In [13]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math
import numpy as np

# Check device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
print(f"PyTorch version: {torch.__version__}")
```

```
Using device: cpu
PyTorch version: 2.11.0+cpu
```

In [14]:
```
class PositionalEncoding(nn.Module):
    """
    Injects positional information into token embeddings.
    Uses sine and cosine functions of different frequencies.
    PE(pos, 2i)   = sin(pos / 10000^(2i/d_model))
    PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))
    """
    def __init__(self, d_model, max_seq_len=5000, dropout=0.1):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Create positional encoding matrix
        pe = torch.zeros(max_seq_len, d_model)                          # (max_seq_len, d_model)
        position = torch.arange(0, max_seq_len).unsqueeze(1).float()    # (max_seq_len, 1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model)
        )

        pe[:, 0::2] = torch.sin(position * div_term)   # even indices
        pe[:, 1::2] = torch.cos(position * div_term)   # odd indices

        pe = pe.unsqueeze(0)  # (1, max_seq_len, d_model) — batch dimension
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x: (batch_size, seq_len, d_model)
        x = x + self.pe[:, :x.size(1), :]
        return self.dropout(x)


# Quick test
pe = PositionalEncoding(d_model=512)
dummy = torch.zeros(2, 10, 512)
out = pe(dummy)
print(f"Positional Encoding output shape: {out.shape}")  # (2, 10, 512)
```

```
Positional Encoding output shape: torch.Size([2, 10, 512])
```

In [15]:
```
def scaled_dot_product_attention(Q, K, V, mask=None):
    """
    Computes Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) * V

    Args:
        Q: Query  — (batch, heads, seq_len, d_k)
        K: Key    — (batch, heads, seq_len, d_k)
        V: Value  — (batch, heads, seq_len, d_v)
        mask:     — optional mask to block certain positions
    Returns:
        output:   — (batch, heads, seq_len, d_v)
        attn_weights: — (batch, heads, seq_len, seq_len)
    """
    d_k = Q.size(-1)

    # Step 1: QK^T / sqrt(d_k)
    scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_k)  # (batch, heads, seq, seq)

    # Step 2: Apply mask (fill -inf so softmax gives ~0)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, float('-1e9'))
```

```
    # Step 3: Softmax over last dimension
    attn_weights = F.softmax(scores, dim=-1)

    # Step 4: Weighted sum of V
    output = torch.matmul(attn_weights, V)

    return output, attn_weights


# Quick test
Q = torch.rand(2, 8, 10, 64)
K = torch.rand(2, 8, 10, 64)
V = torch.rand(2, 8, 10, 64)
out, attn = scaled_dot_product_attention(Q, K, V)
print(f"Attention output shape: {out.shape}")       # (2, 8, 10, 64)
print(f"Attention weights shape: {attn.shape}")     # (2, 8, 10, 10)
```

```
Attention output shape: torch.Size([2, 8, 10, 64])
Attention weights shape: torch.Size([2, 8, 10, 10])
```

In [16]:

```
class MultiHeadAttention(nn.Module):
    """
    Multi-Head Attention splits d_model into h heads,
    runs attention in parallel, then concatenates & projects.
    """
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model    = d_model
        self.num_heads  = num_heads
        self.d_k        = d_model // num_heads  # dimension per head

        # Linear projections for Q, K, V, and output
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def split_heads(self, x):
        """Reshape (batch, seq_len, d_model) → (batch, heads, seq_len, d_k)"""
        batch_size, seq_len, _ = x.size()
        x = x.view(batch_size, seq_len, self.num_heads, self.d_k)
        return x.transpose(1, 2)  # (batch, heads, seq_len, d_k)

    def combine_heads(self, x):
        """Reverse of split_heads: (batch, heads, seq_len, d_k) → (batch, seq_len, d_model)"""
        batch_size, _, seq_len, _ = x.size()
        x = x.transpose(1, 2).contiguous()
        return x.view(batch_size, seq_len, self.d_model)

    def forward(self, Q, K, V, mask=None):
        # Project inputs
        Q = self.split_heads(self.W_q(Q))  # (batch, heads, seq, d_k)
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        # Scaled dot-product attention
        attn_output, attn_weights = scaled_dot_product_attention(Q, K, V, mask)

