Imports and Setup

In [5]:
```python
import torch
import torch.nn as nn
import torch.optim as optim
import math
import random

# Set random seed for reproducibility
SEED = 1234
random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
Using device: cuda
```

Positional Encoding Module

In [6]:
```python
class PositionalEncoding(nn.Module):
    """
    Injects some information about the relative or absolute position of the tokens
    in the sequence. The positional encodings have the same dimension as the embeddings.
    """
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Create constant 'pe' matrix with values dependent on pos and i
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(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)
        pe[:, 1::2] = torch.cos(position * div_term)

        # Register as buffer (not a learnable parameter, but part of state_dict)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        # x: [Batch, Seq_len, Dim]
        # Add positional encoding to the input embedding
        x = x + self.pe[:, :x.size(1), :]
        return self.dropout(x)
```

Seq2Seq Transformer Model Definition

In [7]:
```python
class Seq2SeqTransformer(nn.Module):
    def __init__(self, num_tokens, dim_model, num_heads, num_encoder_layers, num_decoder_layers,
        super().__init__()
```

```
        # Embedding Layer and Positional Encoding
        self.embedding = nn.Embedding(num_tokens, dim_model)
        self.pos_encoder = PositionalEncoding(dim_model, dropout_p)

        # Core: nn.Transformer module (contains both Encoder and Decoder)
        # batch_first=True ensures input format is [Batch, Seq, Dim]
        self.transformer = nn.Transformer(
            d_model=dim_model,
            nhead=num_heads,
            num_encoder_layers=num_encoder_layers,
            num_decoder_layers=num_decoder_layers,
            dropout=dropout_p,
            batch_first=True
        )

        # Final output projection layer
        self.out = nn.Linear(dim_model, num_tokens)
        self.dim_model = dim_model

    def forward(self, src, tgt, src_mask=None, tgt_mask=None, src_padding_mask=None, tgt_padding
        """
        src: Source sequence
        tgt: Target sequence (shifted right)
        src_mask: Mask for source (usually None or all zeros)
        tgt_mask: Mask for target (causal mask to hide future tokens)
        src/tgt_padding_mask: Bool mask where True indicates padding tokens
        """
        # 1. Apply embedding and positional encoding
        src = self.pos_encoder(self.embedding(src) * math.sqrt(self.dim_model))
        tgt = self.pos_encoder(self.embedding(tgt) * math.sqrt(self.dim_model))

        # 2. Pass through the Transformer
        # Note: In PyTorch's nn.Transformer, padding masks are named *_key_padding_mask
        output = self.transformer(
            src, tgt,
            src_mask=src_mask,
            tgt_mask=tgt_mask,
            src_key_padding_mask=src_padding_mask,
            tgt_key_padding_mask=tgt_padding_mask,
            memory_key_padding_mask=src_padding_mask # Mask encoder padding for the decoder
        )

        # 3. Project to vocabulary size
        return self.out(output)
```

Masking Utilities

In [25]:

```
def generate_square_subsequent_mask(sz, device):
    """
    Generates a causal mask (look-ahead mask) for the decoder.
    It prevents positions from attending to subsequent positions.
    """
    mask = (torch.triu(torch.ones((sz, sz), device=device)) == 1).transpose(0, 1)
    mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, float(0.0))
    return mask

def create_mask(src, tgt, device):
    """
    Creates both the causal mask for the target and padding masks for both source and target.
```

```
    """
    src_seq_len = src.shape[1]
    tgt_seq_len = tgt.shape[1]

    # Target requires a causal mask
    tgt_mask = generate_square_subsequent_mask(tgt_seq_len, device)

    # Source does not require a causal mask (encoder sees all tokens)
    src_mask = torch.zeros((src_seq_len, src_seq_len), device=device).type(torch.bool)

    # Padding masks (identify where the token is 0)
    src_padding_mask = (src == 0)
    tgt_padding_mask = (tgt == 0)

    return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask
```

