Demystifying Performer Attention Handle Genome-Length Sequences Efficiently

Shakeel A. Sheikh
The Kashmir Hub for Artificial Intelligence Research (KHAIR)
shakeelzmail608@gmail.com

January 10, 2026

Abstract

This document provides a comprehensive tutorial on attention mechanisms, starting from the fundamental self-attention mechanism and progressing to the efficient **Performer** attention. We explain all mathematical concepts with clarity, using gene sequence analysis as a motivating example throughout. The document includes step-by-step explanations, comparative analyses, practical examples, and complete PyTorch implementation code for **Performer** attention. All concepts are presented in an accessible manner suitable for both beginners and experienced practitioners in machine learning and computational biology. [1]

Contents

[1] *If you find any mistakes in the document, please let me know via email: shakeelzmail608@gmail.com.*

1 Introduction

Attention mechanisms have revolutionized deep learning, particularly in natural language processing and computational biology. However, the quadratic complexity of standard self-attention limits its applicability to long sequences, such as gene sequences, protein sequences,

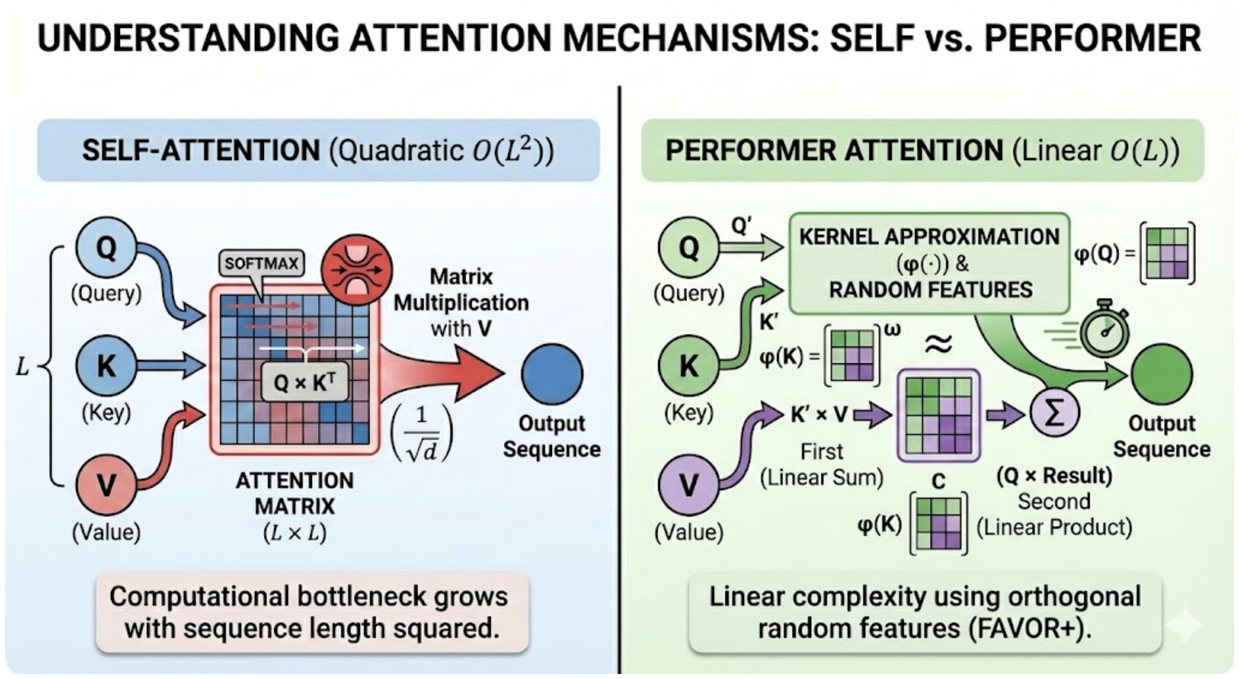
UNDERSTANDING ATTENTION MECHANISMS: SELF vs. PERFORMER
SELF-ATTENTION (Quadratic $O(L^2)$)
PERFORMER ATTENTION (Linear $O(L)$)
Q
(Query)
K
(Key)
V
(Value)
L
SOFTMAX
$Q \times K^T$
ATTENTION MATRIX
$(L \times L)$
Matrix Multiplication with V
$\left(\frac{1}{\sqrt{d}}\right)$
Output Sequence
Computational bottleneck grows with sequence length squared.
Q'
K'
KERNEL APPROXIMATION (φ(·)) & RANDOM FEATURES
φ(Q) =
φ(K) =
ω
≈
K' × V
First (Linear Sum)
C
φ(K)
Σ
(Q × Result) Second (Linear Product)
Linear complexity using orthogonal random features (FAVOR+).

