Sequence-to-Sequence Translation Using Recurrent Neural Networks

CSE 4311 – Neural Networks and Deep Learning
Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

Sequence-To-Sequence Models

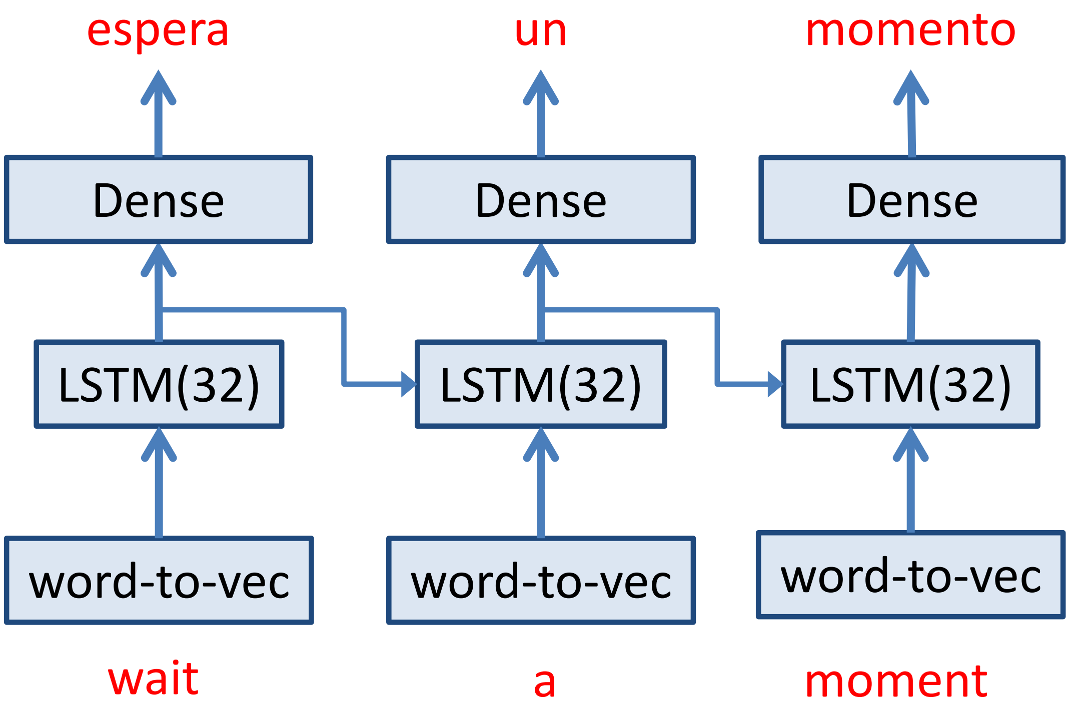
- The text processing models that we have seen so far were designed for classification.
  - Input: text (a movie review document in our example)
  - Output: a class label (“positive” or “negative” in our example).
- Another important family of models are sequence-to-sequence models:
  - Input: text.
  - Output: text.
- The example application that we will build is English-to-Spanish translation.
  - Input: English text.
  - Output: Spanish text.

Sequence-To-Sequence Models

- There are many other applications of sequence-to-sequence models.
- Text summarization:
  - Input: a relatively long piece of text.
  - Output: a summary of the input, shorter and keeping the most important information.
- Question answering:
  - Input: a question.
  - Output: an answer to that question in natural language (as opposed answering a multiple choice question).
- Conversational agents (e.g., chatbots):
  - Input: conversation so far.
  - Output: reply to the user's last entry.

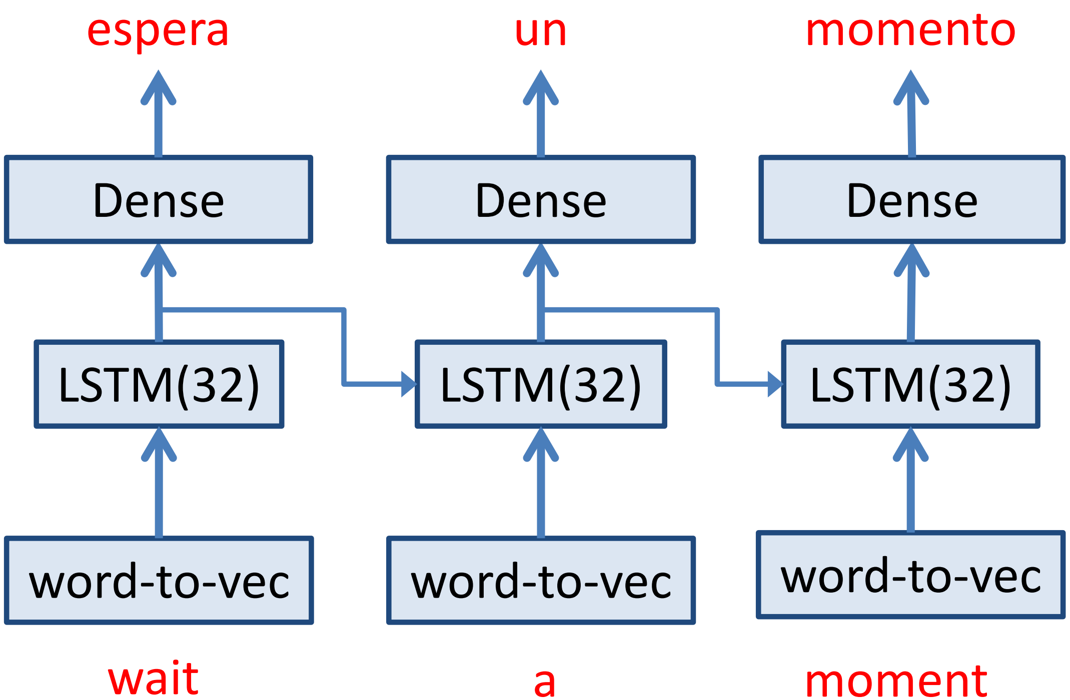
A Simple Approach

- A simple approach (which turns out not to be a good idea) would be to have a simple RNN model to do the translation.
- Here you see a simplified sketch of a model.
  – Note: this sketch shows blocks of units, NOT individual units.



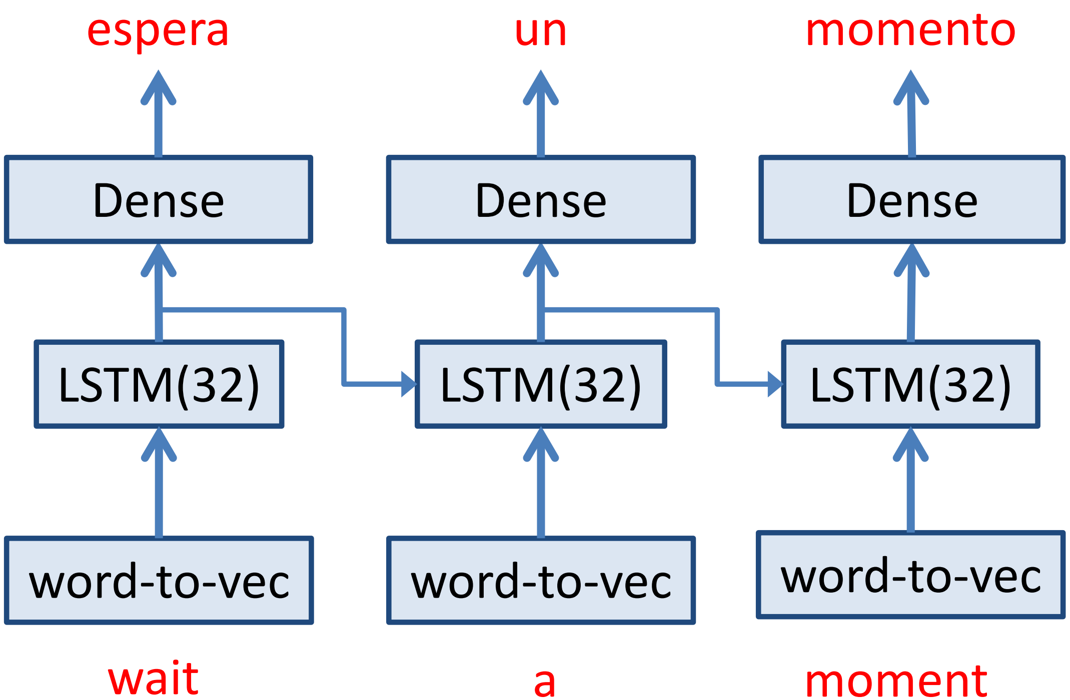
A Simple Approach

- The "word-to-vec" operation simply converts each word to a vector, somehow (we typically use word embeddings).
- The LSTM layer with 32 units is a token recurrent layer, can be replaced by any other recurrent layer or parameters.



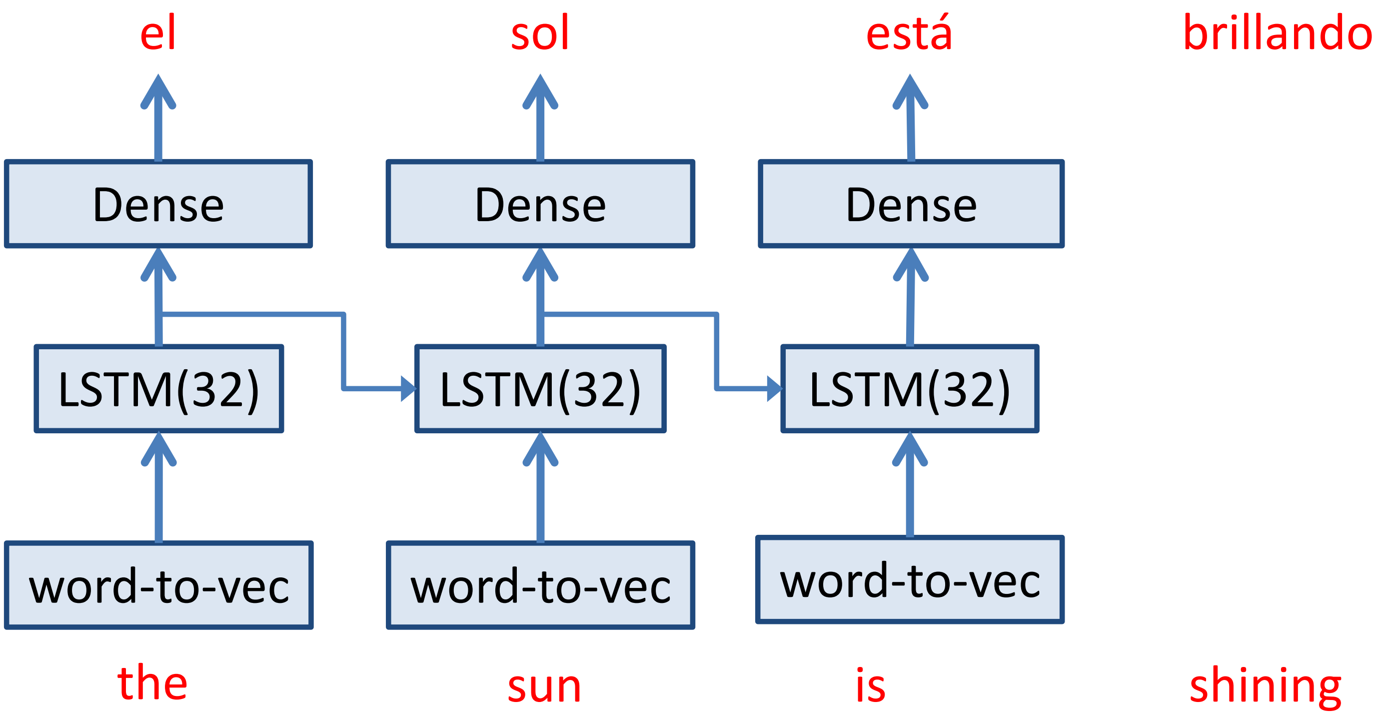
A Simple Approach

- The "Dense" layer produces the classification output.
  - As usual, we have as many output units as the number of classes (in this case, number words in our Spanish vocabulary).
  - We find the unit with the highest output, and we produce the corresponding word.



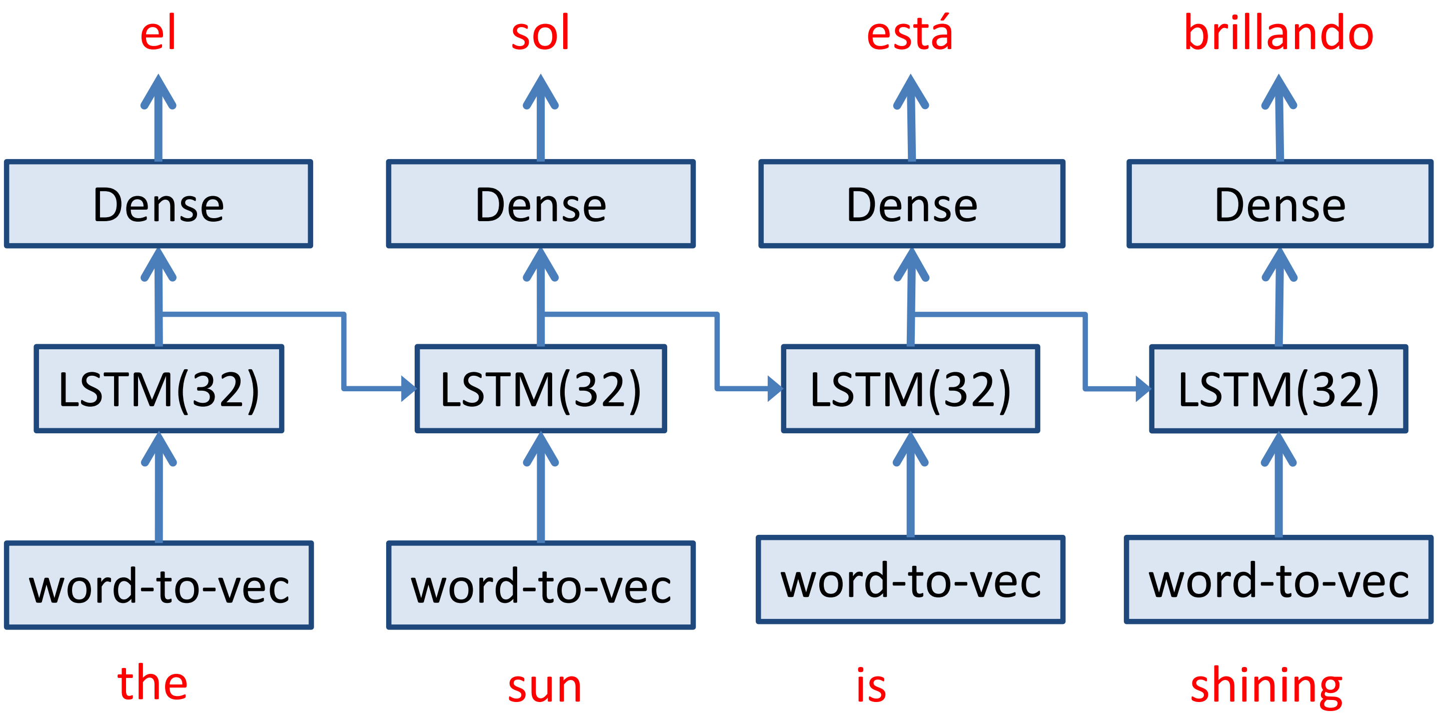
A Simple Approach

- Can this model handle input sentences with four words?



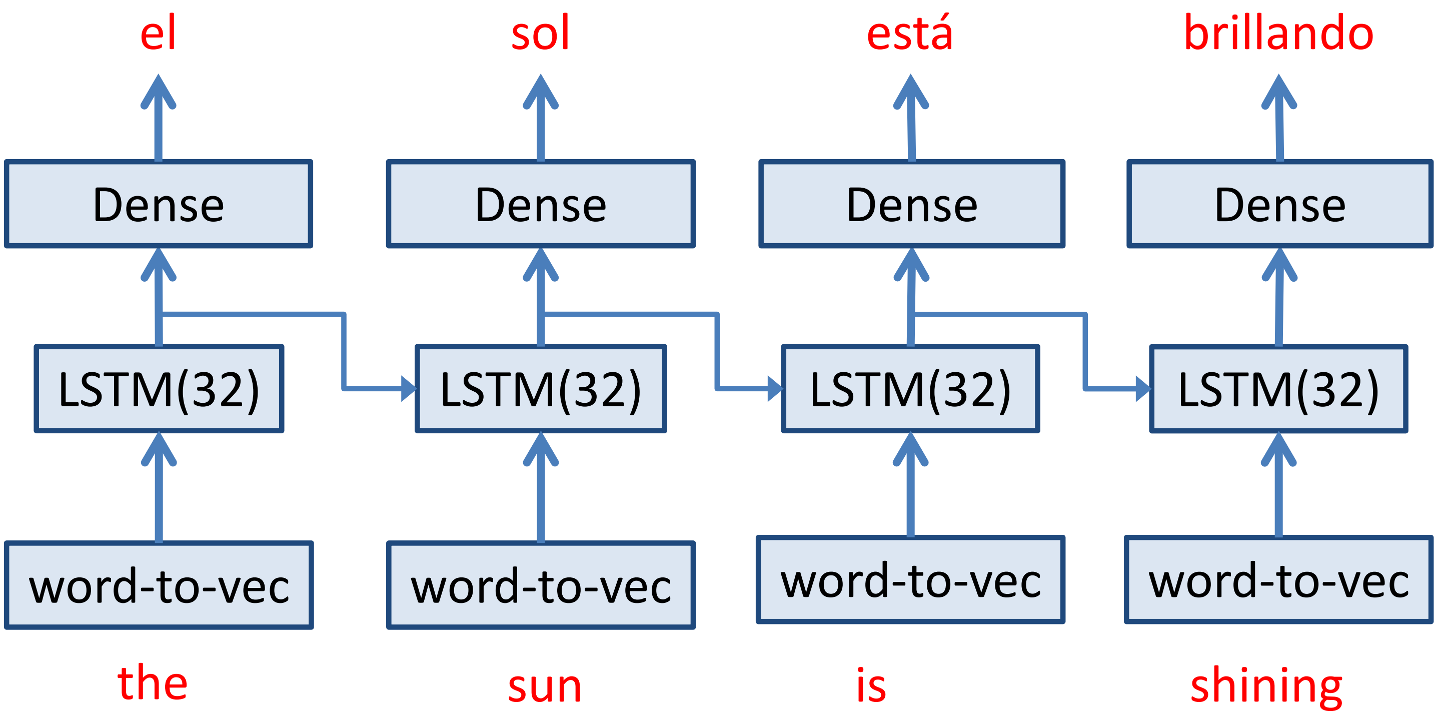
A Simple Approach

- Can this model handle input sentences with four words?
- YES!!! Remember, recurrent models process their input step by step.
- Each step is processed the same way, using the same layers.



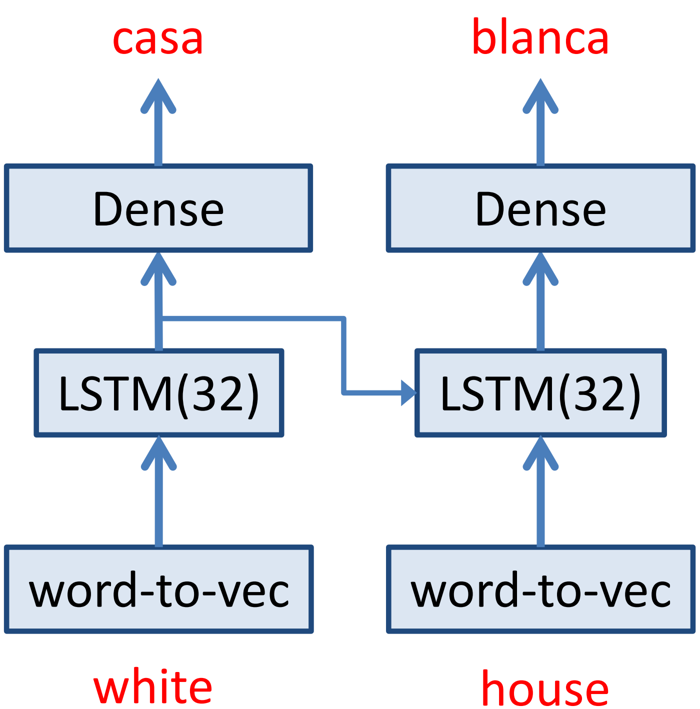
A Simple Approach: Problems

- So, why is this type of model a bad idea?



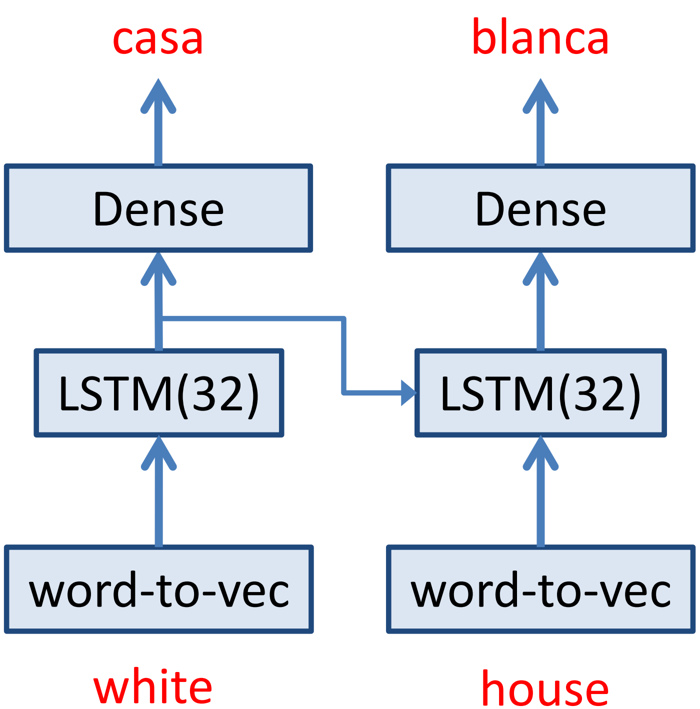
A Simple Approach: Problems

- So, why is this type of model a bad idea?
- One problem: how much of the input sentence has the model seen when it outputs the first word?



A Simple Approach: Problems

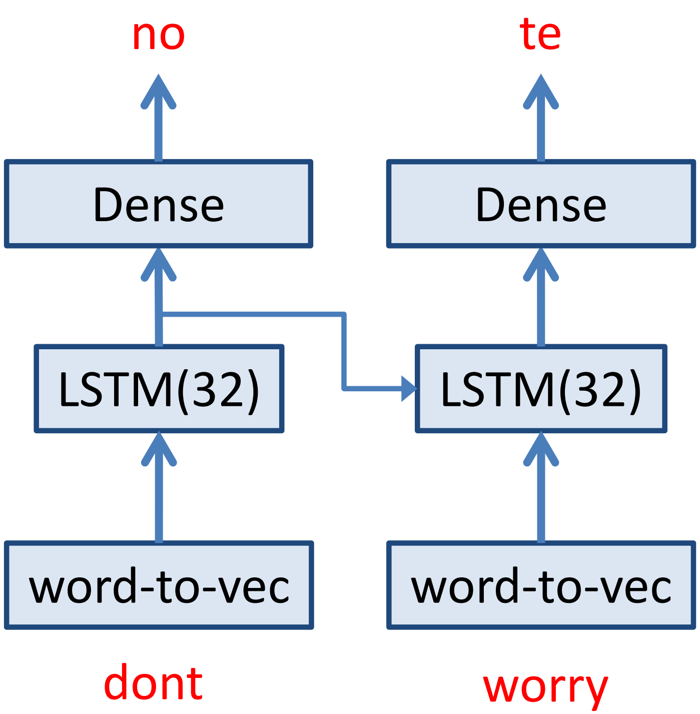
- Remember, the model processes the input time step by time step.
  - Each step produces an output.
  - So, the output of the first step depends only on the first word.



- Here, the correct result should place the noun in front of the adjective.
  - "casa" is "house".
  - "blanca" is "white".
- In general, word order varies a lot among languages.
- When the model sees that the first input word is "white", there is no way to know that the correct first output is the Spanish word for "house".

A Simple Approach: Problems

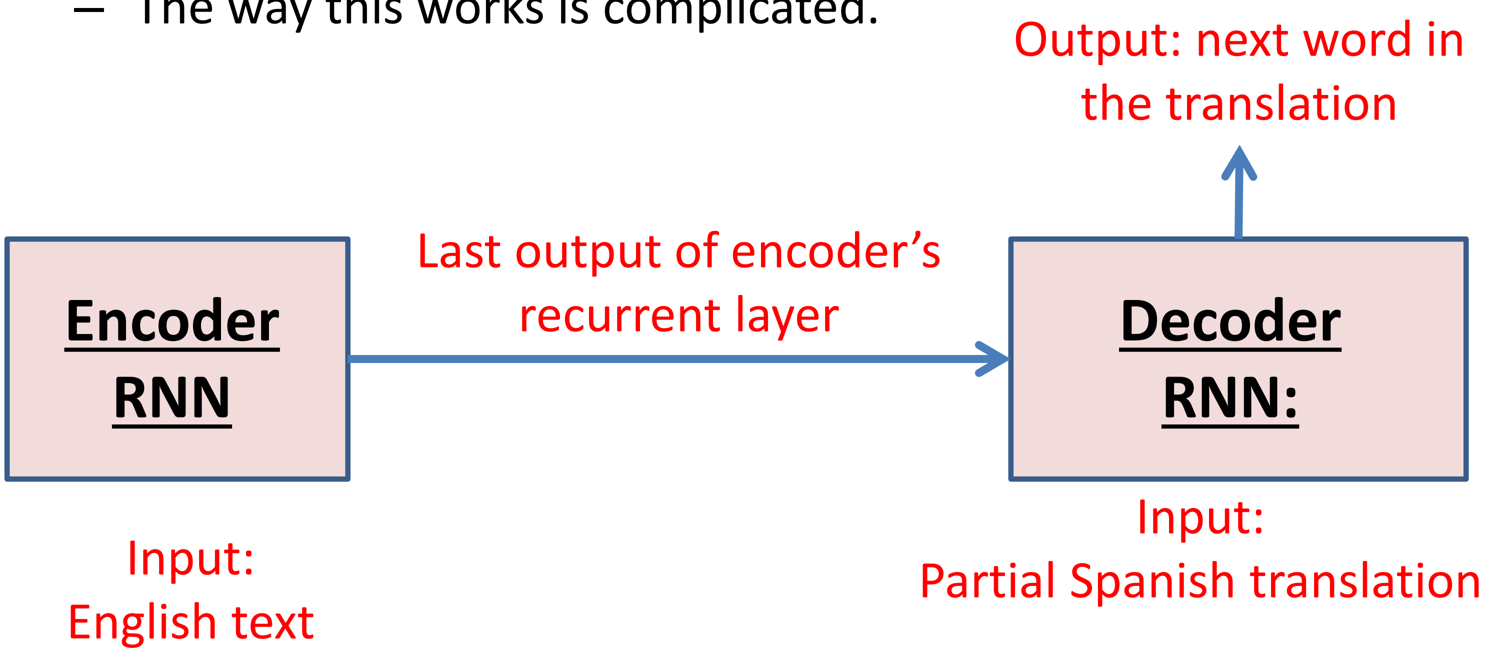
- Another problem: the output has the same length as the input.
- That is not a correct assumption for language-to-language translation.
  - Here, the input is 2 words, the correct translation is 3 words. This model cannot produce the correct translation.



preoccupes

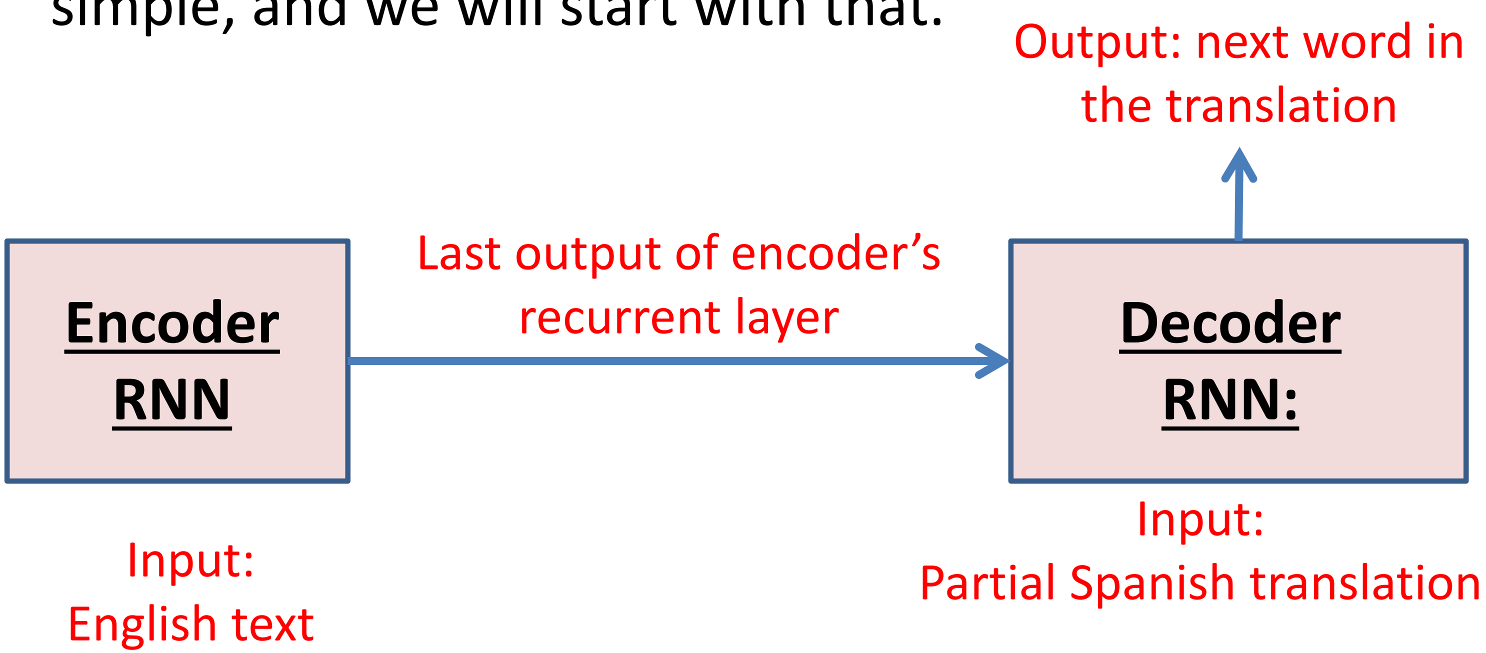
Encoder-Decoder Architecture

- To overcome the limitations of the simple approach, we typically use what is called an **encoder-decoder architecture**.
  - The encoder is an RNN, that takes the English text as input.
  - The output of the encoder is fed into the decoder.
  - The decoder is a **SEPARATE RNN**.
  - The way this works is complicated.



