Enhancing Parameter-Efficient Fine-Tuning of Large Language Models with Alignment Adapters and LoRA

Abdul Baseer Mohammed

Committee

Dr. Hongyang Sun (Chair)
Dr. Prasad Kulkarni
Dr. David O. Johnson

Submitted to the graduate degree program in Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Signature & Date

Signature & Date

Signature & Date

Enhancing Parameter-Efficient Fine-Tuning of Large Language Models with Alignment Adapters and LoRA

Abstract

Large Language Models (LLMs) have become integral to natural language processing, involving initial broad pretraining on generic data followed by fine-tuning for specific tasks or domains. While advancements in Parameter Efficient Fine-Tuning (PEFT) techniques have made strides in reducing resource demands for LLM fine-tuning, they possess individual constraints. This project addresses the challenges posed by PEFT in the context of transformers architecture for sequence-to-sequence tasks, by integrating two pivotal techniques: Low-Rank Adaptation (LoRA) for computational efficiency and adaptive layers for task-specific customization. To overcome the limitations of LoRA, we introduce a simple yet effective hyper alignment adapter, that leverages a hypernetwork to generate decoder inputs based on encoder outputs, thereby serving as a crucial bridge to improve alignment between the encoder and the decoder. This fusion strikes a balance between the fine-tuning complexity and task performance, mitigating the individual drawbacks while improving the encoder-decoder alignment. As a result, we achieve more precise and contextually relevant sequence generation. The proposed solution improves the overall efficiency and effectiveness of LLMs in sequence-to-sequence tasks, leading to better alignment and more accurate output generation.

Introduction

Large Language Models (LLMs) are deep neural network based AI systems, built on the Transformer architecture. The transformer relies on a self-attention mechanism that enables the model to understand and generate human-like text. LLMs, such as T5, GPT, Llama and Mistral, are pre-trained on vast amounts of text data, allowing them to grasp language, context, and world knowledge.

LLM training phases:

1. **Pre-training**: LLMs begin their journey with a pre-training phase. During pre-training, these models are trained on an extensive corpus of text from the internet, which could include books, articles, websites, and more. This phase helps the model learn grammar, semantics, world knowledge, and even some reasoning abilities. The outcome is a highly capable language model that can generate coherent and contextually relevant text

2. **Downstream Tasks**: After pre-training, LLMs can be fine-tuned for specific "downstream tasks." Downstream tasks refer to a wide array of NLP tasks that benefit from the pre-trained language model's understanding of language. These tasks can include text classification, sentiment analysis, language translation, summarization, and more.

3. **Fine-Tuning**: Fine-tuning is crucial for making LLMs useful in real-world applications. For fine-tuning, you typically need a smaller, task-specific dataset that contains labeled or structured examples related to your target task. During fine-tuning, the model's weights are updated to optimize its performance on the specific task. This process leverages the knowledge gained during pre-training and implements it to the task at hand.

4. **Transfer Learning**: Once fine-tuned for a specific task, these models can be easily adapted for related tasks. For example, if you've fine-tuned a model for sentiment analysis on social media data, you can often use the same model with minimal additional training for other text classification tasks, such as spam detection or hate speech identification.

Advent of Transformers architecture

Transformers, a type of neural network architecture, excel in sequence-to-sequence (seq2seq) language tasks. The transformer architecture, widely employed in natural language processing, implements an encoder-decoder structure. The encoder processes input sequences through self-attention and feedforward layers, allowing it to capture intricate dependencies and relationships within the data. Simultaneously, the decoder generates output sequences using masked self-attention, encoder-decoder attention, and additional feedforward layers.

At the heart of the transformer's success lies the attention mechanism. In the attention mechanism, each word in the input sequence is represented as a vector, and attention scores are computed by comparing the word to others, enabling the model to provide a weightage score based on their significance. The resulting attention weights create context vectors for each word, forming a nuanced understanding of the input. The encoder-decoder attention mechanism further refines this process by allowing the decoder to focus on essential information in the input sequence. Figure 1 represents the general architecture of the transformer architecture.

One of the pioneering models that implemented the transformer architecture on a large scale is the Text-To-Text Transfer Transformer (T5). Developed as one of the early models in the transformer family, T5 revolutionized natural language processing by adopting a universal sequence-to-sequence approach. In our project, we will be using a fine-tuned version of the T5 model.

FLAN-T5

FLAN-T5 (Fine-tuned Language Net T5) model is the instruction fine-tuned version of T5 on a variety of language tasks, including translation, summarization, question answering, and code generation, and has been proven to outperform T5 models in most of the tasks.

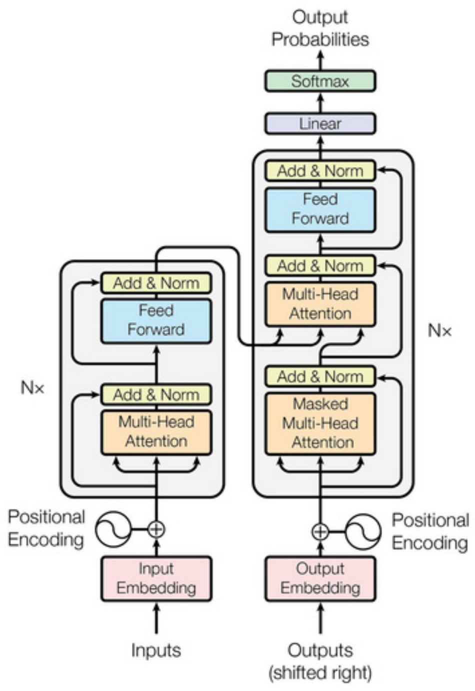


Fig. 1 General architecture of transformers

It has 5 variants:

Small (60M params), Base (250M params), Large (780M params), XL (3B params), XXL (11B params)

These variants have different numbers of encoder-decoder blocks and embedding sizes.

In our project, we are focusing on fine-tuning code to documentation tasks using the FLAN-T5 base model, which consists of around 250M parameters with 12 encoder-decoder blocks and embedding size of 768.

LoRA, Hypernetworks and Adaptive layers

LoRA (Low-Ranked Adaptation) for PEFT (Parameter Efficient Fine-tuning)

Low-Rank Adaptation of Large Language Models is a training method that accelerates the training of LLMs while consuming less memory. It does this by freezing the pre-trained weights of the LLM and introducing trainable rank-decomposition matrices into each layer of the model. These matrices are much smaller than the original model parameters, which makes them easier to train and deploy. LoRA has been shown to achieve comparable or even better performance than traditional fine-tuning methods on a variety of downstream tasks, such as text classification, summarization, and question answering. It is also more efficient in terms of training time and memory
requirements.