Figure 1: Comparison between standard self-attention (left) and Performer attention (right). In self-attention, the input sequence $X \in \mathbb{R}^{N\times d}$ is first projected into Queries (Q), Keys (K), and Values (V). The attention matrix is then computed using the softmax of the similarity scores $QK^\top$, producing a dense $N \times N$ matrix that assigns a weight to every pair of tokens, resulting in quadratic time and memory complexity $O(N^2)$. The final output is obtained as $Z = \mathrm{softmax}(QK^\top)V$. In contrast, Performer attention replaces the softmax kernel with a randomized feature map $\phi(\cdot)$ that approximates the exponential kernel. Queries and keys are transformed into low-dimensional random features $Q' = \phi(Q)$ and $K' = \phi(K)$, allowing the attention computation to be reordered as $Z = Q'(K'^\top V) \oslash (Q'(K'^\top \mathbf{1}))$, which avoids explicitly forming the $N \times N$ attention matrix. This reduces both time and memory complexity from $O(N^2)$ to $O(N)$, enabling efficient modeling of very long sequences such as genomic data or long documents.

or single-cell RNA-seq data. The Performer (Performer Attention) addresses this limitation by providing a linear-time approximation to self-attention through kernel methods and random features.

In this tutorial, we:

1. Explain normal self-attention with intuitive examples
2. Introduce the mathematical foundation of attention mechanisms
3. Detail the Performer attention mechanism step-by-step
4. Compare computational complexities
5. Provide practical examples with gene sequences
6. Include complete PyTorch implementation

2 Normal Self-Attention

2.1 Intuition and Biological Motivation

Consider a set of N genes, where each gene is represented by its expression levels across different conditions or time points. In biological systems, genes interact with each other in complex networks. Self-attention allows each gene to "attend" to all other genes, determining which relationships are most important for understanding its function within a pathway or network.

2.2 Mathematical Formulation

Let $X \in \mathbb{R}^{N\times d}$ represent our input matrix, where:

- N: Number of genes (sequence length)
- d: Number of features per gene (embedding dimension)

2.2.1 Step 1: Linear Projections

We define three learnable weight matrices:

$$\begin{aligned} W^Q &\in \mathbb{R}^{d\times d_k} \quad \text{(Query weights)} \\ W^K &\in \mathbb{R}^{d\times d_k} \quad \text{(Key weights)} \\ W^V &\in \mathbb{R}^{d\times d_v} \quad \text{(Value weights)} \end{aligned}$$

These project the input into query, key, and value representations:

$$Q = XW^Q \in \mathbb{R}^{N\times d_k} \tag{1}$$

$$K = XW^K \in \mathbb{R}^{N\times d_k} \tag{2}$$

$$V = XW^V \in \mathbb{R}^{N\times d_v} \tag{3}$$

assuming dimension $d_k = d_v = d$

Biological Interpretation:

- **Q** (Query): "What information does this gene need?"
- **K** (Key): "What information does this gene provide?"
- **V** (Value): "What is this gene's actual expression profile?"

2.2.2 Step 2: Attention Scores

The attention scores measure similarity between queries and keys:

$$S = \frac{QK^T}{\sqrt{d_k}} \in \mathbb{R}^{N \times N} \tag{4}$$

The scaling factor $\sqrt{d_k}$ prevents extreme values that could cause vanishing gradients in softmax.

2.2.3 Step 3: Softmax Normalization

Apply softmax row-wise to obtain attention weights:

$$A = \text{softmax}(S) = \frac{\exp(S_{ij})}{\sum_{k=1}^{N} \exp(S_{ik})} \in \mathbb{R}^{N \times N} \tag{5}$$

Each row sums to 1, representing a probability distribution over genes to attend to.

2.2.4 Step 4: Weighted Combination

The output is a weighted sum of values:

$$Z = AV \in \mathbb{R}^{N \times d_v} \tag{6}$$

2.3 Example: 5 Genes

Consider 5 genes with 4-dimensional feature vectors representing expression levels:

Gene	Feature 1	Feature 2	Feature 3	Feature 4
G1	1.0	0.5	0.2	1.5
G2	0.8	1.2	0.9	0.3
G3	0.3	0.7	1.8	0.4
G4	1.2	0.4	0.6	1.1
G5	0.9	1.0	0.5	0.8

Table 1: Example gene expression features

Let $d_k = d_v = 4$ for simplicity. After linear projections:

Step 1: Compute Q, K, V (using small random weights)

$$Q = \begin{bmatrix} 0.8 & -0.3 & 1.2 & 0.5 \\ 0.6 & 1.1 & 0.8 & -0.2 \\ -0.1 & 0.5 & 1.5 & 0.3 \\ 1.1 & 0.3 & 0.7 & 0.9 \\ 0.7 & 0.9 & 0.4 & 0.6 \end{bmatrix}, \quad K = \begin{bmatrix} 0.9 & -0.2 & 1.1 & 0.6 \\ 0.7 & 1.0 & 0.9 & -0.1 \\ -0.2 & 0.6 & 1.6 & 0.4 \\ 1.0 & 0.4 & 0.8 & 1.0 \\ 0.8 & 0.8 & 0.5 & 0.7 \end{bmatrix}$$

Step 2: Compute attention scores for Gene 1:

$$\begin{aligned} q_1 &= [0.8, -0.3, 1.2, 0.5] \\ k_1 &= [0.9, -0.2, 1.1, 0.6] \Rightarrow q_1 \cdot k_1 = 2.12 \\ k_2 &= [0.7, 1.0, 0.9, -0.1] \Rightarrow q_1 \cdot k_2 = 0.89 \\ &\vdots \\ S_1 &= [2.12, 0.89, 1.45, 2.01, 1.67]/2 \quad \text{(divided by } \sqrt{4} = 2\text{)} \end{aligned}$$