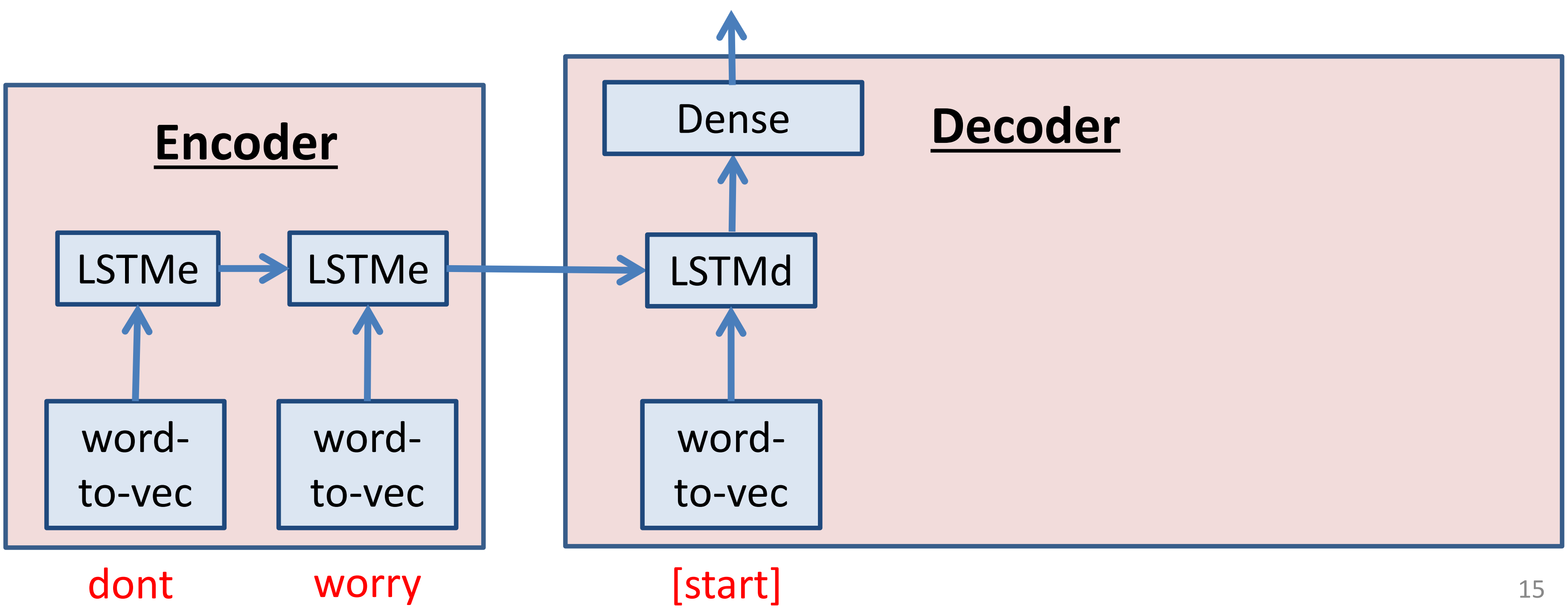
Encoder-Decoder Architecture

- The process we follow during training is a bit different than the process we follow during inference.
  - "Inference" means applying the already trained model to translate a new piece of English text.
- As is often the case, the process during inference is more simple, and we will start with that.



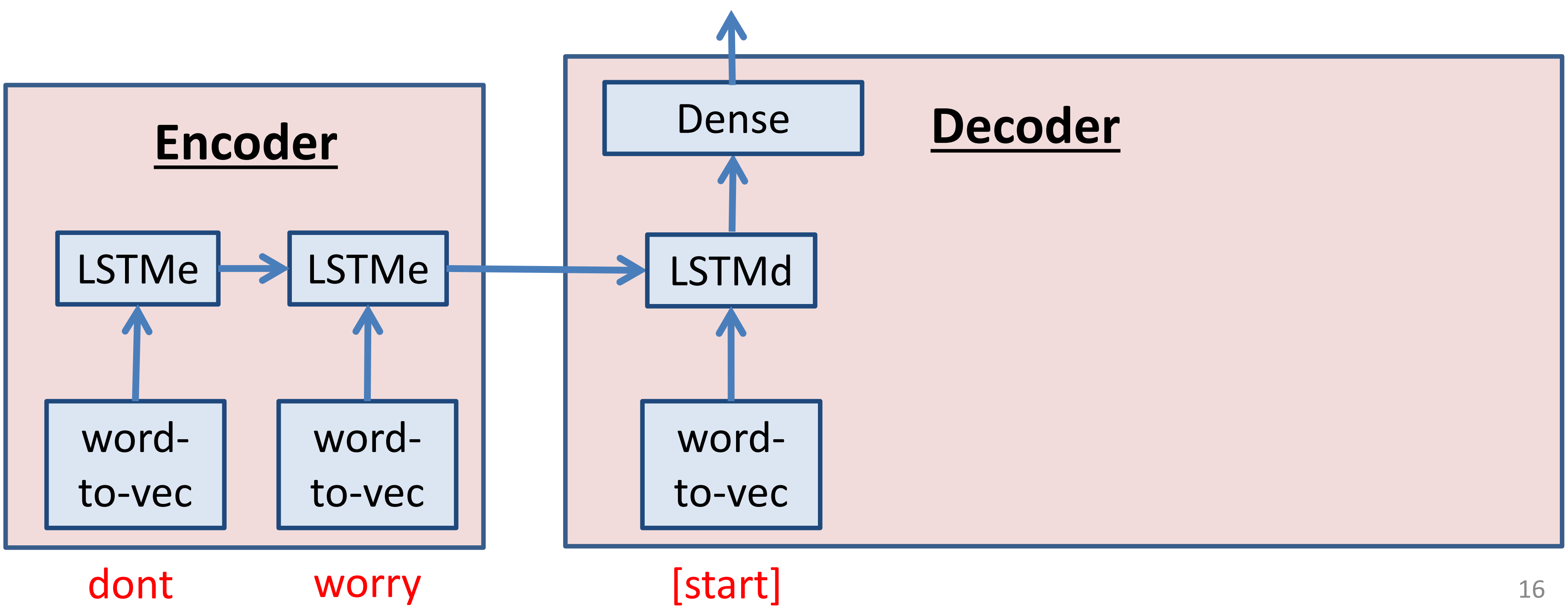
Encoder-Decoder Inference Process

- Input (standardized): “dont worry”
- Desired output: “[start] no te preoccupes [end]”.
- For now, we will be discussing the inference process.
  - We assume the model has already been trained.
  - For simplicity, we assume it will produce the correct output.



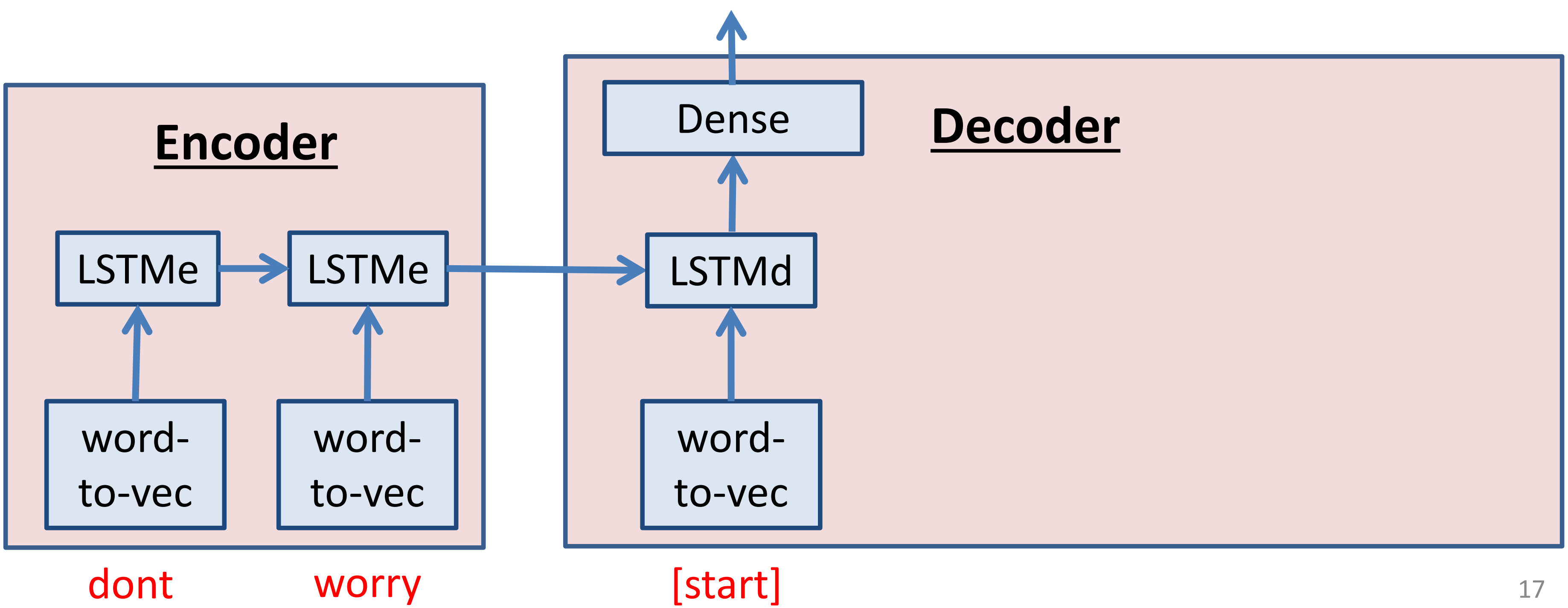
Encoder-Decoder Inference Process

- Notation:
  - LSTMe is the LSTM block of the encoder.
  - LSTMd is the LSTM block of the decoder.
  - We use different notation to emphasize that these blocks are different, they do NOT share the same weights.



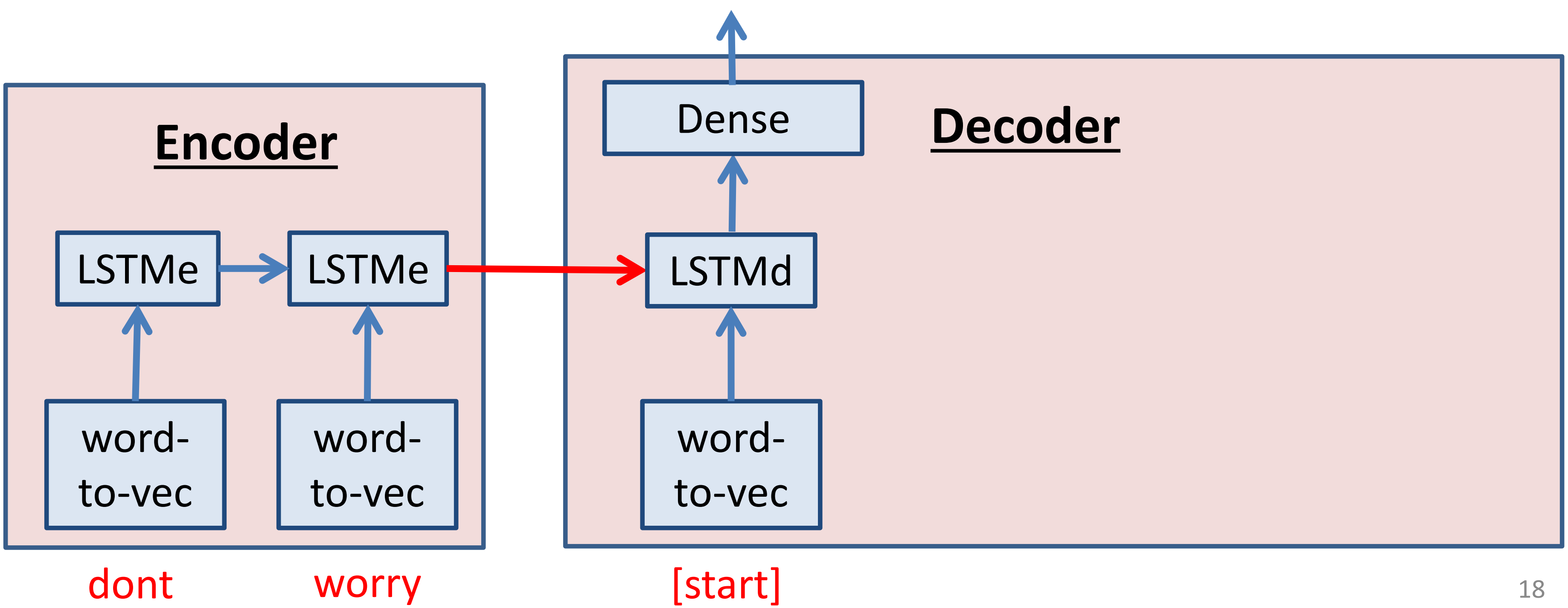
Inference Process: Encoding Part

- The encoder takes as input the words “don’t” and “worry”.
- First step: it processes the word “dont“.
- Second step: it processes the word “worry”.



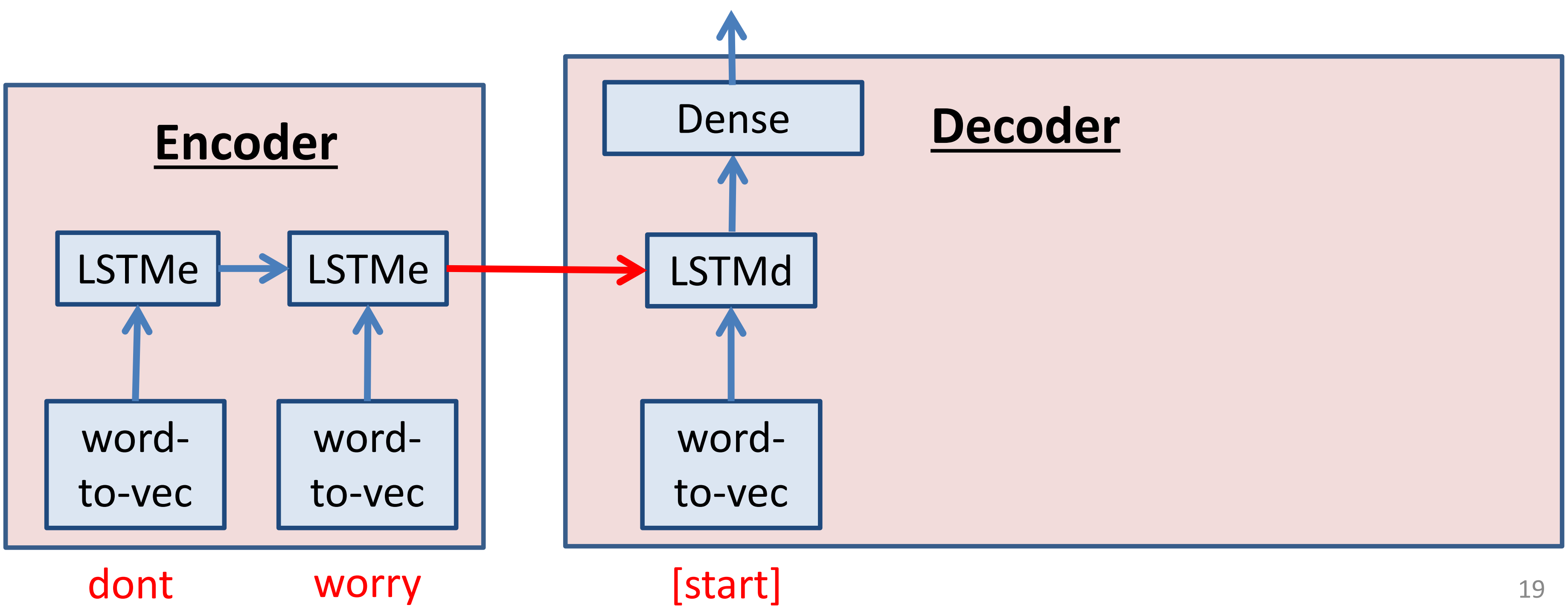
Inference Process: Encoding Part

- The output of LSTMe after the second step is passed as the **<u>recurrent input</u>** to LSTMd for the first step of the decoder.
  - This is something we have not seen before.
  - In our previous models, what was the recurrent input in the first step?



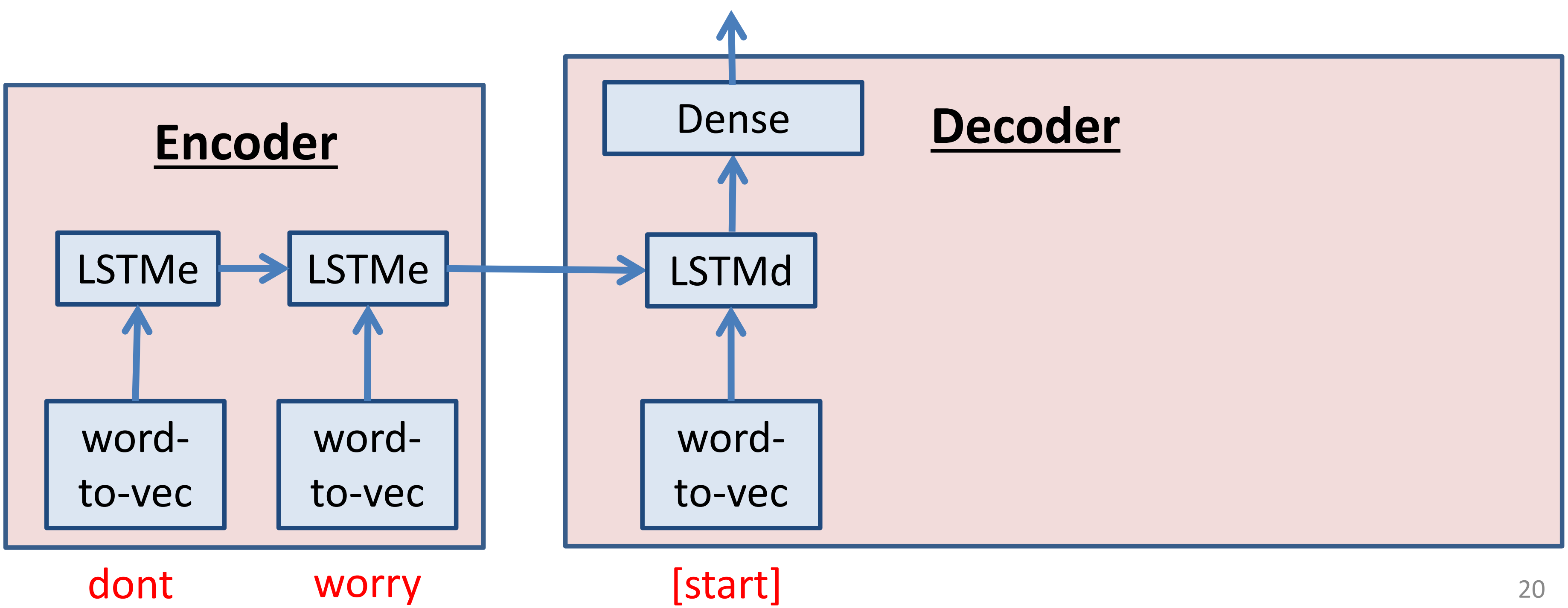
Inference Process: Encoding Part

- The output of LSTMe after the second step is passed as the **recurrent input** to LSTMd for the first step of the decoder.
- This output tells the decoder what it needs to know to produce the translation.



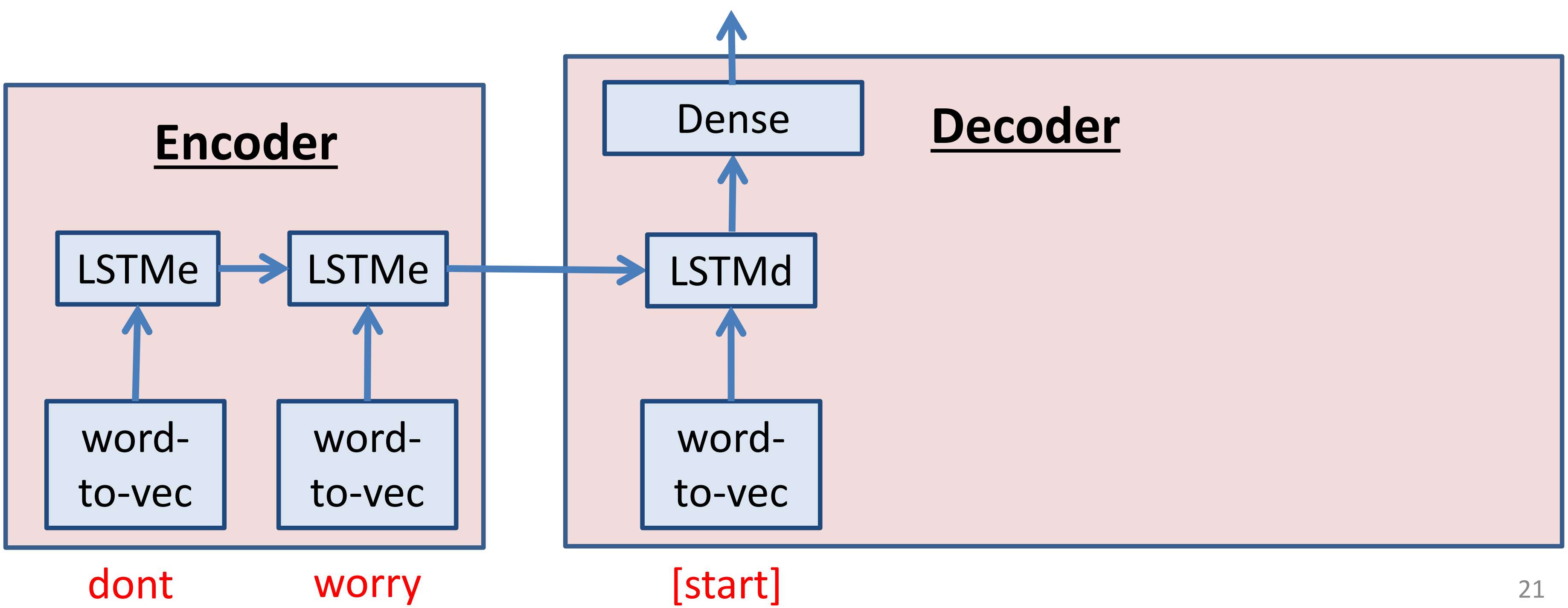
Inference Process: Encoding Part

- The output of LSTMe after the second step is passed as the **recurrent input** to LSTMd for the first step of the decoder.
- The encoder is now done, we move to the decoder.



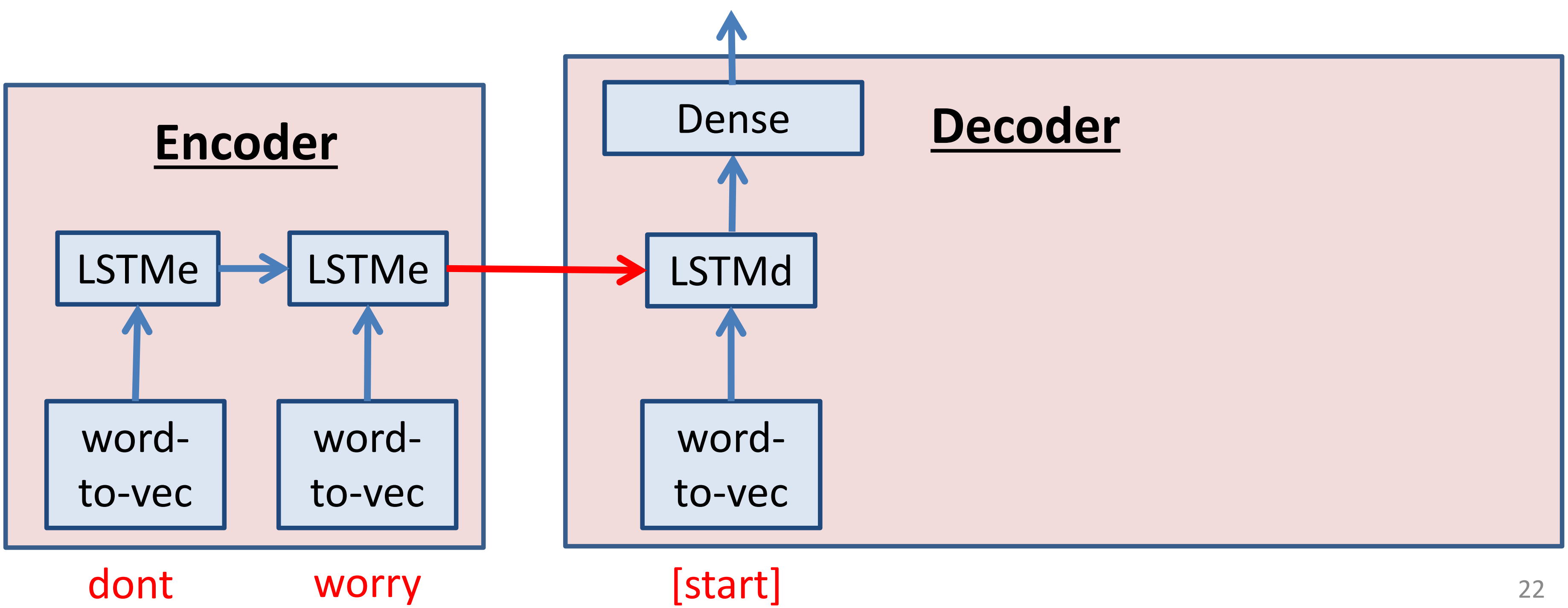
Inference Process: Decoding Part

- At the first step, the decoder receives two types of inputs:
- At its input layer, it receives the special token [start].
  - This token tells the decoder that we want it to output the first word of the translation text.
- How does the decoder know anything about the input text?



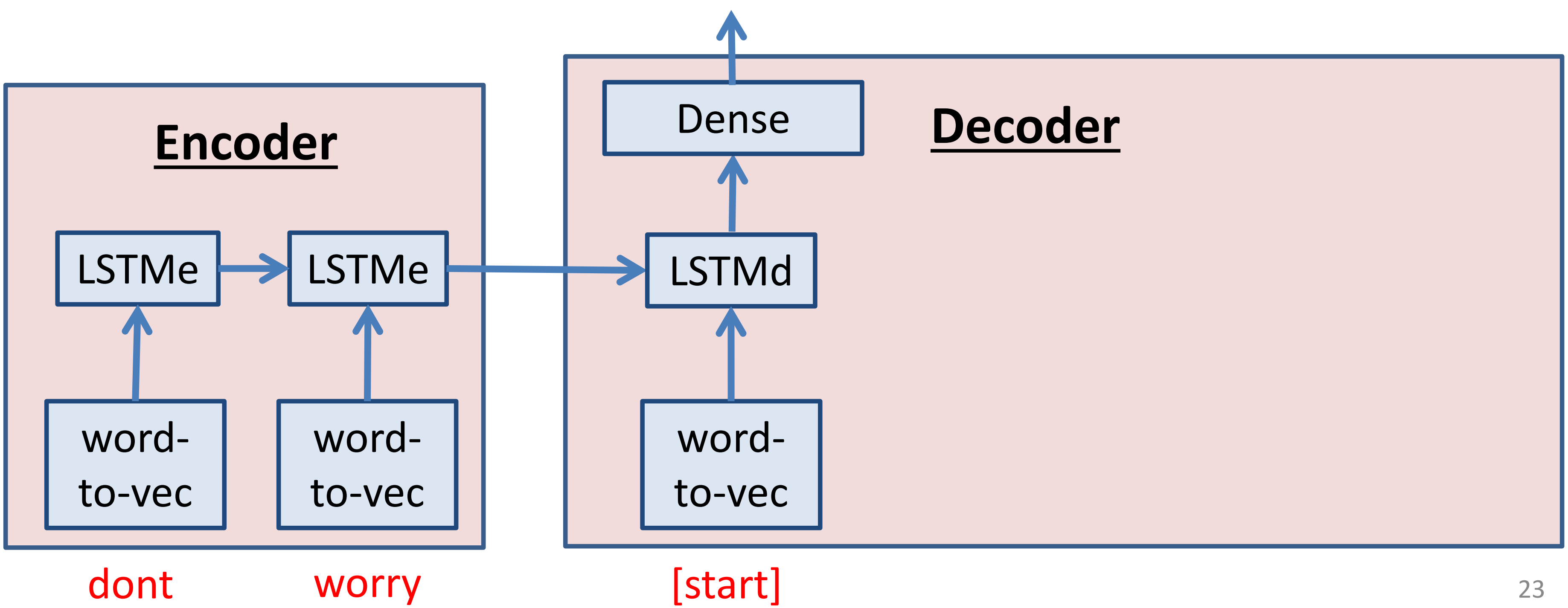
Inference Process: Decoding Part

- At the first step, the decoder receives two types of inputs:
- At its input layer, it receives the special token [start].
- The initial recurrent input comes from the encoder.
  – As we said before, this is how the decoder knows what to translate.



