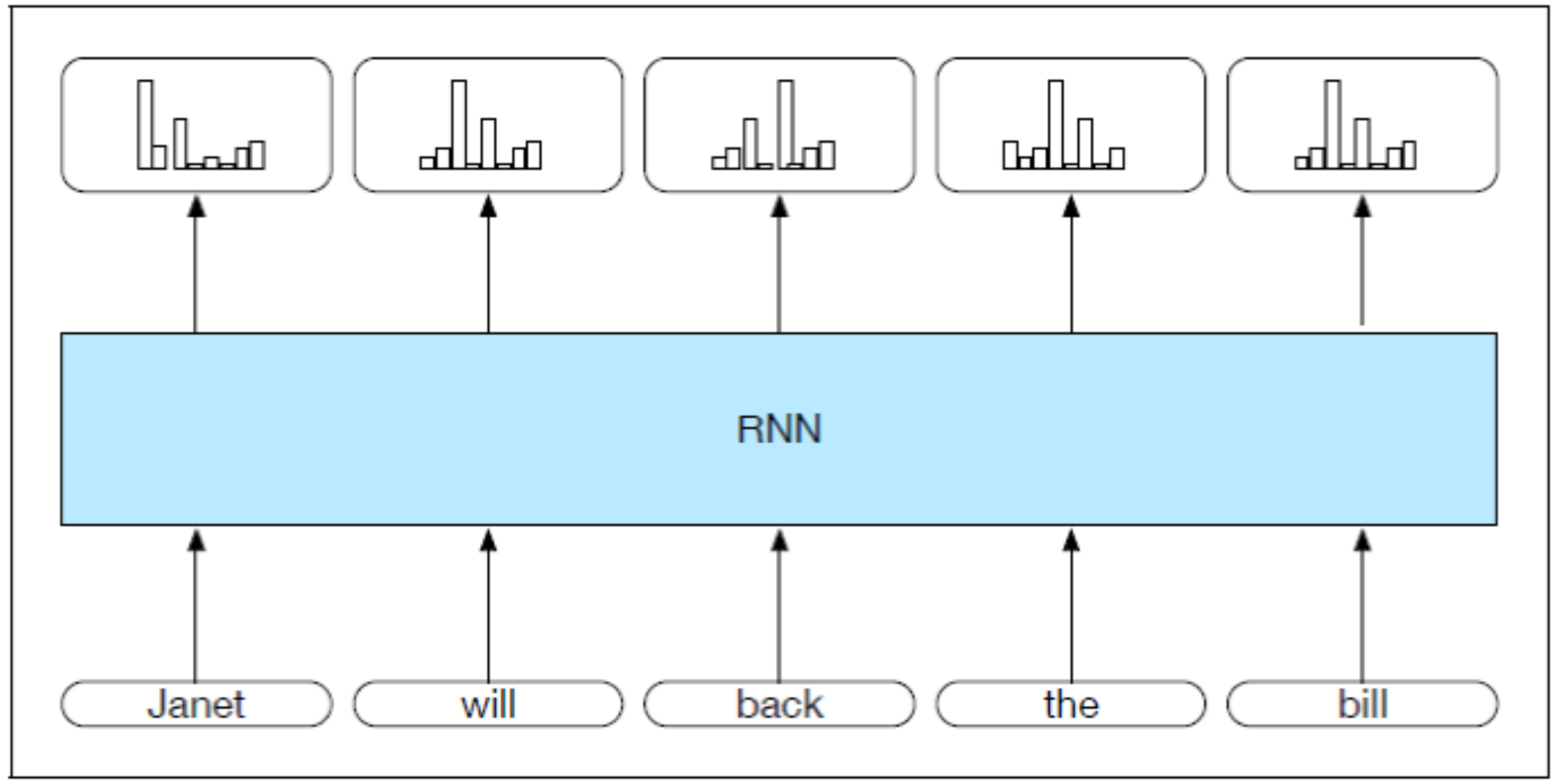
# A primer on transformer models

## Slide sources

- Jurafsky & Martin 3rd Ed (i.e., our textbook on NLP)
- https://jalammar.github.io/illustrated-transformer/

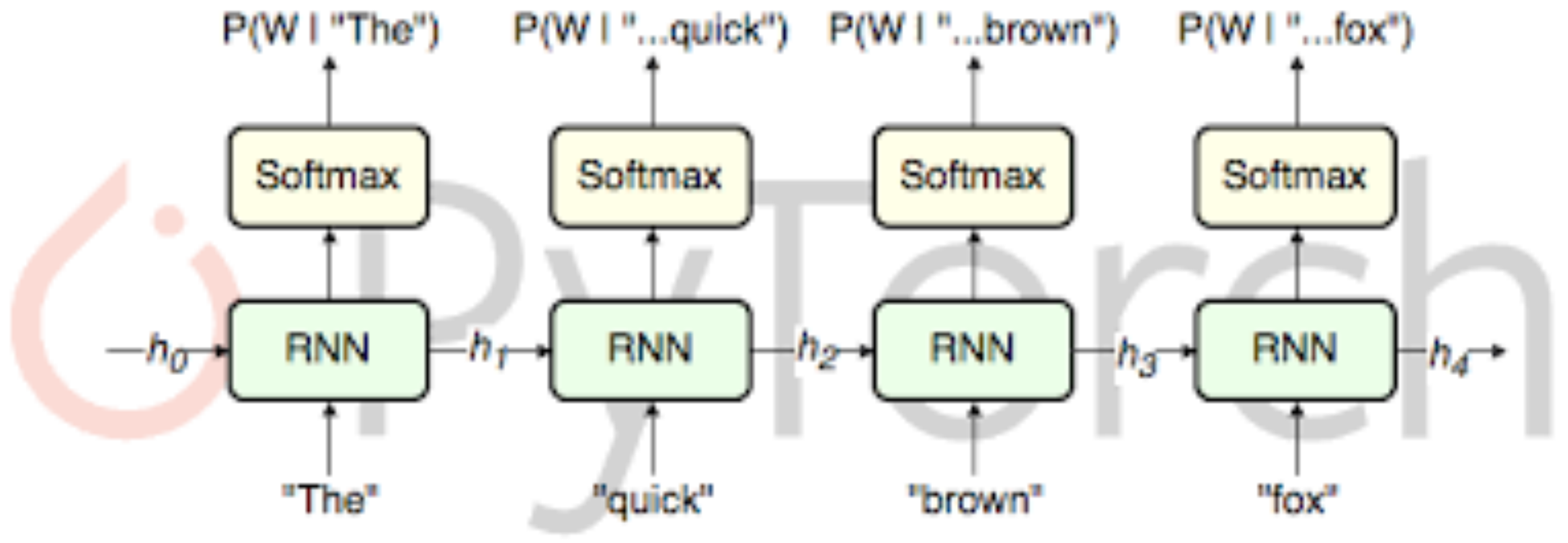
# Encoder-Decoder

**RNN:** input sequence is transformed into output sequence in a one-to-one fashion.

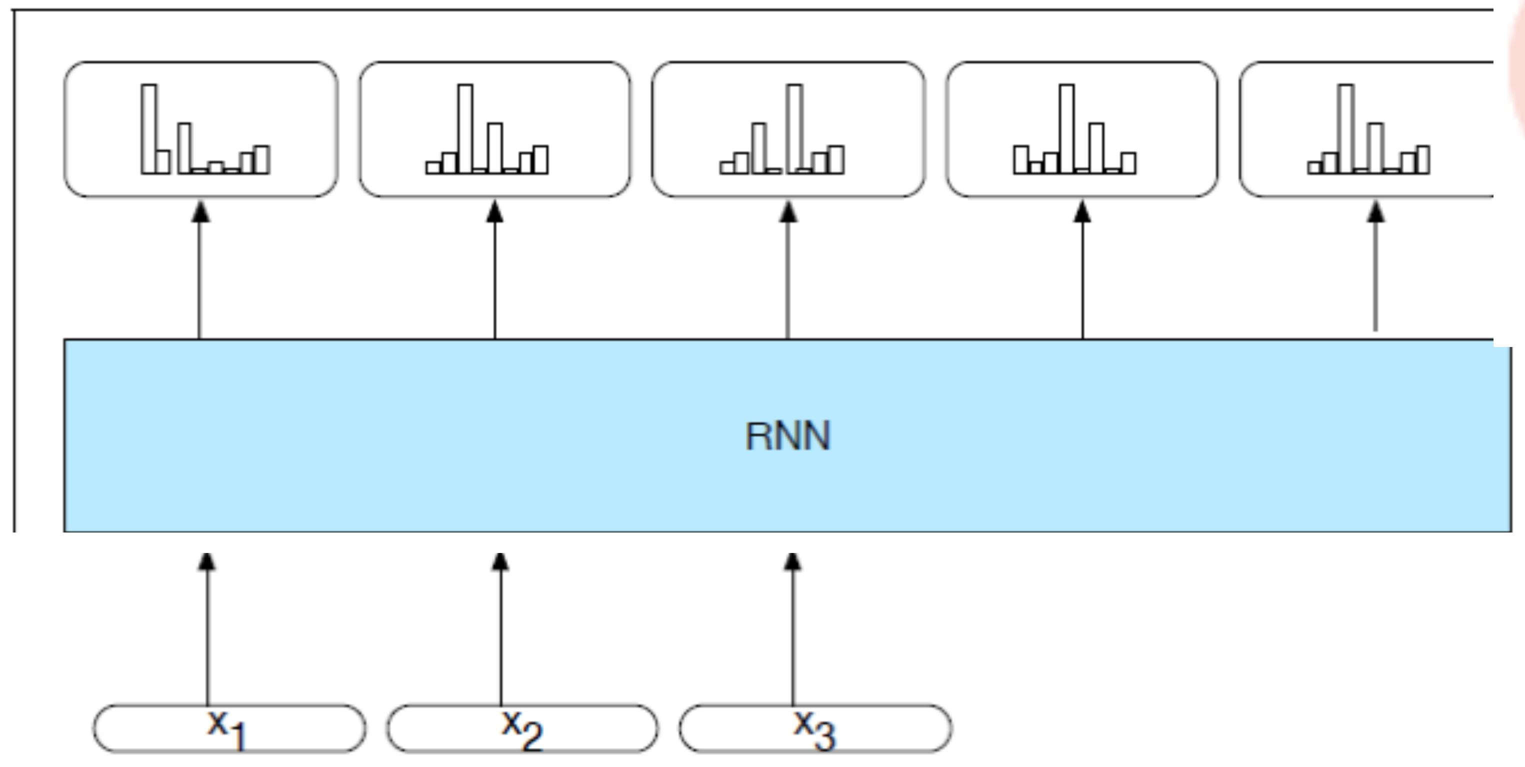


- **Goal:** Develop an architecture capable of generating *contextually appropriate*, *arbitrary length*, output sequences
- **Applications**:
  - Machine translation,
  - Summarization,
  - Question answering,
  - Dialogue modeling.

