Week 8: Pre-Training with nanoGPT

BEH Chuen Yang

Abstract

This report explores language model pre-training with the help of nanoGPT (Karpathy (2022)), a simple and efficient implementation of the GPT architecture in PyTorch. We use a small dataset (Karpathy (2015a)) of ∼1M tokens (Shakespeare's works) to train a small-scale GPT model, and qualitatively evaluate its text generation capabilities.

1 Motivation

Pre-training is the most essential step in creating a language model. It is from this step that the model gains a baseline, causal understanding of language, in which words depend only on the words that precede them.

While training deep neural networks to just "predict next token" is a very straightforward process, pre-training's apparent simplicity belies the difficulty of getting language models to go from outputting garbage to "understanding" text, much less the challenges involved in translating that "understanding" into fluent text generation.

Hence, it is essential to have hands-on experience with pre-training in order to appreciate some of these challenges in greater detail.

2 nanoGPT

In order to gain this experience quickly, we will use nanoGPT (Karpathy (2022)), a simple and efficient implementation of the GPT architecture in PyTorch.

Not only is the code concise and relatively easy to understand, it also comes with a number of features that make it suitable for experimentation:

- Configuration files for different datasets and model architectures, further customisable via command line arguments.
- Automatic model compilation and mixed precision support for training models on less powerful computers.
- One-file training and evaluation scripts that encapsulate the relevant logic.
- WandB (Biewald (2020)) integration for convenient logging.

Obviously, nanoGPT does not come with many of the most advanced features that are in today's state-of-the-art language models. However, it is just as good for witnessing the pre-training process in action, as the way the pre-training works (once the data is prepared) is very similar to how it is done in larger models.

2.1 Training Code Analysis

While the repository comes with a number of files and scripts, we will focus only on the essential ones for this week's report.

The following table contains brief descriptions of said essential files, and we will elaborate on the high-level components they implement in the following subsections.

File Path	Description
data/shakespeare_char/ prepare.py	Script to prepare the Tiny Shakespeare dataset (Karpathy (2015a)) for training.
configurator.py	Assists in loading configuration files and command line arguments.
config/ train_shakespeare_char.py	Sample configuration file for the Shakespeare dataset.
config/ train_shakespeare _char_modified.py	Modified configuration file for the Shakespeare dataset. Changes were restricted to batch size and model specifications.
model.py	Implementation of the GPT model.
train.py	Main training script that handles data loading, training loop, and logging.
evaluate.py	Evaluation script that generates text from the trained model.

Table 1: Essential files in the nanoGPT (Karpathy (2022)) repository

2.2 Dataset & Data Preparation

Corresponding File: `data/shakespeare_char/prepare.py`

The dataset we will focus on for this report is the Tiny Shakespeare dataset (Karpathy (2015a)), which purportedly contains the complete works of William Shakespeare in a single text file (Karpathy (2015b)). [1]

To obtain tokens for training,

- `data/shakespeare_char/prepare.py` downloads this dataset,
- records metadata about the dataset (like the vocabulary, total token counts, etc.),
- splits the data (90% train, 10% validation) into training and validation sets,
- tokenizes each text block on a character level,
- and saves both the (tokenized) training and validation corpuses to disk.

This creates a training dataset with $\sim$ 1M tokens (since there are $\sim$ 1.1M characters in the text), which is vanishingly small relative to traditional pre-training corpuses (Beh (2025a)). However, the small size also allows us to pre-train a small model in a reasonable amount of time, and to quickly iterate on the training process.

During pre-training, only the train set is used, and the validation set is used to provide diagnostic information about the model's performance on unseen data.

2.3 Model Architecture

Corresponding File: `model.py`

Since GPT is its namesake, it should be unsurprising that nanoGPT closely follows the GPT architecture (Radford et al. (2019)), which is itself a relatively simple decoder-only transformer (Vaswani et al. (2017)). Figure 1 shows the architecture of the model at a glance.

There are, however, two notable departures from the GPT architecture:

- The activation function used in the model is Gaussian Error Linear Unit (GELU) (Hendrycks & Gimpel (2016)), rather than the more common ReLU or LeakyReLU (Xu et al. (2015); Nair & Hinton (2010)).

[1] Despite claiming to contain all of Shakespeare's works, the dataset seems to be missing some works, most notably, *Julius Caesar*.