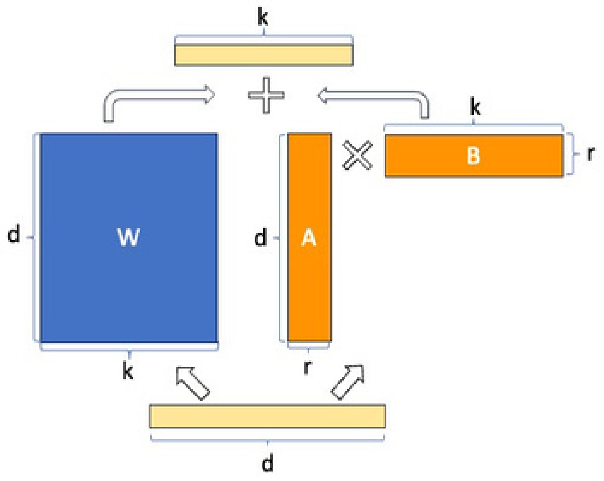


Fig. 2 Implementation and working of LoRA in the attention block

Fig. 2 shows how LoRA performs the forward pass by performing the following h = (W + BA).x
where W - weights from the pre-trained checkpoint (not trainable for the fine-tuning task)
A, B - LoRA matrices that are trained during the fine-tuning
x - Input to the attention block

Working of LoRA:

1. Start with a pre-trained LLM, such as FLAN-T5 base variant.
2. Freeze the pre-trained weights W of the LLM.
3. Introduce trainable rank-decomposition matrices into each layer of the LLM. (the rank of the decomposed matrix is denoted by 'r', indicating the rank of the decomposed matrices.
4. Fine-tune the rank-decomposition matrices on a downstream task.

Hypernetworks

Hypernetworks are neural network architectures designed to facilitate the fine-tuning of other neural networks, particularly for transfer learning tasks. They are essentially meta-networks that generate weights and parameters for another network.
In the context of fine-tuning, hypernetworks provide an elegant approach for efficiently repurposing pre-trained models for specific tasks.
They enhance the adaptability and versatility of deep learning models, making them well-suited for a wide range of applications, particularly when you want to leverage the power of pre-trained models while minimizing the effort and data required for task-specific fine-tuning.

Working of hypernetworks:

1. **Initialization:** Hypernetworks are initialized with random weights.
2. **Target Network Representation:** They take the representation of a target neural network as input.
3. **Parameter Generation:** Hypernetworks generate task-specific parameters, such as weights and biases.
4. **Fine-Tuning:** These parameters are used to fine-tune the target network for specific tasks.
5. **Optimized Adaptation:** Hypernetworks optimize the target network's parameters for efficient learning and adaptation.
6. **Improved Task Performance:** The fine-tuned network performs better on the desired task, leveraging the hypernetwork-generated parameters.

Adaptive layers

Adaptive layers in transformers are a type of neural network layer that can dynamically adjust its parameters during training or inference. This allows the layer to better adapt to the specific data it is being used to process.

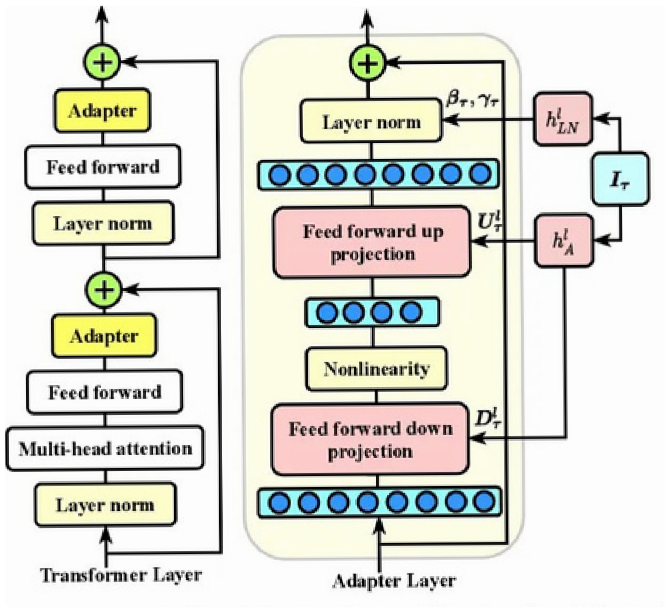


Fig. 3 Implementation of adaptive layers for fine-tuning

Fig. 3 shows the implementation of the adaptive layers that are injected in the encoder-decoder blocks

One way to implement an adaptive layer is to use a hypernetwork. In the case of adaptive layers, the hypernetwork would be used to generate the weights of the adaptive layer itself.

Motivation

Understanding the complexities in the sequence-to-sequence tasks require models that can dynamically adjust their complexity to handle diverse patterns and sequence structures**.** Most of the transformers are trained on seq2seq tasks where the sequences are similar to each other, such as language translation, summarization etc. and have been able to achieve state-of-the-art results in the task. However, when they are tested with seq2seq tasks involving sequences which are completely different from each other (eg: code to documentation), they fail to understand the context and patterns that are important. Therefore, this project delves into the integration of adaptive layers and hypernetworks into transformer models, empowering them to tailor their complexity for different input types. Additionally, the exploration of parameter efficient fine-tuning techniques with LoRA addresses the pressing need for resource optimization. By optimizing model parameters without compromising performance, and harnessing hypernetworks for dynamic layer generation, we aim to create more compact, yet highly adaptable models, specifically designed to enhance the accuracy and efficiency of code-to-documentation translation tasks.

Limitations of LoRA on seq2seq tasks

One of the main limitations of LoRA is that it only trains the attention blocks to be low-rank, and this adaptation may not always align perfectly with the characteristics of the training dataset. This means that the model's attention mechanisms might not fully capture the complex dependencies within the data. Attention mechanisms in neural networks are designed to learn intricate relationships between tokens, and reducing their rank may lead to a loss of some fine-grained information. In practice, LoRA might introduce a trade-off between computational efficiency and model performance, and the choice of the rank reduction hyperparameters could impact the trade-off differently for various tasks and datasets.

Additionally, the effectiveness of LoRA may be limited by the specific task and dataset. In some cases, reducing the rank of attention matrices might lead to a noticeable loss in performance (in our case), as certain tasks may rely heavily on capturing long-range dependencies or intricate relationships between input elements.

The potential reasons for this limitation are:
1. LoRA is typically applied to attention blocks only, which may not be sufficient for sequence-to-sequence tasks where both sequences are completely different from each other.
2. LoRA may not be able to capture the complex relationships between the two sequences, which may result in lower performance compared to fully fine-tuned models.

