Member-only story

Learning Transformers Code First: Part 1 — The Setup

A 4 Part Exploration of Transformers Using nanoGPT as a Starting Point

Lily Hughes-Robinson · Following

Published in Towards Data Science · 8 min read · Jul 7

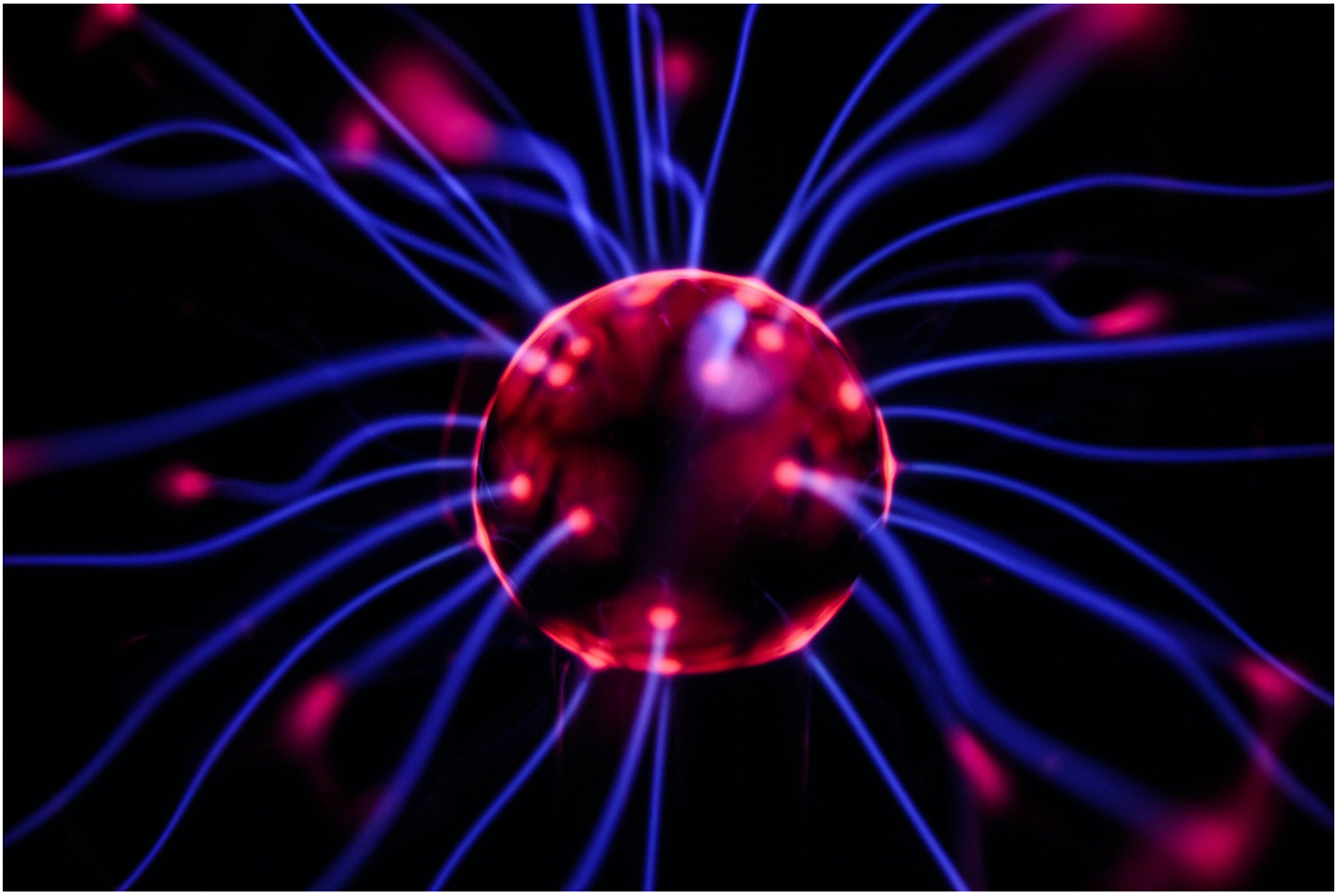


Photo by Josh Riemer on Unsplash

I don't know about you, but sometime looking at code is easier than reading papers. When I was working on AdventureGPT, I started by reading the source code to BabyAGI, an implementation of the ReAct paper in around 600 lines of python.