        # Combine heads and project
        output = self.W_o(self.combine_heads(attn_output))  # (batch, seq, d_model)
        return output, attn_weights


# Quick test
mha = MultiHeadAttention(d_model=512, num_heads=8)
x = torch.rand(2, 10, 512)
out, w = mha(x, x, x)
print(f"MHA output shape: {out.shape}")   # (2, 10, 512)
```

```
MHA output shape: torch.Size([2, 10, 512])
```

In [17]:
```
class FeedForwardNetwork(nn.Module):
    """
    Two linear transformations with a ReLU in between:
    FFN(x) = max(0, xW1 + b1)W2 + b2
    d_ff is typically 4x d_model (e.g. 2048 for d_model=512)
    """
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(FeedForwardNetwork, self).__init__()
        self.linear1  = nn.Linear(d_model, d_ff)
        self.linear2  = nn.Linear(d_ff, d_model)
        self.dropout  = nn.Dropout(dropout)

    def forward(self, x):
        return self.linear2(self.dropout(F.relu(self.linear1(x))))


# Quick test
ffn = FeedForwardNetwork(d_model=512, d_ff=2048)
x = torch.rand(2, 10, 512)
print(f"FFN output shape: {ffn(x).shape}")  # (2, 10, 512)
```

```
FFN output shape: torch.Size([2, 10, 512])
```

In [18]:
```
class EncoderLayer(nn.Module):
    """
    One Encoder block:
        1. Multi-Head Self-Attention  → Add & Norm
        2. Feed-Forward Network       → Add & Norm
    """
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super(EncoderLayer, self).__init__()
        self.self_attn  = MultiHeadAttention(d_model, num_heads)
        self.ffn        = FeedForwardNetwork(d_model, d_ff, dropout)
        self.norm1      = nn.LayerNorm(d_model)
        self.norm2      = nn.LayerNorm(d_model)
        self.dropout    = nn.Dropout(dropout)

    def forward(self, x, src_mask=None):
        # Sub-Layer 1: Self-Attention + Residual
        attn_out, _ = self.self_attn(x, x, x, src_mask)
        x = self.norm1(x + self.dropout(attn_out))

        # Sub-Layer 2: FFN + Residual
        ffn_out = self.ffn(x)
        x = self.norm2(x + self.dropout(ffn_out))

        return x


# Quick test
enc_layer = EncoderLayer(d_model=512, num_heads=8, d_ff=2048)
x = torch.rand(2, 10, 512)
print(f"Encoder Layer output shape: {enc_layer(x).shape}")  # (2, 10, 512)
```

```
Encoder Layer output shape: torch.Size([2, 10, 512])
```

In [19]:
```
class DecoderLayer(nn.Module):
    """
    One Decoder block:
        1. Masked Multi-Head Self-Attention → Add & Norm
        2. Cross-Attention (over encoder output) → Add & Norm
        3. Feed-Forward Network              → Add & Norm
    """
    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super(DecoderLayer, self).__init__()
        self.self_attn   = MultiHeadAttention(d_model, num_heads)
        self.cross_attn  = MultiHeadAttention(d_model, num_heads)
        self.ffn         = FeedForwardNetwork(d_model, d_ff, dropout)
        self.norm1       = nn.LayerNorm(d_model)
        self.norm2       = nn.LayerNorm(d_model)
        self.norm3       = nn.LayerNorm(d_model)
        self.dropout     = nn.Dropout(dropout)

    def forward(self, x, enc_output, src_mask=None, tgt_mask=None):
```

```
        # Sub-Layer 1: Masked Self-Attention + Residual
        attn_out, _ = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(attn_out))

        # Sub-Layer 2: Cross-Attention (Q from decoder, K/V from encoder) + Residual
        cross_out, _ = self.cross_attn(x, enc_output, enc_output, src_mask)
        x = self.norm2(x + self.dropout(cross_out))

        # Sub-Layer 3: FFN + Residual
        ffn_out = self.ffn(x)
        x = self.norm3(x + self.dropout(ffn_out))

        return x


# Quick test
dec_layer = DecoderLayer(d_model=512, num_heads=8, d_ff=2048)
tgt = torch.rand(2, 7, 512)
enc_out = torch.rand(2, 10, 512)
print(f"Decoder Layer output shape: {dec_layer(tgt, enc_out).shape}")  # (2, 7, 512)
```