Step 3: Apply softmax:

$$\begin{aligned} A_1 &= \text{softmax}([1.06, 0.445, 0.725, 1.005, 0.835]) \\ &= [0.286, 0.115, 0.162, 0.239, 0.198] \end{aligned}$$

Step 4: Compute output for Gene 1:

$$z_1 = 0.286 \cdot v_1 + 0.115 \cdot v_2 + 0.162 \cdot v_3 + 0.239 \cdot v_4 + 0.198 \cdot v_5$$

2.4 Computational Complexity

The bottleneck is computing QK^T:

- **Memory**: $O(N^2)$ to store the attention matrix
- **Computation**: $O(N^2 d_k)$ for matrix multiplication

For $N = 10,000$ genes and $d_k = 64$:

$$\text{Memory} = 10,000^2 \times 4 \text{ bytes} \approx 400 \text{ MB}$$
$$\text{Operations} = 10,000^2 \times 64 \approx 6.4 \times 10^9$$

This quadratic scaling makes standard attention impractical for large gene sequences.

3 Performer Attention

3.1 Motivation and Core Idea

The Performer attention mechanism addresses the quadratic complexity problem by:

1. Reformulating attention as a kernel method
2. Using random feature maps for kernel approximation
3. Reordering computations to avoid explicit $N \times N$ matrices

3.2 Mathematical Foundation

3.2.1 Kernel Reformulation

Recall that softmax attention can be written as:

$$\text{Attention}(Q, K, V) = D^{-1} \exp\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{7}$$

where $D = \text{diag}\left(\exp\left(\frac{QK^T}{\sqrt{d_k}}\right) \mathbf{1}_N\right)$.

The key insight is to treat $\exp(q_i^T k_j/\sqrt{d_k})$ as a kernel function:

$$K_{\text{softmax}}(x, y) = \exp(x^T y) \tag{8}$$

Understanding Softmax as a Kernel in Attention

Why Softmax Acts as a Kernel in Attention Mechanisms

To understand why we treat softmax as a kernel, we need to examine the Attention mechanism at the element-wise level. A kernel can be viewed as a function $K(\mathbf{x}, \mathbf{y})$ that takes two vectors and returns a scalar representing their similarity.

1. The Entry-Wise View

In standard attention, we compute the matrix $\mathbf{A} = \exp\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right)$. Focusing on a single entry at position (i, j):

$$A_{ij} = \exp\left(\frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{d_k}}\right)$$

This value A_{ij} represents the raw *affinity* between the i-th query (e.g., Gene A) and the j-th key (e.g., Gene B). In kernel theory, any function computing such similarity can be interpreted as a **Kernel Function** $K(\mathbf{x}, \mathbf{y})$.

2. The Normalization Role of D

Softmax isn't merely an exponential—it includes normalization so each row sums to 1. In standard notation:

$$\text{Softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)}$$

In matrix formulation, this is captured through the diagonal matrix $\mathbf{D}$. Each diagonal entry D_{ii} contains the sum of affinities for row i:

$$D_{ii} = \sum_{j=1}^{N} \exp\left(\frac{\mathbf{q}_i^T \mathbf{k}_j}{\sqrt{d_k}}\right)$$

The final attention weights are obtained by:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{D}^{-1} \exp\left(\frac{\mathbf{QK}^T}{\sqrt{d_k}}\right) \mathbf{V}$$

Multiplication by $\mathbf{D}^{-1}$ divides each element in row i by D_{ii}, exactly implementing the softmax operation. Thus, the softmax attention mechanism can be interpreted as applying a kernel (the exponential of scaled dot-products) followed by row-wise normalization.

3.2.2 Random Feature Maps

The kernel trick approximates the kernel function using random features:

$$K(x, y) = \mathbb{E}[\phi(x)^T \phi(y)] \approx \phi(x)^T \phi(y) \tag{9}$$

For the softmax kernel, we can use trigonometric random features or positive random features.

3.2.3 Positive Random Features (PRF)

The Performer uses:

$$\phi(x) = \frac{1}{\sqrt{m}} \exp\left(Wx - \frac{\|x\|^2}{2}\right) \tag{10}$$

where $W \in \mathbb{R}^{m \times d}$ is a random matrix with orthogonal rows.

Why Random Feature Maps and Positive Random Features (PRF)

Motivation: The Computational Bottleneck of Exact Kernel Methods
The standard attention mechanism with softmax has a quadratic computational complexity $O(N^2)$ in sequence length, as it requires computing all pairwise interactions between queries and keys. This becomes prohibitive for long sequences. Random feature maps provide a solution by approximating the kernel function with linear complexity.

