

---

# TOWARDS SMALLER, FASTER DECODER-ONLY TRANSFORMERS: ARCHITECTURAL VARIANTS AND THEIR IMPLICATIONS

---

**Sathya Krishnan Suresh**  
Puducherry Technological University  
satyakrishnan.s@pec.edu

**Shunmugapriya P**  
Puducherry  
pshunmugapriya@gmail.com

## ABSTRACT

In recent times, the research on Large Language Models (LLMs) has grown exponentially, predominantly focusing on models underpinned by the transformer architecture, as established by [1], and further developed through the decoder-only variations by [2]. Contemporary efforts in this field primarily aim to enhance model capabilities by scaling up both the architecture and data volumes utilized during training. However, the exploration into reduce these model sizes while preserving their efficacy remains scant. In this study, we introduce three modifications to the decoder-only transformer architecture—namely ParallelGPT (*p-gpt*), LinearlyCompressedGPT (*lc-gpt*), and ConvCompressedGPT (*cc-gpt*). These variants demonstrate comparable performance to the conventional architecture in code generation tasks, yet benefit from reduced model sizes and faster training processes. We open-source the model weights and the complete codebase for these implementation for further research.

## 1 Introduction

Since the debut of ChatGPT, there has been a notable increase in research on Large Language Models (LLMs) across a broad range of disciplines, made possible by the accessibility of this technology to a diverse user base. This fastly growing field has largely pursued two distinct paths: one aims at either scaling the model dimensions or the training dataset (or both) to enhance performance, while the other concentrates on refining smaller models (ranging from 1B to 7B parameters) with high-quality data. Despite these advances, investigations into the structural modifications of the transformer architecture itself have been relatively overlooked. Recent studies challenge the necessity of perpetually increasing model sizes by demonstrating that the deeper layers of LLMs may have minimal influence on predictive outcomes.

In this work, we explore modifications to the decoder-only transformer architecture to address current challenges in the scalability and practical application of Large Language Models (LLMs). Recognizing the significant impact of model size on the computational overhead of training and inference, we introduce three compact variants—ParallelGPT (*p-gpt*), LinearlyCompressedGPT (*lc-gpt*), and ConvCompressedGPT (*cc-gpt*)—each designed to reduce parameter count while maintaining, or potentially enhancing, model performance.

Our decision to focus on smaller-sized models, ranging from 5M to 10M parameters, stems from several considerations. Primarily, these dimensions facilitate faster training and inference times, critical for iterative development cycles and real-time applications. Additionally, smaller models circumvent the limitations often encountered with quantized models, which despite their reduced computational demands, frequently underperform compared to their full-precision counterparts. This approach not only ensures efficient local execution without the need for specialized hardware but also aligns with our goal to achieve comparable performance to the original GPT architecture with significantly reduced computational resources.

We have pre-trained each model variant on a specialized dataset tailored for data science code completion, envisioning these models as locally-deployed tools for enhancing productivity through single-line code suggestions. Preliminary results from our experiments suggest a lot of potential research within transformer architecture optimization that could reduce the need for scaling models to prohibitively large sizes - hundreds of billions, or even a trillion parameters-currently dominating the field. The remainder of this paper is organized as follows. The GPT architecture

and its related works are discussed in Section 2. The modifications made to the base architecture and the justifications for the modifications are presented in Section 3. The dataset used is discussed in Section 4. The training pipeline, results and comparisons between the different architectures are presented in Section 5 and finally, the future scope, applications of these architectures and the concluding remarks are presented in Section 6.

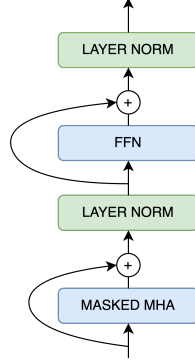


Figure 1: Decoder Block

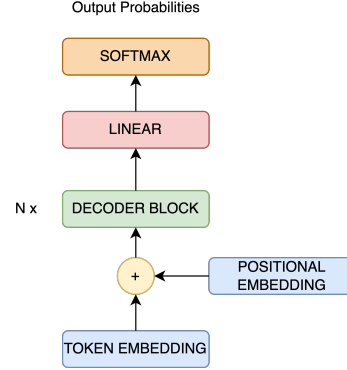


Figure 2: GPT

## 2 Literature survey

### 2.1 Transformer Architecture

Decoder-only transformer architectures represent a significant area of research within the field of natural language processing (NLP). Originating from the broader family of transformer models, which are known for their self-attention mechanisms enabling better parallelization and handling of long-range dependencies, the decoder-only configuration has been specifically tailored for generative tasks.