Recently, I became aware of a recent paper called TinyStories through episode 33 of the excellent Cognitive Revolution Podcast. TinyStories attempts to show that models trained on millions (not billions) of parameters can be effective with high-enough quality data. In the case of the Microsoft researchers in the paper, they utilized synthetic data generated from GPT-3.5 and GPT-4 that would have cost around $10k retail to generate. The dataset and models are available from the author's HuggingFace repo.

I was captivated to hear that a model could be trained on 30M and fewer parameters. For reference, I am running all my model training and inference on a Lenovo Legion 5 laptop with a GTX 1660 Ti. Even just for inference, most models with over 3B parameters are too large to run on my machine. I know there are cloud compute resources available for a price, but I am learning all this in my spare time and can really only afford the modest OpenAI bill I rack up via API calls. Therefore, the idea that there were models I could train on my modest hardware instantly lit me up.

I started reading the TinyStories paper and soon realized that they utilized the now defunct GPT Neo model in there model training. I started digging into the code to see if I could understand it and I realized I needed something even smaller to start from. For context, I am mainly a backend software engineer with just enough machine learning experience to not get completely lost when hearing people talk about neural nets. I am nowhere near a proper ML engineer and this led me to type “gpt from scratch” into my preferred search engine to find a gentler introduction. I found the video below and everything shifted.

This was what I was looking for. In addition to the basic repo linked in the video, there is a polished version called nanoGPT which is still under active development. What is more, **the training code and model code are around 300 lines of python each**. To me, that was even more exciting than the video. I closed the video and started pouring over the source code. nanoGPT utilizes PyTorch, which I've never used before. It also features just enough math to make and machine learning jargon to make the neophyte in me anxious. This was going to be something of a bigger undertaking than I anticipated.

One of the best ways to understand something is to write about. Therefore, I plan on picking apart the code in the nanoGPT repo, reading the famous "Attention is All You Need" paper, and learning transformers in a bottoms-up, hands on way. Whatever I learn along the way I hope to write about in this series. If you want to follow along, clone the nanoGPT repo to your machine (the model can even be trained on CPU, so no hardware excuses) and follow along.

The first thing I did after cloning the repo was follow the README's instructions for training the simplest model, the character-level generation model using the tiny_shakespeare dataset. There is a script to prepare the dataset for training, a script to do the actual training, and a sampling script to output generated text. With a few terminal commands and an hour+ of training, I had simple model to output Shakespearean-sounding text.

Following instructions is all well and good, but I don't really understand something until I modify it to work for my own use case. My goal here was to train a similar character-level model using the TinyStories dataset. This required creating my own data preparation script to get the dataset ready for training. Let's dig into that deeper.

The nanoGPT has two types of data preparation scripts: one for GPT-2 style models and one for character-level models. I grabbed some of the code from the GPT-2 models for downloading from HuggingFace repositories and took everything else from the tiny_shakespeare character-level script. One important point here, tiny_shakespeare is just over 1MB and contains only 40k lines of Shakespeare. TinyStories is over 3GB compressed and contains 39.7M stories. The methods for tokenizing and slicing tiny_shakespeare were not directly transferable, at least not with the 32GB of RAM my laptop has. I crashed my machine several times trying pythonic, easy-to-read methods of preparing TinyStories. The final script uses a few tricks I will detail below.

First off, my preferred solution for processing lists of data is list comprehension, a syntax for generating new lists from existing lists with modifications. The issue with list comprehension in this case is that that 3GB of compressed text becomes closer to 10GB in RAM. Now, list comprehension requires multiple copies of the list in RAM. Not an issue for small data, but unworkable for TinyStories.