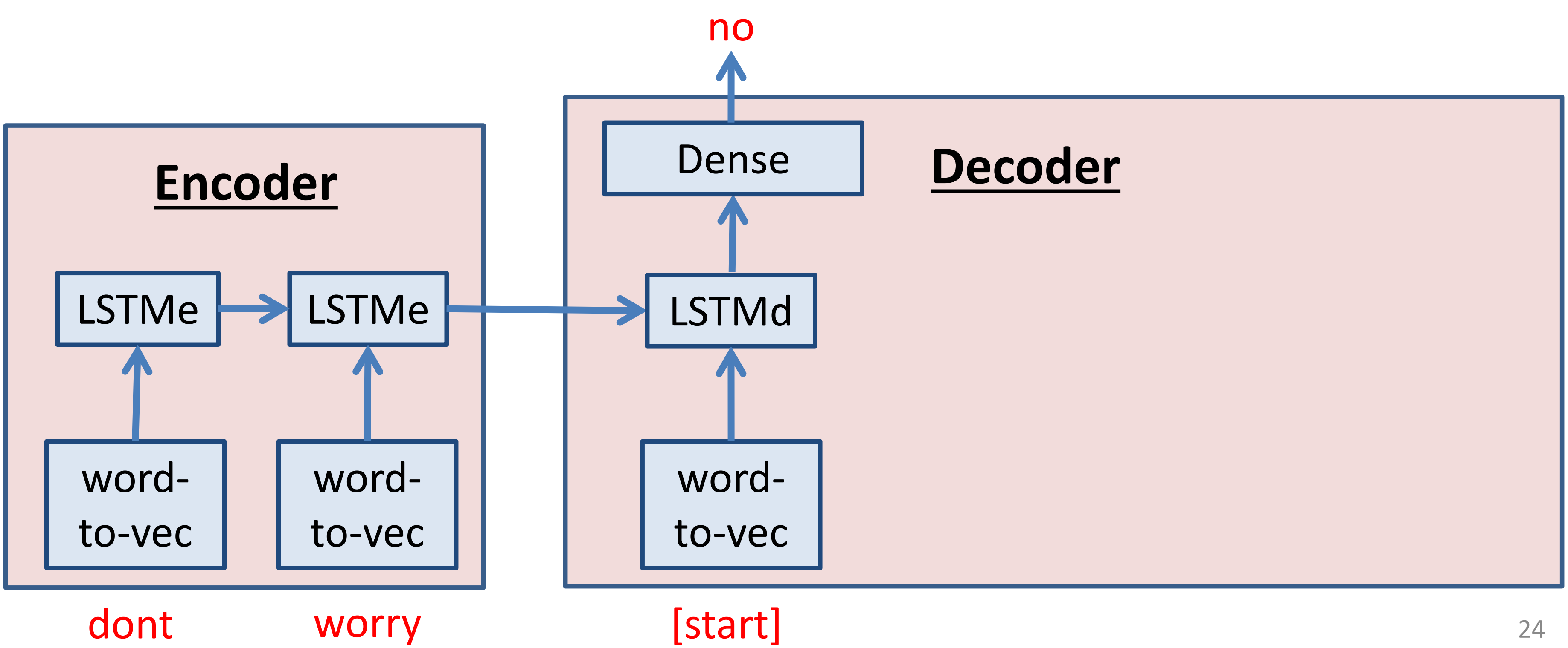
Inference Process: Decoding Part

- The [start] token gets mapped to a vector (word embedding).
- The LSTM block processes its two inputs (word embedding and recurrent input), and passes its output to the Dense layer.
- Then the Dense layer computes its output.



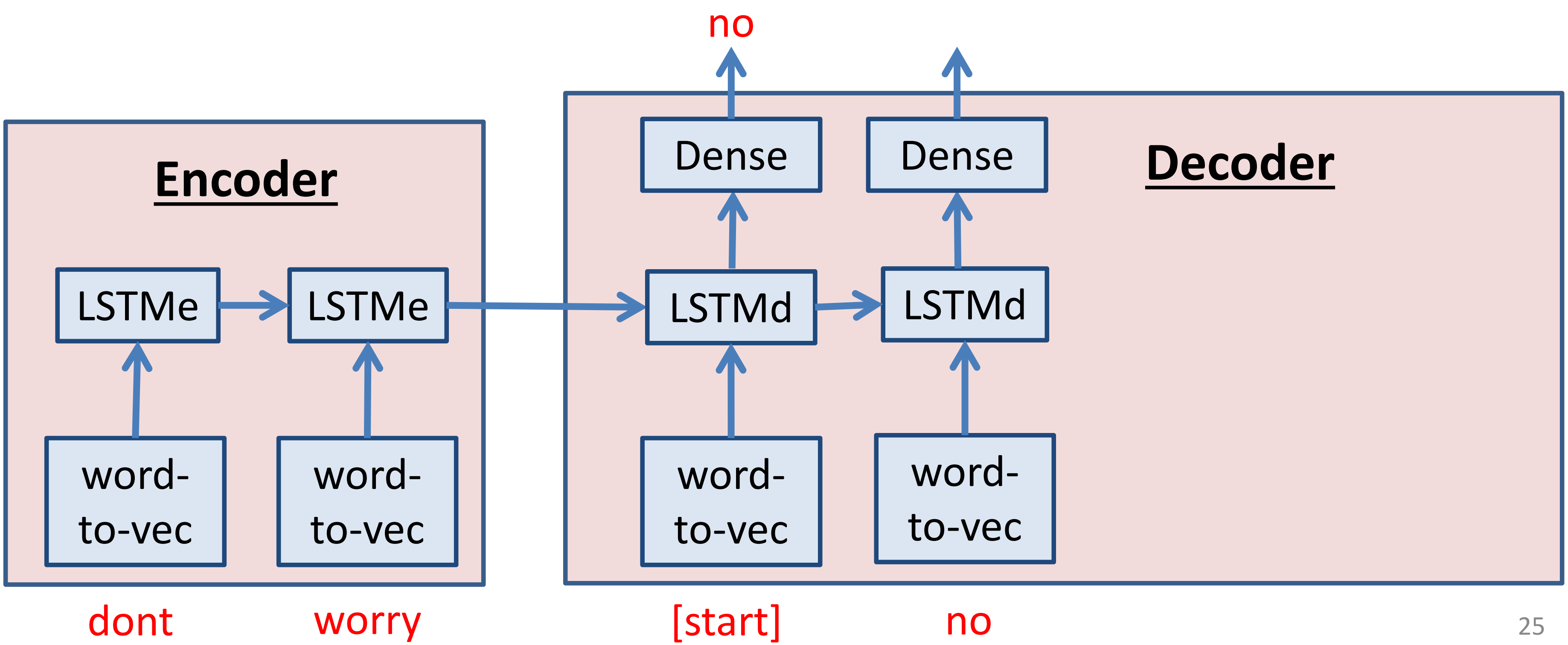
Inference Process: Decoding Part

- The argmax of the Dense layer's output corresponds to the Spanish word "no".
- So, the output of the decoder is the word "no".
- What next?



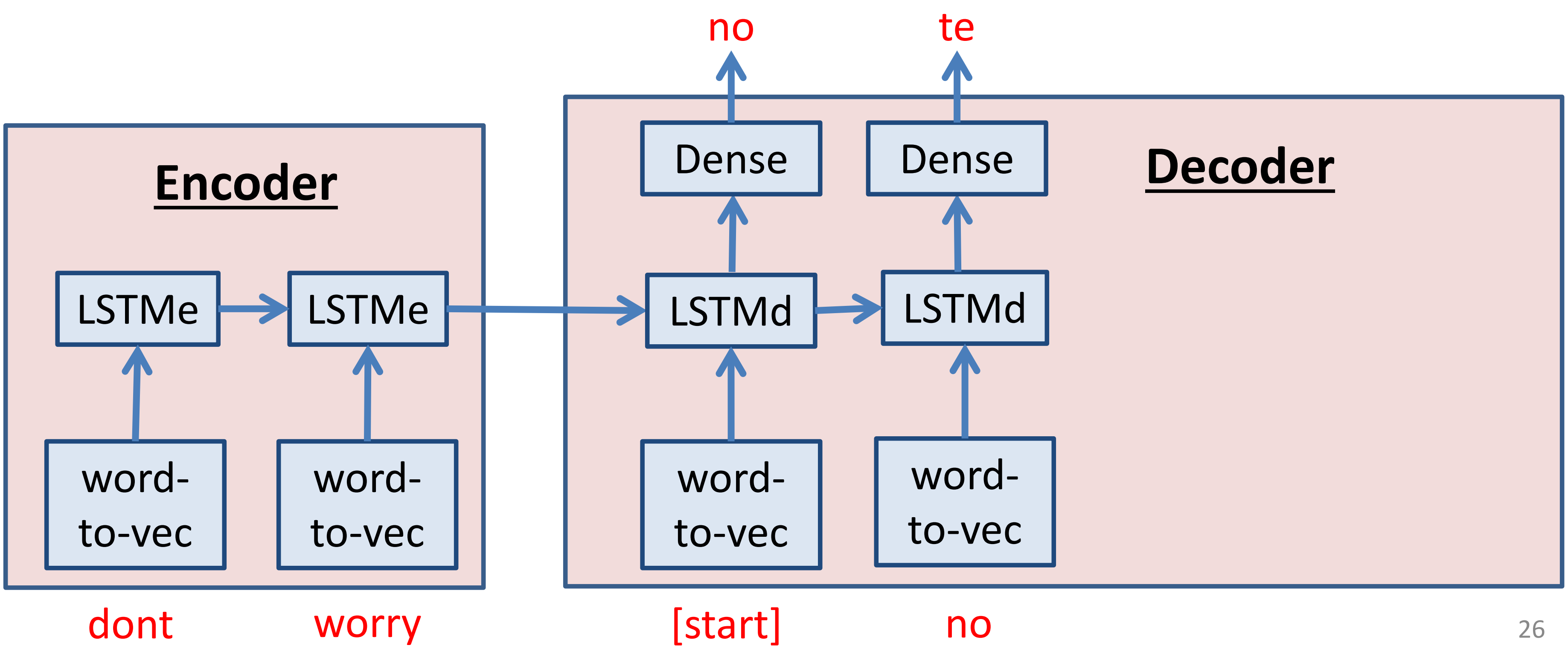
Inference Process: Decoding Part

- The decoder now needs to produce the second word of the translation.
- To do that, we take the word that the decoder just produced, and **<u>we use it as the next input to the decoder</u>**.
- This step is the key to understanding how the decoder works!!!
- So, the decoder processes "no" as its second input.



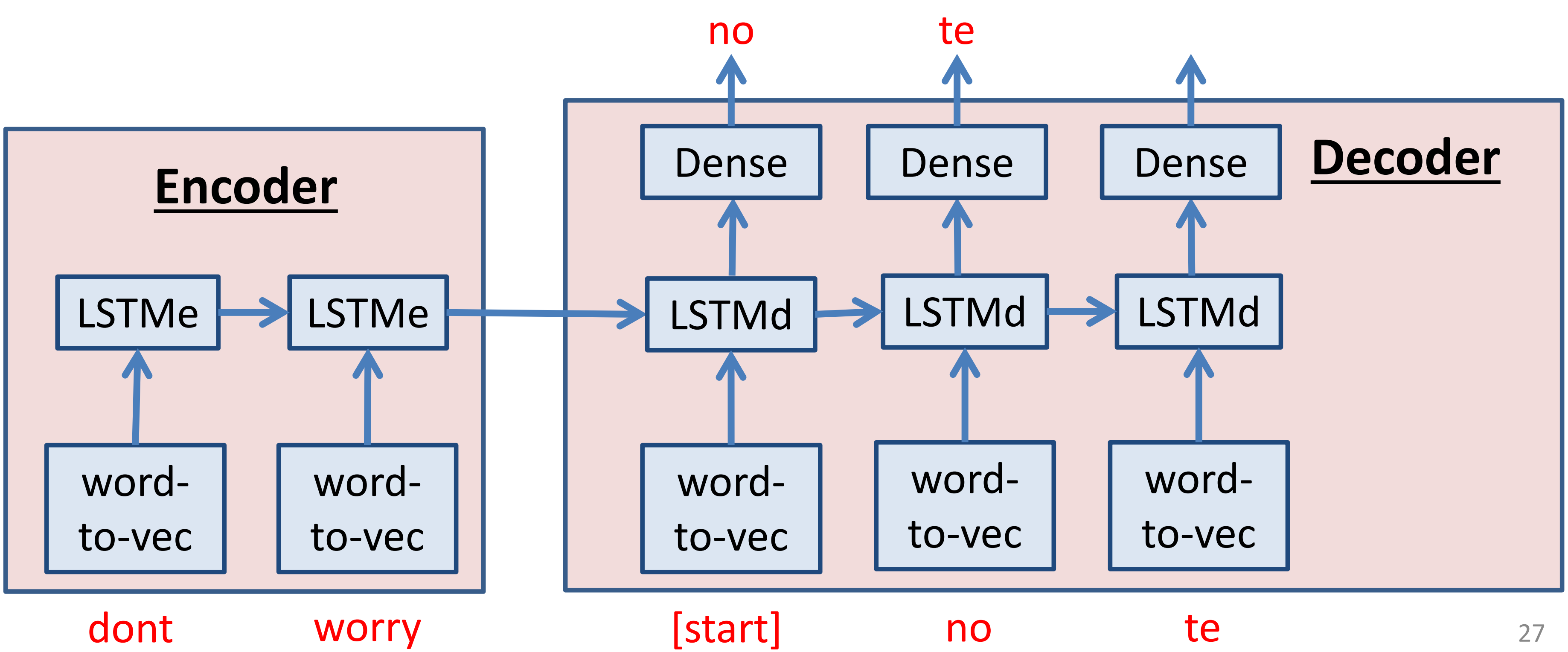
Inference Process: Decoding Part

- The decoder now needs to produce the second word of the translation.
- To do that, we take the word that the decoder just produced, and we use it as the next input to the decoder.
- So, the decoder processes “no” as its second input.
- The output of the second step is “te”.



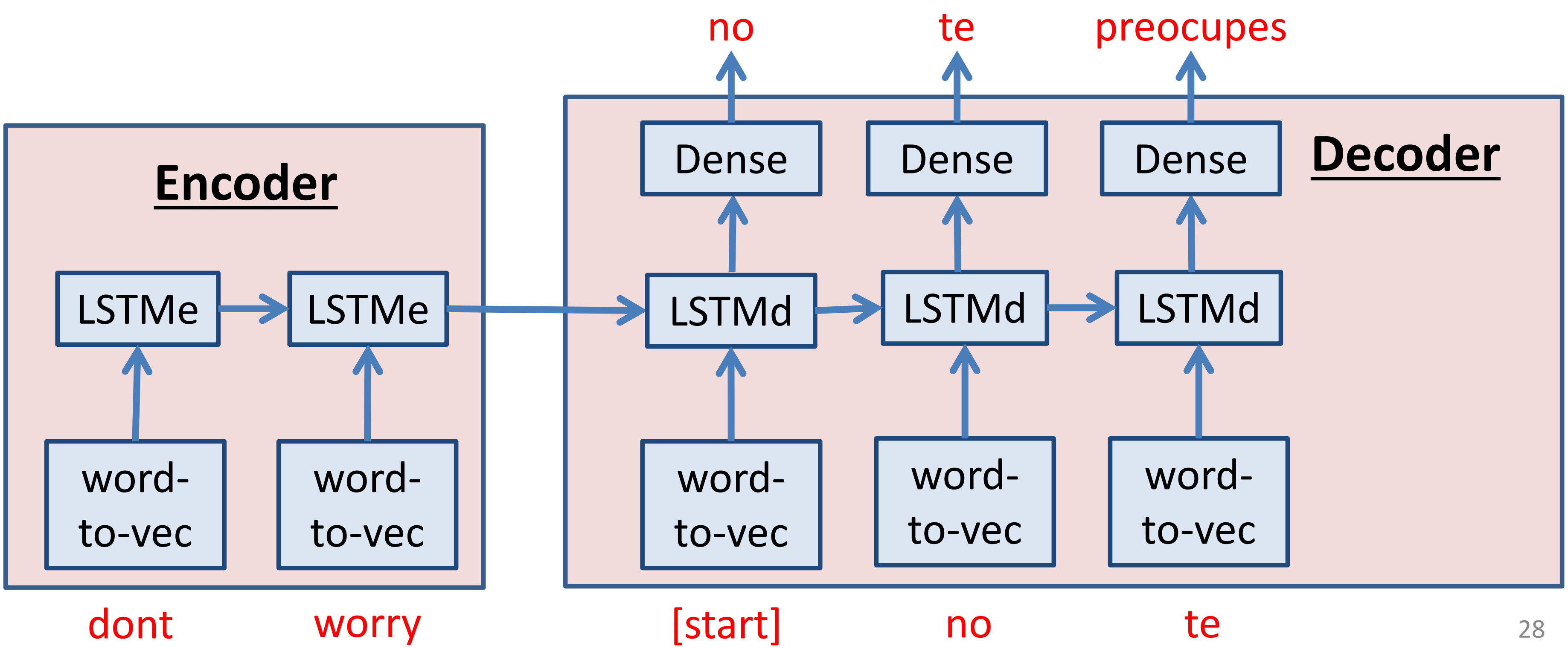
Inference Process: Decoding Part

- Next: produce the third word of the translation.
- Again, we take the word that the decoder just produced, and **we use it as the next input to the decoder**.
- So, the decoder processes “te” as its third input.



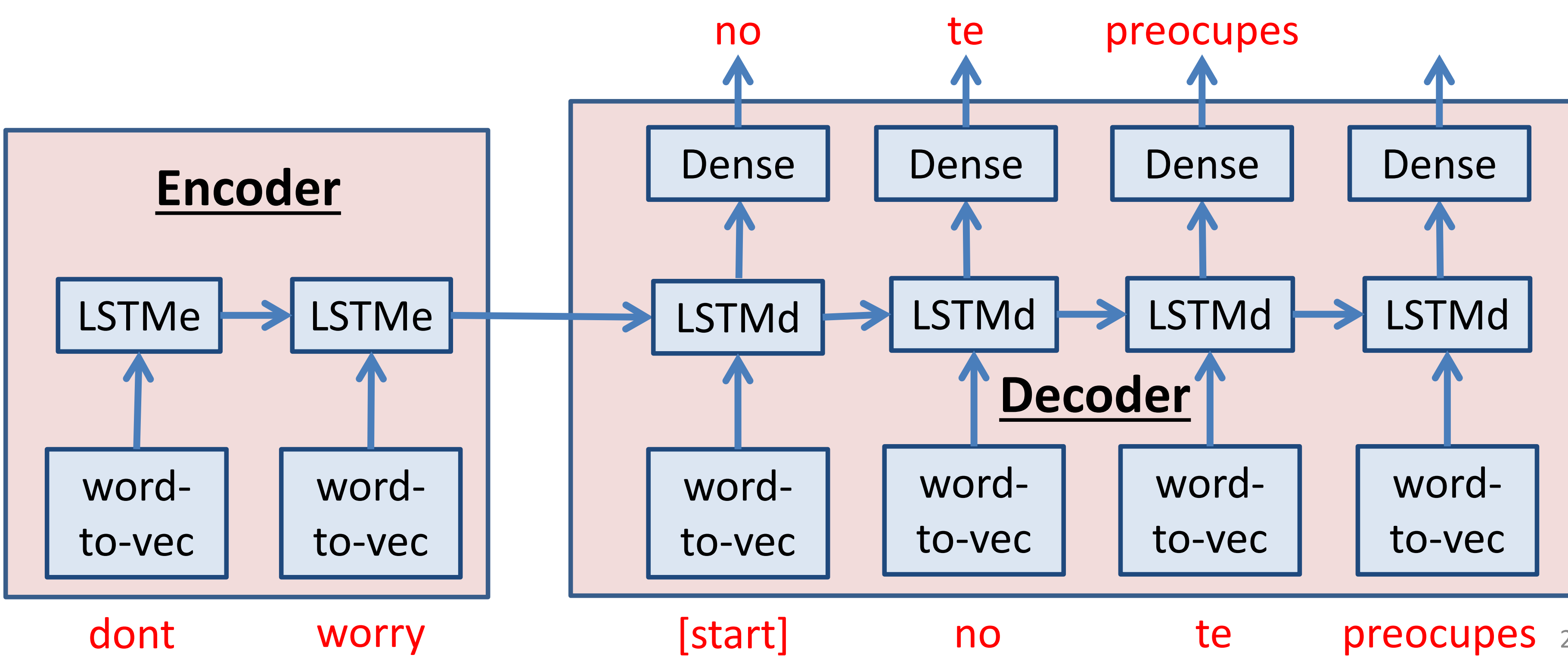
Inference Process: Decoding Part

- Next: produce the third word of the translation.
- Again, we take the word that the decoder just produced, and we use it as the next input to the decoder.
- So, the decoder processes "te" as its third input.
- The output of the third step is "preocupes".



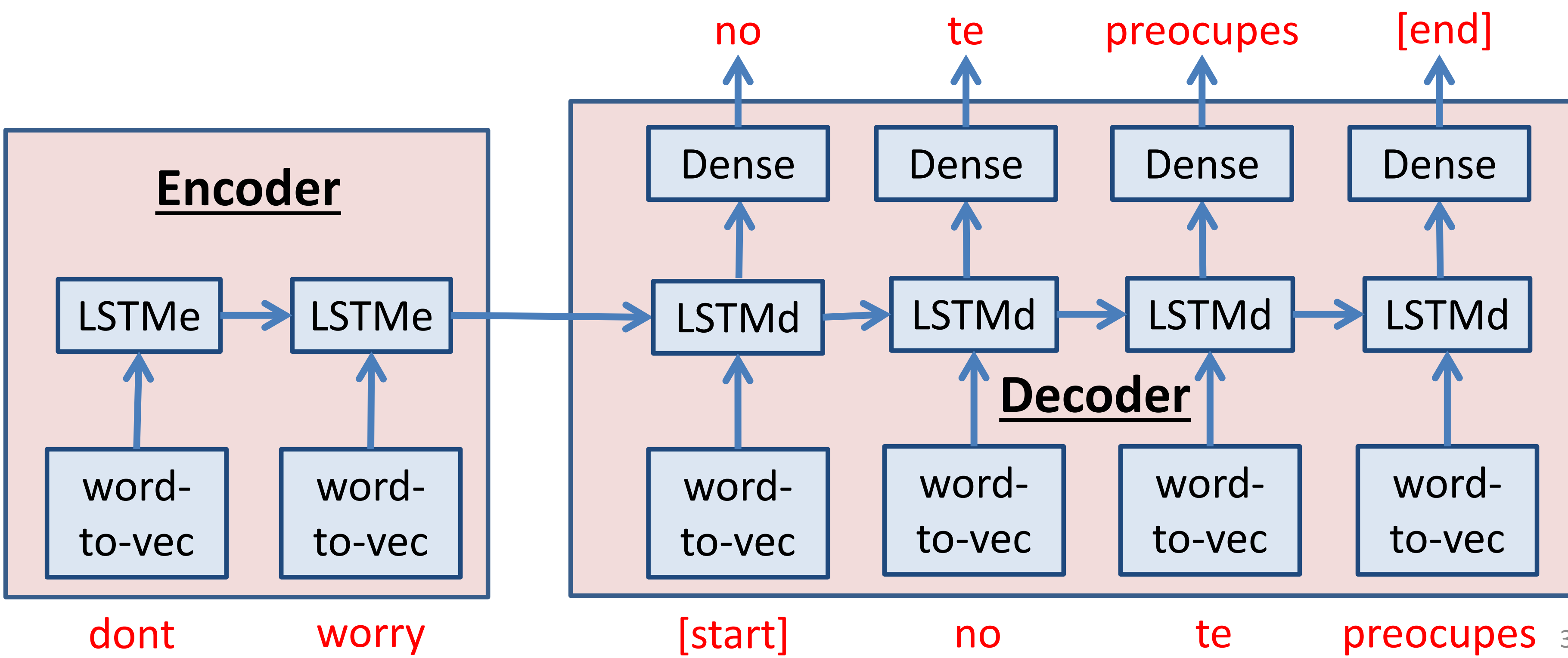
Inference Process: Decoding Part

- Next: produce the fourth word of the translation.
- Again, we take the word that the decoder just produced, and we use it as the next input to the decoder.
- So, the decoder processes "preocupes" as its fourth input.



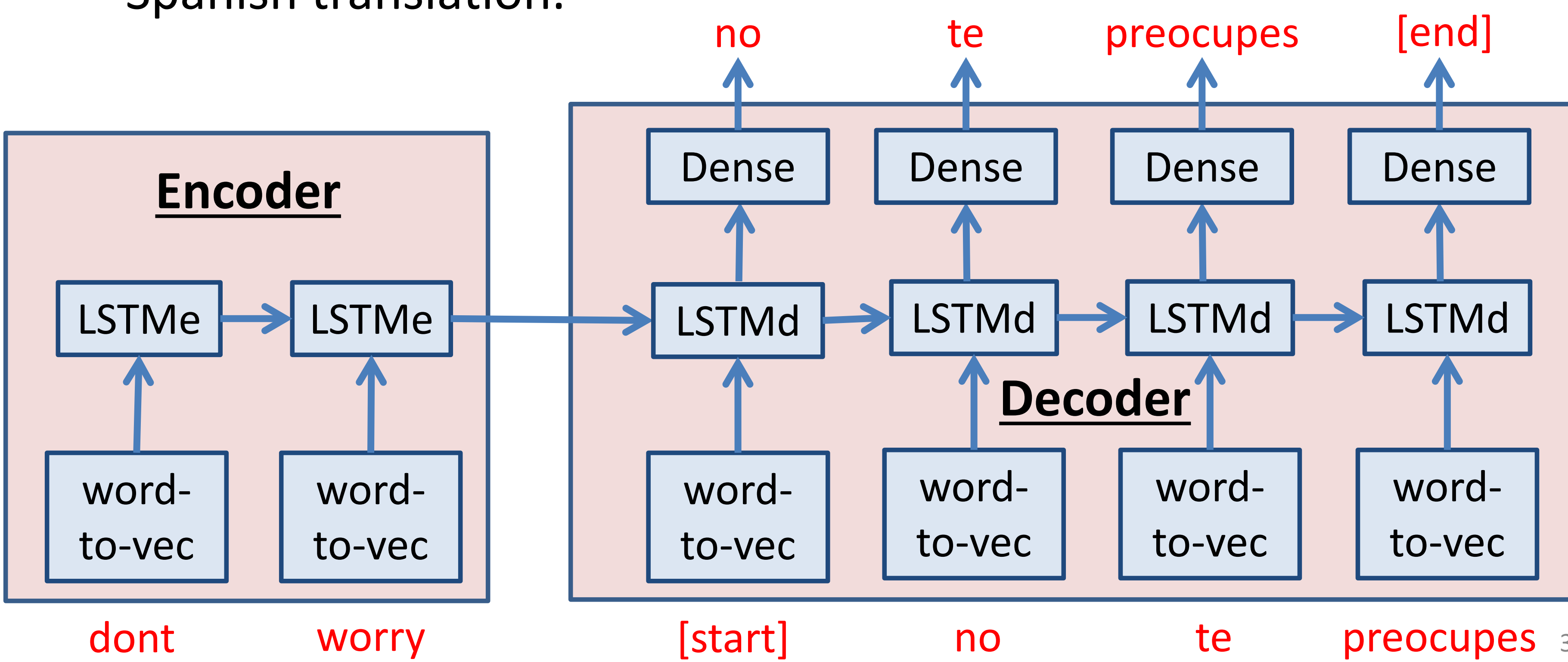
Inference Process: Decoding Part

- Next: produce the fourth word of the translation.
- Again, we take the word that the decoder just produced, and we use it as the next input to the decoder.
- So, the decoder processes "preocupes" as its fourth input.
- The output of the fourth step is [end].



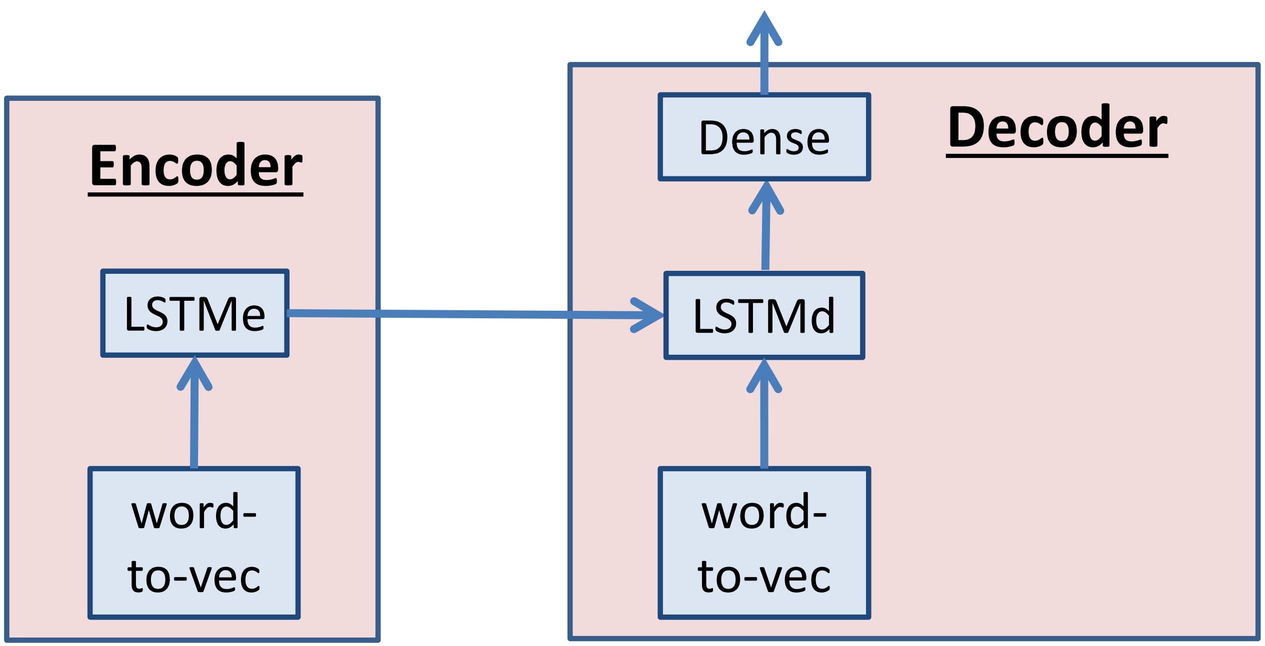
Inference Process: Decoding Part

- Like [start], [end] is a special token.
  - Token [end] is simply the decoder's way to tell us that it has finished the translation.
- The decoder output up to and not including [end] is the actual Spanish translation.



