GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings

Bang Fu, Di Feng
Zilliz Inc.

Abstract

The rise of ChatGPT [1] has led to the development of artificial intelligence (AI) applications, particularly those that rely on large language models (LLMs). However, recalling LLM APIs can be expensive, and the response speed may slow down during LLMs' peak times, causing frustration among developers. Potential solutions to this problem include using better LLM models or investing in more computing resources. However, these options may increase product development costs and decrease development speed. GPTCache [2] is an open-source semantic cache that stores LLM responses to address this issue. When integrating an AI application with GPTCache, user queries are first sent to GPTCache for a response before being sent to LLMs like ChatGPT. If GPTCache has the answer to a question, it quickly returns the answer to the user without having to query the LLM. This approach saves costs on API recalls and makes response times much faster. For instance, integrating GPTCache with the GPT service offered by OpenAI can increase response speed 2-10 times when the cache is hit. Moreover, network fluctuations will not affect GPTCache's response time, making it highly stable. This paper presents GPTCache and its architecture, how it functions and performs, and the use cases for which it is most advantageous.

1 Introduction

Since OpenAI released ChatGPT, large language models have impressed many people and have been frequently integrated into our daily work and lives. At the same time, more open-source enthusiasts and tech companies have invested time and effort into developing open-source LLMs, such as Meta's LLama (Touvron et al., 2023a,b), Google's PaLM (Chowdhery et al., 2022), Stanford's Alpaca (Wang et al., 2023; Taori et al., 2023), and Databrick's Dolly (Conover et al., 2023).

There are two ways to use large language models: online services provided by companies like OpenAI, Claude, and Cohere or downloading open-source models and deploying them on your servers. Both methods require payment. Online services charge you based on tokens, while deploying models on your own server requires purchasing specific computing resources. The choice depends on individual needs.

While online services are more expensive, they are more convenient and effective and provide a better user experience than deploying models yourself. Costs and user experience are two critical considerations for building LLM applications. As your LLM application gains popularity and experiences a surge in traffic, the cost of LLM API calls will increase significantly. High response latency will also be frustrating, particularly during peak times for LLMs, directly affecting the user experience.

GPTCache is an open-source semantic cache designed to improve the efficiency and speed of GPT-based applications by storing and retrieving the responses generated by language models. Unlike traditional cache systems such as Redis, GPTCache employs semantic caching, which stores and retrieves data through embeddings. It utilizes embedding algorithms to transform the queries and LLMs' responses into embeddings and conducts similarity searches on these embeddings using a vector store such as Milvus. GPTCache allows users to customize the cache to their specific requirements, offering a range of choices for embedding, similarity assessment, storage location, and eviction policies. Furthermore, GPTCache supports both the OpenAI ChatGPT interface and the Langchain interface, with plans to support more interfaces in the coming months.

Through experiments using the `paraphrase-albert-small-v2` model (Reimers

[1] https://openai.com/chatgpt
[2] https://github.com/zilliztech/GPTCache

and Gurevych, 2019) to embed input in the onnx runtime environment and running it on a local Mac with i7, 4CPU, and 32G memory, the time consumed when hitting the cache is approximately 0.3 seconds. Compared to accessing OpenAI ChatGPT with an average response latency of 3 seconds, the time consumed is only 1/10. Furthermore, no tokens are consumed when hitting the cache. Different embedding models and similarity evaluation algorithms must be selected in real development scenarios based on the tolerance for cache errors. Even so, the entire consumption time is about 3-4 times faster.

2 Related Works

2.1 Accelerating LLM Inference

Accelerating LLM Inference. Large language models (LLMs) typically take seconds to infer answers, prompting researchers to explore ways to reduce inference time and resource consumption. One approach is quantization (Dettmers and Zettlemoyer, 2023), which decreases the number of bits needed to represent each parameter and, therefore reduces the model size. However, this can result in a trade-off between accuracy and memory footprint. Another method is pruning, which can sparsify large-scale generative pre-trained transformer (GPT) models without retraining, as demonstrated by SparseGPT (Frantar and Alistarh, 2023). Additional methods include Compressing (Xu et al., 2020) and Inference with Reference (Yang et al., 2023).

2.2 Widespread application of Caching

Widespread application of Caching. Caching is a commonly used technique to reduce frequent and computationally expensive data accesses, which can improve system query performance. Many different caching schemes have been proposed for various scenarios. For example, semantic knowledge extracted from data can convert cache misses to cache hits, avoiding unnecessary access to web sources (Lee and Chu, 1999). Another example is in querying multiple databases with sensitive information, where a differentially private cache of past responses can answer the current workload at a lower privacy budget while meeting strict accuracy guarantees (Mazmudar et al., 2022). In addition, a cached memory architecture for new changes to embedding tables has been proposed during embedding. In this architecture, most rows in embeddings are trained at low precision, while the most frequent or recently accessed rows are cached and trained at full precision (Yang et al., 2020). As demonstrated, caching is applied in a variety of real-world development processes.

2.3 Embedding Models

Embedding models (Almeida and Xexéo, 2023) are a type of machine learning model that map discrete symbols or objects (such as text, images, audio, etc.) to continuous vector spaces. These vectors are called embedding vectors and are indispensable in many natural language processing (NLP) and computer vision (CV) tasks.

In NLP tasks, embedding models aim to map text into a low-dimensional continuous vector space. This makes it easier for machine learning models to process the text. The vectors can capture semantic information about the text, such as its meaning in context. In CV tasks, embedding models can map images, videos, or objects into a vector space. This approach allows them to be processed by computer vision algorithms, such as image search and identification. Common text embedding models include BERT (Devlin et al., 2019), GloVe (Pennington et al., 2014), and Word2Vec (Goldberg and Levy, 2014; Mikolov et al., 2013). These models generate embedding vectors by processing large amounts of text data. They can also perform well in many NLP tasks, such as semantic similarity calculation, part-of-speech tagging, named entity recognition, and sentiment analysis.

2.4 Vector Store

A vector database is designed for storing and managing vector data. Vector data consists of sequences of numbers commonly used to represent objects or features in high-dimensional spaces. For example, data types such as images, audio, and natural language text can be represented as vector data.

Vector databases improve the efficiency and accuracy of vector data retrieval by using vector similarity measures to index and query the data. This indexing technique allows the database to quickly find vectors most similar to a query vector, making it useful for various applications such as sentiment analysis, image search, speech recognition, and recommendation systems.

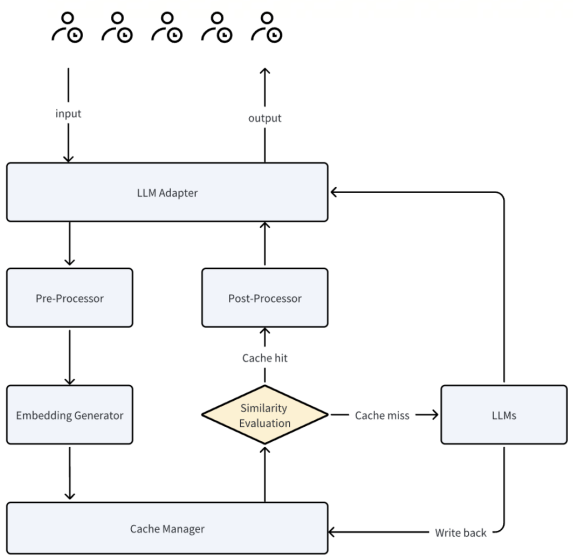

Figure 1: GPTCache: The architecture comprises six core components: adapter, pre-processor, embedding generator, cache manager, similarity evaluator, and post-processor.

3 GPTCache: Semantic Cache for LLMs

The overall workflow of GPTCache follows the general cache pattern - attempting to obtain results from the cache before fetching data or processing requests. If successful, the process terminates immediately. Otherwise, the processing path is the same as if the cache did not exist. However, before returning, the corresponding results are stored in the cache so that repeated actions will retrieve results directly from the cache next time. Using the cache significantly reduces workflow time, which explains why cache designs are ubiquitous in our lives, such as multi-level caches in computers, DNS caches in networks, and Redis/Memcache in management systems.