The outputs of the data preparation scripts is a compressed NumPy array of character level encoding for the train and validation data plus a metadata pickle which includes the full list of unique characters and the encoding/decoding maps to convert these characters to numbers. Using this as reference, we don't need anything other than the final encoded array of numbers once the unique characters are found and mapped to numbers. The best way to do this memory efficiently is to iterate through a the data with a simple for-loop while building these outputs piece-mils. To do this, you initialize an initial variable before the loop which then gets updated each interaction. This prevents multiple versions of the dataset from being held in RAM and only outputs what we need. The final vocab generation code is below:

```
chars_dataset = set([])
len_dataset = 0

# get all the unique characters that occur in this text as well as total length
desc = "Enumerate characters in training set"
for story in tqdm(dataset['train']['text'], desc):
    chars = list(set(story))

    for char in chars:
        chars_dataset.add(char)

    len_dataset += len(story)
```

That said, an array of 30.7M stories (over 4B characters) encoded as numbers still takes up a non-trivial amount of RAM because Python is storing the ints dynamically. Enter NumPy, which has a much more efficient array storage where you can specify the exact size of the ints. In addition to the efficient storage, NumPy also has a memory efficient array concatenation which can be used to build the final encoded array iteratively rather than all at once.

My finishing touch on the script was to add a progress bar using tqdm for each step and I was finally ready to run the script. So, I ran it overnight and came back in the morning. When I came back, the script was still running, with over 100 estimated hours of compute time remaining.

This is when it really hit me: 30.7M stories is small for a language model, but is very much not a toy dataset to be processed on a single thread. It was time to bring in the big guns: parallelization. Parallelism brings in a lot of complexity and overhead, but the performance gains was worth the trade off. Luckily, there are a number of ways to parallelize Python code. Many of these solutions require major rewrites to a serially executed script or complicated abstractions. With a little digging, I found something that allowed me to keep most of my script the same but still run multiple processes to take advantage of all of my threads.

Ray is a library for easily parallelizing methods in Python and can easily be run locally or as a cluster. It handles running tasks in a queue and spinning up worker processes to eat away at that queue. There is an excellent guide to ray below if this has whet your appetite.

Modern Parallel and Distributed Python: A Quick Tutorial on Ray

Ray is an open source project for parallel and distributed Python.

towardsdatascience.com

When it came to choosing what to parallelize, the encode function seemed like a good candidate. It has clear inputs and outputs, no side effects on those inputs, and was easily one of the largest portions of the compute time. Adapting the existing code to work with ray couldn't have been easier: the

function becomes accessible to ray via a decorator, the functional call changes slightly to add a remote attribute, and there is a function to kick off executing all the data. Below is an example of how it looked in my code base initially:

```
import ray

ray.init()

…

# given all the unique characters within a dataset,
# create a unique mapping of characters to ints
stoi = { ch:i for i,ch in enumerate(chars_dataset) }

@ray.remote
def encode(s):
    return [stoi[c] for c in s]

…

encoded_stories = []
for story in dataset['train']['text']:
    encoded_stories.append(encode.remote(story))

ray.get(encoded_stories)

…
```

Armed with all my CPU's power, I forged ahead only to immediately crash my laptop. With the locally distributed call stack used by ray, the entire dataset was in memory several times over. Simply enqueuing the entire dataset caused an out-of-memory error. Annoyed, I used this as an excuse to buy more RAM (64GB here we come!), but continued to tweak the code while the RAM shipped.

The next logical place was to batch the requests being handled by ray into something that could fit inside a reasonable amount of memory. Adding batching logic was reasonably straightforward and is present in the final codebase I will link to at the end of the article. What actually became interesting was experimenting with the batch size. Initially, I chose a random batch size (5000) and it started out well, but it became obvious to me that a fair amount of time was being spent on the single-threaded code during each batch.