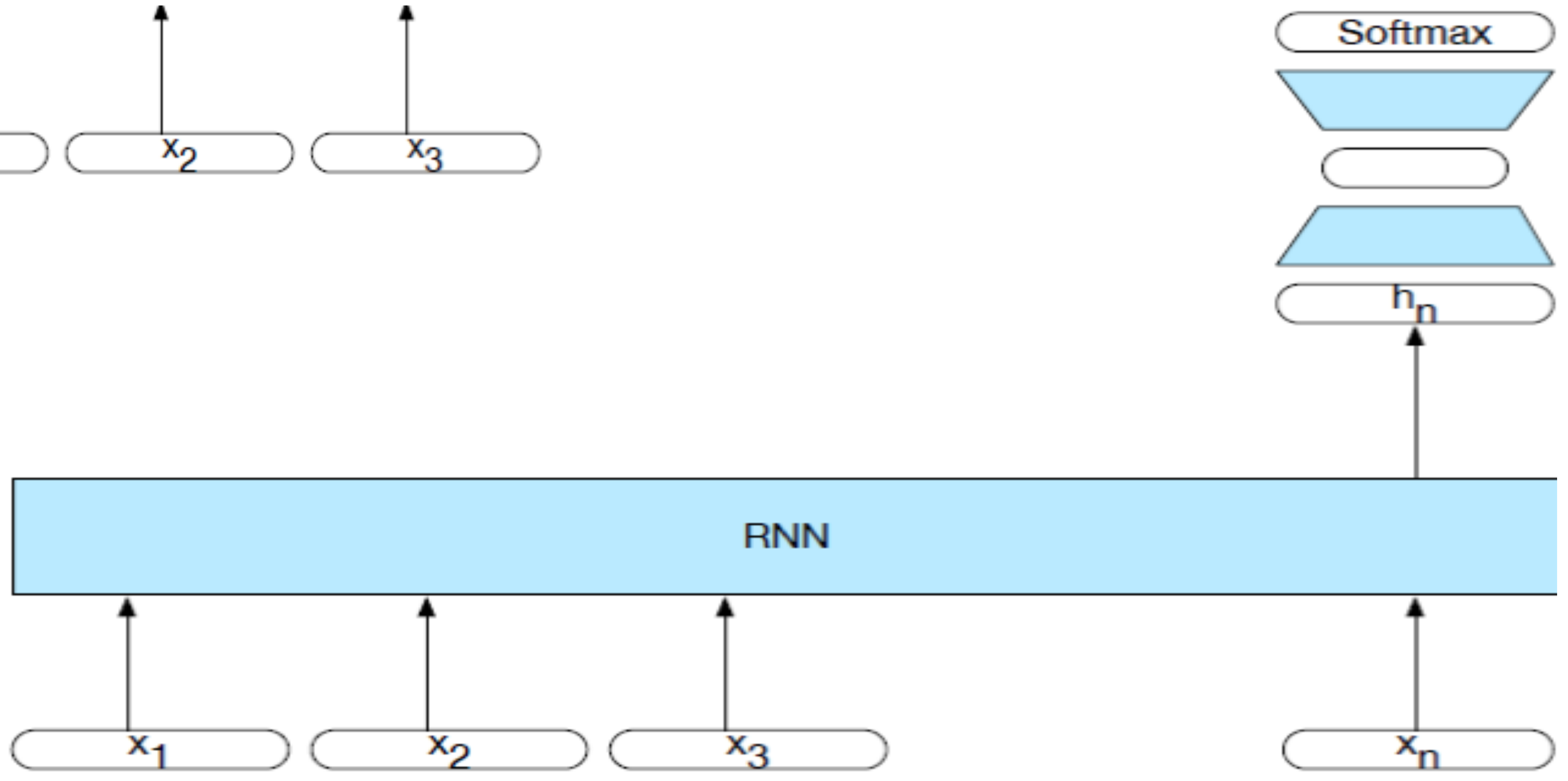
# RNN Applications



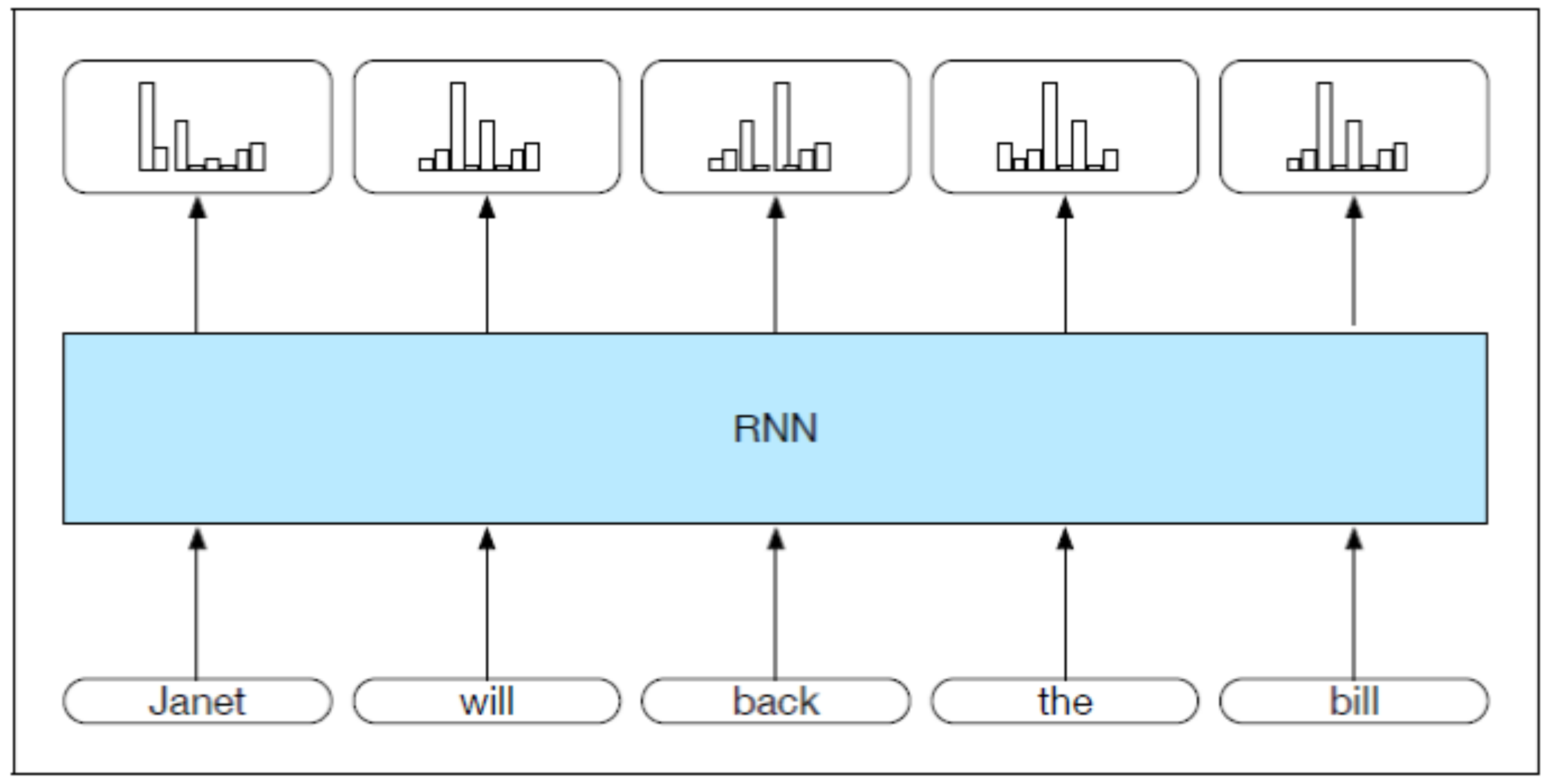
- Language Modeling



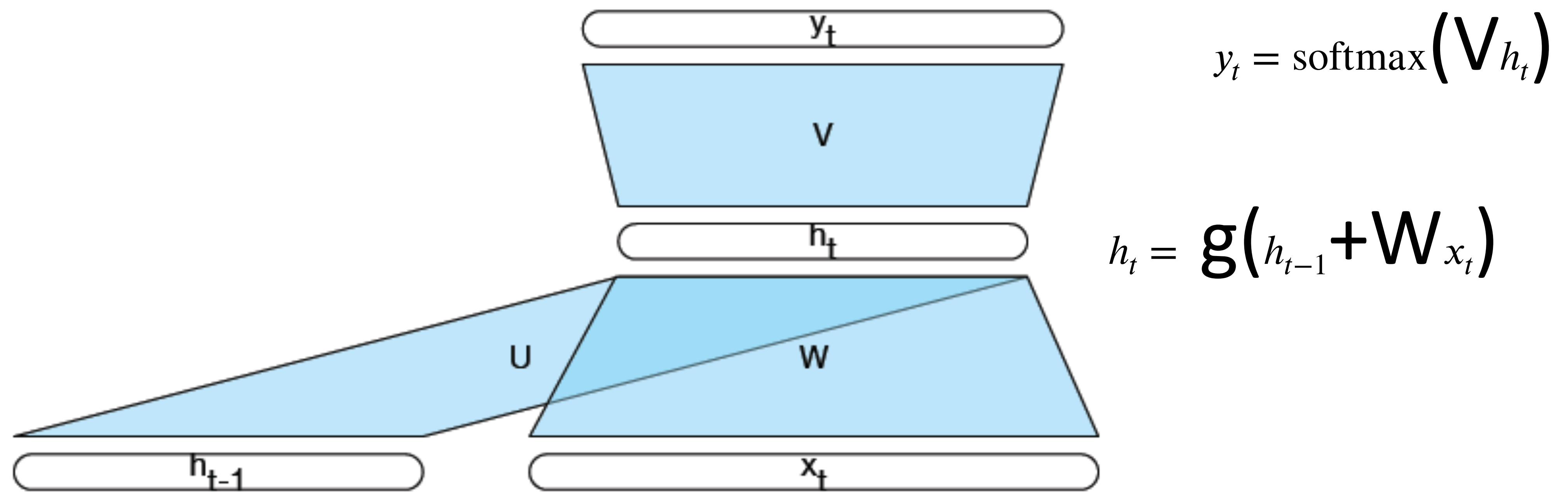
- Sequence Classification (Sentiment, Topic)

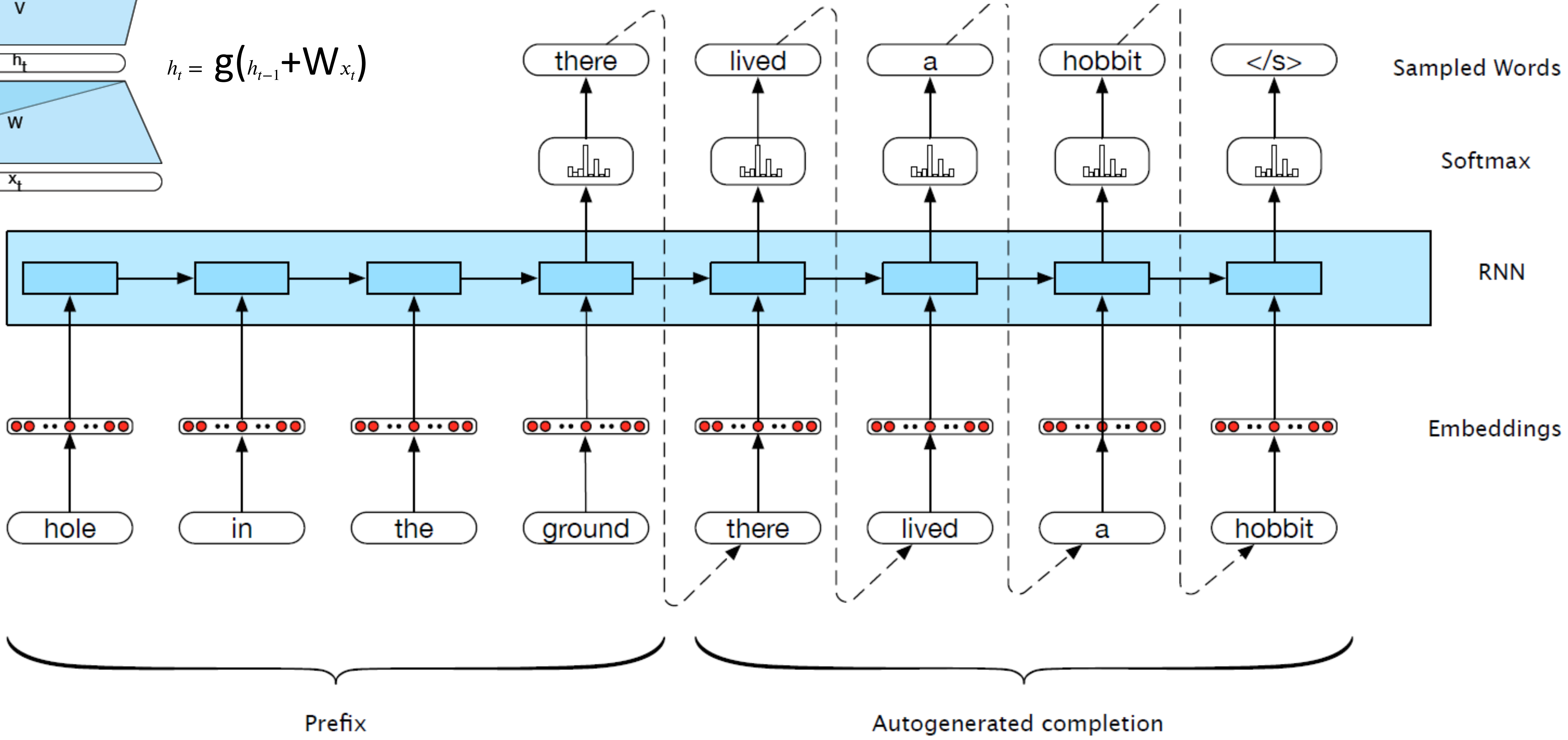


- Sequence to Sequence



# Sentence Completion using an RNN





- **Trained Neural Language Model** can be used to generate novel sequences
- Or to **complete** a given sequence (until end of sentence token <\s> is generated)

# Autoregressive generative models

- Generates sequences of outputs, one by one (e.g., sequences of words)
- Each output element (e.g., word) is generated by keeping into account the previously generated outputs (
  - From this the name “autoregressive”
- A regression model produces an output as a combination of inputs

$$y = a_1x_1 + a_2x_2 + \ldots + a_nx_n + \beta$$

- An autoregressive model produces an output only in terms of previous outputs

$$y_t = a_1y_{t-1} + a_2y_{t-2} + \ldots + a_ny_{t-n} + \epsilon_t + \beta$$

# Extending (autoregressive) generation to Machine Translation

- Word generated at each time step is conditioned on word from previous step.
- Training data are parallel text e.g., English / French

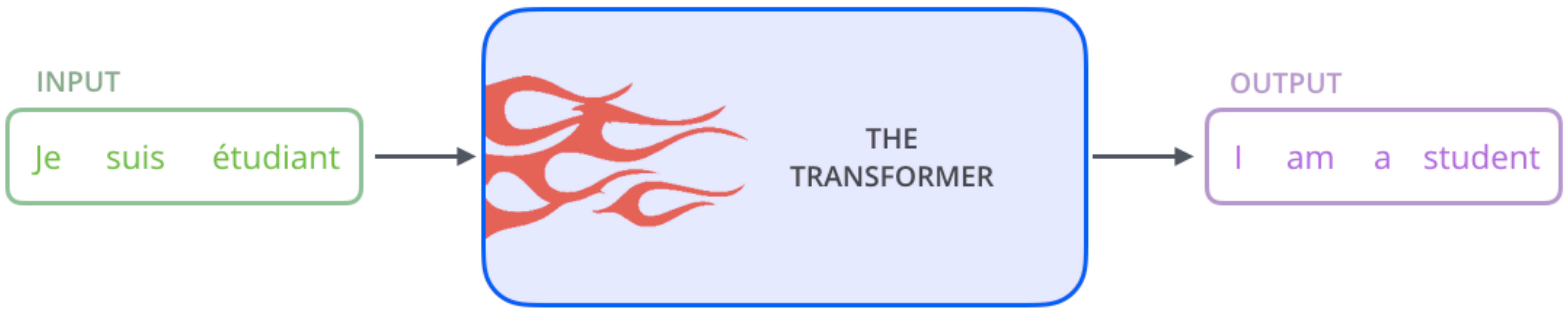
*there lived a hobbit* *vivait un hobbit*

……..

*there lived a hobbit* <\s> *vivait un hobbit* <\s>

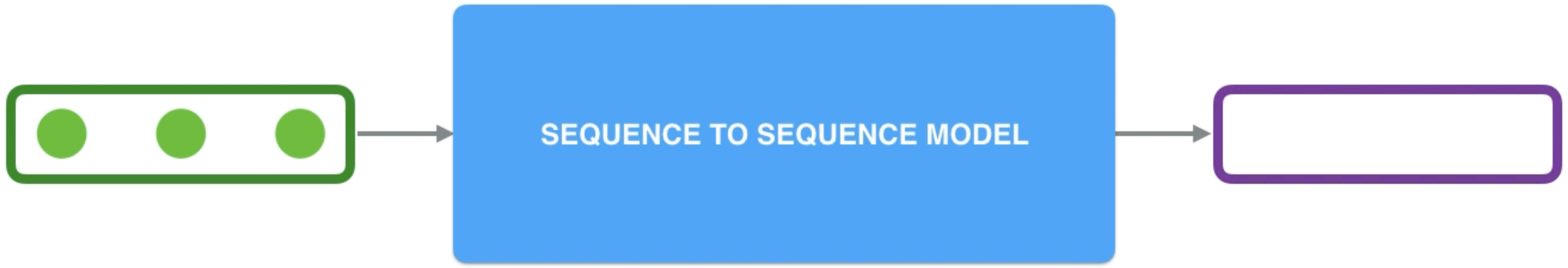
……..

