appendix B References and further reading

Chapter 1

Custom-built LLMs are able to outperform general-purpose LLMs as a team at Bloomberg showed via a version of GPT pretrained on finance data from scratch. The custom LLM outperformed ChatGPT on financial tasks while maintaining good performance on general LLM benchmarks:

- "BloombergGPT: A Large Language Model for Finance" (2023) by Wu et al., https://arxiv.org/abs/2303.17564

Existing LLMs can be adapted and fine-tuned to outperform general LLMs as well, which teams from Google Research and Google DeepMind showed in a medical context:

- "Towards Expert-Level Medical Question Answering with Large Language Models" (2023) by Singhal et al., https://arxiv.org/abs/2305.09617

The following paper proposed the original transformer architecture:

- "Attention Is All You Need" (2017) by Vaswani et al., https://arxiv.org/abs/1706.03762

On the original encoder-style transformer, called BERT, see

- "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" (2018) by Devlin et al., https://arxiv.org/abs/1810.04805

The paper describing the decoder-style GPT-3 model, which inspired modern LLMs and will be used as a template for implementing an LLM from scratch in this book, is

- "Language Models are Few-Shot Learners" (2020) by Brown et al., https://arxiv.org/abs/2005.14165

The following covers the original vision transformer for classifying images, which illustrates that transformer architectures are not only restricted to text inputs:

- "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" (2020) by Dosovitskiy et al., https://arxiv.org/abs/2010.11929

The following experimental (but less popular) LLM architectures serve as examples that not all LLMs need to be based on the transformer architecture:

- "RWKV: Reinventing RNNs for the Transformer Era" (2023) by Peng et al., https://arxiv.org/abs/2305.13048
- "Hyena Hierarchy: Towards Larger Convolutional Language Models" (2023) by Poli et al., https://arxiv.org/abs/2302.10866
- "Mamba: Linear-Time Sequence Modeling with Selective State Spaces" (2023) by Gu and Dao, https://arxiv.org/abs/2312.00752

Meta AI's model is a popular implementation of a GPT-like model that is openly available in contrast to GPT-3 and ChatGPT:

- "Llama 2: Open Foundation and Fine-Tuned Chat Models" (2023) by Touvron et al., https://arxiv.org/abs/2307.092881

For readers interested in additional details about the dataset references in section 1.5, this paper describes the publicly available *The Pile* dataset curated by Eleuther AI:

- "The Pile: An 800GB Dataset of Diverse Text for Language Modeling" (2020) by Gao et al., https://arxiv.org/abs/2101.00027

The following paper provides the reference for InstructGPT for fine-tuning GPT-3, which was mentioned in section 1.6 and will be discussed in more detail in chapter 7:

- "Training Language Models to Follow Instructions with Human Feedback" (2022) by Ouyang et al., https://arxiv.org/abs/2203.02155

Chapter 2

Readers who are interested in discussion and comparison of embedding spaces with latent spaces and the general notion of vector representations can find more information in the first chapter of my book:

- *Machine Learning Q and AI* (2023) by Sebastian Raschka, https://leanpub.com/machine-learning-q-and-ai

The following paper provides more in-depth discussions of how byte pair encoding is used as a tokenization method:

- "Neural Machine Translation of Rare Words with Subword Units" (2015) by Sennrich et al., https://arxiv.org/abs/1508.07909

The code for the byte pair encoding tokenizer used to train GPT-2 was open-sourced by OpenAI:

- https://github.com/openai/gpt-2/blob/master/src/encoder.py

OpenAI provides an interactive web UI to illustrate how the byte pair tokenizer in GPT models works:

- https://platform.openai.com/tokenizer

For readers interested in coding and training a BPE tokenizer from the ground up, Andrej Karpathy's GitHub repository `minbpe` offers a minimal and readable implementation:

- "A Minimal Implementation of a BPE Tokenizer," https://github.com/karpathy/minbpe

Readers who are interested in studying alternative tokenization schemes that are used by some other popular LLMs can find more information in the SentencePiece and WordPiece papers:

- "SentencePiece: A Simple and Language Independent Subword Tokenizer and Detokenizer for Neural Text Processing" (2018) by Kudo and Richardson, https://aclanthology.org/D18-2012/
- "Fast WordPiece Tokenization" (2020) by Song et al., https://arxiv.org/abs/2012.15524

Chapter 3

Readers interested in learning more about Bahdanau attention for RNN and language translation can find detailed insights in the following paper:

- "Neural Machine Translation by Jointly Learning to Align and Translate" (2014) by Bahdanau, Cho, and Bengio, https://arxiv.org/abs/1409.0473

The concept of self-attention as scaled dot-product attention was introduced in the original transformer paper:

- "Attention Is All You Need" (2017) by Vaswani et al., https://arxiv.org/abs/1706.03762

FlashAttention is a highly efficient implementation of a self-attention mechanism, which accelerates the computation process by optimizing memory access patterns. FlashAttention is mathematically the same as the standard self-attention mechanism but optimizes the computational process for efficiency:

- "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness" (2022) by Dao et al., https://arxiv.org/abs/2205.14135
- "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning" (2023) by Dao, https://arxiv.org/abs/2307.08691

PyTorch implements a function for self-attention and causal attention that supports FlashAttention for efficiency. This function is beta and subject to change:

- `scaled_dot_product_attention` documentation: https://mng.bz/NRJd

PyTorch also implements an efficient `MultiHeadAttention` class based on the `scaled_dot_product` function:

- `MultiHeadAttention` documentation: https://mng.bz/DdJV

Dropout is a regularization technique used in neural networks to prevent overfitting by randomly dropping units (along with their connections) from the neural network during training:

- "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" (2014) by Srivastava et al., https://jmlr.org/papers/v15/srivastava14a.html

While using the multi-head attention based on scaled-dot product attention remains the most common variant of self-attention in practice, authors have found that it's possible to also achieve good performance without the value weight matrix and projection layer:

- "Simplifying Transformer Blocks" (2023) by He and Hofmann, https://arxiv.org/abs/2311.01906

Chapter 4

The following paper introduces a technique that stabilizes the hidden state dynamics neural networks by normalizing the summed inputs to the neurons within a hidden layer, significantly reducing training time compared to previously published methods:

- "Layer Normalization" (2016) by Ba, Kiros, and Hinton, https://arxiv.org/abs/1607.06450

Post-LayerNorm, used in the original transformer model, applies layer normalization after the self-attention and feed forward networks. In contrast, Pre-LayerNorm, as adopted in models like GPT-2 and newer LLMs, applies layer normalization before these components, which can lead to more stable training dynamics and has been shown to improve performance in some cases, as discussed in the following papers:

- "On Layer Normalization in the Transformer Architecture" (2020) by Xiong et al., https://arxiv.org/abs/2002.04745
- "ResiDual: Transformer with Dual Residual Connections" (2023) by Tie et al., https://arxiv.org/abs/2304.14802

A popular variant of LayerNorm used in modern LLMs is RMSNorm due to its improved computing efficiency. This variant simplifies the normalization process by normalizing the inputs using only the root mean square of the inputs, without subtracting the mean before squaring. This means it does not center the data before computing the scale. RMSNorm is described in more detail in

- "Root Mean Square Layer Normalization" (2019) by Zhang and Sennrich, https://arxiv.org/abs/1910.07467

The Gaussian Error Linear Unit (GELU) activation function combines the properties of both the classic ReLU activation function and the normal distribution's cumulative distribution function to model layer outputs, allowing for stochastic regularization and nonlinearities in deep learning models:

- "Gaussian Error Linear Units (GELUs)" (2016) by Hendricks and Gimpel, https://arxiv.org/abs/1606.08415

The GPT-2 paper introduced a series of transformer-based LLMs with varying sizes—124 million, 355 million, 774 million, and 1.5 billion parameters:

- "Language Models Are Unsupervised Multitask Learners" (2019) by Radford et al., https://mng.bz/lMgo

OpenAI's GPT-3 uses fundamentally the same architecture as GPT-2, except that the largest version (175 billion) is 100x larger than the largest GPT-2 model and has been trained on much more data. Interested readers can refer to the official GPT-3 paper by OpenAI and the technical overview by Lambda Labs, which calculates that training GPT-3 on a single RTX 8000 consumer GPU would take 665 years:

- "Language Models are Few-Shot Learners" (2023) by Brown et al., https://arxiv.org/abs/2005.14165
- "OpenAI's GPT-3 Language Model: A Technical Overview," https://lambdalabs.com/blog/demystifying-gpt-3

NanoGPT is a code repository with a minimalist yet efficient implementation of a GPT-2 model, similar to the model implemented in this book. While the code in this book is different from nanoGPT, this repository inspired the reorganization of a large GPT Python parent class implementation into smaller submodules:

- "NanoGPT, a Repository for Training Medium-Sized GPTs, https://github.com/karpathy/nanoGPT

An informative blog post showing that most of the computation in LLMs is spent in the feed forward layers rather than attention layers when the context size is smaller than 32,000 tokens is:

- "In the Long (Context) Run" by Harm de Vries, https://www.harmdevries.com/post/context-length/

Chapter 5

For information on detailing the loss function and applying a log transformation to make it easier to handle for mathematical optimization, see my lecture video:

- L8.2 Logistic Regression Loss Function, https://www.youtube.com/watch?v=GxJe0DZvydM

The following lecture and code example by the author explain how PyTorch's cross-entropy functions works under the hood:

- L8.7.1 OneHot Encoding and Multi-category Cross Entropy, https://www.youtube.com/watch?v=4n71-tZ94yk
- Understanding Onehot Encoding and Cross Entropy in PyTorch, https://mng.bz/o05v

The following two papers detail the dataset, hyperparameter, and architecture details used for pretraining LLMs:

- "Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling" (2023) by Biderman et al., https://arxiv.org/abs/2304.01373
- "OLMo: Accelerating the Science of Language Models" (2024) by Groeneveld et al., https://arxiv.org/abs/2402.00838

The following supplementary code available for this book contains instructions for preparing 60,000 public domain books from Project Gutenberg for LLM training:

- Pretraining GPT on the Project Gutenberg Dataset, https://mng.bz/Bdw2

Chapter 5 discusses the pretraining of LLMs, and appendix D covers more advanced training functions, such as linear warmup and cosine annealing. The following paper finds that similar techniques can be successfully applied to continue pretraining already pretrained LLMs, along with additional tips and insights:

- "Simple and Scalable Strategies to Continually Pre-train Large Language Models" (2024) by Ibrahim et al., https://arxiv.org/abs/2403.08763

BloombergGPT is an example of a domain-specific LLM created by training on both general and domain-specific text corpora, specifically in the field of finance:

- "BloombergGPT: A Large Language Model for Finance" (2023) by Wu et al., https://arxiv.org/abs/2303.17564

GaLore is a recent research project that aims to make LLM pretraining more efficient. The required code change boils down to just replacing PyTorch's `AdamW` optimizer in the training function with the `GaLoreAdamW` optimizer provided by the `galore-torch` Python package:

- "GaLore: Memory-Efficient LLM Training by Gradient Low-Rank Projection" (2024) by Zhao et al., https://arxiv.org/abs/2403.03507
- GaLore code repository, https://github.com/jiaweizzhao/GaLore

The following papers and resources share openly available, large-scale pretraining datasets for LLMs that consist of hundreds of gigabytes to terabytes of text data:

- "Dolma: An Open Corpus of Three Trillion Tokens for LLM Pretraining Research" (2024) by Soldaini et al., https://arxiv.org/abs/2402.00159

- "The Pile: An 800GB Dataset of Diverse Text for Language Modeling" (2020) by Gao et al., https://arxiv.org/abs/2101.00027
- "The RefinedWeb Dataset for Falcon LLM: Outperforming Curated Corpora with Web Data, and Web Data Only," (2023) by Penedo et al., https://arxiv.org/abs/2306.01116
- "RedPajama," by Together AI, https://mng.bz/d6nw
- The FineWeb Dataset, which includes more than 15 trillion tokens of cleaned and deduplicated English web data sourced from CommonCrawl, https://mng.bz/rVzy

The paper that originally introduced top-k sampling is

- "Hierarchical Neural Story Generation" (2018) by Fan et al., https://arxiv.org/abs/1805.04833

An alternative to top-k sampling is top-p sampling (not covered in chapter 5), which selects from the smallest set of top tokens whose cumulative probability exceeds a threshold p, while top-k sampling picks from the top k tokens by probability:

- Top-p sampling, https://en.wikipedia.org/wiki/Top-p_sampling

Beam search (not covered in chapter 5) is an alternative decoding algorithm that generates output sequences by keeping only the top-scoring partial sequences at each step to balance efficiency and quality:

- "Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models" (2016) by Vijayakumar et al., https://arxiv.org/abs/1610.02424

Chapter 6

Additional resources that discuss the different types of fine-tuning are

- "Using and Finetuning Pretrained Transformers," https://mng.bz/VxJG
- "Finetuning Large Language Models," https://mng.bz/x28X

Additional experiments, including a comparison of fine-tuning the first output token versus the last output token, can be found in the supplementary code material on GitHub:

- Additional spam classification experiments, https://mng.bz/AdJx

For a binary classification task, such as spam classification, it is technically possible to use only a single output node instead of two output nodes, as I discuss in the following article:

- "Losses Learned—Optimizing Negative Log-Likelihood and Cross-Entropy in PyTorch," https://mng.bz/ZEJA

You can find additional experiments on fine-tuning different layers of an LLM in the following article, which shows that fine-tuning the last transformer block, in addition to the output layer, improves the predictive performance substantially:

- "Finetuning Large Language Models," https://mng.bz/RZJv

Readers can find additional resources and information for dealing with imbalanced classification datasets in the imbalanced-learn documentation:

- "Imbalanced-Learn User Guide," https://mng.bz/2KNa

For readers interested in classifying spam emails rather than spam text messages, the following resource provides a large email spam classification dataset in a convenient CSV format similar to the dataset format used in chapter 6:

- Email Spam Classification Dataset, https://mng.bz/1GEq

GPT-2 is a model based on the decoder module of the transformer architecture, and its primary purpose is to generate new text. As an alternative, encoder-based models such as BERT and RoBERTa can be effective for classification tasks:

- "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" (2018) by Devlin et al., https://arxiv.org/abs/1810.04805
- "RoBERTa: A Robustly Optimized BERT Pretraining Approach" (2019) by Liu et al., https://arxiv.org/abs/1907.11692
- "Additional Experiments Classifying the Sentiment of 50k IMDB Movie Reviews," https://mng.bz/PZJR

Recent papers are showing that the classification performance can be further improved by removing the causal mask during classification fine-tuning alongside other modifications:

- "Label Supervised LLaMA Finetuning" (2023) by Li et al., https://arxiv.org/abs/2310.01208
- "LLM2Vec: Large Language Models Are Secretly Powerful Text Encoders" (2024) by BehnamGhader et al., https://arxiv.org/abs/2404.05961

Chapter 7

The Alpaca dataset for instruction fine-tuning contains 52,000 instruction–response pairs and is one of the first and most popular publicly available datasets for instruction fine-tuning:

- "Stanford Alpaca: An Instruction-Following Llama Model," https://github.com/tatsu-lab/stanford_alpaca

Additional publicly accessible datasets suitable for instruction fine-tuning include

- LIMA, https://huggingface.co/datasets/GAIR/lima
  - For more information, see "LIMA: Less Is More for Alignment," Zhou et al., https://arxiv.org/abs/2305.11206

- UltraChat, https://huggingface.co/datasets/openchat/ultrachat-sharegpt
  - A large-scale dataset consisting of 805,000 instruction–response pairs; for more information, see "Enhancing Chat Language Models by Scaling High-quality Instructional Conversations," by Ding et al., https://arxiv.org/abs/2305.14233
- Alpaca GPT4, https://mng.bz/Aa0p
  - An Alpaca-like dataset with 52,000 instruction–response pairs generated with GPT-4 instead of GPT-3.5

Phi-3 is a 3.8-billion-parameter model with an instruction-fine-tuned variant that is reported to be comparable to much larger proprietary models, such as GPT-3.5:

- "Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone" (2024) by Abdin et al., https://arxiv.org/abs/2404.14219

Researchers propose a synthetic instruction data generation method that generates 300,000 high-quality instruction-response pairs from an instruction fine-tuned Llama-3 model. A pretrained Llama 3 base model fine-tuned on these instruction examples performs comparably to the original instruction fine-tuned Llama-3 model:

- "Magpie: Alignment Data Synthesis from Scratch by Prompting Aligned LLMs with Nothing" (2024) by Xu et al., https://arxiv.org/abs/2406.08464

Research has shown that not masking the instructions and inputs in instruction fine-tuning effectively improves performance on various NLP tasks and open-ended generation benchmarks, particularly when trained on datasets with lengthy instructions and brief outputs or when using a small number of training examples:

- "Instruction Tuning with Loss Over Instructions" (2024) by Shi, https://arxiv.org/abs/2405.14394

Prometheus and PHUDGE are openly available LLMs that match GPT-4 in evaluating long-form responses with customizable criteria. We don't use these because at the time of this writing, they are not supported by Ollama and thus cannot be executed efficiently on a laptop:

- "Prometheus: Inducing Finegrained Evaluation Capability in Language Models" (2023) by Kim et al., https://arxiv.org/abs/2310.08491
- "PHUDGE: Phi-3 as Scalable Judge" (2024) by Deshwal and Chawla, "https://arxiv.org/abs/2405.08029
- "Prometheus 2: An Open Source Language Model Specialized in Evaluating Other Language Models" (2024), by Kim et al., https://arxiv.org/abs/2405.01535

The results in the following report support the view that large language models primarily acquire factual knowledge during pretraining and that fine-tuning mainly enhances their efficiency in using this knowledge. Furthermore, this study explores

how fine-tuning large language models with new factual information affects their ability to use preexisting knowledge, revealing that models learn new facts more slowly and their introduction during fine-tuning increases the model's tendency to generate incorrect information:

- "Does Fine-Tuning LLMs on New Knowledge Encourage Hallucinations?" (2024) by Gekhman, https://arxiv.org/abs/2405.05904

Preference fine-tuning is an optional step after instruction fine-tuning to align the LLM more closely with human preferences. The following articles by the author provide more information about this process:

- "LLM Training: RLHF and Its Alternatives," https://mng.bz/ZVPm
- "Tips for LLM Pretraining and Evaluating Reward Models," https://mng.bz/RNXj

Appendix A

While appendix A should be sufficient to get you up to speed, if you are looking for more comprehensive introductions to deep learning, I recommend the following books:

- *Machine Learning with PyTorch and Scikit-Learn* (2022) by Sebastian Raschka, Hayden Liu, and Vahid Mirjalili. ISBN 978-1801819312
- *Deep Learning with PyTorch* (2021) by Eli Stevens, Luca Antiga, and Thomas Viehmann. ISBN 978-1617295263

For a more thorough introduction to the concepts of tensors, readers can find a 15-minute video tutorial that I recorded:

- "Lecture 4.1: Tensors in Deep Learning," https://www.youtube.com/watch?v=JXfDlgrfOBY

If you want to learn more about model evaluation in machine learning, I recommend my article

- "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning" (2018) by Sebastian Raschka, https://arxiv.org/abs/1811.12808

For readers who are interested in a refresher or gentle introduction to calculus, I've written a chapter on calculus that is freely available on my website:

- "Introduction to Calculus," by Sebastian Raschka, https://mng.bz/WEyW

Why does PyTorch not call `optimizer.zero_grad()` automatically for us in the background? In some instances, it may be desirable to accumulate the gradients, and PyTorch will leave this as an option for us. If you want to learn more about gradient accumulation, please see the following article:

- "Finetuning Large Language Models on a Single GPU Using Gradient Accumulation" by Sebastian Raschka, https://mng.bz/8wPD

This appendix covers DDP, which is a popular approach for training deep learning models across multiple GPUs. For more advanced use cases where a single model doesn't fit onto the GPU, you may also consider PyTorch's Fully Sharded Data Parallel (FSDP) method, which performs distributed data parallelism and distributes large layers across different GPUs. For more information, see this overview with further links to the API documentation:

- "Introducing PyTorch Fully Sharded Data Parallel (FSDP) API," https://mng.bz/EZJR