University of Science and Technology of China, Suzhou campus

School of Software Engineering

Course	**NLP**
Document Type	**Final Report**
Course Level	**Masters**
Student Name	**MD RASEL MAHMUD（周明）**
Student ID	**SL23225002**
Course offered by	**Prof. Guo Yan**

Title

Understanding nanoGPT: From Code to Dataset behind the scenes

Introduction

In the rapidly growing field of artificial intelligence (AI), Generative Pretrained Transformers (GPT) have transformed natural language processing (NLP). Nowadays, many firms employ NLP to solve real-world problems such as translation, text production, grammatical correction, topic summaries, chatbots, customer support and so on. The nanoGPT project is also part of the NLP under the hood. As a result, the purpose of this project is to give a full overview of nanoGPT in order to gain practical experience, including everything from the underlying code to the datasets used.

The nanoGPT is a streamlined and quite good model for practicing or fine-tuning medium-sized GPTs. This version, inspired by minGPT, emphasizes simplicity and speed without compromising effectiveness. nanoGPT is a compact variant of the Generative Pre-trained Transformer (GPT) model. It is a smaller and more lightweight version that aims to achieve comparable language generation capabilities while reducing the computational resources required for training and inference. Typically, it has a smaller number of parameters and model layers compared to its larger counterparts like GPT-3 or GPT-4, Claude-3-Opus, ChatGLM, and so on. This makes nanoGPT suitable for learners who want to learn from scratch to advance using a project basis.

Objectives

[1]. To understand & master the working process of nanoGPT.
[2]. To perform and implementation of nanoGPT different dataset.
[3]. In order to model performance test to use variety of train data size.

Requirements

[1]. Software installations, such as access to Python IDE online Jupiter notebook or localhost (i.e. Cuda, GPU).
[2]. Write and modify code with debugging step by step while execution.
[3]. Demonstration repository code to understand the whole report.

A. Working Procedure

1) Data & Code Preparation

[1]. Dataset Loading:

Downloads the dataset for my case ‘news’ and ‘quotes’ dataset and reads it into a string. Loads a CSV file containing quotes into DataFrame. Extracts the 'quote' column. Concatenates all quotes into a single string, separated by new lines, here is data prepare file quotes and news.

[2]. Mapping, Encode, Decode:

Identifies all unique characters in the data and sorts them. Creates a dictionary mapping each character to a unique integer (stoi) and the reverse mapping (itos). Encoding converts a string into a list of integers based on ‘stoi’. Decoding converts a list of integers back into a string using ‘itos’.

[3]. Data split and export:

Splits the data into training (90%) and validation (10%) sets. Encodes these sets into lists of integers. Converts the integer lists to numpy arrays with uint16 data type. Saves the training and validation data to binary files (train.bin and val.bin). Saves the meta information (vocab_size, itos, stoi) as a pickle file (meta.pkl) for later use in encoding and decoding.

[4]. Config, Checkpoint, Logs

“out_dir” Directory for output files. “eval_interval” Frequency of evaluation during training. “eval_iters: Number of iterations for evaluation. “log_interval” Logging frequency. “wandb_log” Toggle for Weights & Biases logging. “wandb_project” Project name for Weights & Biases. “wandb_run_name” Run name for Weights & Biases.

[5]. Training Params & Model config.

“Dataset” Dataset naming variable. “gradient_accumulation_steps” Number of steps to accumulate gradients. “batch_size” Batch size for training. “block_size” Maximum context length. “n_layer” Number of transformer layers. “n_head” Number of attention heads. “n_embd” Embedding dimension. “dropout” Dropout rate.

[6]. Learning rate & optimize.

learning_rate: Initial learning rate. “max_iters” Maximum iterations. “lr_decay_iters” Iterations after which learning rate starts decaying. “min_lr” Minimum learning rate. “beta2” Second moment decay rate for Adam optimizer. In my project iteration used for “quotes” dataset 5000 and “news” dataset 10000. “warmup_iters” Iterations for learning rate warmup.

[7]. Train & Bench

“train.py” file contains the training loop for the model. It handles loading the data, initializing the model, defining the loss function and optimizer, and iterating through the training data to update the model parameters. This script also includes validation steps to evaluate the model's performance on a separate validation dataset and may contain checkpoints to save the model at various stages of training.

“bench.py” script appears to be designed for benchmarking the performance of the model or specific components of the model. This script likely contains functions and methods to measure the efficiency, speed, and resource utilization of the model during training and inference. Benchmarking is crucial to understand the computational requirements and to optimize the model for better performance on different hardware setups.

[8]. Model, Configurator, sampling

“model.py” file defines the architecture of the model. It contains the implementation of the GPT model, including the transformer layers, attention mechanisms, and any additional layers required for the model. This file is crucial as it outlines how the data flows through the network, how the layers are connected, and how the model processes input data to generate predictions.

“configurator.py” file does configurations and hyperparameters for setting up the model training environment. This includes parameters such as learning rate, batch size, number of epochs, model architecture details (like number of layers and hidden units), and data paths. The configurator ensures that the training process is flexible and can be easily adjusted by modifying the configuration parameters.

“sample.py” script is used for generating text samples from the trained model. It loads the trained model and uses it to generate new text sequences based on a given prompt or seed text. This is an important part of evaluating the model's performance and understanding its ability to generate coherent and contextually relevant text.

2) Model Define

[1]. Transformer Architecture:

Before briefing transformer, we need to know an overview of NLP model timeline. The evolution of natural language processing models began in 1990

with the Elman Network, using recurrent neural networks to encode words as vectors, though it struggled with word disambiguation. In 1992, Jürgen Schmidhuber introduced the Fast Weight Controller, which later proved analogous to the unnormalized linear Transformer. IBM's alignment models for statistical machine translation emerged in 1993, and by 1997, precursors to large language models using LSTMs were proposed. A significant leap occurred in 2001 with the use of a one-billion-word corpus for word disambiguation.

The effectiveness of large neural networks was demonstrated by AlexNet in 2012, prompting a shift from statistical approaches. In 2014, a seq2seq model using LSTMs and GRUs was introduced, along with Bahdanau et al.'s additive attention mechanism. Luong et al. refined attention models in 2015, and by 2016, Google Translate adopted neural networks, surpassing its statistical counterparts. The transformative "Attention is All You Need" paper in 2017 introduced the original transformer model, enhancing training efficiency. Subsequent innovations included ELMo's contextualized embeddings in 2018 and BERT's encoder-only transformer model. By 2020, transformers outperformed RNNs in vision and speech tasks, with the pre-LN Transformer addressing convergence issues. The trend continued with the deployment of massive uni-directional transformer models like GPT-3 in 2023, solidifying the transformer architecture's dominance in NLP.

Transformer architecture is a deep learning model developed by Google, introduced in the 2017 paper "Attention Is All You Need." Unlike earlier models that relied on recurrent neural networks (RNNs) or convolutional neural networks (CNNs), Transformers use a self-attention mechanism to process input data in parallel, making them more efficient and scalable. The architecture consists of an encoder-decoder structure where both components are made up of layers of self-attention and feed-forward neural networks. Key features include multi-head attention, which allows the model to focus on different parts of the input simultaneously, and positional encoding, which provides the model with information about the position of tokens in the sequence. Transformers have revolutionized natural language processing (NLP) by significantly reducing training time and improving performance on a variety of tasks and have been adapted for use in other domains such as computer vision and audio processing. So, nanoGPT uses a transformer model with self-attention mechanisms to process sequences of tokens and generate predictions.

