

Lecture 24 — Large Language Models

Jeff Zarnett

2023-09-19

Large Language Models and You

In November of 2022, OpenAI introduced ChatGPT to the world and suddenly everyone and their best friend was doing some combination of (1) writing news articles about how the advent of AI means we will all be unemployed and lead to the rise of Skynet; (2) founding a startup that used ChatGPT to do something something B2B/B2C SaaS something something profit; (3) rebranding themselves on LinkedIn as “prompt engineering” experts; or (4) generating lots social media cringe content about how to use it to get rich easily¹.

Such large language models have existed before, but ChatGPT ended up a hit because it’s pretty good at being “conversational”, which is to say that it does a good job of responding to input in a way that seems like how a human would respond. For this reason, it’s referred to as NLP – Natural Language Processing. In the words of some experts [KSK⁺23]: “These models are trained on massive amounts of text data and are able to generate human-like text, answer questions, and complete other language-related tasks with high accuracy.” That is only one of many kinds of large language models and there are many different kinds of machine learning systems out there that do other tasks like recognize images.

Of course, just because such a tool can produce an answer to your question, doesn’t mean it is necessarily true or correct. You may enjoy this Legal Eagle video about a lawyer who used ChatGPT for “research” and found that the system returned an answer with completely made-up references. These are sometimes called “hallucinations”. We don’t have time to watch the video in lecture, but I encourage you to watch it to get an understanding of what went wrong: <https://www.youtube.com/watch?v=oqSYLjRYDEM> and why you should not rely on it without checking its output. Remember that in engineering, legally and professionally speaking, the engineer is responsible for understanding how a given tool works and verifying the output is reasonable or correct; a civil engineer who says that the software told them the building was fine will be held liable (both in discipline and legally) if the building was not safe and falls down...

Part of what makes the GPT-3 and GPT-4 models better at producing output that matches our expectations is that it relies on pre-training (it’s the PT part of GPT) on a very large data set, specifically a lot of stuff just out there on the internet [KSK⁺23]. This course is not one on neural networks, large language models, AI, or similar – there are other such technical electives which you may be taking. This does connect to the course content, however, because generating or updating a pre-trained model is computationally challenging... so performance increases matter here.

Parameters. One factor, but certainly not the only one, in how good the model is at responding to requests is the number of parameters. To explain a bit about why it matters, consider this quote from [MB23]: “LLM AI models are generally compared by the number of parameters — where bigger is usually better. The number of parameters is a measure of the size and the complexity of the model. The more parameters a model has, the more data it can process, learn from, and generate. However, having more parameters also means having more computational and memory resources, and more potential for overfitting or underfitting the data. Parameters are learned or updated during the training process, by using an optimization algorithm that tries to minimize the error or the loss between the predicted and the actual outputs. By adjusting the parameters, the model can improve its performance and accuracy on the given task or domain.”

¹Watch this video on the subject of “Get Rich Easy” schemes: <https://www.youtube.com/watch?v=2bq3SdfzCA4>

Optimizing LLMs

The content from this section is based on a guide from “Hugging Face” which describes itself as an AI community that wants to democratize the technology. The guide in question is about methods and tools for training using one GPU [Fac23b] (but we can discuss multi-GPU also). Indeed, you may have guessed by the placement of this topic in the course material that the GPU is the right choice for how to generate or train a large language model.

Okay, but why a GPU? In this case we’re talking about Transformers and there are three main groups of optimizations that it does [Fac23c]: Tensor Contractions, Statistical Normalizations, and Element-Wise Operators. Contractions involve matrix-matrix multiplications and are the most computationally challenging part of the transform; statistical normalizations are a mapping and reduction operation; and element-wise operators are things like dropout and biases and these are not very computationally-intensive. We don’t need to repeat the reasoning as to why GPUs are good at matrix-matrix multiplication and reduction operations since that’s already been discussed.

In discussing the optimizations we can make, we’ll also need to consider what is in memory, since it’s possible that our training of a model might be limited by available GPU memory rather than compute time. Things like the number of parameters and temporary buffers count towards this limit.

