A Simple Implementation of a GPT-2-Like Model for Text Generation on a CPU

Julien Okumu
codewithjulien@gmail.com

Date: March 7, 2025

Institution: Independent Research

Abstract

This paper describes a basic version of the GPT-2 language model, built from scratch in Python using PyTorch, and trained on a small CPU (Intel i5-6200U with 8GB RAM). I aimed to understand how GPT-2 works by coding its key parts—like attention and transformer blocks—and testing it on a tiny dataset (Tiny Shakespeare). The model, with 124 million parameters, was adjusted to fit limited hardware by shortening its context length and using small batches. After one training round, it generated simple but repetitive text. This shows that even a small setup can run a GPT-like model, though it needs more time and data to get smarter. Link with the github code: https://github.com/julienokumu/GPT-2small

---

1. Introduction

Language models like GPT-2, created by OpenAI, can generate human-like text by predicting the next word in a sequence. They're powerful but usually need big computers (GPUs) and lots of data. I wanted to see if we could build a smaller version of GPT-2 and train it on a regular laptop CPU. Our goals were:

1. To code a GPT-2-like model from scratch.
2. To make it work on a basic CPU (Intel i5-6200U, 8GB RAM).
3. To train it on a small text dataset and generate some text.

I based our work on the GPT-2 design but simplified it for our hardware. This paper explains how I did it, what I found, and what it means for learning about AI.

---

2. Methodology

2.1 Model Design

I built a model with these parts:

- **Token Embedding**: Turns words into number vectors (768 numbers each).
- **Positional Embedding**: Adds info about word order (up to 64 words at a time).
- **Transformer Blocks**: 12 layers that process the text using:
    - **Multi-Head Attention**: Lets the model focus on related words (12 "heads" per layer).
    - **Feed-Forward Network**: Adds extra processing (expands to 3072 numbers, then back to 768).
    - **Layer Normalization**: Keeps numbers stable during training.
- **Output Layer**: Predicts the next word from 50,257 possible tokens.

The model has 124 million parameters (numbers it learns), matching the smallest GPT-2 version.

2.2 Adjustments for CPU

My laptop couldn't handle the full GPT-2, so we:

- Set the **context length** to 64 (instead of 1024) to use less memory.
- Used a **batch size** of 1 (one example at a time) to fit in 8GB RAM.
- Made a simple **tokenizer** that turns text into 1000 unique word IDs (not the full 50,257).

2.3 Dataset

I used **Tiny Shakespeare**, a 1MB text file of Shakespeare's plays. It's small but has enough words to test the model.

2.4 Training

I trained the model to predict the next word in the text:

- **Optimizer**: AdamW (a common choice for adjusting the model).

- **Learning Rate**: 0.001 (how fast it learns).
- **Loss Function**: Cross-entropy (measures prediction errors).
- Ran for **1 epoch** (one full pass through the data), about 1000 batches.

Training took ~2 hours on the CPU, using ~2-3GB of RAM.

2.5 Generation

After training, I tested it by starting with “To be or not to be” and asking it to add 20 more words.

---

3. Results

3.1 Training Performance

- **Initial Loss**: ~7 (high, meaning bad predictions at first).
- **Final Average Loss**: ~6.5 (a small drop after 1 epoch).
- The loss shows the model started learning but needs more time to get good.

3.2 Generated Text

Input: “To be or not to be”

Output: “if discontented minister poor. yourselves discontented”

- The text repeats words like “discontented” a lot.
- It’s not meaningful yet, but it uses words from the training data.

3.3 Hardware Usage

- **Time**: ~2 hours for 1 epoch (~5-10 seconds per batch).
- **Memory**: ~2-3GB RAM, fitting well within 8GB.
- No crashes, showing the adjustments worked.

---

4. Discussion

My model successfully ran on a basic CPU, proving you don't need a fancy GPU to experiment with GPT-2-like models. The generated text wasn't great because:

- I only trained for 1 epoch (~2 hours). Real GPT-2 trains for weeks on huge datasets.
- My tokenizer was simple (1000 words vs. 50,257), limiting what the model could say.
- The small context (64 words) meant it couldn't see big patterns.

Still, the loss dropped slightly, showing it learned something. With more epochs (e.g., 5, ~10 hours) or a bigger dataset, it could improve.

---

5. Conclusion

I built and trained a mini GPT-2 on a CPU, showing it's possible with limited hardware. The model, with 124 million parameters, learned a little from Tiny Shakespeare and generated basic text. This experiment teaches us how GPT-2's pieces—like attention and transformers—fit together. For better results, I'd need more training time, a proper tokenizer, and maybe a smaller model (e.g., 6 layers). Future work could test this on bigger texts or optimize it further for CPUs.