Specializing Large Language Models in Finance

Julius L.H. de Clercq
Business Data Science
Vrije Universiteit Amsterdam, Universiteit van Amsterdam, Erasmus University Rotterdam
Gustav Mahlerplein 117, 1082 MS Amsterdam
j.l.h.de.clercq@businessdatascience.nl

Supervised by: dr. Svetlana Borovkova
(Formerly) Vrije Universiteit Amsterdam, Bloomberg

Abstract

While large language models (LLMs) demonstrate impressive general capabilities, their application to specialized domains like finance remains challenging due to shallow domain understanding and hallucination risks. This thesis investigates domain adaptation strategies for financial LLMs through three research questions: (1) the effectiveness of continued pretraining on financial texts, (2) the impact of training sequence length on adaptation efficiency, and (3) the benefits of instruction tuning for domain-adapted models. Llama 3.1 8B is selected as the base model and continued pretraining is performed on SEC filings with both short (2,048 tokens) and long (8,192 tokens) sequences, followed by instruction tuning experiments. Evaluation on four financial NLP benchmarks reveals task-dependent benefits: domain adaptation improves named entity recognition, while offering little to no improvement on classification tasks. Short-sequence training generally outperforms long-sequence training, contradicting prior empirical evidence. Instruction tuning improves base model performance on named entity recognition but shows no synergy with domain adaptation and, in the short-sequence variant, can markedly degrade named entity recognition, suggesting interference between adaptation strategies. These findings indicate that effective financial LLM specialization requires careful alignment between adaptation approaches and intended applications, with domain adaptation being most valuable for information extraction rather than high-level reasoning tasks.

The code for this project can be found on the project's GitHub repository, and the models are collected on Hugging Face.

I wish to express my gratitude to: my supervisor, dr. Svetlana Borovkova, for her expertise, network, and opportunity to work on this project and the valuable educative experience it entailed; Probability & Partners B.V., for their support and belief in this project throughout, with special gratitude going out to Erik Kooistra, for his firm-side supervision and dialogue; Chris Kantos, for lending his technical expertise; prof.dr. Roland Mees, for his keen advice on the thesis process and structure; the SURF Helpdesk and the team High Performance Machine Learning, for their quick responses, unexpectedly generous helpfulness and depth of knowledge; the faculty of Business Data Science, for the incredible quality of education; prof.dr. Sebastian Gryglewicz, prof.dr. Bernd Heidergott, and prof.dr. Paul Smeets for their help in directing my career and academic focus; my family, for the support that I could never have written this thesis without.

Nomenclature

Adam	Adaptive Moment Estimation
AdamW	Adaptive Moment Estimation with Weight-Decay
AGI	Artificial General Intelligence
API	Application Programming Interface
AI	Artificial Intelligence
Bash	Bourne Again Shell
BMP	Basic Multilingual Plane
BOS	Beginning-of-Sequence
CPU	Central Processing Unit
DL	Deep Learning
DRAM	Dynamic Random Access Memory
EOS	End-of-Sequence
FLARE	Financial Language Model Research
FLOPs	Floating Point Operations
FNN	Feed-Forward Neural Network
FM	Foundation Model
GAAP	Generally Accepted Accounting Principles
GLU	Gated Linear Unit
GPT	Generative Pretrained Transformer
GPU	Graphics Processing Unit
HF	Hugging Face
HPC	High-Performance Computing
HTML	Hyper-Text Markup Language
IPO	Initial Public Offering
iXBRL	Inline Extensible Business Reporting Language
JSON	JavaScript Object Notation
JSONL	JavaScript Object Notation Lines
I/O	Input/Output
LLM	Large Language Model
LM	Language Model(ing)
LN	Layer Normalization
LoRA	Low-Rank Adaptation
MLP	Multi-Layer Perceptron
MSL	Maximum Sequence Length
NLP	Natural Language Processing
OOM	Out-Of-Memory
PEFT	Parameter-Efficient Finetuning
QA	Question-Answer
QLoRA	Quantized Low-Rank Adaptation
ReLU	Rectified Linear Unit
RMS	Root Mean Square
RoPE	Rotary Position Embedding
SEC	Securities and Exchange Commission
SiLU	Sigmoid Linear Unit
SFTP	Secure File Transfer Protocol
SPE	Sinusoidal Positional Encoding
SRAM	Static Random Access Memory
SSH	Secured Shell
TAR	Tape Archive
UTF	Unicode Transformation Format
VGP	Vanishing Gradient Problem
VRAM	Video Random Access Memory
XAI	Explainable Artificial Intelligence
XBRL	Extensible Business Reporting Language
XML	Extensible Markup Language

Metric prefixes are used to indicate thousands (k, kilo), millions (M, mega), billions (G, giga) and trillions (T, tera).

Contents

1 Introduction

In early 2023, large language models (LLMs) took the world by storm when ChatGPT, a chatbot based on OpenAI's generative pretrained transformer (GPT) model GPT-3, became the fastest adopted consumer application in history, reaching 100 million users in less than two months after launch (K. Hu, 2023). The revolutionary capabilities of LLMs quickly became apparent, a trend reflected in the stock market: NVIDIA, the world's leading producer of graphics processing units (GPUs) that power LLM training and inference, saw its stock price nearly double within three months of ChatGPT's release and increase about tenfold by the summer of 2025, becoming the world's most capitalized company and the first to surpass a valuation of $4 trillion (Sor, 2025).

LLMs excel at processing vast amounts of textual information, having been trained on internet-scale datasets comprising billions of documents. This unprecedented scale enables them to develop sophisticated linguistic capabilities that surpass previous natural language processing (NLP) approaches. These general capabilities include summarizing complex documents into concise overviews, extracting key information from unstructured text, answering questions based on provided context, generating coherent and contextually appropriate text, and translating between languages.

However, general-purpose, or *generalist* LLMs, trained primarily on web content, exhibit significant limitations when confronted with domain-specific challenges. Their training on diverse internet data results in broad but shallow understanding, leading to a phenomenon known as *hallucination*: the generation of plausible-sounding but factually incorrect or nonsensical outputs. In specialized domains, these models may confidently produce responses that mix accurate information with fabricated details, misapply technical concepts, or fail to recognize the subtle but essential distinctions that experts in the field would immediately identify. This gap between general linguistic competence and domain expertise poses serious challenges for deploying LLMs in professional contexts where accuracy and reliability are paramount (Ling et al., 2023).

The recognition of these limitations has sparked growing demand for *specialist* LLMs tailored to specific domains and industries. Organizations across sectors–from healthcare, law, and finance to engineering and scientific research–seek models that combine the powerful language understanding of generalist LLMs with deep, accurate knowledge of their particular field. This thesis examines the challenge of creating specialist LLMs through the lens of finance: a domain that presents unique challenges due to its technical complexity, rapidly evolving nature, precise terminology, and the high stakes associated with financial decision-making. Finance serves as a highly suitable case study for understanding both the limitations of generalist models and the potential of specialization techniques.

The primary approach to adapting LLMs to specific domains involves *continued pretraining*: rather than training new models from scratch–an approach that is notoriously expensive–researchers can build upon existing generalist LLMs by continuing their training on domain-specific corpora. This technique allows models to acquire specialized knowledge while retaining their general language understanding capabilities, effectively creating systems that bridge the gap between broad linguistic competence and deep domain expertise.

Recent work on continued pretraining for financial domains remains limited in scope and scale. The most prominent effort for a financial LLM to date, BloombergGPT (Wu et al., 2023), demonstrated impressive performance on financial tasks but was trained from scratch using 1.3M GPU hours at an estimated cost of around $3 million (H. Yang et al., 2023), making their approach inaccessible to most researchers and organizations, raising the question of whether domain adaptation may have been performed more efficiently by continually pretraining a base model. Besides, as a proprietary

model, BloombergGPT's architecture, training data, and detailed performance metrics remain largely opaque to the research community, limiting the possibility to build upon their findings or reproduce their results. FinPythia (Y. Xie et al., 2024) demonstrated that continued pretraining of the Pythia base model (S. Biderman et al., 2023) can achieve strong performance on financial NLP benchmarks, at a much lower cost than BloombergGPT. Their work provides valuable evidence for the viability of domain adaptation and establishes important baselines for the field. However, several questions remain unanswered. First, FinPythia builds upon the Pythia architecture, which, while open and well-documented, predates recent architectural innovations that have substantially improved model efficiency and capability. Modern foundation models like Llama 3.1 (Meta, 2024b) incorporate advances such as SwiGLU multilayer perceptrons, Grouped-Query Attention, and improved tokenization strategies that may fundamentally change the dynamics of domain adaptation–both in terms of the baseline capabilities that need less adaptation and the efficiency with which specialized knowledge can be acquired. Second, while FinPythia demonstrates strong benchmark performance, their training corpus combines both financial news and filings data, which raises the question of the added value of the constituent data sources, and the performance that can be achieved through usage of individual sources. The latter is especially relevant due to the increased difficulty and costs of procurement of combined data, and the added complexity it brings to research design and preprocessing. These gaps motivate the first research question (RQ):

RQ1: What improvement in performance on financial NLP benchmarks can be achieved through domain adaptation by continued pretraining on SEC filings?

This research addresses these gaps by adapting Llama 3.1 8B using a focused corpus of SEC (i.e., the United States' Security and Exchange Commission) filings, enabling direct comparison with existing benchmarks while providing a fully reproducible approach using public data.

Beyond the question of whether continued pretraining improves performance, the efficiency of the adaptation process itself merits investigation. Due to the quadratic computational complexity of the attention mechanism with respect to sequence length, the context window size used during training significantly impacts computational requirements (Jin et al., 2023). This consideration is particularly relevant for SEC filings as they can be excessively large; this is especially true for annual reports (10-K filings), which tend to comprise millions of characters.

Despite its importance for training efficiency, research on the effect of context window length during continued pretraining on model performance remains sparse. W. Xiong et al. (2023) found that continued pretraining of Llama 2 (Touvron, Martin, et al., 2023) using longer context windows improves performance on long-context tasks while maintaining or slightly improving short-context performance. However, this remains an isolated study, and with rapid advances in LLM architectures and the unique characteristics of financial text, the relationship between context window size and adaptation effectiveness remains largely unexplored. Understanding this relationship is crucial for practitioners seeking to balance computational costs with model performance when adapting LLMs to financial domains. This motivates the second research question:

RQ2: What difference in performance can be found through domain adaptation using examples with long (8192 tokens) versus short (2048 tokens) sequence lengths?

Recent advancements have demonstrated the effectiveness of instruction tuning, where an LLM is finetuned for following instructions, for improving zero-shot and few-shot task performance (Wei, Bosma, et al., 2022). While continued pretraining can adapt models to domain-specific knowledge and terminology, it primarily optimizes for next-token prediction on domain-specific texts rather than

the ability to follow instructions and complete specific tasks. This distinction is crucial: a model that has undergone continued pretraining on financial corpora may understand financial language and concepts but may not effectively apply this knowledge when prompted to perform specific tasks (Uppaal, 2024).

The development of instruction-tuned LLMs for finance is an emerging area. H. Yang et al. (2023) introduced FinGPT, for which they developed a financial instruction tuning dataset, being one of the few existing instruction tuning datasets dedicated to the financial domain (Lee, 2024; The FinAI Team, 2024). However, this existing work applies instruction tuning directly to general base models, bypassing the potential benefits of first establishing domain expertise through continued pretraining. While W. Xiong et al. (2023) explored instruction tuning after continued pretraining, their work focused primarily on context length effects rather than systematically evaluating the benefits of instruction tuning for domain-specialized capabilities, besides not incorporating recent innovations in LLM design. Jindal et al. (2024) assess the interplay between continued pretraining and instruction tuning, which yields a suggested approach, but their research is not specific to the financial domain.

The key empirical gap lies in quantifying how instruction tuning enhances specific financial capabilities beyond what continued pretraining alone provides. While domain adaptation improves language modeling metrics on financial text, the practical value for end users depends on task performance: can the model accurately extract financial metrics, identify regulatory compliance issues, summarize investment risks, or answer complex queries about financial documents? These capabilities require not just domain knowledge but the ability to apply that knowledge systematically in response to varied prompts. Investigating whether instruction tuning can unlock these practical capabilities in domain-adapted models motivates the third and final research question:

RQ3: What improvement in financial NLP capabilities can be achieved through instruction tuning of domain-adapted LLMs?

To investigate these research questions, this thesis employs a three-stage methodology: domain adaptation through continued pretraining, instruction tuning, and systematic evaluation. For domain adaptation, Llama 3.1 8B is selected as the base model, balancing state-of-the-art performance with computational feasibility for academic research through its moderate model size. The training corpus consists of SEC filings, which provide a comprehensive, publicly available, and professionally authored and archived source of financial text. The data underwent extensive preprocessing to filter noise and machine-readable elements detrimental to domain adaptation (e.g., HTML tags, UU-encodings), and whitelisting to only include the form-types deemed most suitable for domain adaptation. Subsequently, the data was tokenized into two distinct datasets: one with 2,048-token sequences and another with 8,192-token sequences, enabling investigation of context length effects (RQ2). Both datasets were shuffled to decorrelate sequential dependencies inherent in the filings, such as temporal correlation. Continued pretraining was performed on NVIDIA H100 GPUs for approximately 110 hours per configuration, yielding both short-context and long-context domain-adapted models.

The instruction tuning phase aimed to enhance the task-following capabilities of the domain-adapted models. The general-purpose Alpaca dataset for broad instruction-following abilities was combined with TAT-QA: a specialized dataset for financial question answering and numerical reasoning. Instruction tuning, using QLoRA for parameter-efficient fine-tuning, was performed on three model variants: the vanilla Llama 3.1 8B base model and both domain-adapted versions. This approach enables direct comparison of instruction tuning effectiveness across models with and without prior domain adaptation.

For evaluation, the methodology established by Y. Xie et al. (2024) was adopted to ensure direct comparability with FinPythia and BloombergGPT, which represent leading benchmarks for financial LLMs. This evaluation method involves using four financial NLP benchmarks of the FLARE/PIXIU framework (Q. Xie et al., 2023), that assess diverse capabilities from sentiment analysis to named entity recognition. All models–the base model, the two domain adapted models, and the three instruction tuned models–were evaluated using five-shot prompting, providing sufficient examples while testing the models' ability to generalize beyond their training data. This evaluation strategy enables quantification of performance improvements from both continued pretraining and instruction tuning across multiple financial NLP tasks. Figure 1 provides a graphical illustration of the methodology.

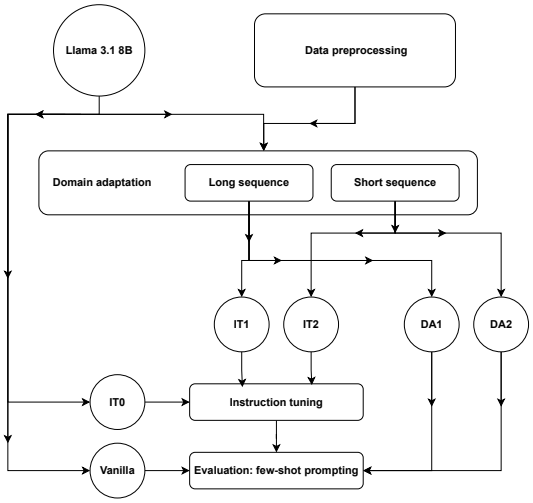


Figure 1

Flowchart of the research methodology. Rectangles with softened corners represent methodological steps, and circles represent models. Five models were evaluated: the base Llama 3.1 8B ("Vanilla"); IT0, the instruction-tuned base model without domain adaptation; DA1 and DA2, the long- and short-sequence domain-adapted models without instruction tuning; and IT1 and IT2, the long- and short-sequence domain-adapted models with instruction tuning.

This research reveals nuanced patterns in financial LLM specialization, demonstrating that the benefits of domain adaptation are highly dependent on task characteristics. While models trained on SEC filings show marked improvements in named entity recognition (NER)–with the strongest NER gains coming from instruction tuning the base model–they exhibit little to no gains in sentiment classification or similar high-level reasoning tasks, aside from small but consistent improvements on the FPB task from short-sequence adaptation (DA2) and its instruction-tuned variant (IT2). On the Headlines task, all assessed models tie and exceed FinPythia but show no improvement across domain-adaptation or instruction-tuning variants. Surprisingly, and contrary to prior empirical evidence, shorter training sequences (2,048 tokens) consistently outperformed longer sequences (8,192 tokens), with the sole exception of IT2 on NER. This advantage did not stem from computational efficiency, which was equal across settings due to FlashAttention, but likely from optimization dynamics on short-context benchmarks. Given that the evaluation benchmarks primarily test short-context tasks, potential benefits of long-sequence training for extensive regulatory documents remain untested. Instruction tuning presents a complex picture: while it dramatically improves NER capabilities when

applied to the base model, it shows no synergistic benefits when combined with domain adaptation and, in the short-sequence variant (IT2), can markedly degrade NER while offering only modest gains on some classification tasks. The study also uncovers significant preprocessing challenges inherent to SEC filings, where the complex mixing of human-readable text with machine-readable markup (XML/HTML/XBRL) introduces substantial noise that likely impacted model performance. Despite significant performance gaps compared to resource-intensive models like BloombergGPT (trained with >1M GPU hours) and FinPythia, versus 110 in this study, the findings provide practical guidance: organizations should prioritize domain adaptation for specific information extraction use cases while carefully considering the unexpected dynamics between different adaptation techniques. These results underscore that specialized financial LLMs require sophisticated preprocessing pipelines, careful consideration of training strategies, computational resources, and alignment with intended applications.

The research in this paper is presented as follows. The theory behind contemporary LLMs is discussed in the next section; for readers not sufficiently familiar with deep learning, an introduction is provided in Appendix A that should give sufficient background to the covered concepts. The data used for this research is discussed in Section 3, though the methodology of data selection, gathering and preprocessing are discussed along with the rest of this paper's methodology in Section 4. Subsequently, the results are presented in Section 5 and discussed in Section 6. Finally, the conclusions of this paper are presented in 7.

2 Theoretical Framework

Understanding fundamental concepts in deep learning (DL) is essential for grasping the theory behind large language models (LLMs). As this background cannot be safely assumed for all readers, while part of the audience is undoubtedly well-versed in this field, an introduction to required DL concepts is provided in Appendix A.

This section starts with discussing the fundamental challenge of natural language processing (NLP), which is to extract information from natural language or text. Subsequently, the transformer and transformer-based architectures are discussed by their fundamental components. Lastly, domain adaptation of LLMs are discussed, followed by the evaluation of financial NLP capabilities of LLMs.

2.1 Natural language processing (NLP)

The goal of NLP is to extract information from (natural) language, or simply text in short. The universal approximation theorem (see Appendix A) implies that any pattern in numerical data may be approximated using a multilayer perceptron (MLP). This means that if we are able to express language numerically, this power may be harnessed to model any pattern hidden in that language. This raises the question of what types of information this numerical representation should be able to capture. A helpful dichotomy in this matter, is that of intrinsic, *lexical* information versus extrinsic, *contextual* information. Lexical information is the meaning of a word as you would find it in a dictionary, independent of context. E.g., the word *"nephew"* denotes a familiar relation, masculinity and relative youth to an aunt or uncle. Contextual information, being context-dependent, has a number of aspects, the most notable being:

- *Semantics*: the meaning of a word that is dependent on its context. For instance, when we consider the meaning of the word "*lady*" in the sentences "*The nice young lady walking her dog*" and "*The wicked lady flying on a broom*", we can easily recognize that this word has a very different meaning due to the different contexts.

- *Syntax*: what sequence of words and punctuation is accepted as valid. E.g., "*On a broom flying the lady*" is not valid syntax in the English language, while another language may actually recognize this as the correct order of words.

So how can these types of information be expressed numerically? Lexical information can be captured by word-vector representations, a.k.a. *embeddings*. Each word is expressed as a vector[1], with each dimension being able to express some aspect of meaning. In fact, even more aspects of meaning can be encoded in these vectors by up- and down-projecting the embeddings in MLP blocks, as is discussed in Section 2.2.5. An oversimplification was made here by referring to words, as it is also possible to assign embeddings to other levels of granularity in language, such as the character level, where an embedding is assigned to each character (e.g., "b", "h", "5", "!"), the subword level (e.g., "pro", "b", "ability"), or even on the sentence level. The units that the embeddings are assigned to are known as *tokens*. The full set of tokens recognized by a language model (LM) is referred to as the LM's *vocabulary*. In

However, if we assign one fixed, static embedding to each token, it would not be possible for a LM to update its representations depending on context. To follow the earlier example: the embedding for "*lady*" would be the same regardless of whether she was walking her dog or was flying on a broom. To allow for incorporation of contextual information, dynamic embeddings are needed. This dynamism is achieved in modern LLMs through the *self-attention* mechanism, which is discussed in Section 2.2.2.

2.2 Transformer-based architecture

Though already an active field of research since at least the 1980s, the seminal paper that caused NLP to leap forward was Vaswani et al. (2017), who introduced the transformer architecture. The transformer, originally designed to translate English to French and German and vice versa, introduced several innovations such as the dot-product self-attention mechanism (see Section 2.2.2) and sinusoidal positional encodings (see Section 2.2.4). But arguably the biggest advantage of the transformer architecture over the then-dominant Recurrent Neural Network (RNN) architectures in NLP, is that RNNs are inherently sequential due to their recurrent nature. The *recurrence* in RNNs refers to the property of the model that the hidden state that was computed for the previous element in the sequence (e.g., the time-step in time series or token in tokenized natural language) is used as input for the hidden state of the current element in the sequence (Goodfellow et al., 2016b); so the previous step must be processed before the next step can be, entailing sequential computation. Because recurrence makes the model inherently sequential, it is impossible to predict the elements of any series or sequence in parallel. This is a fundamental limitation which makes RNNs ill-suited for GPU-based high-performance computing (HPC), which employs highly parallelized computation.

The transformer architecture does not suffer this limitation. Instead, when training a transformer, all elements in the sequence are predicted in parallel. This enables the transformer to be much more efficient in terms of GPU utilization than RNNs, and to be trained much faster as a consequence. Figure 2 shows a schematic representation of the original transformer. All LLMs are based on this architecture, so understanding the fundamental components of the original transformer–that can be clearly identified from this figure–translates to understanding modern LLM design and research. These fun-

[1]This idea was first introduced by Rumelhart et al. (1986), although these were sparse embeddings, with dimensions being binary variables denoting the presence of a word or not, which is why the embeddings were not able to represent detailed or nuanced meaning. Dense embeddings were introduced much later, and popularized by Mikolov et al. (2013). This thirty year-lapse of development needs to be considered in the context of computational power required to make use of dense embeddings; as congruent with Moore's law, computing power has increased massively over those three decades.

damental components are: the embedding space; the MLP block, denoted in the figure as *"Feed Forward"*; the (multi-head) (self-)attention block; the positional encodings; the LM head, which is the section of the model connecting the output of the final transformer block to the output probabilities; and the (layer) normalization block, denoted in the figure as *"Add & Norm"*.

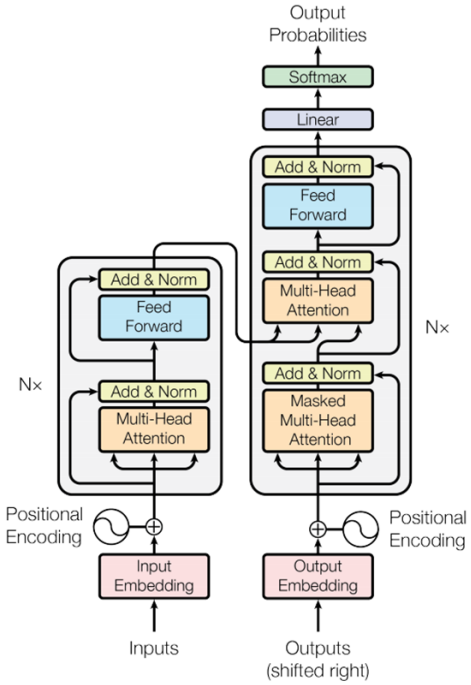


Figure 2

Schematic representation of the original transformer, taken from Vaswani et al. (2017, Figure 1). The left half known as the encoder, and the right half as the decoder. The original transformer was made up of six such blocks, meaning that in this figure $N = 6$.

Each of these components is discussed in a dedicated section below. Prior to that, a first clarification of the figure is that it can be split in a left-half, which is known as the *encoder*, and a right-half, known as the *decoder*. Though the original transformer combines them in an encoder–decoder paradigm, transformer-based architectures are typically based on either one. Most notably, the encoder spawned BERT (Bidirectional Encoder Representations from Transformers; Devlin et al. (2018)) and RoBERTa (Robustly Optimized BERT; Y. Liu (2019)), and the decoder spawned the GPT (Generative Pretrained Transformer) and Llama families of models, among many others. A vital distinction is that the decoder is *generative*, as it performs *causal* or *autoregressive language modeling*, meaning it predicts the next element in the sequence from the preceding ones; this makes the decoder *unidirectional*. The encoder is *bidirectional*, because to predict an element in the sequence it can use both preceding and succeeding elements; this precludes generative capabilities, but does allow learning of richer representations in theory. Contemporary research on LLMs is dominated by decoder-based models, since their generative capacity enables a wide range of use cases. Following this current paradigm, these decoder-based models will be in focus in this research.

Secondly, the grayed areas in Figure 2 spanning multiple attention, MLP, and normalization blocks, are known as *transformer blocks*. The precise make up of these vary from LLM to LLM, but each

LLM does contain a number of such transformer blocks, denoted by N in Figure 2, which is set to six for the original transformer.

2.2.1 Tokenization and embeddings

As introduced in Section 2.1, the input sequence to a transformer is split into *tokens*, each of which has its dedicated vector-representation in the embedding space. The *embedding matrix*, containing the vector-representations of length d of all tokens in the vocabulary $\mathcal{V}$, is denoted as $\boldsymbol{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$. The embedding vectors are learned when the LM is (pre)trained.

This tokenization process requires a *tokenizer*, which is a dedicated model that is trained prior to, and separately from, the LLM. The goal of tokenization is to limit the vocabulary size, while maintaining reasonable representational efficiency of the tokens. This balance is key in the training process of tokenizers. There are various types of tokenizers that have different training strategies, but more detail is omitted here as tokenization strategies fall outside the scope of this research. Figure 3 shows an example sentence that is tokenized twice: sequence 1 with the tokenizer for Llama 2 and sequence 2 with the tokenizer of Llama 3. This figure gives a general illustration of how tokenization works, and that there can be quite some variation in tokenizer functionality. Improvements to the tokenizer were reported to be one of the defining improvements of the Llama 3 model to its predecessor, Llama 2 (Meta, 2024a).

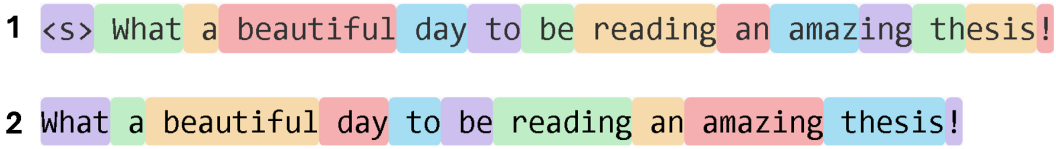


Figure 3

Examples of tokenized sentences, with colors indicating the different tokens. Sequence number 1 is tokenized using Llama 2's tokenizer, and sequence number 2 is tokenized using Llama 3's tokenizer. The applications provided by BelladoreAI (2024) were used to create the graphical representations. Note that sentence 1 displays a beginning-of-sequence (BOS) token while sentence 2 does not, but this is an inconsistency, as in reality both tokenized sequences would begin with a BOS token.

As the tokens themselves are not operated on, but their respective embeddings are, the sequence of tokens needs to be converted to a sequence of embeddings $\boldsymbol{X}$–known as the *input embedding matrix*, which is done through indexing into the embedding matrix $\boldsymbol{E}$:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{E}_{t_1} \\ \boldsymbol{E}_{t_2} \\ \vdots \\ \boldsymbol{E}_{t_n} \end{bmatrix} \in \mathbb{R}^{n \times d}, \tag{1}$$

where n is the number of tokens in the sequence, $\boldsymbol{E}_{t_i}$ denotes the embedding vector corresponding to the token at position i, with $t_i \in \{1, \ldots, |\mathcal{V}|\}$ being the token's index in the vocabulary. This mechanism works much like dictionary lookup, matching the tokens to their learned embeddings.

During training, all input sequences, also called *examples*, tend to be transformed such that they have the same length. Though training on equally sized examples is not strictly necessary–as the training

loop can handle batches with varying sequence lengths–having equal sized examples does stabilize computational load per training iteration, which allows for better resource management. A stable workload allows larger batch sizes without risking out-of-memory (OOM) errors when exceptionally long sequences are encountered. The sequence lengths of the examples are determined by the *maximum sequence length* (MSL), also denoted as n_{max}, which is a chosen hyperparameter. Input sequences shorter than the MSL are *padded*, where non-informative "PAD" tokens are appended to the end of the sequence; input sequences longer than the MSL are *chunked*, meaning that they are cut off at the point where the length of the sequence equals the MSL.

Chunking risks confusing the model, as it introduces sudden breaks in the sequence. As decoder-based models only use the preceding tokens to predict, this does not affect learning of the tokens in the sequence before the break, but it does affect learning of the sequence after the break: the sequence may now appear to start in the middle of a sentence or piece of text, so the model must find to predict these tokens without seeing the context from which they would have logically followed. To address this issue, chunking is often done with *stride*, which is a parameter that determines the number of tokens from the preceding chunk that are used at the beginning of the succeeding chunk. The intuition behind this, is that even though this does not truly avoid this issue of erratic breaks, it does allow the LLM to incorporate contextual information for all tokens in the sequences it is trained on. Another common method is to truncate sequences to fit the MSL, but this is suboptimal as it discards text (and thereby information) that may have been valuable as training data. During inference (e.g., in a chat setting), sequences are often truncated from the beginning, causing the model to lose the oldest context.

2.2.2 Self-attention mechanism

The self-attention mechanism is the core component that enables transformers to model dependencies between all positions in a sequence simultaneously. The self-attention mechanism transforms input embeddings by allowing each embedding to *attend* to all other embeddings in the sequence, leading to the embeddings being updated by their context, which is how LLMs model contextual information. These *contextualized representations* can capture both short- and long-range dependencies in the sequence. Originally introduced by Bahdanau et al. (2014) for neural machine translation, the attention mechanism was later adapted into the self-attention variant by Vaswani et al. (2017), which computes attention weights between all pairs of tokens within the same sequence. The pairwise combinations of all n tokens in the sequence are collected in an $n \times n$ attention matrix, which entails a computational complexity of $\mathcal{O}(n^2)$; this *quadratic complexity* w.r.t. the sequence length is dominant for the entire LLM.

Given an input sequence $\boldsymbol{X} \in \mathbb{R}^{n \times d}$, the mechanism first projects the input into three distinct subspaces: queries, keys, and values. Intuitively, the *query* vectors represent what type of information each token is seeking from other positions in the sequence (e.g., "*I need adjective information*"), the *key* vectors encode what type of information each token can provide (e.g., "*I am an adjective*"), and the *value* vectors contain the actual semantic content (e.g., *beautiful* or *deep*) that will be used to update the embeddings when the query and key align (or match).

The first step in the self-attention computation is to project the input embeddings into query, key, and value representations:

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q, \quad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K, \quad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V \quad , \tag{2}$$

where $\boldsymbol{W}_Q, \boldsymbol{W}_K \in \mathbb{R}^{d \times d_k}$ and $\boldsymbol{W}_V \in \mathbb{R}^{d \times d_v}$ are learned projection matrices, with d_k and d_v being the key/query and value dimensions respectively (in practice these are often chosen such that

$d_k = d_v = d$). This yields $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{n \times d_k}$ and $\boldsymbol{V} \in \mathbb{R}^{n \times d_v}$. Second, attention scores are computed using scaled dot-product attention:

$$\boldsymbol{S} = \frac{\boldsymbol{Q}\boldsymbol{K}^\mathsf{T}}{\sqrt{d_k}} \quad , \tag{3}$$

where $\boldsymbol{S} \in \mathbb{R}^{n \times n}$ contains similarity scores between all token pairs, and the scaling factor $\sqrt{d_k}$ prevents the dot-products from growing too large, helping stabilize the gradients during training.

Note that by taking the outer product of the $\boldsymbol{Q}$ and $\boldsymbol{K}$ matrices, the elements of the resulting matrix are the dot-products of the row vectors of the original matrices. The essential property of the dot-products is that their signs and magnitudes have a geometric interpretation: a negative dot-product implies the angle between the vectors is greater than π (in radians, or 90°), a near-zero dot-product implies an angle around π, and positive means an acute angle. This angle is important, because it is used to measure the similarity between two tokens using the *cosine distance*, which is 0 when the angle is 0, 1 when the angle is π, and -1 when the angle is 2π. This implies that the attention scores can be seen as the similarities between the key and query vectors of the sequence's embeddings, with positive values implying similarity, and negative values implying dissimilarity.

Third, an *attention mask* is applied to control which positions can attend to each other:

$$\boldsymbol{S}' = \boldsymbol{S} + \boldsymbol{M} \quad , \tag{4}$$

where $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ is the attention mask, containing zeros for allowed connections and large negative values (e.g., -10^9) for prohibited ones, ensuring masked positions receive negligible attention weights. The attention mask serves two purposes: preventing attention to padding tokens (padding mask) and, in causal language models (i.e., autoregressive or unidirectional LMs, which all generative LLMs are), blocking attention to future tokens (causal mask). When large negative values are added to the attention scores before insertion into the softmax function, the resulting attention weights for those masked positions become effectively zero, ensuring the model only incorporates information from the appropriate tokens.

Fourth, the attention weights are obtained through softmax normalization:

$$\boldsymbol{A} = \text{softmax}(\boldsymbol{S}') \quad , \tag{5}$$

where each row of $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ represents a probability distribution over all sequence positions, representing the normalized degree to which the tokens attend to each other. Note that definition of the softmax function is provided with Equation 35 in Appendix A.

This is where the geometric interpretation of the dot-products comes in. As the softmax function assigns a larger (smaller) value to positive (negative) inputs, and the signs reflect the cosine similarity between the key and query vectors, this mechanism leads to the most attention being given to the most similar key-query pairs. Hence, this attention mechanism indeed leads to attention being given to aligned keys and queries (as was suggested with the intuitive example earlier), while ignoring those that do not.

Finally, the output is computed as a weighted combination of value vectors, followed by an output projection:

$$\text{Attention}(\boldsymbol{X}) = \boldsymbol{A}\boldsymbol{V}\boldsymbol{W}_O \quad , \tag{6}$$

where $\boldsymbol{W}_O \in \mathbb{R}^{d_v \times d}$ projects the result back to the model dimension, and is known as the *output projection* matrix. Hence, the complete self-attention transformation can be expressed as:

$$\text{Attention}(\boldsymbol{X}) = \text{softmax}\left(\frac{\boldsymbol{X}\boldsymbol{W}_Q(\boldsymbol{X}\boldsymbol{W}_K)^\mathsf{T}}{\sqrt{d_k}} + \boldsymbol{M}\right)\boldsymbol{X}\boldsymbol{W}_V\boldsymbol{W}_O \quad . \tag{7}$$

In practice, transformers employ *multi-head attention*, where the self-attention is computed independently for partitions of the embeddings; the heads refer to the separately applied attention mechanism to each of the partitions, which are all still computed in parallel. What this achieves, is that each of the heads may now focus on different forms of contextual relationships. Rather than using a single set of query, key, and value projections, multi-head attention uses h different attention heads, where each head may learn to capture distinct types of dependencies between tokens, meaning that multiple aspects of contextual information may be separately incorporated in the same self-attention layer.