3.2.4 Random Feature Maps: The Approximation Principle

The core idea comes from the kernel trick, which states that many kernel functions can be approximated by explicit feature maps:

$$K(\mathbf{x}, \mathbf{y}) = \mathbb{E}_{\omega \sim p(\omega)}[\phi_\omega(\mathbf{x})^T \phi_\omega(\mathbf{y})] \approx \phi(\mathbf{x})^T \phi(\mathbf{y}) \tag{11}$$

where $\phi(\mathbf{x})$ is a **random feature map** that projects the input into a higher-dimensional space (dimension m), and the expectation is over some distribution $p(\omega)$ of random parameters.
Why this works: Many kernels (including the softmax/Gaussian kernel) can be expressed as an **inner product** in some implicit feature space. Random feature maps make this explicit, allowing us to:

- Transform queries and keys separately: $\phi(\mathbf{q}_i)$ and $\phi(\mathbf{k}_j)$

- Compute attention as: Attention $\approx \frac{\phi(\mathbf{Q})\phi(\mathbf{K})^T\mathbf{V}}{\text{normalizer}}$
- Achieve $O(Nmd)$ complexity instead of $O(N^2d)$

3.2.5 Positive Random Features (PRF) for the Softmax Kernel

For the softmax kernel $K(\mathbf{x}, \mathbf{y}) = \exp(\mathbf{x}^T\mathbf{y})$, we need a specific type of random features. The Performer model (Choromanski et al., 2020) uses:

$$\phi(\mathbf{x}) = \frac{1}{\sqrt{m}} \exp\left(\mathbf{W}\mathbf{x} - \frac{\|\mathbf{x}\|^2}{2}\right) \tag{12}$$

where $\mathbf{W} \in \mathbb{R}^{m\times d}$ is a random matrix with rows $\mathbf{w}_i \sim \mathcal{N}(0, \mathbf{I}_d)$, often made **orthogonal** for better approximation.
Why this particular form? This stems from the Gaussian integral identity:

$$\exp(\mathbf{x}^T\mathbf{y}) = \mathbb{E}_{\mathbf{w}\sim\mathcal{N}(0,\mathbf{I})}\left[\exp\left(\mathbf{w}^T\mathbf{x} - \frac{\|\mathbf{x}\|^2}{2}\right)\exp\left(\mathbf{w}^T\mathbf{y} - \frac{\|\mathbf{y}\|^2}{2}\right)\right] \tag{13}$$

Key properties of PRF:

- **Positivity:** All features are positive ($\exp(\cdot) > 0$), which is crucial for stable attention computation
- **Unbiased estimator:** $\mathbb{E}[\phi(\mathbf{x})^T\phi(\mathbf{y})] = \exp(\mathbf{x}^T\mathbf{y})$
- **Variance reduction:** Orthogonal rows in $\mathbf{W}$ reduce the variance of the estimator
- **Linearization:** Allows rewriting attention as:
$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \approx \mathbf{D}^{-1}(\phi(\mathbf{Q})(\phi(\mathbf{K})^T\mathbf{V}))$$
where $\mathbf{D}$ is computed from $\phi(\mathbf{Q})\phi(\mathbf{K})^T\mathbf{1}$

Mathematical Derivation: Given $\phi(\mathbf{x}) = \frac{1}{\sqrt{m}}\exp(\mathbf{W}\mathbf{x} - \|\mathbf{x}\|^2/2)$, we have:

$$\phi(\mathbf{x})^T\phi(\mathbf{y}) = \frac{1}{m}\sum_{i=1}^{m}\exp\left(\mathbf{w}_i^T(\mathbf{x}+\mathbf{y}) - \frac{\|\mathbf{x}\|^2 + \|\mathbf{y}\|^2}{2}\right)$$

By the law of large numbers, as $m \to \infty$, this converges to $\exp(\mathbf{x}^T\mathbf{y})$.

Practical Impact: The PRF approach enables linear attention mechanisms that:

- Scale to very long sequences (thousands to millions of tokens)
- Maintain theoretical guarantees of approximation quality
- Can be trained end-to-end like standard transformers
- Have been successfully applied in models like Performer, Linear Transformer, and others

3.3 Step-by-Step Algorithm

Algorithm 1 Performer Attention Algorithm

```
Require: Input $X \in \mathbb{R}^{N \times d}$, random feature dimension $m$
Ensure: Output $Z \in \mathbb{R}^{N \times d}$
 1: // Step 1: Linear projections (same as self-attention)
 2: $Q, K, V \leftarrow \text{LinearProjections}(X)$
 3: // Step 2: Compute random features
 4: Generate random orthogonal matrix $W \in \mathbb{R}^{m \times d}$
 5: $Q' \leftarrow \phi(Q) = \frac{1}{\sqrt{m}} \exp(WQ - \frac{\|Q\|^2}{2})$
 6: $K' \leftarrow \phi(K) = \frac{1}{\sqrt{m}} \exp(WK - \frac{\|K\|^2}{2})$
 7: // Step 3: Reorder computations
 8: // Instead of: $Z = \text{softmax}(QK^T)V$
 9: // We compute: $Z = (Q'(K'^T V)) \oslash (Q'(K'^T \mathbf{1}_N))$
10: numerator $\leftarrow Q' \times (K'^T \times V)$
11: denominator $\leftarrow Q' \times (K'^T \times \mathbf{1}_N)$
12: $Z \leftarrow$ numerator $\oslash$ denominator
13: return $Z$
```

Why Random Feature Maps and Positive Random Features (PRF)

3.4 Detailed Example with 5 Genes

Let's use the same 5 genes from Table 1, with:

- $d = 4$ (original features)
- $m = 8$ (random features, much smaller than $N^2 = 25$)
- $d_k = d_v = 4$

3.4.1 Step 1: Compute Random Matrix W

Generate random orthogonal matrix $W \in \mathbb{R}^{8 \times 4}$:

$$W = \begin{bmatrix} 0.3 & -0.2 & 0.8 & 0.5 \\ -0.4 & 0.7 & 0.1 & -0.6 \\ 0.6 & 0.3 & -0.4 & 0.6 \\ 0.1 & -0.5 & 0.7 & 0.5 \\ -0.7 & 0.1 & 0.5 & -0.5 \\ 0.5 & 0.6 & 0.2 & 0.6 \\ -0.2 & 0.8 & -0.3 & 0.5 \\ 0.4 & 0.2 & 0.6 & -0.7 \end{bmatrix}$$

3.4.2 Step 2: Compute Random Features for Gene 1

After linear projection, suppose $q_1 = [0.8, -0.3, 1.2, 0.5]$.

