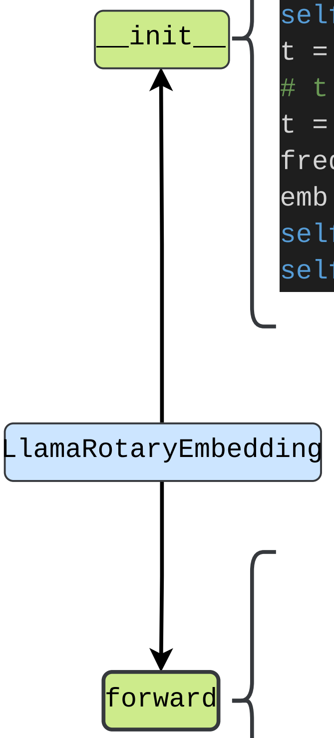


```python
self.scaling_factor = scaling_factor
self.dim = dim
self.max_position_embeddings = max_position_embeddings
self.base = base
inv_freq = 1.0 / (self.base ** (torch.arange(0, self.dim, 2, dtype=torch.int64).float().to(device) / self.dim))
# inv_freq = 1/theta: torch.Size([dim/2])
self.register_buffer("inv_freq", inv_freq, persistent=False)
self.max_seq_len_cached = max_position_embeddings
t = torch.arange(self.max_seq_len_cached, device=device, dtype=torch.int64).type_as(self.inv_freq)
# t: torch.Size([max_position_embeddings])
t = t / self.scaling_factor
freqs = torch.outer(t, self.inv_freq)  # torch.Size([max_position_embeddings, dim/2])
emb = torch.cat((freqs, freqs), dim=-1) # torch.Size([max_position_embeddings, dim])
self.register_buffer("_cos_cached", emb.cos().to(torch.get_default_dtype()), persistent=False)
self.register_buffer("_sin_cached", emb.sin().to(torch.get_default_dtype()), persistent=False)
```

$$\Theta = \{\theta_i = 10000^{-2i/d}, i \in [0, 1, 2, \dots d/2 - 1]\}$$

```python
@torch.no_grad()
def forward(self, x, position_ids):
    # x: [bs, num_attention_heads, seq_len, head_size]
    # position_ids:[1,seq_len]
    inv_freq_expanded = self.inv_freq[None, :, None].float().expand(position_ids.shape[0], -1, 1) # inv_freq_expanded: [1, dim/2, 1]
    position_ids_expanded = position_ids[:, None, :].float() # position_ids_expanded: [1,1,seq_len]
    device_type = x.device.type
    device_type = device_type if isinstance(device_type, str) and device_type != "mps" else "cpu"
    with torch.autocast(device_type=device_type, enabled=False):
        # get m theta
        freqs = (inv_freq_expanded.float() @ position_ids_expanded.float()).transpose(1, 2)
        emb = torch.cat((freqs, freqs), dim=-1) # torch.Size([1, seq_len, dim])
        cos = emb.cos()
        sin = emb.sin()
    return cos.to(dtype=x.dtype), sin.to(dtype=x.dtype)
```

```python
def rotate_half(x):
    """Rotates half the hidden dims of the input."""
    x1 = x[..., : x.shape[-1] // 2]
    x2 = x[..., x.shape[-1] // 2 :]
    return torch.cat((-x2, x1), dim=-1)
```

```python
def apply_rotary_pos_emb(q, k, cos, sin, position_ids=None, unsqueeze_dim=1):
    """Applies Rotary Position Embedding to the query and key tensors."""
    cos = cos.unsqueeze(unsqueeze_dim)
    sin = sin.unsqueeze(unsqueeze_dim)
    q_embed = (q * cos) + (rotate_half(q) * sin)
    k_embed = (k * cos) + (rotate_half(k) * sin)
    return q_embed, k_embed
```

```python
for b in range(q_embed.shape[0]):
    for h in range(q_embed.shape[1]):
        q_slice = q[b][h]
        cos_slice = cos[-1][-1]
        sin_slice = sin[-1][-1]
        len = q_slice.shape[0]
        dim = q_slice.shape[1]
        res_slice = torch.zeros((len,dim))
        for i in range(len):
            for j in range(dim):
                if j < dim / 2:
                    res_slice[i][j] =
                        cos_slice[i][j]*q_slice[i][j]
                        - sin_slice[i][j]*q_slice[i][int(j+dim / 2)]
                else:
                    res_slice[i][j] =
                        cos_slice[i][int(j-dim / 2)]*q_slice[i][j]
                        + sin_slice[i][int(j-dim / 2)]*q_slice[i][int(j-dim / 2)]
        print(res_slice == q_embed[b][h])
```