FRENOS

Whitepaper - Breakdown of Retentive Networks (RetNet)

This paper reviews traditional neural network architectures and introduces one of the first domain-specific analyses of the Retentive Network (RetNet) architecture. RetNet aims to improve upon transformers by integrating features from both recurrent networks and transformers, optimizing for sequence modeling tasks. Frenos pre-trained and fine-tuned our own RetNet model, comparing it to well-known transformer-based models such as Llama 2, Mistral 7B, and Phi-3. This will be the first white paper in a series of research efforts to break down the novel model architectures from a more practical standpoint without diving too deeply into the theory. Future discussions will focus more heavily on the implementation and novel research Frenos is investigating into scalable small language models.

RetNet's recurrent representation enables efficient real-time processing, low-cost inference, and reduced memory usage. The architecture leverages a retention mechanism that supports parallel, recurrent, and chunkwise recurrent computation paradigms, allowing for efficient handling of long sequences and improved scalability compared to traditional transformer models. Experimental results highlight the importance of comprehensive pre-training using diverse datasets, including linguistic and reasoning tasks, to enhance the model's language understanding, contextual reasoning, and domain-specific knowledge embedding. However, due to computing constraints, our RetNet model was only pre-trained on over 4B tokens and fine-tuned for our domain-specific tasks. It performs on par with industry-standard models of its size (approx. 3B parameters). When compared against industry standard models in the context of our application, the RetNet model scaled significantly faster than traditional transformer-based models.

FRENOS ("brake" or "slow down" in Spanish) is founded by a former Fortune 500 cybersecurity architect with over 20 years of experience securing critical infrastructure and an ex-malware developer turned AI/ML leader from Dragos and AWS who have joined forces to build the industry's first attack and defense simulation platform. Frenos was founded to help simulate the threats of today to defend tomorrow.

1. Background and Context

1.1 Overview of Traditional Neural Network Architectures

Neural networks have become fundamental in various machine learning applications, including image recognition, natural language processing (NLP), and predictive analytics. Commonly used architectures include:

Feedforward Neural Networks (FNNs):

These are the simplest types of neural networks where information moves in one direction – forward – from the input nodes, through the hidden nodes, if any, to the output nodes. FNNs have fully connected layers where each neuron in one layer connects to each neuron in the next layer. The primary operation in FNNs is a dot product between the inputs and the weights, followed by a bias addition and applying a nonlinear activation function, such as ReLU or sigmoid. In the simplified mathematical model for a given layer *n*, the output y can be described as:

$$y = \sigma(W^{(n)}x^{(n-1)} + b^{(n)}) \tag{1}$$

where $W^{(n)}$ and $b^{(n)}$ are the weights and biases of the layer *n*, $x^{(n-1)}$ is the input from the previous layer, and σ is the activation function.

Due to their non-cyclic structure, they are fairly straightforward to understand and implement. However, their main limitation is the lack of memory, which makes them unsuitable for tasks where input data is either sequential or time-dependent (e.g., time series forecasting sequence prediction).

Recurrent Neural Networks (RNNs):

RNNs overcome some of the limitations of FNNs by incorporating loops in their network structure, allowing information to persist. This loop structure means that the output from the previous step is fed back into the network as an input to influence predictions at the current step, allowing them to form a memory of the previous outputs. The simplified mathematical model for the state of an RNN at time t, denoted h_t, is updated as:

$$h_t = \sigma(Wx_t + Uh_{t-1} + b) \tag{2}$$

where W, U and b are the weights and bias applicable to the input x_t, and the previous state h_{t-1}, respectively.

This design makes RNNs great for handling sequential and time-dependent data, such as text or speech. The main drawback of RNNs is their tendency to either forget the earlier inputs or suffer from the vanishing or exploding gradient problem. This complicates training and limits their ability to handle long sequences effectively. Techniques such as LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Units) have been introduced to overcome these issues, but at the cost of increased model complexity and computational resource requirements.

Convolutional Neural Networks (CNNs):

CNNs use a convolutional operation that involves a filter or kernel sliding over the input data, computing the dot products at each position. This operation makes them excel in tasks requiring capturing spatial and temporal dependencies through local receptive fields, such as image processing. A simplified output feature map Y, in a given convolutional layer is computed as:

$$Y_{i,j} = \sigma(\sum_m \sum_n W_{m,n} \bullet X_{i+m,j+n} + b) \tag{3}$$

where W is the kernel matrix, X is the input matrix (input image or feature map), b is the bias, and (i, j) indexes the location in the output feature map.

While CNNs are highly effective for grid-like data (e.g., images), they don't typically extend to sequential data processing without modifications since their architecture bias is towards fixed-window spatial correlations rather than temporal dynamics.

Transformers:

Transformer, while not entirely new technology, has significantly revolutionized the field of NLP. They discard both recurrent and convolutional structures, opting instead to fully depend on the self-attention mechanism, which can be represented by the following formula:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (4)$$

where $Q, K,$ and V represent the matrices for the queries, keys, and values. The dimension of the key vectors is represented as d_k. The scaling factor $\sqrt{d_k}$ prevents the dot products from growing too large in magnitude, leading to vanishing gradients during backpropagation.

This approach allows them to assess and prioritize the relevance of every input data segment with one another. Compared to prior architectures, the self-attention mechanisms allow the model to process input sequences in parallel, significantly reducing training times. The architecture can be composed of an encoder and decoder structure or decoder only, each consisting of a stack of identical layers that include multi-head self-attention and position-wise fully connected feed-forward networks.

