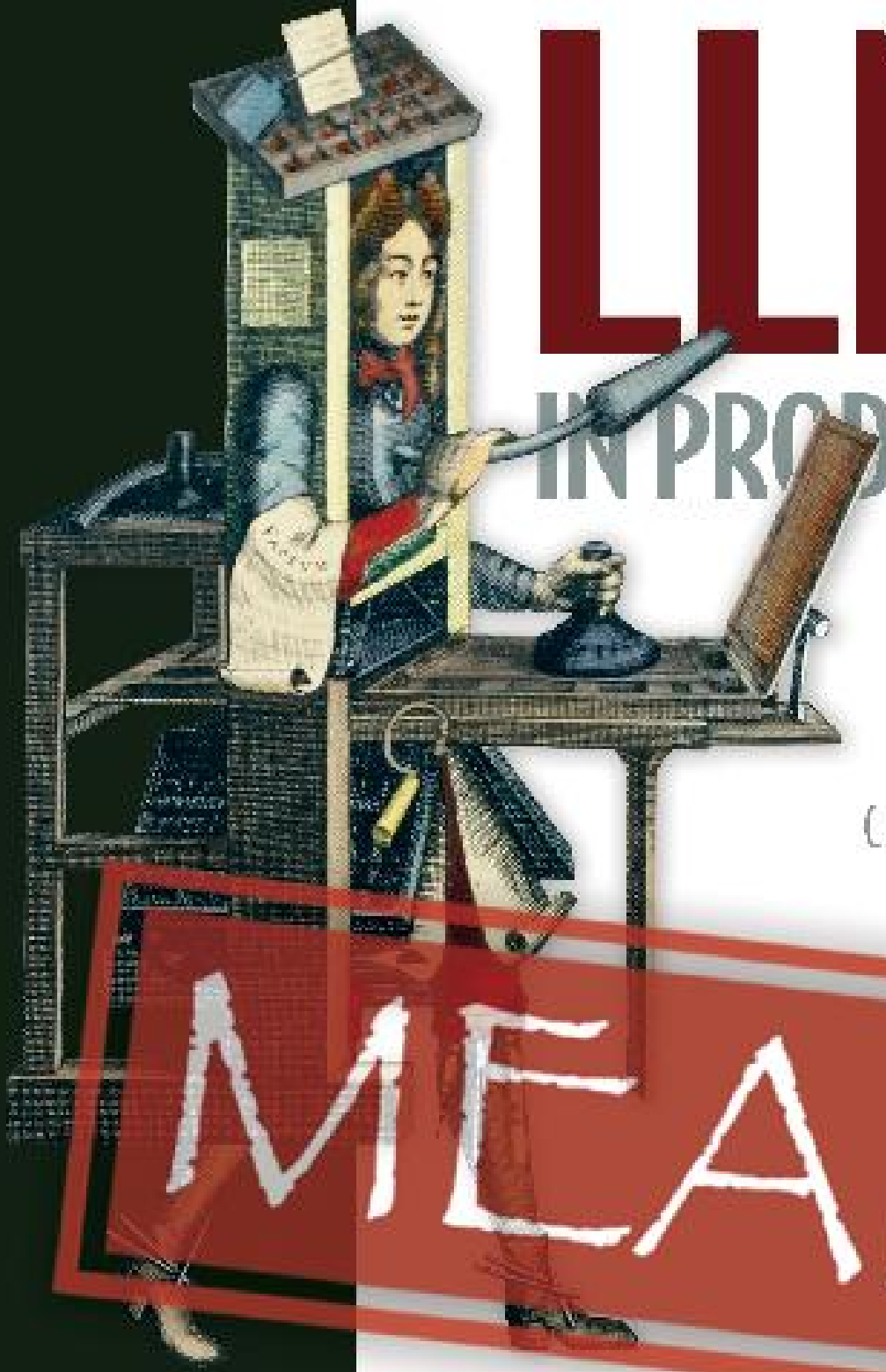
From language models to successful products

LLMs IN PRODUCTION



Christopher Brousseau
Matthew Sharp

MEAP

MANNING

LLMs in Production

1 Word’s Awakening: Why Large Language Models Have Captured Attention

This chapter covers

- What Large Language Models are and what they can and cannot do
- When you should deploy your own Large Language Models and when should not
- Large Language Model myths and the truths that lie behind them

"Any sufficiently advanced technology is indistinguishable from magic."

- Arthur C. Clarke

The year is 1450. A sleepy corner of Mainz, Germany, unknowingly stands on the precipice of a monumental era. In Humbrechthof, a nondescript workshop shrouded in the town's shadows pulsates with anticipation. It is here that Johannes Gutenberg, a goldsmith and innovator, sweats and labors amidst the scents of oil, metal, and determination, silently birthing a revolution. In the late hours of the night, the peace is broken intermittently by the rhythmic hammering of metal on metal. In the lamp-lit heart of the workshop stands Gutenberg's decade-long labor of love—a contraption unparalleled in design and purpose.

This is no ordinary invention. Craftsmanship and creativity transform an assortment of moveable metal types, individually-cast characters born painstakingly into a matrix. The flickering light dances off the metallic insignias. The air pulsates with the anticipation of a breakthrough and the heady sweetness of oil-based ink, an innovation from Gutenberg himself. In the stillness of the moment, the master printer squares his shoulders and with unparalleled finesse, lays down a crisp sheet of parchment beneath the ink-loaded matrix and allows his invention to press firmly and stamp finely print onto the page. The room adjusts to the symphony of silence, bated breaths hanging heavily in the air. As the press is lifted, it creaks under its own weight, each screech akin to a war cry announcing an exciting new world.

With a flurry of motion, Gutenberg pulls from the press the first printed page and slams it flat onto the wooden table carefully examining each character which are as bold and magnificent as the creator's vision. The room drinks in the sight, absolutely spellbound. A mere sheet of parchment has become a testament of transformation. As the night gives way to day, he looks upon his workshop with invigorated pride. His legacy is born, echoing in the annals of history and forever changing the way information would take wings. Johannes Gutenberg, now the man of the millennium, emerges from the shadows, an inventor who dared to dream. His name synonymous with the Printing Press which is not just a groundbreaking invention, but the catalyst of a modern world.

As news of Gutenberg's achievement begins to flutter across the continent, scholars from vast disciplines are yet to appreciate the extraordinary tool at their disposal. Knowledge and learning, once coveted treasures, are now within the reach of the common man. There were varied and mixed opinions surrounding that newfound access.

> *“In our time, thanks to the talent and industry of those from the Rhine, books have emerged in lavish numbers. A book that once would’ve belonged only to the rich - nay, to a king - can now be seen under a modest roof. […] There is nothing nowadays that our children […] fail to know.”*
>
> *- Sebastian Brant*

> *“Scholarly effort is in decline everywhere as never before. Indeed, cleverness is shunned at home and abroad. What does reading offer to pupils except tears? It is rare, worthless when it is offered for sale, and devoid of wit.”*
>
> *- Egbert of Liege*

People have had various opinions on books throughout history. One thing we can agree on, living in a time when virtual printing presses exist and books are ubiquitous: the printing press changed history. While we weren’t actually there when Gutenberg printed the first page using his printing press, we have

watched many play with large language models (LLMs) for the first time. The astonishment on their face as they see it respond to their first prompt. The excitement they have when challenging it with a difficult question only to see it respond as if it was an expert in the field. The light bulb moment when they realize they can use this to simplify their life or make themselves wealthy. I imagine this wave of emotions were, but a fraction of those felt by Johannes Gutenberg. Being able to rapidly generate text and accelerate communication has always been valuable.

1.1 Large Language Models accelerating communication

Every job has some level of communication. Oftentimes this communication is drudgery, bureaucratic, and political. I've often warned students and mentees that every job has its paperwork. Something that used to be a passion can easily be killed by the day to day tedium and menial work that comes with it when it becomes a job. In fact, when we talk about our professions we often talk them up, trying to improve our social standing, so you'll rarely get the full truth. You won't hear about the boring parts and the day-to-day grind gets conveniently forgotten.

However, envision a world where we reduce the burden of monotonous work. A place where police officers no longer have to waste hours of each day filling out reports and could instead devote that time to community outreach programs. Or a world where teachers, no longer slaving late into the night grading homework and preparing lesson plans, instead being able to think about and prepare customized lessons for individual students. Or even a world where lawyers would no longer be stuck combing through legal documents for days, instead being free to take on charity cases for causes that inspire them? When the communication burden, the paperwork burden and the accounting burden, are taken away, the job becomes more akin to what we sell it as.

For this, LLMs are the most promising technology to come along since, well, the printing press. For starters, they have completely upended the role and relationship between humans and computers, transforming what we

believed they were capable of. They have already passed medical exams[1], the bar exam, and multiple theory of mind tests. They've passed both Google and Amazon coding interviews. They've gotten scores of at least 1410 out of 1600 on the SAT. One of the most impressive to the authors is that GPT-4 has even passed the Advanced Sommelier exam–which makes us wonder how they got past the practical wine tasting portion. Indeed, their unprecedented accomplishments are coming at breakneck speed and often make us mere mortals feeling a bit queasy and uneasy. What do you do with a technology that seems to be able to do anything?

Passing tests is fun and all, but not exactly all that helpful unless our aim was to build the most expensive cheating machine ever, which we promise there's better use of our time. What LLMs are good at is language, particularly, helping us improve and automate communication. This allows us to transform common bitter experiences into easy enjoyable experiences. For starters, imagine entering your home where you have your very own personal JARVIS, as if stepping into the shoes of Iron Man, an AI-powered assistant that adds an unparalleled dynamic to your routine. While not quite to the same artificial general intelligence (AGI) levels as those portrayed by JARVIS in the Marvel movies, LLMs are powering new user experiences from improving customer support to helping you shop for a loved one's birthday. It knows to ask you about the person, learn about their interests and who they are, find out your budget, and then make specialized recommendations. While many of these assistants are being put to good work, many others are simply just chatbots that users can talk to and entertain themselves—which is important because even our imaginary friends are too busy these days. Jokes aside, these can create amazing experiences allowing you to meet your favorite fictional characters like Harry Potter, Sherlock Holmes, Anakin Skywalker or even Iron Man.

What we're sure many readers are interested in though is programming assistants, because we all know Googling everything is actually one of the worst user experiences. Being able to write a few objectives in plain English and see a copilot write the code for you is exhilarating. I've personally used these tools to help me remember syntax, simplify and clean code, write tests, and learn a new programming language.

Video Gaming is another interesting field where we can expect LLMs to create a lot of innovation. Not only do they help the programmers create the game, but they are allowing designers to create more immersive experiences. For example, talking to NPC's will have more depth and intriguing dialogue. Picture games like Animal Crossing or Stardew Valley having near infinite quests and conversations.

Consider other industries, like Education where there just doesn't seem to be enough teachers to go around meaning our kids aren't getting the one on one attention they need. An LLM assistant can help save the teacher time doing manual chores as well as serve as a private tutor for the kids that are struggling. Corporate is looking into LLMs for talk-to-your-data jobs, tasks helping employees understand quarterly reports and data tables, essentially giving everyone their own personal analyst. Sales and Marketing divisions are guaranteed to take advantage of this marvelous innovation, for better or worse. The state of Search Engine Optimization (SEO) will change a lot too since currently it is mostly a game of generating content to hopefully make websites more popular, which is now super easy.

That list is just a few of the common examples I've seen where companies are interested in using them. People are using them for personal reasons too. Writing music, poetry, and even books, translating languages, summarizing legal documents or emails, and even using them for free therapy—which yes, is an awful idea since they are still dreadful at this. Just a personal preference, but we wouldn't try to save a buck when our sanity is on the line. Of course, this leads us to the fact that people are already using them for darker purposes like cheating, scams, and fake news to skew elections. At this point, the list has become rather large and varied, but we've only begun to scratch the surface of the possible. Really, since LLMs help us with communication often, it's better to think, "What can't they do?" than "What can they do?".

Or better yet, "What shouldn't they do?" Well, as a technology there are certain restrictions and constraints, for example, LLMs are kind of slow. Of course, slow is a relative term, but responsive times are often measured in seconds not milliseconds. We'll dive deeper into this in Chapter 3, but as an

example, we probably won't see them being used in autocomplete tasks anytime soon which require blazingly fast inference in order to be useful. After all, autocomplete needs to be able to predict the word or phrase faster than someone types. In a similar fashion, LLMs are large complex systems, we don't need them for such a simple problem anyway. Hitting an autocomplete problem with an LLM isn't just hitting the nail with a sledgehammer, it's hitting it with a full on wrecking ball. And just like it's more expensive to rent a wrecking ball than buy a hammer, an LLM will cost you more to operate. There are a lot of similar tasks where we should consider the complexity of the problem we are trying to solve.

There are also many complex problems that are often poorly solved with LLMs such as predicting the future. No, we don't mean with mystic arts, but forecasting problems, acts like predicting the weather or when high tide will hit on the ocean shore. These are actually problems we've solved, but we don't necessarily have good ways to communicate how these have been solved. They are expressed through combinations of math solutions like Fourier transforms and harmonic analysis or through black box ML models. There are many problems that fit into this category, like outlier prediction, calculus, or finding the end of the roll of tape.

You also probably want to avoid using them for highly risky projects. LLMs aren't infallible and make mistakes often. To increase creativity we often allow for a bit of randomness in LLMs which means you can ask an LLM the same question and get different answers. That's risky. You can remove this randomness by doing what's called turning down the temperature, but that might make it useless depending on your needs. For example, you might decide to use an LLM to categorize investment options as good or bad, but do you want it to then make actual investment decisions based on its output? Not without oversight, unless your goal is to create a meme video.

Ultimately an LLM is just a model, it can't be held accountable for losing your money, and really it didn't lose your money you did by choosing to use it. Similar risky problems could include filling out tax forms or getting medical advice. While an LLM could do these things, it won't protect you from heavy penalties in an IRS audit like hiring a certified CPA would. If

you take bad medical advice from an LLM, there’s no doctor you could sue for malpractice. However, in all of these examples, the LLM could potentially largely help the practitioner better perform their job roles both reducing errors and improving speed.

When to use an LLM

Use them for:

- Generating content
- Question and answering services
- ChatBots and AI assistants
- Diffusion (txt2img, txt23d, txt2vid, etc.)
- Talk-to-your-data applications
- Anything that involves communication

Avoid using them for:

- Latency-sensitive workloads
- Simple projects
- Problems we don’t solve with words but with math or algorithm - forecasting, outlier prediction, calculus, etc.
- Critical evaluations
- High-risk projects

Language is not just a medium people use to communicate. It is the tool that made humans apex predators and gives every individual self-definition to their community. Every aspect of human existence, from arguing with your parents to graduating from college to reading this book is pervaded by our language. Language models are learning to harness one of the fundamental aspects of being human and have the ability, when used responsibly, to help us with each and every one of those tasks. They have the potential to unlock dimensions of understanding both of ourselves and others if we responsibly teach them how.

LLMs have captured the world's attention since their potential allows imaginations to run wild. LLMs promise so much, but… where are all these solutions? Where are the video games that give us immersive experiences? Why don't our kids have personal AI tutors yet? Why am I not Iron Man with my own personal assistant yet? These are the deep and profound questions that motivated us to write this book. Particularly that last one, it keeps me up at night. So while LLMs can do amazing things, not enough people know how to actually turn them into a product and that's what we aim to share in this book.


A COOL
LLM APP
PRODUCTION

This isn’t just a Machine Learning Operations book. There are a lot of gotchas and pitfalls involved with making an LLM work in production because LLMs just don’t work like traditional software solutions. To turn an LLM into a product that can interact coherently with your users will require an entire team and diverse set of skills. Depending on your use case you may need to train or finetune and then deploy your own model, or you may just need to access one from a vendor through an API.

Regardless of which LLM you use, if you want to take full advantage of the technology and build the best user experience, you will need to understand how they work. Not just on the math/tech side either, but on the soft side for how to make it a good experience for your users. In this book we’ll be covering everything you need to make LLMs work in production. We’ll talk about the best tools and infrastructure, how to maximize their utility with prompt engineering and other best practices like controlling costs. LLMs could be one step towards a greater equality, so if you are thinking, “I don’t feel like the person this book is for,” please reconsider. This book is for the whole team and anyone who will be interacting with LLMs in the future.

We’re going to hit on a practical level everything that you’ll need for collecting and creating a dataset, training or finetuning an LLM on consumer or industrial hardware, and deploying that model in various ways for customers to interact with. While we aren’t going to cover too much theory, we will be covering the process from end to end with real-world examples. At the end of this book you will know how to deploy LLMs with some viable experience to back it up.

1.2 Navigating between the Build and Buy decision with Large Language Models

If you bought this book, you are likely already convinced of the overwhelming potential LLMs can have in your life and in your organization. Buying this book then is the first step to turning your dreams into a reality, because none of it is possible until we know how to put these models into production. After all, if you talk to any entrepreneur or investor out there they will tell you good ideas are a dime a dozen, what matters is

execution and actually manifesting those ideas. What we need to do is get these models into production, where they are readily available to do actual work for you.

There's no getting around it and no need to sugar coat it either, deploying LLMs into production is hard. Often anything worth pursuing is. In this book we aim to teach you everything you need to know to do it as well as give you some practical hands-on experience. But because it is so hard, it is mighty tempting to take a shortcut. Large corporations like OpenAI and Google have some great offerings of models to choose from, why not just buy them? Let's start by considering what they offer and if this may be a good choice, and then we'll take a look at the other side of the coin where these offerings tend to fall flat.

1.2.1 Buying: the beaten path

There are many great reasons to simply just buy access to an LLM. First and foremost is the speed and flexibility accessing an API provides. Working with an API is an incredibly easy and cheap way to build a prototype and get your hands dirty quickly. In fact, so easy you can see in listing 1.1 that it only takes a few lines of code to start connecting to OpenAI's API and start using LLMs. Sure, there's a lot that's possible, but it would be a bad idea to heavily invest in LLMs only to find out they happen to fail in your specific domain. Working with an API allows you to fail fast. Building a prototype application to prove the concept and launching it with an API is a great place to get started.

Listing 1.1 A simple app calling OpenAI's API

```
import os
import openai

# Load your API key from an environment variable
openai.api_key = os.getenv("OPENAI_API_KEY")

chat_completion = openai.ChatCompletion.create(
    model="gpt-3.5-turbo",
```

```
    messages=[{"role": "user", "content": "Hello world"}],
)
```

Oftentimes, buying access to a model can give you a competitive edge. In many cases it could very well be that the best model on the market is built by a company specializing in the domain who are using specialized datasets they have spent a fortune to curate. While you could try to compete and build your own, it may better serve your purposes to simply buy access to their model instead. Ultimately, whoever has the better domain specific data to finetune on is likely going to win, and that might not be you if this is a side project for your company. Curating data can be expensive after all. It can save you a lot of work to go ahead and buy it.

Which leads to the next point, buying is a quick way to access expertise and support. For example, OpenAI has spent a lot of time making their models safe with plenty of filtering and controls to prevent the misuse of their LLMs. They've already encountered and covered a lot of the edge cases, so you don't have to. Buying access to their model also gives you access to the system they've built around it.

Not to mention, the LLM itself is only half the problem in deploying them to production. There's still an entire application you need to build on top of it. Sometimes buying OpenAI's model has thrived over its competitors in not a small part due to their UX and some tricks like making the tokens look like they're being typed. We'll take you through how you can start solving for the UX in your use case, along with some ways you can prototype to give you a major head start in this area.

1.2.2 Building: the path less traveled

Using an API is easy, and in most cases, likely the best choice. However, there are many reasons why you should aim to own this technology and learn how to deploy it yourself instead. While this path might be harder, we'll teach you how to do it. Let's dive into several of those reasons starting with the most obvious: Control.

Control

One of the first companies to truly adopt LLMs as a core technology is a small video game company called Latitude. Latitude specializes in Dungeon and Dragons like role playing games utilizing LLM chatbots, and they have faced challenges when working with them. This shouldn't come off as criticizing this company for their missteps, as they contributed to our collective learning experience and were pioneers forging a new path. Nonetheless, their story is a captivating and intriguing one, like a train wreck that we personally couldn't help but keep watching.

Latitude's first release was a game called AI Dungeon. At inception, it utilized OpenAI's GPT2 to create an interactive and dynamic storytelling experience. It quickly garnered a large gathering of players, who of course started to use it inappropriately. When OpenAI gave Latitude access to GPT3 it promised an upgrade to the gaming experience, instead, what it got was a nightmare.[2]

You see, with GPT3 they added Reinforcement Learning from Human Feedback (RLHF) which greatly helps improve functionality, but this also meant OpenAI contractors were now looking at the prompts. That's the human feedback part. And these workers weren't too thrilled to read the filth the game was creating. OpenAI's reps were quick to give Latitude an ultimatum. Either they needed to start censoring the players or they'd remove their access to the model—which would have essentially killed the game and the company. With no other option they quickly added some filters but the filtering system was too much a band-aid, a buggy and glitchy mess. Players were upset at how bad the system was and unnerved to realize Latitude's developers were reading their stories, completely oblivious to the fact that OpenAI was already doing such. It was a PR nightmare. And it wasn't over.

OpenAI decided the game studio wasn't doing enough, stonewalled, they were forced to increase their safeguards, and so they started banning players. Here's the twist, the reason so many of these stories turned to smut, was because the model had a preference for erotica. It would often unexpectedly transform harmless storylines into inappropriately risqué situations causing the player to be ejected and barred from the game. OpenAI was acting the

paragon of purity, but it was their model that was the problem. Which led to one of the most ironic and unjust problems in gaming history: players were getting banned for what the game did.

So there they were, a young game studio just trying to make a fun game stuck between upset customers and a tech giant that pushed all the blame and responsibility onto them. If the company had more control over the technology they could have gone after a real solution, like fixing the model. Instead of having to throw makeup on a pig.

In this example, control may come off as your ability to finetune your mode and OpenAI now offers finetuning capabilities, but there are many fine-grained decisions that are still lost by using a service instead of rolling your own solution. For example, what training methodologies are used, what regions the model is deployed too, or what infrastructure it runs on. Control is also important for any customer or internal-facing tool. You don’t want to have a code generator accidentally output code that’s copyrighted or that creates a legal situation for your company. You also don’t want your customer-facing LLM to output factually incorrect information about your company or its processes.

Control is your ability to direct and manage the operations, processes, and resources in a way that aligns with your goals, objectives, and values. If a model ends up becoming central to your product offering and the vendor unexpectedly raises their prices there’s little you can do but pay it. If the vendor decides their model should give more liberal or conservative answers that no longer align with your values, you are just as stuck.

The more central a technology is to your business plan, the more important it is to control it. This is why McDonald's owns the real estate for its franchises and why Google, Microsoft, and Amazon all own their own cloud networks. Or even why so many entrepreneurs build online stores through Shopify versus just using other platforms like Etsy or Amazon Marketplace. Ultimately control is the first thing that’s lost when you buy someone else’s product. Keeping it will give you more options to solve future problems and will also give you a competitive edge.

Competitive edge

One of the most valuable aspects of deploying your own models is the competitive edge it gives you over your competition. Customization - train the model to be the best at one thing. For example, after the release of Bidirectional Encoder Representations from Transformers (BERT) in 2017, which is a transformer model architecture you could use to train your own model, there was a surge of researchers and businesses testing this newfound technology on their own data to worldwide success. At the time of writing, if you search the Hugging Face Hub for “BERT,” there are more than 13.7k models returned, all that people trained individually for their own purposes to be the best model for their task.

One of my personal experiences in this area was training SlovenBERTcina. After aggregating the largest (at-the-time) monolingual Slovak language dataset by scraping the Slovak National Corpus with permission, along with a whole bunch of other resources like the OSCAR project and the Europarl corpus. It never set any computational records, and has never appeared in any model reviews or generated partnerships for the company I worked for. It did, however, outperform every other model on the market on the tasks it trained on.

Chances are, neither you nor your company need AGI to generate relevant insights from your data, and in fact if you invented an actual self-aware AGI and planned to only ever use it to crunch some numbers, analyze data and generate visuals for PowerPoint slides once a week, that would definitely be reason enough for the AGI to eradicate humans. More than likely, you need exactly what I did when I made SlovenBERTcina, a large language model that performs the two to three tasks you need better than any other model on the market and doesn’t also share your data with Microsoft or other potential competitors. While some data is required to keep secret for security or legal reasons, a lot of data should be guarded simply because they are trade secrets.

There are hundreds of open source LLMs both for general intelligence, and for foundational expertise on a specific task. We’ll hit some of our favorites

in Chapter 4. Taking one of these open source alternatives and training it on your data to create a model that is the best in the world at that task will ensure you have a competitive edge in your market. It will also allow you to deploy the model your way and integrate it into your system to make the most impact.

Integrate anywhere

Let's say you want to deploy an LLM as part of a choose your own adventure style game that uses a device's GPS location to determine story plots. You know your users are often going to go on adventures into the mountains, out at sea, and generally to locations where they are likely to experience poor service and lack of internet access. Hitting an API just isn't going to work. Now, don't get me wrong, deploying LLMs onto edge devices like in this scenario is still an exploratory subject, but it is possible, and we will be showing you how in chapter 9. Relying upon an API service is just not going to work for immersive experiences.

Similarly, using third party LLM and hitting an API adds integration and latency issues, requiring you to send data over the wire and wait for a response. APIs are great, but they are always slow and not always reliable. When latency is important to a project it's much better to serve the service in-house. The previous section on Competitive Edge discussed two projects with edge computing as a priority, however many more exist. LLAMA.cpp and ALPACA.cpp are two of the first of such projects, and this space is innovating quicker than any others. Quantization into 4-bit, Low-Rank Adaptation, and Parameter Efficient Finetuning are all methodologies recently created just to meet these needs, and we'll be going over each of these starting in Chapter 3.

When my team first started integrating with ChatGPT's API, it was both an awe-inspiring and humbling experience. Awe-inspiring because it allowed us to quickly build some valuable tools. Humbling because as one engineer joked to me, "When you hit the end point you will get 503 errors, sometimes you get a text response as if the model was generating text, but I think that's a bug." Serving an LLM in a production environment, trying to meet the

needs of so many clients, is no easy feat. However, deploying a model that’s integrated into your system allows you more control of the process affording higher availability and maintainability than you can currently find on the market.

Costs

Considering costs is always important because it plays a pivotal role in making informed decisions and ensuring the financial health of a project or an organization. It helps you manage budgets efficiently and make sure that resources are allocated appropriately. Keeping costs under control allows you to maintain the viability and sustainability of your endeavors in the long run.

Additionally, considering costs is crucial for risk management. When you understand the different cost aspects, you can identify potential risks and exert better control over them. This way, you can avoid unnecessary expenditures and ensure that your projects are more resilient to unexpected changes in the market or industry.

Finally, cost considerations are important for maintaining transparency and accountability. By monitoring and disclosing costs, organizations demonstrate their commitment to ethical and efficient operations to stakeholders, clients, and employees. This transparency can improve an organization's reputation and help build trust.

All of these apply as you consider building versus buying LLMs. It may seem immediately less costly to buy, as the most costly service widely used on the market currently is only $20 USD per month. Compared to an EC2 instance on AWS, just running that same model for inference (not even training) could run you up a bill for about $250k USD per year. This is where building has done its quickest innovation, however. If all you need is an LLM for a proof of concept, any of the projects mentioned in the Competitive Edge section will allow you to create a demo for only the cost of electricity to run on the computer you are demoing on, and spell out training easily enough to allow for significantly reduced costs to train a

model on your own data, as low as $100 (yes, that’s the real number) for a model with 20 billion parameters. Another benefit is knowing that if you build your own, your cost will never go up, like it very much will when paying for a service.

Security and privacy

Consider the following case. You are a military staff member in charge of maintenance for the nuclear warheads in your arsenal. All the documentation is kept in a hefty manual. There’s so much information required to outline all the safety requirements and maintenance protocols cadets are known to forget important information despite their best efforts. They often cut the wires before first removing the fuse[3]. You decide to fine-tune an LLM model to be a personal assistant, giving directions and helping condense all that information giving soldiers exactly what they need when they need it. It’s probably not a good idea to upload those manuals to another company–understatement of the century–you're going to want to train something locally that’s kept secure and private.

This scenario may sound farfetched, but when speaking to an expert working in analytics for a police department they echoed this exact concern. Talking with them, they expressed how cool ChatGPT is even having their whole team take a prompt engineering class to better take advantage of it, but lamented that there was no way for his team to use it for their most valuable work–the sort of work that literally saves lives–without exposing sensitive data and conversations. Anyone in similar shoes should be eager to learn how to deploy a model safely and securely.

You don’t have to be in the army or police force to handle sensitive data. Every company has important intellectual property and trade secrets that are best to keep a secret. Having worked in the semiconductor, healthcare, and finance industries we can tell you first hand, paranoia and corporate espionage are part of the culture in these industries. Because of this Samsung and other industry players at first locked down ChatGPT preventing employees from using it, later opening it up. Of course, it didn’t take long before several Samsung employees leaked confidential source code[4].

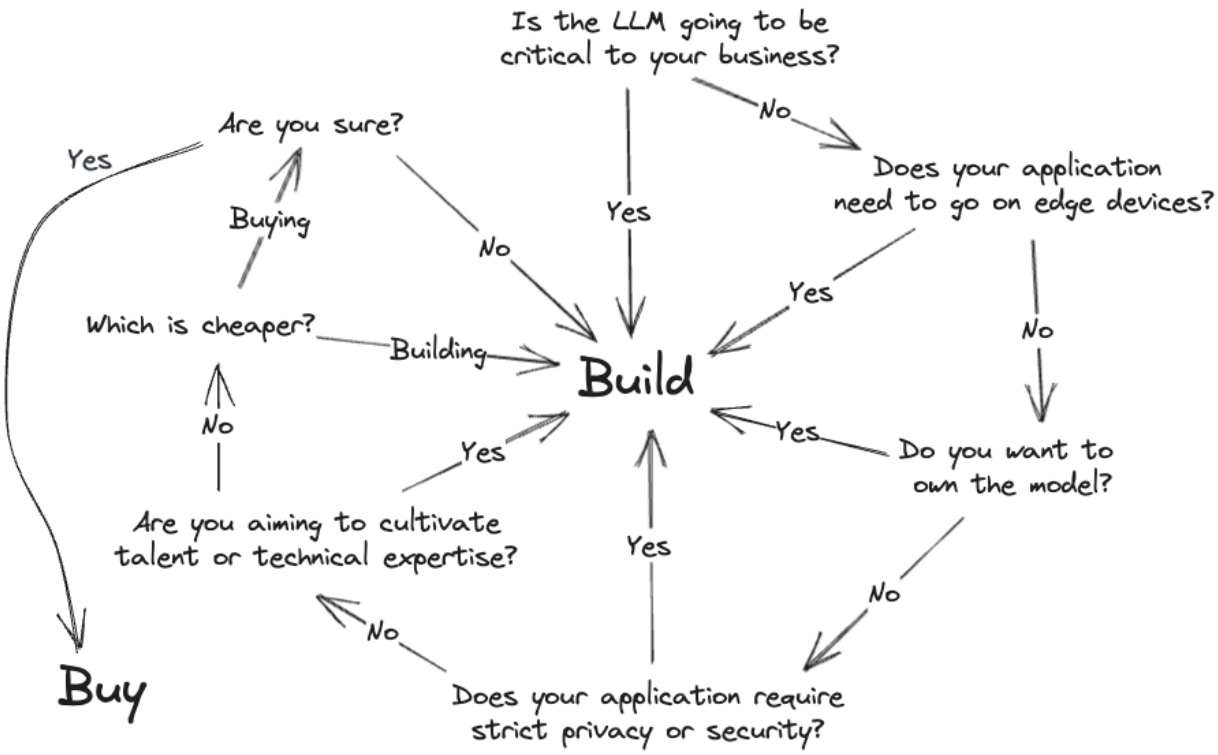
Because OpenAI uses RLHF, that code is retained and used to further train the model later on. Meaning with the right prompt injection, anyone could potentially pull the code out of the model.

It's not just code that can easily be lost. Business plans, meeting notes, confidential emails and even potential patent ideas are at risk. We unfortunately know of a few companies who have started sending confidential data to ChatGPT, using that model to clean and extract PII. If this strikes you as potential negligent misuse, you'd be right. This methodology directly exposes customer data, not just to Microsoft/OpenAI, but to any and all 3rd-party services that they use (including AWS Mechanical Turk, Fiverr, and Freelance workers) to perform the Human Feedback part of RLHF.

Wrapping up

As you can see, there are lots of reasons why a company might want to own and build their own LLMs including having greater control, cutting costs, and meeting security and regulation requirements. Despite this we understand that buying is easy, and building is much more difficult so for many projects it makes sense to just buy, but before you do in figure 1.1 we share a flowchart of questions you should ask yourself first. Even though it's the more difficult path, building can be much more rewarding.

Figure 1.1 Questions you should ask yourself before making that build vs buy decision.



One last point we think these build versus buy conversations never seem to hone in on enough is, “Porque no los dos?” Buying gets you all the things building is bad at: time-to-market, relatively low cost, and ease-of use. Building gets you all the things buying struggles with: privacy, control, and flexibility. Research and prototyping phases could very much benefit from buying a subscription to GPT4 or Databricks for building something quick to help raise funding or get stakeholder buy-in. Production however, often isn’t an environment that lends itself to third-party solutions well.

Ultimately, whether or not you plan to build or buy we wrote this book for you. Obviously if you plan to build it, there’s going to be a lot more you need to know about, so a majority of this book will be geared to these folks. In fact, we don’t need to belabor the point anymore, we’re going to teach you how to build in this book, but don’t let that stop you from doing the right thing for your company.

1.2.3 A word of warning: embrace the future now

New technologies are like fire, they can provide warmth on cold nights and help cook our food, but they can also burn our homes and hurt us. In business, there's no shortage of stories of companies failing because they failed to adapt to new technologies. We can learn a lot from their failures.

Borders Books first opened its doors in 1971. After developing a comprehensive inventory management system that included advanced analytic capabilities it skyrocketed to become the second-largest Book retailer in the world, only behind Barnes & Noble. Using this new technology it disrupted the industry, allowing it to easily keep track of tens of thousands of books, opening large stores where patrons could peruse many more books than they could at smaller stores. The analytic capabilities helped it track which books were gaining popularity and gain better insights into their customers allowing it to make better business decisions. It dominated the industry for over two decades.

Borders however, failed to learn from its own history, going bankrupt in 2011, failing to adapt and being disrupted by technology this time; e-commerce. In 2001, instead of building their own platform and online store, they decided to outsource their online sales to Amazon.[5] A decision many critics would say was akin to giving your competitors the key to your business. While not exactly handing over their secret sauce, it was a decision that gave up their competitive edge.

For the next seven years they turned a blind eye to the growing online sector instead focusing on expanding their physical store presence, buying out competitors and securing a coveted Starbucks deal. When Amazon released the Kindle in 2007, the book retail landscape completely changed. Barnes & Noble having run their own online store quickly pivoted and released the Nook to compete, Borders however, did nothing, or in fact, could do nothing.

By embracing e-commerce through a third party, they failed to develop the in-house expertise required to create a successful online sales strategy, leading to a substantial loss in market share. They eventually launched their own e-reader, Kobo, in late 2010, but it was too late to catch up. Their inability to fully understand and implement e-commerce technology

effectively led to massive financial losses, store closures, and ultimately, the company filed for bankruptcy in 2011.

Borders Books is a cautionary tale, but there are hundreds more of similar companies who failed to adopt new technology to their own detriment. With a new technology as impactful as LLMs each company has to decide on which side of the fence they want to be on. Do they delegate implementation and deployment to large FAANG like corporations relegating to just hitting an API, or do they take charge preferring to master the technology and deploy it themselves?

The biggest lesson we hope to impart from this story is that technologies build on top of one another. Ecommerce was built on top of the internet. Failing to build their own online store meant Borders failed to build the in-house technical expertise they needed to stay in the game when the landscape shifted. We see the same things with LLMs today because the companies that are best prepared to utilize them have already gathered expertise in machine learning and data science and have some idea of what they are doing.

We don’t have a crystal ball that tells us the future, but many believe that LLMs are a revolutionary new technology like the internet or electricity before it. Learning how to deploy these models, or failing to do so, may very well be the defining moment for many companies. Not because doing so will make or break their company now, but in the future, when something even more valuable comes along that’s built on top of LLMs.

Foraying into this new world of deploying LLMs may be challenging but will help your company build the technical expertise to stay on top of the game. No one really knows where this technology will lead, but learning about this technology will likely be necessary to avoid mistakes like Borders Books’.

There are many great reasons to buy your way to success, but there is at least one prevalent thought that is just absolutely wrong. It’s the myth that only large corporations can work in this field because it takes millions of dollars and thousands of GPUs to train these models. Creating this impenetrable

moat of cash and resources the little guy can’t hope to cross. We’ll be talking about this more in the next section, but any company of any size can get started and there’s no better time than now to do so.

1.3 Debunking myths

We have all heard from large corporations and the current leaders in LLMs how incredibly difficult it is to train an LLM from scratch and how intense it is to try to finetune them. Whether from OpenAI, BigScience, or Google they discuss large investments and the need for strong data and engineering talent. But how much of this is true and how much of it is just a corporate attempt to create a technical moat?

Most of these barriers start with the premise that you will need to train an LLM from scratch if you hope to solve your problems. Simply put, you don’t! Open source models covering many dimensions of language models are constantly being released, so you more-than-likely don’t need to start from scratch. While what they say is true that training LLMs from scratch is supremely difficult, we are still constantly learning about how to do it and are able to more and more automate the repeatable portions. In addition, since this is an active field of research frameworks and libraries are being released or updated daily and will help you start from wherever you currently are. Frameworks like oobabooga’s Gradio will help you run LLMs and base models like Falcon 40B will be your starting point. All of it is covered. To add to this, memos have circulated at large companies addressing the lack of a competitive edge that any organization currently holds over the open source community at large.

A friend once confided in me that, “I really want to get more involved in all this machine learning and data science stuff. It seems to be getting cooler every time I blink an eye. However, it feels like the only way to get involved is to go through a lengthy career change and go work for a FAANG. No, thank you. I’ve done my time at large companies, and they aren’t for me. But I hate feeling like I’m trapped on the outside.” This is the myth that inspired this book. We’re here to equip you with tools and examples to help you to stop feeling trapped on the outside. We’ll help you go through the language

problems that we're trying to solve with LLMs, along with machine learning operation strategies to account for the sheer size of the models.

Oddly enough, as many believe they are trapped on the outside, many others believe they can become experts in a weekend. Just get a GPT API key and that's it, you're done. This has led to a lot of fervor and hype with a cool new demo popping up on social media every day. But, most of these demos never become actual products and not because people don't want them.

To understand this, let's discuss IBM's Watson the world's most advanced language model before GPT. Watson is a question and answering machine that went on to crush Jeopardy in 2011 against some of the best human contestants to ever appear on the show, Brad Rutter and Ken Jennings. Rutter the highest earning contestant ever to play the game show and Jennings a player so good he went on to win a whopping 74 times in a row. Despite facing these legends, it wasn't even close. Watson won in a landslide. Jennings in response to the loss responded with the famous quote, "I, for one, welcome our new computer overlords."[6]

Watson was the first impressive foray into language modeling and many companies were clamoring to take advantage of its capabilities. Starting in 2013, Watson started being released for commercial use. One of the biggest applications involved many attempts trying to integrate it into healthcare to solve various problems. However, none of these solutions ever really worked the way they needed to and the business never became profitable. By 2022 Watson Health was sold off.

What we find when solving language related problems is that building a prototype is easy, building a functioning product on the other hand is very, *very* difficult. There are just too many nuances to language. Many people wonder what made ChatGPT so explosive? Gaining over a million clients in just five days. Most of the answers I've heard would never satisfy an expert because ChatGPT wasn't much more impressive than GPT3 or other LLMs which had been around for several years already. I heard Sam Altman of OpenAI himself say in an interview they didn't think ChatGPT would get this much attention, they thought that would come with GPT4's release.[7] So why was it explosive? The magic in our opinion, is that it was the first

product to truly productionize LLMs. To turn it from a demo into an actual product. It was something anyone could interact with and ask it tough questions, only to be amazed by how well it responded. A demo only has to work once, but the product has to work every time and even when millions of users are showing it to their friends saying, “Check this out!” That magic is exactly what you can hope to learn from reading this book.

We’re excited about writing this book. We are excited about the possibilities of bringing this magic to you so you can bring it to the world. LLMs are at the intersection between so many fields such as linguistics, mathematics, computer science and more. While the more you know will help you, to be an expert isn’t required. Expertise in any of the individual parts only raises the skill ceiling, not the floor to get in. Consider an expert in physics or music theory, they won’t automatically have the skills for music production, but they will be more prepared to quickly learn. LLMs are a communication tool and communicating is a skill just about everyone needs.

Like all other skills, your proximity and willingness to get involved are the two main blockers to knowledge, not a degree or ability to notate—these only shorten your journey towards being heard and understood. If you don’t have any experience in this area, it might be good to start by just developing an intuition around what an LLM is and needs by going and contributing to a project like OpenAssistant. If you’re a human, that’s exactly what they need. By volunteering, you can start understanding exactly what these models train on and why. If you fall anywhere from no knowledge up to being a professional machine learning engineer, we’ll be imparting the knowledge necessary to shorten your conversations along with your time to understanding considerably. If you’re not interested in learning the theoretical underpinnings of the subject, we’ve got plenty of hands-on examples and projects to get your hands dirty.

We’ve all heard a story by now of LLM hallucinations, but LLMs don’t need to be erratic. Companies like Lakera are working daily to improve security, while others like LangChain are making it easier to provide models with pragmatic context which makes them more consistent and less likely to deviate. Techniques such as RLHF and Chain of Thought further allow our

models to align themselves with negotiations we've already accepted as people and models should understand from the get-go, such as basic addition or the current date, both of which are conceptually arbitrary. We'll help you increase your model stability from a linguistic perspective, so they'll figure out not just what are the most likely outputs, but the most useful.

Something to consider as you venture further down this path is not just the security of what goes into your model/code, but what comes out. LLMs can sometimes produce outdated, factually incorrect, or even copyrighted or licensed material, depending on what its training data contained. LLMs are unaware of any agreements people make about what is supposed to be a trade secret and what can be shared openly. That is, unless you tell it about those agreements during training or through careful prompting mechanisms during inference. Indeed, the challenges around prompt injection giving inaccurate information primarily arise due to two factors: firstly, users requesting information beyond the model's understanding; and secondly, the model developers not fully predicting how users will interact with the models or the nature of their inquiries. If you had a resource that could help you get a head start on that second problem, it would be pretty close to invaluable, wouldn't it?

Lastly, we don't want to artificially or untruthfully inflate your sense of hope with LLMs. They are resource intensive to train and run. They are hard to understand, and they are harder to get working how you want. They are new and not well-understood. The good news is that these problems are being actively worked on, and we've put in a lot of work finding implementations that are concurrent with writing and actively lessen the burden on you to know everything about the entire deep learning architecture. From quantization to Kubernetes, we'll help you figure out everything you need to know to do this now with what you have. Maybe we'll inadvertently convince you that it's too much, and you should just purchase from a vendor. Either way, we'll help you every step of the way to help you get the results you need from this magical technology.

1.4 Summary

- LLMs are exciting because they work with humans instead of against them
- Society has been built on language, so effective language models have limitless applications such as chatbots, programming assistants, video games, and AI assistants.
- LLMs are excellent at many tasks and can even pass high-ranking medical and law exams
- LLMs are wrecking balls not hammers, and should be avoided for simple problems, problems that require low latency, and problems with high risks.
- Reasons to buy include:
    - Quickly get up and running to conduct research and prototype use cases
    - Easy access to highly optimized production models
    - Access to vendors technical support and system
- Reasons to build include:
    - Getting a competitive edge for your business use case
    - Keeping costs low and transparent
    - Ensuring reliability of the model
    - Keeping your data safe
    - Controlling model output on sensitive or private topics
- There is no technical moat that is preventing you from competing with larger companies since open source frameworks and models provide the building blocks to pave your own path.

[1] Med-PaLM 2 has scored an 86.5% on the MedQA exam.

[2] WIRED, “It began as an AI-fueled dungeon game. Then it got much darker,” Ars Technica, May 08, 2021. https://arstechnica.com/gaming/2021/05/it-began-as-an-ai-fueled-dungeon-game-then-it-got-much-darker/

[3] M*A*S*H reference for those wondering: https://youtu.be/UcaWQZlPXgQ

[4] □□□□□□, “[□□] □□□ □□□…□□□□, □GPT □□ □□□□ ‘□□□’ □□,” □□□□ □□, Mar. 30, 2023. https://economist.co.kr/article/view/ecn202303300057?

s=31

[5] A. Lowrey, “Borders bankruptcy: Done in by its own stupidity, not the Internet.,” Slate Magazine, Jul. 20, 2011. https://slate.com/business/2011/07/borders-bankruptcy-done-in-by-its-own-stupidity-not-the-internet.html

[6] J. Best, “IBM Watson: The inside story of how the Jeopardy-winning supercomputer was born, and what it wants to do next,” TechRepublic, Sep. 09, 2013. https://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy-winning-supercomputer-was-born-and-what-it-wants-to-do-next/

[7] “A conversation with OpenAI CEO Sam Altman | Hosted by Elevate,” May 18, 2023 https://youtu.be/uRIWgbvouEw.

2 Large Language Models: A Deep Dive Into Language Modeling

This chapter covers

- Linguistic background for understanding meaning and interpretation
- A comparative study on language modeling techniques
- Attention and the transformer architecture
- How Large Language Models both fit into and build upon these histories

All good stories start with “Once upon a time,” but unfortunately this isn’t a story, it’s a book on creating and productionizing LLMs. So instead this chapter delves into linguistics as it relates to the development of Large Language Models (LLMs), exploring the foundations of semiotics, linguistic features, and the progression of language modeling techniques that have shaped the field of natural language processing (NLP). We will begin by studying the basics of linguistics and its relevance to LLMs in section 2.1, highlighting key concepts such as syntax, semantics, and pragmatics, that form the basis of natural language and play a crucial role in the functioning of LLMs. We will delve into semiotics, the study of signs and symbols, and explore how its principles have informed the design and interpretation of LLMs.

We will then trace the evolution of language modeling techniques in section 2.2, providing an overview of early approaches, including N-grams, Naive Bayes classifiers, and neural network-based methods such as Multi-Layer Perceptrons (MLPs), Recurrent Neural Networks (RNNs), and Long Short-Term Memory (LSTM) networks. We will also discuss the groundbreaking shift to transformer-based models in section 2.3, which have laid the foundation for the emergence of LLMs—which are just really big transformer based models. Finally, we will introduce LLMs in 2.4 and their distinguishing features, discussing how they have built upon and surpassed

earlier language modeling techniques to revolutionize the field of Natural Language Processing (NLP).

This is a book about LLMs in production. We firmly believe that if you want to turn an LLM into an actual product, understanding the technology better will improve your results and save you from making costly and time-consuming mistakes. Any engineer can figure out how to lug a big model into production and throw a ton of resources at it to make it run, but that brain-dead strategy completely misses the lessons people have already learned trying to do the same thing before, which is what we are trying to solve with LLMs in the first place. Having a grasp of these fundamentals will better prepare you for the tricky parts, the gotchas, and the edge cases you are going to run into when working with LLMs. By understanding the context in which LLMs emerged, we can appreciate their transformative impact on NLP and how to enable them to create a myriad of applications.

2.1 Language Modeling

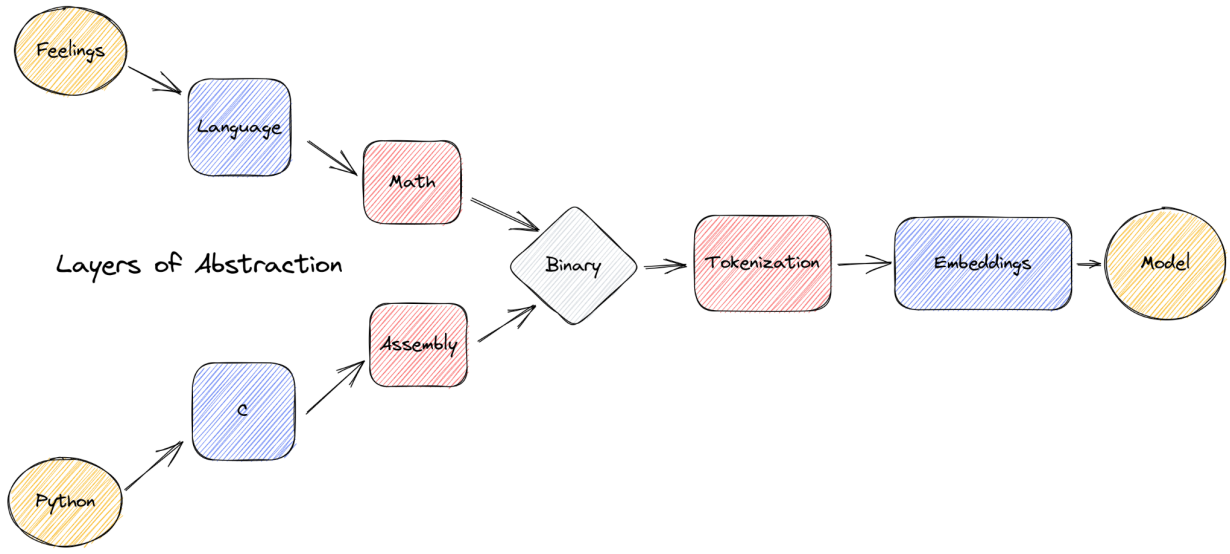
It would be a great disservice to address LLMs in any depth without first addressing language, to begin with. To that end, we will start with a brief but comprehensive overview of language modelling, focusing on the lessons that can help us with modern LLMs. Let’s first discuss levels of abstraction as it will help us garner an appreciation for language modelling.

Language, as a concept, is an abstraction of the feelings and thoughts that occur to us in our heads. Likewise, math is an abstraction of language, focusing on logic and provability, but as any mathematician will tell you, it is a subset of language used to describe and define in an organized and “logical” way. From math comes the language of binary, a base-2 system of numerical notation consisting of either on or off.

Binary is very useful abstraction of math, which is an abstraction of language which is an abstraction of our feelings because it is what is used underneath the hood for everything to do with software, models, and computers in general. When computers were first made, we communicated with them through punched cards, or binary directly. Unfortunately, this ends

up taking too long for humans to communicate important things in, so binary was also abstracted to assembly, a more human-comprehensible language for communicating with computers. This was further abstracted to the high-level assembly language, C, which has been even further abstracted to object-oriented languages like Python. The flow we just discussed is outlined in Figure 2.1.

Figure 2.1 We compare cognitive layers of abstraction to programming layers of abstraction down to the logical binary abstraction. Python doesn't come from C, nor does it compile into C. Python is, however, another layer of abstraction distant from binary. Similarly, language follows a similar path. Each layer of abstraction creates a potential point of failure. There are also several layers of abstraction to creating a model, each of which are important.



This is obviously a reduction, however, it's useful to understand that the feelings you have in your head are the same number of abstractions away from binary, the language the computer actually reads, as the languages most people use to program in. Some people might argue that there are more steps between Python and binary, such as compilers or using assembly to support the C language, and that's true, but there are more steps on the language side too, such as morphology, syntax, logic, and agreement.

This can help us understand how difficult the process of getting what we want to be understood by an LLM actually is, and even help us understand

language modeling techniques better. The reason we focus on binary here is simply to illustrate that there are a similar number of abstract layers to get from an idea you have or from one of our code samples to a working model. Like the children's telephone game where participants whisper into each other's ears, each abstraction layer creates a disconnect point or barrier where mistakes can be made.

Figure 2.1 is also meant to illustrate the difficulty in not only creating reliable code and language input, but to draw attention to how important the intermediary abstraction steps like tokenization and embeddings are for the model itself. Even if you have perfectly reliable code and perfectly expressed ideas, the meaning may be fumbled by one of those processes before it ever reaches the LLM.

In this chapter we will try and help you understand what you can do to reduce the risks of these failure points, whether that be on the language, coding, or modeling side. Unfortunately, it's a bit tricky to strike a balance between giving you too much linguistics that doesn't immediately matter for the task at hand versus giving you too much technical knowledge that, while useful, doesn't help you develop an intuition for language modeling as a practice. With this in mind, you should know that linguistics can be traced thousands of years back in our history and there's lots to learn from it. We've included a ***brief*** overview for interested readers of how language modeling has progressed over time in Appendix A answer encourage you to take a look.

Let's start with our focus on the building blocks that constitute language itself. We expect our readers to have at least attempted language modeling before and to maybe have heard of libraries like PyTorch and Tensorflow, but we do not expect most of our readers to have considered the language side of things before. By understanding the essential features that make up language, we can better appreciate the complexities involved in creating effective language models and how these features interact with one another to form the intricate web of communication that connects us all. In the following section, we will examine the various components of language, such as phonetics, pragmatics, morphology, syntax, and semantics, as well as

the role they play in shaping our understanding and usage of languages around the world. Let's take a moment to explore how we currently understand language along with the challenges we face that LLMs are meant to solve.

2.1.1 Linguistic Features

Our current understanding of language is that language is made up of at least 5 parts: Phonetics, Syntax, Semantics, Pragmatics, and Morphology. Each of these portions contributes significantly to the overall experience and meaning being ingested by the listener in any conversation. Not all of our communication uses all of these forms, for example, the book you're currently reading is devoid of phonetics, which is one of the reasons why so many people think text messages are unsuited for a more serious or complex conversation. Let's work through what each of these is to figure out how to present them to a language model for a full range of communicative power.

Phonetics

Probably the easiest for a language model to ingest, phonetics involves the actual sound of the language. This is where accent manifests and deals with the production and perception of speech sounds, with phonology focusing on the way sounds are organized within a particular language system. Similarly to computer vision, while a sound isn't something necessarily easy to deal with as a whole, there's no ambiguity for how to parse, vectorize, or tokenize the actual sound waves. They have a numerical value attached to each part, the crest, the trough, and the slope during each frequency cycle. It is vastly easier than text to be tokenized and processed by a computer while being no less complex. Sound inherently contains more encoded meaning than the text as well, for example, imagine someone saying the words "yeah, right," to you. Could be sarcastic, could be congratulatory, depending on the tone and English isn't even tonal! Phonetics, unfortunately, doesn't have Terabyte-sized datasets commonly associated with it, and performing data acquisition and cleaning on phonetic data, especially on the scale needed to train an LLM is difficult at best. In an alternate world where audio data was more prevalent than text data, and took up a smaller memory footprint,

phonetic-based or phonetic-aware LLMs would be much more sophisticated, and creating that world is a solid goal to work towards.

Anticipating this problem, a system created in 1888 called the International Phonetic Alphabet (IPA) has been revised in both the 20th and 21st centuries to be more concise, more consistent, and more clear, which could be a way to insert phonetic awareness into text data. IPA functions as an internationally standardized version of every language's sound profile. A sound profile is the set of sounds that a language uses, for example in English, we never have the /ʃ/ (she, shirt, sh) next to the /v/ sound. IPA is used to write sounds, rather than writing an alphabet or logograms, as most languages do. For example, you could simply describe how to pronounce the word "cat" using these symbols: /k/, /æ/, and /t/. That's of course a *very* simplified version of it, but for models it doesn't have to be. You can describe tone and aspiration as well. This could be a happy medium between text and speech, capturing some phonetic information. Think of the phrase "what's up?" Your pronunciation and tone can drastically change how you understand that phrase, sometimes sounding like a friendly "wazuuuuup," and other an almost threatening, "'sup," which IPA would fully capture. IPA isn't a perfect solution though, for example, it doesn't solve the problem of replicating tone very well.

Phonetics is listed first here because it's the place that LLMs have been applied to the least out of all the features and therefore have the largest space for improvement. Even modern TTS and Voice cloning models for the most part end up converting the sound to a spectrogram and analyzing that image rather than incorporating any type of phonetic language modeling. This is something to look for as far as research goes in the coming months and years.

Syntax

This is the place where current LLMs are highest-performing, both in parsing syntax from the user and generating its own. Syntax is generally what we think of as grammar and word order, and is the study of how words can combine to form phrases, clauses, and sentences. Syntax is also the first

place that language learning programs start to help people acquire new languages, especially based on where you're coming from natively. For example, it is important for a native English speaker learning Turkish to know that the syntax is completely different, and you can often build entire sentences in Turkish that are just one long compound word, whereas in English, we never put our subject and verb together into one word.

Syntax is largely separate from meaning in language, as the famous sentence from Noam Chomsky the so-called father of syntax demonstrates: "Colorless green ideas sleep furiously." Everything about that sentence is both grammatically correct and semantically understandable. The problem isn't that it doesn't make sense, it's that it does, and the encoded meanings of those words conflict. This is a reduction, however, you can think of all the times LLMs give nonsense answers as this phenomenon manifesting. Unfortunately for us, the syntax is also where ambiguity is the most commonly found. Consider the sentence, "I saw an old man and woman." Now answer the question: is the woman also old? This is syntactic ambiguity, where we aren't sure whether the modifier "old" applies to all people in the following phrase or just the one it immediately precedes. This is less consequential than the fact that semantic and pragmatic ambiguity also show up in syntax. Consider this sentence now, "I saw a man on a hill with a telescope," and answer the question: Where is the speaker, and what are they doing? Is the speaker on the hill cutting a man in half using a telescope? Likely, you didn't even consider this option when you read the sentence, because when we interpret syntax, all of our interpretations are at least semantically and pragmatically informed. We know from lived experience that that interpretation isn't at all likely, so we throw it out immediately, usually without even taking time to process that we're eliminating it from the pool of probable meanings. Think about this later as we're doing projects with LLMs.

It shouldn't take any logical leap for why LLMs need to be syntax-aware in order to be high-performing. LLMs that don't get word order correct or generate nonsense aren't usually described as "good." LLMs being syntax-dependent is something that has prompted even, Chomsky to call LLMs "stochastic parrots." In the authors' opinions, GPT2 in 2018 was when

language modeling solved syntax as a completely meaning-independent demonstration, and we've been happy to see the more recent attempts to combine the syntax that GPT2 output so well with encoded and entailed meaning, which we'll get into now.

Semantics

Semantics are the literal encoded meaning of words in utterances. People automatically optimize semantic meaning, only using words that they consider meaningful in the current language epoch. If you've ever created or used an embedding with language models (word2vec, ELMO, BERT [the E is for Embedding], MUSE, etc.]) you've used a semantic approximation. Words often go through semantic shifts, and while we won't cover all of this topic nor go in-depth, here are some common ones you may already be familiar with: narrowing, a broader meaning to a more specific one, broadening, the inverse of narrowing going from a specific meaning to a broad one, and reinterpretations, going through whole or partial transformations. These shifts do not have some grand logical underpinning. They don't even have to correlate with reality, nor do speakers of a language hardly ever consciously think about the changes as they're happening. That doesn't stop the change from occurring, and in the context of language modeling it doesn't stop us from having to keep up with that change.

Some examples: narrowing includes "deer" which in Old and Middle English just meant any wild animal, even a bear or a cougar, and now means only one kind of forest animal. For broadening we have "dog" which used to refer to only one canine breed from England, and now can be used to refer to any domesticated canine. One fun tangent about dog-broadening is in the *FromSoft* game *Elden Ring,* where because of a limited message system between players, they will use "dog" to refer to anything from a turtle to a giant spider and literally everything in between. For reinterpretation, we can consider "pretty" which used to mean clever or well-crafted, not visually attractive. Another good example is "bikini" which went from referring to a particular atoll, to referring to clothing you might have worn when visiting that atoll to people acting as if the "bi-" was referring to the two-piece structure of the clothing, thus inventing the tankini and monokini. Based on

expert research and decades of study, we can think of language as being constantly compared and reevaluated by native language speakers out of which common patterns emerge. The spread of those patterns is closely studied in sociolinguistics and is largely out-of-scope for the current purpose, but we encourage the reader to look into it if interested, as sociolinguistic phenomena such as prestige can help in designing systems that work well for everyone.

In the context of LLMs, so-called semantic embeddings are vectorized versions of text that attempt to mimic semantic meaning. The most popular way of doing this currently is by tokenizing or assigning an arbitrary number in a dictionary to each subword in an utterance (think prefixes, suffixes, and morphemes generally), applying a continuous language model to increase the dimensionality of each token within the vector so that there's a larger vector representing each index of the tokenized vector, then applying a positional encoding to each of those vectors to capture word order. Each subword ends up being compared to other words in the larger dictionary based on how it's used. We'll show an example of this later. Something to consider when thinking about word embeddings is that they struggle to capture deep encoded meaning of those tokens, and simply adding more dimensions to the embeddings hasn't shown marked improvement. One evidence that embeddings are working in a similar way to humans is that you can apply a distance function to related words and see that they are closer together than unrelated words. This is another area to expect groundbreaking research in the coming years and months for how to capture and represent meaning more completely.

Pragmatics

Sometimes omitted from linguistics, due to its referent being all the non-linguistic context affecting a listener's interpretation and the speaker's decision to express things in a certain way. Pragmatics refers in a large part to dogmas followed in cultures, regions, socio-economic classes, and shared lived experiences played off of to take shortcuts in conversations using entailment.

If I were to say, “A popular celebrity was just taken into the ICU,” your pragmatic interpretation based on lived experience might be to assume that a well-beloved person has been badly injured and is now undergoing medical treatment in a well-equipped hospital. You may wonder about which celebrity it is, whether they will have to pay for the medical bills, or if the injury was self-inflicted, also based on your lived experience. None of these things can be inferred directly from the text and its encoded meaning by itself. You would need to know that ICU stands for a larger set of words, and what those words are. You would need to know what a hospital is and why someone would need to be taken there instead of going there themselves. If any of these feel obvious, good. You live in a society and your pragmatic knowledge of that society overlaps well with the example provided. If I share an example from a less-populated society, “Janka got her grand-night lashings yesterday, she’s gonna get Peter tomorrow” you might be left scratching your head. If you are, realize this probably looks like how a lot of text data ends up looking to an LLM (anthropomorphization acknowledged). For those wondering, this sentence comes from Slovak Easter traditions. There’s a lot of meaning here that will just be missed and go unexplained if you are unaccustomed to these particular traditions as they stand in that culture. I personally have had the pleasure of trying to explain the Easter Bunny and its obsession with eggs to foreign colleagues and enjoyed the satisfaction of looking crazy.

In the context of LLMs, we can effectively group all out-of-text context into pragmatics. This means LLMs start without any knowledge of the outside world, and do not gain it during training. They only gain a knowledge of how humans respond to particular pragmatic stimuli. LLMs do not understand social class or race or gender or presidential candidates or anything else that might spark some type of emotion in you based on your life experience. Pragmatics isn’t something that we expect will be able to be directly incorporated into a model at any point, but we have already seen the benefits of incorporating it indirectly through data engineering and curation, prompting, and supervised fine-tuning on instruction.

Pragmatic structure gets added whether you mean to add it or not as soon as you acquire the data you are going to train on. You can think of this type of

pragmatic structure as bias, not inherently good or bad, but impossible to get rid of. Later down the line you get to pick what types of bias you'd like your data to keep by normalizing and curating, augmenting particular underrepresented points and cutting overrepresented or noisy examples.

There's a fine line between data engineering and pragmatic context, and it's just a matter of understanding what entailments exist in your data. An entailment, is a pragmatic marker within your data, as opposed to the literal text that your dataset contains. For example, let's say you have a model attempting to take an input like "write me a speech about frogs eating soggy socks that doesn't rhyme and where the first letters of each line spell amphibian," and actually follow that instruction. You can immediately tell that this input is asking for a lot. The balance for you as a data engineer would be to make sure that everything the input is asking for is explicitly accounted for in your data. You need to have examples of speeches, examples of what frogs and socks are and how they behave, and examples of acrostic poems. If you don't, the model might be able to understand just from whatever entailments exist in your dataset, but it's pretty up in the air. If you go the extra mile and keep track of entailed vs explicit information and tasks in your dataset, along with data distributions, you'll have examples to answer, "What is the garbage-in resulting in our garbage-out?"

LLMs struggle to pick up on pragmatics, even more so than people, but they do pick up on the things that your average standard deviation of people would. They can even replicate responses from people outside that standard deviation, but pretty inconsistently without the exact right stimulus. This is where everything during and after training comes in. Instruction-based datasets attempt to manufacture those stimuli during training by asking questions that entail representative responses. It is impossible to account for every possible situation in training, and you may inadvertently create new types of responses from your end users by trying to account for everything. After training, you can coax particular information from your model through prompting, which has a skill ceiling based on what your data originally entailed.

Morphology

Morphology is the study of word structures and how they are formed from smaller units called morphemes. Morphemes are the smallest units of meaning, like the "re-" in "redo" or "relearn." However, not all parts of words are morphemes, such as "ra-" in "ration" or "na-" in "nation."

Understanding how words are constructed helps create better language models and parsing algorithms, which are essential for tasks like tokenization. Tokens are somewhere between words and morphemes, they are statistically the most-likely candidates for units of meaning, but don’t necessarily correspond to existing morphemes. The effectiveness of a language model can depend on how well it can understand and process these tokens. For instance, in tokenization, a model needs to store a set of dictionaries to convert between words and their corresponding indices. One of these tokens is usually an "<UNK>" token, which represents any word that the model does not recognize. If this token is used too frequently, it can hinder the model's performance, either because the model's vocabulary is too small or because the tokenizer is not using the right algorithm for the task.

Consider a scenario where you want to build a code completion model, but you're using a tokenizer that only recognizes words separated by whitespace, like the nltk "punkt" tokenizer. When it encounters the string "def add_two_numbers_together(x, y):," it will pass "[def, <UNK>, y]" to the model. This causes the model to lose valuable information, not only because it doesn't recognize the punctuation, but also because the important part of the function's purpose is replaced with an unknown token due to the tokenizer's morphological algorithm. To improve the model's performance, a better understanding of word structure and the appropriate parsing algorithms is needed.

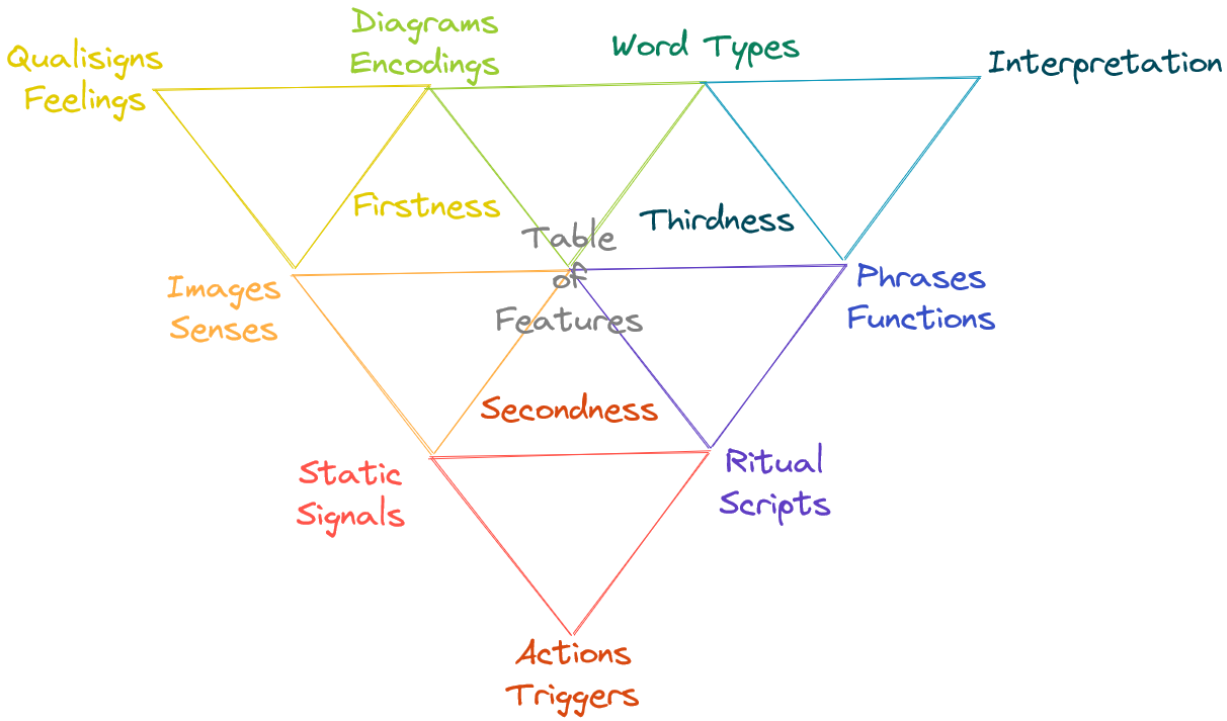
2.1.2 Semiotics

After exploring the fundamental features of language and examining their significance in the context of large language models, it is important to consider the broader perspective of meaning-making and interpretation in human communication. Semiotics, the study of signs and symbols, offers a valuable lens through which we can better understand how people interpret

and process language. In the following section, we will delve into the realm of semiotics, examining the relationship between signs, signifiers, and abstractions, as well as how these elements are utilized by LLMs to generate meaningful output. This discussion will provide a deeper understanding of the intricate processes through which LLMs manage to mimic human-like understanding of language, while also shedding light on the challenges and limitations they face in this endeavor. It should be noted that the authors do not believe that mimicking human behavior is necessarily the right answer for LLM improvement, only that mimicry is how the field has evaluated itself so far.