3.1 Adapter

The adapter serves as the interface for GPTCache to interact with the outside world. It is responsible for converting LLM requests into cache protocols, controlling the entire cache workflow, and transforming cache results into LLM responses. For easy integration of GPTCache into our systems or other ChatGPT-based systems without extra development effort, the adapter should be easy to integrate with all LLMs and flexible enough to integrate more multimodal models in the future.

3.2 Pre-Processor

The pre-processor handles the input of LLM requests primarily by formatting the information as the primary key for the cache data. This includes removing prompt information from inputs, compressing input information, and only retaining the last certain words for long texts or the last round in a multi-round conversation. These operations make the request data more distinguishable from each other and remove redundant and irrelevant information from the requests.

Pre-processing is a critical factor affecting the performance of the cache. For example, suppose both inputs contain a large portion of prompt information, where the key part of the information is only a small portion of the entire input. In that case, the cache cannot obtain the key information without eliminating the prompt. This can result in a high probability that all requests hit the cache. The preprocessed results are passed to the Embedding component for vector conversion.

3.3 Embedding Generator

The embedding generator can convert user queries into embedding vectors for later vector similarity retrieval. There are two methods to achieve this functionality. The first method generates embedding vectors through cloud services (such as OpenAI, Hugging Face, Cohere, etc.). The second method involves generating embedding vectors using local models that can be downloaded from sources such as HuggingFace or GitHub.

3.4 Cache Manager

The cache manager is the core component of GPTCache and has three functions:

- Cache storage: stores user requests and corresponding LLM responses.
- Vector storage: stores embedding vectors and retrieves similar results.
- Eviction management: controls cache capacity and clears expired data according to LRU or FIFO policy when the cache is full.

Before a piece of data is stored, an id will be generated. The id and scalar data will be stored in cache storage, and the id and vector data will be stored in vertor storage. In this way, cache storage and vertor storage are associated. Eviction management also records these IDs. When cache data

needs to be cleared, the data corresponding to cache storage and vertor storage will be deleted based on the id.

The eviction manager releases the cache space by deleting data that has been unused for a long time or is furthest away from using in the GPTCache. If necessary, it removes data from both the cache and vector store. However, frequent deletion operations in the vector store can lead to performance degradation. Therefore, GPTCache only triggers asynchronous operations (e.g., index building, compression, etc.) upon reaching deletion thresholds.

3.5 Similarity Evaluator

GPTCache retrieves the Top-K most similar answers from its cache and uses a similarity evaluation function to determine if the cached answer matches the input query. The similarity evaluation module is also crucial for GPTCache. After research, we eventually adopted the fine-tuned ALBERT model. Of course, there is still room for improvement here, and other language models or LLMs (such as LLaMa-7b) can also be used.

3.6 Post-Processor

The post-processor is responsible for preparing the final response to be returned to the user. It can either return the most similar response or adjust the response's randomness based on the request's temperature parameter. If a similar response is not found in the cache, the LLM will handle the request to generate a response. The generated response will be stored in the cache before being returned to the user.

3.7 Key GPTCache Use Cases

Not all LLM applications are suitable for GPTCache, as the cache hit rate is a crucial factor for the cache's effectiveness. If the cache hit rate is too low, the return on investment cannot balance the input, and there is no need to spend effort on this feature. This is similar to traditional caching scenarios, where caching is usually done only on frequently accessed public nodes to maximize resource utilization and system performance and improve user experience.

This paper introduces three critical practical situations where GPTCache is most beneficial:

1. LLM applications designed for specific domains of expertise, such as law, biology, medicine, finance, and other specialized fields.

2. LLM applications applied to specific use cases, such as internal company ChatBots or personal assistants like chat-pdf and chat-paper. These applications can be enhanced with a cutting-edge AI technology stack called CVP[3] (ChatGPT+Vector DB]+prompt engineering). This combination overcomes the limitations of knowledge bases and enables further expansion and innovation.

3. LLM applications with large user groups can benefit from using the same cache for user groups with the same profile if user profiling and classification can be done. This approach yields good returns.