The primary challenge when using transformers is their quadratic complexity concerning input length, leading to significant computational and memory overhead as sequence length increases. During training, each Q interacts with the other K in the same position or earlier in the sequence. Hence, training naturally has a quadratic dependence on sequence length. This makes scaling transformers particularly difficult without significant access to compute resources.

1.2 Limitations of Traditional Architectures

Traditional neural network architectures have significantly advanced the field of machine learning, providing robust solutions across all domains, including NLP, computer vision, and time series analysis. However, these architectures inherently possess several limitations that can hinder their scalability, efficiency, and speed, particularly as the complexity and size of the data increase. This section outlines the primary constraints of traditional neural network architectures discussed above.

1.2.1 Scalability Issues

FNNs are limited by their static architecture, which requires a predefined and fixed input size. This restricts their application in fields where input dimensions are dynamic, or data is inherently sequential or contextual. Scaling FNNs is also difficult because of the curse of dimensionality as the network depth increases, which leads to overfitting and diminished returns on performance regardless of access to computational resources.

Transformers, known for their self-attention mechanism, calculate pairwise interactions across all elements in a sequence, leading to a computational complexity of $O(n^2)$ where n is the sequence length. This quadratic complexity substantially increases computational and memory demands for longer sequences, challenging the scalability of Transformers in scenarios with extensive data or limited computational resources (e.g., a startup).

1.2.2 Efficiency

The sequential nature of RNNs makes parallel processing difficult, thus limiting their efficiency and scalability. Training RNNs becomes increasingly challenging with long input sequences due to the computational cost and memory requirements of backpropagation through time. Although theoretically capable of processing sequences of any length, RNNs face scalability challenges as sequence lengths increase, often leading to vanishing or exploding gradients. This makes it difficult for RNNs to capture long-term dependencies effectively.

Transformers address some of RNNs' inefficiencies by removing sequential computation in favor of parallel processing. However, this comes at the cost of increased memory usage due to the need to store large key-value pairs in caches for self-attention calculations. These memory requirements are inefficient, making Transformers less suitable for deployment on devices with limited memory capacity.

1.2.3 Inference Speed

The inference speed of transformers is affected by their linear complexity $O(n)$ per inference step, as each token computation during self-attention requires considering all other tokens in the sequence. This makes transformers less ideal for applications requiring fast responses, as the model’s response times scale linearly with sequence length. In decoder-only models, the process is inherently sequential; from a partial sequence, tokens are processed into embeddings and fed into the decoder. The output at the terminal token embedding represents a probability distribution across potential subsequent tokens. In the original transformer paper, they use the softmax function to transform logits into a probability distribution, determining the likelihood of each potential next token in the sequence. This distribution is then used to select and append the next token. The updated sequence is re-fed into the model to generate additional tokens. It is worth discussing the amount of computation needed for appending additional tokens to a sequence that already contains X tokens:

$$P(x_N = x | x_1, ..., x_{N-1}) = \frac{exp(z_{Nx})}{\Sigma_y exp(z_{Ny})} \quad (5)$$

where $P(x_N = x | x_1, ..., x_{N-1})$ is the probability of the next token x at position N given the sequence of previous tokens. Y ranges over all possible tokens in the vocabulary pool, making the denominator a summation over all exp-transformed logits. We can represent logit for the token x at position N, computed as the sum of dot products between the query and all key-value pairs:

$$z_{Nx} = \sum_{m=1}^{N} exp(q_N^T k_m^T) v_m \quad (6)$$

The sequence length decreases progressively. This slowdown is attributable to the linear complexity during inference and quadratic complexity during training, which inherently restricts the maximum sequence length manageable by transformer decoder models.

1.3 Introduction to Retentive Neural Networks (RetNets)

This paper complements the original 'Retentive Network' paper by sharing practical insights and lessons learned rather than exploring the theoretical underpinnings of the architecture. RetNet is designed as a potential successor to the transformer, optimizing for training parallelism, reducing inference costs, and maintaining robust performance. It reinterprets dot product attention as a recurrent neural network, establishing a link between recurrence and attention through a retention mechanism tailored for sequence modeling. This architecture supports parallel, recurrent, and chunkwise recurrent computations, enhancing training efficiency and inference cost-effectiveness by improving decoding speed and reducing latency. The chunkwise approach also handles long sequences with linear complexity, processing segments in parallel and integrating them recurrently. Initial experiments show that RetNet provides scalable training, cost-effective deployment, and effective inference, making it a viable alternative to traditional transformer models for expansive language models.

1.4 The Role of Frenos in Developing RetNets

Frenos, a leader in innovative data solutions, attempted to expand upon the work released by Microsoft Research to develop a robust RetNet model tailored for cybersecurity. Leveraging its expertise in data science and machine learning, Frenos aimed to create a scalable cybersecurity solution that meets the high demands for efficiency and performance in the field. We focused on building a RetNet model from a pre-trained state with specific operational technology domain knowledge, utilizing data across computer science, network security, coding, and use-case-specific graph data.

2. Detailed Description of the RetNet Model

2.1 Retentive Networks

This section will entail a deeper dive into the model. The RetNet model is based on the theoretical connection between recurrence and attention. It introduces a retention mechanism for sequence modeling, supporting three computation paradigms: parallel, recurrent, and chunkwise recurrent. Here's a deeper look into these principles.

2.1.1 Parallel Representation:

The parallel computational framework of RetNet allows for the simultaneous processing of multiple sequence segments. This is particularly beneficial in leveraging the high parallelism capabilities of modern GPUs, thereby accelerating the training process significantly. This mechanism transforms the input sequence into a state vector and output through a recurrent formulation. The transformation is facilitated by context-aware projections utilizing learnable matrices.

