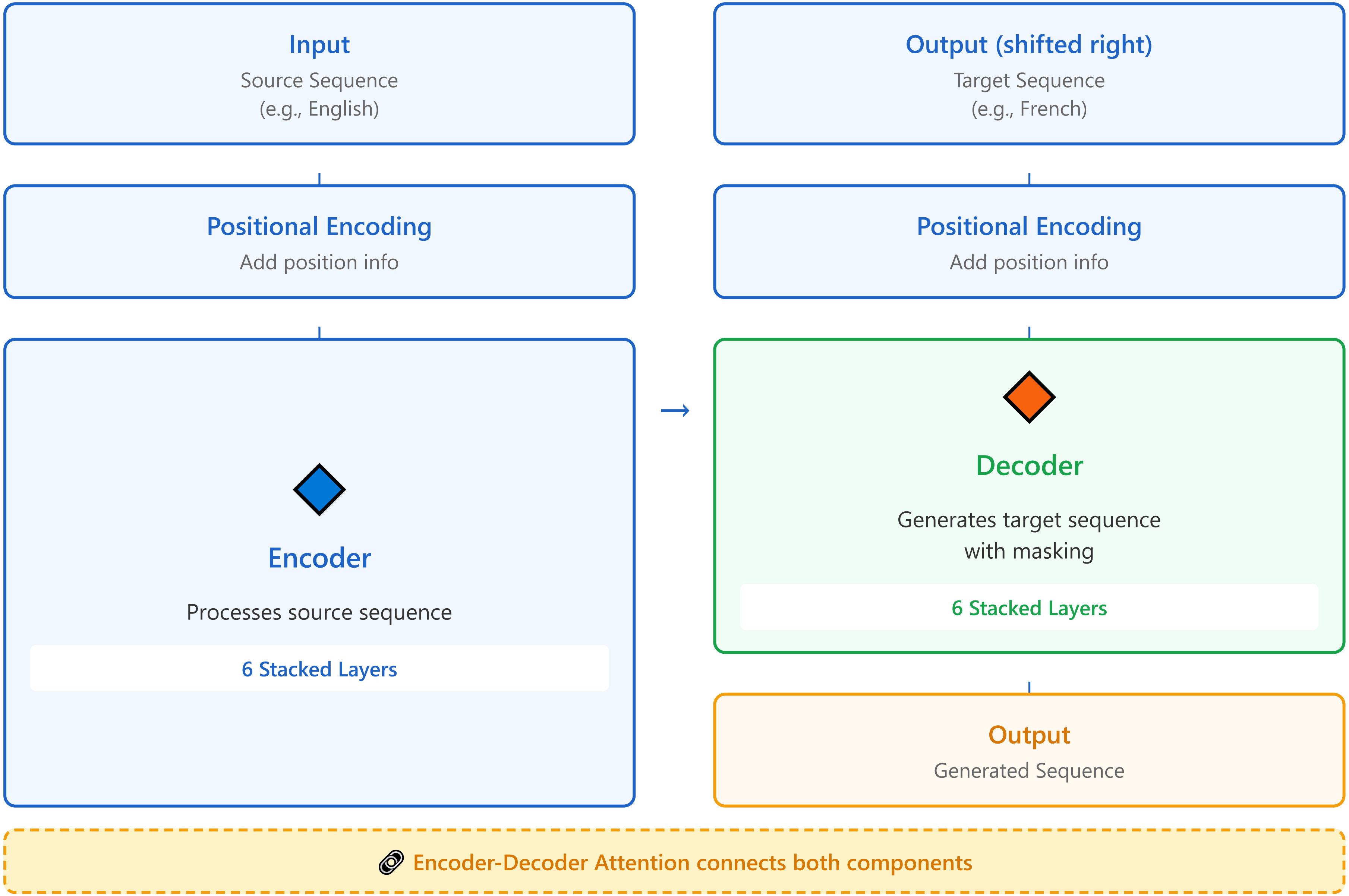
Overall Transformer Structure

Encoder-Decoder Architecture for Sequence-to-Sequence Tasks



Identical layers
with different parameters

Seq2Seq tasks
Translation, Summarization

Decoder uses
masking for causality

Detailed Computation Example

Step-by-step process with actual dimensions (d_model = 4, sequence length = 8)

Step 1: Input Embedding

Source (English): "I love deep learning very much !"

Tokenization: [I, love, deep, learning, very, much, !, <PAD>] → 8 tokens

Embedding Matrix: Vocabulary × d_model → Each token → 4D vector

```
X_embedded (8 × 4) =
[[0.2, 0.5, -0.3, 0.8],  # I
 [0.1, -0.4, 0.6, 0.3],  # love
 [-0.5, 0.2, 0.4, -0.1], # deep
 [0.3, 0.7, -0.2, 0.5],  # learning
 [0.4, -0.3, 0.1, 0.6],  # very
 [-0.2, 0.5, 0.3, -0.4], # much
 [0.6, 0.1, -0.5, 0.2],  # !
 [0.0, 0.0, 0.0, 0.0]]   # <PAD>
```

Step 2: Positional Encoding

Add position information using sin/cos functions:

PE(pos, 2i) = sin(pos / 10000^(2i/d_model))

