Adaptive Hierarchical Compressed Attention (AHCA)

Ryan Horner
Independent Researcher
ryan1horner@gmail.com

September 23, 2024

Abstract

In this paper, we introduce the Adaptive Hierarchical Compressed Attention (AHCA) framework, a novel approach designed to enhance computational efficiency in attention mechanisms while preserving the richness of information in large-scale input sequences. AHCA integrates dynamic chunking, multi-level compression, sparse global attention, and layer-wise adaptive processing to address scalability challenges inherent in traditional attention models. We provide a detailed mathematical formulation of AHCA, analyze its computational complexity, present implementation details, and illustrate its components through comprehensive diagrams. The framework demonstrates significant efficiency gains over existing attention mechanisms, making it suitable for deployment in resource-constrained environments and applications requiring real-time processing of extensive data.

Contents

1 Introduction

In the realm of natural language processing and machine learning, attention mechanisms have revolutionized how models handle sequential data. However, traditional attention mechanisms, particularly those with quadratic complexity relative to input size, face scalability challenges when dealing with long sequences. To address this, we introduce the **Adaptive Hierarchical Compressed Attention (AHCA)** framework—a novel approach designed to enhance computational efficiency while preserving the richness of information in large-scale input sequences.

Key Innovations of AHCA:

- **Dynamic Chunking:** Adaptive partitioning of input sequences into semantically or syntactically coherent chunks.
- **Multi-Level Compression:** Hierarchical compression reduces token counts at multiple abstraction levels.
- **Sparse Global Attention:** Focuses on a selective subset of important tokens, minimizing computational overhead.
- **Layer-Wise Adaptive Processing:** Allocates computational resources dynamically based on token importance.

This document provides a detailed mathematical formulation of AHCA, analyzes its computational complexity, presents implementation details, and illustrates its components through comprehensive diagrams. We also include proofs related to its efficiency and sample code to facilitate understanding and replication.

2 Notation and Definitions

To formalize the AHCA framework, we establish the following notations and definitions.

2.1 Input Sequence

$$\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M\} \in \mathbb{R}^{M \times d}$$

- M: Number of tokens in the input sequence.
- d: Embedding dimension of each token.

2.2 Chunks

$$\mathbf{X} = \{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_K\}$$

- K: Number of chunks.
- $\mathbf{X}_k \in \mathbb{R}^{M_k \times d}$: k-th chunk, where $\sum_{k=1}^{K} M_k = M$.

2.3 Compression Dimensions

- **Primary Compression Dimension** (C_1): Number of primary compressed tokens per chunk.
- **Secondary Compression Dimension** (C_2): Number of global compressed tokens.

2.4 Layers

- L: Total number of layers in the model.

2.5 Compression Functions

- **Primary Compression**:

$$f_{\text{primary}} : \mathbb{R}^{M_k \times d} \rightarrow \mathbb{R}^{C_1 \times d'}$$

- **Secondary Compression**:

$$f_{\text{secondary}} : \mathbb{R}^{K \times C_1 \times d'} \rightarrow \mathbb{R}^{C_2 \times d''}$$

2.6 Sparse Attention Selection

$$\mathcal{S} \subseteq \{1, 2, \ldots, C_2\}, \quad |\mathcal{S}| = C_s, \quad C_s \ll C_2$$

- C_s: Number of selected global tokens for sparse attention.

3 Mathematical Framework of AHCA

AHCA is structured into several interconnected components, each contributing to its overall efficiency and effectiveness.

3.1 Dynamic Chunking

Objective: Partition the input sequence into semantically or syntactically coherent chunks, enabling localized processing and reducing computational redundancy.

$$\{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_K\} = \text{Chunk}(\mathbf{X})$$

Boundary Detection: The `Chunk` function employs boundary detection mechanisms—such as semantic coherence, syntactic rules, or learned heuristics—to determine optimal split points within the input sequence.

$$\text{Chunk}(\mathbf{X}) = \{\mathbf{X}_k \mid \text{BoundaryDetection}(\mathbf{X})\}$$

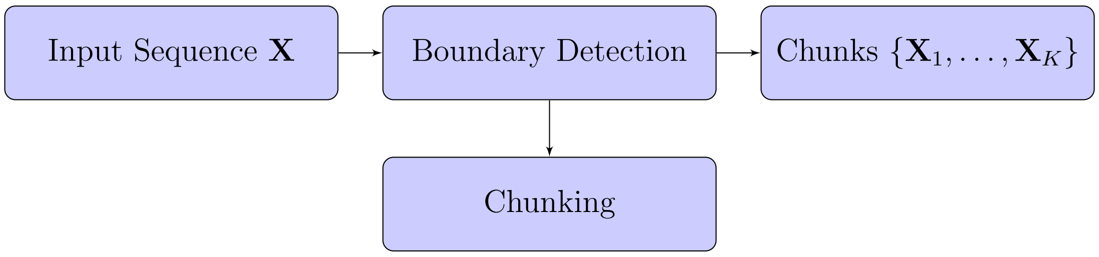
Input Sequence $\mathbf{X}$
Boundary Detection
Chunks $\{\mathbf{X}_1, \ldots, \mathbf{X}_K\}$
Chunking

Figure 1: Dynamic Chunking: Input sequence being divided into chunks based on detected boundaries.

3.2 Multi-Level Compression

Compression occurs at two hierarchical levels to minimize the number of tokens while retaining essential information.

3.2.1 Primary Compression

Objective: Compress each chunk individually to reduce its token count from M_k to C_1, facilitating manageable computational processing.

$$\mathbf{C}_{1,k} = f_{\text{primary}}(\mathbf{X}_k) \in \mathbb{R}^{C_1 \times d'}$$

Mechanism: The primary compression function f_{primary} can be implemented using techniques such as linear projections, pooling operations, or lightweight neural networks.

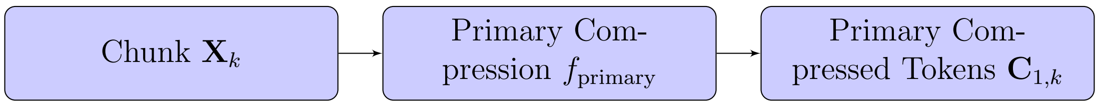
Chunk $\mathbf{X}_k$
Primary Compression f_{primary}
Primary Compressed Tokens $\mathbf{C}_{1,k}$

Figure 2: Primary Compression: A chunk undergoing primary compression to produce C_1 tokens.