4 Experiments

To evaluate GPTCache, we randomly scrape some information from the webpage, and then let chatgpt produce a corresponding data (similar or exactly opposite). And then we created a dataset consisting of three types of sentence pairs:

- Similar sample pairs: two sentences with identical semantics

- Opposite sample pairs: two sentences with related but not identical semantics

- Unrelated sample pairs: two sentences with completely different semantics

Then we evaluate the effectiveness of cache through five indicators, which are:

1. Cache Hit, which successfully finds similar values based on the input, which consists of Positive Hits and Negative Hits.

2. Cache Miss, no similar value was found based on the input

3. Positive Hits, the obtained cache value is confirmed to be similar to the input value

4. Negative Hits, the obtained cache value is found to be not similar through inspection.

[3] https://zilliz.com/blog/ChatGPT-VectorDB-Prompt-as-code

Cache Hit	Cache Miss	Positive Hits	Negative Hits	Hit Latency
876	124	837	39	0.20 s

Table 1: Results for Caching Hit and Miss Samples, Caching Mixed Positive and Negative Queries, and Hit Latency

Cache Hit	Cache Miss	Positive Hits	Negative Hits	Hit Latency
570	590	549	21	0.17 s

Table 2: Results for Caching Hit and Miss Samples, Caching Mixed Positive and Negative Queries, and Hit Latency

5. Hit Latency, it includes pre-processing time, cache data search time, similarity calculation time and post-processing time. The pre-processing and post-processing do not use the model during the test process, and are just simple character or number comparisons.

In addition, we tried different similarity algorithms and found that they had no impact on the results, so we used the common cosine similarity.

First, we cached the keys of all 30,000 positive sample pairs. Next, we randomly selected 1,000 samples and used their peer values as queries. Table 1 presents the results.

We found setting the similarity threshold of GPT-Cache to 0.7 achieves a good balance between hit and positive ratios. So we used this for subsequent tests.

To determine if a cached result is positive or negative to the query, we used the similarity score from ChatGPT with a positive threshold of 0.7. We generated this by prompting:

> Please rate the similarity of the following two questions on a scale from 0 to 1, where 0 means not related and 1 means exactly the same meaning. And questions, "Which app lets you watch live football for free?" and "How can I watch a football live match on my phone?" The similarity score is.

We issued 1,160 queries with 50% positive and 50% unrelated negative samples. Table 2 presents the results. The hit ratio was about 50%, and the negative hit ratio was similar to Experiment 1, indicating GPTCache successfully distinguished related and unrelated queries.

Next, we tried to also cache all negative samples and queried with their peers. Surprisingly, despite high ChatGPT similarity scores (over 0.9) for some pairs, none hit the cache. The cause of the cache error could be the similarity evaluator's fine-tuning on this dataset correctly undervalued the similarity of negative pairs.

The initial experiments demonstrate that GPT-Cache can effectively utilize semantic similarity to cache LLM query-response pairs and achieve significant speedups. We plan to conduct more rigorous evaluations on larger and more diverse datasets. When tuning the similarity threshold, further investigation is required to balance cache hits versus false positives.

5 Future Challenges

One core factor affecting GPTCache's caching effectiveness is the choice of embedding model. Compared to other component selections, the choice of embedding model is crucial because subsequent vector database retrieval relies on the embedding vectors. If the vectors cannot adequately capture the features of the input text, the retrieval results will be very noisy or even counterproductive, returning completely irrelevant cached data. Our testing has shown that even the best cache hit rates do not exceed 90% with current embedding models. This means that negative cache hits are noticeable during use. While this may not greatly impact individual users, it would be unacceptable in real production scenarios. Although other methods, like more strict similarity evaluation, could improve positive cache hit rates, this would also decrease the overall hit rate. Most current embedding models are likely optimized for search scenarios but may not work as well for cache matching. For example, results with semantics opposite to the input text are acceptable in search since they have structural similarity, but this is unacceptable in caching scenarios. Naturally, how to obtain embeddings suitable for caching is an open area for exploration.

