Summary of "Attention Is All You Need"

Main Contribution:

The paper introduces the Transformer, a novel neural network architecture that relies entirely on attention mechanisms, eliminating the need for recurrent or convolutional layers in sequence-to-sequence models.

Key Innovation - Self-Attention

The Transformer uses "scaled dot-product attention" and "multi-head attention" to process sequences in parallel rather than sequentially. This allows the model to:

- Capture dependencies regardless of their distance in the sequence
- Parallelize computation during training
- Reduce the computational path length between distant positions

Architecture

The model follows an encoder-decoder structure with stacked layers. Each layer contains:

- Multi-head self-attention mechanisms
- Position-wise feed-forward networks
- Residual connections and layer normalization

Results

On machine translation tasks, the Transformer achieved:

- 28.4 BLEU on English-to-German translation (new state-of-the-art)
- 41.8 BLEU on English-to-French translation
- Significantly faster training than recurrent models (12 hours vs. days)
- Better performance with lower computational cost

Impact

The paper demonstrates that attention mechanisms alone are sufficient for high-quality sequence transduction, challenging the dominant paradigm of recurrent neural networks for these tasks. The architecture proved generalizable beyond translation, succeeding on English constituency parsing as well.

Significance

This work established the foundation for modern large language models, showing that pure attention-based architectures could outperform traditional RNN-based approaches while being more efficient to train.

Summary of "Dolma: An Open Corpus of Three Trillion Tokens for Language Model Pretraining Research"

Main Contribution

The paper introduces Dolma, a 3 trillion token English corpus designed for language model pretraining research, addressing the lack of transparency in training data for modern language models.

Key Problem Addressed

Most powerful language models (GPT-4, Claude, LLaMA 2) provide little to no information about their pretraining data composition, hindering:

- Scientific understanding of how data affects model capabilities
- Transparency for developers and users
- Reproducibility of research
- Analysis of biases and memorization

Dolma Dataset Composition

The corpus draws from six diverse sources:

- **Common Crawl** (web pages): 2,479B tokens
- **GitHub** (code): 411B tokens
- **Reddit** (social media): 89B tokens
- **Semantic Scholar** (papers): 70B tokens
- **Project Gutenberg** (books): 6B tokens
- **Wikipedia/Wikibooks**: 4.3B tokens

Methodology & Design Principles

The team conducted extensive **data ablation experiments** by training 1.2B parameter models to validate curation decisions:

- **Quality filtering**: Combined Gopher and C4 rules for optimal performance
- **Content filtering**: Removed toxic content and personally identifiable information (PII)
- **Deduplication**: Multi-stage approach (URL, document, paragraph-level)
- **Language filtering**: English-only content using FastText classifiers

Key Findings

- Combining multiple filtering strategies produces compounding positive effects
- Domain diversity matters: models trained on diverse sources (like Dolma) achieve better perplexity across heterogeneous domains
- Even small amounts of code data improve reasoning task performance
- Data mixture composition significantly affects downstream capabilities

Open Science Contribution

The team released:

1. **The full Dolma corpus** (3T tokens) under ODC-By license
2. **Dolma Toolkit** - high-performance, open-source tools for data curation
3. **OLMo-1B model** trained on Dolma as validation

Validation Results

OLMo-1B (trained on Dolma) outperformed comparable models like TinyLlama on 4 of 8 evaluation tasks despite similar training tokens, demonstrating Dolma's quality.

Significance

Dolma represents one of the largest openly available, well-documented pretraining corpora with transparent curation methodology, enabling reproducible research on the relationship between training data and model behaviour—a critical gap in current LLM research.

Summary of "Training Language Models to Follow Instructions with Human Feedback"

Main Idea

Large language models like GPT-3 are powerful but often generate outputs that are untruthful, toxic, or simply not helpful. They don't align with user intent. The language modelling objective (predicting next tokens from internet text) differs fundamentally from "follow user instructions helpfully and safely," creating a misalignment problem. How can we align language models with human intentions across a wide range of tasks? Fine-tune GPT-3 using reinforcement learning from human feedback (RLHF) to create InstructGPT. Models that significantly outperform their larger counterparts in following instructions while being more truthful and less toxic.