Multi-head attention splits the model dimension d across h heads, such that each head operates in a lower-dimensional space of size $d_k = d_v = d/h$. For each head $i \in \{1, \ldots, h\}$, separate projection matrices are learned:

$$\boldsymbol{Q}^{(i)} = \boldsymbol{X}\boldsymbol{W}_Q^{(i)}, \quad \boldsymbol{K}^{(i)} = \boldsymbol{X}\boldsymbol{W}_K^{(i)}, \quad \boldsymbol{V}^{(i)} = \boldsymbol{X}\boldsymbol{W}_V^{(i)} \quad , \tag{8}$$

where $\boldsymbol{W}_Q^{(i)}, \boldsymbol{W}_K^{(i)}, \boldsymbol{W}_V^{(i)} \in \mathbb{R}^{d \times d/h}$ are the projection matrices for head i, which are then used to compute attention independently for each head:

$$\text{head}^{(i)} = \text{softmax}\left(\frac{\boldsymbol{Q}^{(i)}(\boldsymbol{K}^{(i)})^\mathsf{T}}{\sqrt{d/h}} + \boldsymbol{M}\right)\boldsymbol{V}^{(i)} \quad , \tag{9}$$

where $\text{head}^{(i)} \in \mathbb{R}^{n \times d/h}$ represents the output of the i-th attention head.

The outputs from all heads are concatenated and projected through a final linear transformation:

$$\text{MultiHead}(\boldsymbol{X}) = \text{Concat}(\text{head}^{(1)}, \ldots, \text{head}^{(h)})\boldsymbol{W}_O \quad , \tag{10}$$

where $\text{Concat}(\cdot)$ denotes concatenation along the feature dimension, resulting in a matrix of size $\mathbb{R}^{n \times d}$, and $\boldsymbol{W}_O \in \mathbb{R}^{d \times d}$ is the output projection matrix that combines information from all heads. Hence, the complete multi-head self-attention mechanism can be expressed as:

$$\text{MultiHead}(\boldsymbol{X}) = \text{Concat}\left(\text{softmax}\left(\frac{\boldsymbol{X}\boldsymbol{W}_Q^{(i)}(\boldsymbol{X}\boldsymbol{W}_K^{(i)})^\mathsf{T}}{\sqrt{d/h}} + \boldsymbol{M}\right)\boldsymbol{X}\boldsymbol{W}_V^{(i)}\right)_{i=1}^{h}\boldsymbol{W}_O \quad . \tag{11}$$

2.2.3 KV-cache optimization and FlashAttention

During inference, tokens are generated sequentially using the preceding sequence. Of the $\boldsymbol{Q}$ matrix, only the query for the newly generated token is needed, as only the query vector of the newly generated token provides new information, so the queries of the prior tokens may be discarded. The key and value representations for preceding tokens are needed to compute the attention scores for the new token. Instead of recomputing them, they can be reused, which avoids redundant computation. To reuse them, the $\boldsymbol{K}$ and $\boldsymbol{V}$ matrices need to be *cached* (i.e., stored in memory) for the entire sequence. As the sizes of these matrices scale quadratically with sequence length (as explained in the beginning of this section), the *KV-cache* is the main driver of the memory footprint of LLMs during inference, which in turn heavily influences inference speed and computational burden; this is why optimization of the KV-cache is a lively topic of research.

Grouped-query attention (GQA), as used in LLaMA models (Meta, 2024a; Touvron, Martin, et al., 2023), shrinks the KV-cache by allowing multiple attention heads to share the same key and value projections, cutting memory usage by the same factor as the number of heads sharing the same KV projections. Though this limits representational capacity in the attention blocks, this method was found to cost only minimal loss in performance (Ainslie et al., 2023). A highly notable innovation is *multi-head latent attention* (MLA), introduced by A. Liu et al. (2024), the technical report detailing DeepSeek's V2 model design. In MLA, the input matrix is projected into a shared latent space, and only these latent representations are stored in the KV-cache. When computing attention for a new token, the latent vectors are up-projected to reconstruct the head-specific key and value matrices on-the-fly. This approach yields drastic memory savings: for DeepSeek-V2, a decrease in KV-cache size of 93.3% and improvement of generation throughput of up to 5.76× was achieved compared to their earlier DeepSeek 67B model, without virtually any loss of performance.

Another vital innovation of the self-attention mechanism is *FlashAttention* (Dao et al., 2022). The main constraint for GPU speed is not computation–measured in *floating point operations* (FLOPs), but memory bandwidth, which is how much data can be moved per unit of time, measured in bytes per second. Modern GPUs have a vast capacity for arithmetic operations, but the rate at which data can be read from and written to memory is typically the constraining factor, meaning that GPUs are *bandwidth–bound*. Another required realization is that there exists a hierarchy in the types of working memory available to the GPU, ranging from fast but small to large but slow. The fastest available memory consists of registers and *static random access memory* (SRAM), which are built directly on the GPU chip and therefore have very quick access times. By contrast, *video random access memory* (VRAM) consists of off-chip high-bandwidth *dynamic random access memory* (DRAM)[2], which is much larger but also much slower to access compared to on-chip SRAM.

The main culprit for both the computational and memory complexity of the attention mechanism is the attention score matrix $\boldsymbol{S}$, which is of size $n \times n$, and consequently scales quadratically with sequence length. While the quadratic computational complexity is unavoidable for *exact attention*–as all dot products must be computed–the quadratic complexity with respect to memory bandwidth can be avoided. In particular, it is not necessary to *materialize* and store the entire $n \times n$ score matrix in VRAM, which would otherwise involve repeatedly loading and writing this potentially extremely large matrix between slow global memory and the processor.

FlashAttention avoids this by computing attention in smaller submatrices, or *tiles*, which are kept entirely in the much faster on-chip SRAM while all steps of the attention computation are carried out locally within the tile. By working in tiles, FlashAttention eliminates costly read and write operations between SRAM and VRAM for intermediate results, most importantly avoiding that the full attention score matrix $\boldsymbol{S}$ ever needs to be stored in off-chip memory. Concretely, FlashAttention directly computes the product of attention weights with the value vectors:

$$\boldsymbol{AV} = \text{softmax}\left(\frac{\boldsymbol{QK}^\top}{\sqrt{d_k}} + \boldsymbol{M}\right)\boldsymbol{V}\ . \tag{12}$$

The softmax normalization is computed through an *online* technique that allows the normalization statistics to be updated incrementally as new tiles are processed, and the attention mask is applied on the fly by checking for disallowed entries as each block of queries and keys is handled. In practice, a subset of $B \ll n$ query, key, and value vectors are loaded into SRAM at a time. These subsets are processed completely while loaded in fast memory, after which the resulting partial outputs are accumulated, and only the final outputs of the attention block are written back to VRAM.

[2]In the literature VRAM is also referred to as high-bandwidth memory, or HBM.

The essential consequence of this method is that, although FlashAttention still performs all pairwise dot-products–and therefore has the same $\mathcal{O}(n^2)$ computational complexity–it avoids the $\mathcal{O}(n^2)$ memory traffic required for materializing the full attention score matrix. Instead, each input matrix ($\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$) is streamed exactly once from VRAM, and only the final output $\boldsymbol{O}$ is written back, yielding linear (memory) bandwidth complexity with respect to the sequence length: $\mathcal{O}(n)$. Since bandwidth, not computation, is the true performance bottleneck on modern GPUs, FlashAttention thereby changes the runtime scaling of attention from quadratic to effectively linear in practice.

2.2.4 Positional encodings

The self-attention mechanism, nor the MLPs or embedding matrix, have any inherent method of encoding relative positions of tokens in the sequence: they are inherently *position agnostic*, or formally, *permutation-equivariant*. This is crucial, because without such method it would be impossible for the model to be aware of the perturbation of tokens it is fed during training or inference: the model should be able to understand the *relative positions* of tokens (e.g., two tokens before the reference token). To overcome this issue, positions need to be encoded in the model, for which the original transformer from Vaswani et al. (2017) used *sinusoidal positional encodings* (SPEs)[3]. As illustrated in Figure 2, these are added to the input embedding matrix $\boldsymbol{X}$ at the bottom of the model before the transformer blocks. The SPEs are dictated by the following equations:

$$\mathbf{SPE}_{(m,\,2i)} = \sin\left(\frac{m}{10000^{\frac{2i}{d}}}\right), \tag{13}$$

$$\mathbf{SPE}_{(m,\,2i+1)} = \cos\left(\frac{m}{10000^{\frac{2i}{d}}}\right), \tag{14}$$

where m ($m \in \{0, 1, 2, \ldots, n-1\}$) denotes the position of the token in the sequence, ($i \in \{0, 1, 2, \ldots, d-1\}$ denotes the index of the dimension in the embedding, and d is the length of the embedding vectors. Hence, these positional embeddings are collected in a matrix $\mathbf{SPE} \in \mathbb{R}^{n \times d}$, of the same size as the input embedding matrix to which it is added. The effect of these equations are wave-like patterns that form a geometric progression, which are especially pronounced for the lower-indexed dimensions of the embeddings, as visualized in Figure 4. Vaswani et al. (2017) hypothesized that this function would be easily learned by the model, as the positional encoding of a relative positions can be represented as a linear combination of the positional encoding at the reference position (see Appendix C for proof). Note that it is also possible to make the positional encodings learnable parameters. Vaswani et al. (2017) experimented with this and got similar performance as with the sinusoidal functions, and opted for the latter because with it the model has a better chance at being able to extrapolate positional information to positions beyond the maximum sequence length in its training data.

Another important aspect of this choice of patterns is that the positional encodings–being the sine and cosine waves governed by Equations 13 and 14–exhibit *incommensurate frequencies* along the i (i.e., embedding dimension)-axis, meaning that the ratio between two such frequencies is always irrational. This implies that there can never be harmony in these waves, as they never share a common period. This implies that no two positions get the exact same positional encoding, which would easily confound the model by serving as an alias. This incommensurate property is achieved through insertion of the embedding dimension i in the exponent in the denominator in Equations 13 and 14.

However, the core limitation of SPEs is that they embed absolute token positions (e.g., position 5 or position 7) alongside relative positional information. This is problematic because absolute positions

[3]Note that "encodings" and "embeddings" are used interchangeably in the literature, as is done here.

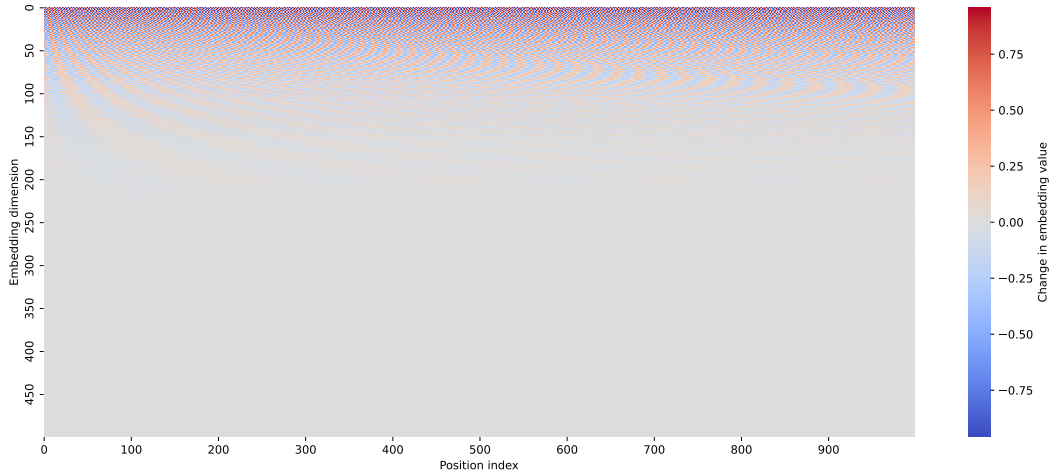


Figure 4

Changes in embedding dimension values through sinusoidal positional encodings, as used in the original transformer architecture (Vaswani et al., 2017), for a hypothetical sequence of 1000 tokens with embeddings of length 500. The wave patterns encode relative positions in the sequence, enabling position-awareness in the LLM.

are usually arbitrary; an input sequence should just as easily be shifted left or right without confounding the model: the relation between position 3 and 5 is now different than the relation between position 23 and 25, though the actual relative positional information should be the same. So, with these absolute positions, SPEs encode *red herrings* in the embeddings: any pattern that the LLM may find in the absolute positional information would be spurious, complicating the training process, which has indeed been found to degrade model performance (K. Sinha et al., 2022). Rather than learning the likelihood of a token appearing at a specific absolute position, the model should learn how likely tokens are to appear near or far from each other. This would allow the model to capture patterns such as articles typically preceding nouns. Concretely, we want positional embeddings to encode *relative* positional information without encoding *absolute* positions. One method that achieves this–used in the LLaMA family of models since LLaMA 1 (Touvron, Lavril, et al., 2023)–is the *rotary position embedding* (RoPE) (Su et al., 2023).

RoPE encodes positional information by rotating the query and key vectors in the self-attention mechanism, using a position-dependent transformation based on the same sinusoidal functions from SPE. Unlike SPEs, integrates positional information *additively* to the uncontextualized embeddings, RoPE integrates positional information *multiplicatively* into the attention computation. This allows encoding relative positions without absolute information, and enables reinsertion of the positional information in each attention block instead of only doing so once at the bottom of the model, which allows for better incorporation of the positional information.

RoPE does this, by partitioning the key and query vectors in two-dimensional pairs: one d-dimensional vector is split into $\frac{d}{2}$ two-dimensional subvectors $\boldsymbol{x}^i$:.

$$\boldsymbol{x}^{(i)} = \begin{pmatrix} x_{2i} \\ x_{2i+1} \end{pmatrix}, \quad i \in \{1, 2, \ldots, \frac{d}{2} - 1\}. \tag{15}$$

For each position m, these 2d-subvectors are rotated by a position-dependent angle θ_i through the rotation matrix $\boldsymbol{R}^d_{\Theta,m} \in \mathbb{R}^{d\times d}$:

$$\boldsymbol{R}^d_{\Theta,m} = \begin{pmatrix} \cos m\theta_1 & -\sin m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta_1 & \cos m\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta_2 & -\sin m\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta_2 & \cos m\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta_{d/2} & -\sin m\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta_{d/2} & \cos m\theta_{d/2} \end{pmatrix}, \tag{16}$$

where the set of rotation frequencies Θ is given by:

$$\Theta = \{\theta_i = 10000^{\frac{-2(i-1)}{d}} \ , \ i \in [1, 2, \ldots, \frac{d}{2}] \}. \tag{17}$$

Note that these frequencies do not depend on the positions, but are multiplied with the position index in the argument of the sine and cosine functions; similar to the SPE equations. In fact, the SPE equations are also used here, but now fashioned to encode the wave-like patterns in the rotations of the subvectors of the embeddings in the sequence. Because the query and key vectors are rotated in this way, the attention score for a query $\boldsymbol{q}_m$ at position m and a key $\boldsymbol{k}_n$ at position n is now computed as:

$$\left(\boldsymbol{R}^d_{\Theta,m}\boldsymbol{q}_m\right)^\top \left(\boldsymbol{R}^d_{\Theta,n}\boldsymbol{k}_n\right) = \boldsymbol{x}_m^\top \boldsymbol{W}_q^\top (\boldsymbol{R}^d_{\Theta,m})^\top \boldsymbol{R}^d_{\Theta,n}\boldsymbol{W}_k\boldsymbol{x}_n = \boldsymbol{q}_m^\top \boldsymbol{R}^d_{\Theta,n-m}\boldsymbol{k}_n \ , \tag{18}$$

with $\boldsymbol{x}_m$ being the input vector for position m, and $\boldsymbol{W}_k$ and $\boldsymbol{W}_q$ being the key and query transformation matrices, respectively. As can be seen from this equation, the rotations of the two vectors combine into one rotation matrix $\boldsymbol{R}^d_{\Theta,n-m}$, where:

$$\boldsymbol{R}^d_{\Theta,n-m} = (\boldsymbol{R}^d_{\Theta,m})^\top \boldsymbol{R}^d_{\Theta,n}. \tag{19}$$

This rotation matrix $\boldsymbol{R}^d_{\Theta,n-m}$ encodes the relative positions $n-m$, and is the only change to the attention score calculation. Hence, this method implicitly encodes relative positional information without tainting the model with absolute positional information: the relative positions 2 to 4 and 4 to 6 both have $n-m=2$ and have the same rotation applied to them which is precisely what is desired from positional embeddings (DeepLearning Hero, 2024). An overview of the RoPE method can be seen in Figure 5. A final note on RoPE, is that the rotation frequencies are incommensurate along the i-axis, as with SPEs, which again ensures uniqueness of the positional embeddings.

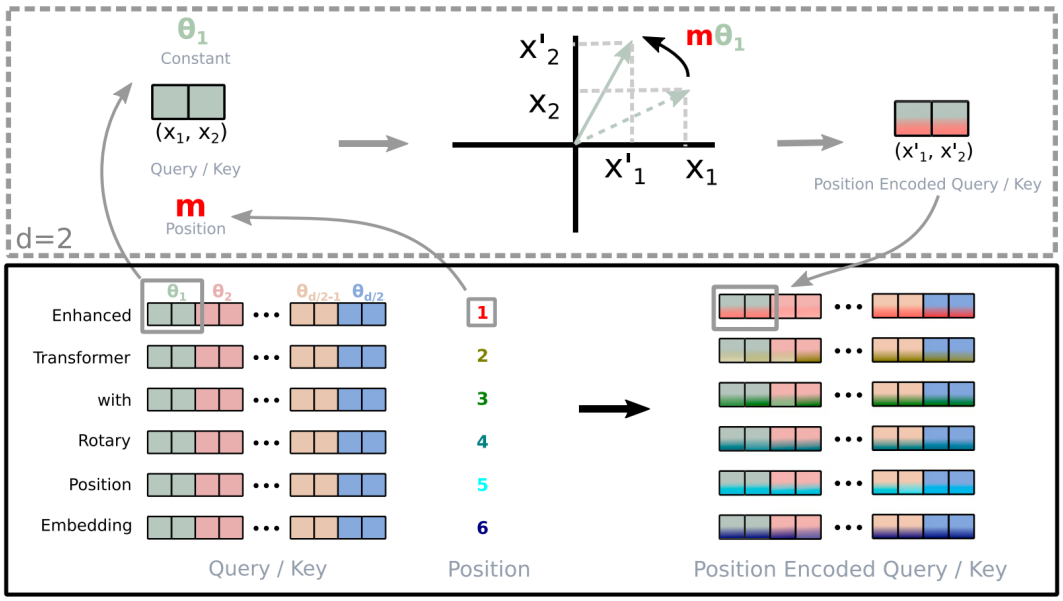


Figure 5

Illustration of rotary position embedding (RoPE), as taken from Su et al. (2023, Figure 1).

2.2.5 Multi-layer perceptron (MLP) blocks

The purpose of the MLP block is to increase the representational capacity of the embeddings in the model. The MLP block works on all embeddings independently, meaning that there is no interaction between the various embeddings of the input sequence, and (as may be worth noting) the same MLP is shared by all embeddings. The MLP works by projecting the input embeddings of dimension d into a higher-dimensional space of dimension h–where typically $h \gg d$, and passing this representation through a nonlinear activation function. After this transformation, the representation is projected back to the original embedding dimension d. This *up-* and *down-projection* enable richer interactions between features and thereby allow the network to model more complex patterns.

The original transformer architecture employed MLP blocks with the *rectified linear unit* (ReLU). ReLU is a non-linear function commonly used as activation in neural networks, which simply takes the maximum of zero and the input: $\text{ReLU}(\cdot) = \max(0, \cdot)$. As is typical in deep learning, this nonlinear function serves as an activation function to allow modeling of any function or pattern, as stipulated in the universal approximation theorem (Cybenko (1989); see Appendix A.3).

$$\text{MLP}_{\text{ReLU}}(\boldsymbol{X}, \boldsymbol{U}, \boldsymbol{D}, \boldsymbol{b}_u, \boldsymbol{b}_d) = \max\big(0,\ \boldsymbol{X}\boldsymbol{U} + \boldsymbol{b}_u\big)\boldsymbol{D} + \boldsymbol{b}_d \quad , \tag{20}$$

where $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ is the input matrix, with n being the sequence (or context) length and d the embedding dimension; $\boldsymbol{U} \in \mathbb{R}^{d \times h}$ is a learnable weight matrix for the up-projection, and $\boldsymbol{D} \in \mathbb{R}^{h \times d}$ is a learnable weight matrix for the down-projection; $\boldsymbol{b}_u \in \mathbb{R}^{1 \times h}$ is a learnable bias vector for the up-projection, and $\boldsymbol{b}_d \in \mathbb{R}^{1 \times d}$ is a learnable bias vector for the down-projection. The $\max(0, \cdot)$ denotes the ReLU activation function applied element-wise. The output of the MLP block is a matrix of the same size as the input matrix $\boldsymbol{X}$.

MLP blocks in contemporary transformers employ a fundamental innovation: the *gated linear unit* (GLU) (Dauphin et al., 2017):

$$\text{GLU}(\boldsymbol{X}, \boldsymbol{G}, \boldsymbol{U}, \boldsymbol{b}_g, \boldsymbol{b}_u) = \sigma(\boldsymbol{X}\boldsymbol{G} + \boldsymbol{b}_g) \otimes (\boldsymbol{X}\boldsymbol{U} + \boldsymbol{b}_u) \quad , \tag{21}$$

where $\boldsymbol{G} \in \mathbb{R}^{d \times h}$ is the learnable weight matrix for the *gate projection*, $\boldsymbol{b}_g$ is the learnable bias vector of length h, σ denotes the sigmoid function, and the $\otimes$ operator denotes the element-wise (Hadamard) product. The output of the GLU function is a matrix in $\mathbb{R}^{L \times h}$ (which is not yet down-projected).

The key intuition is that it is desirable to give the model more control over which signals to suppress and which to allow through. To achieve this, the GLU introduces a *gate*: in parallel to the up-projection $\boldsymbol{U}$, the input matrix is projected into a hidden space of the same dimensionality through $\boldsymbol{G}$; the difference being that this projection is passed through a sigmoid function. The sigmoid maps each element to the interval $[0, 1]$[4], effectively producing a matrix of continuous gating values. This gating matrix is then combined element-wise with the up-projected hidden representation, thereby modulating the information that passes through. This mechanism allows each unit in the hidden layer to be selectively scaled, rather than relying solely on a static nonlinearity such as ReLU. The gate can be interpreted as a dynamic filter: values near 0 suppress activations, while values near 1 let them pass largely unchanged. This additional degree of freedom has been shown to improve the expressiveness of the network and to facilitate optimization by introducing smoother gradients compared to hard thresholding activations, as argued in Dauphin et al. (2017).

Llama 3 employs *SwiGLU* (Meta, 2024a) for its MLP blocks, which uses the Swish function with $\beta = 1$ instead of the sigmoid function as activation in the GLU gate; $Swish_{\beta=1}(x) = x\sigma(x)$ is also known as the *sigmoid linear unit* (SiLU) (Ramachandran et al. (2017); see Appendix B for graphical illustrations of common activation functions and their derivatives). This usage is in line with Shazeer (2020), who found that using SwiGLU in the MLP blocks of transformers yielded the best perplexity scores in his experiments. Hence, the full MLP block with SwiGLU can be represented as:

$$\text{MLP}_{\text{SwiGLU}}(\boldsymbol{X}, \boldsymbol{G}, \boldsymbol{U}, \boldsymbol{D}, \boldsymbol{b}_g, \boldsymbol{b}_u, \boldsymbol{b}_d) = \big[\boldsymbol{X} \otimes \sigma(\boldsymbol{X}\boldsymbol{G} + \boldsymbol{b}_g) \otimes (\boldsymbol{X}\boldsymbol{U} + \boldsymbol{b}_u)\big]\boldsymbol{D} + \boldsymbol{b}_d \quad . \tag{22}$$

2.2.6 Residual connections and layer normalization

All LLMs employ residual connections–the fundamental innovation discussed in Section A.3.3 that enables (very) deep architectures–around each of the MLP and self-attention blocks (as can be seen from Figure 2. While residual connections address the VGP as previously established, they are used in tandem with *layer normalization* (Ba et al., 2016), which tackles another challenge: *internal covariate shift*, where the distribution of layer inputs changes during training. This shift is problematic because differences in input scale (variance) can destabilize learning by causing some layers to dominate others, while differences in location (mean) can push activations into *saturation regions*–where the first derivative of the function is equal or close to zero (see Appendix B–which again leads to the VGP, hampering gradient flow and learning efficiency.

Layer normalization normalizes inputs across the feature dimension for each sequence position independently, stabilizing the distribution of layer inputs throughout training. Unlike batch normalization (Ioffe & Szegedy, 2015), which also reduces internal covariate shift, layer normalization does not depend on batch statistics, making it particularly suitable for sequence modeling tasks where batch sizes

[4]Note that the bounds 0 and 1 are only reached asymptotically: $\lim_{x \to -\infty} \sigma(x) = 0$ and $\lim_{x \to +\infty} \sigma(x) = 1$. However, as discussed in Section 2.5, floating point-numbers that are larger than the range of their format allows, are set to ∞. So ∞ can be reached in practice, meaning that the bounds of the function can actually be reached in digital computation. Therefore the range is denoted as $[0, 1]$ instead of the theoretically correct $(0, 1)$.

may vary, and does not make the critical assumption that the batch is representative for the overall data distribution, which is often unrealistic. Layer normalization is defined as:

$$\text{LayerNorm}(\boldsymbol{X})_{i,:} = \frac{\boldsymbol{X}_{i,:} - \mu_i}{\sigma_i} \otimes \boldsymbol{\gamma} + \boldsymbol{\beta} \quad \forall\, i \in \{1, 2, \ldots, n\} \quad , \tag{23}$$

where $\boldsymbol{X} \in \mathbb{R}^{n \times d}$ is the input sequence matrix with n positions and d features, $\boldsymbol{X}_{i,:} \in \mathbb{R}^d$ is the feature vector (i.e., contextualized embedding) at position i, $\mu_i = \frac{1}{d}\sum_{j=1}^{d} \boldsymbol{X}_{i,j}$ is its mean and $\sigma_i = \sqrt{\frac{1}{d}\sum_{j=1}^{d}(\boldsymbol{X}_{i,j} - \mu_i)^2}$ is its standard deviation, $\boldsymbol{\gamma}, \boldsymbol{\beta} \in \mathbb{R}^d$ are learned scale and shift parameters, respectively, and $\otimes$ denotes the element-wise product.

The integration of these two components in transformer layers has evolved over time. The original transformer architecture (Vaswani et al., 2017) employed *post-layer normalization*, where normalization is applied after the residual connection:

$$\boldsymbol{Z} = \text{LayerNorm}(\boldsymbol{X} + \text{Sublayer}(\boldsymbol{X})) \quad , \tag{24}$$

where $\text{Sublayer}(\cdot)$ represents the attention or MLP blocks. However, this formulation transforms the identity matrix, which breaks with the intuition behind residual connections. Modern LLMs typically employ *pre-layer normalization* (Pre-LN), where normalization is applied before the sublayer:

$$\boldsymbol{Z} = \boldsymbol{X} + \text{LayerNorm}(\text{Sublayer}(\boldsymbol{X})) \quad . \tag{25}$$

This setting keeps the direct gradient pathways established by residual connections intact, and was found to provide superior training stability and gradient flow in very deep networks (R. Xiong et al., 2020). A notable simplification of layer normalization has emerged in recent LLM architectures: *root mean square layer normalization* (RMSNorm) (B. Zhang & Sennrich, 2019), which is employed by the Llama family of models (Touvron, Lavril, et al., 2023). RMSNorm simplifies Pre-LN's computational complexity by eliminating the mean-centering step. This approach is motivated by the observation that for many language modeling tasks, normalizing the scale of activations is more critical than ensuring zero mean, allowing for a more streamlined normalization process. Elimination of mean-centering was motivated by several reasons: (1) contemporary activation functions like Swish are inherently more robust to saturation than classical functions like sigmoid and tanh; (2) residual connections already provide alternative gradient pathways that mitigate vanishing gradients; (3) scale normalization alone sufficiently prevents extreme activation values. This simplification saves a lot of model parameters, significantly cutting the computational complexity contributed by the normalization step. RMSNorm is formulated as:

$$\text{RMSNorm}(\boldsymbol{X})_{i,:} = \frac{\boldsymbol{X}_{i,:}}{\text{RMS}_i} \otimes \boldsymbol{\gamma} \quad \forall\, i \in \{1, 2, \ldots, n\} \quad , \tag{26}$$

where $\text{RMS}_i = \sqrt{\frac{1}{d}\sum_{j=1}^{d} \boldsymbol{X}_{i,j}^2}$ represents the root mean square across the feature dimension for position i. From this equation it can be seen that compared to Pre-LN, both the mean subtraction (μ_i) and the additive bias term ($\boldsymbol{\beta}$) are removed, retaining only the scale parameter $\boldsymbol{\gamma} \in \mathbb{R}^d$.

Sun et al. (2025) found that in Pre-LN, each layer normalizes its input before the sublayer, stabilizing gradient flow but not controlling the *post-residual activations*, whose variance accumulates across layers. As l increases, each residual addition increases variance, leading to excessively large activations in more superficial layers that scales linearly with depth. Their proposed solution, *LayerNorm Scaling*, addresses this by multiplying the normalized outputs (or residual branch) by a depth-

dependent factor $1/\sqrt{l}$, keeping the activation variance roughly constant throughout the network and allowing all layers to train effectively.

2.2.7 Output layer: language modeling (LM) head

The output of a generative transformer is a probability distribution over the vocabulary. During inference, the next token is generated by drawing from the probability distribution for the next token position: $\boldsymbol{p}_t \in \mathbb{R}^{|\mathcal{V}|}$, which is dependent on the preceding tokens. During training, that probability distribution is computed for all tokens in the sequence simultaneously, thereby allowing for parallel computation of the loss for the full sequence. Hence, during training the probability distribution is a matrix $\boldsymbol{P} \in \mathbb{R}^{n \times |\mathcal{V}|}$, where n is the sequence length and $\mathcal{V}$ is the vocabulary.

The language modeling head (LM head) is the final component that transforms the contextualized vector-representations into these probability distributions. Starting with the final hidden layer $\boldsymbol{H} \in \mathbb{R}^{n \times d}$, this layer is transformed to logits $\boldsymbol{Z} \in \mathbb{R}^{n \times |\mathcal{V}|}$ through a learned linear transformation, which are then converted to the probability distribution through softmax normalization. the linear projection of the final hidden state to the logits can be represented as:

$$\boldsymbol{Z} = \boldsymbol{H}\boldsymbol{W}_{\text{LM}} + \boldsymbol{b}_{\text{LM}} \quad , \tag{27}$$

where $\boldsymbol{W}_{\text{LM}} \in \mathbb{R}^{d \times |\mathcal{V}|}$ and $\boldsymbol{b}_{\text{LM}} \in \mathbb{R}^{|\mathcal{V}|}$ are the learnable weight matrix and bias vector for the linear transformation. Second, the logits are converted to probabilities using the softmax function:

$$\boldsymbol{P} = \text{softmax}(\boldsymbol{Z}) \quad . \tag{28}$$

The rest of this section will focus on the LM head during inference; how this probability distribution is used for computing the loss function during training is discussed in Section 2.3. During inference, the next token in the sequence is drawn from this probability distribution $\boldsymbol{P}$; for the token after, the probability distribution is computed using the updated sequence. This is what makes generative LLMs autoregressive: the next token is predicted using the preceding (predicted) tokens.

The probability distribution can be modified through various strategies to control the diversity and quality of generated text. Intuitively, this controls the creativity of the model, determining how erratic it may be in its choice of words. *Temperature scaling* is a technique that adjusts the sharpness of the probability distribution by dividing the logits by a temperature parameter $\tau > 0$ before passing them to the softmax function:

$$P_{t,c} = \frac{\exp\left(z_{t,c}/\tau\right)}{\sum_{j=1}^{|\mathcal{V}|} \exp\left(z_{t,j}/\tau\right)} \quad , \tag{29}$$

where $z_{t,c}$ represents the logit for token c at position t. When $\tau = 1$, the original distribution is preserved; $\tau < 1$ creates a more peaked distribution favoring high-probability tokens (lower entropy), while $\tau > 1$ flattens the distribution (higher entropy). As $\tau \to 0$, the sampling approaches deterministic selection of the highest-probability token, whereas $\tau \to \infty$ results in uniform sampling across the vocabulary. While $\tau = 0$ is undefined due to division by zero, setting temperature to zero is nevertheless possible in most LLM frameworks (such as HF's `transformers` library) to force deterministic output. Such implementations typically bypass the softmax calculation entirely and instead use an argmax function to select the token with the highest logit. This avoids numerical instabilities that would arise from scaling the logits with infinitesimally small temperature values.

Beyond temperature scaling, *top-k sampling* and *top-p sampling* provide alternative approaches to constraining the sampling space. Top-k sampling restricts consideration to only the k most probable tokens at each step, only passing the logits of these k tokens to the softmax function. Top-p sampling

selects the smallest subset of tokens whose cumulative probability exceeds a threshold $p \in (0, 1]$, effectively adapting the vocabulary size based on the confidence of the model's predictions. These techniques can be combined with temperature scaling for fine-grained control over the trade-off between coherence and creativity in the generated text, with top-p sampling being particularly effective at maintaining creativity while avoiding low-probability tokens that may lead to incoherent outputs (H. Zhang et al., 2020).

2.3 (Continued) pretraining

LLMs are trained by optimizing a loss function using gradient descent, as is the case with virtually all DL architectures. Most of the operations required for computing the loss are already covered, but to finalize this venture, this section continues from the analysis of the LM-head at Equation 28. From the probability distribution over the vocabulary $\boldsymbol{P}$, the loss is computed through the cross-entropy function:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n} \log P_{i,\, y_i} \quad , \tag{30}$$

where $y_i \in \{1, \ldots, |\mathcal{V}|\}$ is the index of the (correct) target token's probability at position i in the sequence. This formulation also shows that the loss function is equivalent to the negative log likelihood. The complete LM head transformation to loss can be expressed as:

$$\mathcal{L} = -\frac{1}{n}\sum_{i=1}^{n} \log \frac{\exp\left(\left(\boldsymbol{h}_i \boldsymbol{W}_{\text{LM}} + \boldsymbol{b}_{\text{LM}}\right)_{y_i}\right)}{\sum_{c=1}^{|\mathcal{V}|} \exp\left(\left(\boldsymbol{h}_i \boldsymbol{W}_{\text{LM}} + \boldsymbol{b}_{\text{LM}}\right)_{c}\right)} \quad , \tag{31}$$

where $\boldsymbol{h}_i \in \mathbb{R}^{1\times d}$ is the i-th row of $\boldsymbol{H}$, and the subscripts $_{y_i}$ and $_c$ denote indexing into the logit vector $\boldsymbol{z}_i = \boldsymbol{h}_i \boldsymbol{W}_{\text{LM}} + \boldsymbol{b}_{\text{LM}} \in \mathbb{R}^{|\mathcal{V}|}$ to select the y_i-th and c-th elements, respectively. This is the loss function that is minimized during training.