Hyperparameters and Model Initialization

In [26]:
```
# Special Tokens
PAD_IDX = 0
SOS_IDX = 1
EOS_IDX = 2

# Hyperparameters
VOCAB_SIZE = 20    # 0~19
DIM_MODEL = 128
NUM_HEADS = 4
NUM_LAYERS = 2
BATCH_SIZE = 64
MAX_LEN = 12

# Initialize Model (Dropout=0.0 for toy task specific optimization)
model = Seq2SeqTransformer(
    VOCAB_SIZE, DIM_MODEL, NUM_HEADS, NUM_LAYERS, NUM_LAYERS, dropout_p=0.0
).to(device)

criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)
optimizer = optim.Adam(model.parameters(), lr=0.0005)

print(f"Model Initialized. Parameters: {sum(p.numel() for p in model.parameters() if p.requires_
```

```
Model Initialized. Parameters: 2,510,356
```

Data Generation (Toy Task: Reverse Sequence)

In [27]:
```
def get_batch(bsz):
    """
    Generates a batch of random sequences.
    Task: Reverse the input sequence.
    Input:  [1, 5, 2, 0] (0 is padding)
    Target: [SOS, 2, 5, 1, EOS]
    """
    data = []
    targets = []

    for _ in range(bsz):
        # Length between 3 and (MAX_LEN - 2) to fit SOS and EOS
        seq_len = random.randint(3, MAX_LEN - 2)
```

```
        # Random sequence (3 ~ 19 range to avoid collision with special tokens)
        seq = [random.randint(3, VOCAB_SIZE-1) for _ in range(seq_len)]

        # Source: Sequence + Padding
        src = seq + [PAD_IDX] * (MAX_LEN - len(seq))

        # Target Sequence (Reverse)
        tgt_seq = seq[::-1]

        # Target Input: SOS + Reversed Seq + Padding
        tgt_input = [SOS_IDX] + tgt_seq + [PAD_IDX] * (MAX_LEN - len(seq) - 1)

        # Target Output: Reversed Seq + EOS + Padding
        tgt_out = tgt_seq + [EOS_IDX] + [PAD_IDX] * (MAX_LEN - len(seq) - 1)

        data.append(src)
        targets.append((tgt_input, tgt_out))

    src = torch.tensor(data).to(device)
    tgt_input = torch.tensor([t[0] for t in targets]).to(device)
    tgt_out = torch.tensor([t[1] for t in targets]).to(device)

    return src, tgt_input, tgt_out
```

Training Loop

In [28]:

```
model.train()
print("Training Start...")

EPOCHS = 3000

for epoch in range(EPOCHS):
    # 1. Get batch data
    src, tgt_input, tgt_out = get_batch(BATCH_SIZE)

    # 2. Create masks
    src_mask, tgt_mask, src_pad_mask, tgt_pad_mask = create_mask(src, tgt_input, device)

    optimizer.zero_grad()

    # 3. Forward pass
    logits = model(src, tgt_input, src_mask, tgt_mask, src_pad_mask, tgt_pad_mask)

    # 4. Calculate Loss
    # Flatten output to [Batch * Seq, Vocab] for CrossEntropyLoss
    loss = criterion(logits.reshape(-1, VOCAB_SIZE), tgt_out.reshape(-1))

    # 5. Backward pass and Optimization
    loss.backward()
    optimizer.step()

    if epoch % 50 == 0:
        print(f"Epoch {epoch:3d} | Loss: {loss.item():.4f}")

print("Training Complete.")
```

```
Training Start...
Epoch   0 | Loss: 3.1280
Epoch  50 | Loss: 1.6600
Epoch 100 | Loss: 1.5024
Epoch 150 | Loss: 1.4160
Epoch 200 | Loss: 1.3212
Epoch 250 | Loss: 1.3039
Epoch 300 | Loss: 1.0827
Epoch 350 | Loss: 0.8528
Epoch 400 | Loss: 0.6721
Epoch 450 | Loss: 0.7049
Epoch 500 | Loss: 0.6442
Epoch 550 | Loss: 0.5091
Epoch 600 | Loss: 0.3460
Epoch 650 | Loss: 0.3792
Epoch 700 | Loss: 0.3189
Epoch 750 | Loss: 0.3119
Epoch 800 | Loss: 0.2251
Epoch 850 | Loss: 0.1883
Epoch 900 | Loss: 0.2307
Epoch 950 | Loss: 0.1384
Epoch 1000 | Loss: 0.1169
Epoch 1050 | Loss: 0.1338
Epoch 1100 | Loss: 0.3391
Epoch 1150 | Loss: 0.1215
Epoch 1200 | Loss: 0.0864
Epoch 1250 | Loss: 0.0806
Epoch 1300 | Loss: 0.1881
Epoch 1350 | Loss: 0.0681
Epoch 1400 | Loss: 0.1538
Epoch 1450 | Loss: 0.0435
Epoch 1500 | Loss: 0.0956
Epoch 1550 | Loss: 0.1904
Epoch 1600 | Loss: 0.0732
Epoch 1650 | Loss: 0.4219
Epoch 1700 | Loss: 0.0887
Epoch 1750 | Loss: 0.0342
Epoch 1800 | Loss: 0.0438
Epoch 1850 | Loss: 0.0157
Epoch 1900 | Loss: 0.0205
Epoch 1950 | Loss: 0.0529
Epoch 2000 | Loss: 0.1631
Epoch 2050 | Loss: 0.0955
Epoch 2100 | Loss: 0.0207
Epoch 2150 | Loss: 0.0187
Epoch 2200 | Loss: 0.0885
Epoch 2250 | Loss: 0.4873
Epoch 2300 | Loss: 0.2237
Epoch 2350 | Loss: 0.0661
Epoch 2400 | Loss: 0.0275
Epoch 2450 | Loss: 0.0204
Epoch 2500 | Loss: 0.0579
Epoch 2550 | Loss: 0.0084
Epoch 2600 | Loss: 0.0148
Epoch 2650 | Loss: 0.0391
Epoch 2700 | Loss: 0.1168
Epoch 2750 | Loss: 0.0406
Epoch 2800 | Loss: 0.0570
Epoch 2850 | Loss: 0.0122
Epoch 2900 | Loss: 0.0036
Epoch 2950 | Loss: 0.0214
Training Complete.
```