# The transformer



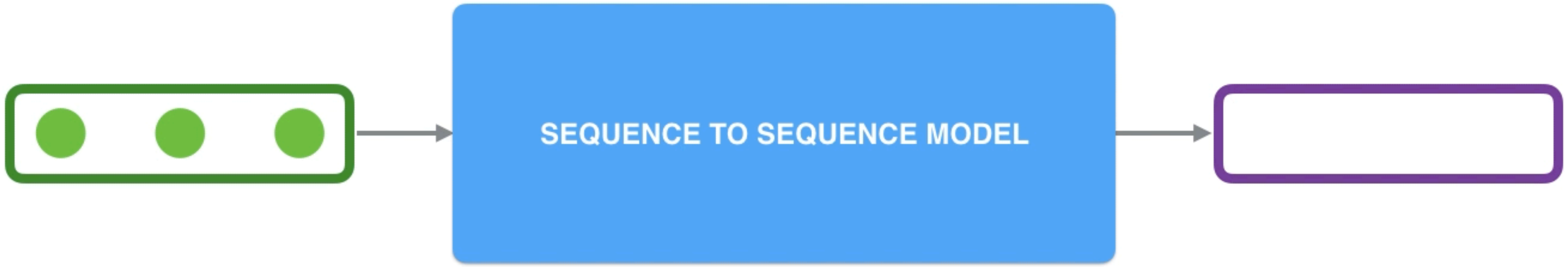
# Seq 2 seq model

Inputs to the network are processed one after the other, as a sequence:

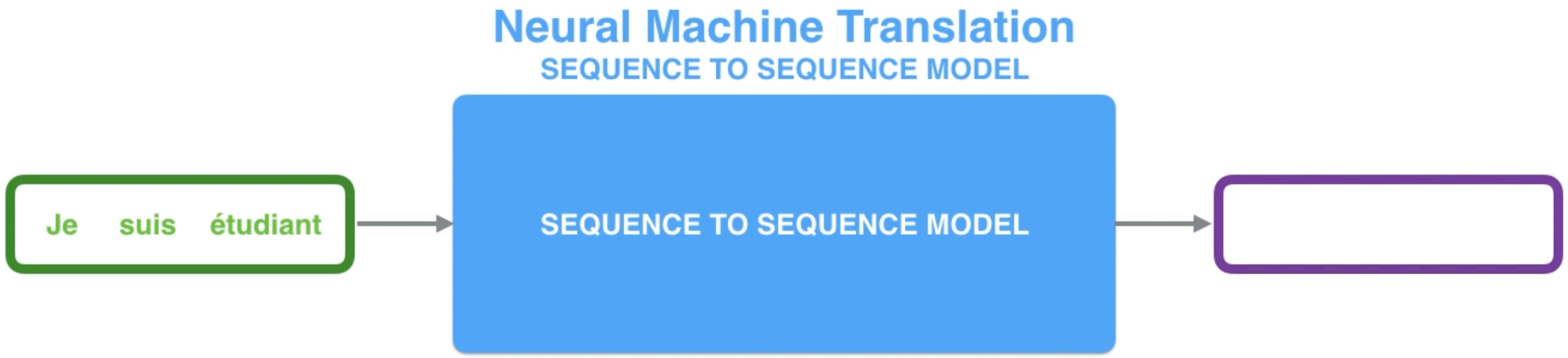


# Seq 2 seq model

Inputs to the network are processed one after the other, as a sequence:

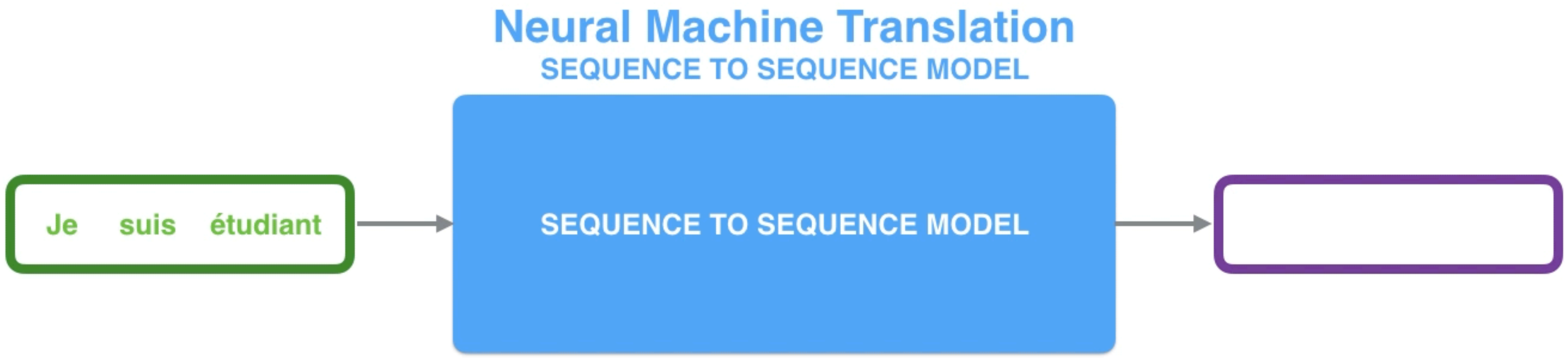


# Sentence translation…


Neural Machine Translation
SEQUENCE TO SEQUENCE MODEL
Je suis étudiant
SEQUENCE TO SEQUENCE MODEL

**What's behind that?**

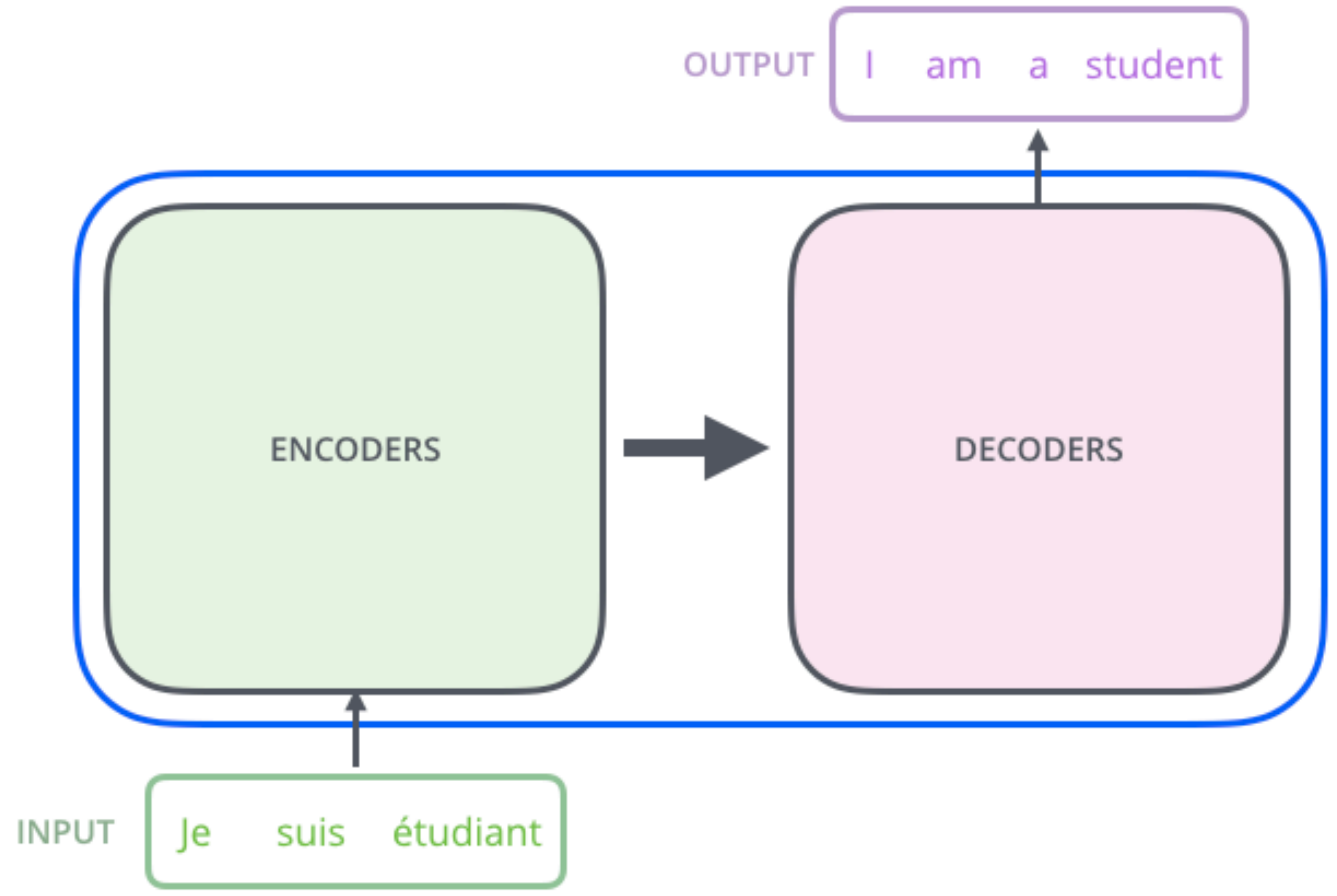
# Sentence translation…


Neural Machine Translation
SEQUENCE TO SEQUENCE MODEL
Je suis étudiant
SEQUENCE TO SEQUENCE MODEL

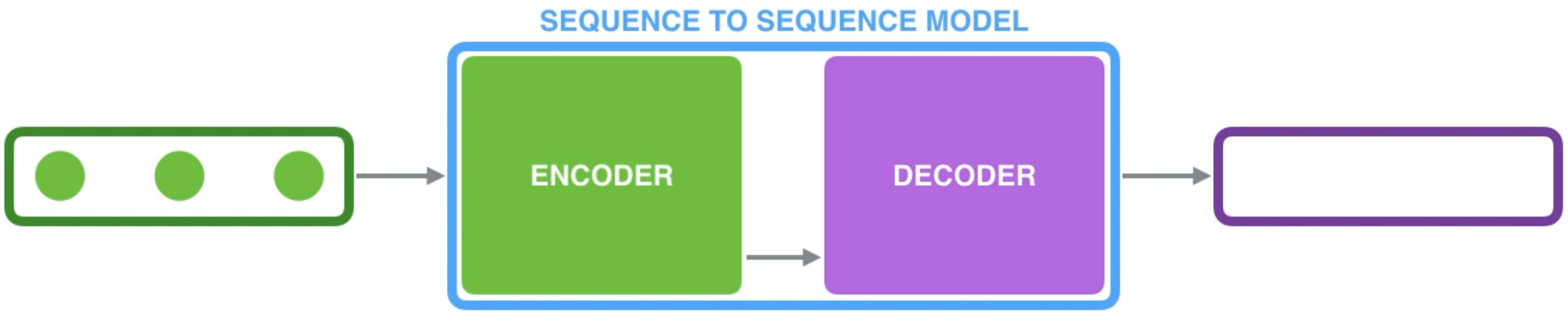
**What's behind that?**

# Encoder/Decoder Model

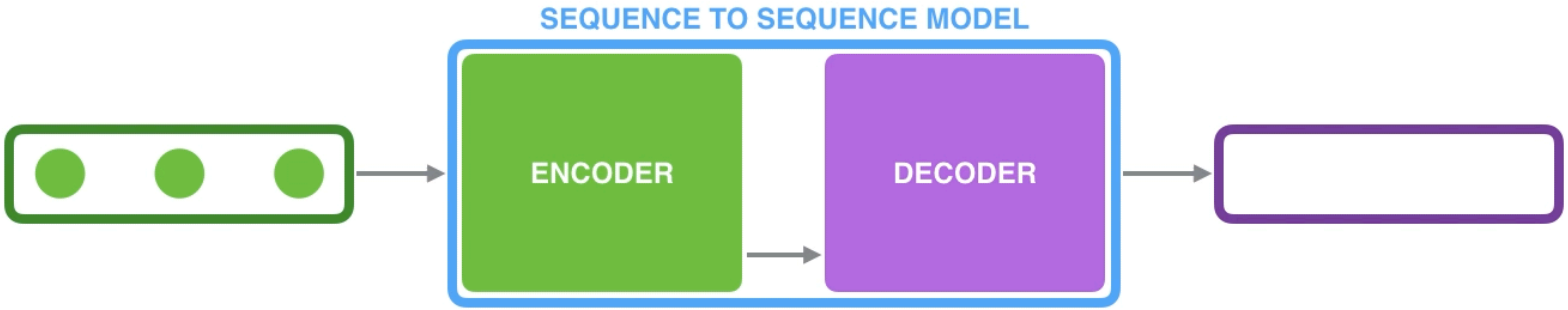
- The encoder processes the input sequence one by one
- The information is captured in a vector named context
- After processing the entire input sequence, the encoder sends the context to the decoder
- The decoder produces the output sequence item by item