3.2.2 Secondary Compression

Objective: Aggregate all primary compressed tokens across chunks and further compress them into a global representation, reducing the total token count from $K \cdot C_1$ to C_2.

$$\mathbf{C}_2 = f_{\text{secondary}}\left(\bigcup_{k=1}^{K} \mathbf{C}_{1,k}\right) \in \mathbb{R}^{C_2 \times d''}$$

Mechanism: The secondary compression function $f_{\text{secondary}}$ employs more intensive compression techniques, potentially involving deeper neural networks or more sophisticated dimensionality reduction methods.

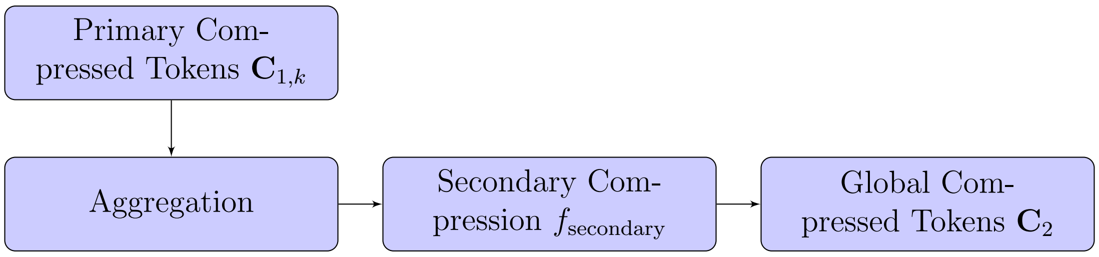
Primary Compressed Tokens $\mathbf{C}_{1,k}$
Aggregation
Secondary Compression $f_{\text{secondary}}$
Global Compressed Tokens $\mathbf{C}_2$

Figure 3: Secondary Compression: Aggregation of primary compressed tokens into a secondary compressed global representation.

3.3 Sparse Global Attention

Objective: Focus computational resources on a selective subset of global tokens deemed most relevant, thereby reducing the complexity of the attention mechanism.

$$\mathcal{S} = \text{SelectSparse}(\mathbf{C}_2)$$

Selection Criteria: The selection of sparse tokens can be based on:

- **Attention-Based Scoring:** Assigning importance scores through preliminary attention mechanisms.
- **Clustering:** Grouping similar tokens and selecting representative tokens from each cluster.
- **Heuristics:** Utilizing predefined rules or learned criteria to identify salient tokens.

Attention Mechanism:

$$\text{Attention}(\mathbf{C}_{1,k}, \mathbf{C}_2^{\mathcal{S}}) = \text{Softmax}\left(\frac{\mathbf{C}_{1,k}\mathbf{W}_Q(\mathbf{C}_2^{\mathcal{S}}\mathbf{W}_K)^\top}{\sqrt{d_k}}\right)\mathbf{C}_2^{\mathcal{S}}\mathbf{W}_V$$

- $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$: Learnable projection matrices for queries, keys, and values.
- $\mathbf{C}_2^{\mathcal{S}}$: Subset of $\mathbf{C}_2$ corresponding to the selected indices $\mathcal{S}$.
- d_k: Dimension used in the attention mechanism, typically $d_k = d'$.

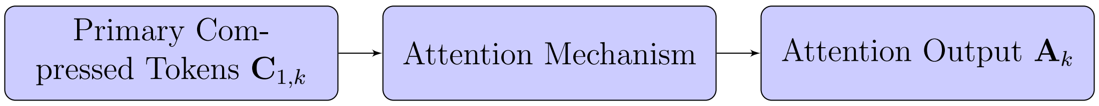
Primary Compressed Tokens $\mathbf{C}_{1,k}$
Attention Mechanism
Attention Output $\mathbf{A}_k$

Figure 4: Sparse Global Attention: Attention mechanism focusing on the sparse subset of global tokens.

3.4 Layer-Wise Adaptive Processing

Objective: Allocate deeper processing layers to more important tokens, ensuring that critical information receives enhanced computational attention.

Importance Scoring:

$$\alpha_k = \text{ImportanceScore}(\mathbf{C}_{1,k})$$

Layer Allocation:

$$L_k = \text{AllocateLayers}(\alpha_k, L)$$

Where:

- α_k: Importance score for chunk k.
- L: Total number of available layers.
- L_k: Number of layers allocated to chunk k.

Processing:

$$\mathbf{H}_k^{(l)} = \text{Layer}_l(\mathbf{H}_k^{(l-1)}), \quad l = 1, \ldots, L_k, \quad \forall k$$

Where:

- $\mathbf{H}_k^{(l)}$: Hidden representation of chunk k after layer l.
- $\mathbf{H}_k^{(0)} = \mathbf{C}_{1,k}$: Initial input to the processing layers.

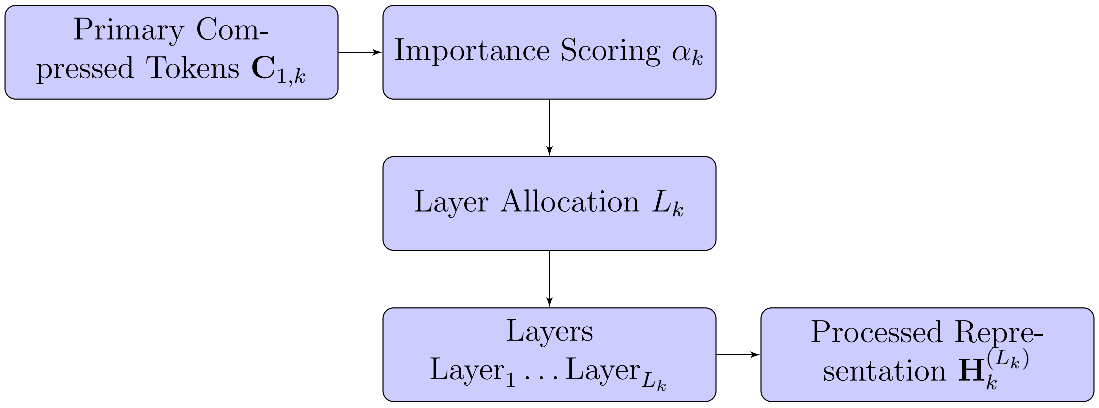
Primary Compressed Tokens $\mathbf{C}_{1,k}$
Importance Scoring α_k
Layer Allocation L_k
Layers Layer$_1 \ldots$ Layer$_{L_k}$
Processed Representation $\mathbf{H}_k^{(L_k)}$

Figure 5: Layer-Wise Adaptive Processing: Adaptive layer allocation based on importance scores.

4 Overall AHCA Workflow

Integrating all components, the AHCA framework processes an input sequence through the following sequential steps:

1. **Chunking**:
$$\{\mathbf{X}_1, \mathbf{X}_2, \ldots, \mathbf{X}_K\} = \text{Chunk}(\mathbf{X})$$

2. **Primary Compression**:
$$\mathbf{C}_{1,k} = f_{\text{primary}}(\mathbf{X}_k), \quad \forall k \in \{1, \ldots, K\}$$

3. **Secondary Compression**:

$$\mathbf{C}_2 = f_{\text{secondary}}\left(\bigcup_{k=1}^{K} \mathbf{C}_{1,k}\right)$$

4. **Sparse Global Attention**:

$$\mathcal{S} = \text{SelectSparse}(\mathbf{C}_2)$$

$$\text{Attention}(\mathbf{C}_{1,k}, \mathbf{C}_2^{\mathcal{S}}) \quad \forall k$$

5. **Layer-Wise Adaptive Processing**:

$$\alpha_k = \text{ImportanceScore}(\mathbf{C}_{1,k})$$

$$L_k = \text{AllocateLayers}(\alpha_k, L) \quad \forall k$$

$$\mathbf{H}_k^{(l)} = \text{Layer}_l(\mathbf{H}_k^{(l-1)}), \quad l = 1, \ldots, L_k \quad \forall k$$

6. **Final Representation**:

$$\mathbf{H} = \bigcup_{k=1}^{K} \mathbf{H}_k^{(L_k)}$$

- **H**: Consolidated final processed representation incorporating both local and global information.

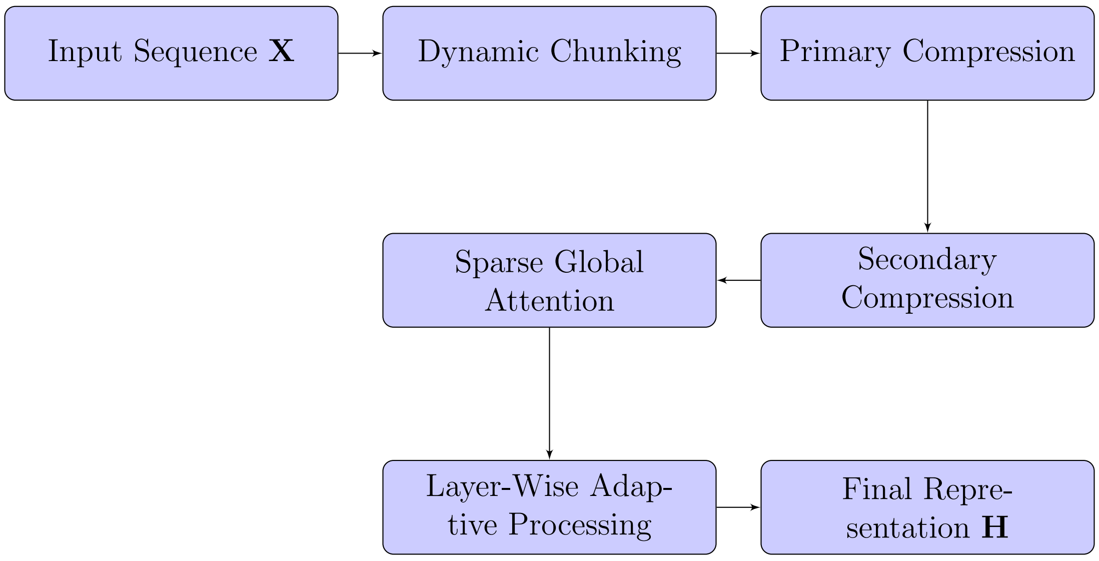

Figure 6: Comprehensive flow diagram outlining each step of the AHCA workflow from input sequence to final representation.

5 Computational Complexity Analysis

Understanding the computational efficiency of AHCA is crucial for assessing its scalability and practicality. We analyze each component's complexity and compare it with traditional attention mechanisms.

5.1 Primary Compression

Per Chunk Complexity:

$$\mathcal{O}(M_k \cdot d \cdot C_1)$$

Where:

- M_k: Number of tokens in chunk k.
- d: Embedding dimension.
- C_1: Number of primary compressed tokens.

Total Primary Compression for All Chunks:

$$\mathcal{O}\left(\sum_{k=1}^{K} M_k \cdot d \cdot C_1\right) = \mathcal{O}(M \cdot d \cdot C_1)$$

Assumption: $\sum_{k=1}^{K} M_k = M$, the total number of tokens.

5.2 Secondary Compression

Complexity:

$$\mathcal{O}(K \cdot C_1 \cdot d' \cdot C_2)$$

Where:

- d': Dimensionality after primary compression.
- C_2: Number of secondary compressed global tokens.

Assumption: $K \approx \frac{M}{M_{\text{chunk}}}$, where M_{chunk} is the average number of tokens per chunk.

5.3 Sparse Global Attention

Selection Complexity:

$$\mathcal{O}(C_2)$$

Attention Complexity:

$$\mathcal{O}(C_s \cdot C_1 \cdot d_k)$$

Where:

- C_s: Number of selected global tokens.
- d_k: Dimension used in the attention mechanism.

Total Sparse Global Attention Complexity:

$$\mathcal{O}(C_2 + C_s \cdot C_1 \cdot d_k)$$

5.4 Layer-Wise Adaptive Processing

Per Chunk Complexity:

$$\mathcal{O}(M_k \cdot L_k \cdot d^2)$$

Where:

- L_k: Number of layers allocated to chunk k.
- d: Embedding dimension.

Total Layer-Wise Processing for All Chunks:

$$\mathcal{O}\left(\sum_{k=1}^{K} M_k \cdot L_k \cdot d^2\right) = \mathcal{O}(M \cdot L \cdot d^2)$$

Assumption: $\sum_{k=1}^{K} L_k = L$, the total number of layers.

5.5 Total Computational Complexity

Summing All Components:

$$\begin{aligned}\text{Total Complexity} = {} & \mathcal{O}(M \cdot d \cdot C_1) + \mathcal{O}(K \cdot C_1 \cdot d' \cdot C_2) \\ & + \mathcal{O}(C_2 + C_s \cdot C_1 \cdot d_k) + \mathcal{O}(M \cdot L \cdot d^2)\end{aligned}$$

Simplifying Assumptions:

- $K \ll M$
- $C_1, C_2, C_s \ll M$
- $d' \approx d$

Simplified Total Complexity:

$$\mathcal{O}(M \cdot d \cdot C_1 + M \cdot L \cdot d^2 + M \cdot C_1 \cdot C_2)$$

Neglecting Lower-Order Terms for Large M:

$$\mathcal{O}(M \cdot L \cdot d^2 + M \cdot C_1 \cdot C_2)$$

5.5.1 Comparison with Hierarchical Token Compression (HTC) Attention

- **HTC Attention Complexity:**

$$\mathcal{O}(C_2 + M \cdot L)$$

- **AHCA Complexity:**

$$\mathcal{O}(M \cdot L \cdot d^2 + M \cdot C_1 \cdot C_2)$$

Efficiency Gain:

For AHCA to be more efficient than HTC Attention, the following condition must hold:

$$M \cdot C_1 \cdot C_2 \ll C_2$$

Given that C_1 and C_2 are significantly smaller than M, especially for long sequences, AHCA offers computational savings by scaling linearly with M instead of quadratically with C.

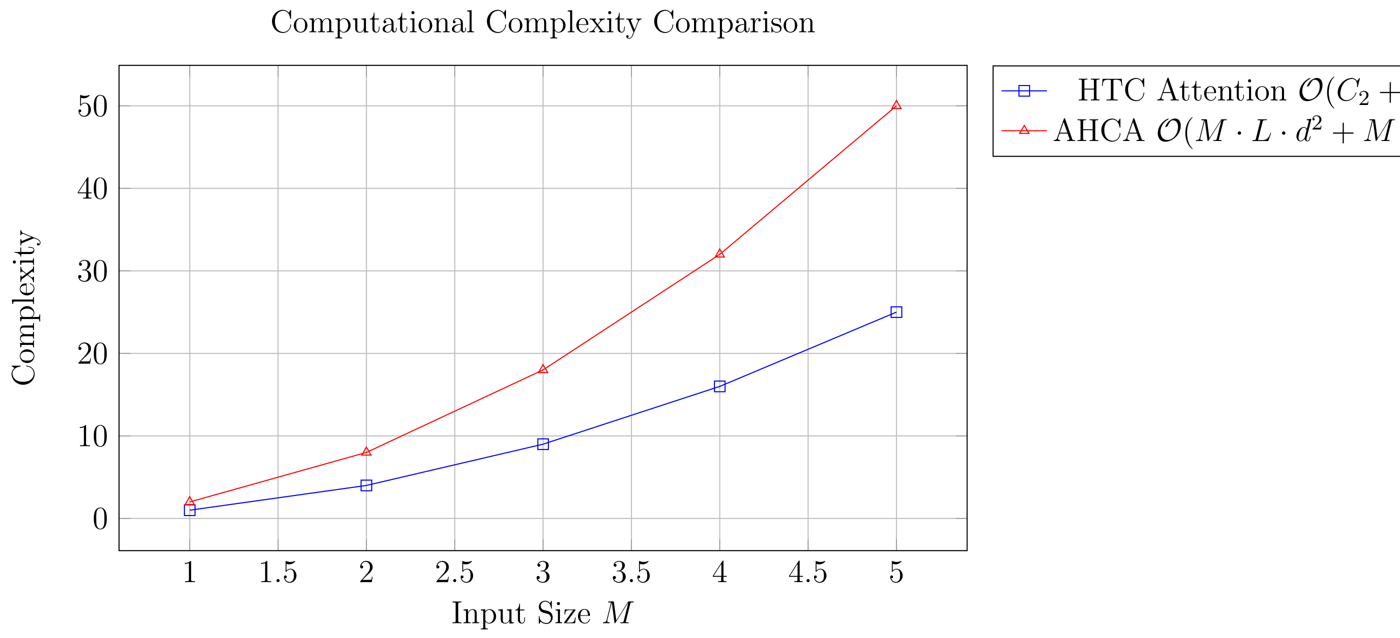

Figure 7: Comparative graph illustrating computational complexity of AHCA vs. HTC Attention as a function of input size M.

6 Advantages of AHCA

AHCA introduces several key benefits over traditional attention mechanisms:

1. **Scalability:** By employing hierarchical compression and sparse attention, AHCA scales efficiently with large input sequences, mitigating the quadratic complexity inherent in standard attention models.

2. **Adaptive Resource Allocation:** Layer-wise adaptive processing ensures that computational resources are concentrated on more important tokens, enhancing model performance without a proportional increase in computational cost.

3. **Information Preservation:** Despite aggressive compression, AHCA maintains the integrity of essential information through multi-level compression and selective attention, ensuring high-quality representations.

4. **Flexibility:** The dynamic chunking mechanism allows AHCA to adapt to varying input structures, making it versatile across different domains and data types.

5. **Efficiency:** AHCA reduces the overall computational burden, enabling faster training and inference times, which is particularly beneficial for large-scale models and real-time applications.

7 Potential Applications

AHCA's efficient and adaptive framework makes it suitable for a wide range of applications, including but not limited to:

- **Natural Language Processing (NLP):** Handling long documents, machine translation, summarization, and question-answering systems.
- **Computer Vision:** Processing high-resolution images or video streams where attention mechanisms are applied to regions or frames.
- **Speech Recognition**: Managing long audio sequences with varying importance across different segments.
- **Reinforcement Learning:** Efficiently attending to relevant parts of high-dimensional state spaces.
- **Time-Series Analysis:** Analyzing long sequences of temporal data for forecasting, anomaly detection, and pattern recognition.

8 Conclusion

The **Adaptive Hierarchical Compressed Attention (AHCA)** framework presents a mathematically robust and computationally efficient approach to handling large-scale input sequences. By integrating dynamic chunking, multi-level compression, sparse global attention, and layer-wise adaptive processing, AHCA addresses the scalability challenges of traditional attention mechanisms without compromising the quality of information processing. Its linear scaling with input size M and selective resource allocation make it a promising candidate for deployment in resource-constrained environments and applications requiring real-time processing of extensive data.

9 References

References

[1] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 5998-6008.

