CHEG 667-013 – CHEMICAL ENGINEERING WITH COMPUTERS

Department of Chemical and Biomolecular Engineering
University of Delaware

Spring 2025

LARGE LANGUAGE MODELS PART I

Key idea:

We will study how Large Language Models (LLMs) work and discuss some of their uses

Key goals:

- Locally run a small transformer-based language model
- Train the model from scratch
- Test model parameters and their effects on text generation
- Develop a better understanding of how these technologies work

---

Large Language Models (LLMs) have rapidly integrated into our daily lives and work. Our goal this week (and the next?) is to learn a bit about how LLMs work. As you have probably become well aware of throughout your studies, engineers often don't take technical solutions for granted. We generally like to "look under the hood" and see how a system, process, or tool does its job – and whether it is giving us accurate and useful solutions. The material we will cover is largely inspired by the rapid adoption of LLMs to help us solve difficult but adjacent problems in our engineering work.

We will use a code repository published by Andrej Karpathy called nanoGPT. GPT stands for Generative Pre-trained Transformer. A transformer is a neural network architecture designed to handle sequences of data using self-attention, which allows it to weigh the importance of different words in a context. The neural network's weights and biases are created beforehand using training and validation datasets (these constitute the training and fine-tuning steps, which often require considerable computational effort, depending on the model size). Generative refers to a model's ability to create new content, rather than just analyzing or classifying existing data. When we generate text, we are running an *inference* on the model. Inference requires much less computational effort.

NanoGPT can replicate the function of the GPT-2 model. Building the model from scratch to a point to that level of performance (which is far lower than the current models) would still require a significant investment in computational effort – Karpathy reports using eight NVIDIA A100 GPUs for four days on the task – or 768 GPU hours. In this introduction, our aspirations will be far lower. We should be able to do simpler work with only a CPU.

Why does AI tend to use GPUs? The math underlying the transformer architecture is largely based on matrix calculations. Originally, GPUs were developed to quickly calculate matrix transformations associated with high-performance graphics applications. (It's all linear algebra!) These processors have since been adapted into general-purpose engines for the parallel computations used in modern AI algorithms.

1 Preliminaries

Dust off those command line skills! There will be no GUI where we're going. I recommend making a new directory (under WSL if you're using a Windows machine) and installing `conda` with a new, clean environment. Call it `llm`. Activate the environment with `conda activate llm`. You will need to install packages like `python`, of course, and `numpy` and `pytorch`.

2 Getting the code

Karpathy's code is at

- https://github.com/karpathy/nanoGPT

Download the code using `git`. `git` is a popular distributed versioning system that lets multiple people track and manage changes to source code or other files over time. (You might have to install it first.) An alternative is to download a `zip` file from the Github page. (Look for the green `Code` button on the site. Clicking this, you will see `Download ZIP` in the dropdown menu.)

```
$ git clone https://github.com/karpathy/nanoGPT
```

You should now have a nanoGPT directory:

```
$ ls
nanoGPT/
```

3 A quick tour

List the directory contents of `./nanoGPT`. You should see something like:

```
$ ls -l nanoGPT
total 696
-rw-r--r--  1 furst  staff    1072 Apr 17 12:44 LICENSE
-rw-r--r--  1 furst  staff   13576 Apr 17 12:44 README.md
drwxr-xr-x  4 furst  staff     128 Apr 17 12:44 assets/
-rw-r--r--  1 furst  staff    4815 Apr 17 12:44 bench.py
drwxr-xr-x  9 furst  staff     288 Apr 17 12:44 config/
-rw-r--r--  1 furst  staff    1758 Apr 17 12:44 configurator.py
drwxr-xr-x  5 furst  staff     160 Apr 17 12:44 data/
-rw-r--r--  1 furst  staff   16345 Apr 17 12:44 model.py
-rw-r--r--  1 furst  staff    3942 Apr 17 12:44 sample.py
-rw-r--r--  1 furst  staff  268519 Apr 17 12:44 scaling_laws.ipynb
-rw-r--r--  1 furst  staff   14857 Apr 17 12:44 train.py
-rw-r--r--  1 furst  staff   14579 Apr 17 12:44 transformer_sizing.ipynb
```

Here's a quick run-down on some of the files and directories:

- `/data` – contains three datasets for training the nanoGPT. Two of these (`/data/openwebtext` and `/data/shakespeare`) encode the training datasets into the GPT-2 tokens (base pair encoding, or bpe). We will focus on the third, `/data/shakespeare_char`, which will generate

a character-level tokenization of the text. (Tokenization is the process of breaking down text into smaller units that a machine learning model can process.)