Pretraining, which follows the training process that has been described earlier in this section, is a massively resource-intensive operation. Pretraining of Llama 3.1 took 1.46M hours for the 8B, 7M hours for the 70B, and 30.84M hours[5] for the 405B parameter model (Meta, 2024b). Because pretraining is an *unsupervised* (or *self-supervised*) *learning* technique, no external prediction target is needed, which means it does not require text to be labeled. This makes data collection far easier, as virtually all text can be used as training data. Because of this abundance of available data, pretraining is usually not done for more than one epoch: better generalization capabilities can be better achieved with more data rather than reusing data that was already trained on.

Though sequences of variable lengths may be used in the training batches, using sequences of equal lengths is preferable as this stabilizes the work- and memory-load to the GPU. This is of particular importance because the computational burden of the self-attention layer scales quadratically with the sequence length n, as discussed in Section 2.2.2. A stable GPU-load is preferable, as this allows for more careful tuning of the per-device batch size, which allows for better GPU utilization.

2.3.1 Scaling laws

A critical observation was made by Kaplan et al. (2020), who found that the *pretraining test loss* scales as a power law with three parameters: *model size*, *dataset size*, and *training compute*. This means that by increasing these three parameters, the performance of the model improves predictably; so predictably, that these relations are called the *scaling laws*. Whereas users of LLMs tend not to measure performance by a loss function but rather by a diverse range of capabilities, this predictable

[5] To put this in perspective: the 150k SBU budget allocated to this project, at 192SBU per H100 GPU hour (SURF Foundation, 2025b), allows for about 780 H100 GPU hours.

improvement of the loss function leads to unpredictable emergent capabilities (e.g., to explain quantum physics, to write poems in the style of some author, to plot graphs in Python) (Wei, Tay, et al., 2022). As a consequence, LLMs truly become more capable as these three parameters are scaled up, but developers of an LLM that pushes the *scale-frontier* also have little control of what new capabilities that LLM will have, which means that they are essentially *buying a mystery box* (Bowman, 2023). Nevertheless, the scaling laws have led to massive investment and effort to scale up these parameters: model sizes have increased to tens and even hundreds of billions of parameters; Llama 1 was pretrained on a corpus of 1.4T tokens (Touvron, Lavril, et al., 2023), Llama 2 on a corpus of 2T tokens (Touvron, Martin, et al., 2023), while Llama 3 was trained on a corpus of 15T tokens (Meta, 2024a); and the demand for computing power from LLMs or artificial intelligence (AI) has led to NVIDIA (the market leader on GPUs) to become the world largest company by market capitalization (Randewich, 2025), and the trillion dollar investments in compute resources have been described as *"a long-unseen mobilization of American industrial might"* (Aschenbrenner, 2024).

However, it remains important to note that scale is not the only way to improve performance. DeepSeek's V3 model caused a sharp drop in valuation of firms invested in AI, as they were able to achieve state-of-the-art performance on important benchmarks while having pretrained using a significantly smaller amount of compute (Chow & Perrigo, 2025). This proved that while the scaling laws imply that *bigger is better*, smarter is also better still.

2.4 Finetuning

Where pretraining is an unsupervised learning technique–as the prediction target is a feature of the input data $\boldsymbol{X}$–LLM *finetuning* is a *supervised learning* technique, as it involves an external prediction target $\boldsymbol{y}$. This target variable can be a class, implying binary or multilabel classification, a continuous variable, implying a regression task, or text, which implies a *sequence-to-sequence* (seq2seq) prediction task.

Raffel et al. (2020) noted that formulating classification or regression prediction tasks as seq2seq prediction tasks is favorable, as the exclusive usage of seq2seq formulations allows the same model to learn from all sorts of prediction tasks, leading to improved *transfer learning* capabilities. Though this was not found to improve prediction accuracy *per se*, it makes the LLM framework much more flexible, allowing to finetune a model on all sorts of prediction tasks without needing to adapt the LLM head, as is normally necessary in DL (see subsection A.3).

A critical risk during finetuning is *catastrophic forgetting*, where a model loses previously acquired capabilities when trained on new tasks. This phenomenon, first observed by McCloskey and Cohen (1989), shows that sequential training on multiple tasks (as is the case with finetuning LLMs) can disrupt earlier knowledge. Though approaches exist to mitigate this issue (Kirkpatrick et al., 2017), this remains a common pitfall and persistant risk for finetuning.

2.4.1 Low-rank adaptation (LoRA)

The continued pretraining and finetuning methods discussed so far can be considered *full parameter finetuning*, as all parameters in the model are updated during training. *parameter efficient finetuning* (PEFT) is a collection of methods aimed at reducing the computational requirements of finetuning by requiring a smaller set of weights to be tuned, which also means that fewer need to be saved, reducing storage requirements. Arguably the most prominent PEFT method is *low-rank adaptation* (LoRA) (E. J. Hu et al., 2022)

As discussed earlier in this section, LLMs are composed of numerous large matrices of learnable parameters. LoRA leverages the insight that weight updates during finetuning often have low *intrinsic*

rank, as most of the weight changes of an update are (near) zero. This implies that a lot of the parameter updates are redundant at any iteration. For a given pretrained weight matrix $\boldsymbol{W}_0 \in \mathbb{R}^{d\times k}$, where d and k represent the input and output dimensions, respectively, LoRA keeps $\boldsymbol{W}_0$ frozen and introduces a low-rank decomposition:

$$\boldsymbol{W} = \boldsymbol{W}_0 + \frac{\alpha}{r}\boldsymbol{B}\boldsymbol{A}, \quad \boldsymbol{B} \in \mathbb{R}^{d\times r}, \quad \boldsymbol{A} \in \mathbb{R}^{r\times k}, \quad r \ll \min(d, k), \tag{32}$$

where $\boldsymbol{B}$ and $\boldsymbol{A}$ are trainable low-rank matrices, and r is the rank of the decomposition. The method can be applied to any weight matrix in the model, such as the embedding matrix $\boldsymbol{E}$, the LM-head projection $\boldsymbol{W}_{\text{LM}}$, the query, key, value projections in attention layers ($\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, $\boldsymbol{W}_V$), or the up-, down-, and gate-projection matrices in the MLP blocks (i.e., $\boldsymbol{U}$, $\boldsymbol{D}$, $\boldsymbol{G}$). The scaling factor α/r helps maintain consistent learning behavior across different rank choices, with α being set to $\alpha = 2r$ in the original paper, leaving only r to be tuned as a hyperparameter. During finetuning, only the parameters of $\boldsymbol{A}$ and $\boldsymbol{B}$ are updated, reducing the number of trainable parameters to typically less than 1% of the original model size. This dramatically decreases memory requirements for gradients and optimizer states, enabling faster training and training on consumer-grade hardware. For inference, the adapted weights can be merged with the frozen weights through addition for practically no added latency, or kept separate to enable dynamic switching between different task-specific adapters.

The value of r (i.e., the (LoRA) rank) is a crucial decision. A higher rank allows for greater model *plasticity* (or flexibility), meaning that it is better able to adapt to its task, but that comes at the cost of greater computational costs, which is exactly what LoRA aims to reduce. E. J. Hu et al. (2022) found that (very) low ranks (e.g., $r = 8$, $r = 16$) do not significantly reduce the performance on the tasks on which finetuning is performed, providing evidence for an actual leap in finetuning efficiency.

2.4.2 LoRA suboptimal for continued pretraining

However, D. Biderman et al. (2024) found that LoRA produces consistently worse results than full parameter finetuning. For finetuning, this was only found for typical low ranks such as $r = 8$, and for continued pretraining the performance gap even remained for high ranks such as $r = 256$. For finetuning, this just implies that higher ranks should be used, but the recommendation for continued pretraining is not to use LoRA or LoRA-based methods at all.

However, these recommendations were the topic of some debate. As pointed out by Han (2023) and Han and Han (2024) (from the creators of `unsloth`; see Section 4.2), the paper's implementation does not adapt all essential layers, namely the gate projection matrices, the embedding matrix, and output projection matrices, and they use a very low rank for the coding adaptation in which LoRA was found to underperform, while the math adaptation in the paper yielded better results while there the authors had set $r = 512$, while for the code adaptation they had set $r = 32$. He argues that for continued pretraining it is still advisable to use PEFT methods such as rank-stabilized LoRA (Kalajdzievski, 2023), as long as the rank is set sufficiently high ($r \geq 256$). Hence, Unsloth acknowledges the paper's findings, but nevertheless incorporates QLoRA in their continued pretraining guidelines (Han & Han, 2024). This is in conflict with the recommendations from the D. Biderman et al. (2024) paper, though they recommend not using LoRA for continued pretraining altogether if the hardware allows, where the Unsloth recommendations are only seconded if the hardware does not.

D. Biderman et al. (2024) paper was considered to be the more credible source of the two, because of its publication in the journal *Transactions on Machine Learning Research*, which implies the rigorous academic standards being upheld such as peer-to-peer review. Besides this argument of authority, the underperformance of LoRA for continued pretraining also makes intuitive sense: freezing most of

the model and using adapter layers of far lower rank inhibits the model's plasticity, as now there are much fewer parameters to tune, though (continued) pretraining requires a great deal of plasticity as it ideally leads to all linguistic patterns to be encoded. Therefore, the overarching conclusion appears to be that LoRA should only be used for continued pretraining if hardware limitations demand lower computational burden, and ranks should not be set unnecessarily low during finetuning, as very low ranks may harm finetuning performance.

2.4.3 Instruction tuning

Most consumer applications and many evaluation frameworks of LLMs involve *prompting*, where a user writes a sequence of text from which the LLM infers a response as a continuation of that sequence[6]. However, while a model that is trained on a vast corpus of text may be good at inferring continuations, this does not necessarily imply that the model is particularly suited to answering prompts. *Instruction tuning* is a technique aimed to bridge this gap, where a model is finetuned on a set of question-answer (QA) pairs, through which it learns to infer the desired response to the given question: instruction tuning tunes the model to following instructions.

2.5 Floating point formats

In essence, LLMs are a vast collection of digital numbers (i.e., parameters) that are processed through some structure. Digital numbers are always stored as strings of bits (ones and zeroes)[7], and the format used for encoding numbers in these *bitstrings* is fundamental to the performance, operational effort and memory requirements of these models. In computing, non-integer real numbers are known as floating points (or floats), and all floats have the following convention: (1) *sign*, indicating positive/negative numbers, (2) *exponent*, indicating the order of magnitude as with scientific notation, (3) *mantissa*, indicating the precise digits of the number. The sign only requires one bit, so each number format ultimately needs to strike a balance between the number's *range*, determined by the bits allocated to the exponent, and *precision*, determined by the bits allocated to the mantissa. The total number of bits used by any number format determines the effort required for operating on these numbers, both for computation and input/output (I/O) operations[8]: 32-bit precision requires double the processing costs for operating on a single value as 16-bit precision. Hence, keeping the total number of bits low is essential for keeping the operational burden manageable. Table 1 compares conventional float formats by their bit allocations.

Format	**FP16**	**BF16**	**FP32**
Total bits	16	16	32
Sign	1	1	1
Exponent	5	8	8
Mantissa	10	7	23

Table 1

Comparison of floating-point formats by the number of bits allocated to the components of the number encoding.

[6]Typically, a *system prompt* is given to the model by the developer, which contains instructions on how to respond. The system prompt serves as the beginning of the sequence of any chat or sequence of prompts and responses

[7]While this is true for classical computing, quantum computing uses qubits, which are not binary but rather exist on a sphere in a two-dimensional complex space (Nielsen & Chuang, 2010). Another line of research explores nonlinear number formats like posits (Raposo et al., 2021), which maximize efficient bit usage, but these remain binary-encoded and therefore compatible with classical computers.

[8]Computation is performed by the processing units–CPUs or GPUs–and I/O operations are performed by the memory hardware, such as DRAM, or disk storage such as a hard disk or solid-state drive (SSD).

The traditional float16 (FP16) number format stores a float as a string of 16 bits, allocating one bit to the sign, five bits to the exponent, and ten bits to the mantissa; so the emphasis in the FP16 format lies on precision rather than range, compared to bfloat16 (BF16). However, an insufficient range will lead to overflow, meaning that numbers that are too large for the range get represented as infinity, and underflow, where numbers too small for the range get represented as zero. This problem is often encountered in gradient descent optimization, and is therefore relevant to virtually all machine learning. The BF16 format, introduced by Google specifically for improving runtimes of machine learning workloads, alleviates precisely this problem (J. Yang et al., 2018). It does so by allocating more bits to the exponent, emphasizing range rather than precision and matching the $(10^{-38}, 3 \cdot 10^{38})$ range of float32 (F32), reducing the risk of over- and underflow (Wortsman et al., 2023). Therefore, BF16 is the current standard floating point format for machine learning applications.

As precision is sacrificed for the sake of range, this raises the question of whether this precision is not missed. However, as will be discussed in the following section, though number precision allows for better encoding of linguistic patterns, parameters may be quantized (i.e., converted to a lower-bit number format) with minimal loss to model performance. This resilience arises because neural networks are generally robust to some degree of noise, and the noise introduced by limited number precision is generally negligible compared to the variability already present in stochastic gradient descent (Merolla et al., 2016). Moreover, using lower precision formats like BF16 substantially reduces memory usage and computational overhead, enabling the training of much larger models within the same hardware constraints.

2.6 Quantized low-rank adaptation (QLoRA)

Because of the greatly decreased computational burden of operating on lower bit representations of numbers, being able to compress higher bit representations of the parameters to lower bit representations–a technique known as *quantization*–poses a promising avenue for efficiency improvements. While the floating point formats discussed in Section 2.5 use 16 or 32 bits per parameter, quantization typically involves reductions to 8, 4, or even fewer bits by mapping continuous values to a discrete set of levels. This dramatically reduces memory requirements but also introduces quantization error through the reduced precision. However, this trade-off has proven remarkably favorable for neural networks due to their inherent noise tolerance, as discussed in the previous section.

Dettmers et al. (2023) found that quantization can be powerfully combined with LoRA. While LoRA dramatically reduces the number of trainable parameters, the base model $\boldsymbol{W}_0$ must still be held in memory during training, typically in 16-bit precision (BF16 or FP16 format). For large models like Llama 1 65B, this alone requires over 130GB of VRAM just for the frozen weights. Dettmers et al. (2023) introduced quantized LoRA (QLoRA) to address this limitation, by combining the parameter efficiency of LoRA with aggressive quantization of the base model.

The key insight is that while finetuning requires higher precision (FP16 or BF16) for gradient updates, the frozen base weights can be quantized to 4-bit precision with minimal impact on the final model quality. QLoRA achieves this through three main innovations. First, it quantizes using a novel 4-bit NormalFloat (NF4) data type, which is information-theoretically optimal for normally distributed weights. Second, it employs *double quantization*, where the quantization constants themselves are quantized, further reducing memory overhead. Third, it uses *paged optimizers* to handle memory spikes during training, preventing OOM errors that would otherwise occur when processing longer sequences.

As illustrated in Figure 6, NF4 is constructed by discretizing the quantiles of a standard normal distribution to ensure equal expected impact from each quantization bin. However, actual weight matrices

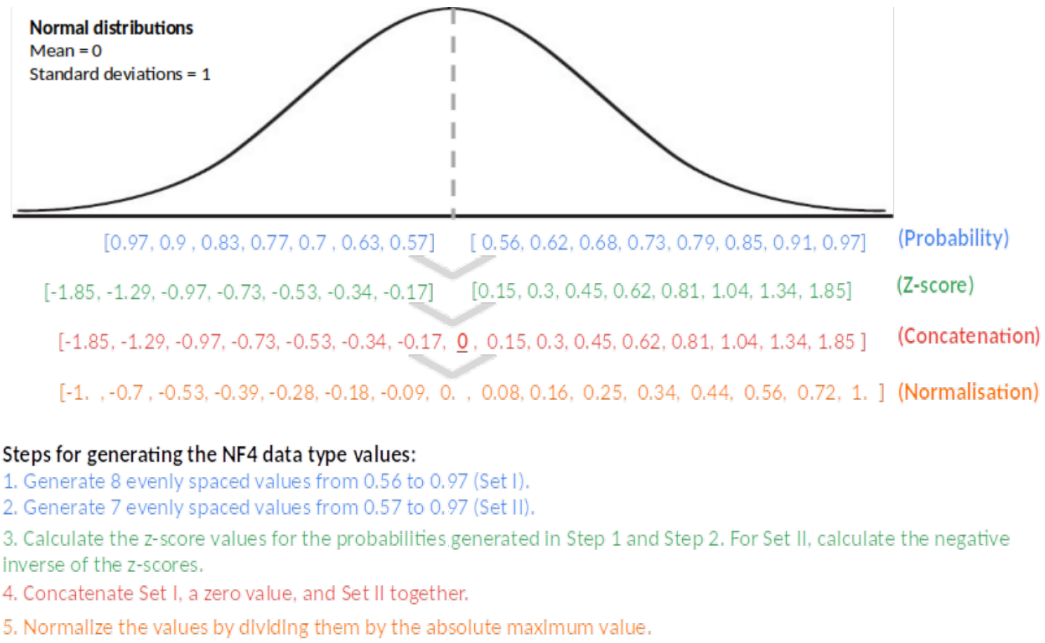


Figure 6

The steps required to construct the NF4 data type, taken from Dettmers et al. (2023, Figure 7), where the caption reads: "Equidistant values of the probability density functions are converted to Z-scores through the quantile function. To ensure the usage of all 16 values the distribution is asymmetric. As such, one half of the normal distribution has 8 and the other 7 probability values. A zero is included to have a discrete zero point. All z-scores and zero are concatenated and then normalized into the range [−1, 1] to receive the final NF4 data type."

in LLMs rarely follow a perfect standard normal distribution; they have different scales and ranges. While layer normalization normalizes the output of each layer, the weight matrices in the layer themselves are not normalized, so some variation between their distributions remains. Hence, to map these weights to the NF4 format, each weight matrix is scaled through a scaling factor known as the *quantization constant* that records its original range. During dequantization, these constants are essential: a quantized value is multiplied by its corresponding constant to reconstruct an approximation of the original weight. The double quantization process reduces the memory footprint of these quantization constants themselves. Instead of storing these constants in FP32, QLoRA quantizes them to 8-bit, with their own second-level quantization constants stored in FP32. For a model with thousands of weight matrices, this hierarchical approach yields significant memory savings with only negligible impact on reconstruction quality.

Paged optimizers–the final innovation introduced by Dettmers et al. (2023)–address the dynamic memory requirements during training by borrowing a concept from operating system design. In computing, *paging* is a memory management technique where data is divided into fixed-size blocks (pages) that can be swapped between the faster random access memory (RAM) and slower disk storage as needed. QLoRA applies this principle to optimizer states (e.g., the momentum and variance

statistics maintained by optimizers like Adam; see Appendix A.3.2): rather than requiring all gradient statistics to reside in GPU memory simultaneously, the paged optimizer divides them into pages that can be moved between GPU and CPU memory dynamically. QLoRA's implementation uses NVIDIA's unified memory feature to automatically manage this paging process. This allows training to run smoothly even when temporary memory requirements would otherwise exceed GPU capacity, as inactive pages are seamlessly offloaded to CPU memory and retrieved when needed.

This method leads to considerable efficiency gains. Finetuning Llama 1 65B, which would require over 780GB of VRAM with full 16-bit precision (Touvron, Lavril, et al., 2023), becomes possible on a single 48GB GPU. The base model is stored in 4-bit (reducing it to approximately 33GB), while the LoRA parameters and their gradients remain in BF16 for stable training. At inference time, the quantized weights are dequantized on-the-fly, with the LoRA updates applied as in standard LoRA.

2.7 Gradient checkpointing

Gradient checkpointing or *activation checkpointing* (T. Chen et al., 2016; Griewank & Walther, 2000) is an important memory optimization technique commonly employed in the training set up of LLMs and foundation models (FMs) in general. Most of the memory footprint of training LLMs comes from the activation gradients. The activations themselves are computed during the forward pass, but are also needed for computing the gradients during the backward pass, and are therefore cached to avoid redundant computation. However, caching the activation gradients is prohibitively memory intensive, while memory tends to be the active constraint in LLM operations. *Gradient checkpointing* addresses this issue by only caching some of the activations (i.e., checkpoints), and recomputing the activations in between. This increases the computational burden, but the memory savings lead to a far greater reduction runtime than the imposed by the added computation of intermediate activations during backward propagation. As stated in Dettmers et al. (2023, Appendix J), for finetuning a Llama 1 7B model on FLAN v2 without gradient checkpointing, the LoRA parameters only took up 26MB while the LoRA input gradients took up 567MB; with gradient checkpointing, the memory footprint of the LoRA input gradients averaged 18MB, significantly saving valuable VRAM capacity.

2.8 Risks and limitations of LLMs

This section serves to highlight a few important considerations on the risks and limitations of LLMs. Awareness of these is important for use cases in any sector, and it should be known that many ramifications of LLMs are still poorly understood.

2.8.1 Anthropic's circuit tracing

The flexibility of DL models, including LLMs, comes at the cost of interpretability: they are *black box* models, as it is generally not possible to determine how the output was derived. Because interpretability is often necessary for their use cases, a field of research exists that is dedicated to explaining the output derivations of DL methods: *explainable AI* (XAI). Anthropic published such an XAI method for explaining how LLMs 'decide' to generate text (Ameisen et al., 2025). With this method, pathways could be identified that were used to derive specific outputs. Circuit tracing works by employing *sparse autoencoders* to decompose the activations into interpretable features, measuring the effect of upstream features on downstream features, and using these measurements to construct *attribution graphs* which formulate the importance of pathways using features as nodes and effect sizes as edge weights.

Some notable discoveries were made when applying this method to an addition case study, which is illustrated in Figure 7. The researchers found that instead of using canonical arithmetic techniques that the LLM has seen during training, it combined an approximation-pathway to determine the range

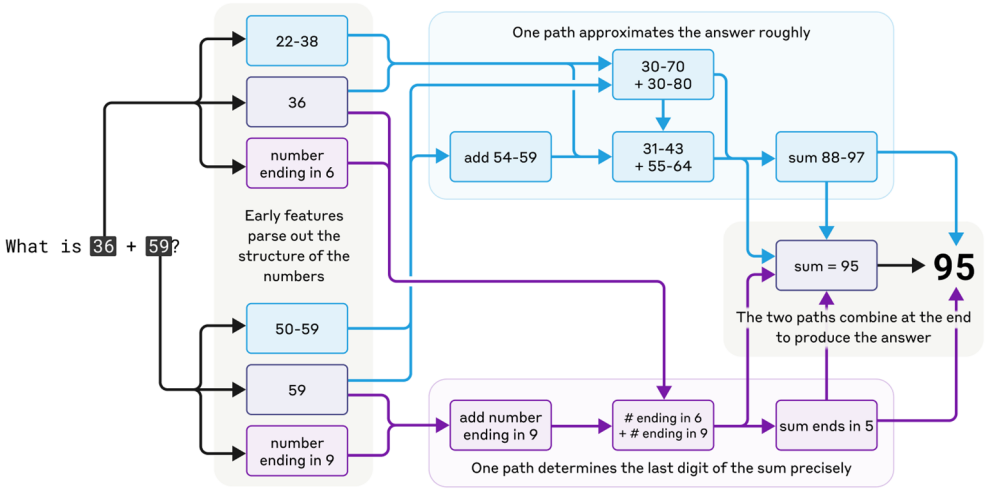


Figure 7

Diagram from Ameisen et al. (2025), illustrating the pathways used by an LLM to perform addition, that were identified through circuit tracing.

that the correct answer must be in, and an exact-pathway by which it determines the final digit that the solution must have. Combined, these pathways led to the correct answer, but the method is somewhat cumbersome compared to established arithmetic. Crucially, when the model was prompted to explain how it derived the answer, it responded that it used the canonical arithmetic techniques that can be found in any textbook on the matter, while circuit tracing revealed it actually employed an entirely different approach. This shows that the LLM is not self-aware of its internal processes, but is inclined to generate plausible-sounding explanations based on the patterns it has learned from training data.

This discrepancy has significant implications for our understanding of LLM capabilities. First, it demonstrates that these models' self-reported explanations of their reasoning processes cannot be taken at face value: they do not know their inner workings and will *confabulate* (i.e., a form of hallucination) explanations that align with established or expected methods. Second, it reveals that despite being trained on vast corpora containing established mathematical techniques, LLMs may develop idiosyncratic strategies that achieve correct results through unconventional means. This illustrates a fundamental issue: the strength of LLMs in learning contextual patterns from text may not lead to internalization of the abstract principles that those texts describe. This shows that abstract reasoning, such as mathematical reasoning, appears to be an inherent weakness of current LLMs. This is in line with the known difficulty of numerical reasoning through NLP (Bylinina & Nouwen, 2020; Thawani et al., 2021), and provides evidence in favor of the "stochastic parrots" view on LLMs: that they are only capable of imitation learning, as will be discussed shortly.

2.8.2 LLMs are stochastic parrots

Bender et al. (2021)[9] discussed societal risks and consequences of LLMs. The most prominent claim they make regarding the limitations of LLMs, is that LLMs are only capable of mimicing their training

[9]Two of the authors were co-leaders of Google's ethical AI team, and were fired for refusing to retract this publication. Google acknowledged their disagreement, stating that it was below their standards for publication. Their firing led to protests of thousands of Google employees, and sparked debate on the juxtaposition of corporate interests versus academic freedom and the risks of AI (Dave & Dastin, 2021).

data: they are *stochastic parrots*, stochastic due to the stochastic nature of inference, and parrots because they mimic their training corpus. Because LLMs tend to generate the most likely language, with probabilities based on their training data, and language reflects ideas and world views, LLMs are also more likely to reflect the dominant world views in their training data. Therefore, it is crucial to know what the training data is, and to consider what world views are dominant in it. Because of the vast corpora needed for LLM pretraining, and the most open and easily accessible source for natural language is the internet, this leads to consider what views are dominant on the internet. The authors state that the young males from developed countries are convincingly overrepresented on the internet. A leading source for natural language through publicly available web sources (NB: or was available in 2020 when the paper was written, at least) is Reddit, a website hosting a collection of fora. The authors state that 67% of Reddit users in the United States are men, and 64% are between ages 18 and 29. Similarly, surveys found that on Wikipedia, another important open web source, only 8.8-15% of contributors are female. Therefore, they argue that the dominant or hegemonic world view expressed by LLMs is that of males in their twenties from developed countries.

This tendency to reflect hegemonic views reinforces the marginalization of underrepresented groups. While Bender et al. (2021) propose balancing training data by demographic representation, such curation entails subjective decisions about which demographic categories (ethnicity, age, geography, political alignment) on which to balance or stratify the data. This subjectivity undermines the viability of demographic stratification as an ethical heuristic to the hegemonic bias.

Bender et al. (2021) also argue that data curation is infeasible for the vast pretraining corpora. This becomes apparent when considering the size of the training corpus of Llama 3.1, which, as stated earlier, comprises 15T tokens. To get a feel for how much text 15T tokens is, at 1.3 tokens per word (the average for Llama 3.1's tokenizer) and assuming 80k words per book, 15T tokens represents about 144M books, which would take the average person (reading at 250 words per minute (Brysbaert, 2019)) about 88 thousand years of continuous reading. Besides, according to Global Book Statistics (Sivo blog) (2025), an estimated 158M unique books have ever been written per 2023. This shows that curating a corpus comprising 15T tokens is an inhuman task–infeasible for human curation by any team with reasonable resources and manpower. This opens a possibility for manipulation and misuse: by populating the data sources for LLMs (i.e., the internet) with text reflecting a certain world view, LLMs may be subverted to reflect this world view as well. This would take sizable effort and resources, but employment of this strategy is not unthinkable given existing geopolitical tensions.

Beyond societal and geopolitical risks, Bender et al. (2021) highlight the environmental costs of LLMs, stating that a single BERT base model has the CO2-equivalent emissions of a trans-American flight. More recently, Llama 3.3 reportedly took 11,390 tonnes CO2-equivalent emissions (Cloud, 2024), which is comparable to 68 fully loaded Boeing 747 flights from New-York to London. The authors point out that these environmental costs are unevenly distributed, as third world countries such as Pakistan or the Maldives are projected to suffer much more severely. This creates a troubling dynamic: those whose perspectives are already marginalized in LLM training data also bear the greatest burden of their environmental consequences.

Lastly, the hypothesized parroting nature of LLMs also implies that they may be limited to *imitation learning*: merely replicating patterns in training data rather than achieving the ability to extrapolate underlying logic that would pose a hallmark of genuine *transfer learning* and of credible progress toward artificial general intelligence (AGI). This distinction is fundamental: transfer learning implies that LLMs extract abstract knowledge from pretraining that generalizes to novel tasks, while imitation learning suggests they merely *interpolate* between memorized patterns. Though some degree

of transfer learning capabilities appears evident, empirical evidence seems to suggest this degree is limited. Recent work by Chan et al. (2022) demonstrate that LLMs can appear to generalize while actually relying on superficial statistical correlations. Similarly, McCoy et al. (2023) found that GPT-4 was much better at predicting high probability outcomes than low, exhibiting simple pattern matching rather than causal reasoning. If LLMs are indeed limited to imitation, this challenges the narrative that scale and compute will eventually lead to AGI (Aschenbrenner, 2024), or that this evolution will be slow. This possibility has profound implications for the future trajectory of and investment in LLM development, both within the financial sector and without.

2.9 Financial domain adaptation

Tang and Yang (2024) show that generalist LLMs exhibit significant performance drops on specialized benchmarks, failing to capture domain-specific linguistic and semantic patterns, thereby providing empirical evidence for the need of domain-specialized LLMs. FinBERT (Y. Yang et al., 2020) pioneered financial domain adaptation through task-specific fine-tuning of BERT on financial communication texts. While this approach demonstrated improved performance on sentiment analysis and classification tasks, narrow, task-specific finetuning risks catastrophic forgetting (Kirkpatrick et al., 2017). However, Howard and Ruder (2018) show that this is less of a problem when the target tasks align more closely with previously learned features; this shows that transfer learning works better for more similar tasks.

A paradigm shift toward large-scale domain-specific pretraining occurred with BloombergGPT (Wu et al., 2023), developed by its namesake Bloomberg, which was pretrained for about 53 days on 512 NVIDIA A100 GPUs. These 1.3M GPU hours come at an estimated cost of about $3M (Wu et al., 2023; H. Yang et al., 2023). In line with the scaling laws, the large amount of resources invested yielded state-of-the-art performance on financial NLP benchmarks.

However, the massive investments in BloombergGPT may have covered inefficiencies in this training approach, and may also be solely available to private parties that only develop LLMs proprietarily. By choosing to pretrain a model from scratch instead of continually pretraining an existing model, natural language had to be learned from base principles instead of only having to focus on financial jargon and domain-specific subtleties. Y. Xie et al. (2024) introduced FinPythia, which continually pretrained a base Pythia model (S. Biderman et al., 2023), and was able to achieve competitive performance on financial NLP benchmarks. FinPythia was trained for 18 days on eight A100 GPUs at an estimated 3.5 thousand GPU hours, on 24B tokens from both SEC filings and financial news sources, and was developed by a team from Amazon (having access to considerable compute through Amazon Web Services). The competitive performance w.r.t. BloombergGPT is remarkable, as the invested GPU hours of FinPythia is approximately 1000$\times$ less. This underscores that continued pretraining can be much more efficient than pretraining from scratch.

A pioneering effort is that of FinGPT (H. Yang et al., 2023), which novelly employed financial instruction tuning. Unlike BloombergGPT and FinPythia, FinGPT focuses on generative capabilities for financial applications, including market sentiment analysis, financial report summarization, and investment advice generation. A number of open-source LLMs (e.g., Llama 2, Llama 3) were instruction-tuned using QLoRA on a financial instruction tuning dataset that was constructed by adapting a number of NLP benchmarks to the QA format used in instruction tuning. Subsequently, a reinforcement learning technique was used to dynamically update the models using stock prices as feedback. Direct comparison of performance with BloombergGPT and FinPythia remains challenging due to differences in evaluation frameworks; while the latter models report results on traditional NLP benchmarks, FinGPT's emphasis on instruction-following capabilities require more complex

evaluation metrics. This highlights a critical gap in standardized benchmarks for instruction-tuned financial models. Nevertheless, FinGPT confirms the potential of financial instruction tuning.

2.10 Evaluating financial specialization

To evaluate the financial capabilities of the sprawling number of base models and their adaptations, it is desirable to have a standardized framework to facilitate fair and transparent comparisons, and to have an evaluation that covers financial NLP as completely as possible. Lin et al. (2025), published by a team of researchers from HF, introduced a leaderboard for financial LLMs, offering benchmarks that cover a seemingly exhaustive set of task categories, as shown in Table 2.

Category	**Tasks**
Information Extraction	Named entity recognition (NER), relation extraction, causal classification.
Textual Analysis	Sentiment analysis, hawkish-dovish classification, argument unit classification.
Question Answering	Answering financial questions from datasets like FinQA and TATQA.
Text Generation	Summarization of financial texts (e.g., reports, filings).
Risk Management	Credit scoring, fraud detection, evaluating financial risks.
Forecasting	Stock movement prediction based on financial news and social media.
Decision-Making	Simulating decision-making tasks, e.g., M&A transactions, trading tasks.

Table 2

Task categories of financial NLP tasks from the Open FinLLM Leaderboard (Lin et al., 2025) with examples of their corresponding tasks.

Due to the novelty of the Open FinLLM leaderboard at the time of writing this research, the Open FinLLM leaderboard does not (yet) offer comparison to relevant financial LLMs such as FinPythia 7B (Y. Xie et al., 2024). A precursor to this framework that was used by FinPythia, is *financial language model research* (FLARE) and its associated PIXIU benchmark suite (Q. Xie et al., 2023), which together form an open-source framework designed to standardize and facilitate research on financial LLMs. FLARE provides a unified training and evaluation pipeline, which facilitates reproducible experimentation across various financial NLP tasks. The task categories covered by the PIXIU benchmarks are: sentiment analysis, classification, knowledge extraction, number understanding, text summarization, credit scoring, and forecasting. Evaluation in the FLARE/PIXIU framework is done through *few-shot prompting*, where the model is given the instruction for the task with a predetermined number of training examples, including both input and correct answer, before being shown the input for which it is to generate the answer itself. The framework was later expanded with FinBen (Q. Xie et al., 2024), offering significantly more benchmarks and covering more task categories. FinBen covers the same categories as Open FinLLM, shown in Table 2, to which it forms the direct predecessor.