The architecture of a decoder-only transformer is characterized by its sequence of stacked decoder layers as shown in 1, each consisting of a masked self-attention mechanism followed by a feed-forward neural network. Unlike the encoder-decoder frameworks where both components process inputs and generate outputs, the decoder-only model focuses solely on output generation. This is achieved by training the model to predict the next token in a sequence given the previous tokens, making it inherently suitable for tasks such as text generation, language modeling, and autoregressive prediction.

The masked self-attention mechanism within decoder-only architectures allows each position in the output sequence to attend to all positions up to and including that one, which is instrumental in capturing the nuances of language patterns. This mechanism, coupled with layer normalization and residual connections, helps in stabilizing the learning process and enhancing the model’s ability to generalize from its training data.

### 2.2 Related Work

The development of Large Language Models (LLMs) has been significantly driven by advancements in the transformer architecture, first introduced by [1] and extended in various decoder-only variations such as those by [2]. These models have traditionally focused on scaling up through increased architectural complexity and extensive training datasets. However, the need for more size- and compute-efficient models has led to innovative approaches in model design, aiming to retain performance while reducing resource demands.

Our study introduces three novel variants of the decoder-only transformer architecture aimed at efficiency: ParallelGPT (*p-gpt*), LinearlyCompressedGPT (*lc-gpt*), and ConvCompressedGPT (*cc-gpt*). These models demonstrate performance on par with traditional architectures in code generation tasks, but with the advantage of smaller model sizes and faster training times. We provide open-source access to both the model weights and the complete codebase to facilitate further exploration in this area.

In parallel, the Funnel-Transformer [3] proposes an architecture that enhances efficiency by progressively compressing the sequence of hidden states, significantly lowering computational costs while potentially increasing model capacity. Unlike traditional models with constant sequence length, the Funnel-Transformer uses a pooling mechanism to reduce

sequence length across layers, which conserves computational resources and allows for model depth or width expansion without additional costs.

Similarly, the "MobiLlama" [4] study presents a small language model optimized for resource-constrained environments. This model incorporates a parameter-sharing scheme within transformer blocks and focuses on reducing both pre-training and deployment costs, aligning well with our objectives of efficiency and reduced resource usage.

Further contributing to the field, Grouped Query Attention [5] restructures the attention mechanism by grouping inputs, allowing attention within these confines and reducing complexity from quadratic to linear. This facilitates handling of longer sequences and larger datasets. Conversely, Multi-Query Attention [6] extends traditional attention mechanisms by allowing multiple queries within a single head, enhancing the model's ability to distill diverse information within the same layer and enriching its expressive capabilities.

These collective advancements depict a clear trajectory towards not only enhancing the performance and scalability of LLMs but also towards making these models more adaptable to constraints of size and computational affordability.

### 3 Architectural modifications

In this section, we introduce three novel architectures derived from the traditional GPT architecture to address various limitations in training and inference. These architectures are designed to enable faster training and inference, overcome common limitations encountered with quantized models, and facilitate local execution without specialized hardware. The three proposed architectures are ParallelGPT (*p-gpt*), LinearlyCompressedGPT (*lc-gpt*), and ConvCompressedGPT (*cc-gpt*).

#### 3.1 ParallelGPT

The  $N$  decoder blocks in a *gpt* architecture are stacked vertically on top of each other and the input to each decoder block will be the output of the previous decoder block. It can be seen here that the time taken for the input to go through this architecture will increase as the number of decoder blocks increase and the dimensionality of the model also has to increase to make sure that the information in the tokens is propagated through the blocks. Recent studies have also shown that the deeper layers [7] have little effect on the predictions made by the model.