Even with a suitable embedding model, positive hit rates are unlikely to reach production requirements, such as 99%, without decreasing cache hits. The similarity evaluation module plays a core role in improving positive cache hit rates by filtering incorrect hits. Our current implementations include vector distance, retrieving distance, cohere rerank

API, and sbert cross-encoder. However, testing shows these methods do not sufficiently distinguish between positive and negative cache hits. To address this, we are using large models to judge sentence similarity and distill them into a small model to obtain a specialized model for textual similarity.

As large language models are widely adopted, their supported token counts have increased from 2k initially to 100k. However, if a single input exceeds the LLM's token count limit, it cannot process the request. Similarly, conversations with total tokens exceeding the limit must drop some information. Large token counts from long texts or conversations pose a challenge for caching, making it difficult to identify key information and generate representative vectors. Currently, we utilize summary models to pre-process and shorten long inputs, but this approach increases cache instability, and its effectiveness is not optimistic. Therefore, special cache lookup methods may be needed for long texts.

As mentioned earlier, is there any alternative to retrieving cache data using vector databases? For example, can we use traditional databases like MySQL, PostgreSQL, SQL Server, or Oracle to store cache data, with textual pre-rocessing to standardize user inputs? For instance, when the inputs are "tell me a joke" and "I want to get a joke", can we convert them to a certain string, like "tell a joke" , or a same number? Cache hits could then utilize string matching or numeric ranges instead of vectors.

6 Conclusion

GPTCache is a caching solution tailored for LLM applications. It brings the following benefits to the LLM app developers:

- Less costs: Most LLM services charge fees based on a combination of the number of requests and token count. GPTCache can effectively minimize expenses by caching query results, thereby reducing the number of requests and tokens sent to the LLM service.

- Faster response times: LLMs utilize generative AI to produce responses in real-time, which can be time-consuming. However, when a similar query is cached, the response time greatly improves, as the result is retrieved directly from the cache without interaction with the LLM service. In most cases, GPTCache can also offer better query throughput than standard LLM services.

- More scalable and available: LLM services often impose rate limits on the number of access requests within a given timeframe. If these limits are exceeded, additional requests are blocked until a cooldown period has elapsed, leading to service outages. GPTCache allows you to easily scale and handle increasing query volumes, ensuring consistent performance as your application's user base expands.

By utilizing semantic similarity search and vector embeddings, GPTCache provides an effective caching solution that enhances performance, reduces costs, and improves scalability for applications that use large language models. Our initial experiments have shown great potential, and we plan to conduct more comprehensive evaluations on diverse real-world datasets and application scenarios.

References

Felipe Almeida and Geraldo Xexéo. 2023. Word embeddings: A survey.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

Mike Conover, Matt Hayes, Ankit Mathur, Jianwei Xie, Jun Wan, Sam Shah, Ali Ghodsi, Patrick Wendell, Matei Zaharia, and Reynold Xin. 2023. Free dolly: Introducing the world's first truly open instruction-tuned llm.

Tim Dettmers and Luke Zettlemoyer. 2023. The case for 4-bit precision: k-bit inference scaling laws. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 7750–7774. PMLR.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive language models can be accurately pruned in one-shot. *arXiv preprint arXiv:2301.00774*.

Yoav Goldberg and Omer Levy. 2014. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *CoRR*, abs/1402.3722.

Dongwon Lee and Wesley W. Chu. 1999. Semantic caching via query matching for web sources. In *Proceedings of the Eighth International Conference on Information and Knowledge Management*, CIKM '99, page 77–85, New York, NY, USA. Association for Computing Machinery.

Miti Mazmudar, Thomas Humphries, Jiaxiang Liu, Matthew Rafuse, and Xi He. 2022. Cache me if you can: Accuracy-aware inference engine for differentially private data exploration.

Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*.

Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-instruct: Aligning language models with self-generated instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada. Association for Computational Linguistics.

Canwen Xu, Wangchunshu Zhou, Tao Ge, Furu Wei, and Ming Zhou. 2020. BERT-of-theseus: Compressing BERT by progressive module replacing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7859–7869, Online. Association for Computational Linguistics.

Jie Amy Yang, Jianyu Huang, Jongsoo Park, Ping Tak Peter Tang, and Andrew Tulloch. 2020. Mixed-precision embedding using a cache.

Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. 2023. Inference with reference: Lossless acceleration of large language models.