Цель состоит в максимизации условной вероятности

$$\max p\left(W|D,\boldsymbol{\theta}\right) = \max\prod_{t=1}^{T} p\left(\boldsymbol{w}_t|D,\boldsymbol{\theta},\boldsymbol{w}_{1:t-1}\right),$$

где $\boldsymbol{\theta}$ — параметры модели, T — длина последовательности, $\boldsymbol{w}_t$ — t-й токен сгенерированного кода.

Метод RLTF использует онлайн-буфер, который динамически обновляется в процессе обучения. Буфер содержит пары данных, состоящие из описания задачи D, сгенерированного кода $\hat{W}$ и обратной связи от компилятора $\mathrm{FB}(\hat{W})$. Для оптимизации используется следующая функция потерь, объединяющая стандартное обучение с подкреплением и обратную связь разной степени детализации:

$$L_{\text{total}} = L_{\text{SL}} + L_{\text{coarse}} + L_{\text{fine}} + L_{\text{adaptive}},$$

где L_{SL} — стандартная функция потерь для обучения с учителем, а L_{coarse}, L_{fine}, L_{adaptive} — компоненты для обучения с подкреплением.

Авторы [62] выделяют три механизма обратной связи:

- *грубая обратная связь* (coarse-grained feedback), где вознаграждение зависит от статуса выполнения кода (успешное выполнение, сбой или синтаксическая ошибка):

$$R_{\text{coarse}}(\hat{W}) = \begin{cases} 1.0, & \text{если } \mathrm{FB}(\hat{W}) = \text{pass},\\ -0.3, & \text{если } \mathrm{FB}(\hat{W}) = \text{failure},\\ -0.6, & \text{если } \mathrm{FB}(\hat{W}) = \text{error (не синтаксическая)},\\ -1.0, & \text{если } \mathrm{FB}(\hat{W}) = \text{syntax error};\end{cases}$$

- *тонкая обратная связь* (fine-grained feedback), где ошибки классифицируются на основе конкретных строк или конкретных причин ошибки и проблем в коде, и вознаграждение определяется как

$$R_{\text{fine}}(\hat{W}) = \begin{cases} 0.0, & \text{если ошибка относится к категории } U_{\text{ignore}},\\ -0.3, & \text{в противном случае.}\end{cases}$$

- *адаптивная обратная связь* (adaptive feedback), где вознаграждение зависит от доли успешно пройденных тестов:

$$R_{\text{adaptive}}(\hat{W}) = -0.3 + 1.3\cdot\frac{N_{\text{pass}}}{N_{\text{pass}} + N_{\text{fail}}},$$

  где N_{pass} и N_{fail} — количество пройденных и проваленных тестов соответственно.

Общая функция потерь объединяет все уровни обратной связи:

$$L_{\text{total}} = L_{\text{SL}} - \sum_{t=1}^{T} R(\hat{W}) \log p \left(\boldsymbol{w}_t | D, \boldsymbol{\theta}, \boldsymbol{w}_{1:t-1} \right),$$

где $R(\hat{W})$ объединяет R_{coarse}, R_{fine} и R_{adaptive}.

Эксперименты на бенчмарках APPS и MBPP продемонстрировали, что RLTF улучшает результаты порождения кода, превосходя другие подходы, в частности CodeRL и PPOCoder, как по качеству порождения, так и по универсальности применения на различных моделях — исследовались модели CodeT5 и CodeGen. Абляционные исследования также подтвердили важность онлайн-обучения и предложенных механизмов обратной связи.

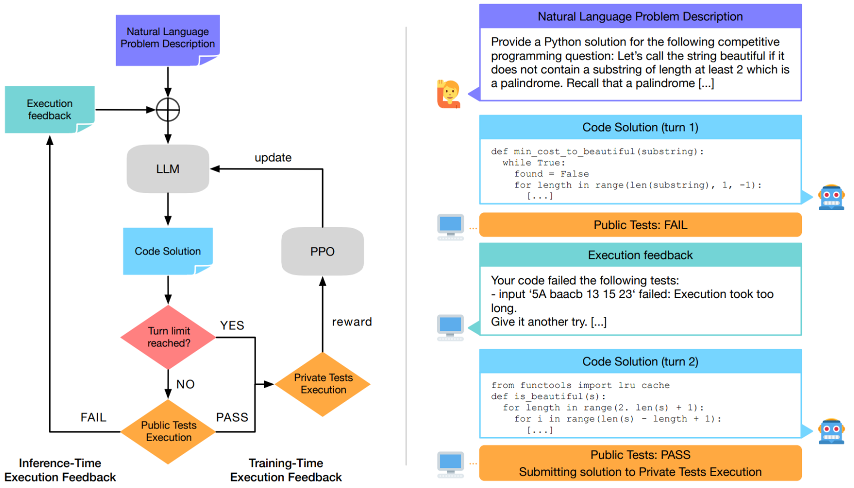
4.3. RLEF. В работе [34] предлагается метод обучения с подкреплением на основе данных из исполнения кода (Reinforcement Learning with Execution Feedback, RLEF), который также использует обучение с подкреплением, но на основе обратной связи, получаемой в процессе выполнения кода, а не из компилятора или из юнит-тестов. Основная идея заключается в том, чтобы представить задачу как частично наблюдаемый марковский процесс принятия решений (MDP), где действиями являются последовательные порождения кода, а наблюдения включают обратную связь о результатах выполнения тестов. Это позволяет модели LLM не только порождать решения, но и эффективно исправлять ошибки на основе предоставленной обратной связи, используя итеративный процесс исправлений, как показано на рис. 12.

Метод RLEF оптимизирует стратегию π с использованием алгоритма Proximal Policy Optimization (PPO). Награда определяется следующим образом:

$$r(s_t, a_t) = \begin{cases} 1, & \text{если все тесты пройдены,} \\ -1, & \text{если хотя бы один тест провален,} \\ -0.2, & \text{если сгенерированный код некорректен.} \end{cases}$$

Общая функция вознаграждения также включает регуляризацию в виде KL-дивергенции между текущей стратегией π и начальной стратегией ρ:

$$R(s_t, a_t) = r(s_t, a_t) - \beta \log \frac{\pi(a_t | c_t)}{\rho(a_t | c_t)},$$



Natural Language Problem Description

Provide a Python solution for the following competitive programming question: Let's call the string beautiful if it does not contain a substring of length at least 2 which is a palindrome. Recall that a palindrome [...]

Code Solution (turn 1)

```
def min_cost_to_beautiful(substring):
  while True:
    found = False
    for length in range(len(substring), 1, -1):
      [...]
```

Public Tests: FAIL

Execution feedback

Your code failed the following tests:
- input '5A baacb 13 15 23' failed: Execution took too long.
Give it another try. [...]

Code Solution (turn 2)

```
from functools import lru cache
def is_beautiful(s):
  for length in range(2. len(s) + 1):
    for i in range(len(s) - length + 1):
      [...]
```

Public Tests: PASS
Submitting solution to Private Tests Execution

Рис. 12. Обучение с подкреплением на основе данных об исполнении кода (RLEF) [34]; слева — общая блок-схема алгоритма, справа — пример «внутреннего диалога».

где c_t — последовательность предыдущих наблюдений и действий, β — коэффициент регуляризации.

Оптимизация проводится на основе функции преимущества (advantage function) $A_t = R(s_t, a_t) - V(c_t)$, где $V(c_t)$ — функция ценности. Для минимизации потерь используется обрезанная целевая функция метода PPO:

$$L_{\text{clipped}}(\theta) = \mathbb{E}_t \left[\min\left(r_t(\theta)A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t\right)\right],$$

где $r_t(\theta) = \frac{\pi_\theta(a_t|c_t)}{\pi_{\theta_{\text{old}}}(a_t|c_t)}$.

Эксперименты, проведённые на сложных бенчмарках, в частности CodeContests, HumanEval+ и MBPP+, показали, что RLEF позволяет улучшить результаты с существенно меньшими вычислительными затратами. Метод обеспечивает улучшение как для малых (8B), так и для крупных (70B) моделей, демонстрируя их способность эффективно исправлять ошибки и адаптироваться к обратной связи в процессе итеративного синтеза кода. Экспериментальное исследование в [34] показало, что RLEF не только увеличивает точность первого порождения, но и значительно улучшает качество последующих исправлений; это открывает широкие возможности для автоматического последовательного исправления и улучшения порождаемого кода.

4.4. $\mathcal{B}$-Coder. Метод $\mathcal{B}$-Coder [129] также представляет собой архитектуру, основанную на методах обучения с подкреплением (RL) для задачи синтеза программного кода, но в отличие от большинства других методов, рассмотренных в этом разделе, $\mathcal{B}$-Coder использует подходы на основе функции значений (value-based RL) вместо алгоритмов оптимизации стратегий (policy-based RL), к которым относится уже не раз упоминавшийся PPO. $\mathcal{B}$-Coder направлен на оптимизацию функциональной корректности программ с минимальными затратами на проектирование функций вознаграждения.

Задача синтеза программ снова формализуется как задача последовательного порождения программы $W = (w_0, w_1, \dots, w_T)$ на основе текстового описания задачи D. Порождения каждого токена w_t интерпретируется как действие a_t, выполненное в состоянии $s_t = (w_{<t}, D)$. Таким образом, процесс обучения модели описывается через марковский процесс принятия решений (MDP), характеризуемый пятеркой (S, A, P, r, γ), где:

$$r(s_T, a_T) = \begin{cases} 1, & \text{если } W \text{ прошло все модульные тесты,} \\ -0.3, & \text{если } W \text{ не прошло хотя бы один тест,} \\ -0.6, & \text{если } W \text{ вызывает ошибку выполнения,} \\ -1.0, & \text{если } W \text{ содержит синтаксическую ошибку.} \end{cases}$$

Задача обучения с подкреплением заключается в максимизации дисконтированной суммы вознаграждений:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \mid \pi\right],$$

где π — стратегия выбора действий.

В отличие от других методов, $\mathcal{B}$-Coder использует Q-обучение для оценки функции ценности состояния-действия

$$Q^\pi(s, a) = \mathbb{E}\left[r(s, a) + \gamma \max_{a'} Q^\pi(s', a')\right],$$

где s' — следующее состояние, получаемое после применения действия a в состоянии s.

Для стабилизации процесса обучения метод использует следующие приёмы:

- *инициализация функции Q с помощью предобученных моделей*: функция Q формируется из логитов $l(s,a)$ языковой модели, преобразованных через функцию softmax:
$$Q(s,a) = \alpha\left(l(s,a) - \max_{a'} l(s,a')\right) + V(s),$$
где $V(s)$ — функция ценности состояния;
- *консервативный оператор Беллмана*: вместо стандартного оптимального оператора Беллмана
$$(B^*Q)(s,a) = r(s,a) + \gamma \max_{a'} Q(s',a')$$
используется консервативный оператор
$$(B_qQ)(s,a) = r(s,a) + \gamma Q(s',a'_q),$$
где $a'_q = \arg\max_{a'} q(a' \mid s')$ — действие, выбираемое фиксированной стратегией q.

Обучение $\mathcal{B}$-Coder проходит в два этапа:

(1) *предобучение* $V(s)$, когда функция состояния обучается через минимизацию ошибки временной разницы (TD-обучение)
$$L_V = \mathbb{E}\left[\left(r(s,a) + \gamma V(s') - Q(s,a)\right)^2\right];$$

(2) *тонкая настройка* $Q(s,a)$, когда оптимизируется комбинированная функция потерь
$$L_{\mathrm{ft}} = L_Q + \beta_{\mathrm{adv}} L_{\mathrm{adv}} + \beta_{\mathrm{ce}} L_{\mathrm{ce}},$$
где L_Q — TD-ошибка с консервативным оператором, L_{adv} — регуляризация функции преимущества $A(s,a)$, а L_{ce} — функция ошибки на основе кросс-энтропии.

Эксперименты в работе [129] на бенчмарках APPS и MBPP показали, что $\mathcal{B}$-Coder превосходит предшествующие подходы, в частности CodeRL, PPOCoder и RLTF, демонстрируя лучшее качество порождения кода. Консервативный оператор Беллмана улучшает стабильность обучения, а использование off-policy данных обеспечивает высокую эффективность выборки.

4.5. PSGPO. В работе [23] представлен вариант метода оптимизации стратегий на основе так называемых процессуальных наград (Process Supervision-Guided Policy Optimization, PSGPO). Главная его мотивация — устранить проблему разреженных сигналов вознаграждения в задачах порождения кода с использованием обучения с подкреплением. Основная новизна PSGPO заключается в разработанной процессуальной модели наград (Process Reward Model, PRM), которая обеспечивает детальную обратную связь на уровне строк кода, что позволяет модели улучшать свои решения на промежуточных этапах порождения.

Проблема порождения кода снова формулируется как последовательное порождение токенов $y = (y_1, y_2, \ldots, y_T)$ на основе входного описания задачи x. Предварительно обученная языковая модель $p_\theta(y \mid x)$ оценивает условное распределение вероятностей:

$$p_\theta(y \mid x) = \prod_{t=1}^{T} p_\theta(y_t \mid y_{<t}, x),$$

где $y_{<t} = (y_1, \ldots, y_{t-1})$.

Цель обучения с подкреплением — максимизация ожидаемой награды

$$\mathbb{E}_{y \sim p_\theta} \left[R(x, y) \right],$$

где $R(x, y)$ — функция вознаграждения, основанная на прохождении тестов.