Inference (Greedy Decoding)

In [29]:

```python
def greedy_decode(model, src, max_len, start_symbol):
    src = src.to(device)
    # Encoder Masking
    src_padding_mask = (src == PAD_IDX).to(device)

    memory = model.transformer.encoder(
        model.pos_encoder(model.embedding(src) * math.sqrt(DIM_MODEL)),
        src_key_padding_mask=src_padding_mask
    )

    ys = torch.ones(1, 1).fill_(start_symbol).type(torch.long).to(device)

    for i in range(max_len-1):
        tgt_mask = generate_square_subsequent_mask(ys.size(1), device)

        out = model.transformer.decoder(
            model.pos_encoder(model.embedding(ys) * math.sqrt(DIM_MODEL)),
            memory,
            tgt_mask=tgt_mask,
            memory_key_padding_mask=src_padding_mask
        )

        prob = model.out(out[:, -1])
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.item()

        ys = torch.cat([ys, torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)

        if next_word == EOS_IDX:
            break

    return ys

# --- Run Inference ---
model.eval()
print("\nTesting (Source -> Reversed Prediction):")

# Test Sequence
test_src_list = [3, 4, 5, 6, 7, 8, 9]
print(f"Input Sequence: {test_src_list}")

src = torch.tensor([test_src_list + [PAD_IDX]*(MAX_LEN - len(test_src_list))]).to(device)

# Perform inference
pred_tensor = greedy_decode(model, src, MAX_LEN, start_symbol=SOS_IDX)

# Post-processing
result = pred_tensor.squeeze().tolist()
print(f"Raw Output: {result}")

# Clean up output
final_res = []
for token in result:
    if token == SOS_IDX: continue
    if token == EOS_IDX: break
    final_res.append(token)
```

```
print(f"Predicted Result: {final_res}")

# Check correctness
if final_res == test_src_list[::-1]:
    print("✅ Success! The sequence is correctly reversed.")
else:
    print("❌ Failed.")
```

```
Testing (Source -> Reversed Prediction):
Input Sequence: [3, 4, 5, 6, 7, 8, 9]
Raw Output: [1, 9, 8, 7, 6, 5, 4, 3, 2]
Predicted Result: [9, 8, 7, 6, 5, 4, 3]
✅ Success! The sequence is correctly reversed.
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: