МЕТОДИКА РАЗРАБОТКИ АВТОМАТИЗИРОВАННЫХ СРЕДСТВ ГЕНЕРАЦИИ ПРОГРАММНОГО КОДА ПОСРЕДСТВОМ НАСТРОЙКИ БОЛЬШИХ ЯЗЫКОВЫХ МОДЕЛЕЙ

Самонов А. В.[1], Бурова И. О.[2]

DOI: 10.21681/2311-3456-2024-3-68-75

Цель исследования: *разработка методического, алгоритмического и программного обеспечения для создания автоматизированных средств генерации программного обеспечения на основе больших языковых моделей.*

Методы исследования: *анализ архитектуры, методов и средств создания, обучения и применения больших языковых моделей, исследование методов и алгоритмов точной настройки и применения больших языковых моделей для генерации программного кода, экспериментальные исследования разработанных алгоритмов и программ на стенде.*

Полученные результаты: *исследованы архитектурные и технологические основы построения и функционирования больших языковых моделей (Large Language Model, LLM). Определены перспективные технологии, методы и средства обучения и точной настройки LLM на решение задач в области программирования. Разработана методика создания автоматизированных средств генерации программного кода посредством реализации итерационной процедуры настройки ограниченного количества значимых параметров базовой LLM на специально подготовленных обучающих наборах данных. Определены ключевые модули и параметры процедуры настройки LLM. Представлены фрагменты программной реализации методики в среде Pytorch. Полученные в ходе экспериментов результаты свидетельствуют о целесообразности применения данного подхода для разработки автоматизированных средств генерации программного кода.*

Научная и практическая значимость: *состоит в разработке методического, алгоритмического и программного обеспечения, предназначенного для создания при ограниченных вычислительных ресурсах на основе больших языков моделей автоматических средств генерации и тестирования программного кода, в которых отсутствуют катастрофическое забывание, риск переобучения, галлюцинации.*

Ключевые слова: *большие языковые модели, глубокое обучение, внимание на себя, нейросетевые модели, трансформер, Large Language Model, self-attention, transformer*

METHODOLOGY FOR THE DEVELOPMENT OF AUTOMATED SOFTWARE CODE GENERATION TOOLS BY FINE-TUNING LARGE LANGUAGE MODELS

Samonov A. V.[3], Burova I. O.[4]

The purpose of research: *the development of methodological, algorithmic and software for the creation of automated software generation tools based on large language models*

Research methods: *analysis of architecture, methods and means of creating, teaching and applying large language models, research of methods and algorithms for fine-tuning and applying large language models to generate program code, experimental studies of developed algorithms and programs on the stand.*

1 Самонов Александр Валерьянович, кандидат технических наук, доцент, старший научный сотрудник, Военно-космическая академия имени А. Ф. Можайского, Санкт-Петербург, Россия. E mail: a.samonov@mail.ru, ORCID: 0000-0002-0390-4481.

2 Бурова Ирина Олеговна младший научный сотрудник, Военно-космическая академия имени А. Ф. Можайского, Санкт-Петербург, Россия, E-mail: burova@smilecom.ru

3 Alexander V. Samonov, Ph.D. in technical sciences, associate professor, senior research scientist Mozhaiskiy Military Space Academy St.Petersburg, Russia. E mail: a.samonov@mail.ru, ORCID: 0000-0002-0390-4481

4 Irina O. Burova. research scientist Mozhaiskiy Military Space Academy St.Petersburg, Russia. E mail: burova@smilecom.ru

The results obtained: *the architectural and technological foundations of the construction and functioning of large language models (LLM) are investigated. Promising technologies, methods and tools for teaching and fine-tuning LLM to solve programming problems have been identified. A methodology has been developed for creating automated software code generation tools by implementing an iterative procedure for configuring a limited number of significant parameters of the basic LLM on specially prepared training datasets. The key modules and parameters of the LLM setup procedure are defined. Fragments of the software implementation of the technique in the Pytorch environment are presented. The results obtained during the experiments indicate the expediency of using this approach to develop automated software code generation tools.*

Scientific and practical significance: *it consists in the development of methodological, algorithmic and software designed to create, with limited computing resources, models of automatic means of generating and testing software code based on large languages models, in which there is no catastrophic forgetting, the risk of retraining, hallucinations.*

Keywords: *large language models, deep learning, neural network models, transformer, Large Language Model, self-attention.*

Введение

Современный этап мирового развития характеризуется активным внедрением в индустрию, науку, образование и другие сферы хозяйственной и общественной жизни технологий искусственного интеллекта. Яркими примерами таких технологий являются методы глубокого обучения и генеративный искусственный интеллект. Созданные с их помощью большие языковые модели (Large Language Model, LLM) и программные комплексы способны обрабатывать и создавать тексты, понимать и синтезировать речь, изображения, музыку, генерировать программный код, решать аналитические, математические и другие нетривиальные задачи. Наиболее известными коммерческими LLM, используемыми в области разработки программного обеспечения, являются: GPT-4, GPT-3.5, Claude 2, Palm 2, AlphaCode 2 [1]. Примерами LLM с открытым исходным кодом, предназначенных для решения задач аналогичного класса, являются: Code Llama, WizardCoder, PhindCodeLlama, StarCoder, CodeGen, CodeGeeX и ряд других [2]. Данные системы могут обнаружить и помочь устранить ошибки в программном коде, предложить вариант решения типовой задачи, а также самостоятельно сгенерировать программный код на таких языках как Python, C++, Java, JavaScript, Go и др. На рис. 1 представлены результаты тестирования возможностей средств генерации программного кода с помощью тестовых наборов HumanEval и MBPP [3]. Как видно из данного графика, целый ряд средств успешно справляются с половиной и более тестовых заданий.

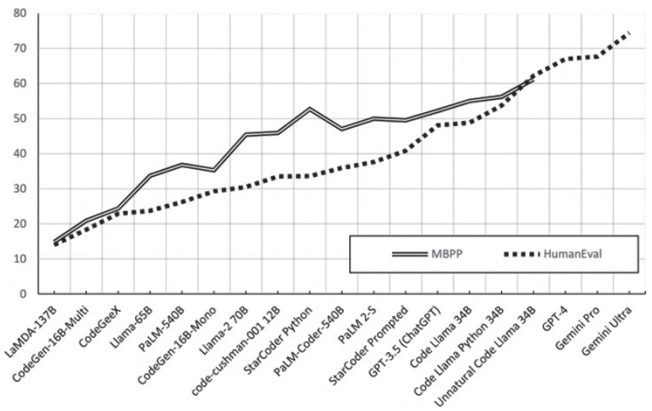

Рис. 1. Результаты сравнительного анализа производительности программных пакетов, используемых для генерации программного кода

Разработка таких средств осуществляется посредством настройки больших языковых моделей на решение задач в области программирования. Как показал проведенный анализ, современные средства генерации программного кода пока способны разрабатывать только относительно простые программы и преимущественно на языке Python. Основными проблемными вопросами, с которыми сталкиваются разработчики систем данного класса, являются: катастрофическое забывание, риск переобучения, галлюцинации создаваемых систем, а также исключительно высокие требования к производительности используемых при обучении LLM вычислительных средств.

В данной статье представлены предложения по совершенствованию и развитию методического, алгоритмического и программного обеспечения процессов создания средств генерации программного кода на основе больших языковых моделей, позволяющие преодолеть существующие трудности и ограничения.

Архитектура, технологии и алгоритмы обучения и функционирования больших языковых моделей

Технологическая цепочка процессов и средств, реализуемых и используемых при создании базовой модели LLM и последующей ее специализации, представлена на рисунке 2. На начальном этапе модель

обучается на неструктурированных и немаркированных данных. Основными источниками данных, которые были использованы при разработке большинства современных LLM, являются: Wikipedia, Common Crawl, BooksCorpus (коллекция текстов книг), OpenWebText (набор статей из сети интернет). В результате получается предварительно обученная (pretrained) LLM общего назначения. Примерами таких моделей являются GPT 4, GPT 3.5, Gemini, Falcon, Llama, Mixtral. На втором этапе осуществляется доработка LLM посредством самостоятельного обучения на специальным образом подготовленных и маркированных данных, настраивающая модель на решение задач определенного класса. На третьем этапе такие модели проходят дополнительное обучение с подкреплением на основе обратной связи с экспертом (Reinforcement Learning from Human Feedback). Примерами наборов данных, используемых при разработке средств генерации и тестирования программного кода, являются CodeTextBook, sql-create-context. В результате создаются предметно-ориентированные программные системы ИИ (СИИ), предназначенные для решения конкретных практических задач в определенных областях. При адаптации LLM на решение задач в области программирования используются такие источники как Github и StackOverflow, наборы данных MathQA-Python, MBPP, APPS, DS1000. В результате создается специализированная (обученная и проверенная) СИИ, включающая несколько взаимосвязанных нейросетей и программных средств, обеспечивающих их настройку и применение для решения задач в области программирования.

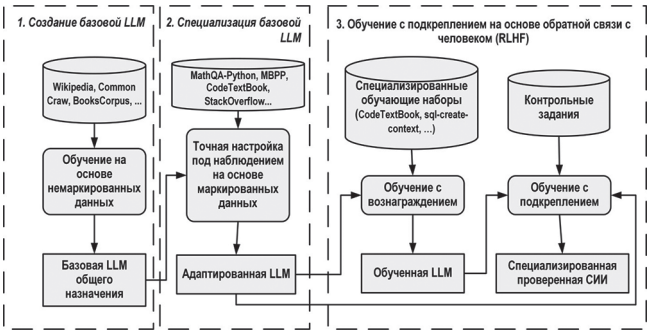

Рис.2. Технологическая цепочка процессов и средств, используемых при создании СИИ

Бурное развитие и широкое применение LLM во многом обязано двум используемым в них техническим решениям: архитектуре «трансформер» (transformer) и механизму «внимание на себя» (self-attention). Состав, структура и механизмы функционирования основных компонентов трансформера представлены на рисунке 3 [4, 5].

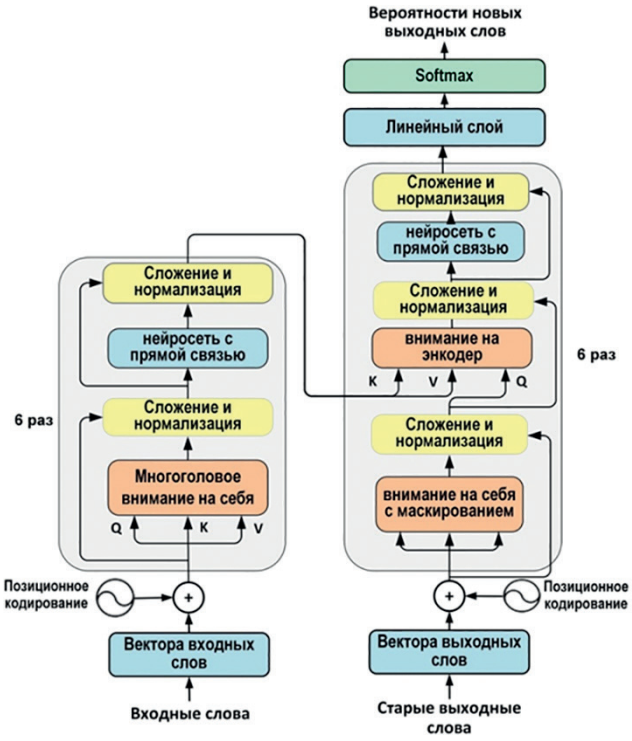

Рис.3. Архитектура трансформера

Устройство трансформера состоит из кодирующего и декодирующего компонентов. На вход принимается некая последовательность, создается ее векторное представление (англ. embedding), прибавляется вектор позиционного кодирования, после чего набор элементов без учета их положения в последовательности поступает в кодирующий компонент (encoder), а затем декодирующий компонент (decoder) получает на вход часть этой последовательности и выход кодирующего. В результате получается новая выходная последовательность. Трансформер-кодировщик переводит исходные векторы в скрытые, которые правильно сохраняют в себе информацию о контексте каждого элемента. Каждый слой энкодера включает следующие модули: внимание на себя (self-attention), сложение и нормализацию (add&normalize), нейросеть прямого распространения (FFN, feed-forward neural network).

Далее трансформер-декодировщик декодирует результат кодировщика в новую последовательность, которая состоит из эмбедингов элементов выходного языка. По эмбедингам генерируются итоговые элементы с помощью вероятностной языковой модели. Результаты работы энкодера принимает модуль декодера «внимание на энкодер». При получении данных модуль декодера формирует запрос Q из данных модуля «внимание на себя с маскированием» и ищет соответствующие ему ключи K и значения V. Модуль «внимание на энкодер» передает работу в нейросеть прямого распространения. Данные проходят через

шесть слоев декодера, которые включают такие же, как и в энкодере модули. С последнего слоя декодера результат попадает на заключительные модули – линейный слой и softmax. Данная процедура выполняется до тех пор, пока входная матрица декодера не заполнится до конца и не сгенерируется сигнал остановки.

Сердцем трансформера является работа модуля «внимание к себе», с помощью которого трансформер определяет контекст обрабатываемого в данный момент слова (токена) и определяет степень его близости с другими словами (токенами) входного набора данных. Первым шагом при вычислении «внимания к себе» является создание из входного вектора трех векторов: $Q(x)$ – запроса, $K(x)$ – ключа и $V(x)$ – значения. Эти векторы создаются путем умножения входного вектора на соответствующие им матрицы: W_Q, W_K и W_V, которые были рассчитаны при обучении LLM [4]:

$$Q_{(x)} = x \cdot W_Q + b_k \quad (1)$$

$$K_{(x)} = x \cdot W_K + b_q \quad (2)$$

$$V_{(x)} = x \cdot W_V + b_v, \quad (3)$$

где W_Q, W_K, $\in \mathrm{R}^{\text{input} \times \text{key}}$, $W_V \in \mathrm{R}^{\text{input} \times \text{val}}$, b_Q, $b_K \in \mathrm{R}^{\text{input} \times \text{key}}$, $b_V \in \mathrm{R}^{\text{input} \times \text{val}}$. (4)