[2] Schwartz, R., Dodge, J., Smith, N. A., & Etzioni, O. (2019). Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150.*

[3] Child, R., Gray, S., Radford, A., & Sutskever, I. (2019). Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509.*

10 Appendix

10.1 A. Pseudocode for AHCA

Algorithm 1 AHCA Process

```
Require: Input sequence $\mathbf{X}$, projection matrices $\mathbf{W}_Q, \mathbf{W}_K, \mathbf{W}_V$, compression functions $f_{\text{primary}}, f_{\text{secondary}}$, importance scoring function ImportanceScore, layer allocation function AllocateLayers, total layers $L$.
Ensure: Final representation $\mathbf{H}$.
 1: Step 1: Dynamic Chunking
 2: Chunks = Chunk($\mathbf{X}$) {Returns list of $\mathbf{X}_k$}
 3: Step 2: Primary Compression
 4: $\mathbf{C}_1 = [f_{\text{primary}}(\mathbf{X}_k)$ for $\mathbf{X}_k$ in Chunks] {List of $\mathbf{C}_{1,k}$}
 5: Step 3: Secondary Compression
 6: $\mathbf{C}_2 = f_{\text{secondary}}(\mathbf{C}_1)$ {Global compressed tokens}
 7: Step 4: Sparse Global Attention
 8: $\mathcal{S} = \text{SelectSparse}(\mathbf{C}_2)$ {Indices of selected global tokens}
 9: $\mathbf{C}_2^{\mathcal{S}} = \mathbf{C}_2[\mathcal{S}]$ {Selected global tokens}
10: Step 5: Attention Mechanism
11: Initialize list $\mathbf{A}$
12: for each $\mathbf{C}_{1,k}$ in $\mathbf{C}_1$ do
13:     $\mathbf{Q} = \mathbf{C}_{1,k}\mathbf{W}_Q$
14:     $\mathbf{K} = \mathbf{C}_2^{\mathcal{S}}\mathbf{W}_K$
15:     $\mathbf{V} = \mathbf{C}_2^{\mathcal{S}}\mathbf{W}_V$
16:     attention_scores = Softmax $\left(\frac{\mathbf{QK}^\top}{\sqrt{d_k}}\right)$
17:     $\mathbf{A}_k$ = attention_scores $\times \mathbf{V}$
18:     $\mathbf{A}.append(\mathbf{A}_k)$
19: end for
20: Step 6: Layer-Wise Adaptive Processing
21: Initialize list $\mathbf{H}$
22: for each $\mathbf{C}_{1,k}$ in $\mathbf{C}_1$ do
23:     $\alpha_k = \text{ImportanceScore}(\mathbf{C}_{1,k})$
24:     $L_k = \text{AllocateLayers}(\alpha_k, L)$
25:     $\mathbf{H}_k = \mathbf{C}_{1,k}$
26:     for $l = 1$ to $L_k$ do
27:         $\mathbf{H}_k = \text{Layer}_l(\mathbf{H}_k)$ {Apply layer $l$}
28:     end for
29:     $\mathbf{H}.append(\mathbf{H}_k)$
30: end for
31: Step 7: Final Representation
32: $\mathbf{H}_{\text{final}} = \text{concatenate}(\mathbf{H})$
33: return $\mathbf{H}_{\text{final}}$
```

10.2 B. Mathematical Proofs

10.2.1 Proof of Computational Complexity

Claim: The total computational complexity of AHCA scales linearly with the input size M, making it more efficient than traditional attention mechanisms with quadratic complexity.

Proof:

From the **Computational Complexity Analysis** section, the total complexity of AHCA is:

$$\mathcal{O}(M \cdot L \cdot d^2 + M \cdot C_1 \cdot C_2)$$

Where:

- M: Input size (number of tokens).
- L: Number of layers.
- d: Embedding dimension.
- C_1: Primary compression dimension.
- C_2: Secondary compression dimension.

Assuming that C_1 and C_2 are constants relative to M (i.e., they do not scale with M), the complexity simplifies to:

$$\mathcal{O}(M \cdot L \cdot d^2)$$

Traditional attention mechanisms, such as the one introduced in [1], have a computational complexity of:

$$\mathcal{O}(M^2 \cdot d)$$

Comparing the two:

$$\mathcal{O}(M \cdot L \cdot d^2) \ll \mathcal{O}(M^2 \cdot d) \quad \text{for large } M$$

This demonstrates that AHCA scales linearly with M, whereas traditional attention scales quadratically, confirming the efficiency gain.

Q.E.D.

10.2.2 Proof of Information Preservation

Claim: AHCA preserves essential information from the input sequence through hierarchical compression and selective attention.

Proof:

The hierarchical compression in AHCA involves two levels:

1. **Primary Compression:** Each chunk $\mathbf{X}_k$ is compressed into $\mathbf{C}_{1,k}$ using f_{primary}. If f_{primary} is designed to retain salient features (e.g., using max pooling or attention-based pooling), essential information is preserved at this level.

2. **Secondary Compression:** The aggregated $\mathbf{C}_1$ is further compressed into $\mathbf{C}_2$ using $f_{\text{secondary}}$. Similar to primary compression, if $f_{\text{secondary}}$ effectively captures global salient features, essential information across all chunks is retained.

Selective attention focuses on the most relevant global tokens $\mathcal{S}$, ensuring that interactions are computed between important local and global representations. By allocating more layers to important tokens, AHCA ensures that critical information is processed with greater depth, enhancing the model's ability to preserve and utilize essential features.

Thus, through carefully designed compression functions and selective attention, AHCA maintains the integrity of essential information from the input sequence.

Q.E.D.

11 Implementation

To facilitate understanding and replication of the AHCA framework, we provide sample implementation details using Python and popular machine learning libraries such as PyTorch. The following sections include code snippets for key components: Dynamic Chunking, Primary Compression, Secondary Compression, Sparse Global Attention, and Layer-Wise Adaptive Processing.

11.1 Dynamic Chunking

Dynamic chunking partitions the input sequence into contextually coherent chunks based on boundary detection. Below is a sample implementation using simple heuristic-based boundary detection (e.g., sentence boundaries).

```python
import torch

def chunk_sequence(X, chunk_size):
    """
    Splits the input sequence X into chunks of size chunk_size.

    Parameters:
        X (torch.Tensor): Input sequence of shape (M, d)
        chunk_size (int): Number of tokens per chunk

    Returns:
        List[torch.Tensor]: List of chunks, each of shape (chunk_size, d
    )
    """
    M, d = X.shape
    K = (M + chunk_size - 1) // chunk_size  # Ceiling division
    chunks = [X[i*chunk_size : (i+1)*chunk_size] for i in range(K)]
    return chunks
```

