OneSparse: A Unified Framework for Sparse Activation Layers in Vision Models

Supplementary Material

A. Visual Analysis of Functional Specialization

We visualize routing probabilities in deep ConvNeXt stages to test whether the Unified Router allocates broad context to Memory Units and fine semantic refinement to Expert Units. Fig. A1 shows a clear separation between the two branches, consistent with the intended adaptive processing strategy.

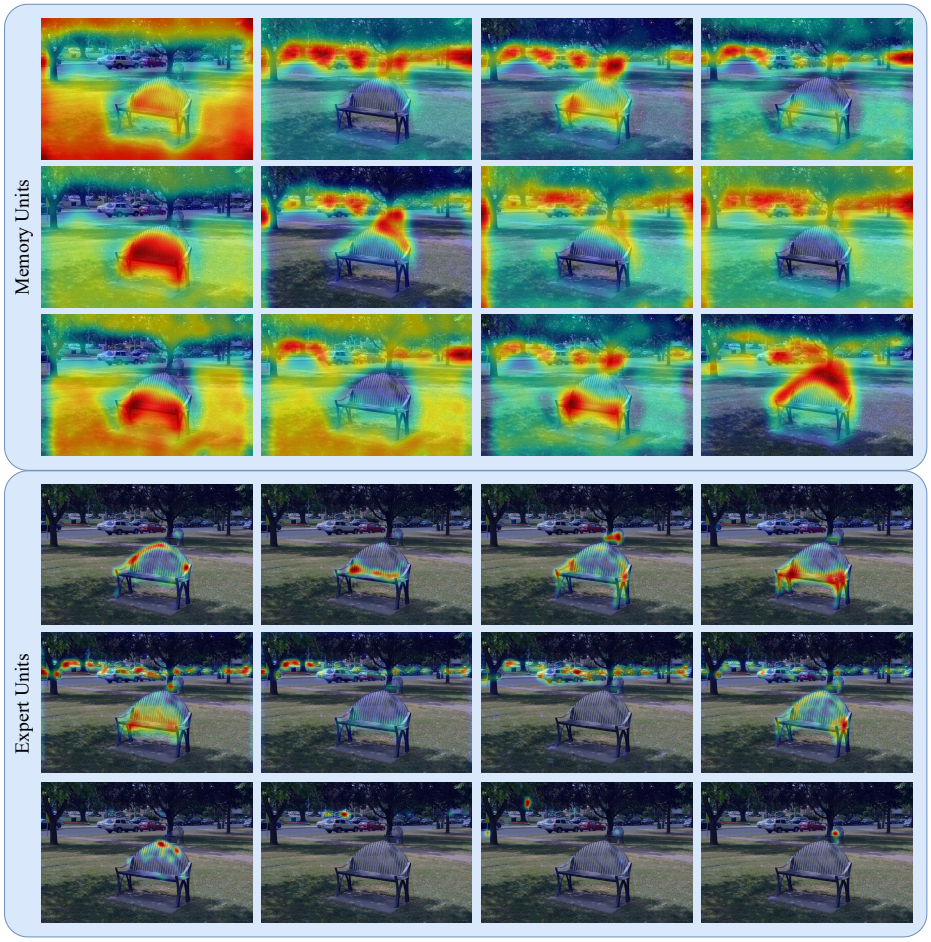

Figure A1. **Visualizing Functional Specialization. Top (Memory Units):** Diffuse, image-wide activations indicate coarse-grained global modeling. **Bottom (Expert Units):** Concentrated responses on salient objects and boundaries indicate fine-grained refinement.

As shown in Fig. A1, memory units exhibit broad image-wide attention, whereas expert units concentrate on salient objects and boundaries. This result supports the view that the Nexus Layer separates coarse context modeling from fine-grained refinement without explicit supervision.

B. Implementation of Nexus Layer

We provide PyTorch-style pseudo-code for the Nexus Layer, separating the core routing procedure from the processing sub-modules.

Algorithm A1 implements *Unified Routing*, while Algorithm A2 defines the expert and memory branches.

```python
class NexusLayer(nn.Module):
    def __init__(self, dim, num_experts, num_memory_units, slots_per_unit):
        super().__init__()
        self.num_experts = num_experts
        self.num_memory = num_memory_units
        self.total_units = num_experts + num_memory_units

        # ============================================================
        # Unified Router Parameters
        # ============================================================
        # Learnable queries for all processing slots.
        # Experts and memory groups are treated uniformly as units.
        # Q: (Total_Units, Slots, Dim)
        self.router_queries = nn.Parameter(torch.randn(self.total_units, slots_per_unit, dim))
        # Temperatures for the dispatch and combine softmaxes
        self.router_temp = nn.Parameter(torch.tensor([1.0, 1.0]))

        # ============================================================
        # Adaptive Processing Units
        # ============================================================
        # Branch 1: Dynamic experts
        self.experts = MultiExpertBlock(dim, num_experts)
        # Branch 2: Static memory
        self.memory = MemoryRetrievalBlock(dim, num_memory_units)

    def forward(self, x):
        """
        Input: x (Batch, N_tokens, Dim)
        Output: y (Batch, N_tokens, Dim)
        """
        # 1. Calculate Affinity Scores (Eq. 10)
        # Project input tokens to the router's query space
        # affinity: (Batch, N, Total_Units, Slots)
        affinity = einsum("b n d, e s d -> b n e s", x, self.router_queries)

        # 2. Generate Routing Tensors
        # Dispatch tensor D: softmax over tokens
        D = torch.softmax(affinity / self.router_temp[0], dim=1)

        # Combine tensor B: softmax over slots
        combine_logits = affinity / self.router_temp[1]
        B = torch.softmax(combine_logits.flatten(start_dim=-2), dim=-1).view_as(affinity)

        # 3. Dispatch Stage (Eq. 11 Inner Sum)
        # Aggregate tokens into slots
        z_in = einsum("b n e s, b n d -> b e s d", D, x)

        # 4. Adaptive Processing (Heterogeneous Execution)
        z_exp_in = z_in[:, :self.num_experts, ...]
        z_mem_in = z_in[:, self.num_experts:, ...]

        # Branch 1: Expert units
        z_exp_out = self.experts(z_exp_in)
        # Branch 2: Memory units
        z_mem_out = self.memory(z_mem_in)

        # Concatenate outputs
        z_out = torch.cat([z_exp_out, z_mem_out], dim=1)

        # 5. Combine Stage (Eq. 11 Outer Sum)
        # Reconstruct final token representations
        y = einsum("b n e s, b e s d -> b n d", B, z_out)

        return y
```

Algorithm A1: Core implementation of the Nexus Layer for unified dispatch and combine.

```python
class MultiExpertBlock(nn.Module):
    """Vectorized implementation of multiple FFN experts."""
    def __init__(self, dim, num_experts, mlp_ratio=4):
        super().__init__()
        # Flexible control over expert capacity
        hidden_dim = int(dim * mlp_ratio)

        # Batched weights for parallel execution of all experts
        self.w1 = nn.Parameter(torch.randn(num_experts, dim, hidden_dim))
        self.w2 = nn.Parameter(torch.randn(num_experts, hidden_dim, dim))

    def forward(self, x):
        # x: (Batch, Num_Experts, Slots, Dim)
        # Standard FFN applied in parallel across the 'Num_Experts' dimension
        x = einsum("b e s d, e d h -> b e s h", x, self.w1)
        x = F.relu(x)
        x = einsum("b e s h, e h d -> b e s d", x, self.w2)
        return x

class MemoryRetrievalBlock(nn.Module):
    """Memory processing via efficient Key-Value retrieval."""
    def __init__(self, dim, num_units, num_keys=1024, topk=4):
        super().__init__()
        self.topk = topk
        self.num_keys = num_keys

        # Keys are used for computing similarity (Content addressing)
        self.mem_keys = nn.Parameter(torch.randn(num_units, num_keys, dim))

        # Values are stored in an EmbeddingBag for highly optimized weighted sum retrieval
        # This avoids explicit gather + multiply + sum operations
        self.value_bag = nn.EmbeddingBag(num_units * num_keys, dim, mode='sum')

    def forward(self, x):
        # x: (Batch, Num_Mem_Units, Slots, Dim)

        # 1. Query the internal memory bank
        # scores: (Batch, Num_Mem_Units, Slots, K_keys)
        scores = einsum("b e s d, e k d -> b e s k", x, self.mem_keys)

        # 2. Retrieve top-k indices and weights
        topk_weights, topk_indices = torch.topk(scores, k=self.topk, dim=-1)
        topk_weights = torch.softmax(topk_weights, dim=-1)

        # Offset indices to align with the flattened EmbeddingBag
        # offset: (Num_Mem_Units,) -> (1, Num_Mem_Units, 1, 1)
        offset = torch.arange(x.shape[1], device=x.device) * self.num_keys
        topk_indices = topk_indices + offset.view(1, -1, 1, 1)

        # 3. Efficient Aggregation using EmbeddingBag
        # Flatten for bag lookup: (Batch * Num_Units * Slots, TopK)
        flat_indices = topk_indices.view(-1, self.topk)
        flat_weights = topk_weights.view(-1, self.topk)

        # The EmbeddingBag handles the weighted sum efficiently
        out = self.value_bag(flat_indices, per_sample_weights=flat_weights)

        # Reshape back to original 4D tensor
        return out.view(x.shape)
```

Algorithm A2: Implementation of the processing sub-modules. The Expert Block performs dynamic FFN computation, while the Memory Block performs efficient key-value retrieval.