Training a "Small" Language Model

Luka Meladze
National University of Singapore
Project: **LLM Post-training with Reinforcement Learning**

Abstract

This report details the process of training a character-level Generative Pre-trained Transformer (GPT) model from scratch using the nanoGPT repository. The objective is to understand the fundamental mechanics of training a Large Language Model (LLM) by working with a smaller model and a limited dataset, i.e. the complete works of William Shakespeare. The report will cover a summary of the training code (train.py) to demonstrate an understanding of the underlying processes, visualization of the model's learning progress through training and validation loss curves, and a comparative analysis of our model's responses against other pre-trained models (e.g. Qwen).

1 Observations and Comparative Analysis of models

1.1 The implementation details of the model training

The train.py script in the nanoGPT repository is the core engine for training the language model. It orchestrates the entire pre-training process. The code assumes that the dataset has been preprocessed into a binary file containing a sequence of token IDs. For our Shakespeare dataset, a separate script (data/shakespeare/prepare.py) handles the initial text loading, character vocabulary creation, and tokenization (mapping each character to a unique integer). The get batch function then is responsible for creating mini-batches of data for both training and validation. For each batch, it randomly selects starting points from the dataset and extracts a sequence of a predefined block_size. The input to the model (x) is this sequence, and the target (y) is the same sequence shifted by one position to the right. This setup trains the model to predict the next character in the sequence. An optimizer is created to update the model's weights during training. The nanoGPT repository uses the AdamW optimizer, which is a variant of the Adam optimizer with improved weight decay regularization. The core of the script is the main training loop, which iterates for a specified number of steps (max_iters). In each iteration:

- Periodically, the model is switched to evaluation mode (model.eval()) and it then estimates the loss on a subset of the training and validation data by averaging the loss over several batches. This provides an unbiased estimate of the model's performance on unseen data and helps in detecting overfitting.
- Forward Pass: A batch of training data is fetched using get_batch. The model performs a forward pass, taking the input sequences (x) and producing logits, which are raw predictions for the next token in the sequence. The loss is then calculated, typically using cross-entropy, which measures the difference between the model's predictions and the actual target tokens (y).
- The gradients of the loss with respect to the model's parameters are calculated using loss.backward(). This is the backpropagation step. Then the optimizer's step() method is called to update the model's weights based on the calculated gradients and the chosen learning rate.

1.2 Training and Validation Curves

The model was trained on the Shakespeare dataset for 5000 iterations. The training and validation losses were logged at regular intervals. (Configuration: max_iters = 5000 and eval_interval = 250 $- >$ 21 dots on the plot below).The training loss (blue line) consistently decreases over the training iterations. This indicates that the model is effectively learning the patterns and structure of the Shakespearean text from the training data. The curve starts steep and gradually flattens, which is typical as the model converges. The validation loss (orange line) also decreases, closely following the training loss. This is a very positive sign, as it suggests that the model is generalizing well to unseen data and is not significantly overfitting. The model is not only memorizing the training data but learning the underlying language patterns. The following plots shows the curves.

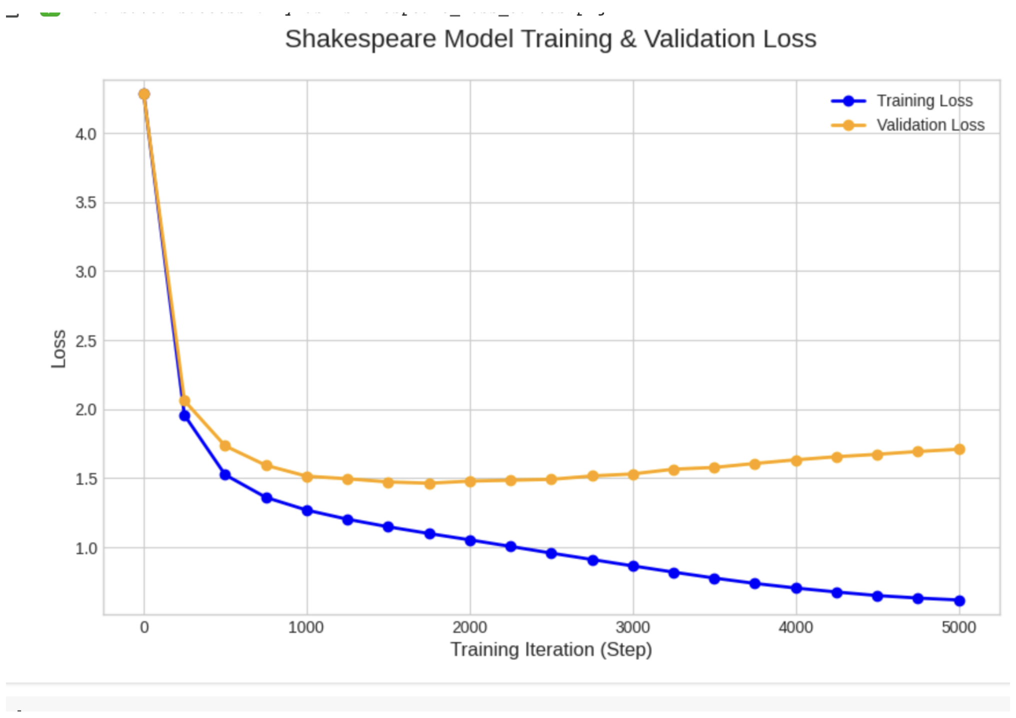

Figure 1:

1.3 Generating Outputs by the Trained Base Model

To generate text from our trained model, we use sample.py. The start variable in this script was modified to the prompts: **1. ”Who are you?” 2. ”The capital of Singapore is”.**

- Prompt: ”The capital of Singapore is”:
  - When run with a generic start or a simple prompt, the model produces text that is stylistically similar to Shakespearean text. However, the model has **no factual knowledge** of other concepts like geography (e.g. capitals). Its entire dataset is from Shakespeare (unlike the Qwen Base and Instruct Models trained before). It completes the prompt not with a fact, but with words and phrases that are statistically likely to follow in a Shakespearean context, resulting in nonsensical, poetic-sounding, but factually incorrect statements. E.g. ”The capital of Singapore is the sea” or ”The capital of Singapore is no less true”.
- **Prompt: ”Who are you?**:

- Model's response (just next token completion):
  ”CORIOLANUS: Hear me, to see me, he must take me again.
  AUFIDIUS: Speak no word a boy.
- **Analysis:** The model correctly interprets this as a prompt that would appear in a play or dialogue. Its response is to generate a new character name (”CORIOLANUS:”) followed by a line of dialogue. This demonstrates it has learned the conversational patterns within its training data.

Analysis & Comparison with General-Purpose LLMs:

1. The outputs of this specialized Shakespearean model are very different from the general-purpose LLMs (like Qwen 0.5B and 0.6B parameters) used previously. The nanoGPT model was only introduced to Shakespearean English. It has no concept of modern countries, technology, or general facts outside of its limited dataset. General-purpose LLMs are trained on terabytes of diverse text from the internet, books, and articles, giving them a comprehensive model of the world.
2. Model Size and Architecture: Our model is tiny, with about 10.65 million parameters. Models like Qwen have billions of parameters. This larger capacity allows them to store vastly more information, understand more complex grammatical and semantic patterns, and reason more effectively.
3. Training Objective & Tokenization: Our model was trained with a simple objective: predict the next character. In contrast, other models were predicting next token (bunch of characters on each iteration) and some of them were already **instruct-tuned.**