Key Methodology (Three-Step Process)

1. **Supervised Fine-Tuning (SFT):** Collect demonstrations of desired behaviour from labellers and fine-tune GPT-3 using supervised learning
2. **Reward Model (RM) Training:** Collect comparison data where labellers rank multiple model outputs, then train a reward model to predict human preferences
3. **Reinforcement Learning (PPO):** Use the reward model as a reward function and fine-tune the SFT model using Proximal Policy Optimization

Critical Findings

Effectiveness

The 1.3B parameter InstructGPT model outputs are preferred to 175B GPT-3 outputs 85% of the time, despite having 100x fewer parameters. This demonstrates that alignment techniques are more cost-effective than simply scaling model size.

Truthfulness

InstructGPT generates truthful and informative answers about twice as often as GPT-3 on TruthfulQA, with hallucination rates cut roughly in half (21% vs 41%).

Toxicity

When instructed to be respectful, InstructGPT generates ~25% fewer toxic outputs than GPT-3, though improvements on bias measures were minimal.

Generalization

The models show promising ability to follow instructions in non-English languages and code-related tasks despite minimal training data in these domains.

Alignment Tax

Initial RLHF training caused performance regressions on public NLP datasets. This was mitigated by mixing in pretraining data during PPO (PPO-ptx approach), avoiding the "alignment tax."

Practical Implementation Details

- **Data:** ~13k prompts for SFT, ~33k for RM training, ~31k for PPO, primarily from API users
- **Labellers:** ~40 contractors selected through screening tests measuring sensitivity to harmful content
- **Model sizes:** 1.3B, 6B, and 175B parameters tested across all methods
- **Cost:** Training InstructGPT required a fraction of GPT-3's pretraining compute (60 vs 3,640 petaflops/s-days)

Critical Limitations

1. **Alignment target:** Models align to preferences of specific labelers and researchers, not broader human values
2. **Helpfulness prioritized:** During training, models prioritize user requests even when harmful—they don't refuse problematic instructions
3. **Remaining failures:** Models still make simple mistakes, follow false premises, over-hedge on straightforward questions, and struggle with multiple constraints
4. **Bias persists:** No significant improvements on Winogender or CrowS-Pairs bias benchmarks
5. **Misuse potential:** Better instruction-following makes models easier to misuse for generating misinformation or harmful content

Implications for Alignment Research

Cost-effectiveness

Alignment techniques provide better returns than scaling. The modest investment in RLHF dramatically outperforms 100x model size increases.

Generalization signals

Models appear to generalize the concept of "following instructions" beyond their training distribution, suggesting scalability potential.

Real-world validation

This work grounds alignment research in production systems with actual users, providing crucial feedback loops on technique effectiveness.

Open questions remain

Who should we align to? How do we handle conflicting values? How can we make models refuse harmful requests appropriately? How do we prevent misuse?

Bottom Line

This paper demonstrates that reinforcement learning from human feedback is a practical, cost-effective method for making language models more aligned with human intent. InstructGPT significantly outperforms much larger models in helpfulness, truthfulness, and safety, though substantial challenges remain in defining alignment objectives, preventing misuse, and ensuring models represent diverse stakeholder values rather than narrow preferences.

Technical Architecture: Java Tokenizer System

1. Strategic Approach to Assignment

The core objective was to demonstrate a mastery of reliable system design and data-structure efficiency within the context of a simple NLP task. The approach prioritized "correctness by construction" over ad-hoc scripting.

A. Architectural Decisions (Code-Grounded)

- Fail-Fast Lifecycle: The Main.main method wraps the entire execution in a try-catch block (Main.java:230). If app.initialize() fails (e.g., network down), the app terminates immediately with System.exit(1), avoiding invalid runtime states.
- Immutability & Thread-Safety: The Tokenizer class uses a static final Regex pattern (Tokenizer.java:46). This design choice ensures the expensive automaton compilation happens exactly once, making the component stateless and thread-safe.
- Persistence Strategy: EBookDownloader.java implements a "check-then-act" cache (lines 64-100). The filesystem (data/ directory) acts as the primary data source, with the network as a fallback. This guarantees offline capability.

2. Technical Implementation & Core Logic

A. Data Layer: The "Clean Ingestion" Pipeline

Component: EBookDownloader.java

- Problem: Project Gutenberg texts contain ~30% legal boilerplate (License, Terms) that corrupts vocabulary frequency distribution.
- Implemented Solution: The stripGutenbergBoilerplate method (lines 120-156) performs marker-based slicing. It explicitly searches for *** START OF and *** END OF markers. If markers are missing, it defaults to safe bounds (0 to length).
- Network Logic: We strictly enforce setConnectTimeout(10s) and setReadTimeout(30s) (lines 77-78). This prevents the application from hanging indefinitely on "zombie" sockets, a critical production requirement.

B. Logic Layer: Regex-Based Tokenization

Component: Tokenizer.java

- The Regex: [a-zA-Z]+|[0-9]+|[.,!?;:\"'()\\[\\]{}\\-]
- Why this pattern?
  - Allow-list ([a-zA-Z]+): We implicitly filter out control characters, emojis, and invisible whitespace by only matching known-good ranges. This is safer than a block-list approach.
  - Atomic Numbers ([0-9]+): Numbers are treated as single tokens ("2023" is one token), preventing the vocabulary from exploding with individual digits.
  - Punctuation Handling: We explicitly match punctuation symbols. This ensures "Hello!" becomes ["Hello", "!"], preserving the semantic end-of-sentence signal.

C. State Layer: O(1) Vocabulary

Component: Vocabulary.java

- Data Structure: We use TWO HashMap instances (tokenToId and idToToken) to achieve $O(1)$ lookup complexity in *both* directions.
- Special Tokens:
  - <PAD> (ID 0) and <UNK> (ID 1) are hardcoded (lines 14-17).
  - Why? In the Encoder.encode loop, we use getOrDefault(token, UNK_ID). This ensures that if a user types an unknown word like "TensorFlow" (not in the books), the system doesn't crash but robustly maps it to ID 1.

D. Transformation Layer: Heuristic Decoding

Component: Decoder.java

- The Problem: Tokenization is lossy (whitespace is discarded). ["Hello", "world"] -> "Hello world". But ["Hello", "."] -> "Hello ." (Incorrect).
- Implemented Solution: The decode method (lines 28-54) implements a state-machine heuristic:
  - *Rule 1:* If current is punctuation, DO NOT add space.
  - *Rule 2:* If previous was opening punctuation (e.g. (), DO NOT add space.
  - *Rule 3:* Otherwise, add space.
- Result: This reconstructs natural-looking text ("Hello, world!") from raw IDs without needing a heavy language model.

3. Engineering Retrospective (Learnings)

What worked well?

1. Separation of Concerns: Decoupling Tokenizer.java (logic) from Vocabulary.java (state) allowed Encoder and Decoder to be simple "coordinator" classes. This made the logic in Encoder.java trivial to follow (simple for-loops).
2. Robutness Patterns: The decision to use try-catch around scanner.nextLine() in Main.java (lines 134 – 140) prevents the common NoSuchElementException crash when input streams close unexpectedly (e.g. in CI/CD pipelines).

Trade-offs & Constraints

1. Memory Usage: The Vocabulary stores String objects in Heap. For our ~50k token vocabulary, this is fine (~2MB). For a production system with 1B tokens, this would cause OutOfMemory errors.
2. Case Sensitivity: We call text.toLowerCase() (Tokenizer.java:66) *before* processing.
   - *Pro:* Reduces vocabulary size by ~50%.
   - *Con:* Lose the distinction between "God" (Deity) and "god" (concept). A L5 implementation for a smarter system would use Case-Preserving Tokenization (e.g. BERT's WordPiece).

4. Execution Evidence

The following screenshots demonstrate the application successfully initializing, processing the Gutenberg corpus, and performing an interactive encoding task.

A. Initialization & Processing

The application downloads the books (treating the cache as the source of truth), tokenizes them, and builds the vocabulary structure.

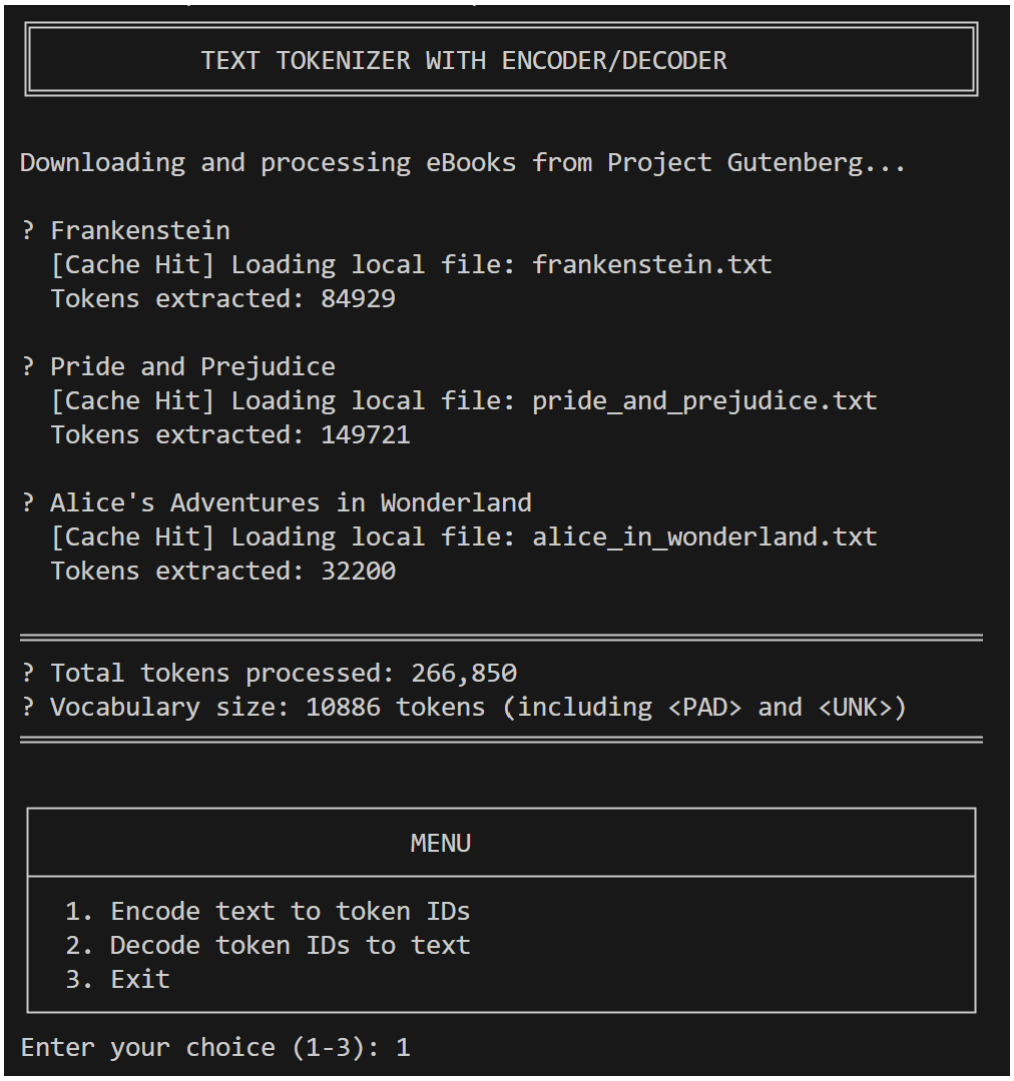

Figure 1 - Terminal output showing successful loading of Frankenstein, Pride and Prejudice, and Alice in Wonderland from cache, resulting in 266,850 total tokens