- `/config` – scripts to train or finetune the model, depending on the tokenization method used.
- `train.py` – a Python script that trains the model. This will build the weights and biases of the transformer.
- `sample` – a Python script that runs inference on the model. This is a "prompt" script that will cause the model to begin generating text.
- `model.py` – a Python script with all of the mathematics of the transformer AI! That's it! There's just 330 lines of code! (*Hint:* type `wc -l model.py`)

4 Preparing the training dataset

These commands will download the training dataset and tokenize it. Type:

```
$ python data/shakespeare_char/prepare.py
```

after a few minutes, you should see:

```
length of dataset in characters: 1,115,394
all the unique characters:
 !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
vocab size: 65
train has 1,003,854 tokens
val has 111,540 tokens
```

Now we see the files in `data/shakespeare_char`,

```
$ ls -l
total 6576
-rw-r--r--  1 furst  staff  1115394 Apr 17 14:54 input.txt
-rw-r--r--  1 furst  staff      703 Apr 17 14:54 meta.pkl
-rw-r--r--  1 furst  staff     2344 Apr 17 12:44 prepare.py
-rw-r--r--  1 furst  staff      209 Apr 17 12:44 readme.md
-rw-r--r--  1 furst  staff  2007708 Apr 17 14:54 train.bin
-rw-r--r--  1 furst  staff   223080 Apr 17 14:54 val.bin
```

The script downloads `input.txt` and tokenizes the text. It splits the tokenized text into two binary files: `train.bin` and `val.bin`. These are the training and validation datasets. `meta.pkg` contains information about the model size and parameters.

Exercise 1: The `prepare.py` script downloads and tokenizes a version of *Tiny Shakespeaere*. How big is the text file? Use the command `wc` to find the number of lines, words, and characters. Examine the text with the command `less`.

5 Training the model

Most of us will be running this code on a CPU, not a GPU. Moreover, as an interpreted language, Python is pretty slow, too. We will need to reduce the size of the model by setting a few of the parameters. After this, we will train the model on our training text.

5.1 Model parameters

The default parameters are in the configuration file `nanoGPT/config/train_shakespeare_char.py`. Examine this file:

```
$ less train_shakespeare_char.py
```

Note the following parameters:

- `n_head` – the number of parallel attention heads in each transformer. Transformer blocks use multiple attention heads to capture diverse patterns in the text.
- `n_layer` – the number of (hidden) layers or transformer blocks stacked in the model.
- `n_embd` – in the model, each token is mapped to a vector of this size. If `n_embd` is too small, the model can't capture complex patterns. If it is too large, the model overfits or wastes capacity and it is more expensive to train. Memory and compute cost may grow approximately quadratically with this dimensionality.
- `block_size` – This is the *context window* or *context length* – how many characters (tokens) the model can "look back" to predict the next one. Larger context allows richer understanding, but increases memory and compute.
- `dropout` – a regularization technique that randomly disables a fraction of neurons during training to prevent overfitting. Values between 0.1–0.5 are common. Note that we set it to zero when we use a small model on the CPU.

A related parameter that is set by the tokenization is the *vocabulary size*. Remember, we're using a character-level tokenization with a vocabulary of 65 tokens.

Exercise 2: What are the default values for the parameters `eval_iters`, `log_interval`, `block_size`, `batch_size`, `n_layer`, `n_head`, `n_embd`, `max_iters`, `lr_decay_iters` and `dropout`?

5.2 A training run

Since we are likely using a CPU, we have to pare down the model from its default values.[1] These can be passed on the command line, or the configuration can be edited. Here are the parameters to start with:

```
$ python train.py config/train_shakespeare_char.py --device=cpu --compile=
   False --eval_iters=20 --log_interval=1 --block_size=64 --batch_size=12
   --n_layer=4 --n_head=4 --n_embd=128 --max_iters=2000 --lr_decay_iters
   =2000 --dropout=0.0
```

[1] Try running `python train.py config/train_shakespeare_char.py --device=cpu --compile=False` to see how slow it is using the default values. Use control-c to quit after a few minutes.

You should see the script output its parameters and other information, then something like this:

```
step 0: train loss 4.1676, val loss 4.1649
iter 0: loss 4.1828, time 2654.72ms, mfu -100.00%
iter 1: loss 4.1373, time 124.87ms, mfu -100.00%
iter 2: loss 4.1347, time 150.66ms, mfu -100.00%
iter 3: loss 4.0995, time 580.57ms, mfu -100.00%
iter 4: loss 4.0387, time 487.72ms, mfu -100.00%
iter 5: loss 3.9758, time 136.06ms, mfu 0.01%
iter 6: loss 3.9126, time 518.57ms, mfu 0.01%
...
```

It's slow! Not only are we running on a CPU and not a highly parallelized GPU, but we also haven't used the just-in-time compilation features that are available in some GPU implementations of PyTorch. So, we're relying on an interpreted Python script. Yikes!