Optimizing. There are two kinds of optimizations that are worth talking about. The first one is the idea of model performance: how do we generate a model that gives answers or predictions quickly? The second is how can we generate or train the model efficiently.

The first one is easy to motivate and we have learned numerous techniques that could be applied here. Examples: Use more space to reduce CPU usage, optimize for common cases, speculate, et cetera. Some of these are more fun than others: given a particular question, can you guess what the followup might be?

Before we get into the subject of how, we should address the question of why you would wish to generate or customize a LLM rather than use an existing one. To start with, you might not want to send your (sensitive) data to a third party for analysis. Still, you can download and use some existing models. So generating a model or refining an existing one may make sense in a situation where you will get better results by creating a more specialized model than the generic one. To illustrate what I mean, ChatGPT will gladly make you a Dungeons & Dragons campaign setting, but you don’t need it to have that capability if you want it to analyze your customer behaviours to find the ones who are most likely to be open to upgrading their plan. That extra capability (parameters) takes up space and computational time and a smaller model that gives better answers is more efficient.

Our first major optimization, and perhaps the easiest to do, is the batch size. The batch size is just telling the GPU how much to do at once. It’s a little bit like when we discussed the idea of creating more threads to increase performance; you may see an improvement by having more workers active but you also may not get any additional benefit from worker $N + 1$ over N since there may not be enough work or other resource conflicts.

I’ve used an example from Hugging Face [Fac23c] with some light modifications to see what we can do with a very simply example using dummy data. Let’s go over and look at that example now. It’s in Python (a lot of LLM, machine learning, etc. content is) but it shouldn’t be too difficult to understand as we walk through it.

```
import numpy as np
import torch
from datasets import Dataset
from pytorch import *
from transformers import AutoModelForSequenceClassification
from transformers import TrainingArguments, Trainer, logging

default_args = {
    "output_dir": "tmp",
    "evaluation_strategy": "no",
    "num_train_epochs": 1,
    "log_level": "error",
    "report_to": "none",
}
```

```

def print_gpu_utilization():
    nvmlInit()
    handle = nvmlDeviceGetHandleByIndex(0)
    info = nvmlDeviceGetMemoryInfo(handle)
    print(f"GPU_memory_occupied:_{info.used//_1024*_2}_MB.")

def print_summary(res):
    print(f"Time:_{res.metrics['train_runtime']:.2f}")
    print(f"Samples/second:_{res.metrics['train_samples_per_second']:.2f}")
    print_gpu_utilization()

print("Starting up. Initial GPU utilization:")
print_gpu_utilization()
torch.ones((1, 1)).to("cuda")
print("Initialized Torch; current GPU utilization:")
print_gpu_utilization()

model = AutoModelForSequenceClassification.from_pretrained("bert-large-uncased").to("cuda")
print_gpu_utilization()

logging.set_verbosity_error()

seq_len, dataset_size = 512, 512
dummy_data = {
    "input_ids": np.random.randint(100, 30000, (dataset_size, seq_len)),
    "labels": np.random.randint(0, 1, dataset_size),
}
ds = Dataset.from_dict(dummy_data)
ds.set_format("pt")

training_args = TrainingArguments(per_device_train_batch_size=4, **default_args)
trainer = Trainer(model=model, args=training_args, train_dataset=ds)
result = trainer.train()
print_summary(result)

```

The bert-large-uncased model [DCLT18] is not a particularly large one – it says on its data sheet that it’s about 340 MB – and it’s trained on a bunch of English language data. It’s uncased because it makes no distinction between capitals and lower-case letters, e.g., it sees “Word” and “word” as equivalent.

