Published in MantisNLP

Popescu Daniel Follow

Dec 21, 2022 · 8 min read · Listen

Prompt Engineering — Part I — How to interact with Large Language Models

This is the first in a series of blog posts that will dive into prompt engineering. The blogs accompany the presentation I did at PyData New York 2022, and will cover how prompt quality can have a dramatic effect on the quality of Large Language Model (LLM) performance, and the results and insights we generated by conducting experiments using different combinations of LLMs, Natural Language Processing (NLP) tasks, and prompts.

Large Language Models

So what exactly is an LLM? Well, the "large" part is rather subjective as larger and larger models are trained, but we can think of LLMs as language models that have been trained on massive amounts of data, using a substantial amount of resources.

We describe the size of a model based on the number of trainable parameters. These are parameters internal to the model for which a value will be learned through the training process. The more parameters, the more training is required, and the more complex problems that the model will be able to solve. For example GPT-3, a model which has around 175 Billion trainable parameters, costs around 5 million dollars to train, and shows impressive performance on a range of NLP tasks, as we will see.

The other axis on which we can evaluate model size is the size of the dataset on which the model was trained. See the table below for a comparison of three models all considered to be large in their time.

	BERT	GPT-J	BLOOM
Model size (params)	350 Million	6 Billion	175 Billion
Dataset size	7 Gigabytes	825 Gigabytes	1.5 Terabytes

What has driven this explosion in the size of models is the transformer architecture. Prior to transformers, a common architecture for NLP tasks was the Recursive Neural Network such as the LSTM (Long Short Term Memory). While groundbreaking in their time, these models did not scale well. Transformers, on the other hand, seem capable of improving performance as more and more data is thrown at them — the limits of this improvement are yet to be hit.

LLMs are typically trained using masked token prediction'. In this task, a portion (e.g. 15% for BERT) of tokens in a large corpus of text are 'masked', and the model is asked to predict the missing token. Hence, for the first generations of LLMs, the only task that the models were put to was text generation, which is essentially what we are asking the model to do during the training process. Researchers quickly realized however, that by providing the LLM with a specific prompt, they could convert most NLP tasks into a text generation task.

When GPT-3 was released, an immense 175B parameter model (115 times bigger than its predecessor GPT-2), it appeared to be a few shot learner. This means that it was capable of taking previously unseen tasks such as text classification, and it would perform reasonably well only having seen a few, say 3–5, training examples. Suddenly you could tell the model to do almost anything, and it kind of worked out of the box! Since then things have diverged in the industry: there are those working on producing ever larger LLMs, such as the Nvidia's Megatron-NLG which contains some 530B trainable parameters; and there are others that are working on

capturing similar performance but on smaller models, for example Flat-T5 which has *just* a few Billion parameters.

The attraction was that you can prompt the model with the problem in natural language and the model figures out by itself (or with very few examples) the solution, and it doesn't matter whether that problem is Question Answering (QA), Named Entity Recognition (NER), Text classification, and so on. This is why the prompt aspect is so important.

For our experiments, we selected 12 LLMs to work with. We selected them based on performance, size and popularity. You can see them in the graph below, ordered by size.

models ordered by size (Billions of parameters)

How to interact with LLMs

As we can see from the previous plot some of these models are gigantic (a 30B model for example takes about 70GB of space on disk!). So hosting them can be expensive and complicated.

Fortunately there are some approaches to solving this problem, and in this blog we'll look at 3 ways you can work with LLMs:

- Using APIs
- Hosting them and using Hugging Face Accelerate
- Using Deepspeed-Mii

LLM APIs

Several companies offer access to models through an API, and this is a really easy way to interact with a model. All you only need to do is call the API with your prompt and maybe some model parameters.

A few examples of APIs:

- OpenAI — offers it's GPT-3 model versions through an API
- Co:here — offers similar services to OpenAI, with a focus on helping you with particular NLP tasks
- AI21labs — has its own jurassic models that are similar in size to GPT-3
- GooseAI has GPT-J and GPT-NeoX-20B
- HuggingFace Inference Endpoints — This allows you to host and call many of the models available on HuggingFace. There is still a bit of work to be done though for big models to be available. Btw, you can easily use HuggingFace Inference through our hugie tool!

Whilst APIs are the quickest way to get started with LLMs, they may not be the best way for every use case. Let's look at the pros and cons of using APIs which will help you decide if it is right for you. We address most of the cons in the following sections.

Pros of using APIs:

- Easy to use
- Reliable uptime
- You get to experiment with a model very fast
- No hosting and security concerns

Cons of using APIs:

- Can become very expensive depending on the amount of calls you make
- Not all models have APIs
- Inference time can be slow
- You might not have control over all the generation parameters

Because of these disadvantages, you might decide that you need to host a model privately for yourself. So let's look at some ways you can do that.

Using HuggingFace Accelerate

We usually want to run LLMs on a Graphics Processing Unit (GPU) for speed, but they are usually too big to fit within the memory of even the largest GPUs. Hugging Face Accelerate allows the layers of the model to be split across multiple GPUs, or part on GPU and part on system memory. When you load the model, Accelerate looks at the devices on your machine and automatically maps the layers of the model to those devices (prioritizing the GPU).

So, let's say I want to load the "flan-t5-xxl" model using Accelerate on an instance with 2 A10 GPUs containing 24GB of memory each. With Accelerate's integration to the transformers library, you can now simply do:

```
model = T5ForConditionalGeneration.from_pretrained(
   "flan-t5-xxl",
   device_map="auto",
)
```

By using **device_map="auto"** we tell it to use Accelerate and to take care of splitting the model between devices. Still, it's good practice (for now) to add a few other parameters:

```
model = T5ForConditionalGeneration.from_pretrained(
   "flan-t5-xxl",
   low_cpu_mem_usage=True,
   torch_dtype=torch.bfloat16,
   device_map="auto",
   max_memory={0: "18GiB", 1: "10GiB"}
)
```

Setting **torch_dtype** to **torch.bfloat16** means we want to use mixed precision meaning that we set the tensors to use smaller data types (usually float16 as opposed to float32) which uses less memory. So the model becomes smaller and also faster, at the cost of very little drop in performance. Setting **low_cpu_mem_usage** means it will prioritize filling the GPU memory first.

Finally, the **max_memory** parameter allows us to specify the maximum memory we want each device to use. This is important because Accelerate will, by default, use as much memory as possible from each device, which can leave us without any memory left to call the generate function resulting in an Out of Memory (OOM) error. Hence, it's good to leave some space on the GPUs (especially on the first one) for when we generate the response. How much depends on the length of the inputs we plan to pass to the model, and the length of sequence we will generate as response, so it requires some experimentation. You can see a representation of this is the figure below.

Visualization of how a model is split onto 2 GPUs

There are a lot of guides on how to use Accelerate on Hugging Face's documentation. It is also still in active development, which means what you see now might be done differently in a few months (in fact, when we started our experiments we had to use some workarounds in the code, which now have been solved: it's now much easier to use).

Using DeepSpeed-MII

Recently, DeepSpeed released DeepSpeed-MII, a wonderful way of loading and making your model available in your own mini API.

It works very easily just by configuring the service:

```
import mii
mii_configs = {"tensor_parallel": 1, "dtype": "fp16"}
mii.deploy(task="text-generation",
           model="bigscience/bloom-560m",
           deployment_name="bloom560m_deployment",
           mii_config=mii_configs)
```

And then you call it either with an API call or with mii functionality:

```
import mii
generator = mii.mii_query_handle("bloom560m_deployment")
result = generator.query({"query": ["DeepSpeed is", "Seattle is"]}, do_sample=True, ma
print(result)
```

Helpfully you can do mixed precision here too with just a few parameters, and deepspeed will also split the model in the same way that Hugging Face's Accelerate does.

All of this goes in a dictionary that represents the deepspeed configuration. For example

```
ds_config = {
    "fp16": {
        "enabled": True
    },
    "zero_optimization": {
        "stage": 3,
        "offload_param": {
            "device": "cpu",
        },
    },
    "train_micro_batch_size_per_gpu": 1,
}
```

Here we enable fp16 (mixed precision on float16), and using **stage=3** in zero_optimization we tell it to automatically partition the model across our available devices.

There's a lot more configuration that can be done with DeepSpeed and DeepSpeed-MII that we will explore in a future blog. It's worth mentioning DeepSpeed Compression here too, which can greatly help to reduce the size of a model.

Outro

In this blog, we've introduced the idea of prompts for LLMs, and talked a little bit about how we can get started with LLMs either through APIs, or by deploying the model on our own hardware. It's the first in our series of blogs about LLMs and prompt engineering. In the next blog we'll look at prompts and how to construct them.

Large Language Models Prompt Engineering Gpt 3 Naturallanguageprocessing