Every 250th iteration, the training script does a validation step. If the validation loss is lower than the previous value, it saves the model parameters.

```
step 250: train loss 2.4293, val loss 2.4447
saving checkpoint to out-shakespeare-char-cpu
...
```

What is happening?

When we train nanoGPT, it starts with randomly assigned weights and biases. This includes token embeddings (each token ID is assigned a random vector of size `n_embd`), attention weights for the query Q, key K, and value V matrices and their output projections, MLP weights in the feedforward network inside each transformer block, and bias terms, which are also randomly initialized (often to zero or small values). Training then tunes these values through gradient descent (using the fused AdamW optimizer – see `model.py`) to minimize loss and produce meaningful predictions.

Exercise 3: As the model trains, it reports the training and validation losses. In a Jupyter notebook, plot these values with the number of iterations. *Hint:* To capture the output when you perform a training run, you could run the process in the background while redirecting its output to a file: `python train.py config/train_shakespeare_char.py [options] > output.txt &`. (Remember, the ampersand at the end runs the process in the background.) You can still monitor the run by typing `tail -f output.txt`. This command will "follow" the end of the file as it is written.

After the training training finishes, we should have the model in `/out-shakespeare-char-cpu`

```
$ ls -l
total 20608
-rw-r--r--  1 furst  staff  9678341 Apr 18 17:41 ckpt.pt
```

In this case, the model is about 9.3 MB. That's not great! Our *training* text was only 1.1 MB! The point of this exercise is to demonstrate, very simply, the basics of a Generative Pre-trained Transformer, not to build an efficient and powerful LLM.

6 Generating text

The script `sample.py` runs inference on the model we just trained. We're using the CPU here, too.

```
$ python sample.py --out_dir=out-shakespeare-char --device=cpu
```

After a short time, the model will begin generating text.

```
I by done what leave death,
And aproposely beef the are and sors blate though wat our fort
Thine the aftior than whating bods farse dowed
And nears and thou stand murs's consel.

MEOF:
Sir, should and then thee.
```

Sounds a little more middle english than Shakespeare! But it has a certain generative charm.

Exercise 4: Examine `sample.py` and find the default parameters. Make a list of them and note their default values.

In the next few sections, we will try changing a few of the parameters in `sample.py`. One recommendation is to edit the number of samples `num_samples` and maybe the number of tokens `num_tokens`. These change the number of times the GPT model is queried and the amount of text that it will generate during each run. It's a little easier to experiment with fewer samples, for instance.

Before we continue, you might be seeing the following warning:

```
nanoGPT/sample.py:39: FutureWarning: You are using torch.load with
   weights_only=False (the current default value), which uses the default
   pickle module implicitly. It is possible to construct malicious pickle
   data which will execute arbitrary code during unpickling (See https://
   github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
   more details). In a future release, the default value for weights_only
   will be flipped to True. This limits the functions that could be
   executed during unpickling. Arbitrary objects will no longer be allowed
    to be loaded via this mode unless they are explicitly allowlisted by
   the user via torch.serialization.add_safe_globals. We recommend you
   start setting weights_only=True for any use case where you don't have
   full control of the loaded file. Please open an issue on GitHub for any
    issues related to this experimental feature.
  checkpoint = torch.load(ckpt_path, map_location=device)
```

This is warning us that PyTorch will soon default to `weights_only=True`, meaning it will only load tensor weights and not any other Python objects unless you explicitly allow them. We can instead use the following line in `sample.py`:[2]

[2]Since the checkpoint is from a trusted source (we trained it), it's safe to use `weights_only=False` also.

```
checkpoint = torch.load(ckpt_path, map_location=device, weights_only=True)
```

6.1 Seed

GPT output is probabilistic. The codes we use generate pseudo-random numbers. Using a `seed`, will cause the program to generate the same pseudo-random sequence. This is useful for testing the effect of other parameters. If you want to generate output that is different each time, comment out the following lines in `sample.py`:

```
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
```

Exercise 5: Remove seed and run `sample.py` a few times. Save your favorite output.

6.2 Temperature

Temperature is an interesting "hyperscaling parameter" of LLMs. Temperature controls the randomness of the model's responses. It influences how the model samples from the probabilities it assigns to possible next words during text generation. A higher temperature amplifies smaller probabilities, making the distribution more uniform, and a lower temperature reduces smaller probabilities, making the distribution more focused on the highest-probability tokens.

Exercise 6: Experiment by changing the model temperature and seeing what text it generates. Here, setting `seed` to a consistent value will help you understand the effect of temperature. At low temperatures, the text tends to repeat itself. At higher temperatures, sometimes the model generates gibberish. Why?

6.3 Start

The parameter `start` is the beginning of the text sequence. The model tries to determine the next most probable token. The default value is `\n`, a linefeed, but you can change `start` using the command line or by editing `sample.py`.

Exercise 7: Experiment with different strings in `start`. Some text is easier to enter in `sample.py` directly.

