Search models, datasets, users...

Datasets: bird-of-paradise / **transformer-from-scratch-tutorial**

like 22

ArXiv: arxiv:1706.03762 License: mit

Dataset card Files xet Community 2

main

transformer-from-scratch-tutorial / Transformer_Implementation_Tutorial.ipynb

bird-of-paradise

`Fix encoder-decoder interaction: use final encoder output for all …`

1f4d6e1 VERIFIED

raw Copy download link history contribute delete Safe 441 kB

Open in Colab

Implementing Transformer Architecture: A Step-by-Step Guide

Paper Reference

- "Attention Is All You Need" (Vaswani et al., 2017)
- Key sections:
  - 3.1: Encoder and Decoder Stacks
  - 3.2: Attention Mechanism
  - 3.3: Position-wise Feed-Forward Networks
  - 3.4: Embeddings and Softmax
  - 3.5: Positional Encoding
  - 5.4: Regularization (dropout strategy)

Implementation Strategy

Breaking down the architecture into manageable pieces and gradually adding complexity:

1. Start with foundational components:
   - Embedding + Positional Encoding
   - Single-head self-attention
2. Build up attention mechanism:
   - Extend to multi-head attention
   - Add cross-attention capability
   - Implement attention masking
3. Construct larger components:
   - Encoder (self-attention + FFN)
   - Decoder (masked self-attention + cross-attention + FFN)
4. Combine into final architecture:
   - Encoder-Decoder stack
   - Full Transformer with input/output layers

Development Tips

1. Visualization and Planning:
   - Draw out tensor dimensions on paper
   - Sketch attention patterns and masks
   - Map each component back to paper equations
   - This helps catch dimension mismatches early!
2. Dimension Cheat Sheet:

- Input tokens: [batch_size, seq_len]
- Embeddings: [batch_size, seq_len, d_model]
- Attention matrices: [batch_size, num_heads, seq_len, seq_len]
- FFN hidden layer: [batch_size, seq_len, d_ff]
- Output logits: [batch_size, seq_len, vocab_size]

3. Common Pitfalls:

- Forgetting to scale dot products by √d_k
- Applying mask too early or too late
- Incorrect mask dimensions or application
- Missing residual connections
- Wrong order of layer norm and dropout
- Tensor dimension mismatches in attention
- Not handling padding properly

4. Performance Considerations:

- Memory usage scales with sequence length squared
- Attention computation is $O(n^2)$ with sequence length
- Balance between d_model and num_heads
- Trade-off between model size and batch size

Testing Strategy

- Test each component independently
- Verify shape preservation
- Check attention patterns
- Confirm mask effectiveness
- Validate gradient flow
- Monitor numerical stability

Remember: The key to successfully implementing the Transformer is understanding how each piece fits together and maintaining clear dimension tracking throughout the implementation.

In [ ]:

```

```

Code Section

Embedding and Positional Encoding

This implements the input embedding from Section 3.4 and positional encoding from Section 3.5 of the paper. Key points:

- Embedding dimension can differ from model dimension (using projection)
- Positional encoding uses sine and cosine functions
- Scale embeddings by √d_model

- Apply dropout to the sum of embeddings and positional encodings

Implementation tips:

- Use `nn.Embedding` for token embeddings
- Store scaling factor as float during initialization
- Remember to expand positional encoding for batch dimension
- Add assertion for input dtype (should be torch.long)

In [1]:
```
import math
import torch
import torch.nn as nn

class EmbeddingWithProjection(nn.Module):
    def __init__(self, vocab_size, d_embed, d_model,
                 max_position_embeddings =512, dropout=0.1):
        super().__init__()
        self.d_model = d_model
        self.d_embed = d_embed
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(self.vocab_size, self.d
        self.projection = nn.Linear(self.d_embed, self.d_mode
        self.scaling = float(math.sqrt(self.d_model))

        self.layernorm = nn.LayerNorm(self.d_model)
        self.dropout = nn.Dropout(p=dropout)

    @staticmethod
    def create_positional_encoding(seq_length, d_model, batch
        # Create position indices: [seq_length, 1]
        position = torch.arange(seq_length).unsqueeze(1).floa

        # Create dimension indices: [1, d_model//2]
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-math.log(10000.0) / d_model)
        )

        # Create empty tensor: [seq_length, d_model]
        pe = torch.zeros(seq_length, d_model)

        # Compute sin and cos
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        # Add batch dimension and expand: [batch_size, seq_le
        pe = pe.unsqueeze(0).expand(batch_size, -1, -1)

        return pe



    def forward(self, x):
        assert x.dtype == torch.long, f"Input tensor must hav
        batch_size, seq_length = x.size() # [batch, seq_lengt

        # token embedding
        token_embedding = self.embedding(x)
        # project the scaled token embedding to the d_model s
        token_embedding =  self.projection(token_embedding) *
```

```
        # add positional encodings to projected,
        # scaled embeddings before applying layer norm and dr
        positional_encoding = self.create_positional_encoding

        # In addition, we apply dropout to the sums of the em
        # in both the encoder and decoder stacks. For the bas
        normalized_sum = self.layernorm(token_embedding + pos
        final_output = self.dropout(normalized_sum)
        return final_output
```

Transformer Attention

Implements the core attention mechanism from Section 3.2.1. Formula: Attention(Q,K,V) = softmax(QK^T/√d_k)V

Key points:

- Supports both self-attention and cross-attention
- Multi-head attention implementation per section 3.2.2
- Handles different sequence lengths for encoder/decoder
- Scales dot products by 1/√d_k
- Applies attention masking before softmax
- Applies dropout after softmax

Implementation tips:

- Use separate Q,K,V projections
- Handle masking through addition (not masked_fill)
- Remember to use braodcasting and reshape for multi-head attention
- Keep track of tensor dimensions at each step

In [3]:

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

import math

class TransformerAttention(nn.Module):
    """
    Transformer Scaled Dot Product Attention Module
    Args:
        d_model: Total dimension of the model.
        num_head: Number of attention heads.
        dropout: Dropout rate for attention scores.
        bias: Whether to include bias in linear projections.

