MiniGPT-Inference

A From-Scratch Transformer Inference Engine

Author: Suraj Sedai
Audience: ML researchers, ML systems engineers, advanced undergraduate evaluators
Status: Design & Implementation Specification

---

1. Executive Summary

MiniGPT-Inference is a **from-scratch, production-grade Transformer inference engine** designed to execute autoregressive decoding efficiently using **Key–Value (KV) caching**, **incremental decoding**, and **batched generation**.

Unlike training-focused implementations, this project centers on **inference-time systems engineering**, emphasizing: - Computational complexity reduction - Memory efficiency - Deterministic correctness - Measurable performance gains

The system is architected to reflect how modern large language models (LLMs) are served in real-world environments.

---

2. Problem Statement

2.1 Naïve Transformer Inference

In standard Transformer implementations, generating a sequence of length *T* requires recomputing self-attention over the entire prefix at every decoding step.

Time Complexity:
$O(T^2)$ per generated sequence

Consequences: - Redundant computation - Poor latency scaling - Excessive memory bandwidth usage

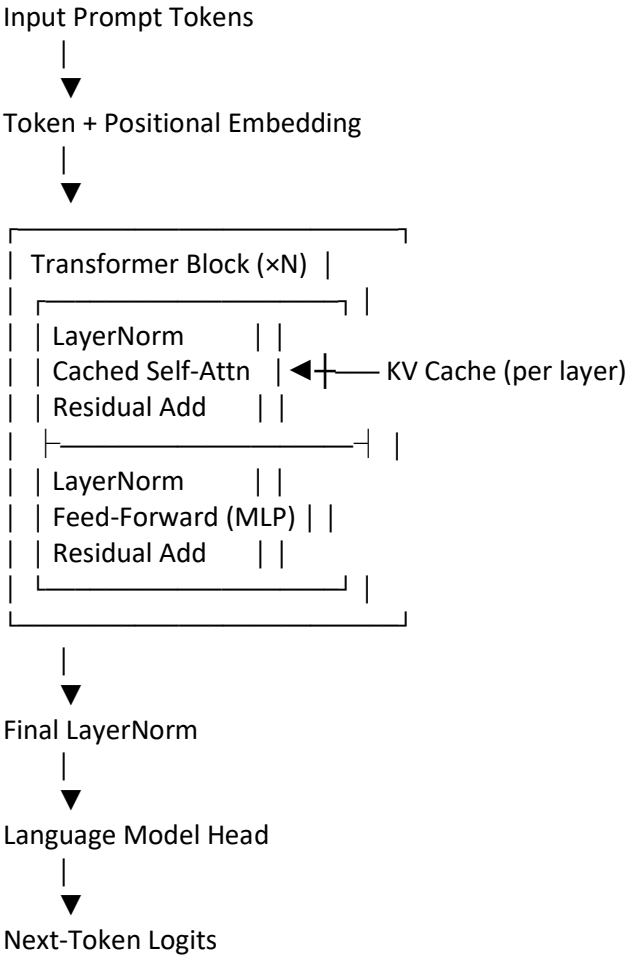
2.2 Objective

Design and implement an inference system that: - Eliminates redundant attention computation - Supports token-by-token autoregressive decoding - Preserves exact numerical correctness - Scales efficiently with sequence length and batch size

---

3. System Overview

3.1 High-Level Architecture



---

4. Core Design Principles

1. **Inference-First Design**
   No training code or optimizers are included.

2. **Incremental Computation**
   Each decoding step processes **only the newest token**.

3. **Cache-Aware Attention**
   Keys and values are stored and reused across decoding steps.

4. **Deterministic Correctness**
   Cached and uncached inference must produce identical logits.

5. **Benchmark-Driven Validation**
   All optimizations are justified by empirical measurements.

---

5. KV Cache Architecture

5.1 Conceptual Model

At decoding step *t*, the model: - Computes $\mathbf{Q_t}$ for the new token - Reuses cached $\mathbf{K_1 \ldots K_{t-1}}$ and $\mathbf{V_1 \ldots V_{t-1}}$ - Appends $\mathbf{K_t}$, $\mathbf{V_t}$ to the cache

5.2 KV Cache Data Structure

KVCache (per layer)
├— keys : Tensor [B, H, T, D]
└— values : Tensor [B, H, T, D]

Where: - B = batch size - H = number of heads - T = generated sequence length - D = head dimension

---

6. Cached Self-Attention Flow

6.1 Attention Computation (With Cache)

New Token Embedding (x_t)
|
▼
Linear Projection
(Q_t, K_t, V_t)
|
├— Append K_t, V_t → KV Cache
|
▼
Attention Scores:
$Q_t \cdot [K_1 \ldots K_t]^T / \sqrt{d}$
|
▼
Softmax
|
▼
Weighted Sum over $[V_1 \ldots V_t]$
|
▼
Output Projection