Compute Wq_1:

$$Wq_1 = \begin{bmatrix} 0.3 \times 0.8 + (-0.2) \times (-0.3) + 0.8 \times 1.2 + 0.5 \times 0.5 \\ -0.4 \times 0.8 + 0.7 \times (-0.3) + 0.1 \times 1.2 + (-0.6) \times 0.5 \\ \vdots \\ 0.4 \times 0.8 + 0.2 \times (-0.3) + 0.6 \times 1.2 + (-0.7) \times 0.5 \end{bmatrix}$$
$$= [1.05, -0.62, 0.78, 0.45, -1.12, 1.21, -0.35, 0.92]$$

Compute $\|q_1\|^2/2 = (0.8^2 + (-0.3)^2 + 1.2^2 + 0.5^2)/2 = 0.955$
Apply transformation:

$$\phi(q_1) = \frac{1}{\sqrt{8}} \exp\left([1.05, -0.62, 0.78, 0.45, -1.12, 1.21, -0.35, 0.92] - 0.955\right)$$
$$= \frac{1}{2.828} \times [\exp(0.095), \exp(-1.575), \ldots, \exp(-0.035)]$$
$$= [0.18, 0.05, 0.12, 0.09, 0.03, 0.21, 0.07, 0.14]$$

3.5 Complexity Analysis

Operation	**Self-Attention**	**Performer**	**Savings**
Memory	$O(N^2)$	$O(Nm)$	$O(N/m)$
Computation	$O(N^2 d_k)$	$O(Nmd_k)$	$O(N/m)$
Matrix Size	$N \times N$	$N \times m$	-

Table 2: Complexity comparison ($m \ll N$)

For $N = 10,000$, $d_k = 64$, $m = 256$:

$$\text{Memory savings} = \frac{N}{m} = \frac{10,000}{256} \approx 39\times$$
$$\text{Computation savings} = \frac{N^2 d_k}{N m d_k} = \frac{N}{m} \approx 39\times$$

4 Comparative Analysis

4.1 Theoretical Differences

4.2 Practical Considerations for Gene Analysis

4.2.1 When to Use Self-Attention:

- Small gene sets ($N < 1,000$)
- When exact attention patterns are crucial

Aspect	**Self-Attention**	**Performer**
Exactness	Exact computation	Approximate via random features
Memory	Quadratic in sequence length	Linear in sequence length
Compute Time	Quadratic in sequence length	Linear in sequence length
Parallelization	Limited by N^2 matrix	Highly parallelizable
Theoretical Guarantees	Exact result	Probabilistic bounds
Biological Interpretation	Exact gene-gene interactions	Approximate interactions

Table 3: Theoretical comparison

- For interpretability studies requiring exact weights
- When computational resources are abundant

4.2.2 When to Use Performer:

- Genome-scale analysis ($N > 10,000$)
- Single-cell RNA-seq with many cells
- Protein sequence analysis
- Real-time biological applications

5 PyTorch Implementation

5.1 Complete Performer Attention Module

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math


class PerformerAttention(nn.Module):
    """
    Performer Attention Module

    Args:
        dim (int): Input dimension
        heads (int): Number of attention heads
        dim_head (int): Dimension per head
        causal (bool): Whether to use causal masking
        kernel_type (str): 'relu' or 'softmax' kernel
        random_features (int): Number of random features (m)
    """

    def __init__(self, dim, heads=8, dim_head=64, causal=False,
                 kernel_type='relu', random_features=256):
        super().__init__()
        self.dim = dim
        self.heads = heads
        self.dim_head = dim_head
        self.causal = causal
        self.kernel_type = kernel_type
        self.random_features = random_features

        # Inner dimension for multi-head attention
        inner_dim = dim_head * heads

        # Linear projections for Q, K, V
        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias=False)

        # Output projection
        self.to_out = nn.Linear(inner_dim, dim)

        # Random projection matrix (not learned, fixed during training)
        self.register_buffer('projection_matrix',
                             self.create_projection_matrix(dim_head,
random_features))

        # Layer normalization for stability
        self.norm = nn.LayerNorm(dim_head)

    def create_projection_matrix(self, dim, random_features):
        """
        Create random orthogonal matrix for kernel approximation

        Args:
            dim: Input dimension
            random_features: Number of random features (m)

        Returns:
            Random orthogonal matrix of shape [dim, random_features]
        """
        # Generate random matrix
        rand_mat = torch.randn(random_features, dim)