3. LoRA may not be able to handle tasks that require a lot of parameter tuning, as it is designed to reduce the number of trainable parameters.
In the context of mitigating the limitations introduced by LoRA and enhancing the alignment of encoder outputs for improved embeddings received by the decoder, adaptive layers can play a crucial role.

Potential advantages of adaptive layers

Adaptive layers are a set of learnable hypernetworks inserted into the architecture of a pre-trained model to adapt it to a specific task. Some of the key advantages of adaptive layers are:

1. **Enhanced Model Adaptation**: Adaptive layers allow the model to fine-tune its parameters specifically for the code-to-documentation task. While LoRA primarily focuses on reducing the computational complexity of attention mechanisms, adaptive layers provide a mechanism to adapt the model to the specifics of the task. By fine-tuning these layers, you can capture the intricacies of the code-to documentation dataset and enhance the model's performance.

2. **Improved Alignment of Encoder Outputs**: The combination of LoRA and adaptive layers can work synergistically. While LoRA helps in reducing the computational demands of the model, adaptive layers can ensure that the encoder's outputs are better aligned with the task
requirements. Adaptive layers can learn to emphasize or de-emphasize certain aspects of the encoder's representations, improving the quality of the embeddings passed to the decoder. This ensures that the decoder receives more relevant and task-specific information, which is vital for generating accurate documentation from code.

Working

One of the possible ways to combine LoRA and adaptive layers in the transformers architecture for fine-tuning tasks can be:

1. Start with a pre-trained LLM, in our case FLAN-T5 base variant.
2. Freeze the pre-trained weights of the LLM.
3. Introduce trainable rank-decomposition matrices using LoRA into some of the attention blocks of the LLM.
4. Create an adaptive hypernetwork block that are injected on top of the encoder layers. In this case, we add the adaptive layer on top of the encoder network. This results in enhanced alignment between the encoder outputs and the decoder inputs.
5. Fine-tune the adaptive layers and LoRA rank-decomposition matrices on a downstream task.

Overall, combining LoRA and adaptive layers in transformers is a promising approach for improving the performance, efficiency, and robustness of LLMs. More research is needed to determine the best way to combine these techniques, but the potential benefits are significant.

Our approach

Our approach integrates PEFT techniques such as adaptive layers and LoRA to improve the performance of fine-tuning tasks. The adaptive layers are modified to perform better alignment in such a way that it can make use of the LoRA training as well. Adaptive layers are widely used for fine-tuning tasks where resources are constrained, as they are more efficient to train and use than LoRA. However, both adaptive layers and LoRA have some limitations. One limitation of adaptive layers is that they can be less robust to overfitting than LoRA. This is because adaptive layers are trained to learn task-specific representations, while LoRA is trained to learn a global adaptation matrix for the entire model.

Another limitation of LoRA is that it can be more computationally expensive to train and use than adaptive layers. This is because LoRA requires training a separate low-rank adaptation matrix for each layer of the model.

Our approach addresses these limitations by combining adaptive layers and LoRA in a novel way. We use adapters to learn the representations used by the decoders.

Improved alignment vectors

Adapters can be used to learn task-specific alignment vectors, which can lead to improved performance on seq2seq tasks. This is because adapters can learn to align the encoder and decoder representations in a way that is specifically tailored to the task at hand.

Consider a task of translating code from one programming language to another. The encoder would take the code in the source programming language as input and produce a representation of the code. The decoder would then take this representation as input and produce a translation of the code in the target programming language.

One challenge in this task is that the encoder and decoder representations may not be aligned. For example, the encoder representation may focus on the low-level details of the code, while the decoder representation may focus on the high-level semantics of the code.

Adapters can be used to learn alignment vectors that can bridge the gap between the encoder and decoder representations. These alignment vectors can be used to transform the encoder representation into an aligned version that can be better understood by the decoder.

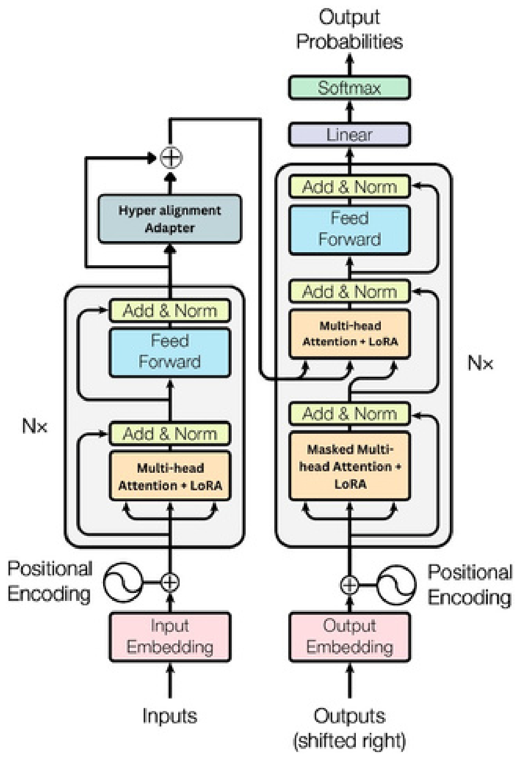


Fig. 4 Our architecture that combines LoRA and adaptive layers

By integrating LoRA to improve computational efficiency and incorporating adaptive layers for task-specific adaptation, we can achieve a balanced trade-off between complexity and performance. This combined approach has the potential to address the limitations inherent to LoRA while ensuring that the encoder's outputs are suitably aligned with the decoder's requirements, ultimately leading to a more effective model for sequence-to-sequence generation tasks. In this context, we implement a *hyper alignment adapter* which serves as a critical bridge between the encoder and decoder. Its primary function is to generate decoder inputs based on the encoder's outputs, acting as a mechanism to align and enhance the encoder's representations for improved understanding by the decoder. By doing so, it ensures that the decoder receives better-aligned information from the encoder, enhancing the model's capabilities for sequence-to-sequence tasks.

Figure 5 shows a simple architecture of a hyper alignment adapter that uses a feedforward layer followed by RMS Normalization and GELU activation. However, to further increase the complexity of the model and improve the performance of the model, the complexity of the alignment adapter can be increased.