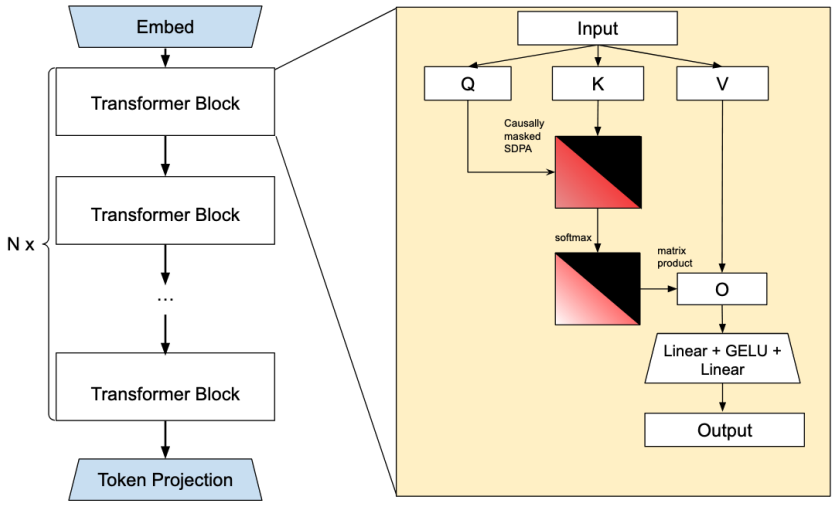

Figure 1: nanoGPT (Karpathy (2022)) model architecture

2.4 Training Loop

Corresponding File: `train.py`

As mentioned in Beh (2025a), language model pre-training is typically formulated as a self-supervised, next-token prediction task, and nanoGPT is no exception. Algorithm **??** shows provides a sketch of the training loop used in nanoGPT.

Algorithm 1 nanoGPT Training Loop (Next-Token Prediction)

Require: Training corpus $\mathcal{T} = [t_1, t_2, \ldots, t_N]$, context window size C, batch size B, Model parameters θ

while not converged **do**
  Sample B indices $\{s_1, \ldots, s_B\}$ uniformly at random such that $\forall b(s_b + C \leq N)$
  for $b = 1$ to B **do**
    $\mathbf{x}^{(b)} \leftarrow [t_{s_b}, t_{s_b+1}, \ldots, t_{s_b+C-1}]$ ▷ Input block
    $\mathbf{y}^{(b)} \leftarrow [t_{s_b+1}, t_{s_b+2}, \ldots, t_{s_b+C}]$ ▷ Target block (shifted by 1)
  end for
  Stack $\{\mathbf{x}^{(b)}\}_{b=1}^{B}$ into input batch $\mathbf{X} \in \mathbb{N}^{B \times C}$
  Stack $\{\mathbf{y}^{(b)}\}_{b=1}^{B}$ into target batch $\mathbf{Y} \in \mathbb{N}^{B \times C}$
  Compute logits $\mathbf{L} = \text{Model}(\mathbf{X}) \in \mathbb{R}^{B \times C \times V}$, where V is vocab size
  Compute loss $\mathcal{L} = \text{CrossEntropy}(\mathbf{L}, \mathbf{Y})$ (averaged over batch and sequence)
  Backpropagate $\mathcal{L}$ and update θ
end while

2.5 Evaluation Loop

Corresponding File: `evaluate.py`

nanoGPT provides autoregressive evaluation of trained models, as is usual for language models. Starting with a user-provided prompt,

- the script generates logits by performing a forward pass through the model
- samples the next token from the **LAST** token's logits,
- appends the sampled token to the prompt,
- and repeats the process until the desired number of tokens is generated.

Hyperparameter	Value
Context Window	512 tokens
Batch Size	32 samples
# Layers	12
# Attention Heads	8
Embedding Dimension	256
Dropout Rate	0.2
Learning Rate	1e-3
Weight Decay	1e-1
Training Steps	5000
Optimizer	AdamW, $\beta_1 = 0.9, \beta_2 = 0.99$ (Loshchilov & Hutter (2019))
Batching Seed	1337

Table 2: Training hyperparameters for the modified Shakespeare model

Since only full blocks of tokens are ever used during training, one would not be faulted for suspecting that the model may not properly learn to generate text in an autoregressive manner, especially if the prompt itself is not a full block of tokens.

However, this is not the case. Further explanation can be found in Appendix A.