        # Orthogonalize using QR decomposition
        q, _ = torch.linalg.qr(rand_mat, mode='reduced')

        # Transpose to get [dim, random_features]
        return q.t()

    def relu_kernel(self, x, is_query=False):
        """
        ReLU kernel approximation
```

```python
        $\phi(x) = max(0, x)$ for both queries and keys
        """
        return F.relu(x)

    def softmax_kernel(self, x, is_query, projection_matrix):
        """
        Softmax kernel approximation using random features

        For queries: $\phi(q) = \frac{1}{\sqrt{m}} * \exp(Wq - ||q||^2)$
        For keys: $\phi(k) = \frac{1}{\sqrt{m}} * \exp(Wk - ||k||^2)$
        """
        # Normalize inputs for numerical stability
        x = F.normalize(x, dim=-1, p=2)

        # Project using random matrix
        projected = torch.matmul(x, projection_matrix)

        # Compute squared norm
        x_norm_squared = (x ** 2).sum(dim=-1, keepdim=True)

        projected = projected - x_norm_squared / 2
        return torch.exp(projected) / math.sqrt(self.random_features)



    def forward(self, x, mask=None):
        """
        Forward pass

        Args:
            x: Input tensor of shape [batch_size, seq_len, dim]
            mask: Optional attention mask

        Returns:
            Output tensor of shape [batch_size, seq_len, dim]
        """
        batch_size, seq_len, _ = x.shape

        # Step 1: Linear projections to get Q, K, V
        qkv = self.to_qkv(x).chunk(3, dim=-1)
        q, k, v = map(
            lambda t: t.reshape(batch_size, seq_len, self.heads, self.
dim_head).transpose(1, 2),
            qkv
        )

        # Normalize for stability
        q = self.norm(q)
        k = self.norm(k)

        # Step 2: Apply kernel approximation
        if self.kernel_type == 'relu':
            q_prime = self.relu_kernel(q, is_query=True)
            k_prime = self.relu_kernel(k, is_query=False)
        else:  # softmax kernel
            q_prime = self.softmax_kernel(q, is_query=True,
                                          projection_matrix=self.projection_matrix)
            k_prime = self.softmax_kernel(k, is_query=False,
                                          projection_matrix=self.projection_matrix)

        # Step 3: Compute attention using kernel trick

        # Transpose K' for efficient multiplication
        k_prime_t = k_prime.transpose(-2, -1)  # [batch, heads, dim_head, seq_len]

        # Compute K'^T V
        ktv = torch.matmul(k_prime_t, v)  # [batch, heads, dim_head, dim_head]

        # Compute Q'(K'^T V)
        numerator = torch.matmul(q_prime, ktv)  # [batch, heads, seq_len, dim_head]

        # Normalization: compute denominator
        # Create ones tensor for denominator calculation
        ones = torch.ones(batch_size, seq_len, 1, 1, device=x.device)

        # Compute K'^T * 1
        kt_ones = torch.matmul(k_prime_t, ones)  # [batch, heads, dim_head, 1]

        # Compute Q'(K'^T 1)
        denominator = torch.matmul(q_prime, kt_ones)  # [batch, heads, seq_len, 1]

        # Avoid division by zero
        denominator = denominator + 1e-8

        # Normalize to get attention output
        out = numerator / denominator

        # Reshape back to original dimensions
        out = out.transpose(1, 2).reshape(batch_size, seq_len, -1)

        # Final linear projection
        return self.to_out(out)


class GenePerformer(nn.Module):
    """
    Complete gene sequence model using Performer attention

    Args:
        num_genes: Number of unique genes in vocabulary
        dim: Embedding dimension
        depth: Number of Performer layers
        heads: Number of attention heads
        dim_head: Dimension per head
        random_features: Number of random features for approximation
    """

    def __init__(self, num_genes, dim=128, depth=6, heads=8,
                 dim_head=64, random_features=256):
        super().__init__()

        # Gene embeddings (learnable representations)
        self.gene_embeddings = nn.Embedding(num_genes, dim)

        # Positional encodings (for sequence order)
        self.position_embeddings = nn.Parameter(torch.randn(1, 1000, dim))

        # Multiple Performer layers
        self.layers = nn.ModuleList([
            PerformerAttention(
                dim=dim,
                heads=heads,
                dim_head=dim_head,
                kernel_type='softmax',
                random_features=random_features
            )
            for _ in range(depth)
        ])

        # Layer normalization
        self.norm = nn.LayerNorm(dim)

        # Output layer for gene prediction tasks
        self.output_layer = nn.Linear(dim, num_genes)

    def forward(self, gene_indices, mask=None):
        """
        Forward pass for gene sequence analysis

        Args:
            gene_indices: Tensor of shape [batch_size, seq_len]
                          containing gene indices
            mask: Optional attention mask

        Returns:
            Logits for gene predictions
        """
        batch_size, seq_len = gene_indices.shape

        # Get gene embeddings
        x = self.gene_embeddings(gene_indices)  # [batch, seq_len, dim]

        # Add positional embeddings
        pos_emb = self.position_embeddings[:, :seq_len, :]
```

```python
        x = x + pos_emb

        # Apply Performer layers with residual connections
        for layer in self.layers:
            # Residual connection
            x = layer(x, mask=mask) + x

        # Final normalization
        x = self.norm(x)

        # Output predictions
        return self.output_layer(x)


def create_gene_attention_model(config):
    """
    Factory function to create gene attention model

    Args:
        config: Dictionary containing model configuration

    Returns:
        Initialized GenePerformer model
    """
    model = GenePerformer(
        num_genes=config['num_genes'],
        dim=config.get('dim', 128),
        depth=config.get('depth', 6),
        heads=config.get('heads', 8),
        dim_head=config.get('dim_head', 64),
        random_features=config.get('random_features', 256)
    )

    # Initialize weights
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)

    return model


# Example usage
if __name__ == "__main__":
    # Configuration
    config = {
        'num_genes': 1000,  # Vocabulary size
        'dim': 128,
        'depth': 6,
        'heads': 8,
        'dim_head': 64,
        'random_features': 256
    }

    # Create model
    model = create_gene_attention_model(config)

    # Create sample batch of gene sequences
    batch_size = 32
    seq_len = 50  # 50 genes per sequence
    gene_sequences = torch.randint(0, config['num_genes'], (batch_size,
seq_len))

    # Forward pass
    print(f"Input shape: {gene_sequences.shape}")
    output = model(gene_sequences)
    print(f"Output shape: {output.shape}")

    # Memory usage comparison
    total_params = sum(p.numel() for p in model.parameters())
    print(f"Total parameters: {total_params:,}")

    # Example of memory savings
    N = seq_len
    m = config['random_features']
    d = config['dim_head']

    normal_memory = N * N * 4  # bytes for float32
    performer_memory = N * m * d * 4

    print(f"\nMemory comparison for seq_len={N}:")
    print(f"Normal attention: {normal_memory:,} bytes")
    print(f"Performer attention: {performer_memory:,} bytes")
    print(f"Savings: {normal_memory/performer_memory:.1f}x")
```

Listing 1: Complete Performer Attention Implementation

5.2 Training Example for Gene Function Prediction

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np


class GeneDataset(Dataset):
    """Dataset for gene sequence analysis"""

    def __init__(self, sequences, labels, max_len=100):
        self.sequences = sequences  # List of gene index sequences
        self.labels = labels        # Corresponding function labels
        self.max_len = max_len

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        seq = self.sequences[idx][:self.max_len]
        label = self.labels[idx]

        # Pad sequence if necessary
        if len(seq) < self.max_len:
            seq = seq + [0] * (self.max_len - len(seq))

        return torch.tensor(seq), torch.tensor(label)


def train_gene_model(model, train_loader, val_loader, config):
    """
    Training function for gene attention model

    Args:
        model: GenePerformer model
        train_loader: DataLoader for training data
        val_loader: DataLoader for validation data
        config: Training configuration
    """

    # Loss function and optimizer
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.AdamW(
        model.parameters(),
        lr=config.get('lr', 1e-4),
        weight_decay=config.get('weight_decay', 0.01)
    )

    # Learning rate scheduler
    scheduler = optim.lr_scheduler.CosineAnnealingLR(
        optimizer,
        T_max=config.get('epochs', 50)
    )

    # Training loop
    for epoch in range(config['epochs']):
        model.train()
        total_loss = 0

        for batch_idx, (sequences, labels) in enumerate(train_loader):
            optimizer.zero_grad()

            # Forward pass
            outputs = model(sequences)
            loss = criterion(outputs.view(-1, outputs.size(-1)),
                             labels.view(-1))

            # Backward pass
            loss.backward()

            # Gradient clipping
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

```

```python
            optimizer.step()

            total_loss += loss.item()

            if batch_idx % 100 == 0:
                print(f"Epoch {epoch}, Batch {batch_idx}, Loss: {loss.item():.4f}")

        # Validation
        model.eval()
        val_loss = 0
        correct = 0
        total = 0

        with torch.no_grad():
            for sequences, labels in val_loader:
                outputs = model(sequences)
                loss = criterion(outputs.view(-1, outputs.size(-1)),
                               labels.view(-1))
                val_loss += loss.item()

                # Calculate accuracy
                _, predicted = outputs.max(-1)
                total += labels.numel()
                correct += predicted.eq(labels).sum().item()

        avg_train_loss = total_loss / len(train_loader)
        avg_val_loss = val_loss / len(val_loader)
        accuracy = 100. * correct / total

        print(f"\nEpoch {epoch} Summary:")
        print(f"Train Loss: {avg_train_loss:.4f}")
        print(f"Val Loss: {avg_val_loss:.4f}")
        print(f"Val Accuracy: {accuracy:.2f}%")

        # Update learning rate
        scheduler.step()

    print("Training complete!")


# Example of creating and training the model
def main():
    # Configuration
    config = {
        'num_genes': 20000,  # Human genome has ~20,000 protein-coding genes
        'dim': 256,
        'depth': 8,
        'heads': 8,
        'dim_head': 64,
        'random_features': 512,
        'lr': 1e-4,
        'epochs': 50,
        'batch_size': 32
    }

    # Create model
    model = create_gene_attention_model(config)

    # Create synthetic dataset (in practice, use real gene data)
    num_samples = 10000
    max_seq_len = 100

    # Generate random gene sequences
    sequences = [
        np.random.randint(0, config['num_genes'],
                         np.random.randint(50, max_seq_len)).tolist()
        for _ in range(num_samples)
    ]

    # Generate random labels (e.g., pathway membership)
    labels = np.random.randint(0, 10, num_samples)  # 10 different
   pathways

    # Split into train/val
    split_idx = int(0.8 * num_samples)
    train_dataset = GeneDataset(sequences[:split_idx], labels[:split_idx])
    val_dataset = GeneDataset(sequences[split_idx:], labels[split_idx:])

    train_loader = DataLoader(train_dataset, batch_size=config['batch_size
   '],
                            shuffle=True)
    val_loader = DataLoader(val_dataset, batch_size=config['batch_size'])

    # Train the model
    train_gene_model(model, train_loader, val_loader, config)

    # Save the model
    torch.save(model.state_dict(), 'gene_performer_model.pth')
    print("Model saved!")


if __name__ == "__main__":
    main()
```