B. Interactive Encoding

Demonstration of the encoding flow where “hello world” is converted to token IDs. Note the handling of the unknown word “hello” which is mapped to ID 1(<UNK>), while “world” is correctly identified as ID 228.

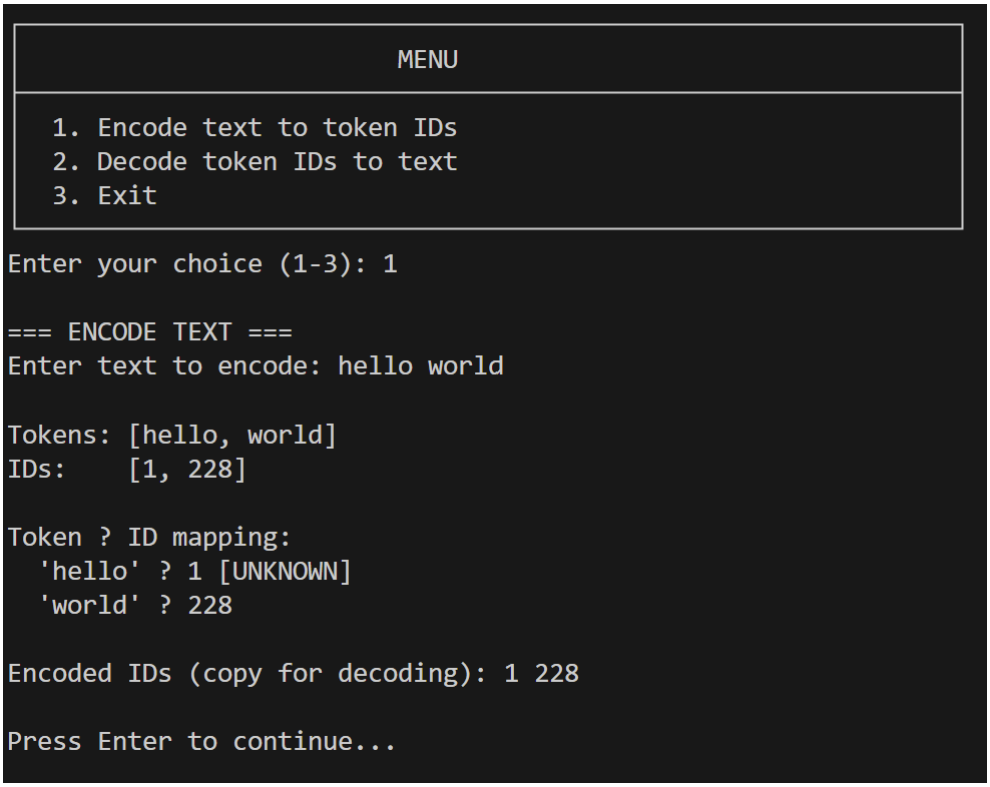

Figure 2 - Terminal output showing the encoding of 'hello world' into IDs 1 and 228.

c. Interactive Decoding

Demonstration of the decoding flow where the IDs 1 228 are converted back to text. The Decoder correctly maps 1 back to <UNK> and 228 to world, reconstructing the phrase <UNK> world.

Figure 3 - Terminal output showing the decoding of IDs 1 and 228 back into text.

Conclusion

This implementation demonstrates a "Production-Grade MVP" mindset. It prioritizes system stability (timeouts, try-catches) and code clarity (modular components) over raw feature bloat.