# Visually…

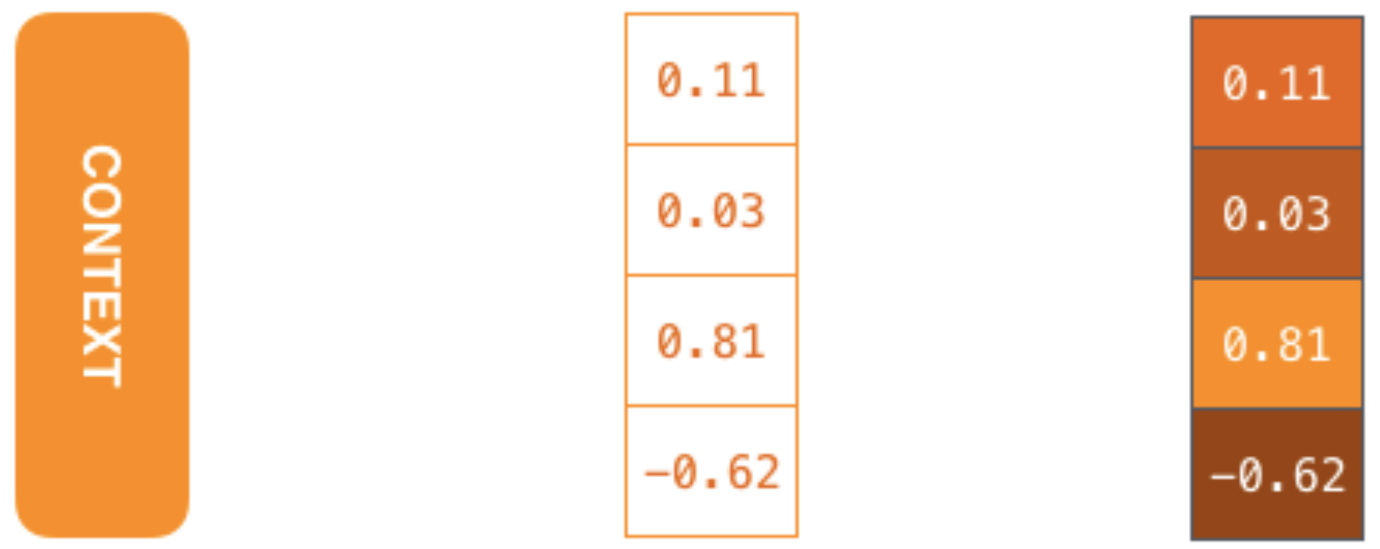


# Visually…

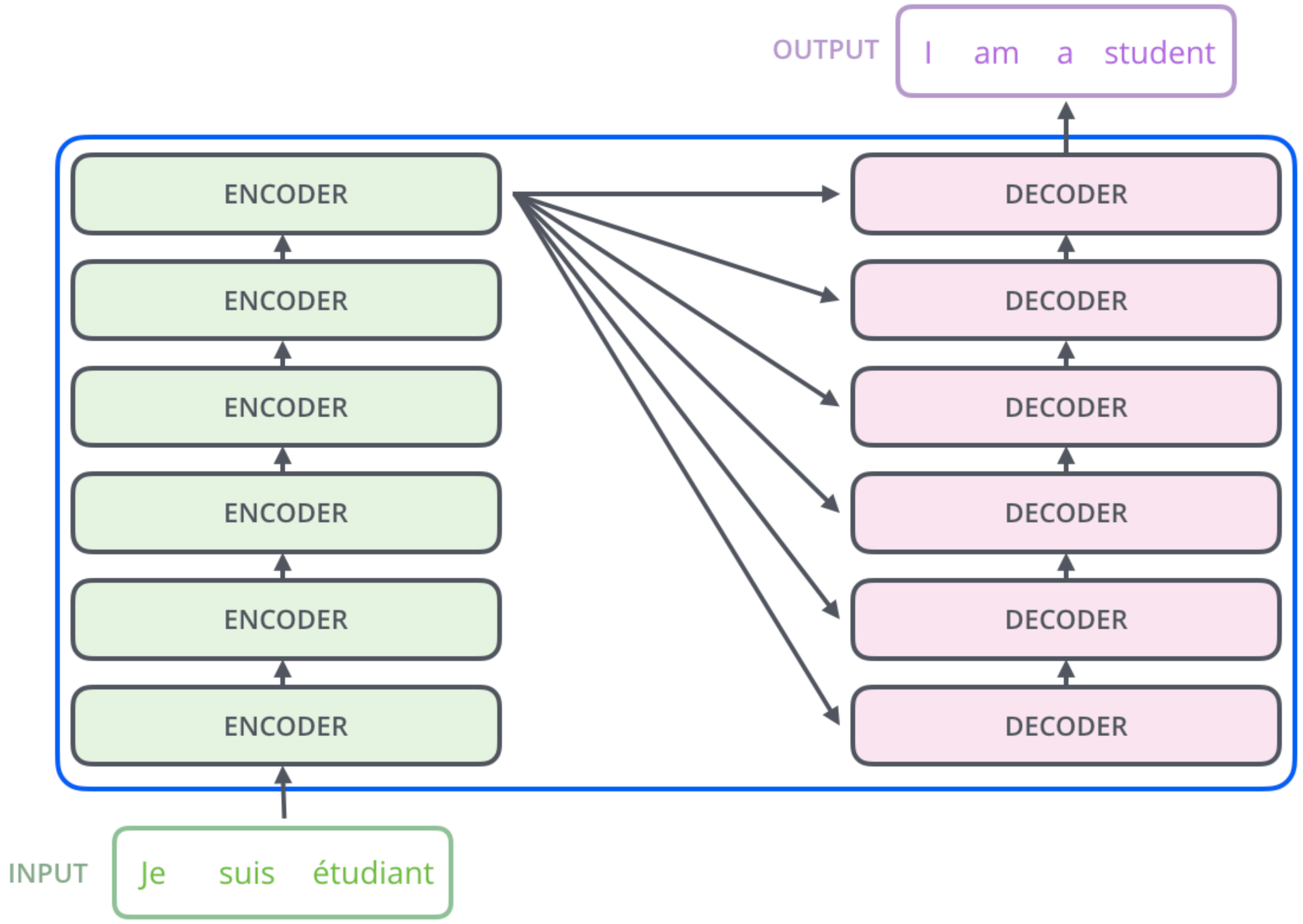


# Context, encoder, and decoder

- Encoder and decoder are recurrent neural networks composed of multiple stages
- The context is a vector, which size is equal to the number of stages in the encoder and decoder
- Realistically, 256, 512, 1024 stages



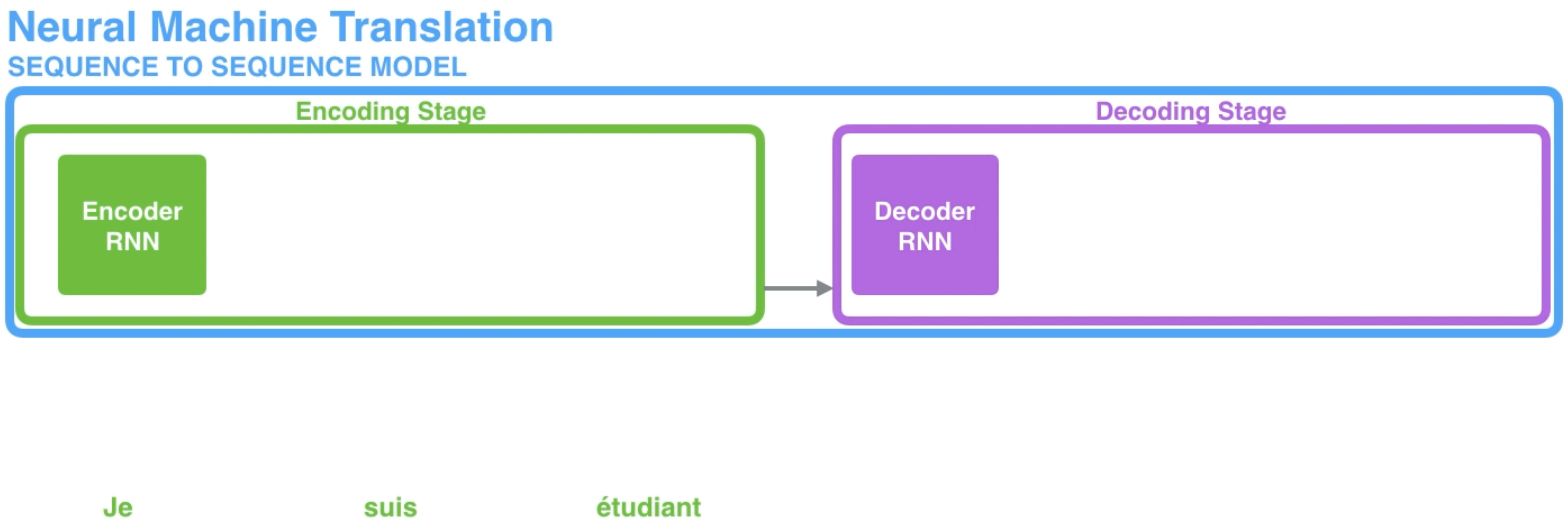
# Encoder and decoder



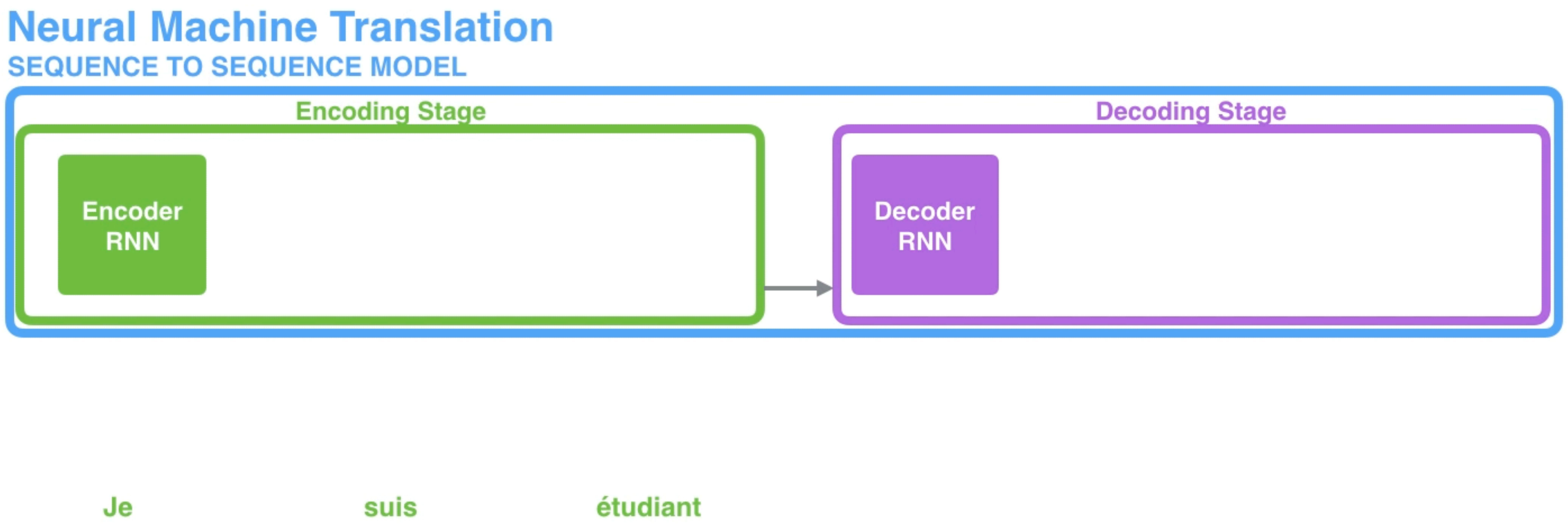
# How each RNN works

- It takes two inputs:
  - A word from the sentence
  - The hidden state
- Words are represented as embedding vectors
- The output of each RNN stage of the encoder (or decoder) is passed to the next one

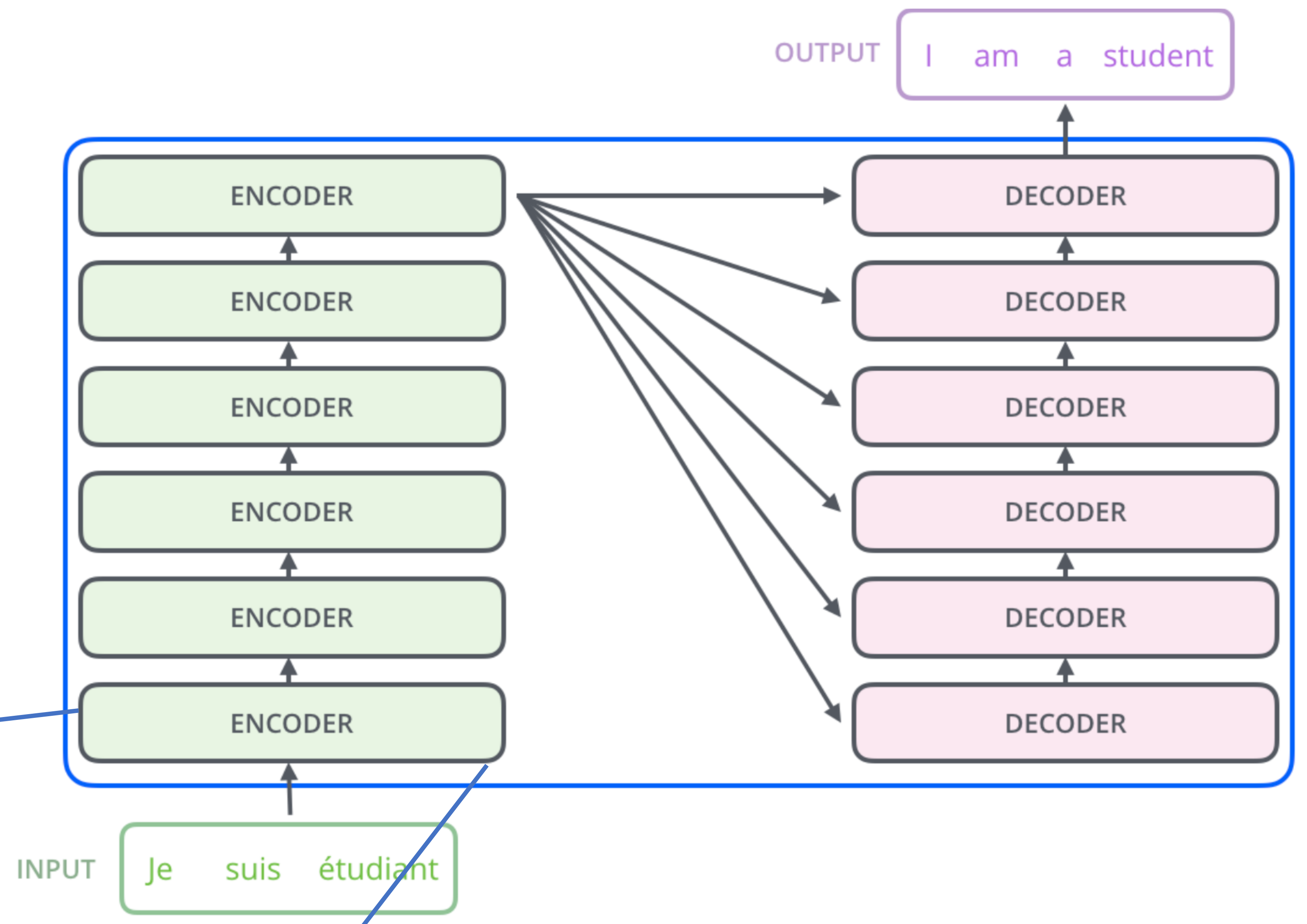
# Visually…

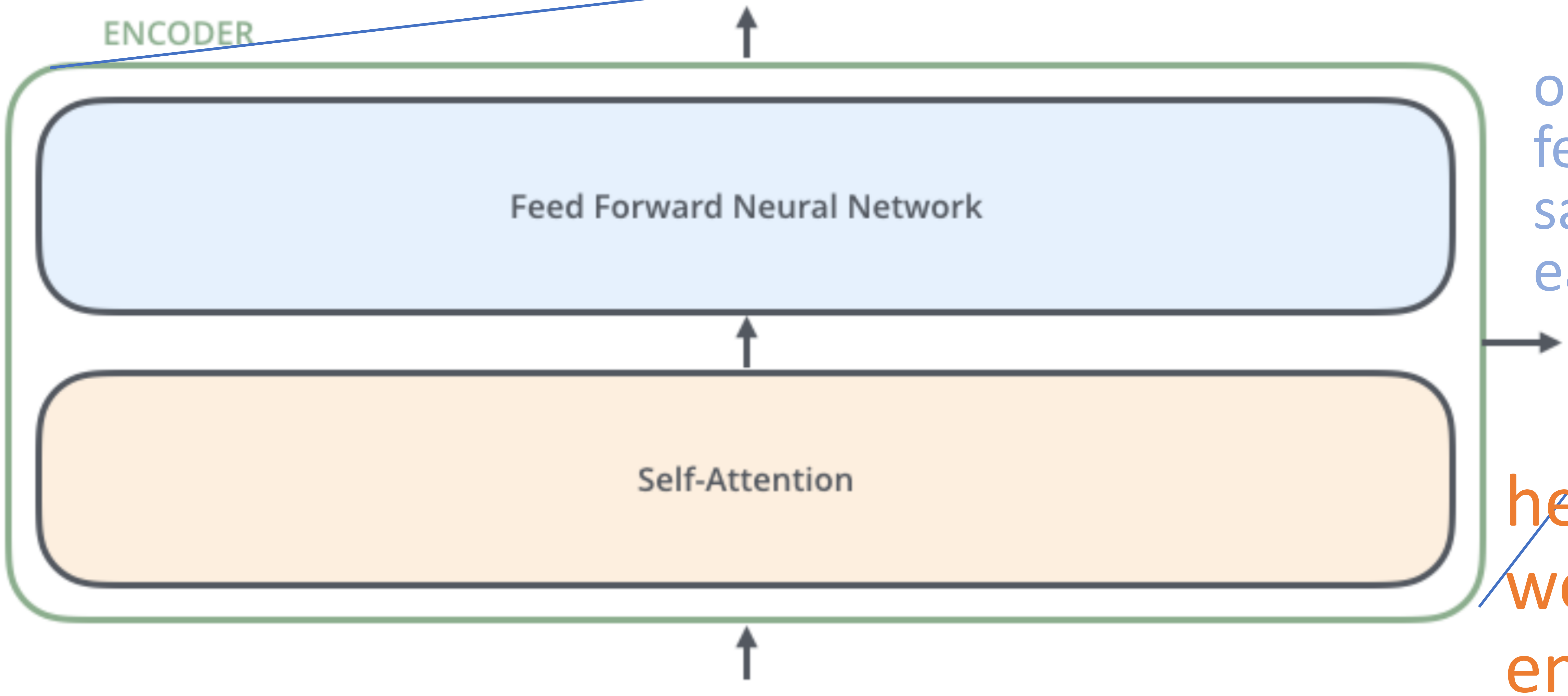

Neural Machine Translation
SEQUENCE TO SEQUENCE MODEL
Encoding Stage
Decoding Stage
Encoder RNN
Decoder RNN
Je
suis
étudiant