Процессуальная модель наград (PRM) предоставляет детальную обратную связь, оценивая корректность каждой строки сгенерированного кода. Для обучения PRM используются автоматические метки, полученные с помощью бинарного поиска. Метка для префикса $y_{\le m}$ определяется как

$$l_m = \begin{cases} +1, & \text{если существует дополнение, проходящее все тесты,} \\ -1, & \text{если все дополнения проваливаются.} \end{cases}$$

PRM обучается минимизировать ошибку в форме среднеквадратичного отклонения:

$$\min_\phi \sum_{(x, y_{\le m}, l_m)} \left(R_\phi(x, y_{\le m}) - l_m \right)^2,$$

где R_ϕ — предсказание PRM.

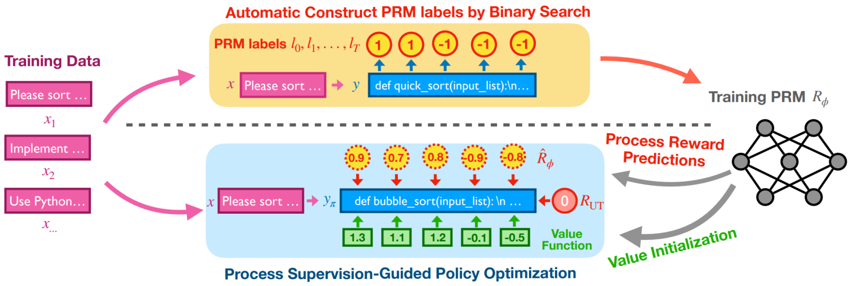


Рис. 13. Оптимизации стратегии на основе процессуальных наград (PSGPO) [23].

Далее в обучении с подкреплением PRM используется двумя способами:

(1) *детальные вознаграждения* (dense rewards): PRM добавляет пошаговые сигналы на уровне строк кода,

$$R_{\mathrm{PRM}}(x, y) = \sum_{m=1}^{T} R_\phi(x, y_{\le m}),$$

и общая функция вознаграждения превращается в

$$R(x, y) = R_{\mathrm{UT}}(x, y) + \lambda R_{\mathrm{PRM}}(x, y),$$

где R_{UT} — бинарная награда за прохождение тестов, λ — вес PRM;

(2) *инициализация функции значений* (value initialization): PRM используется для задания начальных значений функции значений состояний

$$V(x, y_{\le m}) \approx R_\phi(x, y_{\le m}).$$

Процесс автоматического порождения меток PRM и интеграция PRM и процесс обучения с подкреплением проиллюстрированы на рис. 13.

Эксперименты работы [23] на бенчмарках LiveCodeBench и InHouseBench(закрытый бенчмарк ByteDance) показали, что PSGPO с использованием PRM улучшает метрику Pass@1 для длинных решений (>100 токенов) на 9% по сравнению с базовым подходом на основе RL. Комбинация детальных вознаграждений и инициализации функции значений улучшает обучение и за счёт дополнительных сигналов обратной связи, и за счёт улучшения стабильности обучения, и в экспериментах метод PSGPO продемонстрировал значительные преимущества в задачах с длинным горизонтом планирования.

4.6. ILF. Работа [16] применяет к порождению кода разработанный авторами метод имитационного обучения на основе текстовой обратной связи на естественном языке (Imitation Learning from Language Feedback, ILF) [15]. Основная идея ILF заключается в использовании небольшого объёма человеческих аннотаций для обучения модели на этапах порождения и исправления программ, что делает метод эффективным с точки зрения требующихся данных и удобным для использования. На рис. 14 показаны общая схема подхода (рис. 14а) и пример запроса с текстовой обратной связью (рис. 14б).

Задача синтеза программ здесь формализуется как задача обучения вероятностной модели π_θ, параметризованной θ, для порождения программы $x \in V^*$ из описания задачи $t \in T$. Здесь V^* обозначает множество всех возможных последовательностей токенов. Вероятность порождения определяется как

$$\pi_\theta(x \mid t) = \prod_{i=1}^{|x|} \pi_\theta(x_i \mid x_{<i}, t),$$

где $x_{<i}$ — префикс длины $i-1$.

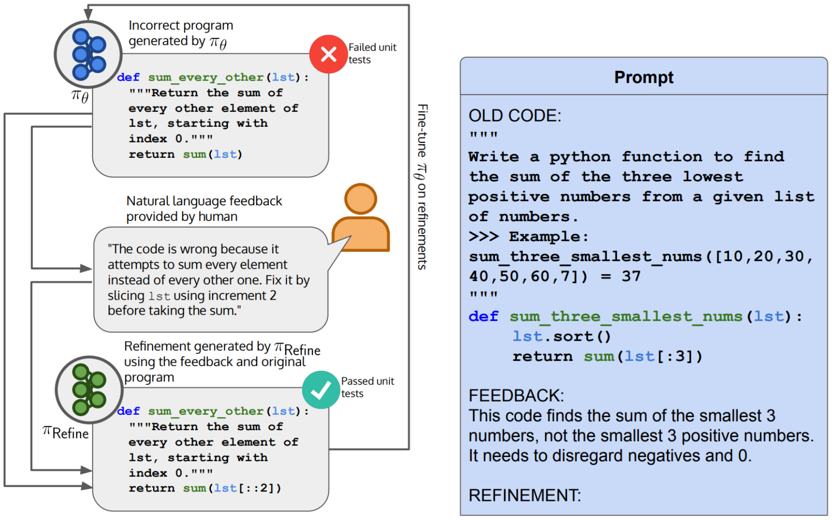
Цель ILF состоит в минимизации дивергенции Кульбака-Лейблера между текущей моделью π_θ и целевым распределением π_t^*:

$$\min_\theta \mathbb{E}_{t \sim p(t)} \left[\mathrm{KL}(\pi_t^* \| \pi_\theta(\cdot \mid t)) \right],$$

где $\pi_t^*(x) \propto \exp(\beta R(x,t))$, а $R(x,t)$ — функция награды, основанная на прохождении модульных тестов.

Сам алгоритм ILF состоит из нескольких этапов:

(1) сэмплирование некорректных программ $x_0 \sim \pi_\theta(\cdot \mid t)$, не проходящих тесты: $eval(x_0, t) = 0$;
(2) аннотирование x_0 с использованием обратной связи f, написанной человеком;
(3) обучение модели π_{Refine} для генерации исправлений x_1, которые проходят тесты, на основе f и x_0;
(4) обучение π_θ на исправлениях x_1, сгенерированных π_{Refine}.



```
def sum_every_other(lst):
 """Return the sum of
 every other element of
 lst, starting with
 index 0."""
 return sum(lst)
```

"The code is wrong because it attempts to sum every element instead of every other one. Fix it by slicing `lst` using increment 2 before taking the sum."

```
def sum_every_other(lst):
 """Return the sum of
 every other element of
 lst, starting with
 index 0."""
 return sum(lst[::2])
```

OLD CODE:

```
"""
Write a python function to find
the sum of the three lowest
positive numbers from a given list
of numbers.
>>> Example:
sum_three_smallest_nums([10,20,30,
40,50,60,7]) = 37
"""
def sum_three_smallest_nums(lst):
    lst.sort()
    return sum(lst[:3])
```

FEEDBACK:
This code finds the sum of the smallest 3 numbers, not the smallest 3 positive numbers. It needs to disregard negatives and 0.

REFINEMENT:

(а) (б)

Рис. 14. Имитационное обучение на основе текстовой обратной связи (ILF) [16]: (а) общая схема подхода; (б) пример запроса с текстовой обратной связью.

Предложенное распределение $q_t(x_1)$, приближающее π_t^*, формализуется как

$$q_t(\boldsymbol{x}_1) = \sum_{\boldsymbol{x}_0, f} \pi_\theta(x_0 \mid t) \cdot p_F\left(f | t, \boldsymbol{x}_0\right) \cdot \pi_{\text{Refine}}\left(\boldsymbol{x}_1 | t, \boldsymbol{x}_0, f\right) \cdot \delta\left(\text{eval}(\boldsymbol{x}_1, t) = 1\right),$$

где $p_F\left(f | t, \boldsymbol{x}_0\right)$ — распределение обратной связи, а δ — дельта-функция, задающая ограничение прохождения тестов.

В работе [16] метод был протестирован на бенчмарке MBPP, где он обеспечил значительное улучшение метрики Pass@1: $+10\%$ в абсолютных величинах и $+38\%$ в относительных по сравнению с базовыми подходами. Анализ показал, что качество человеческой обратной связи играет ключевую роль, тогда как автоматически порождённая обратная связь имеет существенно меньшую эффективность. Таким образом, ILF тоже представляет собой эффективный подход для обучения моделей порождения программного кода, обеспечивая значительные улучшения с использованием ограниченного объёма сделанных вручную аннотаций.

§5. Следование инструкциям

5.1. Постановка задачи. Обучение следованию инструкциям (*instruction tuning*) — это методика машинного обучения, используемая для дообучения (fine-tuning) больших языковых моделей (large language models, LLM). В этом подходе модель дообучается на наборе данных, где задачи представлены в виде инструкций на естественном языке. Цель обучения следованию инструкциям состоит в том, чтобы улучшить способность модели к обобщению на широкий спектр задач, сделав её более адаптированной к выполнению запросов пользователей.

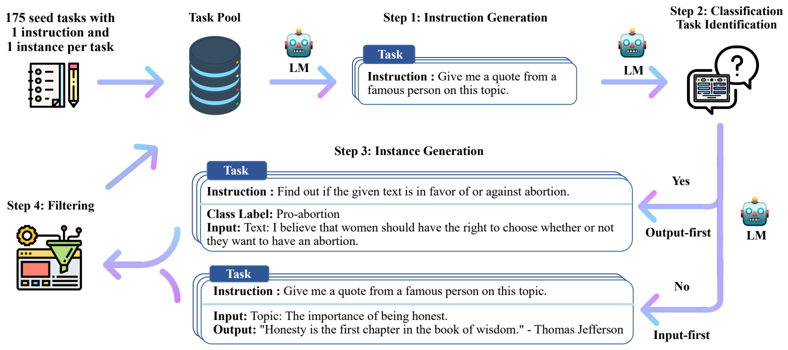
Модель обучается на данных, содержащих примеры задач, описанных в виде инструкций, обычно с входными данными и ожидаемым результатом. Например:

```
Инструкция: Переведите текст "Hello" на французский язык.
Вход: "Hello"
Выход: "Bonjour"
```

Обращаем внимание на разницу между стандартным обучением следованию инструкциям и описанном в разделе 2.1 методе RLHF: здесь речь идёт об обычном обучении с учителем, без методов обучения с подкреплением.

5.2. InstructGPT и Self-Instruct. Одной из ключевых работ в этой области является работа исследователей OpenAI про InstructGPT [79]. Поскольку эта работа также была ключевой для RLHF, мы уже подробно разбирали её в разделе 2.1. InstructGPT улучшал способности языковых моделей следовать инструкциям пользователей, быть полезной и безопасной с помощью специальной процедуры дообучения, в которой было и обучение следованию инструкциям, и RLHF, в три этапа:

(1) дообучение с учителем (SFT), где модель дообучается на основе демонстраций, предоставленных аннотаторами, которые создают идеальные ответы на разнообразные промпты;
(2) моделирование награды, где аннотаторы ранжируют ответы модели по качеству, затем отранжированные ответы используются



Instruction : Give me a quote from a famous person on this topic.

Instruction : Find out if the given text is in favor of or against abortion.

Class Label: Pro-abortion
Input: Text: I believe that women should have the right to choose whether or not they want to have an abortion.

Instruction : Give me a quote from a famous person on this topic.

Input: Topic: The importance of being honest.
Output: "Honesty is the first chapter in the book of wisdom." - Thomas Jefferson

Рис. 15. Схема Self-Instruct. Процесс синтеза и отбора новых инструкций [116].

для обучения модели вознаграждения, предсказывающей предпочтения людей;

(3) обучение с подкреплением методом PPO [96], как описано в разделе 2.1.

Этапы 2 и 3 здесь необходимы для масштабирования количества данных для обучения, так как сбор демонстраций от аннотаторов не масштабируемуй и гораздо дороже, чем моделирования награды и использование PPO метода. Этот метод повысил качество модели; например, аннотаторы предпочитали ответы 175B InstructGPT перед ответами 175B GPT-3 (few-shot) в 71% случаев.

После InstructGPT методы обучения следованию инструкциям активно развивались; см., например, обзор [134]. Настоящий обзор посвящён моделям, связанным с программным кодом, и в следующем разделе мы к ним перейдём, но сначала отметим один важный общий подход.

Метод Self-Instruct [116] представляет собой экономичный, масштабируемый и практически не требующий аннотаций способ обучения следованию инструкций, что является важным шагом в сторону развития открытых больших языковых моделей. Self-Instruct основан на полуавтоматическом итеративном процессе, который схематично показан на рис. 15:

- небольшой набор вручную написанных инструкций используется для запуска порождения новых заданий с помощью обучения на небольшом числе примеров (few-shot learning);
- модель (GPT-3 в случае исходной работы [116]) создаёт, фильтрует и использует эти данные для дообучения;
- авторы различают инструкции, требующие решения задачи классификации и требующие открытого порождения текста, из-за различия в процедурах синтеза входных и выходных данных:
  - для задач классификации сначала синтезируются выходные данные (ответ), а потом подходящий к ним вход (вопрос),
  - а для задач порождения — наоборот, сначала вопрос, а затем ответ к нему; это позволяет достичь большего разнообразия в ответах;
- полученные данные фильтруются эвристическими способами, в частности по метрике текстовой близости ROUGE-L с уже имеющимся набором инструкций, чтобы полученный датасет стал разнообразнее.