7 Exercises

Our output is pretty primitive. If you're willing to spend more time training and generating text, we can make the model a little larger. For instance, on an ARM-based Mac, we can use the GPU to train the model and run inferences. This is significantly faster and enables us to use larger models with noticeably higher fidelity:

```
$ python sample.py --out_dir=out-shakespeare-char-gpu --device=mps
Overriding: out_dir = out-shakespeare-char-gpu
```

```
Overriding: device = mps
number of parameters: 10.65M
Loading meta from data/shakespeare_char/meta.pkl...
RICHARD III::
Upon what!

KING EDWARD IV:
Thou in his old king I hear, my lord;
And commend the bloody, reason aching;
His mother, which doth his facit of his case,
his still, away; for we see heal us told
That seem her and the fall foul jealousing father;
And we shall weep with our napesty together.

FRIAR LAURENCE:
Transpokes her bloody and hour
To the tables of evident matters, her shoes
That the fatal ham to their death: do not high it
To read a passing thing into expeech him.
```

That text is generated using the default model parameters for nanoGPT. Not bad! The model is much larger. It has 10.6 million parameters compared to 800,000 in the smaller CPU-run model. When I train the model with the 'lighter" parameters we use for the CPU-based model, I see about 50-fold faster performance:

```
step 0: train loss 4.1676, val loss 4.1649
iter 0: loss 4.1828, time 764.41ms, mfu -100.00%
iter 1: loss 4.1373, time 34.71ms, mfu -100.00%
iter 2: loss 4.1347, time 19.60ms, mfu -100.00%
iter 3: loss 4.0995, time 18.56ms, mfu -100.00%
iter 4: loss 4.0387, time 20.71ms, mfu -100.00%
iter 5: loss 3.9758, time 17.55ms, mfu 0.07%
iter 6: loss 3.9126, time 17.84ms, mfu 0.07%
...
```

Compare those results to the times reported in section 5.2. By the way, `mfu` stands for *model flop utilization*. It is an estimate of the fraction of the GPU's floating point operation capacity (FLOPs) that the model is using per second. Low numbers like those reported here are typical of unoptimized, small models.

Exercise 8: Train nanoGPT with different parameters. Increase the size of the network, the context length, the length of training, etc.

8 Module project

Exercise 9: Find a different text to train nanoGPT on. It could be more Shakespeare (how about the sonnets?), Beowulf, or other work. What results do you get? *Hint:* https://huggingface.co/datasets has many text datasets to choose from. Present your results to the class.

9 Additional resources and references

9.1 Attention Is All You Need

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, Attention Is All You Need, in Proceedings of the 31st International Conference on Neural Information Processing Systems (Curran Associates Inc., Red Hook, NY, USA, 2017), pp. 6000–6010.

https://dl.acm.org/doi/10.5555/3295222.3295349

This is the paper that introduced the transformer architecture. It's interesting to go back to the source. The transformer architecture discussed in the paper incorporates both *encoder* and *decoder* functions because the authors were testing its performance on machine translation tasks. The transformer architecture's performance in other natural language processing tasks, like language modeling and text generation in the form of unsupervised pretraining and autoregressive generation (as in GPT) was a major subsequent innovation.

9.2 Andrej Karpathy

Andrej Karpathy wrote `nanoGPT`. He posts videos on Youtube that teach basic implementations of GPTs, applications of LLMs, and other topics on machine learning and AI. Karpathy's nanoGPT video shows you how to build it, step-by-step, including the mathematics behind the transformer and masked attention:

- https://www.youtube.com/watch?v=kCc8FmEb1nY

Also see his overview of LLMs, Intro to Large Language Models:

- https://www.youtube.com/watch?v=zjkBMFhNj_g

9.3 Applications in the physical sciences

I recommend watching this roundtable discussion hosted by the AIP Foundation in April 2024: *Physics, AI, and the Future of Discovery.* It addresses AI more broadly than language models.

- https://www.youtube.com/live/cUeEP15KN8M?si=TG6VXmj66lWTJISF

In that event, Prof. Jesse Thaler (MIT) provided some especially insightful (and sometimes funny) remarks on the role of AI in the physical sciences – including an April Fools joke, ChatJesseT. Below are links to his segments if you're short on time:

- https://www.youtube.com/live/cUeEP15KN8M?si=AIdi8sNEgiG7Bhv0&t=2087
- https://www.youtube.com/live/cUeEP15KN8M?si=UngwZpUcpxYkaYCE&t=611

Try ChatJesseT:

- https://chatjesset.com/

9.4 StatQuest guides

These books are informative and accessible resources for understanding the underlying math and vocabulary of transformers:

- Josh Starmer, The StatQuest Illustrated Guide to Neural Networks and AI, 2025
- Josh Starmer, The StatQuest Illustrated Guide to Machine Learning, 2022

10 Notes