Один набор Q, K и V может отражать только один вид зависимостей между токенами, и матрицы извлекают лишь ограниченный набор информации из входных представлений. Чтобы компенсировать эту неоптимальность, в классическую архитектуру трансформера вместо одного слоя внимания включили несколько параллельных с разными весами. Используя тензорную нотацию, процедуру вычисления «многоголового внимания к себе» (MHA, multi-head attention) можно представить в виде следующих формул [6, 7]:

$$MHA(Q,K,V) = Concat(head_1,\ldots,head_h)\ W^O, \quad (5)$$

где Q, K, V – матрицы запросов, ключей и значений соответственно, а W^O матрица с весовыми коэффициентами без смещения на выходе;

$head_i = Attention(xW_iQ, xW_iK, xW_iV)$;

$Attention\ (Q_i,K_i,V_i) = softmax(\frac{Q_i K_i^T}{\sqrt{dk}})\ V_i$;

W_i^Q, W_i^K, W_i^V – матрицы весов для i-й головы модуля внимания.

Вторая часть трансформера – нейросеть прямого распространения (FFN, feed-forward neural network) представляет собой два обычных полносвязных слоя, применяемых независимо к каждому элементу входной последовательности. FFN берет вектор x (скрытое представление в определенной позиции последовательности) и пропускает его через два изученных линейных преобразования (представленных матрицами W_1 и W_2 и векторами смещения b_1 и b_2). Между двумя линейными преобразованиями применяется функция активации:

$$FFN(x) = act(xW_1 + b_1)\ W_2 + b_2. \quad (6)$$

В качестве функции активации act используются ReLU (Rectified Linear Unit), GELU (Gaussian Error Linear Unit) или SwiGLU [8]. Функция Softmax нормализует оценки, чтобы все они были положительными и в сумме равнялись 1.

В результате создается базовая LLM, включающая несколько взаимосвязанных нейросетей и программных средств. Нейросети представлены матрицами W_Q, W_K, W_V. Программные средства включают функции кодирования, декодирования, многоголосового внимания, сложения, нормализации, линеаризации и др. Далее на основе базовой LLM можно создать специализированную систему, предназначенную для решения предметно-ориентированных задач.

Методы и средства создания специализированных интеллектуальных средств для автоматической генерации программного кода

Настройка и специализация больших языковых моделей требует обновления сотен миллионов и миллиардов параметров и сохранения больших копий весовых коэффициентов для каждой задачи, что приводит к увеличению затрат на хранение, совместное использование и обслуживание моделей. При обучении LLM используется гораздо больше памяти, чем при простом ее размещении на графическом процессоре. Это связано с тем, что во время обучения память используется для следующих компонентов LLM: весов модели, состояний оптимизатора, градиентов, переадресации активаций, сохраненных для вычисления градиента, временных буферов. С целью сокращения этих затрат разработаны и используются методы точной настройки значимых (эффективных) параметров (PEFT, Parameter-Efficient Fine-Tuning). Описание современных методов и средств точной настройки LLM представлено в [9–14].

Методы точной настройки эффективных параметров LLM, в отличие от полной настройки модели, обеспечивают обучение только небольшого набора параметров, которые могут быть подмножеством существующих параметров модели или набором добавленных параметров. Эти методы различаются значимостью параметров, эффективностью использования памяти, скоростью обучения, конечным качеством модели и возможными дополнительными затратами при настройке и использовании по назначению. Методы PEFT позволяют повысить корректность и производительность предварительно обученных языковых моделей при решении задач

из определенной области. Основными преимуществами использования методов PEFT являются: сокращение времени обучения, снижение затрат на вычисления и хранение моделей, снижение риска переобучения, преодоление катастрофического забывания, удобство развертывания и переноса на другие устройства.

Наиболее известными методами тонкой настойки являются: LoRA, Prefix tuning, Prompt tuning и Adapters [9–16]. LoRa (Low-Rank Adaptation) – адаптация низкого ранга фокусируется на изменении весов только определенных слоев и параметров базовой LLM, ориентируясь на те, которые наиболее эффективны (полезны) для решения задач данного класса. Это достигается путем применения для настройки весов матриц, имеющих существенно меньший ранг, чем матрицы базовой модели. LoRa применяется только к матрицам запросов и значений трансформера, что означает, что многослойный персептрон заморожен и адаптируются только веса внимания. Функция потерь оптимизируется путем передачи градиента через замороженную модель в адаптеры. Метод QLoRA (Quantization-Aware Low-Rank Adaptation) разработан для развертывания моделей в средах с ограниченными ресурсами. Он позволяет значительно снизить требования к необходимым для развертывания и настройки моделей объемам и производительности памяти GPU и CPU, а также мощностям вычислителей. На рисунке 4 представлена упрощенная схема трансформера (слева) с включенным в него адаптером, реализующим метод LoRA (справа).