    Inputs:
        sequence: input sequence for self-attention and the q
        key_value_state: input for the key, values for cross-
    """
    def __init__(self, d_model, num_head, dropout=0.1, bias=T
        super().__init__()  # Missing in the original impleme
        assert d_model % num_head == 0, "d_model must be divi
        self.d_model = d_model
        self.num_head = num_head
        self.d_head=d_model//num_head
```

```
            self.d_head=d_model//num_head
            self.dropout_rate = dropout  # Store dropout rate sep

            # linear transformations
            self.q_proj = nn.Linear(d_model, d_model, bias=bias)
            self.k_proj = nn.Linear(d_model, d_model, bias=bias)
            self.v_proj = nn.Linear(d_model, d_model, bias=bias)
            self.output_proj = nn.Linear(d_model, d_model, bias=b

            # Dropout layer
            self.dropout = nn.Dropout(p=dropout)

            # Initiialize scaler
            self.scaler = float(1.0 / math.sqrt(self.d_head)) # S


        def forward(self, sequence, key_value_states = None, att_
            """Input shape: [batch_size, seq_len, d_model=num_hea
            batch_size, seq_len, model_dim = sequence.size()

            # Check only critical input dimensions
            assert model_dim == self.d_model, f"Input dimension {
            if key_value_states is not None:
                assert key_value_states.size(-1) == self.d_model,
                f"Cross attention key/value dimension {key_value_


            # if key_value_states are provided this layer is used
            # for the decoder
            is_cross_attention = key_value_states is not None

            # Linear projections and reshape for multi-head
            Q_state = self.q_proj(sequence)
            if is_cross_attention:
                kv_seq_len = key_value_states.size(1)
                K_state = self.k_proj(key_value_states)
                V_state = self.v_proj(key_value_states)
            else:
                kv_seq_len = seq_len
                K_state = self.k_proj(sequence)
                V_state = self.v_proj(sequence)

            #[batch_size, self.num_head, seq_len, self.d_head]
            Q_state = Q_state.view(batch_size, seq_len, self.num_

            # in cross-attention, key/value sequence length might
            K_state = K_state.view(batch_size, kv_seq_len, self.n
            V_state = V_state.view(batch_size, kv_seq_len, self.n

            # Scale Q by 1/sqrt(d_k)
            Q_state = Q_state * self.scaler


            # Compute attention matrix: QK^T
            self.att_matrix = torch.matmul(Q_state, K_state.trans


            # apply attention mask to attention matrix
            if att_mask is not None and not isinstance(att_mask,
                raise TypeError("att_mask must be a torch.Tensor"

            if att_mask is not None:
```

```
            self.att_matrix = self.att_matrix + att_mask

        # apply softmax to the last dimension to get the atte
        att_score = F.softmax(self.att_matrix, dim = -1)

        # apply drop out to attention score
        att_score = self.dropout(att_score)

        # get final output: softmax(QK^T)V
        att_output = torch.matmul(att_score, V_state)

        # concatinate all attention heads
        att_output = att_output.transpose(1, 2)
        att_output = att_output.contiguous().view(batch_size,

        # final linear transformation to the concatenated out
        att_output = self.output_proj(att_output)

        assert att_output.size() == (batch_size, seq_len, sel
        f"Final output shape {att_output.size()} incorrect"

        return att_output
```

Feed-Forward Network (FFN)

Implements the position-wise feed-forward network from Section 3.3:
$FFN(x) = max(0, xW_1 + b_1)W_2 + b_2$

Key points:

- Two linear transformations with ReLU in between
- Inner layer dimension (d_ff) is typically 2048
- Applied identically to each position

Implementation tips:

- Use nn.Linear for transformations
- Remember to include bias terms
- Position-wise means same transformation for each position
- Dimension flow: d_model → d_ff → d_model

In [5]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F


class FFN(nn.Module):
    """
    Position-wise Feed-Forward Networks
    This consists of two linear transformations with a ReLU a

    FFN(x) = max(0, xW1 + b1 )W2 + b2
    d_model: embedding dimension (e.g., 512)
    d_ff: feed-forward dimension (e.g., 2048)

    """
    def __init__(self, d_model, d_ff):
        super().__init__()
```

```
        self.d_model=d_model
        self.d_ff= d_ff

        # Linear transformation y = xW+b
        self.fc1 = nn.Linear(self.d_model, self.d_ff, bias =
        self.fc2 = nn.Linear(self.d_ff, self.d_model, bias =

        # for potential speed up
        # Pre-normalize the weights (can help with training s
        nn.init.xavier_uniform_(self.fc1.weight)
        nn.init.xavier_uniform_(self.fc2.weight)


    def forward(self, input):
        # check input and first FF layer dimension matching
        batch_size, seq_length, d_input = input.size()
        assert self.d_model == d_input, "d_model must be the

        # First linear transformation followed by ReLU
        # There's no need for explicit torch.max() as F.relu(
        f1 = F.relu(self.fc1(input))

        # max(0, xW_1 + b_1)W_2 + b_2
        f2 =  self.fc2(f1)

        return f2
```

In [7]:

```
net = FFN(  d_model = 512,  d_ff =2048)
print(net)
```

```
FFN(
  (fc1): Linear(in_features=512, out_features=2048, bias=Tru
e)
  (fc2): Linear(in_features=2048, out_features=512, bias=Tru
e)
)
```

Transformer Encoder

Implements single encoder layer from Section 3.1, consisting of:

- Multi-head self-attention
- Position-wise feed-forward network
- Residual connections and layer normalization

Implementation tips:

- Apply dropout before adding residual
- Keep model dimension consistent through the layer

In [7]:

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

class TransformerEncoder(nn.Module):
    """
    Encoder layer of the Transformer
```

```
    Sublayers: TransformerAttention
               Residual LayerNorm
               FNN
               Residual LayerNorm
    Args:
            d_model: 512 model hidden dimension
            d_embed: 512 embedding dimension, same as d_model
            d_ff: 2048 hidden dimension of the feed forward n
            num_head: 8 Number of attention heads.
            dropout:  0.1 dropout rate

            bias: Whether to include bias in linear projectio

    """

    def __init__(
        self, d_model, d_ff,
        num_head, dropout=0.1,
        bias=True
    ):
        super().__init__()
        self.d_model = d_model
        self.d_ff = d_ff


        # attention sublayer
        self.att = TransformerAttention(
            d_model = d_model,
            num_head = num_head,
            dropout = dropout,
            bias = bias
        )

        # FFN sublayer
        self.ffn = FFN(
            d_model = d_model,
            d_ff = d_ff
        )

        # Dropout layer
        self.dropout = nn.Dropout(p=dropout)

        # layer-normalization layer
        self.LayerNorm_att = nn.LayerNorm(self.d_model)
        self.LayerNorm_ffn = nn.LayerNorm(self.d_model)


    def forward(self, embed_input, padding_mask=None):

        batch_size, seq_len, _ = embed_input.size()

        ## First sublayer: self attion
        att_sublayer = self.att(sequence = embed_input, key_v
                                att_mask = padding_mask)  # [

        # apply dropout before layer normalization for each s
        att_sublayer = self.dropout(att_sublayer)
        # Residual layer normalization
        att_normalized = self.LayerNorm_att(embed_input + att_

        ## Second sublayer: FFN
        ffn_sublayer = self.ffn(att_normalized)
```

```
        ffn_sublayer = self.dropout(ffn_sublayer)
        ffn_normalized = self.LayerNorm_ffn(att_normalized +


        return ffn_normalized
```

In [9]:
```
net = TransformerEncoder( d_model = 512, d_ff =2048, num_head
print(net)
```