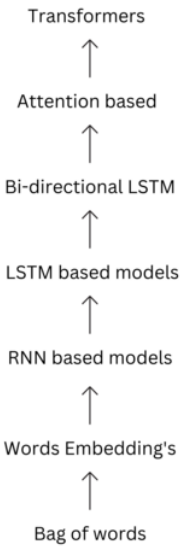

Figure 1-1: *Timeline of NLP*

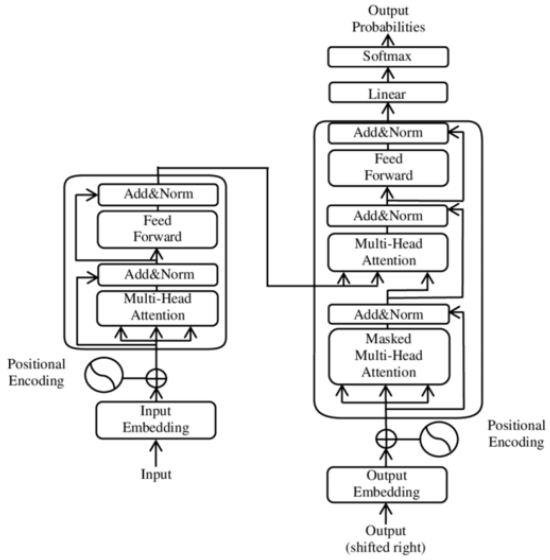

Figure 1-2: *Transformer Architecture*

[2]. Configurable Parameters:

The model size, number of layers, and other hyperparameters are defined based on the specific requirements and computational resources as I explain in code working procedure. Parameters encompass various aspects of training configuration, such as output directories, evaluation intervals, logging frequency, and checkpoint saving criteria. Additionally, the code supports integration with Weights & Biases for experiment tracking.

Key parameters also define the dataset and training process, including dataset selection, gradient accumulation steps, batch size, and block size. Model configuration parameters specify the number of transformer layers, attention heads, embedding dimensions, and dropout rates, which collectively determine the model's complexity and capacity.

Other configuration parameters such as learning rate, maximum iterations, learning rate decay schedule, minimum learning rate, and Adam optimizer settings (beta2) are adjustable to control the training dynamics and convergence stability. Warmup iterations are included to gradually increase the learning rate at the start of training, while device and compilation settings ensure compatibility with various hardware setups, including CPU and GPU.

3) Training Process

[1]. Loss Function:

A loss function plays a critical role in training the model by quantifying the difference between the predicted and actual token sequences. The commonly used loss function is cross-entropy loss, which measures the discrepancy between the model's predicted probability distribution and the true distribution of the target tokens. By minimizing this loss, the model's predictions become more accurate over time. The cross-entropy loss function is particularly effective for classification tasks like language modeling, where the goal is to predict the next token in a sequence. During training, the optimizer adjusts the model's parameters to reduce this loss, thereby improving the model's ability to generate coherent and contextually relevant text.

[2]. Optimization:

Optimization techniques such as the AdamW optimizer are employed to iteratively adjust model parameters, enhancing performance over time. AdamW combines the benefits of adaptive learning rates and weight decay regularization, leading to more stable and efficient training. By dynamically adjusting the learning rate for each parameter and applying weight decay to prevent overfitting, AdamW helps the model converge faster and achieve better generalization, ultimately improving its ability to generate coherent and accurate text.

[3]. Back Propagation:

Backpropagation involves computing gradients of the loss function with respect to the model's parameters and propagating these gradients backward through the network. This process updates the weights of the model, enabling it to learn from errors. By iteratively adjusting the weights based on the gradients, the model improves its predictions over time, effectively minimizing the loss and enhancing its performance in generating accurate and contextually relevant text.

B. Model varying on different Datasets

1. **Model Complexity:** Larger models have more parameters and layers, enabling them to learn and represent more complex patterns in data. This results in better performance but also requires more computational resources and memory.

2. **Training Data:** Different datasets vary in size and complexity. A larger dataset provides more diverse examples, helping the model generalize better but also necessitating more storage and processing power.

3. **Computational Resources**: The available hardware (GPUs, TPUs) can limit the feasible size of the model and the dataset it can process effectively.

4. **Use Case:** Specific applications may require different model sizes. For instance, smaller models may suffice for simple tasks, while more complex tasks benefit from larger models trained on extensive datasets.

5. **Training Time:** Larger models and datasets significantly increase training time, which can be a constraint depending on the project's deadlines and available computational power.

6. **Fine-Tuning:** The pre-trained model can be fine-tuned on specific datasets to adapt it to tasks or domains, enhancing its performance on specialized applications.

These factors collectively determine the optimal model size and the appropriate training dataset to balance performance, efficiency, and resource constraints. Below table-1 shows model training two types of datasets, and it smartly showcases each of them depending on dataset.

Table 1: *Demonstrate the model training variation on different dataset*

	“News” Dataset	“Quotes” Dataset
Iteration	5000	10,000
Loss Function	1.2356	0.0995
No. of Params	11.06M	10.66M
Time (ms)	12518.99	11461.66
MFU (%)	3.77	3.67
Vocab. Size	103	1,136
Train token	351,651	89,736,025
Val token	39,073	9,970,670

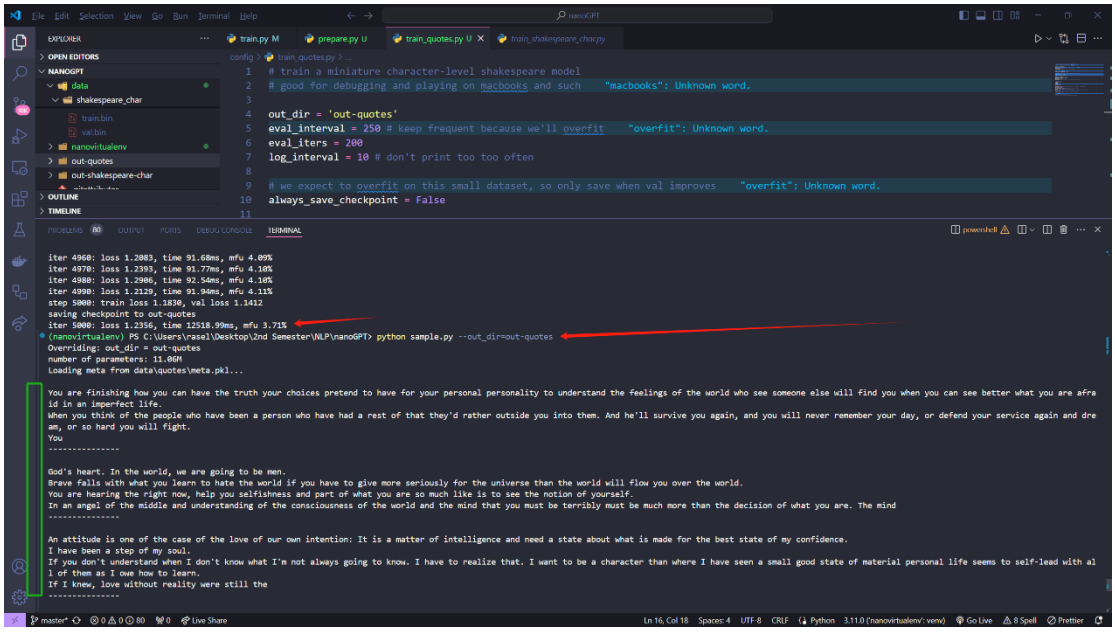

Figure 2-1: *Quotes dataset output sampling*

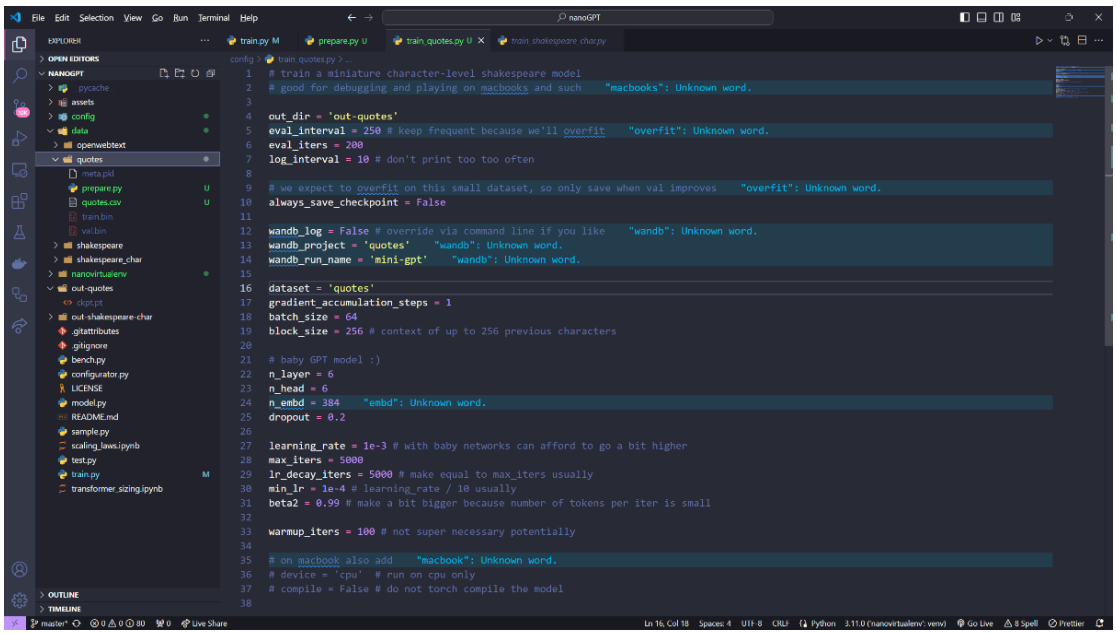

Figure 2-2: *Quotes dataset training file config*

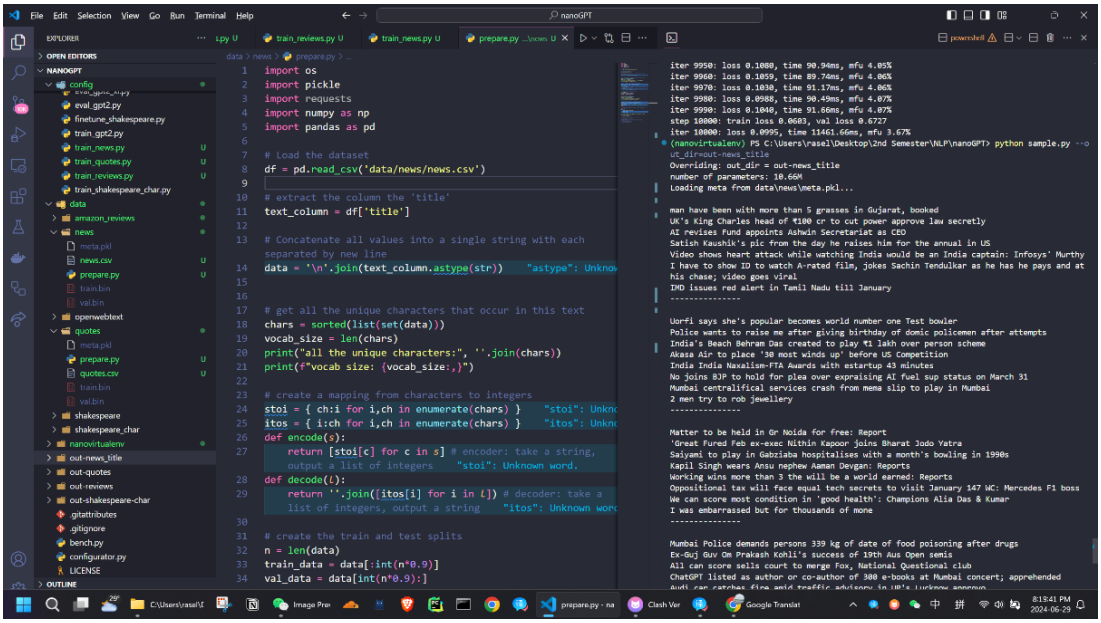

Figure 2-3: News dataset output sampling result

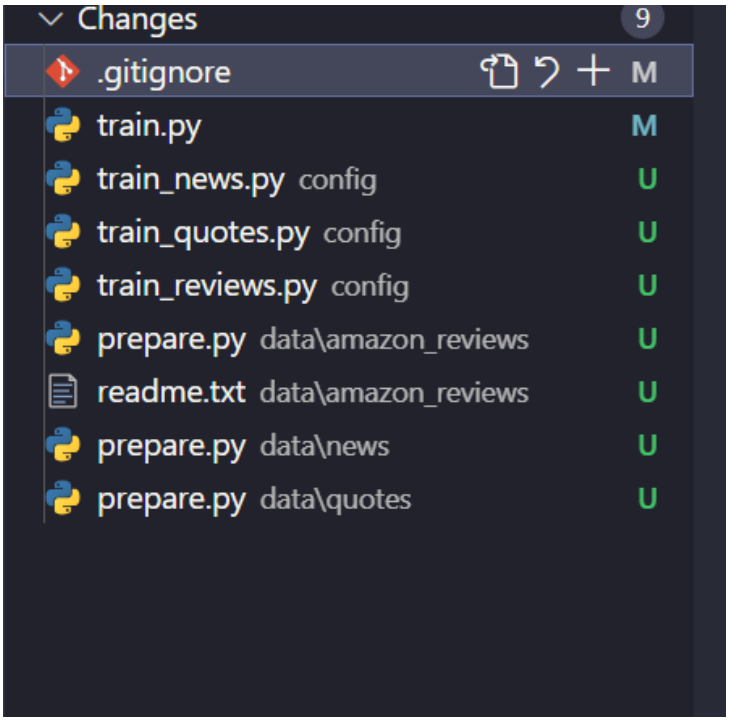

Figure 2-4: For fine-tune modified code files

Conclusion

I successfully trained a nanoGPT model on new datasets of quotes and news. Sampling results and table 1 shows nanoGPT demonstrate its capability to generate corelate and contextually relevant text, reflecting the efficacy of the training process. By undertaking the nanoGPT project, I gained a comprehensive understanding of the transformer architecture and its essential components, such as self-attention mechanisms, which are crucial for natural language processing tasks. I learned the importance of efficient data preprocessing and tokenization, converting text data into a format suitable for model training.

Adjusting various hyperparameters, such as learning rate, batch size, and model depth, taught me how to optimize training performance and adapt the model to different datasets. Exploring optimization techniques, particularly the AdamW optimizer, highlighted the significance of adaptive learning rates and regularization in effective model training. Reinforcing my understanding of backpropagation and gradient descent, I appreciated their roles in updating model weights and minimizing loss functions. Monitoring training progress through metrics like loss and memory footprint utilization (MFU), and evaluating model performance regularly, emphasized the importance of preventing overfitting.

Applying nanoGPT to a new dataset provided practical experience in training and fine-tuning language models, enhancing my ability to generate contextually relevant text. Additionally, using tools like Weights & Biases for experiment tracking demonstrated the value of managing and analyzing training runs efficiently. Overall, this project enriched my understanding of machine learning principles and equipped me with practical skills to apply these concepts in real-world scenarios.

References

[1]. Transformer (deep learning architecture). (2024, June 28). In Wikipedia. https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)
[2]. https://github.com/raselmahmud-coder/nanoGPT
[3]. https://www.kaggle.com/datasets/manann/quotes-500k
[4]. https://www.kaggle.com/datasets/rishikeshfulari/news-dataset-for-headline-generation