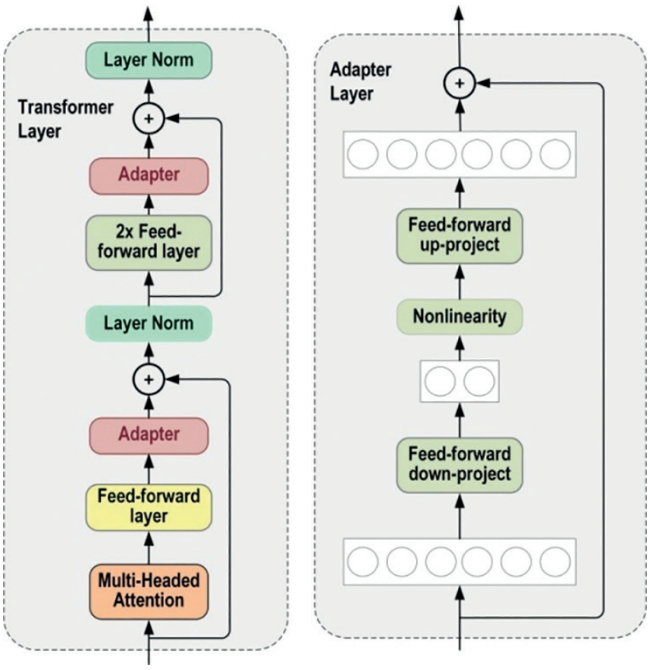

Рис.4. Схема трансформера (слева) с включенным в него адаптером (справа), реализующим метод LoRA

Как показано на рис. 5, LoRa обучает только матрицы A и B, оставляя предварительно обученные веса замороженными [9].

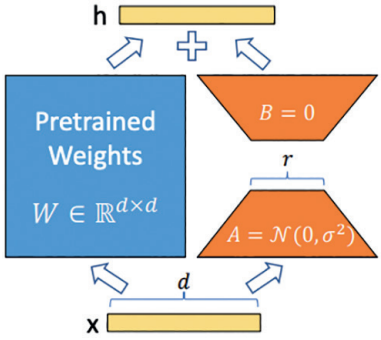

Рис.5. Схема, иллюстрирующая метод LoRA

LoRa позволяет использовать одну и ту же модель для разных задач путем замены весов LoRa, сокращая объем памяти, необходимый для хранения разных моделей. Обучение с помощью LoRa происходит быстрее, поскольку оптимизируются только матрицы LoRa, в отличие от полной точной настройки.

Формула, описывающая в тензорной нотации метод LoRA, имеет следующий вид:

$$W_0 + \Delta W = W_0 + BA,$$

где W_0 – матрица весов предварительно обученной модели; ΔW – обновленные и добавленные весовые коэффициенты во время адаптации исходной модели; r – ранг матрицы с обновляемыми параметрами; $A \in Rr \times k$ – матрица размера rxk, элементами которой являются случайные величины, соответствующие нормальному закону распределения $N(\mu, b^2)$, где μ=0 (среднее значение величины), b (сигма) – среднеквадратическое отклонение; $B \in Rd \times r$ – матрица размера dxr, элементам которой на начальном этапе обучения присваиваются нули.

Важным достоинством LoRa является возможность использовать одну и ту же модель для разных задач путем замены весов в матрицах A и B, сокращая объем памяти, необходимый для хранения разных моделей. QLoRA расширяет метод LoRA посредством квантизации параметров модели, т.е. уменьшения точности весовых коэффициентов, сохраняя при этом необходимую корректность и производительность.

Для специализации LLM в области программирования можно использовать следующие обучающие наборы данных: MBPP, MathQA-Python, MultiPL-MBPP, APPS, DS1000 [17–19]. В результате создается специализированная (обученная и проверенная) СИИ, позволяющая самостоятельно сгенерировать программный код для решения сформулированной на естественном языке задачи или проверить программный код на наличие ошибок и дефектов.