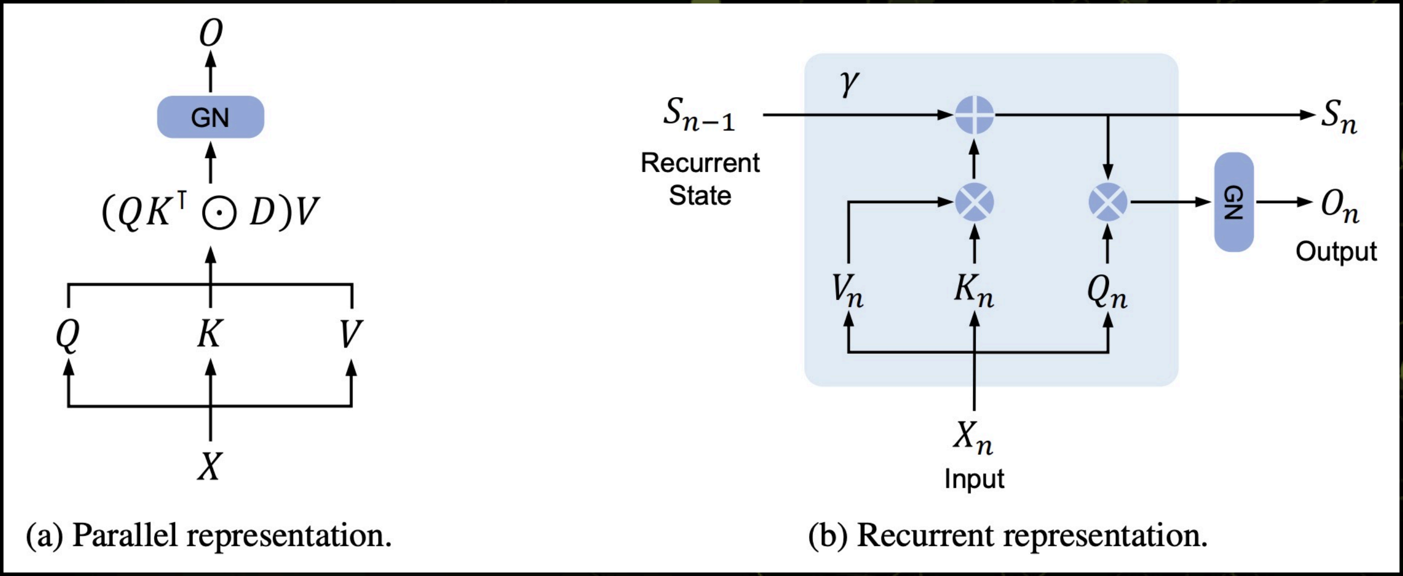


Figure 1. Dual form of RetNet. "GN" is short for GroupNorm. (B) The Proposed mechanism, which can be written as recurrent neural networks (RNNs), is favorable for inference and will be discussed in section 3.2.2.

The retention layer is defined as:

$$\begin{aligned} Q &= (XW_Q) \odot \Theta, \\ K &= (XW_K) \odot \overline{\Theta}, \\ V &= XW_V, \\ \overline{\Theta} &= e^{in\theta} \end{aligned} \tag{7}$$

where X represents the input data matrix, $W_{Q,K}$ is the weight matrix associated with the query and key. Without going through the entire derivation, we have projected Q_n and K_n to be context-aware, meaning the query and key matrices are transformed linearly into new Q and K matrices using learnable matrices:

$$Q = XW_Q \text{ and } K = XW_K \tag{8}$$

where X is our original input sequence and W_Q and W_K are learnable matrices of size d x d. This parallelization regarding the final output state with the retention layer can be represented in the following equation:

$$O_n = \sum_{m=1}^{n} \gamma^{n-m} (Q_n e^{in\theta})(K_m e^{im\theta})^\dagger v_m \tag{9}$$

where † is the conjugate transpose.

Query and key values are made "content aware," Now matrices are integrated using the exponential through the Hadamard Product, which conducts an element-wise multiplication and yields a matrix of identical dimensions to the matrices being applied. As indicated in the output state equations (9) and (10). For key values, the conjugate transpose $(K_m e^{im\theta})^{\dagger}$ is used.

The authors detail the transformation of the original equation into its final form, incorporating an exponential decay method:

$$D_{nm} = \{if\ \gamma^{n-m},\ n \geq m,\ else\ if\ 0,\ n < m\} \tag{10}$$

In sequence modeling tasks, causal masking prevents the model from attending to future elements during prediction. The matrix D_{nm} implements causal masking by setting its values to 0 when $n < m$, ensuring that the model does not use future information in its computations. Additionally, exponential decay is employed to gradually decrease the influence of elements as their distance from the current position increases. This is achieved by modifying the values of D based on the distance between elements, assigning higher importance to closer elements while exponentially reducing the influence of distant elements.

By combining causal masking and exponential decay through the matrix D, the model can effectively capture dependencies and patterns within the sequence while respecting the temporal order and focusing on the most relevant information for predicting the next element. Now, combining everything discussed above, we get our final Retention equation:

$$Retention(X) = (QK^{\mathsf{T}} \odot D)V \tag{11}$$

The computation involves performing a dot product between the Query (Q) and Key (K) matrices, followed by a Hadamard product (element-wise multiplication) with the matrix D. The resulting matrix is then multiplied with the Value (V) matrix using another dot product operation. A brief code snippet below shows the implementation of the parallel retention algorithm.