For the sake of replicability, it is best to evaluate the LLM on deterministic output, instead of letting it generate tokens stochastically by drawing from the probability distribution derived in Equation 28. Instead, the standard practice is to set temperature to zero, which forces deterministic output as discussed in Section 2.2.7.

3 Data

Three sets of data were used for this research: (1) domain adaptation data, comprising a very large corpus of SEC filings; (2) instruction tuning data, which is a one-to-one mixture of the Alpaca (Taori et al., 2023) and TAT-QA (Zhu et al., 2021) datasets; (3) few-shot prompting evaluation data, comprising five financial NLP benchmarks from the FLARE/PIXIU framework (The FinAI Team, 2024). Selection and preprocessing of the domain adaptation data involved many research design choices and methodological challenges that are more appropriately discussed under the methodology section. This section serves to describe the raw and processed data with a concise overview of the processing steps, leaving disquisition on the research design choices for data selection and preprocessing to be discussed in the methodology in Section 4.

3.1 Domain adaptation data

A dataset containing all filings filed between 2012 and 2024 to the U.S.A.'s Securities and Exchange Commission (SEC) was retrieved from the SEC's from the Electronic Data Gathering, Analysis, and Retrieval system (EDGAR) database U.S. Securities and Exchange Commission (2024). On average, each filing in the raw data contained seven documents. These documents were found to contain either plain text, HTML-formatted text, XML-tables, or UU-encodings. Examples of the encountered filing contents may be found in Appendix D. This data was[10] cleaned to remove HTML tags and UU-encodings, and extract text from UU-encoded PDF files if no text extract was already present, as elaborated upon in Section 4.3.3. The raw data contained about 15T characters and the cleaned corpus comprises 6.3T characters. 0.6% of these characters are contained in the filing metadata, which contains information about the filing such as the date of filing, the filing company, the form type, and other contextual information about the filing. 0.4% of the cleaned characters belong to the document metadata, which are tags around each document denoting the start and end of the document, the name and type of the document, and an occasional description of the document's contents. The other 99% of the characters in the cleaned corpus belong to the contents of the documents. An overview of the hierarchical structure of the data can be found in Table 3, and a summary of the character counts and relative sizes of the filing components can be found in Table 4

Table 3

Overview of the hierarchical structure in the SEC filings data used for domain adaptation.

Level	Contents
Corpus	13 years (2012–2024)
Year	∼260 tarballs (one per working day)
Tarball	∼3000 filings on average
Filing	∼7 documents on average
Document	Filing contents (e.g., plain text, XML, UU-encodings)

After cleaning, a selection–or *whitelist*–was made of the form types to be included in the domain adaptation data. A total of 61 out of the 550 form types were assessed, covering 97% of characters and 93% of filings in the cleaned corpus. The whitelist contained 33 form types, thereby keeping 2.3T characters and 3.6M filings, which represent 37% of characters and 39% of filings of the cleaned corpus. Of the cleaned and whitelisted corpus, 41.6% of characters are found in 10-Q forms (i.e.,

[10]Note that data is the plural form of datum, so "data was" is actually not grammatically correct. This spelling is nevertheless used, with the justification that "data" is used as an abbreviation for "dataset".

Table 4

Character counts by totals and filing components. "Raw total" gives the total number of characters in the raw data, and "Cleaned total" gives the total number of characters in the cleaned dataset. The "Fraction" column shows the proportion of characters relative to the row's parent: "Cleaned total" is shown as a fraction of the raw total; the three lower rows are fractions of the cleaned total.

Component	Character count	Fraction
Raw total	1.48×10^{13}	-
Cleaned total	6.32×10^{12}	42.7%
Filing metadata	3.88×10^{10}	0.6%
Document metadata	2.55×10^{10}	0.4%
Content	6.25×10^{12}	99.0%

quarterly reports) and 22.4% are found in 10-K forms (i.e., annual reports), as shown in Table 5. More details on the whitelisting process can be found in Section 4.3.6 and in Appendix F.

Table 5

Distribution of character and filing shares by SEC form types in the cleaned data. Only the ten largest of either metric are reported.

Panel A: Top ten form types by share of characters in the cleaned dataset.

Form type	Share of characters
10-Q	41.6%
10-K	22.4%
485BPOS	8.1%
8-K	4.6%
20-F	3.3%
424B2	2.4%
497	2.4%
424B3	2.2%
S-1/A	1.7%
6-K	1.5%

Panel B: Top ten form types by share of filings in the cleaned dataset.

Form type	Share of filings
8-K	25.4%
424B2	13.4%
6-K	8.2%
497K	7.3%
10-Q	7.2%
FWP	5.5%
497	5.0%
CORRESP	3.7%
485BPOS	2.7%
10-K	2.6%

After cleaning and whitelisting, the data was tokenized, padded and chunked. All tokenization was performed with the tokenizer of the Llama 3.1 8B model. The data was either padded and chunked to create examples with sequence lengths of 2048 tokens, in which case a stride of 128 tokens was maintained, or to sequence lengths of 8192 tokens with stride set to 512 tokens; the tokenization methodology is discussed in Section 4.3.7. Note that these two sets were held completely separate as they were used for independent training rounds. After this tokenization process, the resulting examples were shuffled to decorrelate the data, which prevents overfitting on local correlations (e.g., temporal correlation) during training. Once the data was cleaned, whitelisted, tokenized and shuffled, it was ready to be fed into the data pipeline of the domain adaptation training process. However, despite the data shrinkage through cleaning and whitelisting, the compute budget allocated to this project was not nearly sufficient to allow for training on all preprocessed data, so training was performed on a subset of this data. For both the long and short sequence length-training rounds, 10k

examples of the domain adaptation data were separated to serve as a validation[11] set. Taking an absolute number of examples instead of a fraction of the training data as validation set is not conform conventional practice in deep learning (DL). However, due to the great size of the domain adaptation data, setting the size of the validation set to any meaningful fraction of the training data would lead to excessively large validation sets[12].

3.2 Instruction tuning data

To instruction-tune the LLMs in this project, two Question-Answer (QA) datasets were mixed with a one-to-one ratio: the Alpaca dataset (Taori et al., 2023), which is a dataset of general QA-pairs (e.g., Q: "*What are the three primary colors?*", A: "*The three primary colors are red, blue and yellow.*"), and the TAT-QA dataset (Zhu et al., 2021), which provides questions on financial text with tabular data, focusing on both financial understanding and arithmetic reasoning. TAT-QA contains 13,251 examples in its training split and 1644 examples in its validation split; Alpaca only provides a training split of 52k examples without official validation or evaluation sets. To mix these datasets, the examples needed to be in the same format, which is why the TAT-QA data was converted to the Alpaca format (see Appendix G for a TAT-QA and an Alpaca example in their respective formats). The TAT-QA data uses an average of six QA pairs per input table and text, and stores each set of pairs by their input data in JSON (i.e., JavaScript Object Notation)-format. Alpaca has a separate line for each QA pair in JSONL (i.e., JSON Lines)-format with its elements stored under the keys: *instruction*, defining the question or command; *input*, with optional input for when the instruction requires input to be operated on; and *output*, containing the desired response that the LLM should learn to generate from the provided instruction and input. Hence, the 13,251 TAT-QA training QA-pairs were extracted from the 2207 underlying records and converted to the Alpaca-format; the same was done for the 1644 QA-pairs from the 274 records of the validation split. To make the one-to-one mixture of Alpaca and TAT-QA examples, an equal amount of Alpaca examples were added to the training and validation splits and the splits were randomly shuffled. The resulting instruction tuning dataset contained 26,502 examples in the training split and 3288 examples in the evaluation split.

3.3 Few-shot prompting evaluation data

Four benchmarks are used for the few-shot evaluation of the models assessed in this research; the evaluation method is discussed in Section 4.6. The following datasets, with the numbers of train and test examples reported in Table 6, are employed by these benchmarks:

- **FPB** (Malo et al., 2014; The FinAI, 2024a): *sentiment analysis*. The Financial Phrase Databank (FPB) provides financial phrases taken from financial news, that are to be classified as either positive, negative or neutral. For each example in the data, the instruction is:

  > *"Analyze the sentiment of this statement extracted from a financial news article. Provide your answer as either negative, positive, or neutral."*

  and an example input phrase is:

[11]The validation set is sometimes also referred to as the evaluation set, and is consistently referred to as the evaluation set in the HF `transformers` framework (all arguments referring to it have the prefix "eval"). However, this terminology is inconsistent with the broader field of machine learning, as evaluation data should refer to the data used to evaluate model performance after training–which therefore can be used interchangeably with "test" data–while evaluation/validation of the model performance during training should strictly be referred to as the validation data.

[12]For the 8192-token sequences, with 10k examples validation loss is calculated over $8192 * 10k = 81.92M$ tokens and takes about 80 minutes to compute. With validation set to occur every thousand iterations of the total 5000 training iterations, total validation time sits at about 6 hours out of the total 110 hours of approximate training time; which was deemed a reasonable balance.

"Hearst will be able to consolidate about 20% of all Russian market for advertising in press after the purchase."

- **FiQA SA** (ICE-PIXIU, 2024; Maia et al., 2018): *sentiment analysis*. The Financial QA (FiQA) sentiment analysis (SA) provides the same task of labeling a phrase as positive, negative or neutral. The original FiQA task goes one step further, and assesses *aspect-based sentiment analysis*, whereby an element needs to be identified from a phrase with an accompanying sentiment score in the range $[-1, 1]$. However, the FLARE implementation simplified this task to only assess sentiment without the need to identify the different aspects. This is a point of confusion, as Y. Xie et al. (2024) incorrectly report the FiQA-SA task to be aspect-based, as they use the simplified FLARE task as is used here. For this task, the instruction for all each question in the dataset reads:

  "What is the sentiment of the following financial post: Positive, Negative, or Neutral?"

  with an example input phrase being:

  "Verizon and AT&T accused of hurting rivals"

- **Headlines** (A. Sinha & Khandait, 2021; The FinAI, 2024b): *classification*. This dataset provides headlines from news articles, and instructs for each a task related to the price of gold. The dataset contains multiple queries for each of the headlines it contains. For the headline:

  "dec. gold up $2.50 at $1,053.10/oz on globex"

  a query reads:

  "Look for indications that the price of gold is increasing. In the news headline, can you identify a Asset Comparision pertaining to gold? Please answer Yes or No."

  And for the headline:

  "Gold maintains slight gain after ECB keeps monetary policy unchanged"

  an accompanying query is:

  "Consider if the headline compares gold with other assets. Does the news headline mention anything about a Direction Up in gold commodities? Please answer Yes or No."

- **NER** (Alvarado et al., 2015; The FinAI, 2024c): *named entity recognition*. In the FLARE-NER dataset, each sentence is extracted from financial agreements within SEC filings and contains span-level annotations identifying entities such as `PER` (person), `ORG` (organization), or `LOC` (location). In one training example, a sentence:

  "... Elon Musk, CEO of SpaceX, announced the launch from Cape Canaveral."

  would have to be annotated with entities: "Elon Musk (PER)", "SpaceX (ORG)", "Cape Canaveral (LOC)". As is conform the given instructions:

"In the sentences extracted from financial agreements in U.S. SEC filings, identify the named entities that represent a person ('PER'), an organization ('ORG'), or a location ('LOC'). The required answer format is: 'entity name, entity type'."

Dataset	Train	Test
FPB	3,100	970
FiQA SA	750	235
Headlines	71,891	20,546
NER	408	98

Table 6

Summary of the number of examples in the train and test splits of the datasets used for few-shot prompting evaluation.

4 Methodology

This section is structured as follows: first, the hardware and software used for the project are discussed; domain adaptation is discussed next, after which instruction tuning is discussed, and finally the few-shot prompting evaluation method is discussed. Figure 8 provides a graphical illustration of the methodology.

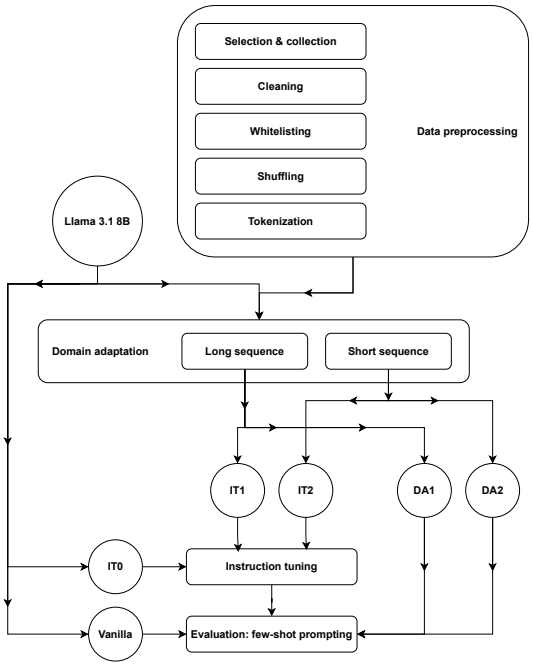


Figure 8

Flowchart of the research methodology. Rectangles with softened corners represent methodological steps, and circles represent models. Five models were evaluated: the base Llama 3.1 8B ("Vanilla"); IT0, the instruction-tuned base model without domain adaptation; DA1 and DA2, the long- and short-sequence domain-adapted models without instruction tuning; and IT1 and IT2, the long- and short-sequence domain-adapted models with instruction tuning.

4.1 Hardware

The NVIDIA H100 GPU partition of the Snellius compute cluster was used for all operations performed on LLMs in this research. GPUs excel at highly parallelized operations, which is massively beneficial to operations on large matrices that make up the many parameters of LLMs. Compared to its predecessor, the NVIDIA A100 GPU, the H100 GPU was found to be about five times faster for training the Llama 2 70B model, four times faster for training GPT-3, can be up to 30 times faster for inference of the largest models, and is six to seven times faster for other high-performance computing (HPC) applications: genome sequencing and 3D Fast Fourier Transforms (NVIDIA Corporation, 2022). The H100 GPU has 80GB VRAM capacity and a memory bandwidth of about 3.3TB/s, while the A100 GPU has 40GB VRAM with a bandwidth of about 2TB/s. The H100 GPU gives a clear advantage over the A100, which has been the industry standard before the H100 supplanted it. The Snellius compute cluster was expanded with 88 H100 GPU nodes in Q2 2024, in addition to the 72 previously installed A100 GPU nodes (SURF Foundation, 2025a). Both the A100 and H100 nodes on Snellius contain four GPUs each, allowing for multi-GPU distributed learning.

The thin[13] Genoa CPU partition was used for tasks requiring much sequential processing, which are about all tasks that do not involve an LLM. Where GPUs excel at highly parallelized operations such as matrix multiplications, CPUs excel at sequential operations. A thin Genoa CPU node contains one CPU with 192 cores (The CPU cores themselves may also be referred to as CPUs) and has 412GB (i.e., 384GiB) of dynamic random access memory (DRAM; this is the working memory, or RAM, available to the CPU), of which roughly 330GB can be allocated to jobs. Also, CPU budget is less expensive than GPU budget, so tasks were to be assigned to the CPU partition where possible.

4.2 Software

This section gives an overview of the software used in this project. The overview is not exhaustive, but serves to inform on the most relevant components.

- **SLURM**. Simple Linux Utility for Resource Management (SLURM) (Yoo et al., 2003) is the software used by the Snellius compute cluster for scheduling jobs (e.g., training an LLM) on its hardware. SLURM is accessed via a command-line interface (CLI) over a Secure Shell (SSH) connection. SLURM commands are run from within a Bash-shell. Bash is an acronym for *Bourne Again SHell*, a widely used Unix shell that extends the original Bourne shell (`sh`) with additional features. Jobs are scheduled through the execution of job scripts, which were all written in `sh`, not relying on Bash-specific features. Each job submitted to SLURM needs to be given a wallclock time; when the runtime exceeds the assigned wallclock time, the job is terminated by SLURM. The maximum wallclock time that may be assigned to a job is five days (120 hours), which posed a hard limit on how much compute resources could be allocated to the domain adaptation process, as discussed in Section 4.4.

- **Python**. A virtual environment (venv) was created that uses Python version 3.11. The versions of other software dependencies may be found in the venv creation job script.

- **CUDA**. CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) developed by NVIDIA (NVIDIA Corporation, 2025). It enables software to utilize NVIDIA GPUs for general-purpose computing. On the Snellius compute cluster, CUDA is available as a module that must be loaded before running GPU-enabled jobs.

[13]The 'thin' and 'fat' qualifications refer to the amount of dynamic random access memory (DRAM) available to the nodes, with the fat nodes having significantly more.

- **Hugging Face (HF)**. Hugging Face is a central platform and open-source ecosystem for natural language processing (NLP) and large language model (LLM) research (Wolf et al., 2020). Its `transformers` library provides high-level APIs for training, fine-tuning, and performing inference with state-of-the-art LLMs, while the `datasets` library offers efficient access to a large collection of preprocessed datasets for machine learning tasks. These tools integrate popular deep learning frameworks such as PyTorch (Paszke et al., 2019) and TensorFlow (Abadi et al., 2016), and depend heavily on PyTorch in particular. Besides, these libraries are optimized for both CPU and GPU execution, the latter of which is most relevant for this research, and also implies dependence on CUDA. Hugging Face also hosts the Hugging Face Hub, a public repository for sharing models, datasets, and training scripts, enabling collaborative research and reproducibility.

- **Unsloth**. Unsloth (Daniel Han & team, 2025) is an open-source library that patches the Hugging Face `transformers` framework with a range of efficiency-oriented optimizations for large language model (LLM) training and inference. It is particularly focused on single-GPU training, where its modifications–such as the custom gradient checkpointing algorithm with activation offloading discussed in Section 2.7–enable substantial reductions in memory usage and compute overhead. With Fast Attention 2 (Han & Han, 2024), Unsloth achieves up to a $2\times$ speedup over the standard Hugging Face implementation while maintaining training stability. However, Unsloth is currently incompatible with multi-GPU training. Although multi-GPU setups can reduce wallclock time proportionally to the number of GPUs used, they increase the total compute budget by the same factor. Since compute budget is the primary constraint in this research, the efficiency gains of Unsloth were prioritized over the speed up from multi-GPU parallelism.

  Though gradient checkpointing, as explained in Section 2.7, is a fairly common memory footprint-optimization for training deep learning architectures such as LLMs, `unsloth` offers a further optimization that drastically reduces VRAM usage. In their gradient checkpointing algorithm, activations are *asynchronously offloaded* to system (or CPU) RAM ("Unsloth Gradient Checkpointing - 4x longer context windows," 2024). Moving the activations to system RAM only costs about 1.9% extra computational overhead, while lowering the burden to VRAM by about 30%; the asynchrony means that the offloading is done in parallel to training such that the process is not blocked while the activations are offloaded. "Unsloth Gradient Checkpointing - 4x longer context windows" (2024) reports that their optimizations allow for maximum context windows that are four times larger than the standard implementation of Hugging Face with Flash Attention 2.

- **LLaMA 3.1 8B**. The LLaMA 3.1 8B model (Meta, 2024b) is an open-source, state-of-the-art large language model developed by Meta. It employs a transformer architecture with rotary position embeddings (RoPE) to enhance the modeling of long-range dependencies in text. This model was selected as the base LLM for this project due to its open-source nature and state-of-the-art performance. While the larger LLaMA 3.1 70B offers higher raw capabilities, the larger size leads to significantly higher computational and memory requirements, which makes it impractical with the available compute budget. The 8B parameter version, while still offering good performance, allows for a wider exploration of configurations and training methods (e.g., continued pretraining with short and long sequences) due to its lower computational burden.

- **FLARE/PIXIU**. The evaluation framework, discussed in Section 2.10. Usage of the FLARE/PIXIU framework revealed several bugs and outdated components, which required patching for implementation. These bug fixes are reported in Appendix I.

4.3 Domain adaptation data preprocessing

4.3.1 Data selection

As one of the three key factors in scaling laws, increasing the amount of data is essential for improving LLM quality. For effective specialization, this data must be rich in financial content–necessarily denser than the general-purpose web crawl datasets typically used for pretraining (Gururangan et al., 2020). Meeting this minimal requirement is not particularly difficult, as web-scale datasets draw from a very broad range of sources, diluting financial information. The real challenge lies in assembling a large *quantity* of high *quality* financial data. To illustrate, while financial web fora (such as on Reddit) offer abundant quantities of text, the lack of professionalism on such fora means that the quality of that financial text may be poor, which risks the LLM learning false information[14] or an unprofessional vocabulary; LLMs are stochastic parrots after all (Bender et al., 2021).

The original plan was to source data from the London Stock Exchange Group (LSEG), which had contractually granted full access to its extensive database–comprising 292 separately marketed datasets (London Stock Exchange Group, n.d.). The most promising among these were "Reuters Financial News" and "Transcripts & Briefs". As the exclusive provider of Reuters' financial news archives–produced by one of the world's largest and most reputable press agencies–LSEG offered a vast trove of high-quality, finance-dense text. The Transcripts & Briefs dataset includes automated transcripts of major business events, such as shareholder and earnings meetings, dating back to 2018 and covering a wide range of global companies. However, due to internal business considerations, LSEG ultimately revoked access and annulled the contract, rendering several months of coordination and planning fruitless.

Hence, a new data source had to be found, for which there were three promising candidates. The New York Times maintains an archive of its news online. Being an old, sizable and respected news organization, their archive promises both quality and quantity (more than 13M articles) of data, also comprising financial news. However, access to this archive lay behind a steep paywall, and even then it would have only been possible to retrieve 100 articles every four weeks.[15] Another candidate was the archive of the Wall Street Journal. Their archive, which spans the period from 1995 onward, is fully accessible with only a standard subscription, making it an excellent candidate for scraping. However, this would not be compliant to their terms and conditions, which explicitly prohibit the usage of their data for training any machine learning algorithm ("WSJ terms of use," 2025, clause 9.4).

The final candidate was the SEC's EDGAR database. As the SEC is a public institution, the filings filed to the SEC are publicly available through its EDGAR database. All firms, funds, and other organizations that are publicly listed in the US need to file legally required documents under the Securities Act of 1933, the Securities Exchange Act of 1934, the Trust Indenture Act of 1939, and

[14] In May 2024, a user asked Google's Gemini LLM: "How many rocks should I eat?", and was told: "According to geologists at UC Berkeley, you should eat at least one small rock per day." This widely cited error is attributed to training on poorly curated text from fora like Reddit, where users often post false information as a joke (Grant, 2024)–a practice known as *shitposting*.

[15] Recently their pricing scheme changed, where the steep paywall and article limit now only applies to articles from 1923–1980, where first it applied to their full archive (2007). This makes scraping of the post-1980 articles feasible, although using these for training any machine learning algorithm does not conform to their terms and conditions without explicit consent of the organization (2024).

the Investment Company Act of 1940, to the SEC, which are then publicized and archived through EDGAR (Securities & Commission, n.d.). The filings available through EDGAR vary widely. Some notable filing types are: annual and quarterly reports (10-K, 10-Q), registration statements for companies intending to conduct an initial public offering (S-1), and proxy statements filed by companies to inform shareholders of upcoming votes on corporate matters (DEF14A). There are about 550 different filing types (as was found in the data), and the database contains millions of such filings. Hence, EDGAR is chosen as the source of finetuning data for this project because the data both has quantity and quality: quantity, with millions of filings being available; quality, as the filings are filed by industry professionals and are validated by the SEC; and open to use, as there are no issues regarding legal usage for the purposes of this research due to the public nature of the database. A potential issue, however, is that filings tend to be legalistic in nature, reflecting regulatory compliance rather than financial analysis and discourse, which might make this data less suited for financial domain adaptation than financial news or transcripts.

4.3.2 EDGAR data scraping and migration to Snellius

U.S. Securities and Exchange Commission (2024) facilitates extraction of data in bulk through a machine-readable feed, where all filings of a single day are compressed in a `.tar.gz` (TAR-GZ)-file, colloquially known as a *tarball*. TAR is an acronym for "tape archive", which is a file format where a set of files are merged into a single file, by appending one to the other; much like frames on a video tape are appended to one another (hence the name). The benefit of this is that it reduces the data usage of file management-overhead, which can be especially detrimental to HPC file management systems (such as SLURM) for large numbers of files (SURF Foundation, n.d.): if millions of files were to be stored separately on a HPC file system, the overhead of the file organization would exacerbate disk usage and cause the system to become critically slow[16]. GZ is short for "GNU zip", which is a data compression tool that works on a single file, and is therefore suitable for combination with a tape archive (unlike the ZIP-format, which both archives and decompresses). A downside of the tarball-format is that decompression of the archive is inherently sequential and can therefore not be parallelized; this issue will be of considerable importance in Section 4.3.3.

With one such tarball for each of the 250 to 260 workdays per year; for thirteen years of data (2012–2024) this means that approximately 3300 such tarballs were to be downloaded, with the size of the tarballs being usually around 200MB, but often exceeding 1GB for the later years in the dataset, and with outliers ranging up to 7.74GB. A scraper was built using Python's `selenium` package (Selenium Project, 2024) to automate downloading of these files. However, the EDGAR server only serviced the scraper with a download speed of about 15GB per hour. This means that downloading one year of data would take several hours up to more than a full day of continuous downloading, making the process prone to network errors or other disruptions. Therefore, the scraper was made robust to interruptions, by having the scraper check which files were already downloaded at the start of each run, making it able to pick up where it left off after it had been disrupted. Another much needed optimization of the scraper was to parallelize the downloading process, where the scraper would download multiple files in parallel to improve the 15GB/h download speed. To do so, it was necessary for the scraper to run multiple `ChromeDriver` instances (a driver is a browser–Google Chrome in this case–controllable with automated software) in parallel. However, when running multiple such

[16] In two cases, once with a failed tokenization prototype and once while building a container from a `singularity` image file of a software package, hundreds of thousands of separate files were created on the project's scratch space. Simple removal commands failed to delete even 1% of the files after hours of runtime, so cleaning up the disk required creation of a separate script that utilized the 192 cores of a Genoa CPU to parallelize deletion. This parallelized deletion script still took several hours to complete in both cases. This illustrates the absolute necessity of archiving file formats such as TAR or ZIP.

instances in parallel, the server recognized the scraper as a bot, prompting the scraper to be denied service. After several failed attempts to fix this, the time needed to solve the issue was considered not to justify the potential gain in downloading speed. For future attempts, a promising approach would be to use different proxies for each of the parallel processes such that the server does not recognize the processes by their identical IP address, which may be implemented using Python modules such as `httpx`, `proxybroker` or `free-proxy`.

Due to the sheer size of the data, even in compressed form at 2.6TB, it was not possible to store it in full on personal, consumer-grade hardware. The project had been granted several terabytes of project space on the Snellius infrastructure: 5TB at first, which was later increased to 15TB. Because of this space and the fact that all operations on this data were to be done on the Snellius compute cluster anyways, it may seem sensible to have downloaded the data directly to Snellius. However, this was considered unwise, because the 15GB/h download speed limit would also be suffered if the scraper were run from Snellius, which means that here too the total runtime of the script would be very long. This runtime on Snellius would therefore cost a sizable amount of compute budget, while also being sensitive to the same interruptions as suffered on a personal computer. Hence, it was deemed wiser to run the scraper from a personal computer, and migrate the data to Snellius afterwards.

This data migration to Snellius was also subject to challenges. Communication and file transfer between Snellius and the personal computer was facilitated by an SSH (i.e., Secured Shell) Protocol for the command line interface and an SFTP (i.e., Secured File Transfer Protocol) for transferring files. Uploading one year of data–with sizes ranging between 83GB and 384GB per year–typically took between six and twelve hours, and was also prone to connection timeouts or transfer interruptions, where the upload would inexplicably stall. Files that were being uploaded at the moment of interruption would still appear in the destination directory on Snellius like any other file, even though these would have incomplete contents and were therefore corrupted (robust implementation of the scraper prevented this from being an issue for the initial download from EDGAR). Through manual inspection these files could sometimes be spotted by notably small file sizes, but this did not offer a foolproof solution. Instead, systematic assessment of file corruption was done through a separate script, `inquisitor.py`, which checked on corruption in both the TAR-structure and GZ-compression. This method identified several corrupted tarballs, which were replaced with new downloads.

The ± 3300 tarballs containing all filings filed in 2012 up to and including 2024, comprises about 2.6TB in its raw, compressed form. Downloading this at the estimated average speed of 15GB/h, implies that downloading continuously would have taken 7 days, 5 hours and 20 minutes. Because of interruptions and waiting time for data migration to Snellius, downloading this data and migrating it to Snellius actually took three to four weeks. Decompressing a sample of about three months of data revealed a compression factor of 7.7, which puts the estimated uncompressed raw data size at roughly 20TB.

4.3.3 Data cleaning strategy

Once all data was downloaded and migrated to Snellius, it had to be cleaned to be made suitable for continued pretraining. As no EDGAR-specific guidelines on data cleaning for LLM pretraining purposes could be found, manual inspection of the underlying filings was committed to assess what cleaning would be necessary. The guiding heuristic for this assessment is that the data should be denoised: non-informative text should be removed where possible (Parthasarathy et al., 2024). This manual inspection led to identification of the following filing components (see Appendix D for examples) and assessment of whether the component is informative or noise:

- **Metadata**: *informative*. Each filing begins with several lines of metadata identifying the company that issued the filing, the date of the filing, etc. This is deemed informative, as it provides useful context for understanding the filing itself. Besides filing-level metadata, the filings also contain document-level metadata. Each filing contains one or more documents (e.g., the filing itself, an encoded picture, an appendix), each with its own limited metadata indicating the name and type of the document, among others. This document-metadata was likewise considered informative, because it provides useful structure to the filing.

- **Plain text**: *informative*. Clearly, this is the main source of information in this data and therefore the main training objective.

- **HTML**: *noise*. HTML (i.e., Hyper-Text Markup Language) tags provide instructions on how to render the layout of a page, detailing, for example, font type and style or text alignment being left, center or right. HTML tags never contain content-related information. Therefore, these tags are not deemed useful for financial domain adaptation, and are therefore considered noise to be removed. The impact of removing these HTML-tags may be highly significant, as spot-checking revealed that more than 90% of the characters in a section of HTML-formatted text may be contained in the HTML-tags–constituting a very significant source of noise. To parse HTML, the well-established Python library `BeautifulSoup` (Richardson, 2024) was used, allowing the extraction of clean text from these sections.

- **XML**: *informative*. XML (i.e., Extensible Markup Language) is similar to HTML in structure, but with one crucial difference: the tags can contain content-related information. XML is often used to store tabular data in tree-like structures, in which the tags indicate the unit of the numerical data. For example, an XML entry may be: `<transactionShares><value>2494</value></transactionShares>` (ignoring newlines and indentations); the number 2494 is enclosed in the XML opening and closing tags, that tell that this number represents the number of shares exchanged in the transaction. Removal of these tags would only leave the plain number without any context to derive meaning from. Hence, removal of XML tags may render XML sections wholly unintelligible, which is why the XML sections were decided to be left intact. However, keeping XML sections intact leads to a considerable issue: HTML-formatted text may be nested in XML sections. In such cases, the HTML sections cannot safely be targeted through `regex` pattern recognition without also targeting XML. Because the risk of introducing nonsensical elements to the filings when removing XML tags was deemed critical, the XML-nested HTML tags were decided to leave be.

  A final note on the XML sections, is that though the tags are informative, they do not constitute natural language (e.g., `<sharesOwnedFollowingTransaction>`). Ideally, these tags would be mapped to their meaning in natural language. As manually mapping is infeasible due to the size of the data, such pre-made mappings were searched for in XML Schema Definitions (XSD), which are defined for each XML scheme used in the filings. However, though these did provide some contextual information to the tags, they did not include annotations with the meanings denoted in natural language. Hence, the only viable option was to leave the XML tags unchanged and hope that all tags are sufficiently intelligible. However, it was later found that sometimes XML is also nested in HTML sections. In these cases, the reverse was conducted, and the XML sections were parsed and cleaned along with the rest of the section, which is a suboptimal outcome. See Appendix D for an example of a cleaned XML table. This inconsistency in the workflow shows that for future work, either XML and

HTML cannot be separately treated, or a more sophisticated parser should be used that is able to separate HTML from XML.

- **UU-encodings**: *noise*. UU (i.e., Unix-to-Unix)-encodings are large blocks of characters that encode files such as images, PDF-documents, or even ZIP-folders. Though they encode actual information, they do not resemble coherent natural language, making them unsuitable for (continued) pretraining of LLMs. In fact, if such encodings were to be used for pretraining LLMs, the model would be completely confounded, as it would be nigh-impossible to learn the decoding scheme through training on masked-token prediction on the elements in the encoding, leaving the encoding for an unintelligible, incoherent mess which the model would be trained to find patterns in. At best, this would slow down training due to useless training cycles; at worst, this would destroy encoded linguistic patterns by training on data that is not at all representative of natural language, destroying predictive ability as congruent with the principle of *garbage in, garbage out)*. Hence, removing all UU-encodings was considered to be of critical importance.

  However, several filings were found that contained a UU-encoded PDF-file as its only document, meaning that removal of the encoding would clear out the filing. Through spot-checking, such PDF-files were found to exclusively be correspondence letters between company representatives and the SEC, that were scanned to store them digitally on EDGAR. Filings in more recent years (since October 2019) were found to contain the digital text extract below the UU-encoding. Therefore, to minimize loss of information, if a filing contained a UU-encoded PDF-file and no text extract was already present, the encoding was decoded and the text was extracted from the PDF-file to replace the UU-encoding. In all other cases, UU-encodings were simply removed.

Besides this document-level cleaning, in some cases filings were removed in their entirety from the data. This was conducted when: (1) the filing was a placeholder for a paper submission (identified by the line: "*This document was generated as part of a paper submission.*"), in which case it had no actual content; (2) extracting text from a UU-encoded PDF in the filing encountered an error, which may be indicative of file corruption; (3) if the filing only contained non-PDF UU-encodings, the case might arise that all content of the filing is removed and only the metadata remains, in which case the filing is discarded as there is no useful content left.

Cleaning this data is a very computationally expensive task. It requires decompressing the tarballs, reading the files, looping over the contents several times to check for matches of the `regex` patterns by which the various sections are identified, processing the sections as described in the list above, and writing these to a new TAR-file; the extensive sequential processing tasks makes these operations CPU-heavy, the raw filing needs to be kept in working memory (DRAM) during processing which is memory intensive, and the terabytes of reading from and writing to disk makes this particularly I/O-heavy. With approximately 3300 tarballs with an uncompressed size of 20TB, this is a considerable challenge that necessitates a supercomputer, which is why the thin Genoa CPUs from the Snellius cluster were used to perform this processing (details on the hardware were given in Section 4.1).