```
Decoder Layer output shape: torch.Size([2, 7, 512])
```

In [20]:

```
class Transformer(nn.Module):
    """
    Full Transformer (Encoder-Decoder) as described in
    'Attention Is All You Need' (Vaswani et al., 2017).
    """
    def __init__(self, src_vocab_size, tgt_vocab_size,
                 d_model=512, num_heads=8, num_layers=6,
                 d_ff=2048, max_seq_len=5000, dropout=0.1):
        super(Transformer, self).__init__()

        # Embeddings
        self.src_embedding = nn.Embedding(src_vocab_size, d_model)
        self.tgt_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.pos_encoding  = PositionalEncoding(d_model, max_seq_len, dropout)

        # Encoder stack
        self.encoder_layers = nn.ModuleList([
            EncoderLayer(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)
        ])

        # Decoder stack
        self.decoder_layers = nn.ModuleList([
            DecoderLayer(d_model, num_heads, d_ff, dropout)
            for _ in range(num_layers)
        ])

        # Final linear projection to vocabulary
        self.output_layer = nn.Linear(d_model, tgt_vocab_size)
        self.d_model = d_model

    def make_src_mask(self, src, pad_idx=0):
        """Mask padding tokens in source"""
        # src: (batch, src_len)
        return (src != pad_idx).unsqueeze(1).unsqueeze(2)  # (batch, 1, 1, src_len)

    def make_tgt_mask(self, tgt, pad_idx=0):
        """Mask both padding AND future tokens in target (causal mask)"""
        batch_size, tgt_len = tgt.size()
        pad_mask  = (tgt != pad_idx).unsqueeze(1).unsqueeze(2)           # (batch, 1, 1, tgt_len)
        look_ahead = torch.tril(torch.ones(tgt_len, tgt_len)).bool()     # (tgt_len, tgt_len)
        look_ahead = look_ahead.unsqueeze(0).unsqueeze(0).to(tgt.device) # (1, 1, tgt_len, tgt_len)
        return pad_mask & look_ahead                                       # combined

    def encode(self, src, src_mask):
        x = self.pos_encoding(self.src_embedding(src) * math.sqrt(self.d_model))
        for layer in self.encoder_layers:
            x = layer(x, src_mask)
        return x

    def decode(self, tgt, enc_output, src_mask, tgt_mask):
```

```
        x = self.pos_encoding(self.tgt_embedding(tgt) * math.sqrt(self.d_model))
        for layer in self.decoder_layers:
            x = layer(x, enc_output, src_mask, tgt_mask)
        return x

    def forward(self, src, tgt, pad_idx=0):
        src_mask = self.make_src_mask(src, pad_idx)
        tgt_mask = self.make_tgt_mask(tgt, pad_idx)

        enc_output = self.encode(src, src_mask)
        dec_output = self.decode(tgt, enc_output, src_mask, tgt_mask)

        return self.output_layer(dec_output)  # (batch, tgt_len, tgt_vocab_size)
```

In [21]:

```
# Hyperparameters (matching the original "base" model)
SRC_VOCAB_SIZE = 10000
TGT_VOCAB_SIZE = 10000
D_MODEL        = 512
NUM_HEADS      = 8
NUM_LAYERS     = 6
D_FF           = 2048
MAX_SEQ_LEN    = 100
DROPOUT        = 0.1

model = Transformer(
    src_vocab_size = SRC_VOCAB_SIZE,
    tgt_vocab_size = TGT_VOCAB_SIZE,
    d_model        = D_MODEL,
    num_heads      = NUM_HEADS,
    num_layers     = NUM_LAYERS,
    d_ff           = D_FF,
    max_seq_len    = MAX_SEQ_LEN,
    dropout        = DROPOUT
).to(device)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters: {total_params:,}")
print(model)
```