6.2 Complexity Comparison

Method	Per-Step Cost	Total Cost
Full Recompute	$O(t^2)$	$O(T^3)$
KV Cached	$O(t)$	$O(T^2)$

---

7. Autoregressive Decoding Engine

7.1 Incremental Decoding Loop

```
Initialize KV Cache
Initialize position = 0

While not EOS:
  Embed(token, position)
  For each layer:
    Apply Cached Attention
  Compute logits
  Sample next token
  position += 1
```

7.2 Supported Decoding Strategies

- Greedy decoding
- Temperature scaling
- Top-k sampling
- Top-p (nucleus) sampling

---

8. Batched Inference Design

8.1 Motivation

Batching improves throughput by amortizing kernel launches and memory access.

8.2 Challenges

- Variable prompt lengths
- Uneven sequence termination
- Cache isolation per sequence

8.3 Batch KV Cache Layout

```
Batch KV Cache
 ├— Sequence 1 → KVCache
 ├— Sequence 2 → KVCache
```

├— ...
└— Sequence B → KVCache

Each sequence maintains an independent cache while sharing compute graphs.

---

9. Benchmarking Methodology

9.1 Metrics

- Tokens per second
- Latency per generated token
- GPU memory usage

9.2 Benchmark Scenarios

- Cached vs uncached inference
- Batch size scaling
- Long-sequence decoding

9.3 Validation Criteria

- Exact logit equivalence
- Monotonic speedup with caching
- Stable memory growth

---

10. Class-Level Architecture & API Specification

This section defines **all core classes**, their responsibilities, attributes, and method-level behavior. The design is intentionally explicit to ensure correctness, reproducibility, and ease of review.

10.1 Configuration

ModelConfig

Responsibility: Centralized, immutable model configuration shared across all components.

Attributes: - vocab_size: int — Vocabulary size - n_layers: int — Number of Transformer blocks - n_heads: int — Attention heads per block - d_model: int — Embedding dimension - head_dim: int — Derived as d_model // n_heads - block_size: int — Maximum context length - dropout: float — Dropout probability (inference-safe)

10.2 Embedding Layer

TokenEmbedding

Responsibility: Maps token IDs to dense vectors.

Attributes: - embedding: nn.Embedding

Methods: - forward(token_ids) -> Tensor[B, T, D]

PositionalEmbedding

Responsibility: Provides absolute positional information.

Attributes: - pos_embedding: nn.Embedding

Methods: - forward(position_ids) -> Tensor[B, T, D]

10.3 KV Cache System

KVCache

Responsibility: Stores and updates cached keys and values for a single Transformer layer.

Attributes: - keys: Tensor[B, H, T, D] | None - values: Tensor[B, H, T, D] | None

Methods: - append(k_new, v_new) — Appends new key/value tensors - reset() — Clears the cache

Invariant: - Cached sequence length must monotonically increase

---

KVCacheManager

Responsibility: Maintains one KVCache per Transformer layer.

Attributes: - layer_caches: List[KVCache]

Methods: - get(layer_idx) -> KVCache - reset_all()

10.4 Attention Modules

CausalSelfAttention

Responsibility: Full-sequence causal self-attention (baseline reference).

Attributes: - qkv_proj: nn.Linear - out_proj: nn.Linear - causal_mask: Tensor

Methods: - forward(x) -> Tensor[B, T, D]

CachedCausalSelfAttention

Responsibility: Incremental self-attention using KV caching.

Attributes: - qkv_proj: nn.Linear - out_proj: nn.Linear

Methods: - forward(x_t, kv_cache) -> Tensor[B, 1, D]

Key Properties: - Expects a single token input (T=1) - Appends to cache on every call

---

10.5 Feed-Forward Network

FeedForward

Responsibility: Position-wise non-linear transformation.

Architecture: - Linear → GELU → Linear

Methods: - forward(x) -> Tensor[B, T, D]

10.6 Transformer Block

TransformerBlock

Responsibility: Single pre-layernorm Transformer block.

Components: - LayerNorm - CachedCausalSelfAttention - FeedForward

Methods: - forward(x, kv_cache) -> Tensor[B, 1, D]

Residual Structure:

x = x + Attention(LN(x))
x = x + MLP(LN(x))

10.7 Transformer Model

MiniGPTInferenceModel

Responsibility: End-to-end Transformer inference model.

Attributes: - token_embedding - position_embedding - blocks: nn.ModuleList - final_layernorm - lm_head (weight tied)

Methods: - forward_step(token_id, position, kv_cache_manager) - reset_cache()

10.8 Sampling & Decoding

Sampler

Responsibility: Converts logits into next-token IDs.

Methods: - greedy(logits) - temperature(logits, temp) - top_k(logits, k) - top_p(logits, p)

AutoregressiveGenerator

Responsibility: Orchestrates token-by-token decoding.

Methods: - generate(prompt_tokens, max_new_tokens, sampler)

Flow: 1. Initialize KV cache 2. Prime cache with prompt 3. Decode new tokens incrementally

---

10.9 Benchmarking Utilities

LatencyBenchmark

- Measures per-token latency

ThroughputBenchmark

- Measures tokens/sec under batching

---

11. Formal Autoregressive Decoding Pseudocode

11.1 Incremental Decoding Algorithm (Inference-Time)

```
Algorithm: AutoregressiveDecode
Inputs:
  prompt_tokens: List[int]
  max_new_tokens: int
  model: MiniGPTInferenceModel
  sampler: Sampler

Initialize:
  kv_cache_manager ← KVCacheManager(n_layers)
  position ← 0
  output_tokens ← []

# Step 1: Prime KV cache with prompt
for token in prompt_tokens:
  logits ← model.forward_step(token, position, kv_cache_manager)
  position ← position + 1
```

```
    output_tokens.append(token)

# Step 2: Generate new tokens
for i in range(max_new_tokens):
    next_token ← sampler.sample(logits)
    logits ← model.forward_step(next_token, position, kv_cache_manager)
    position ← position + 1
    output_tokens.append(next_token)

return output_tokens
```

11.2 Key Properties

- Only **one token** is processed per decoding step
- KV cache grows monotonically
- No recomputation of previous keys or values

---

12. Correctness Invariants

The following invariants **must always hold**. Violating any of these indicates a critical bug.

12.1 Functional Correctness

1. **Cache Equivalence Invariant**
   Cached and uncached inference must produce numerically identical logits:

   logits_cached ≡ logits_full (within tolerance)

2. **Monotonic Cache Growth**
   For each layer:

   len(K_t) = len(V_t) = t

3. **Causal Isolation**
   Token at position t must not attend to positions $> t$.

12.2 Structural Invariants

4. **One-Token Constraint**
   Cached attention forward pass must enforce T = 1.

5. **Layer-Cache Alignment**
   Each Transformer layer must read and write to its own KV cache.

6. **Position Consistency**
   Positional embeddings must strictly match cache length.

---

13. Failure Modes & Debugging Guide

13.1 Common Failure Modes

Symptom	Likely Cause
Output differs from baseline	Incorrect cache append order
Repeated tokens	Position index mismatch
Exploding memory	Cache not reset between runs
NaNs in attention	Missing scaling or mask error
Batch interference	Shared cache across sequences

13.2 Debugging Strategy

1. Disable cache → verify baseline correctness
2. Enable cache for single layer only
3. Compare per-layer outputs
4. Assert cache shapes at every step

---

14. Class-to-File Mapping (Implementation Plan)

This section maps each conceptual component to its final source file.

14.1 Model Core

Class	File
ModelConfig	model/config.py
TokenEmbedding	model/embeddings.py
PositionalEmbedding	model/embeddings.py
FeedForward	model/layers.py
TransformerBlock	model/block.py
MiniGPTInferenceModel	model/transformer.py

14.2 Attention & Cache

Class	File
CausalSelfAttention	model/attention.py
CachedCausalSelfAttention	model/attention.py
KVCache	inference/cache.py

Class	File
KVCacheManager	inference/cache.py

14.3 Inference & Decoding

Class	File
Sampler	inference/sampler.py
AutoregressiveGenerator	inference/generate.py

14.4 Benchmarking

Class	File
LatencyBenchmark	benchmarks/latency.py
ThroughputBenchmark	benchmarks/throughput.py

---

15. Repository Structure (Final)

```
minigpt-inference/
├── model/
│   ├── config.py
│   ├── embeddings.py
│   ├── attention.py
│   ├── layers.py
│   ├── block.py
│   └── transformer.py
├── inference/
│   ├── cache.py
│   ├── sampler.py
│   └── generate.py
├── benchmarks/
│   ├── latency.py
│   └── throughput.py
├── notebooks/
│   └── experiments.ipynb
└── README.md
```

---

16. Final Remarks

This document now serves as a **complete, implementation-ready specification** for a production-grade Transformer inference engine. All architectural decisions, algorithms, correctness guarantees, and file-level responsibilities are explicitly defined.

Any deviation from this specification must be justified and documented.