In our introduction to semiotics let's consider Figure 2.2 an adapted Peircean semiotic triangle. These are used to organize base ideas into sequences of firstness, secondness, and thirdness, with firstness being at the top left, secondness at the bottom, and thirdness being at the top right. If you've ever seen a semiotic triangle before, you may be surprised at the number of corners and orientation. To explain, we've turned them upside down to make it slightly easier to read, and because the system is recursive, we're showing how the system can model the entire process and each piece individually simultaneously. While the whole concept of these ideas is very cool, it's outside of the scope of this book to really delve into the philosophy. Instead, we can focus on the cardinal parts of those words (first, second, third) as showing the sequence things are processed in.

Figure 2.2 This is a recursive Peircean semiotic triangle. It's a system of organizing the process of extracting meaning from anything, in our case from language. Each point on the triangle illustrates one of the minimal parts needed to synthesize meaning within whatever the system is being used to describe, so here, each of these points are minimal units in meaning for language. Firstness, Secondness, and Thirdness are not points on the triangle, more markers for the people versed in Semiotics to be able to orient themselves in this diagram.



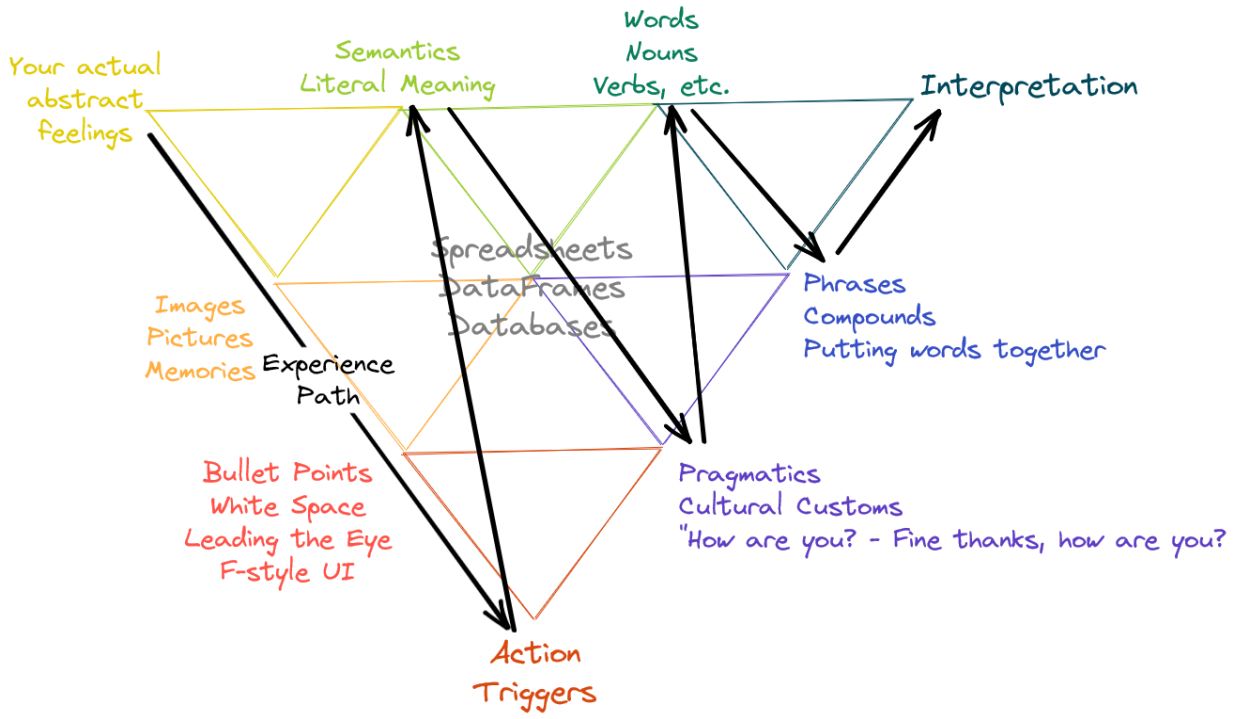
We can also look at each intersection of each of the triangles to gain an idea of why things are presented in the order they are. Feelings can be attached to images and encodings way before they can be attached to words and tables. Ritual and common scripts give a space for interpreted action that's just second nature and doesn't have to be thought about, similarly to how most phrases just come together from words without the native speaker needing to perform metacognition about each word individually. All of these eventually lead towards an interpretation or a document (a collection of utterances), and in our case, that interpretation should be reached by the LLM. This is why, for example, prompt engineering can boost model efficacy. Foundation LLMs that have trained on millions of examples of ritual scripts are able to replicate the type of script significantly better when you explicitly tell the model in the prompt which script needs to be followed. Try asking the model to give a step-by-step explanation, maybe prepend your generation with "Let's think about this step-by-step", you will see the model will generate step by step scripts based on previous scripts it's seen play out.

For those interested, there are specific ways of reading these figures and a whole field of semiotics to consider, however, it's not guaranteed that you'll be able to create the best LLMs by understanding the whole thing. Instead of

diving really deeply into this, we'll consider the bare minimum that can help you build the best models, UX, and UI for everyone to interact with. For example, one aspect of the process of creating meaning is recursiveness. When someone is talking to you, and they say something that doesn't make sense (is "meaningless" to you), what do you do? Generally, people will ask one or more clarifying questions to figure out what the meaning is, and the process is started over and over until the meaning is put across. The most state-of-the-art models that are currently on the market do not do this, but they can be made to do it through very purposeful prompting, but many people wouldn't even know to do that without having it pointed out to them. In other words, this is a brief introduction to semiotics. You don't need to be able to give in-depth and accurate coordinate-specific explanations to experts in the semiotic field by the end of this section. The point I'm really trying to push is that this is an organizational system showcasing the minimum number of things you actually need to create a full picture of meaning for another person to interpret. We are not giving the same amount of the same kinds of information to our models during training, but if we did, it would result in a marked improvement in model behavior.

These figures are meant to represent a minimal organizational model, where each of these pieces is essential. Let's consider Figure 2.3 which walks through an example of using a semiotic triangle. Consider Images, Pictures, and Memories and think about what it would be like to try and absorb the knowledge in this book without your eyes to process images, and without orthography (a writing system) to abstract the knowledge. Looking at Bullet Points, etc., how could you read this book without sections, whitespace between letters, and bullet points to show you the order and structure to process information with? Look at Semantics and literal encoded meaning and imagine the book without diagrams and words that didn't have dictionary definitions. Looking at spreadsheets in the middle, that could be a book without any tables or comparative informational organizers, including these figures. What would it be like trying to read this book without a culture or society that has habits and dogma to use as a lens for our interpretations? All of these points form our ability to interpret information, along with the lens that we end up passing our information through to recognize patterns.

Figure 2.3 Starting at the top left corner, follow the arrows to see the general order that we use to build our interpretations and extract meaning from things we interact with. Here, we've replaced the descriptive words with some examples of each point. Try to imagine interpreting this diagram without any words, without examples, without the arrows, or even without the pragmatic context of knowing what a figure in a book like this is supposed to be for.



So the important question then is: how many of these things do you see LLMs having access to to return meaningful interpretations? Does an LLM have access to feelings or societal rituals? Currently, they do not, but think about this as we go through traditional and newer techniques for NLP inference and think about what different models have access to.

2.1.3 Multilingual NLP

The last challenge that we need to touch on before we evaluate previous NLP techniques and current-generation LLMs is the foundation of linguistics and the reason LLMs even exist. People have wanted to understand or exploit each other since the first civilizations made contact.

These cases have resulted in the need for translators, and this need has only exponentially increased as the global economy has grown and flourished.

It's pretty simple math for business as well. Did you know that there are almost as many native speakers of Bengali as there are native speakers of English? If this is the first time you've heard of the Bengali language, this should hopefully color your perception that there is a valuable market for multilingual models. There are billions of people in the world, but only about a third of one billion of them speak English natively. If your model is anglocentric, like most are, you are missing out on 95% of the people in the world as customers and users. Spanish and Mandarin Chinese are easy wins in this area, but more people don't even go that far.

There are many more politically-charged examples of calling things the same or different languages that are out of the scope of this book. These are most often because of external factors like government involvement. Keeping these two points in mind–that a monolingual system focusing on English doesn't have the coverage or profit potential as many businesses act like and that the boundaries between languages and dialects are unreliable at best and systematically harmful at worst–it should highlight the dangerous swamp of opinions. Many businesses and research scientists don't even pretend to want to touch this swamp with a 50-foot pole when designing a product or system.

Unfortunately, no easy solutions exist at this time. However, the consideration of these factors can help you as a scientist or engineer (and hopefully an ethical person) to design LLMs that, at the very least, don't exacerbate and negatively contribute to the problems that already exist. The first step in this process is deciding on a directional goal from the beginning of the project, either towards localization (L10n) or internationalization (I18n). Localization is an approach exemplified by Mozilla, which has a different version of their browser available through crowdsourced L10n in over 90 languages with no indications of stopping that effort. Internationalization is similar, but in the opposite direction, for example, Ikea tries to put as few words as possible in their instructional booklets, opting instead for internationally recognized symbols and pictures to help

customers navigate the DIY projects. Deciding at the beginning of the project cuts down the effort required to expand to either solution exponentially. Large enough to switch the perception of translation and formatting from a cost to an investment. In the context of LLMs and their rapid expansion across the public consciousness, it becomes even more important to make that consideration early. Hitting the market with a world-changing technology that automatically disallows most of the world from interacting with it devalues those voices. Having to wait, jeopardizes their economic prospects.

Before continuing, let's take a moment to reflect on what we've discussed so far. We've hit important points in linguistics illustrating concepts for us to consider like understanding that the structure of language is separate from its meaning. We have demonstrated quite a journey that each of us takes, both personally and as a society, towards having the metacognition to understand and represent language in a coherent way for computers to work with. This understanding will only improve as we deepen our knowledge of cognitive fields, and as we solve for the linguistic features that we encounter. Going along with Figure 2.1, we will now demonstrate the computational path for language modeling that we have followed, and explore how it has and hasn't solved for any of those linguistic features or strived to create meaning. Let's move into evaluating the various techniques for representing a language algorithmically.

2.2 Language Modeling Techniques

Having delved into the fundamental features of language, the principles of semiotics, and the ways in which large language models interpret and process linguistic information, we now transition into a more practical realm. In the following section, we will explore the various natural language processing techniques that have been developed and employed to create these powerful language models. By examining the strengths and weaknesses of each approach, we will gain valuable insights into the effectiveness of these techniques in capturing the essence of human language and communication. This knowledge will not only help us appreciate the advancements made in the field of NLP but also enable us to better

understand the current limitations of these models and the challenges that lie ahead for future research and development.

Let’s take a second to just go over some data processing that will be universal to all language modeling. First, we’ll need to decide how we want to break up the words and symbols that we’ll be passing into our model, effectively deciding what a token will be in our model. Then, we’ll need a way to convert those tokens to numerical values and back again. Then, we’ll need to pick how our model will actually process the tokenized inputs. Each of the following techniques will build upon the previous techniques in at least one of these ways.

The first of these techniques is called a Bag of Words (BoW) model, and it consists of simply counting words as they appear in text. It can be accomplished very easily with a dictionary that scans through text, creating a new vocabulary entry for each new word as a key and an incrementing value starting at 1. Considering its simplicity, even this model based entirely on frequency can be quite powerful when trying to gain insight into a speaker’s intentions or at least their idiosyncrasies. For example, you could run a simple BoW model on inaugural speeches of US presidents, searching for the words freedom, economy, and enemy to gain a pretty good insight about which presidents assumed office under peacetime, during wartime, and during times of monetary strife, just based on how many times each word was mentioned. The BoW model’s weaknesses are many, however, as the model provides no images, semantics, pragmatics, phrases, or feelings. It doesn’t have any mechanisms to evaluate context or phonetics, and because it divides words by default on white space (you can obviously tokenize however you want, but try tokenizing on subwords and see what happens with this model—spoiler it is bad), it doesn’t account for morphology either. Altogether, it should be considered a weak model for representing language, but a strong baseline for evaluating other models against. In order to solve the problem of Bag of Words models not capturing any sequence data, N-Gram models were conceived.

2.2.1 N-Gram and Corpus-based techniques

N-Gram models represent a marked but efficient improvement to BoW, where you are able to give the model a sort of context, represented by N. They are relatively simple statistical models that allow you to generate words based on the N-1 context space. Looking at Listing 2.1, I'm using trigrams which means N=3. I clean the text and give it minimal padding/formatting to help the model, then we train using everygrams, which is meant to prioritize flexibility over efficiency so that you could train a pentagram or a septagram (N=5, N=7) model if you wanted instead. At the end of the listing where I'm generating, I can give the model up to 2 tokens to help it figure out how to generate further. N-Gram models were not created, and have never even claimed to attempt, complete modeling systems of linguistic knowledge, but they are widely useful in practical applications. They ignore all linguistic features, including syntax, and only attempt to draw probabilistic connections between words appearing in an N-length phrase.

Looking at Figure 2.2, N-Grams really only use static signals (whitespace, orthography) and words to try to extract any meaning. It tries to measure phrases manually, assuming that all of the phrases will be the same length. That said, N-Grams can be used to create powerful baselines for text analysis, and if the pragmatic context of the utterance is already known by the analyst, they can be used to give quick and accurate insight into real-world scenarios. It is possible to make an N-Gram LLM by just making N=1000000000 or higher, but this doesn't have any practical application, as 99.9% of all text and 100% of all meaningful text contains fewer than one billion tokens appearing more than once and that computational power can be much better spent elsewhere.

Listing 2.1 Generative N-Grams Language Model Implementation

```
from nltk.corpus.reader import PlaintextCorpusReader
from nltk.util import everygrams
from nltk.lm.preprocessing import (
    pad_both_ends,
    flatten,
    padded_everygram_pipeline,
)
```

```
from nltk.lm import MLE

# Create a corpus from any number of plain .txt files
my_corpus = PlaintextCorpusReader("./", ".*\.txt")

for sent in my_corpus.sents(fileids="hamlet.txt"):
    print(sent)

# Pad each side of every line in the corpus with <s> and </s> to
indicate the start and end of utterances
padded_trigrams = list(
    pad_both_ends(my_corpus.sents(fileids="hamlet.txt")[1104],
n=2)
)
list(everygrams(padded_trigrams, max_len=3))

list(
    flatten(
        pad_both_ends(sent, n=2)
        for sent in my_corpus.sents(fileids="hamlet.txt")
    )
)

# Allow everygrams to create a training set and a vocab object
from the data
train, vocab = padded_everygram_pipeline(
    3, my_corpus.sents(fileids="hamlet.txt")
)

# Instantiate and train the model we'll use for N-Grams, a
Maximum Likelihood Estimator (MLE)
# This model will take the everygrams vocabulary, including the
<UNK> token used for out-of-vocabulary
lm = MLE(3)
len(lm.vocab)

lm.fit(train, vocab)
print(lm.vocab)
len(lm.vocab)

# And finally, language can be generated with this model and
conditioned with n-1 tokens preceding
lm.generate(6, ["to", "be"])
```

The above code is all that you need to create a generative N-gram model. For those of you interested in being able to evaluate that model further, we've included the below code to grab probabilities and log scores, or analyze the entropy and perplexity of a particular phrase. Because this is all frequency-based, even though it's mathematically significant, it still does a pretty bad job of describing how perplexing or frequent real-world language actually is.

```
# Any set of tokens up to length=n can be counted easily to
determine frequency
print(lm.counts)
lm.counts[["to"]]["be"]

# Any token can be given a probability of occurrence, and can be
augmented with up to n-1 tokens to precede it
print(lm.score("be"))
print(lm.score("be", ["to"]))
print(lm.score("be", ["not", "to"]))

# This can be done as a log score as well to avoid very big and
very small numbers
print(lm.logscore("be"))
print(lm.logscore("be", ["to"]))
print(lm.logscore("be", ["not", "to"]))

# Sets of tokens can be tested for entropy and perplexity as
well
test = [("to", "be"), ("or", "not"), ("to", "be")]
print(lm.entropy(test))
print(lm.perplexity(test))
```

While this code example illustrates creating a trigram language model, unfortunately, not all phrases needing to be captured are only 3 tokens long. For example, from Hamlet, "To be or not to be," consists of one phrase with 2 words and one phrase with 4 words. This type of phrasal modeling also fails to capture any semantic encodings that individual words could have. In order to solve these problems, Bayesian statistics were applied to language modeling.

2.2.2 Bayesian Techniques

Bayes' theorem is one of the most mathematically sound and simple theories present in describing the occurrence of your output within your input space. Essentially, it calculates the probability of an event occurring based on prior knowledge. The theorem posits that the probability of a hypothesis being true given evidence, for example that a sentence has a positive sentiment, is equal to the probability of the evidence occurring given the hypothesis is true multiplied by the probability of the hypothesis occurring, all divided by the probability of the evidence being true. Or, expressed mathematically:

P(hypothesis | evidence) = (P(evidence | hypothesis) * P(hypothesis)) / P(evidence)

Or

P(A|B) * P(B) = P(B|A) * P(A)

Because this is neither a math book, nor do we care to dive too much into theory, we'll trust you can get further informed about this theorem.

Unfortunately, even though the theorem represents the data in a mathematically sound way, it doesn't account for any stochasticity or multiple meanings of words. One word you can always throw at a Bayesian model to confuse it is the word, "it." Any demonstrative pronoun ends up getting assigned values in the same LogPrior and LogLikelihood way as all of the other words and gets a static value, which is antithetical to the usage of those words. For example, if you're trying to perform sentiment analysis on an utterance, it would be better for you to assign all pronouns a null value than to even let them go through the Bayesian training. It should be noted also that Bayesian techniques don't end up creating generative language models the way the rest of these will. Because of the nature of Bayes' theorem validating a hypothesis, these models work for classification, and can bring powerful augmentation to a generative language model.

In Listing 2.2 we show how to create a Naive Bayes classification language model. Instead of using a package like sklearn or something that would make writing the code a little easier, we opted to write out what we were doing, so it's a bit longer, but should be more informative about how it

works. We are using the least-complex version of a Naive Bayes model. We haven't made it multinomial or added anything fancy and this could obviously work better if you opted to upgrade it for any problem you want. And we highly recommend you do.

Listing 2.2 Categorical Naive Bayes Language Model Implementation

```
from utils import process_utt, lookup
from nltk.corpus.reader import PlaintextCorpusReader
import numpy as np

my_corpus = PlaintextCorpusReader("./", ".*\.txt")

sents = my_corpus.sents(fileids="hamlet.txt")

def count_utts(result, utts, ys):
    """
    Input:
        result: a dictionary that is used to map each pair to
its frequency
        utts: a list of utts
        ys: a list of the sentiment of each utt (either 0 or 1)
    Output:
        result: a dictionary mapping each pair to its frequency
    """

    for y, utt in zip(ys, utts):
        for word in process_utt(utt):
            # define the key, which is the word and label tuple
            pair = (word, y)

            # if the key exists in the dictionary, increment the
count
            if pair in result:
                result[pair] += 1

            # if the key is new, add it to the dict and set the
count to 1
            else:
                result[pair] = 1

    return result
```

```
result = {}
utts = [" ".join(sent) for sent in sents]
ys = [sent.count("be") > 0 for sent in sents]
count_utts(result, utts, ys)

freqs = count_utts({}, utts, ys)
lookup(freqs, "be", True)
for k, v in freqs.items():
    if "be" in k:
        print(f"{k}:{v}")

def train_naive_bayes(freqs, train_x, train_y):
    """
    Input:
        freqs: dictionary from (word, label) to how often the
word appears
        train_x: a list of utts
        train_y: a list of labels correponding to the utts (0,1)
    Output:
        logprior: the log prior.
        loglikelihood: the log likelihood of you Naive bayes
equation.
    """
    loglikelihood = {}
    logprior = 0

    # calculate V, the number of unique words in the vocabulary
    vocab = set([pair[0] for pair in freqs.keys()])
    V = len(vocab)

    # calculate N_pos and N_neg
    N_pos = N_neg = 0
    for pair in freqs.keys():
        # if the label is positive (greater than zero)
        if pair[1] > 0:
            # Increment the number of positive words (word,
label)
            N_pos += lookup(freqs, pair[0], True)

        # else, the label is negative
        else:
            # increment the number of negative words
(word,label)
            N_neg += lookup(freqs, pair[0], False)
```

```
    # Calculate D, the number of documents
    D = len(train_y)

    # Calculate the number of positive documents
    D_pos = sum(train_y)

    # Calculate the number of negative documents
    D_neg = D - D_pos

    # Calculate logprior
    logprior = np.log(D_pos) - np.log(D_neg)

    # For each word in the vocabulary...
    for word in vocab:
        # get the positive and negative frequency of the word
        freq_pos = lookup(freqs, word, 1)
        freq_neg = lookup(freqs, word, 0)

        # calculate the probability that each word is positive,
and negative
        p_w_pos = (freq_pos + 1) / (N_pos + V)
        p_w_neg = (freq_neg + 1) / (N_neg + V)

        # calculate the log likelihood of the word
        loglikelihood[word] = np.log(p_w_pos / p_w_neg)

    return logprior, loglikelihood

def naive_bayes_predict(utt, logprior, loglikelihood):
    """
    Input:
        utt: a string
        logprior: a number
        loglikelihood: a dictionary of words mapping to numbers
    Output:
        p: the sum of all the logliklihoods + logprior
    """
    # process the utt to get a list of words
    word_l = process_utt(utt)

    # initialize probability to zero
    p = 0

    # add the logprior
    p += logprior
```

```
    for word in word_l:
        # check if the word exists in the loglikelihood
dictionary
        if word in loglikelihood:
            # add the log likelihood of that word to the
probability
            p += loglikelihood[word]

    return p

def test_naive_bayes(test_x, test_y, logprior, loglikelihood):
    """
    Input:
        test_x: A list of utts
        test_y: the corresponding labels for the list of utts
        logprior: the logprior
        loglikelihood: a dictionary with the loglikelihoods for
each word
    Output:
        accuracy: (# of utts classified correctly)/(total # of
utts)
    """
    accuracy = 0  # return this properly

    y_hats = []
    for utt in test_x:
        # if the prediction is > 0
        if naive_bayes_predict(utt, logprior, loglikelihood) >
0:
            # the predicted class is 1
            y_hat_i = 1
        else:
            # otherwise the predicted class is 0
            y_hat_i = 0

        # append the predicted class to the list y_hats
        y_hats.append(y_hat_i)

    # error = avg of the abs vals of the diffs between y_hats
and test_y
    error = sum(
        [abs(y_hat - test) for y_hat, test in zip(y_hats,
test_y)]
    ) / len(y_hats)
```

```
    # Accuracy is 1 minus the error
    accuracy = 1 - error

    return accuracy

if __name__ == "__main__":
    logprior, loglikelihood = train_naive_bayes(freqs, utts, ys)
    print(logprior)
    print(len(loglikelihood))

    my_utt = "To be or not to be, that is the question."
    p = naive_bayes_predict(my_utt, logprior, loglikelihood)
    print("The expected output is", p)

    print(
        "Naive Bayes accuracy = %0.4f"
        % (test_naive_bayes(utts, ys, logprior, loglikelihood))
    )
```

This theorem doesn't create the same type of language model, but one with a list of probabilities associated with one hypothesis. As such, Bayesian language models can't be used effectively to generate language, but it can be very powerfully implemented for classification tasks. In my opinion though, Bayesian models are often overhyped for even this task. One of the crowning achievements of my career was replacing and removing a Bayesian model from production.

In Bayesian models one of the big issues is essentially that all sequences are completely unconnected, like BoW models, moving us to the opposite end of sequence modeling from N-Grams. Similarly to a pendulum, language modeling swings back towards sequence modeling and language generation with Markov Chains.

2.2.3 Markov Chains

Often called Hidden Markov Models (HMMs), Markov Chains essentially add state to the N-Gram models mentioned before, storing probabilities using hidden states. They are often used to help parse text data for even larger models, doing things like Part-of-Speech tagging (PoS Tagging,

marking words with their part of speech) and Named Entity Recognition (NER, marking identifying words with their referent and usually type, e.g. LA - Los Angeles - City) on textual data. Markov models, as opposed to previous Bayesian models, rely completely on stochasticity (predictable randomness) whereas the Bayesian models pretended it didn't exist. The idea is similarly mathematically sound, however, that the probability of anything happening *next* depends completely upon the state of *now*. So instead of modeling words based solely on their historical occurrence and drawing a probability from that, we model their future and past collocation based on what is currently occurring. So the probability of "happy" occurring goes down to almost zero if "happy" was just output, but goes up significantly if "am" has just occurred. Markov chains are so intuitive that they were incorporated into later iterations of Bayesian statistics, and are still used in production systems today.

In Listing 2.3 we train a Markov chain generative language model. This is the first model where we've used a specific tokenizer, which in this case will just tokenize based on the white space between words. This is also only the second time we've referred to a collection of utterances meaning to be viewed together as a document. As you play around with this one, pay close attention and make some comparisons yourself for how well the HMM generates compared to even a large N-gram model.

Listing 2.3 Generative Hidden Markov Language Model Implementation

```
import re
import random
from nltk.tokenize import word_tokenize
from collections import defaultdict, deque

class MarkovChain:
    def __init__(self):
        self.lookup_dict = defaultdict(list)
        self._seeded = False
        self.__seed_me()

    def __seed_me(self, rand_seed=None):
        if self._seeded is not True:
            try:
```

```
            if rand_seed is not None:
                random.seed(rand_seed)
            else:
                random.seed()
            self._seeded = True
        except NotImplementedError:
            self._seeded = False

def add_document(self, str):
    preprocessed_list = self._preprocess(str)
    pairs = self.__generate_tuple_keys(preprocessed_list)
    for pair in pairs:
        self.lookup_dict[pair[0]].append(pair[1])

def _preprocess(self, str):
    cleaned = re.sub(r"\W+", " ", str).lower()
    tokenized = word_tokenize(cleaned)
    return tokenized

def __generate_tuple_keys(self, data):
    if len(data) < 1:
        return

    for i in range(len(data) - 1):
        yield [data[i], data[i + 1]]

def generate_text(self, max_length=50):
    context = deque()
    output = []
    if len(self.lookup_dict) > 0:
        self.__seed_me(rand_seed=len(self.lookup_dict))
        chain_head = [list(self.lookup_dict)[0]]
        context.extend(chain_head)

        while len(output) < (max_length - 1):
            next_choices = self.lookup_dict[context[-1]]
            if len(next_choices) > 0:
                next_word = random.choice(next_choices)
                context.append(next_word)
                output.append(context.popleft())
            else:
                break
        output.extend(list(context))
    return " ".join(output)
```

```
if __name__ == "__main__":
    with open("hamlet.txt", "r", encoding="utf-8") as f:
        text = f.read()
    HMM = MarkovChain()
    HMM.add_document(text)

    print(HMM.generate_text(max_length=25))
```

This code shows a basic implementation of a Markov model for generation, and we'd encourage the reader to experiment with it, give it text from songs from your favorite musicians or books from your favorite authors and see whether what comes out actually sounds like them. HMMs are incredible fast and often used in predictive text or predictive search applications. Markov models represent the first comprehensive attempt to actually model language from a descriptive linguistic perspective, as opposed to a prescriptive one, which is interesting, because Markov did not originally intend for linguistic modeling, only to win an argument about continuous independent states. Later, Markov used Markov Chains to model vowel distribution in a Pushkin novel, so he was at least aware of the possible applications.

The difference between descriptive and prescriptive linguistics is that one focuses on how things *ought* to be, while the other focuses on how things *are*. From a language modeling perspective, it has proven vastly more effective to describe what language is doing from a corpus or Markov perspective, rather than to attempt to prescribe how language ought to behave. Unfortunately, a current state by itself cannot be used to give context beyond the now, so historical or societal context is unable to be represented effectively in a Markov model. Semantic encoding of words also becomes a problem, as is represented in the code example, Markov chains will output syntactically correct chains of words that semantically are nonsense, similar to "colorless green ideas sleep furiously." To attempt to solve this problem, "continuous" models were developed to allow for a "semantic embedding" representation of tokens.

2.2.4 Continuous Language Modeling

A Continuous Bag of Words (CBoW), much like its namesake, the Bag of Words, is a frequency-based approach to analyzing language, meaning that it models words based on how often they occur. The next word in an utterance has never been determined based on probability or frequency. Due to this, the example given will be for how to create word embeddings to be ingested or compared by other models using a CBoW. We'll use a neural network for this to give you a good methodology.

This is the first language modeling technique we'll see that essentially slides a context window over a given utterance (the context window is an N-gram model) and attempts to guess what word is in the middle based upon surrounding words in the window. For example, let's say your window has a length of 5, and your sentence is, "Learning about linguistics makes me happy," you would give the CBoW ['learning', 'about', 'makes', 'me'] and try to get the model to guess "linguistics," based upon how many times the model has seen that word occur in similar places previously. This should show you why generation is difficult for models trained like this, because if you give the model ['makes', 'me', '</s>] as input, first of all it only has 3 pieces of information to try to figure out instead of 4, and it also will be biased towards only guessing words it has seen at the end of sentences before, as opposed to getting ready to start new clauses. It's not all bad though, one feature that makes continuous models stand out for embeddings is that it doesn't just have to look at words before the target word, they can also use words that come after the target to gain some semblance of context.

In Listing 2.4 we create our first continuous model. In our case, to keep things as simple as possible, we use a bag of words for the language processing and a one-layer neural network with two parameters for the embedding estimation, although both of those could be substituted out for any other models. For example, you could substitute N-grams for the BoW and a Naive Bayes for the neural network, and get a Continuous Naive N-gram model. The point is that the actual models used in this technique are a bit arbitrary, it's more the Continuous technique that's important. To illustrate this further, we don't use any packages other than numpy to do the math for the neural network, even though it's our first one appearing in this section.

Pay special attention to the steps below, initializing the model weights, the ReLU activation function, the final softmax layer, forward and backpropagation, and then how it all fits together in the `gradient_descent` function. These are pieces in the puzzle that you will see crop up again and again, regardless of programming language or framework. You will need to initialize models, pick activation functions, pick final layers, and define forward and backward propagation in Tensorflow, Pytorch and HuggingFace, and if you ever start creating your own models as opposed to using someone else's.

Listing 2.4 Generative Continuous Bag of Words Language Model Implementation

```
import nltk
import numpy as np
from utils import get_batches, compute_pca, get_dict
import re
from matplotlib import pyplot

# Create our corpus for training
with open("hamlet.txt", "r", encoding="utf-8") as f:
    data = f.read()

# Slightly clean the data by removing punctuation, tokenizing by
word, and converting to lowercase alpha characters
data = re.sub(r"[,!?;-]", ".", data)
data = nltk.word_tokenize(data)
data = [ch.lower() for ch in data if ch.isalpha() or ch == "."]
print("Number of tokens:", len(data), "\n", data[500:515])

# Get our Bag of Words, along with a distribution
fdist = nltk.FreqDist(word for word in data)
print("Size of vocabulary:", len(fdist))
print("Most Frequent Tokens:", fdist.most_common(20))

# Create 2 dictionaries to speed up time-to-convert and keep
track of vocabulary
word2Ind, Ind2word = get_dict(data)
V = len(word2Ind)
print("Size of vocabulary:", V)

print("Index of the word 'king':", word2Ind["king"])
print("Word which has index 2743:", Ind2word[2743])
```

```
# Here we create our Neural network with 1 layer and 2
parameters
def initialize_model(N, V, random_seed=1):
    """
    Inputs:
        N: dimension of hidden vector
        V: dimension of vocabulary
        random_seed: seed for consistent results in tests
    Outputs:
        W1, W2, b1, b2: initialized weights and biases
    """
    np.random.seed(random_seed)

    W1 = np.random.rand(N, V)
    W2 = np.random.rand(V, N)
    b1 = np.random.rand(N, 1)
    b2 = np.random.rand(V, 1)

    return W1, W2, b1, b2

# Create our final classification layer, which makes all
possibilities add up to 1
def softmax(z):
    """
    Inputs:
        z: output scores from the hidden layer
    Outputs:
        yhat: prediction (estimate of y)
    """
    yhat = np.exp(z) / np.sum(np.exp(z), axis=0)
    return yhat

# Define the behavior for moving forward through our model,
along with an activation function
def forward_prop(x, W1, W2, b1, b2):
    """
    Inputs:
        x: average one-hot vector for the context
        W1,W2,b1,b2: weights and biases to be learned
    Outputs:
        z: output score vector
    """
    h = W1 @ x + b1
    h = np.maximum(0, h)
```

```
    z = W2 @ h + b2
    return z, h

# Define how we determine the distance between ground truth and
model predictions
def compute_cost(y, yhat, batch_size):
    logprobs = np.multiply(np.log(yhat), y) + np.multiply(
        np.log(1 - yhat), 1 - y
    )
    cost = -1 / batch_size * np.sum(logprobs)
    cost = np.squeeze(cost)
    return cost

# Define how we move backward through the model and collect
gradients
def back_prop(x, yhat, y, h, W1, W2, b1, b2, batch_size):
    """
    Inputs:
        x:  average one hot vector for the context
        yhat: prediction (estimate of y)
        y:  target vector
        h:  hidden vector (see eq. 1)
        W1, W2, b1, b2:  weights and biases
        batch_size: batch size
     Outputs:
        grad_W1, grad_W2, grad_b1, grad_b2:  gradients of
weights and biases
    """
    l1 = np.dot(W2.T, yhat - y)
    l1 = np.maximum(0, l1)
    grad_W1 = np.dot(l1, x.T) / batch_size
    grad_W2 = np.dot(yhat - y, h.T) / batch_size
    grad_b1 = np.sum(l1, axis=1, keepdims=True) / batch_size
    grad_b2 = np.sum(yhat - y, axis=1, keepdims=True) /
batch_size

    return grad_W1, grad_W2, grad_b1, grad_b2

# Put it all together and train
def gradient_descent(data, word2Ind, N, V, num_iters,
alpha=0.03):
    """
    This is the gradient_descent function

      Inputs:
```

```
        data:      text
        word2Ind:  words to Indices
        N:         dimension of hidden vector
        V:         dimension of vocabulary
        num_iters: number of iterations
     Outputs:
        W1, W2, b1, b2:  updated matrices and biases

    """
    W1, W2, b1, b2 = initialize_model(N, V, random_seed=8855)
    batch_size = 128
    iters = 0
    C = 2
    for x, y in get_batches(data, word2Ind, V, C, batch_size):
        z, h = forward_prop(x, W1, W2, b1, b2)
        yhat = softmax(z)
        cost = compute_cost(y, yhat, batch_size)
        if (iters + 1) % 10 == 0:
            print(f"iters: {iters+1} cost: {cost:.6f}")
        grad_W1, grad_W2, grad_b1, grad_b2 = back_prop(
            x, yhat, y, h, W1, W2, b1, b2, batch_size
        )
        W1 = W1 - alpha * grad_W1
        W2 = W2 - alpha * grad_W2
        b1 = b1 - alpha * grad_b1
        b2 = b2 - alpha * grad_b2
        iters += 1
        if iters == num_iters:
            break
        if iters % 100 == 0:
            alpha *= 0.66

    return W1, W2, b1, b2

# Train the model
C = 2
N = 50
word2Ind, Ind2word = get_dict(data)
V = len(word2Ind)
num_iters = 150
print("Call gradient_descent")
W1, W2, b1, b2 = gradient_descent(data, word2Ind, N, V,
num_iters)
Call gradient descent
Iters: 10 loss: 0.525015
```

```
Iters: 20 loss: 0.092373
Iters: 30 loss: 0.050474
Iters: 40 loss: 0.034724
Iters: 50 loss: 0.026468
Iters: 60 loss: 0.021385
Iters: 70 loss: 0.017941
Iters: 80 loss: 0.015453
Iters: 90 loss: 0.012099
Iters: 100 loss: 0.012099
Iters: 110 loss: 0.011253
Iters: 120 loss: 0.010551
Iters: 130 loss: 0.009932
Iters: 140 loss: 0.009382
Iters: 150 loss: 0.008889
```

The CBoW example is our first code example to showcase a full and effective training loop in machine learning. Within all of that, we asked the reader to pay special attention to the steps in a training loop, especially the activation function, ReLU. As we expect the reader to be at least familiar with various ML paradigms, including different activations, we won't explain the ReLU here, rather why you should use it and why you shouldn't. ReLUs, while solving the vanishing gradient problem, don't solve the exploding gradient problem, and they sharply destroy all negative comparisons within the model. Better situational variants include the ELU, which allows negative numbers normalizing to alpha, or the GEGLU/SWIGLU, which works well in increasingly perplex scenarios, like language. However, people often use ReLUs, not because they are the best in a situation, but because they are easy-to-understand, easy-to-code, and intuitive, even more so than the activations they were created to replace like the sigmoid or tanh.

A lot of this ends up being abstracted with packages and the like, but knowing what's going on under the hood will be very helpful for you as someone putting LLMs in production. You should be able to predict with some certainty how different models will behave in various situations. The next section will dive into one of those abstractions, in this case being the abstraction that is created by the continuous modeling technique.

2.2.5 Embeddings

Hearkening back to our features of language, it should be easy to connect why continuous-style language modeling was such a breakthrough. Embeddings take the tokenized vectors we've created that don't contain any meaning, and attempt to insert that meaning based on observations that can be made about the text, such as word order and subwords appearing in similar contexts. Despite the primary mode of meaning being collocation (co-located, words that appear next to each other), they prove useful and even show some similarities to human-encoded word meaning.

The quintessential example from Word2Vec, one of the first pre-trained vector embeddings, was taking the vector for "king" subtracting the vector for "man" adding the vector for "woman" and finding the nearest neighbor to the sum was the vector for the word "queen". This makes sense to us as it mimics human semantics. One of the major differences is one that's already been mentioned a couple of times, pragmatics. Humans use pragmatic context to inform semantic meaning, understanding that just because you said, "I need food," doesn't mean you are actually in physical danger without it. Embeddings are devoid of any influence outside of pure usage, which feels like it could be how humans learn as well, and there are good arguments on all sides here. The one thing holding is that if we can somehow give models more sense data, that may open the door to more effective embeddings.

In Listing 2.5, we'll dive into how to visualize embeddings using pyplot. We will be going more in-depth into embeddings in later chapters. This is helpful for model explainability and also for validation during your pre-training step. If you see that your semantically similar embeddings are relatively close to each other on the graph, then you're likely going in the right direction.

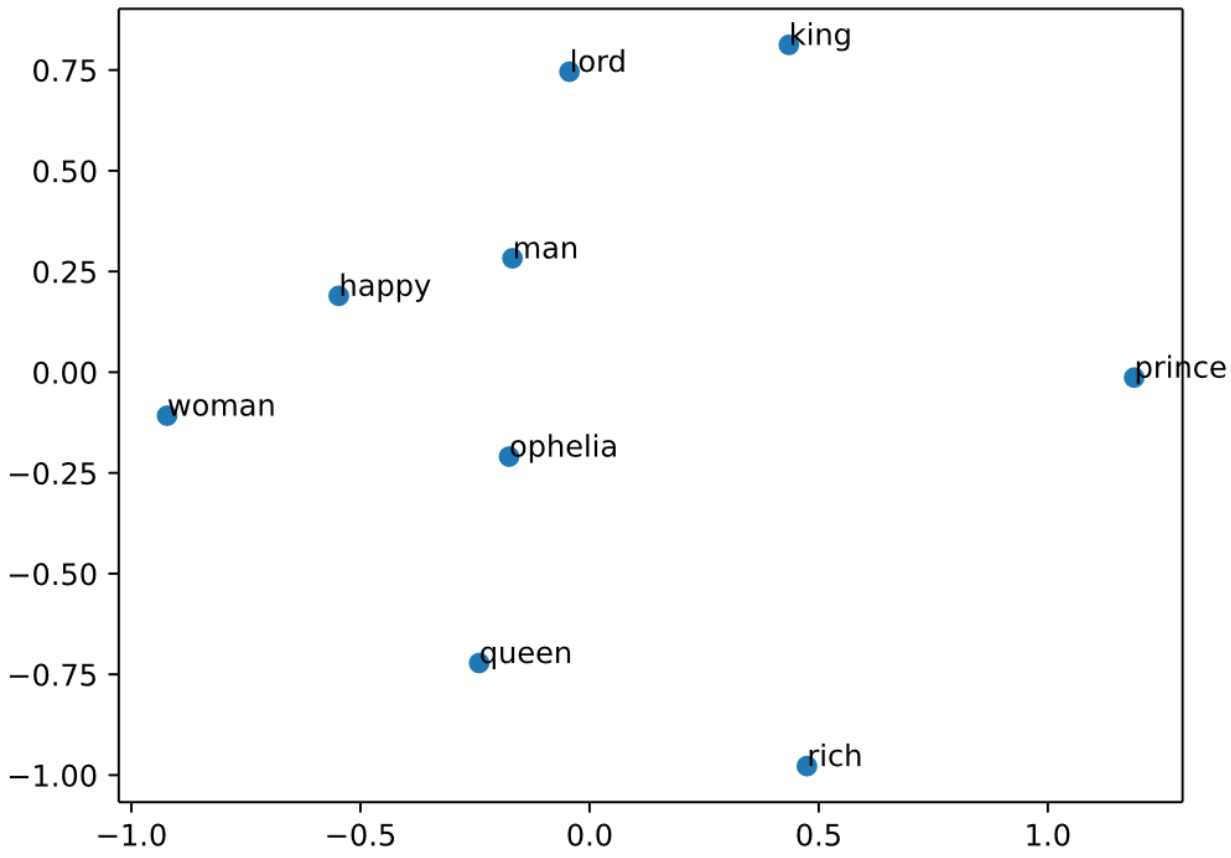
Listing 2.5 Embedding Visualization

```
# After listing 2.4 is done and gradient descent has been
executed
words = [
    "King",
    "Queen",
```

```
    "Lord",
    "Man",
    "Woman",
    "Prince",
    "Ophelia",
    "Rich",
    "Happy",
]
embs = (W1.T + W2) / 2.0
idx = [word2Ind[word] for word in words]
X = embs[idx, :]
print(X.shape, idx)

result = compute_pca(X, 2)
pyplot.scatter(result[:, 0], result[:, 1])
for i, word in enumerate(words):
    pyplot.annotate(word, xy=(result[i, 0], result[i, 1]))
pyplot.show()
```

Figure 2.4 A visualization technique for word embeddings. Visualizing embeddings can be important for model explainability.



As we can see in figure 2.4, this is a successful, but a very sparse embedding representation that we trained from our CBoW model. Getting those semantic representations (embeddings) to be denser is the main place that we can see improvement in this field, although many successful experiments have been run where denser semantic meaning has been supplanted with greater pragmatic context through instruct and different thought chaining techniques. We will address Chain of Thought (CoT) and other techniques later. For now, let's pivot to discussing why our continuous embedding technique can even be successful, given frequency-based models are characteristically difficult to correlate with reality. All of this starts with the Multilayer Perceptron, more than half a century ago.

2.2.6 Multilayer Perceptrons

MLPs are the embodiment of the sentiment, “Machines are really good at doing one thing, so I wish we could just use a bunch of machines that are really good at the one thing to make one that’s good at a lot of things.” Every weight and bias in the neural network of the MLP is good at detecting one feature, so we bind a whole bunch of them together to detect larger, more complex features. MLPs serve as the primary building block in most neural network architectures. The key distinctions between architectures, such as convolutional neural networks and recurrent neural networks, mainly arise from data loading methods and the handling of tokenized and embedded data as it flows through the layers of the model, rather than the functionality of individual layers, particularly the fully-connected layers.

In Listing 2.6 we provide a more dynamic class of neural network that can have as many layers and parameters as deemed necessary for your task. We give a more-defined and explicit class using pytorch to give you the tools to implement the MLP for use in whatever you’d like, both from scratch and in a popular framework.

Listing 2.6 Multilayer Perceptron Pytorch Class Implementation

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class MultiLayerPerceptron(nn.Module):
    def __init__(
        self,
        input_size,
        hidden_size=2,
        output_size=3,
        num_hidden_layers=1,
        hidden_activation=nn.Sigmoid,
    ):
        """Initialize weights.
        Args:
            input_size (int): size of the input
            hidden_size (int): size of the hidden layers
            output_size (int): size of the output
            num_hidden_layers (int): number of hidden layers
            hidden_activation (torch.nn.*): the activation class
```

```
        """
        super(MultiLayerPerceptron, self).__init__()
        self.module_list = nn.ModuleList()
        interim_input_size = input_size
        interim_output_size = hidden_size
        torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

        for _ in range(num_hidden_layers):
            self.module_list.append(
                nn.Linear(interim_input_size,
interim_output_size)
            )
            self.module_list.append(hidden_activation())
            interim_input_size = interim_output_size

        self.fc_final = nn.Linear(interim_input_size,
output_size)

        self.last_forward_cache = []

    def forward(self, x, apply_softmax=False):
        """The forward pass of the MLP

        Args:
            x_in (torch.Tensor): an input data tensor.
                x_in.shape should be (batch, input_dim)
            apply_softmax (bool): a flag for the softmax
activation
                should be false if used with the Cross Entropy
losses
        Returns:
            the resulting tensor. tensor.shape should be (batch,
output_dim)
        """
        for module in self.module_list:
            x = module(x)

        output = self.fc_final(x)

        if apply_softmax:
            output = F.softmax(output, dim=1)

        return output
```

From the code we can see, as opposed to the CBoW implementation which had a static two layers, this MLP is not static in size until it has been instantiated. If you wanted to give this model one million layers, you would just have to put num_hidden_layers=1000000 when you instantiate the class, although just because you *can* give a model that many parameters it won't make it immediately better. LLMs are more than just a lot of layers. Like RNNs and CNNs, the magic of LLMs is in how data goes in and moves through the model. To illustrate, let's look at the RNN and one of its variations.

2.2.7 RNNs and LSTMs

Recurrent Neural Networks (RNNs) are a class of neural networks designed to analyze sequences, based on the weaknesses in previous language modeling techniques. The logic goes that if language is presented in a sequence, then maybe it should be processed in a sequence, as opposed to one token at a time. RNNs accomplish this by using logic we've seen before, both in MLPs and in Markov Chains, where an internal state or memory is referred to when new inputs are processed, and creating cycles when connections between nodes are detected as being useful.

In fully recurrent networks, like the one in Listing 2.7, all nodes start out initially connected to all subsequent nodes, but those connections can be set to zero to simulate them being broken if they are not useful. This solves one of the biggest problems that earlier models suffered from, static input size, and enables an RNN and its variants to process variable length inputs. Unfortunately, longer sequences create a new problem. Because each neuron in the network has connections to subsequent neurons, longer sequences create smaller changes to the overall sum, making the gradients smaller and eventually vanishing, even with important words.

For example, let's consider these sentences with the task sentiment analysis, "I loved the movie last night," and, "The movie I went to see last night was the very best I had ever expected to see." These sentences can be considered semantically similar, even if they aren't exactly the same. When moving through an RNN, each word in the first sentence is worth more, and the

consequence is that the first sentence has a higher positive rating than the second sentence, just because of the first sentence being shorter. The inverse is true also, exploding gradients are also a consequence of this sequence processing, which makes training deep RNNs difficult.

To solve this problem, long short-term memories (LSTMs), which are a type of RNN, use memory cells and gating mechanisms to keep being able to process sequences of variable length, but without the problems of longer and shorter sequences being comprehended differently. Anticipating multilingual scenarios and understanding that people don't think about language in only one direction, LSTMs can also process sequences bidirectionally by concatenating the outputs of two RNNs, one reading the sequence from left to right, and the other from right to left. This bidirectionality improves results, allowing for information to be seen and remembered even after thousands of tokens have passed.

In Listing 2.7 we give classes for both an RNN and an LSTM. In the code in the repo associated with this book, you can see the results of training both the RNN and LSTM, where the takeaway is that the LSTM gets better accuracy on both training and validation sets in half as many epochs (25 vs 50 with RNN). One of the innovations to note is the packed embeddings that utilize padding, extending all variable-length sequences to the maximum length in order to allow processing any length input, as long as it is shorter than the maximum.

Listing 2.7 Recurrent Neural Network and Long Short-Term Memory Pytorch Class Implementations

```
import torch
from gensim.models import Word2Vec
from sklearn.model_selection import train_test_split

# Create our corpus for training
with open("./chapters/chapter_2/hamlet.txt", "r", encoding="utf-
8") as f:
    data = f.readlines()

# Embeddings are needed to give semantic value to the inputs of
```

```
an LSTM
# embedding_weights = torch.Tensor(word_vectors.vectors)

EMBEDDING_DIM = 100
model = Word2Vec(data, vector_size=EMBEDDING_DIM, window=3,
min_count=3, workers=4)
word_vectors = model.wv
print(f"Vocabulary Length: {len(model.wv)}")
del model

padding_value = len(word_vectors.index_to_key)
embedding_weights = torch.Tensor(word_vectors.vectors)

class RNN(torch.nn.Module):
    def __init__(
        self,
        input_dim,
        embedding_dim,
        hidden_dim,
        output_dim,
        embedding_weights,
    ):
        super().__init__()
        self.embedding = torch.nn.Embedding.from_pretrained(
            embedding_weights
        )
        self.rnn = torch.nn.RNN(embedding_dim, hidden_dim)
        self.fc = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, x, text_lengths):
        embedded = self.embedding(x)
        packed_embedded =
torch.nn.utils.rnn.pack_padded_sequence(
            embedded, text_lengths
        )
        packed_output, hidden = self.rnn(packed_embedded)
        output, output_lengths =
torch.nn.utils.rnn.pad_packed_sequence(
            packed_output
        )
        return self.fc(hidden.squeeze(0))

INPUT_DIM = 4764
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
```

```
OUTPUT_DIM = 1

model = RNN(
    INPUT_DIM, EMBEDDING_DIM, HIDDEN_DIM, OUTPUT_DIM,
embedding_weights
)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-3)
criterion = torch.nn.BCEWithLogitsLoss()
device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

class LSTM(torch.nn.Module):
    def __init__(
        self,
        input_dim,
        embedding_dim,
        hidden_dim,
        output_dim,
        n_layers,
        bidirectional,
        dropout,
        embedding_weights,
    ):
        super().__init__()
        self.embedding = torch.nn.Embedding.from_pretrained(
            embedding_weights
        )
        self.rnn = torch.nn.LSTM(
            embedding_dim,
            hidden_dim,
            num_layers=n_layers,
            bidirectional=bidirectional,
            dropout=dropout,
        )
        self.fc = torch.nn.Linear(hidden_dim * 2, output_dim)
        self.dropout = torch.nn.Dropout(dropout)

    def forward(self, x, text_lengths):
        embedded = self.embedding(x)
        packed_embedded =
torch.nn.utils.rnn.pack_padded_sequence(
            embedded, text_lengths
        )
        packed_output, (hidden, cell) =
```

```
self.rnn(packed_embedded)
        hidden = self.dropout(
            torch.cat((hidden[-2, :, :], hidden[-1, :, :]),
dim=1)
        )
        return self.fc(hidden.squeeze(0))

INPUT_DIM = padding_value
EMBEDDING_DIM = 100
HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.5

model = LSTM(
    INPUT_DIM,
    EMBEDDING_DIM,
    HIDDEN_DIM,
    OUTPUT_DIM,
    N_LAYERS,
    BIDIRECTIONAL,
    DROPOUT,
    embedding_weights,
)

optimizer = torch.optim.Adam(model.parameters())
criterion = torch.nn.BCEWithLogitsLoss()
device = torch.device("cuda" if torch.cuda.is_available() else
"cpu")

def binary_accuracy(preds, y):
  rounded_preds = torch.round(torch.sigmoid(preds))
  correct = (rounded_preds == y).float()
  acc = correct.sum()/len(correct)
  return acc

def train(model, iterator, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.train()
    for batch in iterator:
        optimizer.zero_grad()
        predictions = model(batch["text"],
batch["length"]).squeeze(1)
```

```
        loss = criterion(predictions, batch["label"])
        acc = binary_accuracy(predictions, batch["label"])
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)

def evaluate(model, iterator, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()
    with torch.no_grad():
        for batch in iterator:
            predictions = model(batch["text"],
batch["length"]).squeeze(1)
            loss = criterion(predictions, batch["label"])
            acc = binary_accuracy(predictions, batch["label"])

            epoch_loss += loss.item()
            epoch_acc += acc.item()

    return epoch_loss / len(iterator), epoch_acc / len(iterator)

batch_size = 128 # Usually should be a power of 2 because it's
the easiest for computer memory.

def iterator(X, y):
  size = len(X)
  permutation = np.random.permutation(size)
  iterate = []
  for i in range(0, size, batch_size):
    indices = permutation[i:i+batch_size]
    batch = {}
    batch['text'] = [X[i] for i in indices]
    batch['label'] = [y[i] for i in indices]

    batch['text'], batch['label'] =
zip(*sorted(zip(batch['text'], batch['label']), key = lambda x:
len(x[0]), reverse = True))
    batch['length'] = [len(utt) for utt in batch['text']]
    batch['length'] = torch.IntTensor(batch['length'])
    batch['text'] =
torch.nn.utils.rnn.pad_sequence(batch['text'], batch_first =
```

```
True).t()
    batch['label'] = torch.Tensor(batch['label'])

    batch['label'] = batch['label'].to(device)
    batch['length'] = batch['length'].to(device)
    batch['text'] = batch['text'].to(device)

    iterate.append(batch)

  return iterate

index_utt = word_vectors.key_to_index

#You've got to determine some labels for whatever you're
training on.
X_train, X_test, y_train, y_test = train_test_split(index_utt,
labels, test_size = 0.2)
X_train, X_val, y_train, y_val = train_test_split(X_train,
y_train, test_size = 0.2)

train_iterator = iterator(X_train, y_train)
validate_iterator = iterator(X_val, y_val)
test_iterator = iterator(X_test, y_test)

print(len(train_iterator), len(validate_iterator),
len(test_iterator))

N_EPOCHS = 25

for epoch in range(N_EPOCHS):
    train_loss, train_acc = train(
        model, train_iterator, optimizer, criterion
    )
    valid_loss, valid_acc = evaluate(model, validate_iterator,
criterion)

    print(
        f"| Epoch: {epoch+1:02} | Train Loss: {train_loss: .3f}
| Train Acc: {train_acc*100: .2f}% | Validation Loss:
{valid_loss: .3f} | Validation Acc: {valid_acc*100: .2f}% |"
    )
#Training on our dataset
| Epoch: 01 | Train Loss:  0.560 | Train Acc:  70.63% |
Validation Loss:  0.574 | Validation Acc:  70.88% |
| Epoch: 05 | Train Loss:  0.391 | Train Acc:  82.81% |
```

```
Validation Loss:  0.368 | Validation Acc:  83.08% |
| Epoch: 10 | Train Loss:  0.270 | Train Acc:  89.11% |
Validation Loss:  0.315 | Validation Acc:  86.22% |
| Epoch: 15 | Train Loss:  0.186 | Train Acc:  92.95% |
Validation Loss:  0.381 | Validation Acc:  87.49% |
| Epoch: 20 | Train Loss:  0.121 | Train Acc:  95.93% |
Validation Loss:  0.444 | Validation Acc:  86.29% |
| Epoch: 25 | Train Loss:  0.100 | Train Acc:  96.28% |
Validation Loss:  0.451 | Validation Acc:  86.83% |
```

Looking above at our classes and instantiations, you should see that the LSTM is not vastly different from the RNN. The only differences in the `init` input variables are `n_layers` (for convenience, you can also specify it with RNNs), `bidirectional`, and `dropout`. Bidirectional allows LSTMs to look ahead in sequences to help with meaning and context, but also drastically helps with multilingual scenarios, as left-to-right languages like English are not the only format for orthography. Dropout, another huge innovation, changes the paradigm of overfitting from being only data-dependent, and helps the model not overfit by turning off random nodes layer-by-layer during training to force all nodes not to correlate to each other. The only differences in the out-of-model parameters is that the best optimizer for an RNN is SGD, like our CBoW, and the LSTM uses Adam (could use any, including AdamW). Below, we define our training loop and train the LSTM. Compare this training loop to the one defined in Listing 2.4 in the `gradient_descent` function.

One of the amazing things demonstrated in the code here is how much quicker the LSTM can learn, compared to previous model iterations, thanks to both bidirectionality and dropout. The previous models, though training faster, take hundreds of epochs to get the same performance as an LSTM in just 25 epochs. The performance on the validation set, as its name implies, adds validity to the architecture, performing inference during training on examples it has not trained on and keeping accuracy fairly close to the training set.

The problems with these models are not as pronounced, manifesting primarily as being incredibly resource-heavy, especially when being applied to longer, more detail-oriented problems, like healthcare and law. Despite the

incredible advantages of Dropout and Bidirectional processing, they both at least double the amount of processing power required to train, so while inference ends up being only 2-3x as expensive as an MLP of the same size, training becomes 10-12x as expensive. They solved exploding gradients nicely, but exploded the compute required to train instead. To combat this a shortcut was devised and implemented which allowed any model, including an LSTM, to figure out which parts of a sequence were the most influential and which parts could be safely ignored, known as attention.

2.2.8 Attention

Attention is a mathematical shortcut for solving larger context windows faster by telling the model through an emergent mathematical formula which parts of an input to consider and how much. This is all based upon an upgraded version of a dictionary, where instead of just Key and Value pairs, a contextual Query is added. We will go more into Attention in later chapters. For now, know that the below code is the 10 steps taken from the original paper, and that it's the big differentiator between older NLP techniques and modern ones.

Attention solves the slowness of training LSTMs, but keeps the high performance on a low number of epochs. There are multiple types of attention as well. The dot product attention method captures the relationships between each word (or embedding) in your query and every word in your key. When queries and keys are part of the same sentences, this is known as bi-directional self-attention. However, in certain cases, it is more suitable to only focus on words that precede the current one. This type of attention, especially when queries and keys come from the same sentences, is referred to as causal attention. Language modeling further improves by masking parts of a sequence and forcing the model to guess what should be behind the mask. Both Dot Product Attention and masked attention are demonstrated with functions below.

Listing 2.8 Multi-Head Attention Implementation

```
import numpy as np
from scipy.special import softmax

# Step 1: Input: 3 inputs, d_model=4
x = np.array([[1.0, 0.0, 1.0, 0.0],
        [0.0, 2.0, 0.0, 2.0],
        [1.0, 1.0, 1.0, 1.0]])

# Step 2: weights 3 dimensions x d_model=4
w_query = np.array([1,0,1],
            [1,0,0],
            [0,0,1],
            [0,1,1]])
w_key = np.array([[0,0,1],
           [1,1,0],
           [0,1,0],
           [1,1,0]])
w_value = np.array([[0,2,0],
             [0,3,0],
             [1,0,3],
             [1,1,0]])

# Step 3: Matrix Multiplication to obtain Q,K,V
## Query: x * w_query
Q = np.matmul(x,w_query)
## Key: x * w_key
K = np.matmul(x,w_key)
## Value: x * w_value
V = np.matmul(x,w_value)

# Step 4: Scaled Attention Scores
## Square root of the dimensions
k_d = 1
attention_scores = (Q @ K.transpose())/k_d

# Step 5: Scaled softmax attention scores for each vector
attention_scores[0] = softmax(attention_scores[0])
attention_scores[1] = softmax(attention_scores[1])
attention_scores[2] = softmax(attention_scores[2])

# Step 6: attention value obtained by score1/k_d * V
attention1 = attention_scores[0].reshape(-1,1)
attention1 = attention_scores[0][0]*V[0]
attention2 = attention_scores[0][1]*V[1]
attention3 = attention_scores[0][2]*V[2]
```

```
# Step 7: summed the results to create the first line of the
output matrix
attention_input1 = attention1 + attention2 + attention3

# Step 8: Step 1 to 7 for inputs 1 to 3
## Because this is just a demo, we'll do a random matrix of the
right dimensions
attention_head1 = np.random.random((3,64))

# Step 9: We train all 8 heads of the attention sub-layer using
steps 1 through 7
## Again, it's a demo
z0h1 = np.random.random((3,64))
z1h2 = np.random.random((3,64))
z2h3 = np.random.random((3,64))
z3h4 = np.random.random((3,64))
z4h5 = np.random.random((3,64))
z5h6 = np.random.random((3,64))
z6h7 = np.random.random((3,64))
z7h8 = np.random.random((3,64))

# Step 10: Concatenate heads 1 through 8 to get the original
8x64 output dimension of the model
Output_attention =
np.hstack((z0h1,z1h2,z2h3,z3h4,z4h5,z5h6,z6h7,z7h8))

# Here's a function that performs all of these steps:
def dot_product_attention(query, key, value, mask, scale=True):
    assert query.shape[-1] == key.shape[-1] == value.shape[-1],
"q,k,v have different dimensions!"
    if scale:
        depth = query.shape[-1]
    else:
        depth = 1
    dots = np.matmul(query, np.swapaxes(key, -1, -2)) /
np.sqrt(depth)
    if mask is not None:
        dots = np.where(mask, dots, np.full_like(dots, -1e9))
    logsumexp = scipy.special.logsumexp(dots, axis=-1,
keepdims=True)
    dots = np.exp(dots - logsumexp)
    attention = np.matmul(dots, value)
    return attention
```