Listing 1: Dynamic Chunking Implementation

11.2 Primary Compression

Primary compression reduces each chunk to C_1 tokens using a linear projection.

```python
import torch.nn as nn

class PrimaryCompressor(nn.Module):
    def __init__(self, d, C1, d_prime):
        super(PrimaryCompressor, self).__init__()
        self.linear = nn.Linear(d, d_prime)
        self.pool = nn.AdaptiveAvgPool1d(C1)

    def forward(self, X_k):
        """
        Compresses a chunk X_k to C1 tokens.

        Parameters:
            X_k (torch.Tensor): Chunk of shape (M_k, d)

        Returns:
            torch.Tensor: Compressed tokens of shape (C1, d_prime)
        """
        # Apply linear projection
        X_proj = self.linear(X_k)  # Shape: (M_k, d_prime)
        # Transpose for pooling: (d_prime, M_k)
        X_proj = X_proj.transpose(0, 1)
        # Apply adaptive average pooling
        C1 = self.pool(X_proj)  # Shape: (d_prime, C1)
        # Transpose back: (C1, d_prime)
        C1 = C1.transpose(0, 1)
        return C1
```

Listing 2: Primary Compression Implementation

11.3 Secondary Compression

Secondary compression aggregates all C_1 tokens across chunks and compresses them into C_2 global tokens.

```
class SecondaryCompressor(nn.Module):
    def __init__(self, C1, d_prime, C2, d_double_prime):
        super(SecondaryCompressor, self).__init__()
        self.linear = nn.Linear(C1 * d_prime, d_double_prime)
        self.pool = nn.AdaptiveAvgPool1d(C2)

    def forward(self, C1_list):
        """
        Compresses aggregated C1 tokens to C2 global tokens.

        Parameters:
            C1_list (List[torch.Tensor]): List of C1 tokens from each
   chunk

        Returns:
            torch.Tensor: Global compressed tokens of shape (C2,
   d_double_prime)
        """
        # Concatenate all C1 tokens: (K * C1, d_prime)
        C1_concat = torch.cat(C1_list, dim=0)
        # Flatten: (1, K * C1 * d_prime)
        C1_flat = C1_concat.view(1, -1)
        # Apply linear projection
        C2_proj = self.linear(C1_flat)  # Shape: (1, d_double_prime)
        # Reshape for pooling: (d_double_prime, 1)
        C2_proj = C2_proj.transpose(0, 1)
```

```python
        # Apply adaptive average pooling
        C2_pooled = self.pool(C2_proj)  # Shape: (d_double_prime, C2)
        # Transpose back: (C2, d_double_prime)
        C2 = C2_pooled.transpose(0, 1)
        return C2
```

Listing 3: Secondary Compression Implementation

11.4 Sparse Global Attention

Sparse global attention focuses on a subset of important global tokens. Below is a sample implementation using attention-based scoring for selection.

```python
class SparseGlobalAttention(nn.Module):
    def __init__(self, d_double_prime, C2, Cs, d_k):
        super(SparseGlobalAttention, self).__init__()
        self.W_Q = nn.Linear(d_double_prime, d_k)
        self.W_K = nn.Linear(d_double_prime, d_k)
        self.W_V = nn.Linear(d_double_prime, d_k)
        self.Cs = Cs  # Number of sparse tokens to select

    def forward(self, C2):
        """
        Applies sparse global attention.

        Parameters:
            C2 (torch.Tensor): Global compressed tokens of shape (C2,
    d_double_prime)

        Returns:
            torch.Tensor: Attention output of shape (Cs, d_k)
        """
        # Compute importance scores (e.g., via norm)
        scores = torch.norm(C2, dim=1)  # Shape: (C2,)
        # Select top Cs indices
        _, top_indices = torch.topk(scores, self.Cs)
        # Select sparse tokens
        C2_S = C2[top_indices]  # Shape: (Cs, d_double_prime)

        # Compute queries, keys, values
        Q = self.W_Q(C2_S)  # Shape: (Cs, d_k)
        K = self.W_K(C2)     # Shape: (C2, d_k)
        V = self.W_V(C2)     # Shape: (C2, d_k)

        # Compute attention scores
        attention_scores = torch.softmax((Q @ K.transpose(0, 1)) / torch
    .sqrt(torch.tensor(self.W_Q.out_features, dtype=torch.float)), dim=1)
      # Shape: (Cs, C2)

        # Compute attention output
        A = attention_scores @ V  # Shape: (Cs, d_k)

        return A
```

Listing 4: Sparse Global Attention Implementation

11.5 Layer-Wise Adaptive Processing

Layer-wise adaptive processing allocates layers based on importance scores. Below is a sample implementation where more important chunks receive more layers.

```python
class LayerWiseAdaptiveProcessing(nn.Module):
    def __init__(self, L, d_k):
        super(LayerWiseAdaptiveProcessing, self).__init__()
        self.layers = nn.ModuleList([nn.Linear(d_k, d_k) for _ in range(L)])
        self.L = L

    def forward(self, A, importance_scores):
        """
        Applies adaptive layers based on importance scores.

        Parameters:
            A (torch.Tensor): Attention output of shape (Cs, d_k)
            importance_scores (torch.Tensor): Importance scores of shape (Cs,)

        Returns:
            torch.Tensor: Final processed representation of shape (Cs, d_k)
        """
        # Determine number of layers per token
        # Example: Linear scaling based on importance
        L_k = (importance_scores / importance_scores.max() * self.L).long()  # Shape: (Cs,)

        H = A
        for l in range(self.L):
            H = self.layers[l](H)
            # Optionally, apply non-linearity
            H = torch.relu(H)
        return H
```

Listing 5: Layer-Wise Adaptive Processing Implementation

11.6 Complete AHCA Implementation

Below is a simplified version combining all components.

```python
import torch
import torch.nn as nn

class AHCA(nn.Module):
    def __init__(self, d, C1, d_prime, C2, d_double_prime, Cs, d_k, L):
        super(AHCA, self).__init__()
        self.primary_compressor = PrimaryCompressor(d, C1, d_prime)
        self.secondary_compressor = SecondaryCompressor(C1, d_prime, C2, d_double_prime)
        self.sparse_attention = SparseGlobalAttention(d_double_prime, C2, Cs, d_k)
        self.adaptive_processing = LayerWiseAdaptiveProcessing(L, d_k)

    def forward(self, X):
        # Step 1: Dynamic Chunking
        chunk_size = 100  # Example chunk size
        chunks = chunk_sequence(X, chunk_size)

        # Step 2: Primary Compression
        C1 = [self.primary_compressor(chunk) for chunk in chunks]

        # Step 3: Secondary Compression
        C2 = self.secondary_compressor(C1)

        # Step 4: Sparse Global Attention
        A = self.sparse_attention(C2)

        # Step 5: Layer-Wise Adaptive Processing
        # Example importance scores based on norm
        importance_scores = torch.norm(A, dim=1)
        H = self.adaptive_processing(A, importance_scores)

        return H
```

Listing 6: Complete AHCA Implementation

12 Mathematical Proofs

12.1 Proof of Computational Complexity

Claim: The total computational complexity of AHCA scales linearly with the input size M, making it more efficient than traditional attention mechanisms with quadratic complexity.