# Visually…


Neural Machine Translation
SEQUENCE TO SEQUENCE MODEL
Encoding Stage
Decoding Stage
Encoder RNN
Decoder RNN
Je
suis
étudiant

The **encoders** are **all identical in structure** (yet they do not share weights). Each one is broken down into two sub-layers



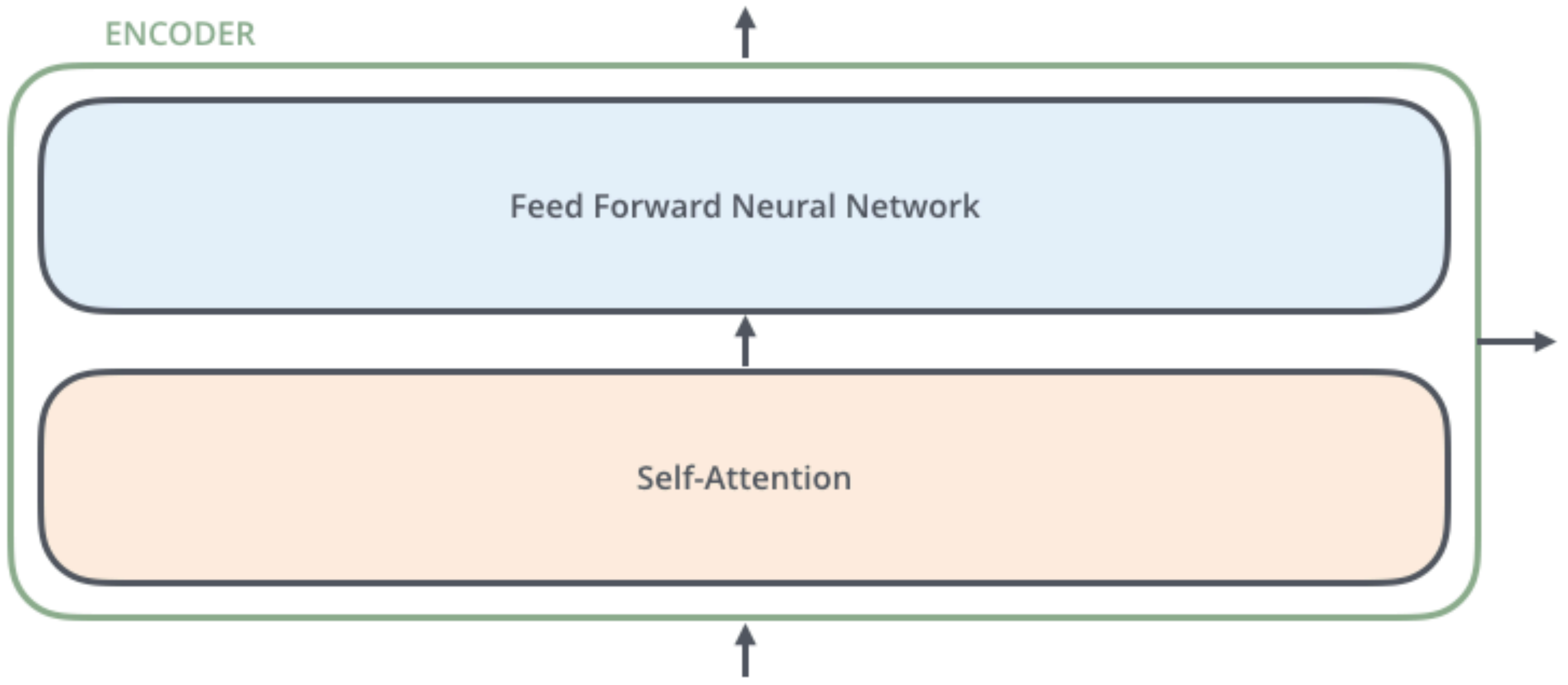


outputs of the self-attention are fed to a feed-forward neural network. The exact same one is independently applied to each position.

helps the encoder look at other words in the input sentence as it encodes a specific word.

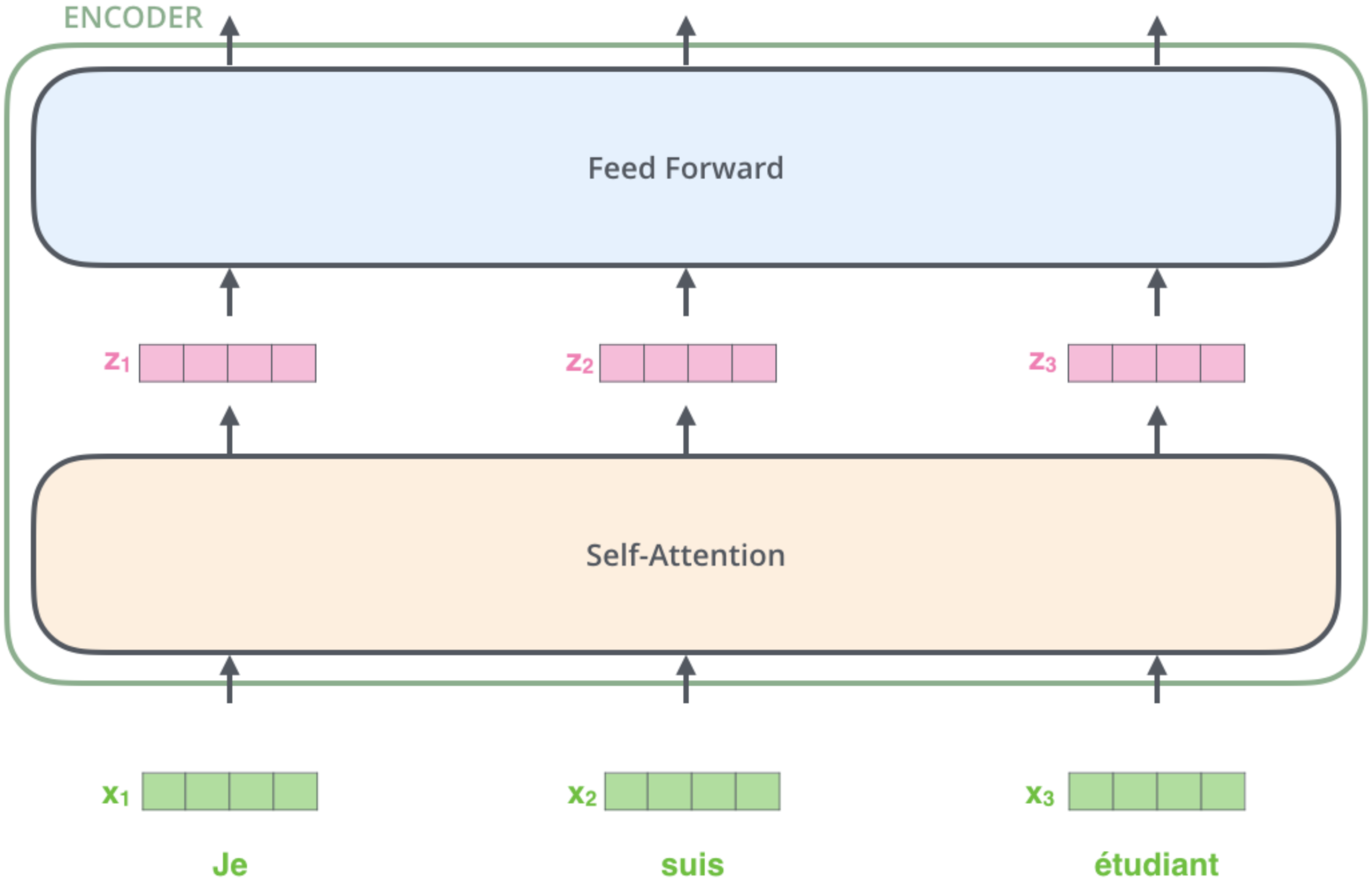
## How the encoder works: the attention layer

Self-attention layer: a layer that helps the encoder look at other words in the input sentence as it encodes a specific word.



**Key property of Transformer**: word in each position flows through its own path in the encoder.

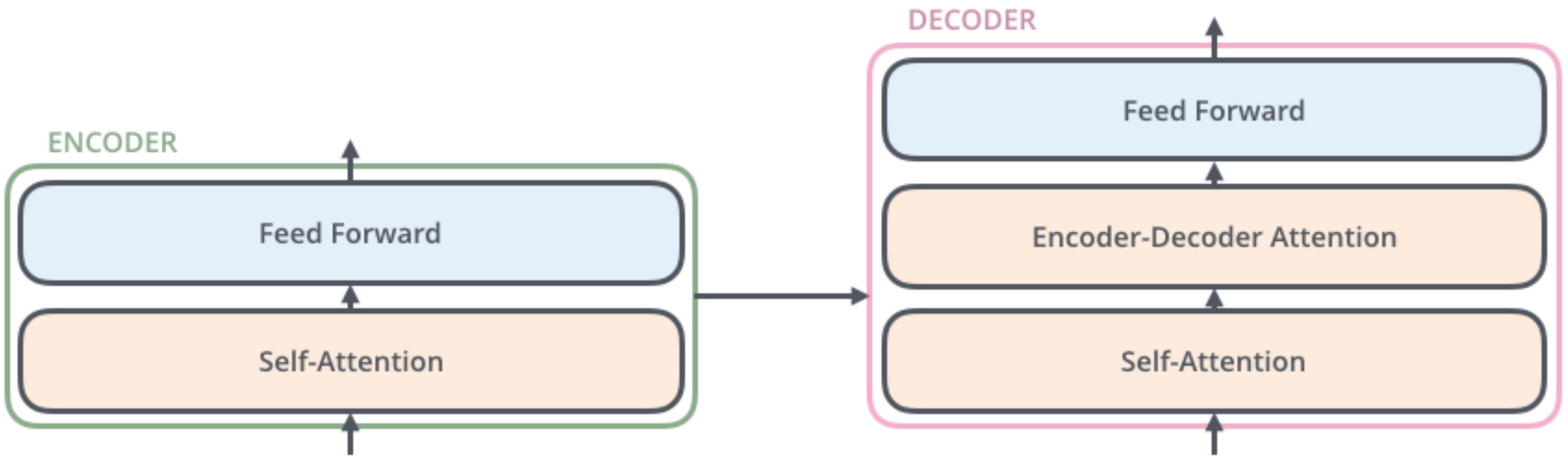
- There are dependencies between these paths in the self-attention layer.
- Feed-forward layer does not have those dependencies => various paths can be executed in parallel !



**Word embeddings**

# Decoder attention layers

- The output of the self attention are forwarded to a feed forward stage and to the decoder
- The decoder has both those layers, but between them is an attention layer that helps the decoder focus on relevant parts of the input sentence



# Recap

An attention model differs from a classic sequence-to-sequence model:

- The encoder passes a lot more data to the decoder (it passes all the hidden states)

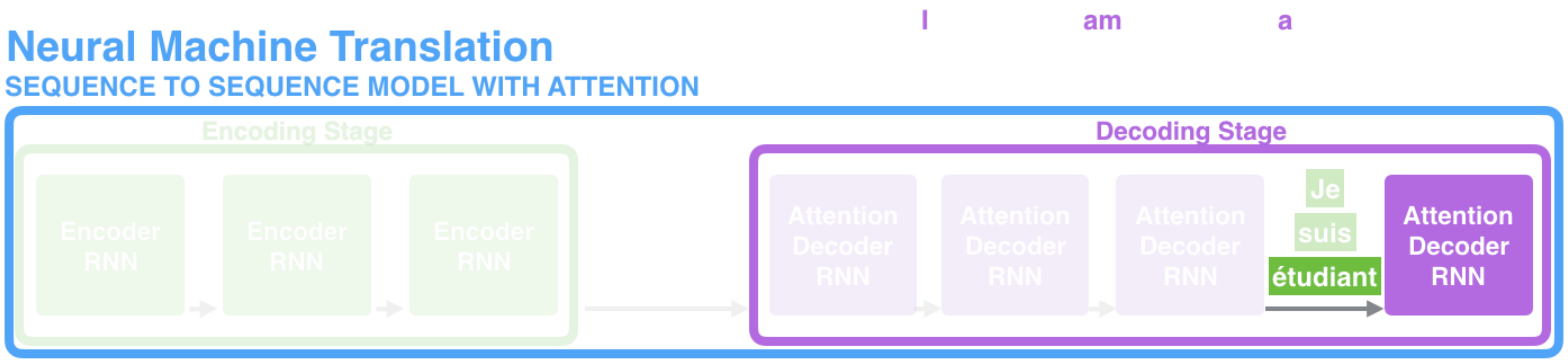
- An attention decoder does an extra step before producing its output. In order to focus on the parts of the input that are relevant to this decoding time step, the decoder does the following:

  1. Looks at the set of encoder hidden states it received – each encoder hidden state is most associated with a certain word in the input sentence