Both of these limitations can be overcome by splitting the decoder part of the *gpt* architecture into two parts (or more, which is for future research), each having an equal number of decoder blocks ( $N/2$ ). To train a model of this architecture the dimensionality of the embedding model was made to be twice the dimensionality of the decoder ( $D_{\text{MODEL}}$ ). Hence, the output vectors of the embedding model will be of dimensions  $D_{\text{MODEL}} * 2$  out of which the first half of each vector is sent to one block and the second half to the other block. This is done to ensure that each block learns a knowledge that is different from the other block. The outputs of the two blocks are combined using a weighted sum and are then fed to the classification head. Equations 1, 2 and 3 give a mathematical description of the above and 3 represents the architecture.

$$x = \text{embedding}(\text{tokens}), \quad x \in \mathbb{R}^{D_{\text{MODEL}} \times 2} \quad (1)$$

$$x_1, x_2 = x[:, :, : D_{\text{MODEL}}], x[:, :, D_{\text{MODEL}} + 1 :], \quad x_1, x_2 \in \mathbb{R}^{D_{\text{MODEL}}} \quad (2)$$

$$\text{probability} = \sigma(\text{linear}(w \cdot \text{decoder}_1(x_1) + (1 - w) \cdot \text{decoder}_2(x_2))) \quad (3)$$

Training a model of this architecture has the following benefits - 1. faster training, as each block can be trained on a separate compute node in parallel, 2. during inference the block with the lesser weight ( $w$  in 3) can be dropped for faster inference but this will result in a slight reduction in the performance of the model, 3. the number of parallel blocks can be increased further which we hypothesize, might produce better results and an even faster training. However with increasing the number of parallel blocks the dimensionality of the embedding also has to be increased to make sure that each block learns a different knowledge, which will increase the number of parameters in the model. This increase in the number of parameters will only be a disadvantage if we don't decide to drop a few blocks during inference.

#### 3.2 LinearlyCompressedGPT

We hypothesize that, much of the information that can be learned from the embeddings of each token might happen in the initial few blocks of the decoder (depending on the dimensions of the model), which brings into question the need for each block having the same dimensions. If each of the  $N$  decoder block has the same dimension this results in the model having much more parameters than that is actually needed to achieve the desired performance.

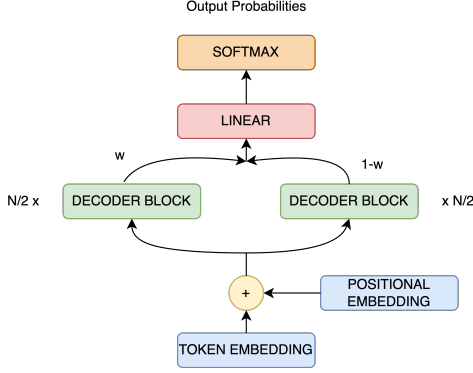


Figure 3: ParallelGPT

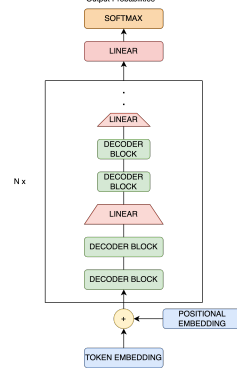


Figure 4: LinearlyCompressedGPT

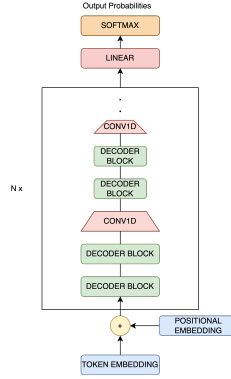


Figure 5: ConvCompressedGPT

To this end, we propose an architecture in which the dimension of the decoder blocks keeps reducing by half as the embeddings pass through the architecture. This concept is inspired by the architectural design of various image classification convolutional networks (CNNs), where the dimensions of an image are systematically reduced as it passes through successive layers of the network. To make sure that the dimensions of the output vectors of a decoder block matches up to the dimensions of the successive decoder block we introduce a dense (linear) layer after two decoder blocks of the same dimension to map the vectors to half of the original dimension. A dense layer is placed after every couple of decoder blocks, since placing them after each decoder block will produce an output vector of a small dimension from which the classification head will have a hard time to predict the next token. The architecture for *lc-gpt* is presented in fig. 4 and eq. 4, 5 represent the equations of the architecture discussed.

$$x = \text{decoder}_2(\text{decoder}_1(x)), x \in \mathbb{R}^{D_{\text{MODEL}}} \quad (4)$$

$$x = \text{linear}(x), x \in \mathbb{R}^{D_{\text{MODEL}}/2} \quad (5)$$

The *lc-gpt* architecture can reduce the number of parameters in the model in terms of millions or billions depending on the size of the original *gpt* architecture with the same number of layers and dimensions. The extra parameters introduced by the dense layers will be negligible compared to the amount of parameters reduced by reducing the dimensions of the decoder blocks. This reduction in dimension also forces the deeper decoder blocks to perform better with the smaller dimension vectors that are fed to it.

### 3.3 ConvCompressedGPT

ConvCompressedGPT builds on the concept of LinearlyCompressedGPT by replacing the dense layers with 1D convolutional layers. This architecture retains the advantages of LinearlyCompressedGPT while leveraging the benefits of convolutional layers, such as weight sharing and improved positional pattern recognition.

Convolutional layers apply filters across sequence positions, effectively capturing positional dependencies and patterns. This approach may enhance the model's ability to predict the next token by incorporating contextual information across the sequence. The architectural design for ConvCompressedGPT is presented in Figure 5.

The ConvCompressedGPT architecture offers multiple benefits, primarily due to its use of 1D convolutional layers. Firstly, this approach leads to a reduction in the total number of model parameters, similar to LinearlyCompressedGPT. By progressively decreasing the dimensions, ConvCompressedGPT requires fewer computational resources and uses less memory, enhancing its efficiency. Secondly, the convolutional layers provide inherent advantages by capturing positional patterns and dependencies within the input sequence, potentially improving the model’s ability to make accurate predictions. This characteristic is especially beneficial for handling sequential data, where the positional context plays a crucial role. Lastly, the ConvCompressedGPT structure is highly scalable, allowing for further extension by adding more convolutional layers. This scalability provides flexibility in adapting the architecture to various contexts and tasks, making it suitable for a wide range of applications.

## 4 Dataset

In this study, we utilize the *codeparrot-ds* dataset from HuggingFace [8], [9], which comprises code snippets that specifically leverage data science libraries such as *numpy*, *pandas*, *scikit-learn*, and *matplotlib*. The selection of this dataset was strategic; it focuses narrowly on four well-defined libraries. This specificity is advantageous for our research objectives, as employing a more expansive dataset encompassing a broader array of libraries might impede the performance of the models we train, particularly given their smaller scale.

For practical and experimental purposes, we tailored the dataset size to suit our computational constraints and to facilitate rapid iteration cycles. Although the original *codeparrot-ds* dataset contains 384,000 training examples and 3,320 validation examples as detailed on the HuggingFace repository, we randomly selected a subset of 10,000 examples for training and 1,000 for validation. This reduction was necessitated by the limited computing resources available to us.

## 5 Training and Results

### 5.1 Training

#### 5.1.1 Tokenization

Tokenization is a critical preprocessing step in natural language processing (NLP) and involves breaking down text into smaller elements known as tokens, which are then mapped to unique integer identifiers. There are three main tokenization strategies: word-level, character-level, and sub-word tokenization. Each strategy has its advantages and disadvantages, depending on the specific application and the required granularity of text representation.

Sub-word tokenization, commonly used by state-of-the-art large language models (LLMs), combines the benefits of word-level and character-level tokenization. It splits words into smaller meaningful units, allowing the model to generalize across similar words while maintaining a manageable vocabulary size. However, sub-word tokenization can lead to ambiguity, especially for out-of-vocabulary words, and may require more complex tokenization algorithms.