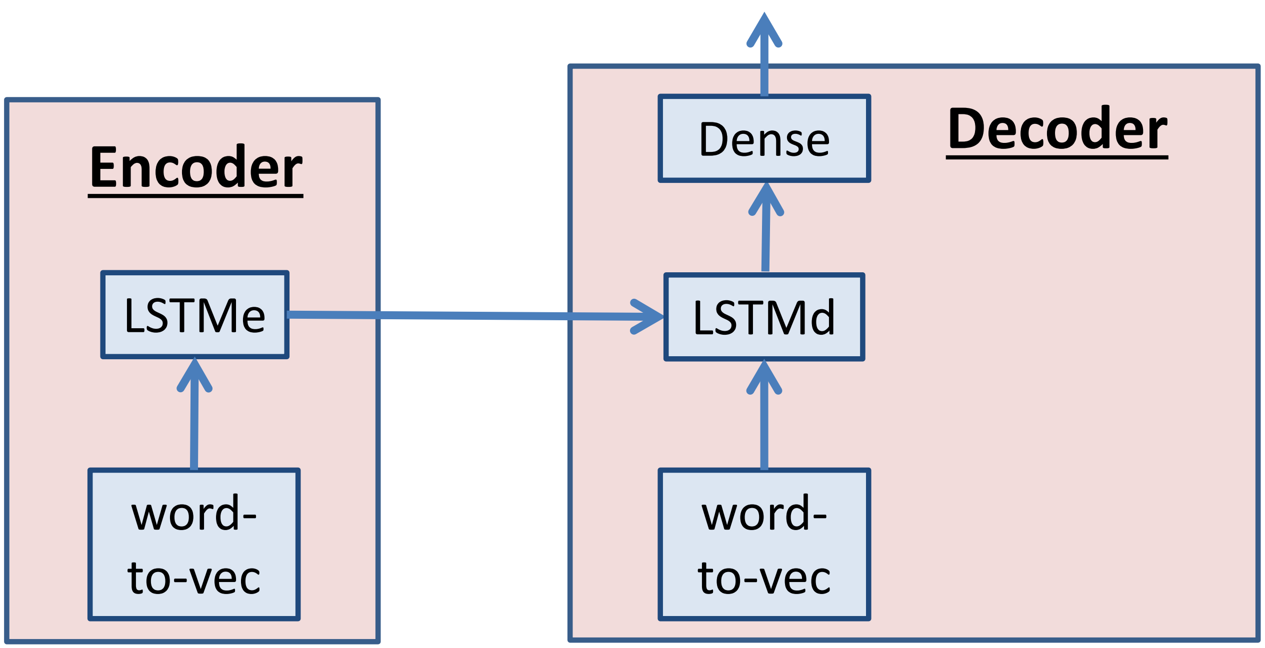
Encoder-Decoder Architecture

- In the previous slides, we drew separately the model blocks that process each time step.
- However, these blocks are all identical, and use the same weights.
- We can simplify the drawing by avoiding the replications.



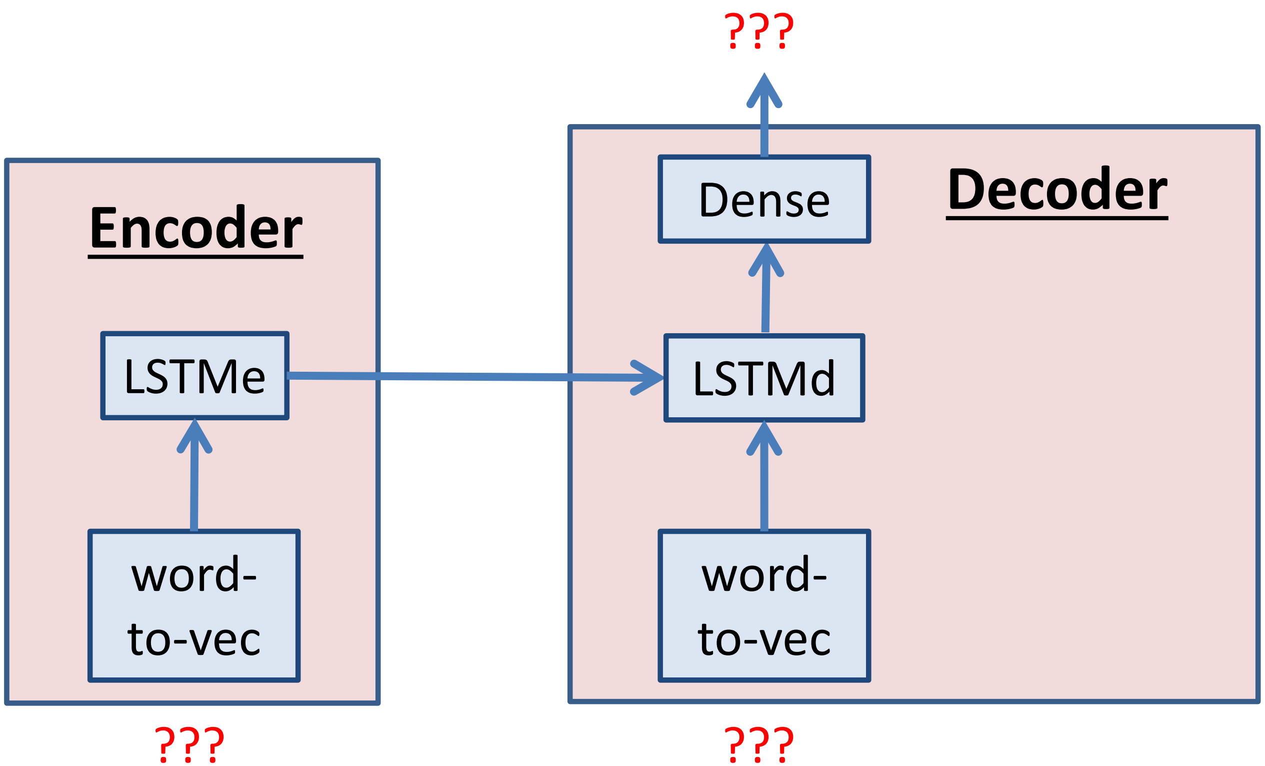
Training: Inputs and Targets

- So far we have discussed the inference process.
  - Assume the model has been trained.
  - See how it produces, step by step, the translation for some new input text.
- Now we will discuss the training process.



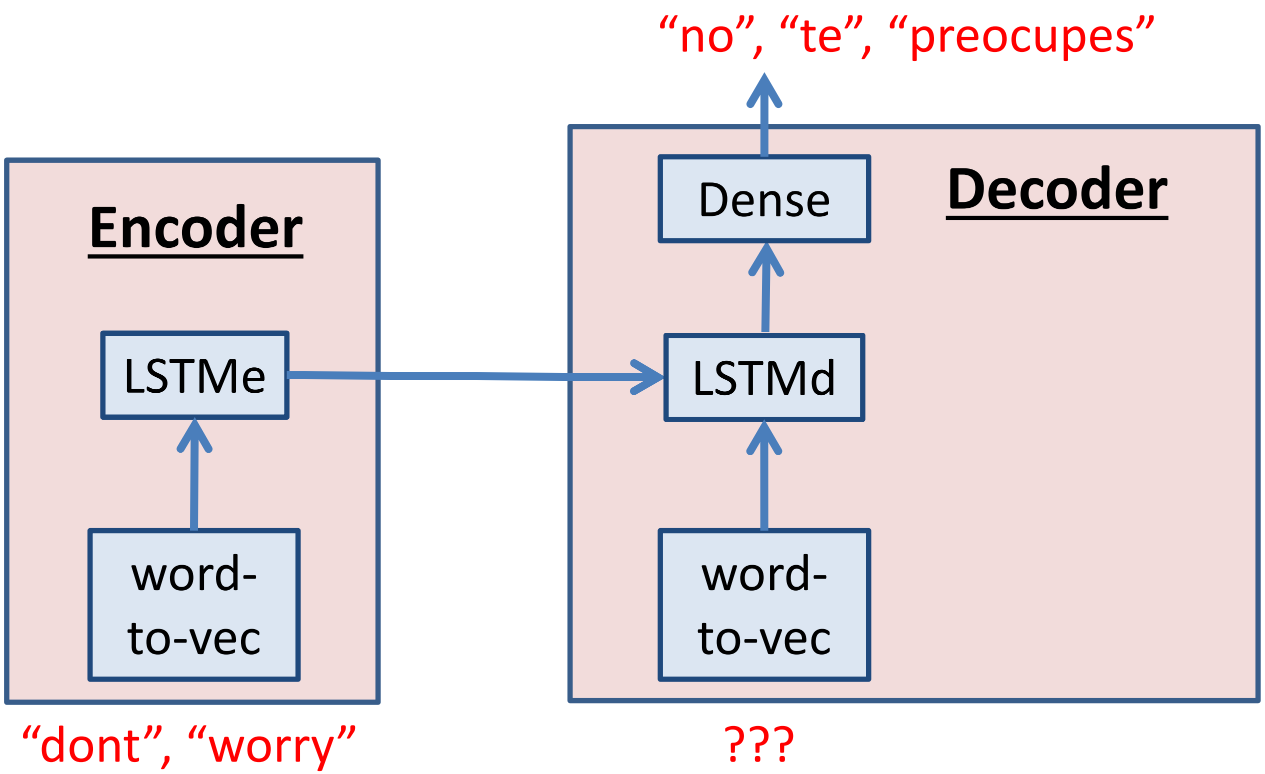
Training: Inputs and Targets

- Question: what are the training inputs, and what are the target outputs?



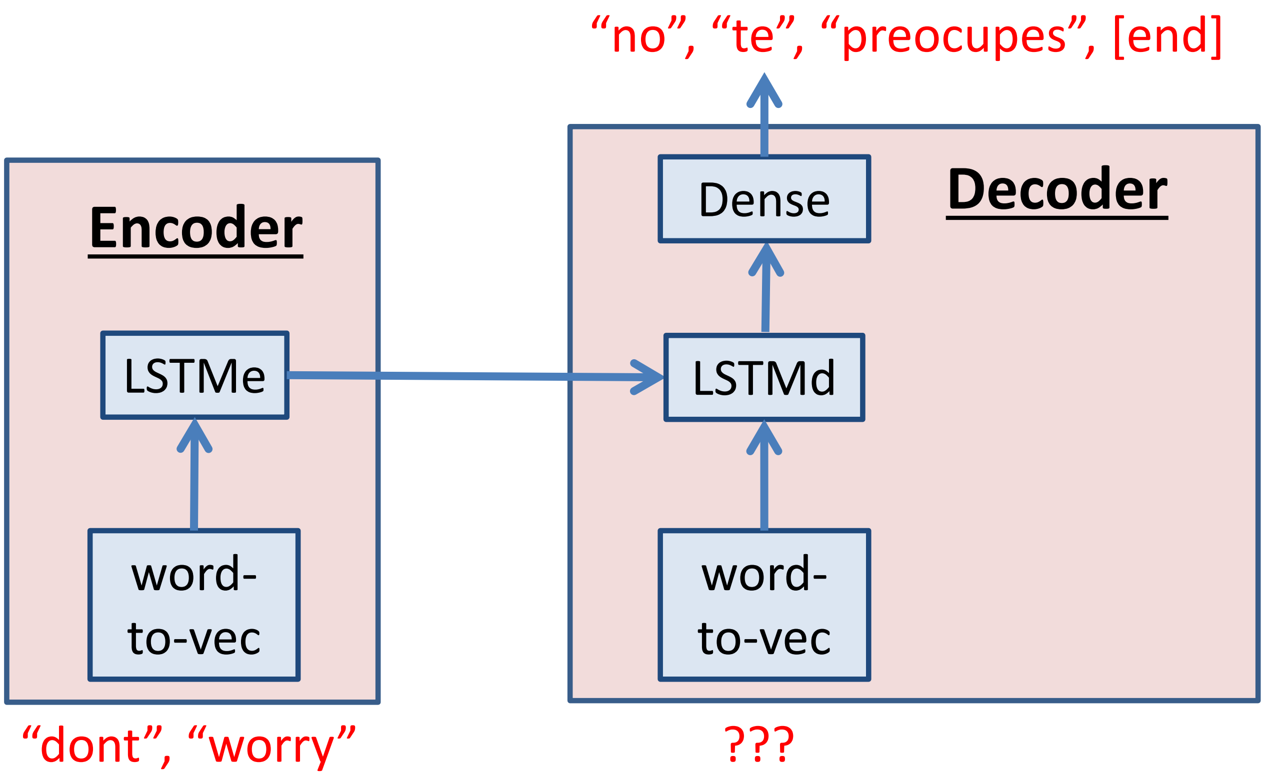
Training: Inputs and Targets

- Clearly, the English text is an input.
- Clearly, the Spanish translation is an output.
- However, this is not the complete picture.



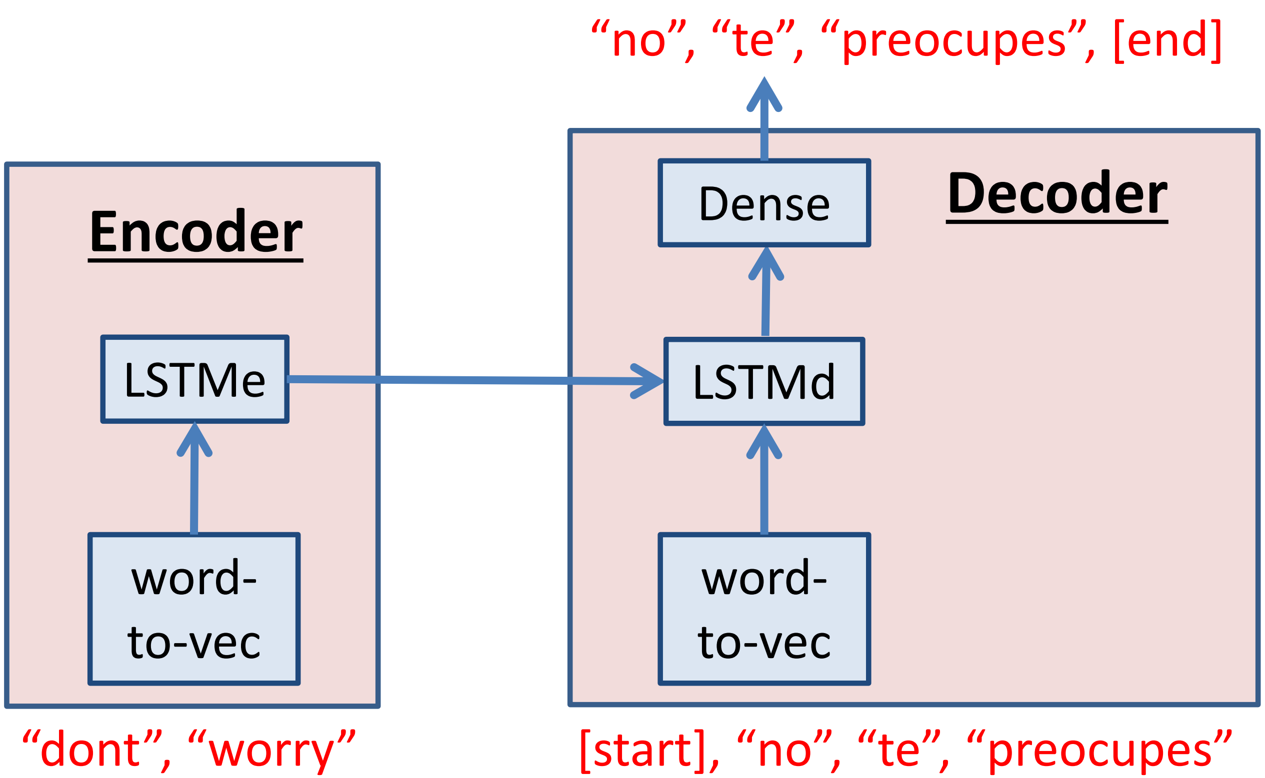
Training: Inputs and Targets

- One easy thing first: the model needs to know when to stop decoding.
- So, the target output should always end with the special token [end].



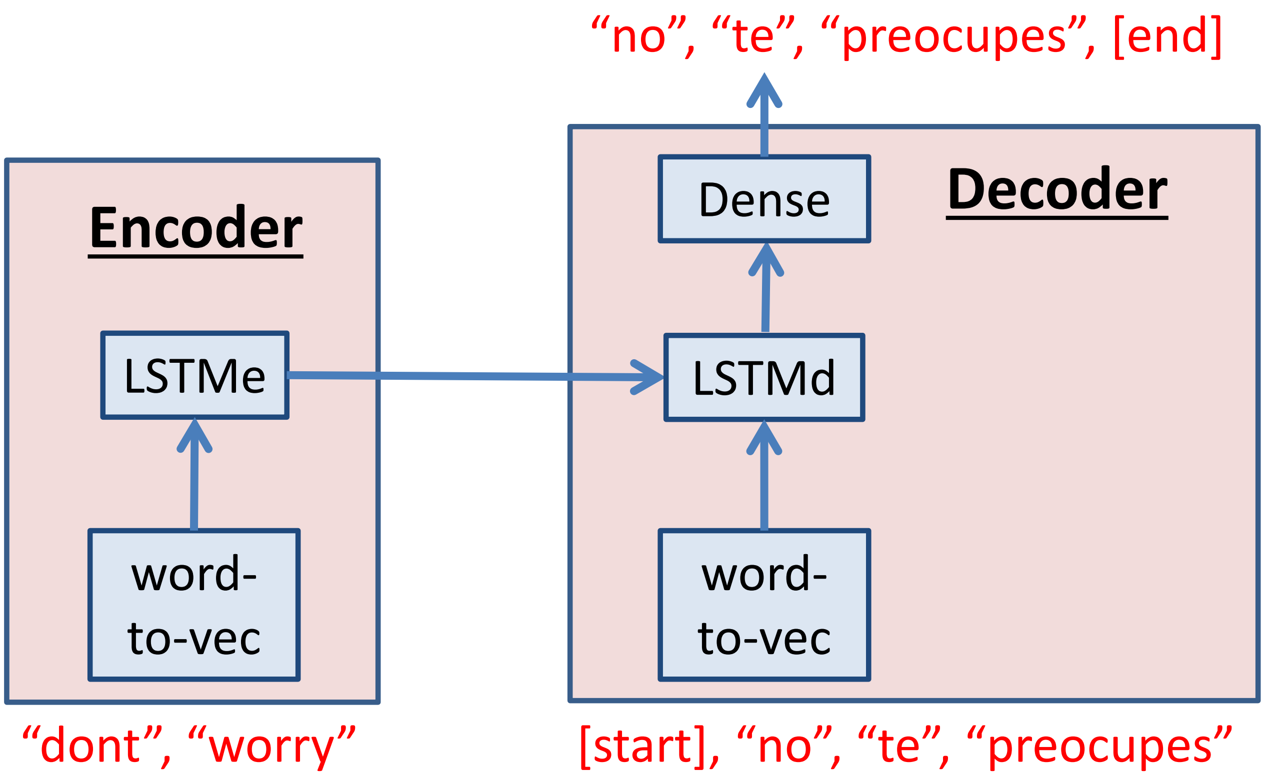
Training: Inputs and Targets

- Now, the confusing part: the output of the decoder is (sort of) also input to the decoder.
  - We said that, at inference time, the output of the decoder at one step is used as input to the next step.



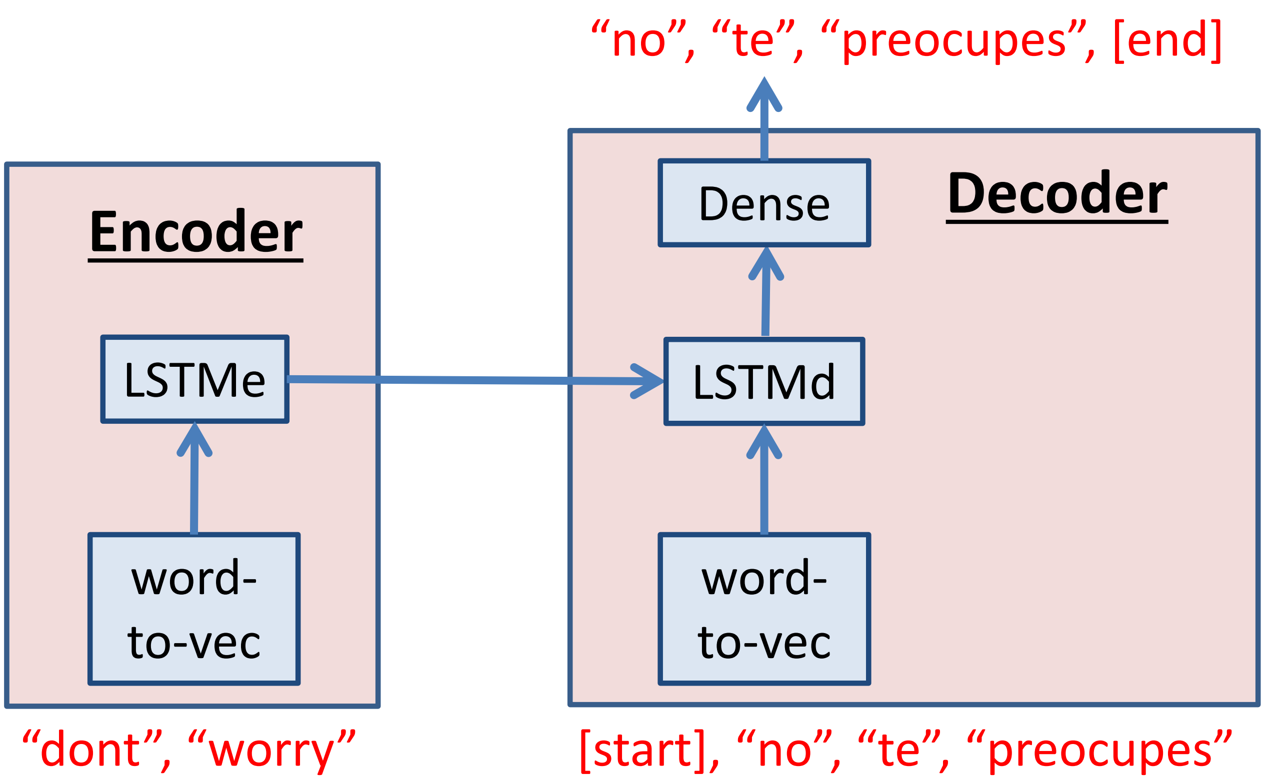
Training: Inputs and Targets

- There is one key difference between the decoder input and the decoder output:
  - The decoder input starts with [start] and ends with the last Spanish word.
  - The target output starts with the first Spanish word and ends with [end].



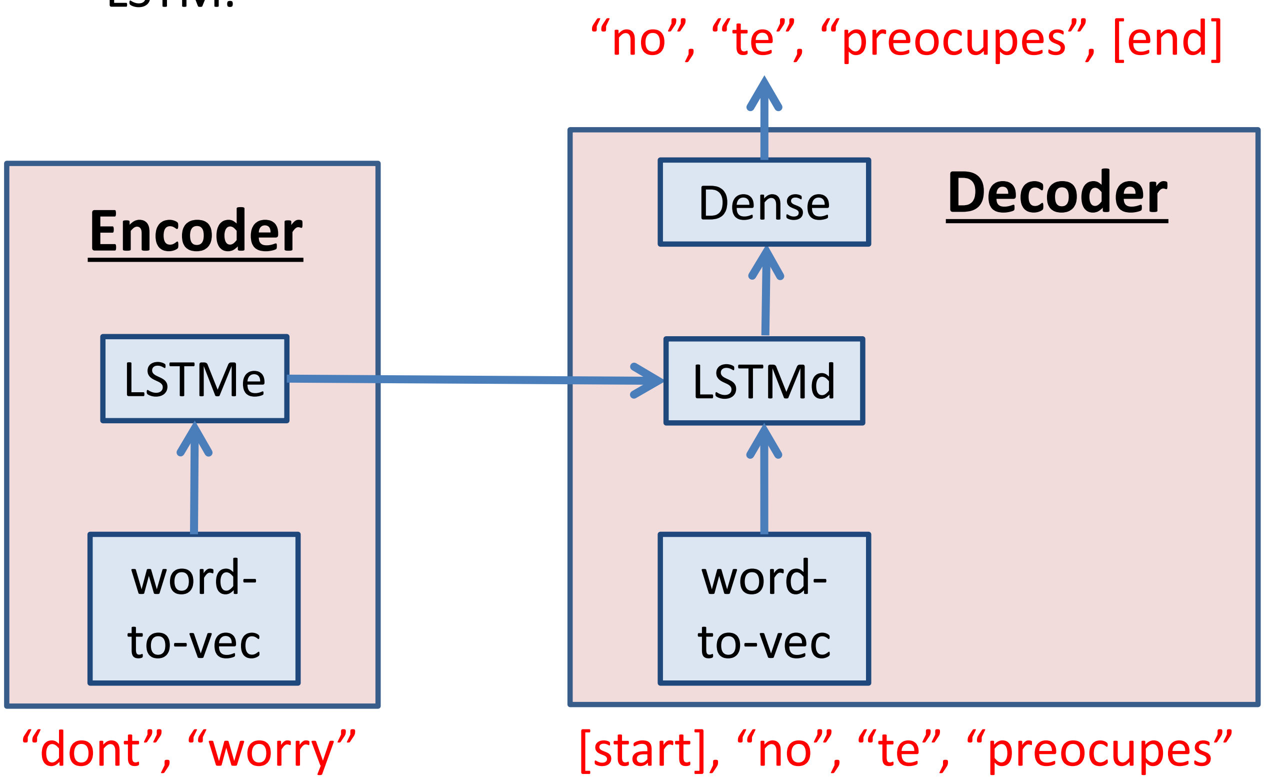
Training: Inputs and Targets

- Consequently, the position of a Spanish word in the target output sequence is **always** one time step before the position of the same word in the decoder input sequence.



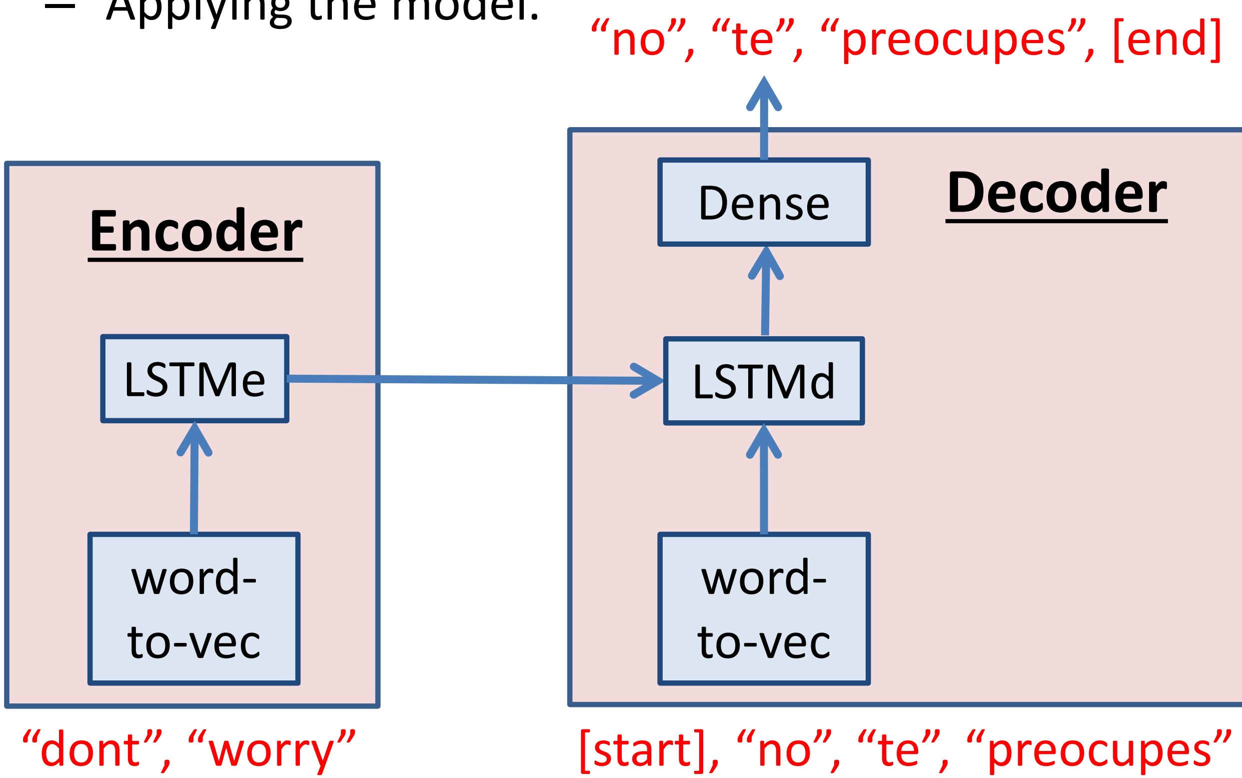
Implementing in Keras

- How do we implement this in Keras? How do we tell Keras that:
  - Some input goes to the encoder.
  - Some input goes to the decoder.
  - The last output of the encoder's LSTM is the first input to the decoder's LSTM.



Implementing in Keras

- We will look at the Keras implementation step by step.
  - Loading the original dataset file, and creating Tensorflow datasets.
  - Specifying the model.
  - Training the model.
  - Applying the model.



The Original Data

- The original file has 118,964 lines of text.
- Each line has the following format:
  - A sentence in English.
  - The TAB (“\t”) character.
  - A translation of the English sentence to Spanish.
- Some lines close to the beginning of the file:

```
I quit.    Renuncié.
I work.    Estoy trabajando.
I'm 19.    Tengo diecinueve.
I'm up.    Estoy levantado.
```

- The initial lines are pretty short, but they get longer.
  - Last line: 47-word English sentence, 51-word Spanish translation.