  2. Gives each hidden state a score

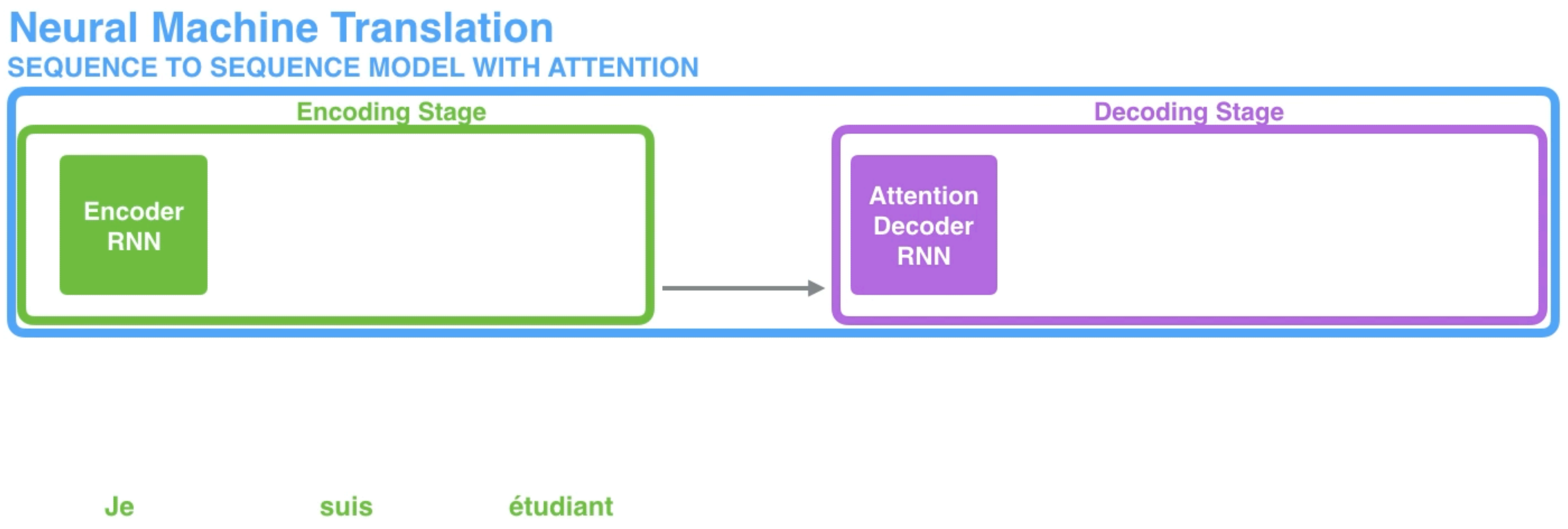
  3. Multiplies each hidden state by its softmaxed score, thus amplifying hidden states with high scores, and drowning out hidden states with low scores

# Attention model

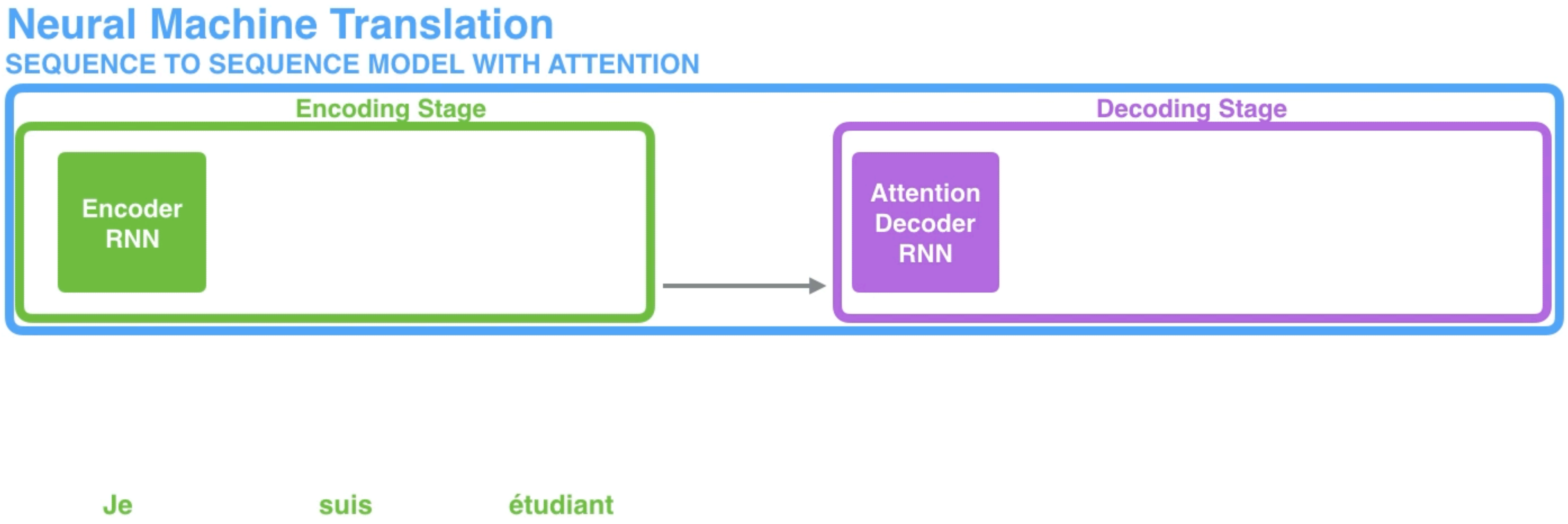


**The attention model let the decoding stage focus on the “étudiant” word when performing the translation**

# Visually…

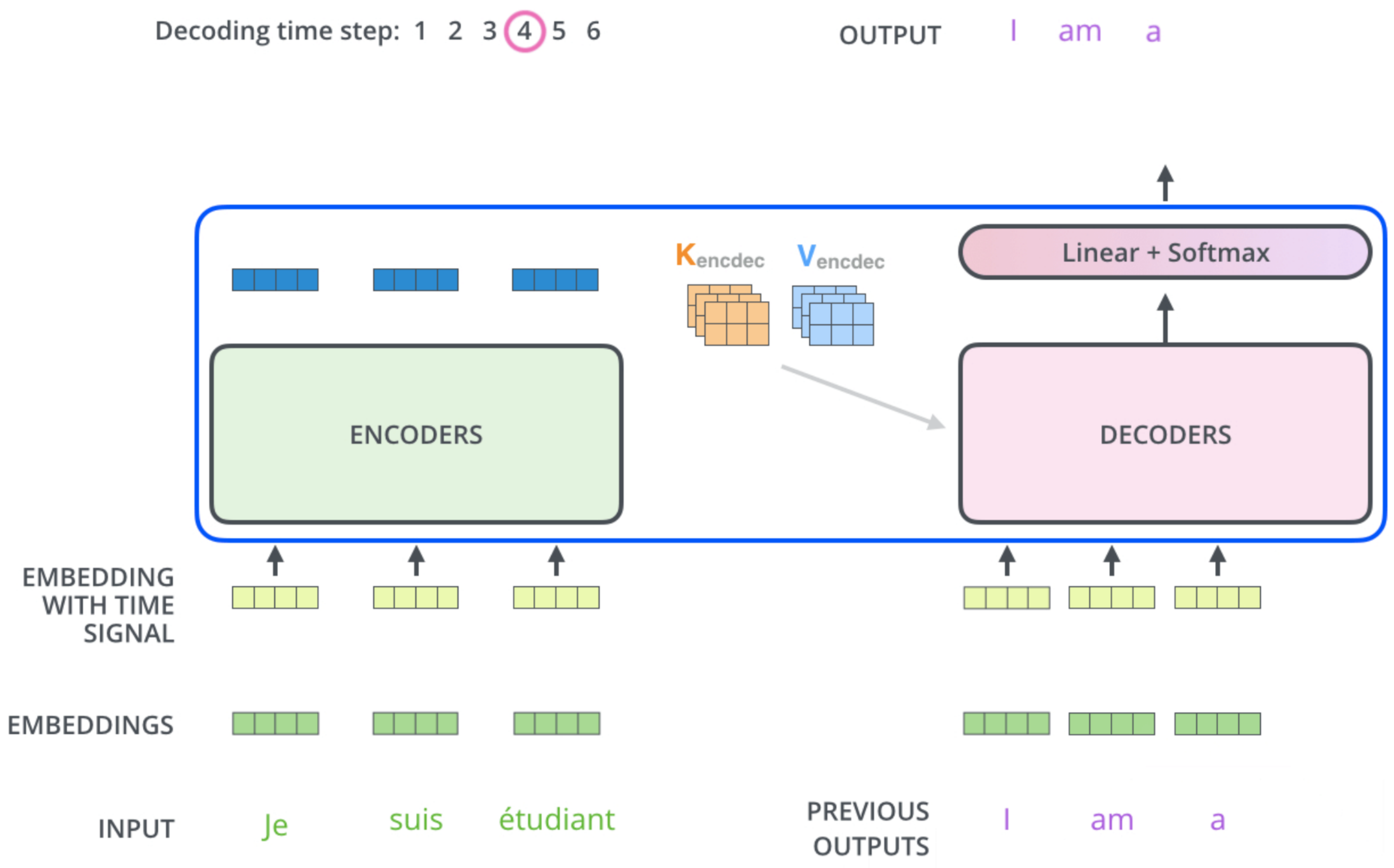

Neural Machine Translation
SEQUENCE TO SEQUENCE MODEL WITH ATTENTION
Encoding Stage
Decoding Stage
Encoder RNN
Attention Decoder RNN
Je
suis
étudiant

# Visually…


Neural Machine Translation
SEQUENCE TO SEQUENCE MODEL WITH ATTENTION
Encoding Stage
Decoding Stage
Encoder RNN
Attention Decoder RNN
Je
suis
étudiant

# The Decoder Side

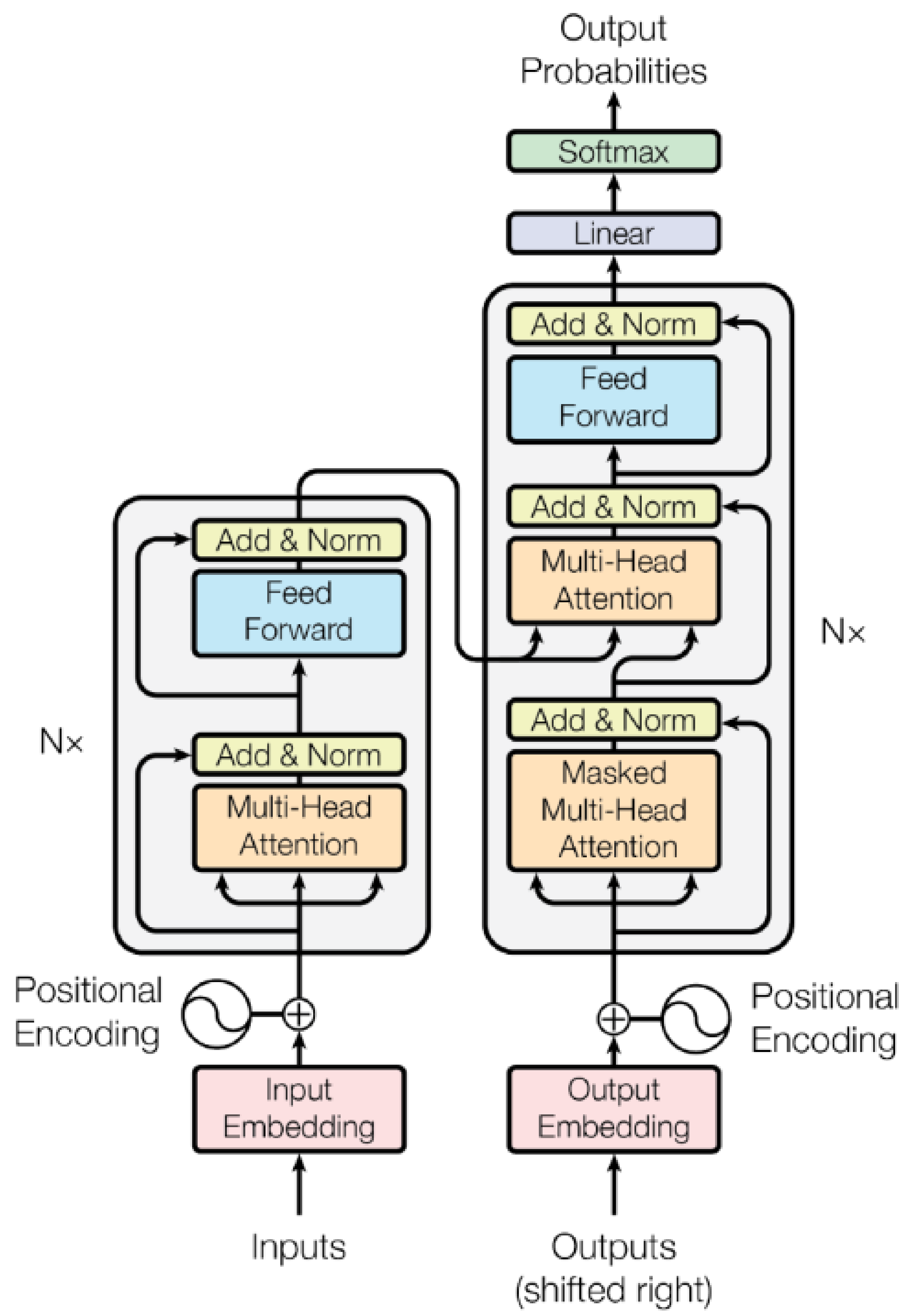
Relies on most of the concepts on the encoder side



## Decoder - Discussion

- Generates the text word by word
- The generation is (in theory) done using a softmax layer that encodes the probability for every word in the vocabulary
  - In practice this process is optimized
- Upon performing the generation, the decoder looks at previously generated words, and takes them into account to generate the next one

# Trasformer Full Picture (from the original paper)



# Not only encoder-decoder

- Not all transformers have both the encoder and the decoder phase
- Some models can be
  - Decoder-only
  - Encoder-only
- This does not necessarily mean one is better than the other!