In this work, we opt for character-level tokenization. This decision is driven by the architecture design, where the reduced dimensions in two of the three proposed models could hinder the ability of the classification head to predict the next token when using word-level or sub-word tokenization. The increased vocabulary size associated with these tokenization strategies could complicate the learning process and lead to slower convergence.

Character-level tokenization offers several advantages for our task. It simplifies the tokenization process, eliminating the need for intricate splitting rules, and it aligns well with our focus on single-line code completion. By employing character-level tokenization, both the training and testing datasets were processed into a vocabulary of 2,117 unique tokens.

#### 5.1.2 Training loop

The training of our models is conducted with a batch size of 64. Given that our objective is to develop a model for single-line code completion, we set the context length to a small 256 tokens. This context length helps maintain the required focus on code snippets while avoiding excessive memory consumption.

To reduce overfitting, we apply a dropout of 0.1 across the layers. For our loss function, we employ cross-entropy loss, a widely used metric for classification tasks. The model weights are updated using the AdamW optimizer with a learning rate of  $3e-4$  and a weight decay of  $1e-3$ . The AdamW optimizer is selected for its robustness in training large models and its ability to converge quickly with minimal parameter tuning.

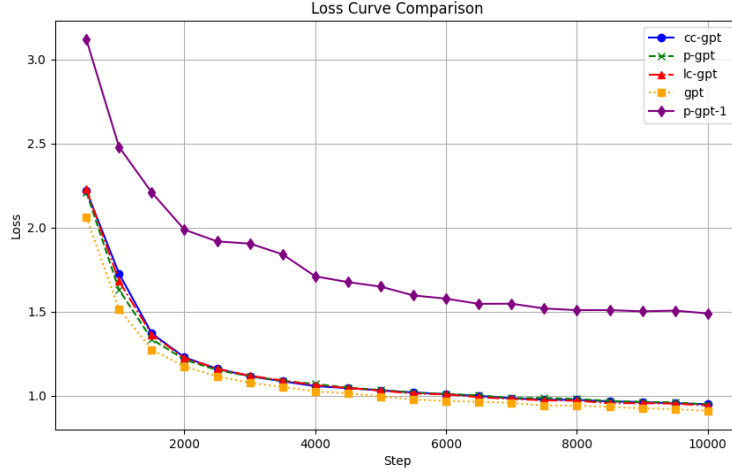


Figure 6: Loss comparison b/w the 4 models

Name	Params (M)	Size (MB)	Training Time (min)
<i>gpt</i>	8.82	33.66	25.35
<i>p-gpt</i>	9.74	37.14	26.15
<i>p-gpt (1-block)</i>	6.19	23.60	26.15
<i>lc-gpt</i>	5.65	21.54	20.68
<i>cc-gpt</i>	5.65	21.54	21.68

Table 1: n\_layers=4, d\_model=384, n\_heads=6

Training proceeds for 10,000 steps and to monitor the model’s performance during training, we evaluate it every 500 steps using the test set, and the results are logged for analysis and validation (refer to Figure 6 for an overview of the loss trends). This evaluation frequency allows us to track the model’s progress and adjust the training approach if needed.

## 5.2 Results

The results depicted in Figure 6 indicate that, with the exception of the *p-gpt-1* model, all other models exhibit performance metrics similar to those of *gpt*. This outcome aligns with our earlier hypotheses regarding model architecture and the parallelization strategy used within the decoder blocks. The degradation in the performance of *p-gpt-1* can be attributed to its inference method. During inference, the model omits the *decoder\_1* block if its weight is less than 0.5, or alternatively the *decoder\_2* block, as illustrated in Equation 3. This intentional reduction in the model’s complexity likely contributes to the observed performance gap. However, the performance of *p-gpt-1* could potentially be improved with extended training, as the loss trends suggest a continued decrease even in the later training stages.

As detailed in Table 1, several of the tested models achieved performance metrics similar to those of *gpt*, the traditional decoder-only transformer model, while employing fewer parameters and yielding faster training times. Notably, both *lc-gpt* and *cc-gpt* demonstrated a **36%** reduction in parameter count compared to *gpt*. This reduction in parameters has significant implications for larger-scale models with more complex architectures, extended context lengths, and additional decoder blocks.