Всего в [116] было получено $\sim$52K разнообразных инструкций, которые связаны с $\sim$82K примеров входных и выходных данных, т.е. хотя датасет получился достаточно обширным, он не идёт ни в какое сравнение с объёмами, требующимися для обучения больших языковых моделей.

Тем не менее, дообучение базовой модели GPT-3 на собранных инструкциях привело к увеличению точности на 33% на тестовом датасете Super-NaturalInstructions [118], что почти достигает уровня модели InstructGPT от OpenAI, использующей обширные пользовательские данные и демонстрации, произведённые аннотаторами. Авторы подчеркивают важность качества данных и разнообразия набора инструкций для улучшения обобщающей способности модели.

В последующих разделах мы увидим, как подобные методы применяются к домену программного кода.

5.3. Инструкции для задач с кодом. Во-первых, подход Self-Instruct был применён напрямую к программному домену, в частности, появился набор инструкций с кодовыми задачами CodeAlpaca [14].

Среди развивающих метод работ отметим модель WizardCoder [68], показывающую, как усложнение обучающих данных с инструкциями позволяет открытой модели обойти коммерческие решения, в частности Claude от Antropic и Bard от Google[7].

В модели WizardCoder используется методика под названием Evol-Instruct [124], адаптированная для задач программирования. Она позволяет порождать постепенно усложняющиеся и более детализированные инструкции для работы с кодом, используя модели семейства ChatGPT с различными промптами. Показано, что прирост производительности стабилизируется после трех раундов эволюции данных.

Метод эволюции инструкций (Code Evol-Instruct) включает в себя следующие шаги.

1. Изначально используется набор данных CodeAlpaca [14] с 20,000 примеров программных задач.
2. Выбирается одна задача из набора данных.
3. Инструкция в задаче усложняется одним из следующих способов:
   - добавление новых требований (например, ограничения по сложности);
   - замена стандартных условий на редкие;
   - увеличение числа логических шагов, требующихся для решения;
   - добавление ошибочного кода для отвлечения;
   - установление более высоких требований к вычислительной сложности решения (по времеми или памяти).
4. Новая инструкция добавляется к текущему набору.
5. Модель обучается на этом усложненном наборе данных.
6. Процесс повторяется несколько раз (до ухудшения результатов на тестовом наборе), чтобы создать финальный набор данных.

Показательно, что усложнение вопросов производится также языковыми моделями, причём при помощи очень простых и прямолинейных

---

[7] Разумеется, такие сравнения справедливы только на конкретную дату публикации — коммерческие закрытые решения постоянно обновляются, и их результаты со временем растут, даже если модели не получают новое название или новый порядковый номер.

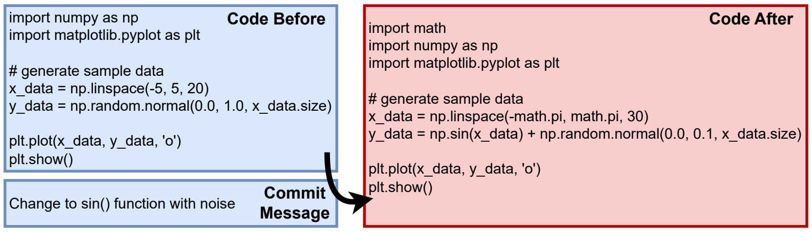
запросов; приведём их в качестве яркого примера. Для усложнения инструкций использовался следующий промпт:

```
Please increase the difficulty of the given programming test
question a bit.

You can increase the difficulty using, but not limited to,
the following methods:
{method}

{question}
```

Здесь `question` представляет текущую инструкцию, которую собираемся усложнить, а `method` — описание способа усложнения. Ниже представлены промпты для усложнения инструкций.

```
Add new constraints and requirements to the original problem,
adding approximately 10 additional words.

Replace a commonly used requirement in the programming task
with a less common and more specific one.

If the original problem can be solved with only a few logical
steps, please add more reasoning steps.

Provide a piece of erroneous code as a reference to increase
misdirection.

Propose higher time or space complexity requirements, but
please refrain from doing so frequently.
```

В работе OctoPack [72] было продемонстрировано, что данные могут быть получены без использования закрытых коммерческих моделей. А именно, набор данных *CommitPackFt* был создан из общедоступных коммитов (commits) на *GitHub*. Были собраны изменения в коде (состояние “до” и “после” коммита) вместе с сообщениями коммитов, которые служат в качестве инструкций, написанных человеком; пример такого извлечения инструкции показан на рис. 16. Для повышения качества данных в набор инструкций включались только коммиты с открытыми лицензиями и понятными сообщениями в императивном стиле; использовались изменения, затрагивающие только один файл.



```
import numpy as np
import matplotlib.pyplot as plt

# generate sample data
x_data = np.linspace(-5, 5, 20)
y_data = np.random.normal(0.0, 1.0, x_data.size)

plt.plot(x_data, y_data, 'o')
plt.show()
```

Change to sin() function with noise

```
import math
import numpy as np
import matplotlib.pyplot as plt

# generate sample data
x_data = np.linspace(-math.pi, math.pi, 30)
y_data = np.sin(x_data) + np.random.normal(0.0, 0.1, x_data.size)

plt.plot(x_data, y_data, 'o')
plt.show()
```

Рис. 16. Пример извлечения инструкции из коммита [72].

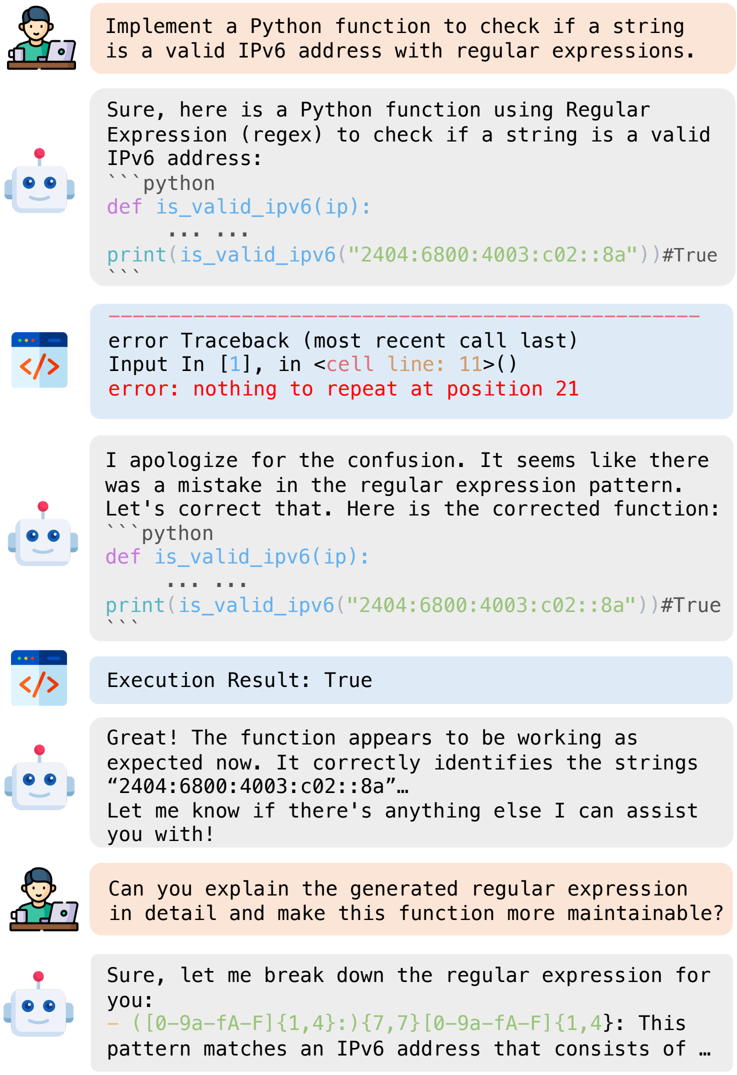
Финальный набор данных после фильтрации охватывает 277 языков программирования.

Дообучение на этом наборе инструкций позволило добиться лучшего результата для StarCoder-16B на HumanEval Python (46.2% pass@1) среди моделей, не обученных на синтетических данных от моделей OpenAI. Для обучения модели StarCoder авторы использовали низкоранговые адаптеры LoRA, описанные подробно в разделе 2.3.

5.4. Использование обратной связи от человека или компилятора. Несмотря на рассмотренные выше достижения в следовании одношаговым пользовательским инструкциям, текущие модели порождения кода имеют ограниченные способности обрабатывать обратную связь от внешнего мира в виде диалога. Обратная связь может быть представлена в двух формах:

(1) обратная связь от компилятора, включающая результаты выполнения программы и сообщения об ошибках;
(2) человеческая обратная связь, состоящая из рекомендаций или инструкций от пользователей.

Обратная связь от компилятора играет ключевую роль, позволяя моделям исправлять синтаксические и логические ошибки в программе, а человеческая обратная связь помогает моделям лучше понимать инструкции пользователей, что облегчает создание решений, которые более точно соответствуют ожиданиям пользователей. Пример такого взаимодействия показан на рис. 17.



```
Implement a Python function to check if a string
is a valid IPv6 address with regular expressions.
```

~~~
Sure, here is a Python function using Regular
Expression (regex) to check if a string is a valid
IPv6 address:
```python
def is_valid_ipv6(ip):
     ... ...
print(is_valid_ipv6("2404:6800:4003:c02::8a"))#True
```
~~~

```
-------------------------------------------------
error Traceback (most recent call last)
Input In [1], in <cell line: 11>()
error: nothing to repeat at position 21
```

~~~
I apologize for the confusion. It seems like there
was a mistake in the regular expression pattern.
Let's correct that. Here is the corrected function:
```python
def is_valid_ipv6(ip):
     ... ...
print(is_valid_ipv6("2404:6800:4003:c02::8a"))#True
```
~~~

```
Execution Result: True
```

```
Great! The function appears to be working as
expected now. It correctly identifies the strings
“2404:6800:4003:c02::8a”…
Let me know if there's anything else I can assist
you with!
```

```
Can you explain the generated regular expression
in detail and make this function more maintainable?
```

```
Sure, let me break down the regular expression for
you:
- ([0-9a-fA-F]{1,4}:){7,7}[0-9a-fA-F]{1,4}: This
pattern matches an IPv6 address that consists of …
```

Рис. 17. Пример использования обратной связи от компилятора и пользователя [141].

В частности, авторы работы OpenCodeInterpreter [141] предложили решение этой проблемы с помощью подготовки набора данных с обратной связью от компилятора и реальных людей. Получение данных выглядит следующим образом:

1. *сбор одношаговых инструкций*, где использовались:
    - запросы из открытых наборов данных (Magicoder[8], ShareGPT[9], Evol-Instruct-Code (обсуждали в разделе 5.3), LeetCode Problems[10], LeetCode Follow Up Questions);
    - фильтрация для выбора сложных запросов с помощью модели Qwen-72B-Chat (использовались запросы, получившие оценки 4–5 по шкале сложности);
2. *преобразование инструкций в диалог одним из следующих способов*:
    - упаковка одиночных запросов — объединение схожих запросов в диалоги, где похожесть определялась с помощью вложений модели BERT-base;
    - симуляция взаимодействий с компилятором или человеком — этот подход работает итеративно:
        - сначала ответ порождается более слабой моделью GPT-3.5,
        - синтезированный программный код запускается,
        - в случае ошибки диагностическая информация от компилятора с началом диалога передаётся в более сильную модель GPT-4 для исправления
        - процесс повторяется до тех пор, пока программа не выполнится успешно; аналогичная процедура выполнялась и с человеческими запросами для исправления возможных проблем в коде

Упомянутые выше наборы синтетических инструкций Magicoder и ShareGPT были получены похожим способом, как мы обсуждали в разделе 5.3. Для преобразования в диалог было выбрано 10 категорий запросов: форматирование, исправление ошибок или уязвимостей, проблемы совместимости и т.д. Кроме того, были созданы синтетические инструкции для исправления кода: с помощью GPT-4 намеренно

---

[8] https://huggingface.co/datasets/ise-uiuc/Magicoder-OSS-Instruct-75K

[9] https://huggingface.co/datasets/ajibawa-2023/Python-Code-23k-ShareGPT

[10] https://leetcode.com/problemset/

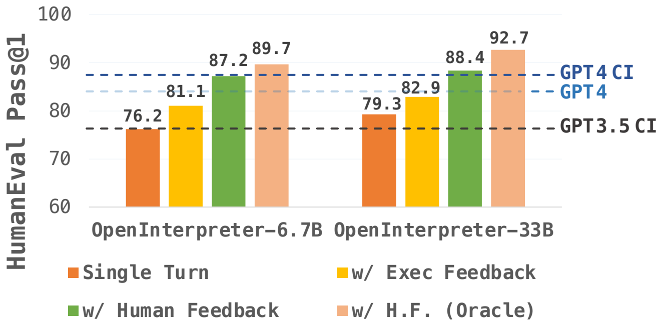


Рис. 18. Результаты OpenCodeInterpreter на HumanEval [141].

добавлялись ошибки в ответы, затем программа исполнялась, и результат выполнения снова передавался в модель для последующего исправления ошибок. Использовались также задачи и посты с форума LeetCode, где можно найти задачи с разной сложностью и решениями, а также обсуждение задач на форуме.

Эта работа важна тем, что устраняет ранее выявленный разрыв между моделями с открытым исходным кодом и лучшими коммерческими моделями, в частности GPT-4 с функцией Code Interpreter[11]. Результаты OpenCodeInterpreter на HumanEval показаны на рис. 18.

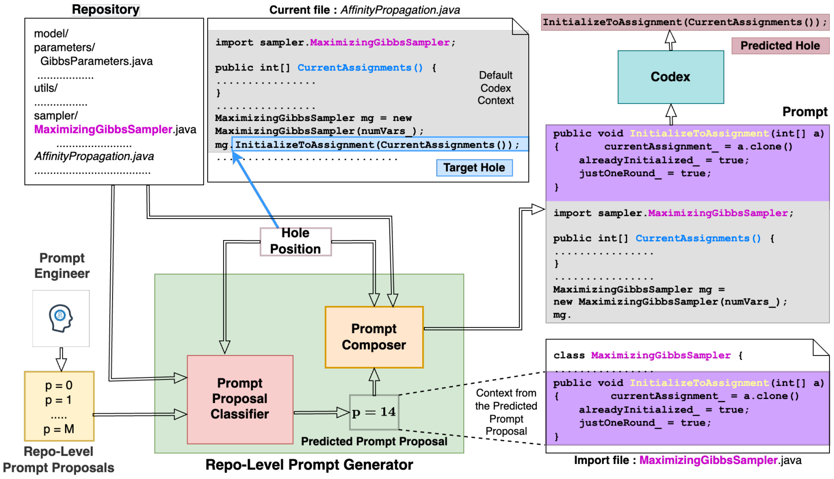
§6. Промптинг

6.1. Постановка задачи. Промптинг (*prompting*) — это метод в машинном обучении, применяемый к большим языковым моделям, который позволяет управлять поведением модели для порождения нужных ответов. Вместо того чтобы повторно обучать модель для выполнения конкретных задач, промптинг модифицирует текстовый ввод (тот самый промпт) так, чтобы выдать инструкции, примеры или контекст в форме, позволяющей получить желаемый ответ. Этот подход использует уже существующие знания модели для выполнения задач без дополнительного дообучения.

В этом разделе мы приведём несколько примеров различных методов сбора контекста для задач работы с программным кодом.

---

[11] https://platform.openai.com/docs/assistants/tools/code-interpreter



```
import sampler.MaximizingGibbsSampler;

public int[] CurrentAssignments() {
................
}
................
MaximizingGibbsSampler mg = new
MaximizingGibbsSampler(numVars_);
mg.InitializeToAssignment(CurrentAssignments());
..............................
```

```
InitializeToAssignment(CurrentAssignments());
```

```
public void InitializeToAssignment(int[] a)
{       currentAssignment_ = a.clone()
    alreadyInitialized_ = true;
    justOneRound_ = true;
}
import sampler.MaximizingGibbsSampler;

public int[] CurrentAssignments() {
................
}
................
MaximizingGibbsSampler mg =
new MaximizingGibbsSampler(numVars_);
mg.
```

```
class MaximizingGibbsSampler {
................
public void InitializeToAssignment(int[] a)
{        currentAssignment_ = a.clone()
    alreadyInitialized_ = true;
    justOneRound_ = true;
}
```

Рис. 19. Для заданного списка категорий контекста (prompt proposals), репозитория и позиции автодополнения (hole position) классификатор предсказывает релевантные участки контекста. Фиолетовым цветом выделен релевантный контекст ($p = 14$), сигнатура метода с телом из импортируемого файла [101].

6.2. Автоматическое порождение промптов. Начнём с интересного подхода порождения промптов на уровне репозитория (Repo-Level Prompt Generator, RLPG) [101], который помогает повысить качество закрытой модели Codex от OpenAI на задаче автодополнения строки кода без переобучения самой модели. Метод сначала порождает возможные участки контекста, используя структуру кода и репозитория. Далее используется классификатор для отсеивания только релевантного контекста для заданной позиции автодополнения (“дырки”, hole).

Автор выделили 10 категорий источников контекста, в том числе импортируемый файл, текущий файл, объемлющий класс, соседние файлы и т.д. Из каждого источника выделяют 7 типов контекста: декларация поля класса, сигнатура метода, сигнатура метода с телом, строковые константы и другие. Комбинация источников и типов контекста дает в результате 63 предопределённых вида участков контекста, которые RLPG может добавлять в промпт модели. Процесс синтеза контекста проиллюстрирован на рис. 19.

Для обучения классификатора были собраны тренировочные данные из 19 репозиториев с открытыми лицензиями из Google Code Archive, которые предположительно не попали в тренировочный набор данных для модели *Codex* (для которой использовались только проекты с *GitHub*). Из каждого файла удалялись пустые строки и комментарии, а “дырки” для автодополнения создавались в середине оставшихся строк (их всего 92721). Для каждого репозитория количество “дырок” ограничивалось величиной в 10,000, чтобы избежать дисбаланса. Подсказки порождались на основе различных источников и типов контекста, которые затем оценивались с использованием *Codex* для определения успешности: если автодополнение совпало с изначальной строкой, то пример получал метку 1, иначе — 0. Всего было сделано около 150K запросов к *Codex* для сбора тренировочных данных.

Использовалась следующая целевая функция:

$$\mathcal{L} = \frac{1}{N}\sum_{h=1}^{N}\mathcal{L}^h = \frac{1}{N}\sum_{h=1}^{N}\frac{1}{M^h}\sum_{p=1}^{M^h} BCE(\hat{y}_p^h, y_p^h)\cdot T_p^h$$

где $M^h = \sum_p^M T_p^h$ обозначает общее число видов контекста, применимых к данной “дырке”, N — общее число дырок в наборе данных, BCE — бинарная кросс-энтропия.

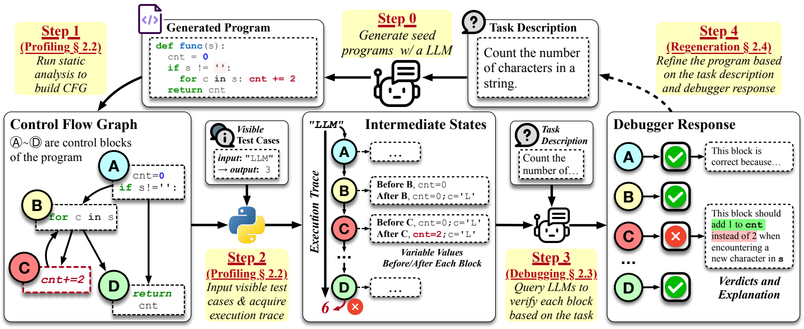
Эксперименты проводились с двумя вариантами модели:

(1) полносвязная нейронная сеть с двумя слоями для проекции вектора контекста в вероятности для каждого вида контекста:

$$\hat{y}_p^h = P(y_p^h = 1|H^h) = \sigma(W^2(\mathrm{ReLU}(W^1(F_\phi(H^h)) + b^1)) + b^2),$$

где H^h — окно вокруг “дырки” (авторы брали две строки сверху и снизу), F_ϕ — BERT-base модель, используемая для получения вектора контекста “дырки”; сама модель при этом не обучалась;

(2) механизм многоглавого внимания (multi-head attention) [112] для вычисления схожести между контекстом “дырки” и участком контекста, извлечённым одним из 63 способов, указанных



```
def func(s):
  cnt = 0
  if s != '':
    for c in s: cnt += 2
  return cnt
```

Рис. 20. Иллюстрация процесса отладки для LLM [142].

выше; полученные оценки схожести передаются в полносвязную сеть для предсказания вероятностей:

$$Q^h = F_\phi(H^h), \quad K_p^h = F_\phi(C_p^h), \quad V_p^h = F_\phi(C_p^h);$$

$$\hat{y}_p^h = P(y_p^h = 1 | H^h, C_p^h) = \sigma\left(W_p G(\mathrm{MHA}(Q^h, K^h, V_p^h)) + b_p\right),$$

где снова использовалась модель BERT-base для кодирования контекста “дырки” H^h и участков контекста C_p^h, а MHA означает многоглавое внимание, выходы которого передаются в модуль G, состоящий из двух полносвязных слоев с ReLU активацией.

Второй вариант показал себя лучше на тестовых данных. В итоге RLPG улучшил качество на 17% по сравнению с *Codex*.

6.3. Обратная связь от инструментов. Еще одним способом улучшения качества модели, который не требует дообучения, является использование в качестве обратной связи информации о компиляции или исполнении программы. Как мы уже обсуждали в разделе 5.4, такая информация помогает LLM решать сложные задачи на порождение кода за несколько шагов взаимодействия с инструментом.

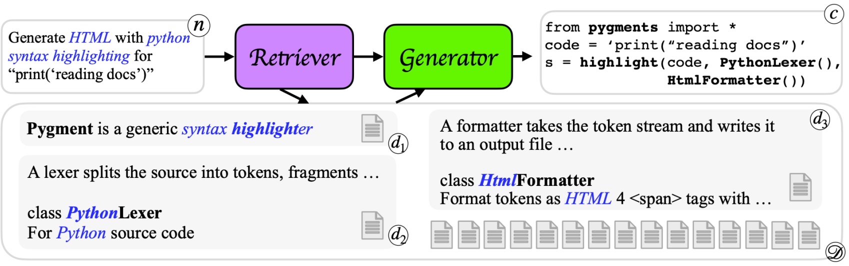
В этом разделе мы рассмотрим один из продвинутых способов сбора этой информации, а именно Large Language Model Debugger (LDB) [142]. Это новый подход для отладки программ, синтезированных большими языковыми моделями, чем-то напоминающий процесс отладки программ человеком.

Основные этапы процесса отладки выглядят следующим образом.

(1) LDB запускает инструментированную программу на публичных тестах для сбора информации о её поведении. На этапе инструментирования генерируется трасса исполнения, в которой узлы представляют собой базовые блоки (basic blocks) — последовательности инструкций с одной точкой входа и выхода. Для каждого базового блока фиксируются промежуточные значения переменных (значения до и после выполнения блока).
(2) Анализируются значения переменных для каждого базового блока с помощью LLM. Для каждого блока модель оценивает его корректность относительно описания задачи и выдает вердикт (True или False), а также пишет пояснение для ошибочных блоков. Этот процесс позволяет сосредоточить внимание модели на небольших, изолированных фрагментах кода, что упрощает диагностику ошибок и их устранение.
(3) Исправление ошибок, основывающееся на отладочных данных. Модель повторно порождает код, учитывая выявленные ошибки. Этот процесс повторяется до тех пор, пока программа не пройдет все публичные тесты или не будет достигнуто максимальное количество итераций отладки.
(4) Валидация. Итоговая версия программы тестируется на закрытых тестах для оценки её корректности.

Описанный выше процесс отладки программ для LLM показан на рис. 20. Существенным недостатком этого метода является то, что он требует наличие тестов, которые позволяют выявить несовпадение с ожиданием пользователя. Тем не менее, данный подход позволил повысить метрику качества открытой модели до 9.8% на бенчмарках HumanEval, MBPP и TransCoder.

6.4. Поиск. RAG (Retrieval Augmented Generation) — это метод сбора контекста для промпта, когда релевантная информация извлекается из внешних источников (например, база исходного кода, документация к библиотеке, веб-поиск и т.д.) и добавляется к инструкции пользователя для улучшения ответа порождающей модели. Этот метод позволяет модели использовать новые знания без обучения; существует множество различных методов и вариаций подхода RAG [6, 29, 33, 45, 56, 136, 137].



Generate *HTML* with *python syntax highlighting* for “print(‘reading docs’)”

```
from pygments import *
code = ‘print(“reading docs”)’
s = highlight(code, PythonLexer(),
              HtmlFormatter())
```

Pygment is a generic *syntax* ***highlighter***

A lexer splits the source into tokens, fragments …

class ***Python*****Lexer**
For *Python* source code

A formatter takes the token stream and writes it to an output file …

class ***Html*****Formatter**
Format tokens as *HTML* 4 <span> tags with …

Рис. 21. Иллюстрация метода DocPrompting [143].

В качестве примера применения RAG к порождению программного кода рассмотрим работу DocPrompting [143], которая добавляет релевантную документацию в промпт модели, тем самым улучшая качество на задаче синтеза программ на Python и Bash. На рис. 21 показан пример использования для языка Python: для заданного запроса на естественном языке n сначала извлекается набор релевантных документаций $\{d_1, d_2, d_3\}$ из корпуса документации D. Затем LLM создаёт код c на основе запроса n и извлечённых документов. DocPrompting позволяет модели обобщаться на ранее не встречавшиеся случаи использования, читая извлечённую документацию. На рис. 21 синим курсивом выделены общие токены между запросом и документацией; жирным шрифтом показаны общие токены между документацией и полученным фрагментом кода.

Для поиска релевантной документации можно использовать либо классическую функцию BM25 [91], либо поиск по векторам, полученным из автокодировщика RoBERTa [65]. Автокодировщик был дообучен с помощью контрастивной функции потерь:

$$\mathcal{L}^r = -\log \frac{\exp\left(\mathrm{sim}(h_n, h_{d_i^+})\right)}{\exp\left(\mathrm{sim}(h_n, h_{d_i^+})\right) + \sum_{d_j^- \in \mathcal{B}/\mathcal{D}_n^*} \exp\left(\mathrm{sim}(h_n, h_{d_i^+})\right)} \tag{14}$$

где $\mathrm{sim}(h_x, h_y)$ — косинусное расстояние между векторами h_x и h_y, h_n — векторное представление запроса, $h_{d_i^+}$ и $h_{d_i^+}$ — векторные представления релевантной и не релевантной документации соответственно. В результате метод улучшил метрики качества генерации *bash*-команд коммерческой моделью Codex на бенчмарке *tldr*.

Подобным образом можно добавлять релевантную документацию внутренних библиотек [131] или похожие примеры кода [84].

