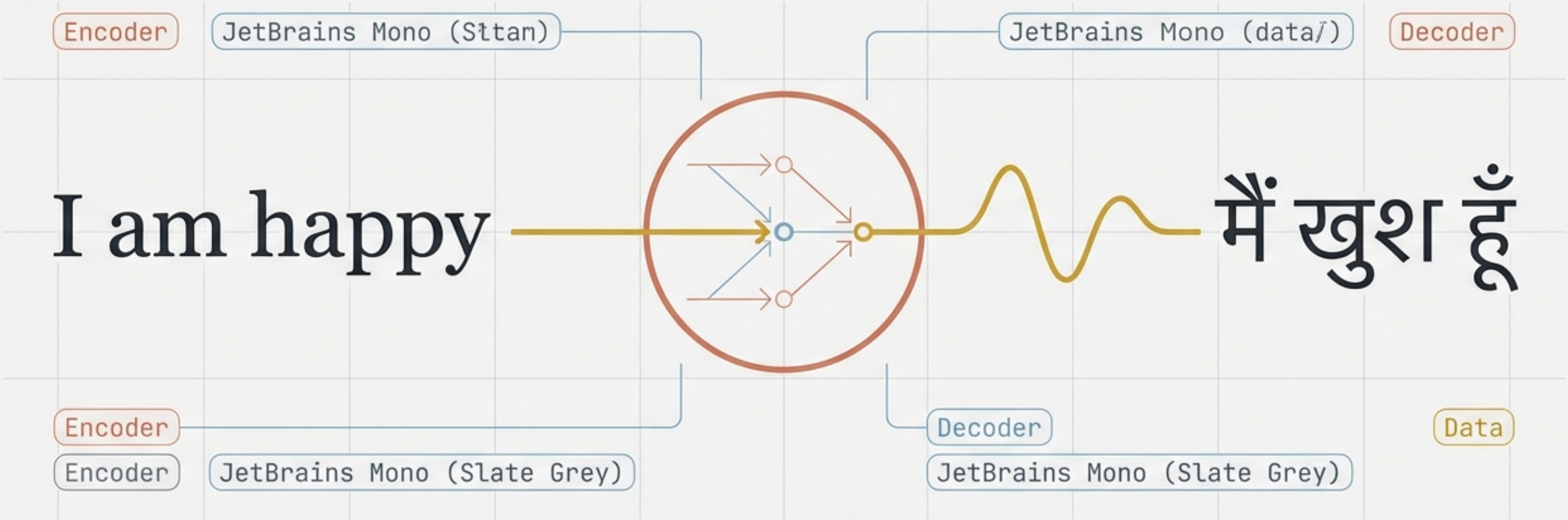
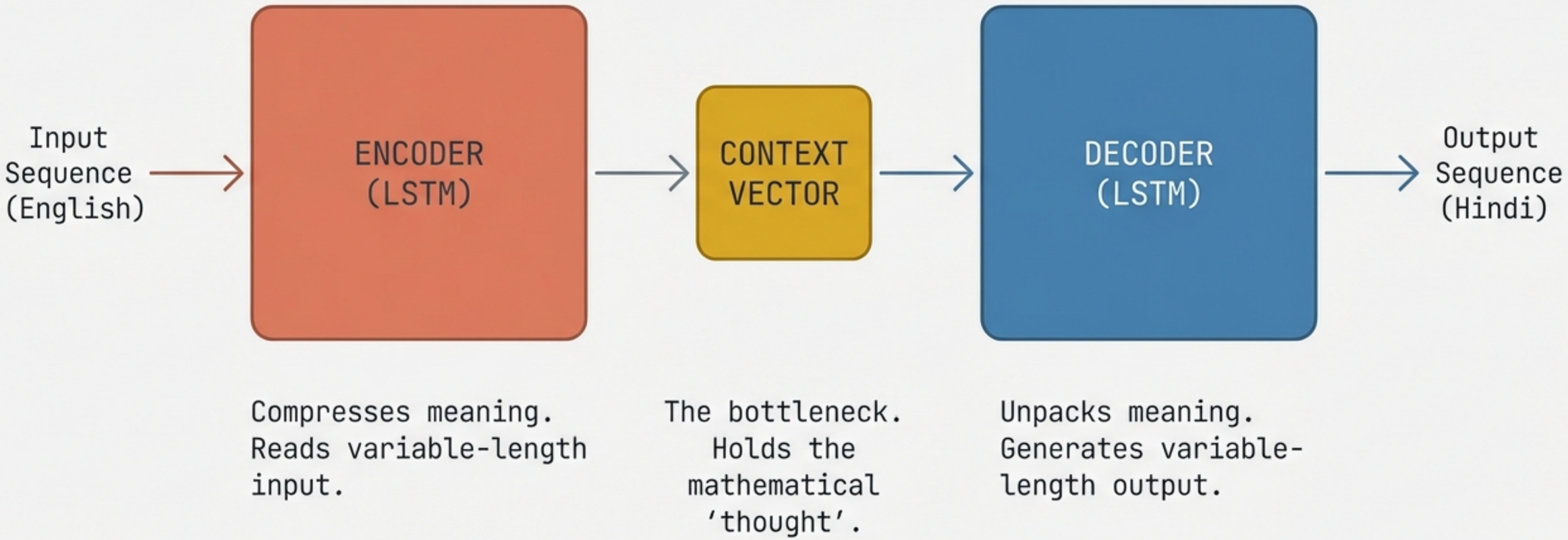
Building a Translator