First I tried to run it on my laptop, but that failed because it does not have any nvidia GPU, which is not surprising. Next I tried to run this on ecetesla0 and I saw the following output (skipped some of the stack trace):

```

jzarnett@ecetesla0:~/github/ece459/lectures/live-coding/L24$ python3 dummy_data.py
Starting up. Initial GPU utilization:
GPU memory occupied: 0 MB.
Initialized Torch; current GPU utilization:
GPU memory occupied: 417 MB.
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at
bert-large-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions
and inference.
GPU memory occupied: 1705 MB.
torch.cuda.OutOfMemoryError: CUDA out of memory. Tried to allocate 20.00 MiB (GPU 0;
7.43 GiB total capacity; 6.90 GiB already allocated; 16.81 MiB free; 6.90 GiB
reserved in total by PyTorch) If reserved memory is >> allocated memory try setting
max_split_size_mb to avoid fragmentation. See documentation for Memory Management
and PYTORCH_CUDA_ALLOC_CONF

```

So the ecetesla0 machine ran out of memory trying to process this. Using nvidia-smi I learned that the card has only 7611MiB of VRAM available and that does not seem like a lot for the kind of work we are trying to do.

The configuration we had asked for a batch size of 4 and it's possible that this is just too much to fit in memory at once for this old card. Reducing batch size to 2 did not help, nor did 1. This is a clear indication that for the model that we want to use, the card isn't going to cut it. Scotty, we need more power.

What I actually did next was change to a smaller version of the model, bert-base-uncased which was significantly smaller (110 MB) and something the card could handle. Here's the output with batch size of 1:

```
jzarnett@ecetesla0:~/github/ece459/lectures/live-coding/L24$ python3 dummy_data.py
Starting up. Initial GPU utilization:
GPU memory occupied: 0 MB.
Initialized Torch; current GPU utilization:
GPU memory occupied: 417 MB.
Some weights of BertForSequenceClassification were not initialized from the model checkpoint at
bert-base-uncased and are newly initialized: ['classifier.weight', 'classifier.bias']
You should probably TRAIN this model on a down-stream task to be able to use
it for predictions and inference.
GPU memory occupied: 887 MB.
{'loss': 0.0028, 'learning_rate': 1.1718750000000001e-06, 'epoch': 0.98}
{'train_runtime': 109.6152, 'train_samples_per_second': 4.671,
 'train_steps_per_second': 4.671, 'train_loss': 0.0027378778694355788, 'epoch': 1.0}
Time: 109.62
Samples/second: 4.67
GPU memory occupied: 3281 MB.
```

Then I needed to experiment some more with batch size to find the ideal for this card. To condense the results a little bit, see the results table below.

Batch Size	Time (s)	Samples/s	Memory Occupied (MB)	Utilization (%)
1	109.62	4.67	3 281	43.1
2	85.82	5.97	3 391	44.6
4	72.18	7.09	4 613	60.6
8	66.70	7.68	7 069	92.9

And given what we know about the GPU in this system, it's not surprising the OOM error returns when the batch size is increased to 9. The other thing that's nice is that the OOM is encountered very quickly on startup so it's easy to just binary search different batch sizes to find the maximum you can process in one go.

We can try some other optimization techniques to see if we can squeeze out a little more performance from this. There are a number of different techniques that we can focus on to try to optimize memory utilization since that's our limiting factor. Focusing on memory utilization is a part of what makes this topic a little different than most of the others we've covered in this course, which tend to be much more compute-focused.

Gradient Accumulation. The idea behind gradient accumulation is to calculate gradients in small increments rather than for the whole batch; doing this can increase the effective batch size, at the risk of slowing down the total process by having too many compute steps [Fac23b].

Experimenting with this and batch size fixed at 8:

Gradient Accumulation Steps	Time (s)	Samples/s	Memory Occupied (MB)	Utilization (%)
1	66.06	7.75	7 069	92.9
2	63.96	8.01	7 509	98.7
4	62.81	8.15	7 509	98.7
8	62.65	8.17	7 509	98.7
16	62.42	8.20	7 509	98.7
32	62.44	8.20	7 509	98.7
128	62.20	8.23	6 637	87.2
1024	61.78	8.29	6 637	87.2
4096	62.16	8.24	6 637	87.2

We can see that we very quickly hit diminishing returns on this, but it seems like increasing the number continues to have a marginal benefit, basically for free, up until we get to around 1024. However, I got suspicious about the 128 dropoff in memory usage and it made me think about other indicators – is it getting worse somehow? The output talks about training loss...