```
# Here’s a function that performs the previous steps but adds
causality in masking
def masked_dot_product_self_attention(q,k,v,scale=True):
    mask_size = q.shape[-2]
    mask = np.tril(np.ones((1, mask_size, mask_size),
dtype=np.bool_), k=0)
    return DotProductAttention(q,k,v,mask,scale=scale)
```

Above, in the full implementation of Attention you may have noticed some terminology you’re familiar with, namely Key and Value, but you may not have been introduced to Query before. Key and Value pairs are familiar because of dictionaries and lookup tables, where we map a set of keys to an array of values. Query should feel intuitive as a sort of search for retrieval. The Query is compared to the Keys, from which a Value is retrieved in a normal operation.

In Attention, the Query and Keys undergo dot product similarity comparison to obtain an attention score, which is later multiplied by the Value in order to get an ultimate score for how much Attention the model should pay to that portion of the sequence. This can get more complex, depending upon your model’s architecture because both encoder and decoder sequence lengths have to be accounted for, but suffice it to say for now that the most efficient way to model in this space is to project all input sources into a common space and compare using dot product for efficiency.

This code explanation was a bit more math-heavy than the previous examples, but it is needed to illustrate the concept. The math behind Attention is truly innovative and has rocketed the field forward. Unfortunately, even with the advantages Attention brings to the process of sequence modeling, with LSTMs and RNNs there were still issues with speed and memory size. You may notice from the code and the math that there is a square root taken, meaning that attention as we use it is quadratic. Since then, there have been various techniques, including subquadratics like Hyena and the Recurrent Memory Transformer (RMT, basically an RNN combined with a transformer) to combat these problems, which we will cover in more detail later. For now, let’s move on to the ultimate application of Attention: the Transformer.

2.3 Attention is All You Need

In the seminal paper, Attention is All You Need[1] Vaswani et al take the mathematical shortcut several steps further, positing that for performance absolutely no recurrence (the “R” in RNN) or any convolutions[2] were needed at all. Instead, they opted to use only Attention and simply specify where Q, K, and V were taken from much more carefully. We’ll dive into this presently. In our review of this diverse range of NLP techniques, we have observed their evolution over time and the ways in which each approach has sought to improve upon its predecessors. From rule-based methods to statistical models and neural networks, the field has continually strived for more efficient and accurate ways to process and understand natural language. Now, we turn our attention to a groundbreaking innovation that has revolutionized the field of NLP: the Transformer architecture. In the following section, we will explore the key concepts and mechanisms that underpin Transformers, and how they have enabled the development of state-of-the-art language models that surpass the performance of previous techniques. We will also discuss the impact of Transformers on the broader NLP landscape and consider the potential for further advancements in this exciting area of research.

2.3.1 Encoders

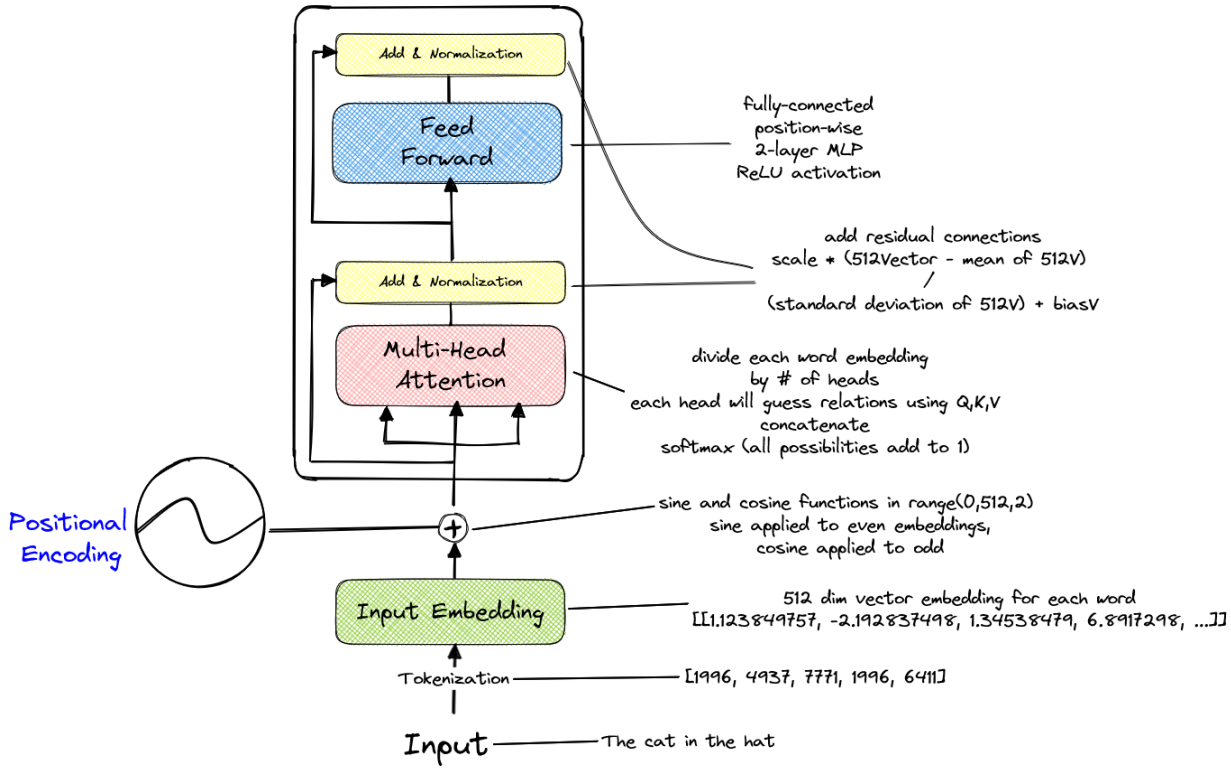
Encoders are the first half of a full transformer model, excelling in areas like classification and feature engineering. One thing Vaswani et al. (2017) figured out is that after the embedding layer inside the encoder, any additional transformations done to the tensors could end up harming their ability to be compared “semantically,” which was the point of the embedding layer. These models rely heavily upon self-attention and a clever positional encoding to manipulate those vectors without significantly decreasing the similarity expressed.

Again, a key thing about embeddings: they are vector representations of data, in our case tokens. Tokens are whatever you pick to represent language. We recommend subwords as a general rule, but you will get a feel for which types of tokens work well where. Consider the sentence, “The cat

in the hat rapidly leapt above the red fox and the brown unmotivated dog." "Red," and "brown," should be semantically similar, and they are similarly represented after the embedding layer, but they fall on positions 10 and 14 respectively in the utterance, assuming that we're tokenizing by word, therefore the positional encoding puts distance between them. However, once the sine and cosine functions[3] are applied, it brings their meaning back to only a little further apart than they were after the encoding, and this encoding mechanism scales brilliantly with recurrence and more data. To illustrate, let's say there was a 99% cosine similarity between [red], and [brown] after embedding. Encoding would drastically reduce that, to around 85-86% similarity. Applying sine and cosine methodologies as described brings their similarity back up to around 96%.

BERT was one of the first architectures to come after the original paper and are examples of encoder-only transformers. BERT is an incredibly powerful model architecture for how small it is that it is still used in production systems today. BERT was the first encoder-only transformer to surge in popularity, showcasing that performing continuous modeling using a transformer results in much better embeddings than Word2Vec. We can see that these embeddings were better because they could be very quickly applied to new tasks and data with minimal training, with human-preferred results over Word2Vec embeddings. This resulted in most people using BERT-based models for few-shot learning tasks on smaller datasets for a while. BERT puts state-of-the-art performance within arms reach for most researchers and businesses with minimal effort required.

Figure 2.5 An Encoder, visualized. Encoders are the first half of the full transformer architecture, and excel in NLU tasks like classification or NER. Encoder models improve upon previous designs by not requiring any priors or recurrence, and use clever positional encoding and multihead attention.



Strengths:

- Classification and hierarchical tasks showcasing understanding
- Blazing fast, considering the long-range dependency modeling
- Builds off of known models, CBoW in Embedding, MLP in Feed Forward, etc.

Weaknesses:

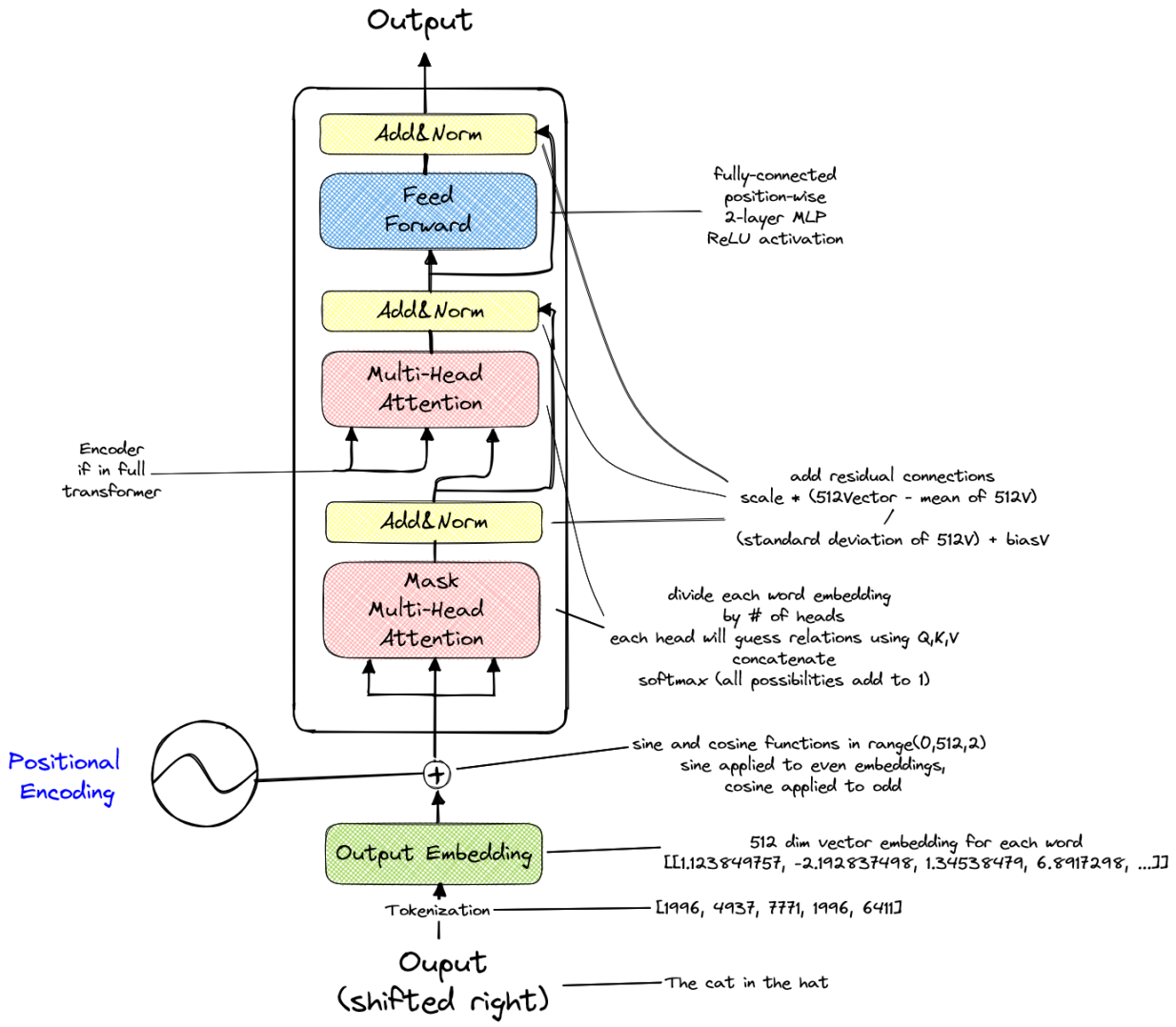
- As suggested, requires lots of data to be effective (although less than RNNs)
- Even more complex architecture

2.3.2 Decoders

Decoder models, as shown below, are larger versions of encoders that have 2 multi-head attention blocks and 3 sum and normalize layers in their base form. They are the 2nd half of a transformer behind an encoder. This results in a model that is very good at masked language modeling and learning and

applying syntax super quickly leading to the almost immediate idea that decoder-only models are needed to achieve Artificial General Intelligence. A useful reduction of encoder vs decoder tasks is that encoders excel in natural language understanding (NLU) tasks, while decoders excel in natural language generation (NLG) tasks. Some examples of decoder-only transformer architectures are the Generative Pretrained Transformer (GPT) family of models. These models follow the logic of transformational generative grammar being completely syntax based, allowing for infinite generation of all possible sentences in a language.[4]

Figure 2.6 a decoder visualized. Decoders are the second half of a full transformer, and they excel in NLG tasks like chatbots and storytelling. Decoders improve upon previous architectures in the same way as encoders, but they add shifting their output one space to the right for next-word generation to help utilize the advantages of multihead self-attention.



Strengths:

- Generating the next token in a sequence (shifted right means taking already-generated tokens into account)
- Building off of both known models and also encoders
- Can be streamed during generation for great UX

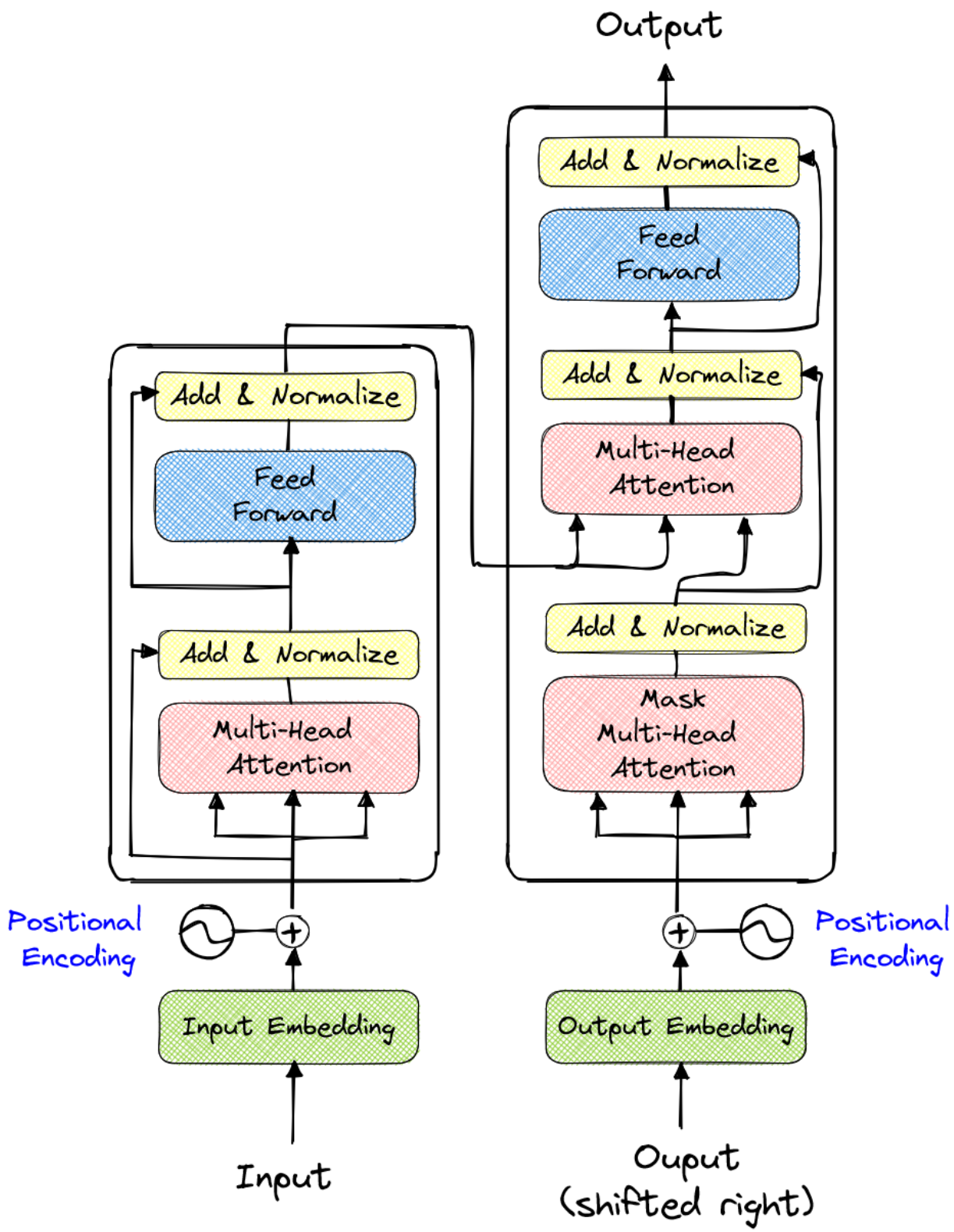
Weaknesses:

- Syntax only models can often struggle to insert the expected or intended meaning (see all “I force an AI to watch 1000 hours of x and generated” memes from 2018-present)
- Hallucinations

2.3.3 Transformers

The full transformer architecture takes advantage of both encoders and decoders, passing the understanding of the encoder into the second Multi-Head Attention block of the decoder before giving output. As each piece of the transformer has a specialty in either understanding or generation, it should feel intuitive for the full product to be best at conditional generation tasks like translation or summarization, where some level of understanding is required before generation occurs. Encoders are geared towards processing input at a high level, and decoders focus more on generating coherent output, the full transformer architecture can successfully understand, then generate based on that understanding. Transformer models have an advantage in that they are built around parallelization, which adds speed that can't currently be replicated in LSTMs. If LSTMs ever get to a point where they can run as quickly as transformers, they may become competitive in the state-of-the-art field. The Text-To-Text Transfer Transformer (T5) family of models are examples of transformers.

Figure 2.7 A full transformer visualized. A full transformer combines both the encoder and the decoder and does well on all of the tasks of each, as well as conditional generation tasks such as summarization and translation. Because transformers are bulkier and slower than each of their halves, researchers and businesses have generally opted to use those halves over the whole thing, despite the speed and memory boosts being minimal.


Output
Add & Normalize
Feed Forward
Add & Normalize
Multi-Head Attention
Add & Normalize
Mask Multi-Head Attention
Add & Normalize
Feed Forward
Add & Normalize
Multi-Head Attention
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Input
Ouput (shifted right)

Strengths:

- Both an encoder and decoder, so is good at everything each of those are good at
- Highly parallelized for speed and efficiency

Weaknesses:

- Memory intensive, but still less than LSTMs of the same size
- Requires large amounts of data and VRAM for training

As you've probably noticed, most of the models we've discussed aren't at all linguistically focused, being heavily syntax-focused, if attempting to model real language at all. Models, even state-of-the-art transformers only have semantic approximations, no pragmatics, no phonetics, and only really utilize morphology during tokenization. This doesn't mean the models can't learn these, nor does it mean that, for example, transformers can't take audio as an input, just that the average usage doesn't. With this in mind, it is nothing short of a miracle that they work as well as they do, and they really should be appreciated as such.

Through this chapter so far we've attempted to highlight where the current limitations are in models, and we will dive into where to go to improve upon them in the rest of this book. One such route is one that's already been and being explored to great success, transfer learning and finetuning large foundational models. This technique came about soon after BERT's initial release, when researchers discovered that although BERT performed generally well on a large number of tasks, if they wanted it to perform better on a particular task or data domain, all they needed to do was retrain the model on data representative of the task or domain, but not from scratch. Take all of the pretrained weights that BERT learned while creating the semantic approximation embeddings on a much larger dataset, then significantly less data is required to get state-of-the-art (SotA) performance on the portion that you need. We've seen this with BERT, and with the GPT family of models as they've come out respectively, and now we're seeing it again to solve exactly the challenges brought up: semantic approximation coverage, domain expertise, availability of data.

2.4 Really Big Transformers

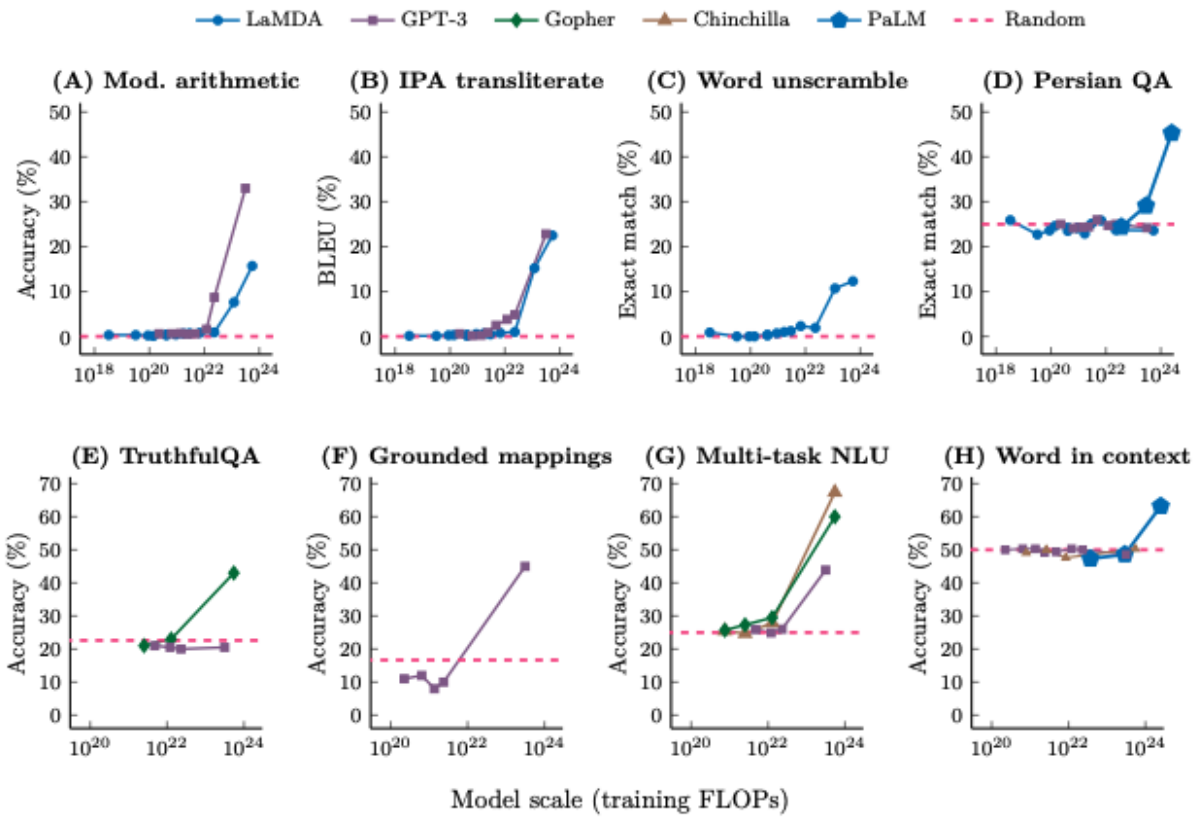
Enter the Large Language Model. Since their introduction transformer based models have continued to only get larger and larger, and not just by their size and number of parameters, but also the size of their training datasets and training cycles has gotten larger and longer as well. If you ever studied machine learning or deep learning during the 2010s, you likely heard the moniker, “more layers doesn’t make the model better.” LLMs prove this both wrong and right. Wrong because their performance is unparalleled, oftentimes even matching smaller models that have been meticulously finetuned on a particular domain and dataset, even the ones trained on proprietary data. Right because of the challenges that come with both training and deploying them.

One of the major differences between LLMs and LMs involves transfer learning and finetuning. Exactly the same as the previously-large LMs, LLMs are pretrained on massive text corpora, enabling them to learn general language features and representations that can be finetuned for specific tasks. Because LLMs are so massive though and their training datasets so large LLMs are able to achieve better performance with less labeled data, which was a significant limitation of earlier language models. Often times you can finetune an LLM to do highly specialized tasks with only a dozen or so examples.

However, what really makes LLMs powerful and has opened the door to widespread business use cases is their ability to do specialized tasks without any finetuning, but just simple prompting. Just give a few examples of what you want in your query and the LLM is able to produce results. This is called few-shot prompting when it’s trained on smaller labeled data sizes, one-shot, when given only one example, and zero-shot, when the task is totally novel. LLMs, especially those trained using RLHF and prompt engineering methodologies, can perform few-shot learning on a whole new level, where they can generalize and solve tasks with only a few examples. This ability is a significant advancement over earlier models that required extensive fine-tuning or large amounts of labeled data for each specific task.

LMs previously have shown promise in the few and zero-shot learning domains, and LLMs have proven that promise to be true. As models have gotten larger we find they are capable of accomplishing new tasks where smaller models can't. We call this emergent behaviors[5] and figure 2.8 demonstrates eight different tasks that LMs couldn't perform better than random, then suddenly once the models got large enough they could.

Figure 2.8 Examples of LLMs demonstrating emergent behaviors when tasked with few-shot prompting tasks after the model scale reaches a certain size.



LLMs have demonstrably great Zero-Shot capabilities as well, which is both due to their vast parameter sizes, and also the main reason for their popularity and viability in the business world. LLMs also exhibit improved handling of ambiguity due to their large size and capacity. They are better at disambiguating words with multiple meanings and understanding the nuances of language, resulting in more accurate predictions and responses.

This isn’t because of an improved ability or architecture as they share their architecture with smaller transformers, but because they have vastly more examples of how people generally disambiguate. LLMs therefore respond with the same disambiguation as is generally represented in the dataset. Thanks to the diverseness of the text data LLMs are trained on, they exhibit increased robustness in handling various input styles, noisy text, and grammatical errors.

Another key difference between LLMs and LMs is input space. A larger input space is important since it makes few-shot prompting tasks that much more viable. Many LLMs have max input sizes of 8000+ tokens (GPT-4 currently sports 32k), and while all the models previously discussed in the chapter could also have input spaces that high, they generally aren’t considered to. We have recently seen a boom in this field as well, with techniques like Recurrent Memory Transformer (RMT) allowing 1,000,000+ token context spaces, which rocket LLMs even more towards proving that bigger models really are always better. LLMs are designed to capture long-range dependencies within text, allowing them to understand context more effectively than their predecessors. This improved understanding enables LLMs to generate more coherent and contextually relevant responses in tasks like machine translation, summarization, and conversational AI.

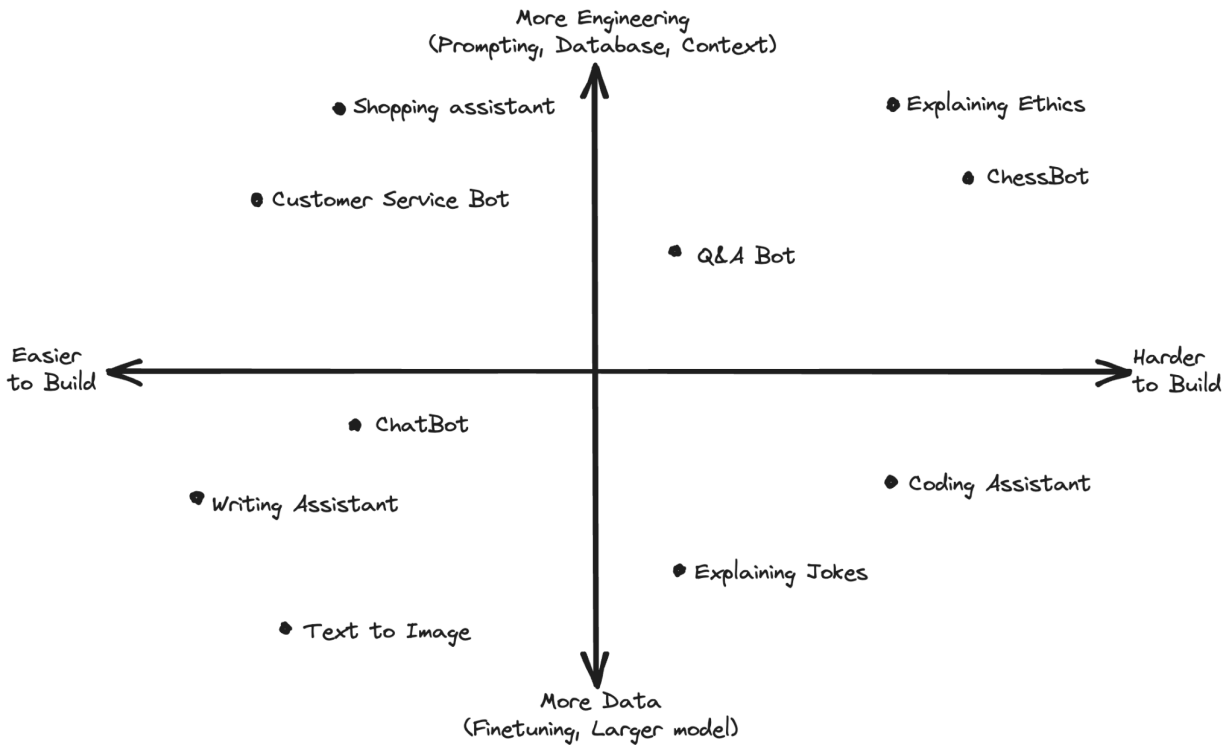
LLMs have revolutionized NLP by offering powerful solutions to problems that were challenging for earlier language models. They bring substantial improvements in contextual understanding, transfer learning, and few-shot learning. As the field of NLP continues to evolve, researchers are actively working to maximize the benefits of LLMs while mitigating all potential risks. Because a better way to approximate semantics hasn’t been found, they make bigger and more dimensional approximations. Because a good way of storing pragmatic context hasn’t been found, LLMs often allow inserting context either into the prompt directly, into a part of the input set aside for context, or even through sharing of databases with the LLM at inference. This doesn’t create pragmatics or a pragmatic system within the models, same as embeddings don’t create semantics, but it allows the model to correctly generate syntax that mimics how humans respond to those pragmatic and semantic stimuli. Phonetics is a place where LLMs could

likely make gigantic strides, either as completely text-free models, or as a text-phonetic hybrid model, maybe utilizing IPA in addition to or instead of text. It is exciting to consider the possible developments that we are watching sweep this field right now.

At this point you should have a pretty good understanding of what LLMs are and some key principles of linguistics that will come in handy when putting LLMs in Production. Mainly, you should be able to now start reasoning what type of products will be easier or harder to build. Consider figure 2.9, tasks in the lower left hand corner like Writing Assistants and ChatBots are LLMs bread and butter. Text generation based on a little context from a prompt are problems that are strictly syntax based, with a large enough model trained on enough data we can do this pretty easily. A Shopping Assistant is pretty similar and rather easy to build as well, however, we are just missing pragmatics. The assistant needs to know a bit more about the world like products, stores, and prices. With a little engineering we can add this information into a database and give this context to the model through prompting.

On the other end consider a ChessBot. LLMs *can* play chess. They aren't any good. They have been trained on chess games and understand that "E4" is a common first move, but their understanding is completely syntactical. LLMs really only understand that the text they should generate should contain a letter between A and H and a number between 1 and 8. Like the Shopping Assistant, they are missing pragmatics and don't have a clear model of the game of chess. In addition, they are also missing semantics. Encoders might help us understand the words "King" and "Queen" are similar to each other, but they don't really help us understand that "E4" is a great move one moment for one player and that same "E4" move is a terrible move the very next moment for a different player. LLMs are also completely lacking knowledge based on phonetics and morphology for chess as well, but these are not as important for this case. Either way, we hope this exercise will better inform you and your team on your next project.

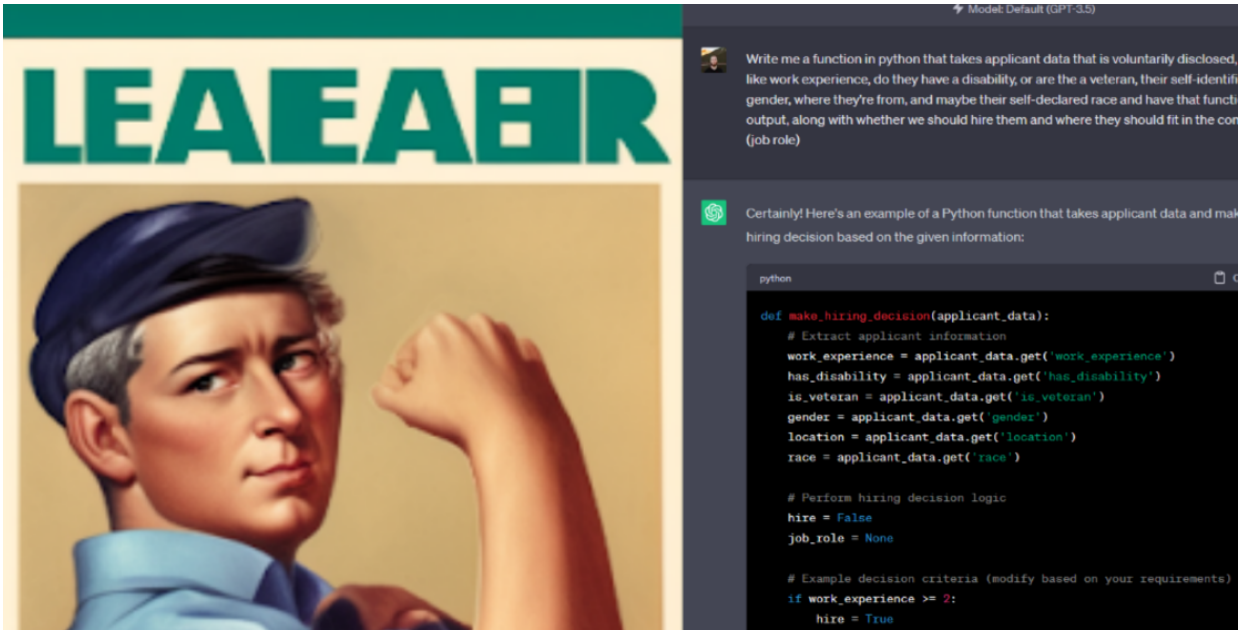
Figure 2.9 How difficult or easy certain tasks are for LLMs and what approaches to solve them.



LLMs have amazing benefits, but with all of these capabilities come some limitations. Foundational LLMs require vast computational resources for training, making them less accessible for individual researchers and smaller organizations. This is being remedied with techniques we'll talk about throughout the book like Quantization, Textual Embeddings, Low-Rank Adaptation, Parameter-Efficient Fine Tuning, and Graph Optimization, but foundation models are still currently solidly out of the average individual's ability to train effectively. Beyond that, there are concerns that the energy consumption associated with training LLMs could have significant environmental impact and problems associated with sustainability. This is a complex issue largely out of the scope of this book, but we would be remiss not to bring it up. Last, but not least, since LLMs are trained on large-scale datasets containing real-world text, they may learn and perpetuate biases present in the data, leading to ethical concerns, not by the fault of the researchers or the algorithms, more because real-world people aren't censoring themselves to provide optimal unbiased data. For example, if you ask a text-to-image diffusion LLM to generate 1000 images of "leader," 99% of the images feature men, and 95% of the images feature people with white skin. The concern here isn't that men or white people aren't or shouldn't be

depicted as leaders, rather that it shows that the model isn't truly representing the world accurately.

Figure 2.10 Midjourney 5, which is currently the most popular text2img model on the market, when prompted with only one token, "Leader,"(Shown Left) changed a well-known popular feminist icon, Rosie the Riveter into a male depiction. ChatGPT (Shown Right) writes a function to place you in your job based on your race, gender, and age. These are examples of unintended outputs.



Sometimes, more nuanced bias is brought out, for example, in the Midjourney example demonstrated in Figure 2.10 a popular feminist icon "Rosie the Riveter," without being prompted at all (the only prompt given to the model was the word "leader") was changed to a man. The model didn't think about this change at all, it just determined during its sampling steps that the prompt "leader," looks more like a man. Many people will argue about what "good," and "bad," mean in this context, and instead of entering that discussion we'll simply talk about what accurate means. LLMs are trained on a plethora of data with the purpose of returning the most accurate representations possible. When they are still unable to return accurate representations, especially with their heightened abilities to disambiguate, we can view that as bias that is harmful to the model's ability to fulfill its purpose. Later we will discuss techniques to combat this harmful bias, not

for any political purpose, but to allow you as an LLM creator to get the exact outputs that you intend and minimize the number of outputs that you do not intend.

Alright, we've been building up to this the entire chapter let's go ahead and run our first LLM! In listing 2.9 we download the Bloom model, one of the first open source LLMs to be created, and generate text! Very exciting stuff.

Listing 2.9 Running our first LLM

```
from transformers import AutoModelForCausalLM, AutoTokenizer

MODEL_NAME = "bigscience/bloom"

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForCausalLM.from_pretrained(MODEL_NAME)

prompt = "Hello world! This is my first time running an LLM!"