Python
```python
def ParallelRetention(
    q, # bsz * num_head * len * qk_dim
    k, # bsz * num_head * len * qk_dim
    v, # bsz * num_head * len * v_dim
    decay_mask # num_head * len * len
    ):
    retention = q @ k.transpose(-1,-2)
    retention = retention * decay_mask
    output = retention @ v
    output = group_norm(output)
    return output
```

This pseudocode represents the ‘ParallelRetention’ implementation, where the function is designed to handle inputs completely parallelly. This method calculates the retention scores by multiplying the query (*q*) and the transposed key (*k*) matrices. The scores

are then element-wise multiplied by a *decay_mask* to modulate the retention based on the distance between sequence elements, thereby simulating a form of attention decay. The final output is obtained by multiplying these adjusted scores with the value (*v*) matrix, followed by normalization using *group_norm*. This approach is highly efficient for scenarios requiring rapid processing with extensive parallel computation resources, such as GPU accelerations.

2.1.2 Recurrent Representation:

The recurrent representation in RetNet can be formulated similarly to Recurrent Neural Networks (RNNs), which are known for their ability to perform inference in constant time $O(1)$.

The recurrent representation in RetNet allows for efficient inference with time complexity $O(1)$ per step. This means that the computational cost of processing each input step remains constant, regardless of the total sequence length. The $O(1)$ complexity is achieved by updating the recurrent representation incrementally at each time step, using only the current input and the previous recurrent state. This is in contrast to self-attention-based models like Transformers, which have a time complexity $O(n^2)$ due to the need to attend to all positions in the sequence. By maintaining a constant-size recurrent representation and updating it incrementally, RetNet avoids the need to store and process the entire sequence at each step, resulting in low-cost inference with $O(1)$ complexity.

The recurrent representation in RetNet significantly reduces GPU memory usage and latency during inference compared to Transformer-based models. RetNet does not need to store the entire sequence in memory and can process the input streaming. In Transformer models, the self-attention mechanism requires computing the attention scores between all pairs of positions in the sequence. This results in memory complexity $O(n^2)$, where n is the sequence length. This can lead to high memory consumption, especially for long sequences. In contrast, RetNet's recurrent representation allows it to maintain a fixed-size memory footprint, regardless of the sequence length. The memory usage remains constant as the model processes the input step by step, updating the recurrent representation incrementally. The provided code snippet demonstrates how the recurrent retention mechanism is implemented. The RecurrentRetention function takes the current query (q), key (k), and value (v) tensors, along with the previous recurrent state (*past_kv*) and a decay factor. It computes the updated recurrent state (*current_kv*) by combining the current key-value pairs with the decayed previous state. The output is then obtained by summing the element-wise product of the query and the current recurrent state. RetNet significantly reduces GPU memory usage and inference latency compared to Transformer models by avoiding the need to store and process the entire sequence at once.

The low-cost inference and reduced memory usage of RetNet make it particularly suitable for real-time processing scenarios, such as online speech recognition, real-time language translation, or interactive conversational agents. In these applications, the input arrives continuously, and the model needs to generate outputs with minimal latency. RetNet's recurrent representation processes the input incrementally, generating outputs on the fly as new input arrives. The constant time complexity per step ensures fast processing, regardless of input length. This is crucial for real-time applications where the system needs to respond promptly. Additionally, reduced memory usage allows RetNet to handle longer sequences and operate efficiently on resource-constrained devices like mobile phones or edge devices. The provided code snippet showcases the efficiency of the recurrent retention mechanism. The RecurrentRetention function can be called iteratively for each input step, updating the recurrent state and generating the output incrementally. This enables efficient real-time processing, as the model can generate outputs as soon as new input arrives without waiting for the entire sequence to be available.

Python
```python
def RecurrentRetention(
        q,k,v # bsz * num_head * len * qkv_dim
        past_kv, # bsz * num_head * qk_dim * v_dim
        decay # num_head * 1 * 1
        ):
"""
Function to compute the recurrent retention of query (q), key (k), and value
(v) with past key and value (past_kv) and decay

Args:
        q (Tensor): Query tensor of shape bsz * num_head * len * qkv_dim
        k (Tensor): Key tensor of shape bsz * num_head * len * qkv_dim
        v (Tensor): Value tensor of shape bsz * num_head * len * qkv_dim
        past_kv (Tensor): Past key-value tensor of shape bsz * num_head * qk_dim
* v_dim
        decay (Tensor): Decay tensor of shape num_head * 1 * 1

Returns: output (Tensor): Output tensor after applying recurrent retention
current_kv (Tensor): Current key-value tensor after combining past and new
key-value """
        current_kv = decay * past_kv + k.unsqueeze(-1) * v.unsqueeze(-2)
        output = torch.sum(q.unsqueeze(-1) * current_kv, dim=-2)
        output = group_norm(output)
        return output, current_kv
```

In *RecurrentRetention*, the function processes the input sequence sequentially, mimicking recurrent neural network behavior. It combines the current input (*k, v*) with a decayed version of the previous key-value states (*past_kv*). This mechanism allows the model to maintain a continuous state across the sequence, thus capturing long-range dependencies efficiently. The recurrence is controlled by a decay factor, adjusting how much of the past state influences the current state. Such an approach is particularly useful for tasks that benefit from temporal continuity, like time-series prediction or language modeling.

2.1.3 Chunkwise Recurrent Representation:

The Chunkwise Recurrent Representation in the RetNet architecture is an innovative computational strategy designed to optimize the processing of long sequences by integrating the benefits of both parallel and recurrent mechanisms. This hybrid approach addresses the scalability challenges and computational bottlenecks typical of traditional sequential models like Transformers, especially regarding computational overhead and memory utilization.

$$Retention(X_{[i]}) = (Q_{[i]}K_{[i]}^{T} \odot D)V_{[i]} + (Q_{[i]}R_{[i]}) \odot \xi \qquad (12)$$

$$\text{where } \xi_{ij} = \gamma^{i+1}$$

$$Retention(X_{[i]}) = (inner\ chunk) + (cross\ chunk)$$

The chunkwise recurrent representation begins by segmenting the input sequence into manageable chunks to be processed independently in parallel. Once each chunk of the input sequence is processed independently and in parallel, allowing for significant gains in processing efficiency, the outputs are synthesized through a recurrent framework. This recurrent process is essential for preserving the continuity and dependencies across chunk boundaries, thereby maintaining the global context of the sequence. The recurrent summarization operates through a sequential transfer of states, where each chunk's output updates a continuously maintained state.

In previous research, the state transition involves sophisticated transformations, typically leveraging nonlinear activation functions and parameterized matrix operations to fuse and forward the contextual information. The update mechanism can include operations such as gated functions or normalization layers to stabilize the learning process and mitigate issues like vanishing or exploding gradients, common in deep sequential models. The RetNet model employs gated multi-scale retention, which will be discussed in the following sections. Below is a pseudo-code implementation of the *ChunkwiseRetention*.

```
Python
def ChunkwiseRetention(
        q,k,v, # bsz * num_head * chunk_size * qkv_dim
        past_kv, # bsz * num_head * qk_dim * v_dim
        decay_mask, # num_head * chunk_size * chunk_size
        chunk_decay # num_head * 1 * 1
        inner_decay # num_head * chunk_size
        ):
""" Perform chunkwise retention computation.

        Args: q (torch.Tensor):
                Query tensor of shape (bsz, num_head, chunk_size, qkv_dim).
                k (torch.Tensor): Key tensor of shape (bsz, num_head, chunk_size,
        qkv_dim)
                v (torch.Tensor): Value tensor of shape (bsz, num_head,
        chunk_size, qkv_dim)
        past_kv (torch.Tensor): Past key-value tensor of shape (bsz,
    num_head, qk_dim, v_dim)
        decay_mask (torch.Tensor): Decay mask tensor of shape (num_head,
    chunk_size, chunk_size)
    chunk_decay (torch.Tensor): Chunk decay tensor of shape (num_head, 1, 1)
        inner_decay (torch.Tensor): Inner decay tensor of shape (num_head,
    chunk_size)

    Returns:
        tuple: A tuple containing:
            - output (torch.Tensor): Output tensor of the chunkwise
        retention computation.
            - current_kv (torch.Tensor): Updated key-value tensor for
        the current chunk.
"""
    retention = q @ k.transpose(-1,-2)
    retention = retention * decay_mask
    inner_retention = retention @ v
    cross_retention = (q @ past_kv) * inner_decay
    retention = inner_retention + cross_retention
    output = group_norm(retention)
    current_kv = chunk_decay * past_kv + k.transpose(-1, -2) @ v
    return output, current_kv
```

The ChunkwiseRetention function represents a hybrid approach that combines parallel and recurrent computation elements. It processes the input data in chunks (chunk_size), applying parallel computation within each chunk while using a recurrent approach to integrate information across chunks. This method optimizes both the computational efficiency and the ability to capture dependencies over longer sequences, making it suitable for applications like video processing or document analysis where both local and global contexts are important.

2.2 Key Features and Components of the RetNet

2.2.1 Retention:

At the core of the retention mechanism is the ability to encode sequence information into contextualized vector representations. This is accomplished by projecting input sequences using a series of transformations sensitive to the input data's content. These projections are not static but dynamically adjusted by content-aware learnable matrices. This adaptability allows the network to tailor its processing strategy to the specific characteristics of the input, enhancing the model's ability to capture relevant patterns and dependencies.

The content-aware aspect of the projection is implemented via matrices (W_Q) and (W_K), which transform the input sequence into query (Q) and key (K) vectors, respectively. This transformation process is pivotal as it determines how information is retained and processed across different sequence parts. RetNet can more effectively manage the relationships and interactions between different sequence elements by making these projections content-aware, leading to more nuanced and powerful sequence modeling.

2.2.2 Multi-Scale Retention (MSR) Module:

The Multi-Scale Retention (MSR) module serves as the core building block of RetNet, replacing the traditional multi-head attention mechanism found in Transformer architectures. The MSR module is designed to capture and model dependencies at various scales within the input sequence. It achieves this by employing different parameter matrices for each retention head, enabling the module to effectively learn and represent multi-scale patterns. These matrices are specifically tailored to address different patterns or features, allowing the MSR to represent multi-scale patterns across the data adeptly.

The MSR module supports three distinct computation paradigms: parallel, recurrent, and chunkwise recurrent. The parallel paradigm leverages GPU capabilities to maximize training efficiency by processing all input positions simultaneously. In contrast, the recurrent paradigm is optimized for inference, promoting low-cost, constant-time complexity operations that enhance decoding throughput and reduce latency and GPU memory usage. The chunkwise recurrent paradigm facilitates efficient long-sequence modeling, encoding each local block in parallel while summarizing global information recurrently to save on GPU memory.

$$MSR(X) = Concat(head_1, head_2, \dots head_n)W^O \tag{13}$$

where $X \in R^{n\,x\,d}$ is the input sequence, $head_i$ represents the output of the i-th retention head, n is the number of retention heads, and $W^O \in R^{hd_v * d}$ is the parameter matrix that combines the outputs of the retention heads. Each head's distinct configuration of parameter matrices supports a sophisticated multi-scale modeling capability, significantly enhancing the model's performance in diverse applications.

```python
# Pseudocode Representation for the the Multi-Scale Retention (MSR) Module
def MultiScaleRentention(X, n_heads, WQ, WK, WV, WO, paradigm='parallel',
chunk_size=None):
"""
        Computes the Multi-Scale Retention (MSR) output for the given input
        sequence X.
        Parameters:
        - X: Input sequence matrix (batch_size, sequence_length, input_dim)
        - n_heads: Number of retention heads
        - WQ, WK, WV, WO: Parameter matrices for queries, keys, values, and
        output combination
```

```
- paradigm: Computation paradigm ('parallel', 'recurrent',
'chunkwise_recurrent')

- chunk_size: Size of chunks for the chunkwise recurrent paradigm
Returns: - MSR output
"""
# Process each head
for i in range(n_heads):
Q = np.dot(X, WQ[i])
K = np.dot(X, WK[i])
V = np.dot(X, WV[i])
if paradigm == 'parallel':
       output = parallel_computation(Q, K, V)
elif paradigm == 'recurrent':
       output = recurrent_computation(Q, K, V)
elif paradigm == 'chunkwise_recurrent' and chunk_size is not None:
       output = chunkwise_recurrent_computation(Q, K, V, chunk_size)
else:
       raise ValueError("Invalid computation paradigm")
heads_outputs.append(output)

# Concatenate outputs from all heads and apply output transformation
matrix
concatenated_outputs = np.concatenate(heads_outputs, axis=-1)
MSR_output = np.dot(concatenated_outputs, WO)
return MSR_output
```

2.2.3 Gated Multi-Scale Retention:

The researchers use $h = \frac{d_{model}}{d}$ retention heads in each layer, where d is the head dimensions. Each of these heads uses different parameter matrices W_Q, W_K, W_V and is assigned a different γ for each head.

The FFN module in the RetNet architecture is similar to that of traditional transformers, consisting of two linear transformations with a non-linearity gated activation function between. The gated activation function used to increase the non-linearity of the retention layers is known as swish:

$$f(x) = x \cdot \sigma(x) \tag{14}$$

where $\sigma(x) = (1 + exp(-x))^{-1}$ is the sigmoid function. See Figure 2 for the Swish graph.

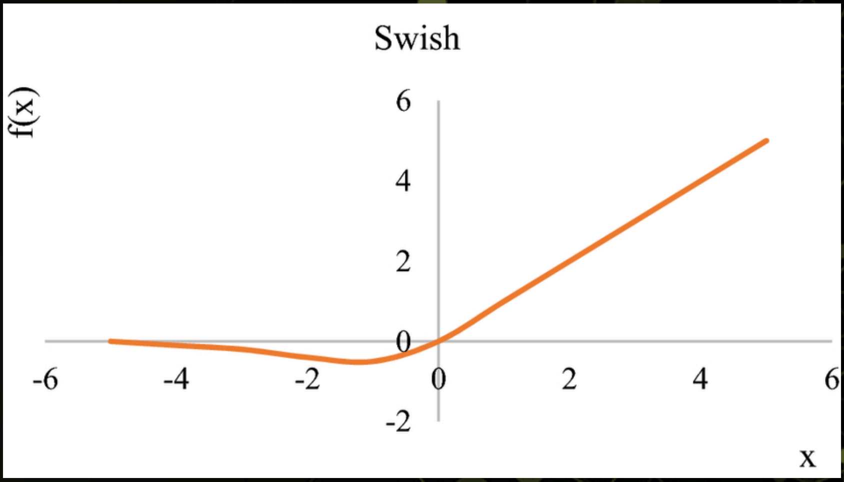


Figure 2. The Swish activation function.

The Swish activation function incorporates a sigmoid gating mechanism that modulates the network's input signal flow. This gating effect is crucial for managing the information dynamics within the network's layers. By allowing each unit to control the flow of its own activation, Swish helps adjust the propagation of gradients during the backpropagation process. This adaptability can be particularly beneficial in deep network architectures, where controlling the gradient flow is essential for efficient training and avoiding issues like vanishing or exploding gradients, which are large problems in recurrent neural networks.

The smoothness of the Swish function facilitates a more continuous and differential behavior across the input domain. Unlike ReLU, which has a non-differentiable point at zero and completely nullifies negative inputs, Swish applies a soft suppression based on the sigmoid function. This smoothness ensures that gradients are more stable across the entire input range, thereby supporting better optimization and convergence behaviors. The non-monotonic property of Swish, where the function decreases in the negative input domain before transitioning to an increasing function, allows it to handle negative and positive inputs in a dynamically balanced manner. This characteristic is advantageous for modeling complex patterns where the presence and absence of certain features (represented by positive and negative signals) are informative for the learning task. This FFN follows the MSR mechanism in each block of the network.

2.2.3 Normalization Techniques and Connectivity:

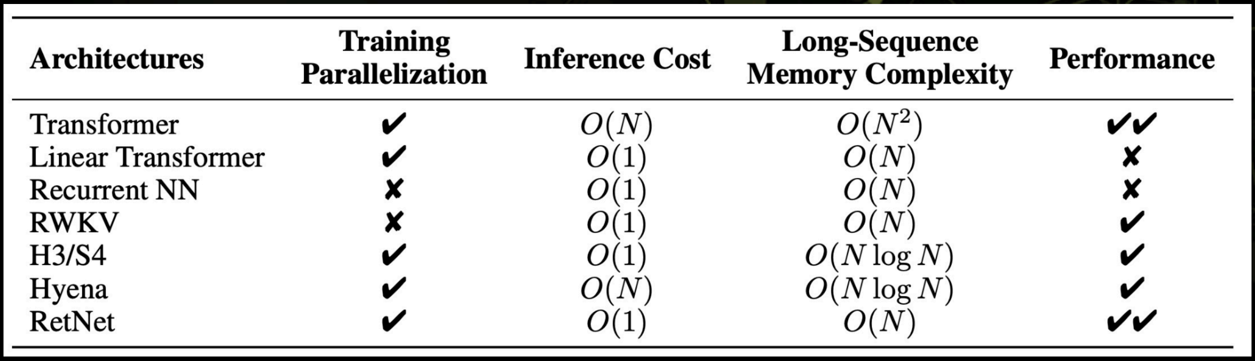
In RetNet, Group Normalization (GroupNorm) is utilized within each retention layer to ensure stable outputs across diverse feature groups. This allows multiplying a scalar value without impacting the outputs and backward gradients. GroupNorm is particularly beneficial in varied batch-size environments, as it operates independently of batch dynamics. Layer Normalization (LayerNorm), uniformly applied across all features of each layer, plays a critical role in maintaining robustness within the RetNet architecture. This normalization adjusts the scale of activations throughout each layer's output, effectively countering internal covariate shifts. LayerNorm facilitates model convergence and addresses the vanishing gradient problem, ensuring stable training across the network's depth.

Residual Connections, drawing on principles from traditional transformers, are used in each RetNet block. These connections create a direct pathway for gradients during backpropagation, which is essential for efficiently training deeper networks. They prevent the loss of information through successive transformations and combat the degradation in learning performance as network depth increases.

2.2.4 Efficient Inference and Memory Usage:

The recurrent representation ensures that the computational complexity during inference is $O(1)$ concerning sequence length, which is well-suited for autoregressive decoding. This is because the model leverages previously computed states, updating them with new information rather than starting from scratch for each input in the sequence. Therefore, the cost per step remains constant regardless of the input sequence length. The inherent sequential processing of the recurrent model is more memory-efficient as it avoids the redundancy of recalculating activations, instead updating its state incrementally with each new input.

Due to the architectural design, the RetNet model leads to a significant reduction in GPU memory during both training and inference. In traditional deep-learning models, memory typically increases linearly with sequence length due to the need to store intermediate outputs for each timestep. The memory complexity for long sequences is $O(N)$ due to efficiently managing the storage of hidden states and gradients during the forward and backward passes. This management entails gradient checkpointing, memory efficient attention, and batch processing, all contributing to overall enhanced efficiency in the RetNet architecture.



Architectures	Training Parallelization	Inference Cost	Long-Sequence Memory Complexity	Performance
Transformer	✔	$O(N)$	$O(N^2)$	✔✔
Linear Transformer	✔	$O(1)$	$O(N)$	✘
Recurrent NN	✘	$O(1)$	$O(N)$	✘
RWKV	✘	$O(1)$	$O(N)$	✔
H3/S4	✔	$O(1)$	$O(N \log N)$	✔
Hyena	✔	$O(N)$	$O(N \log N)$	✔
RetNet	✔	$O(1)$	$O(N)$	✔✔

Table 1. Model architecture comparison. RetNet achieves training parallelization, constant inference cost, linear long-sequence memory complexity, and good performance.

3. Experimentation, Implementation, and Discussion

Pre-training a large language model involves seeding knowledge. At Frenos, we consider pre-training like going through primary school. This is where you learn the foundations of language and reasoning.

In pre-training the RetNet model, Frenos focused on comprehensively seeding knowledge related to language understanding, contextual reasoning, and domain-specific knowledge embedding. The approach involved training the model on diverse datasets to ensure the acquisition of foundational language and reasoning skills. Frenos leverages various linguistic data and knowledge resources, laying a robust foundation for the RetNet model's language understanding capabilities.

The pre-training process also involved fine-tuning the model on specific tasks and domains to enhance its ability to comprehend and assimilate domain-specific knowledge. Frenos utilized the capabilities provided by the Huggingface transformer Python library to fine-tune the RetNet model on task-specific datasets, enhancing its adaptability and proficiency in addressing real-world challenges.

We needed to ensure that at least 25% of the training data involved reasoning tasks to enhance the model's performance. Our varied sources provided rich and comprehensive linguistic and domain-specific knowledge, enabling the model to develop a strong understanding of language and reasoning skills.

One critical factor in enhancing the model's performance was incorporating a mix of coding and math datasets for reasoning tasks. We found that the model's results were unsatisfactory if there was less than a 15% mixture of language seeding texts compared to reasoning texts. Therefore, by carefully curating and including a balanced mixture of text sources, we ensured that the RetNet model acquired a robust foundation in language understanding, contextual reasoning, and domain-specific knowledge embedding.

The comprehensive data collection and analysis techniques enabled us to assess and leverage our datasets' rich linguistic and domain-specific knowledge resources. This approach laid a solid groundwork for the RetNet model, empowering it with the capabilities to comprehend and assimilate diverse sets of information, ultimately enhancing its adaptability and proficiency in addressing real-world challenges.

The biggest challenge was the amount of tokens needed to pre-train the large language model. For our experimentation, we wanted to implement a three billion parameter model, which would be the right size for many downstream tasks. Our first iteration resulted in training the model with 70 million tokens. Results showed that the model could recite pieces of data that it was pre-trained on and was coherent. Yet, after fine-tuning for instruction, we noticed degraded performance when asking the model simple questions and tasks. After analyzing the performance, we realized the model lacked exposure to diverse data and reasoning.