# Encoder-only models

- Use all the available nodes to encode the knowledge from the input
- Useful when the task requires complex analyses of the relationships between words in the inputs, yet the output is relatively simple to generate
- As in encoder-decoder models, the encoder captures the knowledge from the input and stores it in an embedding

# Encoder-only example: BERT

- **Pre-training of Deep Bidirectional Transformers for Language Understanding**
  - https://huggingface.co/docs/transformers/en/model_doc/bert
- Developed by Google
- Trained through masking and next-sentence prediction
- Therefore, they can be applied for tasks whether the model expects to replace the masking token with something
- Problems for which it is well-suited:
  - Text classification (e.g., sentiment analysis)
  - Word prediction

# Decoder-only models

- Use most of the knowledge for the generation tasks
- Take a sequence of tokens and predict the next token in the sequence, step by step, by creating sentences/paragraphs
- Particularly suited for generating complex, arbitrarily long sequences from an input (prompt)
- Autoregressive generation:
  - The model predicts each token based on the previous tokens.
  - This happens smoothly ensuring the generation of coherent sentences

# Decoder-Only Models: GPT

- **GPT: Generative Pretrained Transformer**
  - Developed by OpenAI
  - Version 2 is openly available
  - GPT 3 vas released as an API and with a chatbot (ChatGPT) on November 30, 2022
  - Last versions: GPT 4 (2023), GPT 4-o (2024)
- Applications:
  - Generating long and complex text
  - Instantiating colloquial conversations

# Notes:

When choosing the models, you need to consider:

- Its size, in terms of parameters, that may make the model more powerful but also more expensive to fine-tune (GPUs are almost always needed)
- Its context window in terms of input+output tokens
    - T5-base has a context of 512 tokens only
    - GPT3: 4k tokens
    - GPT 4 turbo, GPT 4-o: 128k tokens

## Transformers - Hands-on

## Main idea and sources

- We will mainly reuse pre-trained models from HuggingFace: https://huggingface.co/
- Where needed, we will fine-tune them
- Sometimes, we will just play with pre-trained models as they are
- Note, HuggingFace can be used with two backends, TensorFlow and Torch
    - We will use Torch which is simpler, yet tutorials show how to use TensorFlow as well (which you mostly know already)

# What is HuggingFace?

- Repository of:
    - Pretrained models (you can search for models for all purposes)
    - Datasets for training/finetuning
    - Convenient high-level APIs that facilitate a lot your work
- Also, once you have created your model, you can push it on HuggingFace
    - Just matter of creating your own account and token on the HuggingFace portal

## What is a pretrained model?

- Model already trained
  - on a very large corpus
  - leveraging conspicuous hardware resources
- Instead of retraining the model from scratch we can:
  - Import the model and use it “as is” (works just fine for some tasks)
  - Fine-tune the model for our specific task

# How are language transformer models pre-trained?

- Supervised training:
  - Done by actually giving several input/output pairs
  - OpenAI has large teams of “trainers” that create them for their models
- Unsupervised denoising training:
  - Trains a “language model” on a text corpus
  - It works by masking text elements on the input and providing the solutions on the output

## Unsupervised denoising training

- The cute dog walks in the park

## Unsupervised denoising training

- The cute dog walks in the park

# Unsupervised denoising training

- The cute dog walks in the park
- Get transformed as follows:
    - input: ”The <extra_id_0> walks in the park”
    - output: ”<extra_id_0>cute dog<extra_id_1>”

# Unsupervised denoising training

- The cute dog walks in the park
- Get transformed as follows:
    - input: ”The <extra_id_0> walks in the park”
    - output: ”<extra_id_0>cute dog<extra_id_1>”
- Where <extra_id_0> is a special token indicating the masked text
- In the output, we add a further special token <extra_id_1> indicating the end of the masked text

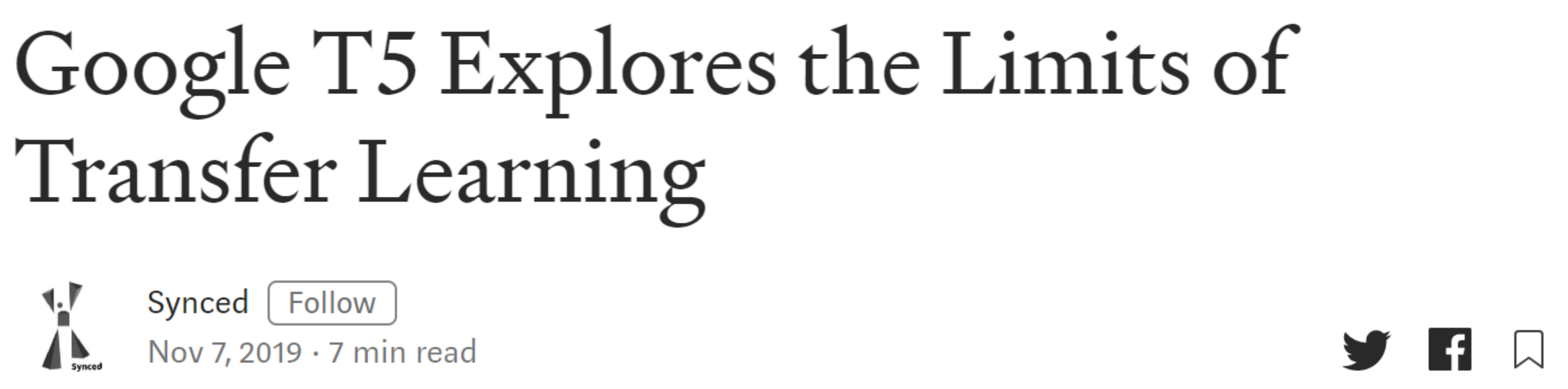
# Similarly, with multiple fragments being masked…

- The cute dog walks in the park
- Get transformed as follows:
  - input: ”The <extra_id_0> walks in <extra_id_1> park”
  - output: ”<extra_id_0>cute dog<extra_id_1>the<extra_id_2>”

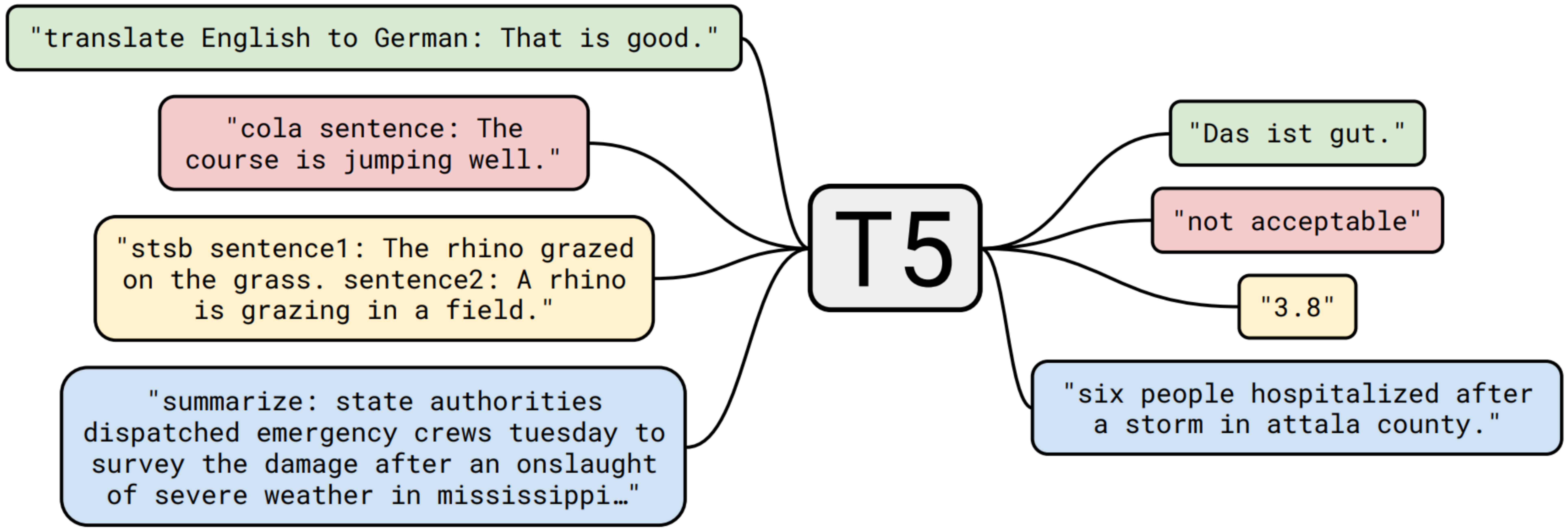
## What’s great here?

- We can just use tons of text and automatically train a language model
- Similar to what we have done for the n-gram models at the beginning of the course

# Model Example: T5


Google T5 Explores the Limits of Transfer Learning

Synced Follow
Nov 7, 2019 · 7 min read


"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

## T5 Model

- Stands for “Text to Text Transfer Transformer”
- Developed by Google AI
- Presented in the paper:
  Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer by Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu.
- Refer to HuggingFace:
  https://huggingface.co/docs/transformers/model_doc/t5

## T5 model description

- T5 is an encoder-decoder model pre-trained on a multi-task mixture of unsupervised and supervised tasks and for which each task is converted into a text-to-text format.
- Different sizes (to load them we just specify one of these names):
  - t5-small
  - t5-base
  - t5-large
  - t5-3b
  - t5-11b

# T5 model: applications

- Any sequence-to-sequence translation
  - I will explain later how sequence to sequence translation works
- The model has been specifically pretrained for some tasks
- In this case, to make it working properly, we need to prepend the input with a “Prompt”
  - For translation: “translate English to German: “
  - For summarization: “summarize: “

## Let’s apply T5 for a simple task…

# Task example: text summarization

- Goal: given an article (or abstract), automatically generate its title
- We will use a T5 model pretrained on the English
- Note: T5 has been already pretrained for many tasks, including:
  - Language translation
  - Summarization
  - Text completion
- We will reuse an existing dataset, but it is very easy to load data from json or csv files through the “dataset” library of HugginFace

## Types of summarization

- Summarization can be:
    - Extractive: extract the most relevant information from a document.
    - Abstractive: generate new text that captures the most relevant information.
- We will focus on the second one which is most challenging and can be achieved with T5 models

## Libraries to install

```
pip install transformers datasets evaluate rouge_score torch
accelerate
```

## Loading online dataset

```
from datasets import load_dataset

billsum = load_dataset("billsum", split=“train")
```

# Split the dataset into training and test

```
billsum = billsum.train_test_split(test_size=0.2)
billsum
```

```
DatasetDict({
    train: Dataset({
        features: ['text', 'summary', 'title'],
        num_rows: 15159
    })
    test: Dataset({
        features: ['text', 'summary', 'title'],
        num_rows: 3790
    })
})
```

## Inspect one sample...

```
billsum['train'][0]
```

```
'text': "SECTION 1. SHORT TITLE.\n\n    This Act may be cited as the ``Justice in India Act''.\n\nSEC. 2. FINDINGS.\n\n    The Congress finds
that--\n            (1) each year, in both Jammu and Kashmir and the Punjab, \n        the Government of India detains thousands of persons under
\n        special or preventive detention laws without informing them of \n        the charges against them;\n            (2) most of these
detainees are political prisoners, \n        including prisoners of conscience;\n            (3) they are often detained for several months and
\n        sometimes even more than a year;\n            (4) detainees are not permitted any contact with lawyers or \n        family members
unless they are remanded to judicial custody and \n        transferred to prison, and only then if the family on its own \n        is able to
locate the detainee;\n            (5) in most cases, these persons are detained under the \n        Terrorist and Disruptive Activities
(Prevention) Act of 1987, \n        the National Security Act of 1980, and the Jammu and Kashmir \n        Public Safety Act of 1978;\n
(6) the Terrorist and Disruptive Activities (Prevention) \n        Act of 1987 authorizes administrative detention without formal \n
charge or trial for up to 1 year for investigation of suspected \n        ``terrorist'' or broadly defined ``disruptive'' activities;\n
(7) the 1-year period of permissible detention before trial \n        violates Article 9 of the International Covenant on Civil and \n
Political Rights, to which India is a party;\n            (8) Article 9 of the International Covenant provides, \n        ``Anyone arrested or
detained on a criminal charge shall be \n        brought promptly before a judge or other officer authorized by \n        law to exercise
judicial power and shall be entitled to trial \n        within a reasonable time or to release.'';\n            (9) under the Terrorist and
Disruptive Activities \n        (Prevention) Act of 1987, all proceedings before a designate \n        court must be conducted in secret ``at any
place other \n        than...[the court's]...ordinary place of sitting'';\n            (10) section 16(2) of the Terrorist and Disruptive \n
Activities (Prevention) Act of 1987 permits the designated \n        court to keep the ``identity and address of any witness \n        secret'';
\n            (11) under the Terrorist and Disruptive Activities \n        (Prevention) Act of 1987, a confession to a senior police \n
officer can be admitted as evidence if there is reason to \n        believe it was made voluntarily;\n            (12) the Terrorist and
Disruptive Activities (Prevention) \n        Act of 1987 amends India's criminal code, which prohibits such \n        confessions, and
substantially increases the risk of torture;\n            (13) the Terrorist and Disruptive Activities (Prevention) \n        Act of 1987
reverses the presumption of innocence, placing the \n        burden on the accused to prove that he or she is not guilty;\n            (14) the
National Security Act of 1980 permits the \n        detention of persons without charge or trial for up to 1 year \n        in order to prevent
them from acting in a manner prejudicial to \n        the security of the state, the maintenancty Act of 1980, \nthe Jammu and Kashmir Public
Safety Act of 1978, the Armed Forces \n(Punjab and Chandigarh) Special Powers Act of 1983, and the Armed \nForces (Jammu and Kashmir) Special
Powers Act of 1990…..'
 'summary': 'Justice in India Act - Terminates all development assistance for India under the Foreign Assistance Act of 1961 (except assistance
for specified health projects) if the President reports to the Congress that India has not repealed certain special and preventive detention
laws.  Provides for the resumption of such assistance if India repeals such laws.',
 'title': 'Justice in India Act'}
```

# Different Dataset

- Taken from a zipped .csv file

```
medium_datasets = load_dataset("csv", data_files="medium-articles.zip")
```

- Similarly, you can read also from json or other formats (just replace ‘csv’ with ‘json’

- See the load_dataset documentation: https://huggingface.co/docs/datasets/loading

# Splitting (and selecting a subset)

```
datasets_train_test = medium_datasets["train"].train_test_split(test_size=3000)
datasets_train_validation = datasets_train_test["train"].train_test_split(test_size=3000)

medium_datasets["train"] = datasets_train_validation["train"]
medium_datasets["validation"] = datasets_train_validation["test"]
medium_datasets["test"] = datasets_train_test["test"]
```

```
medium_datasets["train"] = medium_datasets["train"].shuffle().select(range(10000))
medium_datasets["validation"] = medium_datasets["validation"].shuffle().select(range(1000))
medium_datasets["test"] = medium_datasets["test"].shuffle().select(range(1000))
```

## Discussion

- We have split twice to create both validation and test set
- Then, using

  `shuffle().select(range(10000))`

  we randomized the datasets, and selected only a subset, to make the training less expensive (for this example, but you can use the whole dataset)

# Filtering too long examples

```
medium_datasets_cleaned = medium_datasets.filter(
    lambda example: (len(example['text']) >= 500) and
    (len(example['title']) >= 20)
)
```

## Dataset summary

```
medium_datasets_cleaned

DatasetDict({
    train: Dataset({
        features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
        num_rows: 8559
    })
    validation: Dataset({
        features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
        num_rows: 862
    })
    test: Dataset({
        features: ['title', 'text', 'url', 'authors', 'timestamp', 'tags'],
        num_rows: 861
    })
})
```