```
TransformerEncoder(
  (att): TransformerAttention(
    (q_proj): Linear(in_features=512, out_features=512, bias=
True)
    (k_proj): Linear(in_features=512, out_features=512, bias=
True)
    (v_proj): Linear(in_features=512, out_features=512, bias=
True)
    (output_proj): Linear(in_features=512, out_features=512,
bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (ffn): FFN(
    (fc1): Linear(in_features=512, out_features=2048, bias=Tr
ue)
    (fc2): Linear(in_features=2048, out_features=512, bias=Tr
ue)
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (LayerNorm_att): LayerNorm((512,), eps=1e-05, elementwise_a
ffine=True)
  (LayerNorm_ffn): LayerNorm((512,), eps=1e-05, elementwise_a
ffine=True)
)
```

Transformer Decoder

Implements decoder layer from Section 3.1, with three sub-layers:

- Masked multi-head self-attention
- Multi-head cross-attention with encoder output
- Position-wise feed-forward network

Key points:

- Self-attention uses causal masking
- Cross-attention allows attending to all encoder outputs
- Each sub-layer followed by residual connection and layer normalization
- Apply dropout to the output of previous sub-layer before residual connection and layer normalization

Implementation tips:

- Order of operations matters (masking before softmax)
- Each attention layer has its own projections
- Remember to pass encoder outputs for cross-attention
- Careful with mask dimensions in self and cross attention

- Key implementation detail for causal masking:
- Create causal mask using upper triangular matrix:
  `mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1)`
  `mask = mask.masked_fill(mask == 1, float('-inf'))`

In [9]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class TransformerDecoder(nn.Module):
    """
    Decoder layer of the Transformer
    Sublayers: TransformerAttention with self-attention
               Residual LayerNorm
               TransformerAttention with cross-attention
               Residual LayerNorm
               FNN
               Residual LayerNorm
    Args:
            d_model: 512 model hidden dimension
            d_embed: 512 embedding dimension, same as d_model
            d_ff: 2048 hidden dimension of the feed forward n
            num_head: 8 Number of attention heads.
            dropout:  0.1 dropout rate

            bias: Whether to include bias in linear projectio

    """

    def __init__(
        self, d_model, d_ff,
        num_head, dropout=0.1,
        bias=True
    ):
        super().__init__()
        self.d_model = d_model
        self.d_ff = d_ff


        # attention sublayer
        self.att = TransformerAttention(
            d_model = d_model,
            num_head = num_head,
            dropout = dropout,
            bias = bias
        )

        # FFN sublayer
        self.ffn = FFN(
            d_model = d_model,
            d_ff = d_ff
        )


        # Dropout layer
        self.dropout = nn.Dropout(p=dropout)

        # layer-normalization layer
        self.LayerNorm_att1 = nn.LayerNorm(self.d_model)
```

```
        self.LayerNorm_att2 = nn.LayerNorm(self.d_model)
        self.LayerNorm_ffn = nn.LayerNorm(self.d_model)

    @staticmethod
    def create_causal_mask(seq_len):
        mask = torch.triu(torch.ones(seq_len, seq_len), diago
        mask = mask.masked_fill(mask == 1, float('-inf'))
        return mask


    def forward(self, embed_input, cross_input, padding_mask=
        """
        Args:
        embed_input: Decoder input sequence [batch_size, seq_
        cross_input: Encoder output sequence [batch_size, enc
        casual_attention_mask: Causal mask for self-attention
        padding_mask: Padding mask for cross-attention [batch
        Returns:
        Tensor: Decoded output [batch_size, seq_len, d_model]
        """
        batch_size, seq_len, _ = embed_input.size()

        assert embed_input.size(-1) == self.d_model, f"Input
        assert cross_input.size(-1) == self.d_model, "Encoder


        # Generate and expand causal mask for self-attention
        causal_mask = self.create_causal_mask(seq_len).to(emb
        causal_mask = causal_mask.unsqueeze(0).unsqueeze(1)


        ## First sublayer: self attion
        # After embedding and positional encoding, input sequ
        # Or, the output of the previous encoder/decoder feed
        att_sublayer1 = self.att(sequence = embed_input, key_
                                 att_mask = causal_mask)  # [b
        # apply dropout before layer normalization for each s
        att_sublayer1 = self.dropout(att_sublayer1)
        # Residual layer normalization
        att_normalized1 = self.LayerNorm_att1(embed_input + a

        ## Second sublayer: cross attention
        # Query from the output of previous attention output,
        # Key, Value from output of Encoder of the same layer
        att_sublayer2 = self.att(sequence = att_normalized1,
                                 att_mask = padding_mask)  # [
        # apply dropout before layer normalization for each s
        att_sublayer2 = self.dropout(att_sublayer2)
        # Residual layer normalization
        att_normalized2 = self.LayerNorm_att2(att_normalized1


        # Third sublayer: FFN
        ffn_sublayer = self.ffn(att_normalized2)
        ffn_sublayer = self.dropout(ffn_sublayer)
        ffn_normalized = self.LayerNorm_ffn(att_normalized2 +


        return ffn_normalized
```

In [13]:
```
net = TransformerDecoder( d_model = 512, d_ff =2048, num_head
print(net)
```

```
TransformerDecoder(
  (att): TransformerAttention(
    (q_proj): Linear(in_features=512, out_features=512, bias=
True)
    (k_proj): Linear(in_features=512, out_features=512, bias=
True)
    (v_proj): Linear(in_features=512, out_features=512, bias=
True)
    (output_proj): Linear(in_features=512, out_features=512,
bias=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (ffn): FFN(
    (fc1): Linear(in_features=512, out_features=2048, bias=Tr
ue)
    (fc2): Linear(in_features=2048, out_features=512, bias=Tr
ue)
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (LayerNorm_att1): LayerNorm((512,), eps=1e-05, elementwise_
affine=True)
  (LayerNorm_att2): LayerNorm((512,), eps=1e-05, elementwise_
affine=True)
  (LayerNorm_ffn): LayerNorm((512,), eps=1e-05, elementwise_a
ffine=True)
)
```

Encoder-Decoder Stack

Implements the full stack of encoder and decoder layers from Section 3.1.

Key points:

- Multiple encoder and decoder layers (typically 6)
- Each encoder output feeds into all decoder layers
- Maintains residual connections throughout the stack

Implementation tips:

- Use nn.ModuleList for layer stacks
- Share encoder outputs across decoder layers
- Maintain consistent masking throughout
- Handle padding masks separately from causal masks