The next iteration of training utilized over 4 billion tokens emphasizing reasoning datasets. This ensures a more diverse linguistic and domain-specific knowledge for the model's pre-training. Additionally, we prioritized incorporating a mix of coding and math datasets for reasoning tasks to enhance the model's performance.

Upon fine-tuning the RetNet model with a focus on reasoning tasks and diverse linguistic data, we observed improved results compared to its predecessor. This shift in the pre-training process resulted in an enriched foundational knowledge base for the RetNet model, ultimately contributing to its improved adaptability and proficiency in addressing real-world challenges.

Comparison of RetNet Model and Transformer-Based Architecture

After analyzing the RetNet model's performance, we conducted a comparative study to evaluate its architecture against transformer-based models regarding inference speed. The results clearly illustrate that regardless of the sequence length, the RetNet model exhibits constant processing time, highlighting its efficiency and advantage over transformer-based architectures, which demonstrated quadratic $O(n^2)$ processing time. This comparison underscores the superior inference speed and computational efficiency of the RetNet model's architecture, showcasing its potential for real-world applications that require rapid processing of large sequences.

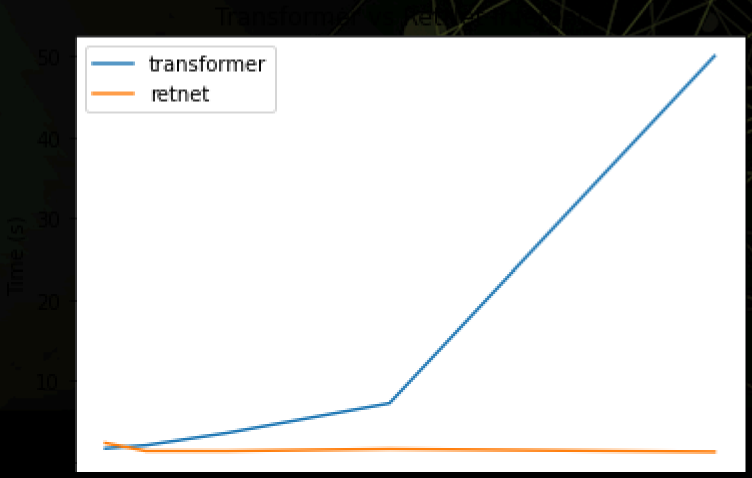


Figure 3. RetNet inference times compared to transformers at various context lengths.

The insights from this comparative analysis emphasize the significance of architectural considerations in achieving optimal model performance for specific tasks. It's been determined that the RetNet model's architecture is the optimal choice when comparing the speed of inference and the length of the token context. In our future endeavors, we aim to refine the RetNet model's architecture to capitalize on its inherent advantages and continue advancing its capabilities for various language understanding and reasoning tasks.

4. Resources

1) Sun, Y., Dong, L., Huang, S., Ma, S., Xia, Y., Xue, J., Wang, J., & Wei, F. (2023). "Retentive Network: A Successor to Transformer for Large Language Models."
2) Ma, S., Wang, H., Huang, S., Wang, W., Chi, Z., Dong, L., Benhaim, A., Patra, B., Chaudhary, V., Song, X., & Wei, F. (2022). "TorchScale: Transformers at Scale." *CoRR*, abs/2211.13184.
3) Vaswani, A., et al. (2017). "Attention is All You Need." *Advances in Neural Information Processing Systems*, arXiv:1706.03762.
4) Lukyanenko, A. (2023). "Paper Review: Retentive Network: A Successor to Transformer for Large Language Models." *Andrey Lukyanenko's Blog*, 24 July 2023.
5) Ramachandran, P., Zoph, B., & Le, Q. V. (2017). "Swish: a self-gated activation function." *arXiv: Neural and Evolutionary Computing*.
6) Hao, Y., Sun, Y., Dong, L., Han, Z., Gu, Y., & Wei, F. (2022). "Structured prompting: Scaling in-context learning to 1,000 examples." *ArXiv*, abs/2212.06713.
7) Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). "Layer normalization." *arXiv preprint arXiv:1607.06450*.
8) Wang, H., Ma, S., Dong, L., Huang, S., Wang, W., Chi, Z., Benhaim, A., Patra, B., Chaudhary, V., Song, X., & Wei, F. (2022). "DeepNet: Scaling Transformers to 1,000 Layers." *arXiv preprint arXiv:2203.00555*.
9) Wang, H., Ma, S., Dong, L., Huang, S., Wang, W., Chi, Z., Benhaim, A., Patra, B., Chaudhary, V., Song, X., & Wei, F. (2023). "LongNet: Scaling Transformers to 1,000,000,000 Tokens." *arXiv preprint arXiv:2307.02486*.
10) Wang, H., Ma, S., Huang, S., Wang, W., Chi, Z., Dong, L., Benhaim, A., Patra, B., Chaudhary, V., Song, X., & Wei, F. (2023). "BitNet: Scaling 1-bit Transformers for Large Language Models." *arXiv preprint arXiv:2310.158*.
11) Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). "Scaling Laws for Neural Language Models." arXiv preprint arXiv:2001.08361.
12) Jurafsky, D., & Martin, J. H. (2023). *Speech and Language Processing*. Draft of February 3, 2024. Chapter 10: Transformers Large Language Models.
13) Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Rae, J. W., Vinyals, O., & Sifre, L. (2022). "Training Compute-Optimal Large Language Models." arXiv, arXiv:2203.15556.