Listing 2: Training Loop for Gene Function Prediction

6 Biological Applications

6.1 Gene-Gene Interaction Networks

Performer attention enables the analysis of large gene interaction networks by:

1. **Scalability**: Handling thousands of genes simultaneously

2. **Attention Weights as Interactions**: The attention matrix approximates gene-gene interaction strengths
3. **Pathway Analysis**: Identifying genes that co-attend to each other in biological pathways

6.2 Single-Cell RNA-Seq Analysis

For single-cell RNA-seq data with N cells and G genes:

- Normal attention: $O(N^2G)$ - impractical for $N > 10,000$ cells
- Performer attention: $O(NmG)$ where $m \approx 256 - 512$
- Enables analysis of large-scale single-cell datasets

6.3 Protein Sequence Analysis

Protein sequences can be very long (up to 35,000 amino acids for Titin):

- Normal attention fails due to quadratic complexity
- Performer attention scales linearly with sequence length
- Enables whole-protein sequence analysis

7 Advanced Topics

7.1 Different Kernel Functions

Kernel	**Random Features**	**Properties**
Softmax	$\phi(x) = \exp(Wx - \|x\|^2/2)$	Matches standard attention
ReLU	$\phi(x) = \max(0, Wx)$	Simpler, faster
Trigonometric	$\phi(x) = [\sin(Wx), \cos(Wx)]$	Theoretical guarantees

Table 4: Kernel functions for Performer attention

7.2 Hyperparameter Selection

- **Random features (m)**: Typically 256-1024, trade-off between accuracy and efficiency
- **Number of heads**: 4-16, depends on task complexity
- **Dimension per head**: Usually 32-128
- **Kernel type**: 'softmax' for exact approximation, 'relu' for speed

8 Conclusion

The Performer attention mechanism represents a significant advance in scalable attention architectures. By reformulating attention as a kernel method and using random feature approximations, it achieves linear time and memory complexity while maintaining competitive performance with standard attention.

For biological applications, particularly in genomics, this enables:

- Analysis of genome-scale datasets
- Large-scale single-cell RNA-seq analysis

The provided PyTorch implementation offers a practical starting point for researchers and practitioners working with large biological sequences. The modular design allows easy integration into existing pipelines and adaptation to specific biological tasks.

8.1 Future Directions

1. **Adaptive random features**: Learning the projection matrix instead of random initialization
2. **Sparse attention patterns**: Combining Performer with sparse attention for even greater efficiency
3. **Biological priors**: Incorporating domain knowledge into attention mechanisms
4. **Multimodal integration**: Combining gene expression with other omics data

Acknowledgments

The Performer attention architecture, proposed by Choromanski et al. in *Rethinking Attention with Performers* (ICLR 2021), introduces a linear-complexity alternative to standard self-attention. This document is designed for ***educational and tutorial purposes***, providing a deep dive into its mathematical foundations via practical small example." This document provides a pedagogical breakdown of those concepts and and its application to long-range sequence modeling.

References

1. Vaswani, A., et al. (2017). Attention is all you need. NeurIPS.
2. Choromanski, K., et al. (2021). Rethinking attention with performers. ICLR.

A Appendix: Mathematical Derivations

A.1 Softmax Kernel Derivation

The softmax kernel is defined as:

$$K_{\text{softmax}}(x, y) = \exp(x^T y) \tag{14}$$

We can rewrite this using the identity:

$$\exp(x^T y) = \exp\left(\frac{\|x\|^2 + \|y\|^2 - \|x - y\|^2}{2}\right) \tag{15}$$

$$= \exp\left(\frac{\|x\|^2}{2}\right) \exp\left(\frac{\|y\|^2}{2}\right) \exp\left(-\frac{\|x - y\|^2}{2}\right) \tag{16}$$

The Gaussian kernel $\exp(-\|x - y\|^2/2)$ can be approximated using random Fourier features.

A.2 Random Feature Maps for Gaussian Kernel

For the Gaussian kernel $K(x, y) = \exp(-\|x - y\|^2/(2\sigma^2))$, we have:

$$K(x, y) = \mathbb{E}_{w \sim \mathcal{N}(0,I)}[\cos(w^T (x - y))] \tag{17}$$

This leads to the random feature map:

$$\phi(x) = \frac{1}{\sqrt{m}}[\cos(w_1^T x), \sin(w_1^T x), \ldots, \cos(w_m^T x), \sin(w_m^T x)] \tag{18}$$