In [15]:

```
class TransformerEncoderDecoder(nn.Module):
    """
    Encoder-Decoder stack of the Transformer
    Sublayers:  Encoder x 6
                Decoder x 6
    Args:
            d_model: 512 model hidden dimension
            d_embed: 512 embedding dimension, same as d_model
            d_ff: 2048 hidden dimension of the feed forward n
            num_head: 8 Number of attention heads.
            dropout:  0.1 dropout rate

            bias: Whether to include bias in linear projectio
```

```
    """
    def __init__(
        self, num_layer,
        d_model, d_ff,
        num_head, dropout=0.1,
        bias=True
    ):
        super().__init__()
        self.num_layer = num_layer
        self.d_model = d_model
        self.d_ff = d_ff
        self.num_head = num_head
        self.dropout = dropout
        self.bias = bias

        # Encoder stack
        self.encoder_stack = nn.ModuleList([ TransformerEncod
                                                d_model = self.d_mode
                                                d_ff = self.d_ff,
                                                num_head = self.num_h
                                                dropout = self.dropou
                                                bias = self.bias) for

        # Decoder stack
        self.decoder_stack = nn.ModuleList([ TransformerDecod
                                                d_model = self.d_mode
                                                d_ff = self.d_ff,
                                                num_head = self.num_h
                                                dropout = self.dropou
                                                bias = self.bias) for


    def forward(self, embed_encoder_input, embed_decoder_inpu
        # Process through all encoder layers first
        encoder_output = embed_encoder_input
        for encoder in self.encoder_stack:
            encoder_output = encoder(encoder_output, padding_

        # Use final encoder output for all decoder layers
        decoder_output = embed_decoder_input
        for decoder in self.decoder_stack:
            decoder_output = decoder(decoder_output, encoder_

        return decoder_output
```

Full Transformer

Combines all components into complete architecture:

- Input embeddings for source and target
- Positional encoding
- Encoder-decoder stack
- Final linear and softmax layer

Key points:

- Handles different vocabulary sizes for source/target
- Shifts decoder inputs for teacher forcing

- Projects outputs to target vocabulary size
- Applies log softmax for training stability

Implementation tips:

- Handle start tokens for decoder input
- Maintain separate embeddings for source/target
- Remember to scale embeddings
- Consider sharing embedding weights with output layer

In [13]:
```
class Transformer(nn.Module):
    def __init__(
        self,
        num_layer,
        d_model, d_embed, d_ff,
        num_head,
        src_vocab_size,
        tgt_vocab_size,
        max_position_embeddings=512,
        dropout=0.1,
        bias=True
    ):
        super().__init__()

        self.tgt_vocab_size = tgt_vocab_size

        # Source and target embeddings
        self.src_embedding = EmbeddingWithProjection(
            vocab_size=src_vocab_size,
            d_embed=d_embed,
            d_model=d_model,
            max_position_embeddings=max_position_embeddings,
            dropout=dropout
        )

        self.tgt_embedding = EmbeddingWithProjection(
            vocab_size=tgt_vocab_size,
            d_embed=d_embed,
            d_model=d_model,
            max_position_embeddings=max_position_embeddings,
            dropout=dropout
        )

        # Encoder-Decoder stack
        self.encoder_decoder = TransformerEncoderDecoder(
            num_layer=num_layer,
            d_model=d_model,
            d_ff=d_ff,
            num_head=num_head,
            dropout=dropout,
            bias=bias
        )

        # Output projection and softmax
        self.output_projection = nn.Linear(d_model, tgt_vocab
        self.softmax = nn.LogSoftmax(dim=-1)

    def shift_target_right(self, tgt_tokens):
        # Shift target tokens right by padding with zeros at
        batch_size, seq_len = tgt_tokens.size()
```

```python
        # Create start token (zeros)
        start_tokens = torch.zeros(batch_size, 1, dtype=tgt_t

        # Concatenate start token and remove last token
        shifted_tokens = torch.cat([start_tokens, tgt_tokens[

        return shifted_tokens

    def forward(self, src_tokens, tgt_tokens, padding_mask=No
        """
        Args:
            src_tokens: source sequence [batch_size, src_len]
            tgt_tokens: target sequence [batch_size, tgt_len]
            padding_mask: padding mask [batch_size, 1, 1, seq
        Returns:
            output: [batch_size, tgt_len, tgt_vocab_size] log
        """
        # Shift target tokens right for teacher forcing
        shifted_tgt_tokens = self.shift_target_right(tgt_toke

        # Embed source and target sequences
        src_embedding = self.src_embedding(src_tokens)
        tgt_embedding = self.tgt_embedding(shifted_tgt_tokens

        # Pass through encoder-decoder stack
        decoder_output = self.encoder_decoder(
            embed_encoder_input=src_embedding,
            embed_decoder_input=tgt_embedding,
            padding_mask=padding_mask
        )

        # Project to vocabulary size and apply log softmax
        logits = self.output_projection(decoder_output)
        log_probs = self.softmax(logits)

        return log_probs
```

In [ ]:

```python

```

Testing Section

In [15]:

```python
## testing on the embedding implemntation
## Tokenlize model input: from batched sentences to batched s
from transformers import AutoTokenizer
from transformers import pipeline

import torch

# layer config
d_model = 768
d_embed = 1024  # Larger embedding dimension
vocab_size=30522

# loading sample data
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english
tokenizer = AutoTokenizer.from_pretrained(checkpoint, use_fas
sequences = ["I've been waiting for a HuggingFace course my w
```

```
# will truncate the sequences that are longer than the model
# (512 for BERT or DistilBERT)
max_position_embeddings = 512
model_inputs = tokenizer(sequences, truncation=True,  padding

# Check vocabulary size from the tokenizer
# Happen to be the same as the default setting for distilbert
vocab_size = tokenizer.vocab_size
print(f"Tokenizer vocabulary size: {vocab_size}")


input = torch.tensor(model_inputs['input_ids'])
embedder = EmbeddingWithProjection(vocab_size=vocab_size, d_e
output = embedder(input)

print(f"Input shape: {input.shape}")
print(f"Embedded shape after projection: {output.shape}")
```

```
/opt/anaconda3/lib/python3.11/site-packages/torchvision/io/im
age.py:13: UserWarning: Failed to load image Python extensio
n: 'dlopen(/opt/anaconda3/lib/python3.11/site-packages/torchv
ision/image.so, 0x0006): Symbol not found: __ZN3c1017Register
OperatorsD1Ev
  Referenced from: <1868C013-6C01-31FA-98D3-E369F1FD0275> /op
t/anaconda3/lib/python3.11/site-packages/torchvision/image.so
  Expected in:     <44DEDA27-4DE9-3D4A-8EDE-5AA72081319F> /op
t/anaconda3/lib/python3.11/site-packages/torch/lib/libtorch_c
pu.dylib'If you don't plan on using image functionality from
`torchvision.io`, you can ignore this warning. Otherwise, the
re might be something wrong with your environment. Did you ha
ve `libjpeg` or `libpng` installed before building `torchvisi
on` from source?
  warn(
/opt/anaconda3/lib/python3.11/site-packages/torchvision/datap
oints/__init__.py:12: UserWarning: The torchvision.datapoints
and torchvision.transforms.v2 namespaces are still Beta. Whil
e we do not expect major breaking changes, some APIs may stil
l change according to user feedback. Please submit any feedba
ck you may have in this issue: https://github.com/pytorch/vis
ion/issues/6753, and you can also check out https://github.co
m/pytorch/vision/issues/7319 to learn more about the APIs tha
t we suspect might involve future changes. You can silence th
is warning by calling torchvision.disable_beta_transforms_war
ning().
  warnings.warn(_BETA_TRANSFORMS_WARNING)
/opt/anaconda3/lib/python3.11/site-packages/torchvision/trans
forms/v2/__init__.py:54: UserWarning: The torchvision.datapoi
nts and torchvision.transforms.v2 namespaces are still Beta.
While we do not expect major breaking changes, some APIs may
still change according to user feedback. Please submit any fe
edback you may have in this issue: https://github.com/pytorc
h/vision/issues/6753, and you can also check out https://gith
ub.com/pytorch/vision/issues/7319 to learn more about the API
s that we suspect might involve future changes. You can silen
ce this warning by calling torchvision.disable_beta_transform
s_warning().
  warnings.warn(_BETA_TRANSFORMS_WARNING)
Tokenizer vocabulary size: 30522
Input shape: torch.Size([2, 16])
Embedded shape after projection: torch.Size([2, 16, 768])
```