The reduction in parameters also correlates with reduced memory requirements for model deployment. This efficiency gain has far-reaching effects, particularly for larger-scale models, offering potential benefits such as faster inference times and enhanced feasibility for on-device deployment. These advantages could play a crucial role in scenarios where computational resources and memory capacity are limited.



## 6 Conclusion

The architectural modifications analyzed in this study not only enhance the efficiency of the transformer models in terms of faster inference and training with fewer parameters, but also underline the need to further explore this area of research. The findings suggest that continued innovation in transformer architectures could significantly expedite the deployment of on-device language models that operate efficiently without specialized hardware. This promises substantial advancements in making AI technologies more accessible and sustainable.

Particularly, further investigations into the *p-gpt* architecture are crucial. A deeper understanding of the knowledge processed by each of the parallel blocks within *p-gpt* could reveal how to structure the training process to tailor each block for specific tasks more effectively. This targeted training approach could lead to breakthroughs in model specialization and efficiency.

Moreover, both *lc-gpt* and *cc-gpt* architectures present rich opportunities for further exploration. Experimenting with scaling up the dimensions—from lower to higher within the processing flow—could provide insights into optimizing computational resources and model performance. Additionally, incorporating pooling layers, as introduced in [3], in place of traditional linear or convolutional layers could offer a novel method to reduce complexity and enhance model learning capabilities. These avenues not only hold the promise of refining the efficiency and effectiveness of transformer models but also pave the way for groundbreaking applications in real-world scenarios.

## References

- [1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.
- [2] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018. Available: [https://cdn.openai.com/research-covers/language-unsupervised/language\\_understanding\\_paper.pdf](https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf).
- [3] Zihang Dai, Guokun Lai, Yiming Yang, and Quoc Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. In *Advances in Neural Information Processing Systems*, 2020.
- [4] Omkar Thawakar, Ashmal Vayani, Salman Khan, Hisham Cholakkal, Rao Muhammad Anwer, Michael Felsberg, Timothy Baldwin, Eric P. Xing, and Fahad Shahbaz Khan. Mobillama: Towards accurate and lightweight fully transparent gpt, 2024.
- [5] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints, 2023.
- [6] Noam Shazeer. Fast transformer decoding: One write-head is all you need. 2019. cite arxiv:1911.02150.
- [7] Andrey Gromov, Kushal Tirumala, Hassan Shapourian, Paolo Gloriosi, and Daniel A. Roberts. The unreasonable ineffectiveness of the deeper layers. *arXiv preprint arXiv:2403.17887*, 2024.
- [8] Hugging Face. Codeparrot-ds-train dataset. <https://huggingface.co/datasets/huggingface-course/codeparrot-ds-train>, 2021. Accessed: 2024-04-16.
- [9] Hugging Face. Codeparrot-ds-valid dataset. <https://huggingface.co/datasets/huggingface-course/codeparrot-ds-valid>, 2021. Accessed: 2024-04-16.
- [10] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio Cesar Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Conti Kauffmann, Gustavo Henrique de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sebastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023.
- [11] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A survey of transformers. *arXiv preprint arXiv:2106.04554*, 2021.
- [12] Wenfeng Zheng, Gu Gong, Jiawei Tian, Siyu Lu, Ruiyang Wang, Zhengtong Yin, Xiaolu Li, and Lirong Yin. Design of a modified transformer architecture based on relative position coding. *International Journal of Computational Intelligence Systems*, 16(1):1–17, 2023.
- [13] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11106–11115. AAAI, 2021.

- [14] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395*, 2024.
- [15] Li Shen and Yangzhu Wang. Tcct: Tightly-coupled convolutional transformer on time series forecasting. *Neuro-computing*, 480:131–145, 2022.
- [16] Andrej Karpathy. nanogpt: Minimal gpt implementation with pytorch. <https://github.com/karpathy/nanoGPT>, 2023.
- [17] Hugging Face. Chapter 7.6: Transformers and transfer learning, 2024. Accessed: 2024-04-19.