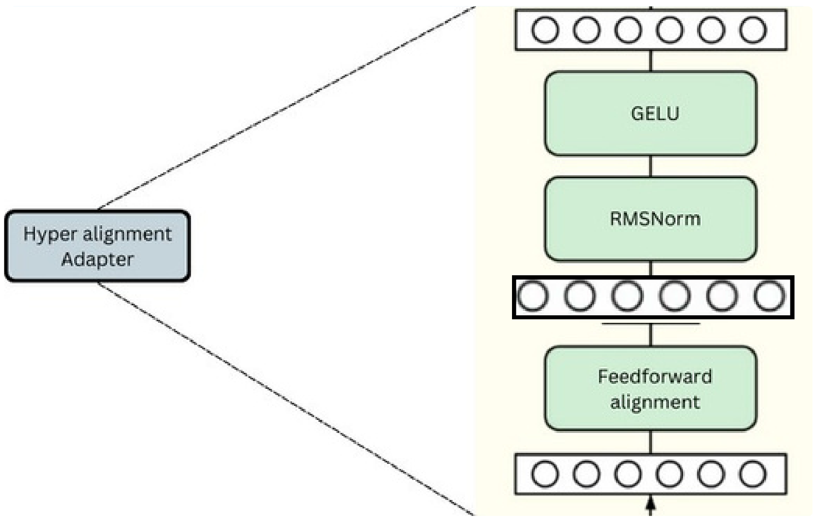


Fig. 5 Architecture of hyper alignment adapter

$$Alignment\ adapter = GELU(\ RMSNorm(\ W_{\Theta} \odot\ X))$$

WΘ - Weight matrix from the feedforward alignment

X - Output embeddings received from the final encoder block

The above formula shows the forward propagation that occurs in the alignment adapter after the encoder outputs. This lets the model learn better alignment representations without significantly necessarily increasing the complexity.

The hyper alignment adapter is implemented using linear feedforward layers that take the input from the last layer of the encoder. The linear layers are then connected to a RMS Normalization layer, followed by a traditional GELU activation.

The feedforward alignment network is a simpler version of the attention blocks that uses a single matrix to provide a simpler combination of the dense and attention mechanism that can help in making the output from the encoder align more with the decoder input. This lets the adapter to perform extra computations that can help in providing better context to the decoder for processing.

The choice of GELU (Gaussian Error Linear Unit) are due to the following reasons:

1. **Smoother gradients**: GELU has smoother gradients than ReLU, which can lead to faster convergence and better performance during training.
2. **Less vanishing gradients**: GELU is less likely to cause vanishing gradients than ReLU, especially in deep networks.
3. **Better regularization**: GELU can act as a regularizer, helping to prevent overfitting.

Hence, it provides a smoother and better gradient flow over other activation functions such as ReLU and LeakyReLU.
The choice of RMS Normalization (RMSNorm) over the Layer Normalization (LayerNorm) is mainly due to the **re-scaling invariance**, which is provided in the RMSNorm.

RMSNorm is computationally simpler than LayerNorm because it does not require centering the input data. RMSNorm is more stable than LayerNorm in some cases. For example, if the input data is scaled by a constant factor, the output of RMSNorm will remain the same. This is not the case with LayerNorm, which can lead to instability during training. They have also been shown to be more robust to noise and outliers in the input data.

Furthermore, we apply LoRA on the Query and Value blocks present in all the attention blocks, with a dropout rate of 0.05. The choice of this is due to the following reasons:

1. **Improved performance**: LoRA has been shown to improve the performance of transformer models on a variety of tasks, including machine translation, question answering, and summarization. This is likely because LoRA allows the model to learn more complex relationships between the input and output sequences.
2. **Reduced memory and computation**: LoRA reduces the memory and computation required to train and deploy transformer models. This is because LoRA only updates a low-rank adaptation layer, rather than the entire transformer model.
3. **Increased interpretability**: LoRA can be used to increase the interpretability of transformer models. This is because the low-rank adaptation layer can be used to identify the most important features for the task.

However, implementing LoRA on all the attention blocks can lead to overfitting. This is because LoRA is a very powerful tool that can be used to learn complex relationships between the input and output sequences. If LoRA is performed on all the attention blocks, then the model may be able to learn even the most subtle patterns in the training data, which can lead to overfitting.

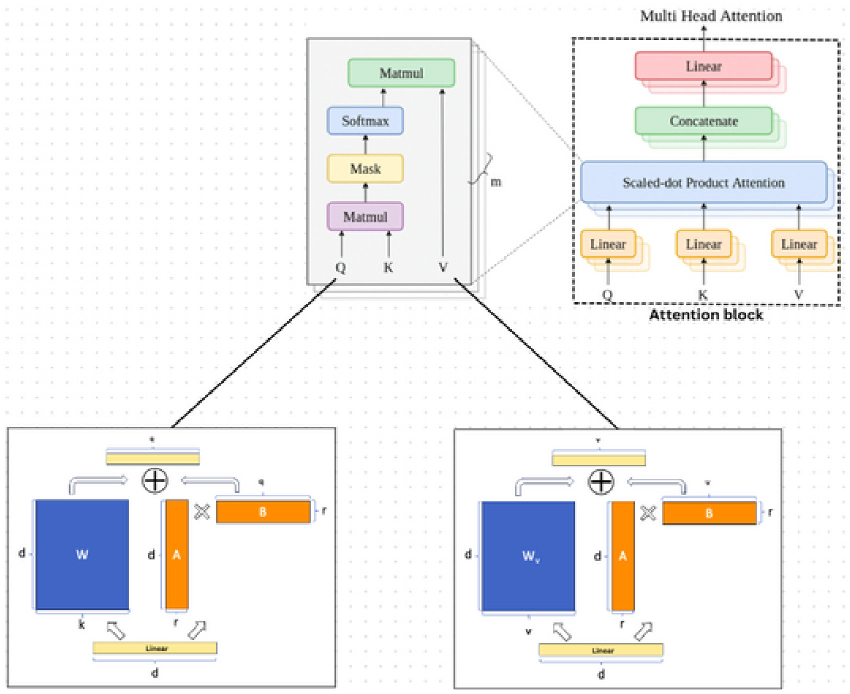


Fig. 6 Implementation of LoRA on Query and Value matrices of the attention mechanism

Figure 6 represents the working of LoRA mechanism on the Query and Value matrices present in each of the attention mechanisms of every encoder and decoder block. Note that the LoRA approach can also be used on the Key and the Feed Forward Network as well. This would increase the number of trainable parameters and can also make the model prone to overfitting, especially if the model is trained on smaller datasets. To keep the training resource-efficient, we apply LoRA only on the query and key matrices.