Апробация методики в среде фрэймворка Pytorh

Схема алгоритма, реализующего процедуру адаптации и настройки базовой LLM для ее использования в качестве средства генерации программного кода, представлена на рисунке 6. Для успешного решения данной задачи требуются высокая производительность вычислительных средств и значительные объемы оперативной памяти GPU и CPU. Это обусловлено большими размерами LLM. Например, для запуска Llama 2 7B требуется GPU объемом 13 ГБ, а для ее точной настройки потребуется около 70 ГБ памяти графического процессора. В связи с этим, на первом шаге осуществляется сокращение размера базовой LLM посредством операции квантизации. В результате квантизации на 4 бита для обучения модели потребуется GPU объемом 24 ГБ. При этом потери точности результата работы настроенной модели не превысят 4 процентов. Ниже представлен фрагмент программы, реализующий данную операцию с помощью класса BitsAndBytesConfig фрэймворка Pytorh.

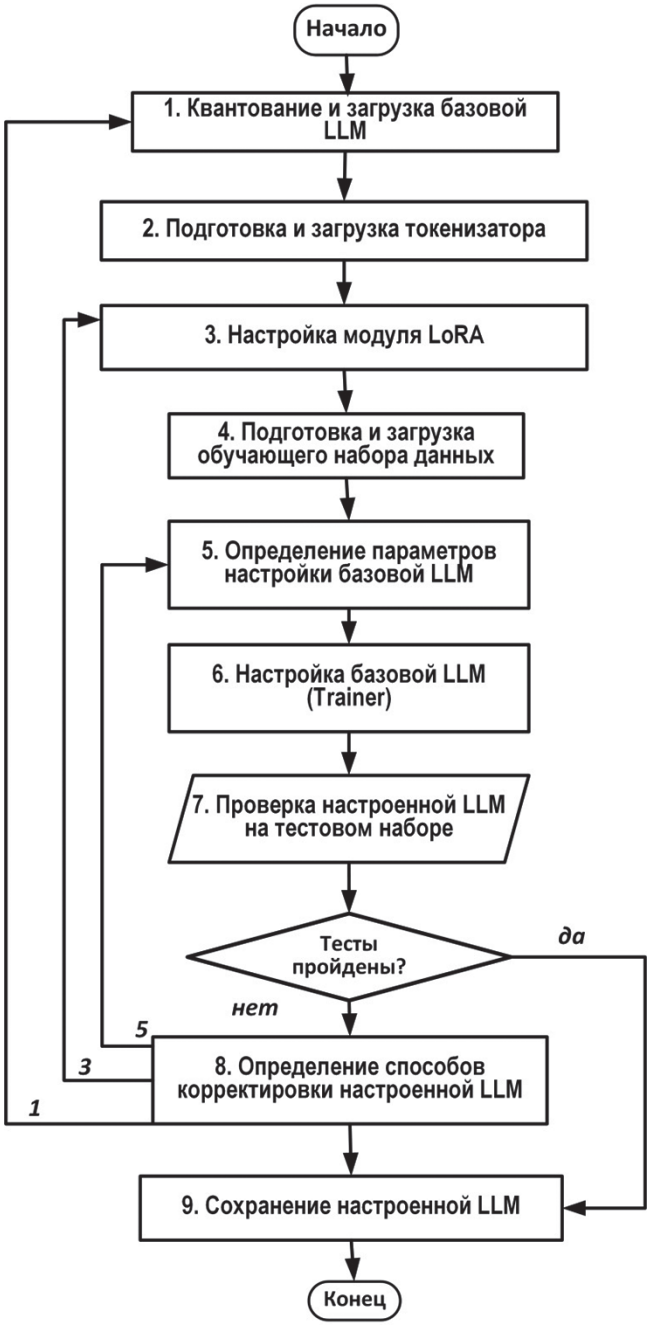

Рис.6. Схема алгоритма, реализующего процедуру точной настройки LLM

```
bnb_config = BitsAndBytesConfig(load_
in_4bit=True, bnb_4bit_use_double_
quant=True, bnb_4bit_quant_type="nf4",
bnb_4bit_compute_dtype=torch.bfloat16)
model = AutoModelForCausalLM.
from_pretrained(base_model, device_
map="auto", trust_remote_code=True,
quantization_config=bnb_config)
```

В результате выполнения данного фрагмента программы линейные слои модели сначала преобразуются в формат fp4/nf4 (`load_in_4bit=True`), а затем выполняется повторная квантизация уже квантизированных весов (`bnb_4bit_use_double_quant=True`).

На втором шаге методики осуществляется загрузка токенизатора настраиваемой LLM:

```
tokenizer = AutoTokenizer.from_
pretrained(base_model)
tokenizer.pad_token = tokenizer.
eos_token
```

На третьем шаге методики осуществляется настройка модуля LoRA. Пример конфигурационного файла представлен ниже:

```
config = LoraConfig(r=8, lora_
alpha=32, target_modules = ["q_proj",
"k_proj", "v_proj", "o_proj"], lora_
dropout = 0.05, bias="none", task_
type="CAUSAL_LM")
```

Основными параметрами являются: r – целое число, определяющее способ обновления матриц, более низкий ранг приводит к менее поддающимся обучению параметрам; `lora_alpha` – коэффициент масштабирования; `lora_dropout` – вероятность выпадения слоев; `task_type` – задает тип используемой модели.