To make good use of this CPU, as many of the 192 cores as possible should be used in parallel. A double level of parallelization was used in this cleaning process: the active cores were assigned on the tarball-level, while each year could also be processed in parallel on separate CPU nodes. The latter parallelization is trivial, but the former is not: parallelizing on the tarball-level makes the process susceptible to delays due to variation in data size in tarballs, which is indeed subject to significant variation: quarterly reports, for instance, tend to be published around the same time for many firms, so

those days contain much more filings with much more data, contributing to this variability. Another option is to parallelize on the filings level, meaning that one tarball would be processed at a time, but that the filings in the tarball are processed in parallel. This parallelization on a more granular level with longer queues allows for a better spread of the workload, as the tasks (i.e., filings) are now more uniform in size and the long queues imply that workers (i.e., processor cores) may perform multiple smaller tasks while another worker is working on a long task. This reduces the risk of some cores becoming idle while others are still processing particularly large tarballs. However, this would require multiple processes needing to access both the same input-tarball and output TAR-file, which may lead to race conditions (i.e., processes blocking each other due to concurrent access of the same object) and broken process pools. Besides, the input-tarball needs to be decompressed, which is an inherently sequential task, and is therefore very likely to pose a bottleneck in the process. Parallelizing on the tarball level does not have these issues, as each worker now has its dedicated input tarball and output TAR-file, which was therefore deemed the safer choice.

But as each CPU needs to operate on a separate filing, as processing on the filings themselves is inherently sequential, the number of active cores also denotes the number of filings that need to be kept in memory. The most DRAM that can be requested for a job scheduled on a thin Genoa CPU is 334GB (empirically established), which means that each active core may use an average of 1.74GB DRAM during the cleaning process. Though this may seem like ample memory, the facts that the raw filings need to be kept in DRAM during processing and that the largest of these filings is extremely large at about 7.8GB, imply that this memory may still be far from sufficient; especially considering that memory usage exceeding the available memory only once will immediately result in the process hitting an OOM-error and getting terminated, so it is the peak memory usage that poses the actual danger. To prevent OOM-kill events, the cleaning script had to be optimized where possible for memory efficiency: tarballs were decompressed one filing at a time instead of decompressing the full tarball at once, usage and storage of intermediate objects was minimized, `regex` objects were pre-compiled, and cleaned filings were written directly to disk. Still, OOM-errors were frequently encountered during prototyping and test runs, especially for the later years in the data, which represent much more data (2024 comprises about four times more data than 2012). Ultimately, 128 cores were used for parallel processing of the years 2012–2021, and 96 cores were used for processing years 2022–2024. Note that regardless of the number of cores used, the full CPU node had to be claimed for the cleaning tasks, as all available DRAM was required.

A notable issue encountered during cleaning, is that not all characters in the data are part of the basic multilingual plane (BMP), which is the set of all characters that can be decoded in UTF-8 (i.e., 8-bit Unicode Transformation Format, the most common character encoding format), which caused an error and thereby leading to termination of the cleaning script. Upon inspection it was found that the character that threw this error was the block character (i.e.,), which was incidentally used as a bullet for bulleted lists. The only way to overcome this issue, was either to remove such non-BMP characters whenever they were encountered, or to replace them with the Unicode replacement character (i.e.,). The former option was chosen, though both could have been reasonably justified. A final peculiar issue, was that the last five days of 2024 showed unusual sizes of the filing information logs, ranging between 21–51GB instead of the 3–20MB range found for the rest of the data. The problem was found to be that EDGAR has shifted from Unix-style line endings: "$\backslash n$", to Windows-style line endings: "$\backslash r \backslash n$". This caused the `regex` patterns to fail in recognizing the line endings, entering the full filing content after following the recognized beginning of the pattern, as datapoints rather than only the text on that same line. As a result, the filing information logs now contained several duplicates of the entire filing instead of just the descriptive information. The `regex` patterns

were adjusted such that they recognize the Windows-style line endings, while remaining robust to recognition of the Unix-style line endings.

4.3.4 Data cleaning results

The cleaned dataset contains 9.2M filings, just 103k fewer than the total in the raw dataset. About 97k of these removed files were placeholders for paper submissions; the remaining 6k represent both the failed PDF extractions (compared to 268k successfully extracted PDFs), and no cases were encountered where all constituent documents were removed. An overview of the processing statistics can be found in Table 7. Additional statistics may be found in Appendix E; character and filing counts per form type may be found in Appendix F, for a subset of the largest form types on these metrics.

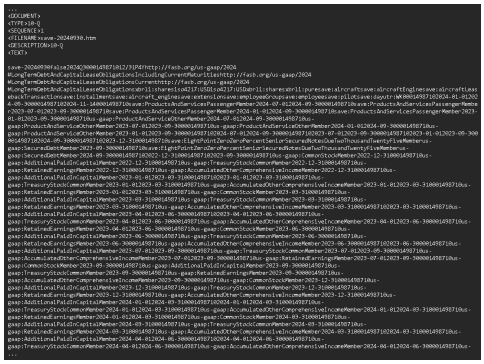
Table 7

Overview of domain adaptation data cleaning outcomes.

Processing result	**Count**
Files kept	9.2×10^6
Documents processed	6.2×10^7
Characters in cleaned data	6.3×10^{12}
HTML characters removed	5.1×10^{12}
HTML documents parsed	3.2×10^7
UU-encoded characters removed	2.7×10^{12}
UU-encoded documents removed	1.7×10^7
UU-encoded PDFs extracted	2.7×10^5
UU-encoded PDF extractions failed	5.9×10^3
UU-encoded PDF characters extracted	1.7×10^{10}
Files removed	1.0×10^5
Files removed - character count	8.8×10^{10}

After preprocessing, the resulting uncompressed TAR-files, still one for each filing day, comprised 5.9TB of data. To put this size in perspective, RoBERTa was pre-trained on 160GB of uncompressed text (Y. Liu, 2019), and the Colossal Clean Crawled Corpus from Raffel et al. (2020) comprises 750GB. To compare this size with more recent training sets is less clear-cut, as these tend to be expressed in number tokens instead of bytes or characters, which is indeed the more relevant unit for LLM training corpora; though the number of tokens does depend on the tokenizer used, so different models will state different sizes for the same corpus. However, an important comparison is that with the pretraining corpus size. As stated in Section 2.3.1, Llama 1 was pretrained on a corpus of 1.4T tokens (Touvron, Lavril, et al., 2023), Llama 2 was trained on a corpus of 2T tokens (Touvron, Martin, et al., 2023), while Llama 3 was trained on a corpus of 15T tokens (Meta, 2024a). As shown in Table 7, the cleaned corpus contains 6.3×10^{12} characters. Llama 3's tokenizer averages 4.47 characters per tokenizing for plain text from the IMDB dataset (Disparate AI, 2024). Using this as estimate, the total number of tokens for this cleaned dataset is estimtated to be $\frac{6.3}{4.47} \times 10^{12} \approx 1.4 \times 10^{12}$ (i.e., 1.4T) tokens. This puts the size of the data on par with the pretraining data of Llama 1, and to about a tenth of the pretraining data of Llama 3. Here it must be noted that since the data is used for continued pretraining, the model does not need to learn general linguistic patterns but only needs to learn nuances specific to financial language. Hence, domain adaptation data should reasonably be much smaller than pretraining corpora. Essentially, the size of the data therefore more than sufficiently meets the requirement of quantity.

(a) *Generic XBRL tags found in a 20-F filing.*



(b) *GAAP-style XBRL tags found in a 10-Q filing.*

Figure 9

Excerpts from filings containing XBRL tags, meant for automatic extraction of structured data.

4.3.5 Residual noise from machine-readable XBRL

After cleaning, manual inspection of some cleaned filings revealed that some filings, especially 10-K and 10-Q forms, contained large amounts of machine readable *XBRL* (i.e., Extensible Business Reporting Language), as shown in Figure 9. XBRL is used for automated extraction of structured data from filings. This has been mandatory for major SEC filings since 2011, but until June 2019 the XBRL was kept separate from the human readable text, so it did not show in the raw filings data. Since June 2019, the SEC has mandated that filings use *inline XBRL* (iXBRL), which means that XBRL is now shown in the raw filings data among the plain text (U.S. Securities and Exchange Commission, 2019). This is problematic, because these machine-readable XBRL tags are not suitable as domain-adaptation data, as they do not contain narrative financial text, so XBRL is essentially noise. Adding to the severity, is that together with iXBRL, HTML tags are also used in XBRL sections, which also pose substantial amounts of noise. As this was discovered late in the process, this noise was not removed from the domain-adaptation data. The share of XBRL and HTML tags in the cleaned data is unknown, but is likely to be substantial, meaning that its impact on the training process is likely to be significant. The employed method of identifying HTML sections with `regex` and removing the tags with `BeautifulSoup` would not work here, as XBRL uses various formats which are not all identifiable with HTML-style opening and closing characters (i.e., < and >), as seen from the GAAP (i.e., Generally Accepted Accounting Principles)-style iXBRL shown in Figure 9b. A method which would work, is using Arelle (Arelle Project, 2010), which is the program used by the SEC to render filings on webpages, which entails parsing XBRL from plain text. This method is recommended for future data cleaning of SEC filings. Another possible method is to use LLMs as preprocessors, as proposed in H. Zhang et al. (2023), but this is computationally more expensive without having the robust and reliable method offered by a dedicated algorithm like Arelle; so Arelle is considered the superior method.

4.3.6 Subsetting domain adaptation data on whitelisted form types

However, the great size of the cleaned filing corpus is both a blessing and a curse: a blessing, because the scaling laws dictate that a larger training corpus leads to better (out-of-sample prediction) performance; a curse, because completion of one training epoch (which means to loop over all tokens of the data in training) would take many times more computation than the compute budget for this project allows. To address this, the notion was used that not all form types are equally useful for domain

adaptation, which mandates selection of the form types which are deemed suitable. This selection, or *whitelisting*, would further distill the corpus to better enable a financial specialization to emerge in the domain-adapted LLM.

As the goal of domain adaptation is to specialize in financial natural language processing, the training data most useful for this purpose is that which comprises *narrative financial language* reflecting financial knowledge and information. This narrative text is dichotomous to *structured numerical data*, which is also prevalent in the filings. Some form types (e.g., 3, 4, ABS-EE, SC 13G) are mainly–sometimes exclusively–made up of numerical, XML-structured data. Though this numerical data does reflect financial information, such as loan default and delinquency rates or transaction volumes, the lack of natural language reflecting a financial narrative obstructs the possibility for the LLM to learn the subtleties of financial language from context. This is not to say that training on such structured, numerical data is entirely useless in domain adaptation, as such training may yield a model that is better able to extract and summarize numerical information from the data structures encountered in the training data, but it is important to realize that LLMs are designed to extract information from natural language rather than structured numerical data. The limitations of LLMs regarding numerical reasoning were discussed in Section 2.8.

According to a finance industry professional[17], the form types most relevant to investors are the 10-K and 10-Q forms: the annual and quarterly reports of publicly traded companies, and the 13F-HR form: the quarterly holdings reports of institutional investors with investment discretion of at least $100M in Section 13F assets (e.g., publicly traded U.S. stocks, options, convertible debt). The Risk Factors sections of the 10-Qs and 10-Ks were said to be of special importance, as here actual market developments are discussed instead of highly legalistic language or accounting figures common in other sections. The 13F-HR forms, however, mainly contain highly structured numerical data. Though relevant for investors, these tend not to contain narrative sections of natural language, making this form type generally unsuitable as LLM finetuning-data.

Hence, preference for narrative text was given over structured numerical data, which became the guiding principle for selecting the SEC form types to be kept for domain adaptation. A secondary criterion was the financial information deemed to be present in this form type: annual reports are likely to contain much financial information, while some form types–such as Form D–are highly legalistic and bureaucratic in nature, meaning that these would not be useful for learning a financial specialization. To make this whitelist, the form types were sorted on both number of filings, of which the largest 50 were assessed, on total number or characters, of which all form types encompassing more than 10M characters were assessed; these are the 34 largest form types in terms of total characters in the cleaned data. Due to obvious correlation between number of filings and total number of characters, this led to the assessment of 61 form types in total, of which 33 were whitelisted. Details on form type selection can be found in Appendix F.

The subset of the corpus based on the whitelist contains 3.6M filings with a total of 2.3T characters, corresponding to an estimated 0.51T tokens. This represents 37% of the characters and 39% of all filings in the cleaned corpus. A notable characteristic of the dataset is the highly skewed distribution of both filings and characters across form types, as can be seen from Tables E.1a and E.1b in the Appendix. The ABS-EE form type (Asset-Backed Securities – Electronic Exhibits) alone accounts for 2.8T characters out of the total 6.3T in the cleaned corpus, despite comprising only 33k filings. Similarly, Form 4 (which reports changes in beneficial ownership for individuals holding over 10%

[17]The consulted industry professional is Chris Kantos, Managing Partner and Head of Quantitative Research at Alexandria Technology, a London-based FinTech firm.

of a firm's stock) contributes 2.4M of the 9.2M total filings. Both of these were excluded due to their structured, largely numerical nature, but this skew is also evident across the rest of the form types. Due to this skew, the 61 assessed form types out of the total 550 encountered form types, still cover 97% of all characters and 93% of all filings in the cleaned corpus. Another important observation, is that in terms of total number of characters, 10-K forms comprise 8% of the cleaned corpus and 22% of the whitelisted corpus, and 10-Q forms comprise 15% of the cleaned corpus and 42% of the whitelisted corpus; this means that annual and quarterly reports make up 64% of the cleaned and whitelisted corpus. This happens to align well with the advice of the industry professional that quarterly and annual reports are most relevant to the industry.

This step was also used to create a single JSONL file with all whitelisted and cleaned filings, instead of keeping them in TAR-files per day. Keeping all text in a single file facilitates both further preprocessing and feeding the training data pipeline during the training process. For the latter, the JSONL-format is convenient, as it is supported by the `transformers.Trainer` object. The resulting JSONL file comprised 2.2TB.

4.3.7 Pre-tokenization, padding and chunking

An early domain adaptation training run revealed a major inefficiency in the training process. As can be seen from Figure 10, GPU utilization periodically dropped to 0% for 24.5% of the runtime of the training run, which was canceled when the issue was found to not waste compute budget. This meant that about one quarter of the entire compute budget allocated to training would be wasted as long as this issue remained, which was considered an unacceptable inefficiency. As the GPU utilization was 0% for the first few minutes of the training run, this hinted on the bottleneck being at the very beginning of the training process, which pointed to the training data pipeline.

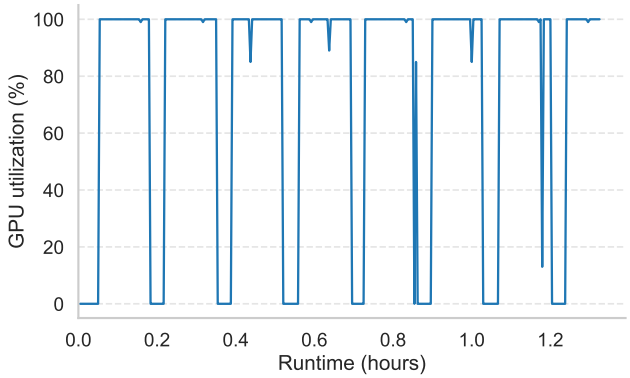


Figure 10

GPU utilization expressed as a percentage of the maximum for an early domain adaptation training run. This plot shows that for 24.5% of runtime, the GPU utilization dropped to about 0%, which revealed a bottleneck in the training data pipeline. The training run was canceled early.

The training data pipeline constituted several steps to transform the cleaned and whitelisted filings in the JSONL file to examples fit for training LLMs. First, the data was streamed from disk with a buffer of 10,000 lines. Streaming was necessary because 2.2TB is too much to load into memory all at once; the buffer helps reduce I/O overhead from repeatedly reading from disk. Second, the data was tokenized using Llama 3.1 8B's tokenizer, which was operated through HF's `transformers.PreTrainedTokenizer` class. This class also has built in chunking and padding

mechanisms. However, it was found that for Llama 3.1 8B's tokenizer, the chunking mechanism was actually not implemented, which resulted in discarding the rest of the filing after the first example was created, instead of raising a warning or error that this, therefore taking some time for the issue to be identified. This meant that a custom chunking mechanism had to be developed. This tokenizer was also found not to have a defined PAD token. The EOS token was used as the PAD token instead. Though there is some discussion on whether this does not prevent correct learning of the EOS token, the attention mask on these EOS tokens used for padding should be enough to avoid that issue (Stack Overflow Community, 2023).

Tests revealed that tokenization was the main bottleneck in the training data pipeline. As this is a necessary step, the only way to improve GPU utilization was to tokenize beforehand. The added benefit of this approach, is that this allowed for tokenization using a CPU instead of a GPU, which therefore saved the more constrained GPU budget. Hence, tokenization, padding and chunking was performed as a preprocessing step rather than leaving it in the data pipeline during training. Hence, the data was *pre-tokenized*. The pre-tokenized data was also stored as a JSONL file, but now with one line for each example. As a result, the data pipeline now only constituted buffered streaming of examples from the JSONL file and feeding these to the training process. Indeed, this massively improved GPU utilization to near maximum, as can be seen from Figure H.2. Hyperparameters used for pre-tokenization are discussed in Section 4.4.1.

4.3.8 Shuffling data

Even if not all of this data may be used in the end for pretraining due to compute budget limitations, a smaller dataset can still be randomly sampled from it. This random sampling makes sure that the data used in training is still representative of the total corpus, as it should have the same distribution. This is especially important considering the *temporal correlation* in the data: after all preceding preprocessing, the data is still sorted by the day on which the filing was added to EDGAR. Importantly, it is reasonable to assume temporal relations in the data: in 2012 the aftermath of the great financial crisis of 2007/2008 may have had far greater impact on the topics discussed–and therefore the language used–in financial reports than in 2022, when the COVID-19 pandemic may have dominated reports. Because of this, training on data in this chronological order would lead to the model learning patterns in the data that disappear later during training, which it would then be incentivized to forget. Essentially, this means that the model would overfit on temporal trends during training, instead of learning the linguistic patterns that hold in general. Therefore, it is necessary to *decorrelate* the data, which is achieved through random sampling or shuffling.

It was decided to shuffle the pre-tokenized (and chunked and padded) examples rather than the filings before tokenization, because as some of the filings are very large (most notably 10-Q's and 10-K's), keeping to the order of examples of the same filing may lead to fitting idiosyncratic patterns in a filing instead of global patterns, which poses the same sort of overfitting during training that would have been caused by the temporal correlation.

The easiest way to achieve decorrelation, would be through the default Fisher-Yates shuffling algorithm (Fisher & Yates, 1938), which is the golden standard of randomization due to its perfectly uniform distribution over all possible permutations. However, virtually all implementations of this algorithm load all data into DRAM, which is not possible given that the dataset comprises several terabytes, which is impossible to fit on the 334GB DRAM that the thin Genoa CPU nodes offer. Using Fisher-Yates to shuffle the index and construct the shuffled data, requires knowing byte offsets beforehand for parallelizing construction, and partitioning the shuffled index into buckets for the parallelized workers to avoid disk contention and race conditions. This solution appears cleaner than the

algorithm that was eventually employed, though it does face much of the same challenges. However, this workaround with byte offsets was found after this step was already completed. Hence, though not employed in this methodology, this is the method recommended for future research.

Another approach that was considered is streaming using a randomized index. However, this was considered suboptimal for two reasons. First, the streaming implementation of HF's `datasets` module does not support random selection, meaning that a custom streaming method would have to be implemented which would be a challenge in itself. The only randomization possible here, is to shuffle the examples contained in the streamed buffer, but that is not enough to break temporal correlation: e.g., this may shuffle a few months of data, but not the entire 13 years. Second, when an algorithm is given a specific line to read, it finds the data of this line by iterating over all lines in the file until the correct line is found. This could be avoided by creating a mapping of the lines to their byte offsets, and using that mapping to directly access lines with Python's `open().seek()` function, which accesses directly and therefore runs in $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$, with n being the number of lines in the data (Stack Overflow Community, 2018). Though this approach seems to offer the most efficient solution as it does not require construction of a shuffled dataset, the requirement of aligning the `datasets.load_dataset()` function with the custom streaming logic was considered too risky to implement.

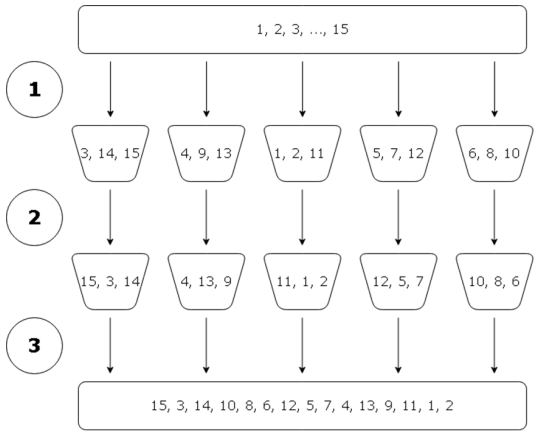


Figure 11

Schematic representation of the bucket shuffling algorithm used to decorrelate the domain adaptation data. The circled numbers indicate the steps in the algorithm. In step 1, observations are randomly distributed over the buckets; in step 2, the observations in the buckets are shuffled to break their original sequence; in step 3, the shuffled buckets are merged back into a single set in random order.

Instead, the chosen solution to this conundrum was *bucket shuffling*, as visualized in Figure 11. The first step of this algorithm is to randomly distribute the data over several buckets, with the main constraint being that the buckets cannot exceed the available DRAM (334GB), which means that the lower bound for the number of buckets is $\lceil \frac{2.2TB}{334GB} \rceil = 7$, but more buckets are preferable to improve the uniformity of perturbation probabilities; so 64 buckets were chosen. The examples are randomly distributed over the buckets, which already constitutes a first randomization, but as the examples are appended to each bucket and the examples are selected in the same order as the original data, the examples in the buckets are still sorted by date. Secondly, the observations within each bucket are shuffled using Fisher-Yates. This is now possible because the bucket does fit in DRAM, and here the

temporal correlation is truly broken. Finally, the buckets are merged in random order (again using Fisher-Yates to determine the order) to construct the full shuffled dataset. This final shuffle is perhaps superfluous, but it may still be helpful to facilitate uniformity of permutation probabilities by breaking macro-structure from the bucket assignment, which may arise from unevenly sized buckets; in any case, this final shuffle comes at negligible computational cost and uniformity of the permutation probability distribution can only be improved by this step.

Importantly, this algorithm had to be parallelized. Specifically, the first step takes very long to compute as it entails looping over the full dataset. To parallelize distribution and to avoid race conditions resulting from multiple workers writing to the same file concurrently, each of the 192 workers (equaling the number of available CPU cores) was given its own partition of the data, and given its own set of 64 buckets over which to distribute the examples in its partition. After all workers were finished with this distribution, the buckets were merged back into one set of 64 buckets by merging all buckets with the same index (all first buckets were merged, all second buckets were merged, etc.). This was the most important parallelization step, and drastically decreased the runtime of the shuffling algorithm. The within-bucket shuffles of the second step could also be parallelized, as multiple buckets could fit into DRAM concurrently, given the high number of buckets. The parallelization of this step used six workers such that six buckets were shuffled concurrently, which leaves some margin of DRAM for overhead.

4.4 Domain adaptation

The longest possible wallclock time that can be assigned to a job on Snellius is five days (120 hours). Therefore, the number of iterations and validations were chosen such that the estimated runtime of the training run would be about 110 hours, keeping some margin for error. For the long sequence training run, the number of training iterations was set to 5000, validating every 1000th iteration. For the short-sequence training run, the number of iterations was set to 20,000, also validating every thousand iterations. The validation set for both the long- and short-sequence runs comprised 10k examples of matching sequence lengths (i.e., 2048 tokens for short- and 8192 tokens for long-sequence training).

An important observation is that decreasing n_{max} by a factor of four also reduces the number of iterations per (roughly equal) runtime by the same factor. This suggests linear complexity with coefficient one, rather than the quadratic complexity known from the self-attention mechanism. This initially puzzling result is explained by FlashAttention–as discussed in Section 2.2.3–which reduces the bandwidth complexity of self-attention to linear. As GPUs are bandwidth- rather than computation-bound, this leads to linear runtime scaling despite the underlying computational complexity remaining quadratic. Because the total batch size is held constant for both long- and short-sequence training, the result is that both runs process roughly the same number of tokens per GPU hour–which is a valid measure of training efficiency. Thus, due to FlashAttention, short-sequence training shows virtually no gain in efficiency, though this did serve as the original motivation for exploring its performance effects compared to long-sequence training (RQ2).

4.4.1 Hyperparameters

For the long-sequence training run, the MSL (i.e., n_{max}) is set to 8192 tokens with a stride of 512 tokens; for the short-sequence training run these are both divided by four, so a MSL of 2048 and stride of 128 tokens are used. Due to pre-tokenization, these two parameter sets translated into two separate training datasets for the long- and short-sequence training runs.

For gradient descent, an adaptive moment with weight decay (AdamW) optimizer was used, for which the moments were set to: $\beta_1 = 0.9$ and $\beta_2 = 0.95$, and the weight decay was set to 0.01. These

are conventional hyperparameter values for LLM pretraining and finetuning (Brown et al., 2020; Touvron, Martin, et al., 2023); although β_2 is somewhat lower than the default value of 0.999 in the PyTorch implementation, which is a typical value in general machine learning, and a weight decay of 0.1 may also

The learning rate schedule has a peak learning rate of 2×10^{-4}, which is attained after a linear warm-up period over the first 5% of iterations, starting from 0. After the warm-up period, the learning rate schedule follows cosine decay to zero over the total number of iterations T, where $T = 5k$ for the long-sequence domain adaptation and $T = 20k$ for short-sequence domain adaptation. The employed learning rate schedule is illustrated in Figure 12. On top of the learning rate schedule, the Euclidean norm of the gradient is clipped at 1.0, which mitigates exploding gradients destabilizing the descent path.

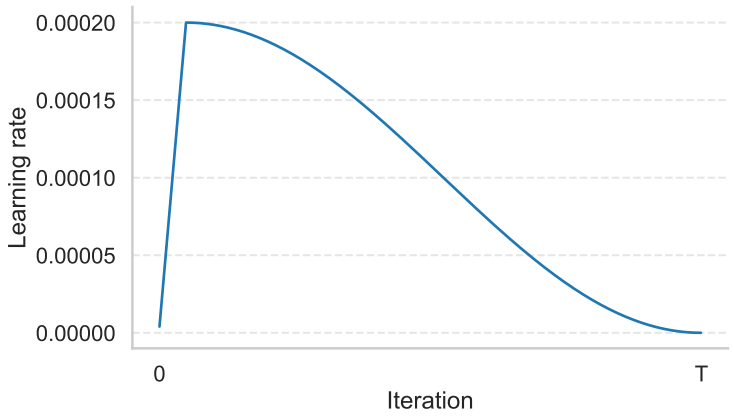


Figure 12

Learning rate schedule with cosine decay, linear warm-up period for the first 5% of training iterations, and a learning rate set to 2×10^{-4}. *T denotes the total number of training iterations, which is 20k for the short-sequence and 5k for the long-sequence training runs.*

To determine the total and per device batch sizes and gradient accumulation steps, the following method is employed: the desired total batch size is determined first, after which the batch size per device is maximized (limited by the GPU's VRAM capacity), and finally the gradient accumulation steps is set to match the desired total batch size. This maximizes GPU usage and minimizes the number of gradient accumulation steps, which scales one-to-one with computation time. The total batch size is chosen to be 64, because this is a somewhat large value compared to a batch size of 32–which is otherwise typically used in DL–while not being so large to necessitate extreme computation times for a perhaps overly cautious gradient descent path. The total batch size can also be expressed in the number of tokens processed for each batch. For the short-sequence training, the batch size is $64 \cdot 2048 = 0.131M$ tokens, and for the long-sequence it is $64 \cdot 8192 = 0.524M$ tokens. This is considerably lower than the pretraining batch size for Llama 3, at 4M and 8M tokens (set differently for different phases of their training run). This lower batch size is more appropriate to the significantly more constrained compute for this project than Llama 3's pretraining, but this deviation may suggest that an overly erratic descent path is used.

The (per device) batch size during training yields OOM errors when being set to 128 or larger. A batch size of 64 is feasible with the NVIDIA H100 GPU, but this yields no speed up compared to a

batch size of 32 with two gradient accumulation steps. In fact, the expected speed up of an increased batch size is also not visible for all increases above the increase from 4 to 8: doubling the batch size also yields a doubled computation time per iteration. Though this shouldn't be the case as the batch processed in parallel, this may indicate that HF's `transformers.Trainer` actually splits batches over multiple gradient accumulation steps under the hood if the batch does not fit in VRAM; though the OOM errors at a batch size of 128 suggests that this mechanism is either limited or that another issue is at play. Besides, if such a mechanism is employed, this invalidates the per device batch size and gradient accumulation steps that are reported during the training run, while no warning is given that this is the case; this would therefore obscure the training process and harm developer control.

Another such issue may be that the data pipeline forms the actual bottleneck. The data is streamed, meaning that one example is loaded at a time from the training data stored on disk[18] Streaming is necessary because the full data is too large to fit in RAM or VRAM, but streaming is necessarily sequential, and disk storage is much slower than RAM: typical DRAM (i.e., Dynamic RAM, the default type of CPU RAM) is usually a thousand times faster than SSD storage[19], which is the typical disk storage hardware. Whatever the reason, the total batch size of 64 is maintained, and a per device batch size of 32 is chosen with two gradient accumulation steps, even though runtime tests indicated negligible effect of different configurations of the latter two hyperparameters.

Surprisingly, prototyping revealed that the greatest feasible (per device) validation batch size*In the HF framework, the (per device) validation batch size is referred to as the evaluation batch size. That is slightly misleading, as validation and evaluation have distinct meanings in the machine learning literature, but this terminology would be especially confusing to use in this thesis because of the prominent features of both concepts. Hence, this batch size is called the validation batch size in this thesis.* before hitting OOM errors was 2, though validation should be much less memory intensive than training, as it does not require storage of gradients or intermediate activations. This can in part be explained due to the activation offloading to DRAM of Unsloth during training, which is not possible for validation as the activations do not need to be stored, but that cannot explain why validation is less memory intensive, only that the difference between memory footprints is smaller than it otherwise would be. A more probable cause, is that the hypothesized mechanism of balancing the per device batch size and gradient accumulation steps is indeed employed by the `transformers.Trainer()` function or its Unsloth patches, which is not possible to employ during validation as no gradients need to be accumulated. However, this would be a problematic mechanism, as discussed above, which makes it doubtful whether this mechanism is indeed employed. Possibly there are other training optimizations employed by Unsloth that reduce VRAM usage during training, but which are not present during evaluation, leading to a small maximum (per device) validation batch size of 2. A summary of the hyperparameters used in domain adaptation is given in Table 8.

This memory savings factor becomes apparent when comparing the training and validation batch sizes that are allowed by VRAM limitations. Gradient checkpointing is an optimization that is only used for calculating gradients, which is done during training but not during evaluation. While evaluation should be less memory intensive because there is no need for calculating gradients, the lack of these optimizations actually make evaluation far more memory intensive than training. While a

[18]Note that batched streaming is not supported by HF's `datasets` module. A custom batched streaming implementation appeared to require wrapping the `Trainer` function in a loop, which is not implemented as it risks clashes with either the standard HF or Unsloth patches of the `Trainer` class.

[19]To build intuition for this speed difference: $1000\times$ speed difference is approximately the difference between the typical movement speeds of a turtle and a fighter jet. This illustrates the importance of using working memory (RAM) instead of storage for the execution of any script where possible.

batch size of 8 can be used during training, during evaluation the greatest possible batch size, before the process gets OOM killed, is 2. This aligns with the reported four-fold memory savings factor.

Category	Hyperparameter	Value
Tokenization (long, short)	Max sequence length ($n_{\max}$)	8192, 2048
	Stride	512, 128
Gradient descent	Learning rate (η)	2e-4
	Warm-up ratio	0.01
	Weight decay (λ)	0.01
	Adam moments ($\beta_1,\ \beta_2$)	0.9, 0.95
	Gradient clipping	1.0
Batching	Total batch size	64
	Per device batch size	16
	Gradient accumulation	4
	Validation batch size	2

Table 8

Hyperparameters used for the domain adaptation training runs, grouped by categories denoting which aspect of the training process they govern. The long-sequence and short-sequence training runs only differ in tokenization-hyperparameters, and are denoted by long and short.

4.4.2 Unfreezing the embedding space

In early training runs, diagnostic output showed that only 93.46% of the parameters were adjustable, meaning that 6.54% of the model parameters were held frozen and therefore were not updated during training. This was puzzling, because after having disabled QLoRA for continued pretraining (as this was found to be suboptimal, as discussed in 2.4.2), no other PEFT method was employed that usually keep parameters frozen. Subsequent diagnostics revealed that the frozen parameters were that of the embedding matrix. This is definitely unwanted for domain adaptation, because the lexical meaning of tokens and words–being the type of information that embeddings are thought to encode–are expected to differ per domain as well, not only their contextual meaning. Therefore, the embedding parameters were manually unfrozen. Problematically, this drastically increased approximate average processing time per iteration: from 20 seconds per iteration with frozen embedding space to 75 seconds per iteration; a 3.75-factor increase. Because the truly limiting factor for the training process is compute budget, which is dependent on computation time, a 3.75 factor increase in computation time implies an equal factor reduction in number of iterations is needed to keep to required limits, and less data can be used for training by that same factor. Since the richness of data for this project is of key importance, this reduction in effective training data is highly undesirable. However, due to the theoretical importance of the embedding space to domain adaptation, keeping the embedding space frozen was deemed suboptimal. Therefore, the embedding space was unfrozen despite the $3.75\times$ reduction in effective training data[20].

4.5 Instruction tuning

The ideal data for instruction tuning in this project would be a purely financial QA dataset that cover a wide range of task categories (e.g., arithmetic reasoning, information extraction, summarization). However, no such ideal candidate was available. Despite promising a Financial Instruct Dataset,

[20]It remains unclear why the embedding space is frozen by default, but considering this massive slowdown it is probably a design choice aimed at avoiding exactly this. For task adaptation this is quite sensible, and as that is the more common method of finetuning LLMs, it is understandable to define this to be the default behavior, which was done by either HF or unsloth.

nicknamed *FIT* and detailing the dataset size and components, Q. Xie et al. (2023) do not actually provide this dataset (yet). A promising alternative would be the FinQA data, which is a dataset of roughly 8000 QA-pairs relating to information from earnings reports (Z. Chen et al., 2021). However, the FinQA data is already used in the five-shot evaluation based on the FLARE/PIXIU framework, so using this data for instruction tuning would invalidate evaluation of the transfer-learning capabilities of the instruction-tuned LLMs, as evaluation requires unseen data. The TAT-QA dataset (Zhu et al., 2021) poses a viable alternative. Though there is significant overlap with the FinQA data in terms of source material–most of the 10-Q and 10-K reports used in FinQA were also used in TAT-QA (Z. Chen et al., 2021)–there is no overlap in the QA pairs as these are independently authored. TAT-QA is designed to train models on arithmetic reasoning using tables, while FinQA is designed to train and test models on multi-step numerical reasoning capabilities. However, though desirable, arithmetic capabilities are not the main objective of instruction tuning; general instruction following and question answering is. The focus of TAT-QA on arithmetic reasoning may insufficiently cover general instruction tuning, which could preclude development of a user-friendly chatbot. To address this issue, the TAT-QA data is supplemented with data from Alpaca, which is a general instruction tuning dataset constructed by a team from Stanford University to tune Llama models specifically (Taori et al., 2023). Alpaca contains vastly more examples than TAT-QA, which risks drowning out the financial specialization of the TAT-QA data. Therefore, all TAT-QA examples are used, and an equal amount of Alpaca examples are added to both the train and evaluation splits of the TAT-QA data.

Instruction tuning is performed using three models: the base Llama 3 8B model (Vanilla), yielding model IT0; the long-sequence domain adapted model (DA1), yielding model IT1; and the short-sequence domain adapted model (DA2), yielding IT2. Because the instruction tuning data is much smaller than the domain adaptation data, it is feasible to finetune for multiple epochs, which is why instruction tuning was chosen to be performed with 10 epochs. Though this arguably high number of epochs risks overfitting, inspection of the validation loss after each epoch will allow for selection of the model state which minimizes validation loss, and is therefore overfitting the least.

For all instruction tuning, the same hyperparameters and learning rate schedule is used as for domain adaptation, shown in Table 8 and Figure 12, respectively. Because training runs for ten epochs, and each epoch takes 415 iterations, the total number of iterations for instruction tuning is $T = 4150$. Regarding tokenization, the QA sequences from the instruction tuning data are much shorter than the filings used for domain adaptation. To make sure no chunking was needed, which would have posed significant complications to instruction tuning–as now multiple chunks would have to predict the same outcome, even though each chunk would only contain a part of the information needed to derive the output–the maximum sequence length was chosen as the smallest power of two (for the sake of hardware compatibility) that is larger than any sequence. This led to selection of a MSL of 4096. Due to the smaller sequences, tokenization is much less resource intensive here and therefore proved no longer to be a bottleneck. This meant that it was no longer needed to pre-tokenize, and that tokenization could remain contained in the data pipeline during training. This did not harm GPU utilization, as can be seen from Figure H.4.

QLoRA was used for instruction tuning with the LoRA rank set to $r = 256$–relatively high as in accordance with D. Biderman et al. (2024), and with adapter matrices being applied to the following matrices: the query, key, value, and output projection matrices in the self-attention mechanism: $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, $\boldsymbol{W}_V$, $\boldsymbol{W}_O$; and the up-, down-, and gate-projection matrices in the MLP blocks: $\boldsymbol{U}$, $\boldsymbol{D}$, and $\boldsymbol{G}$. Note that the scaling factor α was set to $2r$, as was done in the original LoRA implementation.

4.6 Few-shot prompting evaluation

For evaluation of the financial specialization of the assessed models, the methodology of Y. Xie et al. (2024) is followed, which employs the Financial Language Research (FLARE) methodology, also dubbed PIXIU (The FinAI Team, 2024). This evaluation is performed through *five-shot prompting*, in which a model is first shown the instructions, followed by five train examples with accompanying answers, before the actual test example is shown for which it is to make its prediction. Note that this evaluation method formulates all prediction tasks as sequence-to-sequence (seq2seq) prediction tasks, which is in line with Raffel et al. (2020), as discussed in Section 2.4.3. By following their methodology, the results can be compared to their FinPythia 7B model, and to BloombergGPT (Wu et al., 2023), which has been evaluated on the same benchmarks, albeit with different train-test splits which therefore makes the evaluation of this model not perfectly comparable. The following four benchmarks of the FLARE/PIXIU framework, having been discussed in detail in Section 3.3, are employed for evaluation: FPB, FiQA SA, Headlines, and NER.

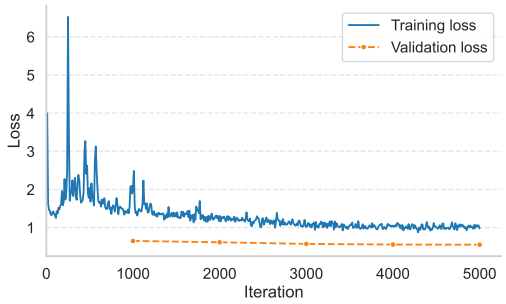
As these datasets will be used in evaluation through five-shot prompting, five train examples will be shown to the model with instructions, input and desired answer, before the model is asked to predict the answer to a test example. The train examples are randomly sampled from the data, but a fixed randomization seed implies that the exact same examples evaluation is performed as for other assessments using this framework, such as FinPythia 7B. Prompting evaluation does not require any parameter updates and therefore does not employ gradient descent, meaning that none of the hyperparameters used in domain adaptation or instruction tuning had to be set here. The only hyperparameters that are relevant for few-shot prompting, are the number of shots, being set to 5 for five-shot prompting, and temperature being set to zero to obtain deterministic results.

5 Results

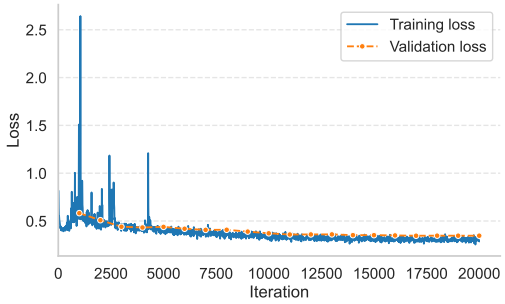
The results are presented in this section in three parts: domain adaptation, showing the continued pretraining results and diagnostics; (2) instruction tuning, showing the results of the supervised fine-tuning on the instruction-tune dataset; (3) few-shot prompting evaluation, containing the evaluation results on the financial NLP benchmarks.

5.1 Domain adaptation results

The long-sequence domain adaptation training process results, measured by training and validation loss development, can be seen from Figure 13a. The final computed training loss was 1.29, which started around 4 and appears to have reached convergence around the 3000-th iteration. Validation loss saw a modest but monotonic improvement over the entire training run, and was consistently lower than the training loss. The short-sequence domain adaptation training results can be seen from Figure 13b. The final computed training loss was 1.29, which started around 4 and appears to have reached convergence around the 3000-th iteration. Validation loss saw a modest but monotonic improvement over the entire training run.



(a) *Long-sequence domain-adaptation loss plot. The final training loss was 1.29 and is reported for every tenth iteration. Validation loss started at 0.65 and finished at 0.55, the latter being the best validation loss of the run.*



(b) *Short-sequence domain-adaptation loss plot. The final training loss was 0.37 and is reported for every tenth iteration. Validation loss started at 0.58 and finished at 0.3463. The best validation loss was found at iteration 17,000, with a validation loss of 0.3459.*

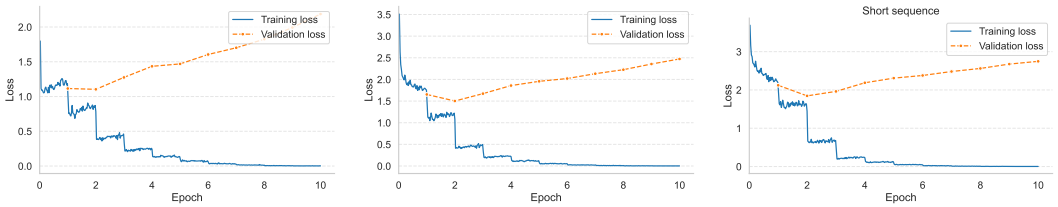
Figure 13

Training and validation cross-entropy losses for the two domain adaptation runs: (a) long-sequence ($n_{\max} = 8192$) and (b) short-sequence ($n_{\max} = 2048$). The validation loss is computed with a precision of seventeen decimals, and training loss is known to the fourth decimal. However, the reported loss values are reported at the level of precision that is deemed to best facilitate comparison.

The Euclidean (i.e., L2) norm of the gradients were found to frequently take on extreme values around 1000 early in the training runs of both the long and short sequence domain adaptation, which shows that the gradient clip at 1.0 played an important role in stabilizing the descent path. The GPU utilization of these training runs was found to be much better than the early prototype run before pre-tokenization was implemented. GPU utilization during the long-sequence training run was found to be above 95% for 94.37% and below 50% for 1.61% of observations. For the short-sequence training run, GPU utilization was above 95% for 87.16% and below 50% for 2.16% of observations. Plots detailing the gradient norms and GPU utilization can be found in Appendix H.

5.2 Instruction tuning results

Training and validation loss during instruction tuning of the base/vanilla, long-sequence domain-adapted, and short-sequence domain-adapted Llama 3.1 8B models are presented in Figure 14. It can be seen that the lowest validation loss was found after the second epoch of the training run. Hence, these states of the models were found to be optimal and were therefore used in subsequent analysis. The vanilla model was found to achieve the best training and validation loss, with the long-sequence domain-adapted model being the runner-up in terms of performance on these metrics, and the short-sequence domain-adapted model having been found to perform the worst of the analyzed configurations.



(a) *Base Llama 3.1 8B model. The final training loss was 0.29. The lowest validation loss was measured after the second epoch, at 1.10.*

(b) *Long-sequence domain-adapted model. The final training loss was 0.40. The lowest validation loss was measured after the second epoch, at 1.50.*

(c) *Short-sequence domain-adapted model. The final training loss was 0.52. The lowest validation loss was measured after the second epoch, at 1.85.*

Figure 14

Training and validation cross-entropy losses during instruction tuning: (a) base Llama 3.1 8B model, (b) long-sequence domain-adapted model, and (c) short-sequence domain-adapted model. Losses are reported for every tenth iteration.

GPU utilization was found to be similar to the utilization during the domain adaptation runs, despite domain adaptation using a streamed dataset which instruction tuning did not, indicating that streaming itself is not a bottleneck in the training process. Also as with domain adaptation, about 1% to 2.5% of observations saw a short and sharp drop in GPU utilization; it is unclear what causes these sudden and shortlived drops. The gradient norms were found not to reach values as extreme as with domain adaptation, but gradient clipping nevertheless appeared to be an important regularizer for the training process. Plots with gradient norms and GPU utilization during the instruction tuning runs can be found in Appendix H.

5.3 Few-shot prompting evaluation results

Evaluation results are shown in Table 9. Performance is found to vary notably by task and training variant across the benchmark tasks. The BloombergGPT and FinPythia 7B models significantly outperform the models the assessed models in this research. Only on the Headlines task, the assessed models exceed FinPythia (0.5414) and tie at 0.5995, but thereby also show no performance improvement from domain adaptation or instruction tuning. FiQA also shows no change with domain adaptation (DA1, DA2: Acc 0.3234; F1 0.1581), a slight drop with instruction tuning of the base model (IT0: Acc 0.3191; F1 0.1565), and a return to baseline when combined with domain adaptation (IT1, IT2: Acc 0.3234; F1 0.1581). FPB exhibits small, consistent gains from short-sequence adaptation: DA2 raises Acc to 0.2866 and F1 to 0.1291 over the vanilla 0.2856/0.1269, and IT2 further improves to Acc 0.2938 and F1 0.1478. For NER, a clear ranking of the assessed models emerges: IT0 (0.4195) > DA2 (0.3923) > DA1 (0.3641) > IT1 (0.3136) > Vanilla (0.3007) > IT2 (0.1794); though BloombergGPT and FinPythia remains unrivaled. Where the scores of DA1 and DA2 differ, the short-sequence model's score exceeds that of the long-sequence model.

Table 9

Five-shot prompting evaluation results on four financial NLP benchmarks. Vanilla is the base Llama 3.1 8B model; DA1 and DA2 are domain-adapted models trained on long- and short-sequence financial text, respectively; IT0 is the instruction-tuned base model; IT1 and IT2 are instruction-tuned long- and short-sequence domain-adapted models, respectively; results for the BloombergGPT and FinPythia 7B models are taken from Y. Xie et al. (2024). Different train-test splits were used for BloombergGPT. Bold values indicates the best score for the metric out of the assessed models, and underlined values indicate the second-best values. Metrics: Acc – classification accuracy; F1 – F1-score; Each model's output was made deterministic by setting the temperature parameter to 0.

Task	Metric	Model							
		Vanilla	DA1	DA2	IT0	IT1	IT2	BloombergGPT	FinPythia 7B
FPB	Acc	0.2856	0.2856	0.2866	0.2856	0.2856	0.2938	–	**0.5990**
	F1	0.1269	0.1269	0.1291	0.1269	0.1269	0.1478	0.5107	**0.6443**
FiQA	Acc	0.3234	0.3234	0.3234	0.3191	0.3234	0.3234	–	**0.5234**
	F1	0.1581	0.1581	0.1581	0.1565	0.1581	0.1581	**0.7507**	0.5304
Headlines	F1	0.5995	0.5995	0.5995	0.5995	0.5995	0.5995	**0.8220**	0.5414
NER	F1	0.3007	0.3641	0.3923	0.4195	0.3136	0.1794	**0.6082**	0.4842

6 Discussion

Several insights about domain adaptation for financial LLMs were revealed by this study. Most notably, adaptation benefits were found to be highly task-dependent rather than universal. Structured prediction tasks like NER benefit substantially from domain knowledge (21-30% relative improvements), while classification tasks remain largely unaffected. This pattern suggests that domain adaptation primarily enhances fine-grained linguistic understanding–recognizing financial entities, parsing technical terminology–rather than high-level semantic reasoning. For practitioners, this implies that intended applications should be carefully considered before committing to investments in domain adaptation.

The unexpected superiority of short-sequence training ($n_{\text{max}} = 2048$) over long-sequence training ($n_{\text{max}} = 8192$) challenges assumptions about context window benefits and their empirical evidence found by W. Xiong et al. (2023). While longer contexts theoretically enable better document understanding, the results suggest that focused learning on shorter segments may be more effective, at least within computational constraints. However, this finding requires careful interpretation: the evaluation benchmarks use relatively short sequences and thus cannot assess long-range dependency modeling, which is crucial for analyzing lengthy regulatory filings. Crucially, as discussed in Section 4.4, the presumed computational efficiency–measured by tokens processed per GPU hour–of short-sequence training did not materialize, despite being the main motivation for investigating long- versus short-sequence training (RQ2). As explained in Section 2.2.3, this is a consequence of FlashAttention, which reduces the bandwidth complexity of attention to linear. Since GPUs are bandwidth-bound, runtime therefore scales linearly with sequence length, even though the underlying computational complexity remains quadratic.

Instruction tuning presents a nuanced picture. Its dramatic improvement on NER when applied to the base model demonstrates the value of instruction tuning. Yet the absence of synergy with domain adaptation–and, in the case of IT2, a marked degradation on NER relative to both DA variants and even the Vanilla model–suggests that stacking adaptation strategies may suffer from interference or catastrophic forgetting. At the same time, instruction tuning can offer modest gains on some classification tasks (e.g., IT2 on FPB), while leaving others largely unchanged (FiQA, Headlines).

The performance gap observed relative to BloombergGPT and FinPythia 7B likely stems from multiple factors. The most significant is the vast difference in computational resources: the 110 GPU hours utilized represent less than 4% of FinPythia's training time and under 0.02% of BloombergGPT's estimated compute. The lower performance compared to these models is therefore perfectly explained with established scaling laws.

Additionally, the batch sizes used (0.131M-0.524M tokens) fall far below those used in modern LLM training (4–8M tokens for Llama 3). Larger batch sizes enable better gradient estimation and more stable optimization dynamics, potentially explaining some performance limitations. The quality and diversity of training data also differ: while this study focused exclusively on SEC filings for reproducibility, both baseline models incorporated diverse financial sources including news articles, reports, and analyses. This diversity matters because different sources capture complementary aspects of financial knowledge–news articles convey market sentiment and real-time developments, analyst reports provide forward-looking insights and valuation perspectives, while SEC filings primarily document historical performance and regulatory compliance. The legalistic nature of SEC filings may prioritize compliance language over financial insight, which could explain the performance gain on NER (identifying regulated entities and terms) while not affecting classification tasks requiring market understanding.

6.1 Limitations and contributions

This study faces several methodological constraints that may have impacted results. Data preprocessing challenges proved pervasive: inconsistent handling of nested XML/HTML structures and residual XBRL formatting likely introduced substantial noise. The complexity of SEC filing formats–mixing human-readable text with machine-readable markup–requires more sophisticated parsing strategies than employed here; future research is advised to use Arelle for parsing XBRL, which might also support parsing of HTML and XML.

Evaluation limitations also constrain the interpretation of results. The benchmarks employed consist primarily of short-context tasks, preventing assessment of long-range dependency modeling that motivated the long-sequence training experiments. The relatively small number of evaluation tasks (four benchmarks) limits the generalizability of findings, and the absence of generation-based or long-form reasoning tasks may not fully capture the benefits of both domain adaptation and, particularly, instruction tuning.

Despite these limitations, this study makes several important contributions to understanding financial LLM adaptation. It provides the first systematic comparison of sequence length effects in financial domain adaptation, demonstrating that shorter contexts can be more effective within computational constraints. The identification of task-dependent benefits–with structured prediction tasks showing substantial gains while classification tasks remain unaffected–offers practical guidance for resource allocation. Additionally, the detailed analysis of SEC filing preprocessing challenges and the unexpected lack of synergy between domain adaptation and instruction tuning provide valuable insights for future research in specialized LLM development. These findings underscore that effective financial NLP requires careful alignment between adaptation strategies, computational resources, and specific task requirements.

7 Conclusion

This study investigated domain adaptation strategies for financial LLMs through three research questions, revealing nuanced patterns in how specialization affects model performance.

RQ1: Domain adaptation effectiveness. Domain adaptation demonstrated clear task-type dependencies rather than universal benefits. Structured prediction tasks, particularly named entity recognition, showed substantial improvements from domain adaptation, though the strongest NER result came from instruction tuning the base model. Classification tasks remained largely unaffected, with the exception of small, consistent gains on FPB from short-sequence adaptation and IT2. This pattern suggests that domain adaptation primarily enhances fine-grained linguistic understanding–recognizing financial entities and parsing technical terminology–rather than high-level semantic reasoning such as sentiment classification. The legalistic nature of SEC filings may have contributed to this effect, improving regulatory term recognition without enhancing broader financial understanding.

RQ2: Sequence length optimization. Contrary to expectations and previous empirical evidence, short-sequence training ($n_{\text{max}} = 2048$) consistently outperformed long-sequence training ($n_{\text{max}} = 8192$), with a single exception being the NER performance of IT2. This advantage did not stem from computational efficiency, which was equal across settings due to FlashAttention, but likely from optimization dynamics on short-context benchmarks. Given the evaluation benchmarks' limitation to short-context tasks, potential benefits of long-sequence training for extensive regulatory documents remain untested.

RQ3: Instruction tuning synergies. Instruction tuning exhibited complex, task-dependent effects without the expected synergies with domain adaptation. While base model instruction tuning greatly improved NER performance, combining it with domain-adapted models showed no additional benefits and, for the short-sequence variant (IT2), a marked degradation on NER, alongside modest gains on FPB and little change on FiQA and Headlines. This suggests interference between sequential adaptation strategies, challenging assumptions about additive improvements from stacked techniques.

These findings must be interpreted within the context of significant resource constraints–110 GPU hours versus tens of thousands to over one million hours for the benchmark models–and methodological limitations including data preprocessing challenges and evaluation gaps. Nevertheless, this study provides valuable contributions: the first systematic comparison of sequence lengths for financial domain adaptation, empirical evidence of task-dependent specialization benefits, and identification of critical preprocessing requirements for SEC filings.

The results underscore that effective financial NLP requires careful alignment between adaptation strategies and intended applications. Organizations should prioritize domain adaptation for information extraction and entity recognition tasks while recognizing its limited value for (sentiment) classification problems. Future research should address the computational and data diversity gaps identified here, employ comprehensive evaluation frameworks including long-context assessments, and investigate whether richer training corpora combining regulatory filings with market-oriented texts can achieve more balanced improvements across task types.

References

(2007). *The New York Times Archive*. https : / / archive . nytimes . com / www . nytimes . com / ref / membercenter/nytarchive.html

(2024). *New York Times - Terms of service*. https : / / help . nytimes . com / hc / en - us / articles / 115014893428-Terms-of-Service

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., … Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

Ainslie, J., Ontanon, S., Alberti, C., Cvicek, V., Fisher, Z., Pham, P., Ravula, A., Sanghai, S. N., Wang, L., & Yang, Y.-T. (2023). Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. https://arxiv.org/abs/2305.13245

Alvarado, J. C. S., Verspoor, K., & Baldwin, T. (2015). Domain adaption of named entity recognition to support credit risk assessment. *Proceedings of the australasian language technology association workshop 2015*, 84–90.

Ameisen, E., Lindsey, J., Pearce, A., Gurnee, W., Turner, N. L., Batson, J., Chen, B., Citro, C., Abrahams, D., Carter, S., et al. (2025). On the biology of a large language model [Section "Addition" in attribution-graphs/biology]. *Transformer Circuits Thread (online)*. https://transformer-circuits.pub/2025/attribution-graphs/biology.html

Arelle Project. (2010). Arelle: Open-source xbrl platform ["Arelle was created in 2010 to improve the accessibility and usability of XBRL™ … the world's only free and open source XBRL platform."].

Aschenbrenner, L. (2024). Situational Awareness: The Decade Ahead. https://situational-awareness.ai/

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

BelladoreAI. (2024). Llama tokenizer demos [Accessed: 2025-04-01].

Bender, E. M., Gebru, T., McMillan-Major, A., & Shmitchell, S. (2021). On the dangers of stochastic parrots: Can language models be too big? *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, 610–623.

Biderman, D., Portes, J., Ortiz, J. J. G., Paul, M., Greengard, P., Jennings, C., King, D., Havens, S., Chiley, V., Frankle, J., et al. (2024). Lora learns less and forgets less. *Transactions on Machine Learning Research*.

Biderman, S., Schoelkopf, H., Anthony, Q. G., Bradley, H., O'Brien, K., Hallahan, E., Khan, M. A., Purohit, S., Prashanth, U. S., Raff, E., et al. (2023). Pythia: A suite for analyzing large language models across training and scaling. *Proceedings of the 40th International Conference on Machine Learning (ICML)*, 2397–2430. https://proceedings.mlr.press/v202/biderman23a/biderman23a.pdf

Blum, A. L., & Rivest, R. L. (1992). Training a 3-node neural network is np-complete. *Neural Networks*, *5*(1), 117–127. https://doi.org/10.1016/0893-6080(92)90011-O

Bowman, S. R. (2023). Eight things to know about large language models. *arXiv preprint arXiv:2304.00612*. https://arxiv.org/abs/2304.00612

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, *33*, 1877–1901.

Brysbaert, M. (2019). How many words do we read per minute? a review and meta-analysis of reading rate. *Journal of Memory and Language*.

Bylinina, L., & Nouwen, R. (2020). Numeral semantics. *Language and Linguistics Compass*, *14*(8), e12390.

Chan, S., Santoro, A., Lampinen, A., Wang, J., Singh, A., Richemond, P., McClelland, J., & Hill, F. (2022). Data distributional properties drive emergent in-context learning in transformers. *Advances in neural information processing systems*, *35*, 18878–18891.

Chen, T., Xu, B., Zhang, C., & Guestrin, C. (2016). Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*.

Chen, Z., Chen, W., Smiley, C., Shah, S., Borova, I., Langdon, D., Moussa, R., Beane, M., Huang, T.-H., Routledge, B., & Wang, W. Y. (2021). Finqa: A dataset of numerical reasoning over financial data. *Proceedings of EMNLP 2021*.

Chow, A. R., & Perrigo, B. (2025). Is the DeepSeek Panic Overblown? *Time*. https://time.com/7211646/is-deepseek-panic-overblown/

Cloud, H. (2024). *What is meta llama 3.3 70b? features, use cases & more* [Accessed: 2025-09-04]. https://www.hyperstack.cloud/blog/thought-leadership/what-is-meta-llama-3-3-70b-features-use-cases-more

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, *2*(4), 303–314. https://doi.org/10.1007/BF02551274

Daniel Han, M. H., & team, U. (2025). *Unsloth*. http://github.com/unslothai/unsloth

Dao, T., Fu, D., Ermon, S., Rudra, A., & Ré, C. (2022). Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, *35*, 16344–16359.

Dauphin, Y. N., Fan, A., Auli, M., & Grangier, D. (2017). Language modeling with gated convolutional networks. *International conference on machine learning*, 933–941.

Dave, P., & Dastin, J. (2021). Google fires second AI ethics leader as dispute over research, diversity grows. *Reuters*. https://www.reuters.com/article/business/google-fires-second-ai-ethics-leader-as-dispute-over-research-diversity-grows-idUSKBN2AM1T1/

DeepLearning Hero. (2024). Rotary positional encoding (rope) explained [Accessed: 2025-07-08]. https://www.youtube.com/watch?v=GQPOtyITy54

Dettmers, T., Pagnoni, A., Holtzman, A., & Zettlemoyer, L. (2023). Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems*, *36*, 10088–10115.

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Disparate AI. (2024, May). *Not all tokens are created equal: A practical analysis of the tokenizers for different llms* [Medium post]. https://medium.com/@disparate-ai/not-all-tokens-are-created-equal-7347d549af4d

Fisher, R. A., & Yates, F. (1938). *Statistical tables for biological, agricultural and medical research* (1st ed.). Oliver; Boyd.

Global Book Statistics (Sivo blog). (2025). How Many Books Are There in the World? [Estimate of 158,464,880 unique books as of 2023]. https://blog.sivo.it.com/global-book-statistics/how-many-books-are-there-in-the-world/

Goodfellow, I., Bengio, Y., & Courville, A. (2016a). Back-propagation and other differentiation algorithms [Available online at http://www.deeplearningbook.org/contents/mlp.html]. In *Deep learning*. MIT Press.

Goodfellow, I., Bengio, Y., & Courville, A. (2016b). Sequence modeling: Recurrent and recursive nets [Chapter 10]. In *Deep learning* (pp. 373–423). MIT Press. http://www.deeplearningbook.org

Grant, N. (2024). Google's A.I. search errors cause a furor online. *The New York Times*. https://www.nytimes.com/2024/05/24/technology/google-ai-overview-search.html

Griewank, A., & Walther, A. (2000). Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software (TOMS)*, *26*(1), 19–45.

Gururangan, S., Marasović, A., Swayamdipta, S., Lo, K., Beltagy, I., Downey, D., & Smith, N. A. (2020). Don't stop pretraining: Adapt language models to domains and tasks. *arXiv preprint arXiv:2004.10964*.

Han, D. (2023, May). "my take on 'lora learns less and forgets less' …" [Accessed: 2025-06-26]. https://x.com/danielhanchen/status/1791900967472140583

Han, D., & Han, M. (2024, June). Continued Pretraining with Unsloth [Accessed: 2025-03-12]. https://unsloth.ai/blog/contpretraining

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.

Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 328–339. https://doi.org/10.18653/v1/P18-1031

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W., et al. (2022). Lora: Low-rank adaptation of large language models. *ICLR*, *1*(2), 3.

Hu, K. (2023). ChatGPT sets record for fastest-growing user base—analyst note. *Reuters*.

Hugging Face. (2025a). Fast vs. slow tokenizers in hugging face transformers [Accessed: 2025-06-30]. https://huggingface.co/docs/transformers/main_classes/tokenizer

Hugging Face. (2025b). Llama 3.1 tokenizer architecture [Accessed: 2025-06-30]. https://huggingface.co/docs/transformers/en/model_doc/llama3

ICE-PIXIU. (2024). Fiqasa: Financial sentiment analysis dataset on HuggingFace. https://huggingface.co/datasets/TheFinAI/flare-fiqasa/

Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International conference on machine learning*, 448–456.

Jin, H., Han, X., Yang, J., Jiang, Z., Chang, C.-Y., & Hu, X. (2023). Growlength: Accelerating llms pretraining by progressively growing training length. *arXiv preprint arXiv:2310.00576*.

Jindal, I., Badrinath, C., Bharti, P., Vinay, L., & Sharma, S. D. (2024). Balancing continuous pre-training and instruction fine-tuning: Optimizing instruction-following in llms. *arXiv preprint arXiv:2410.10739*. https://arxiv.org/html/2410.10739v1

Kalajdzievski, D. (2023). A rank stabilization scaling factor for fine-tuning with lora. *arXiv preprint arXiv:2312.03732*.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, *114*(13), 3521–3526. https://doi.org/10.1073/pnas.1611835114

Lee, S. (2024). Finale: Finance domain instruction-tuning dataset with rationales. *Proceedings of the 2nd Workshop on Financial NLP*. https://aclanthology.org/2024.finnlp-2.9.pdf

Lin, S. C., Tian, F., Wang, K., Zhao, X., Huang, J., Xie, Q., Borella, L., White, M., Wang, C. D., Xiao, K., et al. (2025). Open FinLLM leaderboard: Towards financial AI readiness. *arXiv preprint arXiv:2501.10963*.

Ling, C., Zhao, X., Lu, J., Deng, C., Zheng, C., Wang, J., Chowdhury, T., Li, Y., Cui, H., Zhang, X., et al. (2023). Domain specialization as the key to make large language models disruptive: A comprehensive survey. *arXiv preprint arXiv:2305.18703*.

Liu, A., Feng, B., Wang, B., Wang, B., Liu, B., Zhao, C., Dengr, C., Ruan, C., Dai, D., Guo, D., et al. (2024). Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*.

Liu, Y. (2019). Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, *364*.

London Stock Exchange Group. (n.d.). Financial News Coverage. https://www.lseg.com/en/data-analytics/financial-data/financial-news-coverage

Maia, M., Handschuh, S., Freitas, A., Davis, B., McDermott, R., Zarrouk, M., & Balahur, A. (2018). Www'18 open challenge: Financial opinion mining and question answering. *Companion proceedings of the the web conference 2018*, 1941–1942.

Malo, P., Sinha, A., Korhonen, P., Wallenius, J., & Takala, P. (2014). Good debt or bad debt: Detecting semantic orientations in economic texts. *Journal of the Association for Information Science and Technology*, *65*(4), 782–796.

McCloskey, M., & Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. *Proceedings of the 11th Annual Conference of the Cognitive Science Society*, 651–656.

McCoy, R. T., Yao, S., Friedman, D., Hardy, M., & Griffiths, T. L. (2023). Embers of autoregression: Understanding large language models through the problem they are trained to solve. *arXiv preprint arXiv:2309.13638*.

Merolla, P., Appuswamy, R., Arthur, J., Esser, S. K., & Modha, D. (2016). Deep neural networks are robust to weight binarization and other non-linear distortions. *arXiv preprint arXiv:1606.01981*.

Meta. (2024a). Introducing meta llama 3: The most capable openly available llm to date. *2*(5), 6. https://ai.meta.com/blog/meta-llama-3

Meta. (2024b). Meta llama 3.1 8b-instruct. https://huggingface.co/meta-llama/Llama-3.1-8B-Instruct

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Nature News. (2025). AI trained on decades of financial filings predicts companies' fortunes. *Nature*, *629*, 837. https://doi.org/10.1038/d41586-025-01125-9

Nielsen, M. A., & Chuang, I. L. (2010). *Quantum computation and quantum information*. Cambridge university press.

NVIDIA Corporation. (2022). NVIDIA H100 Tensor Core GPU Datasheet [Accessed: 2025-06-26]. https://resources.nvidia.com/en-us-hopper-architecture/nvidia-tensor-core-gpu-datasheet?ncid=no-ncid

NVIDIA Corporation. (2025). *Cuda toolkit documentation*. https://developer.nvidia.com/cuda-toolkit

Parthasarathy, V. B., Zafar, A., Khan, A., & Shahid, A. (2024). The ultimate guide to fine-tuning llms from basics to breakthroughs: An exhaustive review of technologies, research, best practices, applied research challenges and opportunities. *arXiv preprint arXiv:2408.13296*.

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In S. Dasgupta & D. McAllester (Eds.), *Proceedings of the 30th international conference on machine learning* (pp. 1310–1318, Vol. 28). PMLR. https://proceedings.mlr.press/v28/pascanu13.html

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., … Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems (NeurIPS) 32*. https://papers.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. https://arxiv.org/abs/1910.10683

Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Searching for activation functions. *arXiv preprint arXiv:1710.05941*.

Randewich, N. (2025). NVIDIA's value tops $4 trillion, cementing its status as the world's most valuable company. *Reuters*. https://www.reuters.com/business/nvidia-notches-4-trillion-record-close-market-cap-2025-07-10/

Raposo, G., Tomás, P., & Roma, N. (2021). Positnn: Training deep neural networks with mixed low-precision posit. *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 7908–7912.

Richardson, L. (2024). *Beautiful soup* [Version 4.12.3, Python library for pulling data out of HTML and XML files]. https://www.crummy.com/software/BeautifulSoup/

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, *323*(6088), 533–536.

Sahraei, A., Chamorro, A., Kraft, P., & Breuer, L. (2021). Application of machine learning models to predict maximum event water fractions in streamflow. *Frontiers in Water*, *3*, 652100.

Securities, U., & Commission, E. (n.d.). About EDGAR. *U.S. Securities and Exchange Commission*. https://www.sec.gov/submit-filings/about-edgar

Selenium Project. (2024). Selenium: Python bindings for browser automation [Version 4.15.2].

Shazeer, N. (2020). Glu variants improve transformer. https://arxiv.org/abs/2002.05202

Sinha, A., & Khandait, T. (2021). Impact of news on the commodity market: Dataset and results. *Future of Information and Communication Conference*, 589–601.

Sinha, K., Kazemnejad, A., Reddy, S., Pineau, J., Hupkes, D., & Williams, A. (2022). The curious case of absolute position embeddings. *arXiv preprint arXiv:2210.12574*.

Sor, J. (2025). Nvidia's AI-powered rise to a 4 trillion market cap, in 3 charts. *Business Insider*. https://www.businessinsider.com/nvidia-stock-price-nvda-history-4-trillion-market-cap-chatgpt-2025-7

Stack Overflow Community. (2018). Complexity of f.seek() in python. https://stackoverflow.com/questions/51801213/complexity-of-f-seek-in-python

Stack Overflow Community. (2023, July). *How does one set the pad token correctly (not to eos) during fine-tuning to avoid model not predicting eos?* https://stackoverflow.com/questions/

76633368/how-does-one-set-the-pad-token-correctly-not-to-eos-during-fine-tuning-to-avoi

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2023). Roformer: Enhanced transformer with rotary position embedding. https://arxiv.org/abs/2104.09864

Sun, W., Song, X., Li, P., Yin, L., Zheng, Y., & Liu, S. (2025). The curse of depth in large language models. *arXiv preprint arXiv:2502.05795*.

SURF Foundation. (n.d.). Best practice for data formats in deep learning. *SURF User Knowledge Base*. https://servicedesk.surf.nl/wiki/display/WIKI/Best+Practice+for+Data+Formats+in+Deep+Learning

SURF Foundation. (2025a). Snellius hardware [Accessed: 2025-06-26]. https://servicedesk.surf.nl/wiki/spaces/WIKI/pages/30660208/Snellius+hardware

SURF Foundation. (2025b). Snellius partitions and accounting [Accessed: 2025-07-08]. https://servicedesk.surf.nl/wiki/spaces/WIKI/pages/30660209/Snellius+partitions+and+accounting

Tang, Y., & Yang, Y. (2024). Do we need domain-specific embedding models? an empirical investigation. *arXiv preprint arXiv:2409.18511*.

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., & Hashimoto, T. B. (2023). Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca

Thawani, A., Pujara, J., Szekely, P. A., & Ilievski, F. (2021). Representing numbers in nlp: A survey and a vision. *arXiv preprint arXiv:2103.13136*.

