Embformer: An Embedding-Weight-Only Transformer Architecture

Author: Wu Hecong **Assistant:** Gemini 2.5 Pro **Version:** 1.0 **Date:** 2025-06-25

Abstract

The computational and memory demands of Large Language Models (LLMs), partially driven by dense matrix multiplications in linear layers, pose significant challenges for training and deployment. This research note introduces **Embformer**, a novel Transformer architecture that radically re-imagines model parameterization. Embformer replaces all linear projection layers (for queries, keys, values, and feed-forward networks) with direct lookups from trainable embedding tables. This design transforms the model's core operation from intensive computation to efficient memory retrieval. We first provide a mathematical proof-of-concept demonstrating how a linear layer can be equivalently expressed as a scaled dot-product attention mechanism, thereby motivating our decision to remove the linear layer from the model architecture. Building on this insight, we designed and implemented an Embformer model where all weights are stored in token-specific embedding layers. A preliminary experiment, training a 16-layer model, confirms the architecture's feasibility; the model can be trained and perform inference. However, its performance currently lags behind traditional Transformers of similar parameter counts, a result we attribute to the sparse activation of its parameters during any single forward pass. This work presents a new paradigm for LLM architecture, trading computational density for memory-based lookup, and outlines key challenges and future directions, including optimizing attention mechanisms and managing embedding memory during training. Code is available at https://github.com/HighCWu/embformer.

1. Introduction

The remarkable success of Transformer-based Large Language Models (LLMs) (Vaswani et al., 2017) is intrinsically linked to their immense scale. However, this scale comes at a cost: the vast number of parameters, primarily housed in linear layers, leads to substantial computational overhead (dominated by matrix-matrix multiplications) and high VRAM consumption during both training and inference.

Recent research has explored innovative ways to mitigate these costs. For instance, RWKV's **DeepEmbed** (Peng, 2025) introduces layer-wise, token-specific trainable vectors within the FFN. These vectors, which function as per-token channel-wise scaling factors, can be offloaded to slower memory (RAM/SSD) during inference, drastically reducing the active VRAM footprint. Similarly, Google's **Per-Layer Embeddings (PLE)** in the Gemma 3n model (Google, 2025) follows a comparable principle, demonstrating the industry's interest in hybrid memory-computation models.

These approaches suggest a promising direction: can we extend this "lookup-over-computation" philosophy to its logical conclusion? This research note investigates this question by proposing **Embformer**, an architecture that completely eliminates linear layers in favor of embedding lookups. Our core hypothesis is that the functional role of linear projections can be fulfilled by retrieving pre-trained, token-specific vectors from large but efficient embedding tables.

This note makes the following contributions:

1. A mathematical and code-based demonstration that linear layer operations can be reframed as an attention mechanism.
2. The design of the Embformer architecture, which sources all query, key, and value vectors, as well as FFN parameters, directly from embedding layers.
3. A rapid experimental validation of the architecture, confirming its viability and characterizing its performance trade-offs.

2. Methodology

2.1. Theoretical Foundation: Re-framing Linear Layers as Attention

The foundational insight for Embformer is that the operation of a linear layer, `y = x @ W^T`, can be mathematically reconstructed as a specialized form of attention. This equivalence allows us to substitute the computational primitive of matrix multiplication with the structural primitive of an attention block.

Let's consider a single input vector `x` of size `in_features` and a linear layer weight matrix `W` of size `(out_features, in_features)`. The output `y` is of size `out_features`. We can express this as an attention operation where the input `x` defines the attention weights (`attn_weights`) and the linear layer's weight `W` acts as the value matrix (`v`). In this view, the sequence length of the attention mechanism becomes equivalent to the input feature dimension (`in_features`).

The following Python code using PyTorch demonstrates this equivalence.

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

# Set a seed for reproducibility
torch.manual_seed(42)

# --- Step 1: Compute a standard linear layer result ---
x = torch.randn(4, 8)  # Batch size=4, in_features=8
fc_weight = torch.randn(32, 8)  # out_features=32, in_features=8
y_linear = x @ fc_weight.T

# --- Step 2: Re-frame as an attention operation ---
# The input `x` determines the attention weights. We use its absolute value
normalized
# to form a probability distribution. The sign and total magnitude are
folded into the value `v`.
x_abs = x.abs()
x_sum = x_abs.sum(dim=-1, keepdim=True)
attn_weights = (x_abs / x_sum)[:, None]  # (bsz, 1, seq_len) where
seq_len=in_features

# The linear weight `fc_weight` becomes the base of the value `v`.
# The magnitude and sign of the input `x` are used to scale `v`.
v = fc_weight.T[None] * (x_sum[..., None] * x.sign()[..., None])
# v shape: (bsz, seq_len, out_dim) -> (4, 8, 32)
```

```
# The attention output is calculated by multiplying attn_weights and v.
y_attention_manual = (attn_weights @ v)[:, 0]

# Verification: The results are identical.
assert torch.allclose(y_attention_manual, y_linear)

# --- Step 3: Recovering Q and K for a standard attention function ---
# We can reverse-engineer a valid (q, k) pair that produces our
`attn_weights`
# after a softmax operation. This is not a unique solution, but
demonstrates compatibility.

# Start from our desired attention weights and derive pre-softmax logits.
# We add a random constant to show that softmax is shift-invariant.
logits = torch.log(attn_weights)
logits = logits - torch.randn(1)

# Verify that softmax(logits) recovers our attention weights.
assert torch.allclose(torch.softmax(logits, dim=-1), attn_weights)

# Now, we can factorize the scaled logits `s = logits * sqrt(d_k)` into `q
@ k^T`.
# We choose a random k and solve for q = s @ inv(k^T).
d_k = 8
s = logits * math.sqrt(d_k)
k = torch.randn(4, 8, d_k) # bsz, seq_len, d_k
k_inv_t = torch.linalg.inv(k.transpose(-2, -1))
q = s @ k_inv_t

# --- Step 4: Final verification with F.scaled_dot_product_attention ---
# Using the recovered q, k, and our constructed v, we can use the standard
# PyTorch attention function to get the final result.
y_sdpa = F.scaled_dot_product_attention(q, k, v)[:, 0]

# The final output matches the original linear layer output, confirming the
equivalence.
assert torch.allclose(y_sdpa, y_linear, atol=1e-3)
```

This equivalence, while not requiring the explicit recovery of `q` and `k` in our final model, provides the theoretical justification for replacing a linear layer with an attention block whose `value` matrix is derived from the original layer's weights and whose attention pattern is determined by the input `x`.

2.2. The Embformer Architecture

The Embformer architecture builds upon this principle by systematically replacing all linear layers with embedding lookups that feed into attention mechanisms or perform direct scaling.

Inspiration from Prior Work: Our design is inspired by two key findings:

1. **Layer-wise Knowledge Storage (DeepEmbed, PLE):** The idea that token-specific knowledge can be stored in large, layer-specific embedding tables is a powerful one for model expressivity and inference efficiency.

2. **Information Concentration in Top Layers (Wu & Tu, 2024):** Research has shown that the top-layer Key-Value (KV) cache in Transformers contains the most crucial information for generation. This suggests that focusing parameterization efforts on these representations is a worthwhile strategy. Since the top-layer KV representations are closely related to the original input embeddings, it becomes feasible to directly substitute them with embedding outputs. However, recognizing that removing linear layers severely limits model expressive capability, we retain the use of separate KVs across layers (as in traditional Transformers), but with their values sourced entirely from different embedding layer outputs.

Architectural Design:

- **Removal of Linear Layers:** All `nn.Linear` layers for Query (Q), Key (K), Value (V) projections, as well as those in the Feed-Forward Network (FFN), are completely removed from the standard Transformer block.
- **Embedding-based Projections:**
  - The **query vector** for the top-most attention layer is retrieved directly from a dedicated embedding table: `Q_top = Embedding_Q_top[token_id]`.
  - The **key and value vectors** for *all* layers are sourced from distinct, layer-specific embedding tables: `K_layer_i = Embedding_K_i[token_id]` and `V_layer_i = Embedding_V_i[token_id]`. This maintains layer-wise representational diversity, which is critical for model capacity.
- **Embedding-based FFN:** The traditional FFN (typically a two-layer MLP) is replaced with a DeepEmbed-style channel-wise scaling operation. The output of the multi-head attention block is element-wise multiplied by a vector retrieved from a layer-specific FFN embedding table: `FFN_output = Attention_output * Embedding_FFN_i[token_id]`.
- **Channel Mixing:** Since the removal of linear layers eliminates the mechanism for information to mix across attention heads and channels within the FFN, we introduce a simple, parameter-free channel mixing mechanism after the scaling operation to facilitate partial information exchange.
- **Parameter Tying:** The final `lm_head` used for token prediction is tied to the weights of the initial input embedding layer, meaning that the only linear layer's weights are also derived from the embedding layer.

The resulting model is one where all of trainable parameters are housed in `nn.Embedding` layers.

3. Rapid Experiment

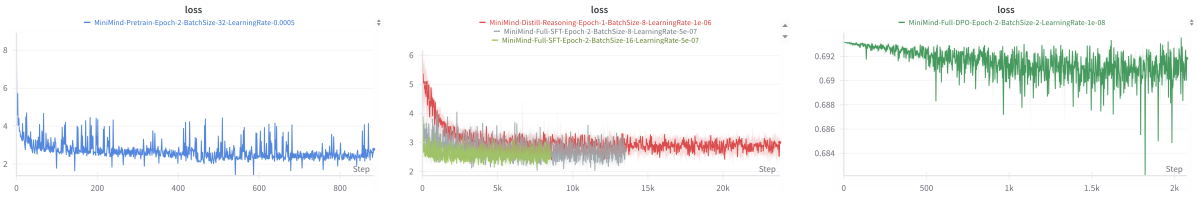
To validate the feasibility of the Embformer concept, we conducted a preliminary training experiment.

3.1. Experimental Setup

- **Base Code:** We modified the Qwen3 model implementation from the Hugging Face `transformers` library.
- **Training Framework:** We utilized the MiniMind framework, which provides a complete training pipeline and associated dataset.