Hence, these adapter blocks can be added to existing transformer architectures to improve fine-tuning performance. The forward and backward passes of adapter blocks work the same way as for other transformer layers. This provides various advantages:

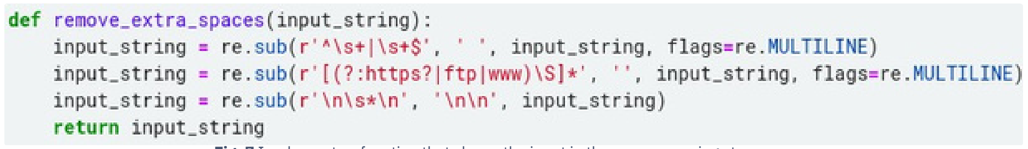
Improved performance: Combining LoRA and adaptive layers can lead to improved performance on a variety of tasks, including machine translation, question answering, and summarization. This is because LoRA allows for more efficient fine-tuning of large language models, while adaptive layers allow the model to learn task-specific representations.

Reduced memory usage: Combining LoRA and adaptive layers can also reduce the memory usage of large language models. This is because LoRA only requires a small number of additional parameters, while adaptive layers can be used to share parameters across different tasks.

Increased flexibility: Combining LoRA and adaptive layers can also make large language models more flexible. This is because LoRA allows for the fine-tuning of specific model components, while adaptive layers allow the model to learn task-specific representations.

Dataset and performance metrics

In our work, we are mainly focusing on working with code snippets that are written in Python programming language. Hence, we tailor our model for the task of document generation for python code snippets.
We train and inference the model using the dataset: code_x_glue_ct_code_to_text (https://zenodo.org/record/7857872/files/python.zip)
The dataset comprises of various code snippets across different programming languages, which includes Go, Java, JavaScript, Ruby, Python, php. Each of the languages had around 150k samples, each of them split into train, test and validation samples. The dataset comprises various details of the code and its corresponding details that includes scraped repo name, url of the code, sha ID, function name and the documentation/ explanation for each of the corresponding code in the form of a JSONL file. For our experimentation, due to the resource limitations, we decided to use the Python code snippets for training, specifically the testing dataset which comprise of around 23k samples of code snippets and its corresponding details.
For preprocessing, we loaded the JSON flies into a Pandas Dataframe for ease of use while cleaning and applying transformations. The unwanted details such as the repo name, sha ID, url from the dataframe, essentially left with the function Code, function name and the corresponding Documentation of it. However, the scraped function code consisted of the documentation in it's code. Hence, we applied transformation on that column of the dataframe using regular expressions to remove the comments/documentation from the code snippets.



```python
def remove_extra_spaces(input_string):
    input_string = re.sub(r'^\s+|\s+$', ' ', input_string, flags=re.MULTILINE)
    input_string = re.sub(r'[(?:https?|ftp|www)\S]*', '', input_string, flags=re.MULTILINE)
    input_string = re.sub(r'\n\s*\n', '\n\n', input_string)
    return input_string
```

Fig. 7 Implements a function that cleans the input in the pre-processing step

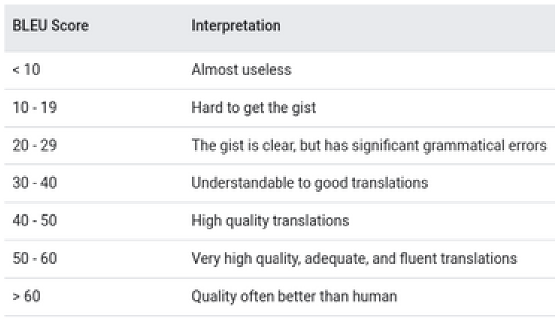
This resulted in a Dataframe containing the pure code snippet, function name and its documentation in different columns. After cleaning the dataset, we combined the function name and Code snippet column into a single column that follows the following template:

This is the function name:
<function_name>
Code:
<code snippet>
Provide the documentation for the following code:

The choice of this template was because of the state-of-the-art results generated by FLAN-T5 with instruct tuning against its T5 predecessor. Therefore, the dataframe was finally cleaned and transformed into 2 columns, the Code-Documentation pairs. This Dataframe was loaded into the dataset pipeline provided by Huggingface for performing the tokenization and padding of the sequences. This creates a simple pipeline for training the LLM in batches.
Regarding the performance evaluation, metrics such as BLEU and ROUGE are used to benchmark the models. Higher scores indicate that the model is better aligned to the expected output data.

BLEU

The BLEU (Bilingual Evaluation Understudy) metric is a commonly used measure for evaluating the quality of machine-generated text, often applied to machine translation tasks. It assesses the similarity between the generated text and reference human-generated text by computing a score based on n-gram[1] overlap, providing a measure of translation accuracy. Higher BLEU scores indicate better translation quality.



BLEU Score	Interpretation
< 10	Almost useless
10 - 19	Hard to get the gist
20 - 29	The gist is clear, but has significant grammatical errors
30 - 40	Understandable to good translations
40 - 50	High quality translations
50 - 60	Very high quality, adequate, and fluent translations
> 60	Quality often better than human

Table 1 BLEU Score Classification Table

ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a metric for evaluating the quality of machine-generated text, primarily used in the context of summarization and text generation tasks. ROUGE measures the overlap of n-grams[1] and other textual units between the generated text and reference summaries, providing a way to assess content overlap and the informativeness of generated text. Higher ROUGE scores indicate better content overlap and summarization quality.
In our experiments, we mainly use ROUGE-L because it calculates the Longest Common Subsequence (LCS) between the predicted and reference summaries.

Experiments

The experiments were all performed on the same training and test dataset to make sure fair benchmarking and training procedure for all the models.

The FLAN-T5 model was quantized to fp16-bit precision.
All the models were trained on the data which contained around 22k code-documentation pairs and the inference was performed using unseen 2k code-documentation pairs.

The training was performed with dual T4 GPUs, which individually provided around 14GB vRAM. The training and inference were performed on the free GPU instances provided by Kaggle. We trained each of our models for almost 12 hours, which is the limit for running a kernel on Kaggle platform. Hence, we trained all the models for a total of 12 hours, by varying the number of epochs (steps and batch size) for each of the models based on the trainable parameters. The full fine-tuning consumed a lot of resources and hence a batch size of 1 per GPU was used. The total number of training steps that it was trained for is around 28,000. Similarly, LoRA hyperformer (ours) consisted of less trainable parameters and hence was able to use a large batch size (4 per GPU). The training steps took less time to complete which let us train the model for higher training steps of 32,000. This indicates that the LoRA and hyperAdapter models are computationally less expensive and easier to train in resource constrained environments.

---

[1] **n-gram** is a contiguous sequence of n items (words, characters, etc.) within a given text. In the context of natural language processing, n-grams are used to model the likelihood of a word or sequence of words occurring in a sequence.
In the context of BLEU score, it is used to assess the overlap between the generated text and reference (human-generated) text by considering n-grams as matching units.
In the context of ROUGE, n-grams are essential for assessing the quality of summaries or generated text. ROUGE measures recall by considering the overlap of n-grams between the generated summary and the reference summary.
A higher BLEU or ROUGE score indicates better similarity between the generated and reference sequences in terms of n-gram matches, providing a quantitative measure of the quality of machine-generated text.