На четвертом шаге осуществляется подготовка и загрузка обучающего набора данных:

```
train_dataset = load_dataset('json',
data_files = 'train_set.jsonl', split =
'train')
```

На пятом шаге формируется конфигурационный файл для программы ***trainer***, осуществляющей непосредственное обучение модели. Пример содержимого конфигурационного файла представлен ниже:

```
training_args = transformers.
TrainingArguments (per_device_train_
batch_size = 1, gradient_accumulation_
steps = 8, num_train_epochs=4,
learning_rate=2e-4, fp16=True, save_
total_limit=3, logging_steps=1, output_
dir="experiments", optim ="paged_
adamw_8bit", lr_scheduler_type=
"cosine", warmup_ratio=0.05)
```

Самыми важными являются следующие параметры:

`gradient_accumulation_steps` – количество шагов обновления для накопления градиентов перед выполнением обратного хода/обновления;

`optim="paged_adamw_32bit"`, – используемый оптимизатор;

`learning_rate` – начальная скорость обучения для AdamW optimizer.

На следующем шестом шаге осуществляется настройка базовой модели с помощью модуля `trainer`. В приведенном ниже тексте программы определены базовая модель, обучающий набор данных, конфигурационный файл, функция, используемая для формирования пакета из списка элементов `train_dataset`.

```
trainer = transformers.Trainer(model=
base_model, train_dataset = data,
args = training_args, data_collator =
transformers.DataCollatorForLanguageMo
deling(tokenizer, mlm=False))
```

На седьмом шаге выполняется тестирование полученной в результате обучения специализированной модели. Для оценки качества используются следующие показатели: полнота и корректность выполнения тестовых заданий, использованные вычислительные ресурсы и затраченное время. На основе анализа результатов тестирования принимается решение о прекращении процесса настройки модели, или о продолжении ее обучения и оптимизации. Во втором случае определяется – какие модули и каким образом следует скорректировать, чтобы получить оптимальный результат. В зависимости от выбранного способа коррекции процедура настройки повторяется, начиная с шагов 1, 3 или 5.

Завершается данный процесс, когда тестовые испытания прошли успешно и характеристики производительности стабилизировались на достаточно высоком уровне. Разработанный комплекс нейросетевых моделей и программного обеспечения сохраняется для дальнейшего использования по назначению с помощью следующих операций:

```
trainer.model.save_pretrained(new_
model)
trainer.tokenizer.save_pretrained
(new_model)
```

В качестве базовой LLM для апробации представленной выше методики и реализующих ее программных средств была выбрана LLM Mixtral 8x7B [20]. Выбор данной LLM обусловлен следующими обстоятельствами. Во-первых, модель имеет небольшой размер. Во-вторых, доступна по лицензии Apache 2.0. В третьих, Mixtral 8x7B по многим тестам превосходит или близка к результатам таких моделей как Llama 2 13B и CodeLlama 7B. Отличительной особенностью Mixtral 8x7B является ее архитектура, представляющая собой сеть с разреженной смесью экспертов (SMoE,Sparse Mixture-of-Experts). Mixtral имеет 46,7 млрд общих параметров, но использует только 12,9 млрд параметров для каждого токена. В таблице 1 представлены результаты тестирования возможностей трех LLM: GPT 3.5, Llama 2 и Mixtral8x7B.

Таблица 1

Результаты сравнения возможностей Mixtral с моделями семейства Llama 2 и базовой моделью GPT 3.5

Метод и средства тестирования	GPT-3.5	LLaMA 2 70B	Mixtral 8x7B
MMLU (MCQ in 57 subject)	70.0%	69.9%	70.6%
MBPP (pass@1)	52.2%	49.8%	60.7%
GSM-8K (5-shot)	57.1%	53.6%	58.4%
MT Bench (for Instruct Model)	8.32	6.86	8.30

Представленные в таблице данные свидетельствую о том, что Mixtral 8x7B соответствует или превосходит Llama 2 70B и GPT 3.5 по большинству показателей.