Reading the File

```
text_file = "spa-eng/spa.txt"
with open(text_file, encoding='utf-8') as f:
    lines = f.read().split("\n")[:-1]
text_pairs = []

for line in lines:
    english, spanish = line.split("\t")
    spanish = "[start] " + spanish + " [end]"
    text_pairs.append((english, spanish))
```

- We separate the file content into lines.
- We separate each line into an English part and a Spanish part.
- **NOTE:** we add the [start] and [end tokens] to the Spanish part.

Training, Validation, Test Data

```
random.shuffle(text_pairs)
num_val_samples = int(0.15 * len(text_pairs))
num_train_samples = len(text_pairs) - 2 * num_val_samples
train_pairs = text_pairs[:num_train_samples]
val_pairs = text_pairs[num_train_samples:num_train_samples + num_val_samples]
test_pairs = text_pairs[num_train_samples + num_val_samples:]
```

- This part of the code:
  - Randomizes the order of the text lines.
  - Splits the data into:
    - Training set: 70% of the examples.
    - Validation set: 15% of the examples.
    - Test set: 15% of the examples.

Text Standardization

- Next, we need to define the text vectorization layers.
- For both English and Spanish:
  – The vocabulary size will be 15,000
  – The output will be a sequence of integers.
  – No value is provided for ngrams, so the default value of 1 is used. Each token will be a single word.

- The text vectorization layer for English is as usual:

```
vocab_size = 15000
source_vectorization = layers.TextVectorization(max_tokens=vocab_size,
                                                output_mode="int")
```

Custom Text Standardization

- The text vectorization layer for Spanish needs to take into account that Spanish has the extra punctuation characters "¿" and "¡“, used at the beginning of questions and exclamations.
- Because of that, we define a customized standardization function.

```
strip_chars = string.punctuation + "¿¡"
strip_chars = strip_chars.replace("[", "")
strip_chars = strip_chars.replace("]", "")

def custom_standardization(input_string):
    lowercase = tf.strings.lower(input_string)
    return tf.strings.regex_replace(
        lowercase, f"[{re.escape(strip_chars)}]", "")
```

Custom Text Standardization

- This code creates the text vectorization layer for Spanish.
- Notice how we tell it to use the **custom_standardization** function that we defined in the previous slide.
- This is an example of how many aspects of Keras modules can be customized as needed.

```
target_vectorization = layers.TextVectorization(max_tokens=vocab_size,
                                                output_mode="int",
                                                standardize=custom_standardization)
```

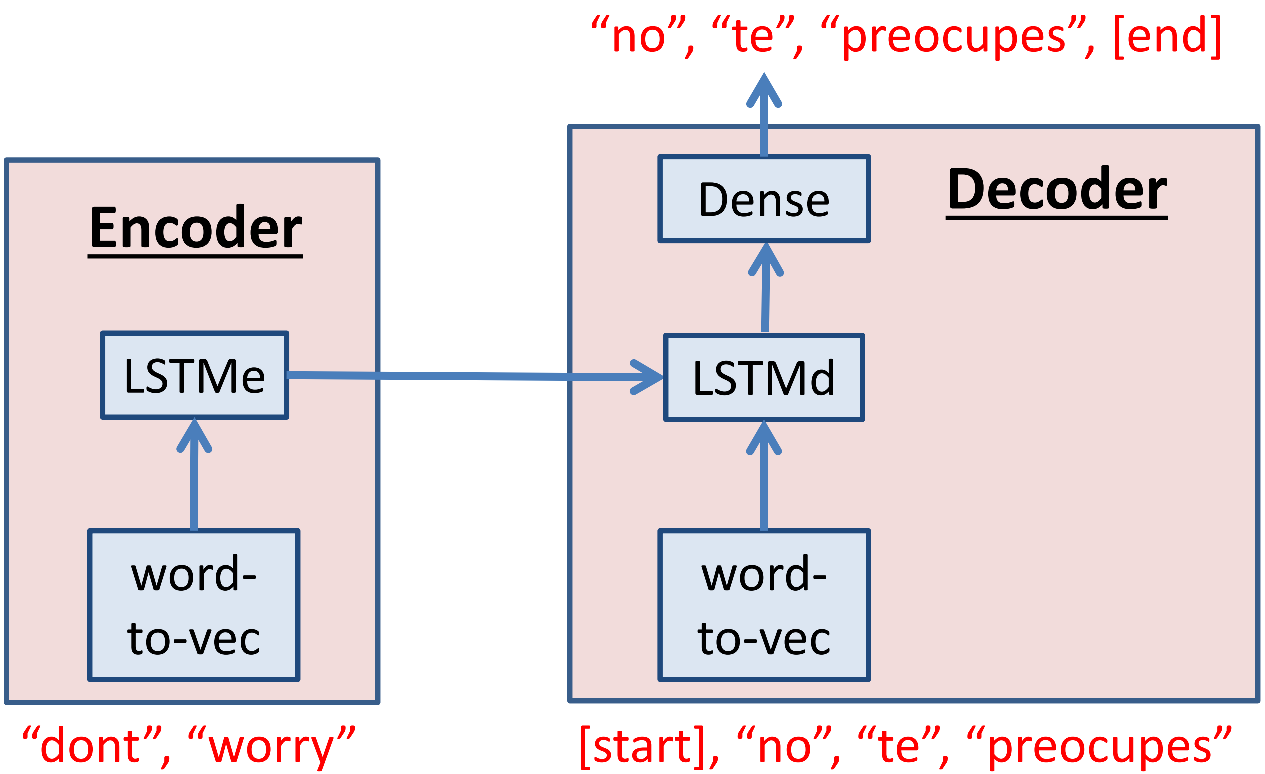
Computing the Vocabularies

- Here, each text vectorization layer computes its vocabulary.
  - We call the **adapt** method, which we have used the same way before.
- So far, our training data is in the **train_pairs** variable.
  - Element **train_pairs[i][0]** is the English sentence.
  - Element **train_pairs[i][1]** is the corresponding Spanish sentence.

```
train_english_texts = [pair[0] for pair in train_pairs]
train_spanish_texts = [pair[1] for pair in train_pairs]
source_vectorization.adapt(train_english_texts)
target_vectorization.adapt(train_spanish_texts)
```

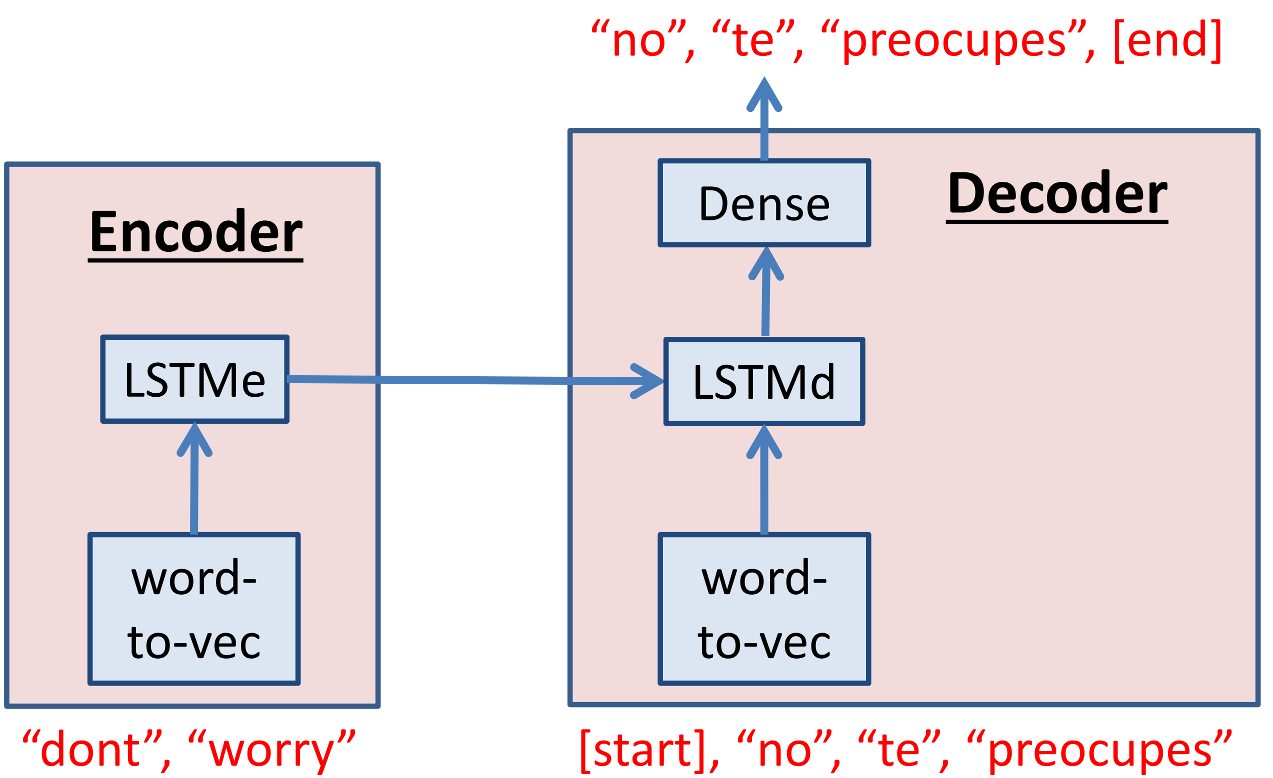
Input and Target Sequences

- For every training object we need to define explicitly:
  - What the input is.
  - What the target output is.
- What are they?



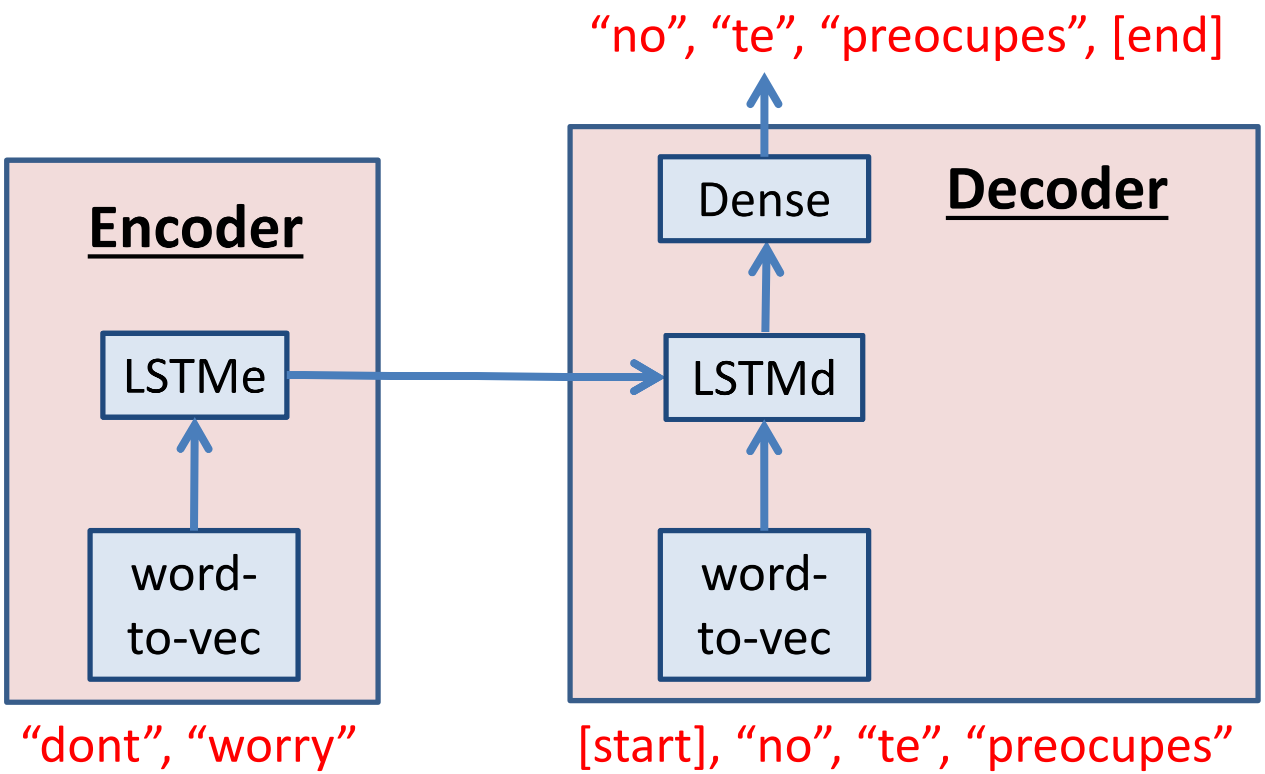
Inputs to the RNN

- The RNN will take two inputs:
  - Encoder input: English text.
  - Decoder input: Spanish text, **except for the [end] token**.



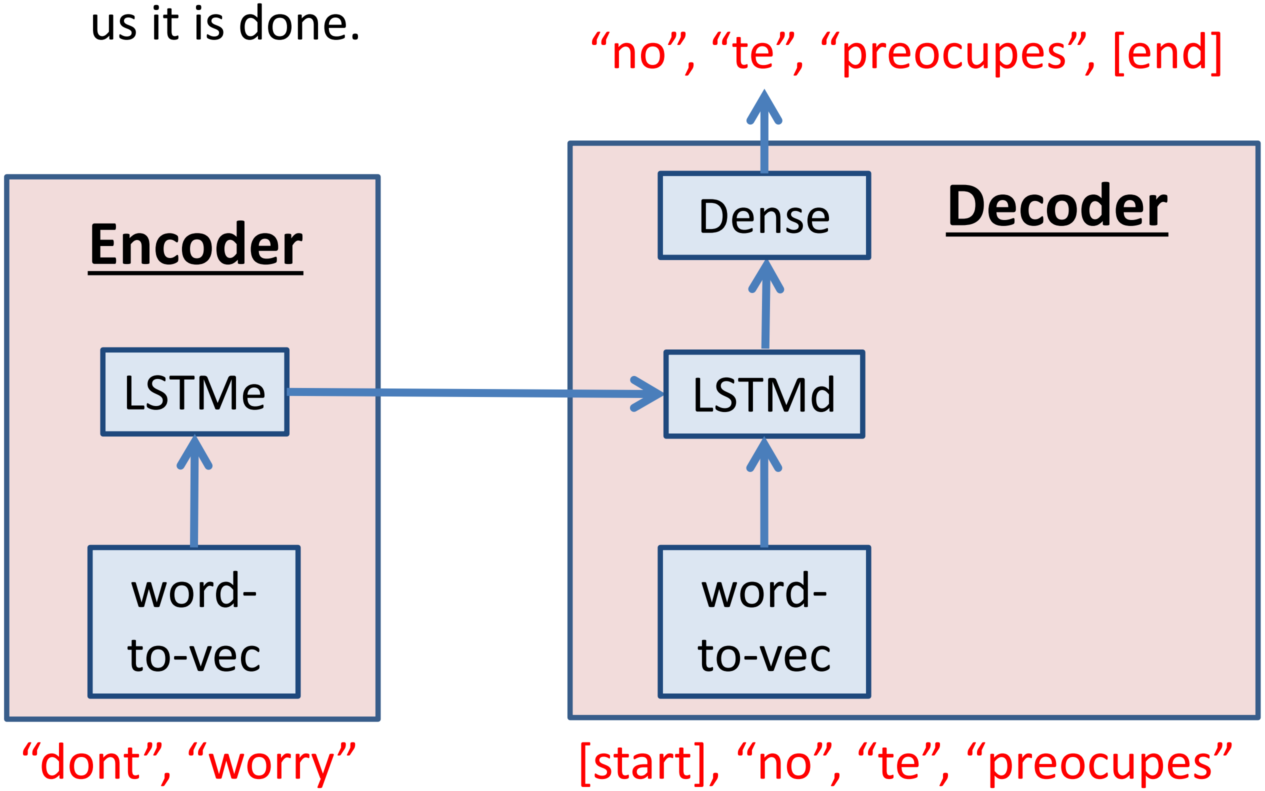
Target Output for the RNN

- Target output: Spanish text, **<u>except for the [start] token</u>**.
- Why do we set up inputs and targets like this?
  - See the slides earlier, describing step by step how the encoder and the decoder work.



Target Output for the RNN

- Remember:
  - [start] is only used as input, not as output. It is our way to tell the decoder to start decoding.
  - [end] is only used as output, not as input. It is the decoder's way of telling us it is done.



Input and Target Sequences: Code

```
def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({"english": eng,
             "spanish": spa[:, :-1]},
            spa[:, 1:])
```

- This function creates input and target sequences so that they match the drawing in the previous slide.
- Input arguments:
  - **eng**: a list of strings. Each string contains text in English.
  - **spa**: a list of strings. Element **spa[i]** is the Spanish translation of **eng[i]**.

Input and Target Sequences: Code

```
def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({"english": eng,
             "spanish": spa[:, :-1]},
            spa[:, 1:])
```

- First, it vectorizes the English and Spanish strings.
- Then, it creates and returns two objects:
- First object: {"english": eng, "spanish": spa[:, :-1]}, a **dictionary**.
  - First item: eng, a list of vectorized English sentences.
  - Second item: spa[:, :-1], a list of vectorized Spanish sentences, **skipping the last element** of each sentence (which is the [end] token).

Input and Target Sequences: Code

```
def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({"english": eng,
             "spanish": spa[:, :-1]},
            spa[:, 1:])
```

- First object: {"english": eng, "spanish": spa[:, :-1]}, a **dictionary**.
- Can you guess why we are returning a dictionary?
- Remember, we will need to somehow tell Keras that we will give a separate input to the encoder and a separate input to the decoder. That is something we have not done so far.
- This dictionary will help us tell Keras exactly what to do.

Input and Target Sequences: Code

```
def format_dataset(eng, spa):
    eng = source_vectorization(eng)
    spa = target_vectorization(spa)
    return ({"english": eng,
             "spanish": spa[:, :-1]},
            spa[:, 1:])
```

- Second return value: list of target outputs, spa[:, 1:].
- Each element here is a sequence of ints representing the Spanish text, **skipping the initial element** of the sequence (which would be the [start] token).

Creating a Tensorflow Dataset

```
def make_dataset(pairs):
    eng_texts, spa_texts = zip(*pairs)
    eng_texts = list(eng_texts)
    spa_texts = list(spa_texts)
    dataset = tf.data.Dataset.from_tensor_slices((eng_texts, spa_texts))
    dataset = dataset.batch(batch_size)
    dataset = dataset.map(format_dataset)
    return dataset.shuffle(2048).prefetch(16).cache()
```

- The **make_dataset** function creates the actual optimized Tensorflow datasets that we will use during training.
  - Here we use the **format_dataset** function that we just discussed.
- Note the use of the **from_tensor_slices** function.
  - It converts lists of numpy arrays into a Tensorflow Dataset object.

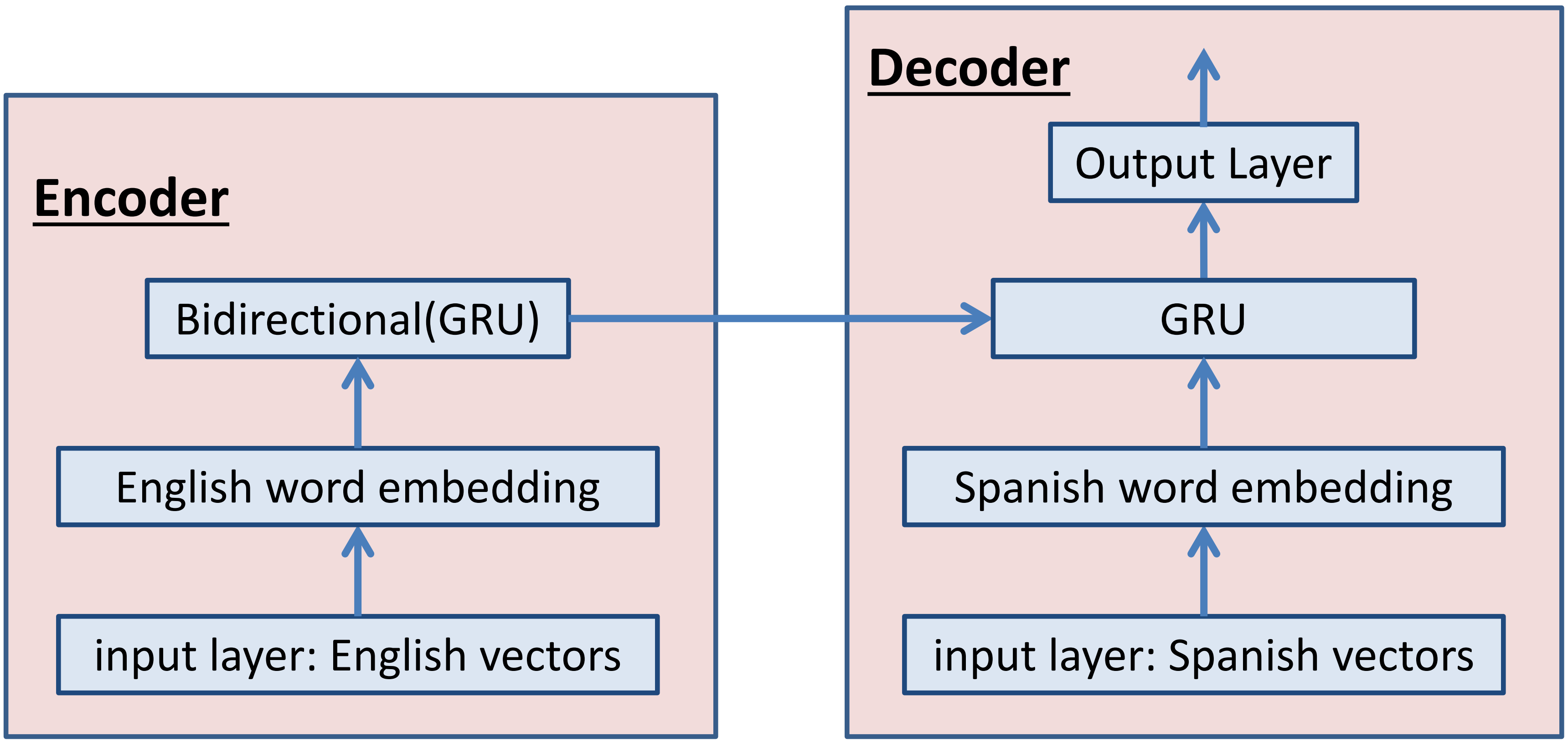
Training and Validation Sets

```
train_ds = make_dataset(train_pairs)
val_ds = make_dataset(val_pairs)
```

- Here we use the **make_dataset** function to create the training and validation sets that we will use during training.
- In both cases, we have inputs and target outputs that follow the specifications we have described.

Specifying the Model

- This is the encoder-decoder model that we will build.
- The next slide will show the Keras code that builds the model.
  - We will see how each line of the code corresponds to this drawing.



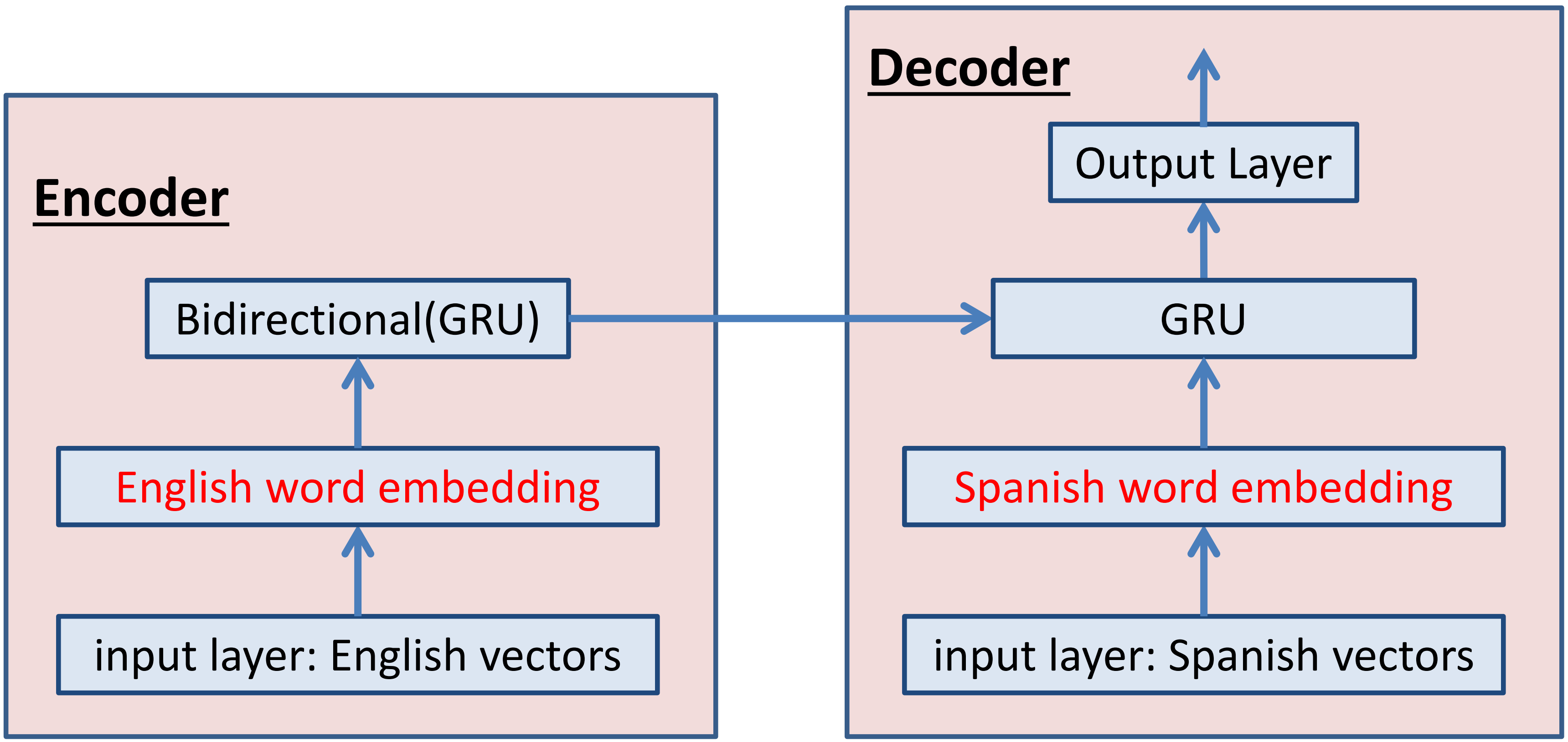
The Model in Keras Code

```
embed_dim = 256
latent_dim = 1024

source = keras.Input(shape=(None,), dtype="int64", name="english")
x1 = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)
encoded_source = layers.Bidirectional(layers.GRU(latent_dim),
                                      merge_mode="sum")(x1)

past_target = keras.Input(shape=(None,), dtype="int64", name="spanish")
x2 = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(past_target)
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
x3 = decoder_gru(x2, initial_state=encoded_source)
x4 = layers.Dropout(0.5)(x3)
target_next_step = layers.Dense(vocab_size, activation="softmax")(x4)
seq2seq_rnn = keras.Model([source, past_target], target_next_step)
```

The Model in Keras Code

```
embed_dim = 256
```

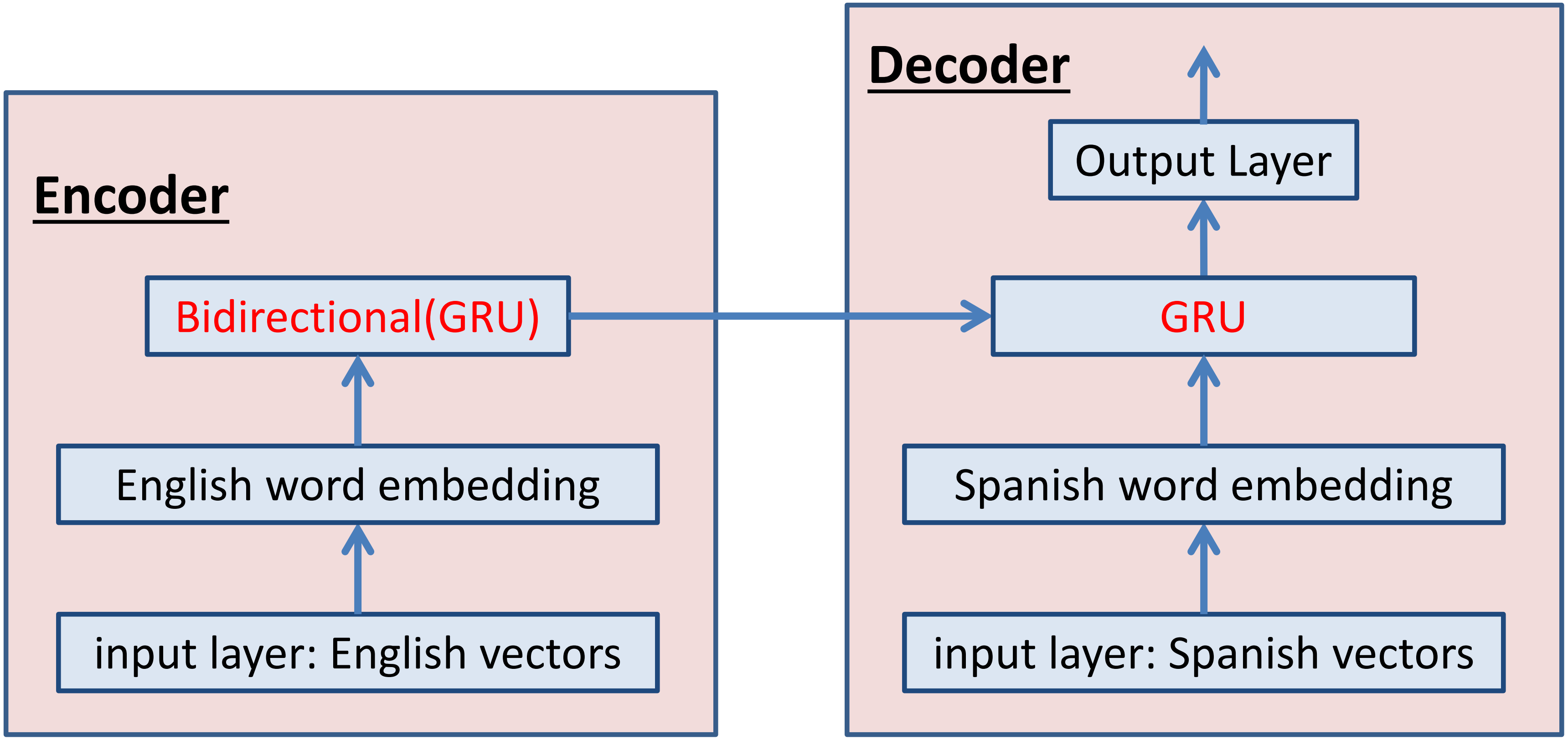
- The word embeddings will have 256 dimensions.



The Model in Keras Code

latent_dim = 1024

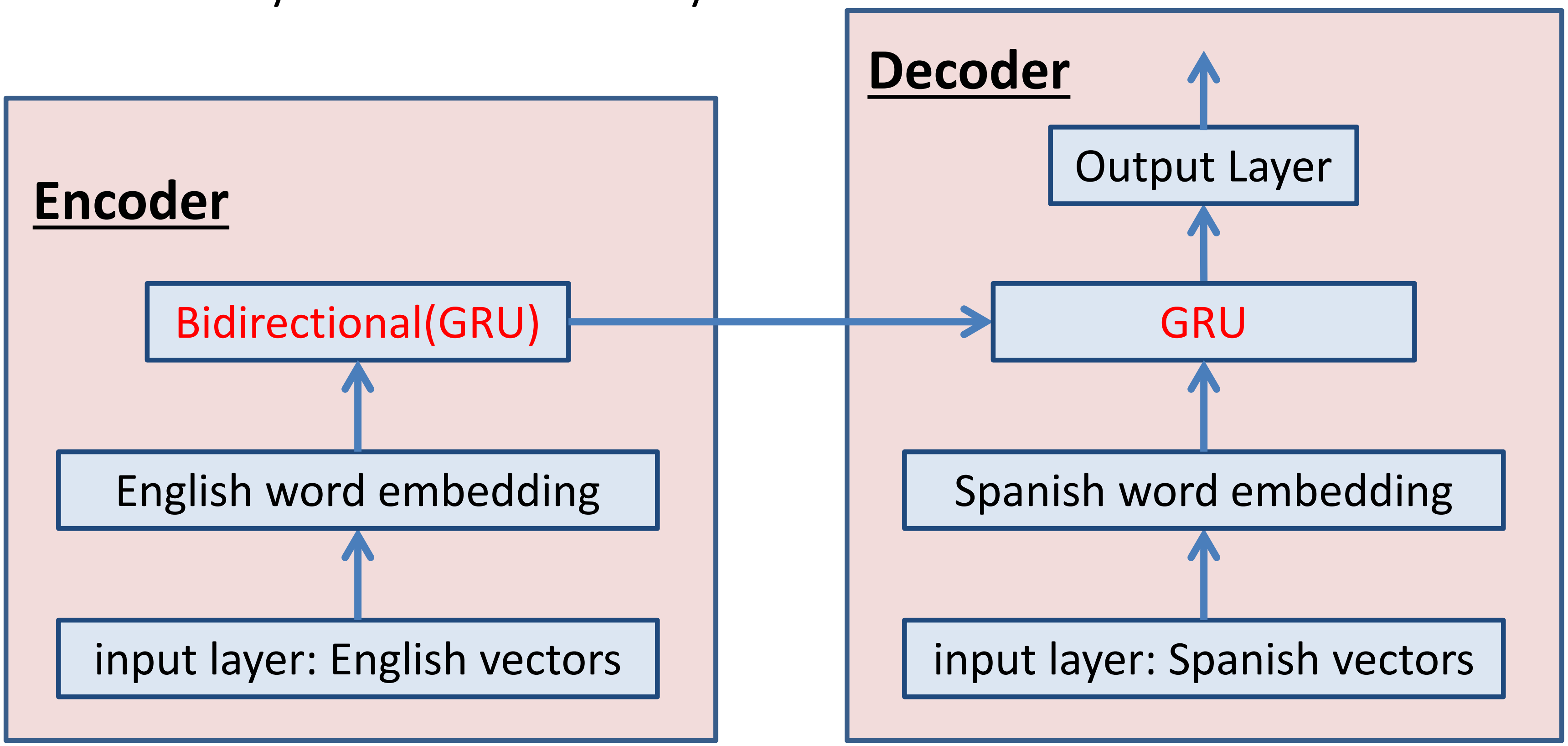
- Both the Encoder GRU layer and the Decoder GRU layer will have 1024 units each.



The Model in Keras Code

latent_dim = 1024

- Note that we will be using GRU layers instead of LSTM layers.
  - They are still recurrent layers.



The Model in Keras Code

- We start by showing the Encoder and Decoder modules as empty.
- Then, for each line of Keras code, we will see what blocks and connections it creates.

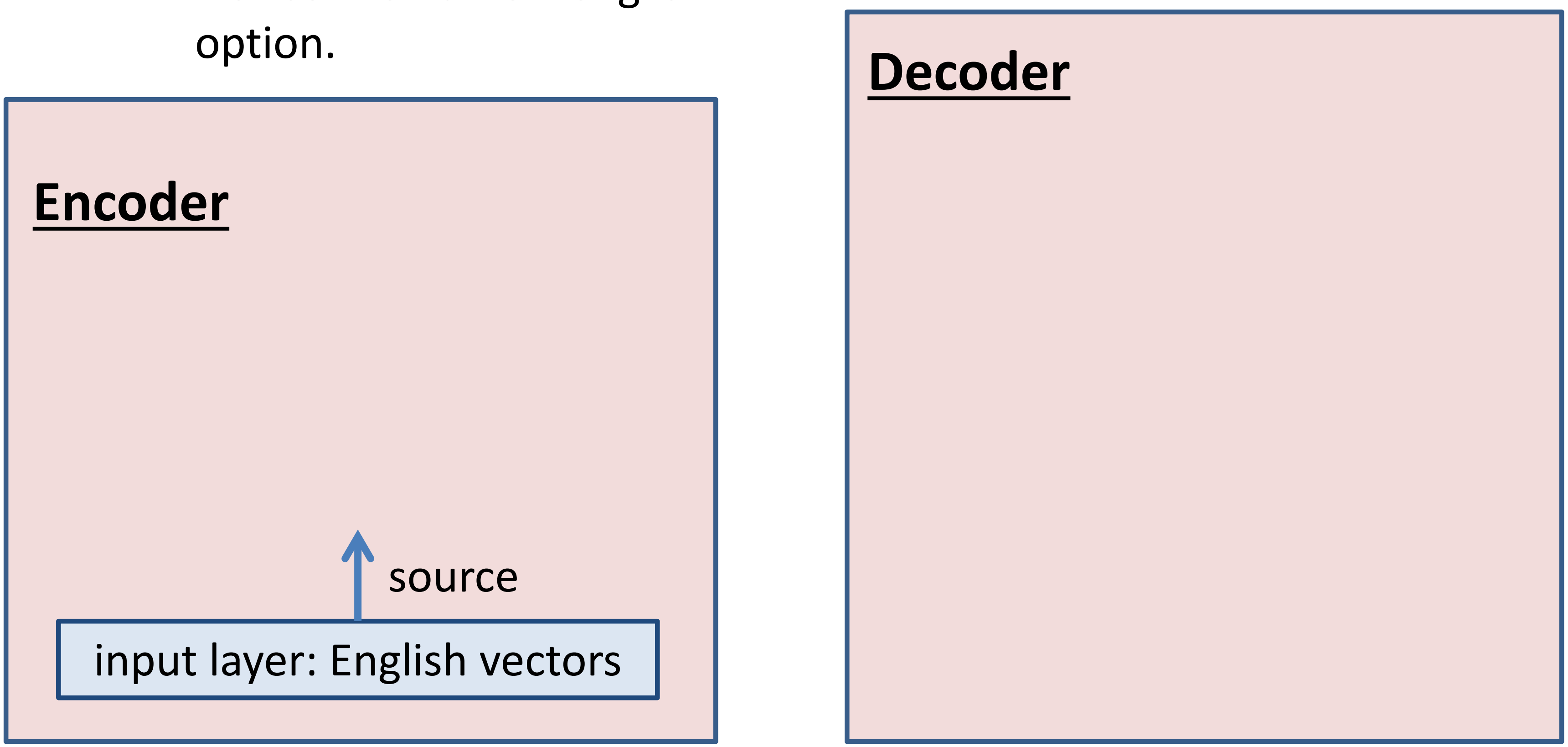
Encoder

Decoder

Specifying Inputs

```
source = keras.Input(shape=(None,), dtype="int64", name="english")
```

- This is the first line, specifying inputs.
  - Notice the name="english“ option.

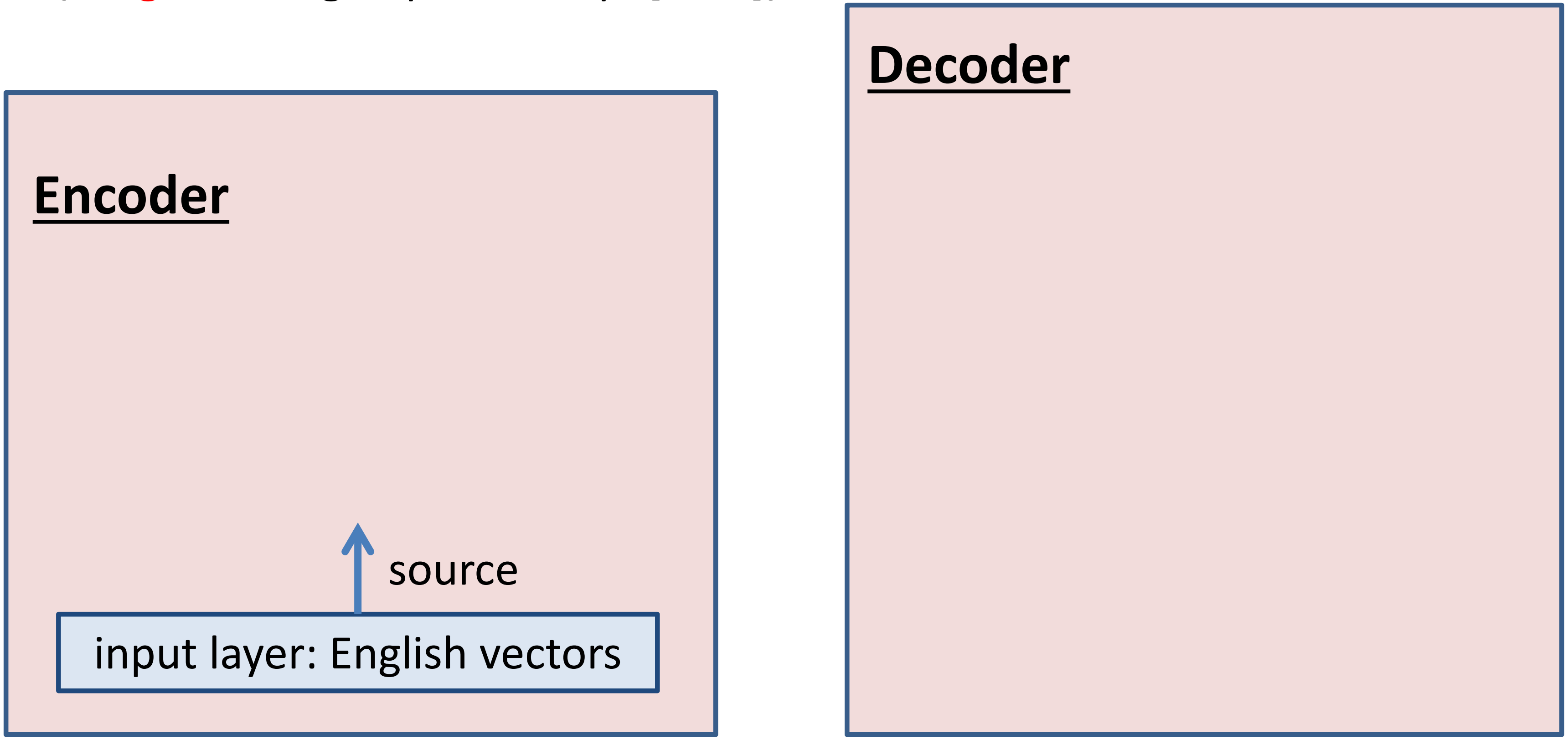


Specifying Inputs

source = keras.Input(shape=(None,), dtype="int64", name="english")

- This name="english" refers to our dictionary of training inputs:

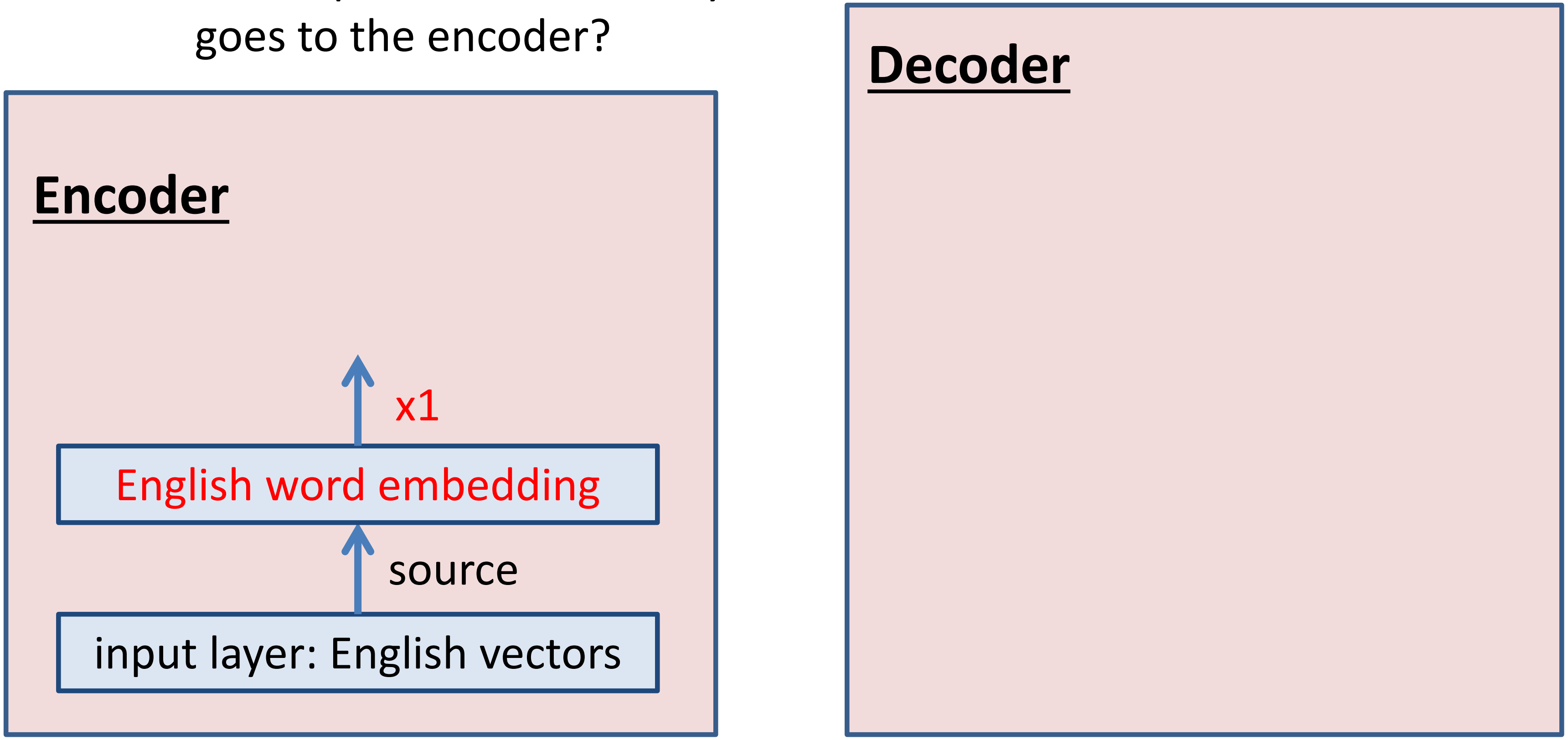
{"english": eng, "spanish": spa[:, :-1]}



Adding Encoder Word Embedding

```
x1 = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)
```

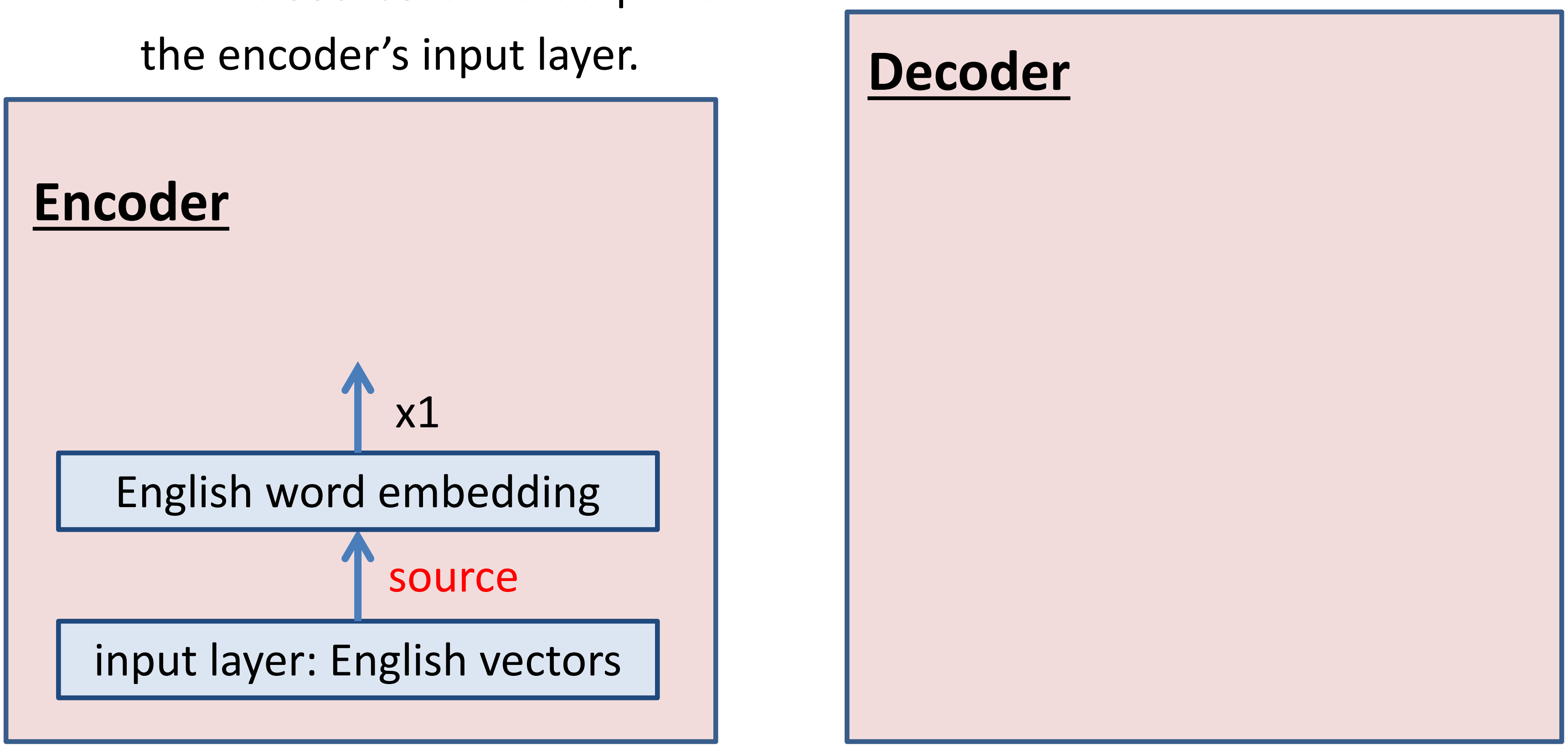
- Here we add the encoder's word embedding layer.
  - What specifies that this layer goes to the encoder?



Adding Encoder Word Embedding

x1 = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(source)

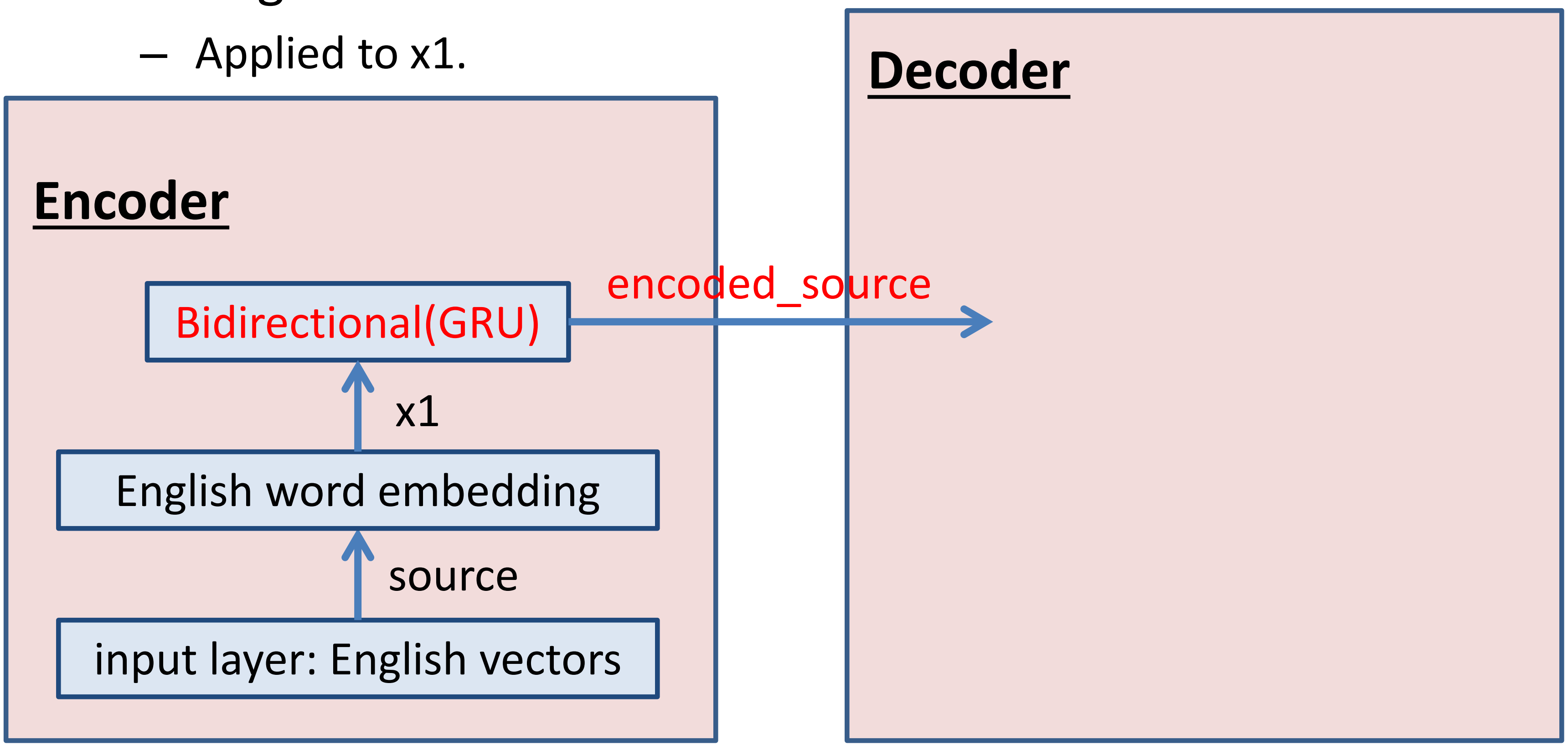
- This layer is applied to variable **source**.
  - This **source** is the output of the encoder's input layer.



Adding Encoder Recurrent Layer

```
encoded_source = layers.Bidirectional(layers.GRU(latent_dim),
                                      merge_mode="sum")(x1)
```

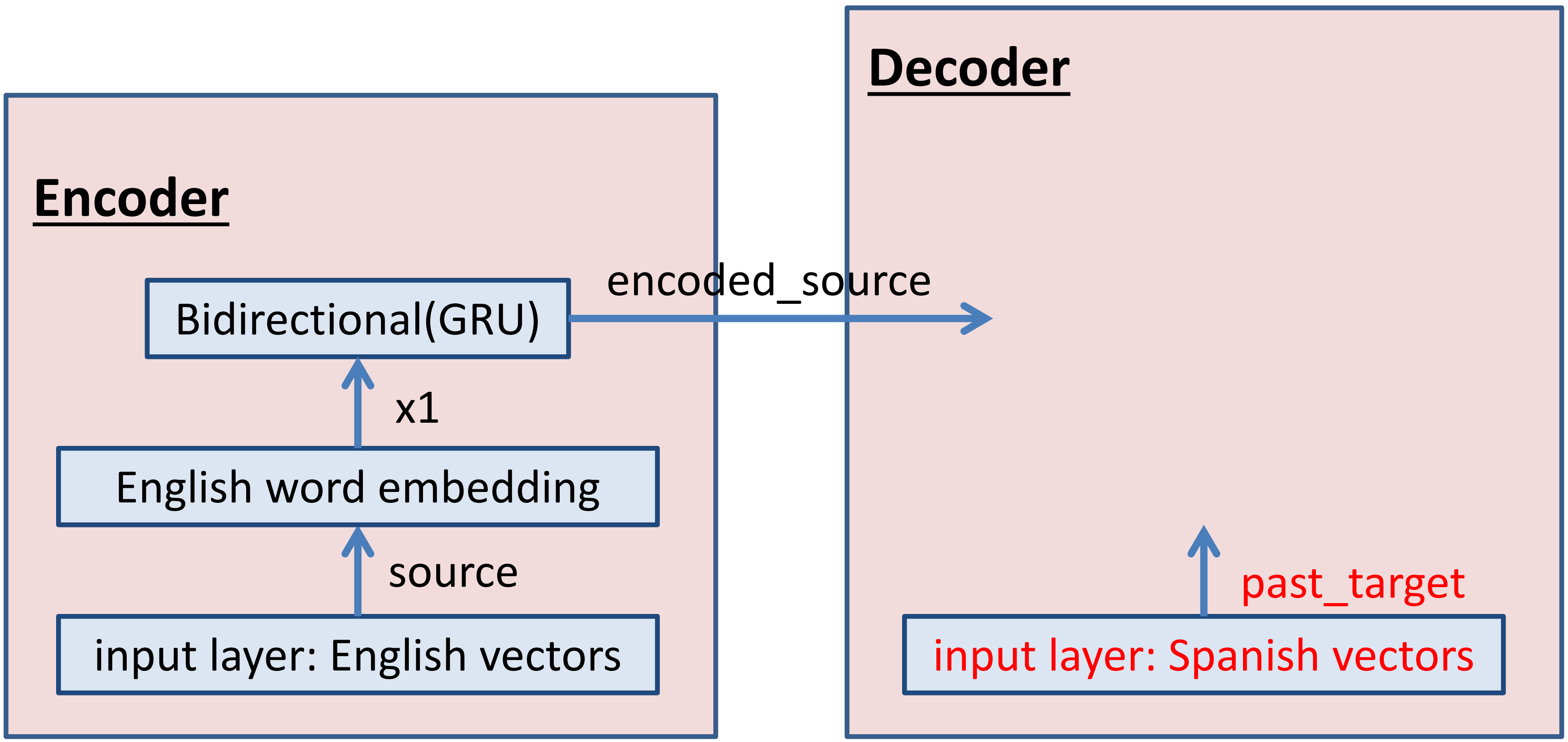
- Adding the Bidirectional GRU.
  - Applied to x1.



Adding Decoder Input

```
past_target = keras.Input(shape=(None,), dtype="int64", name="spanish")
```

- Adding the input layer to the decoder.