```
Total trainable parameters: 59,508,496
Transformer(
  (src_embedding): Embedding(10000, 512)
  (tgt_embedding): Embedding(10000, 512)
  (pos_encoding): PositionalEncoding(
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder_layers): ModuleList(
    (0-5): 6 x EncoderLayer(
      (self_attn): MultiHeadAttention(
        (W_q): Linear(in_features=512, out_features=512, bias=True)
        (W_k): Linear(in_features=512, out_features=512, bias=True)
        (W_v): Linear(in_features=512, out_features=512, bias=True)
        (W_o): Linear(in_features=512, out_features=512, bias=True)
      )
      (ffn): FeedForwardNetwork(
        (linear1): Linear(in_features=512, out_features=2048, bias=True)
        (linear2): Linear(in_features=2048, out_features=512, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (decoder_layers): ModuleList(
    (0-5): 6 x DecoderLayer(
      (self_attn): MultiHeadAttention(
        (W_q): Linear(in_features=512, out_features=512, bias=True)
        (W_k): Linear(in_features=512, out_features=512, bias=True)
        (W_v): Linear(in_features=512, out_features=512, bias=True)
        (W_o): Linear(in_features=512, out_features=512, bias=True)
      )
      (cross_attn): MultiHeadAttention(
        (W_q): Linear(in_features=512, out_features=512, bias=True)
        (W_k): Linear(in_features=512, out_features=512, bias=True)
        (W_v): Linear(in_features=512, out_features=512, bias=True)
        (W_o): Linear(in_features=512, out_features=512, bias=True)
      )
      (ffn): FeedForwardNetwork(
        (linear1): Linear(in_features=512, out_features=2048, bias=True)
        (linear2): Linear(in_features=2048, out_features=512, bias=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
      (norm1): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (norm2): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (norm3): LayerNorm((512,), eps=1e-05, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (output_layer): Linear(in_features=512, out_features=10000, bias=True)
)
```

In [23]:

```python
# Simulate a batch of token sequences
BATCH_SIZE = 4
SRC_LEN    = 20
TGT_LEN    = 15

src = torch.randint(1, SRC_VOCAB_SIZE, (BATCH_SIZE, SRC_LEN)).to(device)  # (4, 20)
tgt = torch.randint(1, TGT_VOCAB_SIZE, (BATCH_SIZE, TGT_LEN)).to(device)  # (4, 15)

model.eval()
with torch.no_grad():
    output = model(src, tgt)

print(f"Input  src shape  : {src.shape}")      # (4, 20)
print(f"Input  tgt shape  : {tgt.shape}")      # (4, 15)
print(f"Output logits shape: {output.shape}")  # (4, 15, 10000)
print("\n Forward pass successful!")
```

```
Input  src shape  : torch.Size([4, 20])
Input  tgt shape  : torch.Size([4, 15])
Output logits shape: torch.Size([4, 15, 10000])

 Forward pass successful!
```

In [24]:

```python
import torch.optim as optim

# Use a smaller config for demo training
demo_model = Transformer(
    src_vocab_size=1000, tgt_vocab_size=1000,
    d_model=128, num_heads=4, num_layers=2,
    d_ff=256, dropout=0.1
).to(device)

optimizer  = optim.Adam(demo_model.parameters(), lr=1e-4, betas=(0.9, 0.98), eps=1e-9)
criterion  = nn.CrossEntropyLoss(ignore_index=0)  # ignore padding index

def train_step(model, src, tgt):
    model.train()
    optimizer.zero_grad()

    tgt_input  = tgt[:, :-1]   # decoder input  (all but last token)
    tgt_output = tgt[:, 1:]    # expected output (all but first token / <sos>)

    logits = model(src, tgt_input)               # (batch, tgt_len-1, vocab)
    logits = logits.reshape(-1, logits.size(-1)) # (batch*(tgt_len-1), vocab)
    tgt_output = tgt_output.reshape(-1)          # (batch*(tgt_len-1),)

    loss = criterion(logits, tgt_output)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
    optimizer.step()
    return loss.item()

# Run a few demo steps with random data
print("Running demo training steps...\n")
for step in range(5):
    src_demo = torch.randint(1, 1000, (8, 20)).to(device)
    tgt_demo = torch.randint(1, 1000, (8, 16)).to(device)
    loss_val = train_step(demo_model, src_demo, tgt_demo)
    print(f"Step {step+1}/5 — Loss: {loss_val:.4f}")

print("\n Training loop working correctly!")
```

```
Running demo training steps...

Step 1/5 — Loss: 7.0503
Step 2/5 — Loss: 7.0589
Step 3/5 — Loss: 6.9596
Step 4/5 — Loss: 7.0665
Step 5/5 — Loss: 7.1114

 Training loop working correctly!
```

In [ ]: