Week 8 Report

Mu Junrong

Abstract

This report focuses on the training process and findings of the small transformer-based language model using the nanoGPT framework. The report explains the training loop in detail, including the model architecture and training loop. The training and validation curves are plotted to analyse the learning progress over time. Inference was performed using a modified `sample.py` script, generating continuations from given prompts.Inference is done and the outputs were compared the trained model from Week 7 report, and the causes of the difference are explained.

1 Introduction to nanoGPT

Currently, large language models (LLMs) are designed based on the Transformer architecture. The GPT (Generative Pre-trained Transformer) models (i.e., which includes only the decoder stack), originally introduced by OpenAI, has demonstrated strong capabilities in text generation, completion, summarization, and various other language tasks. However, training these models from scratch can be complex.

To simplify the process, nanoGPT was designed by Andrej Karpathy (Karpathy (2022)). nanoGPT is a minimal, streamlined, and efficient implementation of the GPT architecture in PyTorch. It focuses on code clarity and performance, offering a relatively small codebase that can train models on modest hardware with a few million parameters. nanoGPT is capable of training functioning generative models on character-level or tokenized datasets.

1.1 nanoGPT architecture

nanoGPT implements the standard decoder-only Transformer model (Vaswani et al. (2017)). The model consists of the following key components:

- Embedding Layer: Converts token IDs into dense vectors.
- Multiple Transformer Blocks: Each block includes multi-head self-attention, feed-forward networks (MLPs), layer normalisation, and residual connections.
- Output Head: A linear projection from hidden states to vocabulary logits for next-token prediction.

In the training process, the model is initialised with 12 layers, 12 attention heads, 768-dimensional embeddings and a block size of 1024 tokens per sequence.

These settings roughly correspond to the configuration of the original GPT-2 small (124M) model.

1.2 Dataset: Shakespeare dataset

For the training experiment, the Shakespeare dataset was used, which is a character-level corpus composed of lines from Shakespearean plays. The dataset is small in size but sufficient for training nanoGPT, that can learn to mimic Shakespearean style and structure at the character level. Each character is treated as a token, and sequences of characters are learned in an autoregressive manner.

2 Training process of nanoGPT

2.1 Training loop

The script `train.py` is the main training step of nanoGPT (Karpathy (2022)), and it supports both single-GPU training and Multi-GPU distributed training using Distributed Data Parallel (DDP). It mainly does the following:

- initialises the model's configuration: number of layers, embedding and heads; checks if Distributed Data Parallel (DDP) is active; Enables TensorFloat32 for faster matmuls on Ampere+ GPUs
- `get_batch(split)`: load data set, and returns x: input tokens and y: target tokens (i.e., next token prediction)
- `estimate_loss()`: runs multiple batches through the model and returns average loss. It is Used to log metrics and save checkpoints
- `get_lr(it)`: Implements cosine decay with warmup
- training loop: the loop is run until `max_iters = 600000`. The training loop mainly fetches a new batch of input/output tokens, runs the model forward to compute loss, does backpropagate gradients to update model weights and log progress.

2.2 Training model and architecture

nanoGPT follows the classic Transformer Decoder architecture with token and position embeddings and multiple transformer blocks (i.e., LayerNorm, masked multi-head self-Attention, MLP with GELU). The code `model.py` is the main model block for nanoGPT, and it mainly does the following:

- `CausalSelfAttention`: implements masked multi-head self-attention. It projects the input into Q, K, V (i.e., query, key, value), computes attention, applies dropout and combines heads and projects the result back to the original dimension
- `MLP(nn.Module)`: Implements a standard 2-layer feedforward network with Gaussian Error Linear Unit (GELU) activation:

$$\text{MLP}(x) = \text{Dropout}(W_2(\text{GELU}(W_1 x + b_1)) + b_2)$$

  . It expands the dimension by 4 and projects back by x $\rightarrow$ Linear (expand) $\rightarrow$ GELU $\rightarrow$ Linear (compress) $\rightarrow$ Dropout.
- `forward (self, idx, targets=None)`: implements forward passing.
- `F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1))` : calculates crossentropy which is the standard token-level loss which predicts the next token in sequence.

3 Analysis of the results

To examine the performance of nanoGPT, NVIDIA GPU and Python 3.9 are used.

3.1 Training and validation loss curves

After the experiment, the training and validation loss curves are plotted using the logs recorded during training.

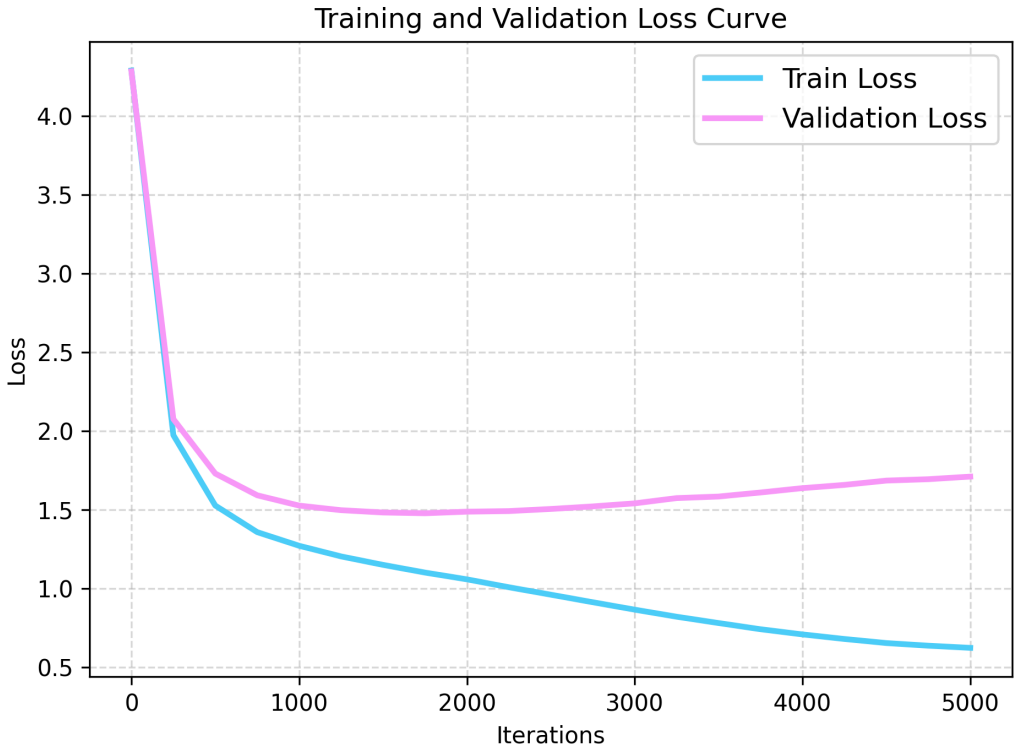

Figure 1: Training and validation losses over time of the experiment

3.2 Analysis of the results

Training loss measures the performance of the model on the training dataset (i.e., the data it sees and learns from during training). It is computed by comparing the model's predictions to the actual next tokens in the training set. The goal of training is to minimize this loss using optimization algorithms.

On the other hand, validation loss measures the performance of the model on unseen data which is not used for training (i.e., the validation dataset). It shows how the model's generalisation ability.

Both losses are calculated using Cross-entropy (Goodfellow et al. (2016))

$$Loss = -\sum_{i=1}^{|V|} y_i \log(p_i)$$

where y_i is the true distribution (i.e., a one-hot vector; 1 for the correct token, 0 for others), and p_i is the predicted probability of token i.

As shown by the curves, the training loss decreases steadily as number of training iterations increases, indicating that the model is learning patterns in the training dataset. The validation loss also decreases initially and tracks the training loss closely, which suggests that the model has good generalsation ability, and is not significantly overfitting.

However, after a number of iterations (i.e., around 2500 iterations), the validation loss slightly increases while training loss continues to drop. This could indicate the model has reached its capacity or begun overfitting.

Overall, the loss curves demonstrate that the model is able to learn meaningful representations from the dataset.

4 Inference and case studies

To examine the model's inference ability, same questions are asked for both Qwen models and nanoGPT. Qwen-2.5-Instruct model is used, with the less creative temperature (i.e., $temperature = 0$ and $temperature = 6$). The following prompts are used for both models:

- "The capital of Singapore is"
- "Who are you?"
- "What is the range of output of tanh?"

4.1 Output analysis

For prompts that are more fact-based, such as "what is the range of the output of tanh", Qwen-2.5-Instruct gives clear analysis and the final answer is correctly stated (i.e., $(-1, 1)$). However, nanoGPT trained on the Shakespeare dataset gives incorrect and meaningless output (i.e., quotation from Shakespeare "GLOUCESTER: All indeed, then I would by my griefs, Where you have no more blazed to thy woes.") (Karpathy (2023)).

For prompts that are open-ended such as "Who are you", Qwen-2.5-Instruct produces the output "What is your purpose in life?" repetitively, which is wrong and has no significance. On the other hand, nanoGPT produces more peotry-styled output (i.e., "EXETER: I am a point of those prince, I cannot ling; Which you will have laid my life to any often."), which is a more relevant response to the prompt (Radford et al. (2018)).

In general, nanoGPT trained on the Shakespeare data set generally generates more Shakespearean and creative output, while responding incorrectly to factual prompts. It is also less coherent and grammatically weaker compared to Qwen-2.5-Instruct.

4.2 Difference analysis

The difference is mainly caused by the model size, training data and fine-tuning process (Radford et al. (2020)).

	nanoGPT	Qwen-2.5-Instruct
Model size and capability	The models has 124 million parameters, which is relatively small	The model has billions of parameters, ensures its high performance
Training data size and type	The model is trained on only Shakespearean plays, which is dialogue-heavy and ancient English style (i.e., 1600s English). For example, factual questions including "range of tanh" never appear in Shakespeare, thus the model is unable to answer unseen math/science queries correctly	The model is trained on diverse internet-scale corpus, including the web, code and instructions. The data set is more general, modern and multilingual
Fine-tuning process	The model is trained from scratch, and there is no instruction tuning involved	The model is fine-tuned with human instructions, for helpful responses

Table 1: nanoGPT v.s. Qwen-2.5-Instruct

References

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Cross-entropy loss derivation used in nanoGPT.

Andrej Karpathy. nanogpt. https://github.com/karpathy/nanoGPT, 2022. The simplest, fastest repository for training/finetuning medium-sized GPTs.

Andrej Karpathy. Tweet comparing model approaches. https://twitter.com/karpathy/status/1645115623017541633, 2023. Key differences in design philosophy.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.