A step-by-step journey inside a Seq2Seq LSTM Model.

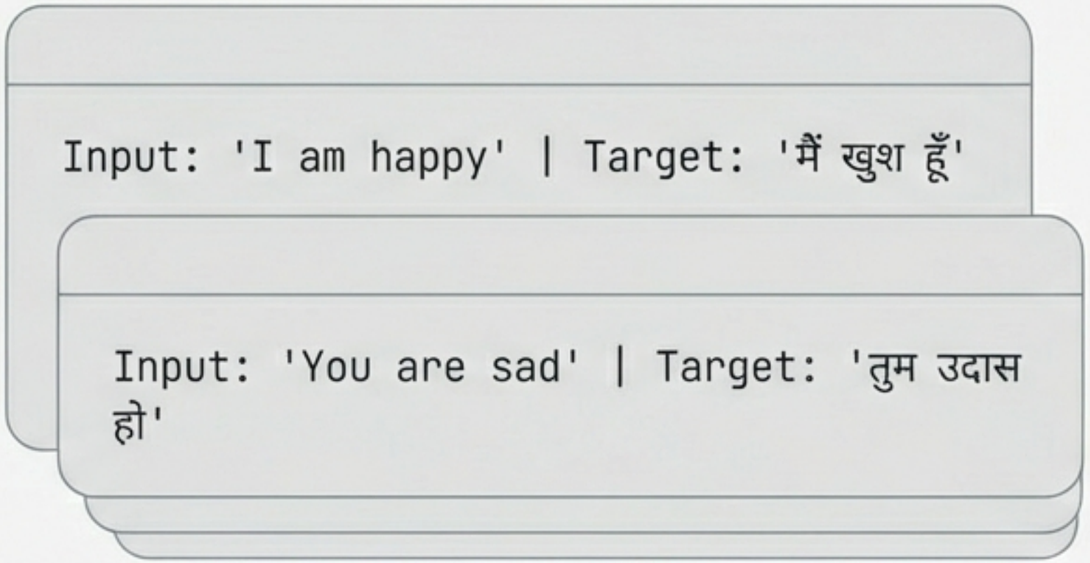


Architecture: Sequence-to-Sequence with Attention-free LSTM · **Framework:** TensorFlow 2.x

The Architecture: The Map of the Journey



The Setup: Imports and Raw Data




```
import tensorflow as tf
import numpy as np

# The Dataset (English -> Hindi)
data = [
    ("I am happy", "मैं खुश हूँ"),
    ("You are sad", "तुम उदास हो"),
    ("She is tired", "वह थकी हुई है"),
    # ... typically millions of pairs
]
```

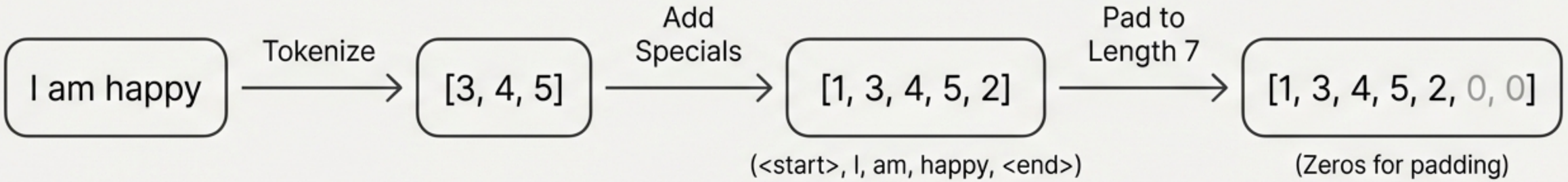
Vocabulary: Building the Dictionary

Lookup Table

Key		Value
<pad>	→	0
<start>	→	1
<end>	→	2
I	→	3
am	→	4
happy	→	5

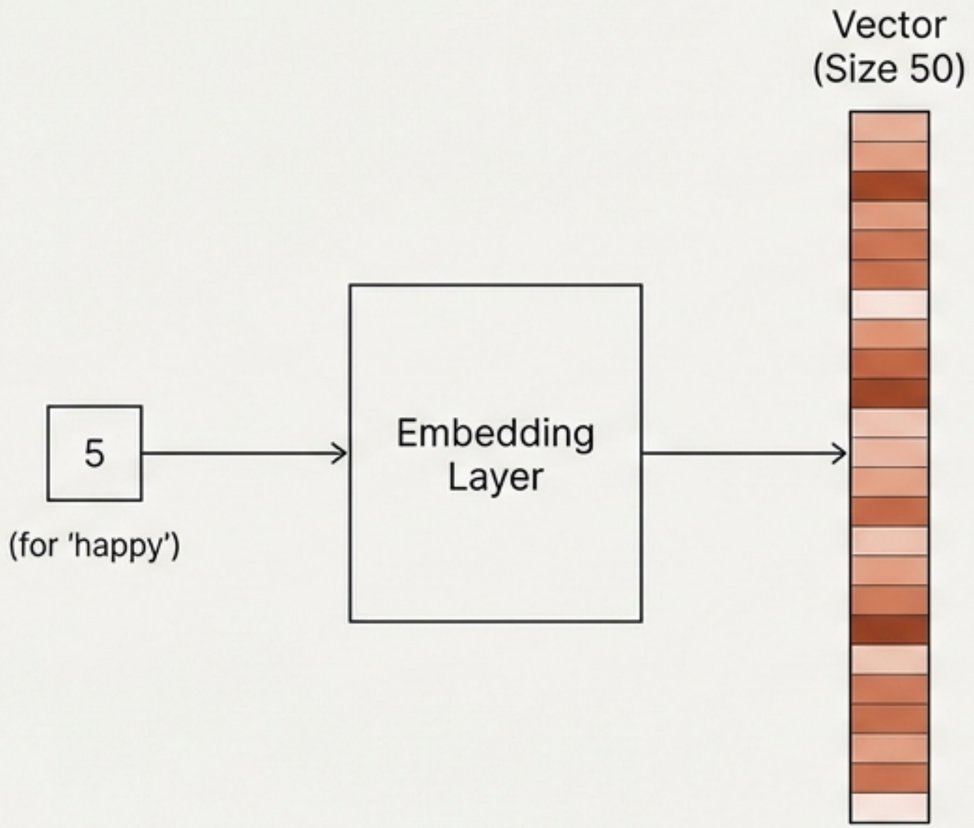
```python
def build_vocab(sentences):
    # Create mapping of words to unique IDs
    vocab = {"<pad>": 0, "<start>": 1, "<end>": 2}

    # Assign IDs to remaining words
    for sentence in sentences:
        for word in sentence.split():
            if word not in vocab:
                vocab[word] = len(vocab)
    return vocab
```