The FinAI. (2024a). en-fpb: English Financial Phrase Bank sentiment dataset. https://huggingface.co/datasets/TheFinAI/en-fpb

The FinAI. (2024b). FLARE-Headlines: Financial LLM classification dataset for headline-level tasks. https://huggingface.co/datasets/TheFinAI/flare-headlines

The FinAI. (2024c). FLARE-NER: Named Entity Recognition dataset from TheFinAI for financial agreements. https://huggingface.co/datasets/TheFinAI/flare-ner

The FinAI Team. (2024). Pixiu: A framework for finetuning and evaluating large language models on financial tasks [Accessed: 2025-06-30]. https://github.com/The-FinAI/PIXIU

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., & Lample, G. (2023). Llama: Open and efficient foundation language models. https://arxiv.org/abs/2302.13971

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Boureau, Y.-L., Chaudhary, V., Lample, G., & Fan, A. (2023). Llama 2: Open foundation and fine-tuned chat models.

Unsloth gradient checkpointing - 4x longer context windows. (2024). *Unsloth*. https://unsloth.ai/blog/long-context

Uppaal, R. (2024). How useful is continued pre-training for generative language models? *Proceedings of the 1st Workshop on Reproducibility in NLP*. https://aclanthology.org/2024.repl4nlp-1.9.pdf

U.S. Securities and Exchange Commission. (2019). Compliance & disclosure interpretations – interactive data (inline xbrl). https://www.sec.gov/divisions/corpfin/guidance/interactivedatainterp

U.S. Securities and Exchange Commission. (2024). EDGAR: Electronic Data Gathering, Analysis, and Retrieval system [Data retrieved via the EDGAR RSS feed at https://www.sec.gov/Archives/edgar/Feed/]. https://www.sec.gov/edgar.shtml

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/1706.03762

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., & Le, Q. V. (2022). Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*. https://arxiv.org/abs/2109.01652

Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al. (2022). Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., … Rush, A. M. (2020). Transformers: State-of-the-art natural language processing. *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 38–45. https://aclanthology.org/2020.emnlp-demos.6

Wortsman, M., Dettmers, T., Zettlemoyer, L., Morcos, A., Farhadi, A., & Schmidt, L. (2023). Stable and low-precision training for large-scale vision-language models. *Advances in Neural Information Processing Systems*, *36*, 10271–10298.

WSJ terms of use. (2025). https://www.dowjones.com/terms-of-use/

Wu, S., Irsoy, O., Lu, S., Dabravolski, V., Dredze, M., Gehrmann, S., Kambadur, P., Rosenberg, D., & Mann, G. (2023). Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564*.

Xie, Q., Han, W., Chen, Z., Xiang, R., Zhang, X., He, Y., Xiao, M., Li, D., Dai, Y., Feng, D., Xu, Y., Kang, H., Kuang, Z., Yuan, C., Yang, K., Luo, Z., Zhang, T., Liu, Z., Xiong, G., … Huang, J. (2024). Finben: A holistic financial benchmark for large language models. https://arxiv.org/abs/2402.12659

Xie, Q., Han, W., Zhang, X., Lai, Y., Peng, M., Lopez-Lira, A., & Huang, J. (2023). Pixiu: A large language model, instruction data and evaluation benchmark for finance. *arXiv preprint arXiv:2306.05443*.

Xie, Y., Aggarwal, K., & Ahmad, A. (2024). Efficient continual pre-training for building domain specific large language models. *Findings of the Association for Computational Linguistics ACL 2024*, 10184–10201.

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., & Liu, T.-Y. (2020). On layer normalization in the transformer architecture. *International Conference on Machine Learning*, 10524–10533.

Xiong, W., Liu, J., Molybog, I., Zhang, H., Bhargava, P., Hou, R., Martin, L., Rungta, R., Sankararaman, K. A., Oguz, B., Khabsa, M., Fang, H., Mehdad, Y., Narang, S., Malik, K., Fan, A., Bhosale, S., Edunov, S., Lewis, M., … Ma, H. (2023). Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*. https://arxiv.org/abs/2309.16039

Yang, H., Liu, X.-Y., & Wang, C. D. (2023). Fingpt: Open-source financial large language models. https://arxiv.org/abs/2306.06031

Yang, J., Chen, Y., Li, D., Zhang, F., Meng, Z., Guo, B., Ren, J., Yu, T., Jia, X., Chen, M., et al. (2018). Bfloat16: The secret to high performance on cloud tpus [Google Research Blog: https://cloud.google.com/blog/topics/inside-google-cloud/bfloat16-data-type-brings-accelerated-training-to-tpus]. *Proceedings of the 35th International Conference on Machine Learning (ICML 2018) Workshop*.

Yang, Y., UY, M. C. S., & Huang, A. (2020). Finbert: A pretrained language model for financial communications. https://arxiv.org/abs/2006.08097

Yoo, A. B., Jette, M. A., & Grondona, M. (2003). SLURM: Simple linux utility for resource management. In D. G. Feitelson, L. Rudolph, & U. Schwiegelshohn (Eds.), *Job scheduling strategies for parallel processing (jsspp)* (pp. 44–60, Vol. 2862). Springer. https://doi.org/10.1007/10968987_3

Zhang, B., & Sennrich, R. (2019). Root mean square layer normalization. *Advances in neural information processing systems*, *32*.

Zhang, H., Dong, Y., Xiao, C., & Oyamada, M. (2023). Large language models as data preprocessors. *arXiv preprint arXiv:2308.16361*.

Zhang, H., Duckworth, D., Ippolito, D., & Neelakantan, A. (2020). Trading off diversity and quality in natural language generation. *arXiv preprint arXiv:2004.10450*.

Zhu, F., Lei, W., Huang, Y., Wang, C., Zhang, S., Lv, J., Feng, F., & Chua, T.-S. (2021). TAT-QA: A question answering benchmark on a hybrid of tabular and textual content in finance. *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 3277–3287. https://doi.org/10.18653/v1/2021.acl-long.254

Appendices

A Introduction to deep learning

This section aims to provide an introduction to concepts in deep learning (DL) that are required background knowledge for the research presented in this paper. Linear regression is assumed to be prior knowledge, which serves as the starting point for the introduction.

A.1 Simple learning: linear and (multinomial) logistic regression

Linear regression serves to find a linear relation between the input data $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ and the target (or output) variable $\boldsymbol{y} \in \mathbb{R}^n$, with the model specification being $\boldsymbol{y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$ in matrix notation, with the $\boldsymbol{\beta} \in \mathbb{R}^p$ vector including both the intercept as β_0 and coefficients, denoting the linear relation. The $\boldsymbol{\beta}$ vector is found by minimizing errors $\boldsymbol{\epsilon} = \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}$, also called *residuals*. To avoid errors canceling out, instead of minimizing the sum of errors directly, the residual sum of squares (RSS) (i.e., sum of squared errors) is minimized: $RSS = (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^\top (\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})$. The RSS is said to be the *loss function*, as it represents the loss of accuracy through the model's abstraction, which is always meant to be minimized. Minimization of the RSS knows a *closed-form solution*: $\hat{\boldsymbol{\beta}} = \arg\min_{\boldsymbol{\beta}}(\text{RSS}) = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$, as the solution can be derived exactly. This means that no iterative procedure is needed to fit the model to the data, as we already exactly know the optimal values for $\boldsymbol{\beta}$ *a priori*. Here it is important to note that *fitting is learning*, so this can already be considered a simple form of (machine) learning.

Optimization becomes more complex when we move on to logistic regression. The goal here is not to fit a continuous numerical variable, but rather a binary outcome (i.e., yes or no, true or false, 1 or 0), used for binary classification. To do so, the outcome $\boldsymbol{y} \in \{0, 1\}^n$ itself is not predicted, but rather the probability of the outcome $\hat{p} = \mathbb{P}[y = 1] \in [0, 1]$. As with linear regression, a linear relation is assumed, but now between the input data $\boldsymbol{X}$ and a latent variable, the *logits* $\boldsymbol{z} \in \mathbb{R}^n$, which provide a measure of the likelihood of the observations belonging to the target class. The input data is linearly transformed to the logits: $\boldsymbol{z} = \boldsymbol{X}\boldsymbol{\beta}$, and the logits are transformed to actual probabilities using the *sigmoid function*: $\hat{\boldsymbol{p}} = \sigma(\boldsymbol{z})$, where

$$\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{1 + e^{z_i}} = \frac{1}{1 + e^{-z_i}} \quad \forall\, i \in \{1, 2, \ldots, n\} \quad , \tag{33}$$

which is a *nonlinear transformation* (see Appendix B for a graphical illustration). To get the predictions for the actual class, a threshold for the probability is chosen, which is typically $\frac{1}{2}$: $\hat{\boldsymbol{y}} = \mathbf{1}_{\{\hat{\boldsymbol{p}} \geq \frac{1}{2}\}}$. For logistic regression, we have another loss function: *cross entropy*, penalizing the sum of differences between the predictions $\hat{\boldsymbol{y}}$ and the actual values $\boldsymbol{y}$:

$$\mathcal{L}(\boldsymbol{\beta}) = -\sum_{i=1}^{n} \left[y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i) \right] \quad . \tag{34}$$

Optimization of this loss function is notably more complex, as it does not have a closed-form solution (because $\boldsymbol{\beta}$ cannot be isolated during derivations). Instead, an iterative method is needed, which is provided through *gradient descent*.

Note that for multinomial (or multilabel) classification, the *softmax* function is used instead of the sigmoid function, as this normalizes the probabilities over all possible classes:

$$\text{softmax}(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \quad . \tag{35}$$

Note that softmax performs row-normalization, as each row sums to 1 after the operation, and is applicable if classes are mutually exclusive (i.e., the observation can belong to only one class). Multinomial classification also requires gradient descent as it has no closed-form solution. Also, to keep to conventional notation, from now on $\boldsymbol{\theta}$ will be used to denote the model parameters instead of $\boldsymbol{\beta}$.

A.2 Gradient descent

Gradient descent works by taking (small) random starting values for the parameters: $\boldsymbol{\theta}_{t=0}$. From these random starting values, the parameters $\boldsymbol{\theta}$ are changed in such a way to lower the loss function at each iteration (or time step) t, $\mathcal{L}(\boldsymbol{\theta}_t)$. The direction in which the loss function should be changed to minimize it, is given by the (negative) *gradient* of the loss function with respect to the parameters $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t)$. The size of the updates can be adjusted through a scaling factor $\eta > 0$, which is known as the *learning rate*. This whole procedure is summarized by the equation:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \cdot \nabla_{\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{\theta}_t) \quad , \tag{36}$$

and this is either repeated until *convergence* to the minimum, found when the loss function no longer shows meaningful decline, or when a predefined maximum iteration T is reached–as is typical in deep learning. For logistic regression, this method reliably finds the global minimum of the loss function, because the loss function is *convex* in $\boldsymbol{\theta}$, meaning that it only has one global minimum. Figure A.1 depicts gradient descent on a convex loss function, as with logistic regression. In deep learning, however, loss functions are no longer convex, making this procedure considerably more complex.

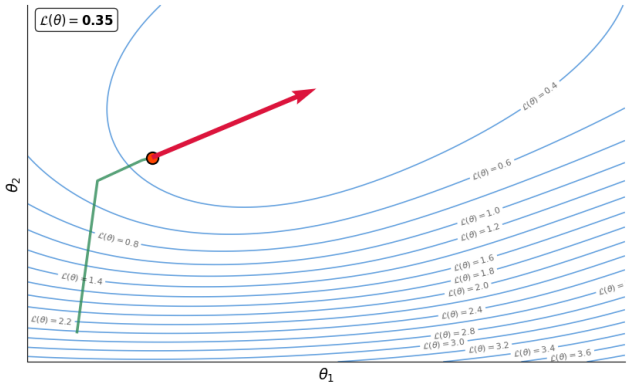


Figure A.1

Plot illustrating gradient descent on a convex loss function on a two-dimensional parameter space. The current position is given by the red dot, the red arrow illustrates the size and direction of the current update, the green line denotes the traveled descent path, and the blue lines indicate the contour lines of the loss function.

A.3 Deep learning: neural networks

The goal of traditional statistics is to *understand* relations in the data (e.g., by how much does unemployment drop when government spending increases?). To achieve this, the models used to model these relations need to be *interpretable*, which implies that they must be relatively simple. Linearity assumptions, as seen in linear and logistic regression, are very helpful in this regard, because the effect of one variable on another (i.e., the derivative) is constant across all values. However, linearity assumptions are often unrealistic. In machine learning (ML), the goal is not to understand relations in the data, but the goal is to *predict* as accurately as possible. As interpretability is no longer needed, this allows for complex and erratic model design, which is justified as long as it yields *predictive power*.

Because linear relations are often unrealistic, prediction of realistic data demands *flexible* models, which are able to fit *nonlinear relations*. A traditional choice for a flexible model is the Taylor series, but these only work for smooth, differentiable functions and require explicit choices such as the order of the expansion and which variables or interaction terms to include. What is really required, is a model that can fit any function: a *universal approximator*; and ideally one that does not require explicit steering of interactions to include.

This is where *neural networks* come in. The most common type of neural network is the feed-forward neural network (FNN), illustrated in Figure A.2. The nodes in the input layer denote scalar values, meaning that the input is a $n \times 1$ vector. The nodes in the *hidden layer(s)*, also called *neurons*, work like logistic regression: they transform a linear combinations of inputs through a nonlinear *activation function*. If the sigmoid function is used as activation function, the operation of each node is exactly the same as logistic regression, as shown in Equation 33. Also similar to logistic regression, the (model) parameters of the neural network are the weights/coefficients used to make the linear combinations. As each neuron, including the output node, features a linear combination of all previous nodes, this quickly leads to neural networks using a vast amount of parameters, as is typically the case. The *multi-layer perceptron* (MLP) is the most common type of FNN, and is defined as a fully connected FNN with at least one hidden layer. This model can be used for all prediction tasks–regression, binary classification and multilabel (or multinomial) classification–by adapting the activation in the output node (i.e., the final activation) to the prediction task: linear for regression, sigmoid for binary classification, and softmax for multilabel classification.

The universal approximation theorem (Cybenko, 1989) states that a neural network with a a feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function (on compact subsets of $\mathbb{R}^N$). This is an incredibly powerful result, which proved that neural networks can be used to fit practically any function. Fitting the neural network (the actual learning) is done through gradient descent, just as with logistic regression.

However, where gradient descent was straightforward and dependable with logistic regression due to the convex loss function, now the optimization is much more complex; complex models create complex optimization problems. With neural networks the loss function is no longer convex, but a complex landscape of hills and valleys. Because of the vast amount of model parameters used in DL architectures and the absence of convexity, deep learning is characterized by truly complex optimization problems. Gradient descent is guaranteed to find some minimum, but because of this complex landscape this is almost certainly a *local minimum*, while finding the *global minimum* is near impossible. Besides, verification of a minimum being global means knowing the loss function in the entire solution space, which is NP-complete (Blum & Rivest, 1992) and therefore tends to be infeasible in the first place.

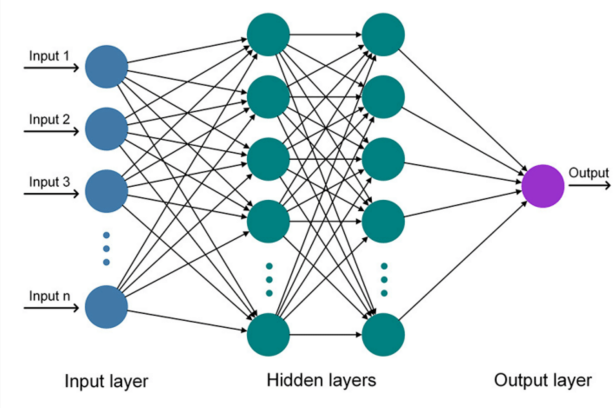


Figure A.2

A schematic representation of a fully-connected feedforward neural network (FNN), taken from Sahraei et al. (2021, Figure 4).

Because training neural networks through gradient descent is a complex and iterative procedure, not necessarily all information is learned from the data from the one iteration it was used for. Instead, the data may be used for multiple times in the same training round. One pass over the entire training dataset is known as an *epoch*, and training for ten epochs is not uncommon in DL. However, this does increase the risk of *overfitting*, which a central problem in machine learning where the model learns patterns that do not generalize outside the training data. Methods aimed at preventing overfitting are known as *regularization* techniques. It is helpful to consider that the data contains an informative *signal* and uninformative *noise*. Clearly, we want the model to learn to identify the signal and to ignore the noise. But by training for multiple epochs, the model will inevitably learn idiosyncratic patterns in the training data (i.e., it will fit the noise), which will lead to poor predictive performance. Overfitting can be easily identified by an increased loss on the *validation data*, which is a set of observations that is not used for training, but on which the model is tested during training to monitor (or validate) performance. It is impossible to know *ex ante* how many epochs are optimal for training, because this is influenced by many factors such as hyperparameter settings and the number of observations in the training data. Hence, if computational resources allow, a valid strategy for picking the best number of epochs is by first training the model for an abundant number of epochs, saving the model's state after each epoch, and to use the model's state for which the lowest validation loss was achieved.

A.3.1 Batching

The complexity of training a DL model opens the opportunity for a plethora of techniques to improve or tweak this process. One aspect of this is *batching*. The batch which refers to the number of observations over which the model is to compute the gradient at each iteration, where the gradient is computed as the average of the gradients w.r.t. the individual observations. The term "batch" is a somewhat overloaded concept: strictly speaking, the batch refers to the complete set of observations, and a subset is called a minibatch; in practice, the term batch is often used synonymously with minibatch (as is the case in this research), since minibatching is the *de facto* standard in machine learning. A similar ambiguity applies to stochastic gradient descent: in a narrow sense, it denotes gradient descent where the gradient is computed using a single randomly selected observation; in a

broader sense, it refers to any gradient descent method that uses a subset of the data to estimate the gradient at each iteration.

For practical implementation of batching in LLM optimization, more terminology has to be introduced. The batch size as introduced earlier is synonymous to the *total batch size*, which is the product of the *batch size per device*, being the number of examples processed in parallel on the device (usually the GPU), and the *gradient accumulation steps*, which determines the number of batches for which the gradient is stored, before the parameters are updated using the average of the stored gradients. A relatively large total batch size is desirable in LLM optimization, because the optimization process is extremely complex: the solution space has as many dimensions as there are (adjustable) model parameters, which is about 8 billion for Llama 3.1 8B model (assuming full parameter training). An eight-billion-dimensional space is impossible to imagine visually, but it undoubtedly makes for a complex landscape wherein there is considerable risk of following a descent path that leads to suboptimal regions in the solution space. A larger total batch size yields a more stable and precise descent path, which is very desirable for navigating this complex landscape. However, this precision comes at the cost of increased computational burden: a total batch size of 64 requires twice as much computation per parameter update as a total batch size of 32. However, this does not yield (significantly) longer computation time, as long as the examples in the batch can be processed in parallel, which is determined by the batch size per device. The computation time does scale one-to-one with the number of gradient accumulation steps. Therefore, it is desirable to minimize gradient accumulation steps while aiming to maximize the per device batch size for a maximized GPU utilization.

A.3.2 Adaptive optimizers and gradient clipping

The complex optimization landscape in DL, combined with the stochasticity introduced by (mini)batching, means that descent paths often traverse steep crevices or cliffs, leading to instability, as well as plateaus, which can cause training to stall. These problems are addressed by more sophisticated optimization algorithms, or *optimizers*, than the simple gradient descent update shown in Equation 36. Arguably the most important optimizer in DL is the *Adaptive Moment Estimation* (Adam) optimizer (Kingma & Ba, 2014), which improves convergence by keeping track of *exponentially decaying averages* of past gradients and squared gradients, by estimating the first and second moments of the gradients: m_t and v_t:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t = (1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 = (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} g_i^2 \end{aligned} \tag{37}$$

where $g_t = \nabla_\theta \mathcal{L}(\theta_t)$ is the gradient at iteration t, and $\beta_1, \beta_2 \in [0, 1)$ are the tunable hyperparameters through which to control the exponential decay rates. The recursively expressions on the right-most sides of these equations show how the moments are indeed exponentially decaying averages. Because $m_0 = v_0 = 0$, these estimates are biased towards zero in early iterations. To correct for this early bias, Adam employs the *bias-corrected moments* $\hat{m}_t$ and $\hat{v}_t$:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad , \tag{38}$$

which are used in the parameter update as:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \quad , \tag{39}$$

where ϵ is a small constant for numerical stability (particularly avoidance of division by zero). Finally, an important modification to Adam (which is also employed in this research) is Adam with *weight-decay* (AdamW), where a penalty term, dependent on the L2 (or Euclidean) norm of the parameter vector, is added to the loss function:

$$\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2}\|\theta\|_2^2 \quad , \tag{40}$$

where λ is a hyperparameter controlling the *weight-decay rate*. By penalizing the norm (or size, in simple terms) of the parameters, this incentivizes parameters to slightly shrink the parameters in at each step. This means that weight decay prevents them from growing too large, which helps stabilize training and reduces overfitting–making this a regularization technique. Hence, $\mathcal{L}_{\text{reg}}$ denotes the regularized loss function, which yields the following parameter update for AdamW:

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} - \eta\lambda\theta_t \tag{41}$$

Another approach to stabilize the descent path is *gradient (norm) clipping* (Pascanu et al., 2013). When a steep area of the optimization landscape is encountered, gradients become large–the size of which is measured by the gradient's norm–and in extreme cases this leads to a phenomenon called *exploding gradients*, where the gradient is orders of magnitude greater than it typically is during the rest of the descent. This is problematic, because these very large gradients tend to over-correct the descent path, disorganizing otherwise carefully tuned parameters. While the Adam optimizer already mitigates this issue somewhat by dampening the effect of large gradients, exploded gradients may nevertheless dominate the moments they affect, and they will in fact have a prolonged effect on the descent path through their contribution to the moments in the iterations after the initial encounter. Gradient clipping is a straightforward strategy that effectively resolves this, by limiting the norm of the gradients to a fixed number (typically clipping at 1.0), and scaling the size of the gradients down to this norm whenever it is exceeded.

A.3.3 The vanishing gradient problem (VGP) and residual networks

One of the most fundamental challenges in deep learning is the *vanishing gradient problem* (VGP). During backpropagation, the chain rule is applied repeatedly through multiple layers, making each layer's gradient the product of those above it. As each of these gradients may approach zero–which the gradients of the activation functions are especially sensitive to– the gradients and parameters updates in the deeper layers tend to diminish exponentially, preventing them from effectively learning meaningful representations.

To understand the VGP mathematically, it helps to consider a deep neural network as a composition of functions, where each layer l implements a transformation $\mathcal{F}_l$. For a network with L layers, the forward pass can be expressed as:

$$\boldsymbol{y} = \mathcal{F}_L(\mathcal{F}_{L-1}(\cdots \mathcal{F}_2(\mathcal{F}_1(\boldsymbol{x}))\cdots)) \quad , \tag{42}$$

with $\boldsymbol{y}$ being the model's output and $\boldsymbol{x}$ being the model's input vector. During backpropagation, the gradient ∇ of the loss $\mathcal{L}$ with respect to the parameters $\boldsymbol{\theta}$ in layer l (i.e., $\nabla_{\boldsymbol{\theta}_l}\mathcal{L}$), requires computing:

$$\nabla_{\boldsymbol{\theta}_l}\mathcal{L} = \left(\frac{\partial \mathcal{F}_l}{\partial \boldsymbol{\theta}_l}\right)^\top \left(\prod_{k=l+1}^{L} \left(\frac{\partial \mathcal{F}_k(\mathcal{F}_{k-1};\boldsymbol{\theta}_k)}{\partial \mathcal{F}_{k-1}(\mathcal{F}_{k-2};\boldsymbol{\theta}_{k-1})}\right)^\top\right)\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}\right) \quad . \tag{43}$$

Note that $\boldsymbol{y} = \mathcal{F}_L(\cdot))$ and that $\mathcal{F}_0 = \boldsymbol{x}$. This expression follows the matrix notation from Goodfellow et al. (2016a). For deeper layers (smaller l), this gradient $\nabla_{\boldsymbol{\theta}_l}\mathcal{L}$ becomes the product of more partial derivatives (or Jacobian matrices). When some of these derivatives tend to zero, the product tends to zero; the risk of which increases exponentially with network depth, which is the defining characteristic of the VGP. The problem is exacerbated by activation functions that can easily get *saturated*, meaning that their derivatives tend to zero for certain ranges of their input. This behavior can be easily seen from the plots of commonly used activation functions, as provided in Appendix B. The derivative of the sigmoid function, which has a maximum value of 0.25 and approaches zero for large $|x|$. The hyperbolic tangent function suffers from the same saturation behavior, while ReLU activations contribute zero gradients for all negative inputs, and the Swish (or SiLU) function shows similar but slightly milder saturation behavior compared to ReLU. The VGP severely constrains the utility of deep layers due to their negligible parameter updates.

The breakthrough solution came with the introduction of residual connections in ResNet (He et al., 2016)[21]. The key innovation is elegantly simple: instead of learning a direct mapping $\mathcal{F}(\boldsymbol{x})$, each layer learns a residual mapping $\mathcal{F}(\boldsymbol{x}) - \boldsymbol{x}$, with the final output of each layer being:

$$\boldsymbol{y} = \boldsymbol{x} + \mathcal{F}(\boldsymbol{x}) \tag{44}$$

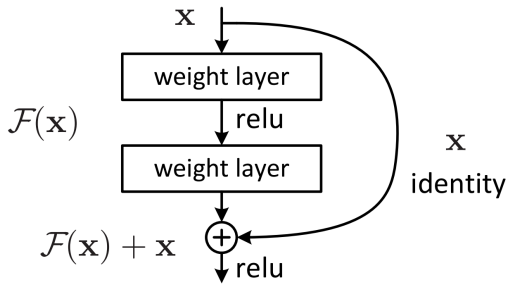


Figure A.3

A building block of the residual neural network (ResNet), from He et al. (2016, Figure 2). The main innovation is the addition of the identity of the input matrix $\boldsymbol{x}$ to the output of the neural network layer $\mathcal{F}(\boldsymbol{x})$, thereby mitigating the vanishing gradient problem (VGP).

This residual formulation fundamentally alters the gradient flow during backpropagation. The gradient of the loss with respect to the input becomes:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{y}}\left(1 + \frac{\partial \mathcal{F}(\boldsymbol{x})}{\partial \boldsymbol{x}}\right) \tag{45}$$

The critical insight is that the gradient always contains the identity term (i.e., 1), ensuring that even if $\frac{\partial \mathcal{F}(\boldsymbol{x})}{\partial \boldsymbol{x}}$ tends to zero, the gradient cannot vanish completely. This creates a direct pathway for

[21] According to Nature News (2025), He et al. (2016) is the most cited paper of the 21st century; Vaswani et al. (2017) came in seventh.

gradients to flow backward through the network, which are also called *skip connections* as the gradients can skip past the neural layer. This simple innovation allowed enabled usage of far deeper architectures; with He et al. (2016) even showing effectiveness of a 52-layered ResNet model. This innovation paved the way for effective training of LLMs.

Return to:

- Section 2: the beginning of the theoretical framework
- Section 2.2.2: softmax function in the self-attention mechanism
- Section 2.2.5: universal approximation theorem noted when discussing MLP blocks

B Activation functions

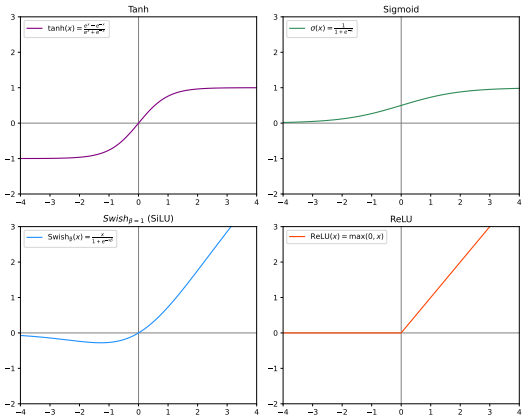


Figure B.1

Graphical representation of common activation functions: hyperbolic tangent (tanh), sigmoid (or logistic), swish with $\beta = 1$ (also known as the Sigmoid Linear Unit, or SiLU), and the Rectified Linear Unit (ReLU).

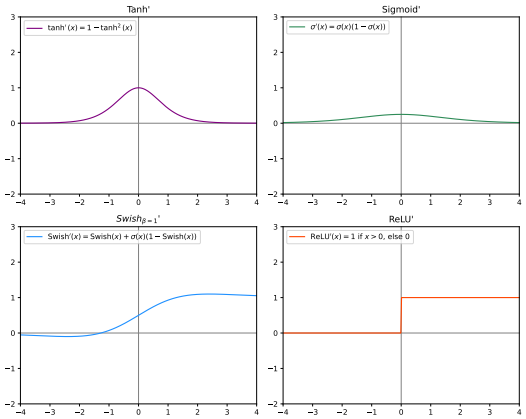


Figure B.2

Graphical representation of the derivatives of common activation functions: hyperbolic tangent (tanh), sigmoid (or logistic), swish with $\beta = 1$, and the Rectified Linear Unit (ReLU).

Return to:

- Section 2.2.5: MLP blocks employing GLU and Swish-GLU
- Section A.1: sigmoid function in logistic regression
- Section A.3.3: role of activations in the vanishing gradient problem

C Sinusoidal positional encodings as linear transformations across relative positions

This section serves to prove the linear combination property of sinusoidal positional encodings using trigonometric identities, to support the hypothesis by Vaswani et al. (2017) that this method allows for easy learning of relative positions. Recall the definitions of the sinusoidal positional encodings for position m and dimension i:

$$\begin{aligned}\text{PE}_{\text{sine}}(m, 2i) &= \sin\left(\omega_i \cdot m\right), \\ \text{PE}_{\text{cos}}(m, 2i+1) &= \cos\left(\omega_i \cdot m\right),\end{aligned}$$

where d is the model dimension, $i \in \{0, 1, ..., \lfloor d/2 \rfloor - 1\}$, and $\omega_i = \frac{1}{10000^{\frac{2i}{d}}}$. Next, consider the embedding at position $m + k$:

$$\begin{aligned}\sin\left(\omega_i(m+k)\right) &= \sin\left(\omega_i m + \omega_i k\right), \\ \cos\left(\omega_i(m+k)\right) &= \cos\left(\omega_i m + \omega_i k\right).\end{aligned}$$

Using the trigonometric addition identities:

$$\begin{aligned}\sin(a+b) &= \sin a \cos b + \cos a \sin b, \\ \cos(a+b) &= \cos a \cos b - \sin a \sin b,\end{aligned}$$

we obtain:

$$\begin{aligned}\sin\left(\omega_i(m+k)\right) &= \sin(\omega_i m)\cos(\omega_i k) + \cos(\omega_i m)\sin(\omega_i k), \\ \cos\left(\omega_i(m+k)\right) &= \cos(\omega_i m)\cos(\omega_i k) - \sin(\omega_i m)\sin(\omega_i k).\end{aligned}$$

This shows that the positional encoding at offset k can be expressed as a *linear combination* of $\sin(\omega_i m)$ and $\cos(\omega_i m)$:

$$\begin{bmatrix}\sin\left(\omega_i(m+k)\right) \\ \cos\left(\omega_i(m+k)\right)\end{bmatrix} = \begin{bmatrix}\cos(\omega_i k) & \sin(\omega_i k) \\ -\sin(\omega_i k) & \cos(\omega_i k)\end{bmatrix} \begin{bmatrix}\sin(\omega_i m) \\ \cos(\omega_i m)\end{bmatrix}.$$

This matrix is a rotation matrix parameterized by $\omega_i k$. Therefore, the model can represent embeddings at any relative position k as a linear transformation of the embedding at position m. This property implies that sinusoidal encodings facilitate reasoning over relative positions, since the model can learn relative offsets by simple linear projections.

Return to:

- Section 2.2.4: Positional embeddings theory

D Filing examples

This section provides examples of the identified components of the SEC filing-files that were used for domain adaptation, and an examples of cleaning results.

```
<SUBMISSION>
<ACCESSION-NUMBER>0000894189-24-006559
<TYPE>497K
<PUBLIC-DOCUMENT-COUNT>3
<FILING-DATE>20241031
<DATE-OF-FILING-DATE-CHANGE>20241031

...


</SERIES-AND-CLASSES-CONTRACTS-DATA>
<DOCUMENT>
<TYPE>497K
<SEQUENCE>1
<FILENAME>brownadvisorytaxexemptsust.htm
<DESCRIPTION>497K
<TEXT>



DocumentBrown Advisory Tax-ExemptSustainable Bond FundClass/Ticker: Institutional Shares / (Not Available for Sale)
Investor Shares / BITEX Advisor Shares / (Not Available for Sale) Summary Prospectus |  October 31, 2024Before you
invest, you may want to review the Fund's Prospectus, which contains more information about the Fund and its risks.
You can find the Fund's Prospectus, reports to shareholders, and other information about the Fund online at

...

arrangement, such as a 401(k) plan or an IRA, and then you may be taxed later upon withdrawal of your investment from
these tax-deferred accounts.Payments to Broker-Dealers and Other Financial IntermediariesIf you purchase the Fund
through a broker-dealer or other financial intermediary (such as a fund-supermarket), the Fund and its related
companies may pay the intermediary for the sale of Fund shares and related services.  These payments may create a
conflict of interest by influencing the broker-dealer or other intermediary and your salesperson to recommend the Fund
over another investment.  Ask your salesperson or visit your financial intermediary's website for more information.7
</TEXT>
</DOCUMENT>
</SUBMISSION>
```

(a) *Example of a filing (type 497K) with filing-level metadata at the top.*

```
<DOCUMENT>
<TYPE>XML
<SEQUENCE>46
<FILENAME>R30.htm
<DESCRIPTION>IDEA: XBRL DOCUMENT
<TEXT>

...

Schedule of Long-term Debt
Long-term debt is comprised of the following: As ofAs ofSeptember 30, 2024December 31, 2023September 30, 2024December 31, 2023(in millions)(weighted-average
interest rates)3.00% senior secured notes due 2025$1,110.0 $1,110.0 8.00 %8.00 %Fixed-rate loans due through 2039 (1)1,011.8 1,093.3 6.44 %5.83 %Unsecured term
loans due in 2031136.3 136.3 1.00 %1.00 %Fixed-rate class A 2015-1 EETC due through 2028245.6 256.6 4.10 %4.10 %Fixed-rate class B 2015-1 EETC due through
2024— 40.0 4.45 %4.45 %Fixed-rate class AA 2017-1 EETC due through 2030160.3 172.2 3.38 %3.38 %Fixed-rate class A 2017-1 EETC due through 203063.4 57.4 3.65 %
3.65 %Fixed-rate class B 2017-1 EETC due through 202644.7 48.2 3.80 %3.80 %Convertible notes due 202525.1 25.1 4.75 %4.75 %Convertible notes due
2026500.0 500.0 1.00 %1.00 %Long-term debt$3,287.2 $3,439.1 Less current maturities1,253.2 315.3 Less unamortized discounts, net47.6 69.0 Total
$1,986.4 $3,054.8 (1) Includes obligations related to 18 aircraft recorded as a failed sale leaseback. Refer to Note 10, Leases for additional information.



Schedule of Maturities of Long-term Debt
At September 30, 2024, long-term debt principal payments for the next five years and thereafter were as follows:September 30, 2024(in millions)Remainder of
2024$33.5 20251,267.5 2026674.1 2027154.7 2028257.2 2029 and beyond900.2 Total debt principal payments$3,287.2



Schedule of Interest Expense, Long-term Debt
Interest expense related to long-term debt and finance leases consists of the following:  Three Months Ended September 30,Nine Months Ended September 30,
2024202320242023(in thousands)8.00% senior secured notes (1)$23,252 $23,252 $69,757 $69,757 Fixed-rate term loans17,275 9,111 52,630 28,012 Unsecured term
loans343 344 1,028 1,029 Class A 2015-1 EETC2,531 2,758 7,649 8,312 Class B 2015-1 EETC— 492 446 1,506 Class C 2015-1 EETC— — — 777 Class AA 2017-1
EETC1,385 1,495 4,208 4,537 Class A 2017-1 EETC503 539 1,517 1,636 Class B 2017-1 EETC438 470 1,324 1,423 Class C 2017-1 EETC— — — 522 Convertible notes
(2)4,432 (1,468)12,795 (8,510)Finance leases8 7 25 24 Commitment and other fees412 415 1,247 1,243 Amortization of deferred financing
costs3,556 3,845 10,625 11,674 Total$54,135 $41,260 $163,251 $121,933 (1) Includes $1.1 million and $3.2 million of accretion and $22.2 million and
$66.6 million of interest expense for the three and nine months ended September 30, 2024, respectively. Includes $1.1 million and $3.2 million of accretion and
$22.2 million and $66.6 million of interest expense for the three and nine months ended September 30, 2023, respectively.(2) Includes $4.4 million of
amortization of the discount for the convertible notes due 2026, as well as interest expense for the convertible notes due 2025 and 2026, for the three months
ended September 30, 2024. Includes $13.3 million of amortization of the discount for the convertible notes due 2026, as well as interest expense for the
convertible notes due 2025 and 2026, partially offset by $0.5 million of favorable mark to market adjustments for the convertible notes due 2026, for the nine
months ended September 30, 2024. Includes $4.4 million and $9.9 million of amortization of the discount for the convertible notes due 2026, as well as interest
expense for the convertible notes due 2025 and 2026, offset by $5.9 million and $18.4 million of favorable mark to market adjustments for the convertible notes
due 2026, for the three and nine months ended September 30, 2023, respectively.

...

</TEXT>
</DOCUMENT>
```

(b) *Example of a cleaned document within a filing.*

Figure D.1

Examples illustrating filing structure and a successfully cleaned document in the corpus. Text truncated at . . ..

```
<DOCUMENT>
<TYPE>10-K/A
<SEQUENCE>1
<FILENAME>fy1410-k_ajanuary2015xdraf.htm
<DESCRIPTION>10-K/A
<TEXT>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
        <head>
                <!-- Document created using Wdesk 1 -->
                <!-- Copyright 2015 Workiva -->
                <title>FY1410-K_AJanuary2015-Draft-1-27-2015</title>
        </head>
        <body style="font-family:Times New Roman;font-size:10pt;">
<a name="sC7079D1AB92A08E3790C31FE0402AD55"></a><div></div><br><div style="line-height:120%;text-align:center;"><hr>
</div><div style="line-height:120%;text-align:center;font-size:12pt;"><font style="font-family:inherit;font-
size:12pt;font-weight:bold;">UNITED STATES</font></div><div style="line-height:120%;text-align:center;font-
size:12pt;"><font style="font-family:inherit;font-size:12pt;font-weight:bold;">SECURITIES AND EXCHANGE
COMMISSION</font></div><div style="line-height:120%;text-align:center;font-size:12pt;"><font style="font-
family:inherit;font-size:12pt;font-weight:bold;">Washington, D.C. 20549</font></div><div style="line-

...

style="line-height:144%;text-align:left;font-size:10pt;"><font style="font-family:inherit;font-size:10pt;">*
Indicates a management contract or compensatory plan or arrangement</font></div><div style="line-height:120%;text-
align:left;font-size:10pt;"><font style="font-family:inherit;font-size:10pt;"><br></font></div><br><div><div
style="line-height:120%;text-align:center;font-size:10pt;"><font style="font-family:inherit;font-
size:10pt;">38</font></div><div style="line-height:120%;text-align:left;font-size:10pt;"><font style="font-
family:inherit;font-size:10pt;"><br></font></div></div> </body>
</html>
</TEXT>
</DOCUMENT>
```

(a) *Document containing HTML (*681,121 *characters).*

```
<DOCUMENT>
<TYPE>3
<SEQUENCE>1
<FILENAME>xslF345X03/edgar240670090_3-lavelle.xml
<DESCRIPTION>OWNERSHIP DOCUMENT
<TEXT>
<XML>
<?xml version="1.0"?>
<ownershipDocument>

    <schemaVersion>X0206</schemaVersion>
...
    <nonDerivativeTable>
        <nonDerivativeHolding>
            <securityTitle>
                <value>Common Shares, no par value (&amp;quot;Common Shares&amp;quot;)</value>
            </securityTitle>
            <postTransactionAmounts>
...
    </nonDerivativeTable>

    <derivativeTable>
        <derivativeHolding>
            <securityTitle>
                <value>Stock Option (right to buy)</value>
...
                </directOrIndirectOwnership>
            </ownershipNature>
        </derivativeHolding>
    </derivativeTable>

    <footnotes>
        <footnote id="F1">The option has vested and is exercisable as of the date hereof.</footnote>
        <footnote id="F2">Represents an exercise price of $1.16 Canadian dollars, converted to U.S. dollars at the
daily average exchange rate of $1.00 to CAD$1.4379 reported by the Bank of Canada on December 30, 2024.</footnote>
    </footnotes>

    <remarks>Exhibit List - Exhibit 24.1 - Power of Attorney</remarks>
...
</ownershipDocument>
</XML>
</TEXT>
</DOCUMENT>
```

(b) *Document containing XML (*3,871 *characters).*

```
<DOCUMENT>
<TYPE>GRAPHIC
<SEQUENCE>3
<FILENAME>image0.jpg
<TEXT>
begin 644 image0.jpg
M_]C_X  02D9)1@ ! 0$ E@"6  #_VP!#  @&l@<&l0@`!P<)"0@*#!0-# L+
MW!D2$PV4'1H2'AT:'!P@)"XG("(L(QP<*#<I+# Q-#0T'R<Y/3@R/"XS-#+_
MVP!# 0D)"0P+#!@-#1@R(1PA,C(R,C(R,C(R,C(R,C(R,C(R,C(R,C(R,C(R
M,C(R,C(R,C(R,C(R,C(R,C(R,C+_P  1" *Q19L# 2(  A$! Q$!_\0
M*P   04! 0$! 0$             $" P0%!@<("0H+_\0 M1   @$# P($ P4%
M1 0   %] 0(#  0111(A,4$&$U%A1R)Q%#*1D:$((T*QP152T? D,V)R@@O*
M%Ax8&1HE)B<H*2HT-38W.#QZ0T1%1D=(24I35%565UA9GF-D9G9G:&E3<W1U
M+G-X>7?HA(G&AX@)BI*3E)G6EYB9F)*CI4$IZBI3K*5%+6VM[BYNL+#Q,7&
MQ\C)KM+3U-76U]C9VN'BX*3EYNJHZ>K(\O/T]?gW'/GZ_\0 I*$  P$! 0$!
M 0$! 0       $" P0%!@<("0H+_\0 M1$  @$"! 0#! <%! 0  0)W  $"
M Q$$!2$Q!A)!40=A<1,B,H$(X$*1H;'!"2,54O 58G+1"A8D-.$E\1<8&1HF

...

M.0KN,^C,.YJX+"S&GH9BT@&K#L\J$1C9'L(*;5Z#:54C'3 QTH \_P!:^Q_\
M*H\:_P!I^1YWFZEYWVG&[5/,?[-NW?Q;/L_EYYV^7MXVUH:UY5G;^+I/(U#
M2&W'6[*3?%,F+6/2Y4@X;]P(\)@99B?,7&!U&WH6CZA>"\O=*L;FZ$3O":W
M1W$; AOW$9VD,P(Z'<J6BZT+i[48-1O-*L,B'@V^3<S6Z/)'M.Y+K$9&^2l
MCH:  _QM_P ASPG_-A5_PWT8NJY>TL+-/WWX,VVO^^5+HT\x(P-DCSPjG'H
MQWOD]3N;U-)B20S/"\L4;O^^)F4$HVTKE?0[689'8D=ZC%A9BSALQ:0"UA
MVy5"(QLCV$%-J]!M*J1CI@8Z4 >?ZU]C_P"%4>-?[3\CSO-U+SOM.-WF>8_V
M;=N_BV?9_+S5M\O;QMOY)lJ4NN^+V?4Y&%K/?&)DU73WGS:_9D/+>c@$&[S^
M"-NX2\Y+BZR[T+1]8O!>7NE6-S="}H!--;H[B-@0R;B,[2&8$+#N/K4EYI.F
MZAx6MQ>Z?:7,]H^^VDFASVA;(.4)&5.5'(]!Z4 <'lJ4-O;Z=!#/=7UW%'HD"
M>(KB2,QR2V $@#LLGSK(G\XD+\P0S9_>>37I%1QP0PO,\44:/,^^5E4 NVT+
MEO4[549/8 =J(((;6WBM[>*.&") D<<:A510,  #@ ##0!)1110 4444 %%%
M% !1110 4444 %%%% !1110 4444 %%%% !1110 4444 %%%% !1110 4444
M %%%% !1110 4444 %%%% !1110 4444 %%%% !1110 4444 %%%% !1110
M4444 %%%% !1110 4444 %%%% !1110 4444 %%%% !1110 4444 %%%% !1
M110 4444 %%%% !1110 4444 %%%% !1110 4444 %%%% !1110 4444 %%%
M% !1110 4444 %%%% !1110 4444 %%%% !1110 4444 %%%% !1110 4444
7 %%%% !1110 4444 %%%% !1110!_]D!

end
</TEXT>
</DOCUMENT>
```

(c) *Document containing a UU-encoded image (*2,065,902 *characters).*

Figure D.2

Examples of different document types in the corpus. Document-level metadata appears at the top and bottom of each document. Text is truncated at the triple dots (. . .) for display purposes.

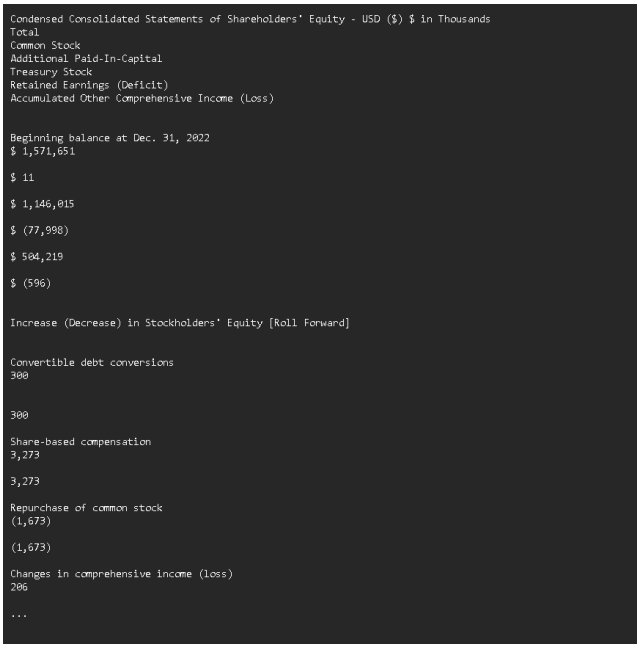


Figure D.3

Example of a cleaned XML table nested in a HTML section. The table is truncated at the triple dots (. . .). The full table covers 789 *lines and contains* 3961 *characters.*

Return to:

- Section 3.1: Domain adaptation data description
- Section 4.3.3: Data cleaning

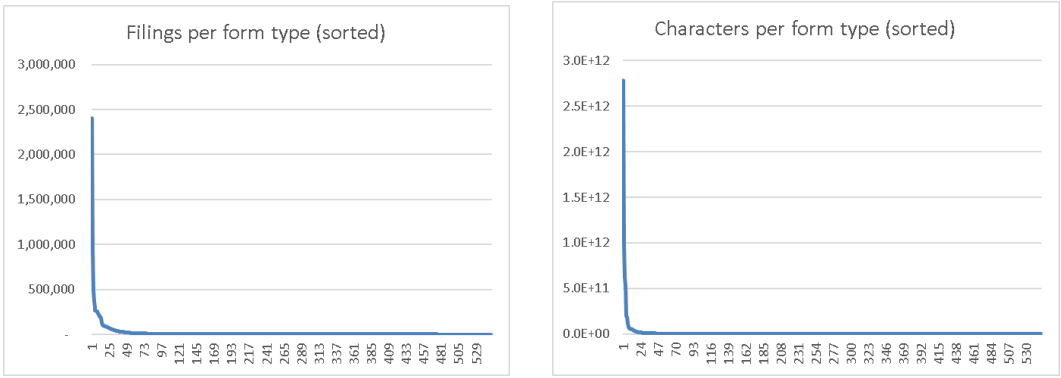
E Additional descriptive statistics

Additional descriptive statistics on the domain adaptation data are provided here.

Table E.1

Processing statistics for domain adaptation data cleaning by year.

Processing Step	2012	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024
Files processed	6.6×10^5	6.7×10^5	6.6×10^5	6.6×10^5	6.3×10^5	6.5×10^5	6.5×10^5	6.6×10^5	7.4×10^5	8.1×10^5	7.9×10^5	8.1×10^5	8.6×10^5
Documents processed	3.6×10^6	4.5×10^6	4.2×10^6	4.2×10^6	3.9×10^6	4.0×10^6	4.2×10^6	4.3×10^6	5.1×10^6	5.8×10^6	6.1×10^6	6.1×10^6	6.4×10^6
HTML characters removed	2.6×10^{11}	3.7×10^{11}	3.1×10^{11}	2.9×10^{11}	3.0×10^{11}	3.0×10^{11}	3.0×10^{11}	3.8×10^{11}	4.6×10^{11}	4.9×10^{11}	5.1×10^{11}	5.3×10^{11}	5.6×10^{11}
HTML documents parsed	2.0×10^6	2.4×10^6	2.4×10^6	2.4×10^6	2.2×10^6	2.2×10^6	2.3×10^6	2.4×10^6	2.5×10^6	2.8×10^6	2.9×10^6	2.9×10^6	2.9×10^6
UU-encoded characters removed	1.3×10^{11}	1.7×10^{11}	1.9×10^{11}	1.7×10^{11}	1.3×10^{11}	1.4×10^{11}	1.6×10^{11}	1.7×10^{11}	2.1×10^{11}	2.6×10^{11}	2.8×10^{11}	3.2×10^{11}	3.6×10^{11}
UU-encoded documents removed	8.8×10^5	9.7×10^5	1.0×10^6	1.1×10^6	1.0×10^6	1.1×10^6	1.2×10^6	1.2×10^6	1.4×10^6	1.7×10^6	1.7×10^6	1.8×10^6	2.0×10^6
UU-encoded PDFs extracted	2.1×10^4	2.3×10^4	1.7×10^4	1.6×10^4	1.9×10^4	1.8×10^4	1.8×10^4	1.4×10^4	2.1×10^4	2.6×10^4	2.4×10^4	2.6×10^4	2.7×10^4
UU-encoded PDF extractions failed	4.0×10^2	4.4×10^2	3.6×10^2	3.5×10^2	4.2×10^2	3.9×10^2	4.1×10^2	2.8×10^2	3.9×10^2	5.1×10^2	5.5×10^2	5.0×10^2	9.0×10^2
UU-encoded PDF characters extracted	5.0×10^8	4.3×10^8	4.1×10^8	5.2×10^8	4.7×10^8	5.6×10^8	8.1×10^8	7.9×10^8	8.0×10^8	8.5×10^8	9.0×10^8	5.1×10^9	4.6×10^9
Files removed	1.2×10^4	1.2×10^4	1.3×10^4	1.3×10^4	9.2×10^3	8.6×10^3	7.4×10^3	5.6×10^3	4.5×10^3	4.4×10^3	4.4×10^3	4.3×10^3	4.3×10^3
Files removed – character count	5.6×10^8	3.1×10^8	2.3×10^9	5.2×10^8	1.5×10^9	3.0×10^9	4.1×10^9	2.9×10^9	7.9×10^9	1.6×10^{10}	2.2×10^{10}	1.3×10^{10}	1.4×10^{10}



(a) *Filings per form type.*

(b) *Characters per form type.*

Figure E.1

Distributions across SEC form types, sorted from largest to smallest. The vertical axis denotes the number of filings (a) or characters (b), while the horizontal axis denotes the index of the form types in this sorting. The corpus contains 550 form types in total.

Return to:

- Section 3.1: Domain adaptation data description
- Section 4.3.4: Data cleaning statistics
- Section 4.3.6: Subsetting data on whitelisted form types

F Domain adaptation data whitelisting specifications

Table F.1

Assessment and selection of form types for inclusion in domain adaptation data. This table shows the 61 assessed form types out of the total 550 form types encountered in the data. The "Whitelist" column shows selected form types with 1 and others by 0. The "Characters" column shows the total number of characters, and the "Filings" column shows the total number of filings for this form type in the cleaned corpus. The form types are sorted by number of characters.

Form type	Characters	Filings	White-listed	Notes
ABS-EE	2.8E+12	33,125	0	Asset-Backed Securities Exhibits. Consists solely of XML tables (as legally required) to provide detailed loan-level data about the underlying assets in a security. No narrative text.
10-Q	9.7E+11	260,817	1	Quarterly Report for publicly listed firms.
10-D	6.2E+11	88,241	0	Monthly Distribution Report that is filed by asset-backed issuers. Mainly structured numerical data.
10-K	5.2E+11	94,504	1	Annual Report for publicly listed firms.
485BPOS	1.9E+11	96,352	1	Amendment to a registration statement for a mutual fund or investment company. Narrative text heavy!
NPORT-P	1.9E+11	255,964	0	Structured holdings data.
8-K	1.1E+11	921,903	1	Announcement of major event to shareholders, such as M&A, bankruptcy, delisting or major purchase.
20-F	7.8E+10	10,005	1	Annual Report for Foreign Issuers. Text heavy, just like 10-K's.
424B2	5.7E+10	484,942	1	Prospectus for Securities
497	5.6E+10	181,091	1	Updated information to shareholders regarding a mutual fund's prospectus or summary prospectus. Text heavy!
N-PX	5.2E+10	43,370	0	Proxy voting records. Minimal natural language, highly structured.
424B3	5.2E+10	85,415	1	Prospectus Supplement to Registration Statement, giving supplemental information to a registration statement such as an S-1. Updating factual information, but informative text-rich.
13F-HR	4.8E+10	266,134	0	Quarterly Report on Institutional Investment Managers. Data-centric, minimal amount of natural language.
S-1/A	3.9E+10	31,933	1	Amendment to registration statement for IPO.
ABS-15G/A	3.6E+10	6,351	0	Amendment for Asset-Backed Securities Reporting. Mainly data-driven.
6-K	3.5E+10	297,054	1	Disclosure of significant information by a foreign private issuer. Disclosure may concern a major event as with 8-K forms, or earnings reports and financial statements that have been disclosed in the issuer's home country.
N-CSR	2.8E+10	42,684	1	Funds shareholder reports. Includes financial statements such as balance sheets and portfolio holdings, AND management's discussion and analysis!

Table F.1 continued

Form type	Characters	Filings	White-listed	Notes
S-1	2.7E+10	12,863	1	Registration Statement for IPO.
10-Q/A	2.3E+10	13,063	1	Amendments to quarterly reports.
10-K/A	2.1E+10	15,224	1	Amendments to annual reports.
4	2.0E+10	2,405,601	0	Changes in beneficial ownership (required for anyone with >10% of the firm's stock).
S-4/A	1.8E+10	6,838	1	Amendment to registration statement for transactions such as mergers and acquisitions.
DEF 14A	1.7E+10	68,151	1	Definitive proxy statement. Discloses all information needed for votes on a shareholder meeting. Text rich.
N-CSRS	1.7E+10	41,730	0	Invesment firms' proxy voting records. Seems much like N-PX.
N-MFP2	1.3E+10	35,951	0	Monthly reports of money market funds, disclosing holdings, liquidity, risk exposure and portfolio valuation. Mainly contains structured, numerical data.
POS AM	1.3E+10	13,375	1	Post-Effective Amendment to a Registration Statement. Updates a registration statement (type S forms), contains narrative text.
485APOS	1.2E+10	28,428	1	Post-Effective Amendment for Investment Companies. Includes fund descriptions, risk factors, and prospectus updates, which are rich in narrative text.
S-4	1.2E+10	4,247	1	Registration statement for transactions such as mergers and acquisitions. Text rich.
40-F	1.2E+10	1,948	1	Annual Report for Canadian Issuers. Text rich, much like a 10-K.
ABS-15G	1.1E+10	19,853	0	Asset-Backed Securities (ABS) Disclosure, contains mainly numerical data on loan performance and repurchase history.
N-MFP	1.1E+10	32,869	0	Monthly Report for Money Market Funds, disclosing holdings and risk exposure. Mainly numerical.
FWP	1.1E+10	197,999	1	Free Writing Prospectus. Providing additional information not formally required in the registration statement. Contains marketing material, informal descriptions, and additional non-required disclosures.
424B5	9.6E+09	34,093	1	Prospectus Supplement, filed as a supplement to an existing registration statement (e.g., S-1). Rich in narrative text.
497K	6.5E+09	266,471	1	Prospectus Supplement for mutual funds and investment companies. Contains natural language, but mainly very technical and niche. Nevertheless deemed useful.
N-Q	6.2E+09	47,576	0	Quarterly Portfolio Holdings Report for investment companies. Mainly numerical.
N-CEN	5.1E+09	20,549	1	Annual Report for Registered Investment Companies. Unlike other annual reports, the text is highly structured and limitedly narrative, but it is likely to contain useful information due to its scope.

Table F.1 continued

Form type	Characters	Filings	White-listed	Notes
8-K/A	4.5E+09	27,335	1	Amendment to event disclosure 8-K. Text-rich, though mainly focussed on correcting errors and supplementing information.
425	3.7E+09	53,138	1	Communications related to M&A and business combinations. Can contain press releases, CEO statements and investor communications. Rich in narrative text.
D	3.1E+09	363,121	0	Notice of exemption from the full registration requirements of the SEC. Very legalistic, minimal narrative, and mostly data-driven.
SC 13G/A	2.9E+09	220,947	0	Amendment to holdings disclosure of large shareholders. Data-driven.
D/A	2.4E+09	229,973	0	Amendment to notice of exemption.
DEFA14A	2.0E+09	73,013	1	Definitive Additional Proxy Soliciting Material. Adds to DEF 14A. Used by public companies to provide additional information to shareholders in relation to a proxy statement. Highly text rich with legal and financial jargon.
SC 13D/A	2.0E+09	55,378	1	Amendment to beneficial ownership report. Contains explanations for acquisitions, which is narrative text.
CORRESP	1.9E+09	134,324	1	Correspondence between issuer and SEC staff. Highly narrative text, though usually very legalistic. But often contains clarifications of (financial) concepts, which would be very useful.
S-8	1.6E+09	29,224	1	Registration Statement for Employee Benefit Plans, registering securities offered to employees. Text rich.
3	1.3E+09	194,593	0	Initial statement of beneficial ownership. Highly structured, data-driven XML content.
SC 13G	1.3E+09	91,826	0	Holdings disclosure of large shareholders. Data-driven.
NSAR-B	9.1E+08	18,458	0	Forerunner of N-CEN report. Not useful as the form type is outdated.
SC 13D	8.4E+08	17,977	1	Beneficial ownership report. Contains explanations for acquisitions, which is narrative text.
NSAR-A	7.7E+08	17,908	0	Forerunner of N-CEN report. Not useful as the form type is outdated.
UPLOAD	6.9E+08	105,786	0	Diverse content, may relate to any other form type.
144	6.6E+08	66,190	0	Reporting proposed sale of securities of insiders of a company. Highly tabular data, little natural language. May contain short descriptions on the circumstances of the sale, but this is not likely to be of much financially informative value.
24F-2NT	5.9E+08	83,988	0	Notice of Securities Sold filed by investment companies (like mutual funds). Used to report the number of securities sold during a fiscal year and pay any related registration fees to the SEC. Little to no natural language.
4/A	4.1E+08	52,415	0	Amendment to form 4, which mainly contains structured numerical data.
S-8 POS	3.5E+08	23,818	1	Post-Effective Amendment to Form S-8.

Table F.1 continued

Form type	Characters	Filings	White-listed	Notes
5	3.4E+08	38,460	0	Annual Statement of Changes in Beneficial Ownership. Much like a consolidated set of type 4 forms, which means it is highly numerical.
485BXT	3.1E+08	23,992	1	Post-Effective Amendment to a Registration Statement, mainly used by investment companies to update a prospectus after effectuation of a registration statement.
497J	2.7E+08	66,554	0	Certification form filed by mutual funds, used to certify (or promise) that the fund's prospectus is consistent with the filed version. Not financially informative.
13F-NT	2.5E+08	80,420	0	Notice of Non-Compliance with 13F requirements. Filed by the issuer to explain the delay in submitting the required 13F form. Not interesting from a financial perspective.
NT 10-Q	1.7E+08	25,854	0	Notification of Late Filing of Quarterly Report. Text-rich, but irrelevant content to financial expertise.
EFFECT	1.1E+08	47,451	0	Registration Statement Effective Notice, filed when a registration statement (e.g., S-1, S-4) comes into effect. Mainly textual, but highly legalistic and not so much financially informative.

Return to:

- Section 3.1: Domain adaptation data description
- Section 4.3.6: Whitelisting methodology

G Instruction tuning data examples

Below is an example JSON entry from the TAT-QA dataset. Note that the examples is truncated at the triple dots (. . .).


```
{
  "table": { # The tabular data in a hybrid context
    "uid": "3ffd9053-a45d-491c-957a-1b2fa0af0570", # The unique id of a table
    "table": [ # The table content which is 2d-array
      [
        "",
        "2019",
        "2018",
        "2017"
      ],
      [
        "Fixed Price",
        "$ 1,452.4",
        "$ 1,146.2",
        "$ 1,036.9"
      ],
      ...
    ]
  },
  "paragraphs": [ # The textual data in a hybrid context
    {
      "uid": "f4ac7069-10a2-47e9-995c-3903293b3d47", # The unique id of a paragraph
      "order": 1, # The order of the paragraph in all associated paragraphs
      "text": "Sales by Contract Type: Substantially all of
       our contracts are fixed-price type contracts.
       Sales included in Other contract types represent cost
       plus and time and material type contracts."
    },
    ...
  ],
  "questions": [
    {
      "uid": "eb787966-fa02-401f-bfaf-ccabf3828b23",
      "order": 2,
      "question": "What is the change in Other in 2019 from 2018?",
      "answer": -12.6,
      "derivation": "44.1 - 56.7",
      "answer_type": "arithmetic",
      "answer_from": "table-text",
      "rel_paragraphs": [
        "2"
      ],
      "req_comparison": false,
      "scale": "million"
    }
  ]
}
```


The following is an example of the Alpaca dataset.

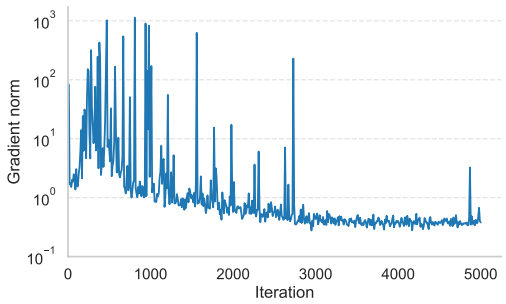

```
{
  "instruction": "Write a short poem about the ocean.",
  "input": "",
  "output": "The ocean whispers to the shore,\nWaves embrace the sand once more.\nBeneath the foam, a world unseen,\nWhere
        coral blooms in depths serene."
}
```


Return to:

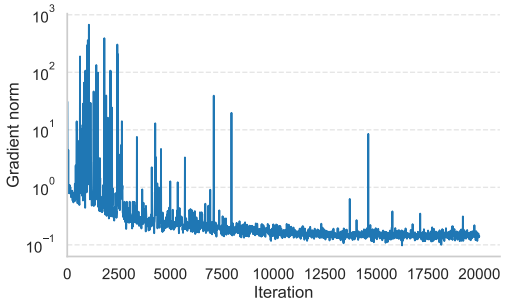
- Section 3.2: Instruction tuning data description

H Additional training statistics

This section provides several plots with additional statistics reported for the domain adaptation and instruction tuning training runs, detailing gradient norms and GPU utilization. The domain adaptation plots are shown first and the instruction tuning plots are shown second.



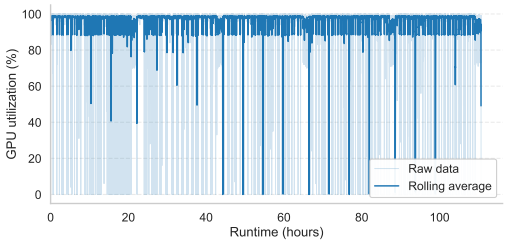
(a) *Long-sequence domain adaptation gradient norms.*



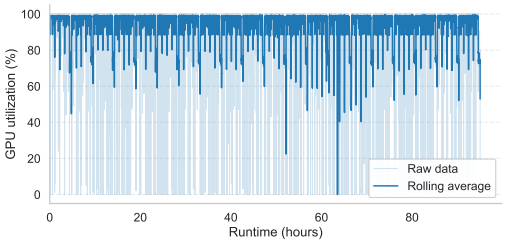
(b) *Short-sequence domain adaptation gradient norms.*

Figure H.1

L2-norm of the gradients during domain adaptation training runs (a) long-sequence and (b) short-sequence, plotted on a logarithmic scale. Statistics are reported every tenth iteration. Due to gradient clipping with the clip set at 1.0, gradients with an L2-norm greater than 1.0 were scaled down to an L2-norm of 1.0. The figures show the L2-norms prior to clipping.



(a) *Long-sequence domain adaptation GPU utilization.*

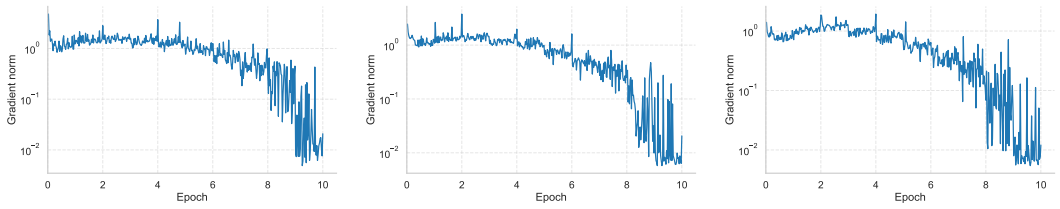


(b) *Short-sequence domain adaptation GPU utilization.*

Figure H.2

GPU utilization during domain adaptation training runs (a) long-sequence and (b) short-sequence. Both the raw data, with statistics reported for every tenth iteration, and a rolling average with a window size of ten observations are shown. For the long-sequence run, average utilization was 96.47%, with 1.26% of observations being below 50% utilization, while for the short-sequence run utilization averaged 94.19% with 2.16% of observations showing utilization below 50%.

The instruction tuning plots are shown below:



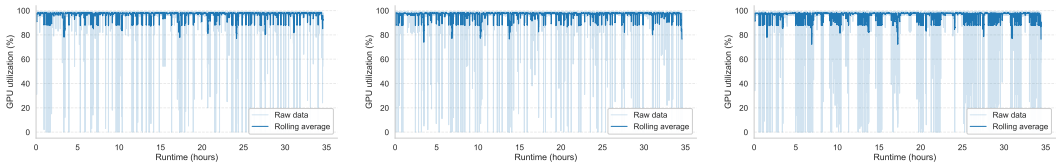
(a) *Base (vanilla) model instruction tuning gradient norms.*

(b) *Long-sequence domain-adapted instruction tuning gradient norms.*

(c) *Short-sequence domain-adapted instruction tuning gradient norms.*

Figure H.3

L2-norm of the gradients during instruction tuning of the Llama 3.1 8B model variants: (a) base/-vanilla, (b) long-sequence domain-adapted, and (c) short-sequence domain-adapted. All plots use a logarithmic scale, with statistics reported every tenth iteration. Due to gradient clipping at 1.0, gradients with an L2-norm greater than 1.0 were scaled down to 1.0. The figures show the L2-norms prior to clipping.



(a) *Base model instruction tuning GPU utilization: averaging 96.48%, with utilization dropping below 50% for 1.19% of observations and above 95% for 96.12% of observations.*

(b) *Long-sequence domain-adapted model instruction tuning GPU utilization: averaging 96.31%, with utilization dropping below 50% for 1.36% of observations and above 95% for 96.25% of observations.*

(c) *Short-sequence domain-adapted model instruction tuning GPU utilization: averaging 95.71%, with utilization dropping below 50% for 2.03% of observations and above 95% for 95.15% of observations.*

Figure H.4

GPU utilization during instruction tuning of the Llama 3.1 8B model variants: (a) base/vanilla, (b) long-sequence domain-adapted, and (c) short-sequence domain-adapted. Both the raw data (statistics reported every tenth iteration) and a rolling average with a window size of ten observations are shown.

Return to:

- Section 4.3.7: Pre-tokenization methodology
- Section 4.5: Instruction tuning methodology
- Section 5.1: Domain adaptation results
- Section 5.2: Instruction tuning results

I FLARE/PIXIU bug fixes

The code of the FLARE/PIXIU framework (The FinAI Team, 2024) proved to be faulty and outdated. To resolve this, the following patches were made:

- The `bart_score` import was incorrectly specified, as it was not defined at top level but under `metrics.BARTScore` in the PIXIU git-repository clone. This error was patched in `PIXIU/src/tasks/flare.py`.
- The PIXIU code uses the outdated `LlamaTokenizer` instead of the now standard `AutoTokenizer` from the `transformers` library, and not the Rust-optimized 'fast' implementation which HF now encourages to use instead (Hugging Face, 2025a, 2025b). These were patched in `PIXIU/src/financial-evaluation/lm_eval/models/huggingface.py`.
- Another issue with `PIXIU/src/financial-evaluation/lm_eval/models/huggingface.py`, is a manual override of special tokens IDs in the `_create_auto_tokenizer` function of the `AutoLlamaCausalLM` class. This override is incompatible with LLama 3, and is clearly outdated: it defines the `unknown`-token though Llama 3 uses a BPE tokenizer which does not need or use an `unknown`-token, and the EOS token ID of Llama 3.1 is 128001 but is manually set to 2. The `unknown`-token definition may be implemented to ensure compatibility with legacy software, but this EOS token ID clearly shows that this software is outdated as this ID was set to 2 in Llama 1 and 2 but was changed to 128001 for Llama 3. These overrides were removed, with the only addition being that the PAD token, which is undefined for Llama 3.1, was set to equal the EOS token.

Return to:

- Section 4.2: List with notable software used in this methodology.