- **Model Configuration:**
  - Number of Layers: 16
  - Hidden Size: 768

- **Vocabulary:** A key aspect of this initial experiment was the use of a small vocabulary of **6,400 tokens**. This was a pragmatic choice to keep the memory footprint of the numerous embedding layers manageable on our available hardware. The memory consumption of this architecture scales linearly with `vocab_size * num_layers * hidden_size`.

3.2. Results and Analysis



The training experiment yielded results that aligned with our expectations. The Embformer model was able to train successfully, and the loss curve demonstrated a clear learning trend. The resulting model could perform inference and generate coherent (though simple) text, confirming that the architecture is functionally viable.

However, its performance was **notably worse** than a traditional Transformer model with a similar configuration (16 layers, 768 hidden size). The primary reason for this performance gap is the **sparse activation of parameters**. In a standard Transformer, the dense weight matrices of the linear layers are fully utilized in the computation for every token. In Embformer, for any given forward pass, only the embedding vectors corresponding to the specific tokens in the input sequence are activated and used. This means that while the total parameter count might be large, only a tiny fraction of the model's "knowledge" is brought to bear on any single prediction. The model is parameter-rich but computation- and activation-poor.

4. Conclusion and Future Work

Our research demonstrates that it is feasible to construct a Transformer-like model, Embformer, by replacing all linear layers with embedding-based lookups. This establishes a proof-of-concept for a new architectural paradigm that trades dense computation for sparse memory access.

The primary challenge is the performance deficit compared to traditional models. We identify two critical areas for future work to bridge this gap and unlock the potential of this architecture:

1. **Improving Attention Efficiency:** In Embformer, the scaled dot-product attention operation becomes the main computational bottleneck, as it is one of the few remaining dense computations. Increasing model depth or width to improve performance will exacerbate this bottleneck. Future work should focus on integrating more efficient attention mechanisms (e.g., linear attention, state-space models, or kernel-based methods) that scale better with sequence length and model size.

2. **Managing Training-Time Memory:** The current design's memory footprint is a direct function of vocabulary size. To train effective models on standard vocabularies (e.g., 32k-128k tokens), the memory requirements for the embedding tables would become prohibitive for most training setups. The next crucial step is to explore and implement strategies for offloading embedding weights to CPU RAM or SSD during training, leveraging techniques akin to DeepSpeed's ZeRO-Offload. Since the architecture is inherently based on lookups, this approach is a natural fit.

In summary, Embformer presents a nascent but promising direction for building highly efficient LLMs. While significant challenges remain, its unique trade-off between computation and memory offers a compelling path toward models that are faster to run and more accessible for deployment on resource-constrained edge devices.

5. References

- Gong, J. (2024). *MiniMind: Train a 26M-parameter GPT from scratch in just 2h*. GitHub repository. https://github.com/jingyaogong/minimind
- Google. (2025). *Announcing Gemma 3n preview: powerful, efficient, mobile-first AI*. Google for Developers Blog. https://developers.googleblog.com/en/introducing-gemma-3n/
- Peng, B. (2025). *DeepEmbed Explanation*. https://x.com/BlinkDL_AI/status/1926941496684519805 and RWKV Official Website. https://www.rwkv.cn/news/read?id=20250527
- Ren, J., Rajbhandari, S., Aminabadi, R. Y., Ruwase, O., Yang, S., Zhang, M., ... & He, Y. (2021). *ZeRO-Offload: Democratizing Billion-Scale Model Training*. arXiv preprint arXiv:2101.06840. https://arxiv.org/abs/2101.06840
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). *Attention is all you need*. Advances in neural information processing systems, 30.
- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., ... & Rush, A. M. (2020). *Transformers: State-of-the-Art Natural Language Processing*. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations (pp. 38-45).
- Wu, H., & Tu, K. (2024). *Layer-Condensed KV Cache for Efficient Inference of Large Language Models*. arXiv preprint arXiv:2405.10637. https://arxiv.org/abs/2405.10637