Essentially, watching my preferred system monitor, I saw a single core pinned for minutes before finally all my laptop's cores lit up for a few seconds before going back to a only a single core being utilized. This lead my to play with the batch size a bit, hoping to feed the starving CPU cores faster and keep them engaged longer. Lowering the batch size didn't help because there was so much synchronous code in each batch used to slice and prepare a batch from the full dataset. That code couldn't be parallelized, so it meant that each batch had a large startup cost time wise generating the chunk. This led me to try the opposite, increasing the chunk size to keep the cores more engaged for longer. This worked, as chunk generation took the same amount of time regardless of chunk size, but each chunk processed more data. Combining this with moving my encoding post-processing into ray functions, I was able to chew through 30% of the training dataset in just a few hours, all on a single laptop.

Finally, after a few more hours, I had a fully prepared, custom dataset to feed to the character-level model. I was pleased that I didn't have to resort to utilizing expensive cloud compute to process the training set, which was my next move if the RAM increase didn't work. What's more, I learned intimately what it meant to create/process a dataset for a character-level model.

In the next article in this series, I will be examining the actual model code, explaining as best I can and linking to copious external resources to provide additional information where my knowledge falls short. Once the article is written, I will go back and provide a link here. In the meantime, I've linked the final version of my dataset preparation script below so you can follow along and see what it takes to process a somewhat large dataset on a limited compute platform.

nanoGPT/data/tinystories_char/prepare.py at master · oaguy1/nanoGPT

The simplest, fastest repository for training/finetuning medium-sized GPTs. - nanoGPT/data/tinystories_char/prepare.py...

github.com

Also, part two of the series is available! Click below to read!

Learning Transformers Code First Part 2—GPT Up Close and Personal

Digging into Generative Pre-Trained Transformers via nanoGPT

towardsdatascience.com

Gpt Transformers Machine Learning AI Data Science

More from the list: "NLP"

Curated by Himanshu Birla

Jon Gi... in Towards Data ...

Characteristics of Word Embeddings

· 11 min read · Sep 4, 2021

Jon Gi... in Towards Data ...

The Word2vec Hyperparameters

· 6 min read · Sep 3, 2021

Jon Gi... in

The Word2ve

· 15 min rea

View list

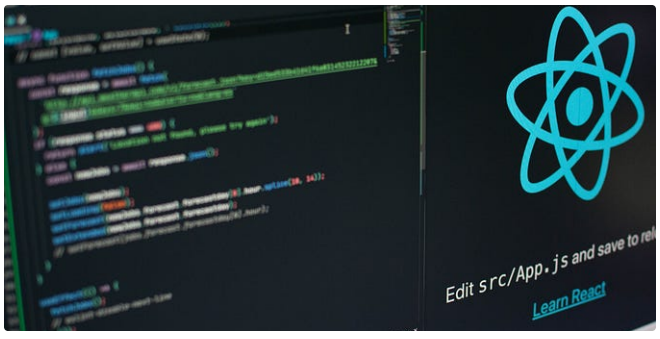
Written by Lily Hughes-Robinson

242 Followers · Writer for Towards Data Science

Following

Lily is a software engineer at a large financial institution. When she isn't building things with AI, she is hanging out with her wife and toddler in Brooklyn.

More from Lily Hughes-Robinson and Towards Data Science



Lily Hughes-Robinson in Better Programming

Antonis Makropoulos in Towards Data Science

Your Personal Copy Editor: Build a LLM-backed App Using...

Turning my idea into a product

✨ · 7 min read · Jul 19

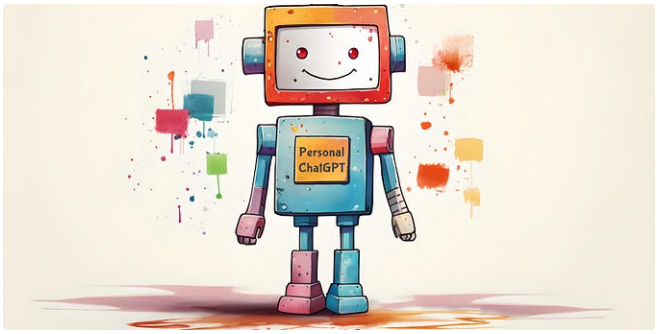
357 6

How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and...