Заключение

В данной статье представлены методы и средства, предназначенные для разработки средств автоматической генерации программного кода, обеспечивающего решение задачи, сформулированной на естественном языке. Основными проблемными вопросами, с которыми сталкиваются разработчики систем данного класса, являются: катастрофическое забывание, риск переобучения, а также исключительно высокие требования

к производительности используемых при этом вычислительных средств – GPU и CPU. Исследование и анализ существующих и перспективных технологий и средств обучения и применения LLM позволил определить наиболее перспективные пути и методы решения и преодоления имеющих проблем и ограничений. В качестве таковых предложено использовать: методы квантизации, точной настройки (LoRA, QLoRA), оптимизации процесса обучения (AdamW, AdaMix) и др. Интегрирующая эти методы и средства методика создания автоматизированных средств генерации программного кода представлена в виде итерационной процедуры настройки (дообучения) ограниченного количества значимых параметров базовой LLM на специально подготовленных наборах данных. Для апробации методики разработана программная реализация в среде Pytorch. Полученные в ходе ее тестирования и применения результаты свидетельствуют о целесообразности применения данного подхода для разработки автоматизированных средств генерации программного кода.

Литература

1. *A Survey of Large Language Models. Wayne Xin Zhao, Kun Zhou, Junyi Li et al. arXiv:2303.18223v13 [cs.CL] 24 Nov 2023.*
2. *Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning https://arXiv:2303.15647v1 [cs.CL] 28 Mar 2023.*
3. *Ankit Yadav, Mayank Singh. Boldly Going Where No Benchmark Has Gone Before: Exposing Bias and Shortcomings in Code Generation Evaluation. arXiv:2401.03855v2 [cs.CL] 23 Feb 2024.*
4. *Attention Is All You Need. Ashish Vaswani, Noam Shazeer, Niki Parmar. arXiv:1706.03762v7 [cs.CL] 2 Aug 2023*
5. *Jay Alammar. The Illustrated Transformer. http://jalammar.github.io/illustrated-transformer.*
6. *Jinjie Ni, Rui Mao, Zonglin Yang. Finding the Pillars of Strength for Multi-Head Attention. arXiv:2305.14380v2 [cs.LG] 15 Oct 2023.*
7. *David Chiang, Alexander M. Rush, and Boaz Barak. 2021. Named tensor notation. ArXiv,abs/2102.13196.*
8. *Noam Shazeer. GLU Variants Improve Transformer. arXiv:2401.03065v1 [cs.SE] 5 Jan 2024.*
9. *LORA: Low-Rank adaptation of large language models. Edward Hu, Yelong Shen, Phillip Wallis and etl., arXiv:2106.09685v2 [cs.CL] 16 Oct 2021.*
10. *LLaMA-Adapter: Efficient Fine-tuning of Language Models with Zero-init Attention. Renrui Zhang, Jiaming Han, Chris Liu, Peng Gao. arXiv:2303.16199v2 [cs.CV] 14 Jun 2023*
11. *Delta tuning: A comprehensive study of parameter efficient methods for pre-trained language models. Ning Ding, Yujia Qin, Guang Yang, Fu Wei, et al. ArXiv, abs/2203.06904*
12. *QLoRA: Quantization-aware low-rank adaptation of large language models Yuhui Xu Lingxi Xie Xiaotao Gu Xin Chen Heng Chang arXiv:2309.14717v2 [cs.LG] 9 Oct 2023.*
13. *QDyLoRA: Quantized Dynamic Low-Rank Adaptation for Efficient Large Language Model Tuning Hossein Rajabzadeh12, Mojtaba Valipour 1, Tianshu Zhu 2, Marzieh Tahaei arXiv:2402.10462v1 [cs.LG] 16 Feb 2024*
14. *Llama 2: Open Foundation and Fine-Tuned Chat Models. Hugo Touvron, Louis Martin, Kevin Stone and et al. arXiv:2307.09288v2 [cs.CL] 19 Jul 2023.*
15. *Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models M. Weyssow, Xin Zhou, K. Kim et al. arXiv:2308.10462v2 [cs.SE] 18 Jan 2024.*
16. *CodePori: Large Scale Model for Autonomous Software Development by Using Multi-Agents. Zeeshan Rasheed, Muhammad Waseem, Mika Saari, Pekka Abrahamsson et al. arXiv:2402.01411v1 [cs.SE] 2 Feb 2024.*
17. *CRUXEval: A Benchmark for Code Reasoning, Understanding and Execution. Alex Gu, Baptiste Roziere, Hugh Leather et al. arXiv:2401.03065v1 [cs.SE] 5 Jan 2024.*
18. *MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. Federico Cassano, John Gouwar, Daniel Nguyen et al. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 49, NO. 7, JULY 2023.*
19. *OOP: Object-Oriented Programming Evaluation Benchmark for Large Language Model. Shuai Wang, Liang Ding, Li Shen et al. arXiv:2401.06628v2 [cs.CL] 21 Feb 2024.*
20. *Mixtral of Experts. Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux. et al. arXiv:2401.04088v1 [cs.LG] 8 Jan 2024.*