In [17]:

```
def test_transformer_encoder():
```

```
# Set random seed for reproducibility
torch.manual_seed(42)

# Test parameters
batch_size = 32
seq_length = 20
d_model = 512
d_ff = 2048
num_heads = 8

# Initialize the transformer encoder
encoder = TransformerEncoder(
    d_model=d_model,
    d_ff=d_ff,
    num_head=num_heads,
    dropout=0.1
)

# Set to evaluation mode to disable dropout
encoder.eval()

# Create input sequence - using ones instead of random va
# for easier interpretation of attention patterns
input_sequence = torch.ones(batch_size, seq_length, d_mod
cross_sequence = torch.ones(batch_size, seq_length, d_mod

# Create attention mask
attention_mask = torch.ones(batch_size, seq_length)
attention_mask[:, 15:] = 0  # Mask last 5 positions
attention_mask =attention_mask.unsqueeze(1).unsqueeze(3)

# Store attention patterns
attention_patterns = []

# Define hook to capture attention scores
def attention_hook(module, input, output):
    # We want to capture the attention scores before they
    # This assumes your attention module returns the atte
    attention_patterns.append(output)

# Register the hook on the attention computation
encoder.att.register_forward_hook(attention_hook)

# Perform forward pass
with torch.no_grad():
    output = encoder(input_sequence, attention_mask)

# Basic shape tests
expected_shape = (batch_size, seq_length, d_model)
assert output.shape == expected_shape, f"Expected shape {

# Print output statistics
print("\nOutput Statistics:")
print(f"Mean: {output.mean():.4f}")
print(f"Std: {output.std():.4f}")
print(f"Min: {output.min():.4f}")
print(f"Max: {output.max():.4f}")

# Analyze attention patterns
if attention_patterns:
    attention_output = attention_patterns[0]
    # Look at the attention patterns for unmasked vs mask
```

```
        unmasked_attention = output[:, :15, :].abs().mean()
        masked_attention = output[:, 15:, :].abs().mean()

        print("\nAttention Analysis:")
        print(f"Unmasked positions mean: {unmasked_attention:
        print(f"Masked positions mean: {masked_attention:.4f}

        # Note: We expect masked positions to still have valu
        # but their patterns should be different from unmaske
        print("\nIs the masking working?", "Yes" if unmasked_

    # Check for any NaN or infinite values
    assert torch.isfinite(output).all(), "Output contains NaN

    print("\nAll tests passed successfully!")
    return output, attention_patterns

# Run the test
output, attention_patterns = test_transformer_encoder()
```

```
Output Statistics:
Mean: -0.0000
Std: 1.0000
Min: -2.7968
Max: 2.8519

Attention Analysis:
Unmasked positions mean: 0.8078
Masked positions mean: 0.8078

Is the masking working? No

All tests passed successfully!
```

In [19]:

```
def test_transformer_decoder():
    torch.manual_seed(42)

    # Test parameters
    batch_size = 32
    seq_length = 20
    encoder_seq_length = 22
    d_model = 512
    d_ff = 2048
    num_heads = 8

    decoder = TransformerDecoder(
        d_model=d_model,
        d_ff=d_ff,
        num_head=num_heads,
        dropout=0.1
    )
    decoder.eval()

    # Create input sequences
    decoder_input = torch.randn(batch_size, seq_length, d_mod
    encoder_output = torch.randn(batch_size, encoder_seq_leng

    # Create padding mask for encoder outputs
    padding_mask = torch.ones(batch_size, seq_length, encoder
    padding_mask[:, :, 18:] = 0  # Mask last 4 positions of e
    padding_mask = padding_mask.unsqueeze(1)  # Add head dime
```

```
    # Store attention scores
    attention_scores = []

    # Define hook to capture attention scores before softmax
    def attention_hook(module, input, output):
        if not attention_scores:  # Only store first layer's
            # Assuming attention scores are computed before t
            attention_scores.append(module.att_matrix.detach(

    # Register hook on the attention layer
    decoder.att.register_forward_hook(attention_hook)

    # Perform forward pass
    with torch.no_grad():
        output = decoder(decoder_input, encoder_output, paddi

    # Basic shape tests
    expected_shape = (batch_size, seq_length, d_model)
    assert output.shape == expected_shape, f"Expected shape {

    # Print output statistics
    print("\nOutput Statistics:")
    print(f"Mean: {output.mean():.4f}")
    print(f"Std: {output.std():.4f}")
    print(f"Min: {output.min():.4f}")
    print(f"Max: {output.max():.4f}")

    # Test shape preservation
    print("\nShape Analysis:")
    print(f"Input shape: {decoder_input.shape}")
    print(f"Output shape: {output.shape}")
    print(f"Expected shape matches: {'Yes' if decoder_input.s

    # Check for any NaN or infinite values
    assert torch.isfinite(output).all(), "Output contains NaN

    print("\nAll tests passed successfully!")
    return output, attention_scores

# Run the test
output, attention_scores = test_transformer_decoder()
```

```
Output Statistics:
Mean: -0.0000
Std: 1.0000
Min: -4.3617
Max: 4.5787

Shape Analysis:
Input shape: torch.Size([32, 20, 512])
Output shape: torch.Size([32, 20, 512])
Expected shape matches: Yes

All tests passed successfully!
```