Pre-processing: Indexing and Padding



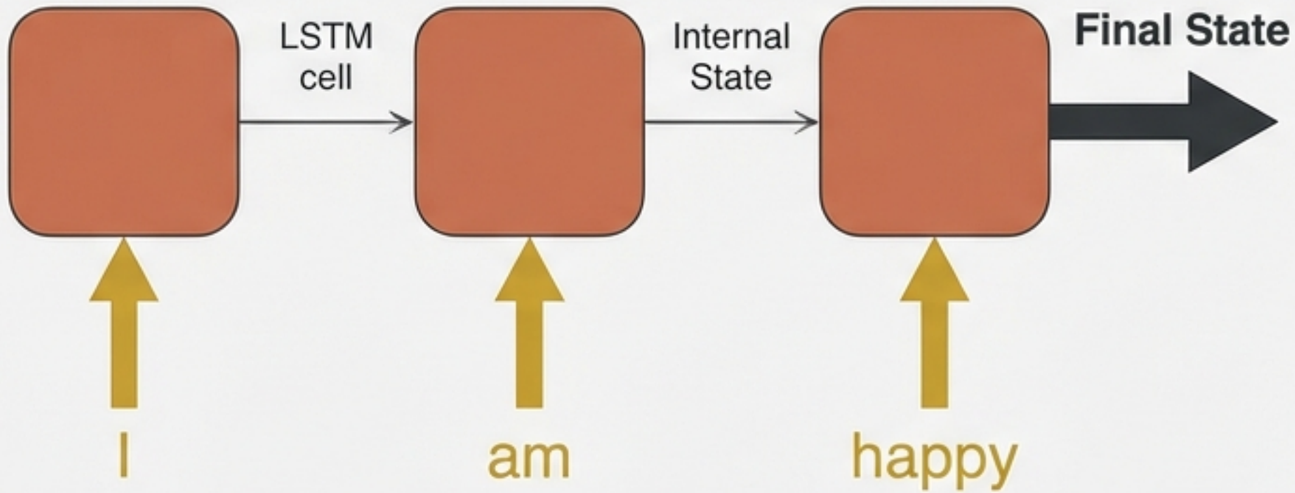

```
# Convert text to sequence of integers
src_data = [sentence_to_indices(pair[0], vocab) for pair in data]

# Pad sequences to ensure equal length (post-padding)
# 0 is the <pad> token
src_padded = tf.keras.preprocessing.sequence.pad_sequences(
    src_data,
    padding='post',
    value=vocab["<pad>"]
)
```

The Encoder Part 1: The Embedding Layer



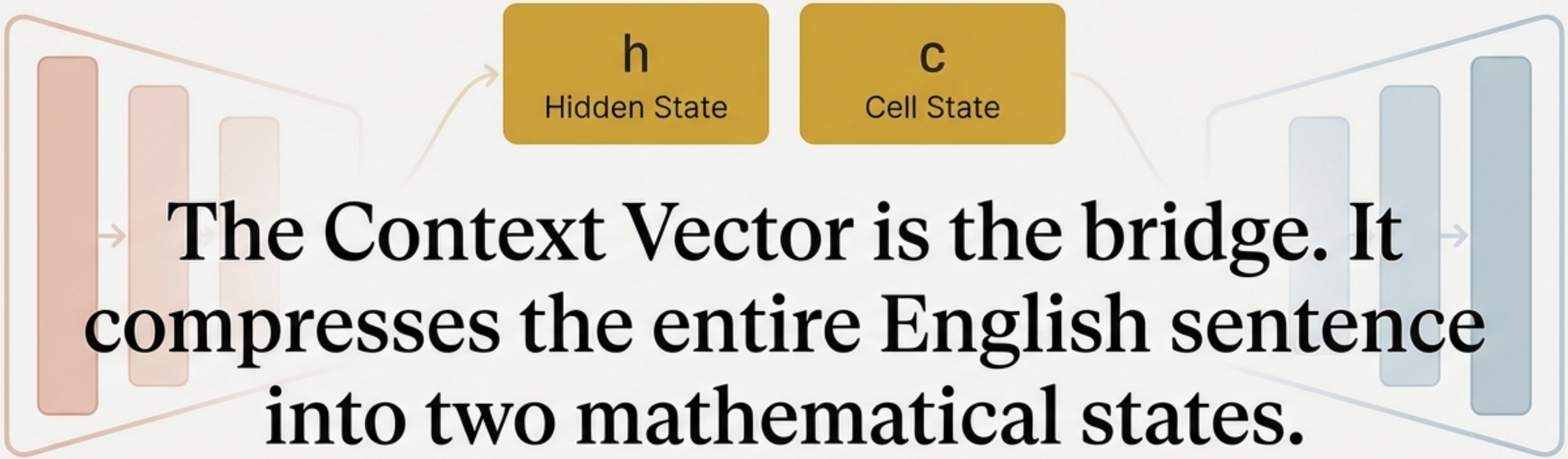

```python
class Encoder(tf.keras.Model):
  def __init__(self, input_size, embed_size, hidden_size):
    super(Encoder, self).__init__()
    # Turns indices (5) into vectors (size 50)
    self.embedding = tf.keras.layers.Embedding(input_size, embed_size)

  def call(self, x):
    # x shape: [batch_size, seq_len]
    embedded = self.embedding(x)
    # embedded shape: [batch_size, seq_len, embed_size]
    return embedded
```

The Encoder Part 2: The LSTM




```python
def __init__(self, ...):
    # return_state=True is crucial!
    self.lstm = tf.keras.layers.LSTM(hidden_size,
        return_state=True)

def call(self, x):
    embedded = self.embedding(x)

    # We ignore 'output' (predictions)
    # We keep 'hidden' and 'cell' (the memory)
    output, hidden, cell = self.lstm(embedded)

    return hidden, cell
```

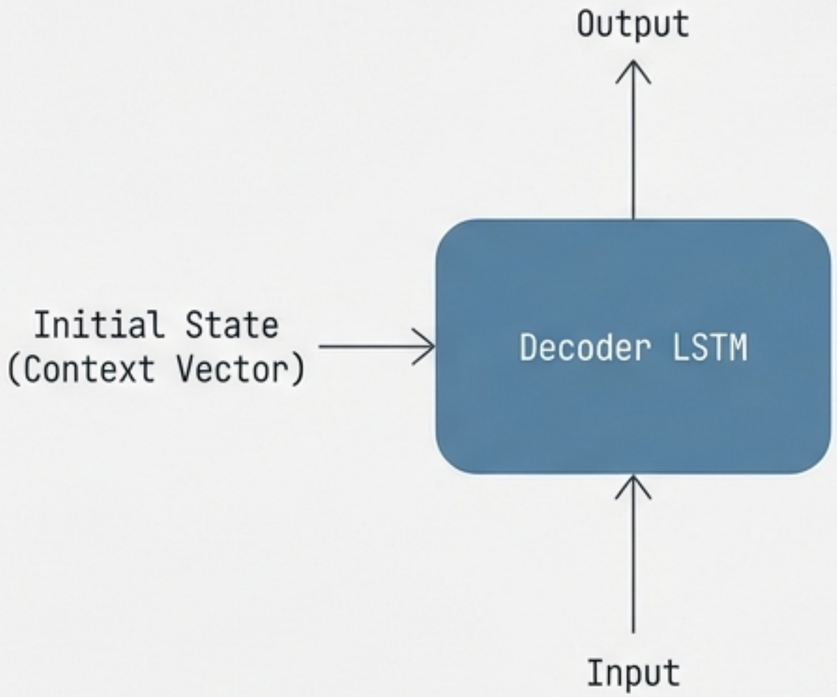
The Handoff: The Context Vector



The Context Vector is the bridge. It compresses the entire English sentence into two mathematical states.

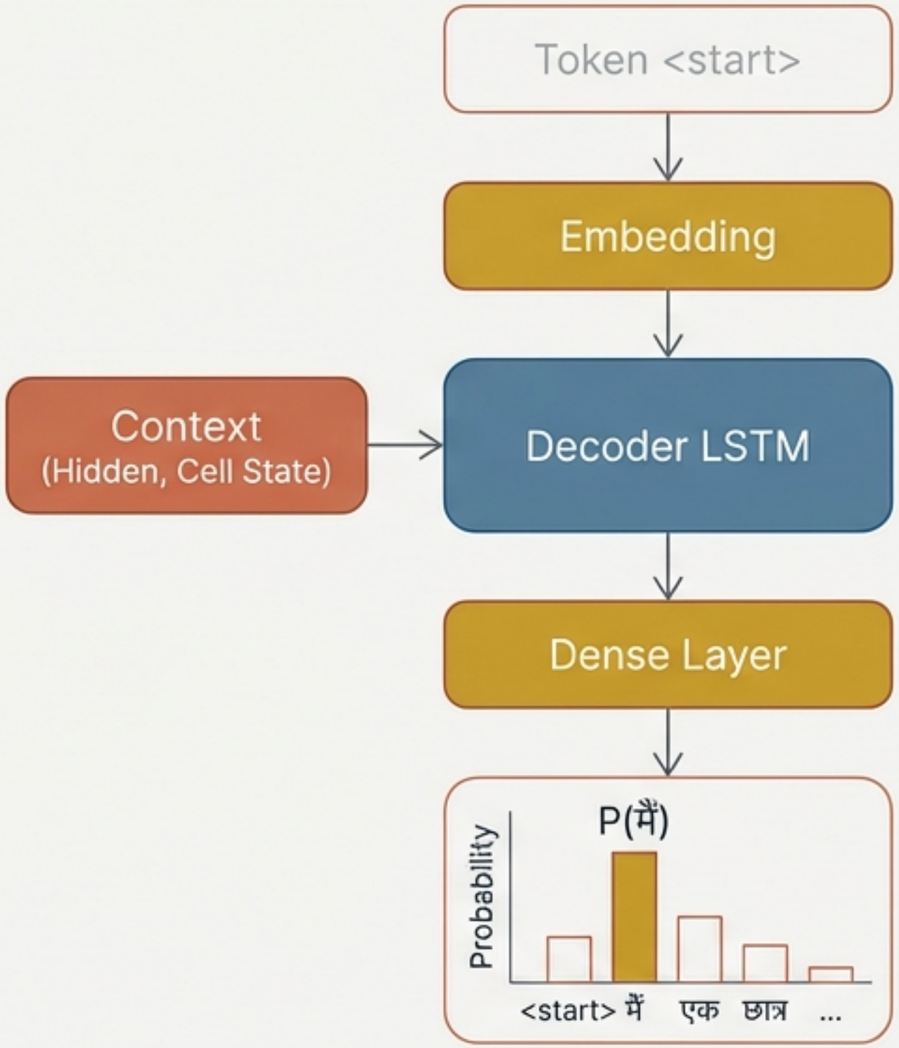
```
# From Encoder
return hidden, cell

# To Decoder
decoder_initial_state = [hidden, cell]
```

The Decoder: Initialization



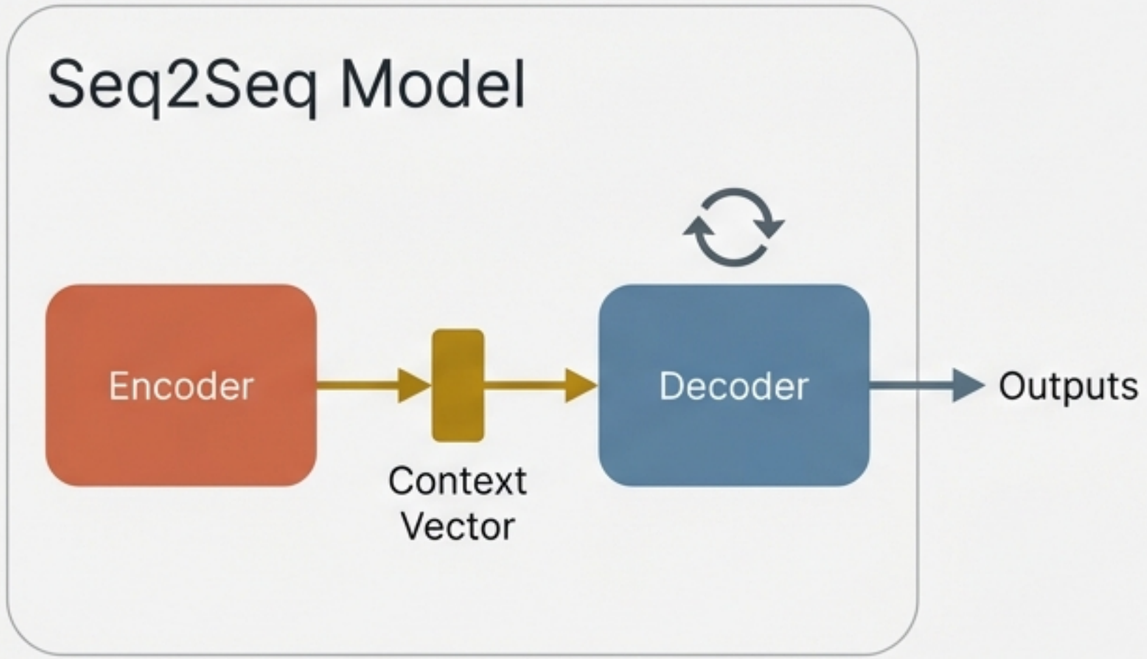

```
class Decoder(tf.keras.Model):
   def __init__(self, output_size, embed_size, hidden_size):
      super(Decoder, self).__init__()
      self.embedding = tf.keras.layers.Embedding(output_size, embed_size)

      # return_sequences=True: We need a word at every step
      self.lstm = tf.keras.layers.LSTM(hidden_size,
                                       return_sequences=True,
                                       return_state=True)

      self.fc = tf.keras.layers.Dense(output_size)
```

The Decoder: Generating Words



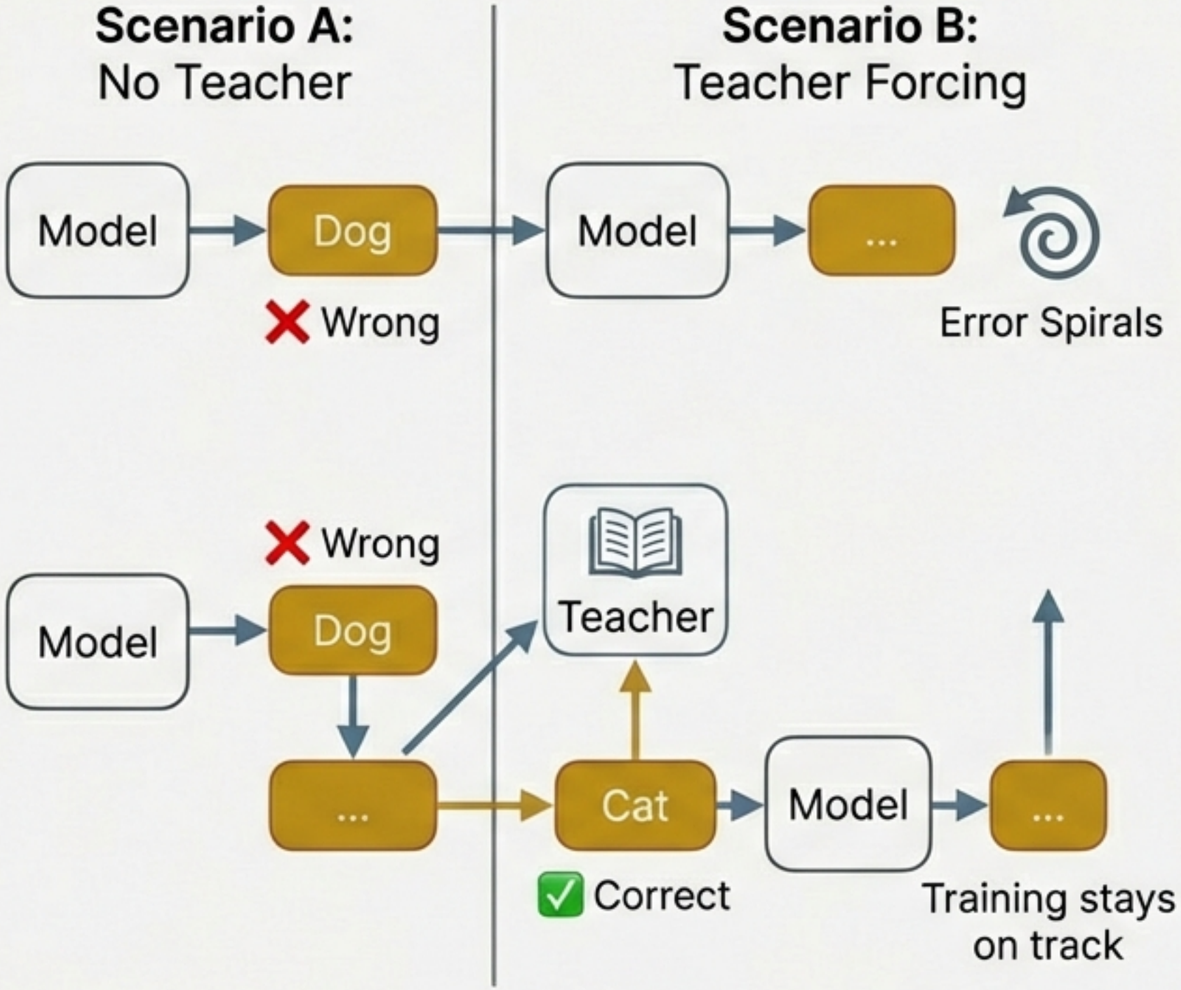

```
def call(self, x, hidden, cell):
    # x is the previous word (or <start>)
    embedded = self.embedding(x)

    # LSTM updates memory, produces output
    lstm_out, hidden, cell = self.lstm(embedded,
        initial_state=[hidden, cell])

    # Dense layer predicts next word probability
    output = self.fc(lstm_out)

    return output, hidden, cell
```

The Wrapper: The Seq2Seq Class



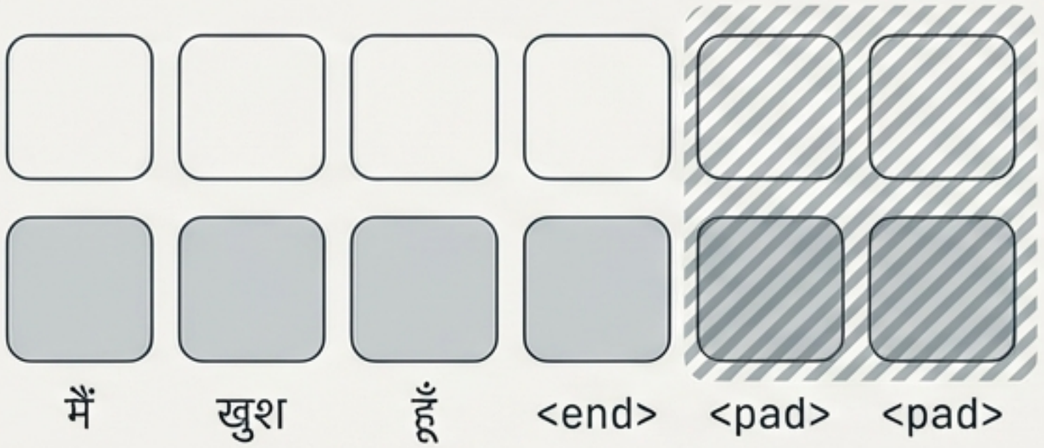

```
class Seq2Seq(tf.keras.Model):
  def call(self, inputs):
    src, tgt = inputs

    # 1. Encode the source
    hidden, cell = self.encoder(src)

    # 2. Decode loop
    outputs = []
    input_token = tgt[:, 0:1] # Start with <start>

    for t in range(1, tgt_len):
       output, hidden, cell =
         self.decoder(input_token, hidden, cell)
       outputs.append(output)
       # ... update input_token for next step ...

    return tf.concat(outputs, axis=1)
```

Training Strategy: Teacher Forcing




```
if training:
    # Teacher Forcing: Feed the ACTUAL
      next word from dataset
    teacher_force = random.random() <
       teacher_forcing_ratio
    input_token = tgt[:, t:t+1] if
       teacher_force else
       predicted_token
else:
    # Inference: Must use model's own
    prediction
    input_token = predicted_token
```

The Loss Function: Grading the Exam



Masking (Ignore Padding)

```
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

# Create mask to ignore padding tokens (0)
# We don't want to penalize the model for padding errors
mask = tf.cast(target_labels != 0, tf.float32)

loss = loss_fn(target_labels, outputs)
loss = loss * mask # Apply mask to zero-out padding loss
```

Inference: Translation in Action

```
Input: "I am happy"

---

Step 1: Predicted "मैं"
Step 2: Predicted "खुश"
Step 3: Predicted "हूँ"
Step 4: Predicted "<end>" -> STOP

---

Result: "मैं खुश हूँ"
```

```python
def translate(sentence):
    # Encode inputs...
    hidden, cell = model.encoder(src_tensor)
    input_token = [start_token]

    for step in range(max_len):
        output, hidden, cell = model.decoder(
          input_token, hidden, cell)
        predicted_token = tf.argmax(output,
          axis=-1)

        if predicted_token == end_token:
            break
        input_token = predicted_token
```

Concepts Mastered

✓	**Tokenization**: Converting text to indices. JetBrains Mono, Slate Grey	✓	**Encoder-Decoder**: The Seq2Seq architecture. JetBrains Mono, Slate Grey
✓	**Padding**: Handling variable sequence lengths. JetBrains Mono, Slate Grey	✓	**Context Vector**: The information bottleneck. JetBrains Mono, Slate Grey
✓	**Embedding**: Dense vector representation of meaning. JetBrains Mono, Slate Grey	✓	**Teacher Forcing**: Stabilizing training. JetBrains Mono, Slate Grey

This Seq2Seq + LSTM architecture laid the groundwork for modern NLP. While Attention mechanisms and Transformers (BERT, GPT) have since evolved, they stand on the shoulders of this logic.