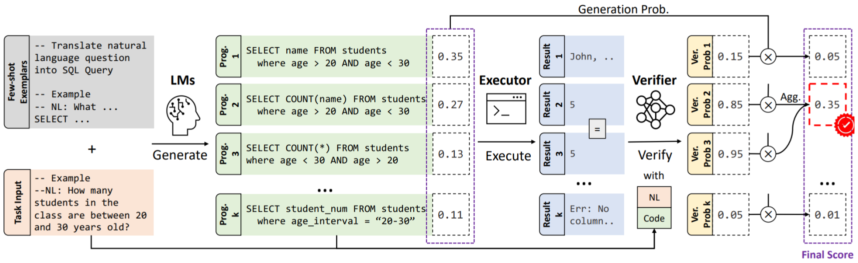
§7. Другие методы

В данном разделе мы обсуждаем несколько методов и направлений исследований, которые не относятся напрямую к предыдущим главам.

7.1. Сэмплирование с верификацией: LEVER. Данный подход, который мы описываем на примере метода LEVER (Learning to Verify Language-to-Code Generation with Execution) [76], основан на следующем наблюдении: большие языковые модели могут производить галлюцинации и часто делают ошибки при порождении кода (да и при ответах на вопросы, и при решении других задач), но если породить достаточно много вариантов ответа, среди них с куда большей вероятностью появится правильный. Это приводит к следующей идее: давайте попробуем обучить отдельную модель *верификатора* (verifier), который будет способен отличать правильные ответы (программы) от неправильных, а затем породим много вариантов ответов на вопрос большой языковой моделью и пропустим их через верификатор. Идея предполагает, что задача верификации ответа должна быть проще, чем задача его порождения с нуля.

Для других применений, например решения текстовых математических задач, такой подход был предложен в работах [22, 98] и позже развивался в работах [41, 51, 58]. Для порождения программного кода задача верификации наверняка ещё проще, чем для математических доказательств: во многих случаях ошибку можно найти уже на этапе проверки того, что программа компилируется, запускается, успешно отрабатывает и проходит простые тесты. Работы, проводящие сравнительные исследования LLM, порождающих программный код, давно отмечали, что при выборе лучшей программы из нескольких результаты обычно существенно улучшаются [7, 17, 28].

Таким образом, подход с сэмплированием и последующей верификацией является естественным способом улучшить результаты за счёт (обычно несущественного) масштабирования вычислительных ресурсов. Метод LEVER [76], предназначенный для улучшения качества порождения кода с помощью больших языковых моделей, представляет собой хороший пример успешного применения такого подхода. LEVER работает следующим образом:



```
-- Translate natural
language question
into SQL Query

-- Example
-- NL: What ...
SELECT ...
```

```
-- Example
--NL: How many
students in the
class are between 20
and 30 years old?
```

```
SELECT name FROM students
  where age > 20 AND age < 30
```

```
SELECT COUNT(name) FROM students
  where age > 20 AND age < 30
```

```
SELECT COUNT(*) FROM students
where age < 30 AND age > 20
```

```
SELECT student_num FROM students
  where age_interval = "20-30"
```

```
Err: No
column..
```

Рис. 22. Обучение верификатора порождённого кода на основе результатов его запуска (LEVER) [76].

(1) *порождение*: из LLM сэмплируется множество программ-кандидатов $\{\boldsymbol{y}_i\}_{i=1}^{n}$ на основе входного описания задачи $\boldsymbol{x}$; для выборки используется температурное сэмплирование

$$\boldsymbol{y}_i \sim p_{\mathrm{LM}}\left(\boldsymbol{y}|\boldsymbol{x}\right),$$

где p_{LM} — распределение вероятностей, обученное LLM; на данном этапе можно сэмплировать с повышенной температурой, чтобы увеличить разнообразие ответов;

(2) *исполнение*: каждая программа $\boldsymbol{y}_i$ исполняется с помощью интерпретатора или компилятора, и результаты исполнения $E(\boldsymbol{y}_i)$ добавляются к входным данным;

(3) *верификация*: обучается дискриминативная модель

$$p_\theta\left(v|\boldsymbol{x}, \boldsymbol{y}, E(\boldsymbol{y})\right),$$

где $v \in \{0, 1\}$ обозначает корректность программы; входом для верификатора служат
   - описание задачи $\boldsymbol{x}$,
   - код программы $\boldsymbol{y}$ и
   - результат её выполнения $E(\boldsymbol{y})$.

Эта последовательность шагов проиллюстрирована на рис. 22.

При применении метода программы ранжируются по совокупному счёту, основанному на вероятности генерации модели и верификации. Верификатор используется для вычисления совокупной вероятности корректности:

$$p_R\left(\boldsymbol{y}|\boldsymbol{x}\right) = p_{\mathrm{LM}}\left(\boldsymbol{y}|\boldsymbol{x}\right) \cdot p_\theta\left(v = 1|\boldsymbol{x}, \boldsymbol{y}, E(\boldsymbol{y})\right).$$

Чтобы результаты не слишком зависели от таких поверхностных факторов, как имена переменных или внешний вид программы, программы с одинаковыми результатами исполнения $E(\boldsymbol{y})$ агрегируются:

$$R(\boldsymbol{x}, \boldsymbol{y}) = \sum_{\boldsymbol{y}' \in S : E(\boldsymbol{y}') = E(\boldsymbol{y})} p_R\left(\boldsymbol{y}' | \boldsymbol{x}\right).$$

Программа с максимальным значением $R(\boldsymbol{x}, \boldsymbol{y})$ выбирается в качестве финального результата:

$$\hat{\boldsymbol{y}} = \arg\max_{\boldsymbol{y} \in S} R(\boldsymbol{x}, \boldsymbol{y}).$$

Для обучения верификатора используются автоматически аннотированные данные. Каждая программа-кандидат $\boldsymbol{y}$ аннотируется меткой v, определяемой сравнением её результата $E(\boldsymbol{y})$ с эталонным значением E^*:

$$v = \begin{cases} 1, & \text{если } E(\boldsymbol{y}) = E^*, \\ 0, & \text{иначе.} \end{cases}$$

Функция потерь верификатора определяется как

$$L_\theta = -\frac{1}{|S|} \sum_{\boldsymbol{y} \in S} \log p_\theta\left(v | \boldsymbol{x}, \boldsymbol{y}, E(\boldsymbol{y})\right).$$

В результате метод LEVER использует результаты исполнения для получения плотной (dense) обратной связи, что повышает точность верификации, эффективно работает в условиях ограниченных данных, показывая улучшения даже при небольшом размере обучающей выборки, и в результате повышает уровень точности на задачах порождения кода, в частности для SQL-парсинга, математических задач и программирования на Python.

LEVER, конечно, не единственный метод, использующий сэмплирование большого числа программ с последующей верификацией. Можно сказать, что система *AlphaCode*, достигшая в 2022 году человеческого уровня на платформе *CodeForces*, тоже работает подобным образом [61], хотя масштабирование вычислительных затрат там гораздо более существенное. Как пример дальнейшего развития этой идеи можно привести работу [59], где сэмплирование из LLM производится с особенным образом построенным перебором запросов. В целом это направление улучшения результатов зачастую может быть применено практически «бесплатно» (особенно если простой верификатор в виде набора юнит-тестов уже существует), и его использование представляется целесообразным во многих практических сценариях.

7.2. Агенты, основанные на LLM. Большая языковая модель может действовать как агент в некоторой окружающей среде, взаимодействуя с ней при помощи текстовых команд. Такие агентные подходы активно развиваются в разных областях применения LLM. Например, *ReAct* [128] применяет агентов, чтобы улучшить способности LLM к рассуждениям, *WebGPT* [73] обучает LLM-агента интерактивно взаимодействовать с веб-браузером, *SayCan* [2] применяет LLM-агентов в роботике, а работы [46, 132] строят LLM-агентов для диалоговых рекомендательных систем. Работа [83] рассматривает агентов как "симуляции людей", которые затем можно использовать для получения синтетических данных; такие подходы уже активно развиваются в рекомендательных системах, где исследователи строят целые виртуальные миры, населённые такими агентами [113, 114, 132]. Последние новости развития коммерческих LLM уже внедрили основанных на LLM агентов в свои сервисы: теперь LLM от OpenAI или Anthropic могут и искать информацию в интернете, и даже интерактивно взаимодействовать с компьютером пользователя на основе последовательных снимков экрана.

В качестве примера применения основанных на LLM агентов к программированию рассмотрим две недавние работы, которые уже успели получили известность и множество цитирований. Метод *Reflexion* [99] предлагает новый подход к обучению агентов на основе больших языковых моделей (LLM), где процесс обучения основан на самоанализе и вербальной обратной связи, а не на обновлении весов. Вместо традиционного обновления параметров модели через градиентный спуск *Reflexion* усиливает агента через сохранение и использование текстовых рефлексий в памяти.

Reflexion использует три модуля:

- *актор* (actor, M_a) порождает текст и действия, взаимодействуя с окружением;
- *оценщик* (evaluator, M_e) оценивает качество действий и выдает сигнал вознаграждения;
- *модель рефлексии* (self-reflection, M_{sr}) порождает текстовые рефлексии на основе траекторий, результатов и сигналов вознаграждения.

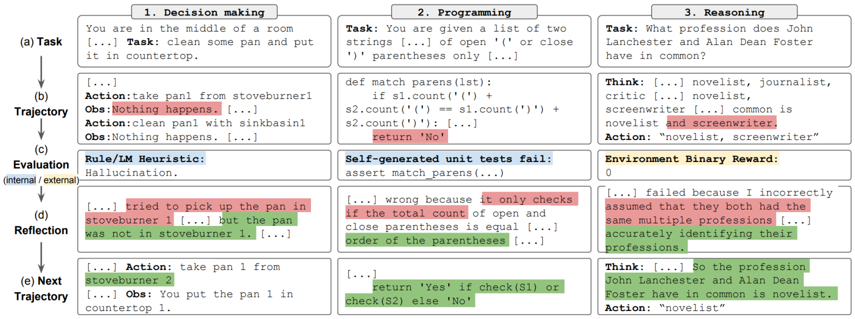


Рис. 23. Иллюстрация метода *Reflexion* [99].

Каждый шаг обучения, проиллюстрированный на рис. 23, состоит из следующих этапов:

(1) *порождение траектории:* актор M_a производит траекторию $\tau_t = [s_0, a_0, s_1, a_1, \ldots, s_T]$, взаимодействуя с окружением;
(2) *оценка траектории:* оценщик M_e вычисляет скалярное вознаграждение $r_t = M_e(\tau_t)$;
(3) *порождение рефлексии:* модель M_{sr} генерирует текстовую рефлексию sr_t, которая сохраняется в долгосрочную память агента;
(4) *обновление стратегии:* в следующих эпизодах агент использует накопленные рефлексии для улучшения действий.

Важным элементом архитектуры агента в *Reflexion* является память, разделённая на два типа:

- *краткосрочная память* хранит текущую траекторию τ_t;
- *долгосрочная память* хранит текстовые рефлексии $[\mathrm{sr}_0, \mathrm{sr}_1, \ldots, \mathrm{sr}_t]$; она ограничена фиксированным числом записей (небольшим, обычно 1–3).

Модель рефлексии интегрирует эти воспоминания для порождения последующих действий, что позволяет агенту делать выводы из прошлых ошибок.

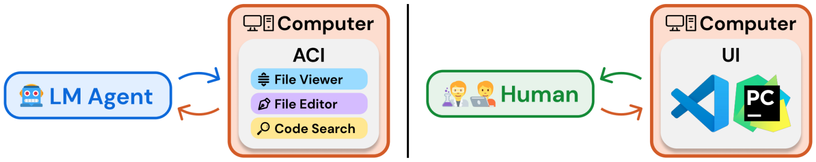
Таким образом, на задачах программирования *Reflexion* обучается через сгенерированные тесты и самоанализ: сначала актор M_a порождает код, затем он оценивается как по результатам компиляции и прохождения тестов, так и моделью оценщика M_e, затем модель M_{sr} порождает текстовую рефлексию, описывающую, какие изменения нужно внести в код, а затем код исправляется на основе рефлексии, и цикл повторяется снова. Такой итеративный подход приносит существенные улучшения: например, на бенчмарке HumanEval *Reflexion* увеличил точность Pass@1 на 11%, до 91%, превзойдя GPT-4, а в других бенчмарках улучшения ещё более серьёзные: успешность решений в AlfWorld [100] выросла на 22%, а решения задач на рассуждения в HotpotQA [127] улучшились на 20%.

Подход на основе агентов с текстовой рефлексией имеет и другие важные преимущества:

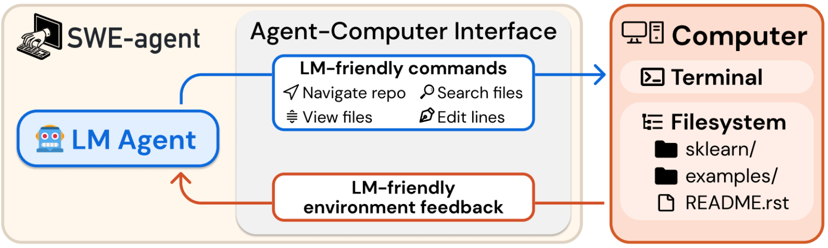
- *интерпретируемость*, поскольку текстовые рефлексии предоставляют явную человекочитаемую информацию о том, какие действия были ошибочными и как их исправить;
- *отсутствие необходимости в дообучении*, ведь метод работает без обновления параметров модели, используя только текстовые подсказки;
- *гибкость*, поскольку *Reflexion* подходит и для программирования, и для принятия решений и рассуждений, и для агентного планирования — адаптироваться к разным типам обратной связи легко, потому что само улучшение агента происходит в любом случае в текстовом виде.