In [21]:
```
def test_transformer_encoder_decoder_stack():
    torch.manual_seed(42)

    # Test parameters
    batch_size = 8
    seq_length = 10
    d_model = 512
```

```
    d_ff = 2048
    num_heads = 8
    num_layers = 6

    # Initialize the transformer encoder-decoder stack
    transformer = TransformerEncoderDecoder(
        num_layer=num_layers,
        d_model=d_model,
        d_ff=d_ff,
        num_head=num_heads,
        dropout=0.1
    )

    # Set to evaluation mode to disable dropout
    transformer.eval()

    # Create input sequences
    encoder_input = torch.randn(batch_size, seq_length, d_mod
    decoder_input = torch.randn(batch_size, seq_length, d_mod

    # Create padding mask
    padding_mask = torch.ones(batch_size, seq_length)
    padding_mask[:, -2:] = 0  # Mask last 2 positions
    padding_mask = padding_mask.unsqueeze(1).unsqueeze(2)  #

    # Store intermediate outputs
    intermediate_outputs = []

    def hook_fn(module, input, output):
        intermediate_outputs.append(output.detach())

    # Register hooks to capture outputs from each encoder and
    for i, (encoder, decoder) in enumerate(zip(transformer.en
        encoder.register_forward_hook(lambda m, i, o, layer=i
        decoder.register_forward_hook(lambda m, i, o, layer=i

    # Perform forward pass
    with torch.no_grad():
        output = transformer(encoder_input, decoder_input, pa

    # Basic shape tests
    expected_shape = (batch_size, seq_length, d_model)
    assert output.shape == expected_shape, f"Expected shape {

    # Print output statistics
    print("\nFinal Output Statistics:")
    print(f"Mean: {output.mean():.4f}")
    print(f"Std: {output.std():.4f}")
    print(f"Min: {output.min():.4f}")
    print(f"Max: {output.max():.4f}")

    # Verify shape preservation through layers
    print("\nShape Preservation Check:")
    print(f"Input shapes - Encoder: {encoder_input.shape}, De
    print(f"Output shape: {output.shape}")

    # Check for any NaN or infinite values
    assert torch.isfinite(output).all(), "Output contains NaN

    # Verify that output is different from input (transformat
    input_output_diff = (output - decoder_input).abs().mean()
    print(f"\nMean absolute difference between input and outp
```

```
    print(f"\nMean absolute difference between input and outp
    print("Transformation occurred:", "Yes" if input_output_d

    # Check if model parameters were used
    total_params = sum(p.numel() for p in transformer.paramet
    print(f"\nTotal number of parameters: {total_params:,}")

    print("\nAll tests passed successfully!")
    return output

# Run the test
output = test_transformer_encoder_decoder_stack()
```

```
Encoder Layer 0 shape: torch.Size([8, 10, 512])

Encoder Layer 1 shape: torch.Size([8, 10, 512])

Encoder Layer 2 shape: torch.Size([8, 10, 512])

Encoder Layer 3 shape: torch.Size([8, 10, 512])

Encoder Layer 4 shape: torch.Size([8, 10, 512])

Encoder Layer 5 shape: torch.Size([8, 10, 512])
Decoder Layer 0 shape: torch.Size([8, 10, 512])
Decoder Layer 1 shape: torch.Size([8, 10, 512])
Decoder Layer 2 shape: torch.Size([8, 10, 512])
Decoder Layer 3 shape: torch.Size([8, 10, 512])
Decoder Layer 4 shape: torch.Size([8, 10, 512])
Decoder Layer 5 shape: torch.Size([8, 10, 512])

Final Output Statistics:
Mean: 0.0000
Std: 1.0000
Min: -3.7172
Max: 4.1310

Shape Preservation Check:
Input shapes - Encoder: torch.Size([8, 10, 512]), Decoder: to
rch.Size([8, 10, 512])
Output shape: torch.Size([8, 10, 512])

Mean absolute difference between input and output: 0.9379
Transformation occurred: Yes

Total number of parameters: 37,834,752

All tests passed successfully!
```

In [23]:
```
def test_complete_transformer():
    # Configuration
    d_model = 768
    d_embed = 1024
    d_ff = 2048
    num_heads = 8
    num_layers = 6
    max_position_embeddings = 512

    # Load tokenizer
    tokenizer = AutoTokenizer.from_pretrained("distilbert-bas
                                              use_fast=True,
                                              use_multiprocessi
    vocab_size = tokenizer.vocab_size
```

```
    vocab_size = tokenizer.vocab_size

    # Create sample source and target sequences
    src_sequences = [
        "I've been waiting for a HuggingFace course my whole
        "So have I!"
    ]
    # Pretend these are translations
    tgt_sequences = [
        "J'ai attendu un cours HuggingFace toute ma vie.",
        "Moi aussi!"
    ]

    # Tokenize source and target sequences
    src_inputs = tokenizer(src_sequences, truncation=True, pa
    tgt_inputs = tokenizer(tgt_sequences, truncation=True, pa

    # Create transformer model
    transformer = Transformer(
        num_layer=num_layers,
        d_model=d_model,
        d_embed=d_embed,
        d_ff=d_ff,
        num_head=num_heads,
        src_vocab_size=vocab_size,
        tgt_vocab_size=vocab_size,
        max_position_embeddings=max_position_embeddings
    )

    # Set to eval mode
    transformer.eval()

    # Create padding mask from attention mask
    padding_mask = src_inputs['attention_mask'].unsqueeze(1).

    print("\nInput Shapes:")
    print(f"Source tokens: {src_inputs['input_ids'].shape}")
    print(f"Target tokens: {tgt_inputs['input_ids'].shape}")

    # Forward pass
    with torch.no_grad():
        output = transformer(
            src_tokens=src_inputs['input_ids'],
            tgt_tokens=tgt_inputs['input_ids'],
            padding_mask=padding_mask
        )

    print("\nOutput Analysis:")
    print(f"Output shape: {output.shape}")  # Should be [batc

    # Verify output is proper probability distribution
    print("\nProbability Distribution Check:")
    print(f"Sum to 1: {torch.allclose(output.exp().sum(dim=-1
    print(f"Max probability: {output.exp().max().item():.4f}"
    print(f"Min probability: {output.exp().min().item():.4f}"

    # Check if we can get predictions
    predictions = output.argmax(dim=-1)
    print("\nSample Predictions:")
    print("Original target:")
    print(tgt_sequences[0])
    print("\nModel output (decoded):")
```

```
    print(tokenizer.decode(predictions[0]))

    # Test backward pass
    transformer.train()
    output = transformer(
        src_tokens=src_inputs['input_ids'],
        tgt_tokens=tgt_inputs['input_ids'],
        padding_mask=padding_mask
    )

    # Calculate loss (cross entropy)
    loss = F.nll_loss(
        output.view(-1, vocab_size),
        tgt_inputs['input_ids'].view(-1)
    )

    # Test backward pass
    loss.backward()

    # Verify gradients
    has_gradients = all(p.grad is not None for p in transform
    print("\nTraining Check:")
    print(f"Loss value: {loss.item():.4f}")
    print(f"Has gradients: {has_gradients}")

    return output, predictions

# Run test
output, predictions = test_complete_transformer()
```

```
Input Shapes:
Source tokens: torch.Size([2, 16])
Target tokens: torch.Size([2, 17])

Output Analysis:
Output shape: torch.Size([2, 17, 30522])

Probability Distribution Check:
Sum to 1: True
Max probability: 0.0005
Min probability: 0.0000

Sample Predictions:
Original target:
J'ai attendu un cours HuggingFace toute ma vie.

Model output (decoded):
##aco bearer barriedate gate spoil lowlands tam navigation gr
owls 1971 painfully demand negativelyzam [unused158] lowlands

Training Check:
Loss value: 10.7329
Has gradients: True
```

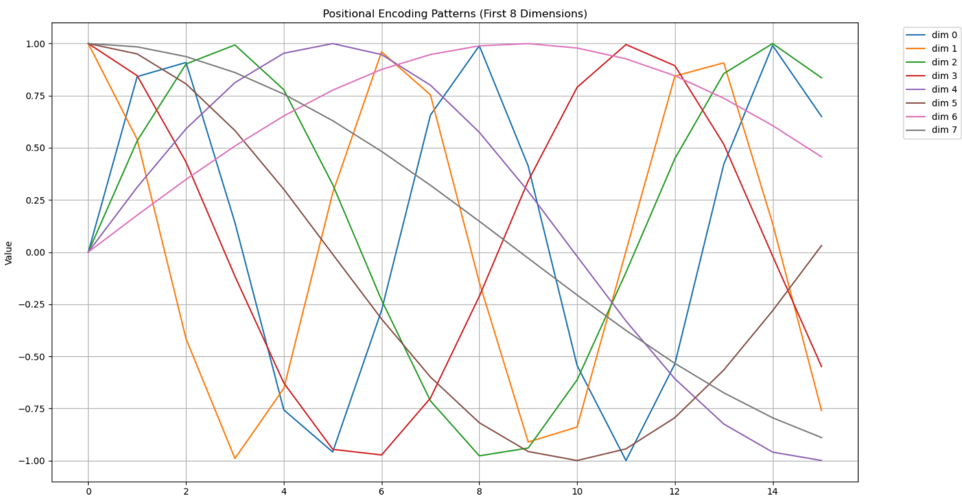
In [ ]:

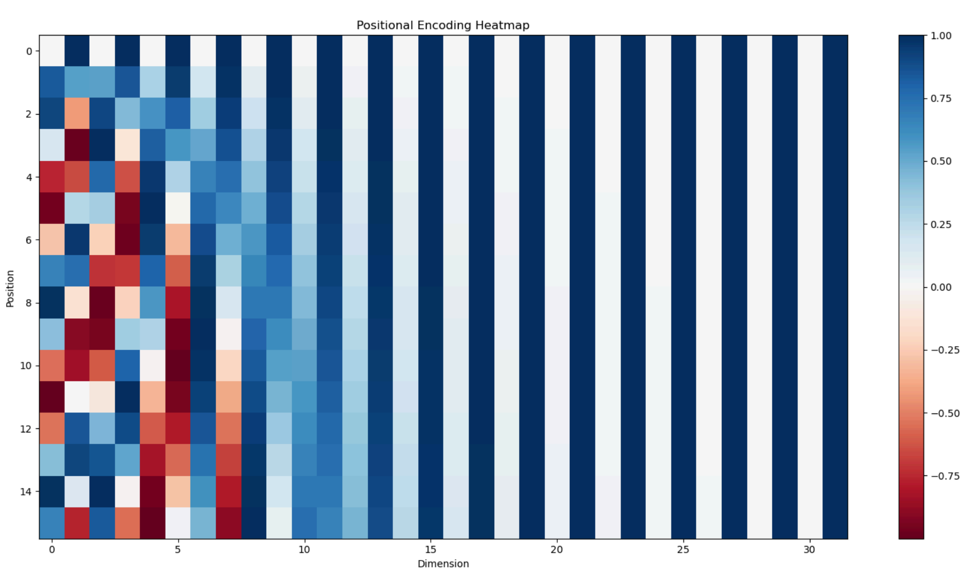
```
```

Visualization Section

Positional Encoding Visualization

In [25]:
```
## Visualize positional encoding
import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"
import matplotlib.pyplot as plt
import numpy as np

def visualize_positional_encoding(seq_length=30, d_model=32):
    # Generate positional encoding
    pe = np.zeros((seq_length, d_model))
    position = np.arange(seq_length)[:, np.newaxis]
    div_term = np.exp(np.arange(0, d_model, 2) * -(np.log(100

    pe[:, 0::2] = np.sin(position * div_term)
    pe[:, 1::2] = np.cos(position * div_term)

    # Create visualization
    plt.figure(figsize=(15, 8))

    # Plot first 8 dimensions
    for dim in range(8):
        plt.plot(pe[:, dim], label=f'dim {dim}')

    plt.xlabel('Position')
    plt.ylabel('Value')
    plt.title('Positional Encoding Patterns (First 8 Dimensio
    plt.legend(bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.grid(True)
    plt.tight_layout()
    plt.show()

    # Also show heatmap of all dimensions
    plt.figure(figsize=(15, 8))
    plt.imshow(pe, cmap='RdBu', aspect='auto')
    plt.colorbar()
    plt.xlabel('Dimension')
    plt.ylabel('Position')
    plt.title('Positional Encoding Heatmap')
    plt.tight_layout()
    plt.show()

# Using your model's positional encoding
seq_length = 16  # From your example
d_model = 768    # From your example

# You might want to use a smaller d_model for visualization
visualize_positional_encoding(seq_length=16, d_model=32)
```





Attention Pattern Visualization

Shows:

- Self-attention patterns (causal masking)
- Cross-attention patterns
- Effect of padding masks
- How attention weights distribute

Implementation insights:

- Use softmax before visualization
- Show masking effects

In [27]:
```
def test_decoder_causal_masking():
    torch.manual_seed(42)

    # Test parameters
    batch_size = 2
    seq_length = 5
    d_model = 512
    d_ff = 2048
    num_heads = 8

    decoder = TransformerDecoder(
        d_model=d_model,
        d_ff=d_ff,
        num_head=num_heads,
        dropout=0.1
    )
    decoder.eval()

    decoder_input = torch.randn(batch_size, seq_length, d_mod
    encoder_output = torch.randn(batch_size, seq_length, d_mo

    attention_scores = []

    def attention_hook(module, input, output):
        if not attention_scores:
            # Apply softmax to get actual attention probabili
            scores = F.softmax(module.att_matrix, dim=-1)
```

```
            scores = F.softmax(module.att_matrix, dim=-1)
            attention_scores.append(scores.detach())

    decoder.att.register_forward_hook(attention_hook)

    with torch.no_grad():
        output = decoder(decoder_input, encoder_output)

    att_weights = attention_scores[0]

    print("\nAttention Matrix Shape:", att_weights.shape)

    # Print attention pattern for first head of first batch
    print("\nAttention Pattern (first head):")
    print(att_weights[0, 0].round(decimals=4))

    # Check future tokens (should be 0)
    future_attention = att_weights[:, :, torch.triu_indices(s
                                    torch.triu_indices(se

    print("\nFuture Token Analysis:")
    print(f"Mean attention to future tokens: {future_attentio
    print(f"Max attention to future tokens: {future_attention
    print("Causal masking working:", "Yes" if future_attentio

    # Check present/past tokens
    present_past = att_weights[:, :, torch.tril_indices(seq_l
                                torch.tril_indices(seq_le

    print("\nPresent/Past Token Analysis:")
    print(f"Mean attention to present/past tokens: {present_p
    print(f"Has non-zero attention patterns:", "Yes" if prese

    # Verify each position's attention sums to 1
    attention_sums = att_weights.sum(dim=-1)
    print("\nAttention Sum Analysis:")
    print(f"Mean attention sum (should be 1): {attention_sums
    print(f"Max deviation from 1: {(attention_sums - 1).abs()

    return att_weights

attention_weights = test_decoder_causal_masking()
```

```
Attention Matrix Shape: torch.Size([2, 8, 5, 5])

Attention Pattern (first head):
tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000],
        [0.4465, 0.5535, 0.0000, 0.0000, 0.0000],
        [0.3403, 0.3496, 0.3101, 0.0000, 0.0000],
        [0.1965, 0.3485, 0.1174, 0.3377, 0.0000],
        [0.1563, 0.1571, 0.1859, 0.1948, 0.3059]])

Future Token Analysis:
Mean attention to future tokens: 0.00000000
Max attention to future tokens: 0.00000000
Causal masking working: Yes

Present/Past Token Analysis:
Mean attention to present/past tokens: 0.3333
Has non-zero attention patterns: Yes

Attention Sum Analysis:
Mean attention sum (should be 1): 1.0000
```

```
Mean attention sum (should be 1): 1.0000
Max deviation from 1: 0.00000024
```

In [29]:

```
def test_decoder_cross_attention():
    torch.manual_seed(42)

    # Test parameters
    batch_size = 2
    decoder_seq_len = 5
    encoder_seq_len = 7  # Different length to make it intere
    d_model = 512
    d_ff = 2048
    num_heads = 8

    decoder = TransformerDecoder(
        d_model=d_model,
        d_ff=d_ff,
        num_head=num_heads,
        dropout=0.1
    )
    decoder.eval()

    # Create input sequences
    decoder_input = torch.randn(batch_size, decoder_seq_len,
    encoder_output = torch.randn(batch_size, encoder_seq_len,

    # Store attention scores
    cross_attention_scores = []

    def attention_hook(module, input, output):
        # We want the second call to att (cross-attention)
        if len(cross_attention_scores) < 2:
            scores = F.softmax(module.att_matrix, dim=-1)
            cross_attention_scores.append(scores.detach())

    decoder.att.register_forward_hook(attention_hook)

    # Forward pass
    with torch.no_grad():
        output = decoder(decoder_input, encoder_output)

    # Get cross-attention weights (second element in list)
    cross_att_weights = cross_attention_scores[1]  # [batch,

    print("\nCross-Attention Matrix Shape:", cross_att_weight

    # Print attention pattern for first head of first batch
    print("\nCross-Attention Pattern (first head):")
    print(cross_att_weights[0, 0].round(decimals=4))

    # Verify each decoder position attends to all encoder pos
    attention_sums = cross_att_weights.sum(dim=-1)
    zero_attention = (cross_att_weights == 0).all(dim=-1)

    print("\nCross-Attention Analysis:")
    print(f"Mean attention weight: {cross_att_weights.mean():
    print(f"Min attention weight: {cross_att_weights.min():.4
    print(f"Max attention weight: {cross_att_weights.max():.4

    print("\nAttention Coverage:")
    print(f"Each position's attention sums to 1: {torch.allcl
    print(f"Every decoder position attends to some encoder po
```

```
    # Check attention distribution
    attention_entropy = -(cross_att_weights * torch.log(cross_
    print(f"\nAttention entropy (higher means more uniform at

    return cross_att_weights

# Run the test
cross_attention_weights = test_decoder_cross_attention()
```

```
Cross-Attention Matrix Shape: torch.Size([2, 8, 5, 7])

Cross-Attention Pattern (first head):
tensor([[0.1308, 0.1502, 0.1380, 0.1131, 0.1987, 0.1117, 0.15
76],
        [0.1303, 0.1041, 0.1502, 0.1756, 0.1679, 0.1589, 0.11
30],
        [0.0896, 0.2159, 0.1142, 0.1718, 0.1797, 0.0844, 0.14
44],
        [0.1250, 0.1650, 0.1607, 0.1053, 0.0868, 0.2349, 0.12
23],
        [0.1637, 0.0842, 0.2093, 0.1223, 0.1274, 0.1392, 0.15
40]])

Cross-Attention Analysis:
Mean attention weight: 0.1429
Min attention weight: 0.0389
Max attention weight: 0.4142

Attention Coverage:
Each position's attention sums to 1: True
Every decoder position attends to some encoder position: True

Attention entropy (higher means more uniform attention): 1.89
17
```

In [31]:

```
def test_decoder_cross_attention_with_padding():
    torch.manual_seed(42)

    # Test parameters
    batch_size = 2
    decoder_seq_len = 5
    encoder_seq_len = 7
    d_model = 512
    d_ff = 2048
    num_heads = 8

    decoder = TransformerDecoder(
        d_model=d_model,
        d_ff=d_ff,
        num_head=num_heads,
        dropout=0.1
    )
    decoder.eval()

    # Create input sequences
    decoder_input = torch.randn(batch_size, decoder_seq_len,
    encoder_output = torch.randn(batch_size, encoder_seq_len,

    # Create padding mask for encoder outputs
    # Mask out last 2 positions (as if they were padding in e
    padding_mask = torch.ones(batch_size, decoder_seq_len, en
```

```
    padding_mask[:, :, -2:] = float('-inf')  # Mask positions
    padding_mask = padding_mask.unsqueeze(1)  # Add head dime

    cross_attention_scores = []

    def attention_hook(module, input, output):
        if len(cross_attention_scores) < 2:
            scores = F.softmax(module.att_matrix, dim=-1)
            cross_attention_scores.append(scores.detach())

    decoder.att.register_forward_hook(attention_hook)

    # Forward pass
    with torch.no_grad():
        output = decoder(decoder_input, encoder_output, paddi

    # Get cross-attention weights (second element)
    cross_att_weights = cross_attention_scores[1]

    print("\nCross-Attention Matrix Shape:", cross_att_weight

    print("\nCross-Attention Pattern (first head):")
    print("(Last two encoder positions should have zero atten
    print(cross_att_weights[0, 0].round(decimals=4))

    # Analyze masked positions (last two columns)
    masked_attention = cross_att_weights[:, :, :, -2:]
    unmasked_attention = cross_att_weights[:, :, :, :-2]

    print("\nMasking Analysis:")
    print(f"Mean attention to masked positions: {masked_atten
    print(f"Max attention to masked positions: {masked_attent
    print(f"Mean attention to unmasked positions: {unmasked_a

    # Verify attention still sums to 1 (only over unmasked po
    attention_sums = cross_att_weights.sum(dim=-1)

    print("\nAttention Coverage:")
    print(f"Each position's attention sums to 1: {torch.allcl

    # Analyze attention distribution over unmasked positions
    print("\nUnmasked Position Analysis:")
    print(f"Min attention to unmasked positions: {unmasked_at
    print(f"Max attention to unmasked positions: {unmasked_at

    return cross_att_weights

# Run the test
cross_attention_weights = test_decoder_cross_attention_with_p
```

```
Cross-Attention Matrix Shape: torch.Size([2, 8, 5, 7])

Cross-Attention Pattern (first head):
(Last two encoder positions should have zero attention)
tensor([[0.1791, 0.2055, 0.1888, 0.1547, 0.2719, 0.0000, 0.00
00],
```