PE(pos, 2i+1) = cos(pos / 10000^(2i/d_model))

```
PE (8 × 4) =
[[0.00, 1.00, 0.00, 1.00],  # pos 0
 [0.84, 0.54, 0.01, 1.00],  # pos 1
```

```
 [0.91, -0.42, 0.02, 1.00], # pos 2
 [0.14, -0.99, 0.03, 1.00], # pos 3
 [-0.76, -0.65, 0.04, 1.00], # pos 4
 [-0.96, 0.28, 0.05, 1.00],  # pos 5
 [-0.28, 0.96, 0.06, 1.00],  # pos 6
 [0.66, 0.75, 0.07, 0.99]]   # pos 7

X_input (8 × 4) = X_embedded + PE
[[0.20, 1.50, -0.30, 1.80],
 [0.94, 0.14, 0.61, 1.30],
 [0.41, -0.22, 0.42, 0.90],
 [0.44, -0.29, -0.17, 1.50],
 [-0.36, -0.95, 0.14, 1.60],
 [-1.16, 0.78, 0.35, 0.60],
 [0.32, 1.06, -0.44, 1.20],
 [0.66, 0.75, 0.07, 0.99]]
```

🔷 Step 3: Encoder Self-Attention (Layer 1)

Multi-Head Attention computation:

1) Linear projections (d_k = d_model / num_heads = 4 / 2 = 2 per head):

```
W_Q (4 × 4), W_K (4 × 4), W_V (4 × 4)
Q = X_input @ W_Q → (8 × 4)
K = X_input @ W_K → (8 × 4)
V = X_input @ W_V → (8 × 4)
```

2) Attention scores:

```
Scores = Q @ K^T / √d_k → (8 × 8)
// Example scores (showing first 4×4 block):
[[2.3, 0.8, 1.2, 1.5, ...],
 [0.8, 2.1, 0.9, 1.3, ...],
```

```
 [1.2, 0.9, 2.5, 1.1, ...],
 [1.5, 1.3, 1.1, 2.2, ...]]

Attention = softmax(Scores) → (8 × 8)
// Softmax normalizes each row to sum to 1
[[0.35, 0.08, 0.12, 0.15, ...], # I attends to all
 [0.10, 0.31, 0.11, 0.16, ...], # love attends to all
 [0.13, 0.09, 0.41, 0.12, ...], # deep attends to all
 ...]
```

3) Apply attention to values:

```
Output = Attention @ V → (8 × 4)
// Each token representation is now context-aware
```

🔄 Step 4: Add & Norm + Feed-Forward

1) Residual connection and Layer Normalization:

```
X1 = LayerNorm(X_input + Attention_output) → (8 × 4)
```

2) Feed-Forward Network (expand to d_ff = 16, then back to 4):

```
FFN(x) = W_2 @ ReLU(W_1 @ x + b_1) + b_2
W_1: (4 × 16), W_2: (16 × 4)

FFN_output = FFN(X1) → (8 × 4)
X_encoder1 = LayerNorm(X1 + FFN_output) → (8 × 4)
```

→ Repeat this process for 6 encoder layers
→ Final encoder output: (8 × 4)

Step 5: Decoder Input & Masked Self-Attention

Target (French): "<BOS> J' aime l' apprentissage profond <PAD>"

Shifted right: [<BOS>, J', aime, l', apprentissage, profond, <PAD>, <PAD>]

After embedding + positional encoding: Y_input (8 × 4)

Masked Self-Attention:

```
Mask prevents attending to future tokens:
[[1, 0, 0, 0, 0, 0, 0, 0],  # <BOS> only sees itself
 [1, 1, 0, 0, 0, 0, 0, 0],  # J' sees <BOS>, J'
 [1, 1, 1, 0, 0, 0, 0, 0],  # aime sees up to aime
 [1, 1, 1, 1, 0, 0, 0, 0],  # l' sees up to l'
 [1, 1, 1, 1, 1, 0, 0, 0],  # apprentissage
 [1, 1, 1, 1, 1, 1, 0, 0],  # profond
 [1, 1, 1, 1, 1, 1, 1, 0],  # <PAD>
 [1, 1, 1, 1, 1, 1, 1, 1]]  # <PAD>

Masked_Attention_output → (8 × 4)
```

Step 6: Encoder-Decoder Cross-Attention

Query from decoder, Key & Value from encoder:

```
Q = Decoder_output @ W_Q → (8 × 4)  # from decoder
K = Encoder_output @ W_K → (8 × 4)  # from encoder
V = Encoder_output @ W_V → (8 × 4)  # from encoder

Cross_Attention = softmax(Q @ K^T / √d_k) @ V
// Decoder attends to ALL encoder positions
// This allows decoder to "look at" the source sentence
```

```
Cross_Attention_output → (8 × 4)
```

📤 Step 7: Final Output & Prediction

After 6 decoder layers:

```
Decoder_final_output → (8 × 4)
```

Linear projection to vocabulary size (e.g., 10,000):

```
Logits = Decoder_output @ W_output → (8 × 10000)
Probabilities = softmax(Logits) → (8 × 10000)
```

For each position, pick highest probability token:

```
Position 0: P("J'") = 0.92 → output "J'"
Position 1: P("aime") = 0.87 → output "aime"
Position 2: P("l'") = 0.91 → output "l'"
Position 3: P("apprentissage") = 0.84 → output "apprentissage"
Position 4: P("profond") = 0.89 → output "profond"
Position 5: P("<EOS>") = 0.95 → output "<EOS>" (stop)

Final Translation: "J' aime l' apprentissage profond"
```

🎯 Key Dimensional Flow Summary

Input tokens (8) → Embedding (8×4) → +PE (8×4) →
Encoder layers (8×4 maintained) → Encoder output (8×4) →

Decoder input (8×4) → Decoder layers (8×4 maintained) →

Linear projection (8×4 → 8×10000) → Softmax → Predictions (8×10000)