Large Language Models (LLMs) – Mô Hình Ngôn Ngữ Lớn

Nguyễn Quản Bá Hồng*

Ngày 15 tháng 5 năm 2025

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Large Language Models – Mô Hình Ngôn Ngữ Lớn.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/large_language_model/NQBH_large_language_model.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/large_language_model/NQBH_large_language_model.tex.
- .
  PDF: URL: .pdf.
  TEX: URL: .tex.

Mục lục

1 Basics LLMs

1.1 [Ras24]. Sebastian Raschka. Build A Large Language Model (From Scratch)

[125 Amazon ratings]

- Amazon review. Learn how to create, train, & tweak LLMs by building 1 from ground up! In *Build a Large Language Model (from Scratch)* bestselling author Sebastian Raschka guides you step by step through creating your own LLM. Each stage is explained with clear text, diagrams, & examples. Go from initial design & creation, to pretraining on a general corpus, & on to fine-tuning for specific tasks.

  Build a Large Language Model (from Scratch) teaches how to:

  - Plan & code all parts of an LLM
  - Prepare a dataset suitable for LLM training
  - Fine-tune LLMs for text classification & with your own data
  - Use human feedback to ensure your LLM follows instructions
  - Load pretrained weights into an LLM

  Build a Large Language Model (from Scratch) takes you inside AI black box to tinker with internal systems that power generative AI. As work through each key stage of LLM creation, develop an in-depth understanding of how LLMs work, their limitations, & their customization methods. Your LLM can be developed on an ordinary laptop, & used as your own personal assistant.

  About technology. Physicist Richard P. Feynman reportedly said, "I don't understand anything I can't build." Based on this same powerful principle, bestselling author Sebastian Raschka guides you step by step as build a GPT-style LLM that can run on laptop. This is an engaging book that covers each stage of process, from planning & coding to training & fine-tuning.

*A Scientist & Creative Artist Wannabe. E-mail: nguyenquanbahong@gmail.com. Bến Tre City, Việt Nam.

About book. *Build a Large Language Model (From Scratch)* is a practical & eminently-satisfying hands-on journey into foundations of generative AI. Without relying on any existing LLM libraries, code a base model, evolve it into a text classifier, & ultimately create a chatbot that can follow your conversational instructions. & really understand it because built it yourself!

What's inside.

- Plan & code an LLM comparable to GPT-2
- Load pretrained weights
- Construct a complete training pipeline
- Fine-tune your LLM for text classification
- Develop LLMs that follow human instructions

- About reader. Readers need intermediate Python skills & some knowledge of ML. LLM you create will run on any modern laptop & can optionally utilize GPUs.
- About author. Sebastian Raschka is a Staff Research Engineer at Lighting AI, where he works on LLM research & develops an open-source software. The technical editor on this book was David Caswell.
- Editorial Reviews.
  - "The most comprehensive book I've seen on building LLMs. Highly recommended!" – Raul Ciotescu, CTO, Netzinkubator Software
  - "A clear, hands-on guide that empowers readers to build their own models & explore cutting edge of AI." – Guillermo Alcántara, Project manager, PepsiCo Global
  - "Must-have resource for quickly getting up to speed on LLMs. Whether you're new to field or looking to deepen your knowledge, it's perfect guide." – Walter Reade, Staff Developer Relations Engineer, Kaggle/Google
  - "A fantastic resource for diving into LLMs – a must-read for anyone eager to get hands-on!" – Dr. Vahid Mirjalili, Senior Data Scientist, FM Global

  Pipeline: 3 main stages of coding a LLM are implementing LLM architecture & data preparation process (stage 1), pretraining an LLM to create a foundation model (stage 2), & fine-tuning foundation model to become a personal assistant or text classifier (stage 3). Each of these stages is explored & implemented in this book.

- Preface. Always been fascinated with language models. > 1 decade ago, journey into AI began with a statistical pattern classification class, which led to 1st independent project: developing a model & web application to detect mood of a song based on its lyrics.

  Fast forward to 2022, with release of ChatGPT, LLMs have taken world by storm & have revolutionized how many of us work. These models are incredibly versatile, aiding in tasks e.g. checking grammar, composing emails, summarizing lengthy documents, & much more. This is owed to their ability to parse & generate human-like text, which is important in various fields, from customer service to content creation, & even in more technical domains like coding & data analysis.

  As their name implies, a hallmark of LLMs: they are "large" – very large – encompassing millions to billions of parameters. (For comparison, using more traditional ML or statistical methods, Iris flower dataset can be classified with > 90% accuracy using a small model with only 2 parameters.) However, despite large size of LLMs compared to more traditional methods, LLMs don't have to be a black box.

  In this book, will learn how to build an LLM 1 step at a time. By end, will have a solid understanding of how an LLM, like ones used in ChatGPT, works on a fundamental level. Believe: developing confidence with each part of fundamental concepts & underlying code is crucial for success. This not only helps in fixing bugs & improving performance but also enables experimentation with new ideas.

  Several years ago, when I started working with LLMs, had to learn how to implement them hard way, sifting through many research papers & incomplete code repositories to develop a general understanding. With this book, hope to make LLMs more accessible by developing & sharing a step-by-step implementation tutorial detailing all major components & developments phases of an LLM.

  Strongly believe: best way to understand LLMs: code one from scratch – & will see that this can be fun too! Happy reading & coding!

- Acknowledgments. Writing a book is a significant undertaking – Viết 1 cuốn sách là 1 công việc quan trọng.
- About this Book. *Build a Large Language Model (From Scratch)* was written to help you understand & create your own GPT-like LLMs from ground up. It begins by focusing on fundamentals of working with text data & coding attention mechanisms & then guides through implementing a complete GPT model from scratch. Book then covers pretraining mechanism as well as fine-tuning for specific tasks e.g. text classification & following instructions. By end of this book, have a deep understanding of how LLMs work & skills to build your own models. While models you'll create are smaller in scale compared to large foundational models, they use same concepts & serve as powerful educational tools to grasp core mechanisms & techniques used in building state-of-art LLMs.

- Who should read this book. *Build a Large Language Model (From Scratch)* is for ML enthusiasts, engineers, researchers, students, & practitioners who want to gain a deep understanding of how LLMs work & learn to build their own models from scratch. Both beginners & experienced developers will be able to use their existing skills & knowledge to grasp concepts & techniques used in creating LLMs.

  What sets this book apart is its comprehensive coverage of entire process of building LLMs, from working with datasets to implementing model architecture, pretraining on unlabeled data, & fine-tuning for specific tasks. As of this writing, no other resource provides such a complete & hands-on approach to building LLMs from ground up.

  To understand code examples in this book, should have a solid grasp of Python programming. While some familiarity with ML, DL, & AI can be beneficial, an extensive background in these areas is not required. LLMs are a unique subset of AI, so even if relatively new to field, will be able to follow along.

  If have some experience with deep neural networks, may find certain concepts more familiar, as LLMs are built upon these architectures. However, proficiency in PyTorch is not a prerequisite. Appendix A provides a concise introduction to PyTorch, equipping with necessary skills to comprehend code examples throughout book.

  A high school-level understanding of mathematics, particularly working with vectors & matrices, can be helpful as explore inner workings of LLMs. However, advanced mathematical knowledge is not necessary to grasp key concepts & ideas presented in this book.

  Most important prerequisite is a strong foundation in Python programming. With this knowledge, will be well prepared to explore fascinating world of LLMs & understand concepts & code examples presented in this book.

- How this book is organized: A roadmap. This book is designed to be read sequentially, as each chap builds upon concepts & techniques introduced in prev ones. Book is divided into 7 chaps that cover essential aspects of LLMs & their implementation.
  * Chap. 1 provides a high-level introduction to fundamental concepts behind LLMs. It explores transformer architecture, which forms basis for LLMs e.g. those used on ChatGPT platform.
  * Chap. 2 lays out a plan for building an LLM from scratch. It covers process of preparing text for LLM training, including splitting text into word & subword tokens, using byte pair encoding for advanced tokenization, sampling training examples with a sliding window approach, & converting tokens into vectors that feed into LLM.
  * Chap. 3 focuses on attention mechanisms used in LLMs. It introduces a basic self-attention framework & progresses to an enhanced self-attention mechanism. Chap also covers implementation of a causal attention module that enables LLMs to generate 1 token at a time, masking randomly selected attention weights with dropout to reduce overfitting & stacking multiple causal attention modules into a multihead attention module.
  * Chap. 4 focuses on coding a GPT-like LLM that can be trained to generate human-like text. It covers techniques e.g. normalizing layer activations to stabilize neural network training, adding shortcut connections in deep neural networks to train models more effectively, implementing transformer blocks to create GPT models of various sizes, & computing number of parameters & storage requirements of GPT models.
  * Chap. 5 implements pretraining process of LLMs. It covers computing training & validation set losses to assess quality of LLM-generated text, implementing a training function & pretraining LLM, saving & loading model weights to continue training an LLM, & loading pretrained weights form OpenAI.
  * Chap. 6 introduces different LLM fine-tuning approaches. It covers preparing a dataset for text classification, modifying a pretrained LLM for fine-tuning, fine-tuning an LLM to identify spam messages, & evaluating accuracy of a fine-tuned LLM classifier.
  * Chap. 7 explores instruction fine-tuning process of LLMs. It covers preparing a dataset for supervised instruction fine-tuning, organizing instruction data in training batches, loading a pretrained LLM & fine-tuning it to follow human instructions, extracting LLM-generated instruction responses for evaluation, & evaluating an instruction-fine-tuned LLM.

- About code. To make it as easy as possible to follow along, all code examples in this book are conveniently available on Manning website at `https://www.manning.com/books/build-a-large-language-model-from-scratch`, as well as in Jupyter notebook format on GitHub at `https://github.com/rasbt/LLMs-from-scratch`. Solutions to all code exercises can be found in appendix C.

  This book contains many examples of source code both in numbered listings & in line with normal text.

  In many cases, original source code has been reformatted; added line breaks & reworked indentation to accommodate available page space in book. In rare cases, even this was not enough, & listings include line-continuation markers. Additionally, comments in source code have often been removed from listings when code is described in text. Code annotations accompany many of listings, highlighting important concepts.

  1 of key goals of this book is accessibility, so code examples have been carefully designed to run efficiently on a regular laptop, without need for any special hardware. But if do have access to a GPU, certain sects provide helpful tips on scaling up datasets & models to take advantage of that extra power.

  Throughout book, use PyTorch as our go-to tensor & a DL library to implement LLMs from ground up. If PyTorch is new to you, I recommend you start with appendix A, which provides an in-depth introduction, complete with setup recommendations.

- Other online resources. Interested in latest AI & LLM research trends? → Check out blog at `https://magazine.sebastianraschka.com/`, where regularly discuss latest AI research with a focus on LLMs.

  Need help getting up to speed with DL & PyTorch? → Offer several free courses on website at `https://sebastianraschka.com/teaching/`. These resources can help you quickly get up to speed with latest techniques.

Looking for bonus materials related to book? → Visit book's GitHub repo at https://github.com/rasbt/LLMs-from-scratch to find additional resources & examples to supplement your learning.

- 1. Understanding LLMs. Cover:
  - High-level explanations of fundamental concepts behind LLMs
  - Insights into transformer architecture from which LLMs are derived
  - A plan for building an LLM from scratch

LLMs, e.g. those offered in OpenAI's ChatGPT, are deep neural network models that have been developed over the past few years. They ushered in a new era for NLP. Before advent of LLMs, traditional methods excelled at categorization tasks e.g. email spam classification & straightforward pattern recognition that could be captured with handcrafted rules or simpler models. However, they typically underperformed in language tasks that demanded complex understanding & generation abilities, e.g. parsing detailed instructions, conducting contextual analysis, & creating coherent & contextually appropriate original text. E.g., previous generations of language models could not write an email from a list of keywords – a task that is trivial for contemporary LLMs.

– LLM, ví dụ như những LLM được cung cấp trong ChatGPT của OpenAI, là các mô hình mạng nơ-ron sâu đã được phát triển trong vài năm qua. Chúng mở ra 1 kỷ nguyên mới cho NLP. Trước khi LLM ra đời, các phương pháp truyền thống đã xuất sắc trong các nhiệm vụ phân loại, ví dụ như phân loại thư rác email & nhận dạng mẫu đơn giản có thể được nắm bắt bằng các quy tắc thủ công hoặc các mô hình đơn giản hơn. Tuy nhiên, chúng thường hoạt động kém trong các nhiệm vụ ngôn ngữ đòi hỏi khả năng hiểu phức tạp & tạo ra, ví dụ như phân tích các hướng dẫn chi tiết, tiến hành phân tích theo ngữ cảnh, & tạo ra văn bản gốc mạch lạc & phù hợp với ngữ cảnh. Ví dụ, các thế hệ mô hình ngôn ngữ trước đây không thể viết email từ danh sách các từ khóa – 1 nhiệm vụ tầm thường đối với LLM đương đại.

LLMs have remarkable capabilities to understand, generate, & interpret human language. However, important to clarify that when we say language models "understand," i.e., they can process & generate text in ways that appear coherent & contextually relevant, not that they possess human-like consciousness or comprehension.

Enabled by advancements in DL, which is a subset of ML & AI focused on neural networks, LLMs are trained on vast quantities of text data. This large-scale training allows LLMs to capture deeper contextual information & subtleties of human language compared to previous approaches. As a result, LLMs have significantly improved performance in a wide range of NLP tasks, including text translation, sentiment analysis, question answering, & many more.

Another important distinction between contemporary LLMs & earlier NLP models is that earlier NLP models were typically designed for specific tasks, e.g. text categorization, language translation, etc. While those earlier NLP models excelled in their narrow applications, LLMs demonstrate a broader proficiency across a wide range of NLP tasks.

Success behind LLMs can be attribute to transformer architecture that underpins many LLMs & vast amounts of data on which LLMs are trained, allowing them to capture a wide variety of linguistic nuances, contexts, & patterns that would be challenging to encode manually.

This shift toward implementing models based on transformer architecture & using large training datasets to train LLMs has fundamentally transformed NLP, providing more capable tools for understanding & interacting with human language.

Following discussion sets a foundation to accomplish primary objective of this book: understanding LLMs by implementing a ChatGPT-like LLM based on transformer architecture step by step in code.

- 1.1. What is an LLM? An LLM is a neural network designed to understand, generate, & respond to human-like text. These models are deep neural networks trained on massive amounts of text data, sometimes encompassing large portions of entire publicly available text on internet.

  "Large" in "LLM" refers to both model's size in terms of parameters & immense dataset on which it's trained. Models like this often have 10s or even hundreds of billions of parameters, which are adjustable weights in network that are optimized during training to predict next word in a sequence. Next-word prediction is sensible because it harnesses inherent sequential nature of language to train models on understanding context, structure, & relationships within text. Yet, it is a very simple task, & so it is surprising to many researchers that it can produce such capable models. In later chaps, discuss & implement next-word training procedure step by step.

  LLMs utilize an architecture called *transformers*, which allows them to pay selective attention to different parts of input when making predictions, making them especially adept at handling nuances & complexities of human language.

  Since LLMs are capable of *generating* text, LLMs are also often referred to as a form of generative AI, often abbreviated as *generative AI* or *GenAI*. As illustrated in Fig. 1.1: As this hierarchical depiction of relationship between different fields suggests, LLMs represent a specific application of DL techniques, using their ability to process & generate human-like text. DL is a specialized branch of ML that focuses on using multilayer neural networks. ML & DL are fields aimed at implementing algorithms that enable computers to learn from data & perform tasks that typically require human intelligence., AI encompasses broader field of creating machines that can perform tasks requiring human-like intelligence, including understanding language, recognizing patterns, & making decisions, & includes subfields like ML & DL.

  - AI: Systems with human-like intelligence
  - ML: Algorithms that learn rules automatically from data

* DL: ML with neural networks consisting of many layers
* LLM: Deep neural network for parsing & generating human-like text
* GenAI: GenAI involves use of deep neural networks to create new content, e.g., text, images, or various forms of media

Algorithms used to implement AI are focus of field of ML. Specifically, ML involves development of algorithms that can learn from & make predictions or decisions based on data without being explicitly programmed. To illustrate this, imagine a spam filter as a practical application of ML. Instead of manually writing rules to identify spam emails, a ML algorithm is fed examples of emails labeled as spam & legitimate emails. By minimizing error in its predictions on a training dataset, model then learns to recognize patterns & characteristics indicative of spam, enabling it to classify new emails as either spam or not spam.

DL is a subset of ML that focuses on utilizing neural networks with 3 or more layers (also called deep neural networks) to model complex patterns & abstractions in data. In contrast to DL, traditional ML requires manual feature extraction. I.e., human experts need to identify & select most relevant features for model.

While field of AI is now dominated by ML & DL, it also includes other approaches – e.g., using rule-based systems, genetic algorithms, expert systems, fuzzy logic, or symbolic reasoning.

Returning to spam classification example, in traditional ML, human experts might manually extract features from email text e.g. frequency of certain trigger words (e.g., "prize", "win", "free"), number of exclamation marks, use of all uppercase words, or presence of suspicious links. This dataset, created based on these expert-defined features, would then be used to train model. In contrast to traditional ML, $\boxed{\text{DL does not require manual feature extraction}}$. I.e., human experts do not need to identify & select most relevant features for a DL model. (However, both traditional ML & DL for spam classification still require collection of labels, e.g. spam or non-spam, which need to be gathered either by an expert or users.)

Look at some of problems LLMs can solve today, challenges that LLMs address, & general LLM architecture we will implement later.

- 1.2. Applications of LLMs. Owing to their advanced capabilities to parse & understand unstructured text data, LLMs have a broad range of applications across various domains. Today, LLms are employed for machine translation, generation of novel texts (see Fig. 1.2: LLM interfaces enable natural language communication between users & AI systems. This screenshot shows ChatGPT writing a poem according to a user's specifications.), sentiment analysis, text summarization, & many other tasks. LLMs have recently been used for content creation, e.g. writing fiction, articles, & even computer code.

  LLMs can also power sophisticated chatbots & virtual assistants, e.g. OpenAI's ChatGPT or Google's Gemini (formerly called Bard), which can answer user queries & augment traditional search engines e.g. Google Search or Microsoft Bing.

  Moreover, LLMs may be used for effective knowledge retrieval from vast volumes of text in specialized areas e.g. medicine or law. This includes sifting through document, summarizing lengthy passages, & answering technical questions.

  In short, LLMs are invaluable for automating almost any task that involves parsing & generating text. Their applications are virtually endless, & as we continue to innovate & explore new ways to use these models, clear: LLMs have potential to redefine our relationship with technology, making it more conversational, intuitive, & accessible.

  Will focus on understanding how LLMs work from ground up, coding an LLM that can generate texts. Will also learn about techniques that allow LLMs to carry out queries, ranging from answering questions to summarizing text, translating text into different languages, & more. I.e., will learn how complex LLM assistants e.g. ChatGPT work by building 1 step by step.

- 1.3. Stages of building & using LLMs. Why should we build our own LLMs? Coding an LLM from ground up is an excellent exercise to understand its mechanics & limitations. Also, it equips us with required knowledge for pretraining or fine-tuning existing open source LLM architectures to our own domain-specific datasets or tasks.

  – Tại sao chúng ta nên xây dựng LLM của riêng mình? Việc mã hóa LLM từ đầu là 1 bài tập tuyệt vời để hiểu cơ chế & hạn chế của nó. Ngoài ra, nó trang bị cho chúng ta kiến thức cần thiết để đào tạo trước hoặc tinh chỉnh các kiến trúc LLM nguồn mở hiện có cho các tập dữ liệu hoặc tác vụ cụ thể theo miền của riêng chúng ta.

  Note 1. *Most LLMs today are implemented using PyTorch DL library, which is what we will use. Readers can find a comprehensive introduction to PyTorch in appendix A.*

  Research has shown: when it comes to modeling performance, customs-built LLMs – those tailored for specific tasks or domains – can outperform general-purpose LLMs, e.g. those provided by ChatGPT, which are designed for a wide array of applications. Examples of these include BloombergGPT (specialized for finance) & LLMs tailored for medical question answering (see appendix B for more details).

  Using custom-built LLMs offers several advantages, particularly regarding data privacy. E.g., companies may prefer not to share sensitive data with 3rd-party LLM providers like OpenAI due to confidentiality concerns. Additionally, developing smaller custom LLMs enables deployment directly on customer devices, e.g. laptops & smartphones, which is something companies like Apple are currently exploring. This local implementation can significantly decrease latency & reduce server-related costs. Furthermore, custom LLMs grant developers complete autonomy, allowing them to control updates & modifications to model as needed.

  – Sử dụng LLM tùy chỉnh mang lại 1 số lợi thế, đặc biệt là về quyền riêng tư dữ liệu. Ví dụ, các công ty có thể không muốn chia sẻ dữ liệu nhạy cảm với các nhà cung cấp LLM bên thứ 3 như OpenAI do lo ngại về tính bảo mật. Ngoài ra, việc phát triển các LLM tùy chỉnh nhỏ hơn cho phép triển khai trực tiếp trên các thiết bị của khách hàng, ví dụ như máy tính xách tay & điện thoại thông minh, đây là điều mà các công ty như Apple hiện đang khám phá. Việc triển khai cục bộ này có thể giảm đáng kể độ trễ & giảm chi phí liên quan đến máy chủ. Hơn nữa, các LLM tùy chỉnh cấp cho các nhà phát triển quyền tự chủ hoàn toàn, cho phép họ kiểm soát các bản cập nhật & sửa đổi mô hình khi cần.

General process of creating an LLM includes pretraining & fine-tuning. "Pre" in "preparing" refers to initial phase where a model like an LLM is trained on a large, diverse dataset to develop a broad understanding of language. This pre-trained model then serves as a foundational resource that can be further refined through fine-tuning, a process where model is specifically trained on a narrower dataset that is more specific to particular tasks or domains. This 2-stage training approach consisting of pretraining & fine-tuning is depicted in Fig. 1.3: Pretraining an LLM involves next-word prediction on large text datasets. A pretrained LLM can then be fine-tuned using a smaller labeled dataset.

1st step in creating an LLM: train it on a large corpus of text data, sometimes referred to as *raw* text. Here, "raw" refers to fact that this data is just regular text without any labeling information. (Filtering may be applied, e.g. removing formatting characters or documents in unknown languages.)

Note 2. *Readers with a background in ML may note: labeling information is typically required for traditional ML models & deep neural networks trained via conventional supervised learning paradigm. However, this is not case for pretraining stage of LLMs. In this phase, LLMs use self-supervised learning, where model generates its own labels from input data.*

This 1st training stage of an LLM is also known as *pretraining*, creating an initial pretrained LLM, often called a *base* or *foundation model*. A typical example of such a model is GPT-3 model (precursor of original model offered in ChatGPT). This model is capable of text completion – i.e., finishing a half-written sentence provided by a user. It also has limited few-shot capabilities, i.e., it can learn to perform new tasks based on only a few examples instead of needing extensive training data.

After obtaining a pretrained LLM from training on large text datasets, where LLM is trained to predict next word in text, we can further train LLM on labeled data, also known as *fine-tuning*.

2 most popular categories of fine-tuning LLMs are *instruction fine-tuning* & *classification fine-tuning*. In instruction fine-tuning, labeled dataset consists of instruction & answer pairs, e.g. a query to translate a text accompanied by correctly translated text. In classification fine-tuning, labeled dataset consists of texts & associated class labels – e.g., emails associated with "spam" & "hot spam" labels.

– 2 loại LLM tinh chỉnh phổ biến nhất là *tinh chỉnh hướng dẫn* & *tinh chỉnh phân loại*. Trong tinh chỉnh hướng dẫn, tập dữ liệu được gắn nhãn bao gồm các cặp hướng dẫn & câu trả lời, ví dụ: truy vấn để dịch văn bản đi kèm với văn bản được dịch chính xác. Trong tinh chỉnh phân loại, tập dữ liệu được gắn nhãn bao gồm các văn bản & nhãn lớp liên quan – ví dụ: email liên quan đến nhãn "spam" & "hot spam".

Cover code implementations for pretraining & fine-tuning an LLM, & delve deeper into specifics of both instruction & classification fine-tuning after pretraining a base LLM.

- 1.4. Introducing transformer architecture. Most modern LLMs rely on *transformer* architecture, which is a deep neural network architecture introduced in 2017 paper "Attention Is All You Need". To understand LLMs, must understand original transformer, which was developed for machine translation, translating English texts to German & French. A simplified version of transformer architecture is depicted in Fig. 1.4: A simplified depiction of original transformer architecture, which is a DL model for language translation. Transformer consists of 2 parts: (a) an encoder that processes input text & produces an embedding representation (a numerical representation that captures many different factors in different dimensions) of text that (b) decoder can use to generate translated text 1 word at a time. This figure shows final stage of translation process where decoder has to generate only final word ("Beispiel"), given original input text ("This is an example") & a partially translated sentence ("Das ist ein"), to complete translation.

  Transformer architecture consists of 2 submodules: an encoder & a decoder. Encoder module processes input text & encodes it into a series of numerical representations or vectors that capture contextual information of input. Then, decoder module takes these encoded vectors & generates output text. In a translation task, e.g., encoder would encode text from source language into vectors, & decoder would decode these vectors to generate text in target language. Both encoder & decoder consist of many layers connected by a so-called self-attention mechanism. You may have many questions regarding how inputs are preprocessed & encoded. These will be addressed in a step-by-step implementation in subsequent chaps.

  – Kiến trúc máy biến áp bao gồm 2 mô-đun con: 1 bộ mã hóa & 1 bộ giải mã. Mô-đun mã hóa xử lý văn bản đầu vào & mã hóa nó thành 1 chuỗi các biểu diễn số hoặc vectơ nắm bắt thông tin ngữ cảnh của đầu vào. Sau đó, mô-đun giải mã lấy các vectơ được mã hóa này & tạo văn bản đầu ra. Trong 1 tác vụ dịch, ví dụ, bộ mã hóa sẽ mã hóa văn bản từ ngôn ngữ nguồn thành các vectơ, & bộ giải mã sẽ giải mã các vectơ này để tạo văn bản ở ngôn ngữ đích. Cả bộ mã hóa & bộ giải mã đều bao gồm nhiều lớp được kết nối bởi cái gọi là cơ chế tự chú ý. Bạn có thể có nhiều câu hỏi liên quan đến cách đầu vào được xử lý trước & mã hóa. Những câu hỏi này sẽ được giải quyết trong phần triển khai từng bước trong các chương tiếp theo.

  A key component of transformers & LLMs is self-attention mechanism (not shown), which allows model to weigh importance of different words or tokens in a sequence relative to each other. This mechanism enables model to capture long-range dependencies & contextual relationships within input data, enhancing its ability to generate coherent & contextually relevant output. However, due to its complexity, will defer further explanation to Chap. 3, where discuss & implement it step & step.

  Later variants of transformer architecture, e.g. BERT (short for *bidirectional encoder representations from transformers*) & various GPT models (short for *generative pretrained transformers*), built on this concept to adapt this architecture for different tasks. If interested, refer to Appendix B for further reading suggestions.

  BERT, which is built upon original transformer's encoder submodule, differs in its training approach from GPT. While GPT is designed for generative tasks, BERT & its variants specialize in masked word prediction, where model predicts masked or hidden words in a given sentence, as shown in Fig. 1.5: A visual representation of transformer's encoder & decoder submodules. On left, encoder segment exemplifies BERT-like LLMs, which focus on masked word prediction & are primarily used for tasks like text classification. On right, decoder segment showcases GPT-like LLMs, designed for generative tasks & producing coherent text sequences. This unique training strategy equips BERT with strengths in text classification tasks, including sentiment prediction & document categorization. As an application of its capabilities, as of this writing, X (formerly Twitter) uses BERT to detect toxic content.

GPT, on other hand, focuses on decoder portion of original transformer architecture & is designed for tasks that require generating texts. This includes machine translation, text summarization, fiction writing, writing computer code, & more.

GPT models, primarily designed & trained to perform text completion tasks, also show remarkable versatility in their capabilities. These models are adept at executing both zero-shot & few-shot learning tasks. Zero-shot learning refers to ability to generalize to completely unseen tasks without any prior specific examples. On other hand, few-shot learning involves learning from a minimal number of examples user provides as input, as shown in Fig. 1.6: In addition to text completion, GPT-like LLMs can solve various tasks based on their inputs without needing retraining, fine-tuning, or task-specific model architecture changes. Sometimes it is helpful to provide examples of target within input, which is known as a few-shot setting. However, GPT-like LLMs are also capable of carrying out tasks without a specific example, which is called zero-shot setting.

Remark 1 (Transformers vs. LLMs)**.** *Today's LLMs are based on transformer architecture. Hence, transformers & LLMs are terms that are often used synonymously in literature. However, note: not all transformers are LLMs since transformers can also be used for computer vision. Also, not all LLMs are transformers, as there are LLMs based on recurrent & convolutional architectures. Main motivation behind these alternative approaches: improve computational efficiency of LLMs. Whether these alternative LLM architectures can compete with capabilities of transformer-based LLMs & whether they are going to be adopted in practice remains to be seen. For simplicity, use term "LLM" to refer to transformer-based LLMs similar to GPT. (Interested readers can find literature references describing these architectures in Appendix B.)*

– LLM ngày nay dựa trên kiến trúc máy biến áp. Do đó, máy biến áp & LLM là những thuật ngữ thường được sử dụng đồng nghĩa trong tài liệu. Tuy nhiên, lưu ý: không phải tất cả máy biến áp đều là LLM vì máy biến áp cũng có thể được sử dụng cho thị giác máy tính. Ngoài ra, không phải tất cả LLM đều là máy biến áp, vì có những LLM dựa trên kiến trúc hồi quy & tích chập. Động lực chính đằng sau các phương pháp tiếp cận thay thế này: cải thiện hiệu quả tính toán của LLM. Liệu các kiến trúc LLM thay thế này có thể cạnh tranh với khả năng của LLM dựa trên máy biến áp & liệu chúng có được áp dụng trong thực tế hay không vẫn còn phải chờ xem. Để đơn giản, hãy sử dụng thuật ngữ "LLM" để chỉ các LLM dựa trên máy biến áp tương tự như GPT. (Những độc giả quan tâm có thể tìm thấy tài liệu tham khảo mô tả các kiến trúc này trong Phụ lục B.)

- 1.5. utilizing large datasets. Large training datasets for popular GPT- & BERT-like models represent diverse & comprehensive text corpora encompassing billions of words, which include a vast array of topics & natural & computer languages. To provide a concrete example, Table 1.1: Pretraining dataset of popular GPT-3 LLM. summarizes dataset used for pretraining GPT-3, which served as base model for 1st version of ChatGPT. Table 1.1 reports number of tokens, where a token is a unit of text that a model reads & number of tokens in a dataset is roughly equivalent to number of words & punctuation characters in text. Chap. 2 addresses tokenization, process of converting text into tokens.

  Main takeaways: scale & diversity of this training dataset allow these models to perform well on diverse tasks, including language syntax, semantics, & context – even some requiring general knowledge.

  Remark 2 (GPT-3 dataset details)**.** *Table 1.1 displays dataset used for GPT-3. Proportions column in table sums up to 100% of sampled data, adjusted for rounding errors. Although subsets in Number of Tokens column total 499 billion, model was trained on only 300 billion tokens. Authors of GPT-3 paper did not specify why model was not trained on all 499 billion tokens.*

  For context, consider size of CommonCrawl dataset, which alone consists of 410 billion tokens & requires about 570 GB of storage. In comparison, later iterations of models like GPT-3, e.g. Meta's LLaMA, have expanded their training scope to include additional data source like Arxiv research papers (92 GB) & StackExchange's code-related Q&As (78 GB).

  Authors of GPT-3 paper did not share training dataset, but a comparable dataset that is publicly available is Dolma: An Open Corpus of 3 Trillion Tokens for LLM Pretraining Research by Soldaini et al. 2024 (https://arxiv.org/abs/2402.00159). However, collection may contain copyrighted works, & exact usage terms may depend on intended use case & country.

  Pretrained nature of these models makes them incredibly versatile for further fine-tuning on downstream tasks, which is why they are also known as base or foundation models. Pretraining LLMs requires access to significant resources & is very expensive. E.g., GPT-3 pretraining cost is estimated to be $4.6 million in terms of cloud computing credits https://www.reddit.com/r/MachineLearning/comments/h0jwoz/d_gpt3_the_4600000_language_model/.

  Good news: many pretrained LLMs, available as open source models, can be used as general-purpose tools to write, extract, & edit texts that were not part of training data. Also, LLMs can be fine-tuned on specific tasks with relatively smaller datasets, reducing computational resources needed & improving performance.

  Will implement code for pretraining & use it to pretrain an LLM for educational purposes. All computations are executable on consumer hardware. After implementing pretraining code, will learn how to reuse openly available model weights & load them into architecture we will implement, allowing us to skip expensive pretraining stage when fine-tune our LLM.

- 1.6. A closer look at GPT architecture. GPT was originally introduced in paper "Improving Language Understanding by Generative Pre-Training" https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf by by Radford et al. from OpenAI. GPT-3 is a scaled-up version of this model that has more parameters & was trained on a larger dataset. In addition, original model offered in ChatGPT was created by fine-tuning GPT-3 on a large instruction dataset using a method from OpenAI's InstructGPT paper https://arxiv.org/pdf/2203.02155. As Fig. 1.6 shows, these models are competent text completion models & can carry out other tasks e.g. spelling correction, classification, or language translation. This is actually very remarkable given that GPT models are pretrained on a relatively simple next-word prediction task, as depicted in Fig. 1.7: In next-word prediction pretraining task for GPT models, system learns to predict upcoming word in a sentence by looking at words that have come before it. This approach helps model understand how words & phrases typically fit together in language, forming a foundation that can be applied to various other tasks.

  Next-word prediction task is a form of self-supervised learning, which is a form of self-labeling. I.e. we don't need to collect labels for training data explicitly but can use structure of data itself: can use next word in a sentence or document as label that model is supposed to predict. Since this next-word prediction task allows us to create labels "on the fly", possible to use massive unlabeled text datasets to train LLMs.

  – Nhiệm vụ dự đoán từ tiếp theo là 1 dạng học tự giám sát, là 1 dạng tự dán nhãn. Tức là chúng ta không cần phải thu thập nhãn cho dữ liệu đào tạo 1 cách rõ ràng nhưng có thể sử dụng cấu trúc của chính dữ liệu: có thể sử dụng từ tiếp theo trong 1 câu hoặc tài liệu làm nhãn mà mô hình được cho là dự đoán. Vì nhiệm vụ dự đoán từ tiếp theo này cho phép chúng ta tạo nhãn "ngay lập tức", có thể sử dụng các tập dữ liệu văn bản không có nhãn lớn để đào tạo LLM.

  Compared to original transformer architecture covered in Sect. 1.4, general GPT architecture is relatively simple. Essentially, just decoder part without encoder (Fig. 1.8: GPT architecture employs only decoder portion of original transformer. It is designed for unidirectional, left-to-right processing, making it well suited for text generation & next-word prediction tasks to generate text in an iterative fashion, 1 word at a time.). Since decoder-style models like GPT generate text by predicting text 1 word at a time, they are considered a type of *autoregressive* model. Autoregressive models incorporate their previous outputs as inputs for future predictions. Consequently, in GPT, each new word is chosen based on sequence that precedes it, which improves coherence of resulting text.

  – So với kiến trúc biến áp ban đầu được đề cập trong Phần 1.4, kiến trúc GPT nói chung tương đối đơn giản. Về cơ bản, chỉ có phần giải mã mà không có phần mã hóa (Hình 1.8: Kiến trúc GPT chỉ sử dụng phần giải mã của biến áp ban đầu. Nó được thiết kế để xử lý một chiều, từ trái sang phải, khiến nó rất phù hợp cho các tác vụ tạo văn bản & dự đoán từ tiếp theo để tạo văn bản theo cách lặp lại, từng từ một.). Vì các mô hình kiểu giải mã như GPT tạo văn bản bằng cách dự đoán văn bản từng từ một, nên chúng được coi là một loại mô hình *tự hồi quy*. Các mô hình tự hồi quy kết hợp các đầu ra trước đó của chúng làm đầu vào cho các dự đoán trong tương lai. Do đó, trong GPT, mỗi từ mới được chọn dựa trên trình tự trước đó, giúp cải thiện tính mạch lạc của văn bản kết quả.

  Architectures e.g. GPT-3 are also significantly larger than original transformer model. E.g., original transformer repeated encoder & decoder blocks 6 times. GPT-3 has 96 transformer layers & 175 billion parameters in total.

  GPT-3 was introduced in 2020, which, by standards of DL & LLM development, is considered a long time ago. However, more recent architectures, e.g. Meta's Llama models, are still based on same underlying concepts, introducing only minor modifications. Hence, understanding GPT remains as relevant as ever, so I focus on implementing prominent architecture behind GPT while providing pointers to specific tweaks employed by alternative LLMs.

  Although original transformer model, consisting of encoder & decoder blocks, was explicitly designed for language translation, GPT models – despite their larger yet simpler decoder-only architecture aimed at next-word prediction – are also capable of performing translation tasks. This capability was initially unexpected to researchers, as it emerged from a model primarily trained on a next-word prediction task, which is a task that did not specifically target translation.

  Ability to perform tasks that model wasn't explicitly trained to perform is called an *emergent behavior*. This capability isn't explicitly taught during training but emerges as a natural consequence of model's exposure to vast quantities of multilingual data in diverse contexts. Fact GPT models can "learn" translation patterns between languages & perform translation tasks even though they weren't specifically trained for it demonstrates benefits & capabilities of these large-scale, generative language models. Can perform diverse tasks without using diverse models for each.

- 1.7. Building a LLM. Now we've laid groundwork for understanding LLMs, code 1 from scratch. Take fundamental idea behind GPT as a blueprint & tackle this in 3 stages, as outlined in Fig. 1.9: 3 main stages of coding an LLM are implementing LLM architecture & data preparation process (stage 1), pretraining an LLM to create a foundation model (stage 2), & fine-tuning foundation model to become a personal assistant or text classifier (stage 3). In stage 1, learn about fundamental data preprocessing steps & code attention mechanism at heart of every LLM. Next, in stage 2, learn how to code & pretrain a GPT-like LLM capable of generating new texts. Also go over fundamentals of evaluating LLMs, which is essential for developing capable NLP systems.

  Pretraining an LLM from scratch is a significant endeavor, demanding thousands to millions of dollars in computing costs for GPT-like models. Therefore, focus of stage 2 is on implementing training for educational purposes using a small dataset. In addition, also provide code examples for loading openly available model weights.

  Finally, in stage 3, take a pretrained LLM & fine-tune it to follow instructions e.g. answering queries or classifying texts – most common tasks in many real-world applications & research.

- Summary.

  * LLMs have transformed field of NLP, which previously mostly relied on explicit rule-based systems & simpler statistical

methods. Advent of LLMs introduced new DL-driven approaches that led to advancements in understanding, generating, & translating human language.

- ∗ Modern LLMs are trained in 2 main steps:
  - · 1st, they are pretrained on a large corpus of unlabeled text by using prediction of next word in a sentence as a label.
  - · Then, they are fine-tuned on a smaller, labeled target dataset to follow instructions or perform classification tasks.
- ∗ LLMs are based on transformer architecture. Key idea of transformer architecture is an attention mechanism that gives LLM selective access to whole input sentence when generating output 1 word at a time.
- ∗ Original transformer architecture consists of an encoder for parsing text & a decoder for generating text.
- ∗ LLms for generating text & following instructions, e.g. GPT-3 & ChatGPT, only implement decoder modules, simplifying architecture.
- ∗ Large datasets consisting of billions of words are essential for pretraining LLMs.
- ∗ While general pretraining task for GPT-like models is to predict next word in a sentence, these LLMs exhibit emergent properties, e.g. capabilities to classify, translate, or summarize texts.
- ∗ Once an LLM is pretrained, resulting foundation model can be fine-tuned more efficiently for various downstream tasks.
- ∗ LLMs fine-tuned on custom datasets can outperform general LLMs on specific tasks.

- 2. Working with text data. Covers:
  - ○ Preparing text for LLM training
  - ○ Splitting text into word & subword tokens
  - ○ Byte pair encoding as a more advanced way of tokenizing text.
  - ○ Sampling training examples with a sliding window approach
  - ○ Converting tokens into vectors that feed into a LLM

  So far, have covered general structure of LLMs & learned that they are pretrained on vast amounts of text. Specifically, our focus was on decoder-only LLMs based on transformer architecture, which underlies models used in ChatGPT & other popular GPT-like LLMs.

  During pretraining stage, LLMs process text 1 word at a time. Training LLMs with millions to billions of parameters using a next-word prediction tasks yields models with impressive capabilities. These models can then be further fine-tuned to follow general instructions or perform specific target tasks. But before we can implement & train LLMs, need to prepare training dataset, as illustration in Fig. 2.1: 3 main stages of coding an LLM. This chap focuses on step 1 of Stage 1: implementing data sample pipeline.

  Learn how to prepare input text for training LLMs. This involves splitting text into individual word & subword tokens, which can then be encoded into vector representations for LLM. Also learn about advanced tokenization schemes like byte pair encoding, which is utilized in popular LLMs like GPT. Lastly, implement a sampling & data-loading strategy to produce input-output pairs necessary for training LLMs.

  - ○ 2.1. Understanding word embeddings. Deep neural network models, including LLMs, cannot process raw text directly. Since text is categorical, it isn't compatible with mathematical operations used to implement & train neural networks. Therefore, need a way to represent words as continuous-valued vectors.

    Note 3. *Readers unfamiliar with vectors & tensors in a computational context can learn more in Appendix A, Sect. A.2.2.*

    Concept of converting data into a vector format is often referred to as *embedding*. Using a specific neural network layer or another pretrained neural network model, can embed different data types – e.g., video, audio, & text, as illustrated in Fig. 2.2: DL models cannot process data formats like video, audio, & text in their raw form. Thus, use an embedding model to transform this raw data into a dense vector representation that DL architectures can easily understand & process. Specifically, this figure illustrates process of converting raw data into a 3D numerical vector. However, important to note: different data formats require distinct embedding models. E.g., an embedding model designed for text would not be suitable for embedding audio or video data.

    At its core, an embedding is a mapping from discrete objects, e.g. words, images, or even entire documents, to points in a continuous vector space – primary purpose of embeddings: convert nonnumeric data into a format that neural networks can process.

    While word embeddings are most common form of text embedding, there are also embeddings for sentences, paragraphs, or whole documents. Sentence or paragraph embeddings are popular choices for *retrieval-augmented generation.* Retrieval-augmented generation combines generation (like producing text) with retrieval (like searching an external knowledge base) to pull relevant information when generation text, which is a technique that is beyond scope of this book. Since goal: train GPT-like LLms, which learn to generate text 1 word at a time, focus on word embeddings.

    Several algorithms & frameworks have been developed to generate word embeddings. 1 of earlier & most popular examples is *Word2Vec* approach. *Word2Vec* trained neural network architecture to generate word embeddings by predicting context of a word given target word or vice versa. Main idea behind Word2Vec: words that appear in similar contexts tend to have similar meanings. Consequently, when projected into 2D word embeddings for visualization purposes, similar terms are clustered together, as shown in Fig. 2.3: If word embeddings are 2D, can plot them in a 2D scatterplot for visualization purposes as shown here. When using word embedding techniques, e.g. Word2Vec, words corresponding to similar concepts often appear close to each other in embedding space. E.g., different types of birds appear closer to each other in embedding space than in countries & cities.

Word embeddings can have varying dimensions, from 1 to thousands. A higher dimensionality might capture more nuanced relationships but at cost of computational efficiency.

– Nhúng từ có thể có nhiều kích thước khác nhau, từ 1 đến hàng nghìn. Kích thước cao hơn có thể nắm bắt được nhiều mối quan hệ sắc thái hơn nhưng phải đánh đổi bằng hiệu quả tính toán.

While can use pretrained models e.g. Word2Vec to generate embeddings for ML models, LLMs commonly produce their own embeddings that are part of input layer & are updated during training. Advantage of optimizing embeddings as part of LLM training instead of using Word2Vec: embeddings are optimized to specific task & data at hand. Will implement such embedding layers later in this chap. (LLMs can also create contextualized output embeddings, as discussed in Chap. 3.)

Unfortunately, high-dimensional embeddings present a challenge for visualization because our sensory perception & common graphical representations are inherently limited to 3D or fewer, which is why Fig. 2.3 shows 2D embeddings in a 2D scatterplot. However, when working with LLMs, typically use embeddings with a much higher dimensionality. For both GPT-2 & GPT-3, embedding size (often referred to as dimensionality of model's hidden states) varies based on specific model variant & size. It is a tradeoff between performance & efficiency. Smallest GPT-2 models (117M & 125M parameters) use an embedding size of 768 dimensions to provide concrete examples. Largest GPt-3 model (175B parameters) uses an embedding size of 12288 dimensions.

Next, walk through required steps for preparing embeddings used by an LLM, which include splitting text into words, converting words into tokens, & turning tokens into embedding vectors.

- 2.2. Tokenizing text. Discuss how split input text into individual tokens, a required preprocessing step for creating embeddings for an LLM. These tokens are either individual words or special characters, including punctuation characters, as shown in Fig. 2.4: A view of text processing steps in context of an LLM. Here, split an input text into individual tokens, which are either words or special characters, e.g. punctuation characters.

  Text we will tokenize for LLM training is "The Verdict," a short story by Edith Wharton, which has been released into public domain & is thus permitted to be used for LLM training tasks. Text is available on Wikisource at https://en.wikisource.org/wiki/The_Verdict, & can copy & paste it into a text file `the-verdict.txt`.

  Alternatively, can find this `the-verdict.txt` file in this book's GitHub repository at [link], can download file with Python code:

```
import urllib.request
url = ("https://raw.githubusercontent.com/rasbt/"
       "LLMs-from-scratch/main/ch02/01_main-chapter-code/"
       "the-verdict.txt")
file_path = "the-verdict.txt"
urllib.request.urlretrieve(url, file_path)
```

  Can load `the-verdict.txt` file using Python's standard file reading utilities.

  Listing 2.1: Reading in a short story as text sample into Python.

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

- 2.3. Converting tokens into token IDs.
- 2.4. Adding special context tokens.
- 2.5. Byte pair encoding.
- 2.6. Data sampling with a sliding window.
- 2.7. Creating token embeddings.
- 2.8. Encoding word positions.

• 3. Coding attention mechanisms.

- 3.1. Problem with modeling long sequences.
- 3.2. Capturing data dependencies with attention mechanisms.
- 3.3. Attending to different parts of input with self-attention.
  * A simple self-attention mechanism without trainable weights.
  * Computing attention weights for all input tokens.

  - 3.4. Implementing self-attention with trainable weights.
    - Computing attention weights step by step.
    - Implementing a compact self-attention Python class.
  - 3.5. Hiding future words with causal attention.
    - Applying a causal attention mask.
    - Masking additional attention weights with dropout.
    - Implementing a compact causal attention class.
  - 3.6. Extending single-head attention to multi-head.
    - Stacking multiple single-head attention layers.
    - Implementing multi-head attention with weight splits.

- 4. Implementing a GPT model from scratch to generate text.
  - 4.1. Coding an LLM architecture.
  - 4.2. Normalizing activations with layer normalization.
  - 4.3. Implementing a feed forward network with GELU activations.
  - 4.4. Adding shortcut connections.
  - 4.5. Connecting attention & linear layers in a transformer block.
  - 4.6. Coding GPT model.
  - 4.7. Generating text.

- 5. Pretraining on unlabeled data.
  - 5.1. Evaluating generative text models.
    - Using GPT to generate text.
    - Calculating text generation loss.
    - Calculating training & validation set losses.
  - 5.2. Training an LLM.
  - 5.3. Decoding strategies to control randomness.
    - Temperature scaling.
    - Top-k sampling.
    - Modifying text generation function.
  - 5.4. Loading & saving model weights in PyTorch.
  - 5.5. Loading pretrained weights from OpenAI.

- 6. Fine-tuning for classification.
  - 6.1. Different categories of fine-tuning.
  - 6.2. Preparing dataset.
  - 6.3. Creating data loaders.
  - 6.4. Initializing a model with pretrained weights.
  - 6.5. Adding a classification head.
  - 6.6. Calculating classification loss & accuracy.
  - 6.7. Fine-tuning model on supervised data.
  - 6.8. Using LLM as a spam classifier.

- 7. Fine-tuning to follow instructions.
  - 7.1. Introduction to instruction fine-tuning.
  - 7.2. Preparing a dataset for supervised instruction fine-tuning.
  - 7.3. Organizing data into training batches.
  - 7.4. Creating data loaders for an instruction dataset.
  - 7.5. Loading a pretrained LLM.
  - 7.6. Fine-tuning LLM on instruction data.
  - 7.7. Extracting & saving responses.
  - 7.8. Evaluating fine-tuned LLM.

- 7.9. Conclusions.
    - What's next?
    - Staying up to data in a fast-moving field.
    - Final words.
- A. Introduction to PyTorch.
- B. References & further reading.
- C. Exercise solutions.
- D. Adding bells & whistles to training loop.
- E. Parameter-efficient fine-tuning with LoRA.

2 Miscellaneous

Tài liệu

[Ras24] Sebastian Raschka. *Build A Large Language Model (From Scratch)*. 1st edition. Manning Publishing, 2024, p. 343.