# 📘 Encoder–Decoder (Seq2Seq) with Attention

## 🟢 How it works:

1. **Encoder** → Reads the input sentence (e.g., "I love dogs").
   * Converts words → numbers → embeddings.
   * Passes them through LSTM → produces hidden states (memory).
2. **Context Vector** → The last hidden state is sent to the decoder.
3. **Decoder** → Starts with a special token <sos> (“start of sentence”).
   * Generates one word at a time (e.g., "J' → "aime" → "les" → "chiens").
   * Keeps using its last output as the next input.

## 📘 Problem (Why Attention is needed?)

* If the input sentence is **short**, Encoder–Decoder works fine.
* If the sentence is **long**, compressing everything into one hidden vector loses information → output becomes inaccurate.

## 📘 Attention Mechanism

👉 Instead of only using the **last hidden state**, the decoder looks at **all encoder states** and learns **where to focus**.

Example:  
- While generating "chiens" (dogs), the decoder gives more weight to "dogs" in the input sentence.  
- While generating "aime" (love), it gives more weight to "love".

This is why translations and text generation improve.

## 📘 Code (Explained Simply)

### Encoder

class Encoder(nn.Module):  
 def \_\_init\_\_(self, ...):  
 self.embedding = nn.Embedding(input\_dim, emb\_dim) # convert word → vector  
 self.lstm = nn.LSTM(emb\_dim, hid\_dim) # LSTM processes input  
  
 def forward(self, src):  
 embedded = self.embedding(src)  
 outputs, (hidden, cell) = self.lstm(embedded)  
 return outputs, (hidden, cell) # all hidden states + last hidden state

👉 **Encoder reads input sentence and gives memory (outputs) + final hidden state.**

### Attention

class Attention(nn.Module):  
 def forward(self, hidden, encoder\_outputs):  
 # Compare decoder hidden state with encoder outputs  
 # Gives attention weights (which words to focus on)  
 return weights

👉 **Attention decides which word in input is important at each decoding step.**

### Decoder

class Decoder(nn.Module):  
 def forward(self, input, hidden, cell, encoder\_outputs):  
 embedded = self.embedding(input) # current word vector  
 attn\_weights = self.attention(hidden, encoder\_outputs)  
 context = torch.bmm(attn\_weights, encoder\_outputs) # weighted input  
 output, (hidden, cell) = self.lstm(torch.cat((embedded, context), dim=2))  
 prediction = self.fc\_out(torch.cat((output, context), dim=1))  
 return prediction, hidden, cell

👉 **Decoder uses:**  
- Current word,  
- Context vector (from attention),  
- Previous hidden state,

To generate the **next word**.

## 📘 Example Flow (Simple)

Input: "I love dogs"  
Output: "J'aime les chiens"

1. Encoder → reads "I love dogs" → produces hidden states.
2. Decoder → starts with <sos> → outputs "J'
   * Looks at encoder states → focuses on "I".
3. Next step → outputs "aime"
   * Focuses on "love".
4. Next step → outputs "les"
   * Focuses on grammar context.
5. Next step → outputs "chiens"
   * Focuses on "dogs".

## ✅ In short:

* **Encoder** → understands the input.
* **Decoder** → generates output step by step.
* **Attention** → helps decoder decide *which part of the input* to use each time.