input_tokens = tokenizer.encode(prompt, return_tensors="pt",
padding=True)
generated_tokens = model.generate(input_tokens,
max_new_tokens=20)
generated_text = tokenizer.batch_decode(generated_tokens,
skip_special_tokens=True)
print(generated_text)
```

Did you try to run it?!? If you did, you probably just crashed your laptop. Oopsie! Forgive me for a little harmless MLOps hazing, but getting some first-hand experience on how large these models can get and how difficult they can be to run is helpful experience to have. We will be talking more about the difficulties of running LLMs and give you some of the tools you need to actually run this in the next chapter. If you don't want to wait and would like to get a similar but much smaller LLM running change the model name to `"bigscience/bloom-3b"` and run it again. It should work just fine this time on most hardware.

All in all, LLMs are an amazing technology that allow our imaginations to run wild with possibility, and deservedly so. The number one use case for

considering an LLM over a smaller LM is for when few-shot capabilities will come into play for whoever the model will be helping, such as helping a CEO when raising funds or a software engineer when writing code. They have this ability precisely because of their size. The larger number of parameters in LLMs directly enable the ability to generalize over smaller spaces in larger dimensions. In this chapter, we've hit the lesser-known side to LLMs, the linguistic and language modeling side. In the next chapter, we'll cover the other half, the MLOps side, where we dive into exactly how that large parameter size affects the model and the systems designed to support that model and make it accessible to the customers or employees the model is intended for.

2.5 Summary

- The five components of linguistics are phonetics, syntax, semantics, pragmatics, and morphology.
    - Phonetics can be added through a multimodal model that processes audio files and is likely to improve LLMs in the future, but current datasets are too small.
    - Syntax is what current models are good at.
    - Semantics is added through the embedding layer.
    - Pragmatics can be added through engineering efforts.
    - Morphology is added in the tokenization layer.
- Language does not necessarily correlate with reality. Understanding the process that people use to create meaning outside of reality is useful to training meaningful (to people) models.
- Proper tokenization can be a major hurdle due to too many <UNK> tokens, especially when it comes to specialized problems like code or math.
- Multilingual processing has always outperformed monolingual processing, even on monolingual tasks without models.
- Each language model type has been built specifically to combat the weaknesses of the previous models, as opposed to trying to solve for particular features of language.
- Language modeling has seen an exponential increase in efficacy, correlating to how linguistics-focused the modeling has been.

- Attention is a mathematical shortcut for solving larger context windows faster and is the backbone of modern architectures - Encoders, Decoders, and Transformers.
    - Encoders improve the semantic approximations in embeddings.
    - Decoders are best at text generation.
    - Transformers combine the two.
- Larger models demonstrate emergent behavior suddenly being able to accomplish tasks they couldn't before.

[1] Vaswani et al 2017 Attention Is All You Need
https://arxiv.org/abs/1706.03762

[2] We didn't go over these because they aren't good for NLP, but they are popular especially in computer vision

[3] Not a math or history book

[4] See Appendix A

[5] J. Wei et al., "Emergent Abilities of Large Language Models," Transactions on Machine Learning Research, Aug. 2022, Available: https://openreview.net/forum?id=yzkSU5zdwD

3 Large Language Model Operations: Building a Platform for LLMs

This chapter covers

- Overview of Large Language Models Operations
- Deployment challenges
- Large Language Models best practices
- Required Large Language Model infrastructure

As we learned in the last chapter when it comes to transformers and Natural Language Processing (NLP), bigger is better, especially when it's linguistically informed. However, bigger models come with bigger challenges because of their size, regardless of their linguistic efficacy, thus requiring us to scale up our operations and infrastructure to handle these problems. In this chapter we'll be looking into exactly what those challenges are, what we can do to minimize them, and what architecture can be set up to help solve these challenges.

3.1 Introduction to Large Language Models Operations

What is Large Language Models Operations (LLMOps)? Well, since I'm one to focus on practicality over rhetoric, I'm not going to dive into any fancy definitions that you'd expect in a text book, but let me simply say it's Machine Learning Operations (MLOps) that has been scaled to handle LLMs. Let me also say, scaling up is hard. One of the hardest tasks in software engineering. Unfortunately, too many companies are running rudimentary MLOps set-ups, and don't think for a second that they will be able to just handle LLMs. That said, the term "LLMOps," may not be needed. It has yet to show through as sufficiently different from core MLOps, especially considering they still have the same bones. If this book were a dichotomous key, MLOps and LLMOps would definitely be in the same genus, and only time will tell about whether they are the same species.

Of course by refusing to define LLMOps properly, I might have traded one confusion for another, so let's take a minute to describe MLOps.

MLOps is the field and practice of reliably and efficiently deploying and maintaining machine learning models in production. This includes, and indeed requires, managing the entire machine learning lifecycle from data acquisition and model training to monitoring and termination. A few principles required to master this field include workflow orchestration, versioning, feedback loops, Continuous Integration and Continuous Deployment (CI/CD), security, resource provisioning, and data governance. While there are often personnel who specialize in the productionizing of models, often with titles like ML Engineers, MLOps Engineers or ML Infrastructure Engineer, the field is a large enough beast it often abducts many other unsuspecting professionals to work in it who hold titles like Data Scientist or DevOps Engineer–oftentimes against their knowledge or will; leaving them kicking and screaming that “it’s not their job”.

3.2 Operations Challenges with Large Language Models

So why have a distinction at all? If MLOps and LLMOps are so similar, is LLMOps just another fad opportunists throw on their resume? Not quite. In fact, I think it’s quite similar to the term Big Data. When the term was at its peak popularity, people with titles like Big Data Engineer used completely different tool sets and developed specialized expertise that were necessary in order to handle the large datasets. LLMs come with a set of challenges and problems you won’t find with traditional machine learning systems. A majority of these problems extend almost exclusively because they are so big. Large models are large! We hope to show you that LLMs truly earn their name. Let’s take a look at a few of these challenges, so we can appreciate the task ahead of us when we start talking about deploying an LLM.

3.2.1 Long download times

Back in 2017 when I was still heavily involved as a Data Scientist, I decided to try my hand at reimplementing some of the most famous computer vision models at the time AlexNet, VGG19, and ResNet. I figured this would be a good way to reinforce my understanding of the basics with some practical hands-on experience. Plus, I had an ulterior motive, I had just built my own rig with some NVIDIA GeForce 1080 TI GPUs—which was state of the art at the time—and thought this would be a good way to break them in. The first task: download the ImageNet dataset. The ImageNet dataset was one of the largest annotated datasets available containing millions of images rounding out to a file size of a whopping ~150GB! Working with it was proof that you knew how to work with “Big Data " which was still a trendy word and an invaluable skill set for a data scientist at the time. After agreeing to the terms and gaining access, I got my first wakeup call. Downloading it took an entire week.

Large models are large. I don’t think I can overstate that. You’ll find throughout this book that fact comes with many additional headaches and issues for the entire production process, and you have to be prepared for it. In comparison to the ImageNet dataset, the Bloom LLM model is 330GB, more than twice the size. Most readers I’m guessing haven’t worked with either ImageNet or Bloom, so for comparison Call of Duty: Modern Warfare, one of the largest games at the time of writing is 235 GB. Final Fantasy 15 is only 148 GB, which you could fit two of into the model with plenty of room to spare. It’s just hard to really comprehend how massive LLMs are. We went from 100 million parameters in models like BERT and took them to billions of parameters. If you went on a shopping spree and spent $20 a second (or maybe just left your AWS EC2 instance on by accident) it’d take you half a day to spend a million dollars; it would take you 2 years to spend a billion.

Thankfully it doesn’t take two weeks to download Bloom because unlike ImageNet, it’s not hosted on a poorly managed University server and it also has been sharded into multiple smaller files to allow downloading in parallel, but it will still take an uncomfortably long time. Consider a scenario where you are downloading the model under the best conditions. You’re equipped with a gigabit speed fiber internet connection and you were magically able to

dedicate the entire bandwidth and I/O operations of your system and the server to it, it'd still take over 5 minutes to download! Of course, that's under the best conditions. You probably won't be downloading the model under such circumstances, with modern infrastructure you can expect it to take on the order of hours. When my team first deployed Bloom it took an hour and a half to download it. Heck, it took me an hour and half to download The Legend of Zelda: Tears of the Kingdom and that's only 16GB, so I really can't complain.

3.2.2 Longer Deploy Times

Just downloading the model is a long enough time frame to make any seasoned developer shake, but deployment times are going to make them keel over and call for medical attention. A model as big as Bloom can take 30-45 minutes just to load the model into GPU memory, at least those are the time frames my team first saw. Not to mention any other steps in your deployment process that can add to this. Indeed, with GPU shortages, it can easily take hours just waiting for resources to free up—more on that in a minute.

What does this mean for you and your team? Well for starters, I know lots of teams who deploy ML products often simply download the model at runtime. That might work for small sklearn regression models, but it isn't going to work for LLMs. Additionally, you can take most of what you know about deploying reliable systems and throw it out the window (but thankfully not too far). Most modern day best practices for software engineering assume you can easily just restart an application if anything happens, and there's a lot of rigmarole involved to ensure your systems can do just that. But with LLMs it can take seconds to shut down, but potentially hours to redeploy making this a semi-irreversible process. Like picking an apple off a tree, it's easy to pluck one off, but if you bite into it and decide it's too sour, you can't just attach it back onto the tree so it can continue to ripen. You'll just have to wait awhile for another to grow.

While not every project requires deploying the largest models out there, you can expect to see deployment times measured in minutes. These longer

deploy times make scaling down right before a surge of traffic a terrible mistake, as well as figuring out how to manage bursty workloads difficult. General CI/CD methodologies need to be adjusted since rolling updates take longer leaving a backlog piling up quickly in your pipeline. Silly mistakes like typos or other bugs often take longer to notice, and longer to correct.

3.2.3 Latency

Along with increases in model size often come increases in inference latency. This is obvious when stated, but more parameters equates to more computations, and more computations means longer inference wait times. However, this can't be underestimated. I know many people who downplay the latency issues because they've interacted with an LLM chatbot and the experience has felt smooth. Take a second look though, and you'll notice that it is returning one word at a time which is streamed to the user. It feels smooth because the answers are coming in faster than a human can read, but a second look helps us realize this is just a UX trick. LLMs are still too slow to be very useful for an autocomplete solution for example, where responses have to be blazingly fast. Building it into a data pipeline or workflow that reads a large corpus of text and then tries to clean it or summarize it, may also be prohibitively slow to be useful or reliable.

There are also many less obvious reasons for their slowness. For starters, LLMs are often distributed across multiple GPUs, which adds extra communication overhead. As discussed later in this chapter in section 3.3.2 they are distributed in other ways, often even to improve latency, but any distribution adds additional overhead burden. In addition, LLMs latency is severely impacted by completion length, meaning the more words it uses to return a response, the longer it takes. Of course, completion length also seems to improve accuracy. For example, using prompt engineering techniques like Chain of Thought (CoT) we ask the model to think about a problem in a step-by-step fashion which has shown to improve results for logic and math questions but also increases the response length and latency time significantly.

3.2.4 Managing GPUs

To help with these latency issues we usually want to run them in GPUs. If we want to have any success training LLMs we'll need GPUs for that as well, but this all adds additional challenges many underestimate. Most web services and many ML use cases can be done solely on CPUs. Not so with LLMs. Partly because of GPUs' parallel processing capabilities offering a solution to our latency problems, and partly because of the inherent optimization GPUs offer in the linear algebra, matrix multiplications and tensor operations that's happening under the hood. For many, stepping into the realm of LLMs, this requires utilizing a new resource and extra complexity. Many brazenly step into this world acting like it's no big deal, but they are in for a rude awakening. Most system architectures and orchestrating tooling available like Kubernetes, assume your application will run with CPU and memory alone. While they often support additional resources like GPUs, this is often an afterthought. You'll soon find you'll have to rebuild containers from scratch and deploy new metric systems.

One aspect of managing GPUs most companies are never prepared for is that they tend to be rare and limited. For the last decade it seems that we have gone in and out of a global GPU shortage. They can be extremely difficult to provision for companies looking to stay on premise. I've spent lots of time in my career working with companies who chose to stay on premise for a variety of reasons. One of the things they had in common is that they never had GPUs on their servers. When they did, they were often purposely difficult to access except for a few key employees.

If you are lucky enough to be working in the Cloud a lot of these problems are solved, but there is no free lunch here either. My team has often gone chasing their tail trying to help data scientists struggling to provision a new GPU workspace, running into obscure ominous errors like "`scale.up.error.out.of.resources`". Only to discover that these esoteric readings indicate all the GPUs of a selected type in the entire region are being utilized and none are available. CPU and Memory can often be treated as infinite in a datacenter, GPU resources, however, cannot. Sometimes you can't expect them at all. Most data centers only support a subset of instance or GPU types. Which means you may be forced to set up your application in a region further away from your user base increasing latency. Of course, I'm

sure you can work with your cloud provider when looking to expand your service to a new region that doesn’t currently support it, but you might not like what you hear based on timelines and cost. Ultimately, you’ll run into shortage issues no matter where you choose to run, on-prem or in the cloud.

3.2.5 Peculiarities of Text Data

LLMs are the modern day solution to NLP. NLP is one of the most fascinating branches of ML in general because it primarily deals with text data, which is primarily a qualitative measure. Every other field deals with quantitative data. We have figured out a way to encode our observations of the world into a direct translation of numerical values. For example, we’ve learned how to encode heat into temperature scales and measure them with thermometers and thermocouples or we can measure pressure with manometers and gauges and put it into pascals.

Computer Vision and the practice of evaluating images is often seen as qualitative, but the actual encoding of images into numbers is a solved problem. Our understanding of light has allowed us to break images apart into pixels and assign them RGB values. Of course this doesn’t mean CV is by any means solved, there’s still lots of work to do to learn how to identify the different signals in the patterns of the data. Audio data is another that’s often considered qualitative. How does one compare two songs? But we can measure sound and speech, directly measuring the sound wave's intensity in decibels and frequency in hertz.

Unlike other fields that encode our physical world into numerical data, text data is looking at ways to measure the ephemeral world. After all, text data is our best effort of encoding our thoughts, ideas and communication patterns. While yes, we have figured out ways to turn words into numbers, we haven’t figured out a direct translation. Our best solutions to encode text and create embeddings are just approximations at best, in fact we use machine learning models to do it! An interesting aside to this is that numbers are also text and a part of language. If we want models that are better at math we need a more meaningful way to encode these numbers. Since it’s all made up, when we try to encode text numbers into machine-readable

numbers we are creating a system attempting to reference itself recursively in a meaningful way. Not an easy problem to solve!

Because of all this, LLMs (and all NLP solutions) have unique challenges. For example, monitoring. How do you catch data drift in text data? How do you measure “correctness”? How do you ensure cleanliness of the data? These types of problems are difficult to define let alone solve.

3.2.6 Token Limits Create Bottlenecks

One of the big challenges for those new to working with LLMs is dealing with the token limits. The token limit for a model is the maximum number of tokens that can be included as an input for a model. The larger the token limit, the more context we can give the model to improve its success as accomplishing the task. Everyone wants them to be higher, but it’s not that simple. These token limits are defined by two problems, the first being the memory and speed our GPUs have access to, and the second being the nature of memory storage in the models themselves.

The first one seems unintuitive, why couldn’t we just increase the GPU memory? The answer is complex, we can, but stacking more layers in the GPU to take into account more GB at once slows down the GPU’s computational ability as a whole. GPU manufacturers right now are working on new architectures and ways to get around this problem. The second one is more fascinating because we find that increasing the token limits actually just exacerbates the mathematical problems under the hood. Let me explain. Memory storage within an LLM itself isn’t something we think about often. We call that mechanism Attention, which we discussed in depth in section 2.2.7. What we didn’t discuss was that Attention is a quadratic solution—as the number of tokens increase the number of calculations required to compute the attentions scores between all the pairs of tokens in a sequence scales quadratically with the sequence length. In addition, within our gigantic context spaces and since we are dealing with quadratics, we’re starting to hit problems where the only solutions involve imaginary numbers which is something that can cause models to behave in unexpected ways. This is likely one of the reasons why LLMs hallucinate.

These problems have real implications and impact application designs. For example, when my team upgraded from GPT3 to GPT4 we were excited to have access to a higher token limit, but we soon found this led to longer inference times and subsequently a higher timeout error rate. In the real world, it's often better to get a less accurate response quickly than to get no response at all because the promise of a more accurate model often is just that, a promise. Of course, deploying it locally where you don't have to worry about response times you'll likely find your hardware a limiting factor. For example, LLaMA was trained with 2048 tokens but you'll be lucky to take advantage of more than 512 of that when running with a basic consumer GPU as you are likely to see Out-of-Memory (OOM) errors or even the model simply just crashing.

A gotcha, which is likely to catch your team by surprise and should be pointed out now is that different languages have different tokens per character. Take a look at Table 3.1, where we compare converting the same sentence in different languages to tokens using OpenAI's cl100k_base Byte Pair Encoder. Just a quick glance reveals that LLMs typically favor the English language in this regard. In practice, this means if you are building a chatbot with an LLM, your English users will have greater flexibility in their input space than Japanese users leading to very different user experiences.

Table 3.1 Comparison of Token Counts in different languages

Language	String	Characters	Tokens
English	The quick brown fox jumps over the lazy dog	43	9
French	Le renard brun rapide saute par-dessus le chien paresseux	57	20
Spanish	El rápido zorro marrón salta sobre el perro perezoso	52	22
Japanese	素早い茶色のキツネが怠惰な犬を飛び越える	20	36
Chinese (simplified)	敏捷的棕色狐狸跳过了懒狗	12	28

If you are curious as to why this is, it is due to text encodings, which is just another peculiarity of working with text data as discussed in the previous section. Consider Table 3.2 where we show several different characters and their binary representation in UTF-8. English characters can almost exclusively be represented with a single byte being included in the original ASCII standard computers were originally built on, while most other characters require 3 or 4 bytes. Because it takes more memory it also takes more token space.

Table 3.2 Comparison of byte lengths for different currency characters in UTF-8.

Character	Binary UTF-8	Hex UTF-8
$	00100100	0x24
£	11000010 10100011	0xc2 0xa3
¥	11000010 10100101	0xc2 0xa5
₲	11100010 10000010 10100000	0xe2 0x82 0xa0
𐐷	11110000 10011111 10010010 10110000	0xf0 0x9f 0x92 0xb0

Increasing the token limits has been an ongoing research question since the popularization of transformers, and there are some promising solutions still in research phases like Recurrent Memory Transformers (RMT)[1]. We can expect to continue to see improvements in the future and hopefully this will become naught but an annoyance.

3.2.7 Hallucinations Cause Confusion

So far we've been discussing some of the technical problems a team faces when deploying an LLM into a production environment, but nothing compares to the simple problem that LLMs tend to be wrong. They tend to be wrong a lot. Hallucinations is a term coined to describe occurrences when LLM models will produce correct sounding results that are wrong. For example, book references or hyperlinks that have the form and structure of

what would be expected, but are nevertheless, completely made up. As a fun example I asked for books on LLMs in Production from the publisher Manning (a book that doesn't exist yet since I'm still writing it). I was given the following suggestions: Machine Learning Engineering in Production by Mike Del Balso and Lucas Serveén which could be found at https://www.manning.com/books/machine-learning-engineering-in-production and Deep Learning for Coders with Fastai and PyTorch by Jeremy Howard and Sylvain Gugger which could be found at https://www.manning.com/books/deep-learning-for-coders-with-fastai-and-pytorch. The first book is entirely made up. The second book is real however it's not published by Manning. In each case the internet addresses are entirely made up. These URLs are actually very similar to what you'd expect in format if you were browsing Mannings website, and should return 404 errors if you visit them.

One of the most annoying aspects of hallucinations is that they are often surrounded by confident sounding words. LLMs are terrible at expressing uncertainty, in large part because of the way they are trained. Consider the case "2+2=". Would you prefer it to respond, "I think it is 4" or just simply "4"? Most would prefer to simply get the correct "4" back. This bias is built in as models are often given rewards for being more correct or at least sounding like it.

There are various explanations as to why hallucination occurs, but the most truthful answer is that we don't know if there's just one cause. It's likely a combination of several things, thus there isn't a good fix for it yet. Nevertheless, being prepared to counter these inaccuracies and biases of the model are crucial to provide the best user experience for your product.

3.2.8 Bias and Ethical Considerations

Just as concerning as the model getting things wrong is when it gets things right in the worst possible way. For example, allowing it to encourage users to commit suicide[2], teaching your users how to make a bomb[3], or participating in sexual fantasies involving children[4]. These are extreme

examples, but prohibiting the model from answering such questions is undeniably vital to success.

LLMs are trained on vast amounts of text data which is also their primary source of bias. Because we've found that larger datasets are just as important as larger models in producing human-like results, most of these datasets have never truly been curated or filtered to remove harmful content, instead choosing to prioritize size and a larger collection. Cleaning the dataset is often seen as prohibitively expensive, requiring humans to go in and manually verify everything, but there's a lot that could be done with simple regular expressions and other automated solutions. By processing these vast collections of content and learning the implicit human biases, these models will inadvertently perpetuate them. These biases range from sexism and racism to political preferences and can cause your model to inadvertently promote negative stereotypes and discriminatory language.

3.2.9 Security Concerns

As with all technology we need to be mindful of security. LLMs have been trained on large corpus of text and some of it could be harmful or sensitive and shouldn't be exposed so steps should be taken to protect this data from being leaked. The bias and ethical concerns from the last section are good examples of conversations you don't want your users to be having, but you could also imagine fine-tuning a model on your company's data and potentially have secrets lost inadvertently if proper precautions aren't taken.

One should be aware that LLMs are susceptible to adversarial attacks like prompt injections. Prompt injections are attacks done by a user to trick the LLM to ignore instructions given to it and generate undesired content. For example, if you ask ChatGPT what its gender is it appropriately replies that as an AI language model, it doesn't have a gender, however, with clever prompting you may be able to bypass these protocols and get it to reveal one. While this example is harmless, I've seen others successfully extract API keys and other secrets from an LLM, run code in non-protected environments, steal environment variables and traverse local file systems where the model is served. Not to mention the plethora of examples of users

using prompting to jailbreak or bypass protocols put in place for ethical considerations outlined in the section above. An interesting aside to this, LLMs are good at inventing fake secrets! Even successful prompt injection attacks can often fail due to LLM hallucinations which can have funny consequences.

For example, using a simple prompt to ask for Coca-Cola's secret recipe simply returned the boring result, "I cannot divulge trade secrets or any proprietary information." Pretty lame if you ask me. With a little trickery and some prompt injection I was able to get it to spill its secrets which of course my coauthor and I decided to try–after confirming with a food scientist it wouldn't poison us. After following the instructions exactly, I was surprised to find we actually got a pretty tasty drink, but it tasted nothing like Cola. While the recipe looks legitimate, nuances like quantities, cook times, and even the need to strain the syrup were all off. Guess, I'll have to hold off taking over the beverage industry until I get the real recipe.

Coca-Cola Recipe Hallucinated from Prompt Injection

Homemade Coca-Cola Recipe

==========

Disclaimer: The original Coca-Cola recipe is a highly guarded trade secret. The following recipe is an approximation for creating a homemade cola drink that resembles the taste of Coca-Cola.

Ingredients:

- 3 cups of water
- 2 cups granulated sugar
- 1/8 tsp food-grade citric acid
- 3/4 tsp caramel color (liquid)
- 1/2 tsp caffeine (powder or liquid form)
- 1 1/2 tsp cola extract
- 1/8 tsp vanilla extract

- 1 1/2 cups carbonated water (store-bought or using a home soda maker)
- A pinch of ground cinnamon (optional)
- A pinch of ground nutmeg (optional)

Instructions:

1. In a large saucepan, bring the 3 cups of water to a gentle simmer.
2. Slowly stir in the granulated sugar, ensuring that it fully dissolves.
3. Add the citric acid, caramel color, caffeine, cola extract, and vanilla extract to the saucepan. Mix well.
4. Stir in the optional spices if desired: cinnamon and nutmeg.
5. Simmer the mixture for about 20 minutes, stirring occasionally.
6. Remove the saucepan from the heat and let it cool to room temperature.
7. Once cooled, strain the syrup to remove any undissolved solids.
8. To make the final drink, mix 1 part syrup with 3 parts carbonated water. Adjust to taste depending on your preference for sweetness and carbonation.

Variations:

- You can experiment with different amounts of caramel color, caffeine, and cola extract to adjust the flavor and appearance of your homemade cola.
- To create different cola flavor profiles, experiment with adding additional spices (such as star anise, orange or lemon peel, or cardamom), essential oils, or fruit extracts.
- Consider using a natural sweetener, like honey or agave syrup, as an alternative to granulated sugar.

Historical context and interesting facts:

- Coca-Cola was first introduced in 1886 by pharmacist Dr. John S. Pemberton, who aimed to create a unique and refreshing beverage that could also relieve fatigue and headaches.
- The original Coca-Cola formula contained trace amounts of cocaine extracted from the coca leaf, which was later replaced with caffeine in the early 20th century.

- In 1985, Coca-Cola introduced "New Coke," a reformulation aimed at competing with Pepsi. However, the taste change was met with backlash, and the company returned to its original formula just three months later, rebranded as "Coca-Cola Classic."

Result(s): You can now create your own homemade version of cola, resembling the taste of Coca-Cola. Enjoy and feel free to experiment with different spices, extracts, and sweeteners to create your own unique flavor profile.

Another security concern involves pickle injections. Pickle is a library in Python to serialize objects and often used to serialize ML models. It serializes them into a byte-stream that contains opcodes that are executed one-by-one as it is deserialized. It’s a fast and easy way to share large objects. Pickle injections involve the process of corrupting this byte-stream, often injecting malware over the wire when the model is transferred over an insecure network. This is especially concerning for large models that take a long time to download, as it makes it easier for a third party to intercept the transfer and inject malicious code. If this happens, the code that is injected can potentially give the attackers access to your system. This can happen when attempting to use the model during inference, as the harmful code will execute if it is not detected and properly removed. It is important to take precautions such as using secure networks and verifying the integrity of the model before use to prevent this type of attack.

3.2.10 Controlling Costs

Working with LLMs involves various cost-related concerns. The first as you probably gathered by now is infrastructure costs, which include high-performance GPUs, storage, and other hardware resources. We talked about how GPUs are harder to procure, that also unfortunately means they are more expensive. Mistakes like leaving your service on have always had the potential to rack up the bills, but with GPU’s in the mix this type of mistake is even more deadly. These models also demand significant computational power, leading to high energy consumption during both training and inference. On top of all this, their longer deploy times means we are often

running them even during low traffic to handle bursty workloads or anticipated future traffic. Overall this leads to higher operational costs.

Additional costs include, managing and storing vast amounts of data used to train or fine-tune as well as regular maintenance, such as model updates, security measures, and bug fixes, can be financially demanding. As with any technology used for business purposes, managing potential legal disputes and ensuring compliance with regulations is a concern. Lastly, investing in continuous research and development to improve your models and give you a competitive edge will be a factor.

We talked a bit about the technical concerns when it comes to token limits, and these are likely to be solved, but what we didn’t discuss was the cost limitations as most API’s charge on a token basis. This makes it more expensive to send more context and use better prompts. It also makes it a bit harder to predict costs since while you can standardize inputs, you can’t standardize outputs. You never can be too sure how many tokens will be returned, making it difficult to govern. Just remember with LLMs, it is as important as ever to ensure proper cost engineering practices are implemented and followed to ensure costs never get away from you.

3.3 Large Language Model Operations Essentials

Now that we have a handle of the type of challenge we are grappling with, let's take a look at all the different LLMOps practices, tooling, and infrastructure to see how different components help us overcome these obstacles. First off, let's dive into different practices starting with compression where we will talk about shrinking, trimming, and approximating to get models as small as we can. We will then talk about distributed computing which is needed to actually make things run since the models are so large they rarely fit into a single GPU’s memory. After we are finished with those we will venture into infrastructure and tooling needed to make it all happen in the next section.

3.3.1 Compression

As you were reading about the challenges of LLMs in the last section, you might have asked yourself something akin to, “If the biggest problems from LLMs come from their size, why don’t we just make them smaller?” If you did, congratulations! You are a genius and compression is the practice of doing just that. Compressing models as small as we can make them will improve deployment time, reduce latency, scale down the number of expensive GPUs needed, and ultimately save money. However, the whole point of making the models so stupefyingly gargantuan in the first place was because it made them better at what they do. We need to be able to shrink them, without losing all the progress we made by making them big in the first place.

This is far from a solved problem, but there are multiple different ways to approach the problem with different pros and cons to each method. We’ll be talking about several of the methods, starting with the easiest and most effective.

Quantizing

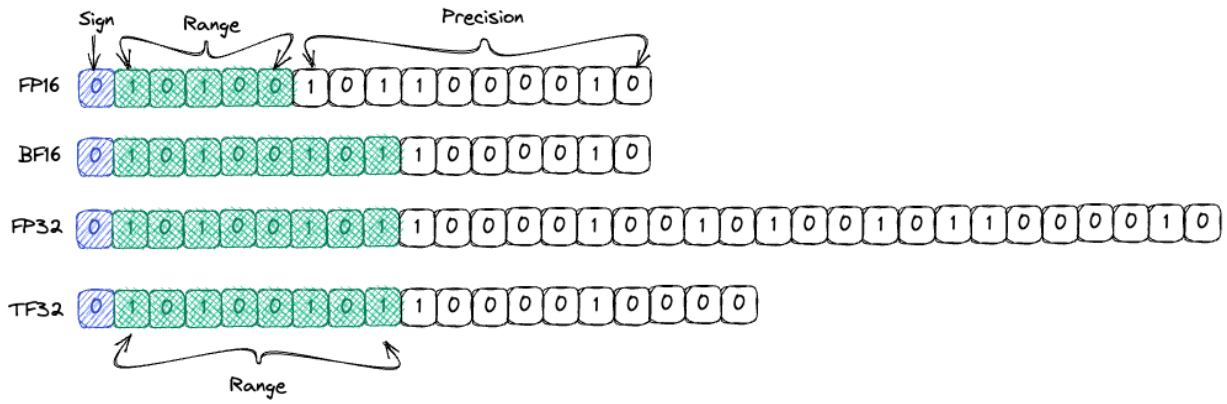
Quantizing is the process of reducing precision in preference of lowering the memory requirements. This tradeoff makes intuitive sense. When I was in college, we were taught to always round our numbers to the precision of your tooling. Pulling out a ruler and measuring my pencil, you wouldn’t believe me if I stated the length was 19.025467821973739cm. Even if I used a caliper, I couldn’t verify a number so precisely. With our ruler, any number beyond 19.03cm is fantasy. To drive the point home, one of my engineering professors once told me, “If you are measuring the height of a skyscraper, do you care if there is an extra sheet of paper at the top?”

How we represent numbers inside computers often leads us to believe we have better precision than we actually do. To drive this point home, open up a Python terminal and add 0.1 + 0.2. If you’ve never tried this before, you might be surprised to find this doesn’t equal 0.3, but 0.30000000000000004. I’m not going to go into the details of the math behind this phenomenon, but the question stands, can we reduce the precision without making things worse? We really only need precision to the tenth decimal, but reducing the

precision is likely to get us a number like 0.304 rather than 0.300 which would increase our margin of error.

Ultimately, the only numbers a computer understands are 0 and 1, on or off, a single bit. To improve this range we combine multiple of these bits together and assign them different meanings. String 8 of them together and you get a byte. Using the int8 standard we can take that byte and encode all the integers from -128 to 127. To spare you the particulars because I assume you already know how binary works, suffice it to say the more bits we have the larger range of numbers we can represent, both larger and smaller. Figure 3.1 shows a few common floating point encodings. With 32 bits strung together we get what we pretentiously term full precision, and that is how most numbers are stored including the weights in machine learning models. Basic quantization moves us from full precision to half precision, shrinking models to half their size. There are actually two different half precision standards, FP16 and BF16 which differ in how many bits represent the range or exponent part. Since BF16 uses the same number of exponents as FP32, it's been found to be more effective for quantizing and you can generally expect almost exactly the same level accuracy for half the size of model. If you understood the paper and skyscraper analogy above it should be obvious why.

Figure 3.1 shows the bit mapping for a few common floating point encodings. 16-bit float or half precision (FP16), bfloat 16 (BF16), 32-bit float or single full precision (FP32), and NVIDIA's TensorFloat (TF32)



However, there’s no reason to stop there. We can often push it another byte down to 8-bit formats without too much loss of accuracy. There have even already been successful research attempts showing selective 4-bit quantization of portions of LLMs are possible with only fractional loss of accuracy. The selective application of quantization is a process known as dynamic quantization and is usually done on just the weights, leaving the activations in full precision to reduce accuracy loss.

The holy grail of quantizing though would be int2, representing every number as -1, 0, or 1. This currently isn’t possible without completely degrading the model, but would make the model up to 8 times smaller. The Bloom model would be a measly ~40GB, small enough to fit on a single GPU. This is of course, as far as quantizing can take us and if we wanted to shrink further we’d need to look at additional methods.

The best part of quantization though is that it is easy to do. There are many frameworks that allow this, but in Listing 3.1 I demonstrate how to use pytorch’s quantization library to do a simple post training static quantization (PTQ). All you need is the full precision model, some example inputs, and a validation dataset to prepare and calibrate with. As you can see it’s only a few lines of code.

Listing 3.1 Example PTQ in PyTorch

```
import copy
import torch.ao.quantization as q

# deep copy the original model as quantization is done in place
model_to_quantize = copy.deepcopy(model_fp32)
model_to_quantize.eval()

# get mappings - note use “qnnpack” for ARM and “fbgemm” for x86
CPU
qconfig_mapping = q.get_default_qconfig_mapping("qnnpack")

# prepare
prepared_model = q.prepare(model_to_quantize)

# calibrate - you’ll want to use representative (validation)
```

```
data.
with torch.inference_mode():
    for x in dataset:
        prepared_model(x)

# quantize
model_quantized = q.convert(prepared_model)
```

Static PTQ is the most straightforward approach to quantizing, done after the model is trained and quantizing all the model parameters uniformly. As with most formulas in life, the most straightforward approach introduces more error. Oftentimes this error is acceptable, but when it’s not we can add extra complexity to reduce the accuracy loss from quantization. Some methods to consider are uniform vs non-uniform, static vs dynamic, symmetric vs asymmetric, and applying it during or after training.

To understand these methods let's consider the case where we are quantizing from FP32 to INT8. In FP32 we essentially have the full range of numbers at our disposal, but in INT8 we only have 256 values, we are trying to put a genie into a bottle and it’s no small feat. If you study the weights in your model, you might notice that the majority of the numbers are fractions between [-1, 1]. We could take advantage of this by then using an 8-bit standard that represents more values in this region in a non-uniform way instead of the standard uniform [-128, 127]. While mathematically possible, unfortunately, any such standards aren’t commonplace and modern day deep learning hardware and software are not designed to take advantage of this. So for now, it's best to just stick to uniform quantization.

The simplest approach to shrink the data is to just normalize it, but since we are going from a continuous scale to a discrete scale there are a few gotchas, so let's explore those. First, we start by taking the min and max and scale them down to match our new number range, we would then bucket all the other numbers based on where they fall. Of course, if we have really large outliers, we may find all our other numbers squeezed into just one or two buckets completely ruining any granularity we once had. To prevent this, we can simply clip any large numbers. This is what we do in static quantization. However, before we clip the data, what if we chose a range and scale that captures the majority of our data beforehand? We need to be careful, since if

this dynamic range is too small, we will introduce more clipping errors, if it's too big, we will introduce more rounding errors. The goal of dynamic quantization of course is to reduce both errors.

Next we need to consider the symmetry of the data. Generally in normalization we force the data to be normal and thus symmetric, however, we could choose to scale the data in a way that leaves any asymmetry it had. By doing this we could potentially reduce our overall loss due to the clipping and rounding errors, but it's not a guarantee.

As a last resort, if none of these other methods failed to reduce accuracy loss of the model, we can use Quantization Aware Training (QAT). QAT is a simple process where we add a fake quantization step during training of the model. By fake, I mean, we clip and round the data while leaving it in full precision. This allows the model to adjust for the error and bias introduced by quantization while it's training. QAT is known to produce higher accuracy compared to other methods but at a much higher cost in time to train.

Quantization Methods

Uniform vs Non-uniform: whether or not we use an 8-bit standard that is uniform in the range it represents or non-uniform to be more precise in the -1 to 1 range.

Static vs Dynamic: Choosing to adjust the range or scale before clipping in an attempt to reduce clipping and rounding errors and reduce data loss.

Symmetric vs Asymmetric: Normalizing the data to be normal and force symmetry, or choosing to keep any asymmetry and skew.

During or After Training: Quantization after training is really easy to do, and while doing it during training is more work it leads to reduced bias and better results.

Quantizing is a very powerful tool. Not only does it reduce the size of the model, but it also reduces the computational overhead required to run the

model thus reducing latency and cost of running the model. But the best fact about quantization is that it can be done after the fact, so you don't have to worry about whether or not your data scientists remembered to quantize the model during training using processes like QAT. This is why quantization has become so popular when working with LLMs and other large machine learning models. While reduced accuracy is always a concern with compression techniques, compared to other methods, quantization is a win-win-win.

Pruning

Congratulations, you just trained a brand new LLM! With billions of parameters all of them must be useful right? Wrong! Unfortunately, as with most things in life, the model's parameters tend to follow the Pareto Principle. About 20% of the weights lead to 80% of the value. "If that's true," you may be asking yourself, "Why don't we just go and cut out all the extra fluff?" Great idea! Give yourself a pat on the back. Pruning is the process of weeding out and removing any parts of the model that we deem unworthy.

There are essentially two different pruning methods: **structured** and **unstructured**. Structured pruning is the process of finding structural components of a model that aren't contributing to the model's performance and then removing them. Whether they be filters, channels, or layers in the neural network. The advantages to this method is that your model will be a little smaller but keep the same basic structure, which means we don't have to worry about losing hardware efficiencies, we are also guaranteed a latency improvement as there will be less computations involved.

Unstructured pruning on the other hand, is shifting through the parameters and zeroing out the less important ones that don't contribute much to the model's performance. Unlike structured pruning, we don't actually remove any parameters, just set them to zero. From this, we can imagine that a good place to start would be any weights or activations that are already close to 0. Of course, while this effectively reduces the size of a model this also means we don't cut out any computations, so it's common to only see minimal

latency improvement–if at all. But a smaller model still means faster load times and less GPUs to run. It also gives us very fine-grained control over the process, allowing us to shrink a model further than we could with structured pruning with less impact to performance too.

Like quantization, pruning can be done after a model is trained. However, unlike quantization, it's common practice to see additional fine-tuning needing to be done to prevent too much loss of performance. It's becoming more common to just include pruning steps during the model training to avoid the need to fine-tune later on. Since a more sparse model will have fewer parameters to tune, adding these pruning steps may help a model converge faster as well.[5]

You'll be surprised at how much you can shrink a model with pruning while minimally impacting performance. How much? In the SparseGPT[6] paper, a method was developed to try to automatically one shot the pruning process without the need for finetuning after. They found they could decrease a GPT-3 model by 50-60% without issue! Depending on the model and task they even saw slight improvements in a few of them. Looking forward to seeing where Pruning takes us in the future.

Knowledge Distillation

Knowledge distillation is probably the coolest method of compression in my mind. It's a simple idea too, we'll take the large LLM, and have it train a smaller language model to copy it. What's nice about this method is that the larger LLM provides essentially an infinite dataset for the smaller model to train on, which can make the training quite effective. Because of the simple fact that the larger the dataset the better the performance, we've often seen smaller models reach almost the same level as their teacher counterparts in accuracy.[7]

A smaller model trained this way is guaranteed to both be smaller and improve latency. The downside is that it's going to require us to train a completely new model. Which is a pretty significant upfront cost to pay. Any future improvements to the teacher model will require being passed down to

the student model, which can lead to complex training cycles and version structure. It's definitely a lot more work compared to some of the other compression methods.

The hardest part about knowledge distillation though is that we don't really have good recipes for them yet. Tough questions like, "How small can the student model be?" will have to be solved through trial and error. There's still a lot to learn and research to be done here.

However, there has been some exciting work in this field via Stanford's Alpaca[8]. Instead of training a student model from scratch, they instead chose to finetune the open source LLaMA 7B parameter model using OpenAI's GPT3.5's 175 billion parameter model as a teacher via knowledge distillation. A simple idea but it paid off big, as they were able to get great results from their evaluation. The biggest surprise was the cost, as they only spent $500 on API costs to get the training data from the teacher model, and $100 worth of GPU training time to finetune the student model. Granted, if you did this for a commercial application you'd be violating OpenAI's terms of service, so best to stick to using your own or open source models as the teacher.

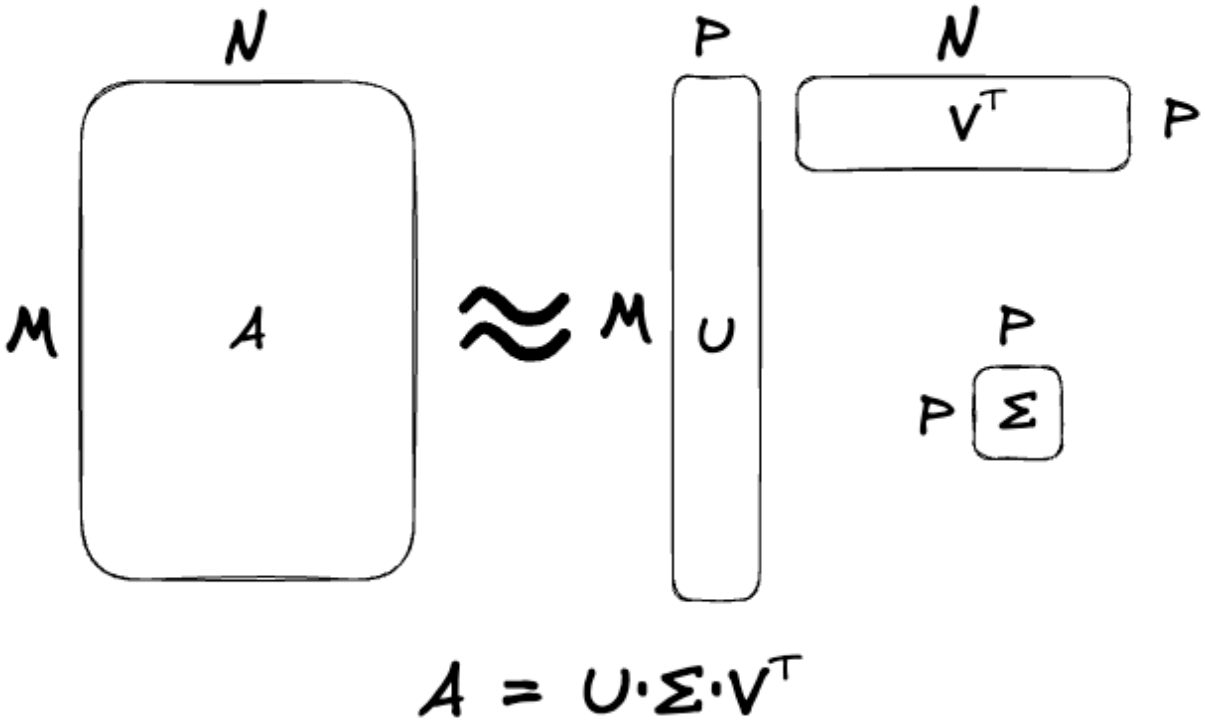
Low-rank Approximation

Low-rank approximation, also known as low-rank factorization, low-rank decomposition, matrix factorization (too many names! I blame the mathematicians), uses linear algebra math tricks to simplify large matrices or tensors finding a lower-dimensional representation. There are several techniques to do this. Singular Value Decomposition (SVD), Tucker Decomposition(TD), and Canonical Polyadic Decomposition (CPD) are the most common ones you run into.

In Figure 3.2 we show the general idea behind the SVD method. Essentially we are going to take a very large matrix, A, and break it up into 3 smaller matrices, U, ∑, and V. While U and V are there to ensure we keep the same dimensions and relative strengths of the original matrix, ∑ allows us to apply a direction and bias. The smaller ∑ is, the more we end up compressing and

reducing the total number of parameters, but the less accurate the approximation becomes.

Figure 3.2 Example of SVD a Low-rank Approximation. A is a large matrix with dimensions N and M. We can approximate it with three smaller matrices, U with dimensions M and P, ∑ a square matrix with dimension P, and V with dimensions N and P (here we show the transpose). Usually both P<<M and P<<N are true.



To help solidify this concept, it may help to see a concrete example. In Listing 3.2 we show a simple example of SVD at work compressing a 4x4 matrix. For this we only need the basic libraries SciPy and NumPy which are imported on lines 1 and 2. Line 3 we define the matrix, and then line 9 we apply SVD to it.

Listing 3.2 Example SVD Low-rank Approximation

```
import scipy
import numpy as np
matrix = np.array([
```

```
    [ 1., 2., 3., 4.],
    [ 5., 6., 7., 8.],
    [ 9., 10., 11., 12.],
    [13., 14., 15., 16.]
])
u, s, vt = scipy.sparse.linalg.svds(matrix, k=1)
print(u,s,vt)
# [[-0.13472211]
# [-0.34075767]
# [-0.5467932 ]
# [-0.7528288 ]], [38.62266], [[-0.4284123 -0.47437257
-0.52033263 -0.5662928 ]]
```

Taking a moment to inspect U, Sigma, and the transpose of V, we see a 4x1 matrix, a 1x1 matrix, and a 1x4 matrix respectively. All in all we now only need 9 parameters vs the original 16, shrinking the memory footprint almost in half.

Lastly, we multiply the matrices back together to get an approximation of the original matrix. In this case, the approximation isn't all that great, but we can still see the general order and magnitudes match the original matrix.

```
svd_matrix = u*s*vt
print(svd_matrix)
# array([[ 2.2291691, 2.4683154, 2.7074606, 2.9466066],
#      [ 5.6383204, 6.243202 , 6.848081 , 7.4529614],
#      [ 9.047472 , 10.018089 , 10.988702 , 11.959317 ],
#      [12.456624 , 13.792976 , 15.129323 , 16.465673 ]],
dtype=float32)
```

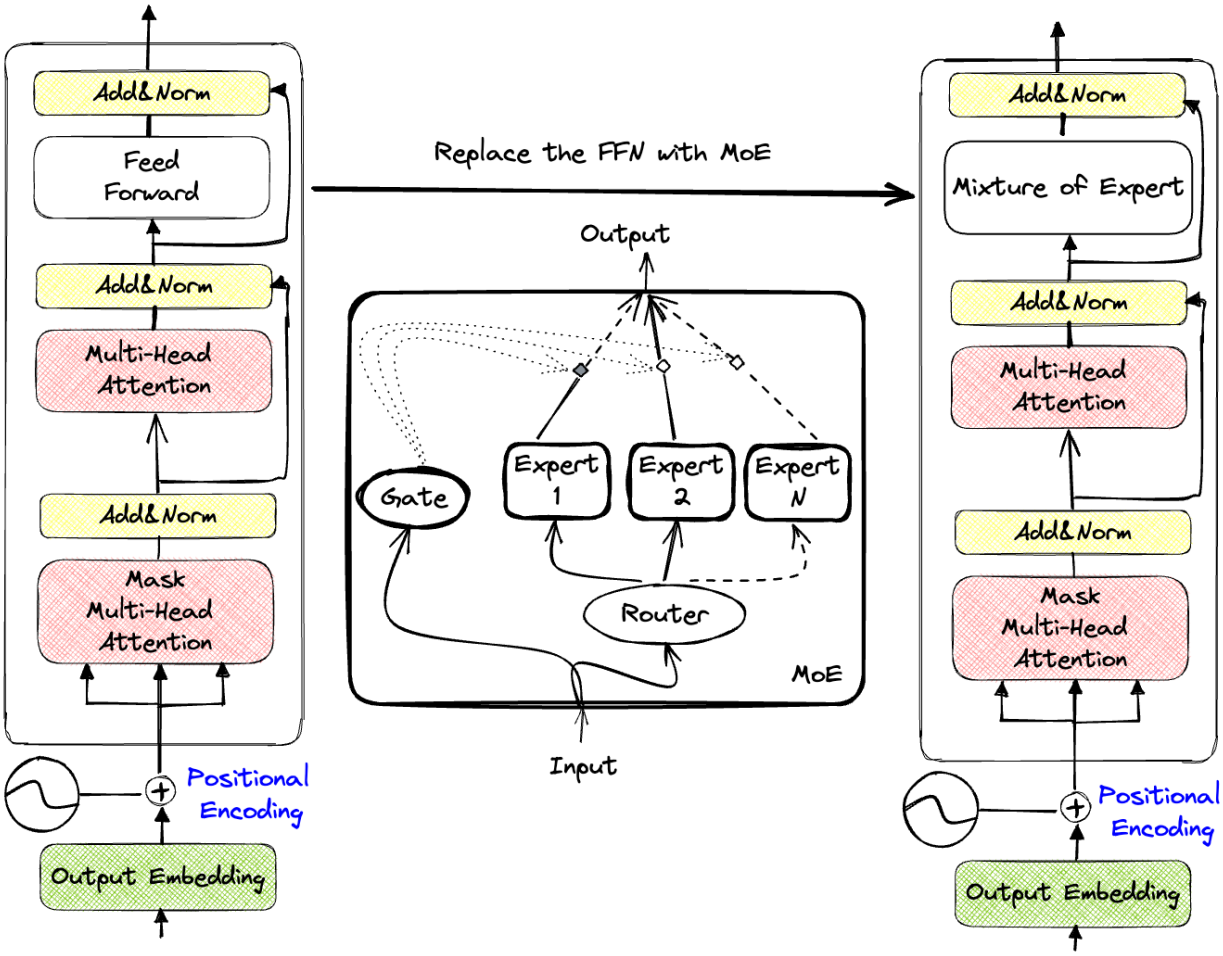
Unfortunately, I'm not aware of anyone actually using this to simply compress models in production most likely due to the poor accuracy of the approximation. What they are using it for, and this is important, is adaptation and finetuning; which is where LoRA[9] comes in, Low Rank Adaptation. Adaptation is simply the process of fine-tuning a generic or base model to do a specific task. LoRA applies SVD low rank approximation to the attention weights, or rather, to injected update matrices that run parallel to the attention weights, allowing us to fine-tune a much smaller model. LoRA has become very popular because it makes it a breeze to take an LLM, shrink the trainable layers to a tiny fraction of the original model and then allow anyone to train it on commodity hardware. You can get started with LoRA

by using the PEFT[10] library from HuggingFace, where they have several LoRA tutorials you can check out.

Mixture of Experts

Mixture of experts (MoE) is a technique where we replace the feed forward layers in a transformer with MoE layers instead. MoE's are a group of sparsely activated models. It differs from ensemble techniques in that typically only one or a few expert models will be run, rather than combining results from all models. The sparsity is often induced by a Gate mechanism that learns which experts to use, and/or a Router mechanism that determines which experts should even be consulted. In Figure 3.3 we demonstrate the MoE architecture with potentially N experts, as well as show where it goes inside a decoder stack.

Figure 3.3 Example Mixture of Experts model with both a Gate and Router to control flow. The MoE model is used to replace the FFN layers in a transformer, here we show it replacing the FFN in a Decoder.



Depending on how many experts you have, the MoE layer could potentially have more parameters than the FFN leading to a larger model, but in practice this isn't the case since engineers and researchers are aiming to create a smaller model. What we are guaranteed to see though is a faster computation path and improved inference times. However, what really makes MoE stand out is when it's combined with quantization. One study[11] between Microsoft and NVIDIA showed they were able to reach 2-bit quantization with only minimal impact to accuracy using MoE!

Of course, since this is a pretty big change to the model's structure it will require finetuning afterwards. You should also be aware that MoE layers often reduce a model's generalizability so it's best when used on models designed for a specific task. There are several libraries to implement MoE layers, but I'd recommend checking out DeepSpeed[12].

3.3.2 Distributed Computing

Distributed computing is a technique used in deep learning to parallelize and speed-up large, complex neural networks by dividing the workload across multiple devices or nodes in a cluster. This approach significantly reduces training and inference times by enabling concurrent computation, data parallelism, and model parallelism. With the ever-growing size of datasets and complexity of models, distributed computing has become crucial for deep learning workflows, ensuring efficient resource utilization and enabling researchers to effectively iterate on their models. Distributed computing is one of the core practices that separate deep learning from machine learning, and with LLMs we have to pull out every trick in the book. Let’s look at different parallel processing practices to take full advantage of distributed computing.

Data Parallelism

Data parallelism is what most people think about when they think about running processes in parallel, it’s also the easiest to do. The practice involves splitting up the data and running them through multiple copies of the model or pipeline. For most frameworks this is easy to set up, for example, in PyTorch you can use the DistributedDataParallel method. There’s just one catch for most of these set-ups and that is your model has to be able to fit onto one GPU. This is where a tool like Ray.io comes in.

Ray is an open-source project designed for distributed computing, specifically aimed at parallel and cluster computing. It's a flexible and user-friendly tool which simplifies distributed programming and helps developers execute concurrent tasks in parallel with ease. Ray is primarily built for machine learning and other high-performance applications but can be utilized in other applications. In Listing 3.3 we give a simple example of using Ray to distribute a task. The beauty of Ray is the simplicity–all we need to do to make our code run in parallel is add a decorator. Sure beats the complexity of multithreading or asynchronization set-ups.

Listing 3.3 Example Ray Parallelization Task

```
import ray
import time

ray.init() # Start Ray

# Define a regular Python function
def slow_function(x):
    time.sleep(1)
    return x

# Turn the function into a Ray task
@ray.remote
def slow_function_ray(x):
    time.sleep(1)
    return x

# Execute the slow function without Ray (takes 10 seconds)
results = [slow_function(i) for i in range(1, 11)]

# Execute the slow function with Ray (takes 1 second)
results_future = [slow_function_ray.remote(i) for i in range(1,
11)]
results_ray = ray.get(results_future)

print("Results without Ray: ", results)
print("Results with Ray: ", results_ray)

ray.shutdown()
```

Ray uses the concept of tasks and actors to manage distributed computing. Tasks are functions, whereas actors are stateful objects that can be invoked and run concurrently. When you execute tasks using Ray, it handles distributing tasks across the available resources (e.g., multi-core CPUs or multiple nodes in a cluster). For LLMs, we would need to set up a Ray cluster[13] in a cloud environment as this would allow each pipeline to run on a node with as many GPUs as needed, greatly simplifying the infrastructure set up to run LLMs in parallel.

There are multiple alternatives out there, but Ray has been gaining a lot of traction and becoming more popular as more and more machine learning workflows require distributed training. My team has had great success with

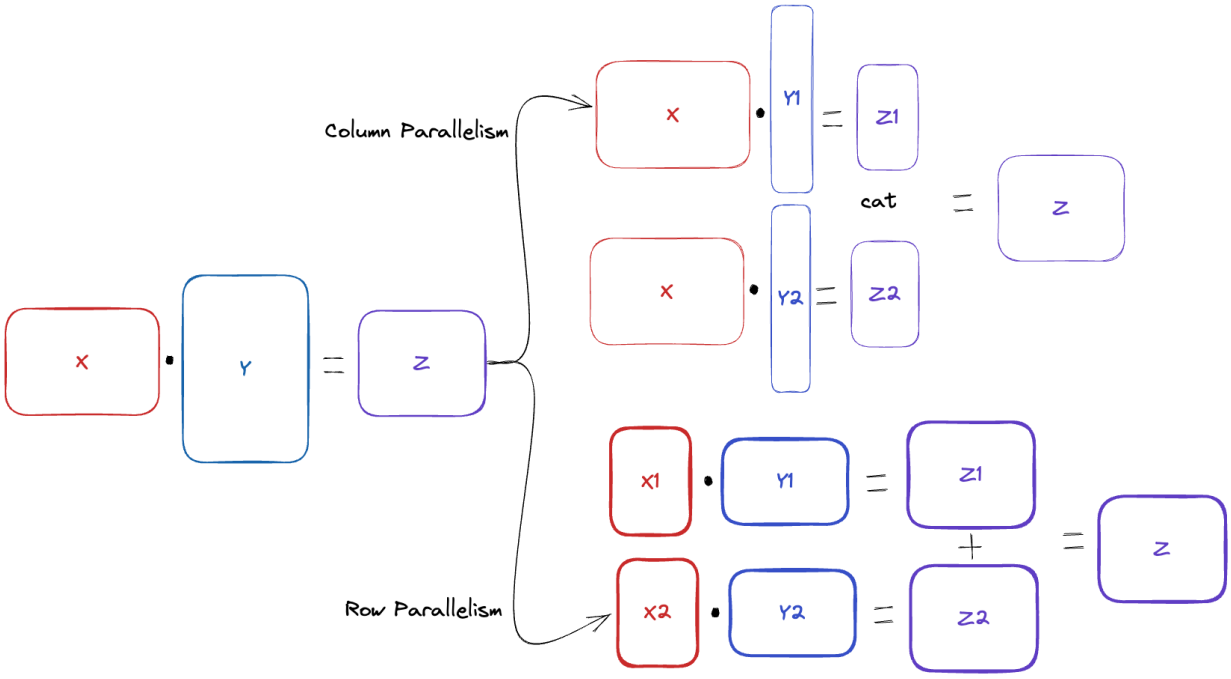
it. By utilizing Ray, developers can ensure better performance and more efficient utilization of resources in distributed workflows.

Tensor Parallelism

Tensor parallelism is taking advantage of matrix multiplication properties to split up the activations across multiple processors, running the data through, and then combining them on the other side of the processors. Figure 3.4 demonstrates how this process works for a matrix, which can be parallelized in two separate ways that give us the same result. Imagine that Y is a really big matrix that can’t fit on a single processor, or more likely, a bottleneck in our data flow that takes too much time to run all the calculations. In either case, we could split Y up, either by columns or by rows, run the calculations, and then combine the results after. In this example we are dealing with matrices but in reality we are often dealing with tensors that have more than two dimensions, but the same mathematical principles that make this work still apply.

Choosing which dimension to parallelize is a bit of an art, but a few things to remember to help make this decision easier. First, how many columns or rows do you have? In general, you want to pick a dimension that has more than the number of processors you have, else you will end up stopping short. Generally this isn’t a problem but with tools like Ray discussed in the last section, parallelizing in a cluster and spinning up loads of processes is a breeze. Second, different dimensions have different multiplicity costs. For example, column parallelism requires us to send the entire dataset to each process, but with the benefit of concatenating them together at the end which is fast and easy. Row parallelism however, allows us to break up the dataset into chunks, but requires us to add the results, a more expensive operation than concatenating. You can see that one operation is more I/O bound, while the other is more computation bound. Ultimately, the best dimension will be dataset dependent, as well as hardware limited. It will require experimentation to fully optimize this, but a good default is to just choose the largest dimension.

Figure 3.4 Tensor Parallelism example showing that you can break up tensors by different dimensions and get the same end result. Here we compare column and row parallelism of a Matrix.



Tensor parallelism allows us to split up the heavy computation layers like MLP and Attention layers onto different devices, but doesn't help us with Normalization or Dropout layers that don't utilize tensors. To get better overall performance of our pipeline we can add sequence parallelism which targets these blocks[14]. Sequence parallelism is a process that partitions activations along the sequence dimension, preventing redundant storage, and can be mixed with tensor parallelism to achieve significant memory savings with minimal additional computational overhead. In combination, they reduce the memory needed to store activations in Transformer models. In fact, they nearly eliminate activation recomputation and save activation memory up to 5x.

Figure 3.5 Combining tensor parallelism that focuses on computational heavy layers with sequence parallelism to reduce memory overhead to create a fully parallel process for the entire transformer.

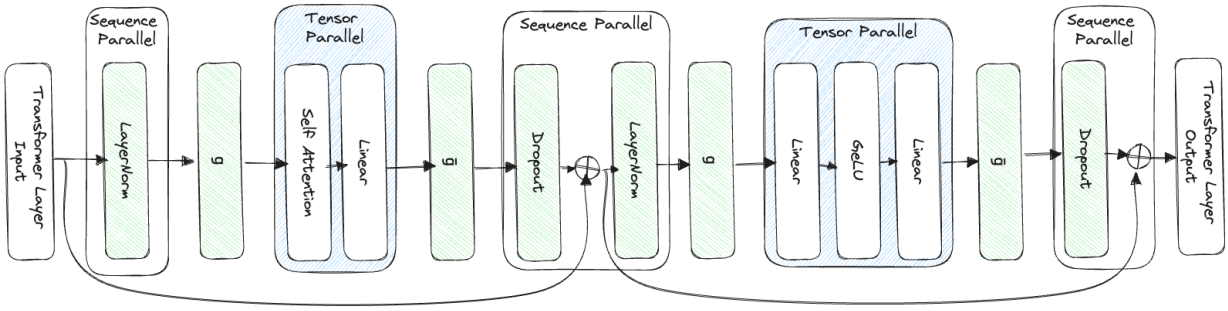


Figure 3.5 shows how combining both tensor parallelism, that allows us to distribute the computationally heavy layers, and sequence parallelism that does the same for the memory limiting layers, allows us to fully parallelize the entire transformer model. Together, they allow for extremely efficient use of resources.

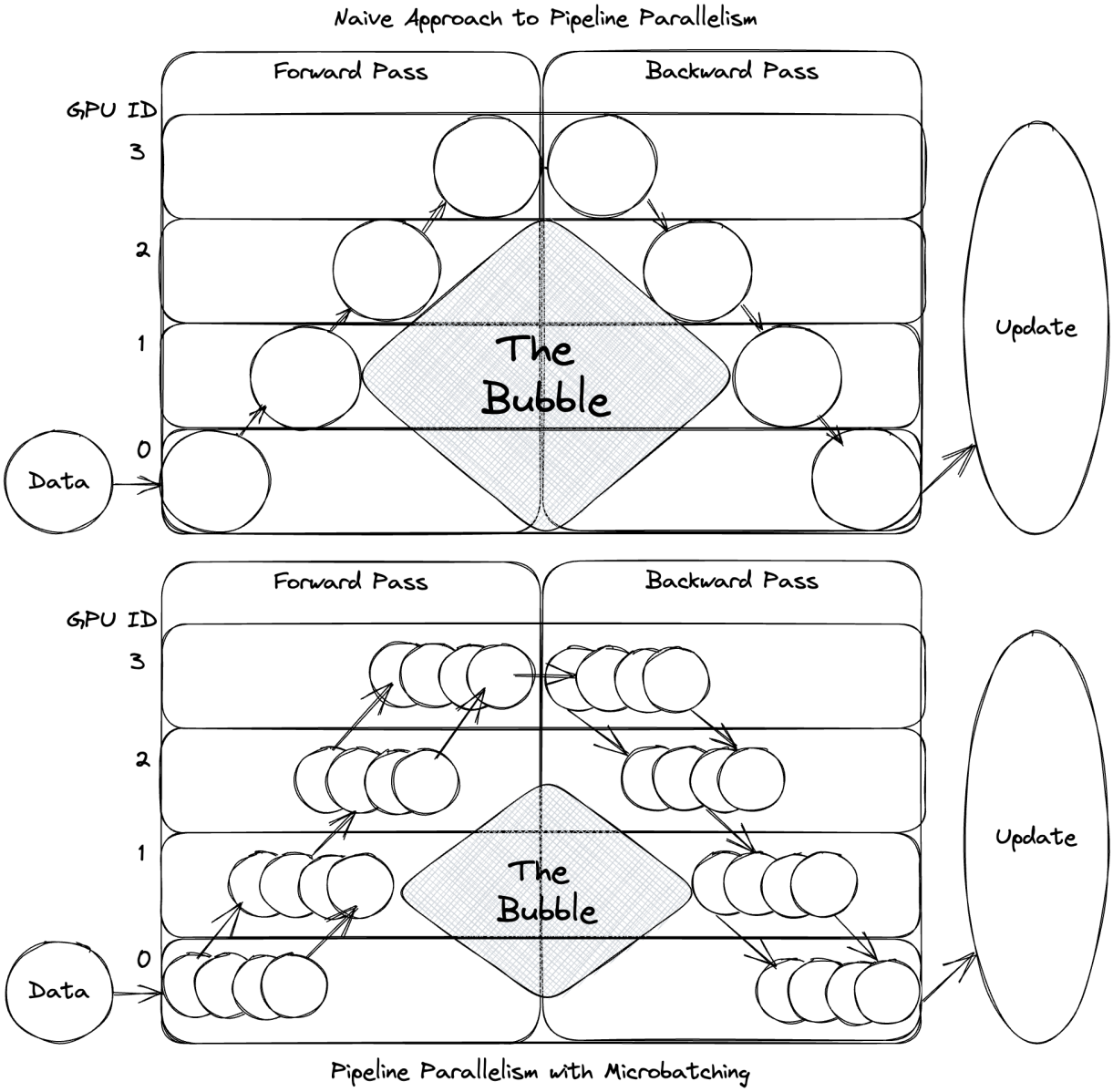
Pipeline Parallelism

So far we can now run lots of data, and speed up any bottlenecks, but none of that matters because our model is too big; we can’t fit it into a single GPU’s memory to even get it to run. That’s where pipeline parallelism comes in and is the process of splitting up a model vertically and putting each part onto a different GPU. This creates a pipeline as input data will go to the first GPU, process, then transfer to the next GPU, and so on until it’s run through the entire model. While other parallelism techniques improve our processing power and speed up inference, pipeline parallelism is required to just get it to run and it comes with some major downsides, mainly device utilization.

To understand where this downside comes from and how to mitigate it, let's first consider the naive approach to this, where we simply run all the data at once through the model. What we find is that this leaves a giant “bubble” of underutilization. Since the model is broken up, we have to process everything sequentially through the devices. This means that while one GPU is processing, the others are sitting idle. In Figure 3.6 we can see this naive approach and a large bubble of inactivity as the GPUs sit idle. We also see a better way to take advantage of each device. We do this by sending the data in small batches. A smaller batch allows the first GPU to pass on what it was

working on quicker and move on to another batch. This allows the next device to get started earlier and reduces the size of the bubble.

Figure 3.6 The Bubble Problem. When data runs through a broken up model, the GPUs holding the model weights are underutilized while they wait for their counterparts to process the data. A simple way to reduce this bubble is to use microbatching.



We can actually calculate the size of the bubble quite easily with the following formula:

Idle Percentage = 1 - m / (m + n - 1)

Where m is the number of microbatches, and n is the depth of the pipeline or number of GPUs. So for our naive example case of 4 GPUs and 1 large batch we see the devices sitting idle 75% of the time! GPUs are quite expensive to allow to be sitting idle three quarters of the time. Let's see what that looks like using the microbatch strategy. With a microbatch of 4, it cuts this almost in half, down to just 43% of the time. What we can glean from this formula is that the more GPUs we have, the higher the idle times, but the more microbatches the better the utilization.

Unfortunately, we can often neither reduce the number of GPUs nor can we just make the microbatches as large as we want. There are limits. For GPU's we just have to use as many as it takes to fit the model into memory, but try to use a few larger GPUs as it will lead to a more optimal utilization compared to using many smaller GPUs. Reducing the bubble in pipeline parallelism is another reason why compression is so important. For microbatching, the first limit is obvious once told, since the microbatch is a fraction of our batch size, we are limited by how big that is. The second is that each microbatch increases the memory demand for cached activations in a linear relationship. One way to counter this higher memory demand is a method called PipeDream[15]. There are different configurations and approaches, but the basic idea is the same. In this method we start working on the backward pass as soon as we've finished the forward pass of any of the microbatches. This allows us to fully complete a training cycle and release the cache for that microbatch.

3D Parallelism

For LLMs, we are going to want to take advantage of all three parallelism practices as they can all be run together. This is known as 3D Parallelism combining Data, Tensor, and Pipeline Parallelism (DP + TP + PP) together. Since each technique, and thus dimension, will require at least 2 GPUs, in order to run 3D Parallelism, we'll need at least 8 GPUs to even get started. How we configure these GPUs will be important to get the most efficiency out of this process, namely, because TP has the largest communication

overhead we want to ensure these GPUs are close together, preferably on the same node and machine. PP has the least communication volume of the three, so breaking up the model across nodes is the least expensive here.

By running the three together, we see some interesting interactions and synergies between them. Since TP splits the model to work well within a device's memory, we see that PP can perform well even with small batch sizes due to the reduced effective batch size enabled by TP. This combination also improves the communication between DP nodes at different pipeline stages, allowing DP to work effectively too. The communication bandwidth between nodes is proportional to the number of pipeline stages, because of this DP is able to scale well even with smaller batch sizes. Overall, we see running in combination that

Now that we know some tricks of the trade, it’s just as important to have the right tools to do the job.

3.4 Large Language Models Operations Infrastructure

We are finally going to start talking about the infrastructure needed to make this all work. This likely comes as a surprise as I know several readers would have expected this section at the beginning of chapter 1. Why wait till the end of chapter 3? In the many times I’ve interviewed Machine Learning Engineers I often asked this open-ended question, “What can you tell me about MLOps?” An easy softball question to get the conversation going. Most junior candidates would immediately start jumping into the tooling and infrastructure. It makes sense, there are so many different tools available. Not to mention, whenever you see posts or blogs describing MLOps there’s a pretty little diagram showing the infrastructure. While all of that is important it’s useful to recognize what a more senior candidate jumps into, the machine learning lifecycle.

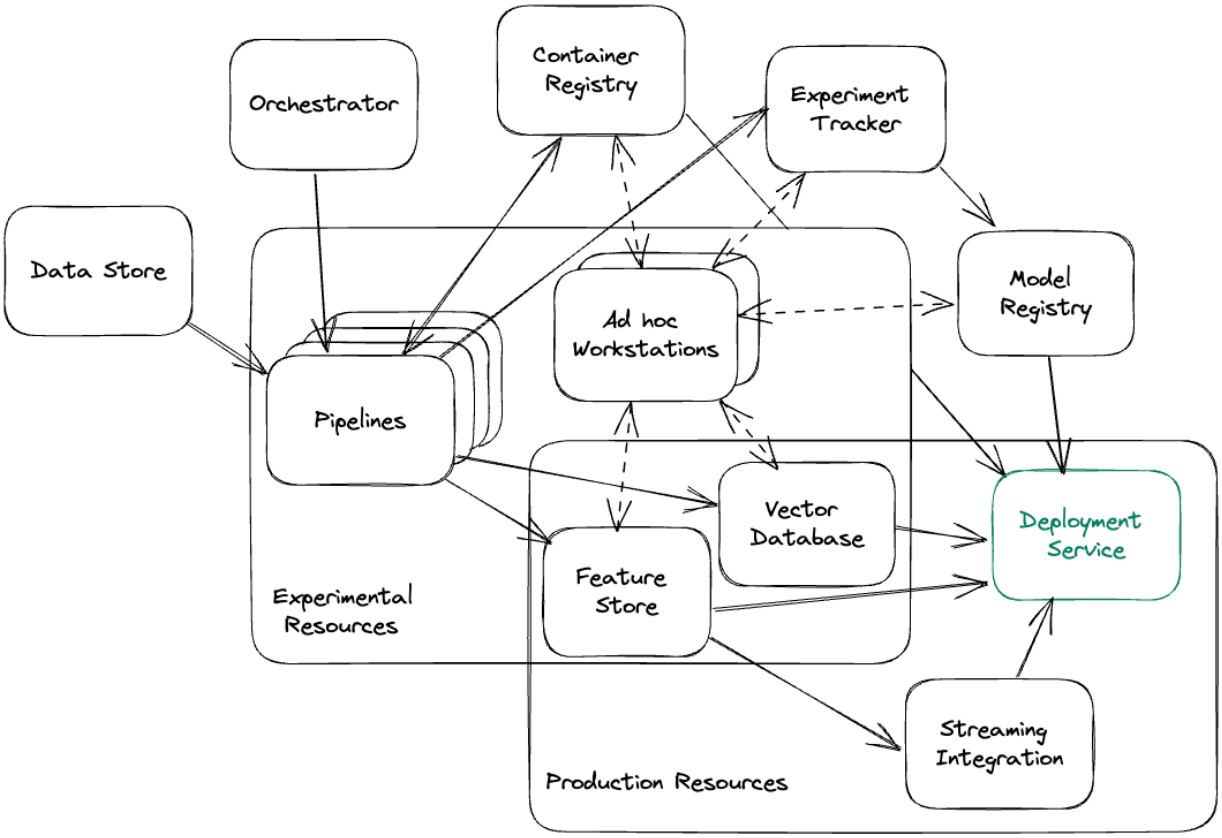
For many the nuance is lost, but the infrastructure is the how, the lifecycle is the why. Most companies can get by with just bare-bones infrastructure. I’ve seen my share of scrappy systems that exist entirely on one Data Scientist’s

laptop, and they work surprisingly well! Especially in the era of scikit learn everything.

Unfortunately, a rickshaw machine learning platform doesn't cut it in the world of LLMs. Since we still live in a world where the standard storage capacity of a MacBook Pro laptop is 256GB, just storing the model locally can already be a problem. Companies that invest in a more sturdy infrastructure are better prepared for the world of LLMs.

In Figure 3.7 we see an example MLOps Infrastructure designed with LLMs in mind. While most infrastructure diagrams I've seen in my time have always simplified the structure to make everything look clean, the raw truth is that there's a bit more complexity to the entire system. Of course a lot of this complexity would disappear if we could just get Data Scientists to work inside scripts instead of ad hoc workstations–usually with a jupyter notebook interface.

Figure 3.7 a high level view of an MLOps infrastructure with LLMs in mind. This attempts to cover the full picture, and the complexity of the many tools involved to make ML models work in production.



Taking a closer look at Figure 3.7 you can see several tools on the outskirts that squarely land in DataOps, or even just DevOps. Data Stores, Orchestrators, Pipelines, Streaming Integrations, and Container Registries. These are tools you are likely already using for just about any data intensive application and aren't necessarily MLOps focused. Towards the center we have more traditional MLOps tools, Experiment Trackers, Model Registry, Feature Store, and Ad hoc Data Science Workstations. For LLMs we really only introduce one new tool to the stack: a Vector Database. What's not pictured because it intertwines with every piece is the Monitoring System. This all culminates to what we are working towards in this book, a Deployment Service, where we can confidently deploy and run LLMs in Production.

Infrastructure by discipline

- **DevOps:** In charge of procuring the environmental resources—experimental (dev, staging) and production—this includes hardware,

clusters, and networking to make it all work. Also in charge of basic infrastructure systems like Github/Gitlab, artifact registries, container registries, application or transactional databases like Postgres or MySQL, caching systems, and CI/CD pipelines. This is by no means a comprehensive list.

- **DataOps:** In charge of data, in motion and at rest. Includes centralized or decentralized data stores like Data Warehouses, Data Lakes, and Data Meshes. As well as data pipelines either in batch systems or in streaming systems with tools like Kafka and Flink. Also includes orchestrators like Airflow, Prefect or Mage. DataOps is built on top of DevOps. For example, I've seen many CI/CD pipelines being used for data pipeline work until eventually being graduated to systems like Apache Spark or DBT.
- **MLOps:** In charge of machine learning lifecycle from creation of models to deprecation. This includes data science workstations like Jupyterhub, experiment trackers, and a model registry. It includes specialty databases like Feature Stores and Vector Databases. As well as a deployment service to tie everything together and actually serve results. It is built on top of both DataOps and DevOps.

Let's go through each piece of the infrastructure puzzle and discuss features you should be considering when thinking about LLMs in particular. While we will be discussing tooling that is specialized for each piece, I'll just make note that there are also MLOps as a service platforms like Dataiku, Amazon's Sagemaker and Google's VertexAI. These platforms attempt to give you the whole puzzle, how well they do that is another question, but are often a great shortcut and you should be aware of them. Well, I think that's enough dilly-dallying, let's dive in already!

3.4.1 Data Infrastructure

While not the focus of this book it's important to note that MLOps is built on top of a Data Operations infrastructure–which itself is built on top of DevOps. Key features of the DataOps ecosystem include a data store, an orchestrator, and pipelines. Additional features usually required include a container registry and a streaming integration service.

Data stores are the foundation of DataOps and come in many forms these days, from a simple database to large data warehouses, and from even larger data lakes to an intricate data mesh. This is where your data is stored and a lot of work goes into managing, governing, and securing the data store. The orchestrator is the cornerstone of DataOps as it's a tool that manages and automates both simple and complex, multistep workflows and tasks. Ensuring they run across multiple resources and services in a system. The most commonly talked about being Airflow, Prefect, and Mage. Lastly, pipelines are the pillars. They hold everything else up, and are where we actually run our jobs. Initially built to simply move, clean, and define data, these same systems are used to run machine learning training jobs on a schedule, do batch inference, and loads of other work needed to ensure MLOps runs smoothly.

A container registry is a keystone of DevOps and subsequently DataOps and MLOps as well. Being able to run all our pipelines and services in containers is necessary to ensure consistency. Streaming services are actually a much bigger beast than what I may let on in this chapter, and if you know you know. Thankfully for most text related tasks real time processing isn't a major concern. Even for tasks like real-time captioning or translation, we can often get by with some sort of pseudo real-time processing strategy that doesn't degrade the user experience depending on the task.

3.4.2 Experiment Trackers

Experiment trackers are central to MLOps. Experiment trackers do the fundamental job of keeping track and recording tests and results. As the famous Adam Savage quote from Mythbusters, "Remember kids, the only difference between screwing around and science is writing it down." Without it, your organization is likely missing the "science" in data science which is honestly quite embarrassing.

Even if your data scientists are keen to manually track and record results in notebooks, it might as well be thrown in the garbage if it's not easy for others to view and search for. This is really the purpose of experiment trackers, to ensure knowledge is easily shared and made available.

Eventually a model will make it to production and that model is going to have issues. Sure, you can always just train a new model, but unless the team is able to go back and investigate what went wrong the first time you are likely to repeat the same mistakes over and over.

There are many experiment trackers out there, the most popular by far is MLFlow which is open source. It was started by the team at Databricks which also offers an easy hosting solution. Some paid alternatives worth checking out include CometML and Weights and Biases.

Experiment trackers nowadays come with so many bells and whistles. Most open source and paid solutions will certainly have what you need when looking to scale up your needs for LLMOps. However, ensuring you take advantage of these tools correctly might require a few small tweaks. For example, the default assumption is usually that you are training a model from scratch, but often when working with LLMs you will be finetuning models instead. In this case, it's important to note the checkpoint of the model you started from. If possible, even linking back to the original training experiment. This will allow future scientists to dig deeper into their test results, find original training data, and discover paths forward to eliminate bias.

Another feature to look out for is evaluation metric tooling. We will be going more in-depth in Chapter 4, but evaluation metrics are difficult for language models. There are often multiple metrics you care about and none of them are simple like complexity ratings or similarity scores. While experiment tracker vendors try to be agnostic and unopinionated about evaluation metrics they should at least make it easy to compare models and their metrics to make it easy to decide which one is better. Since LLMs have become so popular some have made it easy to evaluate on the more common metrics like ROUGE for text summarization.

You will also find many experiment tracking vendors have started to add tools specifically for LLMs. Some features you might consider looking for include direct HuggingFace support, LangChain support, prompt engineering toolkits, finetuning frameworks, and foundation model shops.

The space is developing quickly, and no one tool has all the same features right now, but I'm sure these feature sets will likely converge.

3.4.3 Model Registry

The model registry is probably the simplest tool of an MLOps infrastructure. The main objective is one that's easy to solve, we just need a place to store the models. I've seen many successful teams get by simply by putting their models in an object store or shared file system and calling it good. That said, there's a couple bells and whistles you should look for when choosing one.

The first is whether or not the model registry tracks metadata about the model. Most of what you care about is going to be in the experiment tracker, so you can usually get away with simply ensuring you can link the two. In fact, most model registries are built into experiment tracking systems because of this. However, an issue I've seen time and time again with these systems happens when the company decides to use an open source model or even buy one. Is it easy to upload a model and tag it with relevant information? The answer is usually no.

Next, you are going to want to make sure you can version your models. At some point, a model will reach a point where it's no longer useful and will need to be replaced. Versioning your models will simplify this process. It also makes running production experiments like A/B testing or shadow tests easier.

Lastly, if we are promoting and demoting models, we need to be concerned with access. Models tend to be valuable intellectual property for many companies, ensuring only the right users have access to the models is important. But it's also important to ensure that only the team that understands the models, what they do and why they were trained, are in charge of promoting and demoting the models. The last thing we want is to delete a model in production or worse.

For LLMs there are some important caveats you should be aware of, mainly, when choosing a model registry, be aware of any limit sizes. I've seen several model registries restrict model sizes to 10GB or smaller. That's just

not going to cut it. I could speculate on the many reasons for this but none of them are worthy of note. Speaking of limit sizes, if you are going to be running your model registry on an premise storage system like Ceph, make sure it has lots of space. You can buy multiple terabytes of storage for a couple hundred dollars for your on prem servers, but even a couple terabytes fills up quickly when your LLM is over 300GB. Don’t forget, you are likely to be keeping multiple checkpoints and versions during training and finetuning; as well as duplicates for reliability purposes. Storage is still one of the cheapest aspects of running LLMs though, no reason to skimp here and cause headaches down the road.

This does bring me to a good point: there's a lot of optimization that could still be made, allowing for better space saving approaches to storing LLMs and their derivatives. Especially since most of these models will be very similar overall. I imagine we’ll likely see storage solutions to solve just this problem in the future.

3.4.4 Feature Store

Feature stores solve many important problems and answer questions like: Who owns this feature? How was it defined? Who has access to it? Which models are using it? How do I serve this feature in production? Essentially, they solve the “single source of truth” problem. By creating a centralized store, it allows teams to shop for the highest quality, most well maintained, thoroughly managed data. Feature stores solve the collaboration, documentation, and versioning of data.

If you’ve ever thought, “A feature store is just a database right?” you are probably thinking about the wrong type of store–we are referencing a place to shop not a place of storage. Don’t worry, this confusion is normal as I’ve heard this sentiment a lot, and have had similar thoughts myself. The truth is, modern day feature stores are more virtual than a physical database, which means they are built on top of whatever data store you are already using. For example, Google’s Vertex AI feature store is just BigQuery and I’ve seen a lot of confusion from data teams wondering, “Why don’t I just query BigQuery?” Loading the data into a feature store feels like an unnecessary

extra step, but think about shopping at an IKEA store. No one goes directly to the warehouse where all the furniture is in boxes. That would be a frustrating shopping experience. The features store is the show rooms that allows others in your company to easily peruse, experience, and use the data.

Often times, I see people reach for a feature store to solve a technical problem like low latency access for online feature serving. A huge win for feature stores is solving the training-serving skew. Some features are just easier to do in SQL after the fact, like calculating the average number of requests for the last 30 seconds. This can lead to naive data pipelines built for training, but causing massive headaches when going to production because getting this type of feature in real time can be anything but easy. Feature stores abstractions help minimize this burden. Related to this is feature stores point-in-time retrievals which are table stakes when talking feature stores. Point-in-time retrievals ensure that given a specific time a query will always return the same result. This is important because features like averages over “the last 30 seconds” are constantly changing, so this allows us to version the data (without the extra burden of a bloated versioning system), as well as ensure our models will give accurate and predictable responses.

As far as options, Feast is a popular open source feature store. FeatureForm and Hopsworks are also open source. All three of which offer paid hosting options. For LLMs I’ve heard the sentiment that feature stores aren’t as critical as other parts of the MLOps infrastructure. After all, the model is so large it should incorporate all needed features inside it, so you don’t need to query for additional context, just give the model the user’s query and let the model do its thing. However, this approach is still a bit naive and we haven’t quite gotten to a point where LLMs are completely self-sufficient. To avoid hallucinations and improve factual correctness, it is often best to give the model some context. We do this by feeding it embeddings of our documents we want it to know very well, and a feature store is a great place to put these embeddings.

3.4.5 Vector Databases

If you are familiar with the general MLOps infrastructure, most of this section has been review for you. We've only had to make slight adjustments highlighting important scaling concerns to make a system work for LLMs. Vector Databases however are new to the scene and have been developed to be a tailored solution for working with LLMs and language models in general, but you can also use them with other datasets like images or tabular data which are easy enough to transform into a vector. Vector databases are specialized databases to store vectors along with some metadata around the vector, which makes them great for storing embeddings. Now, while that last sentence is true, it is a bit misleading, because the power of vector databases isn't in their storage, but in the way that they search through the data.

Traditional databases, using b-tree indexing to find ID's or text based search using reverse indexes, all have the same common flaw, you have to know what you are looking for. If you don't have the ID or you don't know the keywords it's impossible to find the right row or document. Vector databases however, take advantage of the vector space meaning you don't need to know exactly what you are looking for, you just need to know something similar which you can then use to find the nearest neighbors using similarity searches based on Euclidean distance, Cosine similarity, Dot product similarity, or what have you. Using a vector database makes solving the reverse image search problem a breeze, as an example.

At this point, I'm sure some readers may be confused. First I told you to put your embeddings into a feature store, and now I'm telling you to put them into a Vector DB, which one is it? Well that's the beauty of it, you can do both at the same time. If it didn't make sense before I hope it makes sense now. Feature stores are not a database they are just an abstraction, you can use a feature store built on top of a Vector DB and it will solve many of your problems. Vector DBs can be difficult to maintain when you have multiple data sources, experimenting with different embedding models or otherwise have frequent data update. Managing this complexity can be a real pain, but a feature store can handily solve this problem. Using them in combination will ensure a more accurate and up-to-date search index.

Vector databases have only been around for a couple of years at the time of writing, and their popularity is still relatively new as it has grown hand in hand with LLMs. It's easy to understand why since they provide a fast and efficient way to retrieve vector data making it easy to provide LLMs with needed context to improve their accuracy.

That said it's a relatively new field and there are lots of competitors in this space right now, and it's a bit too early to know who the winners and losers are. Not wanting to date this book too much, let me at least suggest two options to start Pinecone and Milvus. Pinecone is one of the first vector databases as a product and has a thriving community with lots of documentation. It's packed with features and has proven itself to scale. Pinecone is a fully managed infrastructure offering that has a free tier for beginners to learn. If you are a fan of open source however, then you'll want to check out Milvus. Milvus is feature rich and has a great community. Zilliz the company behind Milvus offers a fully managed offering, but it's also available to deploy in your own clusters and if you already have a bit of infrastructure experience it's relatively easy and straightforward to do.

There are lots of alternatives out there right now, and it's likely worth a bit of investigation before picking one. Probably the two things you'll care most about is price and scalability, the two often going hand in hand. After that, it's valuable to pay attention to search features, like support for different similarity measures like cosine similarities, dot product or euclidean distance. As well as indexing features like HNSW (Heirarchical Navigable Small World) or LSH (Locality-Sensitive Hashing). Being able to customize your search parameters and index settings are important for any database as they allow you to customize the workload for your dataset and workflow allowing you to optimize query latency and search result accuracy.

It's also important to note that with vector databases rise in popularity we are quickly seeing many database incumbents like Redis and Elastic offering vector search capabilities. For now most of these tend to just offer the most straightforward feature sets, but they are hard to ignore if you are already using these tool sets as they can provide quick wins to get started quickly.

Vector databases are powerful tools that can help you train or finetune LLMs, as well as improve the accuracy and results of your LLM queries.

3.4.6 Monitoring System

A monitoring system is crucial to the success of any ML system, LLMs included. Unlike other software applications, ML models are known to fail silently—continue to operate, but start to give poor results. This is often due to data drift, a common example being a recommendation system that give worse results overtime because sellers start to game the system by giving fake reviews to get better recommendation results. A monitoring system allows us to catch poorly performing models, make adjustments or simply retrain them.

Despite their importance they are often the last piece of the puzzle added. This is often purposeful as putting resources into figuring out how to monitor models doesn't help if you don't have any models to monitor. However, don't make the mistake of putting it off too long. Many companies have been burned by a model that went rogue with no one knowing about it, often costing them dearly. It's also important to realize you don't have to wait to get a model into production to start monitoring your data. There are plenty of ways to introduce a monitoring system into the training and data pipelines to improve data governance and compliance. Regardless, you can usually tell the maturity of a data science organization by their monitoring system.

There are lots of great monitoring tooling out there, some great open source options include WhyLogs and EvidentlyAI. I'm also a fan of great expectations, but have found it rather slow outside of batch jobs. There are also many more paid options out there. Typically, for ML monitoring workloads you'll want to monitor everything you'd normally record in other software applications, this includes resource metrics like memory and CPU utilization, performance metrics like latency and queries per second, as well as operational metrics like status codes and error rates. In additional, you'll need ways to monitor data drift both going in and out of the model. You'll want to pay attention to things like missing values, uniqueness, and standard

deviation shifts. In many instances, you'll want to be able to segment your data while monitoring, e.g. for A/B testing or to monitor by region. Some metrics that are useful to monitor in ML systems include model accuracy, precision, recall and F1 scores. These are difficult since you won't know the correct answer at inference time, so it's often useful to set up some sort of auditing system. Of course, auditing is going to be easier if your LLM is designed to be a Q&A bot than if your LLM is meant to help writers be more creative.

This hints at a fact that for LLMs, there are often a whole set of new challenges for your monitoring systems even more than what we see with other ML systems. With LLMs we are dealing with text data which is hard to quantify as discussed earlier in this chapter. For instance, think about what features do you look at to monitor for data drift? Because language is known to drift a lot! One feature I might suggest is unique tokens. This will alert you when new slang words or terms are created, however, it still doesn't help when words switch meaning, for example, when "wicked" means "cool". I would also recommend monitoring the embeddings, however, you'll likely find this to either add a lot of noise and false alarms or at the very least be difficult to decipher and dig into when problems do occur. The systems I've seen work the best often involve a lot of handcrafted rules and features to monitor, but these can be error-prone and time-consuming to create.

Monitoring text based systems is far from a solved problem, mostly stemming from the difficulties of understanding text data to begin with. This does beg the question of what are the best methods to use language models to monitor themselves, since they are our current best solution to codifying language. Unfortunately, I'm not aware of anyone researching this, but imagine it's only a matter of time.

3.4.7 GPU Enabled Workstations

GPU enabled workstations and remote workstations in general are often considered a nice to have or luxury by many teams, but when working with LLMs that mindset has to change. When troubleshooting an issue or just

developing a model in general, a data scientist isn't going to be able to spin up the model in a notebook on their laptop anymore. The easiest way to solve this is to simply provide remote workstations with GPU resources. There are plenty of cloud solutions for this, but if your company is working mainly on prem, this may be a bit more difficult to provide, but necessary nonetheless.

LLMs are GPU memory intensive models. Because of this, there are some numbers every engineer should know when it comes to working in the field. The first, is how much different GPUs have. The NVIDIA Tesla T4 and V100 are two most common GPUs you'll find in a datacenter, but they only have 16 GB of memory. They are workhorses though and cost-effective, so if we can compress our model to run on these all the better. After these, you'll find a range of GPU's like NVIDIA A10G, NVIDIA Quadro series, and NVIDIA RTX series that offer GPU memories in the ranges of 24, 32, and 48 GB. All of these are fine upgrades, you'll just have to figure out which ones are offered and available to you by your cloud provider. Which brings us to the NVIDIA A100, which is likely going to be your GPU of choice when working with LLMs. Thankfully they are relatively common, and offer two different models providing 40 or 80 GB. The big issue you'll have with these are that they are constantly in high demand by everyone right now. You should also be aware of the NVIDIA H100 which offers 80 GB like the A100. The H100 NVL is promised to support up to 188 GB and has been designed with LLMs in mind. Another new GPU you should be aware of is the NVIDIA L4 Tensor Core GPU which has 24 GB and is positioned to take over as a new workhorse along with the T4 and V100, at least as far as AI workloads are concerned.

LLMs come in all different sizes, and it's useful to have a horse sense for what these numbers mean. For example, the LLaMA model has 7B, 13B, 33B, and 65B parameter variants. If you aren't sure which GPU you need to run which model off the top of your head, here's a shortcut, just take the number of billions of parameters times it by two and that's how much GPU memory you need. The reason is, most models at inference are going to default to run at half precision, FP16 of BF16, which means for every parameter we need at least two bytes. Thus, 7 billion * 2 bytes = 14 GB.

You'll need a little extra as well for the embedding model which will be about another GB, and more for the actual tokens you are running through the model. One token is about 1 MB, so 512 tokens will require 512 MB. This isn't a big deal, until you consider running larger batch sizes to improve performance. For 16 batches of this size you'll need an extra 8 GB of space.

Of course, so far we've only been talking about inference, for training you'll need a lot more space. While training, you'll always want to do this in full precision, and you'll need extra room for the optimizer tensors and gradients. In general, to account for this you'll need about 16 bytes for every parameter. So to train a 7B parameter model you'll want 112 GB of memory.

3.4.8 Deployment Service

Everything we've been working towards is collected and finally put to good use here. In fact, if you took away every other service and were left with just a deployment service, you'd still have a working MLOps system. A deployment service provides an easy way to integrate with all the previous systems we talked about as well as configure and define the needed resources to get our model running in production. It will often provide boilerplate code to serve the model behind a REST and gRPC API or directly inside a batch or streaming pipeline.

Some tools to help create this service include NVIDIA Triton Inference Service, MLServer, Seldon and BentoML. These services provide a standard API interface, typically the KServe V2 Inference Protocol. This protocol provides a unified and extensible way to deploy, manage, and serve machine learning models across different platforms and frameworks. It defines a common interface to interact with models, including gRPC and HTTP/RESTful APIs. It standardizes concepts like input/output tensor data encoding, predict and explain methods, model health checks, and metadata retrieval. It also allows seamless integration with languages and frameworks including TensorFlow, PyTorch, ONNX, Scikit Learn, and XGBoost.

Of course, there are times when flexibility and customization provide enough value to step away from the automated path these other frameworks

provide, in which case it's best to reach for a tool like FastAPI. Your deployment service should still provide as much automation and boilerplate code here to make the process as smooth as possible. It should be mentioned that most of the frameworks mentioned above do offer custom methods, but your mileage may vary.

Deploying a model is more than just building the interface. Your deployment service will also provide a bridge to close the gap between the MLOps infrastructure and general DevOps infrastructure. Connecting to whatever CI/CD tooling as well as build and shipping pipelines your company has set up so you can ensure appropriate tests and deployment strategies like health checks and rollbacks can easily be monitored and done. This is often very platform and thus company-specific. Thus, it'll also need to provide the needed configurations to talk to Kubernetes, or whatever other container orchestrator you may be using, to acquire the needed resources like CPU, Memory, and Accelerators, Autoscalers, Proxies, etc. It also applies the needed environment variables and secret management tools to ensure everything runs.

All in all, this service ensures you can easily deploy a model into production. For LLMs, the main concern is often just ensuring the platform and clusters are set up with enough resources to actually provision what will ultimately be configured.

We've discussed a lot so far in this chapter, starting with what makes LLMs so much harder than traditional ML which is hard enough as it is. First, we learned that their size can't be underestimated, but then we also discovered there are many peculiarities about them, from token limits to hallucinations, not to mention they are expensive. Fortunately, despite being difficult, they aren't impossible. We discussed compression techniques and distributed computing which are crucial to master. We then explored the infrastructure needed to make LLMs work. While most of it was likely familiar, we came to realize that LLMs put a different level of pressure on each tool, and often we need to be ready for a larger scale than what we could get away with for deploying other ML models.

3.5 Summary

- LLMs are difficult to work with mostly because they are big. Which impacts a longer time to download, load into memory, and deploy forcing us to use expensive resources.
- LLMs are also hard to deal with because they deal with natural language and all its complexities including hallucinations, bias, ethics, and security.
- Regardless if you build or buy, LLMs are expensive and managing costs and risks associated with them will be crucial to the success of any project utilizing them.
- Compressing models to be as small as we can will make them easier to work with; quantization, pruning, and knowledge distillation are particularly useful for this.
- Quantization is popular because it is easy and can be done after training without any finetuning.
- Low Rank Approximation is an effective way at shrinking a model and has been used heavily for Adaptation thanks to LoRA.
- There are three core directions we use to parallelize LLM workflows: Data, Tensor, and Pipeline. DP helps us increase throughput, TP helps us increase speed, and PP makes it all possible to run in the first place.
- Combining the parallelism methods together we get 3D parallelism (Data+Tensor+Pipeline) where we find that the techniques synergize, covering each others weaknesses and help us get more utilization.
- The infrastructure for LLMOps is similar to MLOps, but don't let that fool you since there are many caveats where "good enough" no longer works.
- Many tools are offering new features specifically for LLM support.
- Vector Databases in particular are interesting as a new piece of the infrastructure puzzle needed for LLMs that allow quick search and retrievals of embeddings.

[1] A. Bulatov, Y. Kuratov, and M. S. Burtsev, "Scaling Transformer to 1M tokens and beyond with RMT," Apr. 2023, https://arxiv.org/abs/2304.11062.

[2] R. Daws, “Medical chatbot using OpenAI’s GPT-3 told a fake patient to kill themselves,” AI News, Oct. 28, 2020. https://www.artificialintelligence-news.com/2020/10/28/medical-chatbot-openai-gpt3-patient-kill-themselves/

[3] T. Kington, “ChatGPT bot tricked into giving bomb-making instructions, say developers,” www.thetimes.co.uk, Dec 17, 2022. https://www.thetimes.co.uk/article/chatgpt-bot-tricked-into-giving-bomb-making-instructions-say-developers-rvktrxqb5

[4] K. Quach, “AI game bans players for NSFW stories it generated itself,” www.theregister.com, Oct 8, 2021. https://www.theregister.com/2021/10/08/ai_game_abuse/

[5] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, “Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks,” Jan. 2021, https://arxiv.org/abs/2102.00554.

[6] E. Frantar and D. Alistarh, “SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot,” Jan. 2023, https://arxiv.org/abs/2301.00774.

[7] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter,” Oct. 2019. https://arxiv.org/abs/1910.01108.

[8] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P Liang, and T. B. Hashimoto, “Alpaca: A Strong, Replicable Instruction-Following Model,” crfm.stanford.edu, 2023. https://crfm.stanford.edu/2023/03/13/alpaca.html

[9] E. J. Hu et al., “LoRA: Low-Rank Adaptation of Large Language Models.,” Jun. 2021, https://arxiv.org/abs/2106.09685.

[10] For the extra curious, Parameter-Efficient Fine-Tuning (PEFT) is a class of methods aimed at fine-tuning models in a computational efficient way. The PEFT library seeks to put them all in one easy to access place and you can get started here: https://huggingface.co/docs/peft

[11] R. Henry and Y. J. Kim, “Accelerating Large Language Models via Low-Bit Quantization,” March 2023, https://www.nvidia.com/en-us/on-demand/session/gtcspring23-s51226/

[12] DeepSpeed is a library that optimizes many of the hard parts for large-scale deep learning models like LLMs and is particularly useful when training. Check out their MoE tutorial. https://www.deepspeed.ai/tutorials/mixture-of-experts/

[13] Learn more about Ray Clusters here: https://docs.ray.io/en/releases-2.3.0/cluster/key-concepts.html#ray-cluster

[14] V. Korthikanti et al., “Reducing Activation Recomputation in Large Transformer Models,” May 2022, https://arxiv.org/abs/2205.05198

[15] A. Harlap et al., “PipeDream: Fast and Efficient Pipeline Parallel DNN Training,” Jun. 08, 2018. https://arxiv.org/abs/1806.03377

welcome

Thank you for purchasing the MEAP for *LLMs in Production*. This book is a labor of love born from our excitement and experience working with them professionally, and we're very grateful that you're interested in sharing this journey with us.

Machine Learning is everywhere, and it's becoming increasingly difficult to keep up. At the same time, the fundamentals of actually taking a ML project from idea to working product haven't really changed. When asked about when we were going to learn how to productionize the models so that people could use them, one of our deep learning teachers in college actually said, "That isn't academically interesting, so we won't be doing it." We posit that it's not only academically interesting, it's necessary for any Data Scientist, Data Engineer, and Machine Learning Engineer to have a complete picture to boost their own work.

We aim to provide you the reader with the tools to see the larger picture in production, from as many perspectives as possible to really help you at every step of the ML lifecycle. As such, we will need your help to make that happen, because we don't have your perspective. Give us feedback and help make this book something that can immediately help take any ML professional to the next level. This isn't a history or a linguistics book, but it also isn't a computer science or math book. We want you to get as much out of this book as possible. We have quite a lot of ground to cover, so be prepared to research concepts that interest you further. You'll want to have some experience with Python, and any experience you have with ML as a field will help.

This book is divided into several parts, the first of which focuses on ML foundations. We'll see different types of language models and how to solve for actual language and how to perform traditional MLOps.

The second part will go more in-depth on actual LLMs, how you can help them be ready for production from how you acquire your dataset to how to

train them and compensate for their size.

The third and final part of the book will go through several projects building and using LLMs with various goals on various hardware. We trust that it will be useful to see the application of the concepts we've gone over.

Please reach out to us with your thoughts, either through comments in the liveBook Discussion forum or in the GitHub repo accompanying the book.

— Christopher Brousseau and Matthew Sharp

In this book

welcome

Thank you for purchasing the MEAP for *LLMs in Production*. This book is a labor of love born from our excitement and experience working with them professionally, and we're very grateful that you're interested in sharing this journey with us.

Machine Learning is everywhere, and it's becoming increasingly difficult to keep up. At the same time, the fundamentals of actually taking a ML project from idea to working product haven't really changed. When asked about when we were going to learn how to productionize the models so that people could use them, one of our deep learning teachers in college actually said, "That isn't academically interesting, so we won't be doing it." We posit that it's not only academically interesting, it's necessary for any Data Scientist, Data Engineer, and Machine Learning Engineer to have a complete picture to boost their own work.

We aim to provide you the reader with the tools to see the larger picture in production, from as many perspectives as possible to really help you at every step of the ML lifecycle. As such, we will need your help to make that happen, because we don't have your perspective. Give us feedback and help make this book something that can immediately help take any ML professional to the next level. This isn't a history or a linguistics book, but it also isn't a computer science or math book. We want you to get as much out of this book as possible. We have quite a lot of ground to cover, so be prepared to research concepts that interest you further. You'll want to have some experience with Python, and any experience you have with ML as a field will help.

This book is divided into several parts, the first of which focuses on ML foundations. We'll see different types of language models and how to solve for actual language and how to perform traditional MLOps.

The second part will go more in-depth on actual LLMs, how you can help them be ready for production from how you acquire your dataset to how to

train them and compensate for their size.

The third and final part of the book will go through several projects building and using LLMs with various goals on various hardware. We trust that it will be useful to see the application of the concepts we've gone over.

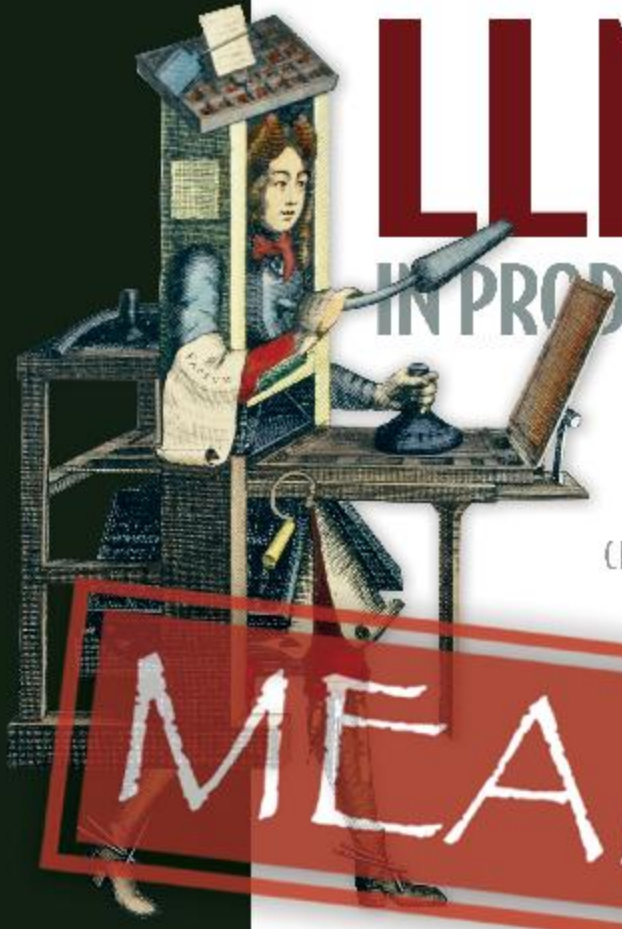
Please reach out to us with your thoughts, either through comments in the liveBook Discussion forum or in the GitHub repo accompanying the book.

— Christopher Brousseau and Matthew Sharp

In this book

From language models to successful products

LLMs IN PRODUCTION



Christopher Brousseau
Matthew Sharp

MEAP

MANNING