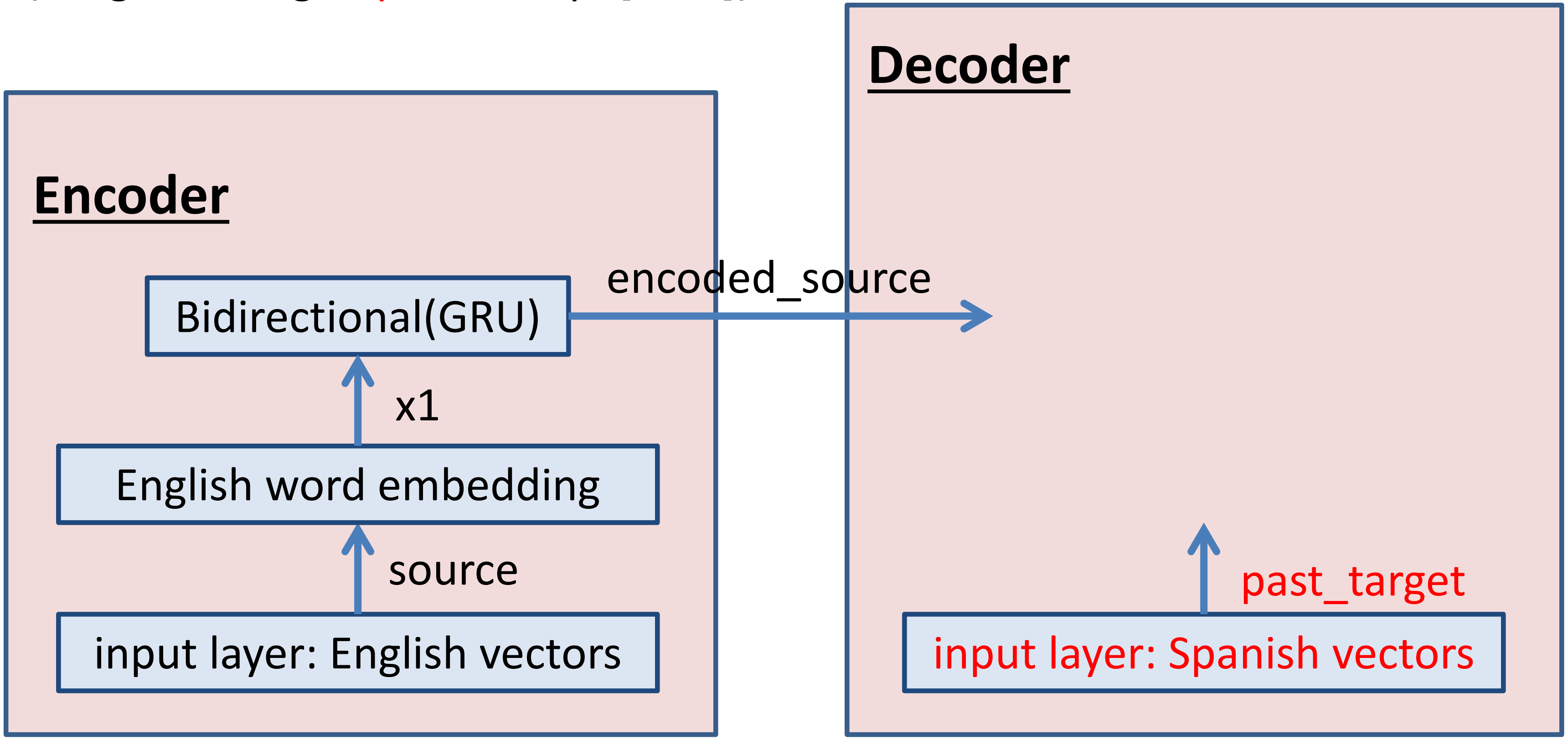


Adding Decoder Input

```
past_target = keras.Input(shape=(None,), dtype="int64", name="spanish")
```

- Again, name="spanish" refers to our dictionary of training inputs:

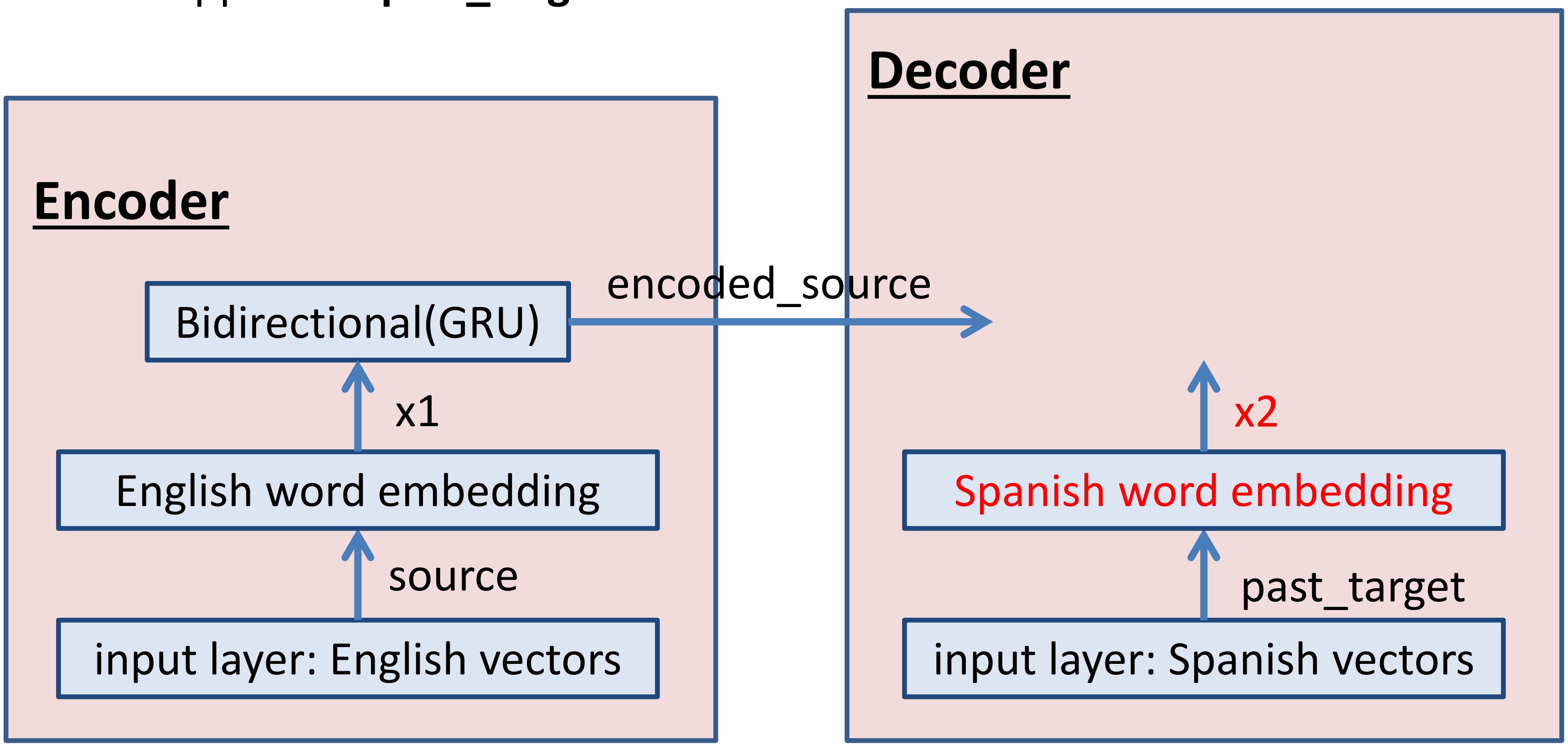
```
{"english": eng, "spanish": spa[:, :-1]}
```



Adding Decoder Word Embedding

```
x2 = layers.Embedding(vocab_size, embed_dim, mask_zero=True)(past_target)
```

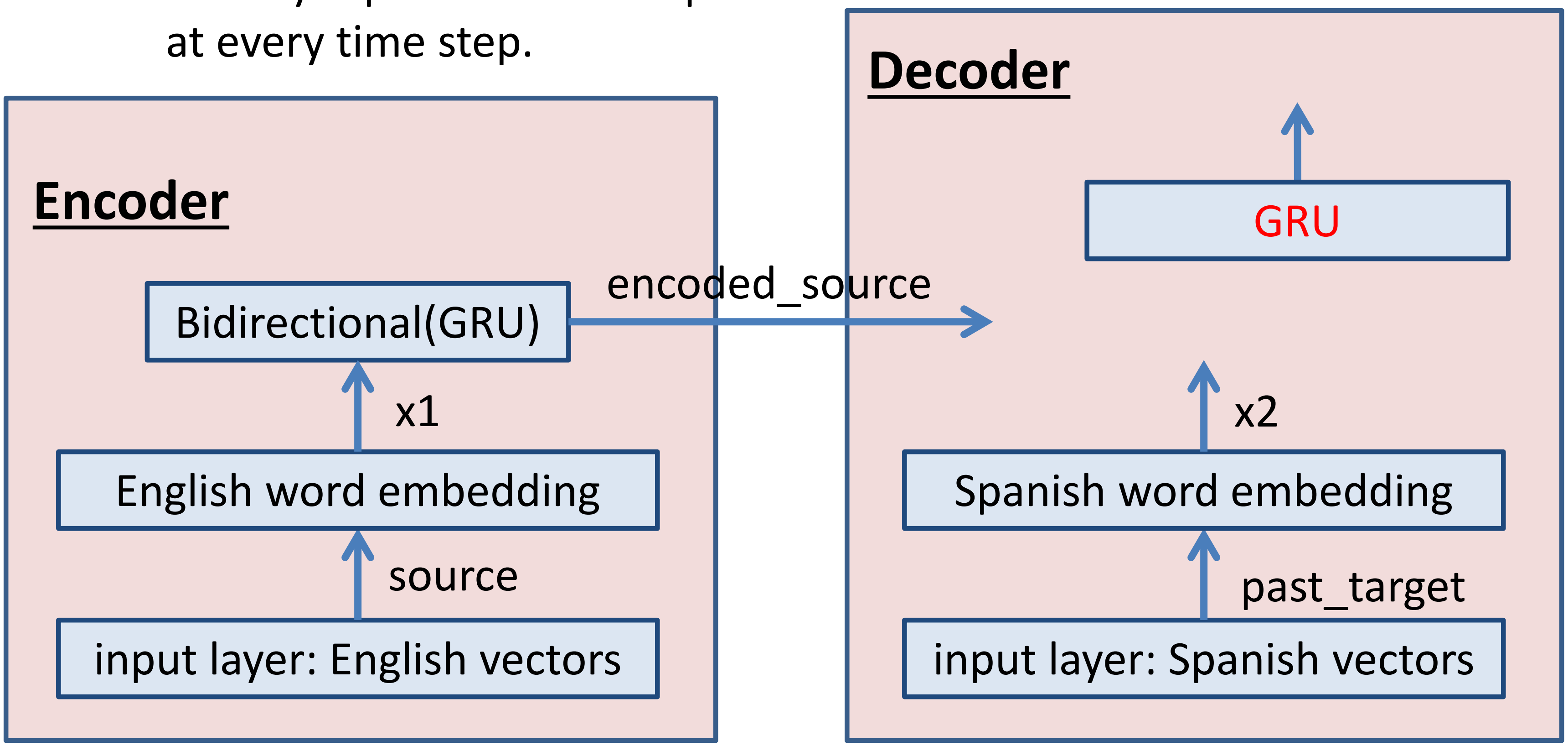
- Here we add the encoder's word embedding layer.
  - Applied to **past_target**.



Adding Decoder Recurrent Layer

```
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
```

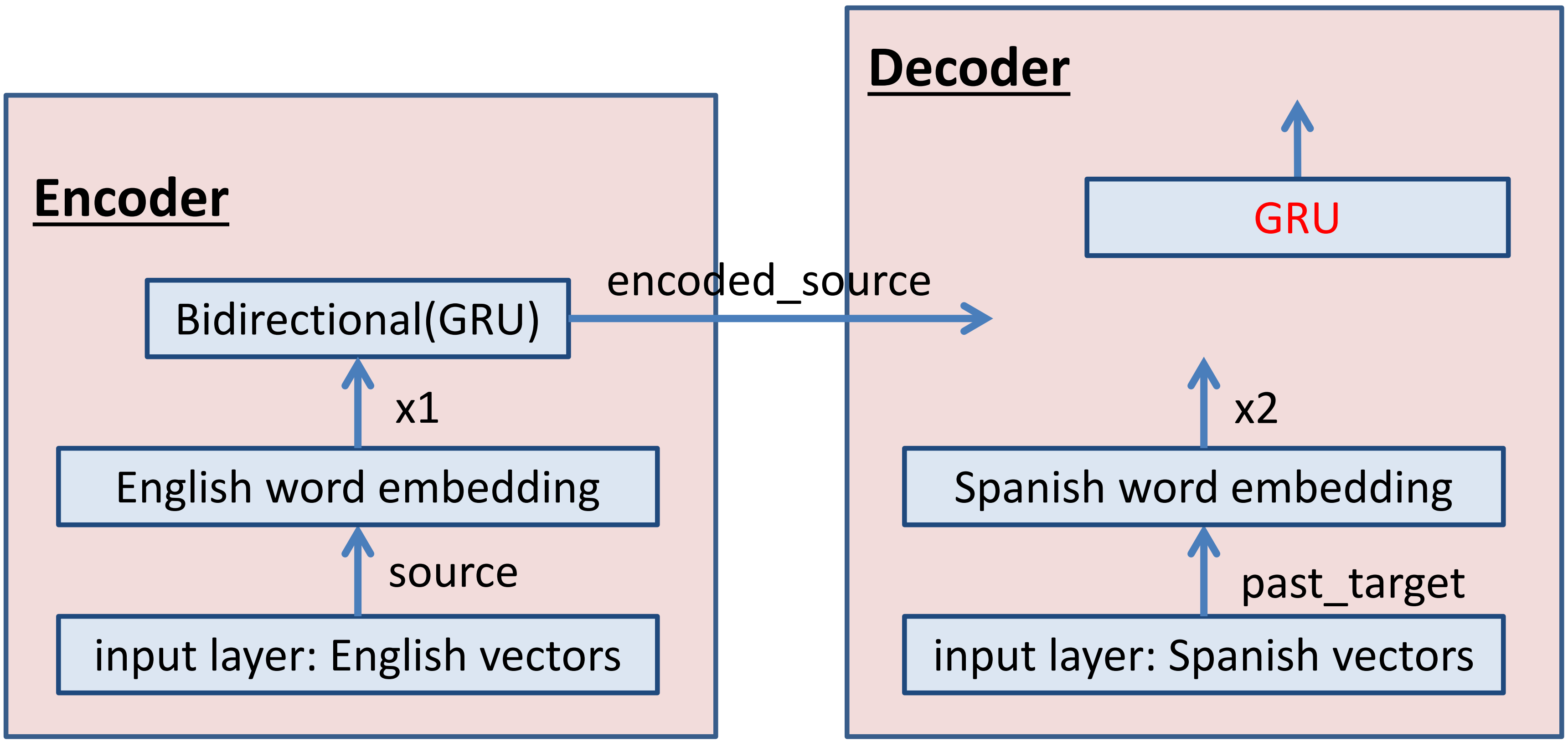
- **Very important!!!** Note the **return_sequences=True** option.
  - This layer produces an output at every time step.



Adding Decoder Recurrent Layer

```
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
```

- Trick question: why is this GRU layer shown as disconnected from the rest of the model?

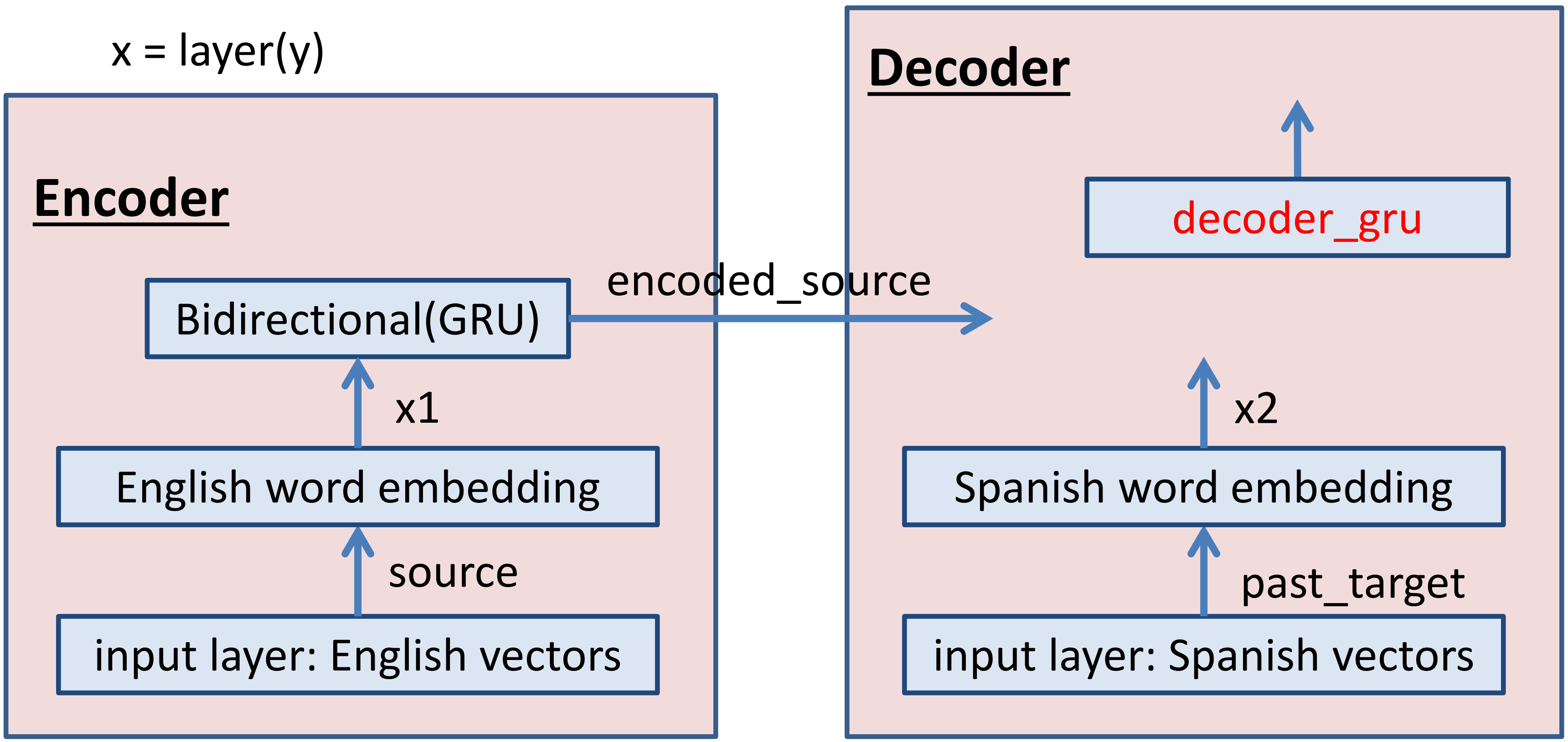


Adding Decoder Recurrent Layer

```
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
```

- This line of code **does not specify the input to this layer**.
  - Previous lines looked like:

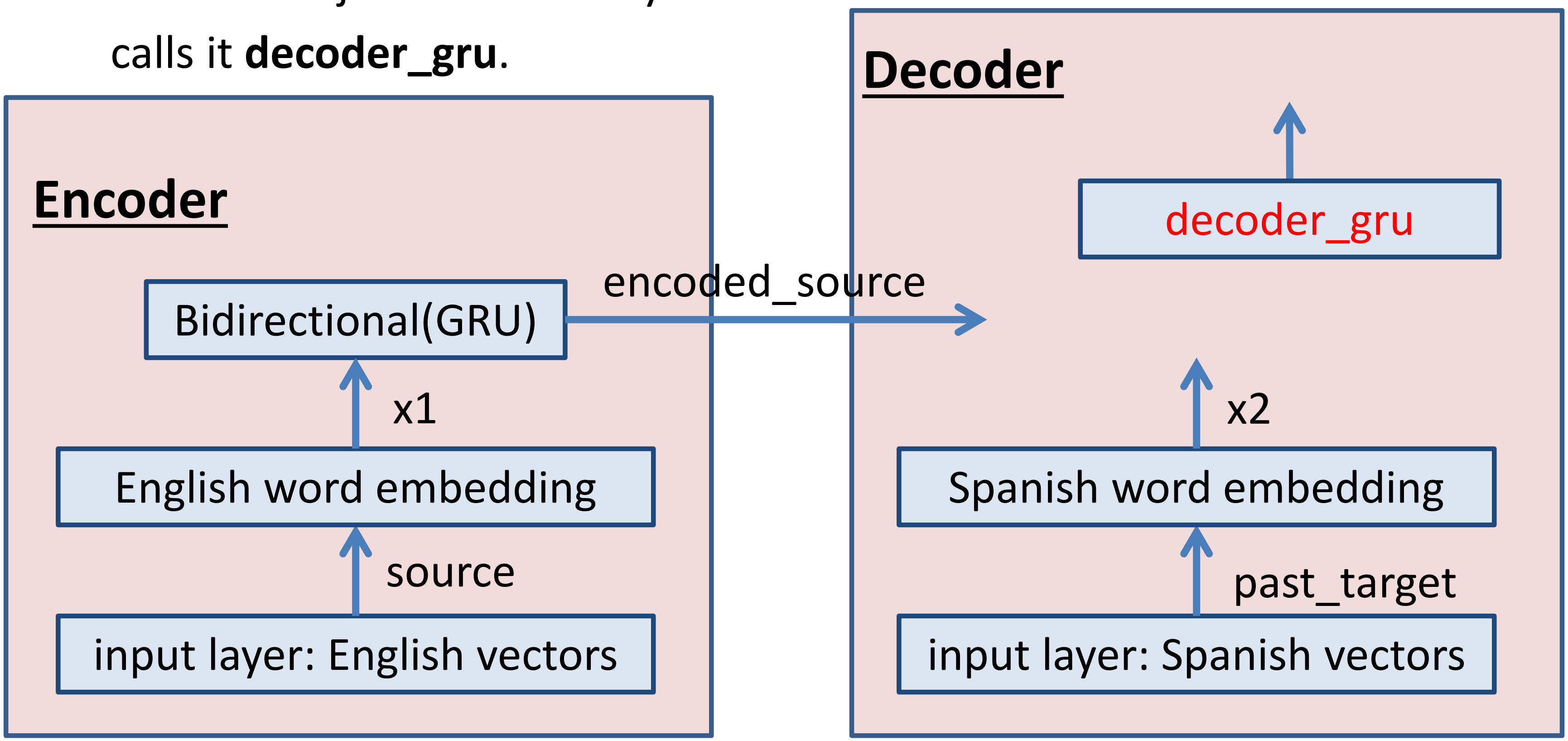
```
x = layer(y)
```



Adding Decoder Recurrent Layer

```
decoder_gru = layers.GRU(latent_dim, return_sequences=True)
```

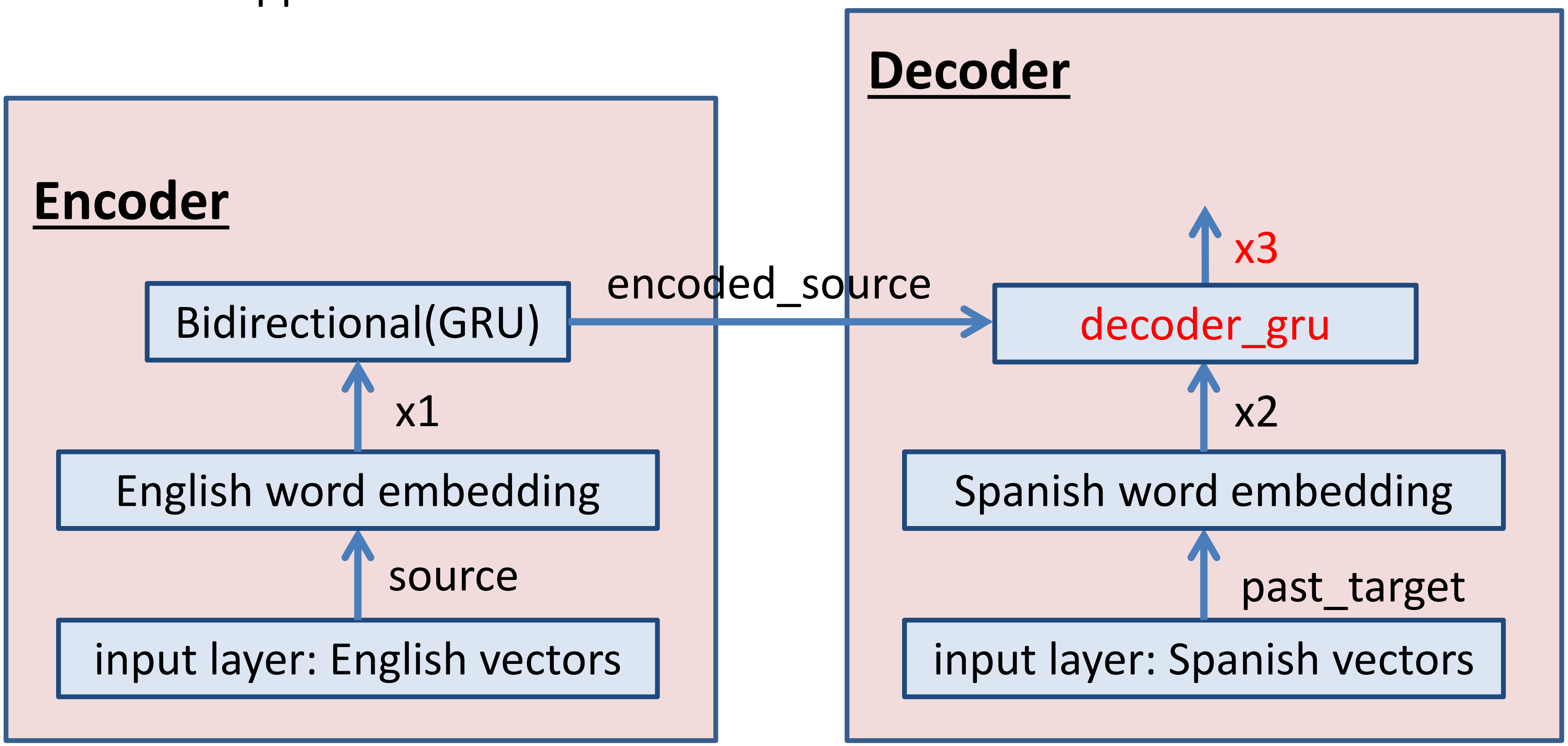
- This line of code **<u>does not specify the input to this layer</u>**.
  - This line just creates a layer and calls it **decoder_gru**.



Adding Decoder Recurrent Layer

```
x3 = decoder_gru(x2, initial_state=encoded_source)
```

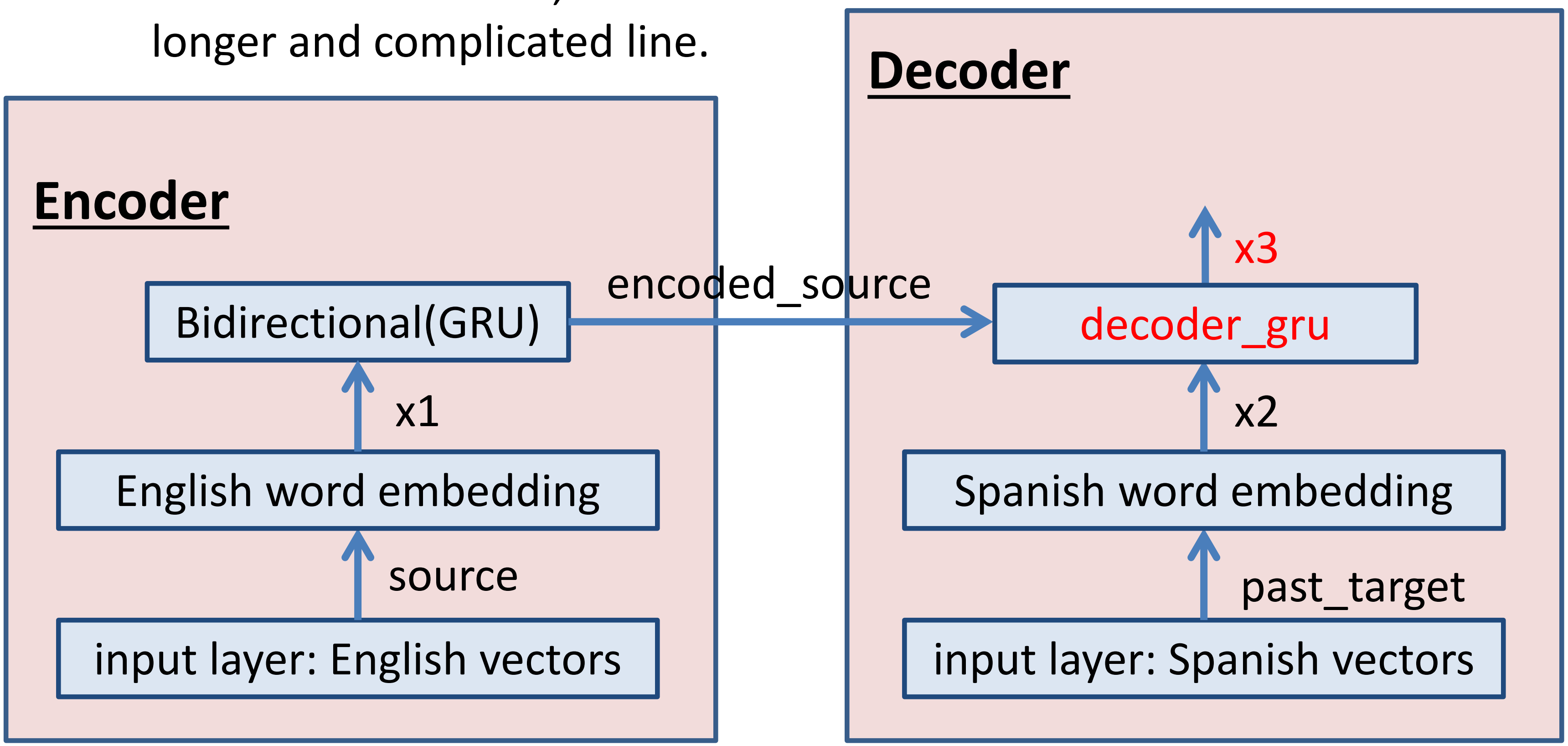
- This line of code connects **decoder_gru** to the rest of the model.
  - It is applied to **x2**.



Adding Decoder Recurrent Layer

```
x3 = decoder_gru(x2, initial_state=encoded_source)
```

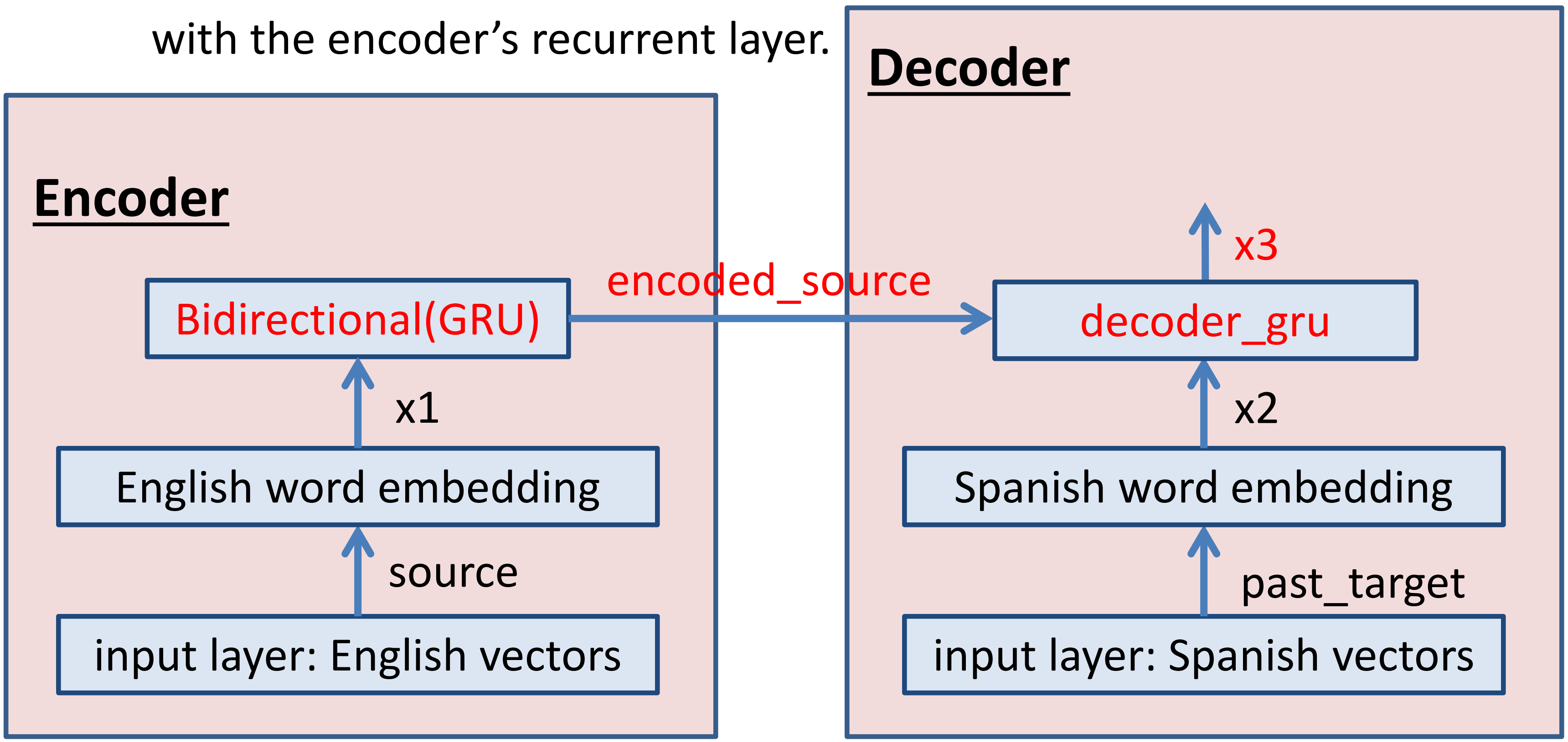
- Why did we use two lines of code to create this layer?
  - Just for convenience, to avoid a longer and complicated line.



Connecting Encoder and Decoder

x3 = decoder_gru(x2, initial_state=encoded_source)

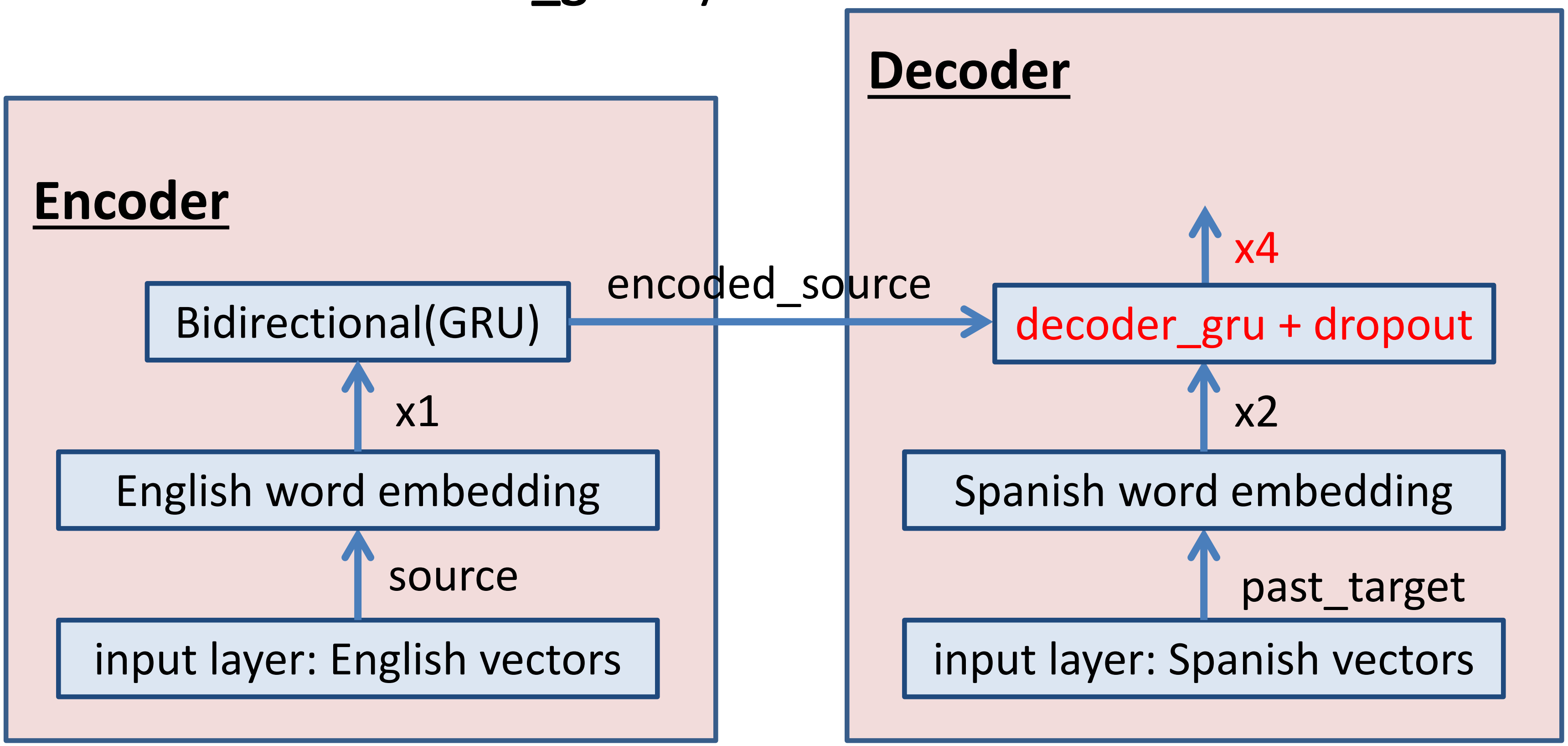
- **Very important!!!** Note the **initial_state=encoded_source** option.
  - This connects the decoder GRU with the encoder's recurrent layer.



Connecting Encoder and Decoder

```
x4 = layers.Dropout(0.5)(x3)
```

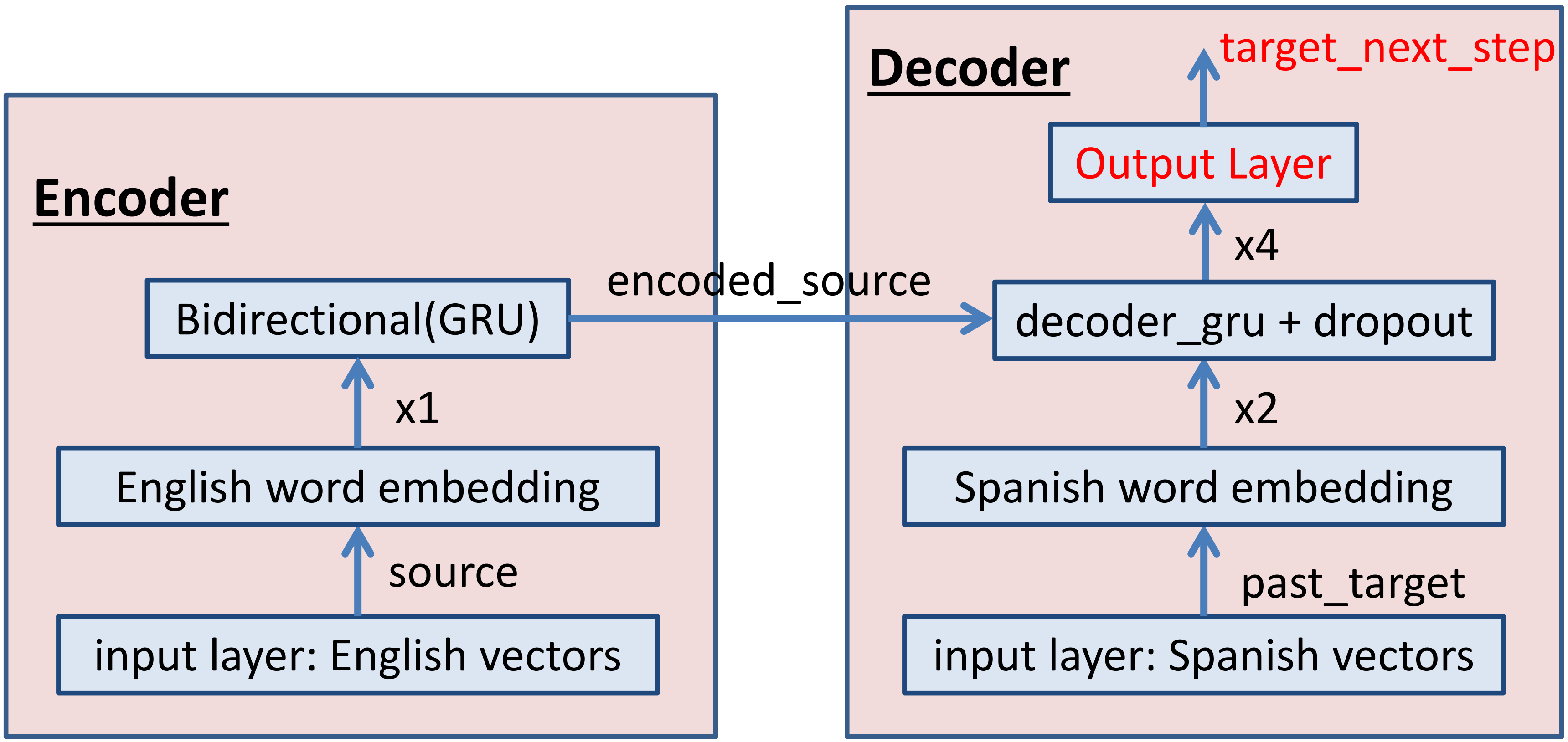
- Here we are just adding a dropout option. We can just include that in the **decoder_gru** layer.



Adding the Output Layer

```
target_next_step = layers.Dense(vocab_size, activation="softmax")(x4)
```

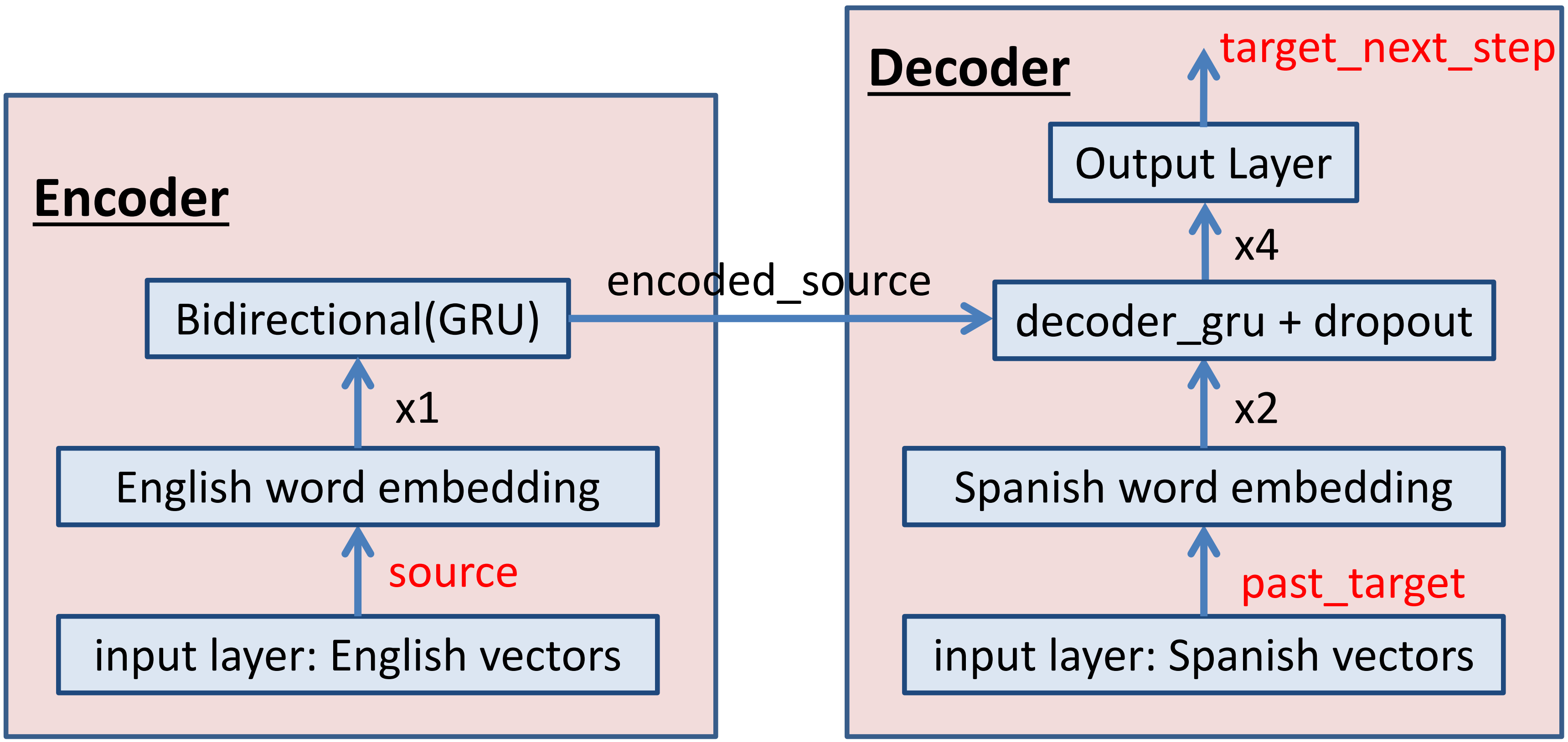
- Here we add the model's output layer.



Adding Decoder Recurrent Layer

seq2seq_rnn = keras.Model([source, past_target], target_next_step)

- Finally, we create the model, by specifying inputs and outputs.



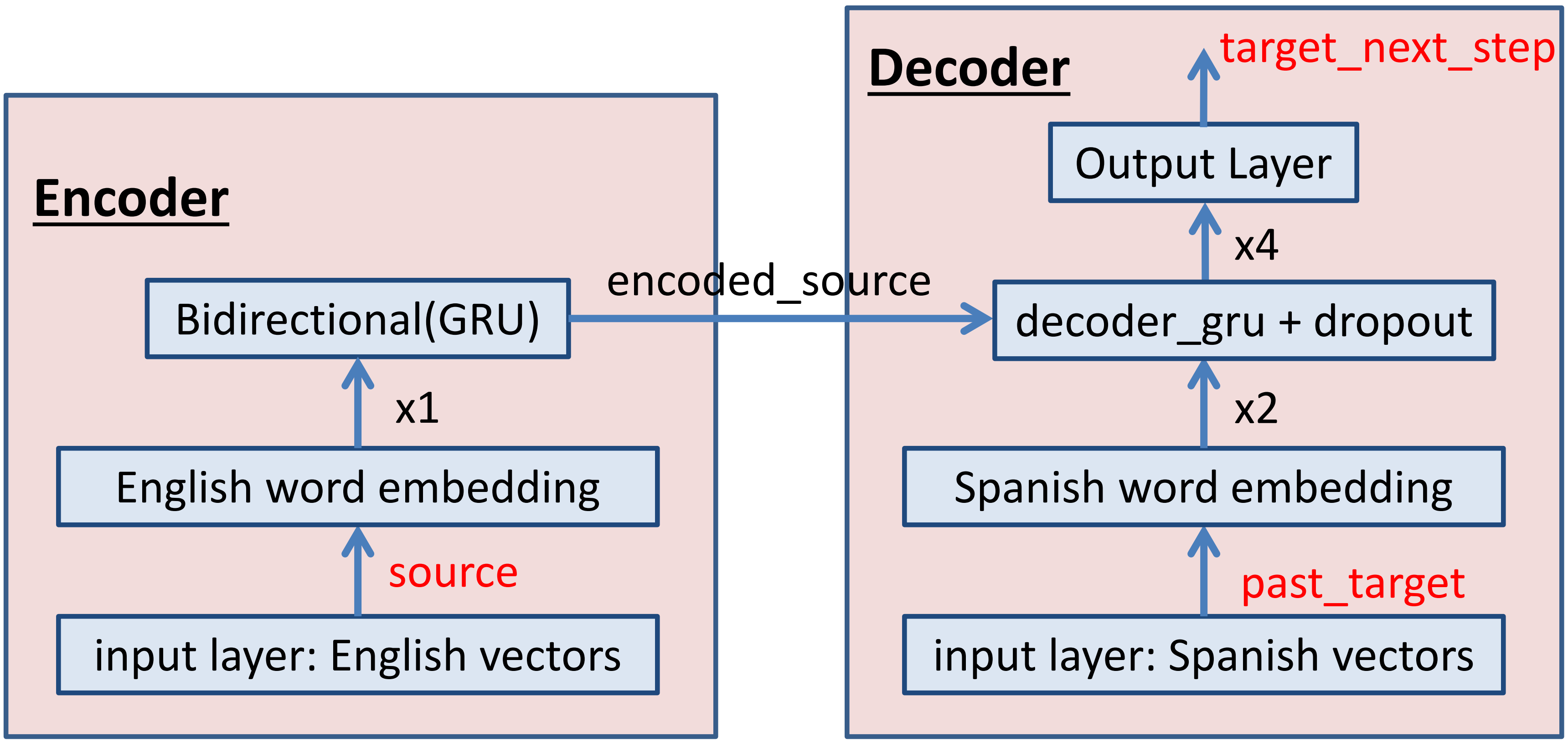
A Model as a Computational Graph

- As we said before, a neural network is a computational graph.
- In the functional API, we specify the graph piece by piece.
  - Vertex by vertex, edge by edge.
- A layer is a vertex in the graph.
- The functional API specifies how these layers connect to each other.
- A line like: x = layer(y)
  - Specifies that we will use variable **x** to refer to the output of that layer.
  - Specifies that the input to the layer comes from variable **y**. That variable **y** should have already been defined as output of another layer.

Adding Decoder Recurrent Layer

seq2seq_rnn = keras.Model([source, past_target], target_next_step)

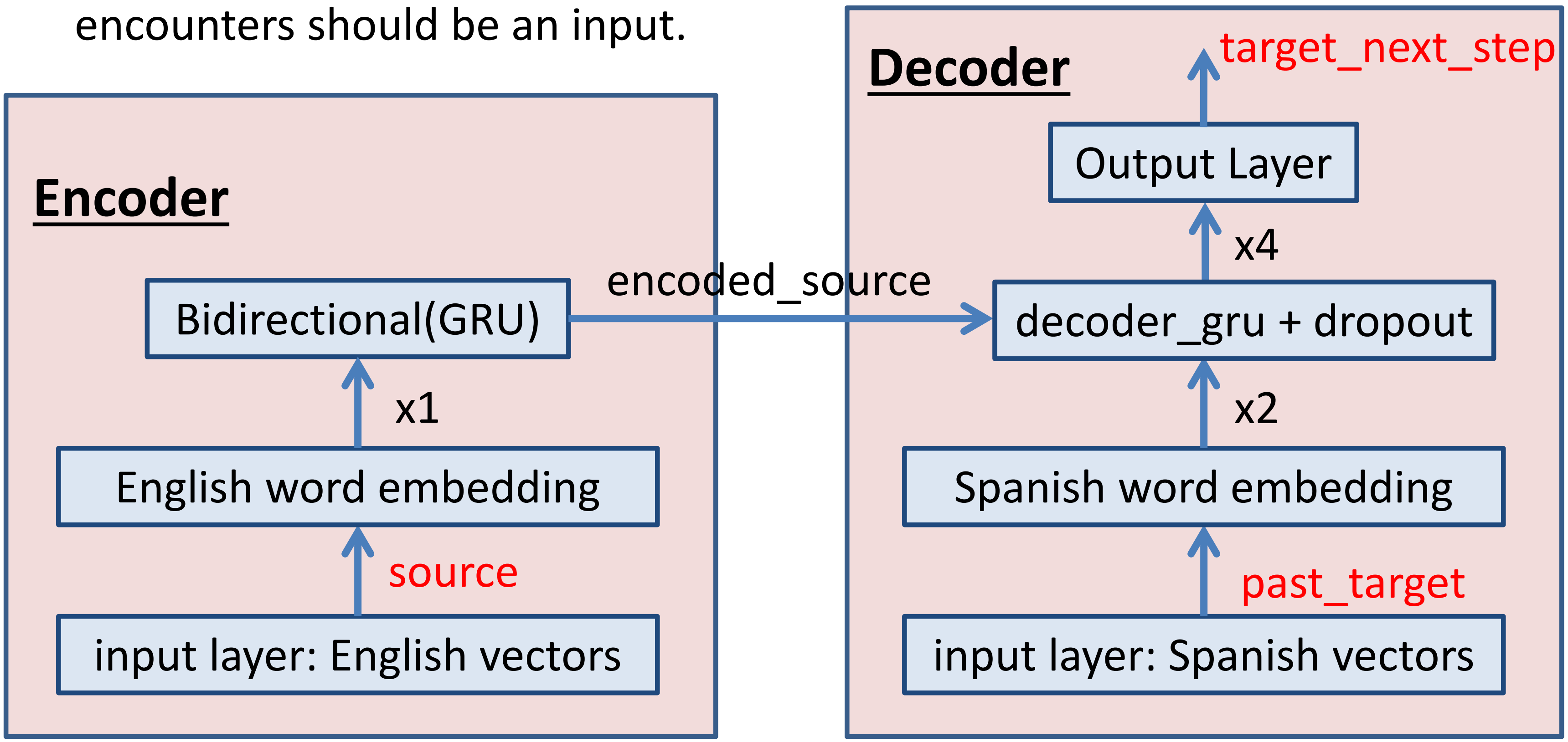
- Question: what algorithm can Keras use to verify that a model can compute the outputs given the inputs?



Adding Decoder Recurrent Layer

seq2seq_rnn = keras.Model([source, past_target], target_next_step)

- One answer that works: breadth-first search, starting at the outputs.
- Every leaf node that the search encounters should be an input.



Training the Network

```
filename = "eng_spa_rnn2_temp.keras"
callbacks = [keras.callbacks.ModelCheckpoint(filename,
                                             save_best_only=True)]

seq2seq_rnn.compile(optimizer="rmsprop",
                    loss="sparse_categorical_crossentropy",
                    metrics=["accuracy"])

seq2seq_rnn.fit(train_ds, epochs=15, validation_data=val_ds,
                callbacks=callbacks)
```

- The code for training is straightforward, there is nothing we haven’t seen before.
- Training is somewhat slow: about 35 minutes per epoch on my computer, close to 9 hours for 15 epochs.

Inference with the Model

- Once we train the model, we can use it to translate new English text to Spanish.
- Let's create some tokenized input:

```
input_sentence = "good morning"
tokenized_input = source_vectorization([input_sentence])
```

- How do we apply the model to translate this input?
- The lines below will not work. Why?

```
translation = model(tokenized_input)

translation = model.predict(tokenized_input)
```

Inference with the Model

```
input_sentence = "good morning"
tokenized_input = source_vectorization([input_sentence])
```

- The lines below will not work. Why?

```
translation = model(tokenized_input)

translation = model.predict(tokenized_input)
```

- Our enconder-decoder model takes two inputs:
  - English text given to the encoder.
  - Partial Spanish text (starting with the [start] token) given to the decoder.
- Given these inputs, our model only outputs a single word.
  - The next word in the translation.

Inference with the Model

```
input_sentence = "good morning"
tokenized_input = source_vectorization([input_sentence])
```

- The way the encoder-decoder model works, we cannot just apply it once to get the entire translation.
- We will use a loop to produce the output word by word.
- First step:
  - Encoder input: tokenized sentence.
  - Decoder input: tokenized “[start]”
  - Output: first word in the translation, (hopefully the correct one) “buenos”.
- Second step???

Inference with the Model

```
input_sentence = "good morning"
tokenized_input = source_vectorization([input_sentence])
```

- First step:
  - Encoder input: tokenized sentence.
  - Decoder input: tokenized “[start]”
  - Output: first word in the translation, (hopefully the correct one) “buenos”.
- Second step:
  - Encoder input: tokenized sentence.
  - Decoder input: tokenized “[start] buenos”
  - Output: second word in the translation, (again, hopefully correct) “días”.

Inference with the Model

```
input_sentence = "good morning"
tokenized_input = source_vectorization([input_sentence])
```

- Second step:
  - Encoder input: tokenized sentence.
  - Decoder input: tokenized “[start] buenos”
  - Output: second word in the translation, (hopefully correct) “días”.
- Third step:
  - Encoder input: tokenized sentence.
  - Decoder input: tokenized “[start] buenos días”
  - Output: third word in the translation, (again, hopefully correct) [end].
- So, if the model produced the correct output, at this point we got the [end] token, and we are done with the translation.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

Input arguments:
- input_sentence, a string of English text.
- max_decoded_length, we will see its use in a bit.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Vectorize the input sentence.
- Note that source_vectorization is a global variable. This function uses several global variables.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_sentence_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Initialize the decoded sentence.
- Eventually it will be the entire translation text.
- Initially it just contains the [start] token.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Entering the main loop.
- This is where we use input argument max_decoded_length, to ensure that the loop will eventually terminate.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- We vectorize the decoded sentence (initially just the [start] token, but it will get longer by a word at each iteration).

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Here we apply our model.
- Note: we pass the two inputs as a list.
- The result, next_token, is a vector, the output of all units in the output layer.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Here we find the position of the output unit with the highest output value.
- That position corresponds to the next word in the translation.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Why is next_token a 3D array?

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- First dimension: index of test object within the batch (here we only have one test object, so its index is 0).

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Second dimension: index of time step. Remember that the decoder RNN produces an output at each time step.
- We only want the last output, which is at position i.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
  input_tokens = source_vectorization([input_sentence])
  decoded_sentence = "[start]"
  for i in range(max_decoded_length):
    target_tokens = target_vectorization([decoded_sentence])
    next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
    sampled_token_index = np.argmax(next_token[0, i, :])
    spa_vocab = target_vectorization.get_vocabulary()
    sampled_token = spa_vocab[sampled_token_index]
    decoded_sentence += " " + sampled_token
    if sampled_token == "[end]":
      break
  return decoded_sentence
```

- For example, if i = 0:
  - The decoded sentence is “[start]”, it has length 1.
  - The decoder processes a single time step, we get the output of that.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- For example, if i = 1:
  - The decoded sentence is “[start] te”, it has length 2.
  - The decoder processes two time steps, we get the last output.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Third dimension: index of output unit. We have as many output units as the number of words in our Spanish vocabulary.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- In these two lines, we look up the word that corresponds to the output unit with the highest value.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- We add the new token to the translation.
- If the new token is [end], we are done, we exit the loop.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

- Done, we return the translation of the input sentence to Spanish.

Inference Code in Keras

```
def decode_sequence(input_sentence, max_decoded_length=20):
    input_tokens = source_vectorization([input_sentence])
    decoded_sentence = "[start]"
    for i in range(max_decoded_length):
        target_tokens = target_vectorization([decoded_sentence])
        next_token = seq2seq_rnn.predict([input_tokens, target_tokens])
        sampled_token_index = np.argmax(next_token[0, i, :])
        spa_vocab = target_vectorization.get_vocabulary()
        sampled_token = spa_vocab[sampled_token_index]
        decoded_sentence += " " + sampled_token
        if sampled_token == "[end]":
            break
    return decoded_sentence
```

In theory, if things go wrong, the decoder may never output the [end] token, or it may output a lot of other tokens before. Using max_decoded_length, we cut off the decoding process early in that case.

Translation Examples

```
max_decoded_length = 20
input_sentence = "Good morning"
print(input_sentence)
print(decode_sequence(input_sentence, max_decoded_length))
```

- This code shows an example where we specify some text in English and we use the **decode_sequence** function (that we just described) to get the Spanish translation.
- You can try this with any text you like.

Translation Examples

- These are some results shown in the textbook.
  - With our models the results may be different, since each model is randomly initialized before training.

Who is in this room?

[start] quién está en esta habitación [end]

- Comment: this looks correct (at least to me, my Spanish is far from perfect).

That doesn't sound too dangerous.

[start] eso no es muy difícil [end]

- Comment: this is wrong, the Spanish translation means “That is not very difficult”.

Translation Examples

- Some more examples from the textbook:

No one will stop me.

[start] nadie me va a hacer [end]

- Comment: this is wrong, does not even make sense in Spanish (at least to me), it means something like “No one is going to make me”.

Tom is friendly.

[start] tom es un buen [UNK] [end]

- Comment: again wrong. Here we get [UNK] as part of the output, so the decoder acknowledges that it was not able to make a complete translation. The rest of the Spanish text means “Tom is a good”.

Summary (1)

- We have used RNNs for sequence-to-sequence translation.
- The translation model combines two RNNs:
  - The encoder RNN, that processes the input text.
  - The decoder RNN, which takes a partial translation as input, and produces the next word as output.
  - The encoder RNN provides its output as initial recurrent input to the decoder RNN.
- To initialize the decoding, we start with a partial translation that only contains the [start] token.
- The decoding process is done when the decoder outputs the [end] token.

Summary (2)

- At training time, we must carefully define the inputs and target outputs.
  - The encoder input is straightforward.
  - The decoder input is the Spanish translation, with the [start] token at the beginning.
  - The target output is the Spanish translation, with the [end] token at the end.
- At inference time, we apply our model repeatedly, to construct the translation word by word.
- This has also been our first example of a model that takes two separate inputs, that go to different parts.
  - At both training time and inference time, we must write appropriate code to specify which input goes to the encoder and which goes to the decoder.