Таким образом, *Reflexion* открывает перспективное направление, комбинирующее мощь LLM с вербальным обучением на основе самоанализа.

Как пример дальнейшего развития агентного подхода приведём недавно разработанную систему *SWE-agent* [126], предназначенную для автоматизации задач разработки программного обеспечения (software engineering, SWE) с использованием языковых моделей в качестве агентов. Основная идея метода заключается в создании специализированного интерфейса взаимодействия между агентом и компьютером, называемого *Agent-Computer Interface* (ACI), который позволяет агенту эффективно решать задачи, связанные с написанием, изменением и тестированием кода.



(а) интерфейс ACI (б) человеческий интерфейс



(в) компоненты интерфейса ACI

Рис. 24. Иллюстрация метода *SWE-agent* [126].

LLM функционирует как агент, взаимодействуя с компьютерной средой через действия (например, редактирование файлов, запуск тестов) и получая обратную связь от окружения. В отличие от традиционных человеко-ориентированных интерфейсов (например, терминалов или IDE), ACI использует

- компактные команды с минимальными параметрами,
- информативную, но лаконичную обратную связь об изменениях, сделанных агентом, и
- механизмы защиты от распространённых ошибок LLM (например, проверка синтаксиса при редактировании).

Это позволяет устранить ограничения стандартных интерфейсов, которые могут быть избыточно сложными для моделей (см. сравнение на рис. 24а и 24б).

SWE-agent включает следующие основные модули (рис. 24в):

- *навигация по кодовой базе:* специальные команды, такие как `find_file` или `search_dir`, позволяют агенту быстро находить релевантные файлы и строки;

- *редактирование файлов:* команда `edit` позволяет агенту заменять диапазоны строк кода, автоматически отображая обновлённое содержимое для проверки;
- *контекстное управление:* система управляет историей действий и наблюдений агента, чтобы минимизировать избыточность в предоставляемых данных;
- *обратная связь:* каждый шаг сопровождается обратной связью о результатах действий агента, включая ошибки синтаксиса или другие проблемы.

На каждом шаге агент порождает сначала “мыслительный процесс” (thought), т.е. высказывание агента о его следующем действии, а затем само действие, конкретную команду, выполняемую в окружении. После выполнения действия агент получает обратную связь, которая помогает ему корректировать своё поведение. Вот пример последовательности работы агента [126]:

- агент использует команду `search_file` для поиска упоминания функции `PVSystem`;
- после нахождения нужного файла он открывает его с помощью `open` и просматривает содержимое;
- агент вносит правки в строки с помощью команды `edit`, при этом синтаксические ошибки автоматически выявляются и возвращаются агенту для исправления;
- после завершения изменений запускается тестирование с помощью `pytest`.

SWE-agent показывает значительные улучшения по сравнению с предыдущими подходами:

- на наборе задач SWE-bench он достигает 12.47% решённых задач, что значительно превосходит результат 3.8%, показанный лучшей из неинтерактивных моделей;
- на бенчмарке HumanEvalFix SWE-agent достигает уровня *87.7%* по метрике Pass@1, что также значительно выше базовых показателей;
- анализ показывает, что правильно спроектированный ACI значительно повышает производительность модели без изменения её внутренних параметров.

SWE-agent иллюстрирует эффективность подхода, объединяющего языковые модели и специализированные интерфейсы, делая их мощным инструментом для автоматизации программной инженерии. В целом, нам представляется, что будущее автоматизированной программной инженерии именно за агентными подходами, комбинирующими выразительность, гибкость и способность к рассуждениям, которые уже сегодня демонстрируют языковые модели, с возможностью итеративной отладки и улучшения в некоторой среде разработки; как показывает SWE-agent, эта среда может быть успешно адаптирована для LLM.

§8. Примеры индустриальных кодовых моделей

8.1. Введение. В этой главе мы рассмотрим, как выглядит процесс дообучения (fine-tuning, post-training) для популярных индустриальных кодовых моделей с открытыми весами. Как мы уже не раз видели в этом обзоре, этап дообучения направлен на преобразование базовой модели в полезного помощника программиста. Так же, как и в базовой работе об InstructGPT (см. раздел 2.1 и саму работу [67]), обычно этот процесс делится на два основных шага: обучение следованию инструкциям (см. раздел 5) и согласование целей и задач модели с человеческими (AI alignment), обычно методами обучения с подкреплением (см. раздел 4).

Несмотря на то, что программный код и данные используемые для обучения в большинстве случаев остаются закрытыми от научного сообщества, имеющаяся информация о процессе дообучения этих моделей может быть полезной для дальнейшей их специализации под конкретную практическую задачу, а также может служить иллюстрацией эффективности тех или иных методов.

В этом разделе мы рассмотрим дообучение на примере двух моделей: Qwen2.5-Coder [47] и DeepSeek-Coder-V2 [26]; мы видели их сравнительные результаты на *LiveCodeBench* в таблицах 10 и 2, где видно, что варианты этих моделей занимают лидирующие позиции на кодовых бенчмарках наравне с коммерческими решениям от OpenAI (семейство GPT) и Anthropic (семейство Claude).

8.2. Qwen2.5-Coder. Шаг *обучения следованию инструкциям* (instruction tuning) для модели Qwen2.5-Coder [47] состоял из следующих этапов.

Рис. 25. Фаза дообучения для модели Qwen2.5-Coder [47].

(1) Обучение на 10 миллионах разнообразных синтезированных инструкциях низкого качества (coarse fine-tuning).
(2) Обучение на 1 миллионе более качественных инструкций для повышения качества (fine fine-tuning). Для этого использовали метод сэмплирования с отклонением (rejection sampling), в котором с помощью LLM синтезировали разные ответы для одной и той же инструкции, а потом отфильтровывали плохие ответы.
(3) Обучение на инструкциях с заполнением пропусков (fill-in-the-middle, FIM [8]). Ввиду того, что инструкции в основном короткие, добавление инструкций с FIM в обучение позволяет сохранить способность модели использовать длинный контекст. Для этого использовали автоматические парсеры из библиотеки *tree-sitter-languages*[12], которые позволяют вырезать из файла различные конструкции с точки зрения языка программирования — выражение, функцию или кодовый блок. После этого с соответствующей инструкцией модели требуется восстановить вырезанный кусочек кода с учетом контекста.

На шаге *согласования* (alignment) использовался метод прямой оптимизации предпочтений (direct preference optimization, DPO) [89], где поведение модели корректируется на основе обратной связи, полученной от исполнения тестов, а также оценок качества другими языковыми моделями; последний метод известен как LLM-as-a-judge [139]. Данный процесс, проиллюстрированный на рис. 25, помогает улучшить качество синтеза кода и соответствие пользовательским ожиданиям.

---

[12] https://pypi.org/project/tree-sitter-languages/

Подготовка данных для дообучения включает сложный процесс сбора, очистки и анализа данных для создания высококачественного набора инструкций.

Сначала были собраны исходные данные из разных источников, включая открытые репозитории GitHub и существующие наборы инструкций для кода, в частности McEval-Instruct[13]. Использовался специально обученный классификатор на основе модели CodeBERT [30], который позволял либо определить, какой язык программирования использовался в данных, либо определить отсутствие кода. Были использованы 100 самых популярных языков программирования.

Для увеличения объема и качества данных использовались синтетические инструкции, созданные при помощи больших языковых моделей. Эти инструкции синтезировались по кодовым фрагментам с GitHub, а затем фильтровались с помощью LLM для повышения качества данных.

Дополнительно была разработана мультиагентная система для создания новых инструкций на разных языках программирования. Агентам назначались языковые специализации, и они могли обмениваться знаниями друг с другом, чтобы улучшать общий корпус данных. Также агентам предоставлялась внешняя память для предотвращения дублирования информации. Такой подход способствовал переиспользованию знаний и концепций программирования между разными языками программирования.

Все данные подвергались финальной оценке на основе чек-листа, который включал такие критерии, как релевантность, сложность, наличие комментариев, образовательная ценность и другие критерии. На завершающем этапе данные по возможности проходили проверку с использованием изолированных окружений для выполнения кода. Этот процесс включал проверку синтаксиса, автоматическое создание модульных тестов и проверку на прохождение этих тестов.

Эти этапы подготовки данных обеспечили создание качественного и многоязычного набора инструкций.

8.3. DeepSeek-Coder-V2. В модели DeepSeek-Coder-V2 [26] на шаге *обучения следованию инструкциям* использовался набор данных, содержащий задачи по программированию и по математике. Сначала авторы выбрали 20 тысяч инструкций по программированию и 30

---

[13] https://huggingface.co/datasets/Multilingual-Multimodal-NLP/McEval-Instruct

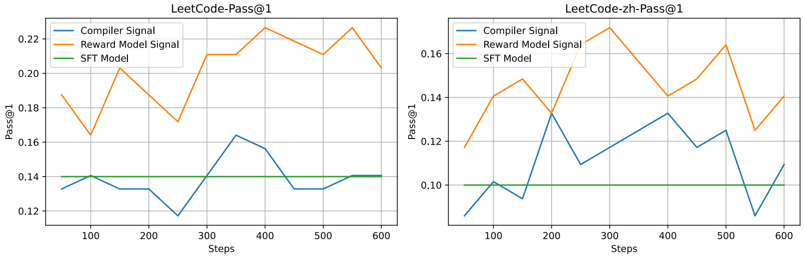


Рис. 26. Метрика качества Pass@1 для разных наград в DeepSeek-Coder-V2 [26].

тысяч математических инструкций из DeepSeek-Coder [35] и DeepSeek-Math [97] соответственно. Чтобы сохранить общие способности модели, добавили часть данных из инструкций для DeepSeek-V2 [25]. Итоговый набор данных содержал 300 миллионов токенов.

На шаге *согласования* использовался метод оптимизации стратегий относительно групп (Group Relative Policy Optimization, GRPO) [97], модификация PPO, менее затратная с точки зрения требующейся памяти. Для этой фазы были также отобраны инструкции с задачами по математике и программированию. Для задач по программированию были написаны модульные тесты. После фильтрации набор данных содержал около 40 тысяч инструкций.

В качестве функции награды для математических задач использовалось сравнение с правильным ответом. Для задач по программированию прохождение тестов уже является бинарной наградой, но некоторые инструкции могут содержать недостаточное количество тестов, не обеспечивающее полного покрытия. Поэтому бинарная величина “прохождение тестов” может быть шумной и не оптимальной для обучения.

По этим соображениям авторы приняли решение обучить модель награды на данных, полученных после запуска тестов, и использовать эту модель награды в процессе RL обучения. Это обеспечивает менее шумный сигнал по сравнению с обратной связью от прохождения тестов. Как показано на рис. 26, на внутренних тестовых данных (Leetcode и Leetcode-zh) использование модели награды для обучения с подкреплением явно превосходит использование данных о прохождении тестов.

Модели Qwen2.5-Coder [47] и DeepSeek-Coder-V2 [26] в настоящее время находятся на переднем крае моделей искусственного интеллекта, работающих с программным кодом; их могут превосходить некоторые закрытые варианты моделей OpenAI и Anthropic, но принципиальной разницы в качестве работы нет (если не говорить о только что появившемся семействе моделей *o1* и ему подобных, которые в этом обзоре мы рассматривать не будем). Поэтому интересно, что опубликованная информация о работе этих моделей хотя и содержит некоторые новые идеи, в целом полностью покрывается основными подходами, рассмотренными в настоящем обзоре.

§9. Заключение

В настоящим обзоре мы рассмотрели современные методы применения больших языковых моделей для порождения программного кода. Основное внимание было уделено подходам дообучения (fine-tuning, post-training), включая построение и обучение адаптеров (раздел 2), обучение с подкреплением (раздел 4), дообучение следованию инструкциям (раздел 5), промптинг (раздел 6) и другие подходы (раздел 7), позволяющие улучшить качество генерации и адаптировать модели к специфическим задачам, связанным с программным кодом; мы также рассмотрели два конкретных подхода к дообучению, применяющихся в лучших на сегодня открытых моделях (раздел 8).

Рассмотренные методы показали, что LLM обладают значительным потенциалом для задач программирования, включая порождение, исправление и оптимизацию кода. Тем не менее, в заключение мы выделяем несколько открытых вопросов, точнее, целых направлений, которые всё ещё требуют дальнейших исследований.

(1) *Разработка более надёжных методов оценки качества моделей, учитывающих реальные сценарии применения.* Текущие методы оценки, такие как pass@k или строгая точность, хорошо работают для стандартных бенчмарков, но не всегда отражают реальное использование моделей в разработке программного обеспечения. Например, реальные задачи программирования часто связаны с многократными итерациями, взаимодействием с пользователем и изменением требований, что плохо моделируется существующими метриками. Для улучшения

здесь было бы полезно разработать методы, учитывающие такие аспекты, как интерактивность, способность модели объяснять свои действия, а также анализ сложных ошибок.

(2) *Минимизация зависимости от больших вычислительных ресурсов.* Современные большие языковые модели требуют огромных вычислительных ресурсов как для обучения, так и для применения, что ограничивает их использование в реальных приложениях, особенно на локальных или ресурсозависимых устройствах. Методы PEFT уже хорошо работают (этому посвящена большая часть обзора), но их производительность всё ещё уступает полному дообучению, и представляется важным продолжать исследования, направленные на разработку ещё более эффективных стратегий, которые позволят достичь сопоставимого качества при значительно меньших затратах.

(3) *Улучшение обработки сложных и нестандартных случаев, включая задачи с ограниченными данными или уникальными требованиями.* Модели зачастую хорошо работают на стандартных задачах из своих обучающих выборок, но испытывают трудности при столкновении с новыми, нестандартными сценариями. Например, задачи, требующие глубокого контекста, специализированных знаний или обработки уникальных структур данных, часто оказываются за пределами возможностей моделей. Здесь важно развивать подходы, способные эффективно обучаться на малом числе примеров (few-shot learning), использовать мультимодальные входы или адаптироваться к уникальным требованиям пользователя.

(4) *Предотвращение потенциальных рисков, связанных с порождением некорректного или вредоносного кода.* Большие языковые модели могут непреднамеренно (или даже преднамеренно, хотя это отдельный большой вопрос) порождать код, содержащий уязвимости, некорректную логику или даже вредоносные элементы. Эти риски усиливаются, если модели применяются без должной валидации. Пока нет надёжных методов автоматического обнаружения и предотвращения подобных сценариев, и требуются решения, которые смогут анализировать порождённый код на наличие таких проблем, учитывать контекст задачи и ограничивать порождения опасного или уязвимого кода.

(5) *Интеграция LLM с инструментами разработки и создание гибридных систем, использующих возможности мультимодальных моделей.* Несмотря на успехи таких инструментов, как GitHub Copilot, текущая интеграция LLM с процессами разработки остаётся на ранней стадии. В частности, отсутствует глубокая интеграция с инструментами отладки, системами контроля версий и другими компонентами DevOps. Кроме того, мультимодальные возможности (например, совместное использование текста, кода, графики) пока используются слабо, что ограничивает потенциал этих моделей. Развитие гибридных систем, которые объединили бы разные источники данных и способы взаимодействия, станет важным шагом для повышения их практической ценности.

Современные LLM уже оказывают значительное влияние на разработку программного обеспечения, автоматизируя рутинные задачи и открывая новые возможности для создания программных продуктов. Однако их потенциал далеко не исчерпан, и мы уверены, что дальнейшее развитие этих технологий способно привести к революционным изменениям в индустрии разработки программного обеспечения.

Список литературы

1. A. Aghajanyan, S. Gupta, L. Zettlemoyer, *Intrinsic dimensionality explains the effectiveness of language model fine-tuning*, Proc. 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers) (Online, 2021), Association for Computational Linguistics, August 2021, pp. 7319–7328.
2. M. Ahn et al., *Do as I can, not as I say: Grounding language in robotic affordances*, arXiv preprint arXiv:2204.01691 (2022).
3. E. L. Aleixo, J. G. Colonna, M. Cristo, E. Fernandes, *Catastrophic forgetting in deep learning: A comprehensive taxonomy*, arXiv preprint arXiv:2312.10549 (2023).
4. Anthropic, *Claude 3.5 sonnet*, `https://www.anthropic.com/news/claude-3-5-sonnet`, 2024.
5. Anthropic, *Introducing the next generation of Claude*, `https://www.anthropic.com/news/claude-3-family`, 2024.
6. A. Asai, S. Min, Z. Zhong, D. Chen, *Retrieval-based language models and applications*, Proc. 61st Annual Meeting of the Association for Computational Linguistics (Volume 6: Tutorial Abstracts) (Toronto, Canada, 2023), Association for Computational Linguistics, July 2023, pp. 41–46.
7. J. Austin et al., *Program synthesis with large language models*, arXiv preprint arXiv:2108.07732 (2021).

8. M. Bavarian, H. Jun, N. Tezak, J. Schulman, C. McLeavey, J. Tworek, M. Chen, *Efficient training of language models to fill in the middle*, arXiv preprint arXiv:2207.14255 (2022).
9. J. Baxter, P. L. Bartlett, *Infinite-horizon policy-gradient estimation.* — J. Artif. Int. Res. **15**, No. 1 (2001), 319–350.
10. R. A. Bradley, M. E. Terry, *Rank analysis of incomplete block designs. I. The method of paired comparisons.* — Biometrika **39** (1952), 324–245.
11. T. Brown et al., *Language models are few-shot learners*, Advances in Neural Information Processing Systems, Vol. 33, Curran Associates, Inc., 2020, pp. 1877–1901.
12. X.-Q. Cai, Y.-J. Zhang, C.-K. Chiang, M. Sugiyama, *Imitation learning from vague feedback*, Proc. 37th International Conference on Neural Information Processing Systems (Red Hook, NY, USA, 2024), Curran Associates Inc.
13. Y. Cao et al., *A comprehensive survey of AI-generated content (AIGC): A history of generative AI from GAN to ChatGPT*, arXiv preprint arXiv:2303.04226 (2023).
14. S. Chaudhary, *Code Alpaca: An instruction-following LLaMA model for code generation*, `https://github.com/sahil280114/codealpaca`, 2023.
15. A. Chen et al., *Learning from natural language feedback.* — Trans. Mach. Learn. Res. (2024).
16. A. Chen et al., *Improving code generation by training with natural language feedback*, arXiv preprint arXiv:2303.16749 (2024).
17. M. Chen et al., *Evaluating large language models trained on code*, arXiv preprint arXiv:2107.03374 (2021).
18. P. F. Christiano et al., *Deep reinforcement learning from human preferences*, Advances in Neural Information Processing Systems, Vol. 30, Curran Associates, Inc., 2017.
19. A. Chronopoulou, M. Peters, J. Dodge, *Efficient hierarchical domain adaptation for pretrained language models*, Proc. 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Seattle, United States, 2022), Association for Computational Linguistics, July 2022, pp. 1336–1351.
20. A. Chronopoulou, M. Peters, A. Fraser, J. Dodge, *AdapterSoup: Weight averaging to improve generalization of pretrained language models*, Findings of the Association for Computational Linguistics: EACL 2023 (Dubrovnik, Croatia, 2023), Association for Computational Linguistics, May 2023, pp. 2054–2063.
21. H. W. Chung et al., *Scaling instruction-finetuned language models.* — J. Mach. Learn. Res. **25**, No. 70 (2024), 1–53.
22. K. Cobbe et al., *Training verifiers to solve math word problems*, arXiv preprint arXiv:2110.14168 (2021).
23. N. Dai et al., *Process supervision-guided policy optimization for code generation*, arXiv preprint arXiv:2410.17621 (2024).
24. M. De Lange et al., *A continual learning survey: Defying forgetting in classification tasks.* — IEEE Trans. Pattern Anal. Mach. Intell. **44**, No. 7 (2022), 3366–3385.
25. DeepSeek-AI et al., *DeepSeek-v2: A strong, economical, and efficient mixture-of-experts language model*, arXiv preprint arXiv:2405.04434 (2024).

26. DeepSeek-AI et al., *DeepSeek-Coder-v2: Breaking the barrier of closed-source models in code intelligence*, arXiv preprint arXiv:2406.11931 (2024).
27. N. Ding et al., *Sparse low-rank adaptation of pre-trained language models*, The 2023 Conference on Empirical Methods in Natural Language Processing (2023).
28. X. Du et al., *Evaluating large language models in class-level code generation*, Proc. IEEE/ACM 46th International Conference on Software Engineering (ICSE '24, New York, NY, USA, 2024), Association for Computing Machinery.
29. W. Fan et al., *A survey on RAG meeting LLMs: Towards retrieval-augmented large language models*, arXiv preprint arXiv:2405.06211 (2024).
30. Z. Feng et al., *CodeBERT: A pre-trained model for programming and natural languages*, Findings of the Association for Computational Linguistics: EMNLP 2020 (Online, 2020), Association for Computational Linguistics, November 2020, pp. 1536–1547.
31. R. M. French, *Catastrophic forgetting in connectionist networks.* — Trends Cogn. Sci. **3**, No. 4 (1999), 128–135.
32. J. Fürnkranz, E. Hüllermeier, W. Cheng, S.-H. Park, *Preference-based reinforcement learning: A formal framework and a policy iteration algorithm.* — Mach. Learn. **89**, No. 1–2 (2012), 123–156.
33. Y. Gao et al., *Retrieval-augmented generation for large language models: A survey*, arXiv preprint arXiv:2312.10997 (2024).
34. J. Gehring et al., *RLEF: Grounding code LLMs in execution feedback with reinforcement learning*, arXiv preprint arXiv:2410.02089 (2024).
35. D. Guo et al., *DeepSeek-Coder: When the large language model meets programming — the rise of code intelligence*, arXiv preprint arXiv:2401.14196 (2024).
36. S. Gururangan, M. Lewis, A. Holtzman, N. A. Smith, L. Zettlemoyer, *DEMix layers: Disentangling domains for modular language modeling*, Proc. 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Seattle, United States, 2022), Association for Computational Linguistics, July 2022, pp. 5557–5576.
37. D. Ha, A. M. Dai, Q. V. Le, *Hypernetworks*, International Conference on Learning Representations (2017).
38. Z. Han et al., *Parameter-efficient fine-tuning for large models: A comprehensive survey*, arXiv preprint arXiv:2403.14608 (2024).
39. J. He, C. Zhou, X. Ma, T. Berg-Kirkpatrick, G. Neubig, *Towards a unified view of parameter-efficient transfer learning*, International Conference on Learning Representations (2022).
40. J. Hoffmann et al., *Training compute-optimal large language models*, arXiv preprint arXiv:2203.15556 (2022).
41. A. Hosseini et al., *V-star: Training verifiers for self-taught reasoners*, arXiv preprint arXiv:2402.06457 (2024).
42. N. Houlsby et al., *Parameter-efficient transfer learning for NLP*, Proc. 36th International Conference on Machine Learning, Vol. 97, Proceedings of Machine Learning Research, PMLR, 2019, pp. 2790–2799.
43. E. J. Hu et al., *LoRA: Low-rank adaptation of large language models*, International Conference on Learning Representations (2022).

44. Y. Hu, Y. Xie, T. Wang, M. Chen, Z. Pan, *Structure-aware low-rank adaptation for parameter-efficient fine-tuning.* — Math. **11**, No. 20 (2023).
45. Y. Hu, Y. Lu, *RAG and RAU: A survey on retrieval-augmented language model in natural language processing*, arXiv preprint arXiv:2404.19543 (2024).
46. X. Huang et al., *Recommender AI agent: Integrating large language models for interactive recommendations*, arXiv preprint arXiv:2308.16505 (2024).
47. B. Hui et al., *Qwen2.5-Coder technical report*, arXiv preprint arXiv:2409.12186 (2024).
48. D. R. Hunter, *MM algorithms for generalized Bradley-Terry models.* — Ann. Stat. **32**, No. 1 (2004), 384–406.
49. A. Jain, S. Sharma, T. Joachims, A. Saxena, *Learning preferences for manipulation tasks from online coactive feedback.* — Int. J. Rob. Res. **34**, No. 10 (2015), 1296–1313.
50. N. Jain et al., *LiveCodeBench: Holistic and contamination-free evaluation of large language models for code*, arXiv preprint arXiv:2403.07974 (2024).
51. S. Kadavath et al., *Language models (mostly) know what they know*, arXiv preprint arXiv:2207.05221 (2022).
52. J. Kaplan et al., *Scaling laws for neural language models*, arXiv preprint arXiv:2001.08361 (2020).
53. J. Lehman et al., *The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities.* — Artif. Life **26**, No. 2 (2020), 274–306.
54. T. Lei et al., *Conditional adapters: Parameter-efficient transfer learning with fast inference*, Advances in Neural Information Processing Systems, Vol. 36, Curran Associates, Inc., 2023, pp. 8152–8172.
55. C. Li, H. Farkhoor, R. Liu, J. Yosinski, *Measuring the intrinsic dimension of objective landscapes*, International Conference on Learning Representations (2018).
56. H. Li et al., *A survey on retrieval-augmented text generation*, arXiv preprint arXiv:2202.01110 (2022).
57. J. Li, P. Zhou, C. Xiong, S. Hoi, *Prototypical contrastive learning of unsupervised representations*, International Conference on Learning Representations (2021).
58. Y. Li et al., *Making language models better reasoners with step-aware verifier*, Proc. 61st Annual Meeting of the Association for Computational Linguistics (Toronto, Canada, 2023), Association for Computational Linguistics, July 2023, pp. 5315–5333.
59. Y. Li, J. Parsert, E. Polgreen, *Guiding enumerative program synthesis with large language models*, Computer Aided Verification (Cham, 2024), Springer Nature Switzerland, pp. 280–301.
60. Y. Li et al., *Competition-level code generation with AlphaCode.* — Science **378**, No. 6624 (2022), 1092–1097.
61. Y. Li et al., *Competition-level code generation with AlphaCode.* — Science **378**, No. 6624 (2022), 1092–1097, `https://www.science.org/doi/pdf/10.1126/science.abq1158`.
62. J. Liu et al., *RLTF: Reinforcement learning from unit test feedback.* — Trans. Mach. Learn. Res. (2023).

63. J. Liu et al., *Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation*, arXiv preprint arXiv:2305.01210 (2023).
64. S.-Y. Liu et al., *DORA: Weight-decomposed low-rank adaptation*, arXiv preprint arXiv:2402.09353 (2024).
65. Y. Liu et al., *RoBERTa: A robustly optimized BERT pretraining approach*, arXiv preprint arXiv:1907.11692 (2019).
66. Z. Liu et al., *ALoRA: Allocating low-rank adaptation for fine-tuning large language models*, arXiv preprint arXiv:2403.16187 (2024).
67. R. Lowe, J. Leike, *Aligning language models to follow instructions*, `https://openai.com/index/instruction-following/`, 2022.
68. Z. Luo et al., *WizardCoder: Empowering code large language models with Evol-Instruct*, arXiv preprint arXiv:2306.08568 (2023).
69. M. Masana et al., *Class-incremental learning: Survey and performance evaluation on image classification*. — IEEE Trans. Pattern Anal. Mach. Intell. **45**, No. 5 (2023), 5513–5533.
70. M. McCloskey, N. J. Cohen, *Catastrophic interference in connectionist networks: The sequential learning problem*, Psychology of Learning and Motivation, Vol. 24, Academic Press, 1989, pp. 109–165.
71. S. Minaee et al., *Large language models: A survey*, arXiv preprint arXiv:2402.06196 (2024).
72. N. Muennighoff et al., *Octopack: Instruction tuning code large language models*, arXiv preprint arXiv:2308.07124 (2024).
73. R. Nakano et al., *WebGPT: Browser-assisted question-answering with human feedback*, arXiv preprint arXiv:2112.09332 (2022).
74. H. Naveed et al., *A comprehensive overview of large language models*, arXiv preprint arXiv:2307.06435 (2024).
75. H. D. Nguyen, F. Chamroukhi, *Practical and theoretical aspects of mixture-of-experts modeling: An overview*. — WIREs Data Min. Knowl. Discov. **8**, No. 4 (2018), e1246.
76. A. Ni et al., *LEVER: Learning to verify language-to-code generation with execution*, Proc. 40th International Conference on Machine Learning, JMLR.org, 2023.
77. OpenAI, *Introducing ChatGPT*, `https://openai.com/index/chatgpt/`, 2022.
78. OpenAI, *GPT-4 technical report*, arXiv preprint arXiv:2303.08774 (2023).
79. L. Ouyang et al., *Training language models to follow instructions with human feedback*, arXiv preprint arXiv:2203.02155 (2022).
80. L. Ouyang et al., *Training language models to follow instructions with human feedback*, Advances in Neural Information Processing Systems, Vol. 35, Curran Associates, Inc., 2022, pp. 27730–27744.
81. P. Pan et al., *Continual deep learning by functional regularisation of memorable past*, Proc. 34th International Conference on Neural Information Processing Systems, Curran Associates Inc., 2020.
82. G. I. Parisi et al., *Continual lifelong learning with neural networks: A review*. — Neural Netw. **113** (2019), 54–71.

83. J. S. Park et al., *Generative agents: Interactive simulacra of human behavior*, arXiv preprint arXiv:2304.03442 (2023).
84. M. R. Parvez et al., *Retrieval augmented code generation and summarization*, Findings of the Association for Computational Linguistics: EMNLP 2021 (Punta Cana, Dominican Republic), November 2021, pp. 2719–2734.
85. J. Pfeiffer et al., *AdapterFusion: Non-destructive task composition for transfer learning*, Proc. 16th Conference of the European Chapter of the Association for Computational Linguistics (Online, 2021), Association for Computational Linguistics, April 2021, pp. 487–503.
86. B. Pfülb, A. Gepperth, *A comprehensive, application-oriented study of catastrophic forgetting in DNNs*, International Conference on Learning Representations, 2019.
87. M. Popović, *chrF: Character n-gram F-score for automatic MT evaluation*, Proc. Tenth Workshop on Statistical Machine Translation (Lisbon, Portugal), September 2015, pp. 392–395.
88. M. Post, *A call for clarity in reporting BLEU scores*, Proc. Third Conference on Machine Translation: Research Papers (Brussels, Belgium), October 2018, pp. 186–191.
89. R. Rafailov et al., *Direct preference optimization: Your language model is secretly a reward model*, arXiv preprint arXiv:2305.18290 (2024).
90. S. Rebuffi et al., *iCaRL: Incremental classifier and representation learning*, 2017 IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), 2017, pp. 5533–5542.
91. S. E. Robertson, K. S. Jones, *Relevance weighting of search terms.* — J. Am. Soc. Inf. Sci. **27** (1976), 129–146.
92. R. Rombach et al., *High-resolution image synthesis with latent diffusion models*, Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. (CVPR), 2022, pp. 10684–10695.
93. B. Roziere et al., *Unsupervised translation of programming languages*, Proc. 34th International Conference on Neural Information Processing Systems, Curran Associates Inc., 2020.
94. J. Schulman et al., *Trust region policy optimization*, Proc. 32nd International Conference on Machine Learning (Lille, France), Vol. 37, PMLR, July 2015, pp. 1889–1897.
95. J. Schulman et al., *Proximal policy optimization algorithms*, arXiv preprint arXiv:1707.06347 (2017).
96. J. Schulman et al., *Proximal policy optimization algorithms*, arXiv preprint arXiv:1707.06347 (2017).
97. Z. Shao et al., *DeepSeekMath: Pushing the limits of mathematical reasoning in open language models*, arXiv preprint arXiv:2402.03300 (2024).
98. J. Shen et al., *Generate & rank: A multi-task framework for math word problems*, Findings of the Association for Computational Linguistics: EMNLP 2021 (Punta Cana, Dominican Republic), November 2021, pp. 2269–2279.
99. N. Shinn et al., *Reflexion: Language agents with verbal reinforcement learning*, Proc. 37th International Conference on Neural Information Processing Systems, Curran Associates Inc., 2024.

100. M. Shridhar et al., *ALFWORLD: Aligning text and embodied environments for interactive learning*, arXiv preprint arXiv:2010.03768 (2021).
101. D. Shrivastava et al., *Repository-level prompt generation for large language models of code*, arXiv preprint arXiv:2206.12839 (2023).
102. A. C. Stickland, I. Murray, *BERT and PALs: Projected attention layers for efficient adaptation in multi-task learning*, Proc. 36th International Conference on Machine Learning (Long Beach, CA, USA), Vol. 97, PMLR, 2019, pp. 5986–5995.
103. N. Stiennon et al., *Learning to summarize with human feedback*, Advances in Neural Information Processing Systems, Vol. 33, Curran Associates Inc., 2020, pp. 3008–3021.
104. R. S. Sutton, A. G. Barto, *Reinforcement learning: An introduction*, MIT Press, 2018.
105. R. S. Sutton et al., *Policy gradient methods for reinforcement learning with function approximation*, Proc. 12th International Conference on Neural Information Processing Systems, MIT Press, 1999, pp. 1057–1063.
106. Gemini Team et al., *Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context*, arXiv preprint arXiv:2403.05530 (2024).
107. E. Todorov, T. Erez, Y. Tassa, *MuJoCo: A physics engine for model-based control*, 2012 IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS), 2012, pp. 5026–5033.
108. H. Touvron et al., *LLaMA: Open and efficient foundation language models*, arXiv preprint arXiv:2302.13971 (2023).
109. H. Touvron et al., *LLaMA 2: Open foundation and fine-tuned chat models*, arXiv preprint arXiv:2307.09288 (2023).
110. M. Valipour et al., *DyLoRA: Parameter-efficient tuning of pre-trained models using dynamic search-free low-rank adaptation*, Proc. 17th Conference of the European Chapter of the Association for Computational Linguistics (Dubrovnik, Croatia), May 2023, pp. 3274–3287.
111. A. van den Oord, Y. Li, O. Vinyals, *Representation learning with contrastive predictive coding*, arXiv preprint arXiv:1807.03748 (2018).
112. A. Vaswani et al., *Attention is all you need*, arXiv preprint arXiv:1706.03762 (2017).
113. L. Wang et al., *User behavior simulation with large language model based agents*, arXiv preprint arXiv:2306.02552 (2024).
114. Y. Wang et al., *RecMind: Large language model powered agent for recommendation*, Findings of the Association for Computational Linguistics: NAACL 2024 (Mexico City, Mexico), June 2024, pp. 4351–4364.
115. Y. Wang et al., *AdaMix: Mixture-of-adaptations for parameter-efficient model tuning*, Proc. 2022 Conference on Empirical Methods in Natural Language Processing (Abu Dhabi, UAE), December 2022, pp. 5744–5760.
116. Y. Wang et al., *Self-Instruct: Aligning language models with self-generated instructions*, arXiv preprint arXiv:2212.10560 (2023).
117. Y. Wang et al., *Super-NaturalInstructions: Generalization via declarative instructions on 1600+ NLP tasks*, arXiv preprint arXiv:2204.07705 (2022).

118. Y. Wang et al., *Super-NaturalInstructions: Generalization via declarative instructions on 1600+ NLP tasks*, Proc. 2022 Conference on Empirical Methods in Natural Language Processing (Abu Dhabi, UAE), December 2022, pp. 5085–5109.
119. F. R. Ward, F. Toni, F. Belardinelli, *On agent incentives to manipulate human feedback in multi-agent reward learning scenarios*, Proc. 21st International Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2022, pp. 1759–1761.
120. C. J. H. Watkins, P. Dayan, *Q-learning*. — Mach. Learn. **8**, No. 3–4 (1992), 279–292.
121. E. Wiewiora, *Reward shaping*, Encyclopedia of Machine Learning (Springer US), 2010, pp. 863–865.
122. R. J. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*. — Mach. Learn. **8**, No. 3–4 (1992), 229–256.
123. C. Wirth et al., *A survey of preference-based reinforcement learning methods*. — J. Mach. Learn. Res. **18**, No. 136 (2017), 1–46.
124. C. Xu et al., *WizardLM: Empowering large language models to follow complex instructions*, arXiv preprint arXiv:2304.12244 (2023).
125. Y. Xu et al., *Preference-based reinforcement learning with finite-time guarantees*, Proc. 34th International Conference on Neural Information Processing Systems, Curran Associates Inc., 2020.
126. J. Yang et al., *SWE-Agent: Agent-computer interfaces enable automated software engineering*, arXiv preprint arXiv:2405.15793 (2024).
127. Z. Yang et al., *HotpotQA: A dataset for diverse, explainable multi-hop question answering*, arXiv preprint arXiv:1809.09600 (2018).
128. S. Yao et al., *ReAct: Synergizing reasoning and acting in language models*, Int. Conf. Learn. Representations (ICLR), 2023.
129. Z. Yu et al., *$\mathcal{B}$-Coder: Value-based deep reinforcement learning for program synthesis*, Twelfth International Conference on Learning Representations (ICLR), 2024.
130. S. E. Yuksel et al., *Twenty years of mixture of experts*. — IEEE Trans. Neural Netw. Learn. Syst. **23**, No. 8 (2012), 1177–1193.
131. D. Zan et al., *When language model meets private library*, Findings of the Association for Computational Linguistics: EMNLP 2022 (Abu Dhabi, UAE), December 2022, pp. 277–288.
132. A. Zhang et al., *On generative agents in recommendation*, arXiv preprint arXiv:2310.10108 (2024).
133. Q. Zhang et al., *AdaLoRA: Adaptive budget allocation for parameter-efficient fine-tuning*, arXiv preprint arXiv:2303.10512 (2023).
134. S. Zhang et al., *Instruction tuning for large language models: A survey*, arXiv preprint arXiv:2308.10792 (2024).
135. H. Zhao et al., *Prototype-based HyperAdapter for sample-efficient multi-task tuning*, Proc. 2023 Conference on Empirical Methods in Natural Language Processing (Singapore), December 2023, pp. 4603–4615.
136. P. Zhao et al., *Retrieval-augmented generation for AI-generated content: A survey*, arXiv preprint arXiv:2402.19473 (2024).

137. R. Zhao et al., *Retrieving multimodal information for augmented generation: A survey*, arXiv preprint arXiv:2303.10868 (2023).
138. W. X. Zhao et al., *A survey of large language models*, arXiv preprint arXiv:2303.18223 (2024).
139. L. Zheng et al., *Judging LLM-as-a-judge with MT-Bench and Chatbot Arena*, arXiv preprint arXiv:2306.05685 (2023).
140. R. Zheng et al., *Secrets of RLHF in large language models part I: PPO*, arXiv preprint arXiv:2307.04964 (2023).
141. T. Zheng et al., *OpenCodeInterpreter: Integrating code generation with execution and refinement*, arXiv preprint arXiv:2402.14658 (2024).
142. L. Zhong et al., *Debug like a human: A large language model debugger via verifying runtime execution step-by-step*, arXiv preprint arXiv:2402.16906 (2024).
143. S. Zhou et al., *DocPrompting: Generating code by retrieving the docs*, arXiv preprint arXiv:2207.05987 (2023).
144. B. Zhu et al., *Principled reinforcement learning with human feedback from pairwise or k-wise comparisons*, Proc. 40th International Conference on Machine Learning (ICML), JMLR.org, 2023.
145. Y. Zhu et al., *Counter-interference adapter for multilingual machine translation*, Findings of the Association for Computational Linguistics: EMNLP 2021 (Punta Cana, Dominican Republic), November 2021, pp. 2812–2823.

Lomshakov V. M., Nikolenko S. I. Large Language Models for Source Code Generation and Editing.

In recent years, large language models (LLMs) have significantly transformed approaches to the automation of software development, providing powerful tools for code generation, correction, and optimization. In this survey, we examine methods for adapting LLMs to programming tasks, including reinforcement learning from human feedback (RLHF), instruction tuning, parameter-efficient fine-tuning (PEFT), and effective prompting strategies. We review modern approaches for fine-tuning and LLM applications, discuss their advantages and limitations, consider relevant datasets for code generation and correction tasks and the corresponding evaluation metrics. Additionally, we describe state of the art open weight models for working with source code.

Поступило 15 ноября 2024 г.

Санкт-Петербургское отделение
Математического института им. В. А. Стеклова РАН, Санкт-Петербург, Россия
E-mail: `vadim.lomshakov@gmail.com`

Университет ИТМО;
Санкт-Петербургское отделение Математического института им. В. А. Стеклова РАН, Санкт-Петербург, Россия
E-mail: `sergey@logic.pdmi.ras.ru`