For this project, we mainly focus on the following 4 models:

1. FLAN-T5 (without fine-tuning) No training
2. Fine-tuned FLAN-T5 (with training)
3. PEFT LoRA (with r= 32, 50)
4. Hyper-aligned LoRA based transformer (ours)

Hyperparameters	**Value**
Training time	12 hours
Batch size	Auto adjusted based on GPU resources remaining (1 to 5 per GPU)
fp16	True
Learning rate	5e-5, 1e-3
Optimizer	AdamW
Learning Rate Scheduler	Linear decay
Lora_alpha	32
Lora_dropout	0.05

Table 2 Hyperparameters used in all the models for the fine-tuning task

The choice of the hyperparameters mentioned in the above table is completely based on the resource constraints (training steps, batch size and fp16) and the remaining hyperparameters are the default parameters used in their respective papers.

FLAN-T5
Pre-trained FLAN-T5 model is taken and inferenced without any change to the model to measure the zero-shot performance of the model. As expected, the model does not perform well on the given task due to varying unseen complexity of the prediction task. Hence, the results achieved were not satisfactory. Hence, we fine-tuned all the parameters of the model.

Fine tuning FLAN-T5
Fine-tuning all the 250M parameters in the model consumes a lot of computational resources with the risk of overfitting. As expected, the model was able to learn and achieve significantly better results when compared to the pre-trained model.
However, the large number of parameters in the model also introduces high training time. The learning rate was set to a smaller value as fine-tuning was required for all the layers with small updates to their weights.

PEFT LoRA
We compare the results of different models by applying PEFT with LoRA with a rank of 50 (r=50) on the query and value matrices of the attention blocks reducing the trainable parameters to a mere **2.18%,** which is around **5.5M** trainable parameters. Consequently, in our approach, we use LoRA with (r=32), which provides trainable parameters of about **3.5M**. This architecture can be efficiently trained in environments with resource constraints. The batch size used for training in this task was 4 times higher than full fine-tuning. Moreover, the learning rate was set to a larger value due to the addition of new decomposition weight matrices. However, due to the low parameters learning and generation of different kinds of seq2seq tasks, the model does not capture all the important dependencies present in this task domain, hence providing average results. We believe that adding task-specific adaptive layers that can provide learnable task specific weights and parameters can improve the performance of the model significantly.

Fine-tuned Lora Hyperformer

In this study, we implement a single linear-layered hypernetwork, which plays a pivotal role in dynamically generating weights for the adaptive layer. This hypernetwork takes into account the properties of the input data provided to it, allowing it to fine-tune the model's parameters according to the specific features and complexities of the code and documentation pairs. These dynamically generated are then applied to the adaptive layer, which processes the input in a manner that aligns with the task's requirements. To ensure stability and consistent performance, the output of the adaptive layer is further passed through an RMS normalization layer, facilitating the model's adaptability and reducing the risk of overfitting. The performance of the model with the same hyperparameters is shown to outperform the LoRA model significantly.

We experimented by reducing the rank of the LoRA model to 32, which reduces the trainable parameters to **1.4%** and introduce our **hyper-alignment adapter** which brings the total number of trainable parameters to **1.61% (4.1M parameters)** of the total number of parameters in the model. This model was able to surpass the performance of the LLM on the code2doc dataset. Hence, supporting our belief in integrating LoRA with adaptive hypernetworks.

Benchmarks

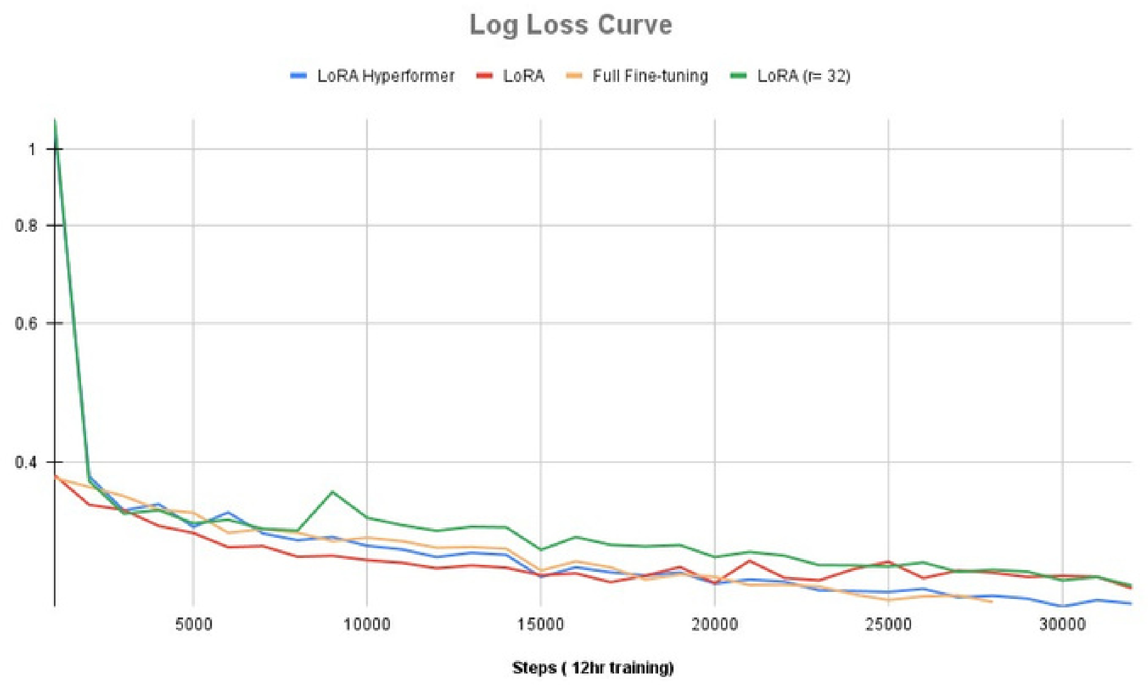


Fig. 8 Log loss training curve of all the models during the fine-tuning task

Figure 8 depicts the learning process of the various models. This curve only indicates that the model is able to learn efficiently and however, should not be confused with the performance of the model. The loss of a model does not always indicate the performance of the model. The importance of the loss function is to assist the model in learning and it does not in anyway indicate the performance of the models accurately.

Model&Method	Trainable parameters (in %)	# Trainable parameters	ROUGE-L	BLEU
FLAN-T5 (no fine-tune)	-	-	8.68	5.49
FLAN-T5 (full fine-tune)	100	247M	23.07	31.1
LoRA (r= 50)	2.18	5.5M	22.34	37.6
LoRA (r= 32)	**1.41**	**3.6M**	22.3	35.8
LoRA Hyperformer (ours, r= 32)	1.61	4.1M	**23.8**	**41.7**

Table 3 Results of different approaches used for the fine-tuning tasks

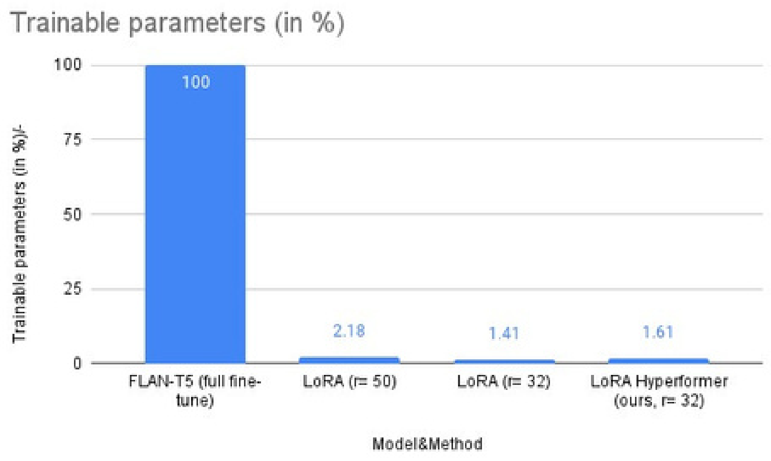


Fig. 9 Comparison of total trainable parameters for the fine-tuning task

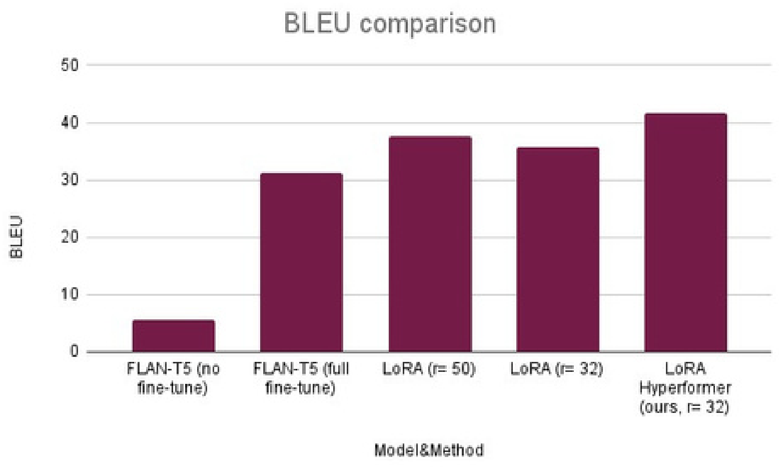


Fig. 10 BLEU Performance comparison of all the models on the test dataset

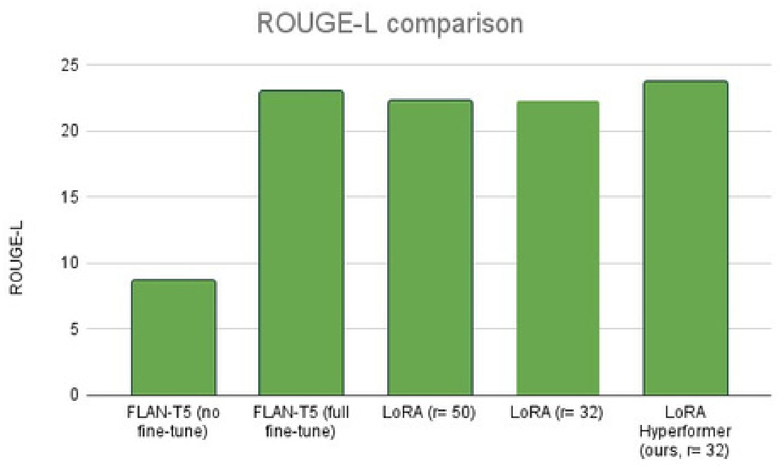


Fig. 11 ROUGE-L Performance comparison of all the models on the test dataset

The above benchmarking is evaluated using the test dataset that was prepared before the fine-tuning. The test dataset was carefully chosen and made sure they are not repeated in the training dataset. Hence, all the models are evaluated on unseen data for a fair evaluation and understanding of the ability of generalization. From the above plots, we can see that the fully fine-tuned model almost had the same log loss curve compared to our LoRA hyperformer after 12 hours. However, the hyperformer as able to significantly outperform the latter, indicating that our approach was able to help the model generalize well.

Conclusion

This project presents a holistic approach that combines LoRA and adaptive layers with hypernetworks to enhance the efficiency and task specific adaptation of transformers architectures for sequence-to-sequence generation tasks. The introduction of the hyper alignment adapter, utilizing hypernetworks to bridge the gap between encoder and decoder, represents a significant step forward. By aligning the outputs of the encoder more effectively with the decoder's requirements, this concept offers a promising avenue for improving the accuracy and relevance of generated sequences.

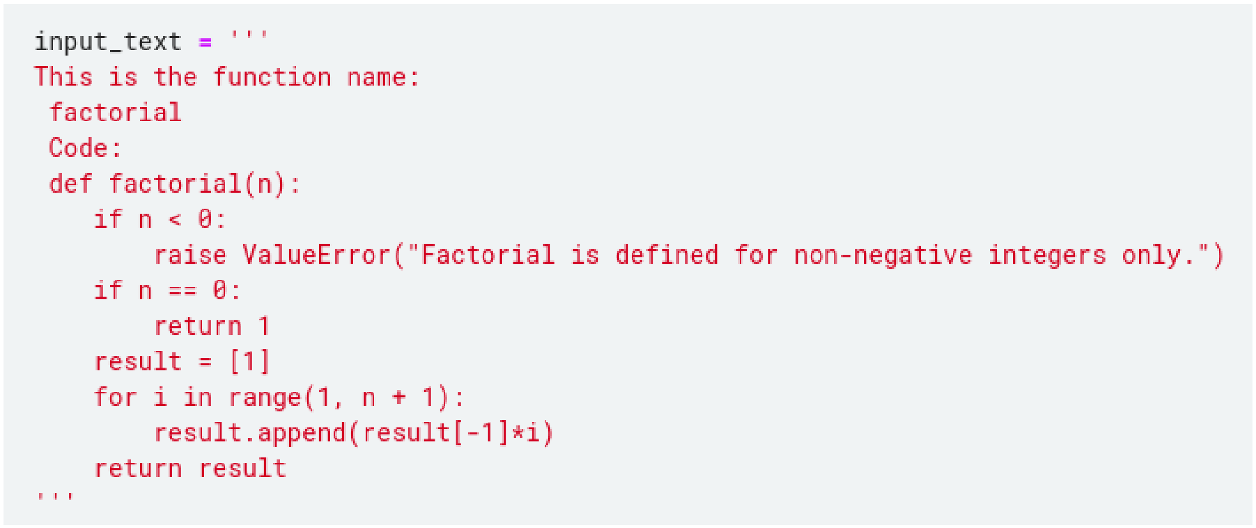
It's crucial to note that this project introduces the foundational concept of hyper alignment adapters in transformers, opening the door to further exploration and refinement. The potential of these adapters in tasks involving diversity between input and output sequences is substantial.

Future research can delve deeper into optimizing the adaptive layers with hypernetworks , exploring different hypernetwork configurations, and leveraging advanced training techniques to fully harness their potential.

APPENDIX:

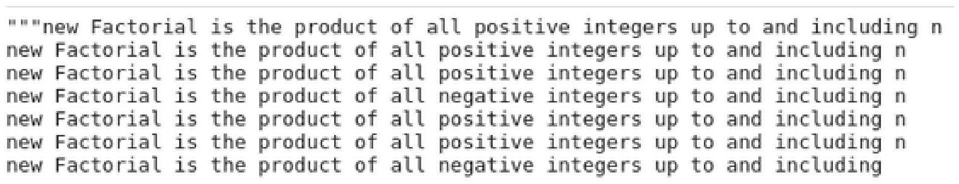
Sample predictions from each of the model:

Input prompt:



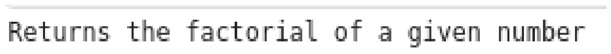
```
input_text = '''
This is the function name:
 factorial
 Code:
 def factorial(n):
    if n < 0:
        raise ValueError("Factorial is defined for non-negative integers only.")
    if n == 0:
        return 1
    result = [1]
    for i in range(1, n + 1):
        result.append(result[-1]*i)
    return result
'''
```

1. **Full fine-tuned model**



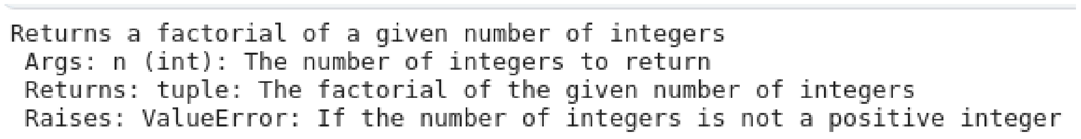
```
"""new Factorial is the product of all positive integers up to and including n
new Factorial is the product of all positive integers up to and including n
new Factorial is the product of all positive integers up to and including n
new Factorial is the product of all negative integers up to and including n
new Factorial is the product of all positive integers up to and including n
new Factorial is the product of all positive integers up to and including n
new Factorial is the product of all negative integers up to and including
```

2. **LoRA model**



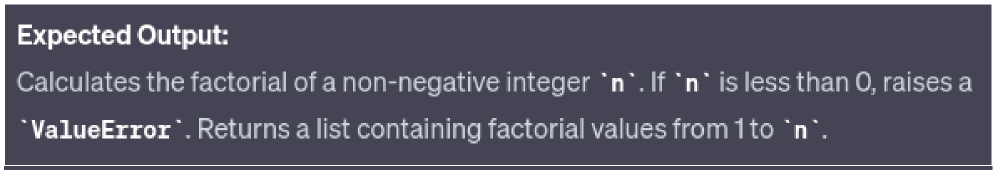
```
Returns the factorial of a given number
```

3. **HyperAdapter model (ours):**



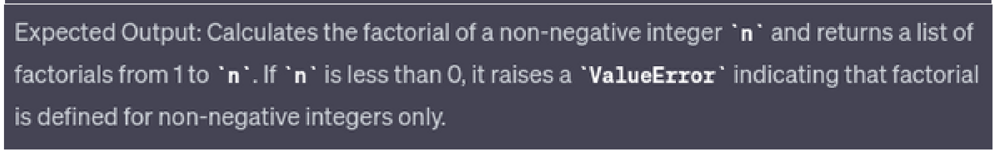
```
Returns a factorial of a given number of integers
 Args: n (int): The number of integers to return
 Returns: tuple: The factorial of the given number of integers
 Raises: ValueError: If the number of integers is not a positive integer
```

4. **Three-shot prompt engineering GPT 3.5**



Expected Output:

Calculates the factorial of a non-negative integer `n`. If `n` is less than 0, raises a `ValueError`. Returns a list containing factorial values from 1 to `n`.



Expected Output: Calculates the factorial of a non-negative integer `n` and returns a list of factorials from 1 to `n`. If `n` is less than 0, it raises a `ValueError` indicating that factorial is defined for non-negative integers only.

References

A Comprehensive Overview of Large Language Models
Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, Ajmal Mian
https://arxiv.org/pdf/2307.06435.pdf

Attention Is All You Need
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin
https://arxiv.org/pdf/1706.03762.pdf

Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer
Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu
https://arxiv.org/pdf/1910.10683.pdf

LoRA: Low-Rank Adaptation of Large Language Models
Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen
https://arxiv.org/pdf/2106.09685.pdf

Instruction Tuning for Large Language Models: A Survey
Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, Guoyin Wang
https://arxiv.org/abs/2308.10792

Fine-Tuning Pretrained Language Models: Weight Initializations, Data Orders, and Early Stopping
Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, Noah Smith
https://arxiv.org/abs/2002.06305

On the Effectiveness of Parameter-Efficient Fine-Tuning
Zihao Fu, Haoran Yang, Anthony Man-Cho So, Wai Lam, Lidong Bing, Nigel Collier
https://arxiv.org/abs/2211.15583

Parameter-efficient Multi-task Fine-tuning for Transformers via Shared Hypernetworks
Rabeeh Karimi Mahabadi, Sebastian Ruder, Mostafa Dehghani, James Henderson
https://arxiv.org/pdf/2106.04489.pdf

Root Mean Square Layer Normalization
Biao Zhang, Rico Sennrich
https://arxiv.org/abs/1910.07467

ROUGE: A Package for Automatic Evaluation of Summaries
Chin-Yew Lin
https://aclanthology.org/W04-1013.pdf

BLEU: a method for automatic evaluation of machine translation
Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu
https://aclanthology.org/P02-1040.pdf