# Inspection...

```
medium_datasets_cleaned["train"][0]
```

```
{'title': 'TailwindCSS and Symfony’s Webpack Encore',
 'text': "TailwindCSS and Symfony’s Webpack Encore\n\nA quick guide to integrating TailwindCSS into Webpack encore\n\nPhoto by Tianyi Ma
on Unsplash\n\nI recently had the chance to choose and work with the TailwindCSS framework for a project at work.\n\nAfter some initial
hesitation about its utility-first CSS concept, I conducted a bit more research and interacted with the tailwind community. I eventually
came away with enough buy-in to revise my initial judgement and give it a shot.\n\nA most pleasant experience!\n\nI’m currently working
on a project that uses PHP’s Symfony framework on the backend and was tasked with integrating Tailwind into Webpack encore, Symfony’s
version of Webpack.\n\nHere’s how I went about integrating them.\n\nDependencies\n\nFirst, we need to install a couple of dependencies:
\n\ntailwindcss itself.\n\nitself. A package called postcss-loader required by tailwind.\n\nrequired by tailwind. The autoprefixer
package that helps with vendor prefixes.\n\nAssuming you already have Webpack-encore installed, you can remove @symfony/webpack-encore ,
otherwise, all dependencies stowed together, we end up with the following command:\n\nyarn add --dev @symfony/webpack-encore tailwindcss
postcss-loader autoprefixer\n\nComposer being our package manager. I’m using the yarn add command but the same outcome could be achieved
with npm via npm install .\n\nConfiguration\n\nThere are four files I configured and/or created:
\n\ntailwind.css\n\ntailwind.config.js\n\npostcss.config.js\n\nwebpack.config.js\n\ntailwind.css: I created this file under my project’s
CSS assets directory — assets/css/tailwind.css — but you can put it anywhere you like. The “@tailwind” directive injects the base ,
components , and utilities styles into your CSS.\n\ntailwind.config.js: This file is optional, but if you need to customize tailwind in
any way (e.g., adding a prefix to class names, customizing a colour or font size, etc), you’ll need to create this file via the tailwind
CLI by running npx tailwind init . This will generate the file in your root folder.\n\npostcss.config.js: I created this file in the
root of my project. This is where we specify the PostCSS plugins we want to use to process our CSS.\n\nwebpack.config.js: If you were
already using Webpack-encore, the file should already be configured, otherwise this could help. The config for this file was just as
swift as the others: I added .addStyleEntry('tailwind', './assets/css/tailwind.css') pointing to our injected styles in tailwind.css and
enabled PostCSS loader, as shown below:\n\nTemplates",
 'url': 'https://medium.com/better-programming/tailwindcss-and-symfonys-webpack-encore-7bfc8c18665b',
 'authors': "['Amir Bizimana']",
 'timestamp': '2020-01-03 17:04:20.498000+00:00',
 'tags': "['CSS', 'Tailwind Css', 'Webpack', 'Symfony', 'Programming']"}
```

## Load tokenizer...

- We use a tokenizer from the “t5-small” model
- Note you could train your own tokenizer from a large text corpus…

```python
from transformers import AutoTokenizer

checkpoint = "t5-small"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

# Text preprocessing and preparation - I

```python
import nltk
nltk.download('punkt')
import string
```

```python
def clean_text(text):
    sentences = nltk.sent_tokenize(text.strip())
    sentences_cleaned = [s for sent in sentences for s in sent.split("\n")]
    sentences_cleaned_no_titles = [sent for sent in sentences_cleaned if len(sent) > 0 and sent[-1] in string.punctuation]
    text_cleaned = "\n".join(sentences_cleaned_no_titles)
    return text_cleaned
```

## Discussion

- The clean_text() function splits sentences by first using the NLTK API you know, and then the new line “\n”
- Then it takes all sentences having a nonzero length that terminate by a punctuation symbol, and aggregate them

# Text preprocessing and preparation - II

```
max_input_length = 512
max_target_length = 64
prefix = "summarize: “

def preprocess_data(examples):
    texts_cleaned = [clean_text(text) for text in examples["text"]]
    inputs = [prefix + text for text in texts_cleaned]
    model_inputs = tokenizer(inputs, max_length=max_input_length, truncation=True)
    labels = tokenizer(examples["title"], max_length=max_target_length, truncation=True)
    model_inputs["labels"] = labels["input_ids"]
    return model_inputs
```

# Discussion - I

- The preprocess_data() is the core of the text preprocessing
- You always need it (the other func is optional, just for this example)
- What it does:
    - Cleanup using the clean_text() as seen before
    - Addition of a prefix (“Prompting”) - will explain later why
    - Tokenization of the text and of the labels using the loaded tokenizer
    - Inputs are truncated by max_input_length
    - Outputs are truncated by max_target_length

# Discussion - II

- model_inputs will contain the input text (taken from the “text” column of the dataset)
- labels will contain the target text (taken from the “title” column of the dataset)
- Finally, labels are added to model_inputs:

```
model_inputs["labels"] = labels["input_ids"]
```

## Prompting

- We prepend the word “summarize: “ to each input example
- This is because:
  - T5 has been pretrained on tasks where it has been asked to perform actions using sentences containing “action verbs”
  - These are called “prompts” (like the prompts you may use with ChatGPT)
  - There are predefined prompts for tasks such as:
    - Summarization, language translation, etc.

# Finally…

- We actually perform the preprocessing using the “map” function, applying it to the whole dataset

```
tokenized_datasets = medium_datasets_cleaned.map(preprocess_data,
                                                 batched=True)
```

# Loading the data collator

- It dynamically pads the sentences at run time, instead of doing it initially
- Much more efficient

```
from transformers import DataCollatorForSeq2Seq

data_collator = DataCollatorForSeq2Seq(tokenizer=tokenizer, model=checkpoint)
```

# Computing the metrics

```python
import evaluate

rouge = evaluate.load("rouge")

import numpy as np

def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
    result = rouge.compute(predictions=decoded_preds, references=decoded_labels, use_stemmer=True)
    prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for pred in predictions]
    result["gen_len"] = np.mean(prediction_lens)
    return {k: round(v, 4) for k, v in result.items()}
```

# Discussion

- First of all, we load the metric we want to use during the evaluation
- For generative models, typical metrics are:
    - exact_match (1 if predicted and label match, 0 otherwise)
    - rouge
    - bleu

# ROUGE

- Recall-Oriented Understudy for Gisting Evaluation
- It is a recall-based set of metrics for comparing generated text against reference one (by computing  some overlap)
- There is a different rouge ROUGE-N for different n-gram lengths, e.g.:
  - ROUGE-1: overlap of unigrams (each word) between the system and reference summaries.
  - ROUGE-2: overlap of bigrams between the system and reference summaries.

# Example

- R (reference summary): The cat is on the mat.
- C (candidate summary): The cat and the dog.
- ROUGE-1 precision: ratio of the number of unigrams in C that appear also in R, over the number of unigrams in C.
- ROUGE-1 recall: ratio of the number of unigrams in R that appear also in C, over the number of unigrams in R.
- ROUGE-1 precision = 3/5 = 0.6
- ROUGE-1 recall = 3/6 = 0.5

## BLEU

- BLEU (bilingual evaluation understudy)
- Algorithm for evaluating the quality of text which has been machine-translated from one natural language to another.
- We will skip the details of of its maths in this course

# Back to the metric function...

```python
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
    result = rouge.compute(predictions=decoded_preds, references=decoded_labels, use_stemmer=True)
    prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for pred in predictions]
    result["gen_len"] = np.mean(prediction_lens)
    return {k: round(v, 4) for k, v in result.items()}
```

## Discussion - I

- The function will take by the optimizer a tuple, with the generated data and the reference one

- We split the tuple

```
predictions, labels = eval_pred
```

- We decode the predictions using the tokenizer (otherwise they are numeric IDs)

```
    decoded_preds = tokenizer.batch_decode(predictions, skip_special_tokens=True)
```

- If the label is not "-100" (OOV) we return the label, otherwise the padding token

```
    labels = np.where(labels != -100, labels, tokenizer.pad_token_id)
```

- Then, we decode the label

```
    decoded_labels = tokenizer.batch_decode(labels, skip_special_tokens=True)
```

# Discussion - II

- We compute the ROUGE metrics:

```
result = rouge.compute(predictions=decoded_preds, references=decoded_labels, use_stemmer=True)
```

- If you are not working on natural language or you want a more precise evaluation, set use_stemmer=False

- We compute the number of non-empty predictions and assign its mean to `result[“gen_len”]`:

```
prediction_lens = [np.count_nonzero(pred != tokenizer.pad_token_id) for pred in predictions]
result["gen_len"] = np.mean(prediction_lens)
```

- We round all metrics to be returned, and return them:

```
  return {k: round(v, 4) for k, v in result.items()}
```

## Loading the model

- We create a seq 2 seq transformer model using the “T5-small” transformer

```
from transformers import AutoModelForSeq2SeqLM, Seq2SeqTrainingArguments, Seq2SeqTrainer

model = AutoModelForSeq2SeqLM.from_pretrained(checkpoint)
```

## Setting the training arguments

```
training_args = Seq2SeqTrainingArguments(
    output_dir=".",
    evaluation_strategy="steps",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    weight_decay=0.01,
    save_total_limit=10,
    num_train_epochs=4,
    predict_with_generate=True,
    fp16=False,
    load_best_model_at_end=True,
    push_to_hub=False)
```

# Discussion

Most of the settings are self-explanatory, but note that:

- The evaluation can done at each step ( evaluation_strategy=“steps”) or at each epoch (evaluation_strategy=“epochs”)
- Keep the learning_rate=2e-5
- You can change the batch size, lowering it if you don’t have enough memory
- You could increase the number of epochs (num_train_epochs)
- Alternatively, you can directly set the number of training steps (max_steps)
- save_total_limit determines the number of checkpoints that are saved
- Set fp16=**False** for CPU**,** it can be set to fp16=**True** for GPU (it sets the floating point computation to 16 bits)
- load_best_model_at_end loads the best model when the training is complete, but requires an evaluation strategy at “steps”
- push_to_hub allows to push the trained model on HuggingFace (you need to set an account and a token)

## More documentation…

https://huggingface.co/docs/transformers/v4.29.1/en/main_classes/trainer#transformers.TrainingArguments

## Preparing the training...

```
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    tokenizer=tokenizer,
    data_collator=data_collator,
    compute_metrics=compute_metrics)
```

## Discussion

You need to pass:

- the model
- the arguments set before
- the training and evaluation dataset
- the tokenizer
- the data collator
- the function to compute the metrics

## Running the training (fine tuning of the model)!

- It will take hours (better to use GPUs)

```
trainer.train()
```

# Saving the model on disk...

```python
import torch
model_name="summarizingModel"
torch.save(model.state_dict(),model_name)
```

# Reloading the model - Setting device type

- First of all, while we may have trained the model on a powerful GPU, we may or may not use it on a machine with GPU
- The inference is doable on CPU too (yet still slower than on GPUs)
- Therefore, we need to determine whether the machine where we do inference has GPUs, and set the device type consequently

```python
import torch
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

# Loading the actual model

- We need to:
  - Load the tokenizer (also used during the training)
  - Load the generic model
  - Reload the saved model parameters from disk (we need to specify the device where it needs to be loaded)

```
checkpoint = "t5-small"
from transformers import AutoModelForSeq2SeqLM, Seq2SeqTrainingArguments, Seq2SeqTrainer
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSeq2SeqLM.from_pretrained(checkpoint)
model.load_state_dict(torch.load("summarizingModel",map_location=device))
```

# Creating the text

- By adding the prompt

```
with open("abstract.txt", "r") as f:
    text = f.read()
prefix=“summarize: “
text=prefix+text
```

# Performing the inference

```
inputs = tokenizer(text, return_tensors="pt").input_ids
inputs=inputs.to(device)

outputs = model.generate(inputs, max_new_tokens=20, num_beams=10,
do_sample=False)
tokenizer.decode(outputs[0], skip_special_tokens=True)
```

- We take an input the input text to the model
- If needed, we preprocess it (using the same preprocessing done for the training instances)
- If needed, we prepend a prompt
- We tokenize the text
- We use the generate() function to generate the prediction
- Finally, we use again the tokenizer to convert the generated token IDs back to words

## Model and input on the same device

- To make Torch working properly, model and inputs must be on the same device

- You can always move inputs and models to a different device using the to(device) function, e.g.

```
model=model.to("cpu") or model=model.to("cuda")

inputs=inputs.to("cpu") or inputs=inputs.to("cuda")
```

# Number of beams (beam size)

- Let’s assume the model generates a single sentence
- It does it word by word, generating every time the most likely word according to the models’ weights
- However, if it starts forming a bad sentence (that, altogether, has a bad confidence in the generation), there’s no way to rollback
- By having a beam size>1, the model generates N sentences in parallel, adding words with different confidence
  - In the end, the sentence with the highest confidence is outputted

# Beam size: example - I

- Suppose the model has to translate the sentence “Comment vas tu?”
- Let’s assume we use a beam size=3
- At the first decoder step, the model outputs an estimated probability for each possible word:
  - How: 75%
  - What: 3%
  - You: 1%

# Beam size: example - II

- We create three copies of the model that will be used to find the next word for each sentence
- The first model will try to find the next word in the sentence “How” by computing this time conditioned probabilities
    - “will”:  36%
    - “are”: 32%
    - “do”: 16%
- The second model will try to complete the sentence starting with “What”
    - “are”: 50%
    - … and so on

## Beam size: example - III

- If the vocabulary has 10,000 words, each model will output 10,000 probabilities
- Next, we compute the probabilities for each of the 30000 two-words sentences
- For example:
    - How: 75%, will: 36% → "how will"=75% x 36% = 27%
- And only the top-3 sentences are kept

## Other parameters of generate

- temperature: low temperature makes the model less “creative” with respect to the training set. A high temperature produces more creative answers

- repetition_penalty: you might want to increase it to values greater than one if the model tends to repeat text in its output

- Example:

```
outputs = model.generate(inputs, max_new_tokens=30, num_beams=10, do_sample=False, temperature=0.6,repetition_penalty=3.0)
```

## Using a pre-trained model as is

## Using pretrained models from scratch

- For some tasks, pretrained models may just work fine without even fine tuning them
- Let’s see it working for a translation example…

## Loading the model (t5-base)

```python
checkpoint = "t5-base"
from transformers import AutoModelForSeq2SeqLM, Seq2SeqTrainingArguments, Seq2SeqTrainer
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSeq2SeqLM.from_pretrained(checkpoint)
```

# Let’s do a translation task

- Which we specify through a prompting:

```
text="The cat is on the table"
prefix="translate English to French: "
text=prefix+text
```

- Usual processing and generation:

```
inputs = tokenizer(text, return_tensors=“pt").input_ids
outputs = model.generate(inputs, max_new_tokens=20, do_sample=False)
tokenizer.decode(outputs[0], skip_special_tokens=True)

'Le chat est sur la table'
```

# Loading models in general..

- You can use the syntax:

```
model.from_pretrained(“user/model_name”)
```

## Other applications

## Other applications

- The idea isn’t much different for:
  - Natural language translation
  - Q&A
- You always need to train the model with an input (e.g., text to be translated, or question, or even text to be completed)
- The label will be the translation, the answer, the text to generate, etc.