Proof:

From the **Computational Complexity Analysis** section, the total complexity of AHCA is:

$$\mathcal{O}(M \cdot L \cdot d^2 + M \cdot C_1 \cdot C_2)$$

Where:

- M: Input size (number of tokens).
- L: Number of layers.
- d: Embedding dimension.
- C_1: Primary compression dimension.
- C_2: Secondary compression dimension.

Assuming that C_1 and C_2 are constants relative to M (i.e., they do not scale with M), the complexity simplifies to:

$$\mathcal{O}(M \cdot L \cdot d^2)$$

Traditional attention mechanisms, such as the one introduced in [1], have a computational complexity of:

$$\mathcal{O}(M^2 \cdot d)$$

Comparing the two:

$$\mathcal{O}(M \cdot L \cdot d^2) \ll \mathcal{O}(M^2 \cdot d) \quad \text{for large } M$$

This demonstrates that AHCA scales linearly with M, whereas traditional attention scales quadratically, confirming the efficiency gain.

Q.E.D.

12.2 Proof of Information Preservation

Claim: AHCA preserves essential information from the input sequence through hierarchical compression and selective attention.

Proof:

The hierarchical compression in AHCA involves two levels:

1. **Primary Compression:** Each chunk $\mathbf{X}_k$ is compressed into $\mathbf{C}_{1,k}$ using f_{primary}. If f_{primary} is designed to retain salient features (e.g., using max pooling or attention-based pooling), essential information is preserved at this level.

2. **Secondary Compression:** The aggregated $\mathbf{C}_1$ is further compressed into $\mathbf{C}_2$ using $f_{\text{secondary}}$. Similar to primary compression, if $f_{\text{secondary}}$ effectively captures global salient features, essential information across all chunks is retained.

Selective attention focuses on the most relevant global tokens $\mathcal{S}$, ensuring that interactions are computed between important local and global representations. By allocating more layers to important tokens, AHCA ensures that critical information is processed with greater depth, enhancing the model's ability to preserve and utilize essential features.

Thus, through carefully designed compression functions and selective attention, AHCA maintains the integrity of essential information from the input sequence.

Q.E.D.

13 Sample Implementations

To facilitate the implementation of AHCA, we provide sample Python code snippets for key components using PyTorch. These implementations are simplified and intended for illustrative purposes. For production use, further optimizations and enhancements may be necessary.

13.1 Dynamic Chunking

Dynamic chunking splits the input sequence into chunks of a specified size.

```python
import torch

def chunk_sequence(X, chunk_size):
    """
    Splits the input sequence X into chunks of size chunk_size.

    Parameters:
        X (torch.Tensor): Input sequence of shape (M, d)
        chunk_size (int): Number of tokens per chunk

    Returns:
        List[torch.Tensor]: List of chunks, each of shape (chunk_size, d
    )
    """
    M, d = X.shape
    K = (M + chunk_size - 1) // chunk_size  # Ceiling division
    chunks = [X[i*chunk_size : (i+1)*chunk_size] for i in range(K)]
    return chunks
```

Listing 7: Dynamic Chunking Implementation

13.2 Primary Compression

Primary compression reduces each chunk to C_1 tokens using a linear projection followed by adaptive average pooling.

```python
import torch.nn as nn

class PrimaryCompressor(nn.Module):
    def __init__(self, d, C1, d_prime):
        super(PrimaryCompressor, self).__init__()
        self.linear = nn.Linear(d, d_prime)
        self.pool = nn.AdaptiveAvgPool1d(C1)

    def forward(self, X_k):
        """
        Compresses a chunk X_k to C1 tokens.

        Parameters:
            X_k (torch.Tensor): Chunk of shape (M_k, d)

        Returns:
            torch.Tensor: Compressed tokens of shape (C1, d_prime)
        """
        # Apply linear projection
        X_proj = self.linear(X_k)  # Shape: (M_k, d_prime)
        # Transpose for pooling: (d_prime, M_k)
        X_proj = X_proj.transpose(0, 1)
        # Apply adaptive average pooling
        C1 = self.pool(X_proj)  # Shape: (d_prime, C1)
        # Transpose back: (C1, d_prime)
        C1 = C1.transpose(0, 1)
        return C1
```

Listing 8: Primary Compression Implementation

13.3 Secondary Compression

Secondary compression aggregates all C_1 tokens across chunks and compresses them into C_2 global tokens.

```python
class SecondaryCompressor(nn.Module):
    def __init__(self, C1, d_prime, C2, d_double_prime):
        super(SecondaryCompressor, self).__init__()
        self.linear = nn.Linear(C1 * d_prime, d_double_prime)
        self.pool = nn.AdaptiveAvgPool1d(C2)

    def forward(self, C1_list):
        """
        Compresses aggregated C1 tokens to C2 global tokens.

        Parameters:
            C1_list (List[torch.Tensor]): List of C1 tokens from each
    chunk

        Returns:
            torch.Tensor: Global compressed tokens of shape (C2,
    d_double_prime)
        """
        # Concatenate all C1 tokens: (K * C1, d_prime)
        C1_concat = torch.cat(C1_list, dim=0)
        # Flatten: (1, K * C1 * d_prime)
        C1_flat = C1_concat.view(1, -1)
        # Apply linear projection
        C2_proj = self.linear(C1_flat)  # Shape: (1, d_double_prime)
        # Reshape for pooling: (d_double_prime, 1)
        C2_proj = C2_proj.transpose(0, 1)
        # Apply adaptive average pooling
        C2_pooled = self.pool(C2_proj)  # Shape: (d_double_prime, C2)
        # Transpose back: (C2, d_double_prime)
        C2 = C2_pooled.transpose(0, 1)
        return C2
```

Listing 9: Secondary Compression Implementation

13.4 Sparse Global Attention

Sparse global attention selects a subset of global tokens based on importance scores and computes attention.

```
class SparseGlobalAttention(nn.Module):
    def __init__(self, d_double_prime, C2, Cs, d_k):
        super(SparseGlobalAttention, self).__init__()
        self.W_Q = nn.Linear(d_double_prime, d_k)
        self.W_K = nn.Linear(d_double_prime, d_k)
        self.W_V = nn.Linear(d_double_prime, d_k)
        self.Cs = Cs  # Number of sparse tokens to select

    def forward(self, C2):
        """
        Applies sparse global attention.

        Parameters:
            C2 (torch.Tensor): Global compressed tokens of shape (C2,
    d_double_prime)

        Returns:
            torch.Tensor: Attention output of shape (Cs, d_k)
        """
        # Compute importance scores (e.g., via norm)
        scores = torch.norm(C2, dim=1)  # Shape: (C2,)
        # Select top Cs indices
        _, top_indices = torch.topk(scores, self.Cs)
        # Select sparse tokens
        C2_S = C2[top_indices]  # Shape: (Cs, d_double_prime)

        # Compute queries, keys, values
        Q = self.W_Q(C2_S)  # Shape: (Cs, d_k)
        K = self.W_K(C2)     # Shape: (C2, d_k)
        V = self.W_V(C2)     # Shape: (C2, d_k)

        # Compute attention scores
        attention_scores = torch.softmax((Q @ K.transpose(0, 1)) / torch
    .sqrt(torch.tensor(self.W_Q.out_features, dtype=torch.float)), dim=1)
      # Shape: (Cs, C2)

        # Compute attention output
        A = attention_scores @ V  # Shape: (Cs, d_k)

        return A
```

Listing 10: Sparse Global Attention Implementation

13.5 Layer-Wise Adaptive Processing

Layer-wise adaptive processing allocates layers based on importance scores.

```
class LayerWiseAdaptiveProcessing(nn.Module):
    def __init__(self, L, d_k):
        super(LayerWiseAdaptiveProcessing, self).__init__()
        self.layers = nn.ModuleList([nn.Linear(d_k, d_k) for _ in range(
    L)])
        self.L = L

    def forward(self, A, importance_scores):
        """
        Applies adaptive layers based on importance scores.

        Parameters:
            A (torch.Tensor): Attention output of shape (Cs, d_k)
            importance_scores (torch.Tensor): Importance scores of shape
     (Cs,)

        Returns:
            torch.Tensor: Final processed representation of shape (Cs,
    d_k)
        """
        # Determine number of layers per token
        # Example: Linear scaling based on importance
        L_k = (importance_scores / importance_scores.max() * self.L).
    long()  # Shape: (Cs,)

        H = A
        for l in range(self.L):
            H = self.layers[l](H)
            # Optionally, apply non-linearity
            H = torch.relu(H)
        return H
```

Listing 11: Layer-Wise Adaptive Processing Implementation

13.6 Complete AHCA Implementation

Combining all components, here is a simplified version of the complete AHCA model.

```python
import torch
import torch.nn as nn

class AHCA(nn.Module):
    def __init__(self, d, C1, d_prime, C2, d_double_prime, Cs, d_k, L):
        super(AHCA, self).__init__()
        self.primary_compressor = PrimaryCompressor(d, C1, d_prime)
        self.secondary_compressor = SecondaryCompressor(C1, d_prime, C2,
    d_double_prime)
        self.sparse_attention = SparseGlobalAttention(d_double_prime, C2
    , Cs, d_k)
        self.adaptive_processing = LayerWiseAdaptiveProcessing(L, d_k)

    def forward(self, X):
        # Step 1: Dynamic Chunking
        chunk_size = 100  # Example chunk size
        chunks = chunk_sequence(X, chunk_size)

        # Step 2: Primary Compression
        C1 = [self.primary_compressor(chunk) for chunk in chunks]

        # Step 3: Secondary Compression
        C2 = self.secondary_compressor(C1)

        # Step 4: Sparse Global Attention
        A = self.sparse_attention(C2)

        # Step 5: Layer-Wise Adaptive Processing
        # Example importance scores based on norm
        importance_scores = torch.norm(A, dim=1)
        H = self.adaptive_processing(A, importance_scores)

        return H
```

Listing 12: Complete AHCA Implementation

13.7 Training AHCA

Below is a sample training loop integrating the AHCA model. This example uses a simple regression task for demonstration purposes.

```python
import torch.optim as optim

# Define model parameters
d = 512
C1 = 64
d_prime = 256
C2 = 32
d_double_prime = 128
Cs = 16
d_k = 64
L = 4

# Initialize AHCA model
model = AHCA(d, C1, d_prime, C2, d_double_prime, Cs, d_k, L)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

```

```
# Sample data
X = torch.randn(1000, d)  # Input sequence of 1000 tokens
Y = torch.randn(Cs, d_k)  # Target output

# Training step
model.train()
optimizer.zero_grad()
H = model(X)  # Forward pass
loss = criterion(H, Y)
loss.backward()
optimizer.step()

print(f"Training Loss: {loss.item()}")
```

Listing 13: AHCA Training Loop Implementation

13.8 Notes on Implementation

- Boundary Detection: The provided ‘$chunk_sequence$‘ function uses fixed-size chunking. For dynamic boundary detection based on semantics or syntax, consider integrating NLP techniques such as sentence tokenization or leveraging pre-trained models to identify meaningful split points.
- Compression Functions: The current implementation uses linear projections and adaptive average pooling. Depending on the application, more sophisticated compression techniques (e.g., attention-based pooling) can be employed to enhance information preservation.
- Attention Mechanism: The sparse attention implementation uses norm-based scoring for simplicity. Alternative methods, such as attention-based scoring or clustering algorithms, can be explored to improve the selection of important tokens.
- **Layer Allocation:** The LayerWiseAdaptiveProcessing class provided applies all layers uniformly for demonstration. To fully implement adaptive layer allocation, modify the forward pass to apply a variable number of layers based on ‘L_k‘.
- **Scalability:** Ensure that the chunk size and compression dimensions C_1, C_2, and C_s are chosen based on the specific application requirements and computational resources.

14 Conclusion

The **Adaptive Hierarchical Compressed Attention (AHCA)** framework presents a mathematically robust and computationally efficient approach to handling large-scale input sequences. By integrating dynamic chunking, multi-level compression, sparse global attention, and layer-wise adaptive processing, AHCA addresses the scalability challenges of traditional attention mechanisms without compromising the quality of information processing. Its linear scaling with input size M and selective resource allocation make it a promising candidate for deployment in resource-constrained environments and applications requiring real-time processing of extensive data.