# What does it take to train a Large Language Model ?

Suppose you wanted to replicate a Large Language Model such as GPT-3

In theory, you know to train a Large Language Model

- Gather a lot of text. Tokenize it.
$$\langle \text{token}_1 \rangle \langle \text{token}_2 \rangle \ldots$$
- Format the text into a training set for the "predict the next token" task
$$\begin{array}{lll} \mathbf{x^{(i)}} & = & \langle \text{token}_1 \rangle \langle \text{token}_2 \rangle \ldots \langle \text{token}_{i-1} \rangle & \text{example } i \text{ features} \\ \mathbf{y^{(i)}} & = & \langle \text{token}_i \rangle & \text{example } i \text{ target} \end{array}$$

  Then just invoke the `fit` method of your model.

You will face some practical impediments

- The training set for most published papers is not public
- Training is going to be
    - very time and compute intensive
    - expensive
    - fraught with unexpected errors

The [Open Pre-Trained Transformer LLM (OPT) (https://arxiv.org/pdf/2205.01068.pdf)](https://arxiv.org/pdf/2205.01068.pdf) is an effort to "democratize" access to LLM's

- Public training data
- Document the training process in detail
- Releases code used for training

It creates a LLM with 175 billion parameters (like GPT-3)

- with the objective that others should be able to replicate the work

It is easy to under-estimate the difficulty of the training process.

This paper demystifies the process.

There are a lot of details (as you would expect for a paper encouraging replication) !

We will highlight some lessons that we find interesting for those planning on training large models.

# Compute environment

OPT is trained on a multiple instances of the most advanced GPU available at the time

- NVIDIA A100 (https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/a100-80gb-datasheet-update-nvidia-us-1521051-r2-web.pdf), with 80GB of GPU memory

These are meant for data centers rather than personal computers (although a PC version is available)

- 400 Watts max power
- requires external cooling (Lots of it !)
- $15K cost: Amazon (https://www.amazon.com/Nvidia-Memory-Graphics-Ampere-A crid=3LOUTWUKFJPCX&keywords=nvidia+a100+80gb&qid=1677615169&s=elec 1) for PC version (at time this was written)

The authors used almost 1000 (992 to be precise) GPU's.

That's almost $15MM of GPU cost alone !

# Training lessons

## Expect hardware failures

When you are training for hours on 1000 machines

- you should expect hardware failure in the middle of training
    - 35 manual restarts over 2 months

Fortunately: the common Deep Learning API's (e.g, Keras, PyTorch) provide *check-pointing*

- can restart training for a checkpoint
- rather than beginning from scratch

## Expect training loss to diverge

In an ideal world: training loss decreases as the number of epochs of training increases.

In practice: this probably won't happen consistently over a long training run.

The authors were able to observe a relationship between Loss Divergence and

- magnitude of activations at the final layer
- the Loss Scalar (explained below) going to 0
    - Loss Scalar: a factor used to scale the true loss
        - to prevent overflow/underflow when using half-precision arithmetic
        - half-precision arithmetic: most throughput on your GPU

This enabled the authors to

- re-start the training
- from a prior checkpoint
- where the values of these two factors were "healthy"

But of course: re-starting in an *identical* training configuration could lead to a repetition of the problem.

The solution was to lower the learning rate

- an example of a *mid-flight correction*

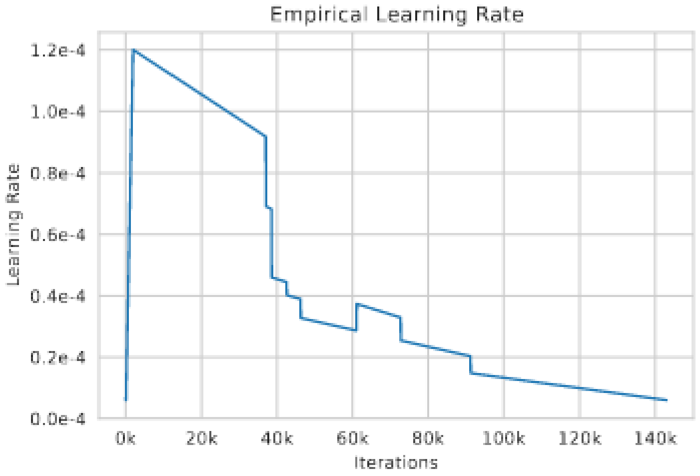


Figure 1: **Empirical LR schedule.** We found that lowering learning rate was helpful for avoiding instabilities.

**Note**

The original plan for the Learning Rate schedule

- Warm-up: 0 to maximum over 2000 steps
    - that's the big initial jump
- Linear decay to 10% of maximum after 300B tokens
- "Mid-flight corrections" are big jumps down

## Aside: the Loss Scalar

- In "Full Precision" arithmetic: 32 bits are used to encode a number.
- "Half Precison" uses only 16 bits. Smaller number of bits means
    - can fit more examples per batch
    - less data transfered from memory
- GPU can execute more operations per second in half-precision
    - faster training
    - but at a cost of reduced range of numbers that can be represented
        - the smallest 16 bit fraction is much larger than the smallest 32 bit fraction
        - so gradients that are small but *not mathematically* zero become zero in the half-precision representation
    - to avoid small gradients from becoming effectively zero:
        - Scale the gradient by a multiplicative factor: the Loss Scalar

See [NVIDIA Mixed Training Docuumentation (https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html#lossscaling)](https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html#lossscaling) if you are interested in the why and how of half-precision.

In [2]:

```
print("Done")
```

```
Done
```