Gradient Accumulation Steps	Loss
1	0.029
2	0.070
4	0.163
8	0.169
16	0.447
32	0.445
128	0.435
1024	0.463
4096	0.014

Does that seem concerning? We won't really know unless we do some validation – and this is random data so validating it won't really work for this scenario. Are we perhaps trading accuracy for time? I think the only way to find out is that we need to have a validation data set. We could get through the first steps here of batch size without giving much thought to this part, but now we're kind of stuck. So let's find out.

We'll follow another guide from [Fac23a] where the goal is to train and validate using some Yelp data. Yes, Yelp, the website that struggling restaurant owners blame for ruining their “gourmet burger” place that charges you \$22 for an unimpressive reheated Sysco hamburger with no side dish. Running this does take significantly longer, but that's to be expected. The training is divided into three epochs and accuracy is calculated at the end of each of those using a training and evaluation set.

```
import evaluate
import numpy as np
import torch
from datasets import load_dataset
from evaluate import evaluator
from pytorch import *
from transformers import AutoModelForSequenceClassification
from transformers import AutoTokenizer
from transformers import TrainingArguments, Trainer, logging

def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length", truncation=True)

def print_gpu_utilization():
    nvmlInit()
    handle = nvmlDeviceGetHandleByIndex(0)
    info = nvmlDeviceGetMemoryInfo(handle)
    print(f"GPU_memory_occupied: {info.used//1024**2} MB.")
```

```

def compute_metrics(eval_pred):
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    computed = metric.compute(predictions=predictions, references=labels)
    print(computed)
    return computed

def print_summary(res):
    print(f"Time:_{res.metrics['train_runtime']:.2f}")
    print(f"Samples/second:_{res.metrics['train_samples_per_second']:.2f}")
    print_gpu_utilization()

print("Starting up._Initial_GPU_utilization:")
print_gpu_utilization()
torch.ones((1, 1)).to("cuda")
print("Initialized_Torch;_current_GPU_utilization:")
print_gpu_utilization()

dataset = load_dataset("yelp_review_full")
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

tokenized_datasets = dataset.map(tokenize_function, batched=True)

small_train_dataset = tokenized_datasets["train"].shuffle(seed=42).select(range(1000))
small_eval_dataset = tokenized_datasets["test"].shuffle(seed=42).select(range(1000))

model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=5)
training_args = TrainingArguments(
    per_device_train_batch_size=8,
    gradient_accumulation_steps=1,
    evaluation_strategy="epoch",
    output_dir="test_trainer",
)
metric = evaluate.load("accuracy")

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=small_train_dataset,
    eval_dataset=small_eval_dataset,
    compute_metrics=compute_metrics,
)

result = trainer.train()
print_summary(result)

```

And our results table with batch size of 8. I've skipped some intermediate results since at 9 minutes to calculate it takes a while to fill in all the values above. But jumping up some levels illustrates the trend.

Gradient Accumulation Steps	Time (s)	Samples/s	Memory Occupied (MB)	Final Accuracy
1	538.37	5.56	7 069	0.621
8	501.89	5.98	7 509	0.554
32	429.70	6.98	7 509	0.347
1024	513.17	5.85	7 509	0.222

I ran the 32 case a few times to check if this was an outlier – the one in the table is the best result. But it's still noticeably lower than that of the case where gradient accumulation is 1. Interesting, right? Increasing the gradient accumulation does change the effective batch size, and as you may know, increasing the batch size too large means less ability to generalize. Which is another way of saying that the model gets stuck at local minima or overfits the data.

That's not to say that smaller batch sizes are always better; models are way more complicated than that – we can also underfit the model. It's part of why it's important to have training and validation data, so we can optimize and find the right balance. In the Yelp example, I get worse accuracy with batch size of 1 than 4, and 4 is worse than 8. There really is no magic number.

Other Ideas

Just in the interests of time, we won't be able to experiment with everything, but the source has some other ideas that are worth mentioning as they relate to other course concepts we've discussed [Fac23b].