3 Training Runs

3.1 Training Configuration

Table 2 summarises the salient hyperparameters used for training. If any more details are desired, they can be found in `config/train_shakespeare_char_modified.py`.

Training was done on 1x NVIDIA RTX A2000 GPU with 8GB of VRAM with BF16 automatic mixed precision. The full run took approximately 15 minutes.

3.2 Loss Curves

Figure 2 shows the training and validation loss curves for the training run.

Figure 2: Training and validation loss curves for the modified Shakespeare model. Note that the losses were only recorded every 250 steps.

Hyperparameter	Value
Sampling Temperature	0.6
# Samples per Prompt	10
Sampling Seed	1337
Prompt 1	"Beware the ides of March." [3]
Prompt 2	"Who are you?" [4]
Prompt 3	"What is the range of output of tanh?" [5]

Table 3: Evaluation hyperparameters for the modified Shakespeare model.

4 Evaluations & Discussion

For evaluations, we allowed the model to autoregressively continue a given prompt for 200 tokens, and repeated this process 10 times for each prompt.

As with the previous section, the salient hyperparameters can be found in Table 3. [2]

We then compared the generated text against texts from the small Qwen models from last week's report (Beh (2025b)), using the same sampling temperature.

For the sake of brevity, we will not include the full text generation results here (although the texts generated by our pre-trained model can be found in `out-shakespeare-char`, and the Qwen models' outputs can be found from Beh (2025b)). Instead, we will summarise the results in the following subsections, referring to the model we pre-trained as "nanoGPT".

4.1 Text Generation Format

As a result of its training set, nanoGPT generates text similar in format to Shakespeare's works (`<NAME>:\ n<LINE 1> \ n<LINE 2> \ n...\ n\ n`), where each line starts with a capital letter, without fail.

However, any similarities appear to be textual, and indeed quite superficial in nature, as the model *frequently repeats the same interlocutors' names* in the generated text, and notably *does not understand the iambic pentameter* structure in which Shakespeare's works are written. Take for example, this contiguous piece of dialogue generated by nanoGPT:

> DUKE OF YORK:
> The letters will the sun sue of England's death. (11 syllables)
> DUKE OF YORK:
> Why looks thou comest to the Duke of Norfolk? (11 syllables)

By contrast, the Qwen models were able to generate text in a wider variety of formats and styles, including multiple choice, and so on. If instruct-tuned, Qwen models are also able to generate text in a more conversational style.

4.2 Text Generation Quality

Surprisingly for a model that is tokenized on a character level, nanoGPT is able to *generate whole words*, with *little to no character-level errors*.

[2] Technically, Karpathy leverages top-k sampling in his script (Karpathy (2022)). However, a manual inspection of the dataset reveals there are only 65 unique characters, much less than his top-k of 200. Hence the top-k is effectively a no-op.

[3] The prompt is derived from the famous line in *Julius Caesar* (Act 1, Scene 2). Since the line was not found in the input corpus, it qualifies as out-of-distribution (OOD) data.

[4] Another OOD prompt to maintain parity with Beh (2025b).

[5] Another OOD prompt to maintain parity with Beh (2025b).

Beyond that, nanoGPT's text is *completely nonsensical* at a phrase and sentence level. nanoGPT also appears to constantly *remix and regurgitate* different variations of quasi-Shakespearean dialogue, using phrases like "I prithee", "sir" and "my lord" frequently, but never in a coherent manner. For example, it might generate sentences like:

> I do not, sir, my lord.

All these indicate that nanoGPT does not understand the meaning of the words it generates, and is simply regurgitating phrases it has seen in the training set.

By contrast, the Qwen models mostly generated cogent and natural sentences, and were on occasion even able to answer questions in a conversational manner (Beh (2025b)).

4.3 Discussion

When analyzing the results side by side with the loss curves, it is quite clear that nanoGPT has somewhat learnt the structure of Shakespeare's works, as evidenced by the steady decrease in training and validation loss over the course of training. Nevertheless, it quite obviously and dismally fails to generate text reminiscent of natural language as it is spoken/written today.

Considering that nanoGPT is a relatively good implementation of the GPT architecture, and that manual inspections have yet to identify any glaring bugs in the model architecture and training loop, it is **likely that the model's failure to generate coherent text stems from**:

- the small size of the training set
- the restricted domain of the training set (i.e. most of Shakespeare's works)
- the small size of the model itself

5 Conclusion

In this report, we have explored the pre-training process of a small language model using nanoGPT (Karpathy (2022)). Through this small yet well-made library, we have seen how the pre-training process works, run a pre-training run on a small dataset, and qualitatively evaluated the model's text generation capabilities. In short, we have effectively witnessed the full pre-training pipeline in action.

While the model was able to learn some of the structure of Shakespeare's works, it ultimately failed to generate coherent text despite an apparently robust implementation and training process. This serves as a reminder of the challenges involved in pre-training language models, and the importance of large datasets and model sizes in achieving good performance.

References

Chuen Yang Beh. Week 5: Introducing large language models to a general audience, 2025a. URL ../wk5/wk5.pdf.

Chuen Yang Beh. Week 7: Comparing base models and instruct-tuned models, 2025b. URL ../wk7/wk.pdf.

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2016. URL https://arxiv.org/abs/1606.08415.

Andrej Karpathy. Tiny shakespeare, 2015a. URL https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt.

Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks, 2015b. URL https://karpathy.github.io/2015/05/21/rnn-effectiveness/#Shakespeare.

Andrej Karpathy. nanogpt, 2022. URL https://github.com/karpathy/nanoGPT.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Vinod Nair and Geoffrey Hinton. Rectified linear units improve restricted boltzmann machines, 2010. URL https://www.cs.toronto.edu/~fritz/absps/reluICML.pdf.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2019. URL https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL https://arxiv.org/abs/1706.03762.

Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network, 2015. URL https://arxiv.org/abs/1505.00853.

A Appendix: Why Full Block Training Lends Itself Well to Autoregressive Generation

Examining the source code, we find that the `CasualSelfAttention` module is designed to project embedding tokens in parallel (line 56).

Moreover, due to the theorem below, the casual self-attention matrix is invariant to the suffix of the input sequence:

Theorem A.1 (Causal Attention Prefix Invariance)**.** Let $\mathbf{X} = (\mathbf{x}_1|\mathbf{x}_2|\dots|\mathbf{x}_n) \in \mathbb{R}^{d\times n}$ be a sequence of token embeddings, and let $\mathbf{W}_Q, \mathbf{W}_K \in \mathbb{R}^{d_{att}\times d}$ be the query and key projection matrices for a self-attention head.

Define the masked attention scores as

$$\mathbf{B}(\mathbf{X})_{ij} = \frac{\exp(\mathbf{A}(\mathbf{X})_{ij})}{\sum_{k=1}^{n}\exp(\mathbf{A}(\mathbf{X})_{ik})} \tag{1}$$

where $\mathbf{A}(\mathbf{X})_{ij} = \begin{cases} \frac{(\mathbf{W}_Q\mathbf{x}_i)^T(\mathbf{W}_K\mathbf{x}_j)}{\sqrt{d_{att}}} & \text{if } j \le i \\ -\infty & \text{if } j > i \end{cases}$

Then for any prefix length $m < n$, $\mathbf{B}(\mathbf{X})_{1:m,1:m} = \mathbf{B}(\mathbf{X}_{1:m})$.

Proof. Since $i \le m$, all positions $k \le i$ are within the prefix $\mathbf{X}_{1:m}$. Therefore:

$$\mathbf{A}(\mathbf{X})_{ij} = \mathbf{A}(\mathbf{X}_{1:m})_{ij} \quad \text{for all } j \le i \le m \tag{2}$$

Since we conduct softmax over the last dimension we only care about the case when $j > m$ but $i \le m$. Here, naturally $j > i$. Hence, we have

$$\begin{aligned}\mathbf{B}(\mathbf{X})_{ij} &= \frac{\exp(\mathbf{A}(\mathbf{X})_{ij})}{\sum_{k=1}^{n}\exp(\mathbf{A}(\mathbf{X})_{ik})} \\ &= \frac{\exp(\mathbf{A}(\mathbf{X}_{1:m})_{ij})}{\sum_{k=1}^{m}\exp(\mathbf{A}(\mathbf{X}_{1:m})_{ik})} \\ &= \mathbf{B}(\mathbf{X}_{1:m})_{ij}\end{aligned}$$

□

As such, it does not matter how much of the model's context window is used to generate the next token. If I had any $m < n$ for a sequence length n, the top left $m \times m$ block of the causal attention matrix would be the same as the causal attention matrix calculated using only the first m tokens of the sequence.

To reiterate, nanoGPT will be able to autoregressively generate text just fine, even if the prompt is not a full block of tokens.