KAN-Former

The First KAN-powered Transformer

```python
class KANFormer(nn.Module):
    def __init__(self, vocabulary_size, hidden_size, num_heads, window_size,
                 d_ff, num_experts, n_experts_per_token, n_blocks, max_seq_len):
        super().__init__()

        self.embedding = nn.Embedding(vocabulary_size, hidden_size)
        head_dim = hidden_size // num_heads
        rotation_matrix = get_rotation_matrix(head_dim, max_seq_len, 10000.0)

        self.blocks = nn.ModuleList([
            KANBlock(hidden_size, num_heads, window_size, d_ff, num_experts,
                n_experts_per_token,rotation_matrix)
            for _ in range(n_blocks)
        ])

        self.out = nn.Linear(hidden_size, vocabulary_size)

    def forward(self, x):
        x = self.embedding(x)
        for block in self.blocks:
            x = block(x)
        output = self.out(x)
        return output
```

TheAiEdge.io

Decoder Block - RMSNorm

```python
class KANBlock(nn.Module):
    def __init__(self, hidden_size, num_heads, window_size, d_ff, num_experts,
                 n_experts_per_token, rotation_matrix) -> None:
        super().__init__()

        self.norm1 = RMSNorm(hidden_size)
        self.norm2 = RMSNorm(hidden_size)
        self.attention = MultiheadKANAttention(hidden_size, num_heads, window_size,
            rotation_matrix)
        self.moe = MoeKANLayer(hidden_size, d_ff, num_experts, n_experts_per_token)

    def forward(self, x):
        x1 = self.attention(self.norm1(x))
        x = x + x1
        x2 = self.moe(self.norm2(x))
        out = x + x2
        return out
```

```python
class RMSNorm(nn.Module):
    def __init__(self, hidden_size: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(hidden_size))

    def _norm(self, x):
        return x * torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)

    def forward(self, x):
        output = self._norm(x.float()).type_as(x)
        return output * self.weight
```

Multihead-KAN-attention Layer

```python
from efficient_kan import KANLinear

class MultiheadKANAttention(nn.Module):
    def __init__(self, hidden_size, num_heads, rotation_matrix):
        super().__init__()

        self.hidden_size = hidden_size
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads

        self.qkv_linear = KANLinear(hidden_size, hidden_size * 3)
        self.out = nn.Linear(hidden_size, hidden_size)
        self.position_emb = RoPE(rotation_matrix)

    def forward(self, x):
        batch_size, seq_length, hidden_size = x.size()

        qkv = self.qkv_linear(x)
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3 * self.head_dim)
        qkv = qkv.transpose(1, 2)
        queries, keys, values = qkv.chunk(3, dim=-1)
        queries, keys = self.position_emb(queries, keys)

        scores = torch.matmul(queries, keys.transpose(2, 3))
        scores = scores / (self.head_dim ** 0.5)
        attention = F.softmax(scores, dim=-1)
        context = torch.matmul(attention, values)
        context = context.transpose(1, 2)
        context = context.reshape(batch_size, seq_length, hidden_size)
        output = self.out(context)
        return output
```

Mixture Of KAN Experts

```python
from efficient_kan import KANLinear

class FeedForward(nn.Module):
    def __init__(self, hidden_size, d_ff):
        super().__init__()
        self.w1 = KANLinear(hidden_size, d_ff)
        self.w2 = KANLinear(hidden_size, d_ff)
        self.w3 = KANLinear(d_ff, hidden_size)

    def forward(self, x) -> torch.Tensor:
        return self.w3(self.w1(x) * self.w2(x))

class MoeKANLayer(nn.Module):
    def __init__(self, hidden_size, d_ff, num_experts, n_experts_per_token):
        super().__init__()
        self.num_experts = num_experts
        self.n_experts_per_token = n_experts_per_token
        self.experts = nn.ModuleList([
            FeedForward(hidden_size, d_ff) for _ in range(num_experts)
        ])
        self.gate = nn.Linear(hidden_size, num_experts, bias=False)

    def forward(self, x):
        gate_logits = self.gate(x)
        weights, selected_experts = torch.topk(gate_logits, self.n_experts_per_token)
        weights = F.softmax(weights, dim=-1, dtype=torch.float).to(x.dtype)
        out = torch.zeros_like(x)
        for i, expert in enumerate(self.experts):
            batch_idx, token_idx, topk_idx = torch.where(selected_experts == i)
            weight = weights[batch_idx, token_idx, topk_idx, None]
            out[batch_idx, token_idx] += weight * expert(
                x[batch_idx, token_idx]
            )
        return out
```

RoPE

```python
def get_rotation_matrix(dim: int, context_size: int, period: float) -> torch.Tensor:
    freqs = 1.0 / (period ** (torch.arange(0, dim, 2) / dim))
    token_indexes = torch.arange(context_size)
    thetas = torch.outer(token_indexes, freqs).float()
    return torch.polar(torch.ones_like(thetas), thetas)

class RoPE(nn.Module):
    def __init__(self, rotation_matrix):
        super().__init__()
        self.rotation_matrix = rotation_matrix

    def forward(self, queries, keys):
        batch_size, num_heads, seq_length, head_dim = queries.size()

        queries = queries.reshape(batch_size, num_heads, seq_length, head_dim // 2, 2)
        keys = keys.reshape(batch_size, num_heads, seq_length, head_dim // 2, 2)

        queries_complex = torch.view_as_complex(queries)
        keys_complex = torch.view_as_complex(keys)

        rotation_matrix = self.rotation_matrix[:seq_length]

        queries_rotated = queries_complex * rotation_matrix
        keys_rotated = keys_complex * rotation_matrix

        new_queries = torch.view_as_real(queries_rotated)
        new_keys = torch.view_as_real(keys_rotated)

        new_queries = new_queries.reshape(batch_size, num_heads, seq_length, head_dim)
        new_keys = new_keys.reshape(batch_size, num_heads, seq_length, head_dim)

        return new_queries, new_keys
```