Gradient Checkpointing. This approach is based around the idea of increasing compute time to reduce memory usage. It might allow us to work with a bigger model even with our fairly limited card memory, but training will take longer; according to the source it might be about 20%. By default, all activations from the forward-pass are saved so they can be used in the backward pass; we could not save them and recalculate them from scratch on the backward pass. That would save the most memory but take the most time. A compromise approach is to save some of the activations so that the total amount to recompute on the backward pass is less.

Trying this out with batch size of 8 and gradient accumulation turned off, the total time goes from 66.70 to 93.07s and the memory from 7 069 down to 3619 MB. As expected, we got slower but used less memory. Actually, more like half the memory. Maybe it means we can increase the batch size? Raising it to 16 means the time was 100.55s but still only 3731 MB.

Increasing the batch size a lot to finish faster might work, although it might require a very large batch size and not really save us anything since it would take quite a lot to fall below the time taken when not using the checkpointing. And no, using this checkpointing even with a batch size of 1 is not sufficient to run the bert - large - uncased model on ecetesla0. And remember that excessively large batch sized make things worse.

Mixed Precision. This is a fairly straightforward tradeoff of accuracy for time; while the default for most things might be 32-bit floating point numbers, if we don't need that level of precision then some of the 32-bit types could be replaced with 16-bit ones and this can speed up calculations.

Data Preloading. If your limiting factor is in getting work to the GPU, data pre-loading is about either pinned memory or multi-threads to get data to the GPU faster. If you recall from the operating systems course you (hopefully) took, pinned memory is pages of memory where the operating system is instructed not to swap those pages to disk (i.e., keep them in RAM) for faster access. And multiple threads, well, this is clear at this point.

This is by no means exhaustive – the guide talks about other ideas that we haven't got time to cover, like Mixture of Experts, which are very deep into the details and beyond what we want to cover here. And finally, we could consider doing things like buying a bigger (better) GPU, or using multiple GPUs for more parallelism. All the things we know about CPU work in this problem domain.

Tradeoffs

More than any other topic, the LLM topic shows the inherent tradeoffs in optimizing things. Do we trade memory for CPU? Do we trade accuracy for time? Do we prefer to err on the side of under- or over-fitting the model and how does that affect our choices on the other dimensions? I imagine that in the next few years our tools and ways of deciding these things will become much more sophisticated and best practices and known-good answers will emerge. But in the meantime, we can have a lot of fun experimenting and learning.

References

- [DCLT18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL: <http://arxiv.org/abs/1810.04805>, arXiv:1810.04805.
- [Fac23a] Hugging Face. Fine-tune a pretrained model (v. 4.33.0), September 2023. Online; accessed 2023-09-16. URL: <https://huggingface.co/docs/transformers/main/training#prepare-a-dataset>.
- [Fac23b] Hugging Face. Methods and tools for efficient training on a single GPU (v. 4.33.0), September 2023. Online; accessed 2023-09-11. URL: https://huggingface.co/docs/transformers/perf_train_gpu_one.
- [Fac23c] Hugging Face. Model Training Anatomy (v. 4.33.0), September 2023. Online; accessed 2023-09-13. URL: https://huggingface.co/docs/transformers/model_memory_anatomy.
- [KSK⁺23] Enkelejda Kasneci, Kathrin Sessler, Stefan Küchemann, Maria Bannert, Daryna Dementieva, Frank Fischer, Urs Gasser, Georg Groh, Stephan Günemann, Eyke Hüllermeier, Stephan Krusche, Gitta Kutyiniok, Tilman Michaeli, Claudia Nerdel, Jürgen Pfeffer, Oleksandra Poquet, Michael Sailer, Albrecht Schmidt, Tina Seidel, Matthias Stadler, Jochen Weller, Jochen Kuhn, and Gjergji Kasneci. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and Individual Differences*, 103:102274, 2023. URL: <https://www.sciencedirect.com/science/article/pii/S1041608023000195>, doi:10.1016/j.lindif.2023.102274.
- [MB23] John Maeda and Matthew Bolaños. What are Models?, May 2023. Online; accessed 2023-09-10. URL: <https://learn.microsoft.com/en-us/semantic-kernel/prompt-engineering/llm-models>.