10 min read · Sep 17

549 11



Robert A. Gonsalves in Towards Data Science

Your Own Personal ChatGPT

How you can fine-tune OpenAI's GPT-3.5 Turbo model to perform new tasks using you...

✨ · 15 min read · Sep 8

595 7

Lily Hughes-Robinson in Towards Data Science

Learning Transformers Code First Part 2—GPT Up Close and...

Digging into Generative Pre-Trained Transformers via nanoGPT

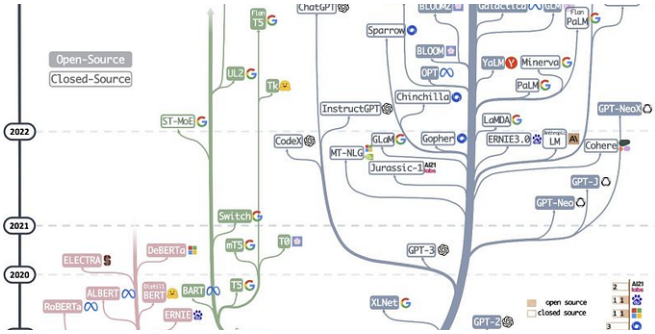
✨ · 13 min read · Jul 13

93 4

See all from Lily Hughes-Robinson

See all from Towards Data Science

Recommended from Medium



Haifeng Li

A Tutorial on LLM

Generative artificial intelligence (GenAI), especially ChatGPT, captures everyone's...

15 min read · Sep 14

372

Jim the AI Whisperer in The Generator

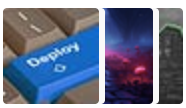
The 19 tell-tale signs an article was written by AI

Avoid these AI cliches (or use them to detect AI writing in the wild)

· 16 min read · 5 days ago

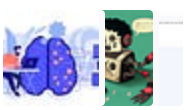
2.3K 55

Lists



Predictive Modeling w/ Python

20 stories · 452 saves

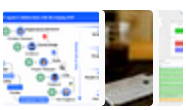


The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves

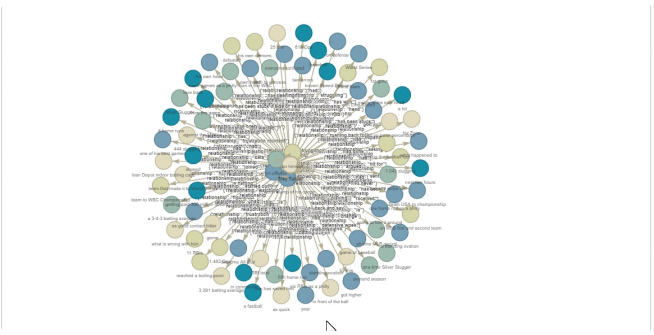
Practical Guides to Machine Learning

10 stories · 519 saves



Natural Language Processing

669 stories · 283 saves



Han HELOIR, Ph.D. in Artificial Corner

MongoDB and Langchain Magic: Your Beginner's Guide to Setting...

Introduction:

✦ · 7 min read · Sep 12

1.4K 12

Wenqi Glantz in Better Programming

7 Query Strategies for Navigating Knowledge Graphs With...

Exploring NebulaGraph RAG Pipeline with the Philadelphia Phillies

✦ · 17 min read · 4 days ago

501 4

Ryan Nguyen in Towards AI

So, You Want To Improve Your RAG Pipeline

Ways to go from prototype to production with LlamaIndex

✦ · 9 min read · Sep 27

176 2

Nikhil Vemu in Mac O'Clock

Change These 12 iOS 17 Settings Right Now For a Superior...

iOS 17's got hell-a-lot features you can't discover thyself

✦ · 9 min read · Sep 26

2.4K 29

See more recommendations